

Undergraduate Topics in Computer Science

Torben Ægidius Mogensen

Introduction to Compiler Design

Third Edition



UTiCS



Springer


Undergraduate Topics in Computer Science


Series Editor

Ian Mackie, University of Sussex, Brighton, UK

Advisory Editors

Samson Abramsky , Department of Computer Science, University of Oxford, Oxford, UK

Chris Hankin , Department of Computing, Imperial College London, London, UK

Mike Hinchey , Lero—The Irish Software Research Centre, University of Limerick, Limerick, Ireland

Dexter C. Kozen, Department of Computer Science, Cornell University, Ithaca, NY, USA

Hanne Riis Nielson , Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kongens Lyngby, Denmark

Steven S. Skiena, Department of Computer Science, Stony Brook University, Stony Brook, NY, USA

Iain Stewart , Department of Computer Science, Durham University, Durham, UK

Joseph Migga Kizza, Engineering and Computer Science, University of Tennessee at Chattanooga, Chattanooga, TN, USA

Roy Crole, School of Computing and Mathematics Sciences, University of Leicester, Leicester, UK

'Undergraduate Topics in Computer Science' (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems, many of which include fully worked solutions.

The UTiCS concept centers on high-quality, ideally and generally quite concise books in softback format. For advanced undergraduate textbooks that are likely to be longer and more expository, Springer continues to offer the highly regarded *Texts in Computer Science* series, to which we refer potential authors.

Torben Ægidius Mogensen

Introduction to Compiler Design

Third Edition

 Springer

Torben Ægidius Mogensen
Datalogisk Institut
Københavns Universitet
Copenhagen, Denmark

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-031-46459-1 ISBN 978-3-031-46460-7 (eBook)
<https://doi.org/10.1007/978-3-031-46460-7>

1st edition: © Springer-Verlag London Limited 2011
2nd & 3rd editions: © Springer International Publishing AG 2017, 2024

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Preface

Language is a process of free creation; its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied. Even the interpretation and use of words involves a process of free creation.

Noam Chomsky (1928–)

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called *machine language*. Since this is a tedious and error-prone process, most programming is done using a high-level *programming language*. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the *compiler* comes in.

A compiler translates (or *compiles*) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to detect and report obvious programmer mistakes.

Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can detect some types of programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level language is that the same program can be compiled to many different machine languages, and hence be brought to run on many different machines.

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs. Additionally, what code is optimal may change when a new version of a processor is made, so to remain optimal, a hand-written machine language program may need to be rewritten, whereas a program in a high-level language just needs to be recompiled using a compiler that optimizes for the new version of the processor.

The Phases of a Compiler

Since writing a compiler is a nontrivial task, it is a good idea to structure the work. A typical way of doing this is to split the compilation into several phases with well-defined interfaces between them. Conceptually, these phases operate in sequence (though in practice, they are often interleaved), each phase (except the first) taking the output from the previous phase as its input. It is common to let each phase be handled by a separate program module. Some of these modules are written by hand, while others may be generated from specifications. Often, some of the modules can be shared between several compilers.

A common division into phases is described below. In some compilers, the ordering of phases may differ slightly, some phases may be combined or split into several phases or some extra phases may be inserted between those mentioned below.

Lexical analysis This is the initial part of reading and analyzing the program text: The text is read and divided into *tokens*, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number. Lexical analysis is often abbreviated to *lexing*.

Syntax analysis This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the *syntax tree*) that reflects the structure of the program. This phase is often called *parsing*.

Type checking This phase analyzes the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared, or if it is used in a context that does not make sense given the type of the variable, such as trying to use a Boolean value as a function pointer.

Intermediate code generation The program is translated to a simple machine-independent intermediate language.

Register allocation The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

Machine code generation The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.

Assembly and linking The assembly language code is translated into binary representation and addresses of variables, functions, etc., are determined.

The first three phases are collectively called *the front end* of the compiler and the last three phases are collectively called *the back end*. The middle part of the compiler is in this context only the intermediate code generation, but this often includes various optimizations and transformations on the intermediate code.

Each phase, through checking and transformation, establishes invariants on the data it passes on to the next phase. For example, the type checker can assume the absence of syntax errors, and the code generation can assume the absence of type errors. These invariants can reduce the burden of writing the later phases.

Assembly and linking are typically done by programs supplied by the machine or operating system vendor, and are hence not part of the compiler itself. We will not further discuss these phases in this book, but assume that a compiler produces its result as symbolic assembly code.

Interpreters

An *interpreter* is another way of implementing a programming language. Interpretation shares many aspects with compiling. Lexing, parsing and type-checking are in an interpreter done just as in a compiler. But instead of generating code from the syntax tree, the syntax tree is processed directly to evaluate expressions, execute statements, and so on. An interpreter may need to process the same piece of the syntax tree (for example, the body of a loop) many times, and hence interpretation is typically slower than executing a compiled program. But writing an interpreter is often simpler than writing a compiler, and an interpreter is easier to move to a different machine, so for applications where speed is not of essence, or where each part of the program is executed only once, interpreters are often used.

Compilation and interpretation may be combined to implement a programming language. For example, the compiler may produce intermediate-level code which is then interpreted rather than compiled to machine code. In some systems, there may even be parts of a program that are compiled to machine code, some parts that are compiled to intermediate code that is interpreted at runtime, while other parts may be interpreted directly from the syntax tree. Each choice is a compromise between speed and space: Compiled code tends to be bigger than intermediate code, which tends to be bigger than syntax, but each step of translation improves running speed.

Using an interpreter is also useful during program development, where it is more important to be able to test a program modification quickly rather than run the program efficiently. And since interpreters do less work on the program before execution

starts, they are able to start running the program more quickly. Furthermore, since an interpreter works on a program representation that is closer to the source code than to the compiled code, error messages can be more precise and informative.

We will discuss interpreters briefly in Chap. 4, but they are not the main focus of this book.

Why Learn About Compilers?

Few people will ever be required to write a compiler for a general-purpose language like C, Java or Haskell. So why do most computer science institutions offer compiler courses and often make these mandatory?

Some typical reasons are:

- (a) It is considered a topic that you should know in order to be “well-cultured” in computer science.
- (b) A good craftsman should know his tools, and compilers are important tools for programmers and computer scientists.
- (c) The techniques used for constructing a compiler are useful for other purposes as well.
- (d) There is a good chance that a programmer or computer scientist will need to write a compiler or interpreter for a domain-specific language.

The first of these reasons is somewhat dubious, though something can be said for “knowing your roots”, even in such a hastily changing field as computer science.

Reason “b” is more convincing: Understanding how a compiler is built will allow programmers to get an intuition about what their high-level programs will look like when compiled, and use this intuition to tune programs for better efficiency. Furthermore, the error reports that compilers provide are often easier to understand when one knows about and understands the different phases of compilation, such as knowing the difference between lexical errors, syntax errors, type errors, and so on.

The third reason is also quite valid. In particular, the techniques used for reading (*lexing* and *parsing*) the text of a program and converting this into a form (*abstract syntax*) that is easily manipulated by a computer, can be used to read and manipulate any kind of structured text such as XML documents, JSON files, address lists, etc.

Reason “d” is becoming more and more important as domain-specific languages (DSLs) are gaining in popularity. A DSL is a (typically small) language designed for a narrow class of problems. Examples are database query languages, text-formatting languages, 3D modeling languages, and languages for setting up economic simulations. The target language for a compiler for a DSL may be traditional machine code, but it can also be another high-level language for which compilers already exist, a sequence of control signals for a machine, or formatted text and graphics in some printer-control language (e.g., PostScript). DSLs are often interpreted instead of compiled. Even so, all DSL compilers and interpreters will have front ends for

reading and analyzing the program text that is similar to those used in compilers and interpreters for general-purpose languages.

In brief, the methods needed to make a compiler front end are more widely applicable than the methods needed to make a compiler back end, but the latter is more important for understanding how a program is executed on a machine.

About the Third Edition of the Book

The third edition has been extended with material about the SSA form, parametric polymorphism (templates), garbage collection, and pattern matching. This extra material is aimed at advanced B.Sc.-level courses or M.Sc.-level courses. Additionally, typos have been fixed and clarifications as well as a few more examples have been added.

To the Lecturer

This book was written for use in the introductory compiler course at DIKU, the Department of Computer Science at the University of Copenhagen, Denmark.

At times, standard techniques from compiler construction have been simplified for presentation in this book. In such cases, references are made to books or articles where the full version of the techniques can be found.

The book aims at being “language neutral”. This means two things:

- With a few exceptions, little detail is given about how the methods in the book can be implemented in any specific language. Rather, the description of the methods is given in the form of algorithm sketches and textual suggestions of how these can be implemented in various types of languages, in particular, imperative and functional languages.
- There is no single through-going example of a language to be compiled. Instead, different small (sub-)languages are used in various places to cover exactly the points that the text needs. This is done to avoid drowning in detail, hopefully allowing the readers to “see the wood for the trees”.

Each chapter has a section on further reading, which suggests additional reading material for interested students. Chapters also have sets of exercises. Few of these require access to a computer, but can be solved on paper or blackboard. After some of the sections in the book, a few easy exercises are listed as suggested exercises. It is recommended that the student attempts to solve these exercises before continuing reading, as the exercises support understanding of the previous sections.

Teaching with this book can be supplemented with project work, where students write simple compilers. Since the book is language neutral, no specific project is given. Instead, the teacher must choose relevant tools and select a project that fits the

level of the students and the time available. Depending on the amount of project work and on how much of the advanced material added in the second and third editions, the book can support course sizes ranging from 5 to 12 ECTS points.

The following link contains extra material for the book, including solutions to selected exercises—sn.pub/hMOU3o.

Copenhagen, Denmark

Torben Ægidius Mogensen

Acknowledgements

Most people return small favors, acknowledge medium ones and repay greater ones—with ingratitude.

Benjamin Franklin (1705–1790)

The author wishes to thank all the people who have been helpful in making this book a reality. This includes the students who have been exposed to earlier versions of the book and notes that made it into the 2nd and 3rd editions of the book at the compiler courses “Dat1E”, “Oversættelse”, “Implementering af programmeringssprog” and “Advanced Language Processing” at DIKU (the Department of Computer Science at The University of Copenhagen), and who have found numerous typos and other errors in the earlier versions. I would also like to thank the co-teachers and instructors of these courses, who have pointed out places where things were not as clear as they could be. Lastly, I would like to thank the people who have used the previous editions of the book and suggested material for this third edition.

Copenhagen, Denmark
August 2023

Torben Ægidius Mogensen

Contents

1	Lexical Analysis	1
1.1	Regular Expressions	2
1.1.1	Shorthands	4
1.1.2	Examples	5
1.2	Nondeterministic Finite Automata	7
1.3	Converting a Regular Expression to an NFA	9
1.3.1	Optimisations	11
1.4	Deterministic Finite Automata	12
1.5	Converting an NFA to a DFA	13
1.5.1	Solving Set Equations	13
1.5.2	The Subset Construction	16
1.6	Size Versus Speed	19
1.7	Minimisation of DFAs	20
1.7.1	Example	21
1.7.2	Dead States	23
1.8	Lexers and Lexer Generators	25
1.8.1	Lexer Generators	29
1.9	Properties of Regular Languages	30
1.9.1	Relative Expressive Power	31
1.9.2	Limits to Expressive Power	32
1.9.3	Closure Properties	33
1.10	Further Reading	34
1.11	Exercises	34
	References	38
2	Syntax Analysis	41
2.1	Context-Free Grammars	42
2.1.1	How to Write Context-Free Grammars	44
2.2	Derivation	46
2.2.1	Syntax Trees and Ambiguity	48

- 2.3 Operator Precedence 50
 - 2.3.1 Rewriting Ambiguous Expression Grammars 52
- 2.4 Other Sources of Ambiguity 55
- 2.5 Syntax Analysis 56
- 2.6 Predictive Parsing 56
- 2.7 *Nullable* and *FIRST* 57
- 2.8 Predictive Parsing Revisited 61
- 2.9 Follow 63
- 2.10 A Larger Example 65
- 2.11 LL(1) Parsing 67
 - 2.11.1 Recursive Descent 68
 - 2.11.2 Table-Driven LL(1) Parsing 69
 - 2.11.3 Conflicts 72
- 2.12 Rewriting a Grammar for LL(1) Parsing 72
 - 2.12.1 Eliminating Left-Recursion 73
 - 2.12.2 Left-Factorisation 75
 - 2.12.3 Construction of LL(1) Parsers Summarised 76
- 2.13 SLR Parsing 76
- 2.14 Constructing SLR Parse Tables 79
 - 2.14.1 Conflicts in SLR Parse-Tables 83
- 2.15 Using Precedence Rules in LR Parse Tables 84
- 2.16 Using LR-Parser Generators 86
 - 2.16.1 Conflict Handling in Parser Generators 87
 - 2.16.2 Declarations and Actions 88
 - 2.16.3 Abstract Syntax 88
- 2.17 Properties of Context-Free Languages 92
- 2.18 Further Reading 93
- 2.19 Exercises 93
- References 97
- 3 Scopes and Symbol Tables 99**
 - 3.1 Symbol Tables 100
 - 3.1.1 Implementation of Symbol Tables 100
 - 3.1.2 Simple Persistent Symbol Tables 101
 - 3.1.3 A Simple Imperative Symbol Table 102
 - 3.1.4 Efficiency Issues 103
 - 3.1.5 Shared or Separate Name Spaces 103
 - 3.2 Further Reading 104
 - 3.3 Exercises 104
 - Reference 104
- 4 Interpretation 105**
 - 4.1 The Structure of an Interpreter 106
 - 4.2 A Small Example Language 106
 - 4.3 An Interpreter for the Example Language 107
 - 4.3.1 Evaluating Expressions 108

- 4.3.2 Interpreting Function Calls 110
- 4.3.3 Interpreting a Program 110
- 4.4 Advantages and Disadvantages of Interpretation 111
- 4.5 Further Reading 113
- 4.6 Exercises 113
- References 115
- 5 Type Checking 117**
 - 5.1 The Design Space of Type Systems 118
 - 5.2 Attributes 119
 - 5.3 Environments for Type Checking 120
 - 5.4 Type Checking of Expressions 120
 - 5.5 Type Checking of Function Declarations 122
 - 5.6 Type Checking a Program 123
 - 5.7 Advanced Type Checking 124
 - 5.8 Further Reading 127
 - 5.9 Exercises 127
 - References 128
- 6 Intermediate-Code Generation 129**
 - 6.1 Designing an Intermediate Language 131
 - 6.2 The Intermediate Language 132
 - 6.3 Syntax-Directed Translation 134
 - 6.4 Generating Code from Expressions 134
 - 6.4.1 Examples of Expression Translation 137
 - 6.5 Translating Statements 138
 - 6.5.1 Example of Statement Translation 141
 - 6.6 Logical Operators 141
 - 6.6.1 Sequential Logical Operators 143
 - 6.6.2 Example of Translation of Conditions 145
 - 6.7 Advanced Control Statements 146
 - 6.8 Translating Structured Data 147
 - 6.8.1 Floating-Point Values 147
 - 6.8.2 Arrays 148
 - 6.8.3 Strings 153
 - 6.8.4 Records/Structs and Unions 154
 - 6.9 Translation of Declarations 154
 - 6.9.1 Simple Local Declarations 155
 - 6.9.2 Translation of Function Declarations 156
 - 6.10 Further Reading 157
 - 6.11 Exercises 157
 - References 160

7	Machine-Code Generation	161
7.1	Conditional Jumps	162
7.2	Constants	163
7.3	Exploiting Complex Instructions	164
7.3.1	Two-Address Instructions	168
7.4	Optimisations	169
7.5	Further Reading	170
7.6	Exercises	171
	References	172
8	Register Allocation	173
8.1	Liveness	174
8.2	Liveness Analysis	175
8.3	Interference	178
8.4	Register Allocation by Graph Colouring	180
8.5	Spilling	181
8.6	Heuristics	183
8.6.1	Removing Redundant Moves	185
8.6.2	Using Explicit Register Numbers	185
8.7	Further Reading	186
8.8	Exercises	186
	References	188
9	Functions	189
9.1	The Call Stack	189
9.2	Activation Records	190
9.3	Prologues, Epilogues and Call-Sequences	191
9.4	Letting the Callee Save Registers	194
9.5	Caller-Saves Versus Callee-Saves	195
9.6	Using Registers to Pass Parameters	196
9.7	Interaction with the Register Allocator	198
9.8	Local Variables	200
9.9	Accessing Non-local Variables	200
9.9.1	Global Variables	201
9.9.2	Call-by-Reference Parameters	202
9.10	Functions as Parameters	203
9.11	Variants	203
9.11.1	Variable-Sized Frames	203
9.11.2	Variable Number of Parameters	204
9.11.3	Direction of Stack-Growth and Position of <i>FP</i>	204
9.11.4	Register Stacks	204
9.12	Optimisations for Function Calls	205
9.12.1	Inlining	205
9.12.2	Tail-Call Optimisation	206
9.13	Further Reading	211

- 9.14 Exercises 212
- References 213
- 10 Data-Flow Analysis and Optimisation 215**
 - 10.1 Data-Flow Analysis 215
 - 10.2 How to Design a Data-Flow Analysis 216
 - 10.3 Liveness Analysis 216
 - 10.3.1 Improving Liveness Analysis 217
 - 10.4 Generalising from Liveness Analysis 218
 - 10.5 Common Subexpression Elimination 219
 - 10.5.1 Available Assignments 219
 - 10.5.2 Example of Available-Assignments Analysis 222
 - 10.5.3 Using Available Assignment Analysis
for Common Subexpression Elimination 223
 - 10.6 Index-Check Elimination 225
 - 10.7 Jump-to-Jump Elimination 228
 - 10.8 Resources Used by Data-Flow Analysis 230
 - 10.9 Pointer Analysis 231
 - 10.10 Limitations of Data-Flow Analyses 235
 - 10.11 SSA Form 236
 - 10.11.1 Transforming to SSA Form 236
 - 10.11.2 Using the SSA Form 240
 - 10.12 Further Reading 241
 - 10.13 Exercises 241
 - References 243
- 11 Optimisations for Loops 245**
 - 11.1 Loops 245
 - 11.2 Code Hoisting 246
 - 11.3 Memory Prefetching 248
 - 11.4 Incrementalisation 250
 - 11.4.1 Rules for Incrementalisation 252
 - 11.5 Further Reading 254
 - 11.6 Exercises 254
 - Reference 255
- 12 More Language Features 257**
 - 12.1 Parametric Polymorphism 257
 - 12.1.1 Implementing Templates 258
 - 12.1.2 Variants 259
 - 12.2 Garbage Collection 260
 - 12.2.1 Identifying Pointers 261
 - 12.2.2 The Freelist 262
 - 12.2.3 Manual Freeing 264
 - 12.2.4 The Mark Phase 265
 - 12.2.5 The Sweep Phase 267

12.2.6	Tying the Knots	268
12.2.7	Remarks About Heap Memory Management	268
12.3	Pattern Matching	268
12.4	Further Reading	271
12.5	Exercises	272
	References	273
Appendix A: Set Notation and Concepts		275
Index		283

List of Figures

Fig. 1.1	Regular expressions and their derivation	3
Fig. 1.2	Some algebraic properties of regular expressions	4
Fig. 1.3	Example of an NFA	8
Fig. 1.4	Constructing NFA fragments from regular expressions	10
Fig. 1.5	NFA for the regular expression $(alb)^*ac$	10
Fig. 1.6	Optimised NFA construction for regular expression shorthands	11
Fig. 1.7	Optimised NFA for $[0-9]^+$	12
Fig. 1.8	Example of a DFA	12
Fig. 1.9	DFA constructed from the NFA in Fig. 1.5	19
Fig. 1.10	Non-minimal DFA	21
Fig. 1.11	Minimal DFA	23
Fig. 1.12	Combined NFA for several tokens	27
Fig. 1.13	Combined DFA for several tokens	28
Fig. 1.14	A 4-state NFA that gives 15 DFA states	32
Fig. 2.1	Converting regular expressions to context-free grammars	45
Grammar 2.2	Simple expression grammar	45
Grammar 2.3	Simple statement grammar	46
Grammar 2.4	Example grammar	47
Fig. 2.5	Derivation of the sequence $aabbbcc$ using Grammar 2.4	47
Fig. 2.6	Leftmost derivation of the sequence $aabbbcc$ using Grammar 2.4	47
Fig. 2.7	Syntax tree for the string $aabbbcc$ using Grammar 2.4	48
Fig. 2.8	Alternative syntax tree for the string $aabbbcc$ using Grammar 2.4	49
Grammar 2.9	Unambiguous version of Grammar 2.4	49
Fig. 2.10	Fully reduced tree for the syntax tree in Fig. 2.7	51

Fig. 2.11	Preferred syntax tree for $2+3*4$ using Grammar 2.2, and the corresponding fully reduced tree	51
Grammar 2.12	Unambiguous expression grammar	54
Fig. 2.13	Syntax tree for $2+3*4$ using Grammar 2.12, and the corresponding fully reduced tree	55
Grammar 2.14	Unambiguous grammar for statements	56
Fig. 2.15	Fixed-point iteration for calculation of <i>Nullable</i>	59
Fig. 2.16	Fixed-point iteration for calculation of <i>FIRST</i>	61
Fig. 2.17	Recursive descent parser for Grammar 2.9	69
Fig. 2.18	Tree-building recursive descent parser for Grammar 2.9	70
Fig. 2.19	LL(1) table for Grammar 2.9	70
Fig. 2.20	Program for table-driven LL(1) parsing	71
Fig. 2.21	Input and stack during table-driven LL(1) parsing	71
Fig. 2.22	Tree-building program for table-driven LL(1) parsing	72
Grammar 2.23	Removing left-recursion from Grammar 2.12	74
Grammar 2.24	Left-factorised grammar for conditionals	75
Fig. 2.25	Example shift-reduce parsing	77
Fig. 2.26	SLR table for Grammar 2.9	79
Fig. 2.27	Algorithm for SLR parsing	79
Fig. 2.28	Example SLR parsing	80
Grammar 2.29	Example grammar for SLR-table construction	80
Fig. 2.30	NFAs for the productions in Grammar 2.29	81
Fig. 2.31	Combined NFA for Grammar 2.29: epsilon transitions are added, and A is the only start state	81
Fig. 2.32	DFA constructed from the NFA in Fig. 2.31	82
Fig. 2.33	DFA table for Grammar 2.9, equivalent to the DFA in Fig. 2.32	82
Fig. 2.34	Summary of SLR parse-table construction	83
Fig. 2.35	Textual representation of NFA states	88
Grammar 4.1	Example language for interpretation	107
Fig. 4.2	Evaluating expressions	109
Fig. 4.3	Evaluating a function call	111
Fig. 4.4	Interpreting a program	112
Fig. 5.1	The design space of type systems	119
Fig. 5.2	Type checking of expressions	121
Fig. 5.3	Type checking a function declaration	123
Fig. 5.4	Type checking a program	124
Grammar 6.1	The intermediate language	132
Grammar 6.2	A simple expression language	134
Fig. 6.3	Translating an expression	137
Grammar 6.4	Statement language	138
Fig. 6.5	Translation of statements	139
Fig. 6.6	Translation of simple conditions	140
Fig. 6.7	Example statements	141

Fig. 6.8	Example statement translation	141
Grammar 6.9	Example language with logical operators	143
Fig. 6.10	Translation of sequential logical operators	144
Fig. 6.11	Example of translating conditions	146
Fig. 6.12	Translation for one-dimensional arrays	149
Fig. 6.13	A two-dimensional array	150
Fig. 6.14	Translation of multi-dimensional arrays	152
Fig. 6.15	Translation of simple declarations	155
Fig. 6.16	Example function definition translation	156
Fig. 7.1	Pattern/replacement pairs for a subset of the RISC-V instruction set	167
Fig. 8.1	Example program for liveness analysis and register allocation	175
Fig. 8.2	Gen and kill sets	176
Fig. 8.3	<i>succ</i> , <i>gen</i> and <i>kill</i> for the program in Fig. 8.1	177
Fig. 8.4	Fixed-point iteration for liveness analysis	178
Fig. 8.5	Interference graph for the program in Fig. 8.1	179
Fig. 8.6	Algorithm 8.6 applied to the graph in Fig. 8.5	182
Fig. 8.7	Program from Fig. 8.1 after spilling variable <i>a</i>	183
Fig. 8.8	Interference graph for the program in Fig. 8.7	183
Fig. 8.9	Colouring of the graph in Fig. 8.8	184
Fig. 9.1	Simple activation record layout	191
Fig. 9.2	Prologue for the header $f(p_1, \dots, p_m)$ using the frame layout shown in Fig. 9.1	192
Fig. 9.3	Epilogue for the instruction RETURN <i>result</i> using the frame layout shown in Fig. 9.1	192
Fig. 9.4	Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using the frame layout shown in Fig. 9.1	193
Fig. 9.5	Activation record layout for callee-saves	194
Fig. 9.6	Prologue for the header $f(p_1, \dots, p_m)$ using callee-saves	194
Fig. 9.7	Epilogue for the instruction RETURN <i>result</i> using callee-saves	195
Fig. 9.8	Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using callee-saves	195
Fig. 9.9	Possible division of registers for a 16-register architecture	196
Fig. 9.10	Activation record layout for the register division shown in Fig. 9.9	197
Fig. 9.11	Prologue for the header $f(p_1, \dots, p_m)$ using the register division shown in Fig. 9.9	197
Fig. 9.12	Epilogue for the instruction RETURN <i>result</i> using the register division shown in Fig. 9.9	197
Fig. 9.13	Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using the register division shown in Fig. 9.9	198

Fig. 9.14 Variable capture when inlining 206

Fig. 9.15 Renaming variables when inlining 206

Fig. 9.16 Recursive inlining 207

Fig. 10.1 Gen and kill sets for available assignments 221

Fig. 10.2 Example code for available-assignments analysis 222

Fig. 10.3 *pred*, *gen* and *kill* for the program in Fig. 10.2 223

Fig. 10.4 Fixed-point iteration for available-assignment analysis 223

Fig. 10.5 The program in Fig. 10.2 after common subexpression elimination 224

Fig. 10.6 *gen* and *kill* sets for index-check elimination 227

Fig. 10.7 Intermediate code for a for-loop with index check 228

Fig. 10.8 Equations for pointer analysis 234

Fig. 10.9 First step of SSA transformation of the code in Fig. 6.16 237

Fig. 10.10 Second step of SSA transformation of the code in Fig. 6.16 237

Fig. 10.11 Third step of SSA transformation of the code in Fig. 6.16 238

Fig. 10.12 Complete SSA transformation of the code in Fig. 6.16 238

Fig. 10.13 Applying constant and copy propagation to the code in Fig. 10.12 240

Fig. 11.1 Incrementalisation of nested loop 251

Fig. 11.2 Eliminating weakly dead variables 252

Fig. 12.1 Two apply functions written in C 258

Fig. 12.2 A C++ template definition 258

Fig. 12.3 Two declarations generated from a template 258

Fig. 12.4 Example heap object representation 262

Fig. 12.5 Testing if a value is a pointer 263

Fig. 12.6 Example freelist object representation 263

Fig. 12.7 Code for allocating an object 264

Fig. 12.8 Code for manual freeing of objects 265

Fig. 12.9 Code for the mark phase 266

Fig. 12.10 Code for the sweep phase 267

Fig. 12.11 Syntax for patterns, rules, and match expressions 269

Chapter 1

Lexical Analysis



I am not yet so lost in lexicography as to forget that words are the daughters of earth, and that things are the sons of heaven. Language is only the instrument of science, and words are but the signs of ideas.

Samuel Johnson (1709–1784)

The word “lexical” in the traditional sense means “pertaining to words”. In terms of programming languages, *words* are entities like variable names, numbers, keywords etc. Such word-like entities are traditionally called *tokens*.

A *lexical analyser*, also called a *lexer* or *scanner*, will as input take a string of individual letters and divide this string into a sequence of classified tokens. Additionally, it will filter out whatever separates the tokens (the so-called *white-space*), i.e., lay-out characters (spaces, newlines etc.) and comments.

The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

- **Efficiency:** A specialised lexer may do the simple parts of the work faster than the parser, which uses more general methods, can. Furthermore, the size of a system that is split in two phases may be smaller than a combined system. This may seem paradoxical but, as we shall see, there is a non-linear factor involved which may make a separated system smaller than a combined system.
- **Modularity:** The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.
- **Tradition:** Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.

It is usually not terribly difficult to write a lexer by hand: You first read past initial white-space, then you, in sequence, test to see if the next token is a keyword, a

number, a variable or whatnot. However, this is not a very good way of handling the problem: You may read the same part of the input repeatedly while testing each possible token, and in some cases it may not be clear where one token ends and the next begins. Furthermore, a handwritten lexer may be complex and difficult to maintain. Hence, lexers are normally constructed by *lexer generators*, that transform (somewhat) human-readable specifications of tokens and white-space into efficient programs.

We will see the same general strategy in the chapter about syntax analysis: Specifications in a well-defined human-readable notation are transformed into efficient programs.

For lexical analysis, specifications are traditionally written using *regular expressions*: An algebraic notation for describing sets of strings. The generated lexers are in a class of extremely simple programs called *finite automata*.

This chapter will describe regular expressions and finite automata, their properties and how regular expressions can be converted to finite automata. Finally, we discuss some practical aspects of lexer generators.

1.1 Regular Expressions

The set of all integer constants or the set of all variable names are examples of sets of strings, where the individual digits or letters used to form these constants or names are taken from a particular *alphabet*, i.e., a set of characters. A set of strings is called a *language*. For integers, the alphabet consists of the digits 0–9 and for variable names, the alphabet contains both letters and digits (and perhaps a few other characters, such as hyphens and underscores).

Given an alphabet, we will describe sets of strings over this alphabet by *regular expressions*, an algebraic notation that is compact and relatively easy for humans to use and understand. The idea is that regular expressions that describe simple sets of strings can be combined to form bigger regular expressions that describe more complex sets of strings. Regular expressions are often called “regexps” for short.

When talking about regular expressions, we will use the letters r , s and t in italics to denote unspecified regular expressions. When letters stand for themselves (i.e., in regular expressions that describe strings that use these letters) we will use typewriter font, e.g., `a` or `b`. The letters u , v and w in italics will be used to denote unspecified single strings, i.e., members of some language. As an example, `abw` denotes any string starting with `ab`. When we say, e.g., “The regular expression `s`” (note the typewriter font) we mean the regular expression that describes a single one-letter string “`s`”, but when we say “The regular expression s ” (note the italics), we mean a regular expression of any form which we just happen to call s . We use the notation $L(s)$ to denote the language (i.e., set of strings) described by the regular expression s . For example, $L(a)$ is the set {“`a`”}.

To find $L(s)$ for a given regular expression s , we use *derivation*: Rules that rewrite a regular expression into a string of letters. These rules allow a single regular expression

to be rewritten into several different strings, so $L(s)$ is the set of strings that s can be rewritten to using these rules. $L(s)$ is often an infinite set, but each string in the set is finite and can be obtained by a finite number of derivation steps. Figure 1.1 shows the different forms of regular expression, the derivation rules for these, and an informal description of what the regular expression forms mean. Note that we use a double arrow (\Rightarrow) to denote derivation. In addition to the specific derivation rules in Fig. 1.1, we also use some general rules to make derivation reflexive and transitive:

$$\begin{array}{ll}
 s \Rightarrow s & \text{Derivation is reflexive} \\
 r \Rightarrow t \text{ if } r \Rightarrow s \text{ and } s \Rightarrow t & \text{Derivation is transitive}
 \end{array}$$

Note that, while we use the same notation for concrete strings and regular expressions denoting one-string languages, the context will make it clear which is meant. We will often show strings and sets of strings without using quotation marks, e.g., write $\{a, bb\}$ instead of $\{\text{"a"}, \text{"bb"}\}$. When doing so, we sometimes use ϵ to denote the empty string, so the derivation $s^* \Rightarrow$ shown in Fig. 1.1 can also be written as $s^* \Rightarrow \epsilon$.

We can use the derivation rules to find the language for a regular expression. As an example, $L(a(b|c)) = \{ab, ac\}$ because $a(b|c) \Rightarrow a(b) = ab$ and $a(b|c) \Rightarrow a(c) = ac$. $L((a|b)^*)$ is infinite and contains any sequence of a s and b s, including the empty sequence. For example, the string ab is in $L((a|b)^*)$ because $(a|b)^* \Rightarrow (a|b)(a|b)^* \Rightarrow a(a|b)^* \Rightarrow a(a|b)(a|b)^* \Rightarrow ab(a|b)^* \Rightarrow ab$.

Parentheses and Precedence Rules

When we use the symbols above to construct composite regular expressions such as $a|ab^*$, it is not a priori clear how the different subexpressions are grouped. We will sometimes (like we did above) use parentheses to make the grouping of symbols explicit such as in $(a|(ab))^*$. Additionally, we use precedence rules, similar to the algebraic convention that multiplication binds stronger than additions, so $3 + 4 \times 5$

Regexp	Derivation rules	Informal description
a		The one-letter string a . No derivation rule, as it is already a string.
ϵ	$\epsilon \Rightarrow$	Denotes and reduces to the empty string.
$s t$	$s t \Rightarrow s$ $s t \Rightarrow t$	Either s or t . Note that this allows multiple different derivations.
st	$st \Rightarrow s't'$, if $s \Rightarrow s'$ and $t \Rightarrow t'$	Something derived from s followed by something derived from t .
s^*	$s^* \Rightarrow$ $s^* \Rightarrow s(s^*)$	A concatenation of any number (including 0) of strings derived from s . The number depends on how many times the second rule is used.

Fig. 1.1 Regular expressions and their derivation

is equivalent to $3 + (4 \times 5)$ and not $(3 + 4) \times 5$. For regular expressions, we use the following conventions: $*$ binds tighter than concatenation, which binds tighter than alternative ($|$). The example $a|a|b^*$ from above is, hence, equivalent to $a|(a|b^*)$.

The $|$ operator is associative and commutative. Concatenation is associative (but obviously not commutative) and distributes over $|$. Figure 1.2 shows these and other algebraic properties of regular expressions, including properties of some of the shorthands introduced below.

Suggested exercise: 1.1.

1.1.1 Shorthands

While the constructions in Fig. 1.1 suffice to describe e.g., number strings and variable names, we will often use extra shorthands for convenience. For example, if we want to describe non-negative integer constants, we can do so by saying that a number constant is a sequence of one or more digits, which is expressed by the regular expression

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

The large number of different digits makes this expression rather verbose. It gets even worse when we get to variable names, where we must enumerate all alphabetic letters (in both upper and lower case).

Hence, we introduce a shorthand for sets of letters. A sequence of letters enclosed in square brackets represents the set of these letters. For example, we use $[ab01]$ as a shorthand for $a|b|0|1$. Additionally, we can use interval notation to abbreviate $[0123456789]$ to $[0-9]$. We can combine several intervals within one bracket and for example write $[a-zA-Z]$ to denote all alphabetic letters in both lower and upper case.

When using intervals, we must be aware of the ordering for the symbols involved. For the digits and letters used above, there is usually no confusion. However, if we

Fig. 1.2 Some algebraic properties of regular expressions

$(r s)t = r s t = r (s t)$	$ $ is associative.
$s t = t s$	$ $ is commutative.
$s s = s$	$ $ is idempotent.
$s^? = s \epsilon$	by definition.
$(rs)t = rst = r(st)$	concatenation is associative.
$s\epsilon = s = \epsilon s$	ϵ is a neutral element for concatenation.
$r(s t) = rs rt$	concatenation distributes over $ $.
$(r s)t = rt st$	concatenation distributes over $ $.
$(s^*)^* = s^*$	$*$ is idempotent.
$s^*s^* = s^*$	0 or more twice is still 0 or more.
$ss^* = s^+ = s^*s$	by definition.
$(s^+)^+ = s^+$	$+$ is idempotent.
$s^+s^* = s^+ = s^*s^+$	still just at least 1

write, e.g., $[0-z]$ it is not immediately clear what is meant. When using such notation in lexer generators, a character set encoding such as ASCII, ISO 8859-1, or UTF-8 is usually implied, so the symbols are ordered as defined by these encodings. To avoid confusion, we will in this book use the interval notation only for intervals of digits or alphabetic letters.

Getting back to the example of integer constants above, we can now write this much shorter as $[0-9][0-9]^*$.

Since s^* denotes *zero or more* occurrences of s , we needed to write the set of digits twice to describe that *one or more* digits are required. Such non-zero repetition is quite common, so we introduce another shorthand, s^+ , to denote one or more occurrences of s . With this notation, we can abbreviate our description of integers to $[0-9]^+$. On a similar note, it is common that we can have zero or one occurrence of something (e.g., an optional sign to a number). Hence we introduce the shorthand $s?$ for $s|\epsilon$. The shorthand symbols $^+$ and $?$ bind with the same precedence as * .

We must stress that these shorthands are just that. They do not allow more languages to be described, they just make it possible to describe some languages more compactly. In the case of s^+ , it can even make an exponential difference: If $^+$ is nested n deep, recursive expansion of s^+ to ss^* yields $2^n - 1$ occurrences of * in the expanded regular expression. For example, $((a^+b)^+c)^+$ expands to $aa^*b(aa^*b)^*c(aa^*b(aa^*b)^*c)^*$.

1.1.2 Examples

We have already seen how we can describe non-negative integer constants using regular expressions. Here are a few examples of other typical programming language elements:

Keywords. A keyword like `if` is described by a regular expression that looks exactly like that keyword, e.g., the regular expression `if` (which is the concatenation of the two regular expressions `i` and `f`).

Variable names. In the programming language C, a variable name consists of letters, digits and the underscore symbol and it must begin with a letter or underscore. This can be described by the regular expression $[a-zA-Z_][a-zA-Z_0-9]^*$.

Integers. An integer constant is an optional sign followed by a non-empty sequence of digits: $[+-]?[0-9]^+$. In some languages, a signed constant is not a single token, but a concatenation of two tokens: the sign and an unsigned number constant. This will usually allow whitespace between the sign and the number, which is not possible with the above.

Floats. In C, a floating-point constant can have an optional sign. After this, the mantissa part is described as a sequence of digits followed by a decimal point and then another sequence of digits. Either one (but not both) of the digit sequences can be empty. Finally, there is an optional exponent part, which is the letter `e` (in upper or lower case) followed by an (optionally signed) integer constant. If there

is an exponent part to the constant, the mantissa part can be written as an integer constant (i.e., without the decimal point). Some examples:

3.14 -3. .23 3e+4 11.22e-3.

This rather involved format can be described by the following regular expression:

$$[+-]?((([0-9]^+.[0-9]^*|.[0-9]^+)([eE][+-]?[0-9]^+)?| [0-9]^+[eE][+-]?[0-9]^+)$$

This regular expression is complicated by the fact that the exponent is optional if the mantissa contains a decimal point, but not if it does not (as that would make the number an integer constant). We can make the description simpler if we make the regular expression for floats also include integers, and instead use other means of distinguishing integers from floats (see Sect. 1.8 for details). If we do this, the regular expression can be simplified to

$$[+-]?((([0-9]^+([0-9]^*)?)?.[0-9]^+)([eE][+-]?[0-9]^+)?)$$

Some languages require digits on both sides of the decimal point (if there is a decimal point). This simplifies the description considerably, as there are fewer special cases:

$$[+-]?((([0-9]^+([0-9]^+)?)([eE][+-]?[0-9]^+)?)$$

String constants. A string constant starts with a quotation mark followed by a sequence of symbols and finally another quotation mark. There are usually some restrictions on the symbols allowed between the quotation marks. For example, line-feed characters or quotes are typically not allowed, though these may be represented by special “escape” sequences of other characters, such as “\n\n” for a string containing two line-feeds. As a (much simplified) example, we can by the following regular expression describe string constants where the allowed symbols are alphanumeric characters and sequences consisting of the backslash symbol followed by a letter (where each such pair is intended to represent a non-alphanumeric symbol):

$$“([a-zA-Z0-9]|\[a-zA-Z]) *”$$

Suggested exercises: 1.2, 1.11(a).

1.2 Nondeterministic Finite Automata

In our quest to transform regular expressions into efficient programs, we use a stepping stone: Nondeterministic finite automata. By their nondeterministic nature, these are not quite as close to “real machines” as we would like, so we will later see how these can be transformed into *deterministic* finite automata, which are easily and efficiently executable on normal hardware.

A finite automaton is, in the abstract sense, a machine that has a finite number of *states* and a finite number of *transitions* between pairs of states. A transition between two states is usually labelled by a character from the input alphabet, but we will also use transitions marked with ε , the so-called *epsilon transitions*.

A finite automaton can be used to decide if an input string is a member in some particular set of strings. To do this, we select one of the states of the automaton as the *starting state*. We start in this state, and in each step we can do one of the following:

- Follow an epsilon transition to another state, or
- Read a character from the input and follow a transition labelled by that character.

When all characters from the input are read, we see if the current state is marked as being *accepting*. If this is the case, the string we have read from the input is in the language defined by the automaton. Otherwise, it is not.

At each step, we may have a choice of several actions: We can choose between either an epsilon transition or a transition on an alphabet character, and if there are several transitions with the same symbol, we can choose between these. This makes the automaton *nondeterministic*, as the choice of action is not determined solely by looking at the current state and the next input character. It may be that some choices lead to an accepting state while others do not. This does, however, not mean that the string is sometimes in the language and sometimes not: We will include a string in the language if it is *possible* to make a sequence of choices that makes the string lead to an accepting state.

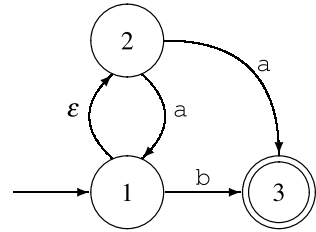
You can think of it as solving a maze with symbols written in the corridors. If you can find the exit while walking over the letters of the string in the correct order, the string is recognised by the maze.

We can formally define a nondeterministic finite automaton by:

Definition 1.1 A *nondeterministic finite automaton* consists of a set S of states. One of these states, $s_0 \in S$, is called the *starting state* of the automaton, and a subset $F \subseteq S$ of the states are *accepting states*. Additionally, we have a set T of *transitions*. Each transition t connects a pair of states s_1 and s_2 and is labelled with a symbol, which is either a character c from an alphabet Σ , or the symbol ε , which indicates an *epsilon-transition*. A transition from state s to state t on the symbol c is written as $s^c t$.

Starting states are sometimes called *initial states* and accepting states can also be called *final states* (which is why we use the letter F to denote the set of accepting

Fig. 1.3 Example of an NFA



states). We use the abbreviations FA for finite automaton, NFA for nondeterministic finite automaton and (later in this chapter) DFA for deterministic finite automaton.

We will mostly use a graphical notation to describe finite automata. States are denoted by circles, optionally containing a number or name that identifies the state. This name or number has, however, no operational significance, it is solely used for identification purposes. Accepting states are denoted by using a double circle instead of a single circle. The initial state is marked by an unlabelled arrow pointing to it from outside the automaton.

A transition is denoted by an arrow connecting two states. Near its midpoint, the arrow is labelled by the symbol (possibly ε) that triggers the transition. Note that the arrow that marks the initial state is *not* a transition and is, hence, not labelled by a symbol.

Repeating the maze analogue, the circles (states) are rooms and the arrows (transitions) are one-way corridors. The double circles (accepting states) are exits, while the unlabelled arrow pointing to the starting state is the entrance to the maze.

Figure 1.3 shows an example of a nondeterministic finite automaton having three states. State 1 is the starting state, and state 3 is accepting. There is an epsilon-transition from state 1 to state 2, transitions on the symbol *a* from state 2 to states 1 and 3, and a transition on the symbol *b* from state 1 to state 3. This NFA recognises the language described by the regular expression $a^*(a|b)$. As an example, the string *aab* is recognised by the following sequence of transitions:

from	to	by
1	2	ε
2	1	<i>a</i>
1	2	ε
2	1	<i>a</i>
1	3	<i>b</i>

At the end of the input we are in state 3, which is accepting. Hence, the string is accepted by the NFA. You can check this by placing a coin at the starting state and follow the transitions by moving the coin.

Note that we sometimes have a choice of several transitions. If we are in state 2 and the next symbol is an *a*, we can, when reading this, either go to state 1 or to state 3. Likewise, if we are in state 1 and the next symbol is a *b*, we can either read this and go to state 3, or we can use the epsilon transition to go directly to state

2 without reading anything. If we, in the example above, had chosen to follow the a -transition to state 3 instead of state 1, we would have been stuck: We would have no legal transition, and yet we would not be at the end of the input (and even if we were, we are not in an accepting state). But, as previously stated, it is enough that there *exists* a path leading to acceptance, so the string aab is accepted by the NFA.

A program that decides if a string is accepted by a given NFA will have to check all possible paths to see if *any* of these accepts the string. This requires either backtracking until a successful path found, or simultaneously following all possible paths. Both of these methods are too time-consuming to make NFAs suitable for efficient recognisers. We will, hence, use NFAs only as a stepping stone between regular expressions and the more efficient DFAs. We use this stepping stone because it makes the construction simpler than direct construction of a DFA from a regular expression.

1.3 Converting a Regular Expression to an NFA

We will construct an NFA *compositionally* from a regular expression, i.e., we will construct the NFA for a composite regular expression from the NFAs constructed from its subexpressions.

To be precise, we will from each subexpression construct an *NFA fragment* and then combine these fragments into bigger fragments. A fragment is not a complete NFA, so we complete the construction by adding the necessary components to make a complete NFA.

An NFA fragment consists of a number of states with transitions between these and additionally two incomplete transitions: One pointing into the fragment and one pointing out of the fragment. The incoming half-transition is not labelled by a symbol, but the outgoing half-transition is labelled by either ϵ or an alphabet symbol. These half-transitions are the entry and exit to the fragment and are used to connect it to other fragments or additional “glue” states.

Construction of NFA fragments for regular expressions is shown in Fig. 1.4. The construction follows the structure of the regular expression by first making NFA fragments for the subexpressions, and then joining these to form an NFA fragment for the whole regular expression. The NFA fragments for the subexpressions are shown as dotted ovals with the incoming half-transition on the left and the outgoing half-transition on the right. The symbol on the outgoing half-transition is not shown when an NFA fragment is shown as a dotted oval (it is “hidden” inside the oval).

When an NFA fragment has been constructed for the whole regular expression, the construction is completed by connecting the outgoing half-transition to an accepting state. The incoming half-transition serves to identify the starting state of the completed NFA. Note that, even though we allow an NFA to have several accepting states, an NFA constructed using this method will have only one: the one added at the end of the construction.

An NFA constructed this way for the regular expression $(a|b)^*ac$ is shown in Fig. 1.5. We have numbered the states for future reference.

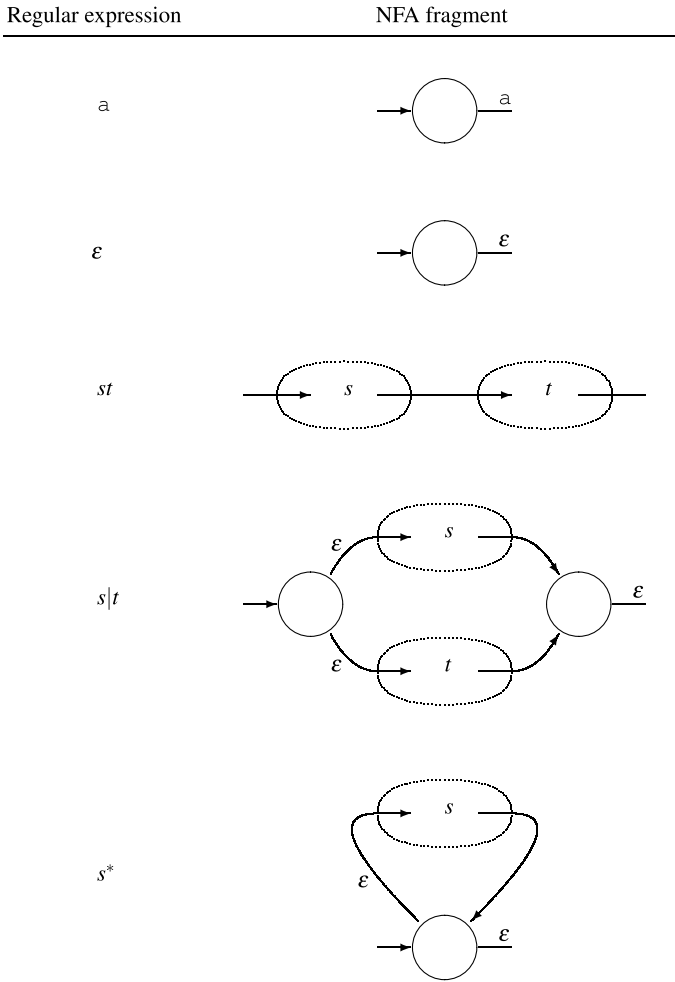


Fig. 1.4 Constructing NFA fragments from regular expressions

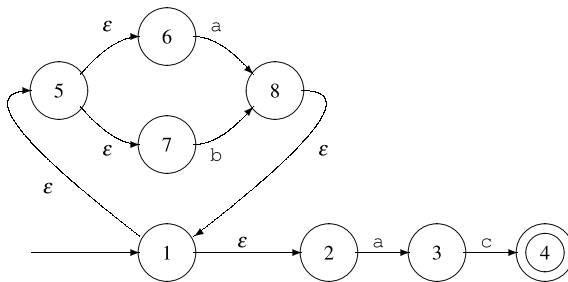
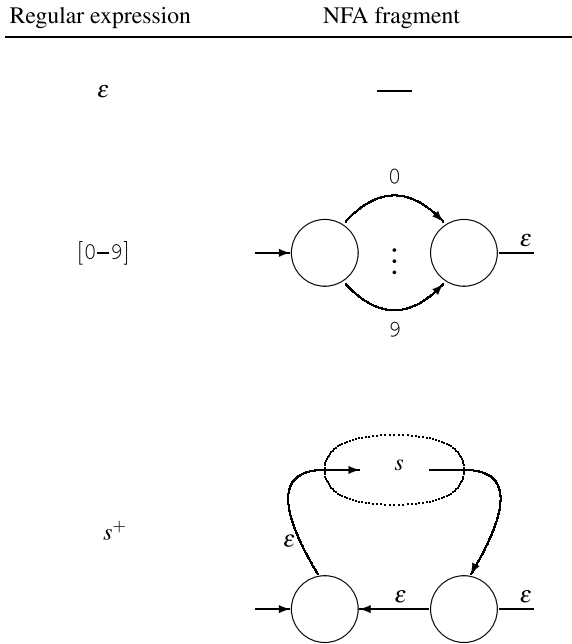


Fig. 1.5 NFA for the regular expression $(a|b)^*ac$

Fig. 1.6 Optimised NFA construction for regular expression shorthands



1.3.1 Optimisations

We can use the construction in Fig. 1.4 for any regular expression by expanding out all shorthand, e.g. converting s^+ to ss^* , $[0-9]$ to $0|1|2|\dots|9$, $s^?$ to $s|\epsilon$, and so on. However, this will result in very large NFAs for some expressions, so we use a few optimised constructions for the shorthands, as shown in Fig. 1.6. Additionally, we show an alternative construction for the regular expression ϵ . This construction does not quite follow the formula used in Fig. 1.4, as it does not have two half-transitions. Rather, the line-segment notation is intended to indicate that the NFA fragment for ϵ just connects the half-transitions of the NFA fragments that it is combined with. In the construction for $[0-9]$, the vertical ellipsis is meant to indicate that there is a transition for each of the digits in $[0-9]$. This construction generalises in the obvious way to other sets of characters, e.g., $[a-zA-Z0-9]$. We have not shown a special construction for $s^?$ as $s|\epsilon$ will do fine when we use the optimised construction for ϵ .

As an example, an NFA for $[0-9]^+$ is shown in Fig. 1.7. Note that while this is *optimised*, it is not *optimal*. You can (in several different ways) make an NFA for this language using only two states.

Suggested exercises: 1.3(a), 1.11(b).

Fig. 1.7 Optimised NFA for $[0-9]^+$

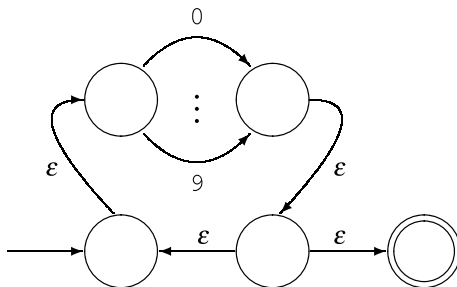
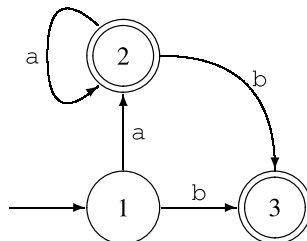


Fig. 1.8 Example of a DFA



1.4 Deterministic Finite Automata

Nondeterministic automata are, as mentioned earlier, not quite as close to “the machine” as we would like. Hence, we now introduce a more restricted form of finite automaton: The deterministic finite automaton, or DFA for short. DFAs are special cases of NFAs that obey a number of additional restrictions:

- There are no epsilon-transitions.
- There may not be two identically labelled transitions out of the same state.

This means that we never have a choice of several next-states: The state and the next input symbol uniquely determine the transition (or lack of same). This is why these automata are called *deterministic*. Figure 1.8 shows a DFA equivalent to the NFA in Fig. 1.3. Using the maze analogy, finding an exit is easy, as you are never in doubt about which corridor to follow.

The transition relation of a DFA is a partial function, and we often write it as a function: $move(s, c)$ is the state (if any) that is reached from state s by a transition on the symbol c . If there is no such transition, $move(s, c)$ is undefined.

It is very easy to implement a DFA on a computer: A two-dimensional table can be cross-indexed by state and symbol to yield the next state (or an indication that there is no transition), essentially implementing the $move$ function by table lookup. Another (one-dimensional) table can indicate which states are accepting.

DFAs have the same expressive power as NFAs: A DFA is a special case of NFA, and any NFA can (as we shall shortly see) be converted to an equivalent DFA. However, the benefit of deterministic transitions comes at a cost: The resulting

DFA can be exponentially larger than the NFA (see Sect. 1.9). In practice (i.e., when describing tokens for a programming language) the increase in size is usually modest, which is why most lexical analysers are based on DFAs.

Suggested exercises: 1.8(a, b), 1.9.

1.5 Converting an NFA to a DFA

As promised, we will show how NFAs can be converted to DFAs such that we, by combining this with the conversion of regular expressions to NFAs shown in Sect. 1.3, can convert any regular expression to a DFA.

The conversion is done by simulating all possible transitions in an NFA at the same time. This means that we operate with sets of NFA states: When we have several choices of a next state, we take all of the choices simultaneously and form a set of the possible next-states. Given a set of NFA states and a symbol, we follow *all* transitions on that symbol from *all* states in the set, which gives us a new set of NFA states. So we get transitions from sets of NFA states to sets of NFA states. The transitions are deterministic because we from one set of NFA states and one symbol have exactly one (possibly empty) set of NFA states that the transition moves to. The idea is that different *sets* of NFA states become different *single* states in the DFA that we construct.

Epsilon-transitions complicate the construction a bit: Whenever we are in an NFA state with an outgoing epsilon-transition, we can always choose to follow the epsilon-transition without reading any symbol. Hence, given a symbol, a next-state can be found by either following a transition with that symbol, or by first doing any number of epsilon-transitions and then a transition with the symbol. We handle this in the construction by extending sets of NFA states by adding all NFA states that can be reached from states in the set using only epsilon-transitions. We define the *epsilon-closure* of a set of NFA states as the set extended with all NFA states that can be reached from these using any number of epsilon-transitions. More formally:

Definition 1.2 Given a set M of NFA states, we define ε -closure(M) to be the least (in terms of the subset relation) set X that is a solution to the set equation

$$X = M \cup \{t \mid s \in X \text{ and } s^\varepsilon t \in T\}$$

Where T is the set of transitions in the NFA.

We will later on see several examples of *set equations* like the one above, so we use some time to discuss how such equations can be solved.

1.5.1 Solving Set Equations

The following is a very brief description of how to solve set equations like the above. If you find it confusing, you can read the Appendix and in particular Sect. A.4 first.

In general, a set equation over a single set-valued variable X has the form

$$X = F(X)$$

where F is a function from sets to sets. Not all such equations are solvable, so we will restrict ourselves to special cases, which we will describe below. We will use calculation of epsilon-closure as the driving example.

In Definition 1.2, we must find a set X that solves the equation

$$X = M \cup \{t \mid s \in X \text{ and } s^\epsilon t \in T\}$$

To cast this equation into the form $X = F(X)$ for a function f , we define F_M to be

$$F_M(X) = M \cup \{t \mid s \in X \text{ and } s^\epsilon t \in T\}$$

There may be several solutions to the equation $X = F_M(X)$. For example, if the NFA has a pair of states that connect to each other by epsilon transitions, adding this pair to a solution that does not already include the pair will create a new solution. The epsilon-closure of M is the *least* solution to the equation (i.e., the smallest set X that satisfies the equation).

F_M has a property that is essential to our solution method: If $X \subseteq Y$ then $F_M(X) \subseteq F_M(Y)$. We say that F_M is *monotonic*.

When we have an equation of the form $X = F(X)$ and F is monotonic, we can find the least solution to the equation in the following way: We first guess that the solution is the empty set (written as \emptyset) and check to see if we are right: We compare \emptyset with $F(\emptyset)$. If these are equal, we are done and \emptyset is the solution. If not, we use the following properties:

- The least solution S to the equation satisfies $S = F(S)$
- $\emptyset \subseteq S$ implies that $F(\emptyset) \subseteq F(S)$

to conclude that $F(\emptyset) \subseteq S$. Hence, $F(\emptyset)$ is either S or a subset of S , so we can use it as a new guess. We now form the chain

$$\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq \dots$$

If at any point an element in the sequence is identical to the previous, we have a fixed-point, i.e., a set S such that $S = F(S)$. This fixed-point of the sequence will be the least (in terms of set inclusion) solution to the equation. This is not difficult to verify, but we will omit the details. Since we are iterating a function until we reach a fixed-point, we call this process *fixed-point iteration*.

If we are working with sets over a finite domain (e.g., sets of NFA states from a specific NFA), we *will* eventually reach a fixed-point, as there can be no infinite chain of strictly increasing sets.

We can use this method for calculating the epsilon-closure of the set $\{1\}$ with respect to the NFA shown in Fig. 1.5. Since we want to find $\epsilon\text{-closure}(\{1\})$, $M = \{1\}$,

so $F_M = F_{\{1\}}$. We start by guessing that X is the empty set:

$$\begin{aligned} F_{\{1\}}(\emptyset) &= \{1\} \cup \{t \mid s \in \emptyset \text{ and } s^\varepsilon t \in T\} \\ &= \{1\} \end{aligned}$$

As $\emptyset \neq \{1\}$, we continue.

$$\begin{aligned} F_{\{1\}}(F_{\{1\}}(\emptyset)) &= F_{\{1\}}(\{1\}) \\ &= \{1\} \cup \{t \mid s \in \{1\} \text{ and } s^\varepsilon t \in T\} \\ &= \{1\} \cup \{2, 5\} = \{1, 2, 5\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(F_{\{1\}}(F_{\{1\}}(\emptyset))) &= F_{\{1\}}(\{1, 2, 5\}) \\ &= \{1\} \cup \{t \mid s \in \{1, 2, 5\} \text{ and } s^\varepsilon t \in T\} \\ &= \{1\} \cup \{2, 5, 6, 7\} = \{1, 2, 5, 6, 7\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(F_{\{1\}}(F_{\{1\}}(F_{\{1\}}(\emptyset)))) &= F_{\{1\}}(\{1, 2, 5, 6, 7\}) \\ &= \{1\} \cup \{t \mid s \in \{1, 2, 5, 6, 7\} \text{ and } s^\varepsilon t \in T\} \\ &= \{1\} \cup \{2, 5, 6, 7\} = \{1, 2, 5, 6, 7\} \end{aligned}$$

We have now reached a fixed-point and found our solution. Hence, we conclude that ε -closure($\{1\}$) = $\{1, 2, 5, 6, 7\}$.

We have done a good deal of repeated calculation in the iteration above: We have calculated the epsilon-transitions from state 1 three times and those from state 2 and 5 twice each. We can make an optimised fixed-point iteration by exploiting that the function is not only monotonic, but also *distributive*: $F(X \cup Y) = F(X) \cup F(Y)$. This means that, when we during the iteration add elements to our set, we in the next iteration need only calculate F for the new elements and add the result to the set. In the example above, we get

$$\begin{aligned} F_{\{1\}}(\emptyset) &= \{1\} \cup \{t \mid s \in \emptyset \text{ and } s^\varepsilon t \in T\} \\ &= \{1\} \\ F_{\{1\}}(\{1\}) &= \{1\} \cup \{t \mid s \in \{1\} \text{ and } s^\varepsilon t \in T\} \\ &= \{1\} \cup \{2, 5\} = \{1, 2, 5\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(\{1, 2, 5\}) &= F_{\{1\}}(\{1\}) \cup F_{\{1\}}(\{2, 5\}) \\ &= \{1, 2, 5\} \cup (\{1\} \cup \{t \mid s \in \{2, 5\} \text{ and } s^\varepsilon t \in T\}) \\ &= \{1, 2, 5\} \cup (\{1\} \cup \{6, 7\}) = \{1, 2, 5, 6, 7\} \end{aligned}$$

$$\begin{aligned} F_{\{1\}}(\{1, 2, 5, 6, 7\}) &= F_{\{1\}}(\{1, 2, 5\}) \cup F_{\{1\}}(\{6, 7\}) \\ &= \{1, 2, 5, 6, 7\} \cup (\{1\} \cup \{t \mid s \in \{6, 7\} \text{ and } s^\varepsilon t \in T\}) \\ &= \{1, 2, 5, 6, 7\} \cup (\{1\} \cup \emptyset) = \{1, 2, 5, 6, 7\} \end{aligned}$$

We can use this principle to formulate a *work-list algorithm* for finding the least fixed-point for an equation over a distributive function F . The idea is that we step-by-step build a set that eventually becomes our solution. In the first step, we calculate $F(\emptyset)$. The elements in this initial set are *unmarked*. In each subsequent step, we take an unmarked element x from the set, mark it and add $F(\{x\})$ (unmarked) to the set. Note that if an element already occurs in the set (marked or not), it is not added again. When, eventually, all elements in the set are marked, we are done.

This is perhaps best illustrated by an example (the same as before). We start by calculating $F_{\{1\}}(\emptyset) = \{1\}$. The element 1 is unmarked, so we pick this, mark it and calculate $F_{\{1\}}(\{1\})$ and add the new elements 2 and 5 to the set. As we continue, we get this sequence of sets:

$$\begin{array}{c}
 \{1\} \\
 \checkmark \\
 \{1, 2, 5\} \\
 \checkmark \quad \checkmark \\
 \{1, 2, 5\} \\
 \checkmark \quad \checkmark \quad \checkmark \\
 \{1, 2, 5, 6, 7\} \\
 \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \\
 \{1, 2, 5, 6, 7\} \\
 \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \\
 \{1, 2, 5, 6, 7\}
 \end{array}$$

Since all elements in the last set are marked, this is a solution to the equation.

We will later also need to solve *simultaneous equations* over sets, i.e., several equations over several sets. These can also be solved by fixed-point iteration in the same way as single equations, though the work-list version of the algorithm becomes a bit more complicated.

1.5.2 The Subset Construction

After this brief detour into the realm of set equations, we are now ready to continue with our construction of DFAs from NFAs. The construction is called *the subset construction*, as each state in the DFA is a subset of the states from the NFA.

Algorithm 1.3 (*The subset construction*) Given an NFA N with states S , starting state $s_0 \in S$, accepting states $F \subseteq S$, transitions T , and alphabet Σ , we construct an equivalent DFA D with states S' , starting state s'_0 , accepting states F' , and a transition function (called “*move*”) by:

$$\begin{aligned}
 s'_0 &= \varepsilon\text{-closure}(\{s_0\}) \\
 \text{move}(s', c) &= \varepsilon\text{-closure}(\{t \mid s \in s' \text{ and } s^c t \in T\}) \\
 S' &= \{s'_0\} \cup \{\text{move}(s', c) \mid s' \in S', c \in \Sigma\} \\
 F' &= \{s' \in S' \mid s' \cap F \neq \emptyset\}
 \end{aligned}$$

The DFA uses the same alphabet Σ as the NFA.

A little explanation:

- The starting state s'_0 of the DFA is the epsilon-closure of the set containing just the starting state s_0 of the NFA, i.e., the states that are reachable from s_0 solely by epsilon-transitions.
- A transition in the DFA on a symbol c is done by finding the set s' of NFA states that comprise the DFA state, following all transitions on c in the NFA from all NFA states s in s' , combining the resulting sets of NFA states, and finally closing this under epsilon transitions.
- The set S' of states in the DFA is the set of DFA states that can be reached from s'_0 using the *move* function. S' is defined as a set equation which can be solved as described in Sect. 1.5.1.
- A state s' in the DFA is an accepting state if *at least one* of the NFA states in s' is accepting.

As an example, we will convert the NFA in Fig. 1.5 to a DFA.

The initial state in the DFA is $\varepsilon\text{-closure}(\{1\})$, which we have already calculated to be $s'_0 = \{1, 2, 5, 6, 7\}$. This is now entered into the set S' of DFA states as unmarked (following the work-list algorithm from Sect. 1.5.1).

We now pick an unmarked element from the uncompleted S' . We have only one choice: s'_0 . We now mark this and calculate the transitions for it. We get

$$\begin{aligned} \text{move}(s'_0, a) &= \varepsilon\text{-closure}(\{t \mid s \in \{1, 2, 5, 6, 7\} \text{ and } s^a t \in T\}) \\ &= \varepsilon\text{-closure}(\{3, 8\}) \\ &= \{3, 8, 1, 2, 5, 6, 7\} \\ &= s'_1 \end{aligned}$$

$$\begin{aligned} \text{move}(s'_0, b) &= \varepsilon\text{-closure}(\{t \mid s \in \{1, 2, 5, 6, 7\} \text{ and } s^b t \in T\}) \\ &= \varepsilon\text{-closure}(\{8\}) \\ &= \{8, 1, 2, 5, 6, 7\} \\ &= s'_2 \end{aligned}$$

$$\begin{aligned} \text{move}(s'_0, c) &= \varepsilon\text{-closure}(\{t \mid s \in \{1, 2, 5, 6, 7\} \text{ and } s^c t \in T\}) \\ &= \varepsilon\text{-closure}(\{\}) \\ &= \{\} \end{aligned}$$

Note that the empty set of NFA states is not an DFA state, so there will be no transition from s'_0 on c .

We now add s'_1 and s'_2 to our incomplete S' , which now is $\{s'_0, s'_1, s'_2\}$. We now pick s'_1 , mark it and calculate its transitions:

$$\begin{aligned}
 \text{move}(s'_1, \text{a}) &= \varepsilon\text{-closure}(\{t \mid s \in \{3, 8, 1, 2, 5, 6, 7\} \text{ and } s^{\text{a}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{3, 8\}) \\
 &= \{3, 8, 1, 2, 5, 6, 7\} \\
 &= s'_1
 \end{aligned}$$

$$\begin{aligned}
 \text{move}(s'_1, \text{b}) &= \varepsilon\text{-closure}(\{t \mid s \in \{3, 8, 1, 2, 5, 6, 7\} \text{ and } s^{\text{b}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{8\}) \\
 &= \{8, 1, 2, 5, 6, 7\} \\
 &= s'_2
 \end{aligned}$$

$$\begin{aligned}
 \text{move}(s'_1, \text{c}) &= \varepsilon\text{-closure}(\{t \mid s \in \{3, 8, 1, 2, 5, 6, 7\} \text{ and } s^{\text{c}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{4\}) \\
 &= \{4\} \\
 &= s'_3
 \end{aligned}$$

We have seen s'_1 and s'_2 before, so only s'_3 is added: $\{s'_0, s'_1, s'_2, s'_3\}$. We next pick s'_2 :

$$\begin{aligned}
 \text{move}(s'_2, \text{a}) &= \varepsilon\text{-closure}(\{t \mid s \in \{8, 1, 2, 5, 6, 7\} \text{ and } s^{\text{a}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{3, 8\}) \\
 &= \{3, 8, 1, 2, 5, 6, 7\} \\
 &= s'_1
 \end{aligned}$$

$$\begin{aligned}
 \text{move}(s'_2, \text{b}) &= \varepsilon\text{-closure}(\{t \mid s \in \{8, 1, 2, 5, 6, 7\} \text{ and } s^{\text{b}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{8\}) \\
 &= \{8, 1, 2, 5, 6, 7\} \\
 &= s'_2
 \end{aligned}$$

$$\begin{aligned}
 \text{move}(s'_2, \text{c}) &= \varepsilon\text{-closure}(\{t \mid s \in \{8, 1, 2, 5, 6, 7\} \text{ and } s^{\text{c}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{\}) \\
 &= \{\}
 \end{aligned}$$

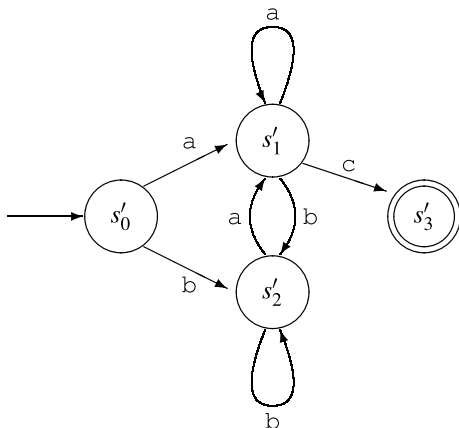
No new elements are added, so we pick the remaining unmarked element s'_3 :

$$\begin{aligned}
 \text{move}(s'_3, \text{a}) &= \varepsilon\text{-closure}(\{t \mid s \in \{4\} \text{ and } s^{\text{a}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{\}) \\
 &= \{\}
 \end{aligned}$$

$$\begin{aligned}
 \text{move}(s'_3, \text{b}) &= \varepsilon\text{-closure}(\{t \mid s \in \{4\} \text{ and } s^{\text{b}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{\}) \\
 &= \{\}
 \end{aligned}$$

$$\begin{aligned}
 \text{move}(s'_3, \text{c}) &= \varepsilon\text{-closure}(\{t \mid s \in \{4\} \text{ and } s^{\text{c}}t \in T\}) \\
 &= \varepsilon\text{-closure}(\{\}) \\
 &= \{\}
 \end{aligned}$$

Fig. 1.9 DFA constructed from the NFA in Fig. 1.5



Since all states are now marked, this completes the construction of $S' = \{s'_0, s'_1, s'_2, s'_3\}$. Only s'_3 contains the accepting NFA state 4, so this is the only accepting state of our DFA. Figure 1.9 shows the completed DFA.

Suggested exercises: 1.3(b), 1.5.

1.6 Size Versus Speed

In the above example, we get a DFA with 4 states from an NFA with 8 states. However, as the states in the constructed DFA are (nonempty) sets of states from the NFA there may potentially be $2^n - 1$ states in a DFA constructed from an n -state NFA. It is not too difficult to construct classes of NFAs that expand exponentially in this way when converted to DFAs, as we shall see in Sect. 1.9.1. Since we are mainly interested in NFAs that are constructed from regular expressions as in Sect. 1.3, we might ask ourselves if these NFAs might not be in a suitably simple class that do not risk exponential-sized DFAs. Alas, this is not the case. Just as we can construct a class of NFAs that expand exponentially, we can construct a class of regular expressions where the smallest equivalent DFAs are exponentially larger. This happens rarely when we use regular expressions or NFAs to describe tokens in programming languages, though.

It is possible to avoid the blow-up in size by operating directly on regular expressions or NFAs when testing strings for inclusion in the languages these define. However, there is a speed penalty for doing so. A DFA can be run in time $k * |v|$, where $|v|$ is the length of the input string v and k is a small constant that is independent of the size of the DFA.¹ Regular expressions and NFAs can be run in time close to $c * |N| * |v|$, where $|N|$ is the size of the NFA (or regular expression) and the

¹ If memory access is assumed to be constant time, regardless of memory size.

constant c typically is larger than k . All in all, DFAs are a lot faster to use than NFAs or regular expressions, so it is only when the size of the DFA is a real problem that one should consider using NFAs or regular expressions directly.

1.7 Minimisation of DFAs

Even though the DFA in Fig. 1.9 has only four states, it is not minimal. It is easy to see that states s'_0 and s'_2 are equivalent: Neither are accepting and they have identical transitions. We can hence collapse these states into a single state and get a three-state DFA.

DFAs constructed from regular expressions through NFAs are often non-minimal, though they are rarely very far from being minimal. Nevertheless, minimising a DFA is not terribly difficult and can be done fairly fast, so many lexer generators perform minimisation.

An interesting property of DFAs is that any regular language (a language that can be expressed by a regular expression, NFA or DFA) has a *unique* equivalent minimal DFA (if states are not labelled). Hence, we can decide equivalence of two regular expressions (or NFAs or DFAs) by converting both to minimal DFAs and compare the results.

As hinted above, minimisation of DFAs is done by collapsing equivalent states. However, deciding whether two states are equivalent is not just done by testing if their immediate transitions are identical, since transitions to different states may be equivalent if the target states turn out to be equivalent. For example, in Fig. 1.9, a transition to s'_0 (if any such existed) should be considered equivalent to a transition to s'_2 (since s'_0 and s'_2 are equivalent). Hence, we use a strategy where we first assume all states to be equivalent and then distinguish them only if we can prove them different. We use the following rules for this:

- An accepting state is *not* equivalent to a non-accepting state.
- If two states s_1 and s_2 have transitions on the same symbol c to states t_1 and t_2 that we have already proven to be different, then s_1 and s_2 are different. This also applies if only one of s_1 or s_2 have a defined transition on c .

This leads to the following algorithm.

Algorithm 1.4 (*DFA minimisation*) Given a DFA D over the alphabet Σ with states S , where $F \subseteq S$ is the set of the accepting states, we construct a minimal DFA D_{min} , where each state is a group of equivalent states from D . The groups in the minimal DFA are *consistent*: For any pair of states s_1, s_2 in a group and any symbol c , $move(s_1, c)$ and $move(s_2, c)$ are both in the same group, or both are undefined. In other words, we can not tell s_1 and s_2 apart by looking at their transitions.

We minimise the DFA D in the following way:

- (1) We start with two groups: the set of accepting states F and the set of non-accepting states $S \setminus F$. Both these groups are initially unmarked.

- (2) We pick any unmarked group G and check if it is consistent. If it is, we mark it. If G is not consistent, we split it into maximal consistent subgroups and replace G by these. A consistent subgroup is maximal if adding any other state from G to it will make it inconsistent. *All* groups (not just the newly added) are then unmarked.
- (3) If there are no unmarked groups left, we are done, and the remaining groups are all consistent, and each group will be a state of the minimal DFA. Otherwise, we go back to step 2.

The starting state of the minimal DFA is the group that contains the original starting state, and any group of accepting states is an accepting state in the minimal DFA.

The time needed for minimisation using Algorithm 1.4 depends on the strategy used for picking groups in step 2. With random choices, the worst case is quadratic in the size of the DFA, but there exist strategies for choosing groups and data structures for representing these that guarantee a worst-case time that is $O(n \times \log(n))$, where n is the number of states in the (non-minimal) DFA. In other words, the method can be implemented so it uses little more than linear time to do minimisation. We will not here go into further detail but just refer to [1] for the optimal algorithm.

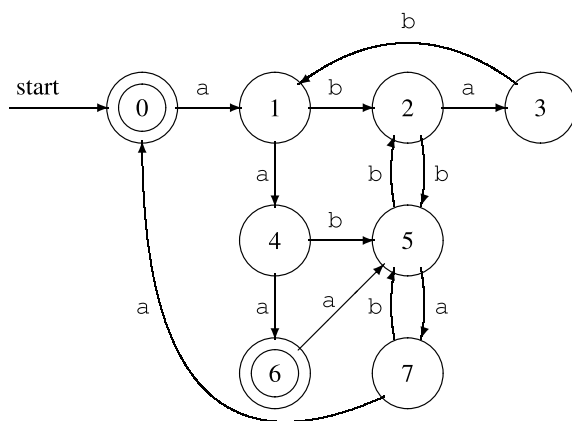
We will, however, note that we can make a slight optimisation to Algorithm 1.4: A group that consists of a single state needs never be split, so we need never select such in step 2, and we can stop when all unmarked groups are singletons.

1.7.1 Example

As an example of minimisation, take the DFA in Fig. 1.10.

We now make the initial division into two groups: The accepting and the non-accepting states.

Fig. 1.10 Non-minimal DFA



$$G_1 = \{0, 6\}$$

$$G_2 = \{1, 2, 3, 4, 5, 7\}$$

These are both unmarked. We next pick any unmarked group, say G_1 . To check if this is consistent, we make a table of its transitions:

G_1	a	b
0	G_2	–
6	G_2	–

This is consistent, so we just mark it and select the remaining unmarked group G_2 and make a table for this

G_2	a	b
1	G_2	G_2
2	G_2	G_2
3	–	G_2
4	G_1	G_2
5	G_2	G_2
7	G_1	G_2

G_2 is evidently *not* consistent, so we split it into maximal consistent subgroups and erase all marks (including the one on G_1):

$$G_1 = \{0, 6\}$$

$$G_3 = \{1, 2, 5\}$$

$$G_4 = \{3\}$$

$$G_5 = \{4, 7\}$$

We now pick G_3 for consideration:

G_3	a	b
1	G_5	G_3
2	G_4	G_3
5	G_5	G_3

This is not consistent either, so we split again and get

$$G_1 = \{0, 6\}$$

$$G_4 = \{3\}$$

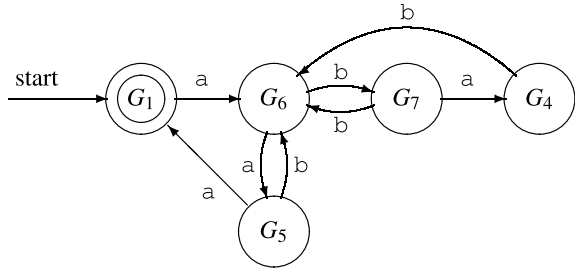
$$G_5 = \{4, 7\}$$

$$G_6 = \{1, 5\}$$

$$G_7 = \{2\}$$

We now pick G_5 and check this:

Fig. 1.11 Minimal DFA



G_5	a	b
4	G_1	G_6
7	G_1	G_6

This is consistent, so we mark it and pick another group, say, G_6 :

G_6	a	b
1	G_5	G_7
5	G_5	G_7

This, also, is consistent, so we have only one unmarked non-singleton group left: G_1 .

G_1	a	b
0	G_6	-
6	G_6	-

As we mark this, we see that there are no unmarked groups left other than singletons. Hence, the groups now form a minimal DFA equivalent to the one in Fig. 1.10. The minimised DFA is shown in Fig. 1.11.

1.7.2 Dead States

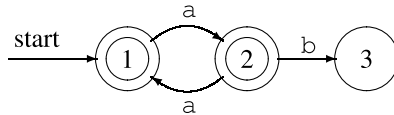
Algorithm 1.4 works under some, as yet, unstated assumptions:

- The *move* function is total, i.e., there are transitions on all symbols from all states, *or*
- There are no *dead states* in the DFA.

A dead state is a state from which no accepting state can be reached. Dead states do not occur in DFAs constructed from NFAs without dead states, and NFAs with dead states can not be constructed from regular expressions by the method shown in Sect. 1.3. Hence, as long as we use minimisation only on DFAs constructed by this

process, we are safe. However, if we get a DFA of unknown origin, we risk that it may contain both dead states and undefined transitions.

A transition to a dead state should rightly be equivalent to an undefined transition, as neither can yield future acceptance. The only difference is that we discover this earlier on an undefined transition than when we make a transition to a dead state. However, Algorithm 1.4 will treat these differently and may hence decree a group to be inconsistent even though it is not. This will make the algorithm split a group that does not need to be split, hence producing a non-minimal DFA. Consider, for example, the following DFA:



States 1 and 2 are, in fact, equivalent, as starting from either one, any sequence of *a*s (and no other sequences) will lead to an accepting state. A minimal equivalent DFA consists of a single accepting state with a transition to itself on *a*.

But Algorithm 1.4 will see a transition on *b* out of state 2 but no transition on *b* out of state 1, so it will not keep states 1 and 2 in the same group. As a result, no reduction in the DFA is made.

There are two solutions to this problem:

- (1) Make sure there are no dead states. This can be ensured by invariant, as is the case for DFAs constructed from regular expressions by the methods shown in this chapter, or by explicitly removing dead states before minimisation. Dead states can be found by a simple reachability analysis for directed graphs (if you can't reach an accepting state from state *s*, *s* is a dead state). In the above example, state 3 is dead and can be removed (including the transition to it). This makes states 1 and 2 stay in the same group during minimisation.
- (2) Make sure there are no undefined transitions. This can be achieved by adding a new dead state (which has transitions to itself on all symbols) and replacing all undefined transitions by transitions to this dead state. After minimisation, the group that contains the added dead state will contain all dead states from the original DFA. This group can now be removed from the minimal DFA (which will once more have undefined transitions). In the above example, a new (non-accepting) state 4 has to be added. State 1 has a transition to state 4 on *b*, state 3 has a transition to state 4 on both *a* and *b*, and state 4 has transitions to itself on both *a* and *b*. After minimisation, state 1 and 2 will be joined, as will state 3 and 4. Since state 4 is dead, all states joined with it are also dead, so we can remove the combined state 3 and 4 from the resulting minimised automaton.

Suggested exercises: 1.6, 1.11(c).

1.8 Lexers and Lexer Generators

We have, in the previous sections, seen how we can convert a language description written as a regular expression into an efficiently executable representation (a DFA). What we want is something more: A program that does lexical analysis, i.e., a *lexer*:

- A lexer has to distinguish between several different types of tokens, e.g., numbers, variables and keywords. Each of these are described by its own regular expression.
- A lexer does not check if its entire input is included in the languages defined by the regular expressions. Instead, it has to cut the input into pieces (tokens), each of which is included in one (or more) of the languages.
- If there are several ways to split the input into legal tokens, the lexer has to decide which of these it should use.

A program that takes a set of token definitions (each consisting of a regular expression and a token name) and generates a lexer is called a *lexer generator*.

The simplest approach would be to generate a DFA for each token definition and apply the DFAs one at a time to the input. This can, however, be quite slow, so we will instead from the set of token definitions generate a single DFA that tests for all the tokens simultaneously. This is not difficult to do: If the tokens are defined by regular expressions r_1, r_2, \dots, r_n , then the regular expression $r_1 | r_2 | \dots | r_n$ describes the union of the languages r_1, r_2, \dots, r_n and the DFA constructed from this combined regular expression will scan for all token types at the same time.

However, we also wish to distinguish between different token types, so we must be able to know *which* of the many tokens was recognised by the combined DFA. We can accomplish this with the following construction of a combined DFA:

- (1) Construct NFAs N_1, N_2, \dots, N_n for each r_1, r_2, \dots, r_n .
- (2) Mark the accepting states of the NFAs by the name of the tokens they accept.
- (3) Combine the NFAs to a single NFA by adding a new starting state which has epsilon-transitions to each of the starting states of the NFAs.
- (4) Convert the combined NFA to a DFA.
- (5) Each accepting state of the DFA consists of a set of NFA states, at least one of which is an accepting state which we marked by token type in step 2. These marks are used to mark the accepting states of the DFA, so each of these will indicate all the token types it accepts.

If the same accepting state in the DFA can accept several different token types, it is because these overlap. This is not unusual, as keywords usually overlap with variable names and a description of floating point constants may include integer constants as well. In such cases, we can do one of two things:

- Let the lexer generator generate an error and require the user to make sure the tokens are disjoint.
- Let the user of the lexer generator choose which of the tokens is preferred.

It can be quite difficult (though always possible) with regular expressions to define, e.g., the set of names that are not keywords. Hence, it is common to let the lexer choose according to a prioritised list. Normally, the order in which tokens are defined in the input to the lexer generator indicates priority (earlier defined tokens take precedence over later defined tokens). Hence, keywords are usually defined before variable names, which means that, for example, the string “`if`” is recognised as a keyword and not a variable name. When an accepting state in a DFA contains accepting NFA states with different marks, the mark corresponding to the highest priority (earliest defined) token is used. Hence, we can simply erase all but one mark from each accepting state. This is a very simple and effective solution to the problem.

When we described minimisation of DFAs, we used two initial groups: One for the accepting states and one for the non-accepting states. As there are now several kinds of accepting states (one for each token), we must use one group for each token, so we will have a total of $n + 1$ initial groups when we have n different tokens.

To illustrate the precedence rule, Fig. 1.12 shows an NFA made by combining NFAs for variable names, the keyword `if`, integers and floats, as described by the regular expressions in Sect. 1.1.2. The individual NFAs are (simplified versions of) what you get from the method described in Sect. 1.4. When a transition is labelled by a set of characters, it is a shorthand for a set of transitions each labelled by a single character. The accepting states are labelled with token names as described above. The corresponding minimised DFA is shown in Fig. 1.13. Note that state G is a combination of states 9 and 12 from the NFA, so it can accept both NUM and FLOAT, but since integers take priority over floats, we have marked G with NUM only. Similarly, state C is a combination of states 4 and 6 in the NFA, so it can accept both IF and ID, but since keywords take precedence over identifiers, we choose to let it accept only IF.

Splitting the Input Stream

As mentioned, the lexer must cut the input into tokens. This may be done in several ways. For example, the string `if17` can be split in many different ways:

- As one token, which is the variable name `if17`.
- As the variable name `if1` followed by the number `7`.
- As the keyword `if` followed by the number `17`.
- As the keyword `if` followed by the numbers `1` and `7`.
- As the variable name `i` followed by the variable name `f17`.
- And several more.

A common convention is that it is the *longest prefix* of the input that matches any token which will be chosen. Hence, the first of the above possible splittings of `if17` will be chosen. Note that the principle of the longest match takes precedence over the order of definition of tokens, so even though the string starts with the keyword `if`, which has higher priority than variable names, the variable name is chosen because it is longer.

Modern languages like C, Java or Haskell follow this convention, and so do most lexer generators, but some (mostly older) languages like FORTRAN do not. There

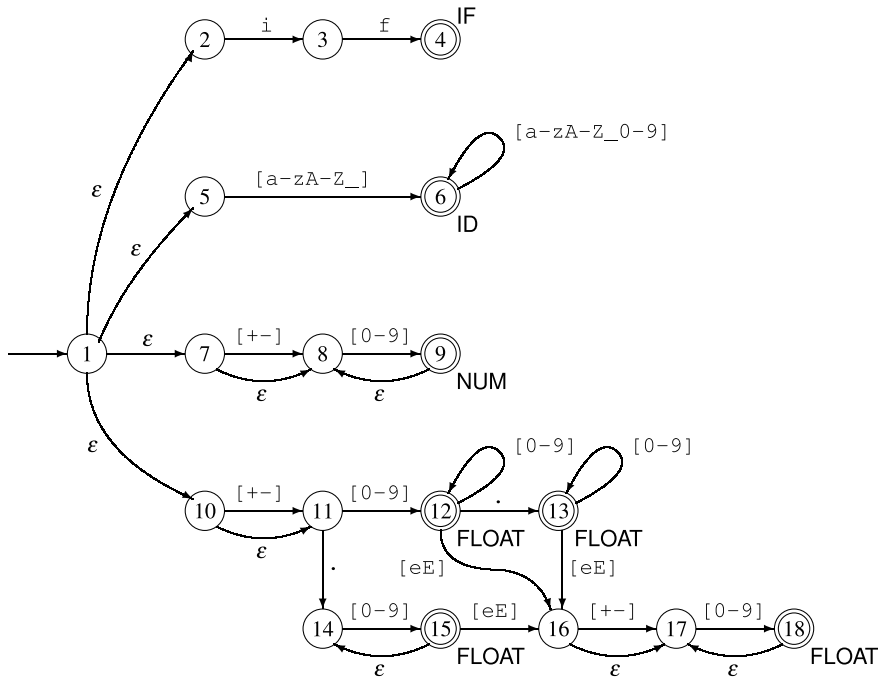


Fig. 1.12 Combined NFA for several tokens

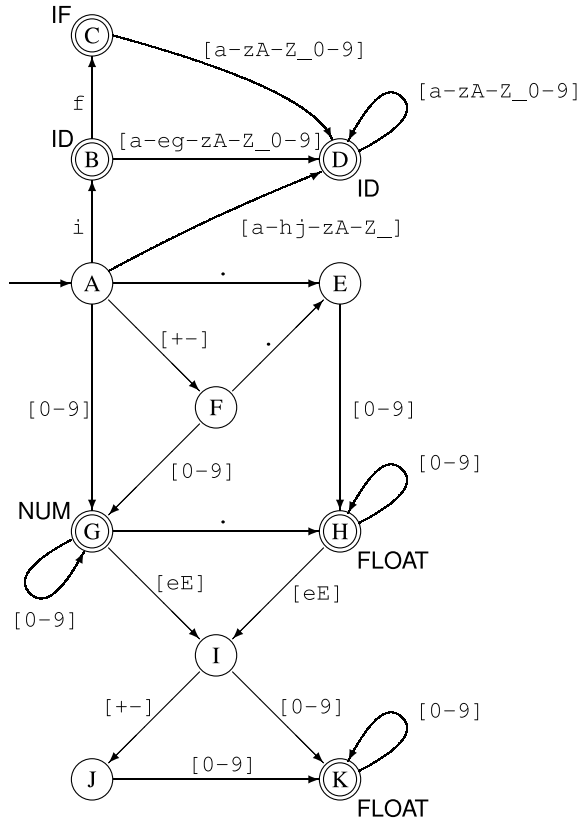
are also fairly new languages that have exceptions to the longest-prefix rule: In F#, the expression `f-1` is split into three tokens: `f`, `-`, and `1` even though `-1` is a valid token for a signed number. For example, `f -1` is split into just two tokens (as the space character is not considered a token), and is read as “the function `f` applied to the number `-1`”, whereas `f-1` is read as “`f` minus one”.

When other conventions are used, lexers must either be written by hand to handle these conventions, or the conventions used by the lexer generator must be overridden. Some lexer generators allow the user to have some control over the conventions used.

The principle of the longest matching prefix is handled by letting the DFA read as far as it can, until it either reaches the end of the input, or no transition is defined on the next input symbol. If the current state at this point is accepting, we are in luck, and can simply output the corresponding token. If not, we must go back to the last time we were in an accepting state and output the token indicated by this. The characters read since then are put back in the input stream. The lexer must, hence, retain the symbols it has read since the last accepting state, so it in such situations can re-insert these in the input. If we are not at the end of the input stream, we restart the DFA (in its initial state) on the remaining input to find the next tokens.

As an example, consider lexing of the string `3e-y` with the DFA in Fig. 1.13. We get to the accepting state G after reading the digit 3. However, we can continue making legal transitions to state I on `e` and then to state J on `-` (as these could be the

Fig. 1.13 Combined DFA for several tokens



start of the exponent part of a real number). It is only when we, in state J, find that there is no transition on y that we realise that this is not the case. We must now go back to the last accepting state (G) and output the number 3 as the first token and re-insert $-$ and e in the input stream, so we can continue with $e-y$ when we look for the subsequent tokens, which will be the identifier e followed by a lexical error, since no prefix of $-y$ match any tokens.

Lexical Errors

If no prefix of the input string forms a valid token, a *lexical error* has occurred. When this happens, the lexer will usually report an error. At this point, it may stop reading the input or it may attempt continued lexical analysis by skipping characters until a valid prefix is found. The purpose of the latter approach is to try finding further lexical errors in the same input, so several of these can be corrected by the user before re-running the lexer. Some of these subsequent errors may, however, not be real errors, but may be caused by the lexer not skipping enough characters (or skipping too many) after the first error is found. If, for example, the start of a

comment is ill-formed, the lexer may try to interpret the contents of the comment as individual tokens, and if the end of a comment is ill-formed, the lexer will read until the end of the next comment (if any) before continuing, hence skipping too much text.

When the lexer finds an error, the consumer of the tokens that the lexer produces (e.g., the rest of the compiler) can not usually itself produce a valid result. However, the compiler may try to find other errors in the remaining input, again allowing the user to find several errors in one edit-compile cycle. Again, some of the subsequent errors may really be spurious errors caused by lexical error(s), so the user will have to guess at the validity of every error message except the first, as only the first error message is guaranteed to be a real error. Nevertheless, such *error recovery* has, when the input is so large that restarting the lexer from the start of input incurs a considerable time overhead, proven to be an aid in productivity by locating more errors in less time. In an integrated development environment, the lexer may work interactively with a text editor, point to a lexical error in the text, allow the user to edit the file, and restart from the first modified position in the file when the user recompiles the program.

1.8.1 *Lexer Generators*

A lexer generator will typically use a notation for regular expressions similar to the one described in Fig. 1.1, but may require alphabet-characters to be quoted to distinguish them from the symbols used to build regular expressions. For example, an `*` intended to match a multiplication symbol in the input is distinguished from an `*` used to denote repetition by quoting the `*` symbol, e.g. as `'*'`, `"*"`, or `'*'`. Additionally, some lexer generators extend regular expressions in various ways, e.g., allowing a set of characters to be specified by listing the characters that are *not* in the set. This is useful, for example, to specify that a comment continues until the next newline character.

The input to the lexer generator will normally contain a list of regular expressions that each denote a token. Each of these regular expressions has an associated *action*. The action describes what is passed on to the consumer (e.g., the parser), typically an element from a token data type, which describes the type of token (NUM, ID, etc.) and sometimes additional information such as the value of a number token, the name of an identifier token, and the position of the token in the input file. The information needed to construct such values is typically provided by the lexer generator through library functions or variables that can be used in the actions.

Normally, the lexer generator requires white-space and comments to be defined by regular expressions. The actions for these regular expressions are typically empty, meaning that white-space and comments are just ignored. Note, however, that in languages (such as Python, Haskell, and F#) where indentation is used to indicate grouping of statements or expressions, whitespace will sometimes have to generate tokens to indicate start and end of such groupings.

An action can be more than just returning a token. If, for example, escape sequences (for defining, e.g., control characters) are allowed in string constants, the actions for string tokens will, typically, translate the string containing these sequences into a string where they have been substituted by the characters they represent. If a language has a large number of keywords, then a DFA that recognises all of these as individual tokens can be fairly large. In such cases, the keywords are not described as separate regular expressions in the lexer definition, but instead treated as special cases of the identifier token. The action for identifiers will then look the name up in a table of keywords and return the appropriate token type (or an identifier token if the name is not found in the table of keywords). A similar strategy can be used if the language allows identifiers to be equal to keywords, so they are distinguished by context.

Another use of non-trivial lexer actions is for nested comments. In principle, a regular expression (or finite automaton) cannot recognise arbitrarily deeply nested comments (see Sect. 1.9), but by using a global counter, the actions for comment tokens can keep track of the nesting level.

Sometimes lexer generators allow several different starting points. In the example in Figs. 1.12 and 1.13, all regular expressions share the same starting state. However, a single lexer may be used, e.g., for both tokens in the programming language and for tokens in the input data to that language. Often, there will be a good deal of sharing between these token sets (the tokens allowed in the input may, for example, be a subset of the tokens allowed in programs). Hence, it is useful to allow these to share a NFA, as this will save space. The resulting DFA will have several starting states. An accepting state may now have more than one token name attached, as long as these come from different token sets (corresponding to different starting points).

In addition to using this feature for several sources of text (program and input), it can be used locally within a single text to read very complex tokens. For example, nested comments and complex-format strings (with nontrivial escape sequences) can be easier to handle if this feature is used.

1.9 Properties of Regular Languages

We have talked about *regular languages* as the class of languages that can be described by regular expressions or finite automata, but this in itself may not give a clear understanding of what is possible and what is not possible to describe by a regular language. We will now state a few properties of regular languages, show some non-obvious examples of regular and non-regular languages, and give informal rules of thumb that can (sometimes) be used to decide if a language is regular.

1.9.1 *Relative Expressive Power*

First, we repeat that regular expressions, NFAs and DFAs have exactly the same expressive power: They all can describe all regular languages and only these. Some languages may, however, have much shorter descriptions in one of these forms than in others.

We have already argued that we from a regular expression can construct an NFA whose size is linear in the size of the regular expression, and that converting an NFA to a DFA can potentially give an exponential increase in size (see below for a concrete example of this). Since DFAs are also NFAs, NFAs are clearly at least as compact as (and sometimes much more compact than) DFAs. Similarly, we can see that NFAs are at least as compact (up to a small constant factor) as regular expressions. But we have not yet considered if the converse is true: Can an NFA be converted to a regular expression of proportional size. The answer is, unfortunately, no: There exist classes of NFAs (and even DFAs) that need regular expressions that are exponentially larger to describe them. This is, however, mainly of academic interest as we rarely have to make conversions in this direction.

If we are only interested in *if* a language is regular rather than the size or efficiency of its description, however, it does not matter which of the formalisms we choose, so we can in each case choose the formalism that suits us best. Sometimes it is easier to describe a regular language using a DFA or NFA instead of a regular expression. For example, the set of binary number strings that represent numbers that divide evenly by 5 can be described by a 6-state DFA (see Exercise 1.10), but it requires a very complex regular expression to do so. For programming language tokens, regular expressions are typically quite suitable.

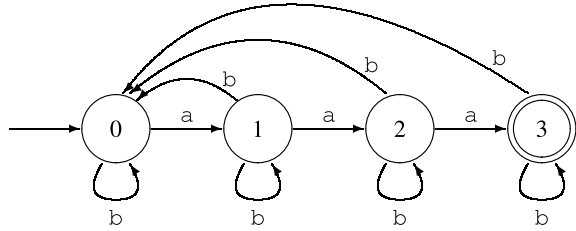
The subset construction (Algorithm 1.3) maps sets of NFA states to DFA states. Since there are $2^n - 1$ non-empty sets of n NFA states, the resulting DFA can potentially have exponentially more states than the NFA. But can this potential ever be realised? To answer this, it is not enough to find one n -state NFA that yields a DFA with $2^n - 1$ states (any one-state NFA does that). We need to find a family of ever bigger NFAs, all of which yield exponentially-sized DFAs. We also need to argue that the resulting DFAs are minimal. One construction that has these properties is the following: For each integer $n > 1$, construct an n -state NFA in the following way:

1. State 0 is the starting state and state $n-1$ is accepting.
2. If $0 \leq i < n-1$, state i has a transition to state $i + 1$ on the symbol a .
3. All states have transitions to themselves *and* to state 0 on the symbol b .

Figure 1.14 shows such an NFA for $n = 4$.

We can represent a set of these states by an n -bit number: Bit i in the number is 1 if and only if state i is in the set. The set that contains only the initial NFA state is, hence, represented by the binary number 1 zero-extended to n bits. We shall see that the way a transition maps a set of states to a new set of states can be expressed as an operation on the number:

Fig. 1.14 A 4-state NFA that gives 15 DFA states



- A transition on *a* maps the number x to $(2x \bmod (2^n))$.
- A transition on *b* maps the number x to itself and $(x \text{ OR } 1)$, using bitwise OR.

This is not hard to verify, so we leave this to the interested reader. It is also easy to see that, starting from the number 1, these two operations can generate any n -bit number. Hence, any subset can be reached by a sequence of transitions, which means that the subset-construction will generate a DFA state for every possible non-empty subset of the NFA states.

But is the DFA minimal? If we look at the NFA, we can see that, if $i < n-1$, an *a* leads from state i to $i+1$, so for each NFA state i there is exactly one sequence of *a*s that leads to the accepting state, and that sequence has $n-1-i$ *a*s. Hence, a DFA state whose subset contains the NFA state i will lead to acceptance on a string of $n-1-i$ *a*s, while a DFA state whose subset does not contain i will not. Hence, for any two different DFA states, we can find an NFA state i that is in one of the sets but not the other, and use that to construct a string that will distinguish the DFA states. Hence, all the DFA states are distinct, so the DFA is minimal.

1.9.2 Limits to Expressive Power

The most basic property of a DFA is that it is *finite*: It has a finite number of states and nowhere else to store information. This means, for example, that any language that requires unbounded counting cannot be regular. An example of this is the language $\{a^n b^n \mid n \geq 0\}$, that is, any sequence of *a*s followed by a sequence of the *same number* of *b*s. If we must decide membership in this language by a DFA that reads the input from left to right, we must, at the time we have read all the *a*s, know how many there were, so we can compare this number to the number of *b*s. But since a finite automaton cannot count arbitrarily high, the language is not regular. A similar non-regular language is the language of matching parentheses. However, if we limit the nesting depth of parentheses to a constant n , we can recognise this language by a DFA that has $n+1$ states (0 to n), where state i corresponds to i unmatched opening parentheses. State 0 is both the starting state and the only accepting state.

Some surprisingly complex languages are regular. As all finite sets of strings are regular languages, the set of all legal Java programs of less than a billion characters is a regular language, though it is by no means a simple one. While it can be argued

that it would be an acceptable limitation for a language to allow only programs of less than a billion characters, it is not practical to describe such a programming language as a regular language: The description would be far too large. Even if we ignore such absurdities, we can sometimes be surprised by the expressive power of regular languages. As an example, given any integer constant n , the set of numbers (written in binary or decimal notation) that divide evenly by n is a regular language (see Exercise 1.10).

1.9.3 Closure Properties

We can also look at closure properties of regular languages. It is clear that regular languages are closed under set union: If we have regular expressions s and t for two languages, the regular expression $s|t$ describes the union of these languages. Similarly, regular languages are closed under concatenation and unbounded repetition, as these correspond to basic operators of regular expressions.

Less obviously, regular languages are also closed under set difference and set intersection. To see this, we first look at set complement: Given a fixed alphabet Σ , the complement of the language L is the set of all strings built from the alphabet Σ , *except* the strings found in L . We write the complement of L as \overline{L} . To get the complement of a regular language L , we first construct a DFA for the language L and make sure that all states have transitions on all characters from the alphabet (as described in Sect. 1.7.2). Now, we simply change every accepting state to non-accepting and *vice versa*, and thus get a DFA for \overline{L} .

We can now (by using the set-theoretic equivalent of De Morgan's law) construct $L_1 \cap L_2$ as $\overline{\overline{L_1} \cup \overline{L_2}}$. Given intersection, we can now get set difference by $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$.

Regular sets are also closed under a number of common string operations, such as prefix, suffix, subsequence and reversal. The precise meaning of these words in the present context is defined below.

Prefix. A prefix of a string w is any initial part of w , including the empty string and all of w . The prefixes of abc are hence ε , a , ab and abc .

Suffix. A suffix of a string is what remains of the string after a prefix has been taken off. The suffixes of abc are hence abc , bc , c and ε .

Subsequence. A subsequence of a string is obtained by deleting any number of symbols from anywhere in the string. The subsequences of abc are hence abc , bc , ac , ab , c , b , a and ε .

Reversal. The reversal of a string is the string read backwards. The reversal of abc is hence cba .

As with complement, these can be obtained by simple transformations of the DFAs for the language.

Suggested exercises: 1.12.

1.10 Further Reading

There are many variants of the method shown in Sect. 1.3. The version presented here has been devised for use in this book in an attempt to make the method easy to understand and manageable to do by hand. Other variants can be found in [2, 3].

It is possible to convert a regular expression to a DFA directly without going through an NFA. One such method [2, 8] actually at one stage during the calculation computes information equivalent to an NFA (without epsilon-transitions), but more direct methods based on algebraic properties of regular expressions also exist [4, 10]. These, unlike NFA-based methods, generalise fairly easily to handle regular expressions extended with explicit set-intersection and set-difference operators. A good deal of theoretic information about regular expressions and finite automata can be found in [5]. An efficient DFA minimisation algorithm can be found in [6].

Lexer generators can be found for most programming languages. For C, the most common are Lex [7] and Flex [11]. Some lexer generators, e.g., Quex [12], generate the states of the DFA as program code instead of using table-lookup. This makes the generated lexers fast, but can use much more space than a table-driven program. Quex is also able to handle Unicode characters.

Finite automata and notation reminiscent of regular expressions are also used to describe behaviour of concurrent systems [9]. In this setting, a state represents the current state of a process and a transition corresponds to an event to which the process reacts by changing state.

1.11 Exercises

Exercise 1.1 Given the regular expression $s = (a|b)(c|d|\epsilon)$,

- Using the derivation rules in Fig. 1.1, show that $L(s)$ contains the string ac .
- Find the complete set $L(s)$.

Exercise 1.2 In the following, a *number-string* is a non-empty sequence of decimal digits, i.e., something in the language defined by the regular expression $[0-9]^+$. The value of a number-string is the usual interpretation of a number-string as an integer number. Note that leading zeroes are allowed.

Make for each of the following languages a regular expression that describes that language.

- All number-strings that have the value 42.
- All number-strings that *do not* have the value 42.
- All number-strings that have a value that is strictly greater than 42.

Exercise 1.3 Given the regular expression $a^*(a|b)aa$:

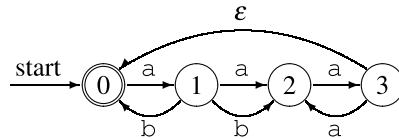
- Construct an equivalent NFA using the method in Sect. 1.3.

(b) Convert this NFA to a DFA using Algorithm 1.3.

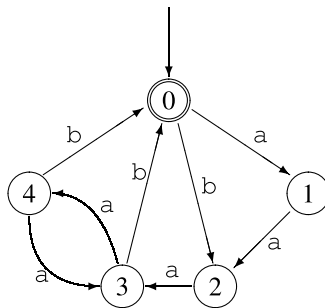
Exercise 1.4 Given the regular expression $((a|b)(a|bb))^*$:

- (a) Construct an equivalent NFA using the method in Sect. 1.3.
- (b) Convert this NFA to a DFA using Algorithm 1.3.

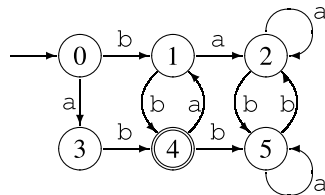
Exercise 1.5 Make a DFA equivalent to the following NFA:



Exercise 1.6 Minimise the following DFA:



Exercise 1.7 Minimise the following DFA:



Exercise 1.8 Construct DFAs for each of the following regular languages. In all cases the alphabet is $\{a, b\}$.

- (a) The set of strings that has exactly 3 bs (and any number of as).
- (b) The set of strings where the number of bs is a multiple of 3 (and there can be any number of as).
- (c) The set of strings where the difference between the number of as and the number of bs is a multiple of 3.

Exercise 1.9 Construct a DFA that recognises balanced sequences of parenthesis with a maximal nesting depth of 3, e.g., $\epsilon, ()(), ((())), ((()))()$ but not $((()))()$ or $()((()))$.

Exercise 1.10 Given that binary number strings are read with the most significant bit first and may have leading zeroes, construct DFAs for each of the following languages:

- (a) Binary number strings that represent numbers that are multiples of 4, e.g., 0, 100 and 10100.
- (b) Binary number strings that represent numbers that are multiples of 5, e.g., 0, 101, 10100 and 11001.

Hint: Make a state for each possible remainder after division by 5 and then add a state to avoid accepting the empty string.

- (c) Given a number n , what is the minimal number of states needed in a DFA that recognises binary numbers that are multiples of n ? Hint: write n as $a * 2^b$, where a is odd.

Exercise 1.11 The empty language, i.e., the language that contains no strings can be recognised by a DFA (any DFA with no accepting states will accept this language), but it can not be defined by any regular expression using the constructions in Sect. 1.1. Hence, the equivalence between DFAs and regular expressions is not complete. To remedy this, a new regular expression ϕ is introduced such that $L(\phi) = \emptyset$.

We will now look at some of the implications of this extension.

- (a) Argue why each of the following algebraic rules, where s is an arbitrary regular expression, is true:

$$\begin{aligned}\phi|s &= s \\ \phi s &= \phi \\ s\phi &= \phi \\ \phi^* &= \varepsilon\end{aligned}$$

- (b) Extend the construction of NFAs from regular expressions to include a case for ϕ .
- (c) What consequence will this extension have for converting the NFA to a minimal DFA? Hint: dead states.

Exercise 1.12 Show that regular languages are closed under prefix, suffix, subsequence and reversal, as postulated in Sect. 1.9. Hint: show how an NFA N for a regular language L can be transformed to an NFA N_p for the set of prefixes of strings from L , and similarly for the other operations.

Exercise 1.13 Which of the following statements are true? Argue each answer informally.

- (a) Any subset of a regular language is itself a regular language.
- (b) Any superset of a regular language is itself a regular language.
- (c) The set of anagrams of strings from a regular language forms a regular language. (An anagram of a string is obtained by rearranging the order of characters in the string, but without adding or deleting any. The anagrams of the string abc are hence abc , acb , bac , bca , cab and cba .)

Exercise 1.14 In Figs. 1.12 and 1.13 we used character sets on transitions as shortcuts for sets of transitions, each with one character. We can, instead, extend the definition of NFAs and DFAs such that such character sets are allowed on a single transition.

For a DFA (to be deterministic), we must require that transitions out of the same state have disjoint character sets.

- Sketch how Algorithm 1.3 must be modified to handle transitions with sets in such a way that the disjointedness requirement for DFAs are ensured.
- Sketch how Algorithm 1.4 must be modified to handle character sets. A new requirement for DFA minimality is that the number of transitions as well as the number of states is minimal. How can this be ensured?

Exercise 1.15 As mentioned in Sect. 1.4, DFAs are often implemented by tables where the current state is cross-indexed by the next symbol to find the next state. If the alphabet is large, such a table can take up quite a lot of room. If, for example, 16-bit Unicode is used as the alphabet, there are $2^{16} = 65536$ entries in each row of the table. Even if each entry in the table is only one byte, each row will take up 64 KB of memory, which may be a problem.

A possible solution is to split each 16-bit Unicode character c into two 8-bit characters c_1 and c_2 . In the regular expressions, each occurrence of a character c is hence replaced by the regular expression c_1c_2 . This regular expression is then converted to an NFA and then to a DFA in the usual way. The DFA may (and probably will) have more states than the DFA using 16-bit characters, but each state in the new DFA use only $1/256$ th of the space used by the original DFA.

- How much larger is the new NFA compared to the old?
- Estimate what the expected size (measured as number of states) of the new DFA is compared to the old. Hint: Some states in the NFA can be reached only after an even number of 8-bit characters are read and the rest only after an odd number of 8-bit characters are read. What does this imply for the sets constructed during the subset construction?
- Roughly, how much time does the new DFA require to analyse a string compared to the old?
- If space is a problem for a DFA over an 8-bit alphabet, do you expect that a similar trick (splitting each 8-bit character into two 4-bit characters) will help reduce the space requirements? Justify your answer.

Exercise 1.16 If L is a regular language, so is $L \setminus \{\varepsilon\}$, i.e., the set of all nonempty strings in L .

So we should be able to transform a regular expression for L into a regular expression for $L \setminus \{\varepsilon\}$. We want to do this with a function *nonempty* that is recursive over the structure of the regular expression for L , i.e., of the form:

$$\begin{aligned}
 \text{nonempty}(\varepsilon) &= \phi \\
 \text{nonempty}(a) &= \dots && \text{where } a \text{ is an alphabet symbol} \\
 \text{nonempty}(s|t) &= \text{nonempty}(s) \mid \text{nonempty}(t) \\
 \text{nonempty}(st) &= \dots \\
 \text{nonempty}(s?) &= \dots \\
 \text{nonempty}(s^*) &= \dots \\
 \text{nonempty}(s^+) &= \dots
 \end{aligned}$$

where ϕ is the regular expression for the empty language (see Exercise 1.11).

- (a) Complete the definition of *nonempty* by replacing the occurrences of “...” in the rules above by expressions similar to those shown in the rules for ε and $s|t$.
- (b) Use this definition to find $\text{nonempty}(a^*b^*)$.

Exercise 1.17 If L is a regular language, so is the set of all prefixes of strings in L (see Sect. 1.9.3).

So we should be able to transform a regular expression for L into a regular expression for the set of all prefixes of strings in L . We want to do this with a function *prefixes* that is recursive over the structure of the regular expression for L , i.e., of the form:

$$\begin{aligned}
 \text{prefixes}(\varepsilon) &= \varepsilon \\
 \text{prefixes}(a) &= a? && \text{where } a \text{ is an alphabet symbol} \\
 \text{prefixes}(s|t) &= \text{prefixes}(s) \mid \text{prefixes}(t) \\
 \text{prefixes}(st) &= \dots \\
 \text{prefixes}(s^*) &= \dots \\
 \text{prefixes}(s^+) &= \dots
 \end{aligned}$$

- (a) Complete the definition of *prefixes* by replacing the occurrences of “...” in the rules above by expressions similar to those shown in the rules for ε , a and $s|t$.
- (b) Use this definition to find $\text{prefixes}(ab^*c)$.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers; Principles, Techniques and Tools. Addison-Wesley (2007)
3. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press (1998)
4. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **1**(4), 481–494 (1964)
5. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation, 2nd edn. Addison-Wesley (2001)
6. Keller, J.P., Paige, R.: Program derivation with verified transformations – a case study. Commun. Pure Appl. Math. **48**(9–10) (1996)
7. Lesk, M.E.: Lex: a Lexical Analyzer Generator. Tech. Rep. 39, AT&T Bell Laboratories, Murray Hill, N. J. (1975)

8. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. *IEEE Trans. Electron. Comput.* **9**(1), 39–47 (1960)
9. Milner, R.: *Communication and Concurrency*. Prentice-Hall (1989)
10. Owens, S., Reppy, J., Turon, A.: Regular-expression derivatives re-examined. *J. Funct. Program.* **19**(2), 173–190 (2009). <https://doi.org/10.1017/S0956796808007090>
11. Paxson, V.: Flex, version 2.5, a fast scanner generator (1995). http://www.gnu.org/software/flex/manual/html_mono/flex.html
12. Schäfer, F.R.: Quex - fast universal lexical analyzer generator (2004–2011). <http://quex.sourceforge.net>. Accessed Sept 2014

Chapter 2

Syntax Analysis



Syntax and vocabulary are overwhelming constraints—the rules that run us. Language is using us to talk—we think we’re using the language, but language is doing the thinking, we’re its slavish agents.

Harry Mathews (1930–2017)

Where lexical analysis splits a text into tokens, the purpose of syntax analysis (also known as *parsing*) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This “something” is typically a data structure called the *syntax tree* of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree, and how the interior nodes of the tree are labelled.

In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting *syntax errors*.

As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy: A notation suitable for human understanding and algebraic manipulation is transformed into a machine-like low-level notation suitable for efficient execution. This process is called *parser generation*.

The notation we use for human manipulation is *context-free grammars*,¹ which is a recursive notation for describing sets of strings and imposing a structure on each such string. This notation can in some cases be translated almost directly into recursive programs, but it is often more convenient to generate *stack automata*. These are similar to the finite automata used for lexical analysis, but they can additionally use a stack, which allows counting and non-local matching of symbols. We shall see

¹ Because derivation is independent of context, as we will see.

two ways of generating such automata. The first of these, LL(1), is relatively simple, but works only for a somewhat restricted class of grammars. The SLR construction, which we present later, is more complex but handles a wider class of grammars. Sadly, neither of these work for *all* context-free grammars. Tools that handle all context-free grammars do exist, but they can incur a severe speed penalty, which is why most parser generators restrict the class of input grammars to what can be parsed efficiently using a specific method. This may require rewriting an otherwise correct grammar for a language into a form that can be handled by the method or by providing additional information to resolve ambiguities. We will show some techniques for doing so.

2.1 Context-Free Grammars

Like regular expressions, context-free grammars describe sets of strings, i.e., languages. Additionally, a context-free grammar also defines structure on the strings in the language it defines. A language is defined over some alphabet, for example the set of tokens produced by a lexer or the set of alphanumeric characters. The symbols in the alphabet are called *terminals*.

A context-free grammar recursively defines several sets of strings. Each set is denoted by a name, which is called a *nonterminal*. The set of nonterminals is disjoint from the set of terminals. One of the nonterminals are chosen to denote the main language described by the grammar. This nonterminal is called the *start symbol* of the grammar, and plays a role similar to the start state of a finite automaton. The sets are described by a number of *productions*. Each production describes some of the possible strings that are contained in the set denoted by a nonterminal. A production has the form

$$N \rightarrow X_1 \dots X_n$$

where N is a nonterminal and $X_1 \dots X_n$ are zero or more symbols, each of which is either a terminal or a nonterminal. The intended meaning of this notation is to say that the set denoted by N contains strings that are obtained by concatenating strings from the sets denoted by $X_1 \dots X_n$. In this setting, a terminal denotes a set consisting of a single string consisting of a single symbol, just like an alphabet character in a regular expression denotes a set consisting of a single string consisting of a single character. We will, when no confusion is likely, equate a nonterminal with the set of strings it denotes, like we did for alphabet characters in regular expressions.

Some examples:

$$A \rightarrow a$$

says that the set denoted by the nonterminal A contains the one-character string a .

$$A \rightarrow aA$$

says that the set denoted by A contains all strings formed by putting an a in front of a string taken from the set denoted by A . Together, these two productions indicate that A contains all non-empty sequences of a s and is hence (in the absence of other productions) equivalent to the regular expression a^+ .

We can define a grammar equivalent to the regular expression a^* by the two productions

$$\begin{aligned} B &\rightarrow \\ B &\rightarrow aB \end{aligned}$$

where the first production indicates that the empty string is part of the set B . Compare this grammar with the definition of s^* in Fig. 1.1.

Productions with empty right-hand sides are called *empty productions*. These are in some variants of grammar notation written with an ε on the right hand side instead of leaving it empty.

So far, we have not described any set that could not just as well have been described using regular expressions. Context-free grammars are, however, capable of expressing much more complex languages. In Sect. 1.9, we noted that the language $\{a^n b^n \mid n \geq 0\}$ is not regular. It is, however, easily described by the grammar

$$\begin{aligned} S &\rightarrow \\ S &\rightarrow aSb \end{aligned}$$

The second production ensures that the a s and b s are paired symmetrically around the middle of the string, so they occur in equal number.

The examples above have used only one nonterminal per grammar. When several nonterminals are used, we must make it clear which of these is the start symbol. By convention (if nothing else is stated), the nonterminal on the left-hand side of the first production is the start symbol. As an example, the grammar

$$\begin{aligned} T &\rightarrow R \\ T &\rightarrow aTa \\ R &\rightarrow b \\ R &\rightarrow bR \end{aligned}$$

has T as start symbol and denotes the set of strings that start with any number of a s followed by a non-zero number of b s and then the same number of a s with which it started.

In some variants of grammar notation, a shorthand notation is used where all the productions of the same nonterminal are combined to a single rule, using the alternative symbol ($|$) from regular expressions to separate the right-hand sides. In this notation, the above grammar would read

$$\begin{aligned} T &\rightarrow R \mid aTa \\ R &\rightarrow b \mid bR \end{aligned}$$

There are still four productions in the grammar, even though the arrow symbol \rightarrow is only used twice. Some grammar notations (such as EBNF, the Extended Backus-Naur Form) also allow equivalents of $?$, $*$ and $^+$ from regular expressions. With such shorthands, we can write the above grammar as

$$T \rightarrow b^+ \mid aTa$$

We will in this book, for simplicity, stick to basic grammar notation without shorthands. In the grammar notation we use, any name with initial capital letter and written in *Italics* (and possibly with subscripts or superscripts) denotes a nonterminal. Any symbol in *typewriter* font denotes itself as a terminal. Any name written in **boldface** denotes a lexical *token* that represents a *set of* concrete strings such as number constants or variable names. In examples of strings that are derived from a grammar, we may replace a boldface token by an element of the corresponding set of strings. For example, if the token **num** represents integer constants, we might write 5 or 7 instead.

2.1.1 How to Write Context-Free Grammars

As hinted above, a regular expression can systematically be rewritten to an equivalent context-free grammar by using a nonterminal for every subexpression in the regular expression, and using one or two productions for each nonterminal. The construction is shown in Fig. 2.1. So, if we can represent a language as a regular expression, it is easy to make a grammar for it.

We will also use context-free grammars to describe non-regular languages. An example of a non-regular language is the kind of arithmetic expressions that are part of most programming languages (and also found on electronic calculators), and which consist of numbers, operators, and parentheses. If arithmetic expressions *do not* have parentheses, the language can be described by a regular expression such as

$$\mathbf{num}((+|-|*|/)\mathbf{num})^*$$

where **num** represents any number constant.

However, if we *do* include parentheses (and these must match), the language can, as mentioned in Sect. 1.9, *not* be described by a regular expression, as a regular expression can not “count” the number of unmatched opening parentheses at a particular point in the string. Even without parentheses, the regular description above is not useful if you want operators to have different precedence, as it treats the expression as a flat string rather than as having structure. Arithmetic expressions with parentheses can be described by context-free grammars such as Grammar 2.2. This grammar, however, does not distinguish operators by precedence. We will look at structure and precedence rules in Sects. 2.2.1 and 2.3.

Form of s_i	Productions for N_i
ϵ	$N_i \rightarrow$
a	$N_i \rightarrow a$
$s_j s_k$	$N_i \rightarrow N_j N_k$
$s_j s_k$	$N_i \rightarrow N_j$ $N_i \rightarrow N_k$
s_j^*	$N_i \rightarrow N_j N_i$ $N_i \rightarrow$
s_j^+	$N_i \rightarrow N_j N_i$ $N_i \rightarrow N_j$
$s_j^?$	$N_i \rightarrow N_j$ $N_i \rightarrow$

Each subexpression of the regular expression is numbered, and subexpression s_i is described by a nonterminal N_i . The productions for N_i depend on the shape of s_i as shown in the table above.

Fig. 2.1 Converting regular expressions to context-free grammars

$$\begin{aligned}
 Exp &\rightarrow Exp + Exp \\
 Exp &\rightarrow Exp - Exp \\
 Exp &\rightarrow Exp * Exp \\
 Exp &\rightarrow Exp / Exp \\
 Exp &\rightarrow \mathbf{num} \\
 Exp &\rightarrow (Exp)
 \end{aligned}$$

Grammar 2.2 Simple expression grammar

Most constructions from programming languages are easily expressed by context-free grammars. In fact, the syntax most modern programming languages are designed to be mostly or completely describable using context-free grammars.

When writing a grammar for a programming language, one normally starts by dividing the constructs of the language into different *syntactic categories*. A syntactic category is a sub-language that embodies a particular language concept. Examples of common syntactic categories in programming languages are:

- Expressions are used to express calculation of values.
- Statements express actions that occur in a particular sequence.
- Declarations define properties of named entities such as variables or functions used in other parts of the program.
- Types are used in declarations to limit the kinds of values a variable can have or the parameters a function can take and the results it can return.

Each syntactic category is denoted by a nonterminal, e.g., *Exp* from Grammar 2.2. More than one nonterminal might be needed to describe a single syntactic category or to provide structure to elements of the syntactic category, as we shall see

Grammar 2.3 Simple statement grammar

$$\begin{aligned} Stat &\rightarrow \mathbf{id} := Exp \\ Stat &\rightarrow Stat ; Stat \\ Stat &\rightarrow \mathbf{if} Exp \mathbf{then} Stat \mathbf{else} Stat \\ Stat &\rightarrow \mathbf{if} Exp \mathbf{then} Stat \end{aligned}$$

later, but a selected nonterminal is the main nonterminal for the syntactic category. Productions for one syntactic category can refer to nonterminals for other syntactic categories. For example, statements may contain expressions, so some of the productions for statements use the main nonterminal for expressions. A simple grammar for statements might look like Grammar 2.3, which refers to the *Exp* nonterminal from Grammar 2.2. The terminal **id** represents variable names.

Suggested exercises: 2.3 (ignore, for now, the word “unambiguous”), 2.21(a).

2.2 Derivation

So far, we have just appealed to intuitive notions of recursion when we describe the set of strings that a grammar produces. To formally define the set of strings that a grammar describes, we use *derivation*, as we did for regular expressions, except that the derivation rules are different. An advantage of using derivations is, as we will later see, that syntax analysis is closely related to derivation.

The basic idea of derivation is to consider productions as rewrite rules: Whenever we have a nonterminal, we can replace this by the right-hand side of *any* single production where the nonterminal appears on the left-hand side. We can do this anywhere in a sequence of symbols (terminals and nonterminals) and repeat doing so until we have only terminals left. The resulting sequence of terminals is a string in the language defined by the grammar. Formally, we define the derivation relation \Rightarrow by the three rules

1. $\alpha N \beta \Rightarrow \alpha \gamma \beta$ if there is a production $N \rightarrow \gamma$
2. $\alpha \Rightarrow \alpha$
3. $\alpha \Rightarrow \gamma$ if there is a β such that $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$

where α , β and γ are (possibly empty) sequences of grammar symbols (terminals and nonterminals). The first rule states that using a production as a rewrite rule (anywhere in a sequence of grammar symbols) is a derivation step. The second states that the derivation relation is reflexive, i.e., that a sequence derives itself. The third rule describes transitivity, i.e., that a sequence of derivations is in itself a derivation.² Some texts about grammars use the symbol \rightarrow^* instead of \Rightarrow for derivation.

We can use derivation to formally define the language that a context-free grammar generates:

² The mathematically inclined will recognise that derivation is a preorder on sequences of grammar symbols.

Grammar 2.4 Example grammar

$$\begin{aligned} T &\rightarrow R \\ T &\rightarrow aTc \\ R &\rightarrow \\ R &\rightarrow RbR \end{aligned}$$

Fig. 2.5 Derivation of the sequence aabbbcc using Grammar 2.4

\underline{T}	
$\Rightarrow a\underline{T}c$	using the production $T \rightarrow aTc$
$\Rightarrow aa\underline{T}cc$	using the production $T \rightarrow aTc$
$\Rightarrow aa\underline{R}cc$	using the production $T \rightarrow R$
$\Rightarrow aaR\underline{b}Rcc$	using the production $R \rightarrow RbR$
$\Rightarrow aaR\underline{b}bcc$	using the production $R \rightarrow$
$\Rightarrow aaRb\underline{R}bcc$	using the production $R \rightarrow RbR$
$\Rightarrow aaRb\underline{R}bRbcc$	using the production $R \rightarrow RbR$
$\Rightarrow aaRb\underline{R}bbRbcc$	using the production $R \rightarrow$
$\Rightarrow aabb\underline{R}bcc$	using the production $R \rightarrow$
$\Rightarrow aabbbcc$	using the production $R \rightarrow$

Fig. 2.6 Leftmost derivation of the sequence aabbbcc using Grammar 2.4

\underline{T}	
$\Rightarrow a\underline{T}c$	using the production $T \rightarrow aTc$
$\Rightarrow aa\underline{T}cc$	using the production $T \rightarrow aTc$
$\Rightarrow aa\underline{R}cc$	using the production $T \rightarrow R$
$\Rightarrow aa\underline{R}bRcc$	using the production $R \rightarrow RbR$
$\Rightarrow aa\underline{R}bRbRcc$	using the production $R \rightarrow RbR$
$\Rightarrow aab\underline{R}bRcc$	using the production $R \rightarrow$
$\Rightarrow aab\underline{R}bRbRcc$	using the production $R \rightarrow RbR$
$\Rightarrow aabb\underline{R}bRcc$	using the production $R \rightarrow$
$\Rightarrow aabbbb\underline{R}cc$	using the production $R \rightarrow$
$\Rightarrow aabbbcc$	using the production $R \rightarrow$

Definition 2.1 Given a context-free grammar G with start symbol S , terminal symbols T and productions P , the language $L(G)$ that G generates is defined to be the set of sequences of terminal symbols that can be obtained by derivation from S using the productions P , i.e., the set $\{w \in T^* \mid S \Rightarrow w\}$.

As an example, we see that Grammar 2.4 generates the sequence aabbbcc by the derivation shown in Fig. 2.5. We have, for clarity, in each sequence of symbols underlined the nonterminal that will be rewritten in the following step.

In this derivation, we have applied derivation steps sometimes to the leftmost nonterminal, sometimes to the rightmost, and sometimes to a nonterminal that was neither. However, since derivation steps are local, the order does not matter. So, we might as well decide to always rewrite the leftmost nonterminal, as shown in Fig. 2.6.

A derivation that always rewrites the leftmost nonterminal is called a *leftmost derivation*. Similarly, a derivation that always rewrites the rightmost nonterminal is called a *rightmost derivation*.

2.2.1 Syntax Trees and Ambiguity

We can draw a derivation as a tree: The root of the tree is the start symbol of the grammar, and whenever we rewrite a nonterminal, we add as its children the symbols on the right-hand side of the production that was used. The leaves of the tree are terminals which, when read from left to right (i.e., following children from left to right), form the derived sequence. If a nonterminal is rewritten using an empty production, the child is shown as an empty leaf node, which is ignored when reading the sequence from the leaves of the tree. Some variants of syntax trees show empty leaf node as ϵ .

When we write such a *syntax tree*, the order of derivation is irrelevant: We get the same tree for left derivation, right derivation or any other derivation order. Only the choice of production for rewriting each nonterminal matters.

As an example, the derivations in Figs. 2.5 and 2.6 yield the same syntax tree, which is shown in Fig. 2.7.

The syntax tree adds structure to the string (sequence of terminal symbols) that it derives. It is this structure that we exploit in the later phases of the compiler.

For compilation, derivation is done backwards: We start with a string and want to produce a syntax tree. This process is called *syntax analysis* or *parsing*.

Even though the *order* of derivation does not matter when constructing a syntax tree, the *choice* of production for a nonterminal does. Obviously, different choices can lead to different strings being derived, but it may also happen that several different syntax trees can be built for the same string. As an example, Fig. 2.8 shows an alternative syntax tree for the same string that was derived in Fig. 2.7.

When a grammar permits several different syntax trees for some strings, we call the grammar *ambiguous*. If our only use of grammar is to describe sets of strings,

Fig. 2.7 Syntax tree for the string `aabbbbcc` using Grammar 2.4

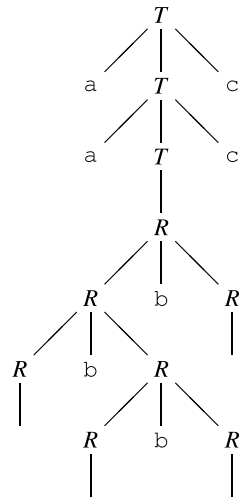
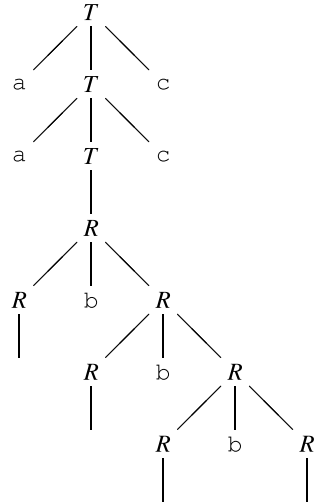


Fig. 2.8 Alternative syntax tree for the string aabbbcc using Grammar 2.4



Grammar 2.9
Unambiguous version
of Grammar 2.4

- $T \rightarrow R$
- $T \rightarrow aTc$
- $R \rightarrow$
- $R \rightarrow bR$

ambiguity is not a problem. However, when we want to use the grammar to impose structure on strings, the structure had better be the same every time. Hence, it is a desirable feature for a grammar to be unambiguous. In most (but not all) cases, an ambiguous grammar can be rewritten to an unambiguous grammar that generates the same set of strings. An unambiguous version of Grammar 2.4 is shown in Grammar 2.9.

An alternative to rewriting an ambiguous grammar to an unambiguous grammar is to apply external rules (not expressed in the grammar) for choosing productions when several are possible. We will return to this in Sect. 2.15.

How do we know when a grammar is ambiguous? Proving ambiguity is conceptually simple: If we can find a string and show two alternative syntax trees for it, the grammar is ambiguous. It may, however, be hard to find such a string and, when the grammar is unambiguous, even harder to show that there are no strings with more than one syntax tree. In fact, the problem is formally undecidable, i.e., there is no method that for all grammars can answer the question “Is this grammar ambiguous?”. But in many cases, it is not difficult to detect and prove ambiguity. For example, if a grammar has productions of the form

$$\begin{aligned}
 N &\rightarrow N\alpha N \\
 N &\rightarrow \beta
 \end{aligned}$$

where α and β are arbitrary (possibly empty) sequences of grammar symbols, the grammar is ambiguous. This is, for example, the case with Grammars 2.2 and 2.4.

We will, in Sects. 2.11 and 2.13, see methods for constructing parsers from grammars. These methods have the property that they only work on unambiguous grammars, so successful construction of a parser is a proof of unambiguity. However, the methods may for some unambiguous grammars fail to produce parsers, so failure to produce a parser is not a proof of ambiguity.

In the next section, we will see ways of rewriting a grammar to get rid of some sources of ambiguity. These transformations preserve the language that the grammar generates at the cost of changing the syntax trees derived from the grammar. By using such transformations (and others, which we will see later), we can create a large set of *equivalent* grammars, i.e., grammars that generate the same language.

Given two grammars, it would be nice to be able to tell if they are equivalent. Unfortunately, like ambiguity, equivalence of context-free grammars is undecidable. Sometimes, equivalence can be proven e.g., by induction over the set of strings that the grammars produce. The converse (i.e., non-equivalence) can be proven by finding an example of a string that one grammar can generate, but the other not. But in some cases, we just have to take claims of equivalence on faith or give up on deciding the issue.

Different, but equivalent, grammars will impose different syntax trees on the strings of their common language, so for compilers they are not equally useful—we want a grammar that imposes the intended structure on programs. Different structure is not exactly the same as different syntax trees: There may be several different grammars that impose the (for some intended purpose) correct structure, even if they do not yield the same syntax trees. We define when two different syntax trees represent the same structure by *reducing* syntax trees: If a node in the syntax tree has only one child, we replace the node by its child (which may be empty). A syntax tree that has no nodes with only one child is *fully reduced*. We deem two syntax trees to represent the same structure if their fully reduced trees are identical except for the names of nonterminals that represent the same syntactic category. Note that a reduced tree is not always a proper syntax tree: The edges do not represent single derivation steps, but rather sequences of derivation steps. Figure 2.10 shows a fully reduced version of the syntax tree in Fig. 2.7.

Suggested exercises: 2.1, 2.2, 2.21(b).

2.3 Operator Precedence

As mentioned in Sect. 2.1.1, we can describe traditional arithmetic expressions by Grammar 2.2. Note that **num** is a terminal that denotes all integer constants and that, here, the parentheses are terminal symbols (unlike in regular expressions, where they are used to impose structure on the regular expressions).

Fig. 2.10 Fully reduced tree for the syntax tree in Fig. 2.7

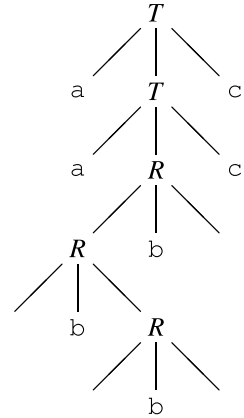
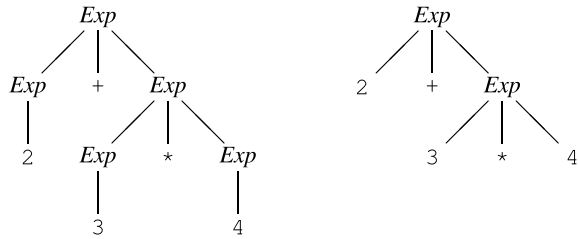


Fig. 2.11 Preferred syntax tree for $2+3*4$ using Grammar 2.2, and the corresponding fully reduced tree



This grammar is ambiguous, as evidenced by, e.g., the productions

$$\begin{aligned} Exp &\rightarrow Exp + Exp \\ Exp &\rightarrow \mathbf{num} \end{aligned}$$

which have the form that in Sect. 2.2.1 was claimed to imply ambiguity. That expressions are ambiguous should not be surprising, as we are used to the fact that an expression like $2+3*4$ can be read in two ways: Either as multiplying the sum of 2 and 3 by 4 or as adding 2 to the product of 3 and 4. Simple electronic calculators will choose the first of these interpretations (as they always calculate from left to right), whereas scientific calculators and most programming languages will choose the second, as they use a hierarchy of *operator precedences* which dictate that the product must be calculated before the sum. The hierarchy can be overridden by explicit parenthesisation, e.g., $(2+3)*4$.

Most programming languages use the same convention as scientific calculators, so we want to make this explicit in the grammar. Ideally, we would like the expression $2+3*4$ to generate the syntax tree shown on the left in Fig. 2.11, which reflects the operator precedences by grouping of subexpressions: When evaluating an expression, the subexpressions represented by subtrees of the syntax tree are evaluated before the topmost operator is applied. A corresponding fully reduced tree is shown on the right in Fig. 2.11.

A possible way of resolving the ambiguity is during syntax analysis to use precedence rules (not stated in the grammar itself) to select among the possible syntax trees. Many parser generators allow this approach, as we shall see in Sect. 2.15. However, some parsing methods require the grammars to be unambiguous, so we have to express the operator hierarchy in the grammar itself.

We first define some concepts relating to infix operators:

- An operator \oplus is *left-associative* if the expression $a \oplus b \oplus c$ must be evaluated from left to right, i.e., as $(a \oplus b) \oplus c$.
- An operator \oplus is *right-associative* if the expression $a \oplus b \oplus c$ must be evaluated from right to left, i.e., as $a \oplus (b \oplus c)$.
- An operator \oplus is *non-associative* if expressions of the form $a \oplus b \oplus c$ are illegal.

By the usual convention, $-$ and $/$ are left-associative, as e.g., $2-3-4$ is calculated as $(2-3)-4$. $+$ and $*$ are associative in the mathematical sense, meaning that it does not matter if we calculate from left to right or from right to left. In programming languages, it can matter, as one order of addition may cause overflow, where the other does not, or one order may cause more loss of precision than another. Generally, we want to avoid ambiguity of expressions, even when they are mathematically equivalent, so we choose either left-associativity or right-associativity even for operators that mathematically are fully associative. By convention (and similarity to $-$ and $/$) we choose to let addition and multiplication be left-associative. Also, having a left-associative $-$ and right-associative $+$ would not help resolving the ambiguity of $2-3+4$, as the operators so-to-speak “pull in different directions”.

List construction operators in functional languages, e.g., $::$ and $@$ in SML, are typically right-associative, as are function arrows in types: $a \rightarrow b \rightarrow c$ is read as $a \rightarrow (b \rightarrow c)$. The assignment operator in C is also right-associative: $a = b = c$ is read as $a = (b = c)$.

In some languages (such as Pascal), comparison operators (such as $<$ and $>$) are non-associative, i.e., you are not allowed to write $2 < 3 < 4$.

2.3.1 Rewriting Ambiguous Expression Grammars

If we have an ambiguous grammar

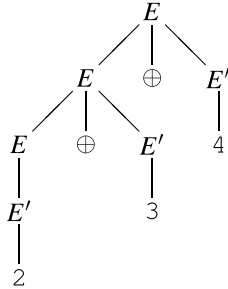
$$\begin{aligned} E &\rightarrow E \oplus E \\ E &\rightarrow \mathbf{num} \end{aligned}$$

and an intended structure on expressions, we can rewrite the ambiguous grammar to an unambiguous grammar that generates the correct structure. As a structure requires a specific associativity of \oplus , we use different rewrite rules for different associativities.

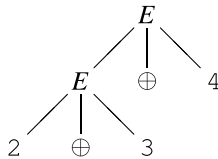
If \oplus is left-associative, we make the grammar *left-recursive* by having a recursive reference to the left only of the operator symbol and replacing the right-recursive reference by a reference to a new nonterminal that represents the non-recursive cases:

$$\begin{aligned}
 E &\rightarrow E \oplus E' \\
 E &\rightarrow E' \\
 E' &\rightarrow \mathbf{num}
 \end{aligned}$$

Now, the expression $2 \oplus 3 \oplus 4$ can only be parsed as



We get a slightly more complex syntax tree than ideally: We would have liked to avoid having two different nonterminals for expressions, and we would prefer to avoid the derivation $E \rightarrow E'$, but the tree certainly reflects the structure that the leftmost application of \oplus binds more tightly than the rightmost application of \oplus . The corresponding fully reduced tree makes this more clear:



We handle right-associativity in a similar fashion: We make the offending production *right-recursive*:

$$\begin{aligned}
 E &\rightarrow E' \oplus E \\
 E &\rightarrow E' \\
 E' &\rightarrow \mathbf{num}
 \end{aligned}$$

Non-associative operators are handled by *non-recursive* productions:

$$\begin{aligned}
 E &\rightarrow E' \oplus E' \\
 E &\rightarrow E' \\
 E' &\rightarrow \mathbf{num}
 \end{aligned}$$

Note that the latter transformation actually changes the language that the grammar generates, as it makes expressions of the form $\mathbf{num} \oplus \mathbf{num} \oplus \mathbf{num}$ illegal.

So far, we have handled only cases where an operator interacts with itself. This is easily extended to the case where several operators with the same precedence and associativity interact with each other, as for example $+$ and $-$:

$$\begin{aligned}
 E &\rightarrow E + E' \\
 E &\rightarrow E - E' \\
 E &\rightarrow E' \\
 E' &\rightarrow \mathbf{num}
 \end{aligned}$$

Operators with the same precedence must have the same associativity for this to work, as mixing left-recursive and right-recursive productions for the same nonterminal makes the grammar ambiguous. As an example, the grammar

$$\begin{aligned}
 E &\rightarrow E + E' \\
 E &\rightarrow E' \oplus E \\
 E &\rightarrow E' \\
 E' &\rightarrow \mathbf{num}
 \end{aligned}$$

seems like an obvious generalisation of the principles used above, giving $+$ and \oplus the same precedence and different associativity. But not only is the grammar ambiguous, it does not even accept the intended language. For example, the string $\mathbf{num} + \mathbf{num} \oplus \mathbf{num}$ is not derivable by this grammar.

In general, there is no obvious way to resolve ambiguity in an expression like $1 + 2 \oplus 3$, where $+$ is left-associative and \oplus is right-associative (or *vice-versa*). Hence, most programming languages (and most parser generators) *require* operators at the same precedence level to have identical associativity.

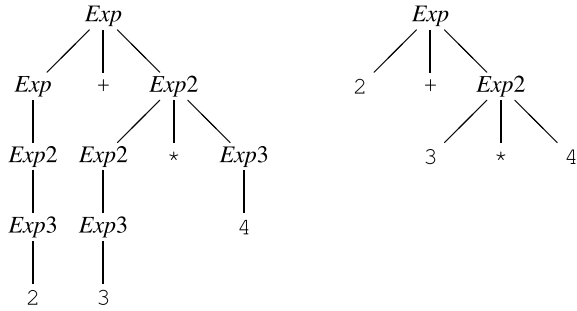
We also need to handle operators with different precedences. This is done by using a nonterminal for each precedence level. The idea is that if an expression uses an operator of a certain precedence level, then its subexpressions cannot use operators of lower precedence (unless these are inside parentheses). Hence, the productions for a nonterminal corresponding to a particular precedence level refers only to nonterminals that correspond to the same or higher precedence levels, unless parentheses or similar bracketing constructs disambiguate the use of these. Grammar 2.12 shows how these rules are used to make an unambiguous version of Grammar 2.2. Figure 2.13 shows the syntax tree for $2 + 3 * 4$ using this grammar and the corresponding reduced tree. Note that the nonterminals Exp , $Exp2$, and $Exp3$ all represent the same syntactic category (expressions), so in the reduced tree we can equate Exp and $Exp2$, making it equivalent to the reduced tree in Fig. 2.11.

Suggested exercises: 2.6.

Grammar 2.12
Unambiguous expression
grammar

$$\begin{aligned}
 Exp &\rightarrow Exp + Exp2 \\
 Exp &\rightarrow Exp - Exp2 \\
 Exp &\rightarrow Exp2 \\
 Exp2 &\rightarrow Exp2 * Exp3 \\
 Exp2 &\rightarrow Exp2 / Exp3 \\
 Exp2 &\rightarrow Exp3 \\
 Exp3 &\rightarrow \mathbf{num} \\
 Exp3 &\rightarrow (Exp)
 \end{aligned}$$

Fig. 2.13 Syntax tree for $2+3*4$ using Grammar 2.12, and the corresponding fully reduced tree



2.4 Other Sources of Ambiguity

Most of the potential ambiguity in grammars for programming languages comes from expression syntax and can be handled by exploiting precedence rules as shown in Sect. 2.3. Another classical example of ambiguity is the “dangling-else” problem.

Imperative languages like Pascal or C often let the else-part of a conditional be optional, like shown in Grammar 2.3. The problem is that it is not clear how to parse, for example,

```
if p then if q then x := 1 else x := 2
```

According to the grammar, the `else` can equally well match either `if`. The usual convention used in programming languages is that an `else` matches the closest not previously matched `if`, which, in the example, will make the `else` match the second `if`.

How do we make this clear in the grammar? We can treat `if`, `then` and `else` in the same way as right-associative operators, as this will make them group to the right, making an `if-then` match the closest `else`. However, the grammar transformations shown in Sect. 2.3 can not directly be applied to Grammar 2.3, as the productions for conditionals do not have the right form.

Instead we use the following observation: When an `if` and an `else` match, all `ifs` that occur between these must have matching `elses`. This can easily be proven by assuming otherwise and concluding that this leads to a contradiction.

Hence, we make two nonterminals: One for matched (i.e. with `else-part`) conditionals and one for unmatched (i.e. without `else-part`) conditionals. The result is shown in Grammar 2.14. This grammar also resolves the associativity of semicolon (right) and the precedence of `if` over semicolon.

An alternative to rewriting grammars to resolve ambiguity is to use an ambiguous grammar and resolve conflicts by using external precedence rules during parsing. We shall look into this in Sect. 2.15.

All cases of ambiguity must be treated carefully: It is not enough that we eliminate ambiguity, we must do so in a way that results in the desired structure: The structure

Grammar 2.14

Unambiguous grammar for statements

$$\begin{aligned} Stat &\rightarrow Stat2 ; Stat \\ Stat &\rightarrow Stat2 \\ Stat2 &\rightarrow Matched \\ Stat2 &\rightarrow Unmatched \\ Matched &\rightarrow \text{if } Exp \text{ then } Matched \text{ else } Matched \\ Matched &\rightarrow \mathbf{id} := Exp \\ Unmatched &\rightarrow \text{if } Exp \text{ then } Matched \text{ else } Unmatched \\ Unmatched &\rightarrow \text{if } Exp \text{ then } Stat2 \end{aligned}$$

of arithmetic expressions is significant, and it makes a difference to which `if` an `else` is matched.

Suggested exercises: 2.3 (focusing now on making the grammar unambiguous).

2.5 Syntax Analysis

The syntax analysis phase of a compiler will take a string of tokens produced by the lexer, and from this construct a syntax tree for the string by finding a derivation of the string from the start symbol of the grammar.

This can be done by guessing derivations (i.e., choosing productions randomly) until the right one is found, but random guessing is hardly an effective method. Even so, some parsing techniques are based on “guessing” derivations. However, these make sure, by looking at the string, that they will always pick the right production. These are called *predictive* parsing methods. Predictive parsers always build the syntax tree from the root down to the leaves and are hence also called (deterministic) top-down parsers.

Other parsers go the other way: They search for parts of the input string that matches right-hand sides of productions and rewrite these to the left-hand nonterminals, at the same time building pieces of the syntax tree. The syntax tree is eventually completed when the string has been rewritten (by inverse derivation) to the start symbol. Also here, we wish to make sure that we always pick the “right” rewrites, so we get deterministic parsing. Such methods are called *bottom-up* parsing methods.

We will in the next sections first look at predictive parsing and later at a bottom-up parsing method called SLR parsing.

2.6 Predictive Parsing

If we look at the left-derivation in Fig. 2.6, we see that, when we replace a nonterminal by the right-hand side of a production, all symbols to the left of this nonterminal are terminals. In other words, we always rewrite the leftmost nonterminal. The terminals to the left of this nonterminal correspond to a prefix of the string that is being parsed.

In a parsing situation, this prefix will be the part of the input that has already been read. The job of the parser is now to choose the production by which the leftmost unexpanded nonterminal should be rewritten. Our aim is to be able to make this choice deterministically based only on the next unmatched input symbol.

If we look at the third line in Fig. 2.6, we have already read two *a*s and (if the input string is the one shown in the bottom line) the next symbol is a *b*. Since the right-hand side of the production

$$T \rightarrow aTc$$

starts with an *a*, we obviously can not use this. Hence, we can only rewrite *T* using the production

$$T \rightarrow R$$

We are not quite as lucky in the next step. None of the productions for *R* start with a terminal symbol, so we can not immediately choose a production based on this. As the grammar (Grammar 2.4) is ambiguous, it should not be a surprise that we can not always choose uniquely. If we, instead, use the unambiguous grammar (Grammar 2.9) we can, when the next input symbol is a *b*, immediately choose the second production for *R*. When all the *b*s are read and we are at the following *c*, we choose the empty production for *R* and match the remaining input with the rest of the derived string.

If we can always choose a unique production based only on the next input symbol, we are able to do predictive parsing without backtracking. We will, below, investigate when we are able to make such unique choices.

2.7 Nullable and FIRST

In simple cases, like the above, the right-hand sides of all productions for any given nonterminal start with distinct terminals, except at most one production whose right-hand side does not start with a terminal (i.e., it is an empty production, or the right-hand side of the production starts with a nonterminal). We chose the production whose right-hand side does not start with a terminal whenever the input symbol does not match any of the terminal symbols that start the right-hand sides other productions. In the example above (using (Grammar 2.9)), we choose the second production for *T* as long as the next input symbol is *a*, and the first production otherwise. Similarly, we choose the second production for *R* when the next input symbol is *b*, and the first production otherwise.

We can extend the method to work also for grammars where more than one production for a given nonterminal have right-hand sides that do not start with terminals. We just need to be able to select between these productions based on the input symbol, even when the right-hand sides do not start with terminal symbols. To do this, we for each right-hand side find the set of strings that the right-hand side can derive.

We then, for each production, find the set of initial characters of the strings in these sets. These sets are called the *FIRST* sets of the productions.

If the *FIRST* sets of different productions for the same nonterminal are disjoint, we can, given an input symbol, choose the production whose *FIRST* set contains this symbol. If the input symbol is not in any of the *FIRST* sets, and there is an empty production, we choose this. Otherwise, we report a syntax error message.

Hence, we define the function *FIRST*, which given a sequence of grammar symbols (e.g., the right-hand side of a production) returns the set of symbols with which strings derived from that sequence can begin:

Definition 2.2 A symbol c is in $FIRST(\alpha)$ if and only if $\alpha \Rightarrow c\beta$ for some (possibly empty) sequence β of grammar symbols.

We extend this definitions to productions, so $FIRST(N \rightarrow \alpha) = FIRST(\alpha)$, and to nonterminals by defining $FIRST(N)$ to be the union of the first sets for all productions for N .

To calculate *FIRST*, we need an auxiliary function *Nullable*, which for a sequence α of grammar symbols indicates whether or not that sequence can derive the empty string:

Definition 2.3 A sequence α of grammar symbols is *Nullable* (we write this as $Nullable(\alpha)$) if and only if $\alpha \Rightarrow \varepsilon$, where ε indicates the empty string.

A production $N \rightarrow \alpha$ is called nullable if $Nullable(\alpha)$.

We describe calculation of *Nullable* by case analysis over the possible forms of sequences of grammar symbols:

Algorithm 2.4

$$\begin{aligned} Nullable() &= true \\ Nullable(a) &= false \\ Nullable(\alpha \beta) &= Nullable(\alpha) \wedge Nullable(\beta) \\ Nullable(N) &= Nullable(\alpha_1) \vee \dots \vee Nullable(\alpha_n), \\ &\quad \text{where the productions for } N \text{ are} \\ &\quad N \rightarrow \alpha_1, \dots, N \rightarrow \alpha_n \end{aligned}$$

where a is a terminal, N is a nonterminal, α and β are sequences of grammar symbols. Note that the first rule handles the empty sequence of grammar symbols.

The equations are quite natural: Any occurrence of a terminal on a right-hand side makes *Nullable* false for that right-hand side, and a nonterminal is nullable if any production has a nullable right-hand side.

Note that this is a recursive definition since *Nullable* for a nonterminal is defined in terms of *Nullable* for its right-hand sides, some of which may contain that same nonterminal. We can solve this in much the same way that we solved set equations in Sect. 1.5.1. We have, however, now booleans instead of sets, and we have several

equations instead of one. Still, the method is essentially the same: We have a set of Boolean equations:

$$\begin{aligned} X_1 &= F_1(X_1, \dots, X_n) \\ &\vdots \\ X_n &= F_n(X_1, \dots, X_n) \end{aligned}$$

We initially assume X_1, \dots, X_n to be all *false*. We then, in any order, calculate the right-hand sides of the equations and update the variable on the left-hand side by the calculated value. We continue until all right-hand sides evaluate to the values on the corresponding left-hand sides. This implies that the equations are solved.

In the Appendix and Sect. 1.5.1, we required the functions to be monotonic with respect to subset. Correspondingly, we now require the Boolean functions to be monotonic with respect to truth: If we make more arguments of a function true, the result will also be more true (i.e., it may stay unchanged or change from *false* to *true*, but never change from *true* to *false*). We also have a property similar to the distributivity property we observed in Sect. 1.5.1: $F_i(X_1, \dots, (p \vee q), \dots, X_n) = F_i(X_1, \dots, p, \dots, X_n) \vee F_i(X_1, \dots, q, \dots, X_n)$ for any i and j . We can also observe that if we represent *false* by the empty set and *true* by the set $\{1\}$, \vee is set union and \wedge is set intersection, and we can solve the equations as set equations.

If we look at Grammar 2.9, we get the following equations by applying the rules for *Nullable* on the right-hand of the productions, and reducing the results using simple logical identities:

$$\begin{aligned} \text{Nullable}(T) &= \text{Nullable}(R) \vee \text{Nullable}(aTc) \\ &= \text{Nullable}(R) \vee (\text{Nullable}(a) \wedge \text{Nullable}(T) \wedge \text{Nullable}(c)) \\ &= \text{Nullable}(R) \vee (\text{false} \wedge \text{Nullable}(T) \wedge \text{false}) \\ &= \text{Nullable}(R) \\ \text{Nullable}(R) &= \text{Nullable}() \vee \text{Nullable}(bR) \\ &= \text{true} \vee (\text{Nullable}(b) \wedge \text{Nullable}(R)) \\ &= \text{true} \end{aligned}$$

In a fixed-point calculation, we initially assume that *Nullable* is false for all nonterminals, and use this as a basis for calculating the right-hand sides of the equations. We repeat recalculating these until there is no change between two iterations. Figure 2.15 shows the fixed-point iteration for the above equations. In each iteration, we evaluate the right-hand sides of the equations using the values from the previous iteration. We continue until two iterations yield the same results. The right-most column shows the final result. Fixed-point iteration is a bit of overkill for this simple example, but it is needed in the general case.

Fig. 2.15 Fixed-point iteration for calculation of *Nullable*

Nonterminal	Initialisation	Iteration 1	Iteration 2	Iteration 3
T	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>
R	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>

To choose productions for predictive parsing, we need to know if the individual productions are nullable. When we know which nonterminals are nullable, it is easy enough to calculate this for the right-hand side of each production using the formulae in Algorithm 2.4. For Grammar 2.9 we get:

Production	Nullable
$T \rightarrow R$	<i>true</i>
$T \rightarrow aTc$	<i>false</i>
$R \rightarrow$	<i>true</i>
$R \rightarrow bR$	<i>false</i>

We can calculate *FIRST* in a fashion similar to the calculation of *Nullable*, i.e., by using formulas for sequences of grammar symbols and recursively defining equations for the nonterminals:

Algorithm 2.5

$$\begin{aligned}
 FIRST() &= \emptyset \\
 FIRST(a) &= \{a\} \\
 FIRST(\alpha\beta) &= \begin{cases} FIRST(\alpha) \cup FIRST(\beta) & \text{if } Nullable(\alpha) \\ FIRST(\alpha) & \text{if not } Nullable(\alpha) \end{cases} \\
 FIRST(N) &= FIRST(\alpha_1) \cup \dots \cup FIRST(\alpha_n) \\
 &\quad \text{where the productions for } N \text{ are} \\
 &\quad N \rightarrow \alpha_1, \dots, N \rightarrow \alpha_n
 \end{aligned}$$

where a is a terminal, N is a nonterminal, and α and β are sequences of grammar symbols. \emptyset denotes the empty set.

The only nontrivial equation is that for $\alpha\beta$. Obviously, anything that can start a string derivable from α can also start a string derivable from $\alpha\beta$. However, if α is nullable, a derivation may proceed as $\alpha\beta \Rightarrow \beta \Rightarrow \dots$, so if α is nullable, anything in $FIRST(\beta)$ is also in $FIRST(\alpha\beta)$. So we have special cases for when α is nullable, and when it is not.

The set-equations are solved in the same general way as the Boolean equations for *Nullable*, but since we work with sets, we initially assume every set to be empty. For Grammar 2.9, we get the following equations:

$$\begin{aligned}
 FIRST(T) &= FIRST(R) \cup FIRST(aTc) \\
 &= FIRST(R) \cup FIRST(a) \\
 &= FIRST(R) \cup \{a\} \\
 FIRST(R) &= FIRST() \cup FIRST(bR) \\
 &= \emptyset \cup FIRST(b) \\
 &= \{b\}
 \end{aligned}$$

The fixed-point iteration is shown in Fig. 2.16. As before, we use the values from the previous iteration when calculating the right-hand sides of the equations.

Fig. 2.16 Fixed-point iteration for calculation of *FIRST*

Nonterminal	Initialisation	Iteration 1	Iteration 2	Iteration 3
<i>T</i>	\emptyset	{a}	{a, b}	{a, b}
<i>R</i>	\emptyset	{b}	{b}	{b}

As for *Nullable*, we need *FIRST* for every production, which we can find by using the formulae in Algorithm 2.5 and the values of *FIRST* for the nonterminals:

Production	<i>FIRST</i>
$T \rightarrow R$	{b}
$T \rightarrow aTc$	{a}
$R \rightarrow$	\emptyset
$R \rightarrow bR$	{b}

We note that the two productions for *T* have disjoint *FIRST* sets, so we can uniquely choose a production based on the input symbol. Since the first production for *T* is nullable, we choose this also on symbols other than *b*, in fact we choose it on all other symbols than *a*, where we choose the second production. The productions for *R* also have disjoint *FIRST* sets. We choose the empty production for *R* when the input symbol is not *b*.

When working with grammars by hand, it is usually quite easy to see for most productions if they are nullable and what their *FIRST* sets are. For example, a production is not nullable if its right-hand side has a terminal anywhere, and if the right-hand side starts with a terminal, the *FIRST* set consists of only that symbol. Sometimes, however, it is necessary to use fixed-point iteration to solve the equations.

Suggested exercises: 2.8 (*Nullable* and *FIRST* only).

2.8 Predictive Parsing Revisited

We have up to now used the following rule for predictive parsing: If the right-hand sides of the productions for a nonterminal have disjoint *FIRST* sets, and the next input symbol is in one of these sets, we choose the corresponding production. If the next input symbol is not in any of these sets, and there is an empty production, we choose this.

We can generalise the case for the empty production, so we in the case where the next input symbol is not found in any *FIRST* set, can select a production if it is *Nullable*. The idea is that a *Nullable* production can derive the empty string, so we can extend the rule for empty productions to cover nullable productions as well. Note that a nullable production can have a non-empty *FIRST* set, so it can be chosen both when the next input symbol is in its *FIRST* set, and when the next input symbol is not in the *FIRST* set of the nonterminal (i.e., not in the *FIRST* set of any of the productions for the nonterminal).

But if there are several *Nullable* productions, we have no way of choosing between them. So, for predictive parsing, a nonterminal can have at most one *Nullable* production.

We said in Sect. 2.2.1 that our syntax analysis methods will detect ambiguous grammars. However, this is not true with the method as stated above: We can get unique choice of production even for some ambiguous grammars, including Grammar 2.4. In the best case, the syntax analysis will just choose one of several possible syntax trees for a given input string. In many cases, we do not consider such behaviour acceptable. In fact, we would very much like our parser construction method to tell us if we by mistake write an ambiguous grammar.

Even worse, the rules for predictive parsing as presented here might—even for unambiguous grammars—give deterministic choice of production, but reject strings that actually belong to the language described by the grammar. If we, for example, change the second production in Grammar 2.9 to

$$T \rightarrow aTb$$

this will not change the choices made by the predictive parser for nonterminal R . However, always choosing the last production for R on a b will lead to erroneous rejection of many strings, including ab .

This kind of behaviour is clearly unacceptable. We should, at least, get a warning that this might occur, so we can rewrite the grammar or choose another syntax analysis method.

Hence, we add to our construction of predictive parsers a test that will reject all ambiguous grammars and those unambiguous grammars that can cause the parser to fail even though a valid parse exists. The test can not tell us in which of these two categories a grammar belongs, though, but in either case we need to rewrite the grammar (or choose a different parsing method).

We have, so far, simply chosen a nullable production if and only if the next input symbol is not in the *FIRST* set of the nonterminal, i.e., if no other choice is valid. But this does not imply that choosing the nullable production is always valid when no other choice is valid. It could well be the case that no choice is valid—which implies that the string we are parsing is not in the language of the grammar. The right thing to do in such cases is to issue an error.

So we must change the rules for choosing productions in such a way that we choose a nullable production only if this is meaningful. So we choose a production $N \rightarrow \alpha$ on symbol c if at least one of the two conditions below are satisfied:

- 1) $c \in \text{FIRST}(\alpha)$, or
- 2) α is nullable, and the sequence Nc can occur somewhere in a derivation starting from the start symbol of the grammar.

The first rule is obvious, but the second requires a bit of explanation: If α is nullable, we can construct a syntax tree for N without reading any input, so it seems like a nullable production could be a valid choice regardless of the next input symbol.

Predictive parsing makes a leftmost derivation, so we always rewrite the leftmost nonterminal N in the current sequence of grammar symbols. If we look at the part of the current sequence of grammar symbols that start with this N , it has the form $N\beta$, where β is any (possibly empty) sequence of grammar symbols. If the next input symbol is c , it must be in $FIRST(N\beta)$, otherwise we can never derive $N\beta$ to a string that starts with c . If c is not in $FIRST(N)$, then N must be nullable and c must be in $FIRST(\beta)$. But β is not necessarily the right-hand side of any production, so we will need to find $FIRST(\beta)$ in some other way. The next section will cover this.

Even with this restriction on choosing nullable productions, we can still have situations where both nullable and non-nullable productions are valid choices. This includes the example above with the modified Grammar 2.9 (since Rb can occur in a derivation). An ambiguous grammar will have either:

1. two or more nullable productions for a given nonterminal, or
2. overlapping $FIRST$ sets for the productions of a nonterminal, or
3. a $FIRST$ set for a non-nullable production that overlaps with the set of characters that makes a nullable production for the same nonterminal a valid choice.

Note that while *absence* of such conflicts proves that a grammar is unambiguous, *presence* of such conflicts does not prove that a grammar is ambiguous.

2.9 Follow

To determine when we can select a nullable production during predictive parsing, we introduce $FOLLOW$ sets for nonterminals.

Definition 2.6 A terminal symbol c is in $FOLLOW(N)$ if and only if there is a derivation from the start symbol S of the grammar such that $S \Rightarrow \alpha N c \beta$, where α and β are (possibly empty) sequences of grammar symbols.

In other words, a terminal c is in $FOLLOW(N)$ if c may follow N at some point in a derivation. Unlike $FIRST(N)$, this is not a property of the productions for N , but of the productions that (directly or indirectly) use N on their right-hand side.

To correctly handle end-of-string conditions, we also want to detect when $S \Rightarrow \alpha N$, i.e., if there are derivations where N can be followed by the end of input. It turns out to be easiest to do this by adding an extra production to the grammar:

$$S' \rightarrow S\$$$

where S' is a new nonterminal that replaces S as start symbol, and $\$$ is a new terminal symbol that represents the end of input. Hence, in the new grammar, $\$$ will be in $FOLLOW(N)$ exactly if $S' \Rightarrow \alpha N \$$, which is the case exactly when $S \Rightarrow \alpha N$.

The easiest way to calculate $FOLLOW$ is to generate a collection of *set constraints*, which are subsequently solved to find the smallest sets that obey the constraints. A production

$$M \rightarrow \alpha N \beta$$

generates the constraint $FIRST(\beta) \subseteq FOLLOW(N)$, since β , obviously, can follow N . Furthermore, if $Nullable(\beta)$ the production also generates the constraint $FOLLOW(M) \subseteq FOLLOW(N)$ (note the direction of the inclusion). The reason is that, if there is a derivation $S' \Rightarrow \gamma M \delta$, then because $M \rightarrow \alpha N \beta$, and β is nullable, we derive $S' \Rightarrow \gamma M \delta \Rightarrow \gamma \alpha N \beta \delta \Rightarrow \gamma \alpha N \delta$, so $FIRST(\delta)$ is also in $FOLLOW(N)$. This is true for any such δ , so $FOLLOW(M) \subseteq FOLLOW(N)$.

If a right-hand side contains several occurrences of nonterminals, we add constraints for all occurrences, i.e., splitting the right-hand side with different choices of α , N and β . For example, the production $A \rightarrow BcB$ generates the constraint $\{c\} \subseteq FOLLOW(B)$ by splitting after the first B , and, by splitting after the last B , we also get the constraint $FOLLOW(A) \subseteq FOLLOW(B)$.

We solve the generated constraints in the following fashion:

We start by assuming empty $FOLLOW$ sets for all nonterminals. First, we then handle the constraints of the form $FIRST(\beta) \subseteq FOLLOW(N)$: We compute $FIRST(\beta)$ and add this to $FOLLOW(N)$. Next, we handle the second type of constraints: For each constraint $FOLLOW(M) \subseteq FOLLOW(N)$, we add all elements of $FOLLOW(M)$ to $FOLLOW(N)$. We iterate these last steps until no further changes happen.

In summary, the steps taken to calculate the $FOLLOW$ sets of a grammar are:

1. Extend the grammar by adding a new nonterminal $S' \rightarrow S\$$, where S is the start symbol for the original grammar. S' is the start symbol for the extended grammar.
2. For every occurrence of a nonterminal N on the right-hand side of a production, i.e., when there is a production $M \rightarrow \alpha N \beta$, where α and β are (possibly empty) sequences of grammar symbols, and N may or may not be equal to M , do the following:
 - 2.1. Let $m = FIRST(\beta)$. If $m \neq \emptyset$, add the constraint $m \subseteq FOLLOW(N)$ to the set of constraints.
 - 2.2. If $M \neq N$ and $Nullable(\beta)$, add the constraint $FOLLOW(M) \subseteq FOLLOW(N)$. Note that if β is empty, $Nullable(\beta)$ is trivially true.

Note that if a production has several occurrences of nonterminals on its right-hand side, step 2 is done for all of these.

3. Solve the constraints using the following steps:
 - 3.1. Start with empty sets for $FOLLOW(N)$ for all nonterminals N (except S' , which doesn't have a $FOLLOW$ set).
 - 3.2. For each constraint of the form $m \subseteq FOLLOW(N)$ constructed in step 2.1, add the contents of m to $FOLLOW(N)$.
 - 3.3. Iterating until a fixed-point is reached, for each constraint of the form $FOLLOW(M) \subseteq FOLLOW(N)$, add the contents of $FOLLOW(M)$ to $FOLLOW(N)$.

We can take Grammar 2.4 as an example of this. We first add the production

$$T' \rightarrow T\$$$

to the grammar to handle end-of-text conditions. The table below shows the constraints generated by each production.

Production	Constraints
$T' \rightarrow T\$$	$\{\$ \} \subseteq FOLLOW(T)$
$T \rightarrow R$	$FOLLOW(T) \subseteq FOLLOW(R)$
$T \rightarrow aTc$	$\{c\} \subseteq FOLLOW(T)$
$R \rightarrow$	
$R \rightarrow RbR$	$\{b\} \subseteq FOLLOW(R)$

In the above table, we have already calculated the required *FIRST* sets, so they are shown as explicit lists of terminals. To initialise the *FOLLOW* sets, we first use the constraints that involve these *FIRST* sets:

$$\begin{aligned} FOLLOW(T) &\supseteq \{\$, c\} \\ FOLLOW(R) &\supseteq \{b\} \end{aligned}$$

and then iterate calculation of the subset constraints. The only such constraint is $FOLLOW(T) \subseteq FOLLOW(R)$, so we get

$$\begin{aligned} FOLLOW(T) &\supseteq \{\$, c\} \\ FOLLOW(R) &\supseteq \{\$, c, b\} \end{aligned}$$

Now all constraints are satisfied, so we can replace subset with equality:

$$\begin{aligned} FOLLOW(T) &= \{\$, c\} \\ FOLLOW(R) &= \{\$, c, b\} \end{aligned}$$

If we return to the question of predictive parsing of Grammar 2.4, we see that, for the nonterminal R , we should choose the empty production on any symbol in $FOLLOW(R)$, i.e., $\{\$, c, b\}$, and choose the non-empty production on the symbols in $FIRST(RbR)$, i.e., $\{b\}$. Since these sets overlap (on the symbol b), we can not uniquely choose a production for R based on the next input symbol. Hence, the revised construction of predictive parsers (see below) will reject this grammar as possibly ambiguous.

2.10 A Larger Example

The above examples of calculating *FIRST* and *FOLLOW* are rather small, so we show a somewhat more substantial example. The following grammar describes even-length strings of a s and b s that are *not* of the form ww where w is any string of a s and b s.

In other words, a string can *not* consist of two identical halves, but otherwise any even-length sequence of as and bs is accepted.

$$\begin{aligned}
 N &\rightarrow A B \\
 N &\rightarrow B A \\
 A &\rightarrow a \\
 A &\rightarrow C A C \\
 B &\rightarrow b \\
 B &\rightarrow C B C \\
 C &\rightarrow a \\
 C &\rightarrow b
 \end{aligned}$$

The grammar is based on the observation that, if the string does not consist of two identical halves, there must be a point in the first part that has an a where the equivalent point in the second part has a b, or vice-versa. The grammar states that one of these is the case. The grammar is ambiguous, so we can not use predictive parsing, but it is used as a nontrivial example of calculation of *FIRST* and *FOLLOW* sets.

First, we note that there are no empty productions in the grammar, so no production can be *Nullable*. So we immediately set up the equations for *FIRST*:

$$\begin{aligned}
 FIRST(N) &= FIRST(A B) \cup FIRST(B A) \\
 &= FIRST(A) \cup FIRST(B) \\
 FIRST(A) &= FIRST(a) \cup FIRST(C A C) \\
 &= \{a\} \cup FIRST(C) \\
 FIRST(B) &= FIRST(b) \cup FIRST(C B C) \\
 &= \{b\} \cup FIRST(C) \\
 FIRST(C) &= FIRST(a) \cup FIRST(b) \\
 &= \{a, b\}
 \end{aligned}$$

which we solve by fixed-point iteration. We initially set the *FIRST* sets for the nonterminals to the empty sets, and iterate evaluation:

Nonterminal	Iteration 1	Iteration 2	Iteration 3
<i>N</i>	\emptyset	{a, b}	{a, b}
<i>A</i>	{a}	{a, b}	{a, b}
<i>B</i>	{b}	{a, b}	{a, b}
<i>C</i>	{a, b}	{a, b}	{a, b}

The last iteration did not add anything, so the fixed-point is reached. We now add the production $N' \rightarrow N\$$, and set up the constraints for calculating *FOLLOW* sets:

Production	Constraints
$N' \rightarrow N\$$	$\{\$ \} \subseteq FOLLOW(N)$
$N \rightarrow A B$	$FIRST(B) \subseteq FOLLOW(A), FOLLOW(N) \subseteq FOLLOW(B)$
$N \rightarrow B A$	$FIRST(A) \subseteq FOLLOW(B), FOLLOW(N) \subseteq FOLLOW(A)$
$A \rightarrow a$	
$A \rightarrow C A C$	$FIRST(A) \subseteq FOLLOW(C), FIRST(C) \subseteq FOLLOW(A),$ $FOLLOW(A) \subseteq FOLLOW(C)$
$B \rightarrow b$	
$B \rightarrow C B C$	$FIRST(B) \subseteq FOLLOW(C), FIRST(C) \subseteq FOLLOW(B),$ $FOLLOW(B) \subseteq FOLLOW(C)$
$C \rightarrow a$	
$C \rightarrow b$	

We first use the constraint $\{\$ \} \subseteq FOLLOW(N)$ and the constraints of the form $FIRST(\dots) \subseteq FOLLOW(\dots)$ to get the initial sets:

$$\begin{aligned} FOLLOW(N) &\supseteq \{\$ \} \\ FOLLOW(A) &\supseteq \{a, b\} \\ FOLLOW(B) &\supseteq \{a, b\} \\ FOLLOW(C) &\supseteq \{a, b\} \end{aligned}$$

and then use the constraints of the form $FOLLOW(\dots) \subseteq FOLLOW(\dots)$. If we do this in top-down order, we get after one iteration:

$$\begin{aligned} FOLLOW(N) &\supseteq \{\$ \} \\ FOLLOW(A) &\supseteq \{a, b, \$ \} \\ FOLLOW(B) &\supseteq \{a, b, \$ \} \\ FOLLOW(C) &\supseteq \{a, b, \$ \} \end{aligned}$$

Another iteration does not add anything, so the final result is

$$\begin{aligned} FOLLOW(N) &= \{\$ \} \\ FOLLOW(A) &= \{a, b, \$ \} \\ FOLLOW(B) &= \{a, b, \$ \} \\ FOLLOW(C) &= \{a, b, \$ \} \end{aligned}$$

Suggested exercises: 2.8 (*FOLLOW* only).

2.11 LL(1) Parsing

We have, in the previous sections, looked at how we can choose productions based on *FIRST* and *FOLLOW* sets, i.e., using the rule that we choose a production $N \rightarrow \alpha$ on input symbol c if either

- $c \in FIRST(\alpha)$, or
- $Nullable(\alpha)$ and $c \in FOLLOW(N)$.

If we can always choose a production uniquely by using these rules, this is called LL(1) parsing—the first L indicates the reading direction (left-to-right), the second L indicates the derivation order (left), and the (1) indicates that there is a one-symbol lookahead, i.e., that decisions require looking only at one input symbol (the next input symbol). A grammar where strings can be unambiguously parsed or rejected using LL(1) parsing is called an LL(1) grammar.

In the rest of this section, we shall see how we can implement LL(1) parsers as programs. We look at two implementation methods: Recursive descent, where grammar structure is directly translated into the program structure, and a table-based approach that encodes the production choices in a table, so a simple grammar-independent program can use the table to do parsing.

2.11.1 Recursive Descent

As the name indicates, *recursive descent* uses recursive functions to implement predictive parsing. The central idea is that each nonterminal in the grammar is implemented by a function in the program.

Each such function looks at the next input symbol in order to choose one of the productions for the nonterminal, using the criteria shown in the beginning of Sect. 2.11. The right-hand side of the chosen production is then used for parsing in the following way:

- A terminal on the right-hand side is matched against the next input symbol. If they match, we move on to the following input symbol and the next symbol on the right hand side, otherwise an error is reported.
- A nonterminal on the right-hand side is handled by calling the corresponding function and, after this call returns, continuing with the next symbol on the right-hand side.

When there are no more symbols on the right-hand side, the function returns.

As an example, Fig. 2.17 shows pseudo-code for a recursive descent parser for Grammar 2.9. We have constructed this program by the following process:

We have first added a production $T' \rightarrow T\$$ and calculated *FIRST* and *FOLLOW* for all productions.

T' has only one production, so the choice is trivial. However, we have added a check on the input symbol anyway, so we can report an error if it is not in $FIRST(T')$. This is shown in the function `parseT'`. The function `match` takes as argument a symbol, which it tests for equality with the symbol in the variable `input`. If they are equal, the following input symbol is read into the variable `input`. We assume `input` is initialised to the first input symbol before `parseT'` is called.

```

function parseT'() =
  if input = 'a' or input = 'b' or input = '$' then
    parseT() ; match('$')
  else reportError()

function parseT() =
  if input = 'b' or input = 'c' or input = '$' then
    parseR()
  else if input = 'a' then
    match('a') ; parseT() ; match('c')
  else reportError()

function parseR() =
  if input = 'c' or input = '$' then
    (* do nothing, just return *)
  else if input = 'b' then
    match('b') ; parseR()
  else reportError()

```

Fig. 2.17 Recursive descent parser for Grammar 2.9

For the `parseT` function, we look at the productions for T . As $FIRST(R) = \{b\}$, the production $T \rightarrow R$ is chosen on the symbol b . Since R is also *Nullable*, we must choose this production also on symbols in $FOLLOW(T)$, i.e., c or $\$$. $FIRST(aTc) = \{a\}$, so we select $T \rightarrow aTc$ on an a . On all other symbols we report an error.

For `parseR`, we must choose the empty production on symbols in $FOLLOW(R)$ (c or $\$$). The production $R \rightarrow bR$ is chosen on input b . Again, all other symbols produce an error.

The program in Fig. 2.17 does not build a syntax tree—it only checks if the input is valid. It can be extended to construct a syntax tree by letting the parse functions return the sub-trees for the parts of input that they parse. Pseudo-code for this is shown in Fig. 2.18. Note that, while decisions are made top-down, the syntax tree is built bottom-up by combining sub-trees from recursive calls. We use the functions `tNode` and `nNode` to build nodes in the syntax tree. `tNode` takes as argument a terminal symbol and builds a leaf node equal to that terminal. `nNode` takes as arguments the name of a nonterminal and a list of subtrees and builds a tree with the nonterminal as root and the subtrees as children. Lists are shown in square brackets with elements separated by commas.

2.11.2 Table-Driven LL(1) Parsing

In table-driven LL(1) parsing, we encode the selection of productions into a table instead of in the program text. A simple non-recursive program uses this table and a stack to perform the parsing.

```

function parseT'() =
  if input = 'a' or input = 'b' or input = '$' then
    let tree = parseT() in
      match('$');
      return tree
  else reportError()

function parseT() =
  if input = 'b' or input = 'c' or input = '$' then
    let tree = parseR() in
      return nNode('T', [tree])
  else if input = 'a' then
    match('a') ;
    let tree = parseT() in
      match('c') ;
      return nNode('T', [tNode('a'), tree, tNode('c')])
  else reportError()

function parseR() =
  if input = 'c' or input = '$' then
    return nNode('R', [])
  else if input = 'b' then
    match('b') ;
    let tree = parseR() in
      return nNode('R', [tNode('b'), tree])
  else reportError()

```

Fig. 2.18 Tree-building recursive descent parser for Grammar 2.9

Nonterminal	a	b	c	\$
T'	$T' \rightarrow T\$$	$T' \rightarrow T\$$		$T' \rightarrow T\$$
T	$T \rightarrow aTc$	$T \rightarrow R$	$T \rightarrow R$	$T \rightarrow R$
R		$R \rightarrow bR$	$R \rightarrow$	$R \rightarrow$

Fig. 2.19 LL(1) table for Grammar 2.9

The table is cross-indexed by nonterminal N and terminal a and contains for each such pair the production (if any) that is chosen for N when a is the next input symbol. This decision is made just as for recursive descent parsing: The production $N \rightarrow \alpha$ is written in the table at position (N, a) if either $a \in \text{FIRST}(\alpha)$, or if both $\text{Nullable}(\alpha)$ and $a \in \text{FOLLOW}(N)$. For Grammar 2.9 we get the table shown in Fig. 2.19.

Fig. 2.20 shows a program that uses this table to parse a string. It uses a stack, which at any time (read from top to bottom) contains the part of the current derivation that has not yet been matched to the input. When this eventually becomes empty, the parse is finished. If the stack is non-empty, and the top of the stack contains a terminal, that terminal is matched against the input and popped from the stack. Otherwise, the top of the stack must be a nonterminal, which we cross-index in the table with the next input symbol. If the table-entry is empty, we report an error. If

```

stack := empty ; push(T',stack)
while stack <> empty do
  if top(stack) is a terminal then
    match(top(stack)) ; pop(stack)
  else if table(top(stack),input) = empty then
    reportError
  else
    rhs := rightHandSide(table(top(stack),input)) ;
    pop(stack) ;
    pushList(rhs,stack)

```

Fig. 2.20 Program for table-driven LL(1) parsing

input	stack
aabbbcc\$	T'
aabbbcc\$	T\$
aabbbcc\$	aTc\$
abbbcc\$	Tc\$
abbbcc\$	aTcc\$
bbbcc\$	Tcc\$
bbbcc\$	Rcc\$
bbbcc\$	bRcc\$
bbcc\$	Rcc\$
bbcc\$	bRcc\$
bcc\$	Rcc\$
bcc\$	bRcc\$
cc\$	Rcc\$
cc\$	cc\$
c\$	c\$
\$	\$

Fig. 2.21 Input and stack during table-driven LL(1) parsing

not, we pop the nonterminal from the stack and replace this by the right-hand side of the production in the table entry. The list of symbols on the right-hand side are pushed such that the first symbol will end up at the top of the stack.

As an example, Fig. 2.21 shows the input and stack at each step during parsing of the string aabbbcc\$ using the table in Fig. 2.19. The stack is shown horizontally with the top to the left.

The program in Fig. 2.20, like the one in Fig. 2.17, only checks if the input is valid. It, too, can be extended to build a syntax tree. Figure 2.22 shows pseudo-code for this. The stack now holds nodes in the syntax tree instead of grammar symbols. The match function now matches a terminal (leaf) node with the next input symbol. The function makeNodes takes a list of grammar symbols and creates a list of nodes, one for each grammar symbol in its argument. A nonterminal node is created with an *uninitialised* mutable field for the list of its children. The T' -node that is pushed to the initial stack is a nonterminal node corresponding to the nonterminal T'. Note

```

stack := empty ; push(T'-node, stack)
while stack <> empty do
  node := pop(stack) ;
  if node is a terminal node then
    match(node) ; pop(stack)
  else
    nonterminal := node.nonterminal ;
    if table(nonterminal, input) = empty then
      reportError
    else
      rhs := rightHandSide(table(nonterminal, input)) ;
      node.children := makeNodes(rhs) ;
      pushList(node.children, stack)

```

Fig. 2.22 Tree-building program for table-driven LL(1) parsing

that the tree is built by first (in a top-down manner) creating nonterminal nodes with uninitialised fields and later (in a bottom-up manner) overwriting these uninitialised fields with subtrees.

2.11.3 Conflicts

When a symbol a allows several choices of production for nonterminal N we say that there is a *conflict* on that symbol for that nonterminal. Conflicts may be caused by ambiguous grammars (indeed all ambiguous grammars will cause conflicts) but there are also unambiguous grammars that cause conflicts. An example of this is the unambiguous expression grammar (Grammar 2.12). We will in the next section see how we can rewrite this grammar to avoid conflicts, but it must be noted that this is not always possible: There are languages for which there exist unambiguous context-free grammars but where no grammar for the language generates a conflict-free LL(1) table. Such languages are said to be non-LL(1). It is, however, important to note the difference between a non-LL(1) language and a non-LL(1) grammar: A language may well be LL(1) even though a grammar used to describe it is not. This just means that there is another grammar (which is LL(1)) for the same language. Our goal is to take a non-LL(1) grammar for a LL(1) language and transform it into an equivalent LL(1) grammar.

2.12 Rewriting a Grammar for LL(1) Parsing

In this section we will look at methods for rewriting grammars such that they are more palatable for LL(1) parsing. In particular, we will look at *elimination of left-recursion* and at *left factorisation*.

It must, however, be noted that not all unambiguous grammars can be rewritten to allow LL(1) parsing. In these cases stronger parsing techniques must be used. We will not cover parsing of ambiguous grammars in this book.

2.12.1 Eliminating Left-Recursion

As mentioned above, the unambiguous expression grammar (Grammar 2.12) is not LL(1). The reason is that all productions in Exp and $Exp2$ have the same *FIRST* sets. Overlap like this will always happen when there are directly or indirectly left-recursive productions in the grammar, as the *FIRST* set of a left-recursive production will include the *FIRST* set of the nonterminal itself and hence be a superset of the *FIRST* sets of all the other productions for that nonterminal. To solve this problem, we must avoid left-recursion in the grammar. We start by looking at elimination of direct left-recursion.

When we have a nonterminal with some left-recursive productions and some productions that are not left-recursive, i.e.,

$$\begin{aligned} N &\rightarrow N \alpha_1 \\ &\vdots \\ N &\rightarrow N \alpha_m \\ N &\rightarrow \beta_1 \\ &\vdots \\ N &\rightarrow \beta_n \end{aligned}$$

where the β_i do not start with N , we observe that the nonterminal N generates all sequences that start with one of the β_i and continue with any number (including 0) of the α_j . In other words, the grammar is equivalent to the regular expression $(\beta_1 | \dots | \beta_n)(\alpha_1 | \dots | \alpha_m)^*$. Some LL(1) parser generators accept grammars with right-hand sides of this form. When using such parser generators, no further rewriting is required. When using simple grammar notation, more rewriting is required, which we will look at below.

We saw in Fig. 2.1 a method for converting regular expressions into context-free grammars that generate the same set of strings. By following this procedure and simplifying a bit afterwards, we get this equivalent grammar:

$$\begin{aligned} N &\rightarrow \beta_1 N_* \\ &\vdots \\ N &\rightarrow \beta_n N_* \\ \\ N_* &\rightarrow \alpha_1 N_* \\ &\vdots \\ N_* &\rightarrow \alpha_m N_* \\ N_* &\rightarrow \end{aligned}$$

where N_* is a new nonterminal that generates a (possibly empty) sequence of α s.

Grammar 2.23 Removing
left-recursion from
Grammar 2.12

$$\begin{aligned} Exp &\rightarrow Exp2 Exp_* \\ Exp_* &\rightarrow + Exp2 Exp_* \\ Exp_* &\rightarrow - Exp2 Exp_* \\ Exp_* &\rightarrow \\ Exp2 &\rightarrow Exp3 Exp2_* \\ Exp2_* &\rightarrow * Exp3 Exp2_* \\ Exp2_* &\rightarrow / Exp3 Exp2_* \\ Exp2_* &\rightarrow \\ Exp3 &\rightarrow \mathbf{num} \\ Exp3 &\rightarrow (Exp) \end{aligned}$$

Note that, since the β_i do not start with N , there is no direct left-recursion in the first n productions. Since N_* is a new nonterminal, no α_j can start with this, so the last m productions can't be directly left-recursive either.

There may, however, still be *indirect* left-recursion: If an α_j is nullable, the corresponding production for N_* is indirectly left-recursive. If a β_i can derive something starting with N , the corresponding production for N is indirectly left-recursive. We will briefly look at indirect left-recursion below.

While we have eliminated direct left-recursion, we have also changed the syntax trees that are built from the strings that are parsed. Hence, after parsing, the syntax tree must be re-structured to obtain the structure that the original grammar describes. We will return to this in Sect. 2.16.

As an example of left-recursion removal, we take the unambiguous expression Grammar 2.12. This has left recursion in both Exp and $Exp2$, so we apply the transformation to both of these to obtain Grammar 2.23. The resulting Grammar 2.23 is now LL(1), which can be verified by generating an LL(1) table for it.

Indirect Left-Recursion

The transformation shown in Sect. 2.12.1 is only applicable in the simple case where there only *direct left-recursion*. Indirect left-recursion can have several forms:

1. There are mutually left-recursive productions

$$\begin{aligned} N_1 &\rightarrow N_2\alpha_1 \\ N_2 &\rightarrow N_3\alpha_2 \\ &\vdots \\ N_{k-1} &\rightarrow N_k\alpha_{k-1} \\ N_k &\rightarrow N_1\alpha_k \end{aligned}$$

2. There is a production $N \rightarrow \alpha N\beta$ where α is *Nullable*.

or any combination of the two. More precisely, a grammar is (directly or indirectly) left-recursive if there is a non-empty derivation sequence $N \Rightarrow N\alpha$, i.e., if a nonterminal derives a sequence of grammar symbols that start by that same nonterminal.

Grammar 2.24

Left-factorised grammar for conditionals

$$\begin{aligned} Stat &\rightarrow \mathbf{id} := Exp \\ Stat &\rightarrow \mathbf{if} Exp \mathbf{then} Stat \mathbf{ElsePart} \\ \\ ElsePart &\rightarrow \mathbf{else} Stat \\ ElsePart &\rightarrow \end{aligned}$$

If there is indirect left-recursion, we must first rewrite the grammar to make the left-recursion direct and then use the transformation above.

Rewriting a grammar to turn indirect left-recursion into direct left-recursion can be done systematically, but the process is a bit complicated. Details can be found in [2]. We will not go into this here, as in practice most cases of left-recursion are direct left-recursion.

2.12.2 Left-Factorisation

If two productions for the same nonterminal begin with the same sequence of symbols, they obviously have overlapping *FIRST* sets. As an example, in Grammar 2.3 the two productions for `if` have overlapping prefixes. We rewrite this in such a way that the overlapping productions are made into a single production that contains the common prefix of the productions and uses a new auxiliary nonterminal for the different suffixes. See Grammar 2.24. In this grammar,³ we can uniquely choose one of the productions for *Stat* based on one input token.

For most grammars, combining productions with common prefix will solve the problem. However, in this particular example the grammar still is not LL(1): We can not uniquely choose a production for the auxiliary nonterminal *ElsePart*, since `else` is in *FOLLOW*(*ElsePart*) as well as in the *FIRST* set of the first production for *ElsePart*. This should not be a surprise to us, since, after all, the grammar is ambiguous and ambiguous grammars can not be LL(1). The equivalent unambiguous grammar (Grammar 2.14) can not easily be rewritten to a form suitable for LL(1), so in practice Grammar 2.24 is used anyway and the conflict is handled not by rewriting to an unambiguous grammar, but by using an ambiguous grammar and resolving the conflict by prioritising productions. If the non-empty production for *ElsePart* has higher priority than the empty production, we will choose the non-empty production when the next input symbol is `else`. This gives the desired behaviour of letting an `else` match the nearest `if`.

Whenever an LL(1) table would have multiple choices of production for the same nonterminal/terminal pair, we use the priorities to select a single production. Most LL(1) parser generators prioritise productions by the order in which they are written, so Grammar 2.24 will give the desired behaviour. Unfortunately, few conflicts in

³ We have omitted the production for semicolon, as that would only muddle the issue by introducing more ambiguity.

LL(1) tables can be removed by prioritising productions without also changing the language recognised by the grammar. For example, operator precedence ambiguity can not be resolved by prioritising productions. Attempting to do so will cause parse errors for some valid expressions.

2.12.3 Construction of LL(1) Parsers Summarised

Constructing an LL(1) parser from a given grammar is done in the following steps.

1. Eliminate ambiguity that can not be resolved by prioritising productions.
2. Eliminate left-recursion.
3. Perform left factorisation where required.
4. Add an extra start production $S' \rightarrow S\$$ to the grammar.
5. Calculate *FIRST* for every production and *FOLLOW* for every nonterminal.
6. For nonterminal N and input symbol c , choose production $N \rightarrow \alpha$ when:
 - $c \in \text{FIRST}(\alpha)$, or
 - $\text{Nullable}(\alpha)$ and $c \in \text{FOLLOW}(N)$.

This choice is encoded either in a table or a recursive-descent program.

7. Use production priorities to eliminate conflicts where appropriate.

Suggested exercises: 2.14.

2.13 SLR Parsing

A problem with LL(1) parsing is that most grammars need extensive rewriting to get them into a form that allows unique choice of production. Even though this rewriting can, to a large extent, be automated, there are still a large number of grammars that can not be automatically transformed into LL(1) grammars.

LR parsers is a class of bottom-up methods for parsing that can solve the parsing problem for a much larger class of grammars than LL(1) parsing, though still not all grammars. The main advantage of LR parsing is that less rewriting is required to get a grammar in acceptable form for LR parsing than is the case for LL(1) parsing. Furthermore, as we shall see in Sect. 2.15, LR parsers allow external declarations for resolving operator precedences, instead of requiring the grammars themselves to be rewritten.

We will look at a simple form of LR-parsing called SLR parsing. The letters “SLR” stand for “Simple”, “Left” and “Right”. “Left” indicates that the input is read from left to right and the “Right” indicates that a rightmost derivation is built.

LR parsers are also called *shift-reduce parsers*. They are table-driven bottom-up parsers and use two kinds of “actions” involving the input stream and a stack:

Fig. 2.25 Example shift-reduce parsing

stack	input	action
	aabbbcc	shift 5 times
aabbb	cc	reduce with $R \rightarrow$
aabbbR	cc	reduce with $R \rightarrow bR$
aabbR	cc	reduce with $R \rightarrow bR$
aabR	cc	reduce with $R \rightarrow bR$
aaR	cc	reduce with $T \rightarrow R$
aaT	cc	shift
aaTc	c	reduce with $T \rightarrow aTc$
aT	c	shift
aTc		reduce with $T \rightarrow aTc$
T		

shift: A terminal symbol is read from the input and pushed on the stack.

reduce: The top n elements of the stack hold symbols identical to the n symbols on the right-hand side of a specified production. These n symbols are by the reduce action replaced by the nonterminal at the left-hand side of the specified production. Contrary to LL(1) parsers, the stack holds the right-hand-side symbols such that the *last* symbol on the right-hand side is at the top of the stack.

If the input text does not conform to the grammar, there will at some point during the parsing be no applicable actions, and the parser will stop with an error message. Otherwise, the parser will read through all the input and leave a single element (the start symbol of the grammar) on the stack. To illustrate shift-reduce parsing, Fig. 2.25 shows a sequence of shift and reduce actions corresponding to parsing the string aabbbcc using Grammar 2.9. The stack is shown growing left to right, so the rightmost stack element is the top. In this example, we do not explain how we select between shift and reduce, but we want to make well-informed choices and to detect potential ambiguity. This is done using SLR parsing.

As with LL(1), our aim is to make the choice of action depend only on the next input symbol and the symbol on top of the stack. To help make this choice, we use a DFA. Conceptually, this DFA reads the contents of the stack (which contains both terminals and nonterminals), starting from the bottom up to the top. The state of the DFA when the top of the stack is reached is, together with the next input symbol, used to determine the next action. Like in LL(1) parsing, this is done using a table, but we use a DFA state instead of a nonterminal to select the row in the table, and the table entries are not productions but actions.

If the action is a shift action, there is no need to start over from the bottom of the stack to find the next action: We just push the input symbol and follow a transition from the current DFA state on this symbol, which gives us the DFA state we need for the next choice.

If the action is a reduce action, we pop off the stack symbols corresponding to the right-hand side of the selected production, and then push the nonterminal on the left-hand side of this production. To make a DFA transition on this nonterminal, we need to know the state of the DFA when reading the stack from bottom to the new

top. To avoid having to start over from the bottom, we remember the transitions we have already made, so the stack holds not only grammar symbols but also states. When we pop a symbol off the stack, we can find the previous DFA state on the new stack top. This is similar to how a lexical analyser remembers past states so it can find the most recent accepting state if a transition fails (see Sect. 1.8). So, when a reduce action pops off the symbols corresponding to the right-hand side of a production, we can find the DFA state that we use to make a transition on the nonterminal on the left-hand side of the production. This nonterminal is pushed on to the new stack together with the new state.

With these optimisations, the DFA only has to make one transition when an action is made: A transition on a terminal when a shift action is made, and a transition on a nonterminal when a reduce action is made.

We represent the DFA as a table, where we cross-index a DFA state with a symbol (terminal or nonterminal) and find one of the following actions:

- shift n*: Push the current input symbol and then state n on the stack, and read the next input symbol. This corresponds to a transition on a terminal.
- go n*: Push the nonterminal indicated by the column and then state n on the stack. This corresponds to a transition on a nonterminal.
- reduce p*: Reduce with the production numbered p : Pop symbols (interleaved with state numbers) corresponding to the right-hand side of the production off the stack. This is always followed by a *go* action on the left-hand side nonterminal using the DFA state that is found *after* popping the right-hand side off the stack.
- accept*: Parsing has completed successfully.
- error*: A syntax error has been detected. This happens when no *shift*, *accept* or *reduce* action is defined for the input symbol.

Note that the current state is always found at the top of the stack.

An example SLR table is shown in Fig. 2.26. The table has been produced from Grammar 2.9 by the method shown below in Sect. 2.14. The actions have been abbreviated to their first letters and *error* is shown as a blank entry. The rows are indexed by DFA states (0 to 7) and the columns are indexed by terminals (including \$) or nonterminals.

The algorithm for parsing a string using the table is shown in Fig. 2.27. The shown algorithm just determines if a string is in the language generated by the grammar. It can, however, easily be extended to build a syntax tree: Instead of grammar symbols, the stack contains syntax trees. When performing a *reduce* action, a new syntax tree is built by using the nonterminal from the reduced production as root and the syntax trees stored at the popped-off stack elements as children. The new tree is pushed on the stack instead of just pushing the nonterminal.

Figure 2.28 shows an example of parsing the string `aabbbcc` using the table in Fig. 2.26. The “stack” column represents the stack contents with the stack bottom shown to the left and the stack top to the right. We interleave grammar symbols and states on the stack, always leaving the current state on the top (at the right). At each step, we look at the current input symbol (at the left end of the string in the input column) and the state at the top of the stack (at the right end of the sequence in the

State	a	b	c	\$	T	R
0	s3	s4	r3	r3	g1	g2
1				a		
2			r1	r1		
3	s3	s4	r3	r3	g5	g6
4		s4	r3	r3		
5			s7			
6			r4	r4		
7			r2	r2		

Fig. 2.26 SLR table for Grammar 2.9

```

stack := empty ; push(0,stack) ; read(input)
loop
  case table[top(stack),input] of
    shift s: push(input,stack) ;
              push(s,stack) ;
              read(input)

    reduce p: n := the left-hand side of production p ;
              r := the number of symbols
                on the right-hand side of p ;
              pop 2*r elements from the stack ;
              let go s = table[top(stack),n];
              push(n,stack) ;
              push(s,stack)

    accept:  terminate with success

    error:   reportError
  endloop

```

Fig. 2.27 Algorithm for SLR parsing

stack column). We look up the pair of input symbol and state in the table and find the action (shown in the action column) that leads to the stack and input shown in next row. When the action is a *reduce* action, we also show the reduction used (in parentheses), and after a semicolon also the *go* action that is performed after the reduction. At the end, the root nonterminal *T* is found as the second stack element. If a syntax tree is built, this will be placed here.

2.14 Constructing SLR Parse Tables

An SLR parse table has a DFA as its core. Constructing this DFA from the grammar is similar to constructing a DFA from a regular expression, as shown in Chap. 2: We

stack	input	action
0	aabbbcc\$	s3
0a3	abbbcc\$	s3
0a3a3	bbbcc\$	s4
0a3a3b4	bbcc\$	s4
0a3a3b4b4	bcc\$	s4
0a3a3b4b4b4	cc\$	r3 ($R \rightarrow$); g6
0a3a3b4b4b4R6	cc\$	r4 ($R \rightarrow bR$); g6
0a3a3b4b4R6	cc\$	r4 ($R \rightarrow bR$); g6
0a3a3b4R6	cc\$	r4 ($R \rightarrow bR$); g2
0a3a3R2	cc\$	r1 ($T \rightarrow R$); g5
0a3a3T5	cc\$	s7
0a3a3T5c7	c\$	r2 ($T \rightarrow aTc$); g5
0a3T5	c\$	s7
0a3T5c7	\$	r2 ($T \rightarrow aTc$); g1
0T1	\$	accept

Fig. 2.28 Example SLR parsing

- 0: $T' \rightarrow T$
- 1: $T \rightarrow R$
- 2: $T \rightarrow aTc$
- 3: $R \rightarrow$
- 4: $R \rightarrow bR$

Grammar 2.29 Example grammar for SLR-table construction

first construct an NFA using techniques similar to those in Sect. 1.3 and then convert this into a DFA using the construction shown in Sect. 1.5.

Before we construct the NFA, we extend the grammar with a new starting production. Doing this to Grammar 2.9 yields Grammar 2.29.

The next step is to make an NFA for each production. This is done as in Sect. 1.3, treating both terminals and nonterminals as alphabet symbols. The accepting state of each NFA is labeled with the number of the corresponding production. The result is shown in Fig. 2.30. Note that we have used the optimised construction for ϵ (the empty production) as shown in Fig. 1.6. For identification purposes, we label the states with letters.

The NFAs in Fig. 2.30 make transitions both on terminals and nonterminals. Transitions by terminal corresponds to *shift* actions and transitions on nonterminals correspond to *go* actions. A *go* action happens after a reduction, so before we can make a transition on a nonterminal, we must on the stack have symbols corresponding to a right-hand side of a production for that nonterminal. So whenever an NFA can make a transition on a nonterminal, we add epsilon transitions to the NFAs for the right-hand sides of the productions for that nonterminal. This way, we can make transitions for a right-hand side, make a reduction, and then a transition on the nonterminal.

Fig. 2.30 NFAs for the productions in Grammar 2.29

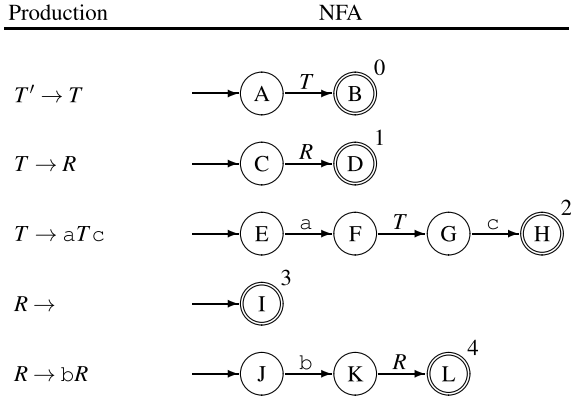
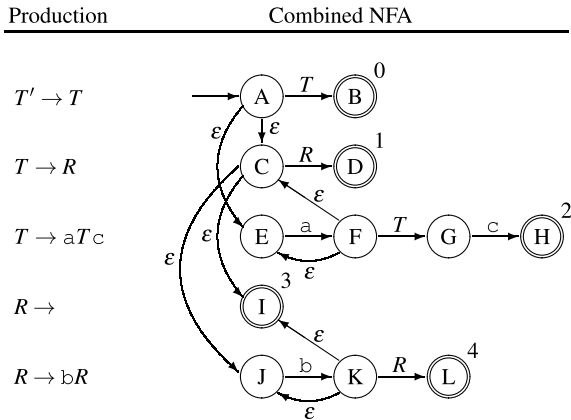


Fig. 2.31 Combined NFA for Grammar 2.29: epsilon transitions are added, and A is the only start state



Finally, we combine the NFAs to a single NFA by letting A (the start state of the production for the added start symbol T') be the only initial state. The result is shown in Fig. 2.31.

We must now convert this NFA into a DFA using the subset construction shown in Sect. 1.5.2. The result is shown in Fig. 2.32. The states are labelled with the sets of NFA states that are combined into the DFA states.

From this DFA, we construct a table where transitions on terminals are shown as *shift* actions and transitions on nonterminals as *go* actions. We use state number 0 for the starting state of the DFA. The order of the other states is not important, but we have numbered them in the order they were generated using the work-list algorithm from Sect. 1.5.2. The table looks similar to Fig. 2.26, except that it has an extra column for sets of NFA states and that no *reduce* or *accept* actions are present yet. Figure 2.33 shows the table constructed from the DFA in Fig. 2.32. The sets of NFA states that form each DFA state is shown in the second column. We will need these below for adding *reduce* and *accept* actions, but once this is done, we will not need them anymore, so we can remove them from the final table.

Fig. 2.32 DFA constructed from the NFA in Fig. 2.31

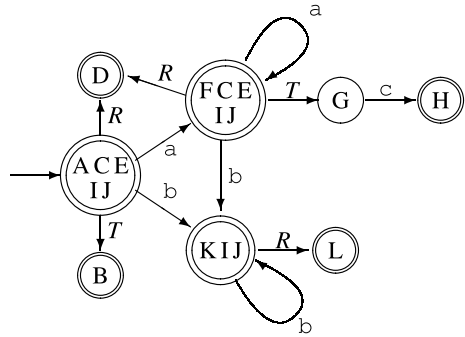


Fig. 2.33 DFA table for Grammar 2.9, equivalent to the DFA in Fig. 2.32

DFA state	NFA states	Transitions				
		a	b	c	T	R
0	A, C, E, I, J	s3	s4		g1	g2
1	B					
2	D					
3	F, C, E, I, J	s3	s4		g5	g2
4	K, I, J		s4			g6
5	G			s7		
6	L					
7	H					

To add *reduce* and *accept* actions, we first need to compute the *FOLLOW* sets for each nonterminal, as described in Sect. 2.9. For purpose of calculating *FOLLOW*, we add yet another extra start production: $T'' \rightarrow T'\$,$ to handle end-of-text conditions as described in Sect. 2.9. This gives us the following result:

$$\begin{aligned}
 FOLLOW(T') &= \{\$ \} \\
 FOLLOW(T) &= \{c, \$ \} \\
 FOLLOW(R) &= \{c, \$ \}
 \end{aligned}$$

We now add *reduce* actions by the following rule: If a DFA state s contains the accepting NFA state for a production $p : N \rightarrow \alpha,$ we add *reduce p* as action to s on all symbols in $FOLLOW(N).$ Reduction for production 0 (the extra start production that was added before constructing the NFA) on the $\$$ symbol is written as *accept*.

In Fig. 2.33, state 0 contains NFA state I, which accepts production 3. Hence, we add r3 as actions at the symbols c and $\$$ (as these are in $FOLLOW(R).$) State 1 contains NFA state B, which accepts production 0. Since $FOLLOW(T') = \{\$, \}$ we add a reduce action for production 0 at $\$.$ As noted above, this is written as *accept* (abbreviated to “a”). In the same way, we add reduce actions to state 3, 4, 6 and 7. The result is shown in Fig. 2.26.

Figure 2.34 summarises the SLR construction.

1. Add the production $S' \rightarrow S$, where S is the start symbol of the grammar.
2. Make an NFA for the right-hand side of each production.
3. If an NFA state s has an outgoing transition on a nonterminal N , add epsilon-transitions from s to the starting states of the NFAs for the right-hand sides of the productions for N .
4. Make the start state of the NFA for the production $S' \rightarrow S$ the single start state of the combined NFA.
5. Convert the combined NFA to a DFA.
6. Build a table cross-indexed by the DFA states and grammar symbols (terminals including $\$$ and nonterminals). Add *shift* actions for transitions on terminals and *go* actions for transitions on nonterminals.
7. Calculate *FOLLOW* for each nonterminal. For this purpose, we add one more start production: $S'' \rightarrow S'\$$.
8. When a DFA state contains an accepting NFA state marked with production number p , where the left-hand side nonterminal for p is N , find the symbols in *FOLLOW*(N) and add a *reduce* p action in the DFA state at all these symbols. If $p = 0$, add an *accept* action instead of a *reduce* p action.

Fig. 2.34 Summary of SLR parse-table construction

2.14.1 Conflicts in SLR Parse-Tables

When *reduce* actions are added to SLR parse-tables, we might add a *reduce* action where there is already a *shift* action, or we may add *reduce* actions for two or more different productions to the same table entry. When either of these happen, we no longer have a unique choice of action, i.e., we have a *conflict*. The first situation is called a *shift-reduce conflict* and the other case a *reduce-reduce conflict*. Both can occur in the same table entry.

Conflicts are often caused by ambiguous grammars, but (as is the case for LL-parsers) some non-ambiguous grammars can generate conflicts. If a conflict is caused by an ambiguous grammar, it is usually (but not always) possible to find an equivalent unambiguous grammar. Methods for eliminating ambiguity were discussed in Sects. 2.3 and 2.4. Sometimes, operator precedence declarations can be used to disambiguate an ambiguous grammar, as we shall see in Sect. 2.15. In rare cases, a language is simply not SLR, so no language-preserving rewrites or use of precedence declarations will eliminate conflicts.

When a conflict is found, inspection of the NFA states that form the problematic DFA state will often help identifying the exact nature of the problem, which is the first step towards solving it. Sometimes, changing a production from left-recursive to right-recursive may help, even though left-recursion in general is not a problem for SLR-parsers, as it is for LL(1)-parsers. It may also help to rewrite the grammar in the following way: If there are productions of the form

$$\begin{aligned}
 A &\rightarrow \alpha B \beta \\
 A &\rightarrow \alpha \gamma_1 \delta \\
 B &\rightarrow \gamma_1 \\
 &\vdots \\
 B &\rightarrow \gamma_n
 \end{aligned}$$

and there is overlap between $FIRST(\delta)$ and $FOLLOW(B)$, then there will be a shift-reduce conflict after reading $\alpha \gamma_1$, as both reduction with $B \rightarrow \gamma_1$ and shifting on any symbol in $FIRST(\delta)$ is possible, which gives a conflict for all symbols in $FIRST(\delta) \cap FOLLOW(B)$. This conflict can be resolved by splitting the first production above into all the possible cases for B :

$$\begin{aligned}
 A &\rightarrow \alpha \gamma_1 \beta \\
 &\vdots \\
 A &\rightarrow \alpha \gamma_n \beta \\
 A &\rightarrow \alpha \gamma_1 \delta
 \end{aligned}$$

The shift-reduce conflict we had when having read $\alpha \gamma_1$ is now gone, as we have postponed reduction until we have read more input, which can determine if we should reduce by the production $A \rightarrow \alpha \gamma_1 \beta$ or by the production $A \rightarrow \alpha \gamma_1 \delta$. See also Sects. 2.15 and 2.16.1.

Suggested exercises: 2.16.

2.15 Using Precedence Rules in LR Parse Tables

We saw in Sect. 2.12.2, that the conflict arising from the dangling-else ambiguity could be removed by removing one of the entries in the LL(1) parse table. Resolving ambiguity by deleting conflicting actions can also be done in SLR-tables. In general, there are more cases where this can be done successfully for SLR-parsers than for LL(1)-parsers. In particular, ambiguity in expression grammars like Grammar 2.2 can be eliminated this way in an SLR table, but not in an LL(1) table. Most LR-parser generators allow declarations of precedence and associativity for tokens used as infix-operators. These declarations are then used to eliminate conflicts in the parse tables.

There are several advantages to this approach:

- Ambiguous expression grammars are more compact and easier to read than unambiguous grammars in the style of Sect. 2.3.1.
- The parse tables constructed from ambiguous grammars are often smaller than tables produced from equivalent unambiguous grammars.
- Parsing using ambiguous grammars is (slightly) faster, as fewer reductions of the form $Exp2 \rightarrow Exp3$ etc. are required.

Using precedence rules to eliminate conflicts is very simple. Grammar 2.2 will generate several conflicts:

- 1) A conflict between shifting on + and reducing by the production
 $Exp \rightarrow Exp + Exp.$
- 2) A conflict between shifting on + and reducing by the production
 $Exp \rightarrow Exp * Exp.$
- 3) A conflict between shifting on * and reducing by the production
 $Exp \rightarrow Exp + Exp.$
- 4) A conflict between shifting on * and reducing by the production
 $Exp \rightarrow Exp * Exp.$

And several more of similar nature involving – and /, for a total of 16 conflicts. Let us take each of the four conflicts above in turn and see how precedence rules can be used to eliminate them. We use the usual convention that + and * are both left-associative and that * binds more strongly than +.

- 1) This conflict arises from expressions like $a+b+c$. After having read $a+b$, the next input symbol is +. We can now either choose to reduce $a+b$, grouping around the first addition before the second, or shift on the plus, which will later lead to $b+c$ being reduced, and hence grouping around the second addition before the first. Since the convention is that + is left-associative, we prefer the first of these options and, hence, eliminate the shift-action from the table and keep only the reduce-action.
- 2) The offending expressions here have the form $a*b+c$. Since convention make multiplication bind stronger than addition, we, again, prefer reduction over shifting.
- 3) In expressions of the form $a+b*c$, the convention, again, makes multiplication bind stronger, so we prefer a shift to avoid grouping around the + operator and, hence, eliminate the reduce-action from the table.
- 4) This case is identical to case 1, where an operator that by convention is left-associative conflicts with itself. We, as in case 1, handle this by eliminating the shift.

In general, elimination of conflicts by operator precedence declarations can be summarised into the following rules:

- a) If the conflict is between two operators of different priority, eliminate the action with the lowest priority operator in favour of the action with the highest priority. In a reduce action, the operator associated with a reduce-action is an operator used in the production that is reduced. If several operators are used in the same production, the operator that is closest to the end of the production is used.⁴
- b) If the conflict is between operators of the same priority, the associativity (which must be the same, as noted in Sect. 2.3.1) of the operators is used: If the operators are left-associative, the shift-action is eliminated and the reduce-action retained.

⁴ Using several operators with declared priorities in the same production should be done with care.

If the operators are right-associative, the reduce-action is eliminated and the shift-action retained. If the operators are non-associative, both actions are eliminated (which will reduce the set of accepted programs).

Prefix and postfix operators can be handled similarly. Associativity only applies to infix operators, so only the precedence of prefix and postfix operators matters.

Note that only shift-reduce conflicts are eliminated by the above rules. Some parser generators allow also reduce-reduce conflicts to be eliminated by precedence rules (in which case the production with the highest-precedence operator is preferred), but this is not as obviously useful as the above.

The dangling-else ambiguity (Sect. 2.4) can also be eliminated using precedence rules. If we have read `if Exp then Stat` and the next symbol is a `else`, we want to shift on `else`, so the `else` will be associated with the `then`. Giving `else` a higher precedence than `then` or giving them both the same precedence and making them right-associative will ensure that a shift is made on `else` when we need it.

Not all conflicts should be eliminated by precedence rules. If you blindly add precedence rules until no conflicts are reported, you risk eliminating actions that are required to parse certain strings, so the parser will accept only a subset of the intended language. Normally, you should only use precedence declarations to specify operator hierarchies, unless you have analysed the parser actions carefully and found that there is no undesirable consequences of adding the precedence rules.

Suggested exercises: 2.18.

2.16 Using LR-Parser Generators

Most LR-parser generators use an extended version of the SLR construction called LALR(1). The “LA” in the abbreviation is short for “lookahead” and the (1) indicates that the lookahead is one symbol, i.e., the next input symbol. LALR(1) parser tables have fewer conflicts than SLR parser tables.

We have chosen to present the SLR construction instead of the LALR(1) construction for several reasons:

- It is simpler.
- In practice, SLR parse tables rarely have conflicts that would not also be conflicts in LALR(1) tables.
- When a grammar is in the SLR class, the parse-table produced by an SLR parser generator is identical to the table produced by an LALR(1) parser generator.
- If you use an LALR(1) parser generator and you do not get any conflicts, you do not need to worry about the difference.
- If you use an LALR(1) parser generator and you *do* get conflicts, understanding SLR parsing is sufficient to deal with the conflicts (by adding precedence declarations or by rewriting the grammar).

In short, the practical difference is small, and knowledge of SLR parsing is sufficient when using LALR(1) parser generators.

Most LR-parser generators organise their input in several sections:

- Declarations of the terminals and nonterminals used.
- Declaration of the start symbol of the grammar.
- Declarations of operator precedence.
- The productions of the grammar.
- Declaration of various auxiliary functions and data-types used in the actions (see below).

2.16.1 Conflict Handling in Parser Generators

For all but the simplest grammars, the user of a parser generator should expect conflicts to be reported when the grammar is first presented to the parser generator. These conflicts can be caused by ambiguity or by the limitations of the parsing method. In any case, the conflicts can normally be eliminated by rewriting the grammar or by adding precedence declarations.

Most parser generators can provide information that is useful to locate where in the grammar the problems are. When a parser generator reports a conflict, it will tell in which state in the table the conflict occur. Information about this state can be written out in a (barely) human-readable form as a set of NFA-states. Since most parser generators rely on pure ASCII, they can not actually draw the NFAs as diagrams. Instead, they rely on the fact that each state in a NFA corresponds to a position in a production in the grammar. If we, for example, look at the NFA states in Fig. 2.30, these would be written as shown in Fig. 2.35. Note that a ‘.’ is used to indicate the position of the state in the production. State 4 of the table in Fig. 2.33 will hence be written as

```
R -> b . R
R -> .
R -> . bR
```

The set of NFA states, combined with information about on which symbols a conflict occurs, can be used to find a remedy, e.g., by adding precedence declarations. Note that a dot at the end of a production indicates an accepting NFA state (and, hence, a possible reduce action) while a dot before a terminal indicates a possible shift action. That both of these appear (as above) in the same DFA state does not imply a conflict—the symbols on which the reduce action is taken may not overlap the symbols on which shift actions are taken.

If all efforts to eliminate conflicts fail, a practical solution may be to change the grammar so it unambiguously accepts a larger language than the intended language, and then post-process the syntax tree to reject “false positives”. This elimination can be done at the same time as type-checking (which, too, may reject programs).

Fig. 2.35 Textual representation of NFA states

NFA-state	Textual representation
A	T' -> . T
B	T' -> T .
C	T -> . R
D	T -> R .
E	T -> . aTc
F	T -> a . Tc
G	T -> aT . c
H	T -> aTc .
I	R -> .
J	R -> . bR
K	R -> b . R
L	R -> bR .

Some programming languages allow programs to declare precedence and associativity for user-defined operators. This can make it difficult to handle precedence during parsing, as the precedences are not known when the parser is generated. A possible solution is to parse all operators using the same precedence and associativity, and then restructure the syntax tree afterwards. See Exercise 2.20 for other approaches.

2.16.2 Declarations and Actions

Each nonterminal and terminal is declared and associated with a data-type. For a terminal, the data-type is used to hold the values that are associated with the tokens that come from the lexer, *e.g.*, the values of numbers or names of identifiers. For a nonterminal, the type is used for the values that are built for the nonterminals during parsing (at reduce-actions), typically syntax trees.

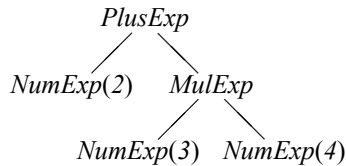
While, conceptually, parsing a string produces a syntax tree for that string, parser generators usually allow more control over what is actually produced. This is done by assigning an *action* to each production. The action is a piece of program text that is used to calculate the value of a production that is being reduced by using the values associated with the symbols on the right-hand side. For example, by putting appropriate actions on each production, the numerical value of an expression may be calculated as the result of parsing the expression. Indeed, compilers can be made such that the value produced during parsing is the compiled code of a program. For all but the simplest compilers it is, however, better to build a (possibly abstract) syntax tree during parsing and then later operate on this representation.

2.16.3 Abstract Syntax

The syntax trees described in Sect. 2.2.1 are not always optimally suitable for compilation. They contain a lot of redundant information: Parentheses, keywords used

for grouping purposes only, and so on. They also reflect structures in the grammar that are only introduced to eliminate ambiguity or to get the grammar accepted by a parser generator (such as left-factorisation or elimination of left-recursion). Hence, actions usually generate *abstract syntax trees* instead of precise syntax trees.

Abstract syntax keeps the essence of the structure of the text but omits the irrelevant details. An *abstract syntax tree* is a tree structure where each node corresponds to one or more nodes in the (concrete) syntax tree. For example, the concrete syntax tree shown in Fig. 2.13 may be represented by the following abstract syntax tree:



Here the names *PlusExp*, *MulExp* and *NumExp* may be constructors in a data-type, they may be elements from an enumerated type used as tags in a union-type or they may be names of subclasses of an *Exp* class. The names indicate which production is chosen, so there is no need to retain the subtrees that are implied by the choice of production, such as the subtree from Fig. 2.13 that holds the symbol $+$. Likewise, the sequence of nodes *Exp*, *Exp2*, *Exp3*, 2 at the left of Fig. 2.13 are combined to a single node *NumExp(2)* that includes both the choice of productions for *Exp*, *Exp2* and *Exp3* and the value of the terminal node. In short, each node in the abstract syntax tree corresponds to one or more nodes in the concrete syntax tree.

A designer of a compiler or interpreter has much freedom in the choice of abstract syntax. Some use abstract syntax that retain all of the structure of the concrete syntax trees plus additional positioning information used for error-reporting. Others prefer abstract syntax that contains only the information necessary for compilation or interpretation, skipping parentheses and other (for compilation or interpretation) irrelevant structure, like we did above.

Exactly how the abstract syntax tree is represented and built depends on the parser generator used. Normally, the action assigned to a production can access the values of the terminals and nonterminals on the right-hand side of a production through specially named variables (often called $\$1$, $\$2$, etc.) and produces the value for the node corresponding to the left-hand-side either by assigning it to a special variable ($\$0$) or letting it be the value of an action expression.

The data structures used for building abstract syntax trees depend on the language. Most statically typed functional languages support tree-structured datatypes with named constructors. In such languages, it is natural to represent abstract syntax by one datatype per syntactic category (e.g., *Exp* above) and one constructor for each instance of the syntactic category (e.g., *PlusExp*, *NumExp* and *MulExp* above). In Pascal, a syntactic category can be represented by a variant record type and each instance as a variant of that. In C, a syntactic category can be represented by a union of structs, each struct representing an instance of the syntactic category and the union covering all possible instances. In object-oriented languages such as Java, a syntactic

category can be represented as an abstract class or interface where each instance in a syntactic category is a concrete class that implements the abstract class or interface. Alternatively, a syntactic category can be represented by an enumerated type with properties.

In most cases, it is fairly simple to build abstract syntax using the actions for the productions in the grammar. It becomes complex only when the abstract syntax tree must have a structure that differs nontrivially from the concrete syntax tree.

One example of this is if left-recursion has been eliminated for the purpose of making an LL(1) parser. The preferred abstract syntax tree will in most cases be similar to the concrete syntax tree of the original left-recursive grammar rather than that of the transformed grammar. As an example, the left-recursive grammar

$$\begin{aligned} E &\rightarrow E + \mathbf{num} \\ E &\rightarrow \mathbf{num} \end{aligned}$$

gets transformed by left-recursion elimination into

$$\begin{aligned} E &\rightarrow \mathbf{num} E_* \\ E_* &\rightarrow + \mathbf{num} E_* \\ E_* &\rightarrow \end{aligned}$$

Which yields a completely different syntax tree. We can use the actions assigned to the productions in the transformed grammar to build an abstract syntax tree that reflects the structure in the original grammar.

In the transformed grammar, E_* can return an abstract syntax tree with a *hole*. The intention is that this hole will eventually be filled by another abstract syntax tree:

- The second production for E_* (the empty production) returns just a hole.
- In the first production for E_* , the $+$ and \mathbf{num} terminals are used to produce a tree for a plus-expression (i.e., a *PlusExp* node) with a hole in place of the first subtree. This tree is itself used to fill the hole in the tree returned by the recursive use of E_* , so the abstract syntax tree is essentially built outside-in. The result is a new tree with a hole.
- In the production for E , the hole in the tree returned by the E_* nonterminal is filled by a *NumExp* node with the number that is the value of the \mathbf{num} terminal.

The best way of building trees with holes depends on the type of language used to implement the actions. Let us first look at the case where a functional language is used.

The actions shown below for the original grammar will build an abstract syntax tree similar to the one shown in the beginning of this section.

$$\begin{aligned} E &\rightarrow E + \mathbf{num} \{ \text{PlusExp} (\$1, \text{NumExp} (\$3)) \} \\ E &\rightarrow \mathbf{num} \{ \text{NumExp} (\$1) \} \end{aligned}$$

We now want to make actions for the transformed grammar that will produce the same abstract syntax trees as the above actions for the original grammar will.

In functional languages, an abstract syntax tree with a hole can be represented by a function. The function takes as argument what should be put into the hole, and returns a syntax tree where the hole is filled with this argument. The hole is represented by the argument variable of the function. We can write this as actions to the transformed grammar:

$$\begin{aligned} E &\rightarrow \mathbf{num} E_* \quad \{ \$2 (\text{NumExp} (\$1)) \} \\ E_* &\rightarrow + \mathbf{num} E_* \quad \{ \lambda x. \$3 (\text{PlusExp} (x, \text{NumExp} (\$2))) \} \\ E_* &\rightarrow \quad \quad \quad \{ \lambda x. x \} \end{aligned}$$

where an expression of the form $\lambda x.e$ is an anonymous function that takes x as argument and returns the value of the expression e . The empty production returns the identity function, which works like a top-level hole. The non-empty production for E_* applies the function $\$3$ returned by the E_* on the right-hand side to a subtree, hence filling the hole in $\$3$ by this subtree. The subtree itself has a hole x , which is filled when applying the function returned by the right-hand side. The production for E applies the function $\$2$ returned by E_* to a subtree that has no holes and, hence, returns a tree with no holes.

In SML, $\lambda x.e$ is written as `fn x => e`, in F# as `fun x -> e`, in Haskell as `\x -> e`, and in Scheme as `(lambda (x) e)`.

An imperative version of the actions in the original grammar could be

$$\begin{aligned} E &\rightarrow E + \mathbf{num} \quad \{ \$0 = \text{PlusExp} (\$1, \text{NumExp} (\$3)) \} \\ E &\rightarrow \mathbf{num} \quad \quad \quad \{ \$0 = \text{NumExp} (\$1) \} \end{aligned}$$

In this setting, `NumExp` and `PlusExp` can be class constructors or functions that allocate and build nodes and return pointers to these. In most imperative languages, anonymous functions of the kind used in the above solution for functional languages can not be built, so holes must be an explicit part of the data-type that is used to represent abstract syntax. These holes will be overwritten when the values are supplied. E_* will, hence, return a record holding both an abstract syntax tree (in a field named `tree`) and a pointer to the hole that should be overwritten (in a field named `hole`). As actions (using C-style notation), this becomes

$$\begin{aligned} E &\rightarrow \mathbf{num} E_* \quad \{ \$2->\text{hole} = \text{NumExp} (\$1); \\ &\quad \quad \quad \$0 = \$2.\text{tree} \} \\ E_* &\rightarrow + \mathbf{num} E_* \quad \{ \$0.\text{hole} = \text{makeHole}(); \\ &\quad \quad \quad \$3->\text{hole} = \text{PlusExp} (\$0.\text{hole}, \text{NumExp} (\$2)); \\ &\quad \quad \quad \$0.\text{tree} = \$3.\text{tree} \} \\ E_* &\rightarrow \quad \quad \quad \{ \$0.\text{hole} = \text{makeHole}(); \\ &\quad \quad \quad \$0.\text{tree} = \$0.\text{hole} \} \end{aligned}$$

where `makeHole()` creates a node that can be overwritten. This may look bad, but left-recursion removal is rarely needed when using LR-parser generators.

An alternative approach is to let the parser build an intermediate (semi-abstract) syntax tree from the transformed grammar, and then let a separate pass restructure the intermediate syntax tree to produce the intended abstract syntax. Some LL(1) parser generators can remove left-recursion automatically and will afterwards restructure the syntax tree so it fits the original grammar.

2.17 Properties of Context-Free Languages

In Sect. 1.9, we described some properties of regular languages. Context-free languages share some, but not all, of these.

For regular languages, deterministic (finite) automata cover exactly the same class of languages as nondeterministic automata. This is not the case for context-free languages: Nondeterministic stack automata do indeed cover all context-free languages, but deterministic stack automata cover only a strict subset. The subset of context-free languages that can be recognised by deterministic stack automata are called deterministic context-free languages. Deterministic context-free languages can be recognised by LR parsers (but not necessarily by SLR parsers).

We have noted that the basic limitation of regular languages is finiteness: A finite automaton can not count unboundedly, and hence can not keep track of matching parentheses or similar properties that require counting. Context-free languages are capable of such counting, essentially using the stack for this purpose. Even so, there are limitations: A context-free language can only keep count of one thing at a time, so while it is possible (even trivial) to describe the language $\{a^n b^n \mid n \geq 0\}$ by a context-free grammar, the language $\{a^n b^n c^n \mid n \geq 0\}$ is not a context-free language. The information kept on the stack follows a strict LIFO order, which further restricts the languages that can be described. It is, for example, trivial to represent the language of palindromes (strings that read the same forwards and backwards) by a context-free grammar, but the language of strings that can be constructed by concatenating a string with itself is not context-free.

Context-free languages are, as regular languages, closed under union: It is easy to construct a grammar for the union of two languages given grammars for each of these. Context-free languages are also closed under prefix, suffix, subsequence and reversal. Indeed, a language that consists of all subsequences of a context-free language is actually a regular language. However, context-free languages are *not* closed under intersection or complement. For example, the languages $\{a^n b^n c^m \mid m, n \geq 0\}$ and $\{a^m b^n c^n \mid m, n \geq 0\}$ are both context-free while their intersection $\{a^n b^n c^n \mid n \geq 0\}$ is not, and the complement of the language described by the grammar in Sect. 2.10 is not a context-free language.

2.18 Further Reading

Context-free grammars were first proposed as a notation for describing natural languages (e.g., English or French) by the linguist Noam Chomsky [4], who defined this as one of three grammar notations for this purpose. The qualifier “context-free” distinguishes this notation from the other two grammar notations, which were called “context-sensitive” and “unconstrained”. In context-free grammars, derivation of a nonterminal is independent of the context in which the terminal occurs, whereas the context can restrict the set of derivations in a context-sensitive grammar. Unrestricted grammars can use the full power of a universal (Turing-complete) computer, so unrestricted grammars can represent all languages with decidable membership.

Context-free grammars are too weak to describe natural languages, but were adopted for defining the Algol 60 programming language [3], using a notation which is now called Backus-Naur form. Since then, variants of context-free grammars have been used for defining or describing almost all programming languages.

Some languages have been designed with specific parsing methods in mind: Pascal [6] was designed for LL(1) parsing while C [7] was originally designed to fit LALR(1) parsing. This property was lost in later versions of the language, which have more complex grammars.

Most parser generators are based on LALR(1) parsing, but some use LL parsing. An example of this is ANTLR [8].

“The Dragon Book” [2] tells more about parsing methods than the present book.

Several textbooks, e.g., [5] describe properties of context-free languages.

The methods presented here for rewriting grammars based on operator precedence uses only infix operators. If prefix or postfix operators have higher precedence than all infix operators, the method presented here will work (with trivial modifications), but if there are infix operators that have higher precedence than some prefix or postfix operators, it breaks down. A method for rewriting grammars with arbitrary precedences of infix, prefix and postfix operators to unambiguous form is presented in [1], along with a proof of correctness of the transformation.

2.19 Exercises

Exercise 2.1 Figures 2.7 and 2.8 show two different syntax trees for the string `aabbbcc` using Grammar 2.4. Draw a third, different syntax tree for `aabbbcc` using the same grammar, and show the left-derivation that corresponds to this syntax tree.

Exercise 2.2 Draw the syntax tree for the string `aabbbcc` using Grammar 2.9.

Exercise 2.3 Write an unambiguous grammar for the language of balanced parentheses, i.e., the language that contains (among other) the equences

ε (i.e., the empty string)

- ()
- (())
- () ()
- (() (()))

but *no* unbalanced sequences such as

- (
-)
-) (
- ((
- () ())

Exercise 2.4 Write grammars for each of the following languages:

- a) All sequences of *as* and *bs* that contain the same number of *as* and *bs* (in any order).
- b) All sequences of *as* and *bs* that contain strictly more *as* than *bs*.
- c) All sequences of *as* and *bs* that contain a different number of *as* and *bs*.
- d) All sequences of *as* and *bs* that contain twice as many *a*s as *b*s.

Exercise 2.5 We extend the language of balanced parentheses from Exercise 2.3 with two symbols: *[* and *]*. *[* corresponds to exactly two normal opening parentheses and *]* corresponds to exactly two normal closing parentheses. A string of mixed parentheses is legal if and only if the string produced by replacing *[* by *((* and *]* by *)*) is a balanced parentheses sequence. Examples of legal strings are

- ε
- () ()
- (())
- []
- () ()
- [()]

- a) Write a grammar that recognises this language.
- b) Draw the syntax trees for *[] ()* and *[()]*.

Exercise 2.6 Show that the grammar

$$\begin{aligned}
 A &\rightarrow -A \\
 A &\rightarrow A - \mathbf{id} \\
 A &\rightarrow \mathbf{id}
 \end{aligned}$$

is ambiguous by finding a string that has two different syntax trees.

Now make two different unambiguous grammars for the same language:

- a) One where prefix minus binds stronger than infix minus.
- b) One where infix minus binds stronger than prefix minus.

Show, using the new grammars, syntax trees for the string you used to prove the original grammar ambiguous. Show also fully reduced syntax trees.

Exercise 2.7 In Grammar 2.2, replace the operators $-$ and $/$ by $<$ and $:$. These have the following precedence rules:

- $<$ is non-associative and binds less tightly than $+$ but more tightly than $:$.
- $:$ is right-associative and binds less tightly than any other operator.

Write an unambiguous grammar for this modified grammar using the method shown in Sect. 2.3.1. Show the syntax tree and the fully reduced syntax tree for $2 : 3 < 4 + 5 : 6 * 7$ using the unambiguous grammar.

Exercise 2.8 Extend Grammar 2.14 with the productions

$$\begin{aligned} Exp &\rightarrow \mathbf{id} \\ Matched &\rightarrow \end{aligned}$$

then calculate *Nullable* and *FIRST* for every production in the grammar. Add an extra start production as described in Sect. 2.9 and calculate *FOLLOW* for every nonterminal in the grammar.

Exercise 2.9 Calculate *Nullable*, *FIRST* and *FOLLOW* for the nonterminals A and B in the grammar

$$\begin{aligned} A &\rightarrow BAa \\ A &\rightarrow \\ B &\rightarrow bBc \\ B &\rightarrow AA \end{aligned}$$

Remember to extend the grammar with an extra start production when calculating *FOLLOW*.

Exercise 2.10 Eliminate left-recursion from Grammar 2.2.

Exercise 2.11 Calculate *Nullable* and *FIRST* for every production in Grammar 2.23.

Exercise 2.12 Add a new start production $Exp' \rightarrow Exp \$$ to the grammar produced in Exercise 2.10 and calculate *FOLLOW* for all nonterminals in the resulting grammar.

Exercise 2.13 Make a LL(1) parser-table for the grammar produced in Exercise 2.12.

Exercise 2.14 Consider the following grammar for postfix expressions:

$$\begin{aligned} E &\rightarrow E E + \\ E &\rightarrow E E * \\ E &\rightarrow \mathbf{num} \end{aligned}$$

- a) Eliminate left-recursion in the grammar.
- b) Do left-factorisation of the grammar produced in question a.
- c) Calculate *Nullable*, *FIRST* for every production and *FOLLOW* for every nonterminal in the grammar produced in question b.
- d) Make a LL(1) parse-table for the grammar produced in question b.

Exercise 2.15 Extend Grammar 2.12 with a new start production as shown in Sect. 2.14 and calculate *FOLLOW* for every nonterminal. Remember to add an extra start production for the purpose of calculating *FOLLOW* as described in Sect. 2.9.

Exercise 2.16 Make NFAs (as in Fig. 2.30) for the productions in Grammar 2.12 (after extending it as shown in Sect. 2.14) and add epsilon-transitions as in Fig. 2.31. Convert the combined NFA into an SLR DFA like the one in Fig. 2.33. Finally, add reduce and accept actions based on the *FOLLOW* sets calculated in Exercise 2.15.

Exercise 2.17 Extend Grammar 2.2 with a new start production as shown in Sect. 2.14 and calculate *FOLLOW* for every nonterminal. Remember to add an extra start production for the purpose of calculating *FOLLOW* as described in Sect. 2.9.

Exercise 2.18 Make NFAs (as in Fig. 2.30) for the productions in Grammar 2.2 (after extending it as shown in Sect. 2.14) and add epsilon-transitions as in Fig. 2.31. Convert the combined NFA into an SLR DFA like the one in Fig. 2.33. Add reduce actions based on the *FOLLOW* sets calculated in Exercise 2.17. Eliminate the conflicts in the table by using operator precedence rules as described in Sect. 2.15. Compare the size of the table to that from Exercise 2.16.

Exercise 2.19 Consider the grammar

$$\begin{aligned} T &\rightarrow T -> T \\ T &\rightarrow T * T \\ T &\rightarrow \mathbf{int} \end{aligned}$$

where $->$ is considered a single terminal symbol.

- a) Add a new start production as shown in Sect. 2.14.
- b) Calculate *FOLLOW*(*T*). Remember to add an extra start production.
- c) Construct an SLR parser-table for the grammar.
- d) Eliminate conflicts using the following precedence rules:
 - $*$ binds tighter than $->$.
 - $*$ is left-associative.
 - $->$ is right-associative.

Exercise 2.20 In Sect. 2.16.1 it is mentioned that user-defined operator precedences in programming languages can be handled by parsing all operators with a single fixed precedence and associativity and then using a separate pass to restructure the syntax tree to reflect the declared precedences. Below are two other methods that have been used for this purpose:

- a) An ambiguous grammar is used and conflicts exist in the SLR table. Whenever a conflict arises during parsing, the parser consults a table of precedences to resolve this conflict. The precedence table is extended whenever a precedence declaration is read.
- b) A terminal symbol is made for every possible precedence and associativity combination. A conflict-free parse table is made either by writing an unambiguous grammar or by eliminating conflicts in the usual way. The lexical analyser uses a table of precedences to assign the correct terminal symbol to each operator it reads.

Compare all three methods. What are the advantages and disadvantages of each method?

Exercise 2.21 Consider the grammar

$$\begin{aligned} A &\rightarrow a A a \\ A &\rightarrow b A b \\ A &\rightarrow \end{aligned}$$

- a) Describe the language that the grammar defines.
- b) Is the grammar ambiguous? Justify your answer.
- c) Construct an SLR parse table for the grammar.
- d) Can the conflicts in the table be eliminated without changing the language? Justify your answer.

Exercise 2.22 The following ambiguous grammar describes Boolean expressions:

$$\begin{aligned} B &\rightarrow \mathbf{true} \\ B &\rightarrow \mathbf{false} \\ B &\rightarrow B \vee B \\ B &\rightarrow B \wedge B \\ B &\rightarrow \neg B \\ B &\rightarrow (B) \end{aligned}$$

- a) Given that negation (\neg) binds tighter than conjunction (\wedge) which binds tighter than disjunction (\vee) and that conjunction and disjunction are both right-associative, rewrite the grammar to be unambiguous.
- b) Write a grammar that accepts the subset of Boolean expressions that are equivalent to **true** (i.e., tautologies). Hint: Modify the answer from question a) and add an additional nonterminal F for false Boolean expressions.

References

1. Aasa, A.: Precedences in specifications and implementations of programming languages. *Theor. Comput. Sci.* **142**(1), 3–26 (1995). [http://dx.doi.org/10.1016/0304-3975\(95\)90680-J](http://dx.doi.org/10.1016/0304-3975(95)90680-J), <http://www.sciencedirect.com/science/article/pii/030439759590680J>

2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers; Principles. Addison-Wesley, Techniques and Tools* (2007)
3. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vaquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language. *Algol* **60** (1963)
4. Chomsky, N.: Three models for the description of language. *IRE Trans. Inf. Theory* **IT-2**(3), 113–124 (1956)
5. Hopcroft, J.E., Motwani, R., Ullman, J.D.: *Introduction to Automata Theory, Languages and Computation*, 2nd edn. Addison-Wesley (2001)
6. Jensen, K., Wirth, N.: *Pascal User Manual and Report*, 2nd edn. Springer-Verlag (1975)
7. Kernighan, B.W., Ritchie, D.M.: *The C Programming Language*. Prentice-Hall (1978)
8. Parr, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages*, 1st edn. Pragmatic Programmers, Pragmatic Bookshelf (2007)

Chapter 3

Scopes and Symbol Tables



The scope of thrift is limitless.
Thomas Edison (1847–1931)

An important concept in programming languages is the ability to *name* items such as variables, functions and types. Each such named item will have a *declaration*, where the name is defined as a synonym for the item and where properties of the item are described. This is called *binding*. Each name will also have a number of *uses*, where the name is used as a reference to the item to which it is bound.

Often, the declaration of a name has a limited *scope*: a portion of the program where the name will be visible. Such declarations are called *local declarations*, whereas a declaration that makes the declared name visible in the entire program is called *global*. It may happen that the same name is declared in several nested scopes. In this case, it is normal that the declaration closest to a use of the name will be the one that defines that particular use. In this context *closest* is related to the syntax tree of the program: The scope of a declaration will be a sub-tree of the syntax tree, and nested declarations will give rise to scopes that are nested sub-trees. The closest declaration of a name is, hence, the declaration corresponding to the smallest sub-tree that encloses the use of the name. As an example, look at this C statement block:

```
{
  int x = 1;
  int y = 2;
  {
    double x = 3.14159265358979;
    y += (int)x;
  }
  y += x;
}
```


The two lines immediately after the first opening brace declare integer variables x and y with scope until the closing brace in the last line. A new scope is started by the second opening brace, and a floating-point variable x with an initial value close to π is declared. This will have scope until the first closing brace, so the original x variable is not visible until the inner scope ends. The assignment $y += (\text{int})x;$ will add 3 to y , so its new value is 5. In the next assignment $y += x;$, we have exited the inner scope, so the original x is restored. The assignment will, hence, add 1 to y , which will have the final value 6.

Scoping based on the structure of the syntax tree, as shown in the example, is called *static* or *lexical* binding and is the most common scoping rule in modern programming languages. We will in the rest of this chapter (indeed, the rest of this book) assume that static binding is used. A few languages have *dynamic* binding, where the declaration that was most recently encountered during execution of the program defines the current use of the name. By its nature, dynamic binding can not be resolved at compile-time, so the techniques that in the rest of this chapter are described as being used in a compiler will have to be used at run-time if the language uses dynamic binding.

A compiler will need to keep track of names and the items these are bound to, so that any use of a name will be attributed correctly to its declaration. This is typically done using a *symbol table* (or *environment*, as it is sometimes called).

3.1 Symbol Tables

A symbol table is a table that binds names to properties of the items the names are bound to. We need a number of operations on symbol tables to accomplish this:

- We need an *empty* symbol table, in which no name is defined.
- We need to be able to *bind* a name to the properties of an item. In case the name is already defined in the symbol table, the new binding takes precedence over the old.
- We need to be able to *look up* a name in a symbol table to find the properties of the item to which the name is bound. If the name is not defined in the symbol table, we need to be told that.
- We need to be able to *enter* a new scope.
- We need to be able to *exit* a scope, reestablishing the symbol table to what it was before the scope was entered.

3.1.1 Implementation of Symbol Tables

There are many ways to implement symbol tables, but the most important distinction between these is how scopes are handled. This may be done using a *persistent* (or

functional) data structure, or it may be done using an *imperative* (or destructively-updated) data structure.

A persistent data structure has the property that no operation on the structure will destroy it. Conceptually, a new modified copy is made of the data structure whenever an operation updates it, hence preserving the old structure unchanged. This means that it is trivial to reestablish the old symbol table when exiting a scope, as it has been preserved by the persistent nature of the data structure. In practice, only a small portion of the data structure representing a persistent symbol table is copied when a modified symbol table is created, most of the structure is shared with the previous version.

In the imperative approach, only one copy of the symbol table exists, so explicit actions are required to store the information needed to restore the symbol table to a previous state. This can be done by using an auxiliary stack. When an update is made, the old binding of a name that is overwritten is recorded (pushed) on the auxiliary stack. When a new scope is entered, a marker is pushed on the auxiliary stack. When the scope is exited, the bindings on the auxiliary stack (down to the marker) are used to reestablish the old symbol table. The bindings and the marker are popped off the auxiliary stack in the process, returning the auxiliary stack to the state it was in before the scope was entered.

Below, we will look at simple implementations of both approaches and discuss more advanced approaches that are more efficient than the simple approaches.

3.1.2 *Simple Persistent Symbol Tables*

In functional languages like Standard ML, F#, Scheme or Haskell, persistent data structures are the norm rather than the exception (which is why persistent data structures are sometimes called *functional* data structures). For example, when a new element is added to the front of a list or an element is taken off the front of the list, the old list still exists and can be used elsewhere. A list is a natural way to implement a symbol table in a functional language: A binding is a pair of a name and its associated properties, and a symbol table is a list of such pairs. The operations are implemented in the following way:

- empty: An empty symbol table is an empty list.
- bind: A new binding (name/properties pair) is added (consed) to the front of the list.
- lookup: The list is searched until a pair with a matching name is found. The properties paired with the name is then returned. If the end of the list is reached, an indication that this happened is returned instead. This indication can be made by raising an exception or by letting the lookup function return a special value representing “not found”. This requires a type that can hold both normal property information and this special value, i.e., an option type (Standard ML, F#) or a Maybe type (Haskell).

enter: The old list is remembered, i.e., a reference is made to it.
 exit: The old list is recalled, i.e., the above reference is used.

The latter two operations are not really explicit operations, as the variable used to hold the symbol table before entering a new scope will still hold the same symbol table after the scope is exited. So all that is needed is a variable to hold (a reference to) the symbol table.

As new bindings are added to the front of the list and the list is searched from the front to the back, bindings in inner scopes will automatically take precedence over bindings in outer scopes.

Another functional approach to symbol tables is using functions: A symbol table is quite naturally seen as a function from names to information. The operations are:

empty: An empty symbol table is a function that returns an error indication (or raises an exception) no matter what its argument is.
 bind: Adding a binding of the name n to the properties p in a symbol table t is done by defining a new symbol-table function t' in terms t and the new binding. When t' is called with a name $n1$ as argument, it compares $n1$ to n . If they are equal, t' returns the properties p . Otherwise, t' calls t with $n1$ as argument and returns the result that this call yields. In Standard ML, we can define a binding function this way:

```
fun bind (n,p,t)
  = fn n1 => if n1=n then p else t n1
```

lookup: The symbol-table function is called with the name as argument.
 enter: The old function is remembered (referenced).
 exit: The old function is recalled (by using a reference).

Again, the latter two operations are mostly implicit.

3.1.3 A Simple Imperative Symbol Table

Imperative symbol tables are natural to use if the compiler is written in an imperative language. A simple imperative symbol table can be implemented as a stack, which works in a way similar to the list-based functional implementation:

empty: An empty symbol table is an empty stack.
 bind: A new binding (name/properties pair) is pushed on top of the stack.
 lookup: The stack is searched top-to-bottom until a matching name is found. The properties paired with the name is then returned. If the bottom of the stack is reached, we instead return an error-indication.
 enter: We push a marker on the top of the stack.

exit: We pop bindings from the stack until a marker is found. This is also popped from the stack.

Note that since the symbol table is itself a stack, we don't need the auxiliary stack mentioned in Sect. 3.1.1.

This is not quite a persistent data structure, as leaving a scope will destroy its symbol table. For simple languages, this won't matter, as a scope isn't needed again after it is exited. But language features such as classes, modules and lexical closures can require symbol tables to persist after their scope is exited. In these cases, a real persistent symbol table must be used, or the needed parts of the symbol table must be copied and stored for later retrieval before exiting a scope.

3.1.4 *Efficiency Issues*

While all of the above implementations are simple, they all share the same efficiency problem: Lookup is done by linear search, so the worst-case time for lookup is proportional to the size of the symbol table. This is mostly a problem in relation to libraries: It is quite common for a program to use libraries that define literally hundreds of names.

A common solution to this problem is *hashing*: Names are hashed (processed) into integers, which are used to index an array. Each array element is then a linear list of the bindings of names that share the same hash code. Given a large enough hash table, these lists will typically be very short, so lookup time is basically constant.

Using hash tables complicates entering and exiting scopes somewhat. While each element of the hash table is a list that can be handled like in the simple cases, doing this for *all* the array-elements at every entry and exit imposes a major overhead. Instead, it is typical for imperative implementations to use a single auxiliary stack (as described in Sect. 3.1.1) to record all updates to the table so they can be undone in time proportional to the number of updates that were done in the local scope. Functional implementations typically use persistent hash-tables or persistent binary search trees, which eliminates the problem.

3.1.5 *Shared or Separate Name Spaces*

In some languages (like Pascal) a variable and a function in the same scope may have the same name, as the context of use will make it clear whether a variable or a function is used. We say that functions and variables have *separate name spaces*, which means that defining a name in one space doesn't affect the same name in the other space, even if the names occur in the same scope. In other languages (e.g. C, F#, or Standard ML) the context can not (easily) distinguish variables from functions. Hence, declaring a local variable will hide a function with the same name declared in

an outer scope or vice versa. These languages have a *shared name space* for variables and functions.

Name spaces may be shared or separate for all the kinds of names that can appear in a program, e.g., variables, functions, types, exceptions, constructors, classes, field selectors, etc. Which name spaces are shared is language-dependent.

Separate name spaces are easily implemented using one symbol table per name space, whereas shared name spaces naturally share a single symbol table. However, it is sometimes convenient to use a single symbol table even if there are separate name spaces. This can be done fairly easily by adding name-space indicators to the names. A name-space indicator can be a textual prefix to the name or it may be a tag that is paired with the name. In either case, a lookup in the symbol table must match both name and name-space indicator of the symbol that is looked up with both name and name-space indicator of the entry in the table.

Suggested exercises: 3.1.

3.2 Further Reading

Most algorithms-and-data-structures textbooks include descriptions of methods for hashing strings and implementing hash tables. A description of efficient persistent data structures for functional languages can be found in [1].

3.3 Exercises

Exercise 3.1 Pick some programming language that you know well and determine which of the following items share name spaces: Variables, functions, procedures and types. If there are more kinds of named items (labels, data constructors, modules, etc.) in the language, include these in the investigation.

Exercise 3.2 Implement, in a programming language of your choice, data structures and operations (empty, bind, lookup, enter and exit) for simple symbol tables.

Exercise 3.3 In some programming languages, identifiers are case-insensitive so, e.g., `size` and `SIZE` refer to the same identifier. Describe how symbol tables can be made case-insensitive.

Reference

1. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)

Chapter 4

Interpretation



Any good software engineer will tell you that a compiler and an interpreter are interchangeable.

Tim Berners-Lee (1955–)

After lexing and parsing, we have the abstract syntax tree of a program as a data structure in memory. But a program needs to be executed, and we have not yet dealt with that issue.

The simplest way to execute a program is *interpretation*. Interpretation is done by a program called an *interpreter*, which takes the abstract syntax tree of a program and executes it by inspecting the syntax tree to see what needs to be done. This is similar to how a human evaluates a mathematical expression: We insert the values of variables in the expression and evaluate it bit by bit, starting with the innermost parentheses and moving out until we have the result of the expression. We can then repeat the process with other values for the variables.

There are some differences, however. Where a human being will copy the text of the formula with variables replaced by values, and then write a sequence of more and more reduced copies of a formula until it is reduced to a single value, an interpreter will keep the formula (or, rather, the abstract syntax tree of an expression) unchanged and use a symbol table to keep track of the values of variables. Instead of reducing a formula, the interpreter is a function that takes an abstract syntax tree and a symbol table as arguments and returns the value of the expression represented by the abstract syntax tree. The function can call itself recursively on parts of the abstract syntax tree to find the values of subexpressions, and when it evaluates a variable, it can look its value up in the symbol table.

This process can be extended to also handle statements and declarations, but the basic idea is the same: A function takes the abstract syntax tree of the program and, possibly, some extra information about the context (such as a symbol table or the input to the program) and returns the output of the program. Some input and output may be done as side effects by the interpreter.

We will in this chapter assume that the symbol tables are persistent, so no explicit action is required to restore the symbol table for the outer scope when exiting an inner scope. In the main text of the chapter, we don't need to preserve symbol tables for inner scopes once these are exited (so a stack-like behaviour is fine), but in one of the exercises we will need symbol tables to persist after their scope is exited.

4.1 The Structure of an Interpreter

An interpreter will typically consist of one function per syntactic category. Each function will take as arguments an abstract syntax tree from the syntactic category and, possibly, extra arguments such as symbol tables. Each function will return one or more results, which can be the value of an expression, an updated symbol table, or nothing at all (relying on side effects to, e.g., symbol tables).

These functions can be implemented in any programming language for which we already have an implementation. This implementation can also be an interpreter, or it can be a compiler that compiles to some other language. Eventually, we will need to either have an interpreter written in machine language or a compiler that compiles to machine language. For the moment, we just write interpretation functions in a notation reminiscent of a high-level programming language and assume an implementation of this exists. Additionally, we want to avoid being specific about how abstract syntax is represented, so we will use a notation that looks like concrete syntax to represent abstract syntax.

4.2 A Small Example Language

We will use a small (somewhat contrived) language to show the principles of interpretation. The language is a first-order functional language with recursive definitions. The syntax is given in Grammar 4.1. The shown grammar is clearly ambiguous, but we assume that any ambiguities have been resolved during parsing, so we have an unambiguous abstract syntax tree.

In the example language, a program is a list of function declarations. The functions are all mutually recursive, and no function may be declared more than once. Each function declares its result type and the types and names of its arguments. Types are `int` (integer) and `bool` (Boolean). There may not be repetitions in the list of parameters for a function. Functions and variables have separate name spaces. The body of a function is an expression, which may be an integer constant, a variable name, a sum-expression, a comparison, a conditional, a function call or an expression with a local declaration. Comparison is defined both on booleans (where *false* is considered smaller than *true*) and integers, but addition is defined only on integers.

Grammar 4.1 Example language for interpretation

$$\begin{aligned}
 \text{Program} &\rightarrow \text{Funs} \\
 \text{Funs} &\rightarrow \text{Fun} \\
 \text{Funs} &\rightarrow \text{Fun Funs} \\
 \text{Fun} &\rightarrow \text{TypeId} (\text{TypeIds}) = \text{Exp} \\
 \text{TypeId} &\rightarrow \text{int } \mathbf{id} \\
 \text{TypeId} &\rightarrow \text{bool } \mathbf{id} \\
 \text{TypeIds} &\rightarrow \text{TypeId} \\
 \text{TypeIds} &\rightarrow \text{TypeId} , \text{TypeIds} \\
 \text{Exp} &\rightarrow \mathbf{num} \\
 \text{Exp} &\rightarrow \mathbf{id} \\
 \text{Exp} &\rightarrow \text{Exp} + \text{Exp} \\
 \text{Exp} &\rightarrow \text{Exp} < \text{Exp} \\
 \text{Exp} &\rightarrow \text{if } \text{Exp} \text{ then } \text{Exp} \text{ else } \text{Exp} \\
 \text{Exp} &\rightarrow \mathbf{id} (\text{Exps}) \\
 \text{Exp} &\rightarrow \text{let } \mathbf{id} = \text{Exp} \text{ in } \text{Exp} \\
 \text{Exps} &\rightarrow \text{Exp} \\
 \text{Exps} &\rightarrow \text{Exp} , \text{Exps}
 \end{aligned}$$

A program must contain a function called `main`, which has one integer argument and which returns an integer. Execution of the program is by calling this function with the input (which must be an integer). The result of this function call (also an integer) is the output of the program.

4.3 An Interpreter for the Example Language

An interpreter for this language must take the abstract syntax tree of the program and an integer (the input to the program) and return another integer (the output of the program). Since values can be both integers or booleans, the interpreter uses a value type that contains both integers and booleans (and enough information to tell them apart). We will not go into detail about how such a type can be defined but simply assume that there are operations for testing if a value is a boolean or an integer and operating on values known to be integers or booleans. If we during interpretation find that we are about to, say, add a boolean to an integer, we stop the interpretation with an error message. We do this by letting the interpreter call a function called `error()`.

We will start by showing how we can evaluate (interpret) expressions, and then extend this to handle the whole program.

4.3.1 Evaluating Expressions

When we evaluate expressions, we need, in addition to the abstract syntax tree of the expression, also a symbol table *vtable* that binds variables to their values. Additionally, we need to be able to handle function calls, so we also need a symbol table *ftable* that binds function names to the abstract syntax trees of their declarations. The result of evaluating an expression is the value of the expression.

For terminals (variable names and numeric constants) with attributes, we assume that there are predefined functions for extracting these attributes. Hence, **id** has an associated function *getname*, that extracts the name of the identifier. Similarly, **num** has a function *getvalue*, that returns the value of the number.

Figure 4.2 shows a function $Eval_{Exp}$, that takes an expression *Exp* and symbol tables *vtable* and *ftable*, and returns a value, which may be either an integer or a boolean. Also shown is a function $Eval_{Exps}$, that evaluates a list of expressions to a list of values. We also use a function $Call_{Fun}$ that handles function calls. We will define this later.

The main part of $Eval_{Exp}$ is a case-expression (in some languages called `switch` or `match`) that identifies which kind of expression the top node of the abstract syntax tree represents. The patterns are shown as concrete syntax, but you should think of it as pattern matching on the abstract syntax. The column to the right of the patterns shows the actions needed to evaluate the expressions. These actions can refer to parts of the pattern on the left. An action is a sequence of definitions of local variables followed by an expression (in the interpreter) that evaluates to the result of the expression that was given (in abstract syntax) as argument to $Eval_{Exp}$.

We will briefly explain each of the cases handled by $Eval_{Exp}$.

- The value of a number is found as the *value* attribute to the node in the abstract syntax tree.
- The value of a variable is found by looking its name up in the symbol table for variables. If the variable is not found in the symbol table, the lookup-function returns the special value *unbound*. When this happens, an error is reported and the interpretation stops. Otherwise, it returns the value returned by *lookup*.
- At a plus-expression, both arguments are evaluated, then it is checked that they are both integers. If they are, we return the sum of the two values. Otherwise, we report an error (and stop).
- Comparison requires that the arguments have the same type. If that is the case, we compare the values, otherwise we report an error.
- In a conditional expression, the condition must be a boolean. If it is, we check if it is **true**. If so, we evaluate the then-branch, otherwise, we evaluate the else-branch. If the condition is not a boolean, we report an error.
- At a function call, the function name is looked up in the function environment to find its definition. If the function is not found in the environment, we report an error. Otherwise, we evaluate the arguments by calling $Eval_{Exps}$ and then call $Call_{Fun}$ to find the result of the call.

Fig. 4.2 Evaluating expressions

$Eval_{Exp}(Exp, vtable, ftable) = \text{case } Exp \text{ of}$	
num	$getvalue(\mathbf{num})$
id	$v = lookup(vtable, getname(\mathbf{id}))$ if $v = unbound$ then error() else v
$Exp_1 + Exp_2$	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$ if v_1 and v_2 both are integers then $v_1 + v_2$ else error()
$Exp_1 < Exp_2$	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$ if v_1 and v_2 both are integers then if $v_1 < v_2$ then true else false else if v_1 and v_2 both are booleans then if $v_1 = \mathbf{false}$ then v_2 else false else error()
if Exp_1 then Exp_2 else Exp_3	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ if v_1 is a boolean then if $v_1 = \mathbf{true}$ then $Eval_{Exp}(Exp_2, vtable, ftable)$ else $Eval_{Exp}(Exp_3, vtable, ftable)$ else error()
id ($Exps$)	$def = lookup(ftable, getname(\mathbf{id}))$ if $def = unbound$ then error() else $args = Eval_{Exps}(Exps, vtable, ftable)$ $Call_{Fun}(def, args, ftable)$
let id = Exp_1 in Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\mathbf{id}), v_1)$ $Eval_{Exp}(Exp_2, vtable', ftable)$
$Eval_{Exps}(Exps, vtable, ftable) = \text{case } Exps \text{ of}$	
Exp	$[Eval_{Exp}(Exp, vtable, ftable)]$
$Exp, Exps$	$Eval_{Exp}(Exp, vtable, ftable)$ $:: Eval_{Exps}(Exps, vtable, ftable)$

- A `let`-expression declares a new variable with an initial value defined by an expression. The expression is evaluated and the symbol table for variables is extended using the function `bind` to bind the variable to the value. The extended

table is used when evaluating the body-expression, which defines the value of the whole expression. Note that we do not explicitly restore the symbol table after exiting the scope of the `let`-expression. The old symbol table is implicitly preserved.

$Eval_{Exps}$ builds a list of the values of the expressions in the expression list. The notation is taken from SML and F#: A list is written in square brackets, and the infix operator `::` adds an element to the front of a list. This operator can also be used in patterns to match non-empty lists and bind variables to the head and tail of this list.

Suggested exercises: 4.1.

4.3.2 Interpreting Function Calls

A function declaration explicitly declares the types of the arguments. When a function is called, we must check that the number of arguments is the same as the declared number, and that the values of the arguments match the declared types.

If this is the case, we build a symbol table that binds the parameter variables to the values of the arguments and use this in evaluating the body of the function. The value of the body must match the declared result type of the function.

$Call_{Fun}$ is also given a symbol table for functions, which is passed to the $Eval_{Exp}$ when evaluating the body.

$Call_{Fun}$ is shown in Fig. 4.3, along with the functions for $TypeId$ and $TypeIds$, which it uses. The function $GetTypeId$ just returns a pair of the declared name and type, and $BindTypeIds$ checks the declared type against a value and, if these match, builds a symbol table that binds the name to the value (and reports an error if they do not match). $BindsTypeIds$ also checks if all parameters have different names by seeing if the current name is already bound. $emptytable$ is an empty symbol table. Looking any name up in the empty symbol table returns *unbound*. The underscore used in the last rule for $BindTypeIds$ is a wildcard pattern that matches anything, so this rule is used when the number of arguments do not match the number of declared parameters.

4.3.3 Interpreting a Program

Running a program is done by calling the `main` function with a single argument that is the input to the program. So we build the symbol table for functions, look up `main` in this and call $Call_{Fun}$ with the resulting definition and an argument list containing just the input.

Hence, $Run_{Program}$, which runs the whole program, calls a function $Build_{f\text{table}}$ that builds the symbol table for functions. This, in turn, uses a function Get_{fname} that finds the name of a function. All these functions are shown in Fig. 4.4.

This completes the interpreter for our small example language.

$$\begin{array}{l}
 \text{Call}_{Fun}(Fun, args, ftable) = \text{case } Fun \text{ of} \\
 \hline
 \text{TypeId} (TypeIds) = \text{Exp} \left| \begin{array}{l}
 (f, t_0) = \text{Get}_{TypeId}(\text{TypeId}) \\
 vtable = \text{Bind}_{TypeIds}(\text{TypeIds}, args) \\
 v_1 = \text{Eval}_{Exp}(\text{Exp}, vtable, ftable) \\
 \text{if } v_1 \text{ is of type } t_0 \\
 \text{then } v_1
 \end{array} \right. \\
 \\
 \text{Get}_{TypeId}(\text{TypeId}) = \text{case } \text{TypeId} \text{ of} \\
 \hline
 \text{int } \mathbf{id} \quad | \quad (\text{getname}(\mathbf{id}), \text{int}) \\
 \hline
 \text{bool } \mathbf{id} \quad | \quad (\text{getname}(\mathbf{id}), \text{bool}) \\
 \\
 \text{Bind}_{TypeIds}(\text{TypeIds}, args) = \text{case } (\text{TypeIds}, args) \text{ of} \\
 \hline
 (\text{TypeId}, \quad | \quad (x, t) = \text{Get}_{TypeId}(\text{TypeId}) \\
 [v]) \quad | \quad \text{if } v \text{ is of type } t \\
 \quad | \quad \text{then } \text{bind}(\text{emptytable}, x, v) \\
 \quad | \quad \text{else } \mathbf{error}() \\
 \hline
 (\text{TypeId} , \text{TypeIds}, \quad | \quad (x, t) = \text{Get}_{TypeId}(\text{TypeId}) \\
 (v :: vs)) \quad | \quad vtable = \text{Bind}_{TypeIds}(\text{TypeIds}, vs) \\
 \quad | \quad \text{if } \text{lookup}(vtable, x) = \text{unbound} \text{ and } v \text{ is of type } t \\
 \quad | \quad \text{then } \text{bind}(vtable, x, v) \\
 \quad | \quad \text{else } \mathbf{error}() \\
 \hline
 \text{---} \quad | \quad \mathbf{error}()
 \end{array}$$

Fig. 4.3 Evaluating a function call

While we have illustrated interpretation mainly by a single example, the methods carry over to other languages: We build one or more function for each syntactic category. These may, in addition to the abstract syntax tree, also take other parameters such as symbol tables, and they return values that are used in other parts of the interpreter (or represent part of the output).

Suggested exercises: 4.5.

4.4 Advantages and Disadvantages of Interpretation

Once you have a abstract syntax tree, interpretation is often the simplest way of executing a program. However, it is also a relatively slow way to do so. When we perform an operation in the interpreted program, the interpreter must first inspect the abstract syntax tree to see what operation it needs to perform, then it must check that the types of the arguments to the operation match the requirements of the operation, and only then can it perform the operation. Additionally, each value must include

Fig. 4.4 Interpreting a program

$$\frac{}{Run_{Program}(Program, input) = \text{case } Program \text{ of}}$$

<i>Funs</i>	$f_{table} = Build_{f_{table}}(Funs)$ $def = lookup(f_{table}, main)$ if $def = unbound$ then error () else $Call_{Fun}(def, [input], f_{table})$
-------------	---

$$\frac{}{Build_{f_{table}}(Funs) = \text{case } Funs \text{ of}}$$

<i>Fun</i>	$f = Get_{f_{name}}(Fun)$ $bind(empty_{table}, f, Fun)$
<i>Fun Funs</i>	$f = Get_{f_{name}}(Fun)$ $f_{table} = Build_{f_{table}}(Funs)$ if $lookup(f_{table}, f) = unbound$ then $bind(f_{table}, f, Fun)$ else error ()

$$\frac{}{Get_{f_{name}}(Fun) = \text{case } Fun \text{ of}}$$

<i>TypeId (TypeIds)</i>	$Exp \mid (f, t_0) = Get_{TypeId}(TypeId)$ f
---------------------------	---

sufficient information to identify its type, so after doing, say, an addition, we must add type information to the resulting number.

It should be clear from this that we spend much more time on figuring out what to do, and whether doing it is O.K., than on actually doing it.

To get faster execution, we can use the observation that a program that executes each part of the program only once will finish quite quickly. In other words, any time-consuming program will contain parts that are executed many times. The idea is that we can do the inspection of the abstract syntax tree and the type checking only once for each program part, and only repeat the actual operations that are performed in this part. Since performing the operations is a small fraction of the total time spent in an interpreter, we can get a substantial speedup by doing this. This is the basic idea behind *static type checking* and *compilation*.

Static type checking checks the program *before* it is executed for *potential* mismatches between the types of values and the requirements of operations. It does this check for the whole program regardless of whether all parts will actually be executed, so it may report errors even if an interpretation of the program would finish without errors. So static type checking puts extra limitations on programs, but reduces the time needed at runtime to check for errors and, as a bonus, reports potential problems before a program is executed, which can help when debugging a program. We look at static type checking in Chap. 5. Static type checking does, however, need some time to do the checking before we can start executing the program, so the time from doing an edit in a program to executing it will increase slightly.

Compilation gets rid of the abstract syntax tree of the source program by translating it into a target program (in a language of which we already have an implementation) that only performs the operations in the source program, having done (most of) the checks during compilation. Usually, the target language is a low-level language such as machine code, but it can also be another high-level language. Like static checking, compilation must (at least conceptually) complete before execution can begin, so it adds delay between editing a program and running it.

Usually, static checking and compilation go hand in hand, but you can find compilers for languages with dynamic (run-time) type checking as well as interpreters for statically typed languages.

Some implementations combine interpretation and compilation: The first few times a function is called, it is interpreted, but if it is called sufficiently often, it is compiled and all subsequent calls to the function will execute the compiled code. This is often called *just-in-time compilation*, though this term was originally used for just postponing compilation of a function until just before the first time it is called, hence reducing the delay from editing a program to its execution, but at the cost of adding small delays for compilation during execution. We will in this book only look at “pure” interpretation and compilation, though.

4.5 Further Reading

A step-by-step construction of an interpreter for a LISP-like language is shown in [2]. A survey of programming language constructs (also for LISP-like languages) and their interpretation is shown in [1].

4.6 Exercises

Exercise 4.1 We extend the language from Sect. 4.2 with Boolean operators. We add the following productions to Grammar 4.1:

$$\begin{aligned} \text{Exp} &\rightarrow \text{not Exp} \\ \text{Exp} &\rightarrow \text{Exp and Exp} \end{aligned}$$

When evaluating `not e`, we first evaluate e to a value v that is checked to be a boolean. If it is, we return $(\neg v)$, where \neg is logical negation.

When evaluating e_1 and e_2 , we first evaluate e_1 and e_2 to values v_1 and v_2 that are both checked to be booleans. If they are, we return $(v_1 \wedge v_2)$, where \wedge is logical conjunction.

Extend the interpretation function in Fig. 4.2 to handle these new constructions as described above.

Exercise 4.2 Add the productions

$$\begin{aligned} \text{Exp} &\rightarrow \text{floatconst} \\ \text{TypeId} &\rightarrow \text{float id} \end{aligned}$$

to Grammar 4.1. This introduces floating-point numbers to the language. The operator $+$ is overloaded so it can do integer addition or floating-point addition, and $<$ is extended so it can also compare pairs of floating point numbers.

- (a) Extend the interpretation functions in Figs. 4.2–4.4 to handle these extensions.
- (b) We now add implicit conversion of integers to floats to the language, using the rules: Whenever an operator has one integer argument and one floating-point argument, the integer is converted to a float. Extend the interpretation functions from question a) above to handle this.

Exercise 4.3 The language defined in Sect. 4.2 declares types of parameters and results of functions. The interpreter in Sect. 4.3 adds explicit type information to every value, and checks this before doing any operations on values. So, we could omit type declarations and rely solely on the type information in values.

Replace in Grammar 4.1 *TypeId* by **id** and rewrite the interpretation functions in Fig. 4.3 so they omit checking types of parameters and results, but still check that the number of arguments match the declaration and that no parameter name is repeated.

Exercise 4.4 In the language defined in Sect. 4.2, variables bound in `let`-expressions have no declared type, so it is possible to write a program where a `let`-bound variable in a `let`-expression sometimes is bound to an integer and at other times to a Boolean value.

Write an example of such a program.

Exercise 4.5 We extend the language from Sect. 4.2 with functional values. These require *lexical closures*, which is a record/struct/tuple that contains an argument name, an expression, and an environment. We assume symbol tables are fully persistent, so environments in closures persist. We add the following productions to Grammar 4.1:

$$\begin{aligned} \text{TypeId} &\rightarrow \text{fun id} \\ \text{Exp} &\rightarrow \text{Exp Exp} \\ \text{Exp} &\rightarrow \text{fn id} => \text{Exp} \end{aligned}$$

The notation for anonymous functions is taken from Standard ML. Evaluating `fn x=>e` in an environment *vtable* produces a functional value *f*, which is a lexical closure consisting of the name *x*, the expression *e*, and the environment *vtable*. When *f* is applied to an argument *v*, it is checked that *v* is an integer. If this is the case, *e* is evaluated in *vtable* extended with a binding that binds *x* to *v*. We then check if the

result w of this evaluation is an integer, and if so use it as the result of the function application.

When evaluating $e_1 e_2$, we evaluate e_1 to a functional value f and e_2 to an integer v , and then apply f to v as described above.

Extend the interpreter from Fig. 4.3 to handle these new constructions as described above.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. MIT Press (1996). Also downloadable from <https://mitpress.mit.edu/sicp/full-text/sicp/book/>
2. Steele, G.L., Sussman, G.J.: The Art of the Interpreter or, The Modularity Complex. Tech. Rep. AIM-453, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)

Chapter 5

Type Checking



The most touching epitaph I ever encountered was on the tombstone of the printer of Edinburgh. It said simply: He kept down the cost and set the type right.

Gregory Nunn (1955–)

Lexing and parsing will reject many texts as not being correct programs. However, many languages have well-formedness requirements that can not be handled exclusively by the techniques seen so far. These requirements can, for example, be static type correctness or a requirement that pattern-matching or case-statements are exhaustive.

These properties are most often not context-free, i.e., they can not be checked by membership of a context-free language. Consequently, they are checked by a phase that conceptually comes after syntax analysis (though it may be interleaved with it). These checks may happen in a phase that does nothing else, or they may be combined with the actual execution or translation to another language. Often, translation to another language (such as machine language) may exploit or depend on type information, which makes it natural to combine calculation of types with the actual translation or to pass type information from a type-check phase to the translation phase. In Chap. 4, we covered type-checking during execution, which is normally called *dynamic typing*. We will in this chapter assume that type checking and related checks are done in a phase previous to execution or translation (i.e., *static typing*), and similarly assume that any information gathered by this phase is available in subsequent phases.

5.1 The Design Space of Type Systems

We have already discussed the difference between static and dynamic typing, i.e., if type checks are made *before* or *during* execution of a program. Additionally, we can distinguish *weakly* and *strongly* typed languages.

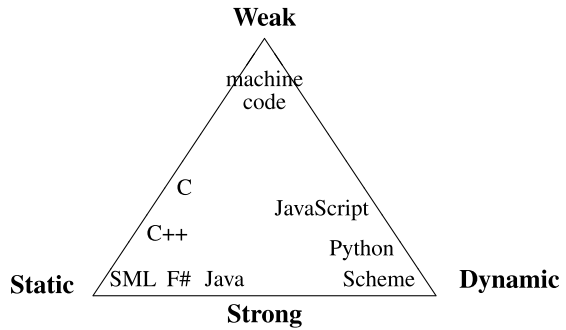
Strong typing means that the language implementation ensures that, whenever an operation is performed, the arguments to the operation are of types on which the operation is defined, so you, for example, do not try to concatenate a string and a floating-point number. This is independent of whether this is ensured statically (prior to execution) or dynamically (during execution).

In contrast, a weakly typed language gives no guarantee that operations are performed on arguments that make sense for the operation. The archetypical weakly typed language is machine code: Operations are performed with no prior checks, and if there is any concept of type at the machine level, it is fairly limited: Registers may be divided into integer, floating point and (possibly) address registers, and memory is (if at all) divided into only code and data areas. Weakly typed languages are mostly used for system programming, where you need to manipulate, move, copy, encrypt or compress data without regard to what that data represents.

Many languages combine both strong and weak typing or both static and dynamic typing: Some types are checked before execution and other types are checked during execution, and some types are not checked at all. For example, C is a statically typed language (since no checks are performed during execution), but not all types are checked, so it is somewhat weakly typed. For example, you can store an integer in a union-typed variable and read it back as a pointer or floating-point number. While Java is generally considered a statically typed language, even this has run-time type checks: When you do a downcast, this is in essence a dynamic type check. A more extreme example is JavaScript: If you try to multiply two strings, the interpreter will see if the strings contain sequences of digits and, if so, convert the strings to numbers and multiply these. This is a form of weak typing, as the multiplication operation is applied to arguments (strings) where multiplication, mathematically speaking, does not make sense. But instead of, like machine code, blindly trying to multiply the machine representations of the strings as if they were numbers, JavaScript performs a dynamic check and a conversion to make the values conform to the operation. I will still call this behaviour weak typing, as there is nothing that indicates that converting strings to numbers before multiplication makes any more sense than just multiplying the machine representations of the strings. The main point is that the language, instead of reporting a possible problem, silently does something that might make no sense.

Figure 5.1 shows a diagram of the design space of static versus dynamic and weak versus strong types, placing some well-known programming languages in this design space. Note that the design space is shown as a triangle: If you never check types, you do so neither statically nor dynamically, so at the weak end of the weak/strong axis, the distinction between static and dynamic is meaningless. We have put Standard ML (SML) in the bottom-left corner as it does all type checking at compile time and guarantees that no type-incorrect operation ever happens. Scheme is in the bottom-

Fig. 5.1 The design space of type systems



right corner as all type checks are done at runtime, but errors are reported if the program attempts to do type-incorrect operations. The languages along the left edge are all statically typed, while those along the right edge are all dynamically typed.

5.2 Attributes

The checking phase operates on the abstract syntax tree of the program and may make several passes over this. Typically, each pass is a recursive walk over the syntax tree, gathering information, or using information gathered in earlier passes. Such information is often called *attributes* of the syntax tree. Typically, we distinguish between two types of attributes: *Synthesised attributes* are passed upwards in the syntax tree, from the leaves up to the root. *Inherited attributes* are, conversely, passed downwards in the syntax tree. Note, however, that information that is synthesised in one subtree may be inherited by another subtree or, in a later pass, by the same subtree. An example of this is a symbol table: This is synthesised by a declaration and inherited by the scope of the declaration. When declarations are recursive, the scope may be the a syntax tree that contains the declaration itself, in which case one pass over this tree will build the symbol table as a synthesised attribute, while a second pass will use it as an inherited attribute.

Typically, each *syntactic category* (represented by a type in the data structure for the abstract syntax tree or by a group of related nonterminals in the grammar) will have its own set of attributes. When we write a checker as a set of mutually recursive functions, there will be one or more such functions for each syntactic category. Each of these functions will take inherited attributes (including the syntax tree itself) as arguments, and return synthesised attributes as results.

We will, in this chapter, focus on type checking, and only briefly mention other properties that can be checked. The methods used for type checking can in most cases easily be modified to handle such other checks.

We will use the language from Sect. 4.2 as an example for static type checking.

5.3 Environments for Type Checking

In order to type check the program, we need symbol tables that bind variables and functions to their types. Since there are separate name spaces for variables and functions, we will use two symbol tables, one for variables and one for functions. A variable is bound to one of the two types `int` or `bool`. A function is bound to its type, which consists of the list of types of its arguments and the type of its result. Function types are written as a parenthesised list of the argument types, an arrow and the result type, e.g. $(\text{int}, \text{bool}) \rightarrow \text{int}$ for a function taking two parameters (of type `int` and `bool`, respectively) and returning an integer.

We will assume that symbol tables are persistent, so no explicit action is required to restore the symbol table for the outer scope when exiting an inner scope. We don't need to preserve symbol tables for inner scopes once these are exited (so a stack-like behaviour is fine).

5.4 Type Checking of Expressions

When we type check expressions, the symbol tables for variables and functions are inherited attributes. The type (`int` or `bool`) of the expression is returned as a synthesised attribute. To make the presentation independent of any specific data structure for abstract syntax, we will (like in Chap. 4) let the type-checker functions for pattern-matching purposes use a notation similar to the concrete syntax. But you should still think of it as abstract syntax, so all issues of ambiguity, etc., have been resolved.

For terminals (variable names and numeric constants) with attributes, we assume that there are predefined functions for extracting these. Hence, **id** has an associated function *getname*, that extracts the name of the identifier. Similarly, **num** has a function *getvalue*, that returns the value of the number. The latter is not required for static type checking, but we used it in Chap. 4, and we will use it again in Chap. ??.

For each nonterminal, we define one or more functions that take an abstract syntax subtree and inherited attributes as arguments, and return the synthesised attributes.

In Fig. 5.2, we show the type-checking function for expressions. The function for type checking expressions is called *Check_{Exp}*. The symbol table for variables is given by the parameter *vtable*, and the symbol table for functions by the parameter *ftable*. The function **error** reports a type error. To allow the type checker to continue and report more than one error, we let the error-reporting function return.¹ After reporting a type error, the type checker can make a guess at what the type should have been and return this guess, allowing type checking to continue for the rest of the program. This guess might, however, be wrong, which can cause spurious type errors to be reported later on. Hence, all but the first type error message should be taken with a grain of salt.

¹ Unlike in Chap. 4, where the **error** function stops execution.

Fig. 5.2 Type checking of expressions

$Check_{Exp}(Exp, vtable, ftable) = \text{case } Exp \text{ of}$	
num	int
id	$t = \text{lookup}(vtable, \text{getname}(\mathbf{id}))$ if $t = \text{unbound}$ then error() ; int else t
$Exp_1 + Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ if $t_1 = \text{int}$ and $t_2 = \text{int}$ then int else error() ; int
$Exp_1 < Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ if $t_1 = t_2$ then bool else error() ; bool
if Exp_1 then Exp_2 else Exp_3	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $t_3 = Check_{Exp}(Exp_3, vtable, ftable)$ if $t_1 = \text{bool}$ and $t_2 = t_3$ then t_2 else error() ; t_2
id ($Exps$)	$t = \text{lookup}(ftable, \text{getname}(\mathbf{id}))$ if $t = \text{unbound}$ then error() ; int else $((t_1, \dots, t_n) \rightarrow t_0) = t$ $[t'_1, \dots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ if $m = n$ and $t_1 = t'_1, \dots, t_n = t'_n$ then t_0 else error() ; t_0
let id = Exp_1 in Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = \text{bind}(vtable, \text{getname}(\mathbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$
$Check_{Exps}(Exps, vtable, ftable) = \text{case } Exps \text{ of}$	
Exp	$[Check_{Exp}(Exp, vtable, ftable)]$
$Exp, Exps$	$Check_{Exp}(Exp, vtable, ftable)$ $:: Check_{Exps}(Exps, vtable, ftable)$

We will briefly explain each of the cases handled by $Check_{Exp}$.

- A number has type `int`.
- The type of a variable is found by looking its name up in the symbol table for variables. If the variable is not found in the symbol table, the lookup-function returns the special value `unbound`. When this happens, an error is reported and

the type checker arbitrarily guesses that the type is `int`. Otherwise, it returns the type returned by *lookup*.

- A plus-expression requires both arguments to be integers and has an integer result.
- Comparison requires that the arguments have the same type. In either case, the result is a boolean.
- In a conditional expression, the condition must be of type `bool` and the two branches must have identical types. The result of a condition is the value of one of the branches, so it has the same type as these. If the branches have different types, the type checker reports an error and arbitrarily chooses the type of the `then`-branch as its guess for the type of the whole expression. Note that the dynamic type checking done in Chap. 4 does not require that the branches have the same type: it only requires that the type of the chosen branch is consistent with how it is later used.
- At a function call, the function name is looked up in the function environment to find the number and types of the arguments as well as the return type. The number of arguments to the call must coincide with the expected number and their types must match the declared types. The resulting type is the return-type of the function. If the function name is not found in *fable*, an error is reported and the type checker arbitrarily guesses the result type to be `int`.
- A `let`-expression declares a new variable, the type of which is that of the expression that defines the value of the variable. The symbol table for variables is extended using the function *bind*, and the extended table is used for checking the body-expression and finding its type, which in turn is the type of the whole expression. A `let`-expression can not in itself be the cause of a type error (though its subexpressions may), so no testing is done.

Since $Check_{Exp}$ mentions the nonterminal $Exps$ and its related type-checking function $Check_{Exps}$, we have included $Check_{Exps}$ in Fig. 5.2.

$Check_{Exps}$ builds a list of the types of the expressions in the expression list. The notation is taken from SML: A list is written in square brackets with elements separated by commas. The operator `::` adds an element to the front of a list.

Suggested exercises: 5.1.

5.5 Type Checking of Function Declarations

A function declaration explicitly declares the types of the arguments to the function. This information is used to build a symbol table for variables, which is used when type checking the body of the function. The type of the body must match the declared result type of the function. The type check function for functions, $Check_{Fun}$, has as inherited attribute the symbol table for functions, which is passed down to the type check function for expressions. $Check_{Fun}$ returns no information, it just checks for errors. $Check_{Fun}$ is shown in Fig. 5.3, along with the functions for $TypeId$ and $TypeIds$, which it uses. The function Get_{TypeId} just returns a pair of the declared name and type,

Fig. 5.3 Type checking a function declaration

$$\begin{array}{l}
\text{Check}_{Fun}(Fun, ftable) = \text{case } Fun \text{ of} \\
\hline
TypeId (TypeIds) = Exp \quad \left| \begin{array}{l} (f, t_0) = Get_{TypeId}(TypeId) \\ vtable = Check_{TypeIds}(TypeIds) \\ t_1 = Check_{Exp}(Exp, vtable, ftable) \\ \text{if } t_0 \neq t_1 \\ \text{then } \mathbf{error}() \end{array} \right. \\
\\
\\
\text{Get}_{TypeId}(TypeId) = \text{case } TypeId \text{ of} \\
\hline
\text{int } \mathbf{id} \quad \left| (getname(\mathbf{id}), \text{int}) \right. \\
\hline
\text{bool } \mathbf{id} \quad \left| (getname(\mathbf{id}), \text{bool}) \right. \\
\\
\\
\text{Check}_{TypeIds}(TypeIds) = \text{case } TypeIds \text{ of} \\
\hline
TypeId \quad \left| \begin{array}{l} (x, t) = Get_{TypeId}(TypeId) \\ bind(emptytable, x, t) \end{array} \right. \\
\hline
TypeId, TypeIds \quad \left| \begin{array}{l} (x, t) = Get_{TypeId}(TypeId) \\ vtable = Check_{TypeIds}(TypeIds) \\ \text{if } lookup(vtable, x) = \text{unbound} \\ \text{then } bind(vtable, x, t) \\ \text{else } \mathbf{error}(); vtable \end{array} \right.
\end{array}$$

and $Check_{TypeIds}$ builds a symbol table from such pairs. $Check_{TypeIds}$ also verifies that all parameters have different names by checking that a name is not already bound before adding it to the table. $emptytable$ is an empty symbol table. Looking any name up in the empty symbol table returns *unbound*.

5.6 Type Checking a Program

A program is a list of functions, and is deemed type correct if all the functions are type correct, and there are no two function definitions defining the same function name. Additionally, there must be a function called `main` with one integer argument and an integer result.

Since all functions are mutually recursive, each of these must be type checked using a symbol table where all functions are bound to their type. This requires two passes over the list of functions: One to build the symbol table, and one to check the function definitions using this table. Hence, we need two functions operating over $Funs$ and two functions operating over Fun . We have already seen one of the latter, $Check_{Fun}$. The other, Get_{Fun} , returns the pair of the function's declared name and type, which consists of the types of the arguments and the type of the result. It uses an auxiliary function Get_{Types} to find the types of the arguments. The two functions for the syntactic category $Types$ are Get_{Types} , which builds the function symbol table and checks for duplicate definitions, and $Check_{Types}$, which calls $Check_{Fun}$ for all functions. These functions and the main function $Check_{Program}$, which ties the loose ends, are shown in Fig. 5.4.

Fig. 5.4 Type checking a program

$$\begin{array}{c}
 \hline
 \text{Check}_{Program}(Program) = \text{case } Program \text{ of} \\
 \hline
 \text{Funs} \quad \left| \begin{array}{l}
 \text{ftable} = \text{Get}_{Funs}(Funs) \\
 \text{Check}_{Funs}(Funs, \text{ftable}) \\
 \text{if } \text{lookup}(\text{ftable}, \text{main}) \neq (\text{int}) \rightarrow \text{int} \\
 \text{then } \mathbf{error}()
 \end{array} \right. \\
 \\
 \text{Get}_{Funs}(Funs) = \text{case } Funs \text{ of} \\
 \hline
 \text{Fun} \quad \left| \begin{array}{l}
 (f, t) = \text{Get}_{Fun}(Fun) \\
 \text{bind}(\text{emptytable}, f, t)
 \end{array} \right. \\
 \hline
 \text{Fun Funs} \quad \left| \begin{array}{l}
 (f, t) = \text{Get}_{Fun}(Fun) \\
 \text{ftable} = \text{Get}_{Funs}(Funs) \\
 \text{if } \text{lookup}(\text{ftable}, f) = \text{unbound} \\
 \text{then } \text{bind}(\text{ftable}, f, t) \\
 \text{else } \mathbf{error}(); \text{ftable}
 \end{array} \right. \\
 \\
 \text{Get}_{Fun}(Fun) = \text{case } Fun \text{ of} \\
 \hline
 \text{TypeId } (TypeIds) = \text{Exp} \quad \left| \begin{array}{l}
 (f, t_0) = \text{Get}_{TypeId}(TypeId) \\
 [t_1, \dots, t_n] = \text{Get}_{Types}(TypeIds) \\
 (f, (t_1, \dots, t_n) \rightarrow t_0)
 \end{array} \right. \\
 \\
 \text{Get}_{Types}(TypeIds) = \text{case } TypeIds \text{ of} \\
 \hline
 \text{TypeId} \quad \left| \begin{array}{l}
 (x, t) = \text{Get}_{TypeId}(TypeId) \\
 [t]
 \end{array} \right. \\
 \hline
 \text{TypeId TypeIds} \quad \left| \begin{array}{l}
 (x_1, t_1) = \text{Get}_{TypeId}(TypeId) \\
 [t_2, \dots, t_n] = \text{Get}_{Types}(TypeIds) \\
 [t_1, t_2, \dots, t_n]
 \end{array} \right. \\
 \\
 \text{Check}_{Funs}(Funs, \text{ftable}) = \text{case } Funs \text{ of} \\
 \hline
 \text{Fun} \quad \left| \text{Check}_{Fun}(Fun, \text{ftable}) \right. \\
 \hline
 \text{Fun Funs} \quad \left| \begin{array}{l}
 \text{Check}_{Fun}(Fun, \text{ftable}) \\
 \text{Check}_{Funs}(Funs, \text{ftable})
 \end{array} \right.
 \end{array}$$

This completes type checking of our small example language.

Suggested exercises: 5.5.

5.7 Advanced Type Checking

Our example language is very simple and obviously does not cover all aspects of type checking. A few examples of other features and brief explanations of how they can be handled are listed below.

Assignments

When a variable is given a value by an assignment, it must be verified that the type of the value is the same as the declared type of the variable. Some compilers may additionally check if a variable can be used before it is given a value, or if a variable is not used after its assignment. While not exactly type errors, such behaviour is likely to be undesirable. Testing for such behaviour does, however, require somewhat more complicated analysis than the simple type checking presented in this chapter, as it relies on non-structural information. It can be done by *data-flow analysis*, see Chap. 10.

Data Structures

A data structure declaration may define a value with several components (e.g., an *array*, *struct*, *tuple*, or *record*), or a value that may be of different types at different times (e.g. a *union*, *variant*, or *sum*). To type check such structures, the type checker must be able to represent their types. Hence, the type checker may need a data structure that describes complex types. This may be similar to the data structure used for the abstract syntax trees of type declarations. Operations that build or take apart structured data need to be tested for correctness. If each operation on structured data has (implicitly or explicitly) declared types for its arguments and a declared type for its result, this can be done in a way similar to how function calls are tested. For example, the representation of an array type must include the number of dimensions of the array and the type of the elements. When the array is declared, the representation of the type is constructed, and when an array is used, e.g., at an array lookup, it is checked that the number of dimensions used in the lookup is correct, and the type of the result is found in the representation of the array type. The representation of a struct type must contain information about the names and types of the fields. These are found when the struct type is declared. When a field of the record is accessed, the representation of the struct type is used to verify that the field name is one of the valid field names for the type, and the type of this field is returned as the type of the field access.

Overloading

Overloading means that the same name is used for several different operations over several different types. We saw a simple example of this in the example language, where `<` was used both for comparing integers and for comparing booleans. In many languages, arithmetic operators like `+` and `-` are defined both over integers and floating point numbers, and possibly other types as well. If these operators are predefined, and there is only a finite number of cases they cover, all the possible cases may be tried in turn, just like in our example.

This, however, requires that the different instances of the operator have disjoint argument types. If, for example, there is a function *read* that reads a value from a text stream, and this is defined to read either integers or floating point numbers, the

argument (the text stream) alone can not be used to select the right operator. Hence, the type checker must pass the *expected* type of each expression down as an inherited attribute, so this (possibly in combination with the types of the arguments) can be used to select the correct instance of the overloaded operator.

It may not always be possible to send down an expected type due to lack of information. In our example language, this is the case for the arguments to `=` (as these may be either `int` or `bool`), and the first expression in a `let`-expression (since the variable bound in the `let`-expression is not declared to be a specific type). If the type checker for this or some other reason is unable to pick a unique operator, it may report “unresolved overloading” as a type error, or it may pick a default instance.

Type Conversion

A language may have operators for converting a value of one type to a value of another type, e.g. an integer to a floating point number. Sometimes these operators are explicit in the program and hence easy to check. However, many languages allow implicit conversion of integers to floats, such that, for example, `3 + 3.12` is well-typed with the implicit assumption that the integer `3` is converted to a float before the addition. This can be handled as follows: If the type checker discovers that the arguments to an operator do not have the correct type, it can try to convert one or both arguments to see if this helps. If there is a small number of predefined legal conversions, this is no major problem. However, a combination of user-defined overloaded operators and user-defined types with conversions can make the type-checking process quite difficult, as the information needed to choose correctly may not be available at compile-time. This is typically the case in object-oriented languages, where method selection is often done at run-time. We will not go into details of how this can be done.

Polymorphism/Generic Types

Some languages allow a function to be *polymorphic* or *generic*, that is, to be defined over a large class of similar types, e.g. over all arrays no matter what the types of the elements are. A function can explicitly declare which parts of the type is generic/polymorphic, or this can be implicit (see below). The type checker can insert the actual types at every use of the generic/polymorphic function to create *instances* of the generic/polymorphic type. This mechanism is different from overloading, as the instances will be related by a common generic type, and because a polymorphic/generic function can be instantiated by (almost) *any* type, not just by a limited list of declared alternatives as is the case with overloading. We will, briefly, look at polymorphism in Sect. 12.1.

Implicit Types

Some languages (like Standard ML and Haskell) require programs to be well-typed, but do not require explicit type declarations for variables or functions. For such to work, a *type inference* algorithm is used. A type inference algorithm gathers information about *uses* of functions and variables and uses this information to infer

the types of these. For example, if one branch of an if-then-else expression is a number constant, then both the other branch and the result of the if-then-else expression must be of number type. If there are inconsistent uses of a variable or function, or if the branches of an if-then-else expression have different inferred types, a type error is reported. A simple case is found in our example language, where the type of a variable bound in a let-expression is not declared, but is implicit the type of the expression to which it is bound.

Suggested exercises: 5.2.

5.8 Further Reading

Overloading of operators and functions is described in Sect. 6.5 of [1]. Section 6.7 of same describes how polymorphism can be handled.

Some theory and a more detailed algorithm for inferring types in a language with implicit types and polymorphism can be found in [2]. Types in general are covered in detail by [4]. See also [3].

5.9 Exercises

Exercise 5.1 We extend the language from Sect. 4.2 with Boolean operators as described in Exercise 4.1.

Extend the type-check function in Fig. 5.2 to handle these new constructions as described above.

Exercise 5.2 We extend the language from Sect. 4.2 with floating-point numbers as described in Exercise 4.2.

- (a) Extend the type checking functions in Figs. 5.2, 5.3 and 5.4 to handle these extensions.
- (b) We now add implicit conversion of integers to floats to the language, using the rules: Whenever an operator has one integer argument and one floating-point argument, the integer is converted to a float. Similarly, if a condition expression (`if-then-else`) has one integer branch and one floating-point branch, the integer branch is converted to floating-point. Extend the type checking functions from question a) above to handle this.

Exercise 5.3 The type check function in Fig. 5.2 tries to guess the correct type when there is a type error. In some cases, the guess is arbitrarily chosen to be `int`, which may lead to spurious type errors later on. A way around this is to have an extra type: `unknown`, which is only used during type checking. If there is a type error,

and there is no basis for guessing a correct type, `unknown` is returned (the error is still reported, though). If an argument to an operator is of type `unknown`, the type checker should not report this as a type error but continue as if the type is correct. The use of an `unknown` argument to an operator may make the result `unknown` as well, so these can be propagated arbitrarily far.

Change Fig. 5.2 to use the `unknown` type as described above.

Exercise 5.4 We look at a simple language with an exception mechanism:

$$\begin{aligned} S &\rightarrow \text{throw } \mathbf{id} \\ S &\rightarrow \text{try } S \text{ catch } \mathbf{id} \Rightarrow S \\ S &\rightarrow S \text{ or } S \\ S &\rightarrow \text{other} \end{aligned}$$

A `throw` statement throws a named exception. This is caught by the nearest enclosing `try-catch` statement (i.e., where the `throw` statement is in the left sub-statement of the `try-catch` statement) that uses the same name, whereby the statement after the arrow in the `try-catch` statement is executed. An `or` statement is a non-deterministic choice between the two statements, so either one can be executed. `other` is a statement that does not throw any exceptions.

We want the type checker to ensure that all possible exceptions are caught and that no `try-catch` statement is superfluous, i.e., that the exception it catches can, in fact, be thrown by its left sub-statement.

Write type-check functions that implement these checks. Hint: Let the type of a statement be the set of possible exceptions it can throw.

Exercise 5.5 In Exercise 4.5, we extended the example language with functional values and implemented these in the interpreter.

Extend the type-checking functions in Figs. 5.2, 5.3 and 5.4 to statically type check the same extensions.

Hint: You should check a function definition when it is declared rather than when it is used.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers; Principles, Techniques and Tools. Addison-Wesley (2007)
2. Milner, R.: A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **17**(3), 348–375 (1978)
3. Mogensen, T.Æ.: Programming Language Design and Implementation. Springer, Cham, Switzerland (2022). <https://link.springer.com/book/10.1007/978-3-031-11806-7>
4. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)

Chapter 6

Intermediate-Code Generation



The art of free society consists first in the maintenance of the symbolic code; and secondly in fearlessness of revision, to secure that the code serves those purposes which satisfy an enlightened reason.

Alfred North Whitehead (1869–1947)

The ultimate goal of a compiler is to get programs written in a high-level language to run on a computer. This means that, eventually, the program will have to be expressed as machine code that can run on the computer. This does not mean that we need to translate directly from the high-level abstract syntax to machine code. Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called *intermediate code*.

Apart from structuring the compiler into smaller jobs, using an intermediate language has other advantages:

- If the compiler needs to generate code for several different machine architectures, only one translation to intermediate code is needed. Only the translation from intermediate code to machine language (i.e., the *back-end*) needs to be written in several versions.
- If several high-level languages need to be compiled, only the translation to intermediate code need to be written for each language. They can all share the back-end, i.e., the translation from intermediate code to machine code.
- Instead of translating the intermediate language to machine code, it can be *interpreted* by a small program written in machine code or in a language for which a compiler or interpreter already exists.

- A lot of optimisations are more easily done at the intermediate-code level than at the source-code level or the machine-code level. And they can be shared by all compilers that use the intermediate code.

The advantage of using an intermediate language is most obvious if many languages are to be compiled to many machines. If translation is done directly, the number of compilers is equal to the product of the number of languages and the number of machines. If a common intermediate language is used, one front-end (i.e., compiler to intermediate code) is needed for every language and one back-end (interpreter or code generator) is needed for each machine, making the total number of front-ends and back-ends equal to the sum of the number of languages and the number of machines. Additionally, each of these is typically simpler than a compiler that compiles directly to machine code.

If an interpreter for an intermediate language is written in a language for which there already exist implementations for the target machines, the same interpreter can be interpreted or compiled for each machine. This way, there is no need to write a separate back-end for each machine. The advantages of this approach are:

- No actual back-end needs to be written for each new machine, as long as the machine is equipped with an interpreter or compiler for the implementation language of the interpreter for the intermediate language.
- A compiled program can be distributed in a single intermediate form for all machines, as opposed to shipping separate binaries for each machine.
- The intermediate form may be more compact than machine code. This saves space both in distribution and on the machine that executes the programs (though the latter is somewhat offset by requiring the interpreter to be kept in memory during execution).

The disadvantage is speed: Interpreting the intermediate form will in most cases be (sometimes a lot) slower than executing translated code directly. Nevertheless, the approach has seen some success, e.g., in early implementations of Java and in the Moscow ML compiler for Standard ML.

Some of the speed penalty can be eliminated by translating the intermediate code to machine code immediately before or during execution of the program. This hybrid form is called *just-in-time compilation* and is often used in modern implementations of Java for executing the intermediate code (the Java Virtual Machine).

We will in this book, however, focus mainly on using the intermediate code for traditional compilation, where the intermediate form will be translated to machine code by the back-end of the compiler.

6.1 Designing an Intermediate Language

An intermediate language should, ideally, have the following properties:

- It should be easy to translate high-level languages to the intermediate language. This should be the case for a wide range of different source languages.
- It should be easy to translate the intermediate language to machine code. This should be true for a wide range of different target architectures.
- The intermediate format should be suitable for optimisations.

The first two of these properties can be somewhat hard to reconcile. A language that is intended as target for translation from a high-level language should be fairly high level itself. However, this may be hard to achieve for more than a small number of similar high-level languages. Furthermore, a high-level intermediate language puts more burden on the back-ends. A low-level intermediate language may make it easy to write back-ends, but puts more burden on the front-ends. A low-level intermediate language, also, may not fit all machines equally well, though this is usually less of a problem than the similar problem for front-ends, as machine languages typically are more similar than high-level languages.

A solution that may reduce the translation burden, though it does not address the other problems, is to have two intermediate levels: One, which is fairly high-level, is used for the front-ends and the other, which is fairly low-level, is used for the back-ends. A single shared translator is then used to translate between these two intermediate formats.

When the intermediate format is shared between many compilers, it makes sense to do as many optimisations as possible on the intermediate format. This way, the (often substantial) effort of writing good optimisations is done only once instead of in every compiler.

Another thing to consider when choosing an intermediate language is the “granularity”: Should an operation in the intermediate language correspond to a large amount of work or to a small amount of work?

The first of these approaches is often used when the intermediate language is interpreted, as the overhead of decoding instructions is amortised over more actual work, but it can also be used for compiling. In this case, each intermediate-code instruction is typically translated into a sequence of machine-code instructions. When coarse-grained intermediate code is used, there is typically a fairly large number of different intermediate-code instructions.

The opposite approach is to let each intermediate-code instruction be as simple as possible. This means that each intermediate-code instruction is typically translated into a single machine-code instruction or that several intermediate-code instructions can be combined into one machine-code instruction. The latter can, to some degree, be automated as each machine-code instruction can be described as a sequence of intermediate-code instructions. When intermediate-code is translated to machine-code, the code generator can look for sequences that match machine-code instructions. By assigning cost to each machine-code instruction, this can be turned into a

combinatorial optimisation problem, where the least-cost solution is found. We will return to this in Chap. 7.

6.2 The Intermediate Language

For this chapter, we have chosen a fairly low-level fine-grained intermediate language, as it is best suited to convey the techniques we want to cover.

We will not treat translation of function calls until Chap. 9, so a “program” in our intermediate language will, for the time being, keep function definitions, calls and returns as primitive constructions in the intermediate language. In Chap. 9, we will see how these can be translated into lower-level code.

The grammar for the intermediate language is shown in Grammar 6.1.

A program is a sequence of function definitions, each of which consists of a header and a body. The header defines the name of the function and its arguments, and the body is a list of instructions. The instructions are:

- A label. This has no effect but serves only to mark the position in the program as a target for jumps.
- An assignment of an atomic expression (constant or variable) to a variable.

Grammar 6.1 The intermediate language

```

Program    → Functions
Functions  → Function Functions
Functions  → Function
Function   → Header Body
Header     → functionid(Args)
Body       → [ Instructions ]
Instructions → Instruction
Instructions → Instruction , Instructions
Instruction → LABEL labelid
Instruction → id := Atom
Instruction → id := unop Atom
Instruction → id := id binop Atom
Instruction → id := M[Atom]
Instruction → M[Atom] := id
Instruction → GOTO labelid
Instruction → IF id relop Atom THEN labelid ELSE labelid
Instruction → id := CALL functionid(Args)
Instruction → RETURN id

Atom       → id
Atom       → num

Args       → id
Args       → id , Args
  
```


- A unary operator applied to an atomic expression, with the result stored in a variable.
- A binary operator applied to a variable and an atomic expression, with the result stored in a variable.
- A transfer from memory to a variable. The memory location is an atomic expression.
- A transfer from a variable to memory. The memory location is an atomic expression.
- A jump to a label.
- A conditional selection between jumps to two labels. The condition is found by comparing a variable with an atomic expression by using a relational operator ($=$, \neq , $<$, $>$, \leq or \geq).
- A function call. The arguments to the function call are variables and the result is assigned to a variable. This instruction is used even if there is no actual result (i.e., if a procedure is called instead of a function), in which case the result variable is a dummy variable.
- A return statement. This returns the value of the specified variable as the result of the current function. We require that a function will always exit by executing a RETURN instruction, i.e, not by “falling out” of the last instruction in its body. In practice, this means that the last instruction in a function body must be either a RETURN instruction, a GOTO instruction or an IF-THEN-ELSE instruction, the branches of which are also in these three categories.

An atomic expression is either a variable or a constant.

We have not specified the set of unary and binary operations, but we expect these to include normal integer arithmetic and bitwise logical operations.

Variables (including function parameters) are local to the function definition in which they are used, and they do not have to be declared in advance. We assume that all values are integers. Adding floating-point numbers and other primitive types (such as Booleans, characters or pointers) is not difficult, though: Variables and operators should be given types. If this is done, memory accesses should use pointer, and there should be explicit operators for converting between the different types. To keep things simple, we stick to integers, which are assumed to be of a size bounded by one machine word (typically 64 bits).

A label is local to the function in which it occurs. There can not be two LABEL instructions with the same label name in the same function definition.

Jumps are local to the function definition in which they occur, so you can not jump into or out of a function definition. If there is a jump to a label l in a function definition, there must also be a LABEL l instruction in the same function definition.

6.3 Syntax-Directed Translation

We will generate code using translation functions for each syntactic category, similar to the functions we used for interpretation and type checking. We generate code for a syntactic construct independently of the constructs around it, except that the parameters of a translation function may hold information about the context (such as symbol tables) and the result of a translation function may (in addition to the generated code) hold information about how the generated code interfaces with its context (such as which variables it uses). Since the translation closely follows the syntactic structure of the program, it is called *syntax-directed translation*.

Given that translation of a syntactic construct is mostly independent of the surrounding and enclosed syntactic constructs, we might miss opportunities to exploit synergies between these and, hence, generate less than optimal code. We will try to remedy this in later chapters by using various optimisation techniques.

6.4 Generating Code from Expressions

Grammar 6.2 shows a simple language of expressions, which we will use as our initial example for translation. Again, we have let the set of unary and binary operators be unspecified but assume that the intermediate language includes all those used by the expression language. We assume that there is a function *transop* that translates the name of an operator in the expression language into the name of the corresponding operator in the intermediate language. The tokens **unop** and **binop** have the names of the actual operators as attributes, accessed by the function *getopname*.

When writing a compiler, we must decide what needs to be done at compile-time and what needs to be done at run-time. Ideally, as much as possible should be done at compile-time, but some things need to be postponed until run-time, as they need access to the actual values of variables, etc., which are not known at compile-time. When we, below, explain the workings of the translation functions, we might use phrasing like “the expression is evaluated and the result stored in the variable”. This describes actions that are performed at run-time by the code that is generated at compile-time. At times, the textual description may not be 100% clear as to what

Grammar 6.2 A simple expression language

$$\begin{aligned}
 \text{Exp} &\rightarrow \mathbf{num} \\
 \text{Exp} &\rightarrow \mathbf{id} \\
 \text{Exp} &\rightarrow \mathbf{unop} \text{Exp} \\
 \text{Exp} &\rightarrow \text{Exp} \mathbf{binop} \text{Exp} \\
 \text{Exp} &\rightarrow \mathbf{id}(\text{Exps}) \\
 \text{Exps} &\rightarrow \text{Exp} \\
 \text{Exps} &\rightarrow \text{Exp} , \text{Exps}
 \end{aligned}$$

happens at which time, but the notation used in the translation functions makes this clear: Intermediate-language code (or equivalent machine language) is executed at run-time, the rest is done at compile time. Intermediate-language instructions may refer to values (constants and variable names) that are generated at compile time. When instructions have operands that are written in *italics*, these operands are variables in the compiler that contain compile-time values that are inserted into the generated code. For example, if *place* holds the variable name `t14` and *v* holds the value 42, then the code template [*place* := *v*] will generate the code [`t14 := 42`].

When we want to translate the expression language to the intermediate language, the main complication is that the expression language is tree-structured while the intermediate language is flat, requiring the result of every operation to be stored in a variable and every (non-constant) argument to be fetched from a variable. We use a function *newvar* at compile time to generate new intermediate-language variable names. Whenever *newvar* is called, it returns a previously unused variable name, so it is not a function in the mathematical sense (as a mathematical function would return the same value every time).

We will describe translation of expressions by a translation function using a notation similar to the notation we used for type-checking functions in Chap. 5.

Some attributes for the translation function are obvious: The translation function must return the code as a synthesised attribute. Furthermore, it must translate the names of variables and functions used in the expression language to the names these correspond to in the intermediate language. This can be done by symbol tables *vtable* and *ftable* that bind variable and function names in the expression language into the corresponding names in the intermediate language. The symbol tables are passed as inherited attributes to the translation function. In addition to these attributes, the translation function must use attributes to determine where it should put the values of subexpressions. This can be done in two ways:

- (1) The locations (variables) of the values of a subexpression can be passed up as a synthesised attribute to the parent expression, which decides on a location for its own value and returns this as synthesised attribute.
- (2) The parent expression can determine in which variables it wants to find the values of its subexpressions, and pass this information down to the subexpressions as inherited attributes.

In both cases, new variables will usually be generated for the intermediate values, though it can be tempting to reuse variables.

When generating code for a variable expression by method 1, we might want to simply pass the (intermediate-code version of) that variable up as the location of the value of the subexpression. This, however, only works under the assumption that the variable is not updated before the value is used by the parent expression. If expressions can have side effects, this is not always the case, as the C expression “`x+ (x=3)`” shows. If *x* has the value 5 prior to evaluation, the intended result is 8. But if the addition just uses the location of *x* as both arguments for the addition, it will return 6. We do not have assignments in our expression language, but to prepare

for later extensions, it is best to copy the value of a variable into a new variable whenever it is used.

Method 2 will have a similar problem if we add assignments: When generating code for an assignment (using C-like notation) “ $x = e$ ”, where e is an expression, it could be tempting to just pass (the intermediate-code version of) x down as the location where to store the value of e . This will work only under the assumption that the code for e does not store anything in its given location until the end of the evaluation of e . While this may seem like a reasonable assumption, it is better to be safe and generate a new variable and only copy the value of this to x when the assignment is made. Generation of new variables is done at compile time, so it does not cost anything at run time. Copying values from one variable to another may cost something at run time, but as we shall see in Chaps. 7, 8, and 10, most of the copying can be eliminated in later phases.

If new variables are generated for all intermediate values, both methods will give the same code (up to renaming of variables). Notationally, method 2 is slightly less cumbersome, so we will use this for our translation function $Trans_{Exp}$, which is shown in Fig. 6.3. We will, however, use method 1 for the translation function $Trans_{Exps}$, that generates code for a list of expressions, partly to illustrate both styles, but also because it is slightly more convenient to use method 1 for $Trans_{Exps}$.

In $Trans_{Exp}$, the inherited attribute *place* is the intermediate-language variable that the result of the expression must be stored in.

If the expression is just a number, the value of that number is stored in the *place*.

If the expression is a variable, the intermediate-language equivalent of this variable is found in *vtable*, and an assignment copies it into the intended *place*.

A unary operation is translated by first generating a new intermediate-language variable to hold the value of the argument of the operation. Then the argument is translated using the newly generated variable for its *place* attribute. We then use an **unop** operation in the intermediate language to assign the result to the inherited *place*. The operator ++ concatenates two lists of instructions.

A binary operation is translated in a similar way. Two new intermediate-language variables are generated to hold the values of the arguments, then the arguments are translated, and finally a binary operation in the intermediate language assigns the final result to the inherited *place*.

A function call is translated by first translating the arguments, using the auxiliary function $Trans_{Exps}$. Then a function call is generated using the argument variables returned by $Trans_{Exps}$, with the result assigned to the inherited *place*. The name of the function is looked-up in *fable* to find the corresponding intermediate-language name.

$Trans_{Exps}$ generates code for each argument expression, storing the results into new variables. These variables are returned along with the code, so they can be put into the argument list of the call instruction.

Fig. 6.3 Translating an expression

$Trans_{Exp}(Exp, vtable, ftable, place) = \text{case } Exp \text{ of}$	
num	$v = \text{getvalue}(\mathbf{num})$ $[place := v]$
id	$x = \text{lookup}(vtable, \text{getname}(\mathbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = \text{newvar}()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = \text{transop}(\text{getopname}(\mathbf{unop}))$ $code_1 ++ [place := op\ place_1]$
Exp_1 binop Exp_2	$place_1 = \text{newvar}()$ $place_2 = \text{newvar}()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = \text{transop}(\text{getopname}(\mathbf{binop}))$ $code_1 ++ code_2 ++ [place := place_1\ op\ place_2]$
id ($Exps$)	$(code_1, [a_1, \dots, a_n])$ $\quad = Trans_{Exps}(Exps, vtable, ftable)$ $fname = \text{lookup}(ftable, \text{getname}(\mathbf{id}))$ $code_1 ++ [place := \text{CALL}\ fname(a_1, \dots, a_n)]$
$Trans_{Exps}(Exps, vtable, ftable) = \text{case } Exps \text{ of}$	
Exp	$place = \text{newvar}()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
$Exp, Exps$	$place = \text{newvar}()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 ++ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

6.4.1 Examples of Expression Translation

Translation of expressions is always relative to symbol tables and a place for storing the result. In the examples below, we assume a variable symbol table that binds x , y and z to v_0 , v_1 and v_2 , respectively, and a function table that binds f to $_f$. The place for the result is t_0 , and we assume that calls to $\text{newvar}()$ return, in sequence, the variable names t_1 , t_2 , t_3 , ...

We start by the simple expression $x-3$. This is a binop-expression, so we first call $\text{newvar}()$ twice, giving $place_1$ the value t_1 and $place_2$ the value t_2 . We then call $Trans_{Exp}$ recursively with the expression x and $place_1$ (which is equal to t_1) as the intended location of the result. When translating this, we first look up x in the variable symbol table, yielding v_0 , and then return the code $[t_1 := v_0]$. Back in the translation of the subtraction expression, we assign this code to $code_1$ and once more call $Trans_{Exp}$ recursively, this time with the expression 3 . This is translated to the code $[t_2 := 3]$, which we assign to $code_2$. The final result is produced

by $code_1 ++ code_2 ++ [t_0 := t_1 - t_2]$ which yields $[t_1 := v_0, t_2 := 3, t_0 := t_1 - t_2]$. The source-language operator $-$ is by *transop* translated to the intermediate-language operator $-$.

The resulting code looks quite sub-optimal, and could, indeed, be shortened to $[t_0 := v_0 - 3]$. When we generate intermediate code, we want, for simplicity, to treat each subexpression independently of its context. This may lead to such superfluous assignments. We will look at ways of getting rid of these when we treat machine code generation, register allocation, and data-flow analysis in Chaps. 7, 8 and 10.

A more complex expression is $3 + f(x - y, z)$. Using the same assumptions as above, this yields the code

```

t1 := 3
  t4 := v0
  t5 := v1
  t3 := t4 - t5
  t6 := v2
  t2 := CALL_f(t3, t6)
t0 := t1 + t2

```

We have, for readability, laid the code out on separate lines rather than using a comma-separated list. The indentation indicates the depth of calls to *Trans_{Exp}* that produced the code in each line.

Suggested exercises: 6.1.

6.5 Translating Statements

We now extend the expression language in Grammar 6.2 with statements. The extensions are shown in Grammar 6.4. Note that we use $:=$ for assignment to distinguish assignment from equality comparison. The grammar is ambiguous, but we work on abstract syntax, where the ambiguities have been resolved.

When translating statements, we will need the symbol table for variables (for translating assignment), and since statements contain expressions, we also need *ftable* so we can pass both symbol tables on to *Trans_{Exp}*.

Grammar 6.4 Statement language

```

Stat  → Stat ; Stat
Stat  → id := Exp
Stat  → if Cond then Stat
Stat  → if Cond then Stat else Stat
Stat  → while Cond do Stat
Stat  → repeat Stat until Cond
Cond  → Exp relop Exp

```

$$Trans_{Stat}(Stat, vtable, ftable) = \text{case } Stat \text{ of}$$

$Stat_1 ; Stat_2$	$code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 ++ code_2$
$id := Exp$	$place = newvar()$ $x = lookup(vtable, getname(id))$ $Trans_{Exp}(Exp, vtable, ftable, place) ++ [x := place]$
$\text{if } Cond$ $\text{then } Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_1 ++ [LABEL label_1] ++ code_2$ $++ [LABEL label_2]$
$\text{if } Cond$ $\text{then } Stat_1$ $\text{else } Stat_2$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_3 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 ++ [LABEL label_1] ++ code_2$ $++ [GOTO label_3, LABEL label_2]$ $++ code_3 ++ [LABEL label_3]$
$\text{while } Cond$ $\text{do } Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_2, label_3, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $[LABEL label_1] ++ code_1$ $++ [LABEL label_2] ++ code_2$ $++ [GOTO label_1, LABEL label_3]$
$\text{repeat } Stat_1$ $\text{until } Cond$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond, label_2, label_1, vtable, ftable)$ $[LABEL label_1] ++ code_1$ $++ code_2 ++ [LABEL label_2]$

Fig. 6.5 Translation of statements

Just like we use *newvar* to generate new, not previously used, variables, we use a similar function *newlabel* to generate new labels. The translation function for statements is shown in Fig. 6.5. It uses an auxiliary translation function shown in Fig. 6.6 for translating conditions.

A sequence of two statements is translated by putting the code for these in sequence.

$$\frac{Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) = \text{case } Cond \text{ of}}{Exp_1 \text{ relop } Exp_2 \quad \left| \begin{array}{l} t_1 = \text{newvar}() \\ t_2 = \text{newvar}() \\ code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1) \\ code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2) \\ op = \text{transop}(\text{getopname}(\text{relop})) \\ code_1 ++ code_2 ++ [\text{IF } t_1 \text{ op } t_2 \text{ THEN } label_t \text{ ELSE } label_f] \end{array} \right.}$$

Fig. 6.6 Translation of simple conditions

An assignment is translated by translating the right-hand-side expression to code that places the result in a new variable, and then copying this to the left-hand-side variable.

When translating statements that use conditions, we use an auxiliary function $Trans_{Cond}$. $Trans_{Cond}$ translates the arguments to the condition and generates an IF-THEN-ELSE instruction using the same relational operator as the condition (but translated to an intermediate-code operator). The target labels of this instruction are inherited attributes to $Trans_{Cond}$.

An if-then statement is translated by first generating two labels: One for the then-branch, and one for the code following the if-then statement. The condition is translated by $Trans_{Cond}$, which is given the two labels as attributes. When (at run-time) the condition is true, the first of these are selected, and when false, the second is chosen. Hence, when the condition is true, the then-branch is executed followed by the code after the if-then statement. When the condition is false, we jump directly to the code following the if-then statement, hence bypassing the then-branch.

An if-then-else statement is treated similarly, but now the condition must choose between jumping to the then-branch or the else-branch. At the end of the then-branch, a jump bypasses the code for the else-branch by jumping to the label at the end. Hence, there is need for three labels: One for the then-branch, one for the else-branch and one for the code following the if-then-else statement.

If the condition in a while-do loop is true, the body must be executed, otherwise the body is by-passed and the code after the loop is executed. Hence, the condition is translated with attributes that provide the label for the start of the body and the label for the code after the loop. When the body of the loop has been executed, the condition must be re-tested for further passes through the loop. Hence, a jump is made to the start of the code for the condition. A total of three labels are thus required: One for the start of the loop, one for the loop body and one for the end of the loop.

A repeat-until loop is slightly simpler. The body precedes the condition, so there is always at least one pass through the loop. If the condition is true, the loop is terminated and we continue with the code after the loop. If the condition is false, we jump to the start of the loop. Hence, only two labels are needed: One for the start of the loop and one for the code after the loop.

Fig. 6.7 Example statements

```
f := 1;
n := 7;
repeat
  f := f * n;
  n := n - 1
until n = 0
```

```
t1 := 1
v0 := t1
t2 := 7
v1 := t2
LABEL L1
  t3 := v0
  t4 := v1
v0 := t3 * t4
  t5 := v1
  t6 := 1
v1 := t5 - t6
  t7 := v1
  t8 := 0
  IF t7 = v1 THEN L2 ELSE L1
LABEL L2
```

Fig. 6.8 Example statement translation

6.5.1 Example of Statement Translation

Using the translation rules in Figs. 6.5 and 6.6, the statements in Fig. 6.7 get translated into the code shown in Fig. 6.8, using a symbol table that binds f to $v0$ and n to $v1$.

Suggested exercises: 6.2.

6.6 Logical Operators

Logical conjunction, disjunction and negation are often available for conditions, so we can write, e.g., $(x = y \text{ or } y = z)$, where **or** is a logical disjunction operator. There are typically two ways to treat logical operators in programming languages:

- (1) Logical operators are similar to arithmetic operators: The arguments are evaluated, and the operator is applied to find the result.
- (2) The second operand of a logical operator is only evaluated if the first operand is insufficient to determine the result. This means that a logical **and** will only evaluate its second operand if its first argument evaluates to **true**, and a logical

or will only evaluate its second operand if its first argument is **false**. This variant is called *sequential logical operators*.

The C language has both variants. The arithmetic logical operators are called `&` and `|`, and the sequential variants are called `&&` and `||`.

The first variant is typically implemented by using bitwise logical operators and uses 0 to represent **false** and some nonzero value (typically 1 or -1) to represent **true**. In C, there is no separate Boolean type, so integers are used even at the source-code level to represent truth values. While any nonzero integer is treated as logical truth by conditional statements, comparison operators return 1, and bitwise logical operators `&` (bitwise **and**) and `|` (bitwise **or**) are used to implement the corresponding logical operations, so 1 is the preferred representation of logical truth. Logical negation is *not* handled by bitwise negation, as the bitwise negation of 1 is not 0. Instead, a special logical negation operator `!` is used that maps any non-zero value to 0, and 0 to 1. We assume an equivalent operator is available in the intermediate language. Some languages use -1 to represent logical truth, as all bits in this value are 1 (assuming two's complement representation is used, which is normally the case). This makes bitwise negation a valid implementation of logical negation.

Adding non-sequential logical operators to our language is not too difficult if we simply assume that the intermediate language includes the required relational operators, bitwise logical operations, and logical negation. We can now simply allow any expression to be used as a condition by adding the production

$$Cond \rightarrow Exp$$

to Grammar 6.4. If there is a separate Boolean type, we assume that a type checker has verified that the expression is of Boolean type. If, as in C, there is no separate Boolean type, the expression must be of integer type.

We then extend the translation function for conditions as follows:

$$\begin{array}{l} \text{Trans}_{Cond}(Cond, label_t, label_f, vtable, ftable) = \text{case } Cond \text{ of} \\ \hline \begin{array}{l|l} Exp_1 \text{ **relop** } Exp_2 & \begin{array}{l} t_1 = \text{newvar}() \\ t_2 = \text{newvar}() \\ code_1 = \text{Trans}_{Exp}(Exp_1, vtable, ftable, t_1) \\ code_2 = \text{Trans}_{Exp}(Exp_2, vtable, ftable, t_2) \\ op = \text{transop}(\text{getopname}(\text{relop})) \\ code_1 ++ code_2 ++ [\text{IF } t_1 \text{ } op \text{ } t_2 \text{ THEN } label_t \text{ ELSE } label_f] \end{array} \\ \hline Exp & \begin{array}{l} t = \text{newvar}() \\ code_1 = \text{Trans}_{Exp}(Exp, vtable, ftable, t) \\ code_1 ++ [\text{IF } t \neq 0 \text{ THEN } label_t \text{ ELSE } label_f] \end{array} \end{array} \end{array}$$

We need to convert the numerical value returned by Trans_{Exp} into a choice between two labels, so we generate an IF instruction that does just that.

The rule for relational operators is now actually superfluous, as the case it handles is covered by the second rule (since relational operators are assumed to be included

in the set of binary arithmetic operators in the intermediate language). However, we can consider it an optimisation, as the code it generates is shorter than the equivalent code generated by the second rule. It will also be natural to keep comparison as a special case when we add sequential logical operators.

6.6.1 Sequential Logical Operators

We will use the same names for sequential logical operators as C, i.e., `&&` for logical **and**, `||` for logical **or**, and `!` for logical negation. The extended language is shown in Grammar 6.9. Note that we allow an expression to be a condition as well as a condition to be an expression. This grammar is highly ambiguous (not least because **binop** overlaps **relop**). As before, we assume such ambiguity to be resolved by the parser before code generation. We also assume that the last productions of *Exp* and *Cond* are used when no other productions apply, as this will yield the best code.

The revised translation functions for *Exp* and *Cond* are shown in Fig. 6.10. Only the new cases for *Exp* are shown.

As expressions, `true` and `false` are the numbers 1 and 0.

A condition *Cond* is translated into code that chooses between two labels. When we want to use a condition as an expression, we must convert this choice into a number. We do this by first assuming that the condition is false by assigning 0 to the target location. We then, if the condition is true, jump to code that assigns 1 to the target location. If the condition is false, we jump around this code, so the value remains 0. We could equally well have done things the other way around, i.e., first assign 1 to the target location and modify this to 0 when the condition is false. Note

Grammar 6.9 Example language with logical operators

$$\begin{aligned}
 \textit{Exp} &\rightarrow \mathbf{num} \\
 \textit{Exp} &\rightarrow \mathbf{id} \\
 \textit{Exp} &\rightarrow \mathbf{unop} \textit{Exp} \\
 \textit{Exp} &\rightarrow \textit{Exp} \mathbf{binop} \textit{Exp} \\
 \textit{Exp} &\rightarrow \mathbf{id}(\textit{Exps}) \\
 \textit{Exp} &\rightarrow \mathbf{true} \\
 \textit{Exp} &\rightarrow \mathbf{false} \\
 \textit{Exp} &\rightarrow \textit{Cond} \\
 \\
 \textit{Exps} &\rightarrow \textit{Exp} \\
 \textit{Exps} &\rightarrow \textit{Exp} , \textit{Exps} \\
 \\
 \textit{Cond} &\rightarrow \textit{Exp} \mathbf{relop} \textit{Exp} \\
 \textit{Cond} &\rightarrow \mathbf{true} \\
 \textit{Cond} &\rightarrow \mathbf{false} \\
 \textit{Cond} &\rightarrow \mathbf{!} \textit{Cond} \\
 \textit{Cond} &\rightarrow \textit{Cond} \mathbf{\&\&} \textit{Cond} \\
 \textit{Cond} &\rightarrow \textit{Cond} \mathbf{||} \textit{Cond} \\
 \textit{Cond} &\rightarrow \textit{Exp}
 \end{aligned}$$

$Trans_{Exp}(Exp, vtable, ftable, place) = \text{case } Exp \text{ of}$	
⋮	
true	[<i>place</i> := 1]
false	[<i>place</i> := 0]
<i>Cond</i>	$label_1 = \text{newlabel}()$ $label_2 = \text{newlabel}()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ [<i>place</i> := 0] ++ $code_1$ ++ [LABEL $label_1$, <i>place</i> := 1] ++ [LABEL $label_2$]
$Trans_{Cond}(Cond, label_l, label_f, vtable, ftable) = \text{case } Cond \text{ of}$	
$Exp_1 \mathbf{relop} Exp_2$	$t_1 = \text{newvar}()$ $t_2 = \text{newvar}()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = \text{transop}(\text{getopname}(\mathbf{relop}))$ $code_1 ++ code_2 ++ [\text{IF } t_1 \text{ } op \text{ } t_2 \text{ THEN } label_l \text{ ELSE } label_f]$
true	[GOTO $label_l$]
false	[GOTO $label_f$]
! <i>Cond</i> ₁	$Trans_{Cond}(Cond_1, label_f, label_l, vtable, ftable)$
<i>Cond</i> ₁ && <i>Cond</i> ₂	$arg_2 = \text{newlabel}()$ $code_1 = Trans_{Cond}(Cond_1, arg_2, label_f, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_l, label_f, vtable, ftable)$ $code_1 ++ [\text{LABEL } arg_2] ++ code_2$
<i>Cond</i> ₁ <i>Cond</i> ₂	$arg_2 = \text{newlabel}()$ $code_1 = Trans_{Cond}(Cond_1, label_l, arg_2, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_l, label_f, vtable, ftable)$ $code_1 ++ [\text{LABEL } arg_2] ++ code_2$
<i>Exp</i>	$t = \text{newvar}()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1 ++ [\text{IF } t \neq 0 \text{ THEN } label_l \text{ ELSE } label_f]$

Fig. 6.10 Translation of sequential logical operators

that this code assigns to the *place* location before evaluating the condition, so it is important that *place* is not the name of a variable that might be used in the condition.

It gets a bit more interesting in $Trans_{Cond}$, where we translate conditions. We have already seen how comparisons and expressions are translated, so we move directly to the new cases.

The constant `true` condition just generates a jump to the label for true conditions, and, similarly, `false` generates a jump to the label for false conditions.

Logical negation generates no code by itself, it just swaps the attribute-labels for true and false when translating its argument. This effectively negates the argument condition.

Sequential logical **and** (`&&`) is translated as follows: The code for the first operand is translated such that if it is false, the second condition is not tested. This is done by jumping straight to the label for false conditions when the first operand is false. If the first operand is true, a jump to the code for the second operand is made. This is handled by using the appropriate labels as arguments to the call to *TransCond*. The call to *TransCond* for the second operand uses the original labels for true and false. Hence, both conditions have to be true for the combined condition to be true.

Sequential **or** (`||`) is similar: If the first operand is true, we jump directly to the label for true conditions without testing the second operand, but if it is false, we jump to the code for the second operand. Again, the second operand uses the original labels for true and false.

Note that the translation functions now work even if **binop** and **unop** do not contain relational operators or logical negation, as we can just choose the last rule for expressions whenever the **binop** rules do not match. However, we can not in the same way omit arithmetic (bitwise) **and** and **or**, as these always evaluate both arguments, which the sequential equivalents do not. Replacing an arithmetic logical operator with a sequential ditto may seem like an optimisation, but there is a visible difference in behaviour: If the second argument has side effects (for example function calls), it is observable whether or not this is evaluated. So the two types of logical operators are not interchangeable.

We have, in the above, used two different nonterminals for conditions and expressions, with some overlap between these and consequently ambiguity in the grammar. It is possible to resolve this ambiguity by rewriting the grammar and get two non-overlapping syntactic categories in the abstract syntax. Another solution is to join the two nonterminals into one, e.g., *Exp* and use two different translation functions for this nonterminal: Whenever an expression is translated, the translation function most appropriate for the context is chosen. For example, `if-then-else` will choose a translation function similar to *TransCond* while assignment will choose one similar to the current *TransExp*.

6.6.2 Example of Translation of Conditions

Using the rules in Figs. 6.5 and 6.10 and a symbol table that binds `x` to `v0` and `y` to `v1`, the following statement

$$y := x < 0 \mid \mid !(x < 9)$$

gets translated into the code shown in Fig. 6.11.

Fig. 6.11 Example of translating conditions

```

t1 := 0
t2 := v0
t3 := 0
IF t2 < t3 THEN L1 ELSE L3
LABEL L3
t4 := v0
t5 := 9
IF t4 < t5 THEN L2 ELSE L1
LABEL L1
t1 := 1
LABEL L2
v1 := t1

```

Note that the ! operator doesn't generate code of its own, it just swaps the branches of the comparison between `t4` and `t5`.

Suggested exercises: 6.3, 6.4.

6.7 Advanced Control Statements

We have, so far, shown translation of simple conditional statements and loops, but some languages have more advanced control features. We will briefly discuss how such can be implemented, without showing actual translation functions.

Goto and Labels

Source-code labels are stored in a symbol table that binds each source-code label to a corresponding label in the intermediate language. A jump to a label will generate a `GOTO` statement to the corresponding intermediate-language label. Unless labels are declared before use, an extra pass may be needed to build the symbol table before the actual translation. Alternatively, an intermediate-language label can be chosen, and an entry in the symbol table be created at the first occurrence of the label even if it is in a jump rather than a declaration. Subsequent jumps or declarations of that label will use the intermediate-language label that was chosen at the first occurrence. By setting a mark in the symbol-table entry when the label is declared, it can be checked that all labels are declared exactly once. This check ought to have been done during the type-checking phase (see Chap. 5), though.

The scope of labels can be controlled by the symbol table, so labels can be local to a procedure or block.

Break, Exit and Continue

Some languages allow exiting loops from the middle of the loop-body by a `break` or `exit` statement. To handle these, the translation function for statements must have an extra inherited attribute which is the label to which a `break` or `exit` statement

must jump. This attribute is changed whenever a new loop is entered. Before the first loop is entered, this attribute is undefined. The translation function should check for this, so it can report an error if a `break` or `exit` occurs outside loops. This should, rightly, be done during type-checking, though.

C's `continue` statement, which jumps to the start of the current loop, can be handled similarly: Using an inherited attribute to set the label to which a `continue` statement should jump.

Case-Statements

A `case`-statement evaluates an expression and selects one of several branches (statements) based on the value of the expression. In most languages, the `case`-statement will be exited at the end of each of these statements. In this case, the `case`-statement can be translated as an assignment that stores the value of the expression followed by a nested `if-then-else` statement, where each branch of the `case`-statement becomes a `then`-branch of one of the `if-then-else` statements (or, in case of the default branch, the final `else`-branch).

In C, the default is that *all* `case`-branches following the selected branch are executed, unless the `case`-expression (called `switch` in C) is explicitly terminated with a `break` statement (see above) at the end of the branch. In this case, the `case`-statement can still be translated to a nested `if-then-else`, but the branches of these are now `GOTO`'s to the code for each `case`-branch. The code for the branches are placed in sequence after the nested `if-then-else`, with `break` handled by `GOTO`'s as described above. Hence, if no explicit jump is made, one branch will fall through to the next.

We will look at translation of a variant of case statements with pattern matching in Sect. 12.3.

6.8 Translating Structured Data

So far, the only values we have used are integers and booleans. However, most programming languages provide floating-point numbers and structured values like arrays, records (structs), unions, lists or tree-structures. We will now look at how these can be translated. We will first look at floats, then at one-dimensional arrays, multi-dimensional arrays and finally other data structures.

6.8.1 *Floating-Point Values*

Floating-point values are, in a computer, typically stored in a different set of registers than integers. Apart from this, they are treated the same way we treat integer values: We use temporary variables to store intermediate expression results and assume

the intermediate language has arithmetic operators for floating-point numbers. The register allocator will have to make sure that the temporary variables used for floating-point values are mapped to floating-point registers. For this reason, it may be a good idea to let the intermediate code indicate which temporary variables hold floats. This can be done by giving them special names or by using a symbol table to hold type information.

6.8.2 Arrays

We extend our example language with one-dimensional arrays by adding the following productions:

$$\begin{aligned} \textit{Exp} &\rightarrow \textit{Indexed} \\ \textit{Stat} &\rightarrow \textit{Indexed} := \textit{Exp} \\ \textit{Indexed} &\rightarrow \mathbf{id}[\textit{Exp}] \end{aligned}$$

Indexed is an array element, which can be used the same way as a variable, either as an expression or as the left part of an assignment statement.

We will initially assume that arrays are zero-based (i.e., the lowest index is 0).

Arrays can be allocated statically, i.e., at compile-time, or *dynamically*, i.e., at run-time. In the first case, the *base address* of the array (the address at which index 0 is stored) is a compile-time constant. In the latter case, a variable will contain the base address of the array. In either case, we assume that the symbol table for variables binds an array name to the constant or variable that holds its base address.

Most modern computers are byte-addressed, while integers typically are 32 or 64 bits long. This means that the index used to access array elements must be multiplied by the size of the elements (measured in bytes), e.g., 4 or 8, to find the actual offset from the base address. In the translation shown in Fig. 6.12, we use 8 for the size of integers. We show only the new parts of the translation functions for *Exp* and *Stat*.

We use a translation function $Trans_{\textit{Indexed}}$ for array elements. This returns a pair consisting of the code that evaluates the address of the array element and the variable that holds this address. When an array element is used in an expression, the contents of the address is transferred to the target variable using a memory-load instruction. When an array element is used on the left-hand side of an assignment, the right-hand side is evaluated, and the value of this is stored at the address using a memory-store instruction.

The address of an array element is calculated by multiplying the index by the size of the elements (here, 8) and adding this to the base address of the array. Note that *base* can be either a variable or a constant (depending on how the array is allocated, see below), but since both are allowed as the second operator to a **binop** in the intermediate language, this is no problem.

As an example, the assignment $a[x] := a[y]$ is, given a symbol table that binds *a*, *x*, and *y* to $\forall 1$, $\forall 2$, and $\forall 1$, respectively, translated into the code

$$\begin{array}{l}
\text{Trans}_{Exp}(Exp, vtable, ftable, place) = \text{case } Exp \text{ of} \\
\hline
Indexed \quad \left| \begin{array}{l}
(code_1, address) = \text{Trans}_{Indexed}(Indexed, vtable, ftable) \\
code_1 ++ [place := M[address]]
\end{array} \right. \\
\\
\text{Trans}_{Stat}(Stat, vtable, ftable) = \text{case } Stat \text{ of} \\
\hline
Indexed := Exp \quad \left| \begin{array}{l}
(code_1, address) = \text{Trans}_{Indexed}(Indexed, vtable, ftable) \\
t = \text{newvar}() \\
code_2 = \text{Trans}_{Exp}(Exp, vtable, ftable, t) \\
code_1 ++ code_2 ++ [M[address] := t]
\end{array} \right. \\
\\
\text{Trans}_{Indexed}(Indexed, vtable, ftable) = \text{case } Indexed \text{ of} \\
\hline
id[Exp] \quad \left| \begin{array}{l}
base = \text{lookup}(vtable, \text{getname}(id)) \\
t = \text{newvar}() \\
code_1 = \text{Trans}_{Exp}(Exp, vtable, ftable, t) \\
code_2 = code_1 ++ [t := t * 8, t := t + base] \\
(code_2, t)
\end{array} \right.
\end{array}$$

Fig. 6.12 Translation for one-dimensional arrays

```

t2 := v2
t2 := t2 * 8
t2 := t2 + v1
t3 := v3
t3 := t3 * 8
t3 := t3 + v1
t1 := M[t3]
M[t2] := t1

```

6.8.2.1 Allocating Arrays

So far, we have only hinted at how arrays are allocated. As mentioned, one possibility is static allocation, where the base-address and the size of the array are known at compile-time. The compiler, typically, has a large address space where it can allocate statically allocated objects. When it does so, the new object is simply allocated after the end of the previously allocated objects.

Dynamic allocation can be done in several ways. One is allocation local to a procedure or function, such that the array is allocated when the function is entered and deallocated when it is exited. This typically means that the array is allocated on a stack and popped from the stack when the procedure is exited. If the sizes of locally allocated arrays are fixed at compile-time, their base addresses are constant offsets from the stack top (or from the *frame pointer*, see Chap. 9) and can be calculated by adding the constant offset to the stack (or frame) pointer at every array-lookup, or once only at the entry of the function and then stored in a local variable. If the sizes

Fig. 6.13 A two-dimensional array

	1st column	2nd column	3rd column	...
1st row	a [0] [0]	a [0] [1]	a [0] [2]	...
2nd row	a [1] [0]	a [1] [1]	a [1] [2]	...
3rd row	a [2] [0]	a [2] [1]	a [2] [2]	...
⋮	⋮	⋮	⋮	⋮

of these arrays are given at run-time, the offset from the stack or frame pointer to an array is not constant. So we need to use a variable to hold the base address of each array. The base address is calculated when the array is allocated and then stored in a local variable. This can subsequently be used as described in *Trans_{Indexed}* above. At compile-time, the array-name will, in the symbol table, be bound to the variable that at runtime will hold the base-address.

Dynamic allocation can also be done globally, so the array will survive until the end of the program or until it is explicitly deallocated. In this case, there must be a global address space available for run-time allocation. Often, this is handled by the operating system which handles memory-allocation requests from all programs that are running at any given time. Such allocation may fail due to lack of memory, in which case the program must terminate with an error or release memory enough elsewhere to make room. The allocation can also be controlled by the program itself, which initially asks the operating system for a large amount of memory and then administrates this itself. This can make allocation of arrays faster than if an operating system call is needed every time an array is allocated. Furthermore, it can allow the program to use *garbage collection* to automatically reclaim the space used for arrays that are no longer accessible. We will return to garbage collection in Sect. 12.2.

6.8.2.2 Multi-dimensional Arrays

Multi-dimensional arrays can be laid out in memory in two ways: *row-major* and *column-major*. The difference is best illustrated by two-dimensional arrays, as shown in Fig. 6.13. A two-dimensional array is addressed by two indices, e.g., (using C-style notation) as $a[i][j]$. The first index, i , indicates the *row* of the element and the second index, j , indicates the *column*. The first row of the array is, hence, the elements $a[0][0]$, $a[0][1]$, $a[0][2]$, ... and the first column is $a[0][0]$, $a[1][0]$, $a[2][0]$, ...¹

In row-major form, the array is laid out one row at a time and in column-major form it is laid out one column at a time. In a 3×2 array, the ordering for row-major is

$$a[0][0], a[0][1], a[1][0], a[1][1], a[2][0], a[2][1]$$

¹ Note that the coordinate system is rotated 90° clockwise compared to mathematical tradition.

For column-major the ordering is

$a[0][0], a[1][0], a[2][0], a[0][1], a[1][1], a[2][1]$

If the size of an element is *size* and the sizes of the dimensions in an *n*-dimensional array are $dim_0, dim_1, \dots, dim_{n-2}, dim_{n-1}$, then in row-major format an element at index $[i_0][i_1] \dots [i_{n-2}][i_{n-1}]$ has the address

$$base + ((\dots (i_0 * dim_1 + i_1) * dim_2 \dots + i_{n-2}) * dim_{n-1} + i_{n-1}) * size$$

In column-major format the address is

$$base + ((\dots (i_{n-1} * dim_{n-2} + i_{n-2}) * dim_{n-3} \dots + i_1) * dim_0 + i_0) * size$$

Note that column-major format corresponds to reversing the order of the indices of a row-major array. i.e., replacing i_0 and dim_0 by i_{n-1} and dim_{n-1} , i_1 and dim_1 by i_{n-2} and dim_{n-2} , and so on.

We extend the grammar for array-elements to accommodate multi-dimensional arrays:

$$\begin{aligned} Indexed &\rightarrow \mathbf{id}[Exp] \\ Indexed &\rightarrow Indexed[Exp] \end{aligned}$$

and extend the translation functions as shown in Fig. 6.14. This translation is for row-major arrays. We leave column-major arrays as an exercise.

With these extensions, the symbol table must return both the base-address of the array and a list of the sizes of the dimensions. Like the base-address, the dimension sizes can either be compile-time constants or variables that at run-time will hold the sizes. We use an auxiliary translation function $Calc_{Indexed}$ to calculate the position of an element. In $Trans_{Indexed}$ we multiply this position by the element size and add the base address. As before, we assume the size of elements is 8.

In some cases, the sizes of the dimensions of an array are not stored in separate variables, but in memory next to the space allocated for the elements of the array. This uses fewer variables (which may be an issue when these need to be allocated to registers, see Chap. 8) and makes it easier to return an array as the result of an expression or function, as only the base-address needs to be returned. The size information is normally stored just before the base-address so, for example, the size of the first dimension can be at address $base-8$, the size of the second dimension at $base-16$ and so on. Hence, the base-address will always point to the first element of the array no matter how many dimensions the array has. If this strategy is used, the necessary dimension sizes must be loaded into variables when an index is calculated. Since this adds several extra (somewhat costly) loads, optimising compilers often try to re-use the values of previous loads, e.g., by doing the loading once outside a loop and referring to variables holding the values inside the loop.

$$\begin{array}{l}
\text{Trans}_{Exp}(Exp, vtable, ftable, place) = \text{case } Exp \text{ of} \\
\hline
\text{Indexed} \left| \begin{array}{l}
(code_1, address) = \text{Trans}_{Indexed}(Indexed, vtable, ftable) \\
code_1 ++ [place := M[address]]
\end{array} \right. \\
\\
\text{Trans}_{Stat}(Stat, vtable, ftable) = \text{case } Stat \text{ of} \\
\hline
\text{Indexed} := Exp \left| \begin{array}{l}
(code_1, address) = \text{Trans}_{Indexed}(Indexed, vtable, ftable) \\
t = \text{newvar}() \\
code_2 = \text{Trans}_{Exp}(Exp_2, vtable, ftable, t) \\
code_1 ++ code_2 ++ [M[address] := t]
\end{array} \right. \\
\\
\text{Trans}_{Indexed}(Indexed, vtable, ftable) = \\
\hline
\begin{array}{l}
(code_1, t, base, []) = \text{Calc}_{Indexed}(Indexed, vtable, ftable) \\
code_2 = code_1 ++ [t := t * 8, t := t + base] \\
(code_2, t)
\end{array} \\
\\
\text{Calc}_{Indexed}(Indexed, vtable, ftable) = \text{case } Indexed \text{ of} \\
\hline
\text{id}[Exp] \left| \begin{array}{l}
(base, dims) = \text{lookup}(vtable, \text{getname}(\text{id})) \\
t = \text{newvar}() \\
code = \text{Trans}_{Exp}(Exp, vtable, ftable, t) \\
(code, t, base, \text{tail}(dims))
\end{array} \right. \\
\hline
\text{Indexed}[Exp] \left| \begin{array}{l}
(code_1, t_1, base, dims) = \text{Calc}_{Indexed}(Indexed, vtable, ftable) \\
dim_1 = \text{head}(dims) \\
t_2 = \text{newvar}() \\
code_2 = \text{Trans}_{Exp}(Exp, vtable, ftable, t_2) \\
code_3 = code_1 ++ code_2 ++ [t_1 := t_1 * dim_1, t_1 := t_1 + t_2] \\
(code_3, t_1, base, \text{tail}(dims))
\end{array} \right.
\end{array}$$

Fig. 6.14 Translation of multi-dimensional arrays

6.8.2.3 Index Checks

The translations shown so far do not test if an index is within the bounds of the array. Index checks are fairly easy to generate: Each index must be compared to the size of (the dimension of) the array and if the index is too big, a jump to some error-producing code is made. If the comparison is made on unsigned numbers, a negative index will look like a very large index. Hence, a single conditional jump using unsigned comparison is inserted at every index calculation.

This is still fairly expensive, but various methods can be used to eliminate some of these tests. For example, if the array-lookup occurs within a `for`-loop, the bounds of the loop-counter may guarantee that array accesses using this variable will be within bounds. More generally, it is possible to make an analysis that finds cases where the index-check condition is subsumed by previous tests, such as the exit test for a loop, the test in an `if-then-else` statement or previous index checks. We will return to this in Chap. 10.

6.8.2.4 Non-zero-based Arrays

We have assumed our arrays to be zero-based, i.e., that the indices start from 0. Some languages (such as Pascal) allow indices to be arbitrary intervals, e.g., -10 to 10 or 10 to 20 . If such are used, the starting-index must be subtracted from each index when the address is calculated. In a one-dimensional array with known size and base-address, the starting-index can be subtracted (at compile-time) from the base-address instead. In a multi-dimensional array with known dimensions, the starting-indices can be multiplied by the sizes of the dimensions and added together to form a single constant that is subtracted from the base-address, instead of subtracting each starting-index from each index. Even if the bounds are not known at compile time, a single offset can be calculated when the array is allocated.

6.8.3 Strings

Strings are often implemented in a fashion similar to one-dimensional arrays. In some languages (e.g. C or pre-ISO-standard Pascal), strings *are* just arrays of characters. This assumes a character is of fixed size, typically one byte. In some languages, such as Haskell, a string is a list of characters.

However, strings often differ from arrays in various ways:

- Different characters may have different size. For example, in UTF-8 encoding, characters in the ASCII subset are represented as one byte, and other characters as up to four bytes.
- Operations such as concatenating strings or extracting substrings are more common than similar operations on arrays.
- Many languages support lexicographic (alphabetic) comparison of strings, but not of arrays. Lexicographic ordering may even need to cater for local rules, e.g., where non-ASCII letters are placed in the alphabet and how certain ligatures (such as œ) are alphabetized.

So strings are often represented unlike arrays, both to cater for non-constant element size and to optimise operations such as concatenation and substring extraction. The representation can be a binary tree where leaf nodes contain single characters or short strings, and inner nodes store pointers to two subtrees as well as the size of the string stored in the left subtree. This allows relatively fast indexing and very fast concatenation.

Regardless of representation, operations on strings, such concatenation, comparison, and substring extraction, are typically implemented by library functions.

6.8.4 Records/Structs and Unions

Records (structs) have many properties in common with arrays. They are typically allocated in a similar way (with a similar choice of possible allocation strategies), and the fields of a record are typically accessed by adding an offset to the base-address of the record. The differences are:

- The types (and hence sizes) of the fields may be different.
- The field-selector is known at compile-time, so the offset from the base address can be calculated at this time.

The offset for a field is simply the sum of the sizes of all fields that occur before it. For a record-variable, the symbol table for variables must hold both the type and the base-address of the record. The symbol table for types must for a record type hold the types and offsets for each field in the record type. When generating code for a record field access, the compiler uses the symbol table for variables to find the base address and the type, which is used with the symbol table for types to find the field offset. Alternatively, the symbol table for variables can hold all this information.

In a union (sum) type, the fields are not consecutive, but are stored at the same address, i.e., the base-address of the union. The size of a union is the maximum of the sizes of its fields.

In some languages, union types include a *tag*, which identifies which variant of the union is stored in the variable. This tag is stored as a separate field before the union-fields. Some languages (e.g., Standard ML) enforce that the tag is tested when the union is accessed, others (e.g., Pascal) leave this as an option to the programmer.

Suggested exercises: 6.8.

6.9 Translation of Declarations

In the translation functions used in this chapter, we have several times required that “The symbol table must contain ...”. It is the job of the compiler to ensure that the symbol tables contain the information necessary for translation. When a name (variable, label, type, etc.) is declared, the compiler must, in the symbol-table entry for that name, keep the information necessary for compiling any use of that name. For scalar variables (e.g., integers), the required information is the intermediate-language variable that holds the value of the variable. For array variables, the information includes the base-address and dimensions of the array. For records, it is the offsets for each field and the total size. If a type is given a name, the symbol table must for that name provide a description of the type, such that variables that are declared to be that type can be given the information they need for their own symbol-table entries.

The exact nature of the information that is put into the symbol tables, and how this information is split among the symbol tables for types and the symbol table for

variables or functions, will depend on the translation functions that use these tables, so it is usually a good idea to write first the translation functions for *uses* of names and then translation functions for their declarations.

6.9.1 Simple Local Declarations

We extend the statement language by the following productions:

$$\begin{aligned} Stat &\rightarrow Decl ; Stat \\ Decl &\rightarrow \text{int } \mathbf{id} \\ Decl &\rightarrow \text{int } \mathbf{id}[\mathbf{num}] \end{aligned}$$

We can, hence, declare integer variables and one-dimensional integer arrays for use in the following statement. An integer variable should be bound to a location in the symbol table, so this declaration should add such a binding to *vtable*. An array should be bound to a variable containing its base address. Furthermore, code must be generated for allocating space for the array. We assume arrays are heap allocated and that the intermediate-code variable *HP* points to the first free element of the (upwards growing) heap. Figure 6.15 shows the translation of these declarations using the simplifying assumption that there is enough space in the heap. A real compiler would need to insert code to check this, and take appropriate action if there is not enough space. See Sect. 12.2 shows more detail about heap allocation.

$$\begin{array}{l} \hline Trans_{Stat}(Stat, vtable, ftable) = \text{case } Stat \text{ of} \\ Decl ; Stat_1 \quad \left| \begin{array}{l} (code_1, vtable_1) = Trans_{Decl}(Decl, vtable) \\ code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable) \\ code_1 ++ code_2 \end{array} \right. \\ \hline \\ Trans_{Decl}(Decl, vtable) = \text{case } Decl \text{ of} \\ \hline \text{int } \mathbf{id} \quad \left| \begin{array}{l} t_1 = \text{newvar}() \\ vtable_1 = \text{bind}(vtable, \text{getname}(\mathbf{id}), t_1) \\ ([], vtable_1) \end{array} \right. \\ \hline \text{int } \mathbf{id}[\mathbf{num}] \quad \left| \begin{array}{l} t_1 = \text{newvar}() \\ vtable_1 = \text{bind}(vtable, \text{getname}(\mathbf{id}), t_1) \\ ([t_1 := HP, HP := HP + (8 * \text{getvalue}(\mathbf{num}))], vtable_1) \end{array} \right. \\ \hline \end{array}$$

Fig. 6.15 Translation of simple declarations

6.9.2 Translation of Function Declarations

Given that the intermediate language includes function declarations, translating simple function definitions is quite easy: We translate a function declaration just by mapping the function and argument names to intermediate-language names in *vtable* and *ftable*, make a function header using the new names and then translating the body statement or expression using *vtable* and *ftable* as symbol tables. If the body is a statement, we just extend $Trans_{Stat}$ to translate a `return` statement into a `RETURN` instruction. If the body is an expression, we translate this and add a `RETURN` instruction to return its value. Local variable declarations are translated like the local declarations above.

As an example, the function definition

```

fac(n)
  f := 1;
  repeat
    f := f * n;
    n := n - 1
  until n = 0
  return f

```

is translated into the code shown in Fig. 6.16, using a *vtable* that binds `f` to `v0` and `n` to `v1`, and an *ftable* that binds `fac` to `_fac`. Note that the body of the function is very similar to the statement in Sect. 6.5.

Fig. 6.16 Example function definition translation

```

_fac (v1)
  t1 := 1
  v0 := t1
  LABEL L1
    t3 := v0
    t4 := v1
    v0 := t3 * t4
    t5 := v1
    t6 := 1
    v1 := t5 - t6
    t7 := v1
    t8 := 0
    IF t7 = t8 THEN L2 ELSE L1
  LABEL L2
  RETURN v0

```


If functions can call functions that are declared later, we use two passes: One to build *fable* and another to translate the function definitions using this *fable*. This is similar to how we in Chap. 5 type-checked mutually recursive function definitions.

At some later point, we will need to expand the intermediate-level function definitions, calls and returns into lower-level code. We will return to this in Chap. 9.

Suggested exercises: 6.13.

6.10 Further Reading

A comprehensive discussion about intermediate languages can be found in [6].

Functional and logic languages often use high-level intermediate languages, which are in many cases translated to lower-level intermediate code before emitting actual machine code. Examples of such intermediate languages can be found in [2, 3] and [1].

A well-known high-level intermediate language is the Java Virtual Machine [5], abbreviated JVM. This language evaluates expressions using a stack instead of temporary variables, and has single instructions for such complex things as virtual method calls and creating new objects. The high-level nature of JVM was chosen for several reasons:

- By letting common complex operations be done by single instructions, the code is smaller, which reduces transmission time when sending the code over the Internet.
- JVM was originally intended for interpretation, and doing more work in a single instruction also helps reduce the overhead of interpretation.
- A program in JVM is *validated* (essentially type-checked) before interpretation or further translation. This is easier when the code is high-level.

The Java Virtual Machine has been criticised for making too many assumptions about the source language, which makes it difficult to use for languages that are dissimilar to Java. Since JVM was designed specifically for Java, this is not surprising. A less language-specific intermediate language is The Low-Level Virtual Machine [4], abbreviated LLVM. Where JVM uses a stack for temporary values, LLVM (like the intermediate language used in this chapter) uses temporary variables, and it uses *static single assignment form* (SSA), which we will look at in Sect. 10.11.

6.11 Exercises

Exercise 6.1 Use the translation functions in Fig. 6.3 to generate code for the expression $2 + g(x + y, x * y)$. Use a *vtable* that binds x to $v0$ and y to $v1$ and an *fable* that binds g to $_g$. The result of the expression should be put in the intermediate-code variable r (so the *place* attribute in the initial call to $Trans_{Exp}$ is r).

Exercise 6.2 Use the translation functions in Figs. 6.5 and 6.6 to generate code for the statement

```
x:=2+y;
if x<y then x:=x+y;
repeat
  y:=y*2;
  while x>10 do x:=x/2
until x<y
```

use the same *vtable* as in Exercise 6.1.

Exercise 6.3 Use the translation functions in Figs. 6.5 and 6.10 to translate the following statement

```
if x<=y && !(x=y || x=1)
then x:=3
else x:=5
```

use the same *vtable* as in Exercise 6.1.

Exercise 6.4 De Morgan's law tells us that $!(p \parallel q)$ is equivalent to $(!p) \&\&(!q)$. Show that the two conditions generate identical code when compiled with *Trans_{Cond}* from Fig. 6.10.

Exercise 6.5 Show that, in any code generated by the functions in Figs. 6.5 and 6.10, every IF-THEN-ELSE instruction will be followed by one of the target labels.

Exercise 6.6 Extend Fig. 6.5 to include a `break`-statement for exiting loops, as described in Sect. 6.7, i.e., extend the statement syntax by

$$Stat \rightarrow \text{break}$$

and add a rule for this to *Trans_{Stat}*. Add whatever extra attributes you may need to do this.

Exercise 6.7 We extend the statement language with the following statements:

$$Stat \rightarrow \text{labelid} :$$

$$Stat \rightarrow \text{goto labelid}$$

for defining and jumping to labels.

Extend Fig. 6.5 to handle these as described in Sect. 6.7. Labels have scope over the entire program (statement) and need not be defined before use. You can assume that there is exactly one definition for each used label.

Exercise 6.8 Show translation functions for multi-dimensional arrays in column-major format. **Hint:** Starting from Fig. 6.14, it may be a good idea to rewrite the productions for *Index* so they are right-recursive instead of left-recursive, as the address formula for column-major arrays groups to the right. Similarly, it is a good idea to reverse the list of dimension sizes, so the size of the rightmost dimension comes first in the list.

Exercise 6.9 When statements are translated using the functions in Fig. 6.5, it will often be the case that the statement immediately following a label is a GOTO statement, i.e., we have the following situation:

```
LABEL label1
      GOTO label2
```

It is clear that any jump to *label₁* can be replaced by a jump to *label₂*, and that this will result in faster code. Hence, it is desirable to do so. This is called jump-to-jump optimisation, and can be done after code-generation by a post-process that looks for these situations. However, it is also possible to avoid most of these situations by modifying the translation function.

This can be done by adding an extra inherited attribute *endlabel*, which holds the name of a label that can be used as the target of a jump to the end of the code that is being translated. If the code is immediately followed by a GOTO statement, *endlabel* will hold the target of this GOTO rather than a label immediately preceding this.

- (a) Add the *endlabel* attribute to *Trans_{Stat}* from Fig. 6.5 and modify the rules so *endlabel* is exploited for jump-to-jump optimisation. Remember to set *endlabel* correctly in recursive calls to *Trans_{Stat}*.
- (b) Use the modified *Trans_{Stat}* to translate the following statement:

```
while x>0 do
  x := x-1;
  if x>10 then x := x/2
```

The extent of the while loop is indicated by indentation.

Use the same *vtable* as Exercise 6.1 and use *endlab* as the *endlabel* for the whole statement.

Exercise 6.10 In Fig. 6.5, *while* statements are translated in such a way that every iteration of the loop executes an unconditional jump (GOTO) in addition to the conditional jumps in the loop condition.

Modify the translation so each iteration only executes the conditional jumps in the loop condition, i.e., so an unconditional jump is saved in every iteration. You may have to add an unconditional jump outside the loop.

Exercise 6.11 In mathematics, logical conjunction is associative:

$$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$$

Show that this also applies to the sequential conjunction operator $\&\&$ when translated as in Fig. 6.10, i.e., that $p \ \&\& \ (q \ \&\& \ r)$ generates the same code (up to renaming of labels) as $(p \ \&\& \ q) \ \&\& \ r$.

Exercise 6.12 Figure 6.14 shows translation of multi-dimensional arrays in row-major layout, where the address of each element is found through multiplication and addition. On machines with fast memory access but slow multiplication, an alternative implementation of multi-dimensional arrays is sometimes used: An array with dimensions $dim_0, dim_1, \dots, dim_n$ is implemented as a one-dimensional array of size dim_0 with pointers to dim_0 different arrays each of dimension dim_1, \dots, dim_n , which again are implemented in the same way (until the last dimension, which is implemented as a normal one-dimensional array of values). This takes up more room, as the pointer arrays need to be stored as well as the elements. But array-lookup can be done using only addition and memory accesses.

- Assuming pointers and array elements need eight bytes each, what is the total number of bytes required to store an array of dimensions $dim_0, dim_1, \dots, dim_n$?
- Write translation functions for array-access in the style of Fig. 6.14 using this representation of arrays. Use addition to multiply numbers by 8 for scaling indices by the size of pointers and array elements.

Exercise 6.13 We add function declarations and function return to the example language by adding the productions

$$\begin{aligned} FunDec &\rightarrow \mathbf{id} \ (\ Params \) \ Stat \\ Params &\rightarrow \mathbf{id} \\ Params &\rightarrow \mathbf{id} \ , \ Params \\ Stat &\rightarrow \mathbf{return} \ Exp \end{aligned}$$

Using the informal explanation in Sect. 6.9.2, extend $Trans_{Stat}$ and write translation functions $Trans_{FunDec}$ and $Trans_{Params}$ to implement these extensions. You can assume that you already have a *fable* that maps source-level function names to intermediate-code function names, so this can be used as inherited attribute by $Trans_{FunDec}$. You can also assume that there are no repeated parameter names, as this would have been detected by a type checker.

References

- Ait-Kaci, H.: Warren's Abstract Machine – A Tutorial Reconstruction. MIT Press (1991)
- Appel, A.W.: Compiling with Continuations. Cambridge University Press (1992)
- Jones, S.L.P., Lester, D.: Implementing Functional Languages – A Tutorial. Prentice Hall (1992)
- Lattner, C.: LLVM language reference manual (2011). <http://llvm.org/docs/LangRef.html>
- Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading, Massachusetts (1999)
- Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)

Chapter 7

Machine-Code Generation



The machine does not isolate man from the great problems of nature but plunges him more deeply into them.

Antoine de Saint-Exupéry (1900–1944)

The intermediate language we have used in Chap. 6 is quite low-level and similar to the type of machine code you can find on modern RISC processors, with a few exceptions:

- We have used an unbounded number of variables, where a processor will have a bounded number of registers.
- We have used high-level instructions for function definitions, calls and return.
- In the intermediate language, the IF-THEN-ELSE instruction has two target labels, where, on most processors, the conditional jump instruction has only one target label, and simply falls through to the next instruction when the condition is false.
- We have assumed that any constant can be an operand to an arithmetic instruction. Typically, RISC processors allow only small constants as operands.

The problem of mapping a large set of variables to a small number of registers is handled by *register allocation*, as explained in Chap. 8. Functions are treated in Chap. 9. We will look at the remaining two problems below.

The simplest solution for generating machine code from intermediate code is to translate each intermediate-language instruction into one or more machine-code instructions. However, it is often possible to find a machine-code instruction that covers two or more intermediate-language instructions. We will in Sect. 7.3 see how we can exploit complex instructions in this way.

Additionally, we will briefly discuss other optimisations.

7.1 Conditional Jumps

Conditional jumps come in many forms on different machines. Some conditional jump instructions embody a relational comparison between two registers (or a register and a constant) and are, hence, similar to the IF-THEN-ELSE instruction in our intermediate language. Other types of conditional jump instructions require the condition to be already resolved and stored in special condition registers or flags. However, it is almost universal that conditional jump instructions specify only one target label (or address), typically used when the condition is true. When the condition is false, execution simply continues with the instructions immediately following the conditional jump instruction.

Converting two-way branches to one-way branches is not terribly difficult: IF c THEN l_t ELSE l_f can be translated to

```
branch_if_c  lt
jump        lf
```

where `branch_if_c` is a conditional instruction that jumps when the condition c is true and `jump` is an unconditional jump.

Often, an IF-THEN-ELSE instruction is immediately followed by one of its target labels. In fact, this will always be the case if the intermediate code is generated by the translation functions shown in Chap. 6 (see Exercise 6.5). If this label happens to be l_f (the label taken for false conditions), we can simply omit the unconditional `jump` from the code shown above. If the following label is l_t , we can negate the condition of the conditional jump and make it jump to l_f , i.e., as

```
branch_if_not_c  lf
```

where `branch_if_not_c` is a conditional instruction that jumps when the condition c is false.

Hence, the code generator (the part of the compiler that generates machine code) should see which (if any) of the target labels follow an IF-THEN-ELSE instruction and translate it accordingly. Alternatively, a post-processing pass can be made over the generated machine code to remove superfluous jumps.

Some instruction sets include comparison of two registers or a register and a small constant as part of a branch instruction. Some instruction sets (such MIPS) only support equality and inequality, so more complex comparisons have to be computed into a register, which is then compared to 0 in a subsequent branch instruction. For example, IF $t_1 < t_2$ THEN l_t ELSE l_f can be compiled to the sequence

```
slt  t3, t1, t2
bne  t3, lt
j    lf
```

where `slt` is short for “set less than” and t_3 is a temporary variable.

Some instruction sets, e.g., PowerPC, compute conditions into numbered one-bit condition registers, which can be tested in branch instructions.

Yet other instruction sets compute conditions as side effects of arithmetic instructions. An arithmetic or comparison instruction will set arithmetic flags such as Zero, Sign, and Overflow, which in specific combinations can be used to test for equality and signed or unsigned less-than or less-than-or-equal-to. This is the default for x86 and x86-64, Sparc, ARMv7 (32-bit), and ARMv8 (64-bit). Some of these (such as ARMv8) additionally have comparison of a register to 0 embedded in a branch instruction. On ARMv7, `IF $t_1 < t_2$ THEN l_t ELSE l_f` can be compiled to the sequence

```
CMP   $t_1$ ,  $t_2$ 
BLT   $l_t$ 
BAL   $l_f$ 
```

where `CMP t_1 , t_2` is an instruction that subtracts t_2 from t_1 but doesn't store the result anywhere, so its only effect is to set the condition codes. `BLT l_t` jumps to l_t if the sign flag (N) is different from the overflow flag (V), implementing a signed less-than comparison. `BAL` (branch always) is an unconditional jump.

Usually, any `IF-THEN-ELSE` instruction can be translated into at most two instructions: One that does the comparison, and one that does the conditional jump. If neither of the branch labels in the `IF-THEN-ELSE` directly follow this, an additional unconditional jump may be needed, as seen in the ARMv7 example above.

7.2 Constants

The intermediate language allows arbitrary constants as operands to binary or unary operators. This is not always the case in machine code.

For example, MIPS allows only 16-bit constants in operands even though integers are 32 bits (64 bits in some versions of the MIPS architecture). To build larger constants, MIPS includes instructions to load 16-bit constants into the upper half (the most significant bits) of a register. With help of these, an arbitrary 32-bit integer can be entered into a register using two instructions. On ARMv7, a constant can be an 8-bit number positioned at any even bit boundary. It may take up to four instructions to build a 32-bit number using these. On ARMv8 and RISC-V, a constant is usually a simple 12-bit constant, but a special instruction (load-upper-immediate) uses a 20-bit constant shifted by 12 bits, so any 32-bit value can be built in at most two instructions.

x86 and x86-64 support word-size constants by storing constants in words after the instruction word. Unlike the other mentioned architectures, x86 and x86-64 have instructions of varying sizes, from one byte to 15 bytes. Since we generate code in symbolic instruction format, we will not consider instruction size except as it concerns limitations on immediate constants.

When an intermediate-language instruction uses a constant, the code generator must check if the constant fits into the constant field (if any) of the equivalent machine-code instruction. If it does, the code generator generates a single machine-code

instruction. If not, the code generator generates a sequence of instructions that builds the constant in a register or memory location, followed by an instruction that uses this register or memory location in place of the constant. If a complex constant is used inside a loop, it may be a good idea to move the code for generating the constant outside the loop and keep it in a register inside the loop. This can be done as part of a general optimisation to move code out of loops, see Sect. 7.4 and Chap. 11.

7.3 Exploiting Complex Instructions

Most instructions in our intermediate language are *atomic*, in the sense that each instruction corresponds to a single operation which can not sensibly be split into smaller steps. The exceptions to this rule are the instructions `IF-THEN-ELSE`, which we in Sect. 7.1 described how to handle, and `CALL` and `RETURN`, which will be detailed in Chap. 9.

CISC (Complex Instruction Set Computer) processors like x86-64 have composite (i.e., non-atomic) instructions in abundance. And while the philosophy behind RISC (Reduced Instruction Set Computer) processors like RISC-V and ARM advocates that machine-code instructions should be simple, most RISC processors include at least a few non-atomic instructions, typically memory-access instructions.

We will in this chapter use a subset of the RISC-V instruction set as an example. A description of the RISC-V instruction set can be found online [4]. If you are not already familiar with the RISC-V instruction set, it would be a good idea to read the description before continuing.

To exploit composite instructions, several intermediate-language instructions can be grouped together and translated into a single machine-code instruction. For example, the intermediate-language instruction sequence

$$\begin{aligned} t_2 &:= t_1 + 116 \\ t_3 &:= M[t_2] \end{aligned}$$

can be translated into the single RISC-V instruction

```
lw x3, 116(x1)
```

where `x1` and `x3` are the registers chosen for t_1 and t_3 , respectively. We will, for now, not assign specific registers to intermediate-language variables, but do so in Chap. 8, so we will use assembly instructions with intermediate-language variables, so the instruction above will in this chapter be written

```
lw t3, 116(t1)
```


It is, however, only possible to combine the two instructions if the value of the intermediate variable t_2 is not required later, as the combined instruction does not store this value anywhere.

We will, hence, need to know if the contents of a variable is required for later use, or if the variable is *dead* after a particular use. When generating intermediate code, most of the temporary variables introduced by the compiler will be assigned and used exactly once, and can be marked as dead after this use. Alternatively, last-use information can be obtained by analysing the intermediate code using a *liveness analysis*, which we will describe in Chap. 8. For now, we will just assume that the last use of any variable is marked in the intermediate code by superscripting it with *last*, such as t^{last} , which indicates the last use of the variable t .

Our next step is to describe each machine-code instruction in terms of one or more intermediate-language instructions. We call the sequence of intermediate-language instructions a *pattern*, and the corresponding machine-code instruction its *replacement*, since the idea is to find sequences in the intermediate code that matches the pattern and replace these sequences by instances of the replacement. When a pattern uses variables such as k , t or r_d , these can match any intermediate-language constants, variables or labels, and when the same variable is used in both pattern and replacement, it means that the corresponding intermediate-language constant or variable/label name is copied to the machine-code instruction, where it will represent a constant, a named register or a machine-code label.

For example, the RISC-V `lw` (load word) instruction can be described by the pattern/replacement pair

$$\begin{array}{l} t := r_s + k \\ r_t := M[t^{last}] \end{array} \quad \Bigg| \quad \text{lw } r_t, k(r_s)$$

where t^{last} in the pattern indicates that the contents of t must not be used afterwards, i.e., that the intermediate-language variable that is matched against t must have a *last* annotation at this place. A pattern can only match a piece of intermediate code if all *last* annotations in the pattern are matched by *last* annotations in the intermediate code. The converse, however, need not hold: It is not harmful to store a value in a register even if it is not used later, so a *last* annotation in the intermediate code need not be matched by a *last* annotation in the pattern.

The list of patterns, that in combination describe the machine-code instruction set, must cover the intermediate language in full (excluding function calls, which we handle in Chap. 9). In particular, each single intermediate-language instruction (with the exception of `CALL` and `RETURN`, which we handle separately in Chap. 9) must be covered by at least one pattern. This means that we must include the RISC-V instruction `lw` $r_t, 0(r_s)$ to cover the intermediate-code instruction $r_t := M[r_s]$, even though we have already listed a more general form of `lw`. If there is an intermediate-language instruction for which there are no equivalent single

machine-code instruction, a sequence of machine-code instructions must be given for this. Hence, an instruction-set description is a list of pairs, where each pair consists of a *pattern* (a sequence of intermediate-language instructions) and a *replacement* (a sequence of machine-code instructions).

When translating a sequence of intermediate-code instructions, the code generator can look at the patterns and pick the pattern that covers the largest prefix of the intermediate code. A simple way of ensuring that the longest prefix is matched is to list the pairs such that longer patterns are listed before shorter patterns. The first pattern in the list that matches a prefix of the intermediate code will now also be the longest matching pattern.

This kind of algorithm is called *greedy*, because it always picks the choice that is best for immediate profit, i.e., the sequence that “eats” most of the intermediate code in one bite. It will, however, not always yield the best possible solution for the total sequence of intermediate-language instructions.

If costs are given for each machine-code instruction sequence in the pattern/replacement pairs, optimal (i.e., least-cost) solutions can be found for straight-line (i.e., jump-free) code sequences. The least-cost sequence that covers the intermediate code can be found, e.g., using a dynamic-programming algorithm. For RISC processors, a greedy algorithm will typically get close to optimal solutions, so the gain from using a better algorithm is small. Hence, we will go into detail only for the greedy algorithm.

As an example, Fig. 7.1 describes a subset of the instructions for the RISC-V microprocessor architecture as a set of pattern/replacement pairs. Note that we exploit the fact that register 0 (written as `x0`) in RISC-V is hardwired to be the value 0 to, e.g., use the `addi` instruction. Note, also, that the jump instruction `j label` is actually a pseudoinstruction that expands to `jal x0, label`.

We assume that we, at this point, have already handled the problem of too-large constants, so any constant that now remains in the intermediate code can be used as an immediate constant in an instruction such as `addi`. Note that we make special cases for `IF-THEN-ELSE` when one of the labels immediately follows the test. Note, also, that we need (at least) two instructions from our RISC-V subset to implement an `IF-THEN-ELSE` instruction that uses `<` as the relational operator, while we need only one for comparison by `=`. Figure 7.1 does not cover all of the intermediate language, but it can fairly easily be extended to do so. It is also possible to add more special cases to exploit a larger subset of the RISC-V instruction set.

The instructions in Fig. 7.1 are listed so that, when two patterns overlap, the longest of these is listed first. Overlap can happen if the pattern in one pair is a prefix of the pattern for another pair, as is the case with the pairs involving `addi` and `lw/sw` and for the different instances of `beq`, `bne`, `blt`, and `bge`.

We can try to use Fig. 7.1 to select RISC-V instructions for the following sequence of intermediate-language instructions:

$t := r_s + k,$ $r_t := M[t^{last}]$		lw	$r_t, k(r_s)$
$r_t := M[r_s]$		lw	$r_t, 0(r_s)$
$r_t := M[k]$		lw	$r_t, k(\times 0)$
$t := r_s + k,$ $M[t^{last}] := r_t$		sw	$r_t, k(r_s)$
$M[r_s] := r_t$		sw	$r_t, 0(r_s)$
$M[k] := r_t$		sw	$r_t, k(\times 0)$
$r_d := r_s + r_t$		add	r_d, r_s, r_t
$r_d := r_t$		add	$r_d, \times 0, r_t$
$r_d := r_s + k$		addi	r_d, r_s, k
$r_d := k$		addi	$r_d, \times 0, k$
GOTO <i>label</i>		j	<i>label</i>
IF $r_s = r_t$ THEN <i>label_t</i> ELSE <i>label_f</i> , LABEL <i>label_f</i>		beq	$r_s, r_t, label_t$ <i>label_f</i> :
IF $r_s = r_t$ THEN <i>label_t</i> ELSE <i>label_f</i> , LABEL <i>label_t</i>		bne	$r_s, r_t, label_f$ <i>label_t</i> :
IF $r_s = r_t$ THEN <i>label_t</i> ELSE <i>label_f</i>		beq	$r_s, r_t, label_t$ j <i>label_f</i>
IF $r_s < r_t$ THEN <i>label_t</i> ELSE <i>label_f</i> , LABEL <i>label_f</i>		blt	$r_s, r_t, label_t$ <i>label_f</i> :
IF $r_s < r_t$ THEN <i>label_t</i> ELSE <i>label_f</i> , LABEL <i>label_t</i>		bge	$r_t, r_s, label_f$ <i>label_f</i> :
IF $r_s < r_t$ THEN <i>label_t</i> ELSE <i>label_f</i>		blt	$r_s, r_t, label_t$ j <i>label_f</i>
LABEL <i>label</i>			<i>label</i> :

Fig. 7.1 Pattern/replacement pairs for a subset of the RISC-V instruction set

```

a := a + blast
d := c + 8
M[dlast] := a
IF a = c THEN label1 ELSE label2
LABEL label2

```

Only one pattern (for the `add` instruction) in Fig. 7.1 matches a prefix of this code, so we generate an `add` instruction for the first intermediate instruction. We now have two matches for prefixes of the remaining code: One using `sw` and one using `addi`.

Since the pattern using `sw` is listed first in the table, we choose this to replace the next two intermediate-language instructions. Finally, a `beq` instruction matches the last two instructions. Hence, we generate the code

```

    add a, a, b
    sw  a, 8(c)
    beq a, c, label1
label2 :

```

Note that we retain `label2` even though the resulting sequence does not refer to it, because some other part of the code might jump to it. We could include single-use annotations for labels like we use for variables, but it is hardly worth the effort, as labels do not generate actual code and hence cost nothing.¹

7.3.1 Two-Address Instructions

In the above we have assumed that the machine code is three-address code, i.e., that the destination register of an instruction can be distinct from the two operand registers. It is, however, not uncommon that processors use two-address code, where the destination register is the same as the first operand register. To handle this, we use pattern/replacement pairs like these:

$r_t := r_s$	<code>mov</code> r_t, r_s
$r_t := r_t + r_s$	<code>add</code> r_t, r_s
$r_d := r_s + r_t$	<code>mov</code> r_d, r_s
	<code>add</code> r_d, r_t

where `mov` r_t, r_s copies r_s into r_t to add copy instructions in the cases where the destination register is not the same as the first operand. As we will see in Chap. 8, the register allocator will often be able to remove the added copy instruction by allocating r_t and r_s in the same register.

Processors that divide registers into data and address registers or integer and floating-point registers can be handled in a similar way: Add instructions that copy the arguments of an operation to new registers before the instruction that does the operation, and let register allocation allocate these to the right kind of registers (and eliminate as many of the moves as possible).

Suggested exercises: 7.2.

¹ This is, strictly speaking, not entirely true, as superfluous labels might inhibit later optimisations.

7.4 Optimisations

Optimisations can be done by a compiler in three places: In the source code (i.e., on the abstract syntax), in the intermediate code, and in the machine code. Some optimisations can be specific to the source language or the machine language, but otherwise it makes sense to perform optimisations mainly in the intermediate language, as such optimisations can be shared among all compilers that use the same intermediate language. Also, the intermediate language is typically simpler than both the source language and the machine language, making the effort of doing optimisations smaller.

Optimising compilers have a wide array of optimisations that they can employ, but we will mention only a few and just hint at how they can be implemented. We will return to some of these in later chapters.

Common Subexpression Elimination

In the statement `a[i] := a[i]+2`, the address for `a[i]` is calculated twice. This double calculation can be eliminated by storing the address in a temporary variable when the address is first calculated, and then use this variable instead of calculating the address again. Simple methods for common subexpression elimination work on *basic blocks*, i.e., straight-line code without jumps or labels, but more advanced methods can eliminate duplicated calculations even across jumps. We will look more closely at common subexpression elimination in Chap. 10.

Code Hoisting

If part of the computation inside a loop is independent of the variables that change inside the loop, it can be moved outside the loop and only calculated once. For example, in the loop

```
while (j < k) {
    sum = sum + a[i][j];
    j++;
}
```

a large part of the address calculation for `a[i][j]` can be done without knowing `j`. This part can be moved outside the loop so it will only be calculated once. Note that this optimisation can not be done on source-code level, as the address calculations are not visible there.

If `k` may be less than or equal to `j`, the loop body may never be entered and we may, hence, unnecessarily execute the code that was moved out of the loop. This might even generate a run-time error. Hence, we can unroll the loop once to

```
if (j < k) {
    sum = sum + a[i][j];
```

```

    j++;
    while (j < k) {
        sum = sum + a[i][j];
        j++;
    }
}

```

The loop-independent part(s) may now without risk be calculated in the unrolled part and reused in the non-unrolled part. We look more closely at code hoisting in Chap. 11.

Constant Propagation

A variable may, at some points in the program, have a value that is always equal to a known constant. When such a variable is used in a calculation, this calculation can often be simplified after replacing the variable by the constant that is guaranteed to be its value. Furthermore, the variable that holds the results of this computation may now also become constant, which may enable even more compile-time reduction.

Constant-propagation algorithms first trace the flow of constant values through the program, and then perform or simplify calculations using these. More advanced methods also look at conditions, so they can exploit that after a test on, e.g., $x = 0$, x is, indeed, the constant 0.

Index-Check Elimination

As mentioned in Chap. 6, some compilers insert run-time checks to catch cases when an index is outside the bounds of the array. Some of these checks can be removed by the compiler. One way of doing this is to see if the tests on the index are subsumed by earlier tests or ensured by assignments. For example, assume that, in the loop shown above, a is declared to be a $k \times k$ array. This means that the entry test for the loop will ensure that j is always less than the upper bound on the array, so this part of the index test can be eliminated. If j is initialised to 0 before entering the loop, we can use this to conclude that we do not need to check the lower bound either. We look more closely at index-check elimination in Chap. 10.

7.5 Further Reading

Code selection by pattern matching normally uses a tree-structured intermediate language instead of the linear instruction sequences we use in this book. This can avoid some problems where the order of unrelated instructions affect the quality of code generation. For example, if the two first instructions in the example at the end of Sect. 7.3 are interchanged, our simple prefix-matching algorithm will not include the address calculation in the `sw` instruction and, hence, needs one more instruction.

If the intermediate code is tree-structured, the order of independent instructions is left unspecified, and the code generator can choose whichever ordering gives the best code. See [3] or [2] for more details.

Descriptions of and methods for implementation of a large number of different optimisations can be found in [1, 3] and [2]. See also Chap. 10.

The instruction set of the RISC-V microprocessor architecture is available online [4].

7.6 Exercises

Exercise 7.1 Add extra inherited attributes to $Trans_{Cond}$ in Fig. 6.10 that, for each of the two target labels, indicates if this label immediately follows the code for the condition, i.e., a Boolean-valued attribute for each of the two labels. Use this information to make sure that the false-destination labels of an IF-THEN-ELSE instruction follow immediately after the IF-THEN-ELSE instruction.

You can use the function *negate* to negate relational operators so, e.g., $negate(<) = \geq$.

Make sure the new attributes are maintained in recursive calls and modify $Trans_{Stat}$ in Fig. 6.5 so it sets these attributes when calling $Trans_{Cond}$.

Exercise 7.2 Use Fig. 7.1 and the method described in Sect. 7.3 to generate code for the following intermediate code sequence:

```

d := c + 8
a := a + blast
M[dlast] := a
IF a < c THEN label1 ELSE label2
LABEL label1

```

Compare this to the example in Sect. 7.3.

Exercise 7.3 In Figs. 6.3 and 6.5, identify guaranteed last-uses of temporary variables, i.e., places where *last* annotations can be inserted safely.

Exercise 7.4 Choose an instruction set (other than RISC-V) and make patterns for the same subset of the intermediate language as covered by Fig. 7.1. Use this to translate the intermediate-code example from Sect. 7.3.

Exercise 7.5 In some microprocessors, arithmetic instructions use only two registers, as the destination register is the same as one of the argument registers. As

an example, copy and addition instructions of a hypothetical such processor can be described as follows (using notation like in Fig. 7.1):

$r_d := r_t$	mov r_d, r_t
$r_d := r_d + r_t$	add r_d, r_t
$r_d := r_d + k$	addi r_d, k

As in RISC-V, register 0 ($x0$) is in this processor hardwired to the value 0.

Add to the above table sufficient extra pattern/replacement pairs to allow translation of the following intermediate-code instructions to sequences of machine-code instructions using only special cases of `mov`, `add` and `addi` instructions in the replacement sequences:

$$\begin{aligned}
 r_d &:= k \\
 r_d &:= r_s + r_t \\
 r_d &:= r_s + k
 \end{aligned}$$

Note that neither r_s nor r_t have the *last* annotation, so their values must be preserved. Note, also, that the intermediate-code instructions above are not a sequence, but a list of separate instructions, so you should generate code separately for each instruction.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers; Principles. Addison-Wesley, Techniques and Tools* (2007)
2. Appel, A.W.: *Modern Compiler Implementation in ML. Cambridge University Press* (1998)
3. Muchnick, S.S.: *Advanced Compiler Design and Implementation. Morgan Kaufmann* (1997)
4. Waterman, A., Asanović, K.: *The RISC-V instruction set manual, vol. i: User-level isa, document version 2.2* (2017). <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

Chapter 8

Register Allocation



Just in terms of allocation of time resources, religion is not very efficient. There's a lot more I could be doing on a Sunday morning.

Bill Gates (1955–)

When generating intermediate code in Chap. 6, we have freely used as many variables as we found convenient. In Chap. 7, we have simply translated variables in the intermediate language one-to-one into registers in the machine language. Processors, however, do not have an unlimited number of registers, so we need *register allocation* to handle this conflict. The purpose of register allocation is to map a large number of variables into a small(ish) number of registers. This can often be done by letting several variables share a single register, but sometimes there are simply not enough registers in the processor. In this case, some of the variables must be temporarily stored in memory. This is called *spilling*.

Register allocation can be done in the intermediate language prior to machine-code generation, or it can be done in the machine language. In the latter case, the machine code initially uses symbolic names for registers, which the register allocation turns into register numbers. Doing register allocation in the intermediate language has the advantage that the same register allocator can easily be used for several target machines (it just needs to be parameterised with the set of available registers).

However, there may be advantages to postponing register allocation to after machine code has been generated. In Chap. 7, we saw that several instructions may be combined to a single instruction, and in the process a variable may disappear. There is no need to allocate a register to this variable, but if we do register allocation in the intermediate language, we will do so. Furthermore, when an intermediate-language instruction needs to be translated into a sequence of machine-code instructions, the machine code may need extra registers for storing temporary values, such as the register needed to store the result of the `slt` instruction when translating a jump on the `<` condition to MIPS code. Hence, the register allocator must make sure that

there are enough spare registers for temporary storage when expanding instructions. Usually, this is only one or two registers, but on a processor with a small number of registers, this can be significant.

The techniques used for register allocation are more or less the same regardless of whether register allocation is done on intermediate code or on machine code. So, in this chapter, we will describe register allocation in terms of the intermediate language introduced in Chap. 6, with the understanding that register allocation can also be done later.

As in Chap. 6, we operate on the body of a single procedure or function, so when we below use the word “program”, we mean it to be such a body. In Chap. 9, we will look at how to handle programs consisting of several functions that can call each other.

8.1 Liveness

In order to answer the question “When can two variables share a register?”, we must first define the concept of *liveness*:

Definition 8.1 A variable is *live* at some point in the program if the value it contains at that point might conceivably be used in future computations. Conversely, it is *dead* if there is no way its value will be used in the future.

We have already hinted at this concept in Chap. 7, when we talked about last-uses of variables.

Intuitively, two variables may at any given program point share a register if they are not both live. If we don’t want a variable to reside in different registers at different points, two variables can share a register only if there is *no* point in the program where they are both live. We will make a more precise definition later.

We can use some rules to determine when a variable is live:

- 1) If an instruction uses the contents of a variable, that variable is *live* at the start of that instruction.
- 2) If a variable is assigned a value in an instruction, and the same variable is not used as an operand in that instruction, then the variable is *dead* at the start of the instruction, as the value it has at this time is not used before it is overwritten.
- 3) If a variable is live at the end of an instruction, and that instruction does not assign a value to the variable, then the variable is also live at the start of the instruction.
- 4) A variable is live at the end of an instruction if it is live at the start of any of the immediately succeeding instructions.

Rule 1 tells how liveness is *generated*, rule 2 how liveness is *killed*, and rules 3 and 4 how liveness is *propagated*.

8.2 Liveness Analysis

We can formalise the above rules as equations over sets of variables. The process of solving these equations is called *liveness analysis*, and will at any given point in the program determine which variables are live at this point. To better speak of points in a program, we number all instructions in a procedure as shown in Fig. 8.1, which will be our running example in this chapter. The program is a function that calculates the N th Fibonacci number.

For every instruction in the program, we have a set of *successors*, i.e., instructions that may immediately follow the instruction during execution. We denote the set of successors to the instruction numbered i as $\text{succ}[i]$. We use the following rules to find $\text{succ}[i]$:

- 1) If the instruction numbered i is *not* a GOTO, IF-THEN-ELSE, or RETURN instruction, the instruction (if any) numbered j that is listed just after instruction number i is in $\text{succ}[i]$. In a well-formed program, there will always be an instruction after instruction i , and this will be numbered $i + 1$, so $\text{succ}[i] = \{i + 1\}$.
- 2) If instruction number i is of the form GOTO l , and there is an instruction j : LABEL l , then $j \in \text{succ}[i]$. In a well-formed program, there will be exactly one such LABEL instruction so $\text{succ}[i] = \{j\}$.
- 3) If instruction i is IF c THEN l_t ELSE l_f , and there are instructions j : LABEL l_t and k : LABEL l_f , j and k are in $\text{succ}[i]$. In a well-formed program, $\text{succ}[i] = \{j, k\}$.
- 4) If instruction i is of the form RETURN x , $\text{succ}[i] = \emptyset$.

Note that we assume that both outcomes of an IF-THEN-ELSE instruction are possible, so both possible destinations can succeed the instruction. If this happens not to be the case (i.e., if the condition is always true or always false), our liveness

Fig. 8.1 Example program for liveness analysis and register allocation

```

    fb(n) [
1: a := 0
2: b := 1
3: z := 0
4: LABEL loop
5: IF n = z THEN end ELSE body
6: LABEL body
7: t := a + b
8: a := b
9: b := t
10: n := n - 1
11: z := 0
12: GOTO loop
13: LABEL end
14: RETURN a
    ]

```

analysis may claim that a variable is live when it is in fact dead. This is no major problem, as the worst that can happen is that we use a register to keep the value of a variable that, after all, is not going to be used. The converse (claiming a variable dead when it is, in fact, live) is worse, as we may overwrite a value that could be used later on, and hence get wrong results from the program. Precise liveness information depends on knowing exactly which paths a program may take through the code when executed, and this is not possible to compute exactly (it is a formally undecidable problem), so it is quite reasonable to allow imprecise results from a liveness analysis, as long as we err on the side of safety, i.e., calling a variable live unless we can prove it to be dead.

We require that a function will always exit by executing a RETURN instruction, i.e. not by “falling out” of the last instruction in its body. So if the last instruction in the body of a function is not a RETURN, GOTO or IF-THEN-ELSE instruction, we add a RETURN instruction to the end. Hence, the only instructions that have empty *succ* sets are RETURN instructions.

For every instruction i , we have a set $gen[i]$, which lists the variables that may be read by instruction i and, hence, are live at the start of the instruction. In other words, $gen[i]$ is the set of variables that instruction i generates liveness for. We also have a set $kill[i]$ that lists the variables that are written to by the instruction. Figure 8.2 shows which variables are in $gen[i]$ and $kill[i]$ for the types of instruction found in intermediate code. x , y and z are (possibly identical) variables and k denotes a constant.

Figure 8.3 shows *succ*, *gen* and *kill* sets for the instructions in the program shown in Fig. 8.1.

For each instruction i , we use two sets to hold the actual liveness information: $in[i]$ holds the variables that are live at the start of i , and $out[i]$ holds the variables that are live at the end of i . We define these by the following equations:

Fig. 8.2 Gen and kill sets

Instruction i	$gen[i]$	$kill[i]$
LABEL l	\emptyset	\emptyset
$x := y$	$\{y\}$	$\{x\}$
$x := k$	\emptyset	$\{x\}$
$x := \mathbf{unop} \ y$	$\{y\}$	$\{x\}$
$x := \mathbf{unop} \ k$	\emptyset	$\{x\}$
$x := y \ \mathbf{binop} \ z$	$\{y, z\}$	$\{x\}$
$x := y \ \mathbf{binop} \ k$	$\{y\}$	$\{x\}$
$x := M[y]$	$\{y\}$	$\{x\}$
$x := M[k]$	\emptyset	$\{x\}$
$M[x] := y$	$\{x, y\}$	\emptyset
$M[k] := y$	$\{y\}$	\emptyset
GOTO l	\emptyset	\emptyset
IF $x \ \mathbf{relop} \ y$ THEN l_t ELSE l_f	$\{x, y\}$	\emptyset
$x := \mathbf{CALL} \ f(args)$	$args$	$\{x\}$
RETURN x	$\{x\}$	\emptyset

Fig. 8.3 *succ*, *gen* and *kill* for the program in Fig. 8.1

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2		a
2	3		b
3	4		z
4	5		
5	6, 13	n, z	
6	7		
7	8	a, b	t
8	9	b	a
9	10	t	b
10	11	n	n
11	12		z
12	4		
13	14		
14		a	

$$in[i] = gen[i] \cup (out[i] \setminus kill[i]) \quad (8.1)$$

$$out[i] = \bigcup_{j \in succ[i]} in[j] \quad (8.2)$$

These equations are recursive. We solve these by fixed-point iteration, as shown in the Appendix A: We initialise all $in[i]$ and $out[i]$ to be empty sets and repeatedly calculate new values for these until no changes occur. This will eventually happen, since we work with subsets of a finite set (the set of variables), and because adding elements to the sets $out[i]$ or $in[j]$ on the right-hand sides of the equations can not reduce the number of elements in the sets on the left-hand sides (i.e., the functions are monotonic). Hence, each iteration will either add elements to some set (which we can do only a finite number of times) or leave all sets unchanged (in which case we are done). It is also easy to see that the resulting sets form a solution to the equation—the last iteration essentially verifies that all equations hold. This is a simple extension of the reasoning used in Sect. 1.5.1.

To find in and out set for Fig. 8.1, initialise these to the empty set and iterate applying (8.2) and (8.1) as assignments until we reach a fixed point.

The order in which we treat the instructions does not matter for the final result of the iteration, but it may influence how quickly we reach the fixed-point. Since the information in (8.1) and (8.2) flows backwards through the program, it is a good idea to do the evaluation in reverse instruction order and to calculate $out[i]$ before $in[i]$. In the example, this means that we will in each iteration calculate the sets in the order

$$out[14], in[14], out[13], in[13], \dots, out[1], in[1]$$

Figure 8.4 shows the fixed-point iteration using this backwards evaluation order. Note that the most recent values are used when calculating the right-hand sides of (8.1)

i	Initial		Iteration 1		Iteration 2		Iteration 3	
	out[i]	in[i]	out[i]	in[i]	out[i]	in[i]	out[i]	in[i]
1			n, a	n	n, a	n	n, a	n
2			n, a, b	n, a	n, a, b	n, a	n, a, b	n, a
3			n, z, a, b	n, a, b	n, z, a, b	n, a, b	n, z, a, b	n, a, b
4			n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b
5			a, b, n	n, z, a, b	a, b, n	n, z, a, b	a, b, n	n, z, a, b
6			a, b, n	a, b, n	a, b, n	a, b, n	a, b, n	a, b, n
7			b, t, n	a, b, n	b, t, n	a, b, n	b, t, n	a, b, n
8			t, n	b, t, n	t, n, a	b, t, n	t, n, a	b, t, n
9			n	t, n	n, a, b	t, n, a	n, a, b	t, n, a
10				n	n, a, b	n, a, b	n, a, b	n, a, b
11					n, z, a, b	n, a, b	n, z, a, b	n, a, b
12					n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b
13			a	a	a	a	a	a
14				a		a		a

Fig. 8.4 Fixed-point iteration for liveness analysis

and (8.2), so, when a value comes from a higher instruction number, the value from the same column in Fig. 8.4 is used, otherwise the value is taken from the previous column.

We see that the result after iteration 3 is the same as after iteration 2, so we have reached a fixed point. We note that n is live-in at instruction 1, which is to be expected, as n is the input parameter. If a variable that is not an input parameter is live at the start of a function, it might in some executions be used before it is initialised, which is generally considered an error (since it can lead to unpredictable results and even security holes). Some compilers issue warnings about use of potentially uninitialised variables, and some compilers enforce initialisation of such variables to a default value (usually 0).

Suggested exercises: 8.1(a, b).

8.3 Interference

We can now define precisely the condition needed for two variables to share a register. We first define *interference*:

Definition 8.2 A variable x interferes with a variable y if $x \neq y$ and there is an instruction i such that $x \in \text{kill}[i]$, $y \in \text{out}[i]$, and instruction i is not $x := y$.

Two different variables can share a register precisely if neither interferes with the other. This is almost the same as saying that they should not be live at the same time, but there are small differences:

- After $x := y$, x and y may be live simultaneously, but as they contain the same value, they can still share a register.
- It may happen that x is not in $out[i]$ even if x is in $kill[i]$, which means that we have assigned to x a value that is definitely not read from x later on. In this case, x is not technically live after instruction i , but it still interferes with any y in $out[i]$. This interference prevents an assignment to x overwriting a live variable y .

The first of these differences is essentially an optimisation that allows more sharing than otherwise, but the latter is important for preserving correctness. In some cases, assignments to dead variables can be eliminated, but in other cases the instruction may have another visible effect (e.g., setting condition flags or accessing memory) and hence can not be eliminated without changing program behaviour.

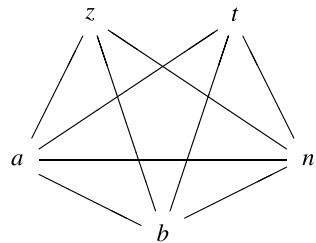
We can use Definition 8.2 to generate interference for each assignment statement in the program in Fig. 8.1:

Instruction	Left-hand side	Interferes with
1	a	n
2	b	n, a
3	z	n, a, b
7	t	b, n
8	a	t, n
9	b	n, a
10	n	a, b
11	z	n, a, b

Even though interference is defined in an asymmetric way in Definition 8.2, the conclusion that the two involved variables cannot share a register is symmetric, so interference defines a symmetric relation between variables. A variable can never interfere with itself, so the relation is not reflective. Because of the symmetry, we can draw interference as an undirected graph, where each node in the graph is a variable, and there is an edge between nodes x and y if x interferes with y (or *vice versa*, as the relation is symmetric). The *interference graph* for the program in Fig. 8.1 is shown in Fig. 8.5.

We will do *global register allocation*, i.e., find for each variable a register that it can stay in at all points in the program (procedure, actually, since a “program” in terms

Fig. 8.5 Interference graph for the program in Fig. 8.1



of our intermediate language corresponds to a procedure in a high-level language). This means that, for the purpose of register allocation, two variables interfere if they do so at *any* point in the program.

8.4 Register Allocation by Graph Colouring

Two variables can share a register if they are not connected by an edge in the interference graph. Hence, we must assign to each node in the interference graph a register number such that:

- 1) Two nodes that are connected by an edge have different register numbers.
- 2) The total number of different register numbers is no higher than the number of available registers.

This problem is well-known in graph theory, where it is called *graph colouring* (in this context, a “colour” is a register number). It is known to be NP-hard, which means that no effective (i.e., polynomial-time) method for doing this optimally is known. In practice, this means that we need to use a heuristic method, which will often find a solution, but may give up in some cases even when a solution does exist. This is no great disaster, as we must deal with non-colourable graphs anyway (by moving some variables to memory), so at worst we get slightly slower programs than we would get if we could colour the interference graphs optimally.

The basic idea of the heuristic method we use is simple: If a node in the graph has strictly fewer than N edges, where N is the number of available colours (i.e., registers), we can set this node aside and colour the rest of the graph. When this is done, the (at most $N-1$) nodes connected by edges to the selected node can not possibly use all N colours, so we can always pick a colour for the selected node from the colours not used by the neighbours.

We can use this method to four-colour the interference graph from Fig. 8.5:

- 1) z has three edges, which is strictly less than four. Hence, we remove z from the graph.
- 2) Now, a has less than four edges, so we also remove this.
- 3) Only three nodes are now left (b , t and n), so we can give each of these a number, e.g., 1, 2 and 3 respectively for nodes b , t and n .
- 4) Since three nodes (b , t and n) are connected to a , and these use colours 1, 2 and 3, we must choose a fourth colour for a , e.g., 4.
- 5) z is connected to a , b and n , so we choose a colour that is different from 4, 1 and 3. Giving z colour 2 works.

The problem comes if there are no nodes that have less than N edges. This in itself does not imply that the graph is uncolourable. As an example, a graph with four nodes arranged and connected as the corners of a square can, even though all nodes have two neighbours, be coloured with two colours by giving opposite corners the same colour. This leads to the following so-called “optimistic” colouring heuristics:

Algorithm 8.3

initialise: Start with an empty stack.

simplify: If there is a node with less than N edges, put this on the stack along with a list of the nodes to which it is connected by edges, and remove the node and its edges from the graph.

If there is no node with less than N edges, pick any node and do as above.

If there are still nodes left in the graph, continue with **simplify**, otherwise go to **select**.

select: Pop a node and its list of connected nodes off the stack. If possible, give the node a colour that is different from the colours of the connected nodes (which are all coloured at this point). If this is not possible, colouring fails and we mark the node for spilling (see below).

If there are still nodes on the stack, continue with **select**, otherwise we are done.

If any nodes are marked for spilling, we must handle this (See Sect. 8.5) and redo graph colouring.

The idea in this algorithm is that, even though a node has N or more edges, some of the nodes it is connected to may have been given identical colours, so the total number of colours used for these nodes is less than N . If this is the case, we can use one of the unused colours. If not, we must mark the node for spill.

There are several things left unspecified by Algorithm 8.3:

- Which node to choose in **simplify**, when none have less than N edges, and
- Which colour to choose in **select**, if there are several choices.

If we choose perfectly in both cases, Algorithm 8.3 will do optimal colouring. But perfect choices are costly to compute so, in practice, we will sometimes have to guess. We will, in Sect. 8.6, look at some ideas for making qualified guesses. For now, we just make arbitrary choices.

Suggested exercises: 8.1(c, d), 8.6.

8.5 Spilling

If the **select** phase is unable to find a colour for a node, Algorithm 8.3 cannot colour the graph. This means we must give up on keeping all variables in registers throughout the program. We must, instead, select some variables that will reside in memory (except for brief periods). This process is called *spilling*. Obvious candidates for spilling are variables corresponding to nodes that are not given colours by **select**. We simply mark these as *spilled* and continue **select** with the rest of the stack, ignoring spilled nodes when selecting colours for the remaining nodes. When we finish Algorithm 8.3, several variables may be marked as spilled.

When we have chosen one or more variables for spilling, we change the program so these are kept in memory. To be precise, for each spilled variable x we:

- 1) Choose a memory address $address_x$ where the value of x is stored. $address_x$ should either be a constant address or a constant offset from the current stack top or frame pointer (see Chap. 9).
- 2) In every instruction i that reads or assigns x , we locally in this instruction rename x to x_i .
- 3) Before an instruction i that reads x_i , insert the instruction $x_i := M[address_x]$.
- 4) After an instruction i that assigns x_i , insert the instruction $M[address_x] := x_i$.
- 5) If x is an input parameter, add an instruction $M[address_x] := x$ to the start of the function. Note that we use the original name for x here.

After this rewrite of the program, we do register allocation again. This includes redoing the liveness analysis, since we have added new variables x_i and changed the liveness of x . We may optimise this a bit by repeating the liveness analysis only for the affected variables (x_i and x), as the results will not change for the other variables.

It may happen that the subsequent new register allocation will generate additional spilled variables. There are several reasons why this may be:

- We have ignored spilled variables when selecting colours for a node in the **select** phase. When the spilled variables are replaced by new variables, these may use colours that would otherwise be available, so we may end up with no choices where we originally had one or more colours available.
- The choices of nodes to remove from the graph in the **simplify** phase and the colours to assign in the **select phase** can change, and we might be less lucky in our choices, so we get more spills.

If we have at least as many registers as the number of variables used in a single instruction, all variables can be loaded just before the instruction, and the result can be saved immediately afterwards, so we will eventually be able to find a colouring by repeated spilling. If we ignore the CALL instruction, no instruction in the intermediate language uses more than two variables, so this is the minimum number of registers that we need. A CALL instruction can use an unbounded number of variables as arguments, possibly even more than the total number of registers available, so it is unrealistic to expect all arguments to function calls to be in registers. We will look at this issue in Chap. 9.

If we take our example from Fig. 8.1, we can attempt to colour its interference graph (Fig. 8.5) with only three colours. The stack built by the **simplify** phase of Algorithm 8.3 and the colours chosen for these nodes in the **select** phase are shown in Fig. 8.6. The stack grows upwards, so the first node chosen by **simplify** is at

Fig. 8.6 Algorithm 8.3 applied to the graph in Fig. 8.5

Node	Neighbours	Colour
n		1
t	n	2
b	t, n	3
a	b, n, t	spill
z	a, b, n	2

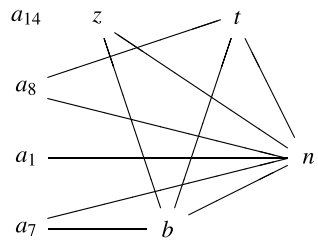
Fig. 8.7 Program from Fig. 8.1 after spilling variable a

```

fib(n) [
1: a1 := 0
   M[addressa] := a1
2: b := 1
3: z := 0
4: LABEL loop
5: IF n = z THEN end ELSE body
6: LABEL body
   a7 := M[addressa]
7: t := a7 + b
8: a8 := b
   M[addressa] := a8
9: b := t
10: n := n - 1
11: z := 0
12: GOTO loop
13: LABEL end
   a14 := M[addressa]
14: RETURN a14
]

```

Fig. 8.8 Interference graph for the program in Fig. 8.7



the bottom. The colours (numbers) are, conversely, chosen top-down as the stack is popped. We can choose no colour for a , as all three available colours are in use by the neighbours b , n and t . Hence, we mark a as spilled. Figure 8.7 shows the program after spill code has been inserted. Figure 8.8 shows the interference graph for the program in Figs. 8.7 and 8.9 shows the stack used by Algorithm 8.3 for colouring this graph, showing that colouring with three colours is now possible.

Suggested exercises: 8.1(e).

8.6 Heuristics

When the **simplify** phase of Algorithm 8.3 is unable to find a node with less than N edges, a node which has at least N neighbours must be chosen. So far, we have chosen arbitrarily, but we may apply some heuristics (qualified guesswork) to the

Fig. 8.9 Colouring of the graph in Fig. 8.8

Node	Neighbours	Colour
n		1
t	n	2
a_8	t, n	3
b	t, n	3
a_7	b, n	2
z	b, n	2
a_1	n	2
a_{14}		1

choice in order to make colouring more likely, or to reduce the number of other variables that will spilled later:

- We may choose a node with close to N neighbours, as this is likely to be colourable in the **select** phase anyway. For example, if a node has exactly N neighbours, it will be colourable if just two of its neighbours get the same colour.
- We may choose a node with many neighbours that have close to N neighbours of their own, as spilling this node may allow many of these neighbours to be coloured.
- We may look at the program and select a variable that does not cost so much to spill, e.g., a variable that is not used inside a loop.

These criteria (and possibly more) may be combined into a single heuristic by giving a numeric value to each node describing how well the corresponding variable fits each criterion, multiplying each with a weight and then adding the results to give a weighted sum.

We have also made arbitrary choices when we pick colours for nodes in the **select** phase. We can try to make it more likely that the rest of the graph can be coloured by choosing a colour that is already used elsewhere in the graph instead of picking a colour that is used nowhere else. This will make it less likely that the nodes connected to an as yet uncoloured node will use all the available colours. A simple instance of this idea is to always use the lowest-numbered available colour.

A more advanced variant of this idea is to look at the uncoloured nodes connected to the node we are about to colour. If we have several choices of colour for the current node, we would like to choose a colour that makes it more likely that its uncoloured neighbours can later be coloured. If an uncoloured neighbour has neighbours of its own that are already coloured, we would like to use one of the colours used among these, as this will not increase the number of colours for nodes that neighbour the uncoloured neighbour, so we will not make it any harder to colour this later on. If the current node has several uncoloured neighbours, we can find the set of neighbour-colours for each of these and select a colour that occurs in as many of these sets as possible. In other words, we look at nodes at distance 2 from

the current node (i.e., two edges away) and use the colour that is used by most of these.

8.6.1 Removing Redundant Moves

An assignment of the form $x := y$ can be removed from the code if x and y use the same register (as the instruction in that case will have no effect). Most register allocators (or later optimisations) attempt to remove such redundant move instructions, and some register allocators try to increase the number of assignments that can be removed by trying to allocate x and y in the same register whenever possible.

If x has already been given a colour by the time we need to select a colour for y , we can choose the same colour for y , as long as it is not used by any variable that y interferes with (including, possibly, x). Similarly, if x is uncoloured, we can give it the same colour as y , if this colour is not used for a variable that interferes with x (including y itself). This is called *biased colouring*.

Another method of achieving the same goal is to combine x and y (if they do not interfere) into a single node *before* colouring the graph, and only split the combined node if the **simplify** phase can not otherwise find a node with less than N edges. This is called *coalescing*.

The converse of coalescing (called *live-range splitting*) can be used as well: Instead of spilling a variable, we can split its node by giving each occurrence of the variable a different name and inserting assignments between these when necessary. The cost of these assignments is likely to be less than the cost of the loads and stores inserted by spilling. Live-range splitting is not quite as effective at increasing the chance of colouring as spilling (since these variables will be live longer than spilled variables, which are live only around a single instruction), and where spilling can reduce the number of required colours down to two (if we disregard CALL instructions), live-range splitting can only reduce it down to the maximum number of variables that are live at the same time at some point in the program.

8.6.2 Using Explicit Register Numbers

Some machine code instructions may require their arguments or results to be in specific registers. For example, the integer multiplication instruction in Intel's IA-32 (x86) processors require the first argument to be in the `eax` register and puts the 64-bit result in the `eax` and `edx` registers. Also, as we shall see in Chap. 9, function calls can require arguments and results to be in specific registers.

Variables used as arguments and results to such operations must, hence, be assigned to these registers *a priori*, before the register allocation begins. We say that these nodes are *pre-coloured* in the interference graph. If two nodes that are pre-coloured to the same register interfere, we can not make a legal colouring of the

graph. One solution would be to spill one or both so they no longer interfere, but that is rather costly.

A better solution is a form of live-range splitting: We insert move instructions that move the (for now, uncoloured) variables to and from the required (pre-coloured) registers immediately before and after an instruction that requires specific registers. Only if this fails to remove interference, do we spill the variable. The specific registers must still be included as pre-coloured nodes in the interference graph, but are not removed from it in the **simplify** phase. Once only pre-coloured nodes remain in the graph, the **select** phase starts. When the **select** phase needs to colour a node, it must avoid colours used by all neighbours to the node—whether they are pre-coloured or just coloured earlier in the **select** phase. The register allocator can try to remove some of the inserted moves by using the techniques described in Sect. 8.6.1.

8.7 Further Reading

Preston Briggs' Ph.D. thesis [2] shows several variants of the register-allocation algorithm shown here, including many optimisations and heuristics as well as considerations about how the various phases can be implemented efficiently. The compiler textbooks [3] and [1] show some other variants and a few newer developments. A completely different approach to register allocation that exploits the structure of a program is suggested in [4].

8.8 Exercises

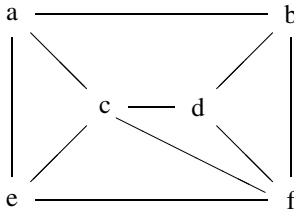
Exercise 8.1 Given the following program:

```

gcd(a, b) [
1: LABEL start
2: IF a < b THEN next ELSE swap
3: LABEL swap
4: t := a
5: a := b
6: b := t
7: LABEL next
8: z := 0
9: b := b mod a
10: IF b = z THEN end ELSE start
11: LABEL end
12: RETURN a
]
```

- a) Show *succ*, *gen* and *kill* for every instruction in the program.
- b) Calculate *in* and *out* for every instruction in the program. Show the iteration as in Fig. 8.4.
- c) Draw the interference graph for *a*, *b*, *t* and *z*.
- d) Make a three-colouring of the interference graph. Show the stack as in Fig. 8.6.
- e) Attempt, instead, a two-colouring of the graph. Select variables for spill, do the spill-transformation as shown in Sect. 8.5, and redo the complete register allocation process on the transformed program. If necessary, repeat the process until register allocation is successful.

Exercise 8.2 Three-colour the following graph. Show the stack as in Fig. 8.6. The graph *is* three-colour-able, so try making different choices if you get spill.



Exercise 8.3 Combine the heuristics suggested in Sect. 8.6 for selecting nodes in the **simplify** phase of Algorithm 8.3 into a formula that gives a single numerical score for each node, such that higher scores is given to stronger candidates for spill. Justify your answer.

Exercise 8.4 Some processors (such as Motorola 68000) have two types of registers: data registers and address registers. Some instructions (such as load and store) expect their arguments or put their results in address registers, while other instructions (such as multiplication and division) expect their arguments or put their results in data registers. Some operations (like addition and subtraction) can use either type of register. There are instructions for moving values between address and data registers.

We handle this by adding the registers as nodes in the interference graph, and make variables interfere with some of these registers.

- a) Describe how a variable that is required to be in an address register can, by adding interference, be prevented from being allocated in a data register, and vice-versa.
- b) The answer above is likely to cause spilling of variables that are used as both address and data. Describe how this can be avoided by a form of live-range splitting.
- c) If there are not enough registers of one type, but there are still available registers of the other type, describe how you can spill a variable to a register of the other type instead of to memory.

Exercise 8.5 Some processors have instructions that operate on values that require two registers to hold. Such processors usually require these values to be held in pairs

of adjacent registers, so an instruction only needs to specify one register number per value (as the other part of the value is implicitly stored in the following register).

We will now look at register allocation where some values must be allocated in register pairs. We note that liveness analysis is unaffected, so only colouring and spill is affected. Hence, we start with an interference graph where some nodes are marked as requiring register pairs.

- a) Modify Algorithm 8.3 to take register pairs into account. Focus on correctness, not efficiency. You can assume “colours” are numbers, so you can talk about adjacent colours, the next colour, etc.
- b) Describe for the **simplify** phase of Algorithm 8.3 heuristics that take into account that some nodes require two registers.
- c) Describe for the **select** phase of Algorithm 8.3 heuristics that take into account that some nodes require two registers.

Exercise 8.6 Describe the class of undirected graphs that can be coloured using at most two colours.

References

1. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press (1998)
2. Briggs, P.: Register allocation via graph coloring, tech. rept. cpc-tr94517-s. Ph.D. thesis, Rice University, Center for Research on Parallel Computation (1992)
3. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
4. Thorup, M.: All structured programs have small tree-width and good register allocation. Inf. Comput. **142**(2), 159–181 (1998)

Chapter 9

Functions



Cats are intended to teach us that not everything in nature has a function.

Garrison Keillor (1942–)

In Chap. 6 we have shown how to translate the body of a single function. Function calls and returns were left (mostly) untranslated by using the `CALL` and `RETURN` instructions in the intermediate code. Nor did we in Chap. 7 show how these instructions should be translated to machine code.

We will, in this chapter, remedy these omissions. We will initially assume that all variables are local to the function that accesses them and that parameters are *call-by-value*, meaning that the *value* of an argument expression is passed to the called function. This is the default parameter-passing mechanism in most languages, and in many languages (e.g., C or SML) it is the only one.

9.1 The Call Stack

A single procedure body uses (in most languages) a finite number of variables. We have seen in Chap. 8 that we can map these variables into a (possibly smaller) set of registers. A program that uses recursive procedures or functions may, however, use an unbounded number of variables, as each recursive invocation of the function has its own set of variables, and there is no bound on the recursion depth. We can not hope to keep all these variables in registers, so we will use memory for some of these. The basic idea is that only variables that are local to the active (most recently called) function will be kept in registers. All other variables will be kept in memory.

When a function is called, all the live variables of the calling function (which we will refer to as the *caller*) need to be stored in memory, so the registers will be

free for use by the called function (the *callee*). When the callee returns, the stored variables are loaded back into registers.

It is convenient to use a stack for this storing and loading, pushing register contents on the stack when they must be saved, and popping them back into registers when they must be restored. Since a stack is (in principle) unbounded, this fits well with the idea of unbounded recursion.

The stack can also be used for other purposes:

- Space can be set aside on the stack for variables that need to be spilled to memory. In Chap. 8, we used a constant address ($address_x$) for spilling a variable x . When a stack is used, $address_x$ is actually an offset relative to a pointer into the stack. This makes the spill-code slightly more complicated, but has the advantage that spilled registers are already saved on the stack when or if a function is called, so they do not need to be stored again.
- Parameters to function calls can be passed on the stack, i.e, written to the top of the stack by the caller, and read from there by the callee.
- The address of the instruction where execution must be resumed after the call returns (the *return address*) can be stored on the stack.
- Since we decided to keep only local variables in registers, non-local variables must reside in memory, which may be global memory or the stack.
- Arrays and records that are allocated locally in a function can be allocated on the stack, as hinted in Sect. 6.8.2.1.

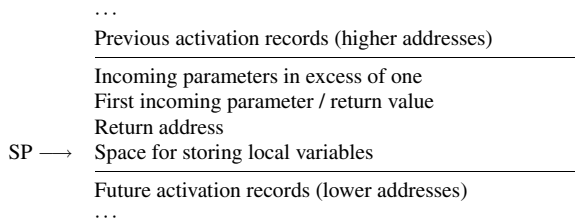
We shall look at each of these in more detail later on.

Most operating systems define a system stack that the operating system uses the store information when a system routine is called or an interrupt is made. The system stack is commonly also used by function calls in user programs. There is no conflict in this, as long as both the operating system and the user program obey the *stack discipline*: When a function or system call returns, the stack pointer (that points to the top of the stack) is restored to the value it had immediately prior to the call. Also, with a few exceptions, the contents of the stack below the stack top is not modified by the call. Hence, we must make sure to compile function calls so we obey the stack discipline. This means that we must move the stack pointer *before* storing values at the top of the stack, as otherwise an interrupt might overwrite these values by storing its own values relative to the not-yet-updated stack pointer.

9.2 Activation Records

Each function invocation will allocate a chunk of memory on the stack to cover all of the function's needs for storing values on the stack. This chunk is called the *activation record* or *frame* for the function invocation. We will use these two names interchangeably. Activation records will typically have the same overall structure for all functions in a program, though the sizes of the various fields in the records may

Fig. 9.1 Simple activation record layout



differ. Often, the machine architecture (or operating system) will dictate a *calling convention* that standardises the layout of activation records. This allows a program to call functions that are compiled with other compilers or even written in a different language, as long as all the involved compilers follow the same calling convention.

We will start by defining very simple activation records and then extend and refine these later on. Our first model uses the assumption that all information is stored in memory when a function is called. This includes parameters, return address and the contents of registers that need to be preserved. A possible layout for such an activation record is shown in Fig. 9.1.

We use a stack that grows downwards in memory (to lower addresses). SP is short for “stack pointer”, and points to the last used space of the stack. When a new function is called, its activation record is placed below this, and SP is moved down to the new stack top. Note that the direction of stack growth (up or down in memory) and position of stack pointer relative to the stack top (last used element or first unused element) can differ from system to system.

In the layout shown in Fig. 9.1, the first (top) words of the activation record holds the incoming parameters, with the first parameter at the lowest address (nearer the stack top). Below these, the return address is stored. The function will typically move the parameters to registers (except for parameters that have been spilled by the register allocator) before executing its body. The space used for the first incoming parameter is also used for storing the return value of the function call (if any). Below the return address, the activation record has space for storing other local variables, e.g., spilled variables and local arrays, or for preserving variables across future function calls. We assume words are 64 bits and memory is byte-addressed, so to skip to the next word in memory, you must add 8 to the address.

9.3 Prologues, Epilogues and Call-Sequences

In Chap. 6, we kept function definitions, function calls, and function returns basically untranslated, assuming parameters and results are passed in named intermediate-code variables.

But, now that parameters and results are passed through the activation record, we need to translate a function header into code that reads parameters from the activation

Fig. 9.2 Prologue for the header $f(p_1, \dots, p_m)$ using the frame layout shown in Fig. 9.1

<code>LABEL f</code>	Label for function entry point
<code>$p_1 := M[SP + 8]$</code>	Load parameters into variables
<code>...</code>	
<code>$p_m := M[SP + 8 * m]$</code>	
<code>$SP := SP - framesize_f$</code>	Make room for local variables

Fig. 9.3 Epilogue for the instruction `RETURN $result$` using the frame layout shown in Fig. 9.1

<code>$SP := SP + framesize_f$</code>	Free space for local variables
<code>$M[SP + 8] := result$</code>	Store result on stack
<code>GOTO $M[SP]$</code>	Return to caller

record into variables. This code is called the *prologue* of the function. Likewise, a RETURN statement should be translated into code to store the return value in the activation record and jumps to the return address that was stored in the activation record by the caller. This is called the *epilogue* of the function.

For the activation-record layout shown in Fig. 9.1, a suitable prologue and epilogue is shown in Figs. 9.2 and 9.3. The prologue is for a function with a header $f(p_1, \dots, p_m)$ and the epilogue is for a return statement of the form `RETURN $result$` . $framesize_f$ is the size of the frame for the function f , excluding space for parameters and return address.

Note that, though we have used a notation similar to the intermediate language introduced in Chap. 6, we have extended this a bit: We now use $M[]$ and GOTO with general expressions as arguments. The first is just notational convenience, since we could evaluate the expression to a variable before using it as an address. The latter requires jumps to calculated addresses, which the original intermediate language does not support. In any case, the prologue and epilogue are usually generated directly as machine language, where jumps to calculated addresses (indirect jumps) are usually available.

If a function has several RETURN statements, each of these will generate an epilogue. But they are all identical except for the result variable that is copied to the frame. So it is common to have only a single epilogue at the end of the code for the function and let all RETURN statements share this. The code for each return statement will now just copy its result variable to a common result variable used by the shared epilogue and then jump to this.

In Chap. 6, we used a single intermediate-language instruction to implement a function call. This function-call instruction must be translated into a *call-sequence* of instructions that will save registers, put parameters in the activation record, etc. A call-sequence suitable for the activation-record layout shown in Fig. 9.1 is shown in Fig. 9.4. The code is an elaboration of the intermediate-language instruction $x := \text{CALL } g(a_1, \dots, a_n)$, called from a function f .

First, all registers that can be used to hold f 's variables are stored in f 's frame. In Fig. 9.4, R_0 – R_k are assumed to hold such variables. These are stored in the space set aside in f 's frame for storing its local variables.

Before storing the parameters a_1, \dots, a_n and the return address in g 's frame, we must move SP down to make space for these, as otherwise a system interrupt (that

Fig. 9.4 Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using the frame layout shown in Fig. 9.1

$M[SP] := R0$	Save caller's local variables
...	
$M[SP + 8 * k] := Rk$	
$SP := SP - 8 * (n + 1)$	Make space for parameters and return address
$M[SP + 8] := a_1$	Save parameters on stack
...	
$M[SP + 8 * n] := a_n$	
$M[SP] := \text{returnaddress}$	Save return address on stack
GOTO g	Jump to callee
LABEL returnaddress	
$x := M[SP + 8]$	Fetch result from stack
$SP := SP + 8 * (n + 1)$	Restore caller's local variables
$R0 := M[SP]$	
...	
$Rk := M[SP + 8 * k]$	

uses the same stack) might overwrite the values. For the same reason, we don't restore SP to its previous value before having read the function result from g 's frame.

After adjusting SP , the parameters and the return address are stored in the prescribed locations in the new frame. Finally, a jump to the address of the function g is made. When the function call returns, the result is read from the frame into the variable x , SP is restored to its previous value, and the saved registers are read back from f 's frame.

Keeping all the parameters in register-allocated variables until just before the call, and only then storing them in the new frame can require a lot of registers to hold the parameters (as these are all live up to the point where they are stored), so if a function has many parameters, it is likely that one or more of a_1, \dots, a_n will be spilled before they are stored in the new frame. It would seem better to store each parameter in the new frame as soon as it is evaluated, so only one of the variables a_1, \dots, a_n will be live at any one time. This requires that SP is modified before the parameters values are calculated (so the stored values are not overwritten by interrupts or other calls), so if any of the parameter calculations need to use values from the current frame (e.g. for accessing spilled variables or local arrays or for making other function calls), the offsets from SP used in these calculations must be modified to take this into account. Exercise 9.1 returns to this issue, but for now we just assume that there are registers enough.

In this simple call-sequence, we save in the caller frame all registers that can hold the caller's register-allocated variables (except x , that is overwritten by the result of the call), so these are preserved across the function call. This may save more registers than needed, as not all registers will hold values that are required after the call (i.e. they may be dead). We will return to this issue in Sects. 9.5 and 9.7.

Suggested exercises: 9.1.

9.4 Letting the Callee Save Registers

The convention used by the activation record layout in Fig. 9.1 is that, before a function is called, the caller saves all registers that must be preserved. Hence, this strategy is called *caller-saves*. An alternative strategy is to let the called function (the callee) save the contents of the registers that need to be preserved, and restore these immediately before the function returns. This strategy is called *callee-saves*.

Stack-layout, prologue, epilogue and call sequence for the callee-saves strategy are shown in Figs. 9.5, 9.6, 9.7 and 9.8. $framesize_f$ does not include the space to store the $k+1$ registers that need to be preserved (as this space is explicitly accounted for by other modifications to SP), only the space for local variables. If $framesize_f$ is known at compile time, offsets like $framesize_f + 8 * (k + m)$ can be calculated at compile time, so all offsets to SP can be constants.

Note that it may not be necessary to store *all* registers that can be used to allocate variables, only those that the function actually uses to hold its local variables. We will discuss this issue below and again in Sect. 9.7.

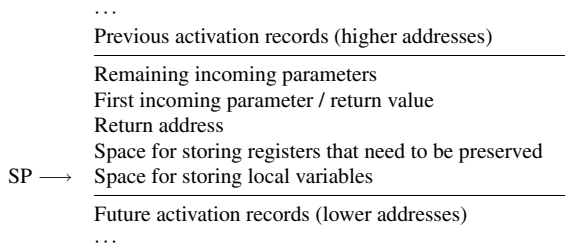


Fig. 9.5 Activation record layout for callee-saves

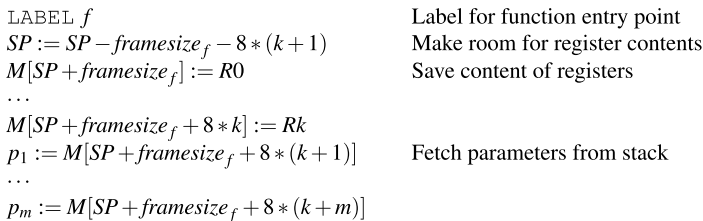


Fig. 9.6 Prologue for the header $f(p_1, \dots, p_m)$ using callee-saves

```

M[SP + framesizef + 8 * (k + 1)] := result    store result on stack
R0 := M[SP + framesizef]                      Restore register content
...
Rk := M[SP + framesizef + 8 * (k + m)]
SP := SP + framesizef + 8 * (k + 1)          Free space for register content
GOTO M[SP]                                       Return to caller
    
```

Fig. 9.7 Epilogue for the instruction RETURN *result* using callee-saves

```

SP := SP - 8 * (n + 1)    Make space for parameters
                          and return address
M[SP + 8] := a1         Store parameters on stack
...
M[SP + 8 * n] := an
M[SP] := returnaddress   Store return address on stack
GOTO g                   Jump to callee
LABEL returnaddress
x := M[SP + 8]           Fetch result from stack
SP := SP + 8 * (n + 1)  Free space for parameters
    
```

Fig. 9.8 Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using callee-saves

9.5 Caller-Saves Versus Callee-Saves

So far, the only difference between caller-saves and callee-saves is *when* registers are saved. However, once we refine the strategies to save only a subset of the registers that can be used to hold variables, other differences emerge: Caller-saves need only save the registers that hold variables that are *live* after the call returns, and callee-saves need only save the registers that the callee will actually use. We will in Sect. 9.7 return to how this can be done, but at the moment just assume these optimisations are made.

Caller-saves and callee-saves each have their advantages (described above) and disadvantages: When caller-saves is used, we might save a live variable in the frame even though the callee does not use the register that holds this variable. On the other hand, with callee-saves we might save some registers that do not actually hold live values. We can not avoid these unnecessary saves, as each function is compiled independently and, hence, do not know the register usage of their callers and callees. We can, however, try to *reduce* unnecessary saving of registers by using a mixed caller-saves and callee-saves strategy:

Some registers are designated caller-saves and the rest as callee-saves. If any live variables are held in caller-saves registers, it is the caller that must save these to its own frame (as in Fig. 9.4, though only registers that are both designated caller-saves *and* hold live variables are saved). If a callee uses any callee-saves registers in its body, it must save these before using them, as in Fig. 9.6. Only callee-saves registers that are actually used in the body need to be saved.

Calling conventions typically specify which registers are caller-saves and which are callee-saves, as well as the layout of the activation records.

9.6 Using Registers to Pass Parameters

In both call sequences shown (in Figs. 9.4 and 9.8), parameters are stored in the frame, and in both prologues (Figs. 9.2 and 9.6), these are immediately loaded back into registers. It will save a good deal of memory traffic if we pass the parameters in registers instead of memory.

Normally, only a few (4–8) registers are used for parameter passing. These are used for the first parameters of a function, while the remaining parameters are passed on the stack, as we have done above. Since most functions have fairly short parameter lists, most parameters will normally be passed in registers. The registers used for parameter passing are typically a subset of the caller-saves registers, as parameters are not live after the call and hence do not have to be preserved.

A possible division of registers for a 16-register architecture is shown in Fig. 9.9. Note that the return address is also passed in a register. Most RISC architectures have jump-and-link (function-call) instructions that leave the return address in a register, so this is only natural. However, if a new function call is made inside the body of the callee, this register is overwritten, so the return address must be saved in the activation record before any calls are made, so the return-address register is marked as callee-saves in Fig. 9.9. In this manner, the return-address register is just like any other register that must be preserved in the frame of the callee if it is used in the body, i.e., if the callee calls any functions.

Activation record layout, prologue/epilogue and call sequence for a calling convention using the register division in Fig. 9.9 are shown in Figs. 9.10, 9.11, 9.12 and 9.13.

U is the number of callee-saves registers that need to be stored in the frame. Note that the offsets to SP for storing registers are not simple functions of their register numbers, as only a subset of the registers need to be saved, so we simply name them $offset_{R4}$ and so on.

$R15$ (which holds the return address) is, like any other callee-saves register, saved in the prologue and restored in the epilogue if it is used inside the body (i.e, if the body makes a function call). It is stored at the bottom of the frame, as required by the layout in Fig. 9.9.

Register	Saved by	Used for
0	caller	parameter 1 / result / local variable
1-3	caller	parameters 2 - 4 / local variables
4-12	callee	local variables
13	caller	temporary storage (unused by register allocator)
14	callee	SP
15	callee	return address

Fig. 9.9 Possible division of registers for a 16-register architecture

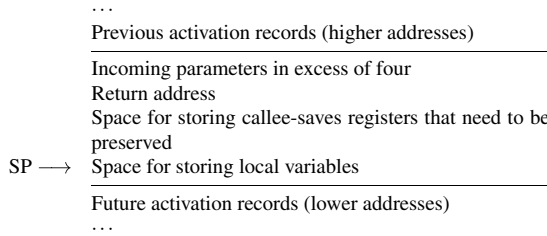


Fig. 9.10 Activation record layout for the register division shown in Fig. 9.9

<i>LABEL f</i>	Label for function entry point
$SP := SP - framesize_f - 8 * U$	Make room for register contents
$M[SP + framesize_f + offset_{R4}] := R4$	Save callee-saves register,
...	if used in body
$M[SP + framesize_f + offset_{R12}] := R12$	Save callee-saves register,
	if used in body
$M[SP + framesize_f] := R15$	Save return-address register,
	if used in body
$p_1 := R0$	Fetch first four parameters
$p_2 := R1$	from registers
$p_3 := R2$	
$p_4 := R3$	
$p_5 := M[SP + framesize_f + 8 * U]$	Fetch remaining parameters
...	from memory
$p_m := M[SP + framesize_f + 8 * (U + m - 5)]$	

Fig. 9.11 Prologue for the header $f(p_1, \dots, p_m)$ using the register division shown in Fig. 9.9

$R0 := result$	Store result in register
$R4 := M[SP + framesize_f + offset_{R4}]$	Restore callee-saves register,
...	if it was saved in prologue
$R12 := M[SP + framesize_f + offset_{R12}]$	Restore callee-saves register,
	if it was saved in prologue
$R15 := M[SP + framesize_f]$	Restore return-address register,
	if it was saved in prologue
$SP := SP + framesize_f + 8 * U$	Free space for storing registers
GOTO R15	Return to caller

Fig. 9.12 Epilogue for the instruction RETURN *result* using the register division shown in Fig. 9.9

In a call-sequence, the instructions

```

R15 := returnaddress
GOTO g
LABEL returnaddress
    
```

can on most RISC processors be implemented by a jump-and-link instruction.

$M[SP + offset_{live_1}] := live_1$	Save live variable,
...	if allocated to a caller-saves register
$M[SP + offset_{live_k}] := live_k$	Save live variable,
	if allocated to a caller-saves register
$SP := SP - \max(0, 8 * (n - 4))$	Make space for parameters
$R0 := a_1$	Store first four parameters
...	in registers
$R3 := a_4$	
$M[SP + 8] := a_5$	Store remaining parameters
...	on stack
$M[SP + 8 * (n - 4)] := a_n$	
$R15 := returnaddress$	Store return address in register
GOTO g	Jump to callee
LABEL $returnaddress$	
$x := R0$	Fetch result from register
$SP := SP + \max(0, 8 * (n - 4))$	Free space used for parameters
$live_1 := M[SP + offset_{live_1}]$	Restore live variable,
...	if allocated to a caller-saves register
$live_k := M[SP + offset_{live_k}]$	Restore live variable,
	if allocated to a caller-saves register

Fig. 9.13 Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using the register division shown in Fig. 9.9

9.7 Interaction with the Register Allocator

As we have hinted above, the register allocator can be used to optimise function calls, as it can provide information about which registers need to be saved.

The register allocator can tell which variables are live after the function call. In a caller-saves strategy (or for caller-saves registers in a mixed strategy), only the (caller-saves) registers that hold variables that are live across the function call need to be saved before the function call.

Likewise, the register allocator can return information about which registers are used by the function body of the callee, so only these need to be saved in a callee-saves strategy.

If a mixed strategy is used, variables that are live across a function call should, if possible, be allocated to callee-saves registers. This way, the caller does not have to save these and, with luck, they do not have to be saved by the callee either (as the callee might not use these registers in its body). If all variables that are live across function calls are made to interfere with all caller-saves registers, the register allocator will not allocate these variables in caller-saves registers, which achieves the desired effect. If no callee-saves register is available, the variable will be spilled and hence, effectively, be saved across the function call. This way, the call sequence will not need to worry about saving live variables stored in caller-saves registers, as the register allocator ensures that there will be none.

As spilling may be somewhat more costly than local save/restore around a function call, it is a good idea to have plenty of callee-saves registers for holding variables that are live across function calls, so less spilling is required. Hence, most calling conventions specify more callee-saves registers than caller-saves registers.

Note that, though the prologues shown in Figs. 9.2, 9.6 and 9.11 load all stack-passed parameters into registers, this should actually not be done for parameters that are spilled—these should just stay spilled in the activation record. Likewise, a register-passed parameter that needs to be spilled should be transferred to that variable's spill slot in the frame instead of to a symbolic register.

In Figs. 9.2, 9.6 and 9.11, we have moved register-passed parameters from the numbered registers or stack locations to named registers, to which the register allocator must assign numbers. Similarly, in the epilogue we move the function result from a named variable to *R0*. This means that these parts of the prologue and epilogue must be included in the code that the register allocator analyses (so the named variables will be replaced by numbers). This will also automatically handle the issue about spilled parameters mentioned above, as spill-code is inserted immediately after the parameters are (temporarily) transferred to registers. This may cause some extra memory transfers when a spilled stack-passed parameter is first loaded into a register and then immediately stored into its spill slot. If the spill slot is at a different address than the parameter slot, this will need to be done anyway, though.

It may seem odd that we move register-passed parameters to named registers instead of just letting them stay in the registers in which they are passed. But the parameter-passing registers may be needed for other function calls, which gives problems if a parameter allocated to one of these needs to be preserved across the call. As mentioned above, variables that are live across function calls should not be allocated to caller-saves registers, and parameter registers are usually caller-saves registers. By moving the parameters to named registers, the register allocator is free to allocate these to callee-saves registers if needed. If this is not needed, the register allocator may allocate the named variable to the register that the parameter was passed in and eliminate the (now superfluous) register-to-register move. As mentioned in Sect. 8.6, modern register allocators will eliminate most such moves anyway, so we might as well exploit this.

In summary, given a good register allocator, the compiler needs to do the following to compile a function:

- 1) Generate code for the body of the function, using symbolic names for variables (except pre-coloured variables used for parameter-passing in call sequences or for instructions that require specific registers, see Sect. 8.6.2). This code should use call sequences instead of `CALL` instructions, and all `RETURN` instructions should be replaced by moves from named variables to the numbered variable for return values followed by jumps to the end of the procedure body, where a common epilogue will be made.
- 2) Add code in the prologue for moving parameters from numbered registers and stack locations into the named variables used for accessing the parameters in the body of the function.

- 3) Call the register allocator with this extended function body and the stack offset for placing the first spilled variable, should any be spilled. The register allocator should be aware of the register division (caller-saves/callee-saves split), and allocate variables that are live across function calls only to callee-saves registers. The register allocator should return both the set of used callee-saves registers and the number of spilled variables.
- 4) To the register-allocated code, add code in prologue and epilogue for saving and restoring the callee-saves registers (including the return-address register) that the register allocator indicates have been used in the extended function body, and for updating the stack pointer with the size of the frame (including space for saved registers and spilled variables).
- 5) Add a function label at the beginning of the code, and a return jump at the end of the epilogue.

9.8 Local Variables

Local variables that correspond to single machine words are typically register allocated and will only need to be stored in the frame if they are spilled or across function calls to free the registers that they occupy. But larger values such as arrays, strings and records will normally be stored in memory.

If such local values are not required to survive after the function returns, they can be stored in the frame. If their sizes are known at compile time, the total size of the frame and the offsets relative to the stack pointer where these values are stored can also be calculated at compile time. This means that the base address of a stack-allocated array and the address of any field of a stack-allocated record is a compile-time constant offset from the stack pointer, so we can use the frame layouts, prologues, epilogues and call sequences above without modification—except for adding the sizes of stack-allocated arrays and records to the frame size.

But if the size of, say, local arrays can depend on run-time values, storing these in the frame will make both the size of the frame and the offsets to the start of each array unknown at compile time. This means that we need run-time variables to store both the sizes and offsets. For arrays, it is typical to use intermediate-language variables to store their base addresses, so these will typically be kept in registers (unless spilled). The size of the frame needs to be stored in a dedicated register that can not be spilled, as you need the size of the frame to calculate the address of spilled variables. See Sect. 9.11.1 for an alternative implementation of variable-sized frames.

9.9 Accessing Non-local Variables

We have up to now assumed that all variables used in a function are local to that function, but most high-level languages also allow functions to access variables that are not declared locally in the functions themselves. We will look at two simple instances of this: Global variables and reference parameters.

9.9.1 Global Variables

In C, variables are either global, which means that they can be accessed by any function in the program, or local to a function, so they can be accessed only by this function. Local variables are treated exactly as we have described above, i.e, stored in a register or in the frame. Global variables will, on the other hand, be stored in memory. The location of each global variable can be calculated at compile-time or link-time. Hence, a use of a global variable x generates the code

$$x := M[address_x]$$

instruction that uses x

The global variable is loaded into a (register-allocated) temporary variable and this will be used in place of the global variable in the instruction that needs the value of the global variable.

An assignment to a global variable x is implemented as

$$x := \text{the value to be stored in } x$$

$$M[address_x] := x$$

Note that a global variable is treated almost like a spilled variable: Its value is loaded from memory into a register immediately before any use, and stored from a register into memory immediately after an assignment. Like with spill, it is possible to use different register-allocated variables for each use of x .

If a global variable is used often within a function, it can be loaded into a local variable at the beginning of the function and stored back again when the function returns. However, a few extra considerations need to be made:

- The variable must be stored back to memory whenever a function is called, as the called function may read or change the global variable. Likewise, the global variable must be read back from memory after the function call, so any changes to the global variable will be registered in the local copy. Local copies of global variables are, hence, not live across function calls, and can thus be allocated in caller-saves registers.
- If the language allows *call-by-reference* parameters (see below) or pointers to global variables, there may be more than one way to access a global variable: Either through its name, or via a call-by-reference parameter or pointer. If we cannot exclude the possibility that a call-by-reference parameter or pointer can access a global variable, it must be stored/retrieved before/after any access to a call-by-reference parameter or any access through a pointer. It is possible to make a global *alias analysis* that determines if global variables, call-by-reference parameters or pointers may point to the same location (i.e, may be *aliased*). However, this is a fairly complex and rather costly analysis, so many compilers simply assume that a global variable may be aliased with *any* call-by-reference parameter or pointer

and that any two of the latter may be aliased. We will look at alias analysis in Chap. 10.

The above tells us that accessing local variables (including call-by-value parameters) is faster than accessing global variables. Hence, good programmers will use global variables sparingly.

9.9.2 *Call-by-Reference Parameters*

Some languages allow parameters to be passed by *call-by-reference*. In Pascal, these are called var-parameters, and in FORTRAN, this is the default parameter-passing method. A parameter passed by call-by-reference must be a variable, an array element, a field in a record or, in general, anything that is allowed at the left-hand-side of an assignment statement. Inside the function that has a call-by-reference parameter, values can be assigned to the parameter, and these assignments actually update the variable, array element or record-field that was passed as parameter, such that the changes are visible to the caller. This differs from assignments to call-by-value parameters, as these update only local variables.

Call-by-reference is implemented by passing the *address* of the variable, array element or whatever that is given as parameter instead of its *value*. Any access (use or definition) to the call-by-reference parameter must be through this address.

In C, there are no explicit call-by-reference parameters, but it is possible to explicitly pass *pointers* to variables, array-elements, etc. as parameters to a function by using the & (address-of) operator, which finds the address of an assignable variable, element or field. When the value of the variable is to be used or updated, this pointer must be explicitly followed, using the * (dereference) operator. So, apart from a more verbose notation and a higher potential for programming errors, this is not significantly different from “real” call-by-reference parameters such as found in Pascal or FORTRAN.

In any case, a variable that is passed as a call-by-reference parameter or has its address passed via a & operator, must reside in memory (since registers don’t have addresses). This means that, at the time of the call, the variable must be spilled or allocated to a caller-saves register, so it will be stored before the call and restored afterwards (which is pretty much the same).

It also means that passing a result back from callee to the caller by a call-by-reference or pointer parameter can be slower than using the return value, as the return value can be passed in a register. Hence, like global variables, call-by-reference and pointer parameters should be used sparingly.

Each of these on their own have the same aliasing problems as when combined with global variables.

9.10 Functions as Parameters

If a function is declared globally, it can access only global variables and its own local variables. Such a function can be passed as an argument or returned as a result just by passing/returning its address. This is the mechanism in C, where all functions are declared globally.

While most processors have either a jump-and-link instruction that stores the return address in a register or a call instruction that stores the return address on the stack, these typically require the address of the called function to be specified in the instruction, i.e., as a constant. The address of a function passed in as a parameter will typically reside in a register and not as a constant in the code, so we may not be able to use the built-in jump-and-link or call instruction. So it may be necessary to explicitly store the return address, as done in, e.g., Fig. 9.13 and use a jump instruction that takes its destination address in a register or memory location. Alternatively, the destination address can be put in a specific register (not used by the register allocator) and use a normal jump-and-link or call instruction that jumps to code that immediately jumps to the address in the register. It costs an extra jump, but may be simpler and shorter than code for explicitly storing the return address.

If functions can be declared locally inside other functions, we need more complex mechanisms that are beyond the scope of this book.

9.11 Variants

We have so far seen activation records with sizes known at compile time, that are stored in stacks that grow downwards in memory, and where *SP* points to the last used element of the stack. There are, however, reasons why you sometimes may want to change some of these details.

9.11.1 Variable-Sized Frames

If local arrays are allocated on the stack, the size of the activation record depends on the size of the arrays. If these sizes are not known at compile-time, neither will the size of the activation records. Hence, we need a run-time variable to store the size of the frame or, equivalently, to point to the opposite end of the frame than the stack pointer does (so the difference between this register and the stack pointer is the size of the frame). This pointer is typically called the *frame pointer*, shortened to *FP*. When a function is called, the new *FP* takes the value of the old *SP*, but we must restore the old value of *FP* when the function returns. We do this by storing the old value of *FP* in a location with a fixed offset from the new *FP*. This can, for example, be next to where the return address is stored. See also Exercise 9.1.

9.11.2 Variable Number of Parameters

Some languages (e.g., C and LISP) allow a function to have a variable number of parameters. This means that the function can be called with a different number of parameters at each call. In C, the `printf` function is an example of this. Normally, a function with a variable number of parameters has a fixed minimum number of parameters known at compile time, and any call to the function can supply an arbitrary number of extra parameters on top of this.

The prologue shown in Fig. 9.11 can easily be modified to handle this: The fixed parameters are transferred or loaded as shown, and the body of the function will fetch the remaining parameters from the parameter registers or the frame as needed.

The call sequence needs no modification, as the caller obviously knows that actual number of parameters.

9.11.3 Direction of Stack-Growth and Position of *FP*

There is no particular reason why a stack has to grow downwards in memory, though this is the most common choice. Sometimes the choice is arbitrary, but at other times there is an advantage to have the stack growing in a particular direction. Some instruction sets have memory-access instructions that include a constant offset from a register-based address. If this offset is unsigned (as it is on, e.g., IBM System/370), it is an advantage that all fields in the activation record are at non-negative offsets from a register. If the stack grows down in memory, all offsets from *SP* to the frame are at non-negative offsets, so a downwards-growing stack is good if offsets can not be negative. In an upwards growing stack, we can use a frame pointer to point to the low-address end of the frame and all offsets can be relative to this.

If, on the other hand, offsets are signed but have a small range (as on Digital's Vax, where the range is -128 to $+127$), it is an advantage to use both positive and negative offsets. This can be done by letting *FP* point, for example, 128 bytes into the frame, so the full range of offsets can be used.

9.11.4 Register Stacks

Some processors, e.g., Oracle's Sparc and Intel's IA-64 (Itanium) have on-chip stacks of registers. The intention is that frames are kept in the stack of registers rather than in a stack in memory. At call or return of a function, the register-stack pointer is adjusted. Since the register stack has a finite size, which is often smaller than the total size of the call stack, it may overflow. This is signalled by the processor and trapped by the operating system, which stores part of the register stack in memory and shifts the rest down (or up) to make room for new elements. If the register stack

underflows (at a pop from an empty register stack), the OS will restore earlier saved parts of the stack.

9.12 Optimisations for Function Calls

A function call imposes an overhead for passing parameters and results, storing registers, restoring them afterwards, and jumping to and from the function. This overhead does not directly contribute to the calculations done by the program, so it makes sense to try to reduce it. We will look at some common ways to do so.

9.12.1 *Inlining*

Inlining attempts to completely eliminate the overhead of a function call by, at compile time, replacing a function call with a copy of the body of the called function. This can, obviously, only be done if the identity of the called function is known at compile time, so calls to functions stored in variables or parameters can not be inlined.

An issue to be aware of when inlining is *variable capture*: Let us say that the calling function f has a local variable x and the called function g accesses a global variable x . Inlining the call to g will put the body of g into the body of f , so by the usual scoping rules, the x in the copy of the body g will now refer to the local variable in f rather than to the global variable. This is in Fig. 9.14 illustrated by an example in the language C. Variable capture can be avoided by, prior to inlining, renaming all local variables in both the calling function and the inlined copy of the called function, so they all have names that do not occur anywhere else in the program. We can now describe the correct procedure for inlining in a C-like language:

1. Rename all local variables in the calling function f .
2. Make a copy g' of the called function g and rename all local variables in g' .
3. Replace the call to g in f by a block that declares local variables corresponding to the parameters of g' and initialise these with the argument expressions of the call to g . These declarations are followed by the body of g' .
4. g' is no longer needed and can be removed.

This is illustrated in Fig. 9.15.

While, on the surface, inlining eliminates the overhead of function calls, it does not always make a program faster. If the called function has a very large body, replacing a call with a copy of the function body, especially if this is done several times, will increase the total code size. Enlarging the program can change it from fitting in the instruction cache to not fitting in the instruction cache, which can make it run a slower. Inlining also increases the size of function bodies and the number of variables local to a function. This can cause register allocation to spill variables.

<pre> int x = 17; int g(int a) { return a + x; } int f(int x) { return g(x); } </pre>	<pre> int x = 17; int g(int a) { return a + x; } int f(int x) { { int a = x; return a + x; } } </pre>
Original program	Incorrect inlining

Fig. 9.14 Variable capture when inlining

<pre> int x = 17; int g(int a) { return a + x; } int g'(int a_4624) { return a_4624 + x; } int f(int x_1473) { return g(x_1473); } </pre>	<pre> int x = 17; int g(int a) { return a + x; } int f(int x_1473) { { int a_4624 = x_1473; return a_4624 + x; } } </pre>
Renaming variables	Correct inlining

Fig. 9.15 Renaming variables when inlining

If the spilled variable is accessed in a loop, this may end up costing more than the overhead of calling a function. For this reason, most compilers inline only small functions, so inlining doesn't increase the code size by very much.

There is also a problem with inlining recursive functions. If a call from a function f to a small recursive function g is inlined, the resulting modified f will still contain calls to g . Since g is small, we want to inline the call, but again the result will have calls to g and so on, ad infinitum. A simple solution would be to stop inlining after one or two rounds, but the gain of inlining a recursive function this way is modest: Most of the recursive calls are to the unmodified g , so there is only savings in the first few (inlined) calls. It is better to not inline calls to g from f , but instead inline calls to g from g . One step of such inlining will reduce the overhead of every second recursive call. Such recursive inlining is illustrated in Fig. 9.16, which uses ML-style syntax for function definitions and local declarations.

9.12.2 Tail-Call Optimisation

Back when BASIC programming on home computers was a common pastime, a standard optimisation trick was to replace the statements `GOSUB 1000: RETURN` (where `GOSUB` is short for “go to subroutine” and the colon separates statements) by

<pre> fun g n = if n = 0 then 1 else n * g (n - 1) fun f x = g (x+3) </pre> <p style="text-align: center;">Original program</p>	<pre> fun g n1 = if n1 = 0 then 1 else n1 * (let val n2 = n1 - 1 in if n2 = 0 then 1 else n2 * g (n2 - 1) end) fun f x = g (x+3) </pre> <p style="text-align: center;">Recursive inlining of g</p>
---	--

Fig. 9.16 Recursive inlining

the single statement `GOTO 1000`. The reasoning was that the `GOSUB` statement will jump to line 1000 but also push a return address pointing to the `RETURN` statement. When the subroutine at line 1000 returns, it will return to the `RETURN` statement, which immediately returns to the next return address on the stack. By using `GOTO` instead, no new return address is pushed, so when the subroutine at line 1000 returns, it does so directly to where the (now eliminated) `RETURN` statement would. The net effect is the same, but we save both space on the stack for the return address, time to move the return address to and from the stack, and an extra return jump. A similar optimisation is common in assembly-language programming.

A call to a subroutine, procedure or function that is immediately followed by a return from the calling subroutine, procedure or function is called a *tail call*, and the optimisation described above is called *tail-call optimisation*.

In C and similar languages, the equivalent to the `gosub-return` sequence is a statement of the form `return f(e1, ..., en)`; which calls a function and immediately returns, or when the last that happens in the body of a procedure is a call to another procedure. But, unlike in BASIC, we can not at the source-language level replace a call to a function or procedure by a jump, as C does not provide jumps to procedures. In the intermediate language described in Sect. 6.2, `return f(e1, ..., en)`; will appear as the instruction sequence `[x := CALL f(a1, ..., an), RETURN x]`, where `a1, ..., an` are intermediate-language variables holding the values of the expressions `e1, ..., en` and `x` is a temporary variable for holding the result of the call. Even if we allow jumps to function labels, we can not in the intermediate language handle parameter passing in jumps, so we need to apply the optimisation when translating the intermediate language to machine language, where we have call sequences, prologues and epilogues. As we saw in Sect. 9.3, the above sequence of intermediate-language instructions compile to code of the following form:

1. Code for saving live variables that are stored in caller-saves registers.
2. Code for transferring the parameters from the local variables `a1, ..., an` to the registers or stack locations used to pass parameters.
3. Code for storing the return address, and jumping to the function label `f`.
4. Code for transferring the result of the call to `f` from the register or stack location used for passing function results to the local variable `x`.

5. Code for restoring live variables that are stored in caller-saves registers.
And, after the call returns,
6. Code for transferring the return value from the local variable x to the register or stack location used for passing function results.
7. Code for restoring from the stack the subset of the callee-saves registers that were used in the body of the current function. This may include the return address, if this is passed in a callee-saves register.
8. Code for freeing on the stack the space used for storing variables, return addresses, and other things.
9. A jump to the return address.

The first thing we note is that no live variables need to be saved before the call to f : The only variable live after the call to f is x , and it gets its value from the call, so it doesn't need saving. So we can look at this somewhat simpler sequence:

1. Code for transferring the parameters from the local variables a_1, \dots, a_n to the registers or stack locations used to pass parameters.
2. Code for storing the return address, and jumping to the function label f .
3. Code for transferring the result of the call to f from the register or stack location used for passing function results to the local variable x .

And, after the call returns,

4. Code for transferring the return value from the local variable x to the register or stack location used for passing function results.
5. Code for restoring from the stack the subset of the callee-saves registers that were used in the body of the current function. This may include the return address if this is passed in a callee-saves register.
6. Code for freeing space on the stack the space used for storing variables, return addresses, and other things.
7. A jump to the return address.

We then note that we transfer the result of the call to f from the location used for function results to x only to immediately move x back to this location. Since x is not used afterwards, we can eliminate both these steps to get:

1. Code for transferring the parameters from the local variables a_1, \dots, a_n to the registers or stack locations used to pass parameters.
2. Code for storing the return address, and jumping to the function label f .

And, after the call returns,

3. Code for restoring from the stack the subset of the callee-saves registers that were used in the body of the current function. This may include the return address if this is passed in a callee-saves register.
4. Code for freeing space on the stack the space used for storing variables, return addresses, and other things.
5. A jump to the return address.

If registers are used to pass parameters, these registers are almost invariably caller-saves registers. Assuming this, we can move the restoration of the callee-saves registers and the freeing of stack space up to before the jump to f . If the return address is passed in a callee-saves register (the link register), we must, for now, postpone restoration of this until after the call to f . After moving restoration of callee-saves registers to before the call, the code sequence looks like this:

1. Code for transferring the parameters from the local variables a_1, \dots, a_n to the registers or stack locations used to pass parameters.
2. Code for restoring from the stack the subset of the callee-saves registers that were used in the body of the returning function. This does *not* include the register (if any) used to pass the return address.
3. Code for freeing space on the stack the space used for storing variables and other things. This does not include the space for storing the register (if any) used to pass the return address.
4. Code for storing the return address, and jumping to the function label f .

And, after the call returns,

5. If the return address was passed in a callee-saves register, code for restoring this from the stack and freeing the space for it.
6. A jump to the return address.

As noted in the beginning of this section, the central idea in tail-call optimisation is to reuse the current return address instead of adding a new return address. We are now ready to apply this optimisation: Since we don't need the return address after calling f (since we don't intend to return to the code after the call), we can free also the space used to store the old return address, and instead of storing a new return address, we re-use the old. The code below applies these optimisations:

1. Code for transferring the parameters from the local variables a_1, \dots, a_n to the registers or stack locations used to pass parameters.
2. Code for restoring from the stack the subset of the callee-saves registers that were used in the body of the returning function. This *can* include the return address register.
3. Code for freeing space on the stack the space used for storing variables and other things. This *can* include the space used to store the return address.
4. Code for storing the *old* return address, and jumping to the function label f .

Note that nothing now happens after the jump to f .

If the return address is passed in a link register, restoring the old return address from the stack will place it in this register, so no explicit store is required before the jump to f . If the return address is passed on the stack, it will usually already be in the right place when we jump to f . So the code for freeing the space for the old return address and storing it again on the stack can in most cases also be eliminated.

If a_1, \dots, a_n are already allocated to the registers used for parameter passing, and no callee-saves registers need restoring, all that is needed is the jump to f .

Functional languages use recursion with tail calls (tail recursion) instead of loops, so tail-call optimisation is important. Compilers for functional languages allocate as many variables as they can in caller-saves registers, so the situation above where a call can be implemented by just a jump instruction is a common case.

Conversely, compilers for non-functional languages like C and Java often omit general tail-call optimisation because programmers are expected to use loops when full recursion is not required. But even in such languages, tail calls are common enough to merit tail-call optimisation.

9.12.2.1 Tail-Recursion Optimisation

If, in a tail call, the caller and callee is the same function, we can optimise even more: If we jump not to the function label f but instead to right after the point in the prelude where the callee-saves registers are saved, we can omit restoring the callee-saves registers before the jump. Also, instead of moving the arguments a_1, \dots, a_n to the registers or stack locations used for parameter passing, we can move them directly to the local variables used for the incoming parameters, and then also skip the step from the prelude that moves parameters from the standard parameter-passing locations to local variables. This is called *tail-recursion optimisation*. Tail-recursion optimisation can be done already in the intermediate language before translation to machine language: If a function with header $f(a_1, \dots, a_n)$ contains the sequence $x := \text{CALL } f(b_1, \dots, b_n), \text{ RETURN } x$, we can add the instruction LABEL ENTRY to the start of the function body, and replace the call-return sequence above by $a_1 := b_1, \dots, a_n := b_n, \text{ GOTO ENTRY}$.

If b_i is the same as a_j where $j < i$, the assignment $a_j := b_j$ will overwrite b_i , which we will need later. So we might have to reorder the assignments or use a temporary variable to hold the value of b_i until it is used. A simple strategy is to use temporary variables for all the assignments: $t_1 := b_1, \dots, t_n := b_n$ followed by $a_1 := t_1, \dots, a_n := t_n$ will be safe. If any of the temporary assignments are unnecessary, they will usually be eliminated by the register allocator or by later optimisations.

Many compilers support tail-recursion optimisation even if they don't support general tail-call optimisation.

9.12.2.2 Identifying Tail Calls

Above, we said that a tail call is when a CALL instruction is immediately followed by a RETURN instruction in the intermediate code. This is, however, a fairly restrictive definition. A more general definition is that a tail call is when nothing *observable* happens between a CALL instruction and a RETURN instruction. Observable in this context is anything that can affect visible behaviour of the program. For the interme-

diate language we use, this means that we allow a sequence of unconditional jumps between the CALL instruction and the RETURN instruction. We could extend this to include assignments to dead variables (as these are not observable), but it is easier to assume that such assignments have already been eliminated before the intermediate language is translated to machine language.

For example, in the C function

```
int f(int x) {
    if (x<0)
        x = abs(x);
    else
        x = x + x;
    return x;
}
```

the call to `abs` is a tail call, as it is followed only by an unconditional jump to the end of the if-then-else statement which is immediately followed by the `return` statement.

However, the second call to `abs` in the C function

```
int g(int x) {
    while (abs(x) > 10)
        x = abs(x/2);
    return x;
}
```

is *not* a tail call, as it will be followed by the loop test and, possibly, further executions of the loop body. So, in a high-level language, textual adjacency of a call and a return does not imply that the call is a tail call, nor does textual distance imply that the call is not a tail call.

Suggested exercises: 9.4.

9.13 Further Reading

Calling conventions for various architectures are usually documented in the manuals provided by the vendors of these architectures. For example, the calling convention of the ARM processor is described in [4]. Additionally, the calling convention for the RISC-V microprocessor is shown in [1].

Functions declared locally inside other functions require more complex mechanisms than described above, especially if they can be passed as arguments or returned as function values. See [2, 3, 5] for how this can be done.

9.14 Exercises

Exercise 9.1 In Sect. 9.3 an optimisation is mentioned whereby each parameter is stored in the new frame as soon as it is evaluated instead of just before the call. It is mentioned that the required early modification of *SP* can give complications for parameter expressions that need to access the frame.

An alternative is to use an extra pointer *FP* (frame pointer), which points to a fixed place in the frame. All local variables are addressed at offsets to *FP* instead of *SP*, so modification to *SP* does not complicate access to variables stored in the frame.

Modify the frame layout in Fig. 9.1 to include *FP* and modify the prologue, epilogue and call sequences from Figs. 9.2, 9.3 and 9.4 to use this modified layout. Make sure that the value of *FP* is not “lost” across a function call.

Exercise 9.2 Find documentation for the calling convention of a processor of your choice and modify Figs. 9.9, 9.10, 9.11, 9.12 and 9.13 to follow this convention.

Exercise 9.3 Many functions have a body consisting of an if-then-else statement or expression, where one or both branches use only a subset of the variables used in the body as a whole. As an example, assume the body is of the form

```

IF cond THEN label1 ELSE label2
LABEL label1
code1
GOTO label3
LABEL label2
code2
LABEL label3
RETURN x

```

The condition *cond* is a simple comparison between variables (which may or may not be callee-saves).

A normal callee-saves strategy will in the prologue save (and in the epilogue restore) all callee-saves registers used in the body. But since only one branch of the if-then-else is taken, some registers are saved and restored unnecessarily.

We can, as usual, from the register allocator get information about variable use in the different parts of the body (i.e., *cond*, *code*₁ and *code*₂).

We will now attempt to combine the prologue and epilogue with a function body of the above form in order to reduce the number of *callee-saves* registers that need to be saved.

Place the code in Figs. 9.11 and 9.12 around the above body. Then modify the combined code so parts of saving and restoring registers *R4–R12* and *R15* is moved into the branches of the if-then-else structure. Be precise about which registers are saved and restored where. You can use clauses like “if used in *code*₁”.

Exercise 9.4 In Sect. 9.12.2, we identified a tail call as an instruction sequence of the form $x := \text{CALL } f(a_1, \dots, a_n), \text{ RETURN } x$, where the result of the call to f is immediately returned.

- a. Explain why tail-call optimisation can not be applied to a sequence such as $x := \text{CALL } f(a_1, \dots, a_n), \text{ RETURN } y$, where the result x of the call to f is not used, and a different value (y) is returned.
- b. Some functions do not return any result. In C, this is indicated by the return type `void` and in Pascal by using a `procedure` declaration instead of a `function` declaration. In the intermediate language described in Fig. 6.1, all function calls return values, so procedures and functions without results just return arbitrary values which are ignored by their callers. Let us say that we are compiling a function/procedure without result, and we get the call-return sequence from above, then we know that the result y of the return statement above is ignored by all callers. Can tail-call optimisation be applied now? Justify your answer.

References

1. Calling Convention (2015). <https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf>
2. Appel, A.W.: *Compiling with Continuations*. Cambridge University Press (1992)
3. Appel, A.W.: *Modern Compiler Implementation in ML*. Cambridge University Press (1998)
4. ARM Limited: Procedure call standard for the ARM architecture (2009). http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042d/IHL0042D_aapcs.pdf
5. Mogensen, T.Æ.: *Programming Language Design and Implementation*. Springer, Cham (2022). <https://link.springer.com/book/10.1007/978-3-031-11806-7>

Chapter 10

Data-Flow Analysis and Optimisation



*It is a capital mistake to theorise before one has data.
Sir Arthur Conan Doyle (1859–1930)*

Most compilers perform some optimisations on the code that generated by the main phases of the compiler. These optimisations often follow a common recipe: First, the code is analysed to find opportunities for applying a specific optimisation, and then the optimisation is applied in the instances that are found.

Usually, an instance found by the analysis is a small collection of instructions that can be replaced by a shorter or faster collection of instructions. Each such replacement may give only a modest improvement, but if many such instances are found, the combined improvement can be substantial.

The instructions that form an instance for optimisation need not be close to each other, and the information needed to enable the optimisation may be collected from a large piece of code, usually a single procedure but sometimes even the entire program. The analysis that collects the information is called a *data-flow analysis*.

10.1 Data-Flow Analysis

As the name indicates, a data-flow analysis analyses flow of information through a program. This information can be an approximation of values calculated during execution, but it can also be information about where values are stored, if values are going to be used later during execution, information about whether a piece of code is ever going to be executed, or any kind of information that can be useful for optimisation.

Basically, the data-flow analysis tries for all points in the program to answer the question “Is it possible to safely apply a specific optimisation here?”. In many cases, the answer to such questions is formally undecidable, so any finite-time analysis will sometimes have to answer “I don’t know”. Since we only apply an optimisation if the answer is “definitely safe”, it doesn’t make any difference to us whether the analysis answers “I don’t know” or “definitely unsafe”. Hence, many analyses do not distinguish between the latter two answers, so the possible answers returned by the analyses are “definitely safe” and “possibly unsafe”.

Though data-flow analysis and the optimisations enabled by data-flow analysis can be done on all levels from source code to machine code, it is most commonly done on intermediate code, both because the intermediate code is usually simpler than both source code and machine code, and because optimisations for intermediate code can be shared by all compilers that use this intermediate code. We will use the intermediate language described in Sect. 6.2.

10.2 How to Design a Data-Flow Analysis

The first step in designing a data-flow analysis is to identify opportunities for optimisation. Usually, the intermediate-code generation phase generates code from each piece of the program largely independently of the other pieces of the program, so sometimes the code is overly general: It has to work regardless of what happens in the rest of the program. So we can often identify cases where knowledge about the rest of the program (or function) can allow more specialised code to be generated. Rather than identifying and optimising such cases during code generation, we can generate general, context independent code that works for all cases, and subsequently optimise this code through special cases that are found by data-flow analysis. We will illustrate this approach with some examples in the sections below.

In most cases, data-flow analysis is local to a single function body, partly because analysis of a whole program can be very costly, and partly because a compiler may only compile a single module at a time and, hence, not have information about all the functions called from the module, nor about all calls to functions inside the module. We will, nevertheless, in Sect. 10.9 look at an analysis that is often done for a whole program.

10.3 Liveness Analysis

We have already looked at liveness analysis in Chap. 8. Liveness analysis is in most compilers used for register allocation [4], and in Chap. 7, we saw use of liveness information when combining multiple intermediate-language instructions into a single machine-code instruction. But liveness information can be used for other forms of optimisations as well, including:

- If x is dead after the assignment $x := e$, and e has no side effects, the assignment can be eliminated. Note that any access to memory is considered a side effect, as memory access can fail or be used to control memory-mapped devices. Similarly, function calls are considered side effects because functions can access memory. If arithmetic operations trap on overflow, even addition can have a side effect. Also, for security reasons a variable x that holds secret data should be cleared before a program exits, so in such cases the assignment $x := 0$ should not be removed. A compiler flag might be used to indicate that clearing of variables should remain (or even be enforced).
- An instruction sequence $x := e; y := x$ can be shortened to $y := e$ if x is dead afterwards. This can be generalised to other cases such as $x := k; y := z + x$ being shortened to $y := z + k$ if x is dead afterwards. This is similar to the instruction selection we did in Chap. 7.

10.3.1 Improving Liveness Analysis

Even if x is dead after the assignment $x := y + z$, the analysis described in Chap. 8 will generate liveness for y and z . But (unless the addition traps on overflow), we can eliminate the assignment, which would cause y and z to be potentially dead (depending on what happens to them after the assignment).

After eliminating the assignment, we can redo liveness analysis to determine if y and z are dead. If they are found to be so, and we therefore eliminate assignments to them, we get more potentially dead variables, so we would have to redo liveness analysis again, and so on. This can not continue forever, as we only redo liveness analysis if an assignment is eliminated, and that can only be done a finite number of times. But it would be more convenient if we could find all eliminable assignments in a single analysis.

Such cascading effects are common: One optimisation can trigger opportunities for more optimisations, so after doing optimisations, it might be worthwhile to repeat data-flow analysis to see if more optimisation is possible. This is not restricted to repeating the same analysis—some optimisations can benefit from other optimisations. It is possible to keep repeating all optimisations until no changes occur, but that can be costly. Hence, it is often worthwhile to make an analysis that finds such cascading effects in a single analysis.

We can do this for liveness analysis by defining a weaker criterion for being dead:

Definition 10.1 A variable x is *weakly dead* at a program point if the only uses of x after that program point are in assignments to weakly dead variables, and where the expressions on the right-hand sides of these assignments do not have side effects. A variable that is not weakly dead is called *strongly live*.

An analysis for strong liveness can use the same definitions of $kill[i]$, $gen[i]$ and $out[i]$ as normal liveness analysis, but we will have to modify Eq. 8.1 for assignments of the form $x := e$:

$$in[i] = \begin{cases} gen[i] \cup (out[i] \setminus kill[i]), & x \in out[i] \text{ or } e \text{ has side effects} \\ out[i], & x \notin out[i] \text{ and } e \text{ has no side effects} \end{cases} \quad (10.1)$$

For all other instructions, we use Eq. 8.1 unchanged.

Register allocation is not safe if strong liveness is used instead of normal liveness *unless* we eliminate all assignments of the form $x := e$, where x is weakly dead and e has no side effects, because otherwise the assignment might overwrite a live variable that shares a register with x . But if we remove such assignments immediately after liveness analysis, we *can* subsequently use the results of the strong liveness analysis for graph colouring.

10.4 Generalising from Liveness Analysis

The liveness analysis described in Chap. 8 consists of several steps:

1. Find information about which instructions can follow others, i.e., the successors $succ[i]$ of each instruction i .
2. For each instruction i , find $gen[i]$ and $kill[i]$ sets that describe how data-flow information is created and destroyed by the instruction.
3. Define equations for $in[i]$ and $out[i]$ sets by describing how data-flow information flows through and between instructions.
4. Initialise the $in[i]$ and $out[i]$ sets.
5. Do fixed-point iteration to find the minimal solution to the data-flow equations.

We will use the same template for other data-flow analyses, but the details might differ. For example:

1. Some analyses require information about the predecessors of an instruction instead of its successors. We define $pred[i]$ (the predecessors of instruction i) by the bi-implication $j \in pred[i] \Leftrightarrow i \in succ[j]$.
2. Where liveness analysis uses sets of variables, other analyses might use sets of instructions, sets of variable/value pairs, sets of labels, or some other information about the code.
3. The equations for $in[i]$ and $out[i]$ might differ. For example, they may use intersection instead of union to combine information from several successors or predecessors of an instruction, and they might depend on instruction type (as it did in the analysis for strong liveness).
4. Where liveness analysis initialises $in[i]$ and $out[i]$ to empty sets, other analyses might initialise the sets to, for example, the set of all instructions in the function. If we want the minimal solution to the equations, we initialise with empty sets,

but if we want the maximal solution to the equations, we initialise with the set of all relevant values. See appendix A for more details about minimal and maximal solutions to set equations.

5. The optimal order in which instructions are visited during fixed-point iteration may differ.

We will see examples of such variations of the general theme in the following sections.

10.5 Common Subexpression Elimination

After translation to intermediate code, there might be several occurrences of the same calculations, even when this is not the case in the source program. For example, the assignment `a[i] := a[i]+19` can in many languages not be simplified at the source level, but we may be able to optimise the generated code. The assignment might be translated to the following intermediate-code sequence:

```
t1 := 8*i
t2 := a+t1
t3 := M[t2]
t4 := t3+19
t5 := 8*i
t6 := a+t5
M[t6] := t4
```

Note that both the multiplication by 8 and the addition of the result to `a` is repeated, and while we can easily see that the expressions `a+t1` and `a+t5` have the same value, they are not textually identical. Also, it is only because `i` is unchanged between the assignments to `t1` and `t5` that the two occurrences of `i*8` calculate the same value. So identifying expressions that always evaluate to the same value is not just a matter of comparing them as text.

Our ultimate goal is to eliminate both the second calculation of `8*i` and the calculation of `a+t5`, but we will start by a simpler analysis that only eliminates the second occurrence of `8*i`, and then discuss how the simple analysis can be extended to also eliminate `a+t5`.

10.5.1 Available Assignments

If we want to replace an expression by a variable that holds the value of the expression, we need to keep track of the set of expressions that are *available* (i.e., already

calculated), and what variables hold the values of these expressions. At each program point, we want to have a set of pairs of expressions and variables, where the expressions are available and their values contained in the variables to which they are paired. Since each such expression/variable pair originates in an assignment of the expression to the variable, we can, equivalently, use a set of assignment instructions. We call this analysis *available assignments*.

It is clear that information flows forwards from assignments to uses of expressions, so for each instruction i in the program, $in[i]$ is the set of assignments available *immediately before* the instruction is executed, and $out[i]$ set is the set of assignments available *immediately after* the instruction is executed. $gen[i]$ and $kill[i]$ should, hence, describe which new assignments become available and which assignments are no longer available. An assignment $x := e$ makes itself available *unless* the variable on the left-hand side also occurs on the right-hand side (because the assignment would make a later occurrence of the expression have a different value). All assignments $y := e'$ (apart from $x := e$) where $x = y$ or x occurs in e' are invalidated:

- An assignment $x := e'$ will after the assignment $x := e$ no longer be available, because x no longer contains the value of e' .
- If x occurs in an expression e' , then an assignment $y := e'$ is no longer available, as the value of e' changes, so y no longer contains the current value of e' .

Figure 10.1 shows the gen and $kill$ sets for each kind of instruction in the intermediate language.

Note that a copy instruction $x := y$ does not generate any available assignment, as replacing a later occurrence of y by x is not an optimisation. Note, also, that any store of a value to memory kills *all* instructions that load from memory. We need to make this rather conservative assumption because we do not know which memory locations loads and stores affect. Similarly, we assume that a function call can change any memory location.

The next step is to define the equations for in and out sets:

$$in[i] = \bigcap_{j \in pred[i]} out[j] \quad (10.2)$$

$$out[i] = gen[i] \cup (in[i] \setminus kill[i]) \quad (10.3)$$

As mentioned above, the assignments that are available after an instruction (i.e., in the out set) are those that are generated by the instruction and those that were available before, except for those that are killed by the instruction. The available assignments before an instruction (i.e., in the in set) are those that are available at *all* predecessors, so we take the intersection of the sets available at the predecessors.

Note that, compared to liveness analysis, the roles of in and out are reversed, since available assignments is a forwards analysis, where liveness analysis is a backwards analysis.

We need to initialise the in and out sets for the fixed-point iteration. We want to find the maximal solution to the equations: Consider a loop where an assignment is

Instruction i	$gen[i]$	$kill[i]$
LABEL l	\emptyset	\emptyset
$x := y$	\emptyset	$assg(x)$
$x := k$	$\{x := k\}$	$assg(x)$
$x := \mathbf{unop} y$ where $x \neq y$	$\{x := \mathbf{unop} y\}$	$assg(x)$
$x := \mathbf{unop} x$	\emptyset	$assg(x)$
$x := \mathbf{unop} k$	$\{x := \mathbf{unop} k\}$	$assg(x)$
$x := y \mathbf{binop} z$ where $x \neq y$ and $x \neq z$	$\{x := y \mathbf{binop} z\}$	$assg(x)$
$x := y \mathbf{binop} z$ where $x = y$ or $x = z$	\emptyset	$assg(x)$
$x := y \mathbf{binop} k$ where $x \neq y$	$\{x := y \mathbf{binop} k\}$	$assg(x)$
$x := x \mathbf{binop} k$	\emptyset	$assg(x)$
$x := M[y]$ where $x \neq y$	$\{x := M[y]\}$	$assg(x)$
$x := M[x]$	\emptyset	$assg(x)$
$x := M[k]$	$\{x := M[k]\}$	$assg(x)$
$M[x] := y$	\emptyset	$loads$
$M[k] := y$	\emptyset	$loads$
GOTO l	\emptyset	\emptyset
IF $x \mathbf{relop} a$ THEN l_t ELSE l_f	\emptyset	\emptyset
$x := \mathbf{CALL} f(args)$	\emptyset	$assg(x) \cup loads$
RETURN x	\emptyset	\emptyset

where $assg(x)$ is the set of all assignments in which x occurs on either left-hand or right-hand side, and $loads$ is the set of all assignments of the form $z := M[\cdot]$ for any variable z .

Fig. 10.1 Gen and kill sets for available assignments

available before the loop and no variable in the assignment is changed inside the loop. We would want the assignment to be available inside the loop, but if we initialise to the empty set the *out* set of the jump from the end of the loop to its beginning, the intersection of the assignments available before the loop and those available at the end of the loop will be empty, and remain that way throughout the iteration. So, instead we initialise the *in* and *out* sets for all instructions—except the first—to the set of all assignments in the program (so we find the maximal solution). The *in* set for the first instruction remains empty, as no assignment is available at the beginning of the program.

Generally, when a data-flow analysis uses the intersection of the values for the predecessors/successors, we will always initialise sets (except for instructions with no predecessors/successors) to the largest possible set, so we do not get overly conservative results for loops. If a data-flow analysis uses union of the values for the predecessors/successors, we will always initialise sets (except for instructions with no predecessors/successors) to the smallest possible set, which is the empty set.

10.5.2 Example of Available-Assignments Analysis

Figure 10.2 shows a function body that doubles all elements of an array p .

Figure 10.3 shows $pred$, gen and $kill$ sets for each instruction in this program. We represent an assignment by the number of the assignment instruction, so gen and $kill$ sets are sets of numbers. We will, however, identify textually identical assignments with the same number, so both the assignment in instruction 2 and the assignment in instruction 13 are identified with the number 2, as can be seen in the gen set of instruction 13 and by the fact that no instruction kills instruction 13.

Note that each assignment kills itself (or its alias), but since it also in most cases generates itself, the net effect in these cases is to remove all conflicting assignments (including the assignment itself) and then adding the assignment itself. Assignment 12 ($i := i + 1$) does not generate itself (or anything else), since i also occurs on the right-hand side. Note, also, that the memory write in instruction 11 kills instruction 7, as this loads from memory.

For the fixed-point iteration, we initialise the in set of instruction 1 to the empty set and all other in and out sets to the set of all assignments that are actually generated by some instructions, i.e., $\{1, 2, 5, 6, 7, 8, 9, 10\}$. We then iterate Eqs. 10.2 and 10.3 as assignments until we reach a fixed-point. Since information flow is forwards, we process the instructions by increasing number and calculate $in[i]$ before $out[i]$. The iteration is shown in Fig. 10.4. For space reasons, the final iteration (which is identical to iteration 2) is not shown.

Fig. 10.2 Example code for available-assignments analysis

```

1:  $i := 0$ 
2:  $a := n * 3$ 
3: IF  $i < a$  THEN loop ELSE end
4: LABEL loop
5:  $b := i * 8$ 
6:  $c := p + b$ 
7:  $d := M[c]$ 
8:  $e := d * 2$ 
9:  $f := i * 8$ 
10:  $g := p + f$ 
11:  $M[g] := e$ 
12:  $i := i + 1$ 
13:  $a := n * 3$ 
14: IF  $i < a$  THEN loop ELSE end
15: LABEL end
16: RETURN  $p$ 

```

i	$pred[i]$	$gen[i]$	$kill[i]$
1		1	1, 5, 9, 12
2	1	2	2
3	2		
4	3, 14		
5	4	5	5, 6
6	5	6	6, 7
7	6	7	7, 8
8	7	8	8
9	8	9	9, 10
10	9	10	10
11	10		7
12	11		1, 5, 9, 12
13	12	2	2
14	13		
15	3, 14		
16	15		

Fig. 10.3 $pred$, gen and $kill$ for the program in Fig. 10.2

i	Initialisation		Iteration 1		Iteration 2	
	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$
1		1,2,5,6,7,8,9,10		1		1
2	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1	1,2	1	1,2
3	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2	1,2	1,2	1,2
4	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2	1,2	2	2
5	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2	1,2,5	2	2,5
6	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5	1,2,5,6	2,5	2,5,6
7	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6	1,2,5,6,7	2,5,6	2,5,6,7
8	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,7	1,2,5,6,7,8	2,5,6,7	2,5,6,7,8
9	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,7,8	1,2,5,6,7,8,9	2,5,6,7,8	2,5,6,7,8,9
10	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9	1,2,5,6,7,8,9,10	2,5,6,7,8,9	2,5,6,7,8,9,10
11	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,8,9,10	2,5,6,7,8,9,10	2,5,6,8,9,10
12	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	1,2,5,6,8,9,10	2,6,8,10	2,5,6,8,9,10	2,6,8,10
13	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	2,6,8,10	2,6,8,10	2,6,8,10	2,6,8,10
14	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	2,6,8,10	2,6,8,10	2,6,8,10	2,6,8,10
15	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	2	2	2	2
16	1,2,5,6,7,8,9,10	1,2,5,6,7,8,9,10	2	2	2	2

Fig. 10.4 Fixed-point iteration for available-assignment analysis

10.5.3 Using Available Assignment Analysis for Common Subexpression Elimination

If instruction i is of the form $x := e$ for some expression e , and $in[i]$ contains an assignment $y := e$, then we can replace $x := e$ by $x := y$.

If we apply this idea to the program in Fig. 10.2, we see that, at instruction 9 ($f := i * 8$), we have assignment 5 ($b := i * 8$) available, so we can replace instruction 9 by ($f := b$). At instruction 13 ($a := n * 3$), we have assignment 2 ($a := n * 3$) available,

Fig. 10.5 The program in Fig. 10.2 after common subexpression elimination

```

1:  $i := 0$ 
2:  $a := n * 3$ 
3: IF  $i < a$  THEN loop ELSE end
4: LABEL loop
5:  $b := i * 8$ 
6:  $c := p + b$ 
7:  $d := M[c]$ 
8:  $e := d * 2$ 
9:  $f := b$ 
10:  $g := p + f$ 
11:  $M[g] := e$ 
12:  $i := i + 1$ 
14: IF  $i < a$  THEN loop ELSE end
15: LABEL end
16: RETURN  $p$ 

```

so we can replace instruction 13 by $a := a$, which we can eliminate entirely as it is a no-operation. The optimised program is shown in Fig. 10.5.

Note that, while we could eliminate identical expressions, we could not eliminate the expression $p + f$ in instruction 10, even though it (because $b = f$) has the same value as the right-hand side of the available assignment 6 ($c := p + b$). A way of eliminating this recomputation also is to first replace all uses of f by b (which we can do since the only assignment to f is $f := b$), and then repeat common subexpression elimination on the resulting program. If a large source-level expression has multiple occurrences, we might have to repeat this a large number of times to get the optimal result. An alternative is to keep track of sets of variables that have the same value (a technique called *value numbering*), which allows large common subexpressions to be eliminated in one pass.

Another limitation of the available assignment analysis is when two different predecessors to an instruction have the same expression available, but in different variables, e.g., if one predecessor of instruction i has the available assignments $\{x := a + b\}$ and the other predecessor has the available assignments $\{y := a + b\}$. These sets have an empty intersection, so the analysis would show no available assignments at the entry of instruction i . One way to make the expression $a + b$ available in this situation is to replace all instructions of the form $x := a + b$ by the sequence $v721 := a + b; x := v721$, where $v721$ is a variable name generated by hashing the expression $a + b$ (so identical expressions generate the same variable name). This way, all assignments that have the same expression on the right-hand side will also have the same variable on the left-hand side, so if an expression is available in all predecessors of an instruction, identical assignments using this expression are available in all predecessors too. If, after common subexpression elimination is performed, $v721$ is dead at the end of the sequence $v721 := a + b; x := v721$, we

can combine the two instructions into $x := a + b$ again. Alternatively, we can leave it to the register allocation phase to allocate x and $v721$ in the same register, so the assignment $x := v721$ can be eliminated.

Suggested exercises: 10.1, 10.2.

10.6 Index-Check Elimination

When a programming language requires bounds-checking of array accesses, the compiler must insert tests before each array access to verify that the index is in range. Index-check elimination aims to remove these index-in-range checks where they are guaranteed to not fail, because the validity of the condition in the check is implied by previous assignments or conditional jumps.

To find unnecessary tests, we collect for each program point a set of inequalities that hold at this point. If the validity of a bounds check is implied by this set of inequalities, we can eliminate it.

We use the conditions in IF-THEN-ELSE instructions as a source for inequalities. Note that, since index checks are translated into such instructions, this set of conditions includes all index checks as well as conditions that are found in the source code of the program.

Conditions in our intermediate language all have the form $x \text{ **relop** } p$, where x is a variable, and p is either a constant or a variable. We only consider inequalities, i.e., conditions of the form $p < q$ or $p \leq q$, where p and q are either variables or constants (but not both constants). Each condition that occurs in the program is translated into a set of inequalities of this form. For example, the equality test $x = y$ is translated into the inequalities $x \leq y$ and $y \leq x$. To gain information from a condition that evaluates to false, we also generate inequalities for the negation of each condition found in the program, so the condition $x < 10$ generates the inequalities $x < 10$ and $10 \leq x$. A condition $x = y$ generates no inequalities for its negation, as $x \neq y$ can not be expressed as a conjunction of inequalities. Similarly, the condition $x \neq y$ generates only inequalities for its negation. The set of inequalities generated by all conditions and their negations gives us a universe Q of inequalities for the analysis.

At each point in the program, we want to find which of the inequalities from Q that are guaranteed to hold at this point. When the analysis is complete, we see if a condition in an IF-THEN-ELSE instruction is implied to be true or implied to be false by the inequalities that hold at entry to the instruction. If this is the case, we can replace the IF-THEN-ELSE instruction with an unconditional jump. If, furthermore, that unconditional jump is to an immediately following label, the jump can be eliminated entirely.

To find equalities that hold, we exploit that, when executing the instruction IF c THEN L_1 ELSE L_2 , those inequalities from Q that are implied by c will be true if we jump to L_1 , and those inequalities from Q that are implied by the negation of

c will be true if we jump to L_2 . Also, after an assignment $x := 0$, we know that the inequalities from Q of the form $x \leq k$ will hold if $0 \leq k$, inequalities of the form $x < k$ will hold if $0 < k$, inequalities of the form $k \leq x$ will hold if $k \leq 0$, and inequalities of the form $k < x$ will hold if $k < 0$, but no other inequalities from Q involving x will be valid after this assignment. So assignments can both add and remove inequalities. Conditional jumps can add (but not remove) inequalities, but the added inequalities depend on which branch is taken. So, where most instructions have a single *kill* and *gen* set of inequalities that are killed and generated by the instruction, jumps have one *gen* set (called gen_L) per target label L (but a single, empty *kill* set). Similarly, all instructions will have a single set $in[i]$ of inequalities that hold at entry to the instruction, but while most instructions also have a single $out[i]$ set, jumps have an $out_L[i]$ for each exit label L that the jump may jump to. So the equations for $in[i]$ and $out[i]$ are different for different instructions:

$$\begin{aligned}
 in[i] &= \bigcap_{j \in pred[i]} out_L[j], & \text{if } i \text{ is LABEL } L \\
 in[i] &= out[pred[i]], & \text{otherwise} \\
 out_L[i] &= in[i], & \text{if } i \text{ is GOTO } L \\
 out_L[i] &= in[i] \cup gen_L[i], & \text{if } i \text{ is IF } c \text{ THEN } p \text{ ELSE } q \text{ and } L \in \{p, q\} \\
 out[i] &= gen[i] \cup (in[i] \setminus kill[i]), & \text{otherwise}
 \end{aligned}$$

To make this work, labels should only be reachable by jumps, so if a predecessor to a LABEL instruction is not a jump, an unconditional jump is inserted. An example of this can be seen in Fig. 10.7, where an otherwise unnecessary jump is inserted before instruction 3. These jumps can be removed again after the index-check elimination is completed. Other analyses may also benefit from having these unconditional jumps instead of fall-through to labels, so it is common to insert them before all optimisation passes and only remove them after all these are complete. We will see another example of needing these unconditional jumps in Sect. 10.11.

Figure 10.6 shows $gen[i]$, $gen_L[i]$ and $kill[i]$ for different types of instructions. We use the following auxiliary definitions:

- $when(Q, c)$ is the set of inequalities in Q implied by the condition c .
- $whennot(Q, c)$ is the set of inequalities in Q implied by the negation of the condition c .
- $conds(Q, x)$ is the set of inequalities from Q that involve x .
- $equal(Q, x, p)$, where p is a variable or a constant, is the set of inequalities from Q that are implied by the equality $x = p$. For example, if $Q = \{x < 10, 10 \leq x, 0 < x, x \leq 0\}$ then $equal(Q, x, 7) = \{x < 10, 0 < x\}$.
- $upper(Q, x)$ is the set of inequalities from Q that puts an upper bound on x , i.e., have the form $x < p$ or $x \leq p$, where p is a variable or a constant.
- $lower(Q, x)$ is the set of inequalities from Q that puts a lower bound on x , i.e., have the form $p < x$ or $p \leq x$, where p is a variable or a constant.

In most cases, an assignment to a variable invalidates all inequalities involving that variable, but we have made some exceptions: If we assign a constant or variable to

Instruction i	$gen[i]$	$gen_L[i]$	$kill[i]$
LABEL A	\emptyset	—	\emptyset
GOTO A	—	$gen_A[i] = \emptyset$	\emptyset
IF c THEN A ELSE B	—	$gen_A[i] = when(Q, c)$ $gen_B[i] = whennot(Q, c)$	\emptyset
$x := p$	$equal(Q, x, p)$	—	$conds(Q, x)$
$x := x + k$	\emptyset	—	$upper(Q, x)$
$x := x - k$	\emptyset	—	$lower(Q, x)$
$x := e$	\emptyset	—	$conds(Q, x)$
$M[p] := e$	\emptyset	—	\emptyset
RETURN e	\emptyset	—	\emptyset

where k is a constant greater than 0, p is either a variable or a constant, and e is any expression not covered by the previous rules. — means that the set is undefined (and not used). Generally, $gen[i]$ and $gen_L[i]$ are not both defined.

Fig. 10.6 gen and $kill$ sets for index-check elimination

a variable, we kill all inequalities involving the variable, but add those inequalities from Q that are implied by the assignment. Also, if x increases, we invalidate all inequalities that bound x from above, but keep those that bound x from below, and if x decreases, we invalidate the inequalities that bound x from below but keep those that bound x from above. We can add more special cases to make the analysis more precise, but the above are sufficient for the most common cases.

We initialise all *in* and *out* sets to Q , except the *in* set for the first instruction, which is initialised to the empty set.

After the data-flow analysis reaches a fixed-point, the inequalities in $in[i]$ are guaranteed to hold at instruction i . So, if we have an instruction i of the form IF c THEN A ELSE B and c is implied by an inequality in $in[i]$, we can replace the instruction by GOTO A . If the negation of c is implied by an inequality in $in[i]$, we can replace the instruction by GOTO B . Note that we only consider when a *single* inequality in $in[i]$ implies a condition, not when the combined set of inequalities do so. The reason for this is that checking if a set of inequalities imply a single inequality is complex and time-consuming, where checking each inequality in $in[i]$ in isolation is simple and fast. It is also sufficient in most cases.

This optimisation can leave some parts of the code unreachable. Such unreachable code can be removed.

We illustrate the analysis by an example. Consider the following for-loop, and assume that the array a is declared to go from 0 to 10.

```
for i := 0 to 9 do
  a[i] := 0;
```

This loop can be translated (with index checks) into the intermediate code shown in Fig. 10.7. Note the otherwise unnecessary jump in instruction 2, which has been added to make all labels accessible only through jumps.

Fig. 10.7 Intermediate code for a for-loop with index check

```

1:  $i := 0$ 
2: GOTO For1
3: LABEL For1
4: IF  $i \leq 9$  THEN For2 ELSE For3
5: LABEL For2
6: IF  $i < 0$  THEN Error ELSE Ok1
7: LABEL Ok1
8: IF  $i > 10$  THEN Error ELSE Ok2
9: LABEL Ok2
10:  $t := i * 8$ 
11:  $t := a + t$ 
12:  $M[t] := 0$ 
13:  $i := i + 1$ 
14: GOTO For1
15: LABEL For3

```

The set Q of possible inequalities in the program are derived from the conditions in the three IF-THEN-ELSE instructions and their negations, so $Q = \{i \leq 9, 9 < i, i < 0, 0 \leq i, 10 < i, i \leq 10\}$.

We leave the fixed-point iteration and check elimination as an exercise to the reader, but note that the assignment $i := 0$ in instruction 1 implies the inequalities $\{i \leq 9, 0 \leq i, i \leq 10\}$, and that the assignment $i := i + 1$ in instruction 13 invalidates the inequalities that bound i from above, so $kill[13] = \{i \leq 9, i < 0, i \leq 10\}$. All other inequalities are unaffected by the assignment.

Suggested exercises: 10.3.

10.7 Jump-to-Jump Elimination

When we have an instruction sequence like

```

LABEL  $L_1$ 
GOTO  $L_2$ 

```

we would like to replace all jumps to L_1 by jumps to L_2 , as this will reduce two consecutive jumps to a single jump. There may be chains of such jumps to jumps, e.g.,

```

LABEL  $L_1$ 
GOTO  $L_2$ 
...
LABEL  $L_2$ 
GOTO  $L_3$ 
...
LABEL  $L_3$ 
GOTO  $L_4$ 

```

Such chains may be generated from nested conditionals and loops. For example, a conditional inside a while-loop might jump to the end of the loop, which immediately jumps to the start of the loop. We want, in the above example, to replace a jump to L_1 with a jump to L_4 directly. To do this, we make a data-flow analysis that for each jump finds its ultimate destination. Rather than keeping separate information for each program point, we maintain a global mapping J that maps each label to a set of labels that is either empty or contains a single label (the final destination). In the example above, we want the analysis to end with $J[L_1] = J[L_2] = J[L_3] = J[L_4] = \{L_4\}$.

J is initialised to map all labels to the empty set. We then repeatedly apply the following rule on all labels in the program until J stabilises: If the instruction LABEL L is immediately followed by the instruction GOTO M , $J[L] := J[M]$. Otherwise, $J[L] := \{L\}$. This will eventually stabilise.

If there is a circular chain of jumps like

```

LABEL  $L_1$ 
GOTO  $L_2$ 
...
LABEL  $L_2$ 
GOTO  $L_3$ 
...
LABEL  $L_3$ 
GOTO  $L_1$ 

```

J will in the fixed-point still map all the involved labels (L_1 , L_2 , L_3) to the empty set. A label that is not part of such an infinite loop is mapped to a singleton set containing the ultimate destination of that label. If $J[L] = \{M\}$, we can replace all jumps to label L by jumps to label M . Furthermore, if $J[L] \neq \{L\}$, we can remove the instruction LABEL L and the following GOTO instruction, as these are never reached. This is an example of dead-code elimination.

Suggested exercises: 10.4.

10.8 Resources Used by Data-Flow Analysis

For most analyses, the size of the sets used in the analysis is proportional to the size of the code that is analysed: The number of different variables is roughly proportional to the size of the code, and the number of definitions or inequalities used by common-subexpression elimination and index-check elimination are also proportional to the code size. Since we usually have *in* and *out* sets for each instruction in the code, the combined size of these sets can be quadratic in the size of the code. So we want to use compact representations of sets.

A commonly used representation is bit vectors: A set is represented by a vector of bits, where each bit position represents an element from the universe of values (e.g., a variable, an assignment, or an inequality). If the bit at the position is 1, the element is in the set, otherwise not. Not only does this give a compact representation of the sets, it also gives fast implementations of set operations such as union, intersection, and set difference, which are used in the equations for *in* and *out* sets: Union is bitwise OR, intersection is bitwise AND, and set difference is bitwise ANDNOT.

In many analyses, a typical set contains far fewer elements than the universe of values. For example, few variables are live for very long, and few assignments are available very long. Bit vectors are very efficient for dense sets, i.e., sets that contain a large fraction of the possible elements, but the advantage is reduced if sets are sparse, i.e., they contain far fewer elements than the universe. Sparse sets are sometimes better represented by linked lists of values or by compressed bit vectors. But since the per-element space and time used by such representations is higher than for dense bit vectors, the sets have to be *very* sparse for this to be an advantage. In a linked list, you normally need 10-50 times the space and time per element compared to dense bit vectors, so it is only for very sparse sets that linked lists are better. Compressed bit vectors are somewhere in between lists and normal bit vectors in both time and space use.

Each iteration of a data-flow analysis updates the *in* and *out* sets for each instruction. Normally, the time used to do a single update is roughly proportional to the sizes of these sets, so a rough estimate of the time used to do a single iteration is the combined size of all the *in* and *out* sets (i.e., quadratic in the size of the code). In the worst case, one iteration changes one set by adding or removing one element. So the worst case number of iterations is also quadratic in the code size. This makes the worst case total time $O(n^4)$, where n is the size of the code. Hence, there is good reason to try to reduce this.¹

We noted already that it is a good idea for an iteration of a data-flow analysis to go through the instructions in the same direction that data flows, i.e., backwards for liveness analysis and forwards for available assignments and index-check elimination. This will propagate changes in the sets much faster than the worst case, so in most cases the number of iterations to find a fixed-point is very small, typically less than six.

¹ It is also a common observation that very few problems are naturally $O(n^4)$, so if an algorithm is $O(n^4)$, it can in all likelihood be optimised to $O(n^3)$ or less.

Instead of using a fixed order of going through instructions, we can use a work-list algorithm: The work list holds instructions that require recalculation of *in* and *out*. It initially holds all instructions, and whenever an *in* or *out* set changes (depending on whether the analysis is forwards or backwards), the successors or predecessors of the instruction are added to the set. For a forwards analysis, a work-list algorithm can look like this:

```

W := [1..n]; /* all instructions in forwards order */
while W is nonempty do
  i := head W; W := tail W;
  recalculate in[i]; recalculate out[i];
  if out[i] changed value then W := succ[i] ++ W;
end

```

where ++ is list concatenation and `succ [i]` is the list of successors to instruction *i*. For backwards analysis, the initial list of instructions would be in backwards order, *out*[*i*] would be calculated before *in*[*i*] and the predecessors of *i* would be added to the work list when *in*[*i*] changes.

Usually, the time used by a work-list algorithm is less than the time used to do three full iterations. But even with a constant number of iterations, the time and space used by data-flow analysis is usually quadratic in the size of the code. This is one reason data-flow analysis is often restricted to one function body at a time instead of analysing the complete program: The average size of functions in a program is mostly independent of the total size of the program, so by doing the analysis per function, the total time is roughly proportional to the size of the whole program. However, some analyses such as pointer analysis (see Sect. 10.9) will not be accurate enough unless they are done across function borders, so they are sometimes done for a whole compilation unit (module) or the whole program.

Suggested exercises: 10.5.

10.9 Pointer Analysis

In the available-assignments analysis described in Sect. 10.5.1, any store of a value to memory invalidates all assignments that load from memory. This is because, not knowing anything about the addresses used in memory operations, any store instruction can potentially overwrite any memory location. So loading from the same location twice can potentially give different values if, between these, a value is stored to memory.

Memory is large, so the chance that one particular store will actually change the value of one particular load is quite small. So it is useful to analyse when two pointers to memory will definitely *not* point to the same location. Such an analysis is called a

pointer analysis or *alias analysis*, as different pointers that point to the same location are called *aliases* for that location.

Since pointers are often passed as arguments or results between functions, a pointer analysis that is local to a single function is of limited usefulness. So we describe an analysis that analyses all functions in a program at the same time.

The main idea of the analysis is that pointers that point to memory that is allocated at different points in the program can not possibly point to the same location. This assumes that pointers never leave the bounds of the allocated chunks of memory they point to. Most programming languages either ensure this with index checks, or define the behaviour of going outside the bounds as undefined (which relieves an optimisation of responsibility for preserving behaviour when this happens), so this is normally a safe assumption.

Memory can be allocated in various ways:

- Global variables are typically allocated to a static address before the program starts.
- A local variable *can* be allocated in memory (usually the stack), and *will* be in memory if its address is taken. In C, the address of a variable can be taken by using the `&` (address of) operator, in other languages like Pascal, the address can be taken by passing the variable as a reference parameter to a function.
- Local arrays or records are usually allocated in memory (often, the stack).
- Memory may be explicitly allocated by calling/applying a constructor or calling a library routine like C's `malloc` function.

In the intermediate language defined in Sect. 6.2, all variables including function parameters are equivalent to registers, so all memory access is explicit. We have already defined instructions for memory access, but since the intermediate language has no access to the stack pointer we can not directly represent stack allocation using the normal instructions. Heap allocation is in many languages done by calling a function, but analysing heap-allocation functions will in all likelihood be so imprecise that all pointers to heap-allocated memory will be classified as aliases. Instead, we will use a pseudo-instruction `alloc x`, which marks that `x` is made to point to freshly allocated memory, so `x` at this point is not aliased to other pointer variables. We won't distinguish whether the allocation is static, on the stack or on the heap, we don't care when (or if) the memory is freed again afterwards, and we don't care about the size of allocated memory blocks. We only assume that offsets from pointers do not create pointers to outside the allocated block, and we assume that allocated blocks are disjoint. In all cases, the number of the instruction that contains an allocation pseudo-instruction is used to identify the allocation point. Addresses may be stored to and retrieved from variables or memory blocks. We use the term "location" as a synonym to an allocation point.

Note that, since the same program point may be reached several times during execution (by looping or recursion), pointers to many different non-overlapping blocks can be allocated by a single program point. So by identifying all blocks that are allocated at a program point, we get only approximate alias information: The analysis

will say that two pointers allocated at the same program point at different times during execution will be aliases, even though they always point to different block. Also, because they are allocated at the same time, two fields of a record or two elements of an array will be classified as aliases, even though they can never be at the same address. In spite of these limitations, the analysis will find many cases where aliasing can never occur.

The pointer analysis tracks sets of pairs each consisting of a variable and an allocation point: If $in[i]$ contains a pair (x, p) , it means that at entry to instruction number i , the variable x can point into memory allocated at the allocation point p (which is also an instruction number). We also keep a global set Mem of pairs of two allocation points: If Mem contains a pair (p, q) , it means that a pointer into something allocated at q may be stored somewhere in a block allocated at p . Additionally, we also have global sets A and R that handle function parameters and return values. If R holds a pair (f, p) , the function f can return a pointer into a block allocated at p . If A holds a triple (f, n, p) , the n th argument of f may be passed a pointer into a block allocated at p .

Rather than defining *gen* and *kill* sets, we define different equations for out , Mem , A , and R for different instruction types. Figure 10.8 shows these. We also need an equation for each function header $f(a_1, \dots, a_n)$:

$$in[1] = \{(a_1, p) \mid (f, 1, p) \in A\} \cup \dots \cup \{(a_n, p) \mid (f, n, p) \in A\}$$

where 1 is the number of the first instruction in the body of f . For all other instructions, the equation for in is

$$in[i] = \bigcup_{j \in pred[i]} out[j]$$

All sets are initialised to the empty set.

Note that we, in the rules for unary and binary operators, assume that if a pointer into a block allocated at p is used in the argument, the result can also be a pointer into this block. This is strictly speaking not true, as, for example, the difference of two pointers is an integer and the negation of a pointer is not a pointer. We can make special cases for certain operators, or we can use type information: If a variable is not of pointer type, it can not point to anything, so an assignment to a non-pointer should not add any pairs to the out set, even if it is given a value that depends on a pointer. This refinement would, however, not be valid in a language like C that allows integers to be converted to pointers. But in “cleaner” languages like Java or SML, it will work.

We can use the results of the pointer analysis to get more precise results from other analyses. For example, in the available assignments analysis in Sect. 10.5.1, any store $M[x] := y$ kills *all* load instructions. With the information gathered by the pointer analysis, we can refine this so only loads from addresses that may be affected by the store instruction are killed.

Instruction i	Equations
LABEL L	$out[i] = in[i]$
$x := y$	$out[i] = \{(x, p) \mid (y, p) \in in[i]\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
$x := k$	$out[i] = \{(x, k)\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
alloc x	$out[i] = \{(x, i)\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
$x := \mathbf{unop} \ y$	$out[i] = \{(x, p) \mid (y, p) \in in[i]\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
$x := \mathbf{unop} \ k$	$out[i] = \{(x, k)\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
$x := y \ \mathbf{binop} \ z$	$out[i] = \{(x, p) \mid (y, p) \in in[i] \vee (z, p) \in in[i]\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
$x := y \ \mathbf{binop} \ k$	$out[i] = \{(x, k)\} \cup \{(x, p) \mid (y, p) \in in[i]\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
$x := M[y]$	$out[i] = \{(x, q) \mid (y, p) \in in[i] \wedge (p, q) \in Mem\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
$x := M[k]$	$out[i] = \{(x, p) \mid (k, p) \in Mem\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
$M[x] := y$	$Mem = Mem \cup \{(p, q) \mid (x, p) \in in[i] \wedge (y, q) \in in[i]\}$ $out[i] = in[i]$
$M[k] := y$	$Mem = Mem \cup \{(k, q) \mid (y, q) \in in[i]\}$ $out[i] = in[i]$
GOTO L	$out[i] = in[i]$
IF $x \ \mathbf{relop} \ y$ THEN T ELSE F	$out[i] = in[i]$
$x := \mathbf{CALL} \ f(a_1, \dots, a_n)$	$A = A \cup \{(f, 1, p) \mid (a_1, p) \in in[i]\} \cup \dots \cup \{(f, n, p) \mid (a_n, p) \in in[i]\}$ $out[i] = \{(x, p) \mid (f, p) \in R\} \cup (in[i] \setminus \{(x, p) \mid (x, p) \in in[i]\})$
RETURN x	$R = R \cup \{(f, p) \mid (x, p) \in in[i]\}$ where f is the current function.

Fig. 10.8 Equations for pointer analysis

Where we had:

Instruction i	$gen[i]$	$kill[i]$
$M[x] := y$	\emptyset	$loads$
$M[k] := y$	\emptyset	$loads$
$x := \mathbf{CALL} \ f(args)$	\emptyset	$assg(x) \cup loads$

we can now refine this to

Instruction i	$gen[i]$	$kill[i]$
$M[x] := y$	\emptyset	$\bigcup \{loads(p) \mid (x, p) \in (in[i] \cup Mem)\}$
$M[k] := y$	\emptyset	$loads(k)$
$x := \mathbf{CALL} \ f(args)$	\emptyset	$assg(x) \cup loads$

where $loads(p)$ is the set of assignments j of the form $j: x := M[a]$, where $(a, p) \in (in[j] \cup Mem)$. Note that, in this context, $in[i]$ and $in[j]$ are the results

found by the pointer analysis, not those (later) found by the available assignments analysis, so we can, using the results of the pointer analysis, compute the kill sets for the available assignments analysis before starting the fixed-point iteration for the available assignments analysis.

While the pointer analysis presented above tracks pointer information across function calls, it does not distinguish between different calls to the same function. For example, if a function is defined as

```
f(x)
[RETURN x]
```

and one call passes a pointer to location a as parameter to f , the analysis will assume that *all* calls to f can return a . It is possible to make analyses that distinguish different calls to the same function, but they are more complex and use more resources.

Another limitation of the shown analysis is that all pointers to the same allocated block of memory are assumed to be potential aliases. But in many cases, they are not. For example, when a record, struct or object containing two or more fields is allocated, pointers to different fields are definitely not aliases, even though they share the same allocation point. It is possible to extend the analysis to consider both the allocation point *and* the offset from the start of the allocated memory, but, again, this makes the analysis more complex and costly.

Suggested exercises: 10.6.

10.10 Limitations of Data-Flow Analyses

All of the data-flow analyses we have seen above are approximations: They will not always reflect accurately what happens at runtime: The liveness analysis may say that a variable is live when it is, in fact, dead, the index-check analysis may fail to remove a redundant index check, the available assignment analysis may say an assignment is unavailable when it really is available, and the pointer analysis may say that two pointers can point to the same location, when they never can.

In all cases, the approximations err on the safe side: It is better to miss an opportunity for optimisation than to make an incorrect optimisation. For liveness analysis, this means that if you don't know whether a variable is live or dead, you had better assume that it is live, as assuming it dead might cause its value to be overwritten when it may actually be needed later on. When available assignment analysis is used for common subexpression elimination, saying that an assignment is available when it is not may make the optimisation replace an expression by a variable that does not always hold the same value as the expression, so it is better to leave an assignment out of the set if you are in doubt. And if a pointer analysis fails to identify two pointers that point to the same location, optimisations made on the assumption that they do not may cause a program to change behaviour.

It can be shown that no compile-time analysis that seeks to uncover nontrivial information about the run-time behaviour of programs can ever be completely exact. You can make more and more complex analyses that get closer and closer to the exact result, but there will always be programs where the analysis is not precise. So a compiler writer will have to be satisfied with analyses that find most cases where an optimisation can be applied, but misses some. In most cases, it is possible to get more precise results by using an analysis that uses more resources, so the choice of analysis is a compromise between precision and resources. The law of diminishing returns applies, so once an analysis finds most of the cases where an optimisation can apply, there is little point in throwing more resources at a more precise analysis. Some compilers give the user a choice between different optimisation levels, where the higher levels use far more resources than the lower levels and often give only modest (if any) improvements.

10.11 SSA Form

Many modern compilers (such as the LLVM compiler framework) use an intermediate representation called *SSA form*, which is short for “static single assignment form”. In SSA form, a variable has exactly one place of definition (i.e., it will be assigned to exactly once). The advantage of the SSA form is that many optimisations are simpler to do for programs in SSA form than for programs not in SSA form.

10.11.1 Transforming to SSA Form

Starting from the intermediate representation in Fig. 6.1, we make the following changes:

1. Labels may get parameters, so all occurrences of **labelid** are replaced by a non-terminal *Labelid* with the productions

$$\begin{aligned} \textit{Labelid} &\rightarrow \mathbf{labelid} \\ \textit{Labelid} &\rightarrow \mathbf{labelid}(\textit{Args}) \end{aligned}$$

so labels can have optional parameters. In label definitions (instructions beginning with the LABEL keyword), the arguments are distinct identifiers but in jumps to labels, repeated variables and constant parameters are allowed.

2. We add the restriction that an identifier can only be assigned to once. An assignment is either an explicit assignment using the := operator or an occurrence of a variable in a function header or a label definition.

Fig. 10.9 First step of SSA transformation of the code in Fig. 6.16

```

_fac (v1)
LABEL L0
  t1 := 1
  v0 := t1
GOTO L1
LABEL L1
  t3 := v0
  t4 := v1
  v0 := t3*t4
  t5 := v1
  t6 := 1
  v1 := t5 - t6
  t7 := v1
  t8 := 0
  IF t7 = t8 THEN L2 ELSE L1
LABEL L2
RETURN v0

```

Fig. 10.10 Second step of SSA transformation of the code in Fig. 6.16

```

_fac (v11)
LABEL l0
  t1 := 1
  v01 := t1
GOTO L1
LABEL L1
  t3 := v0
  t4 := v1
  v02 := t3*t4
  t5 := v1
  t6 := 1
  v12 := t5 - t6
  t7 := v1
  t8 := 0
  IF t7 = t8 THEN L2 ELSE L1
LABEL L2
RETURN v0

```

The first step of the translation is to add a label L0 immediately after the function header, and to make explicit jumps to labels that the code just “falls through” to, as we did in Sect. 10.6. Starting from the code in Fig. 6.16, we get the code in Fig. 10.9.

The second step of the translation is to add subscripts to definitions of variables that are defined multiple times. After doing this, we get the code shown in Fig. 10.10.

The third step is at label definitions to identify if two differently subscripted versions of a variable can flow into the labels. At label L1, this is true for both v0 and v1, but at label L2 only one version of v0 and v1 (subscripted by 2) flow into

Fig. 10.11 Third step of SSA transformation of the code in Fig. 6.16

```

_fac(v11)
LABEL L0
  t1 := 1
  v01 := t1
  GOTO L1(v0, v1)
LABEL L1(v03, v13)
  t3 := v0
  t4 := v1
  v02 := t3*t4
  t5 := v1
  t6 := 1
  v12 := t5 - t6
  t7 := v1
  t8 := 0
  IF t7 = t8 THEN L2 ELSE L1(v0, v1)
LABEL L2
RETURN v0

```

Fig. 10.12 Complete SSA transformation of the code in Fig. 6.16

```

_fac(v11)
LABEL L0
  t1 := 1
  v01 := t1
  GOTO L1(v01, v11)
LABEL L1(v03, v13)
  t3 := v03
  t4 := v13
  v02 := t3*t4
  t5 := v13
  t6 := 1
  v12 := t5 - t6
  t7 := v12
  t8 := 0
  IF t7 = t8 THEN L2 ELSE L1(v02, v12)
LABEL L2
RETURN v02

```

the label. If different versions of a variable can flow into a label, this variable (with a new subscript, as the label is a definition point) is added as a parameter. In the example, $v0_3$ and $v1_3$ are added as parameters to L1. At jumps to this label, we add these variables (without subscript) as arguments to the label. The result is shown in Fig. 10.11.

Finally, every use of a variable with multiple subscripts is given the subscript of the version that flows into the statement that uses the variable. This will be unique. The resulting final SSA form is shown in Fig. 10.12.

The only nontrivial parts of the SSA transformation are figuring out which labels need parameters, and which versions of subscripted variables flow to which uses of the variables. To do this, we define a few concepts:

Definition 10.2 A *basic block* is a sequence of instructions that begin by a label definition and ends with a (conditional or unconditional) jump or a RETURN

statement, and there are no jumps between these. We identify a basic block by the label of its initial label definition.

Note that, after inserting jumps where the code falls through to a label (as we did in step 1 of the SSA transformation), a basic block contains no label definitions apart from its initial label definition.

In Fig. 10.9, there are three basic blocks, starting with the labels L0, L1, and L2,

Definition 10.3 A basic block L_i is a *predecessor* of a basic block L_j if L_i ends with a (conditional or unconditional) jump to L_j .

In Fig. 10.9, L0 has no predecessors, L0 and L1 are predecessors of L1, and L1 is a predecessor of L2.

Definition 10.4 A basic block L_i *dominates* a basic block L_j if all paths from the function header to L_j must pass through L_i . A basic block dominates itself. A basic block L_i *strictly dominates* a basic block L_j if L_i dominates L_j and $L_i \neq L_j$.

In Fig. 10.9, L0 dominates itself, L1, and L2. L1 dominates itself and L2, and L2 dominates only itself.

We note that if two basic blocks L_i and L_j dominate a basic block L_k , then either L_i dominates L_j or L_j dominates L_i . This leads to the following definition:

Definition 10.5 the *closest strict dominator* of a basic block L_j is a strict dominator L_i of L_j that is not itself dominated by another dominator of L_j . The basic block that immediately follows the function header has no strict dominator, and hence no closest strict dominator. Any other basic block will have a closest strict dominator.

Definition 10.6 A basic block L_i is in the *dominance frontier* of a basic block L_j if L_i dominates a predecessor of L_j but doesn't strictly dominate L_j .

In Fig. 10.9, L0 and L2 have empty dominance frontiers, and the dominance frontier of L1 contains only L1.

This leads to the following theorem:

Theorem 10.7 *Whenever a basic block L_i contains a definition of a subscripted variable v , then any basic block L_j in the dominance frontier of L_i needs v (with a new subscript) as a parameter to its label definition.*

In Fig. 10.9, the dominance frontier of L1 contains L1, and L1 has definitions of the subscripted variables v_0 and v_1 , so the label definition L1 needs these as parameters (with new subscripts), as shown in Fig. 10.11.

Note that parameters to label definitions are also definitions, so adding these may trigger yet more additions of label parameters.

The final step is to add subscripts to *uses* of subscripted variables. Before this step, only *definitions* of subscripted variables have subscripts. We use the following rules to find the subscript of a use of a variable v in a basic block L_i :

1. If v is defined in L_i before its use, the index of last definition of v that precedes the use of v is added to the use of v .
2. Otherwise, if L_i is the basic block immediately following the function header, add the index of v in the function header.
3. Otherwise, find the closest strict dominator L_j of L_i and apply the rules to L_j .

In Fig. 10.12, we have used rule 1 to see that v_0 should have subscript 1 in the GOTO statement, and we have used rule 2 to see that v_1 should have subscript 1 in the GOTO statement. In the assignments to t_3 , t_4 , t_5 , and t_7 and in the conditional jump to L1, we have used rule 1 to find the subscripts of the uses of v_0 and v_1 . Finally, we have used rule 3 to find the subscript of v_0 in the RETURN statement. The closest strict dominator of L2 is L1, and the subscript of the last definition of v_0 in L1 is 2.

10.11.2 Using the SSA Form

The single-assignment property of the SSA form makes some optimisations simple. We will, here, just look at *constant and copy propagation*, and leave other uses of the SSA form to further reading.

We use the observation that, if the right-hand e side of an assignment $x := e$ is either a variable y or a constant c , the SSA property implies that all occurrences of x in the program can be replaced by e . We can iterate this to eliminate all assignments of constants and variables to variables. Applying this to the code in Fig. 10.12 yields the code in Fig. 10.13. Note that we, in the GOTO statement use the constant 1 as a parameter to L1.

Suggested exercises: 10.7.

Fig. 10.13 Applying constant and copy propagation to the code in Fig. 10.12

```

_fac(v11)
LABEL L0
GOTO L1(1, v11)
LABEL L1(v03, v13)
  v02 := v03 * v13
  v12 := v13 - 1
  IF v12 = 0 THEN L2 ELSE L1(v02, v12)
LABEL L2
RETURN v02

```

10.12 Further Reading

We have covered only a small portion of the analyses and optimisations that are found in optimising compilers. More examples (including value numbering) can be found in advanced compiler textbooks, such as [1–3, 9]. That data-flow analyses can never be 100% precise for all programs is a consequence of Rice’s Theorem, which is covered in, e.g., [7].

More details about the SSA form (including an efficient algorithm for computing the dominance frontiers of basic blocks) can be found in [3, 5]. The LLVM framework is described in [6]. Note that some definitions of the SSA form don’t use parameters to labels but instead use something called ϕ -nodes to achieve the same effect.

A detailed treatment of program analysis can be found in [10]. The book [8] has good articles on program analysis and transformation.

Additionally, the conferences “Compiler Construction” (CC), “Programming Language Design and Implementation” (PLDI), and other programming-language conferences often present new optimisation techniques, so proceedings from these are good sources for advanced optimisation methods.

10.13 Exercises

Exercise 10.1 In the program in Fig. 10.2, replace instructions 13 and 14 by

```
13:  $h := n * 3$ 
14: IF  $i < h$  THEN loop ELSE end
```

- a) Repeat common subexpression elimination on this modified program.
- b) Repeat, again, common subexpression elimination on the modified program, but, prior to the fixed-point iteration, initialise all sets to the empty set instead of the set of all assignments.

What differences does this make to the final result of fixed-point iteration, and what consequences do these differences have for the optimisation?

Exercise 10.2 In Sect. 10.5 we did not generate any assignments for a copy instruction of the form $x := y$. But we might exploit that x now holds the values of any expression that y holds the value of, so if an assignment is later made to y , this value can still be found in x . In other words, if the set of available assignments before the copy statement contains an assignment $y := e$, it can after the copy statement also contain $x := e$. A set of available assignments may now actually contain assignments that are not part of the original program.

- a) Make a special case of Eq. 10.3 for copy statements (of the form $i: x := y$) that does what is described above. Note that $gen[i]$ is empty.

- b) Describe the set of assignments that with this modification should be used to initialise $in[]$ and $out[]$ sets.

Exercise 10.3 Regarding the index-check analysis,

- Find gen , gen_l and $kill$ sets for the instructions in Fig. 10.7.
- Do the fixed-point iteration for index-check elimination on the result to find the in sets.
- Eliminate the redundant tests.

Exercise 10.4 Write a program that has jumps to jumps, and perform jump-to-jump optimisation of it as described in Sect. 10.7.

Exercise 10.5 In Sect. 10.8, compressed bit vectors are suggested for representing sparse sets. A possible compression scheme is to split the bit vector v of size $n > 1$ into two bit vectors v_1 and v_2 , of size $n/2$ and $n - n/2$, respectively. This is then compressed into a bit vector of the following form:

- If both v_1 and v_2 contain only zeroes, the compressed bit vector has the form 00.
- If v_1 contains only zeroes and v_2 contains at least one one, the compressed bit vector has the form 01 followed by the compressed bit vector for v_2 .
- If v_2 contains only zeroes and v_1 contains at least one one, the compressed bit vector has the form 10 followed by the compressed bit vector for v_1 .
- If both v_1 and v_2 contain ones, the compressed bit vector has the form 11 followed by the compressed bit vectors for v_1 and v_2 , in that order.

A sub-vector of size 1 needs no representation, as its value is implied by the prefix. Since the size of the uncompressed bit vector is known, it is also known when a sub-vector has size 1. A compressed bit vector can be more than one machine word, so a linked list of compressed bit vectors may be required. But a large, sparse bit vector will usually compress into shorter list of machine words than the original.

- Describe how you can inspect a compressed bit vector to find the value of bit j in the uncompressed vector (where indices are between 0 and $n - 1$).
- Describe how union and intersection of compressed bit vectors can be done without fully decompressing the bit vectors.

You do not need to consider the case where the compressed bit vector is more than one machine word.

Exercise 10.6 In Fig. 10.8, the equations for **binop** means that an instruction like $x := y - z$, where y and z are pointers, will add pairs that make x point to anything that y and z may point to. Discuss how this rule can be modified to give more precise results.

Exercise 10.7 Change the function definition shown in Sect. 6.9.2 to use a while loop instead of a repeat loop, generate code as shown in Sect. 6.9.2, and convert the result to SSA form. Show the result after each of the four steps shown in Sect. 10.11.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers; Principles*. Addison-Wesley, Techniques and Tools (2007)
2. Allen, J.R., Kennedy, K.: *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann (2001)
3. Appel, A.W.: *Modern Compiler Implementation in ML*. Cambridge University Press (1998)
4. Briggs, P.: *Register allocation via graph coloring*, tech. rept. cpc-tr94517-s. Ph.D. thesis, Rice University, Center for Research on Parallel Computation (1992)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* **13**(4), 451–490 (1991). <https://doi.org/10.1145/115372.115320>
6. Lattner, C., Adve, V.: *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California (2004)
7. Mogensen, T.Æ.: *Programming Language Design and Implementation*. Springer, Cham (2022). <https://link.springer.com/book/10.1007/978-3-031-11806-7>
8. Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.): *The essence of computation: complexity, analysis, transformation*. Springer-Verlag, New York Inc, New York, NY, USA (2002)
9. Muchnick, S.S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann (1997)
10. Nielson, F., Nielson, H.R., Hankin, C.: *Principles of Program Analysis*. Springer-Verlag, New York Inc, Secaucus, NJ, USA (1999)

Chapter 11

Optimisations for Loops



If you optimize everything, you will always be unhappy.
Donald Knuth (1938–)

Since programs tend to spend most of their time in loops, it is worthwhile to study optimisations specific for loops. We will in this chapter look at three specific optimisations for loops: *Code hoisting*, which moves loop-invariant code out of a loop, *memory prefetching*, which improves cache utilisation, and *incrementalisation*, which replaces multiplication in a loop by addition.

11.1 Loops

Let us first define what we mean by a loop. At first glance a loop is any part of the program that is repeatedly executed, but this is a bit too general a definition, as this includes recursive function calls and unstructured (“spaghetti”) code that can not be expressed as high-level language loops. Intuitively, we want loops to correspond to high-level language structures, even if we work at the intermediate-language level. In particular, we want a loop to have a single entry point, as this will allow us to move code from inside the loop to just before the entry of the loop.

In terms of the intermediate language shown in Sect. 6.2, the entry point of a loop is a LABEL instruction. But not all labels are entry points of loops, so we additionally require execution to be able to return to the entry point. Hence, we define a loop by the following criteria:

Definition 11.1 A *loop* is defined by a LABEL instruction called the *entry point* of the loop and a set of instructions that form the *body* of the loop. All the instructions in the body of the loop must obey two criteria:

1. There are paths from the entry point of the loop to all instructions in its body.
2. From any instruction in the loop body, there must be a path back to the entry point of the loop. This path must use only instructions from the loop body.
3. Any path from an instruction outside the loop to an instruction inside the loop body must pass through the entry point of the loop.

A path is a sequence of instructions related by the successor relation as defined in Sect. 8.1.

In a high-level program, any code that is translated into intermediate code obeying Definition 11.1 is considered a loop. Note that some high-level loops (such as `for` loops) include things like initialisation of variables, which by the definition above occur *before* the loop rather than as *part of* it. Also, an unconditional jump out of the loop and any code leading up to this is not considered part of the loop, as there is no path back to the entry point that does not leave the loop.

Loops may overlap in several ways: A loop may be nested inside another, or two loops may share a header and some body instructions, but not all of these. In the latter case, the union of the two loops also forms a (larger) loop.

Some of our examples will use high-level syntax for loops, while other examples use the intermediate language from Sect. 6.2.

11.2 Code Hoisting

One optimisation for loops is finding computations that are repeated in every iteration of the loop without changing the values involved, i.e., loop-invariant computations, and then lift these computations outside the loop, so they are performed only once before the loop is entered. This is called *code hoisting*.

We saw an example of this in Sect. 10.5.3, where calculation of $n * 3$ was done once before the loop, and subsequent re-computations were replaced by a reference to the variable a that holds the value of $n * 3$ computed before the loop. However, it was only because there already was an explicit computation of $n * 3$ before the loop, that we could avoid re-computation inside the loop: Otherwise, the occurrence of $n * 3$ inside the loop would not have any available assignment that can replace the calculation.

So our aim is to move or copy loop-invariant assignments to before the loop, so their result can be reused inside the loop. Moving a computation to before the loop may, however, cause it to be computed even when the loop is not entered. In addition to causing unnecessary computation (which goes against the wish for optimisation), such computations can potentially cause errors when the precondition (the loop-entry condition) is not satisfied. For example, if the invariant computation is a memory access, the address may be valid only if the loop is entered.

A common solution to this problem is to unroll the loop once: A loop of the form (using C-like syntax):

```
while (cond) {
    body
}
```

is transformed to

```
if (cond) {
    body
    while (cond) {
        body
    }
}
```

Similarly, a test-at-bottom loop of the form

```
do
    body
while (cond)
```

can be unrolled to

```
body
while (cond) {
    body
}
```

Now, we can safely calculate the invariant parts in the extracted copy of the body (which represents the first loop iteration), and reuse the results in the remaining loop iterations. If the compiler unrolls loops this way and afterwards does common subexpression elimination, this is all that is required to do code hoisting. Unrolling of loops is most easily done at source-code level (i.e., on the abstract syntax tree), while common-subexpression elimination is done on the intermediate code, so unrolling naturally happen before common-subexpression elimination. Unrolling loops as above will, of course, increase the size of the compiled program, so it should be done with care if the loop body is large.

Loop unrolling at the level of intermediate code is not too difficult either:

1. Make two copies of all instructions in the loop, including the entry-point label.
2. Rename all labels in the second copy systematically, so jumps from the second copy to labels inside the loop (including the entry point) are to renamed labels. Jumps to outside the loop are kept unchanged.
3. In the first copy of the loop, make all jumps to the entry point of the loop jump to the (renamed) entry-point label of the second copy.
4. Place the second copy directly after an instruction in the first copy that jumps to the entry point of the second copy. This placement may avoid a jump when the intermediate code is translated to machine language.

Suggested exercises: [11.1](#).

11.3 Memory Prefetching

If a loop goes through a large array, it is likely that parts of the array will not be in the cache of the processor. Since access to non-cached memory is *much* slower than access to cached memory, we would like to avoid this.

Many modern processors have *memory prefetch instructions* that tell the processor to load the contents of an address into cache, but unlike a normal load, a memory prefetch returns immediately without waiting for the load to complete, and it does not cause errors if the address is invalid (it just has no effect in this case). So a way to ensure that an array element is in the cache by the time it is used is to issue a prefetch of the array element well in advance of its use, but not so well in advance that it is likely that it will be evicted from the cache between the prefetch and the use. Given modern cache sizes and timings, 50 to 100 000 cycles ahead of the use is a reasonable time for prefetching—less than 50 increases the risk that the prefetch is not completed before the array element is accessed, and more than 100 000 increases the chance that the value will be evicted from the cache before it is used.

So the idea is that, if a loops goes sequentially through the elements of an array, we will prefetch elements that will be processed later. We want most array elements of be prefetched by the time we use them, so we want to start prefetching relatively close to the start of the array, but not so close that the elements do not have time to be read into the cache. A prefetch instruction usually loads an entire cache line, which is typically four or eight words, so we do not have to explicitly prefetch every array element—every fourth or eight element is enough. Since prefetching is harmless and relatively cheap, it is better to prefetch a bit more often than required, so if we don't know anything about the size of cache lines, a safe bet is every fourth element.

So, assume we have a loop that adds up the elements of an array:

```
sum = 0;
for (i=0; i<100000; i++)
    sum += a[i];
}
```

we can rewrite this to

```
sum = 0;
for (i=0; i<100000; i++) {
    if (i&3 == 0) prefetch a[i+32];
    sum += a[i];
}
```

where `prefetch a[i+32]` prefetches the element of `a` that is 32 places after the current element. The number 32 is rather arbitrary, but makes the number of cycles between prefetch and use lie in the interval mentioned above. We have kept to the low end of the interval so the early iterations of the loop benefit from prefetching. Note that we used the test `i&3==0`, which is equivalent to `i%4==0`, but somewhat faster.

We do not have to worry about prefetching past the end of the array – prefetching will never cause runtime errors, so at worst we prefetch something that we will not need.

While this transformation adds a test (that takes time), the potential savings by having nearly all array elements in cache before use are much larger. The overhead of testing can be reduced by unrolling the loop body:

```
sum = 0;
for (i=0; i<100000; i++) {
    prefetch a[i+32];
    sum += a[i];
    i++;
    sum += a[i];
    i++;
    sum += a[i];
    i++;
    sum += a[i];
}
```

We have, in the above, exploited that the number of iterations is a multiple of 4, so the exit test is not needed at every increment of *i*. If the upper limit is unknown, the exit test must be replicated after each increase of *i*:

```
sum = 0;
for (i=0; i<n; i++) {
    prefetch a[i+32];
    sum += a[i];
    i++;
    if (i >= n) break;
    sum += a[i];
    i++;
    if (i >= n) break;
    sum += a[i];
    i++;
    if (i >= n) break;
    sum += a[i];
}
```

Note that this unrolling is different from the unrolling shown in Sect. 11.2, as the last copy of the body jumps back to the original entry point. The loop unrolling used in Sect. 11.2 is akin to non-recursive inlining, while the loop unrolling above is more akin to recursive inlining (see Sect. 9.12.1). To avoid code explosion, unrolling should only be done if the loop body is small.

In a nested loop that accesses a multi-dimensional array, you can prefetch the next row while processing the current. For example, the loop

```

sum = 0;
for (i=0; i<1000; i++)
    for (j=0; j<1000; j++)
        sum += a[i][j];
    }
}

```

can be transformed to

```

sum = 0;
for (i=0; i<1000; i++)
    for (j=0; j<1000; j++)
        if (j&3 == 0) prefetch a[i+1][j];
        sum += a[i][j];
    }
}

```

Like above, we can unroll the body of the loop to reduce the overhead of testing:

```

sum = 0;
for (i=0; i<1000; i++)
    for (j=0; j<1000; j++)
        prefetch a[i+1][j];
        sum += a[i][j];
        j++;
        sum += a[i][j];
        j++;
        sum += a[i][j];
        j++;
        sum += a[i][j];
    }
}

```

11.4 Incrementalisation

Incrementalisation is a technique where $f(x + \delta)$ is calculated from $f(x)$ and δ using fewer operations than calculating $f(x + \delta)$ from scratch. For example, $(x + 1)^2 = x^2 + 2x + 1$, so calculating $(x + 1)^2$ can be done by adding $2x + 1$ to the already computed result of x^2 . Hence, an addition and a multiplication is replaced by either three additions ($v + x + x + 1$) or a shift and two additions ($v + (x \ll 1) + 1$), where \ll is the binary shift-left operator. Incrementalisation is also called *reduction in strength* because it can replace an expensive operation (like multiplication) by less expensive operations (like addition).

If arithmetic overflow causes errors or exceptions, we must take care that the transformed code causes overflow exactly when the original code does. Also, if

<pre> sum := 0 i := 0 LABEL loop1 IF i >= 100 THEN exit1 ELSE body1 LABEL body1 j := 0 LABEL loop2 IF j >= 100 THEN exit2 ELSE body2 LABEL body2 t1 := i*800 t2 := t1+a t3 := j*8 t4 := t2+t3 t5 := M[t4] sum := sum+t5 j := j+1 GOTO loop2 LABEL exit2 i := i+1 GOTO loop1 LABEL exit1 </pre>	<pre> sum := 0 i := 0 t1 := 0 t2 := a LABEL loop1 IF i >= 100 THEN exit1 ELSE body1 LABEL body1 j := 0 t3 := 0; LABEL loop2 IF j >= 100 THEN exit2 ELSE body2 LABEL body2 t4 := t2+t3 t5 := M[t4] sum := sum+t5 j := j+1 t3 := t3+8 GOTO loop2 LABEL exit2 i := i+1 t1 := t1+800 t2 := t2+800 GOTO loop1 LABEL exit1 </pre>
(a)	(b)

Fig. 11.1 Incrementalisation of nested loop

operating on floating-point numbers, rounding errors may accumulate when using incrementalisation, so precision may not be preserved. Hence, most compilers only do incrementalisation on integer calculations that do not trap on overflows, but instead calculates results modulo 2^n .

A common and safe use of incrementalisation is array index calculation. Calculating the address of an element of a multi-dimensional array can involve multiplication. For example, in a 100×100 array a of 64-bit integers, the address of the element $a[i][j]$ is $a+i*800+j*8$, where a is the base address of the array. But if we already know the address of $a[i][j]$, we can calculate the address of $a[i+1][j]$ simply by adding 800 to the address of $a[i][j]$, hence replacing one multiplication, one binary shift (to multiply j by 8), and two additions by a single addition. Using the intermediate language shown in Sect. 6.2 and no optimisation, the loop

```

sum = 0;
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        sum += a[i][j];
    }
}

```

generates the intermediate code shown in Fig. 11.1a.

We note that $t1$ and $t2$ change only when i changes, and $t3$ changes only when j changes, so we initialise $t1$ and $t2$ when we initialise i , update $t1$ and $t2$ when we update i , and similarly initialise and update $t3$ when we initialise and update j . This gives the code in Fig. 11.1b.

<pre> sum := 0 i := 0 t2 := a LABEL loop1 IF i >= 100 THEN exit1 ELSE body1 LABEL body1 j := 0 t3 := 0; LABEL loop2 IF j >= 100 THEN exit2 ELSE body2 LABEL body2 t4 := t2+t3 t5 := M[t4] sum := sum+t5 j := j+1 t3 := t3+8 GOTO loop2 LABEL exit2 i := i+1 t2 := t2+800 GOTO loop1 LABEL exit1 </pre>	<pre> sum := 0 a8 := 80000+a t2 := a LABEL loop1 IF t2 >= a8 THEN exit1 ELSE body1 LABEL body1 t3 := 0; LABEL loop2 IF t3 >= 800 THEN exit2 ELSE body2 LABEL body2 t4 := t2+t3 t5 := M[t4] sum := sum+t5 t3 := t3+8 GOTO loop2 LABEL exit2 t2 := t2+800 GOTO loop1 LABEL exit1 </pre>
(a)	(b)

Fig. 11.2 Eliminating weakly dead variables

We note that, at both assignments to $t1$, it is weakly dead according to definition 10.1. Hence, we can eliminate these assignments, as shown in Fig. 11.2a.

In the transformed code, the inner loop uses four additions, where the inner loop of the original code uses two multiplications (one of which can be implemented by a binary shift) and three additions.

A further optimisation can be made by observing that, assuming it is dead at exit from the loop, j is only used to update itself, and for the exit test of the loop. We can exploit that, at the time of the test, $t3=j*8$ and replace the test $j \geq 100$ by $t3 \geq 800$, at which point j is weakly dead and can be eliminated. Similarly, we observe that (assuming it is dead after the loop) also i is only used to update itself, and in the exit condition of the outer loop. The relation between i and $t2$ is $t2=i*800+a$, so we can replace the test $i \geq 100$ by $t2 \geq 80000+a$. Since a doesn't change inside the loop, we can precompute $a8 = 80000+a$ and get the code shown in Fig. 11.2b.

11.4.1 Rules for Incrementalisation

The example above is rather informal, so we need to clarify when and how we can do the transformation. We start by some definitions:

Definition 11.2 The *code in consideration* is any code that we wish to transform using incrementalisation. It is usually a loop or several nested loops, but it need not be. The code under consideration needs not be a complete function body.

Definition 11.3 A *loop counter* is a variable i where all assignments to i are of the form $i := k$ or $i := i + k$, where k is not changed anywhere in the code under consideration.

In the code in Fig. 11.1a, i and j are loop counters.

Definition 11.4 A *variable derived from a loop counter* i is a variable x to which there in the code under consideration is exactly one assignment, which is of the form $x := y + k$, $x := y * k$, or $x := y + z$, where k is not changed anywhere in the code under consideration, and y and z are either i or variables derived from i (other than x itself). Furthermore, x must be dead at entry to and exit from the code under consideration, and no assignment to i can happen between the assignment to x and any use of x .

In the code in Fig. 11.1a, t_1 and t_2 are derived from i , and t_3 is derived from j . t_4 is not a derived variable, since t_2 and t_3 are derived from different loop counters. t_5 is not a derived variable, as the assignment does not have the required form.

The idea is that a variable x derived from a loop counter i will have a value of the form $i * p + q$, where p and q are not changed during the code in consideration. In Fig. 11.1a, $t_1 = i * 800 + 0$, $t_2 = i * 800 + a$ and $t_3 = j * 8 + 0$. We can easily find the values of derived variables from their assignments:

- If the assignment to x is of the form $x := i + k$, then $x = i * 1 + k$.
- If the assignment to x is of the form $x := i * k$, then $x = i * k + 0$.
- If the assignment to x is of the form $x := y + k$, and $y = i * p + q$, then $x = i * p + (q + k)$.
- If the assignment to x is of the form $x := y * k$, and $y = i * p + q$, then $x = i * (pk) + (qk)$.
- If the assignment to x is of the form $x := y + z$, $y = i * p + q$, and $z = i * p' + q'$, then $x = i * (p + p') + (q + q')$.

If a derived variable x has the value $i * p + q$, we can remove its original assignment, and instead add assignments to x immediately after assignments to i :

- If the assignment to i is of the form $i := k$, add the assignment $x := (pk) + q$.
- If the assignment to i is of the form $i := i + k$, add the assignment $x := x + (pk)$.
- Furthermore, if i is live at entry to the code in consideration, add the assignment $x := i * p + q$ just before the code in consideration.

Note that, since pk is not changed anywhere in the code under consideration, it can be precomputed and, if it is not a constant, assigned to a variable before the code under consideration.

If k doesn't change during the code under consideration, we can, if $p > 0$, replace a test of the form $i < k$ with the test $x < (pk) + q$ and, if $p < 0$, by $x > (pk) + q$. If $p = 0$ or the sign of p is not known, we can not make the replacement. Similar replacements can be made for tests using \leq , $>$, \geq and $=$. Note that $(pk) + q$ doesn't

change in the code in consideration, so it can be precomputed and put into a variable, like we did with the expression $80000 * i + a$ in Fig. 11.2b. There may be several candidates for a derived variable to use instead of i in the test, but it is best to use a variable that is used for something other than defining another derived variable. Since $t1$ in the example above is only used to define $t2$, and $t2$ is used in the assignment $t4 := t2 + t3$, where $t4$ is not a derived variable, $t2$ is the best choice to replace i in the test.

After these transformations, we can remove assignments to variables that are weakly dead. Note that, if a loop counter is live after exit from the code in consideration, it is not weakly dead, so we can not eliminate it.

The above definition of derived variables can be used to do finite differencing of array address calculations but not for finite differencing of assignments like $x := i * i$, as this doesn't have the required form. $x := i * i$ can be incrementalised using more advanced methods by exploiting that $(i + k)^2 = i^2 + 2ki + k^2$. Generally, any polynomial over a loop counter can be incrementalised, but this can get rather complex. Furthermore, the incrementalised version may generate overflows where the original does not, so the transformation should only be used for arithmetic modulo 2^n .

Incrementalisation is usually not used for floating-point calculations, as rounding errors can accumulate to make the result less precise.

Suggested exercises: 11.2.

11.5 Further Reading

The suggestions for further reading in Chap. 10 are also good sources for optimisations for loops. A more general method for incrementalisation can be seen in [1].

11.6 Exercises

Exercise 11.1 In C and similar languages, it is possible to exit a loop from anywhere inside the loop using a `break` statement. In Sect. 11.2, transformations are shown that unroll high-level loops once. These are for loops that exit with tests either at the top or the bottom of the loop and do not consider the possibility of `break` statements.

- What modifications are required for the high-level loop-unfolding transformations if the body of a loop can contain `break` statements?
- Are exit jumps from anywhere inside a loop allowed by Definition 11.1?
- If such exit jumps can occur, are any modifications needed to the rule shown at the end of Sect. 11.2 for unfolding intermediate-code loops?

Exercise 11.2 Consider the following intermediate code:

```
sum := 0
i := 0
LABEL loop
IF i >= 100 THEN exit ELSE body
LABEL body
t1 := i*800
t2 := a+t1
t3 := i+1
t4 := t3*8
t5 := t2+t4
t6 := M[t5]
sum := sum+t6
i := i + 2
GOTO loop
LABEL exit
```

- a) Identify the loop counter(s) in the code.
- b) Identify variables derived from these loop counter(s) and their values in the form $i * p + q$, where i is a loop counter and p and q are invariant.
- c) Perform incrementalisation of the derived variables.
- d) Replace the test `i >= 100` with a test on a derived variable.
- e) Eliminate weakly dead variables.

Reference

1. Liu, Y.A.: Efficiency by incrementalization: An introduction. *High. Ord. Sym. Comput.* **13**, 289–313 (2000). <https://doi.org/10.1023/A:1026547031739>, www.dl.acm.org/citation.cfm?id=369129.369135

Chapter 12

More Language Features



*Normal people... believe that if it ain't broke, don't fix it.
Engineers believe that if it ain't broke, it doesn't have enough
features yet.*

Scott Adams

In this chapter, we will look at some programming language features not covered in depth by the previous chapters. We avoid complex features such as classes with inheritance and virtual method calls and focus on features that are commonly found in programming languages and which are not terribly difficult to implement.

12.1 Parametric Polymorphism

Consider the following two functions written in C that are shown in Fig. 12.1.

They both apply a function that is given as a parameter (see Sect. 9.10) to all elements of an array, they only differ in the types of the function and array: One uses integers and one uses floats. Note that the bodies of the two functions are identical.

We might also want similar functions for other types such as `long` and `double`, and we will have to write separate functions for each type. This is why C++ introduced *templates*, which is a form of *parametric polymorphism*. The idea is to give function definitions not only value parameters but also type parameters, so you can write one function that works on multiple types. In C++, a polymorphic (generic) version of the two functions above is written as shown in Fig. 12.2.

The template declaration states that the following function declaration is parameterised over a typename `T`. In order to use this function declaration, we must explicitly pass a concrete type as parameter, so you can, e.g., write

```

void applyI(int (*f)(int), int a[], int size) {
    for (int i = 0; i < size; i++) a[i] = f(a[i]);
}

void applyF(float (*f)(float), float a[], int size) {
    for (int i = 0; i < size; i++) a[i] = f(a[i]);
}

```

Fig. 12.1 Two apply functions written in C

```

template <typename T>
void apply(T (*f)(T), T a[], int size) {
    for (int i = 0; i < size; i++) a[i] = f(a[i]);
}

```

Fig. 12.2 A C++ template definition

```

void apply_675(int (*f)(int), int a[], int size) {
    for (int i = 0; i < size; i++) a[i] = f(a[i]);
}

void apply_676(double (*f)(double), double a[], int size) {
    for (int i = 0; i < size; i++) a[i] = f(a[i]);
}

```

Fig. 12.3 Two declarations generated from a template

```

apply<int>(abs, xs, 100);
apply<double>(fabs, ys, 100);

```

to call the apply function with integer and floating point types, respectively.

12.1.1 Implementing Templates

In C++, template function declarations are parsed but not type checked when they are declared, and no code is generated. When the compiler sees a call to a template function, it will generate source code for a specialised version of the function. So the two calls to apply shown above will cause the declarations shown in Fig. 12.3 to be generated.

The numbers are added to the function names to make them distinct. The new declarations are placed in the top level scope before the function from which the

apply function is called. The compiler will then parse, type check and generate code for the two declarations. The two calls will be replaced by

```
apply_675(abs, xs, 100);
apply_676(fabs, ys, 100);
```

which are then parsed, type checked and generated code for.

12.1.2 Variants

The reason that template declarations in C++ are not type checked when they are declared is that C++ allows overloaded operators. For example `+` is defined to work on both two integers, two floating point values, and a pointer and an integer (but not two pointers). A template declaration is allowed to use overloaded operators, but these can not be properly type checked before the type parameter is known. A variant of parametric polymorphism called *uniform parametric polymorphism* requires that a polymorphic declaration is well-typed on *all* possible type parameters. This allows the template declaration to be type checked when it is declared—essentially by type checking it with a type on which no operations are defined (a type invented for just that purpose). This will work well for the `apply` function defined above, but not for, say, a polymorphic sorting function that is defined only for types where `<` is defined. That can be handled with *bounded polymorphism*, where type parameters can be instantiated to any type that obeys certain constraints, e.g., that `<` is implemented for the type. In F#, bounded polymorphism is used for the `List.sort` function, which has the type

```
('a list -> 'a list) when 'a : comparison
```

The `when` clause limits the type variable `'a` to be instantiated only to types that implement `comparison`. A sort function in Haskell would, equivalently, have the type

```
Ord a => [a] -> [a]
```

which says that the sorting function maps a list of elements of type `a` to another list if the same type, but only if `a` implements the `Ord` (ordered) type class (where a type class is similar to an interface in Java and C#). Note the different notations for list types and type variables in F# and Haskell—in Haskell, type variables are written in lower case, while concrete types use upper-case initials, and list type are indicated by wrapping square brackets around the element type. In F#, type variables begin with a single quote, and list types are indicated by suffixing the element type with the type constructor `list`. If type variables are bounded explicitly, it is possible to type check a template (or other form of explicitly polymorphic definition) when it is declared: As with uniform parametric polymorphism, we need only check it on a single type that obeys the stated bounds—and has no other properties than this.

Another variant is *implicit polymorphic types*. In languages with implicit polymorphic types, the programmer does not need to declare the types of function arguments

and results, and the compiler will analyse the function to find out which type they have. This will often give the function a polymorphic type. This is called *type inference* and is mostly found in functional programming languages in the ML family, such as Standard ML, Haskell, OCaml, and F#. For example, the following declaration in Haskell

```
map f [] = []
map f (x:xs) = f x : map f xs
```

which applies a function to all elements of a list will be inferred to have the type `map :: (a -> b) -> [a] -> [b]`, i.e., stating that `map` takes a function from any type `a` to any type `b` and a list with elements of type `a`, and then return a list with elements of type `b`. Note that the type parameters are implicit: All type variables used in the type are type parameters. Similarly, the type parameters need not be explicitly instantiated when the `map` function is applied. We can, for example, simply write `map not listOfBools` where `not` is a function from Booleans to Booleans and `listOfBools` is a list of Booleans. This instantiates both `a` and `b` to be the type `Bool`.

In F#, the `map` function can be defined by

```
let rec map f =
  function
  | [] -> []
  | (x :: xs) -> f x :: map f xs
```

and the type is derived to be `map : ('a -> 'b) -> 'a list -> 'b list`.

We leave the process of type inference to further reading.

Suggested exercises: [12.1](#).

12.2 Garbage Collection

In Chap. 6, we briefly discussed allocation of arrays and covered static (global) allocation and stack allocation. Dynamic memory allocation (also called heap allocation), including garbage collection, was briefly mentioned, but not covered in detail. Since most modern languages use automatic memory management (mostly using garbage collection) instead of manual memory management (using `malloc()` and `free()` or equivalent functions), we will, here, mainly look at garbage collection for managing heap memory.

Dynamic memory allocation is used where the lifetime of memory-allocated objects such as arrays and records do not follow program structure, unlike stack allocation where the lifetime of an object is limited by the lifetime of the function in which it is allocated, so it is no longer available when the function returns. This

means that a function can not as results return objects that are allocated in the function. To remedy this, dynamic memory allocation completely decouples where an object is allocated from where it is freed.

With manual memory management, freeing is done explicitly using a call to a function, such as `free()` in C or a destructor in C++. This is error-prone, as a programmer by mistake can free an object that will be used later on, or keep an object allocated long after it is no longer used. Freeing an object too early can cause the memory used for the object to be allocated for some other object, which overwrites the original object. Such errors can be very difficult to track down. Freeing an object too late is less harmful, but it can cause space leaks, where a long-running program uses more and more memory until it runs out.

With automatic memory management, the runtime system of the programming language will automatically free an object when it is certain that it will no longer be accessed. Since knowing the exact time when an object will no longer be used is not always possible, garbage collection uses a safe approximation: When an object no longer *can* be accessed, it is freed. An object *can*, in this approximation, be accessed if there is a path of pointers from a program variable to the object. The set of program variables include all variables allocated in registers, spilled on the stack, saved across procedure calls, or allocated globally. Collectively, this is called the *root set*. Garbage collection will trace all paths from heap pointers in the root set to objects in the heap, and further through pointers in these objects until all reachable objects are found. All non-reachable objects are then reclaimed for later use. Garbage collection is usually initiated when an allocation fails to find sufficient memory for the requested object allocation, but many systems also allows garbage collection to be manually triggered.

There are many different algorithms for garbage collection, but we will show only one: A variant of *conservative collection* using the *mark-sweep* algorithm.

12.2.1 Identifying Pointers

To trace paths from the root set, we need to know which variables in the root set contain pointers into the heap. There are multiple ways of doing so, using type information or using a mark bit in every variable to indicate that it is a heap pointer, but we will use a *conservative* identification: If a variable *looks like* a heap pointer, it will be treated as such. We use two indicators of whether or not a machine word is a heap pointer:

1. It is in the correct interval of numbers that *can* be heap pointers. The heap is assumed to be a single contiguous space in memory, and anything number between the first and last address of this space *might* be a heap pointer. But it is also possible that it is an integer or floating point value that just happen to be in this interval (when viewed as a machine word). If 64 bits are used for pointers, and the heap size is “only” a few gigabytes, the chance of a random number being in the correct interval is fairly small.

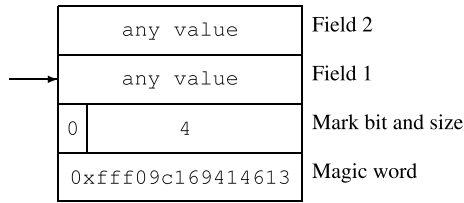


Fig. 12.4 Example heap object representation

- When viewed as a heap pointer, it points to something that is likely to be a heap-allocated object. To make this test likely to be correct, we let the first word of every heap-allocated object be a *magic number*: A pre-chosen number that is unlikely to be used for something else. A random 64-bit machine word that does not represent common integer or floating point values will do fine. One of the many rarely used representations of NaN (not a number) in the 64-bit IEEE 754 format for floating point values is probably a good idea. We use `0xffff09c169414613`, as this is a signalling NaN with non-zero payload in a pattern unlikely to occur in real life.

We can use these indicators not only to identify which members of the root set are heap pointers, but also to identify which fields of heap-allocated objects are pointers. Since the identification of heap pointers is not precise, the garbage collection algorithm is not allowed to change the values of machine words that it thinks are heap pointers: If it by chance is not a heap pointer, doing so would change the behaviour of the program. This implies that a heap object can not be moved by the garbage collector: Its address has to be preserved.

In addition to starting with the magic word, a heap-allocated object needs another header word (placed immediately after the magic word). This header word contains the size of the object and a bit (called the *mark bit*) that is used during garbage collection. After that, there is a number of 64-bit words containing the actual object that is allocated. The size is given as the total number of words of the object including the two header words, so it is at least 3. We use the most significant bit of the word holding the size as mark bit. Pointers to heap allocated objects point to the first word after the two header words, i.e., the first word that can be used by the program. An example of a heap object is shown in Fig. 12.4. Addresses grow upwards.

We will use two global variables `heapStart` and `heapEnd` to indicate the first and last address in the heap. Given the format suggested in Fig. 12.4, the function shown in Fig. 12.5 checks whether a value is (likely to be) a heap pointer.

12.2.2 The Freelist

At any given time, un-allocated memory in the heap will consist of a linked list of heap objects using the format described in Fig. 12.4, using the first field as a pointer

```
int isHeapPointer(uint64_t *value) {
    if (value < heapStart) return 0;
    if (value > heapEnd) return 0;
    if (*(value-2) != 0xffff09c169414613) return 0;
    return 1;
}
```

Fig. 12.5 Testing if a value is a pointer

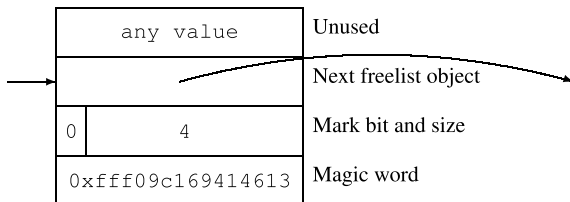


Fig. 12.6 Example freelist object representation

to the next object in the freelist, as shown in Fig. 12.6. By invariant, these will always be heap pointers, so we do not have to check this. Allocating an object consists of searching through the freelist for the first object that is large enough to hold the requested size s of object. The size is measured in 64-bit words and must include space for the two header words. Three different things can happen:

1. We reach the end of the freelist without having found a suitable object. In this case, we trigger garbage collection and try again. If this also fails, allocation fails.
2. We find an object that is between s and $s + 2$ words in size. In this case, we take the object out of the freelist and return its address to the user.
3. We find an object that is more than $s + 2$ words in size. In this case, we split the object in two, one holding s words and another holding the rest. The object holding s words is initialised by setting the two header words to `0xffff09c169414613` and the size s , respectively. The other object simply reduces its size field by s words and stays in the freelist. The address of the object of size s is returned to the user.

The reason we don't split objects of size $s + 1$ or $s + 2$ is that the remainder is too small to hold two header words and a pointer to the next freelist object. So there may be a bit of wasted space, which is called *internal fragmentation*.

We can code up this in the function shown in Fig. 12.7. The first thing we do is add 2 to the size, to make room for the header words. We have two pointers to the freelist, pointing to the current and previous object under examination. The outer loop will retry a failed allocation after garbage collection, but only once. If it fails again, 0 is returned to indicate this. The inner loop walks the freelist looking for a free object of sufficient size. If it finds an object that is bigger by at least 3 than the required size, we split the object into two parts, where one remains in the freelist and (a pointer to) the other returned as successful allocation. If it finds an object that is


```

uint64_t *allocate(uint64_t size) {
    size += 2; /* make room for header words */
    for (int i = 0; i<2; i++) {
        uint64_t *previous = NULL;
        uint64_t *current = freelist;
        uint64_t currentSize = 0;

        while (current != NULL) {
            currentSize = *(current-1);
            if (currentSize > size+2) { /* split object */
                *(current+currentSize-size-2) = 0xfff09c169414613;
                *(current+currentSize-size-1) = size;
                *(current-1) -= size;
                return current+(currentSize-size);
            } else if (currentSize >= size) { /* return object */
                if (previous == NULL) freelist = (uint64_t*) *current;
                else *previous = *current;
                return current;
            } else { /* find next object in freelist */
                previous = current;
                current = (uint64_t*) *current;
            }
        }
        if (i > 0) return 0; /* already tried calling gc */
        gc(); /* call garbage collector */
    }
}

```

Fig. 12.7 Code for allocating an object

large enough, but by at most 2, it removes it from the freelist and returns a pointer to it. When removing the object from the freelist, the “next” field in the previous object is made to point to the object after the current object (with a special case when the current object is the first). If the current object is too small, it continues with the next object in the freelist.

Initially, the freelist contains only one object which contains the entire heap memory. After a garbage collection, it can contain multiple objects that are not adjacent in memory.

12.2.3 Manual Freeing

We will, briefly, describe how a `free()` function can be coded using the freelist representation shown in Fig. 12.6, and then return to how freeing can be automated using garbage collection.

We first note that the allocation code in Fig. 12.7 needs to be modified slightly: Since there is no garbage collector, the conditional call to `gc()` at the end of the

Fig. 12.8 Code for manual freeing of objects

```

void myFree(uint64_t *pointer) {
    if (isHeapPointer(pointer)) {
        *pointer = (uint64_t) freelist;
        freelist = pointer;
    }
    else exit(0);
}

```

procedure should be omitted, and after one iteration of the outer loop, it returns 0, so the outer loop can be omitted entirely. Secondly, we can free an object just by adding it to the front of the freelist. But, to detect attempts to call `free()` with something that are not heap pointers, we first check if the argument is a valid heap pointer. If we detect a non-pointer, we stop the program and report an error. The code is shown in Fig. 12.8.

We can also (partially) guard against freeing objects that are already freed by setting the mark bit when the object is allocated and clearing it when it is free. An attempt to free an object that is already freed can then be detected by checking the mark bit.

Note that this simple procedure does not combine adjacent free objects to larger objects, so eventually the freelist will consist of many small objects, making allocation of a large object fail, even though there is sufficient contiguous free memory for the object. This is called *external fragmentation*. It is possible to extend the `free()` procedure to combine memory-adjacent free objects, but we leave this to further reading.

Suggested exercises: 12.2.

12.2.4 The Mark Phase

The first phase of garbage collection identifies and marks the heap objects that are reachable from the *root set*: the set of registers, stack-allocated variables and global variables that can contain pointers into the heap. We assume that the garbage collector has pointers to the top and bottom of the call stack and to the start and end of the space that contains global variables. It will then add all the heap pointers found there to a stack that is used as a work set during garbage collection. We assume, also, that all pointers are allocated at 64-bit aligned locations, so we can go through the root set in 64-bit steps.

After this, we repeat the following steps until the work space is empty:

1. remove a pointer p from the workspace.
2. If p points to a marked object, do nothing.
3. Otherwise, mark the object that p points to and add all heap-pointer fields in the object to the work set.

```

void mark() {
    uint64_t *stack[stack_size];
    uint64_t stackPointer = 0;
    uint64_t *tmp;
    uint64_t size;

    /* add root set to work stack */
    for (uint64_t *i = firstGlobal; i <= lastGlobal; i++)
        if (isHeapPointer((uint64_t*) *i))
            stack[stackPointer++] = (uint64_t*) *i;
    for (uint64_t *i = callStackBottom; i <= callStackTop; i++)
        if (isHeapPointer((uint64_t*) *i))
            stack[stackPointer++] = (uint64_t*) *i;

    while (stackPointer > 0) {
        tmp = stack[--stackPointer];
        size = *(tmp-1);
        if ((size & 0x8000000000000000) == 0) {
            /* if mark bit is clear */
            /* add pointer fields to work stack */
            for (uint64_t *i = tmp; i < tmp + size - 2; i++)
                if (isHeapPointer((uint64_t*) *i))
                    stack[stackPointer++] = (uint64_t*) *i;
            *(tmp-1) |= 0x8000000000000000; /* and set mark bit */
        }
    }
}

```

Fig. 12.9 Code for the mark phase

When the work set is empty, all reachable objects are marked. We can code up the mark phase as shown in Fig. 12.9.

We have used a variable `stack_size` for the size of the work space. The work space can, in the worst case, hold nearly all pointers that point into the heap, including pointer fields in the heap itself. Though this happens rarely, the stack should be large enough to accommodate the worst case, so it should be the same size as the heap. Since the call stack (where local arrays are allocated) is usually not large enough to hold arrays of this size, it is better to either use a globally allocated array or ask the operating system for temporary storage to hold the work-space stack (which is freed after the mark phase). There are methods that store the work space in the heap objects themselves, temporarily modifying pointer fields, but these are complex, so we leave this to further reading.

```

void sweep() {
    uint64_t *current = heapStart;
    uint64_t size, size2;
    freelist = NULL;
    while (current <= heapEnd) {
        size = *(current-1);
        if (size & 0x8000000000000000) { /* mark is set */
            size &= 0x7fffffff;
            *(current-1) = size; /* clear mark */
            current += size;
        } else { /* mark is clear */
            while (current+size < heapEnd &&
                ((size2 = *(current+size-1))
                 & 0x8000000000000000) == 0) {
                *(current+size-2) = 0; /* clear magic word */
                size += size2; /* join blocks */
            }
            /* add block to freelist */
            *(current-1) = size;
            *current = (uint64_t) freelist;
            freelist = current;
            current += size;
        }
    }
}

```

Fig. 12.10 Code for the sweep phase

12.2.5 The Sweep Phase

The purpose of the sweep phase is to find all heap objects that are not marked (and, hence, not reachable from the root set) and add them to the free list. While doing so, it can also combine adjacent free objects into larger objects.

The method is quite simple: Scan the entire heap memory from start to finish, and if an object is marked, simply remove the mark. If an object is not marked, it is checked if the following objects in memory are not marked either, and if this is the case, they are joined into a single block. Regardless, the object is added to the free list. Code for the sweep phase is shown in Fig. 12.10.

Note that the freelist is cleared at the start of the sweep phase, even though it might contain free objects. These objects will be added to the freelist during the sweep phase, so to avoid duplicated, the freelist must be emptied first.

Even though adjacent free objects are joined in the sweep phase, this does not prevent fragmentation: If free objects are separated by in-use objects, the free blocks can not be joined.

12.2.6 *Tying the Knots*

All that is left is to define garbage collection as a sweep phase followed by a mark phase:

```
void gc() {
    mark(); sweep();
}
```

12.2.7 *Remarks About Heap Memory Management*

We have covered only very simple methods for memory management in this chapter. See the Further Reading section below for where to find more detailed information and more advanced algorithms. Some of these algorithms can avoid fragmentation, but at the cost of using twice as much memory as the maximally allocatable heap memory. Given that free lists can take up nearly as much space as the heap memory, this is not significantly worse than the method described above. Furthermore, *generational collection* algorithms can reduce some of the overhead of using garbage collection, and *concurrent collection* algorithms can keep the program running while garbage collection is done, where the method described here stops the program while garbage collection is done. On modern computers, garbage collection of even a few gigabytes of memory takes only a fraction of a second, so this is mainly a problem for applications where even small pauses are unacceptable, such as programs that control machines. There are type-based alternatives to garbage collection: A type system can keep track of when objects can be freed. This comes at the same cost as static type systems always do: You need to write your program with the type system in mind.

Suggested exercises: [12.3](#).

12.3 Pattern Matching

Functional languages in the ML family of languages (and a number of other languages) support *pattern matching*. We saw some examples of pattern matching in the definitions of map functions in Haskell and F# in Sect. [12.1.2](#), where the function distinguishes between empty and non-empty lists by using pattern matching that is integrated in the function definition. Pattern matching can also be a part of a case/switch/match expression, as seen in this alternative way of writing the map function in F#:

Fig. 12.11 Syntax for patterns, rules, and match expressions

$$\begin{aligned}
 \textit{Pattern} &\rightarrow \mathbf{num} \\
 \textit{Pattern} &\rightarrow \mathbf{id} \\
 \textit{Pattern} &\rightarrow [] \\
 \textit{Pattern} &\rightarrow (\textit{Pattern} \ :: \ \textit{Pattern}) \\
 \\
 \textit{Rule} &\rightarrow | \ \textit{Pattern} \ \rightarrow \ \textit{Exp} \\
 \\
 \textit{Rules} &\rightarrow \ \textit{Rule} \\
 \textit{Rules} &\rightarrow \ \textit{Rule} \ \textit{Rules} \\
 \\
 \textit{Exp} &\rightarrow \ \mathbf{match} \ \textit{Exp} \ \mathbf{with} \ \textit{Rules}
 \end{aligned}$$

```

let rec map f ys =
  match ys with
  | [] -> []
  | (x :: xs) -> f x :: map f xs

```

The match expression evaluates the expression between the `match` and `with` keywords and then in order tries to match the rules that each start with the `|` symbol followed by a pattern, the `->` symbol, and an expression. If the pattern in the rule matches the value of the expression, the expression after the arrow symbol is evaluated, and the value of this is the value of the entire match expression. If the pattern does not match, the next rule is tried and so on until a match is found or all rules have been tried without success, in which case an error is reported. If a pattern contains variables (as in the pattern $(x :: xs)$), these are new variables that are bound to the corresponding part of the value when evaluating the expression after the arrow symbol. For example, if `ys` has the value $(3 :: (5 :: []))$ (which is equivalently written as `[3; 5]`), `x` is bound to 3 and `xs` is bound to $(5 :: [])$ (which is equivalently written as `[5]`) when evaluating the expression `f x :: map f xs`. A variable can not occur multiple times in a pattern.

Patterns can be nested. For example, the pattern $(a :: (b :: []))$ (equivalently written as `[a; b]`) matches a list with exactly two elements and binds `a` to the first element and `b` to the second.

In the following, we will limit patterns to be over integers and lists, but the principles carry over to patterns over other data structures. We assume that type checking has already been performed, so a number-constant pattern will only be used to match numbers, list pattern will only be used to match list values, and so on.

We will represent the empty list by the number 0, and a value of the form $(n :: ns)$ by a (non-null) pointer to a heap-allocated structure that holds the number `n` in its first field and `ns` (which is either 0 or a non-null pointer) in its second field. In the pair $(n :: ns)$, `n` is found at offset `head` and `ns` at offset `tail` from where the pointer points into the heap-allocated structure. If the garbage collector shown in Sect. 12.2 is used, `head` would be 0 and `tail` would be 1. Patterns, rules and match expressions will be of the forms shown in Fig. 12.11.

The remaining expression forms are not relevant for the translation below, so we omit these and refer to Fig. 6.3 for how to translate them.

We will use the notation from Chap. 6 and extend Fig. 6.3 with a rule for translating a match expression:

$$\begin{array}{l} \text{Trans}_{Exp}(Exp, vtable, ftable, place) = \text{case } Exp \text{ of} \\ \hline \text{match } Exp_1 \quad \left| \begin{array}{l} place_1 = \text{newvar}() \\ \text{with } Rules_1 \quad \left| \begin{array}{l} label_s = \text{newlabel}() \\ code_1 = \text{Trans}_{Exp}(Exp_1, vtable, ftable, place_1) \\ code_2 = \text{Trans}_{Rules}(Rules_1, vtable, ftable, place_1, place, label_s) \\ code_1 ++ code_2 ++ [\text{LABEL } label_1] \end{array} \right. \end{array} \right. \end{array}$$

Trans_{Rules} will generate code to match the rules against the value in $place_1$, and at a successful match the code will place the value of the right-hand side in $place$ and jump to $label_s$. Rules are tried in sequence, and if no rule matches, the code will jump to a label that reports a “match failed” error:

$$\begin{array}{l} \text{Trans}_{Rules}(Rules, vtable, ftable, place_1, place_2, label) = \text{case } Rules \text{ of} \\ \hline Rule_1 \quad \left| \text{Trans}_{Rule}(Rule_1, vtable, ftable, place_1, place, label, \text{NoMatch}) \right. \\ \hline Rule_1 \quad Rules_1 \quad \left| \begin{array}{l} label_1 = \text{newlabel}() \\ code_1 = \text{Trans}_{Rule}(Rule_1, vtable, ftable, place_1, place, label, label_1) \\ code_2 = \text{Trans}_{Rules}(Rules_1, vtable, ftable, place_1, place, label) \\ code_1 ++ [\text{LABEL } label_1] ++ code_2 \end{array} \right. \end{array}$$

The last two parameters to Trans_{Rule} are labels to jump to if the rule matches or doesn’t match, respectively. The last rule will jump to the error label if it doesn’t match, but other rules will just jump to the next rule. All rules will on success jump to the same label (after evaluating their right-hand sides). A rule is translated by the following translation function:

$$\begin{array}{l} \text{Trans}_{Rule}(Rules, vtable, ftable, place_1, place_2, label_s, label_f) = \text{case } Rule \text{ of} \\ \hline \left| \begin{array}{l} Pattern_1 \quad \left| \begin{array}{l} (code_1, vtable_1) = \text{Trans}_{Pattern}(Pattern_1, vtable, place_1, label_f) \\ \rightarrow Exp_1 \quad \left| \begin{array}{l} code_2 = \text{Trans}_{Exp}(Exp_1, vtable_1, ftable, place) \\ code_1 ++ code_2 ++ [\text{GOTO } label_s] \end{array} \right. \end{array} \right. \end{array} \right. \end{array}$$

The pattern will extend the symbol table, and the right-hand side expression will be evaluated in the extended symbol table and a jump to $label_s$ will be made. If the pattern doesn’t match, a jump to $label_f$ will be made. Patterns are translated using the following translation function:

$$\text{Trans}_{\text{Pattern}}(\text{Pattern}, \text{vtable}, \text{place}, \text{label}) = \text{case Pattern of}$$

num	$v = \text{getvalue}(\text{num})$ $\text{label}_1 = \text{newlabel}()$ $([\text{IF } \text{place} = v \text{ THEN } \text{label}_1 \text{ ELSE } \text{label}, \text{LABEL } \text{label}_1], \text{vtable})$
id	$\text{vtable}_1 = \text{bind}(\text{vtable}, \text{getname}(\text{id}), \text{place})$ $([], \text{vtable}_1)$
[]	$\text{label}_1 = \text{newlabel}()$ $([\text{IF } \text{place} = 0 \text{ THEN } \text{label}_1 \text{ ELSE } \text{label}, \text{LABEL } \text{label}_1], \text{vtable})$
(Pattern₁ :: Pattern₂)	$\text{label}_1 = \text{newlabel}()$ $\text{code}_1 = [\text{IF } \text{place} \neq 0 \text{ THEN } \text{label}_1 \text{ ELSE } \text{label}, \text{LABEL } \text{label}_1]$ $\text{place}_1 = \text{newvar}()$ $\text{code}_2 = [\text{place}_1 := \text{place} + \text{head}, \text{place}_1 := M[\text{place}_1]]$ $\text{place}_2 = \text{newvar}()$ $\text{code}_3 = [\text{place}_2 := \text{place} + \text{tail}, \text{place}_2 := M[\text{place}_2]]$ $(\text{code}_4, \text{vtable}_1) = \text{Trans}_{\text{Pattern}}(\text{Pattern}_1, \text{vtable}, \text{place}_1, \text{label})$ $(\text{code}_5, \text{vtable}_2) = \text{Trans}_{\text{Pattern}}(\text{Pattern}_2, \text{vtable}_1, \text{place}_2, \text{label})$ $(\text{code}_1 ++ \text{code}_2 ++ \text{code}_3 ++ \text{code}_4 ++ \text{code}_5, \text{vtable}_2)$

If the pattern is a number, it is checked if the number is equal to the value and, if not, a jump to *label* is made. If the number is a variable, the symbol table is extended by binding the variable to the value. If the pattern is the empty list, it is checked if the value is 0 (which represents the empty list) and, if not, a jump to *label* is made. Finally, if the pattern is a *::* pattern, the patterns for the head and tail are matched with the values for the head and tail. If the value is the empty list, a jump to *label* is made. Note that the symbol table that is extended by the first pattern is extended again in the second pattern.

While the above completes code generation for pattern matching, it must be noted that the code is far from optimal. For example, if the first pattern is `[]` and the second pattern is a pattern for non-empty lists (a very common case), the value is compared to the 0 in the first pattern, and if this fails, it is compared to 0 again in the second pattern. If patterns are nested, there can be a lot of such redundant checks. It is possible to generate code without redundant checks, but we leave this to further reading.

Suggested exercises: [12.4](#), [12.5](#).

12.4 Further Reading

Read more about polymorphism and type inference in [5, 6]. The latter has a thorough theoretical treatment of all sorts of type systems.

More detail of manual memory management can be found in [2, 3, 5].

More algorithms for automatic memory management can be found in [1, 5, 9]. Two different approaches to type-based memory management are described in [8] and [4].

A method for generating pattern-matching code without redundant checks can be found in [7].

12.5 Exercises

Exercise 12.1 Given the template definition in Sect. 12.1, show the generated (non-template) function definition and call when the program contains a call

```
apply<char>(toupper, text, strlen(text))
```

where `toupper` is a function from characters to characters and `text` is an array of characters.

Exercise 12.2 This exercise considers manual freeing of objects, as discussed in Sect. 12.8.

- In Sect. 12.8, it is mentioned that the code in Fig. 12.7 must be modified for manual freeing (so no call to `gc`, and no outer loop). Show the code with these modifications.
- A bit later, it is mentioned that the mark bit can be used to partially guard against freeing already-freed objects. Modify the code you made above and the code for `free` in 12.8 to implement this.

Exercise 12.3 The mark procedure in Fig. 12.9 can, as mentioned, use stack space proportional to the total amount of heap pointers in the root set and the heap itself. Suggest a modification that reduces this to one word per heap-allocated object. Since an object can contain multiple heap pointers, this can be significantly smaller.

Hint: The code in Fig. 12.9 can add the same pointer multiple times to the stack.

Exercise 12.4 Generate code for the following match statement, using a *vtable* that binds `xs` to `v0`.

```
match xs with
| (x :: []) -> x+1
| (1 :: ys) -> 1
| ys       -> 0
```

Exercise 12.5 Most programming languages that support pattern matching also support *wildcards*, written as underscores. A wildcard matches anything, but does not bind any variables. Extend the *TransPattern* function from Sect. 12.3 with a rule for wildcard patterns.

References

1. Jones, R.E., Lins, R.D.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley (1996)
2. Kamp, P.H.: Malloc(3) revisited. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '98, pp. 36. USENIX Association, Berkeley, CA, USA (1998). <http://dl.acm.org/citation.cfm?id=1268256.1268292>
3. Knuth, D.: The Art of Computer Programming, vol. 1: Fundamental Algorithms. Addison-Wesley (1997)
4. Matsakis, N.D., Klock, F.S., II.: The Rust language **34**(3), 103–104 (2014). <https://doi.org/10.1145/2692956.2663188>
5. Mogensen, T.Æ.: Programming Language Design and Implementation. Springer, Cham, Switzerland (2022). <https://link.springer.com/book/10.1007/978-3-031-11806-7>
6. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)
7. Sestoft, P.: ML pattern match compilation and partial evaluation. In: Danvy, O., Glück, R., Thiemann, P. (eds.) Partial Evaluation, pp. 446–464. Springer, Berlin (1996)
8. Tofte, M., Talpin, J.P.: Region-based memory management. *Inf. Comput.* **132**(2), 109–176 (1997). <https://doi.org/10.1006/inco.1996.2613>, www.sciencedirect.com/science/article/pii/S0890540196926139
9. Wilson, P.R.: Uniprocessor garbage collection techniques. In: IWMM '92: Proceedings of the International Workshop on Memory Management, pp. 1–42. Springer-Verlag, London, UK (1992)

Appendix A

Set Notation and Concepts

In mathematics you don't understand things. You just get used to them.

John von Neumann (1903–1957)

This appendix is primarily a brief run-through of basic concepts from set theory, but it also, in Sect. A.4, mentions set equations, which are not always covered when introducing set theory.

A.1 Basic Concepts and Notation

A set is a collection of items. You can write a set by listing its elements (the items it contains) inside curly braces. For example, the set that contains the numbers 1, 2 and 3 can be written as $\{1, 2, 3\}$. In sets, the order of elements do not matter, so the same set can be written as $\{2, 1, 3\}$, $\{2, 3, 1\}$ or using any permutation of the elements. The number of occurrences also does not matter, so we could also write the set as $\{2, 1, 2, 3, 1, 1\}$, or in an infinity of other ways. All of these describe the same set. We will normally write sets without repetition, but the fact that repetitions do not matter is important to understand the operations on sets.

We will typically use uppercase letters to denote sets and lowercase letters to denote elements in a set, so we could write $M = \{2, 1, 3\}$ and $x = 2$ as an element of M . The empty set can be written either as an empty list of elements ($\{\}$) or using the special symbol \emptyset . The latter is more common in mathematical texts.

A.1.1 Operations and Predicates

We will often need to check if an element belongs to a set, or select an element from a set. We use the same notation for both of these: $x \in M$ is read as “ x is an element of M ” or “ x is a member of M ”. The negation is written as $x \notin M$, which is read as “ x is not an element of M ” or “ x is not a member of M ”.

We can use these in conditional statements like “if $3 \in M$ then ...”, for asserting a fact “since $x \notin M$, we can conclude that ...”, or for selecting an element from a set, as in “select $x \in M$ ”, which will select an arbitrary element from M and let x be equal to this element.

We can combine two sets M and N into a single set that contains all elements from both sets. We write this as $M \cup N$, which is read as “ M union N ” or “the union of M and N ”. For example, $\{1, 2\} \cup \{5, 1\} = \{1, 2, 5, 1\} = \{1, 2, 5\}$. The following statement holds for membership and union:

$$x \in (M \cup N) \Leftrightarrow x \in M \vee x \in N$$

where \Leftrightarrow is bi-implication (“if and only if”) and \vee is logical disjunction (“or”).

We can also combine two sets M and N into a set that contains only the elements that occur in both sets. We write this as $M \cap N$, which is read as “ M intersect N ” or “the intersection of M and N ”. For example, $\{1, 2\} \cap \{5, 1\} = \{1\}$. The following statement holds for membership and intersection:

$$x \in (M \cap N) \Leftrightarrow x \in M \wedge x \in N$$

where \wedge is logical conjunction (“and”).

We can also talk about set difference (or set subtraction), which is written as $M \setminus N$, which is read as “ M minus N ” or “ M except N ”. $M \setminus N$ contains all the elements that are members of M but not members of N . For example, $\{1, 2\} \setminus \{5, 1\} = \{2\}$. The following statement holds for membership and set difference:

$$x \in (M \setminus N) \Leftrightarrow x \in M \wedge x \notin N$$

Just like arithmetic operators, set operators have precedence rules: \cap binds more tightly than \cup (just like multiplication binds tighter than addition). So writing $A \cup B \cap C$ is the same as writing $A \cup (B \cap C)$. Set difference has the same precedence as union (just like subtraction has the same precedence as addition).

If all the elements of a set M are also elements of a set N , we call M a subset of N , which is written as $M \subseteq N$. This can be defined by

$$M \subseteq N \Leftrightarrow (x \in M \Rightarrow x \in N)$$

where \Rightarrow is logical implication (“only if”).

The converse of subset is superset: $M \supseteq N \Leftrightarrow N \subseteq M$.

A.1.2 Properties of Set Operations

Just like we have mathematical laws saying that, for example $x + y = y + x$, there are also similar laws for set operations. Here is a selection of the most commonly used laws:

$A \cup A = A$	union is idempotent
$A \cap A = A$	intersection is idempotent
$A \cup B = B \cup A$	union is commutative
$A \cap B = B \cap A$	intersection is commutative
$A \cup (B \cup C) = (A \cup B) \cup C$	union is associative
$A \cap (B \cap C) = (A \cap B) \cap C$	intersection is associative
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	union distributes over intersection
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	intersection distributes over union
$A \cup \emptyset = A$	the empty set is a unit element of union
$A \cap \emptyset = \emptyset$	the empty set is a zero element of intersection
$A \subseteq B \Leftrightarrow A \cup B = B$	subset related to union
$A \subseteq B \Leftrightarrow A \cap B = A$	subset related to intersection
$A \subseteq B \Leftrightarrow A \setminus B = \emptyset$	subset related to set difference
$A \subseteq B \wedge B \subseteq A \Leftrightarrow A = B$	subset is anti-symmetric
$A \subseteq B \wedge B \subseteq C \Rightarrow A \subseteq C$	subset is transitive
$A \setminus (B \cup C) = (A \setminus B) \setminus C$	corresponds to $x - (y + z) = (x - y) - z$

Since \cup and \cap are associative, we will often omit parentheses and write, e.g., $A \cup B \cup C$ or $A \cap B \cap C$.

A.2 Set-Builder Notation

We will often build a new set by selecting elements from other sets and doing operations on these elements. We use the very flexible *set-builder notation* for this. A set builder has the form $\{e \mid p\}$, where e is an expression and p is a list of predicates separated by commas. Typically, p will contain predicates of the form $x \in M$, which defines x to be any element of M . The set builder will evaluate the expression e for all elements x of M that fulfils the other predicates in p and build a set of the results. We read $\{e \mid p\}$ as “the set of all elements of the form e where p holds”, or just “ e where p ”. Some mathematical texts use a colon instead of a bar, i.e., writing $\{e : p\}$ instead of $\{e \mid p\}$.

A simple example is

$$\{x^3 \mid x \in \{1, 2, 3, 4\}, x < 3\}$$

which builds the set $\{1^3, 2^3\} = \{1, 8\}$, as only the elements 1 and 2 from the set $\{1, 2, 3, 4\}$ fulfil the predicate $x < 3$.

We can take elements from more than one set, for example

$$\{x + y \mid x \in \{1, 2, 3\}, y \in \{1, 2, 3\}, x < y\}$$

which builds the set $\{1 + 2, 1 + 3, 2 + 3\} = \{3, 4, 5\}$. We use all combinations of elements from the two sets that fulfil the predicate.

We can separate the predicates in a set builder by \wedge or “and” instead of commas. So the example above can, equivalently, be written as

$$\{x + y \mid x \in \{1, 2, 3\}, y \in \{1, 2, 3\} \text{ and } x < y\}$$

A.3 Sets of Sets

The elements of a set can be other sets, so we can, for example, have the set $\{\{1, 2\}, \{2, 3\}\}$ which is a set that has the two sets $\{1, 2\}$ and $\{2, 3\}$ as elements. We can “flatten” a set of sets to a single set, which is the union of the element sets, using the “big union” operator:

$$\bigcup\{\{1, 2\}, \{2, 3\}\} = \{1, 2, 3\}$$

Similarly, we can take the intersection of the element sets using the “big intersection” operator:

$$\bigcap\{\{1, 2\}, \{2, 3\}\} = \{2\}$$

We can use these “big” operators together with set builders, for example

$$\bigcap\{\{x^n \mid n \in \{0, 1, 2\}\} \mid x \in \{1, 2, 3\}\}$$

which evaluates to $\bigcap\{\{1\}, \{1, 2, 4\}, \{1, 4, 9\}\} = \{1\}$.

When a big operator is used in combination with a set builder, a special abbreviated notation can be used: $\bigcup\{e \mid p\}$ and $\bigcap\{e \mid p\}$ can be written, respectively, as

$$\bigcup_p e \quad \text{and} \quad \bigcap_p e$$

For example,

$$\bigcap\{\{x^n \mid n \in \{0, 1, 2\}\} \mid x \in \{1, 2, 3\}\}$$

can be written as

$$\bigcap_{x \in \{1, 2, 3\}} \{x^n \mid n \in \{0, 1, 2\}\}$$

A.4 Set Equations

Just like we can have equations where the variables represent numbers, we can have equations where the variables represent sets. For example, we can write the equation

$$X = \{x^2 \mid x \in X\}$$

which states that X is the set of squares of elements from X , i.e., itself. This particular equation has several solutions, including $X = \{0\}$, $X = \emptyset$ and $X = \{0, 1\}$, and even $X = [0, 1]$, where $[0, 1]$ represents the interval of real numbers between 0 and 1. Usually, we have an implied universe of elements that the sets can draw from. For example, if we only want sets of integers as solutions, we won't consider intervals of real numbers as valid solutions.

When there are more than one solution to a set equation, we are often interested in a solution that has the minimum or maximum possible number of elements. In the above example (assuming we want sets of integers), there is a unique minimal (in terms of number of elements) solution, which is $X = \emptyset$, and a unique maximal solution $X = \{0, 1\}$.

Not all equations have unique minimal or maximal solutions. For example, the equation

$$X = \{1, 2, 3\} \setminus X$$

has no solution at all, and the equation

$$X = \{1, 2, 3\} \setminus \{6/x \mid x \in X\}$$

has exactly two solutions: $X = \{1, 2\}$ and $X = \{1, 3\}$, so there are no unique minimal or maximal solutions.

A.4.1 Monotonic Set Functions

The set equations we have seen so far are of the form $X = F(X)$, where F is a function from sets to sets. A solution to such an equation is called a *fixed-point* for F .

As we have seen, not all such equations have solutions, and when they do, there are not always unique minimal or maximal solutions. We can, however, define a property of the function F that guarantees a unique minimal and a unique maximal solution to the equation $X = F(X)$.

Definition A.1 We say that a set function F is *monotonic* if $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$.

Theorem A.2 *If we draw elements from a finite universe U , and F is a monotonic function over sets of elements from U , then there exist natural numbers m and n , so the unique minimal solution to the equation $X = F(X)$ is equal to $F^m(\emptyset)$, and the unique maximal solution to the equation $X = F(X)$ is equal to $F^n(U)$.*

Where $F^i(A)$ is F applied i times to A . For example, $F^3(A) = F(F(F(A)))$.

Proof It is trivially true that $\emptyset \subseteq F(\emptyset)$. Since F is monotonic, this implies $F(\emptyset) \subseteq F(F(\emptyset))$. This again implies $F(F(\emptyset)) \subseteq F(F(F(\emptyset)))$ and, by induction, $F^i(\emptyset) \subseteq F^{i+1}(\emptyset)$. So we have a chain

$$\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq F(F(F(\emptyset))) \subseteq \dots$$

Since the universe U is finite, the sets $F^i(\emptyset)$ can not all be different. Hence, there exist an m such that $F^m(\emptyset) = F^{m+1}(\emptyset)$, which means $X = F^m(\emptyset)$ is a solution to the equation $X = F(X)$. To prove that it is the unique minimal solution, assume that another (not necessarily minimal) solution A exist. Since $A = F(A)$, we have $A = F^m(A)$. Since $\emptyset \subseteq A$ and F is monotonic, we have $F^m(\emptyset) \subseteq F^m(A) = A$. This implies that $F^m(\emptyset)$ is a subset of all solutions to the equation $X = F(X)$, so there can not be a minimal solution different from $F^m(\emptyset)$. \square

The proof for the maximal solution is left as an exercise.

A.4.1.1 Fixed-Point Iteration

The proof provides an algorithm for finding minimal solutions to set equations of the form $X = F(X)$, where F is monotonic and the universe is finite: Simply compute $F(\emptyset)$, $F^2(\emptyset)$, $F^3(\emptyset)$ and so on until $F^{m+1}(\emptyset) = F^m(\emptyset)$. This is easy to implement on a computer:

```
X :=  $\emptyset$ ;
repeat
  Y := X;
  X := F(X)
until X = Y;
return X
```

A.4.2 Distributive Functions

A function can have a stronger property than being monotonic: A function F is *distributive* if $F(X \cup Y) = F(X) \cup F(Y)$ for all sets X and Y . This clearly implies

monotonicity, as $Y \supseteq X \Leftrightarrow Y = X \cup Y \Rightarrow F(Y) = F(X \cup Y) = F(X) \cup F(Y) \supseteq F(X)$.

We also solve set equations over distributive functions with fixed-point iteration, but we exploit the distributivity to reduce the amount of computation we must do: If we need to compute $F(A \cup B)$, and we have already computed $F(A)$, then we need only compute $F(B)$ and add the elements from this to $F(A)$. We can implement an algorithm for finding the minimal solution that exploits this:

```

X := ∅;
W := F(∅);
while W ≠ ∅ do
    pick x ∈ W;
    W := W \ {x};
    X := X ∪ {x};
    W := W ∪ (F({x}) \ X);
return X
    
```

We maintain a work set W that by invariant is equal to $F(X) \setminus X$. A solution must include all $x \in W$, so we move an x from W to X while keeping the invariant by adding $F(x) \setminus X$ to W . When W becomes empty, we have $F(X) = X$ and, hence, a solution. While the algorithm is more complex than the simple fixed-point algorithm, we can compute F one element at a time, and we avoid computing F twice for the same element.

A.4.3 Simultaneous Equations

We sometimes need to solve several simultaneous set equations:

$$\begin{aligned}
 X_1 &= F_1(X_1, \dots, X_n) \\
 &\vdots \\
 X_n &= F_n(X_1, \dots, X_n)
 \end{aligned}$$

If all the F_i are monotonic in all arguments, we can solve these equations using fixed-point iteration. To find the unique minimal solution, start with $X_i = \emptyset$ for $i = 1 \dots n$, and then iterate applying all F_i until a fixed-point is reached. The order in which we do this doesn't change the solution we find (it will always be the unique minimal solution), but it might affect how fast we find the solution. Generally, we need only recompute X_i if a variable used by F_i changes. If all F_i are distributive in all arguments, we can use a work-set algorithm similar to the algorithm for a single distributive function.

If we want the maximal solution, we initialise all X_i with the universe U of elements.

Exercises

Exercise A.3 What set is built by the set builder

$$\{x^2 + y^2 \mid x \in \{1, 2, 3, 4\}, y \in \{1, 2, 3, 4\}, x < y^2\} ?$$

Exercise A.4 What set is built by the set expression

$$\bigcup_{x \in \{1, 2, 3\}} \{x^n \mid n \in \{0, 1, 2\}\} ?$$

Exercise A.5 Find all solutions to the equation

$$X = \{1, 2, 3\} \setminus \{x + 1 \mid x \in X\}$$

Hint: Any solution must be a subset of $\{1, 2, 3\}$, so you can simply try using all the eight possible subsets of $\{1, 2, 3\}$ as candidates for X , and see for which of these the equation holds.

Exercise A.6 Prove that, if elements are drawn from a finite universe U , and F is a monotonic function over sets of elements from U , then there exists an n such that $X = F^n(U)$ is the unique maximal solution to the set equation $X = F(X)$.

Index

A

Abstract syntax, 88, 106
Accept, 78, 82
Action, 29, 88, 89
Activation record, 190
Alias, 201, 202
Alias analysis, 232
Allocation, 149, 203
Alphabet, 2
ARMv7, 163
ARMv8, 163
Array, 148, 248
Assembly, vii
Assignment, 132
Associative, 52, 54
Attribute, 119
 inherited, 119
 synthesised, 119
Available assignments, 219

B

Back-end, 129
Backus-Naur Form, 44
Basic block, 238
Biased colouring, 185
Binding
 dynamic, 100
 static, 100

C

C, viii, 26, 52, 55, 89, 91, 93, 103, 135, 142, 143, 147, 150, 153, 201, 202, 204, 247, 254
Cache, 248
Cache line, 248

Call-by-reference, 201, 202
Call-by-value, 189
Callee-saves, 194, 195
Caller/callee, 189
Caller-saves, 194, 195
Calling convention, 191
Call-sequence, 192
Call stack, 189
Cascading effects of optimisations, 217
Closure, 103, 114
Coalescing, 185
Code generator, 162, 166
Code hoisting, 169, 245, 246
Column-major, 150
Comments
 nested, 30
Common subexpression elimination, 169, 219, 223, 247
Compilation, 112
Compile-time, 134
Conflict, 72, 75, 83, 85, 87
 reduce-reduce, 83, 86
 shift-reduce, 83, 86
Complex Instruction Set Computer (CISC), 164
Conservative collection, 261
Consistent, 20
Constant and copy propagation, 240
Constant in operand, 163
Constant propagation, 170
Context-free
 grammar, 41, 42, 46
 language, 92

D

Dangling-else, 55, 84, 86

Data-flow analysis, 215, 235
 Dead-code elimination, 229
 Dead variable, 165, 174
 Declaration, 99
 global, 99
 local, 99
 Derivation, 46–48, 56, 70
 left, 48, 68
 leftmost, 47
 of regular expressions, 2
 right, 48
 rightmost, 47, 76
 DFA, 8, 12, 32, 77, 79
 combined, 26
 converting NFA to, 13, 16
 equivalence of, 20
 minimisation, 20, 20, 26
 unique minimal, 20
 Digital Vax, 204
 Distributive, 15
 Domain specific language, viii
 Dominance, 239
 Dominance frontier, 239
 Dynamic memory allocation, 260
 Dynamic programming, 166

E

Environment, 100, 120
 Epilogue, 192
 Epsilon-closure, 13
 Epsilon transition, 7
 Execution, 105
 Extended Backus Naur Form (EBNF), 44
 External fragmentation, 265

F

F#, 27, 29, 101, 103
 Finite Automaton (FA), 2, 7, 8
 deterministic, 12
 graphical notation, 8
 nondeterministic, 7
 FIRST, 58, 61
 Fixed-point, 14, 59, 60, 177
 Flag, 162
 arithmetic, 163
 Floating-point constant, 5
 Floating-point numbers, 133
 FOLLOW, 63
 FORTRAN, 26, 202
 Fragmentation, 263, 265
 Frame, 190

Frame pointer, 203
 Free(), 264
 Freelist, 262
 Front-end, 130
 Functional, 101
 Function calls, 133, 161, 189

G

Garbage collection, 260
 Gen and kill sets, 176
 Generational collection, 268
 Generic types, 126
 Global variable, 201
 Go, 78, 80
 Grammar, 56
 ambiguous, 48, 49, 51, 54, 57, 62, 72, 83
 equivalent, 50
 Graph colouring, 180, 180
 Greedy algorithm, 166

H

Hashing, 103
 Haskell, viii, 26, 29, 91, 101, 153
 Heap allocation, 260
 Heuristics, 180, 183

I

IA-32, 185
 IBM System/370, 204
 Imperative, 101
 Implicit polymorphic types, 259
 Implicit types, 126
 In and out sets, 176
 Incrementalisation, 245, 250
 Index check, 152
 elimination, 170, 225
 translation of, 152
 Inlining, 205
 Instruction set description, 166
 Integer, 5, 133
 Interference, 178
 Interference graph, 179
 Intermediate code, vi, 129, 173
 Intermediate language, 131, 161, 169
 tree-structured, 170
 Internal fragmentation, 263
 Interpretation, 105
 of expressions, 108
 of function calls, 110
 of programs, 110
 Interpreter, vii, 105, 129, 131

J

Java, [viii](#), [26](#), [89](#), [130](#), [157](#)
 JavaScript, [118](#)
 Jump, [133](#)
 conditional, [133](#), [162](#)
 Jump-to-jump optimisation, [159](#), [228](#)
 Just-in-time compilation, [130](#)
 JVM, [157](#)

K

Keyword, [5](#)

L

Label, [132](#)
 LALR(1), [86](#), [93](#)
 Language, [2](#), [46](#)
 context-free, [92](#)
 high-level, [129](#)
 Left-associative, [52](#), [85](#)
 Left-derivation, [56](#)
 Left-factorisation, [75](#)
 Left-recursion, [52](#), [54](#), [74](#), [90](#)
 elimination of, [73](#)
 indirect, [74](#)
 Lexer, [25](#), [56](#)
 Lexer generator, [25](#), [29](#)
 Lexical, [1](#)
 analysis, [vi](#), [1](#)
 error, [28](#)
 Lexing, [117](#)
 Linking, [vii](#)
 LISP, [204](#)
 Liveness, [174](#)
 Liveness analysis, [175](#), [216](#)
 Live-range splitting, [185](#)
 Live variable, [174](#), [189](#)
 LL(1), [42](#), [68](#), [69](#), [72](#), [76](#), [84](#), [90](#), [93](#)
 LLVM, [157](#), [236](#)
 Local variables, [189](#)
 Longest prefix, [26](#)
 Lookahead, [68](#)
 Loop, [245](#)
 LR, [76](#)

M

Machine code, [vi](#), [129](#), [131](#), [161](#)
 Machine language, [173](#)
 Malloc(), [260](#)
 Manual freeing, [264](#)
 Mark, [265](#)
 Mark bit, [262](#)

Mark-sweep, [261](#)
 Memory prefetching, [245](#), [248](#)
 Memory transfer, [133](#)
 MIPS, [162](#), [163](#)
 Monotonic, [14](#)

N

Name space, [103](#), [106](#)
 NFA, [8](#), [80](#), [87](#)
 combined, [25](#), [26](#)
 converting to DFA, [13](#), [16](#)
 fragment, [9](#)
 Non-associative, [52](#), [86](#)
 Non-local variable, [200](#)
 Non-recursive, [53](#)
 Nonterminal, [42](#)
 Nullable, [58](#), [61](#)

O

Operator, [133](#)
 Operator hierarchy, [51](#), [52](#)
 Optimisations, [169](#)
 Overloading, [125](#)

P

Parametric polymorphism, [257](#)
 Parser, [50](#)
 generator, [52](#), [86](#), [87](#)
 predictive, [56](#), [62](#)
 shift-reduce, [76](#)
 table-driven, [76](#)
 top-down, [56](#)
 Parsing, [41](#), [48](#), [117](#)
 bottom-up, [56](#)
 predictive, [61](#), [67](#), [68](#)
 table-driven, [69](#)
 Pascal, [52](#), [55](#), [89](#), [93](#), [103](#), [153](#), [154](#), [202](#)
 Pattern, [165](#)
 Pattern matching, [268](#)
 Persistent, [100](#), [101](#)
 Pointer, [201](#), [202](#)
 Pointer analysis, [232](#)
 Polymorphism, [126](#), [257](#)
 PowerPC, [162](#)
 Precedence, [44](#), [50](#), [51](#), [54](#), [76](#), [84](#)
 declaration, [84](#), [85](#), [88](#)
 rules, [52](#)
 Predecessor, [239](#)
 Prefetch, [248](#)
 Production, [42](#), [43](#)
 empty, [43](#), [61](#)

nullable, 58, 62
 Prologue, 192
 Python, 29

R

Recursive descent, 68
 Reduce, 77, 78, 82
 Reduction in strength, 250
 Regexp, 2
 Register, 173
 for passing function parameters, 196
 Register allocation, vi, 161, 173
 by graph colouring, 180
 global, 179
 Register allocator, 198
 Regular expression, 2, 29
 converting to NFA, 9
 equivalence of, 20
 Regular language, 20, 30
 Return address, 190, 196
 Right-associative, 52, 86
 Right-recursion, 53, 54
 RISC, 161, 164, 196
 RISC-V, 163–165, 166, 171, 211
 Root set, 261
 Row-major, 150
 Run-time, 134

S

Scheme, 91, 101
 Scope, 99
 Select, 181
 Sequential logical operators, 142, 143
 Set constraints, 63
 Set equation, 13, 13
 Set-builder notation, 277
 Shift, 77, 78, 80
 Simple Left Right (SLR), 42, 76, 83, 84
 algorithm, 78
 construction of table, 79, 83
 Simplify, 181
 Standard ML (SML), 52, 91, 101, 103, 130
 Space leak, 261
 Sparc, 163
 Spill, 191
 Spill code, 183
 Spilling, 173, 181
 SSA form, 236
 Stack automaton, 41, 92
 Stack pointer, 190, 191
 Standard ML, 101, 103, 130, 154

Starting state, 7
 Start symbol, 42, 56
 State, 7, 8
 accepting, 7–9, 17, 20, 25
 dead, 23
 final, 7
 initial, 7, 8
 starting, 7, 9, 17, 25
 Strongly live variable, 217
 Subset construction, 16
 Sweep, 267
 Symbol table, 100, 100, 108, 120
 implemented as function, 102
 implemented as list, 101
 implemented as stack, 102
 Syntactic category, 45, 106, 119
 Syntax analysis, vi, 1, 41, 46, 48, 56
 Syntax tree, 41, 48, 56, 74

T

Tail call, 207, 210
 Tail-call optimisation, 207
 Tail recursion, 210
 Tail-recursion optimisation, 210
 Templates, 257
 Terminal, 42
 Token, 1, 25, 26, 29, 44, 56
 Transition, 7, 8, 17, 20
 epsilon, 7, 80
 Translation
 of arrays, 148
 of break/exit/continue, 146
 of case-statements, 147
 of declarations, 154
 of expressions, 134
 of function, 199
 of goto, 146
 of index checks, 152
 of logical operators, 141, 143
 of multi-dimensional arrays, 150
 of non-zero-based arrays, 153
 of records/structs, 154
 of statements, 138
 of strings, 153
 Type checking, vi, 112, 117, 120
 of assignments, 125
 of data structures, 125
 of expressions, 120
 of function declarations, 122
 of programs, 123
 Type conversion, 126
 Type error, 120

Type inference, [260](#)

U

Undecidable, [49](#)

Uniform parametric polymorphism, [259](#)

V

Value numbering, [224](#), [241](#)

Variable

 global, [201](#)

 non-local, [200](#)

Variable capture, [205](#)

Variable name, [5](#)

W

Weakly dead variable, [217](#)

White-space, [1](#), [29](#)

Word length, [148](#)

Work-list algorithm, [16](#)

X

X86, [163](#), [185](#)

X86-64, [163](#)