Undergraduate Topics in Computer Science

Torben Ægidius Mogensen

Introduction to Compiler Design

Springer

Undergraduate Topics in Computer Science

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 all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

For further volumes:
www.springer.com/series/7592

Torben Ægidius Mogensen

Introduction to Compiler Design

Springer

Torben Ægidius Mogensen
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
torbenm@diku.dk
url: http://www.diku.dk/~torbenm

ISSN 1863-7310
ISBN 978-0-85729-828-7 e-ISBN 978-0-85729-829-4
DOI 10.1007/978-0-85729-829-4
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2011933601

Cover design: VTeX UAB, Lithuania

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

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, instead, 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 spot 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 spot some obvious 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.

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. 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 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 analysing 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.

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 analyses 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 frontend* of the compiler and the last three phases are collectively called *the backend*. The middle part of the compiler is in this context only the intermediate code generation, but this often includes various optimisations and transformations on the intermediate code.

Each phase, through checking and transformation, establishes stronger invariants on the things it passes on to the next, so that writing each subsequent phase is easier than if these have to take all the preceding into account. For example, the type

checker can assume absence of syntax errors and the code generation can assume absence of type errors.

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, so we will not further discuss these phases in this book.

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 and 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 the interpreter is easier to move to a different machine, so for applications where speed is not of essence, interpreters are often used.

Compilation and interpretation may be combined to implement a programming language: 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, which is interpreted at runtime while other parts may be kept as a syntax tree and interpreted directly. Each choice is a compromise between speed and space: Compiled code tends to be bigger than intermediate code, which tend 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 representation that is closer to the source code than is 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 SML. 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, 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 data-base query languages, text-formatting languages, scene description languages for ray-tracers 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). Even so, all DSL compilers will share similar front-ends for reading and analysing the program text.

Hence, 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.

The Structure of This Book

The first chapters of the book describes the methods and tools required to read program text and convert it into a form suitable for computer manipulation. This process is made in two stages: A lexical analysis stage that basically divides the input text into a list of "words". This is followed by a syntax analysis (or *parsing*) stage that analyses the way these words form structures and converts the text into a data structure that reflects the textual structure. Lexical analysis is covered in Chap. 1 and syntactical analysis in Chap. 2.

The remainder of the book (Chaps. 3–9) covers the middle part and back-end of interpreters and compilers. Chapter 3 covers how definitions and uses of names (*identifiers*) are connected through *symbol tables*. Chapter 4 shows how you can implement a simple programming language by writing an interpreter and notes that this gives a considerable overhead that can be reduced by doing more things before executing the program, which leads to the following chapters about static type checking (Chap. 5) and compilation (Chaps. 6–9. In Chap. 6, it is shown how expressions and statements can be compiled into an *intermediate language*, a language that is close to machine language but hides machine-specific details. In Chap. 7, it is discussed how the intermediate language can be converted into "real" machine code. Doing this well requires that the registers in the processor are used to store the values of variables, which is achieved by a *register allocation* process, as described in Chap. 8. Up to this point, a "program" has been what corresponds to the body of a single procedure. Procedure calls add some issues, which are discussed in Chap. 9.

The book uses standard set notation and equations over sets. Appendix contains a short summary of these, which may be helpful to those that need these concepts refreshed.

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:

- 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. Each chapter has a set of exercises. Few of these require access to a computer, but can be solved on paper or black-board. 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 supplementary material, the book can support course sizes ranging from 5 to 7.5 ECTS points.

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 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 at the compiler courses "Dat 1E" and "Oversættere" at DIKU, and who have found numerous typos and other errors in the earlier versions. I would also like to thank the instructors at Dat 1E and Oversættere, who have pointed out places where things were not as clear as they could be.

Copenhagen, Denmark Torben Ægidius Mogensen

Contents

List of Figures

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 objects 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 its input take a string of individual letters and divide this string into 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 lexer may do the simple parts of the work faster than the more general parser can. Furthermore, the size of a system that is split in two 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

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_1, © Springer-Verlag London Limited 2011

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 the next token ends. Furthermore, a handwritten lexer may be complex and difficult to maintain. Hence, lexers are normally constructed by *lexer generators*, which transform 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 sets of strings, where the individual letters are taken from a particular alphabet. Such 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 underscore).

Given an alphabet, we will describe sets of strings by *regular expressions*, an algebraic notation that is compact and easy for humans to use and understand. The idea is that regular expressions that describe simple sets of strings can be combined to form regular expressions that describe more complex sets of strings.

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`. Hence, when we say, e.g., "The regular expression `s`" we mean the regular expression that describes a single one-letter string "`s`", but when we say "The regular expression s", 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`"}.

Figure 1.1 shows the constructions used to build regular expressions and the languages they describe:

- A single letter describes the language that has the one-letter string consisting of that letter as its only element.
- The symbol ε (the Greek letter *epsilon*) describes the language that consists solely of the empty string. Note that this is not the empty set of strings (see Exercise 1.10).
- $s|t$ (pronounced "s or t") describes the union of the languages described by s and t.

Regular expression	Language (set of strings)	Informal description
`a`	{"`a`"}	The set consisting of the one-letter string "`a`".
ε	{""}	The set containing the empty string.
$s\|t$	$L(s) \cup L(t)$	Strings from both languages
st	$\{vw \mid v \in L(s), w \in L(t)\}$	Strings constructed by concatenating a string from the first language with a string from the second language. Note: In set-formulas, "|" is not a part of a regular expression, but part of the set-builder notation and reads as "where".
s^*	$\{\text{""}\} \cup \{vw \mid v \in L(s),\ w \in L(s^*)\}$	Each string in the language is a concatenation of any number of strings in the language of s.

Fig. 1.1 Regular expressions

- st (pronounced "s t") describes the concatenation of the languages $L(s)$ and $L(t)$, i.e., the sets of strings obtained by taking a string from $L(s)$ and putting this in front of a string from $L(t)$. For example, if $L(s)$ is {"`a`", "`b`"} and $L(t)$ is {"`c`", "`d`"}, then $L(st)$ is the set {"`ac`", "`ad`", "`bc`", "`bd`"}.
- The language for s^* (pronounced "s star") is described recursively: It consists of the empty string plus whatever can be obtained by concatenating a string from $L(s)$ to a string from $L(s^*)$. This is equivalent to saying that $L(s^*)$ consists of strings that can be obtained by concatenating zero or more (possibly different) strings from $L(s)$. If, for example, $L(s)$ is {"`a`", "`b`"} then $L(s^*)$ is {"", "`a`", "`b`", "`aa`", "`ab`", "`ba`", "`bb`", "`aaa`", ...}, i.e., any string (including the empty) that consists entirely of `a`s and `b`s.

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 {"`a`", "`bb`"}. When doing so, we will use ε to denote the empty string, so the example from $L(s^*)$ above is written as {ε, `a`, `b`, `aa`, `ab`, `ba`, `bb`, `aaa`, ...}. 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, `ab`w denotes any string starting with `ab`.

Precedence Rules When we combine different constructor symbols, e.g., in the regular expression `a|ab`*, it is not *a priori* clear how the different subexpressions are grouped. We can 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 $3 + 4 * 5$ means 3 added to the product of 4 and 5 and not multiplying the sum of 3 and 4 by 5. For regular expressions, we use the following conventions: * binds tighter than concatenation, which binds tighter than alternative (|). The example `a|ab`* from above, hence, is equivalent to `a|(a(b`*`))`.

The | operator is associative and commutative (as it corresponds to set union, which has these properties). Concatenation is associative (but obviously not commutative) and distributes over |. Figure 1.2 shows these and other algebraic properties of regular expressions, including definitions of some of the shorthands introduced below.

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 it is 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. Sequences of letters within square brackets represent 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 write, e.g., `[0–z]` it is not immediately clear what is meant. When using such notation in lexer generators, standard ASCII or ISO 8859-1 character sets are usually used, with the hereby implied ordering of symbols. To avoid confusion, we will 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 allowed. Such non-zero repetition is quite common, so we introduce another shorthand, s^+, to denote one or more

$(r\|s)\|t = r\|s\|t = r\|(s\|t)$	$\|$ is associative.
$s\|t = t\|s$	$\|$ is commutative.
$s\|s = s$	$\|$ is idempotent.
$s? = s\|\varepsilon$	by definition.
$(rs)t = rst = r(st)$	concatenation is associative.
$s\varepsilon = s = \varepsilon s$	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.

Fig. 1.2 Some algebraic properties of regular expressions

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 $s|\varepsilon$. $^+$ and ? bind with the same precedence as *.

We must stress that these shorthands are just that. They do not add anything to the set of languages we can describe, they just make it possible to describe a language 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.

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, the sign is a separate symbol and not

part of the constant itself. This will allow whitespace between the sign and the number, which is not possible with the above.

Floats. 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

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

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):

$$"([a-zA-Z0-9]|\backslash[a-zA-Z])^*"$$

Suggested exercises: 1.1, 1.10(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 these. A transition between 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 so, the string we have read from the input is in the language defined by the automaton.

We may have a choice of several actions at each step: 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 input. 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 recognized 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 the 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 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, possibly 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 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 marked 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 unmarked arrow 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:

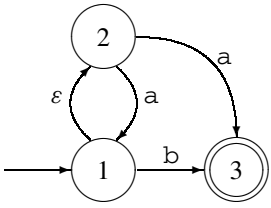
from	to	by
1	2	ε
2	1	`a`
1	2	ε
2	1	`a`
1	3	`b`

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. But, as previously stated, it is enough that there *exists* a path leading to acceptance, so the string `aab` is still accepted.

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 which are too time-consuming to make NFAs suitable for efficient recognisers. We will, hence, use NFAs only as a stepping stone between regular expressions

Fig. 1.3 Example of an NFA



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.

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.

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|⋯|9` and $s?$ to $s|\varepsilon$, etc. However, this will result in very large NFAs for some expressions, so we use a few optimised constructions for the shorthands. 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

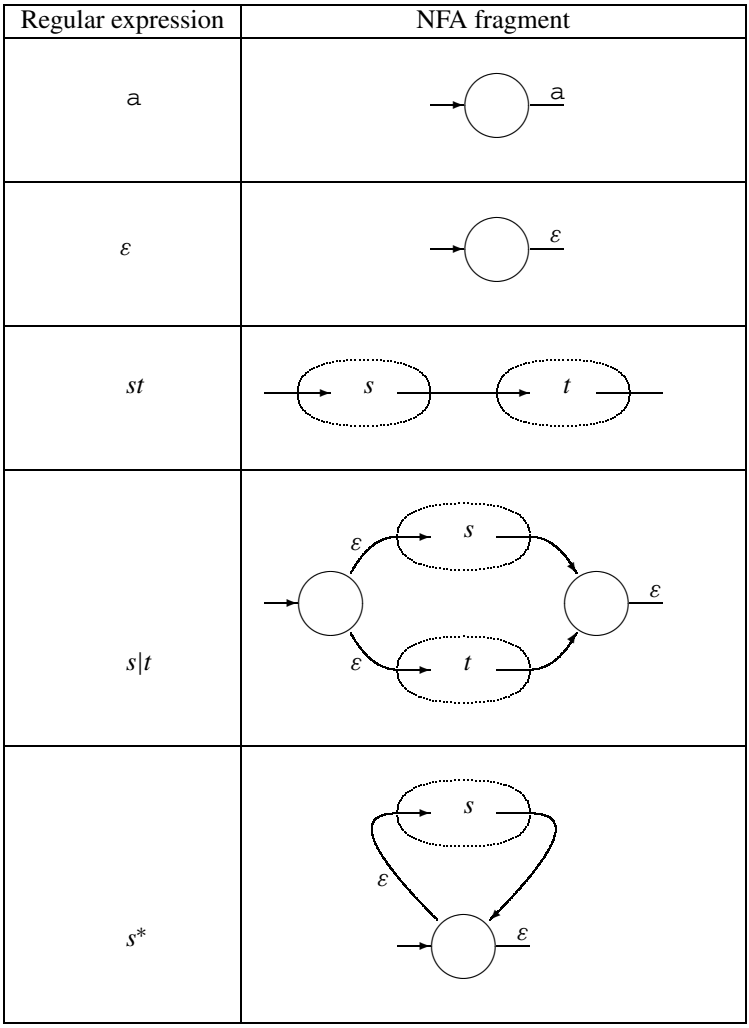


Fig. 1.4 Constructing NFA fragments from regular expressions

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 $s|\varepsilon$ will do fine if we use the optimised construction for ε.

The optimised constructions are shown in Fig. 1.6. 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 make an NFA for this language using only two states.

Suggested exercises: 1.2(a), 1.10(b).

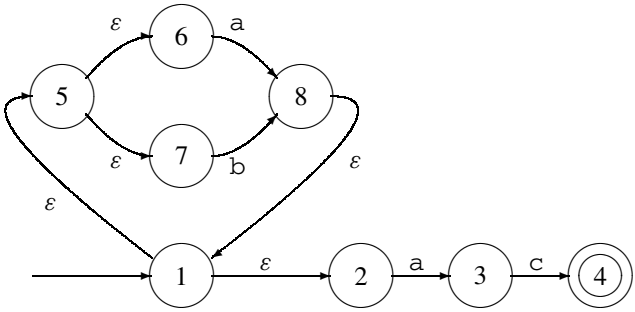


Fig. 1.5 NFA for the regular expression (a|b)*ac

Regular expression	NFA fragment
ε	—
[0–9]	0 … 9, ε
s^+	s, ε, ε, ε

Fig. 1.6 Optimised NFA construction for regular expression shorthands

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

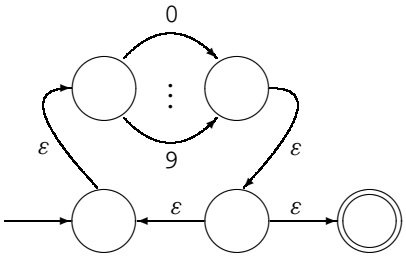


Fig. 1.7 Optimised NFA for $[0\text{–}9]^+$

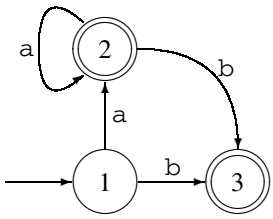


Fig. 1.8 Example of a DFA

automaton: The deterministic finite automaton, or DFA for short. DFAs are NFAs, but 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.

The transition relation of a DFA is a (partial) function, and we often write it as such: $move(s, c)$ 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: 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, this 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: 1.7(a, b), 1.8.

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 paths in an NFA at once. 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. The idea is that such a set of NFA states will become a single DFA state. For any given symbol we form the set of all possible next-states in the NFA, so we get a single transition (labelled by that symbol) going from one set of NFA states to another set. Hence, the transition becomes deterministic in the DFA that is formed from the sets of NFA states.

Epsilon-transitions complicate the construction a bit: Whenever we are in an NFA state we can always choose to follow an 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 first extending the set of NFA states with those you can reach from these using only epsilon-transitions. Then, for each possible input symbol, we follow transitions with this symbol to form a new set of NFA states. We define the *epsilon-closure* of a set of states as the set extended with all 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 $\varepsilon\text{-}closure(M)$ to be the least (in terms of the subset relation) solution to the set equation

$$\begin{aligned}&\varepsilon\text{-}closure(M)\\&\quad = M \cup \{t \mid s \in \varepsilon\text{-}closure(M) \text{ and } s^{\varepsilon} t \in T\}\end{aligned}$$

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 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, ε-*closure*(M) is the value we have to find, so we make an equation such that the value of X that solves the equation will be ε-*closure*(M):

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

So, if we define F_M to be

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

then a solution to the equation $X = F_M(X)$ will be ε-*closure*(M).

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*.

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 X that satisfies the equation).

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 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 a new guess at S. We now form the chain

$$\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq \ldots$$

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), 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 ε-*closure*($\{1\}$), $M = \{1\}$, so $F_M = F_{\{1\}}$. We start by guessing the empty set:

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

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

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

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

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

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

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

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

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

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

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:

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

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 *move* by:

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

The DFA uses the same alphabet as the NFA.

A little explanation:

- The starting state of the DFA is the epsilon-closure of the set containing just the starting state of the NFA, i.e., the states that are reachable from the starting state by epsilon-transitions.
- A transition in the DFA is done by finding the set of NFA states that comprise the DFA state, following all transitions (on the same symbol) in the NFA from all these NFA states and finally combining the resulting sets of states and 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 in the DFA is an accepting state if at least one of the NFA states it contains 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

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

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

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

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

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:

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

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

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

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

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

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

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

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

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

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

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

Which now 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: 1.2(b), 1.4.

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 might not be in a suitably simple class that do not risk

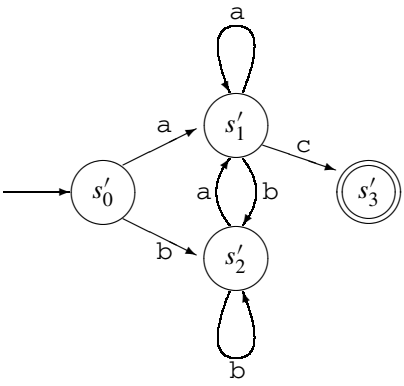


Fig. 1.9 DFA constructed from the NFA in Fig. 1.5

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 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 minimal DFA.

[1] If we do not consider the effects of cache-misses etc.

Hence, we can decide equivalence of 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. 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 states from D. The groups in the minimal DFA are *consistent*: For any pair of states s_1, s_2 in the same group G_1 and any symbol c, $move(s_1, c)$ is in the same group G_2 as $move(s_2, c)$ or both are undefined. In other words, we can not tell s_1 and s_2 apart by looking at their transitions.

We minimize 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 $S \setminus F$. These are 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. *All* groups are then unmarked. A consistent subgroup is maximal if adding any other state to it will make it inconsistent.
3) If there are no unmarked groups left, we are done and the remaining groups are the states 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 $O(n * \log(n))$, 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.

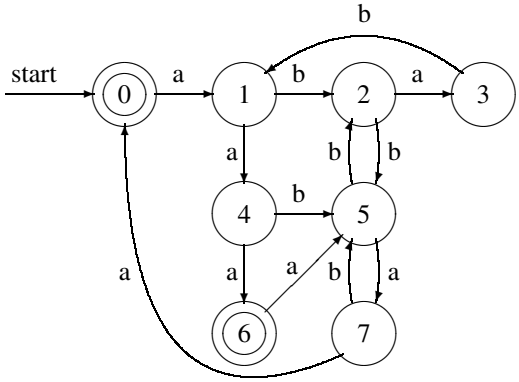


Fig. 1.10 Non-minimal DFA

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.

$$G_1 = \{0, 6\}$$
$$G_2 = \{1, 2, 3, 4, 5, 7\}$$

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):

$$G_1 = \{0, 6\}$$
$$G_3 = \{1, 2, 5\}$$
$$G_4 = \{3\}$$
$$G_5 = \{4, 7\}$$

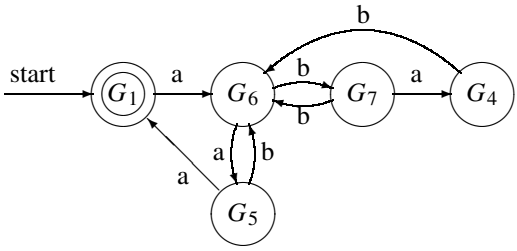


Fig. 1.11 Minimal DFA

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:

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 (except the singletons). Hence, the groups form a minimal DFA equivalent to the one in Fig. 1.10. The minimised DFA is shown in Fig. 1.11.

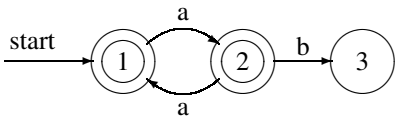
1.7.2 Dead States

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. Such 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 has only one 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: 1.5, 1.10(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 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 $r_1, r_2, \ldots, r_n$, then the regular expression $r_1 \mid r_2 \mid \ldots \mid r_n$ describes the union of the languages $r_1, r_2, \ldots, r_n$ 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 DFA. We can accomplish this with the following construction of a combined DFA:

1) Construct NFAs $N_1, N_2, \ldots, N_n$ for each of $r_1, r_2, \ldots, r_n$.
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.

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`

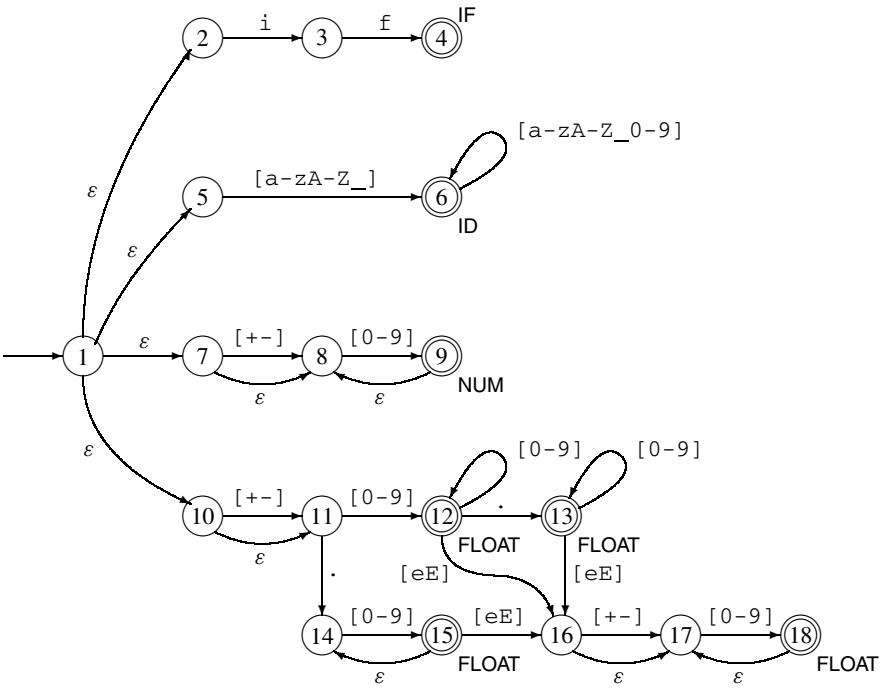


Fig. 1.12 Combined NFA for several tokens

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 SML follow this convention, and so do most lexer generators, but some (mostly older) languages like FORTRAN do not. 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 side-stepped. 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 can re-insert these in the input in such situations. 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.

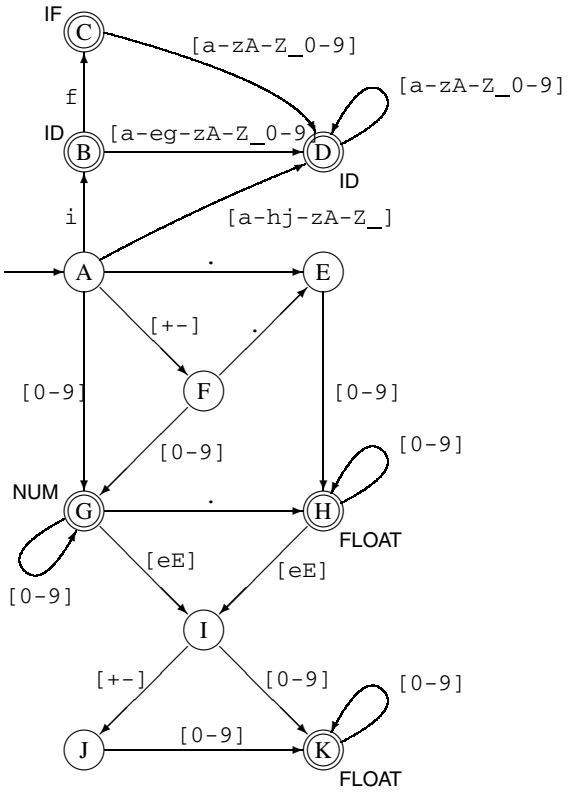


Fig. 1.13 Combined DFA for several tokens

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 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.

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. Less commonly, the lexer may work interactively with a text editor and restart from the point at which an error was spotted after the user has tried to fix the error.

1.8.1 Lexer Generators

A lexer generator will typically use a notation for regular expressions similar to the one described in Sect. 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 `'*'`. 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 the symbols inside a comment up to the terminating character(s).

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, perhaps, 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.

An action can be more than just returning a token. If, for example, a language has a large number of keywords, then a DFA that recognises all of these individually 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 a keyword). A similar strategy can be used if the language allows identifiers to shadow keywords.

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

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 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. Hence, we will now state a few properties of regular languages and give some examples of some 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

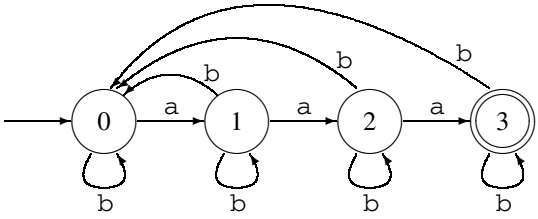


Fig. 1.14 A 4-state NFA that gives 15 DFA states

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 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 1.9), but it requires a very complex regular expression to do so. For programming language tokens, regular expression 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. 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 is 1 in the number if and only if state i is in the set. The set that contains only the initial NFA state is, hence, represented by the number 1. 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:

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

This is not hard to verify, so we leave this to the interested reader. It is also easy to see that these two operations can generate any n-bit number from the number 1. 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 an a leads from state i to $i+1$ (if $i < n-1$), so for each NFA state i there is exactly one sequence of as that leads to the accepting state, and that sequence has $n-1-i$ as. Hence, a DFA state whose subset contains the NFA state i will lead to acceptance on a string of $n-1-i$ as, 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 $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$, that is, any sequence of as followed by a sequence of the *same number* of bs. 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 as, know how many there were, so we can compare this to the number of bs. 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 million lines 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 million lines, it is not practical to describe 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 1.9).

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 this intersection construction, 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: 1.11.

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] and [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 minimization 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]. The latter generates 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.

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 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.

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

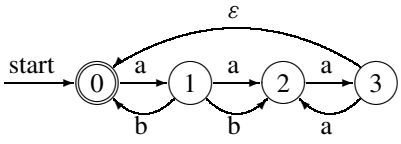
Exercise 1.2 Given the regular expression `a`*`(a|b)aa`:

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

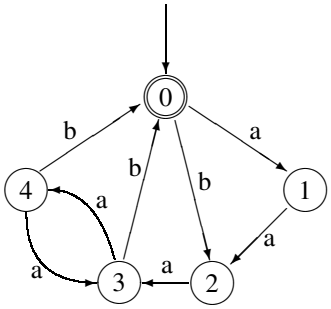
Exercise 1.3 Given the regular expression `((a|b)(a|bb))`*:

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

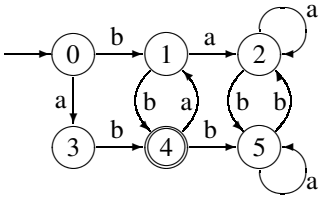
Exercise 1.4 Make a DFA equivalent to the following NFA:



Exercise 1.5 Minimise the following DFA:



Exercise 1.6 Minimise the following DFA:



Exercise 1.7 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 `b`s (and any number of `a`s).
b) The set of strings where the number of `b`s is a multiple of 3 (and there can be any number of `a`s).
c) The set of strings where the difference between the number of `a`s and the number of `b`s is a multiple of 3.

Exercise 1.8 Construct a DFA that recognises balanced sequences of parenthesis with a maximal nesting depth of 3, e.g., ε, ()(), (()(())) or (()())()() but not (((()))) or (()(()(()))).

Exercise 1.9 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 1.10 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 1.11 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 1.12 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 1.13 In Figs. 1.12 and 1.13 we used character sets on transitions as shorthands 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.

a) 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.
b) 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 1.14 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 $c_1 c_2$. 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 1/256th of the space used by the original DFA.

a) How much larger is the new NFA compared to the old?
b) 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?
c) Roughly, how much time does the new DFA require to analyse a string compared to the old?
d) 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 1.15 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:

$$
\begin{aligned}
\mathit{nonempty}(\varepsilon) &= \phi \\
\mathit{nonempty}(\mathtt{a}) &= \ldots \qquad \text{where } \mathtt{a} \text{ is an alphabet symbol} \\
\mathit{nonempty}(s|t) &= \mathit{nonempty}(s) \,|\, \mathit{nonempty}(t) \\
\mathit{nonempty}(s\,t) &= \ldots \\
\mathit{nonempty}(s?) &= \ldots \\
\mathit{nonempty}(s^*) &= \ldots \\
\mathit{nonempty}(s^+) &= \ldots
\end{aligned}
$$

where ϕ is the regular expression for the empty language (see Exercise 1.10).

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 $\mathit{nonempty}(\mathtt{a}^*\mathtt{b}^*)$.

Exercise 1.16 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:

$$\begin{aligned}
&\mathit{prefixes}(\varepsilon) &&= \varepsilon \\
&\mathit{prefixes}(\mathtt{a}) &&= \mathtt{a}? && \text{where } \mathtt{a} \text{ is an alphabet symbol} \\
&\mathit{prefixes}(s|t) &&= \mathit{prefixes}(s) \,|\, \mathit{prefixes}(t) \\
&\mathit{prefixes}(s\,t) &&= \ldots \\
&\mathit{prefixes}(s^*) &&= \ldots \\
&\mathit{prefixes}(s^+) &&= \ldots
\end{aligned}$$

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, $\mathtt{a}$ and $s|t$.
b) Use this definition to find $\mathit{prefixes}(\mathtt{ab}^*\mathtt{c})$.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers; Principles, Techniques and Tools. Addison-Wesley, Reading (2007)
3. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge (1998)
4. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the 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, Reading (2001)
6. Keller, J.P., Paige, R.: Program derivation with verified transformations—a case study. Communications in Pure and Applied Mathematics **48**(9–10), 1053–1113 (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 Transactions on Electronic Computers **9**(1), 39–47 (1960)
9. Milner, R.: Communication and Concurrency. Prentice-Hall, New York (1989)
10. Owens, S., Reppy, J., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. **19**(2), 173–190 (2009). doi: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

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–)

Where lexical analysis splits the input 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 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 two ways of generating such automata. The first of these, LL(1), is relatively simple,

[1] The name refers to the fact that derivation is independent of context.

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_2, © Springer-Verlag London Limited 2011

but works only for a somewhat restricted class of grammars. The SLR construction, which we present later, is more complex but accepts a wider class of grammars. Sadly, neither of these work for all context-free grammars. Tools that handle all context-free grammars exist, but they can incur a severe speed penalty, which is why most parser generators restrict the class of input grammars.

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 language described by the grammar. This is called the *start symbol* of the grammar. 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

$$N \to X_1 \dots X_n$$

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 singleton set, just like alphabet characters in regular expressions. We will, when no confusion is likely, equate a nonterminal with the set of strings it denotes

Some examples:

$$A \to \mathtt{a}$$

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

$$A \to \mathtt{a}A$$

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 $\mathtt{a}^+$.

We can define a grammar equivalent to the regular expression $\mathtt{a}^*$ by the two productions

$$B \to$$
$$B \to \mathtt{a}B$$

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 sometimes 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 $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ is not regular. It is, however, easily described by the grammar

$$\begin{aligned} S &\rightarrow \\ S &\rightarrow \mathtt{a}S\mathtt{b} \end{aligned}$$

The second production ensures that the as and bs 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

$$\begin{aligned} T &\rightarrow R \\ T &\rightarrow \mathtt{a}T\mathtt{a} \\ R &\rightarrow \mathtt{b} \\ R &\rightarrow \mathtt{b}R \end{aligned}$$

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

Sometimes, 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

$$\begin{aligned} T &\rightarrow R \mid \mathtt{a}T\mathtt{a} \\ R &\rightarrow \mathtt{b} \mid \mathtt{b}R \end{aligned}$$

There are still four productions in the grammar, even though the arrow symbol $\rightarrow$ is only used twice.

2.1.1 How to Write Context Free Grammars

As hinted above, a regular expression can systematically be rewritten as a 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 think of a way of expressing a language as a regular expression, it is easy to make a grammar for it. However, we will also want to use grammars to describe non-regular languages. An example is the kind of arithmetic

Fig. 2.1 From regular expressions to context free grammars

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

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

Grammar 2.2 Simple expression grammar

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

expressions that are part of most programming languages (and also found on electronic calculators). Such expressions can be described by Grammar 2.2. Note that, as mentioned in Sect. 1.9, the matching parentheses can not be described by regular expressions, as these can not "count" the number of unmatched opening parentheses at a particular point in the string. However, if we did not have parentheses in the language, it could be described by the regular expression

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

Even so, the regular description 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. We will look at structure in Sects. 2.2.1 and 2.3.

Most constructions from programming languages are easily expressed by context free grammars. In fact, most modern languages are designed this way.

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 concept. Examples of common syntactic categories in programming languages are:

Grammar 2.3 Simple statement grammar

$$
\begin{aligned}
&\mathit{Stat} \to \textbf{id} := \mathit{Exp}\\
&\mathit{Stat} \to \mathit{Stat}\,;\,\mathit{Stat}\\
&\mathit{Stat} \to \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat}\ \textbf{else}\ \mathit{Stat}\\
&\mathit{Stat} \to \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat}
\end{aligned}
$$

Expressions	are used to express calculation of values.
Statements	express actions that occur in a particular sequence.
Declarations	express properties of names used in other parts of the program.

Each syntactic category is denoted by a main nonterminal, e.g., *Exp* from Grammar 2.2. More nonterminals might be needed to describe a syntactic category or provide structure to it, as we shall see, and 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.

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. Since the productions are similar to recursive set equations, we might expect to use the techniques from Sect. 1.5.1 to find the set of strings denoted by a grammar. However, though these methods in theory apply to infinite sets by considering limits of chains of sets, they are only practically useful when the sets are finite. Instead, we below introduce the concept of *derivation*. An added advantage of this approach 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 production in which 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 $N \to \gamma$
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

Grammar 2.4 Example grammar

$$
\begin{aligned}
T &\to R \\
T &\to \mathtt{a}T\mathtt{c} \\
R &\to \\
R &\to R\mathtt{b}R
\end{aligned}
$$

Fig. 2.5 Derivation of the string `aabbbcc` using Grammar 2.4

$$
\begin{aligned}
&\underline{T} \\
\Rightarrow\ &\mathtt{a}\underline{T}\mathtt{c} \\
\Rightarrow\ &\mathtt{aa}\underline{T}\mathtt{cc} \\
\Rightarrow\ &\mathtt{aa}\underline{R}\mathtt{cc} \\
\Rightarrow\ &\mathtt{aa}R\mathtt{b}\underline{R}\mathtt{cc} \\
\Rightarrow\ &\mathtt{aa}\underline{R}\mathtt{bcc} \\
\Rightarrow\ &\mathtt{aa}R\mathtt{b}\underline{R}\mathtt{bcc} \\
\Rightarrow\ &\mathtt{aa}R\mathtt{b}\underline{R}\mathtt{b}R\mathtt{bcc} \\
\Rightarrow\ &\mathtt{aa}\underline{R}\mathtt{bb}R\mathtt{bcc} \\
\Rightarrow\ &\mathtt{aabb}\underline{R}\mathtt{bcc} \\
\Rightarrow\ &\mathtt{aabbbcc}
\end{aligned}
$$

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]

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

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 strings 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 string `aabbbcc` by the derivation shown in Fig. 2.5. We have, for clarity, in each sequence of symbols underlined the nonterminal that is 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]The mathematically inclined will recognise that derivation is a preorder on sequences of grammar symbols.

Fig. 2.6 Leftmost derivation of the string `aabbbcc` using Grammar 2.4

$$
\begin{aligned}
&\underline{T}\\
\Rightarrow\ &\texttt{a}\underline{T}\texttt{c}\\
\Rightarrow\ &\texttt{aa}\underline{T}\texttt{cc}\\
\Rightarrow\ &\texttt{aa}\underline{R}\texttt{cc}\\
\Rightarrow\ &\texttt{aa}\underline{R}\texttt{b}R\texttt{cc}\\
\Rightarrow\ &\texttt{aa}\underline{R}\texttt{b}R\texttt{b}R\texttt{cc}\\
\Rightarrow\ &\texttt{aab}\underline{R}\texttt{b}R\texttt{cc}\\
\Rightarrow\ &\texttt{aab}\underline{R}\texttt{b}R\texttt{b}R\texttt{cc}\\
\Rightarrow\ &\texttt{aabb}\underline{R}\texttt{b}R\texttt{cc}\\
\Rightarrow\ &\texttt{aabbb}\underline{R}\texttt{cc}\\
\Rightarrow\ &\texttt{aabbbcc}
\end{aligned}
$$

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, form the derived string. If a nonterminal is rewritten using an empty production, an ε is shown as its child. This is also a leaf node, but is ignored when reading the string from the leaves of the tree.

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 that it derives. It is this structure that we exploit in the later phases of the compiler.

For compilation, we do the derivation 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 that 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, 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, or external rules can be applied to decide which of the many possible syntax trees is the "right one". An unambiguous version of Grammar 2.4 is shown in Fig. 2.9.

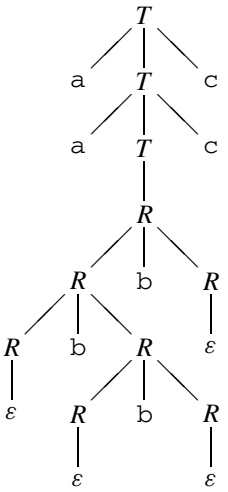


Fig. 2.7 Syntax tree for the string `aabbbcc` using Grammar 2.4

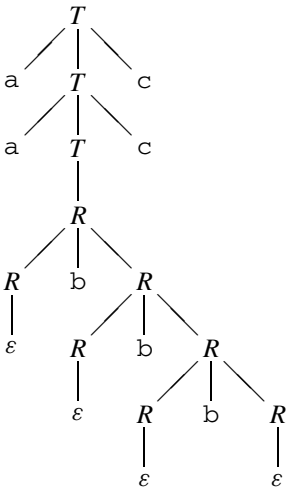


Fig. 2.8 Alternative syntax tree for the string `aabbbcc` using Grammar 2.4

Grammar 2.9
Unambiguous version of Grammar 2.4

$$T \to R$$
$$T \to \mathtt{a}T\mathtt{c}$$
$$R \to$$
$$R \to \mathtt{b}R$$

How do we know if a grammar is ambiguous? If we can find a string and show two alternative syntax trees for it, this is a proof of ambiguity. It may, however, be hard to find such a string and, when the grammar is unambiguous, even harder to show that this is the case. 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, any grammar that has a production of the form

$$N \to N\alpha N$$

where α is any sequence of grammar symbols, 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 also fail on certain unambiguous grammars, so they can not be used to prove 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. 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 (though they may impose different structures on the strings of the language).

Given two grammars, it would be nice to be able to tell if they are equivalent. Unfortunately, no known method is able to decide this in all cases, but, unlike ambiguity, it is not (at the time of writing) known if such a method may or may not theoretically exist. Sometimes, equivalence can be proven e.g. by induction over the set of strings that the grammars produce. The converse 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.

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).

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

$$Exp \to Exp + Exp$$

which has the form that in Sect. 2.2.1 was claimed to imply ambiguity. This ambiguity is not 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

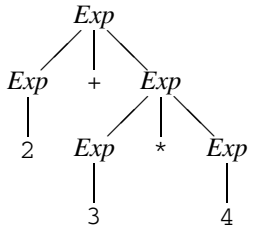


Fig. 2.10 Preferred syntax tree for `2+3*4` using Grammar 2.2

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 in Fig. 2.10, 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 possible way of resolving the ambiguity is to use precedence rules during syntax analysis 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. To clarify this, we first define some concepts:

- 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. However, to avoid ambiguity we have to choose one of these. By convention (and similarity to `-` and `/`) we choose to let these be left-associative as well. 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 -> b -> c` is read as `a -> (b -> c)`. The assignment operator in C is also right-associative: `a=b=c` is read as `a=(b=c)`.

In some languages (like Pascal), comparison operators (like $<$ or $>$) 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

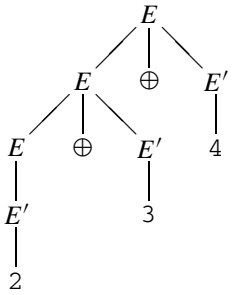
$$E \to E \oplus E$$
$$E \to \textbf{num}$$

we can rewrite this to an unambiguous grammar that generates the correct structure. As this depends on the 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:

$$E \to E \oplus E'$$
$$E \to E'$$
$$E' \to \textbf{num}$$

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



We get a slightly more complex syntax tree than in Fig. 2.10, but not enormously so.

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

$$E \to E' \oplus E$$
$$E \to E'$$
$$E' \to \textbf{num}$$

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

$$E \to E' \oplus E'$$
$$E \to E'$$
$$E' \to \textbf{num}$$

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

Grammar 2.11
Unambiguous expression grammar

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

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 −:

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

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

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

seems like an obvious generalisation of the principles used above, giving + and ⊕ 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 **num**+**num**⊕**num** is not derivable by this grammar.

In general, there is no obvious way to resolve ambiguity in an expression like `1+2⊕3`, where + is left-associative and ⊕ 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 2.11 shows how these rules are used to make an unambiguous version of Grammar 2.2. Figure 2.12 show the syntax tree for `2+3*4` using this grammar.

Suggested exercises: 2.6.

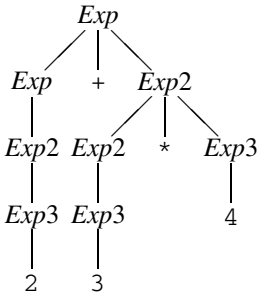


Fig. 2.12 Syntax tree for `2+3*4` using Grammar 2.11

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 s1 else s2
```

According to the grammar, the `else` can equally well match either `if`. The usual convention 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` as a kind of right-associative operators, as this would 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 `if`s that occur between these must have matching `else`s. 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 2.13. 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 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 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).

$$
\begin{array}{ll}
\mathit{Stat} & \rightarrow \mathit{Stat}2\,;\,\mathit{Stat} \\
\mathit{Stat} & \rightarrow \mathit{Stat}2 \\
\mathit{Stat}2 & \rightarrow \mathit{Matched} \\
\mathit{Stat}2 & \rightarrow \mathit{Unmatched} \\
\mathit{Matched} & \rightarrow \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Matched}\ \texttt{else}\ \mathit{Matched} \\
\mathit{Matched} & \rightarrow \mathbf{id} := \mathit{Exp} \\
\mathit{Unmatched} & \rightarrow \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Matched}\ \texttt{else}\ \mathit{Unmatched} \\
\mathit{Unmatched} & \rightarrow \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Stat}2
\end{array}
$$

Grammar 2.13 Unambiguous grammar for statements

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 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 guess right. 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, to the left of the rewritten nonterminals, there are only terminals. These terminals 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 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

$$T \to \texttt{a}T\texttt{c}$$

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

$$T \to R$$

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 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 on the next input symbol, we are able to do predictive parsing without backtracking.

2.7 *Nullable* and *FIRST*

In simple cases, like the above, all productions for a nonterminal start with distinct terminals except at most one production that does not start with a terminal. We chose the latter whenever the input symbol did not match any of the terminal symbols starting the other productions. We can extend the method to work also for grammars where several productions start with nonterminals. We just need to be able to select between them based on the input symbol. In other words, if the strings these productions can derive begin with symbols from disjoint sets, we check if the input symbol is in one of these sets and choose the corresponding production if it is. If not, 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 $\mathit{FIRST}(\alpha)$ if and only if $\alpha \Rightarrow c\beta$ for some (possibly empty) sequence β of grammar symbols.

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 $\mathit{Nullable}(\alpha)$) if and only if $\alpha \Rightarrow \varepsilon$.

A production $N \to \alpha$ 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

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

where $\mathtt{a}$ is a terminal, N is a nonterminal, α and β are sequences of grammar symbols and ε represents 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, but 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, 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 several equations instead of one. Still, the method is essentially the same: We have a set of boolean equations:

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

We initially assume $X_1, \ldots, X_n$ 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 equations are satisfied. In 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 true, the result will also be more true (i.e., it may stay unchanged, change from *false* to *true*, but never change from *true* to *false*).

If we look at Grammar 2.9, we get these equations for nonterminals and right-hand sides:

$$\begin{aligned}
Nullable(T) &= Nullable(R) \vee Nullable(\mathtt{a}T\mathtt{c}) \\
Nullable(R) &= Nullable(\varepsilon) \vee Nullable(\mathtt{b}R)
\end{aligned}$$

$$\begin{aligned}
Nullable(R) &= Nullable(R) \\
Nullable(\mathtt{a}T\mathtt{c}) &= Nullable(\mathtt{a}) \wedge Nullable(T) \wedge Nullable(\mathtt{c}) \\
Nullable(\varepsilon) &= true \\
Nullable(\mathtt{b}R) &= Nullable(\mathtt{b}) \wedge Nullable(R)
\end{aligned}$$

Right-hand side	Initialisation	Iteration 1	Iteration 2	Iteration 3
R	*false*	*false*	*true*	*true*
$\mathtt{a}T\mathtt{c}$	*false*	*false*	*false*	*false*
ε	*false*	*true*	*true*	*true*
$\mathtt{b}R$	*false*	*false*	*false*	*false*
Nonterminal				
T	*false*	*false*	*true*	*true*
R	*false*	*true*	*true*	*true*

Fig. 2.14 Fixed-point iteration for calculation of *Nullable*

In a fixed-point calculation, we initially assume that *Nullable* is false for all nonterminals and use this as a basis for calculating *Nullable* for first the right-hand sides and then the nonterminals. We repeat recalculating these until there is no change between two iterations. Figure 2.14 shows the fixed-point iteration for the above equations. In each iteration, we first evaluate the formulae for the right-hand sides and then use the results of this to evaluate the nonterminals. The right-most column shows the final result.

We can calculate *FIRST* in a similar fashion to *Nullable*:

Algorithm 2.5

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

where $\mathtt{a}$ is a terminal, N is a nonterminal, α and β are sequences of grammar symbols and ε represents the empty sequence of grammar symbols.

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 $\alpha\beta \Rightarrow \beta \Rightarrow \cdots$, so anything in $\mathit{FIRST}(\beta)$ is also in $\mathit{FIRST}(\alpha\beta)$.

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:

Right-hand side	Initialisation	Iteration 1	Iteration 2	Iteration 3
R	$\emptyset$	$\emptyset$	{b}	{b}
aTc	$\emptyset$	{a}	{a}	{a}
ε	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
bR	$\emptyset$	{b}	{b}	{b}
Nonterminal				
T	$\emptyset$	{a}	{a, b}	{a, b}
R	$\emptyset$	{b}	{b}	{b}

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

$$
\begin{aligned}
\mathit{FIRST}(T) &= \mathit{FIRST}(R) \cup \mathit{FIRST}(\texttt{a}T\texttt{c}) \\
\mathit{FIRST}(R) &= \mathit{FIRST}(\varepsilon) \cup \mathit{FIRST}(\texttt{b}R)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{FIRST}(R) &= \mathit{FIRST}(R) \\
\mathit{FIRST}(\texttt{a}T\texttt{c}) &= \mathit{FIRST}(\texttt{a}) \\
\mathit{FIRST}(\varepsilon) &= \emptyset \\
\mathit{FIRST}(\texttt{b}R) &= \mathit{FIRST}(\texttt{b})
\end{aligned}
$$

The fixed-point iteration is shown in Fig. 2.15.

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 go through the motions of solving the equations. When working by hand, it is often useful to simplify the equations before the fixed-point iteration, e.g., reduce $\mathit{FIRST}(\texttt{a}T\texttt{c})$ to {a}.

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

2.8 Predictive Parsing Revisited

We are now ready to construct predictive parsers for a wider class of grammars: If the right-hand sides of the productions for a nonterminal have disjoint *FIRST* sets, we can use the next input symbol to choose among the productions.

In Sect. 2.6, we picked the empty production (if any) on any symbol that was not in the *FIRST* sets of the non-empty productions for the same nonterminal. We can extend this, so we in case of no matching *FIRST* sets can select a production if it is *Nullable*. The idea is that a *Nullable* production can derive the empty string, so the input symbol need not be read by the production itself.

But if there are several *Nullable* productions, we have no way of choosing between them. Hence, we do not allow more than one production for a nonterminal to be *Nullable*.

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. The syntax analysis will in this case 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 some 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

$$T \to \mathtt{a}T\mathtt{b}$$

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 erroneously.

We have so far simply chosen a nullable production if and only if no other choice is possible. This is, however, not always the right thing to do, so we must change the rule to say that we choose a production $N \to \alpha$ on symbol c if one of the two conditions below are satisfied:

1) $c \in \mathit{FIRST}(\alpha)$.
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 it without reading any input, so it seems like a nullable production could be a valid choice regardless of the next input symbol. Not only would this give multiple valid choices of production whenever there are both nullable and non-nullable productions for the same nonterminal, but it is also not always correct to choose the nullable production: Predictive parsing makes a leftmost derivation so we always rewrite the leftmost nonterminal N in the current sequence of grammar symbols. So whatever input is not matched by N must be matched by the sequence of grammar symbols that occurs after N in the current sequence. If this is not possible, we have made a bad choice when deriving N. In particular, if N derives to the empty sequence, the next input symbol c should begin the derivation of the sequence that follows N. So at the very least, the sequence of symbols that follow N should have a derivation that begins with c. If we (using a different derivation order) derive the symbols after N before deriving N, we should, hence, see the sequence Nc during the derivation. If we don't, we can not rewrite

N to the empty sequence without later getting stuck when rewriting the remaining sequence. Since the derivation order doesn't change the syntax tree, we can see that if the alternative derivation order gets stuck, so will the leftmost derivation order. Hence, we can only rewrite N to the empty sequence if the next input symbol c can occur in combination with N in a legal derivation. We will in the next section see how we can determine this.

Note that a nullable production $N \to \alpha$ can validly be selected if $c \in \mathit{FIRST}(\alpha)$.

Even with the 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 $R\texttt{b}$ can occur in a derivation) and all ambiguous grammars that are not detected as being ambiguous by the original method where we only choose nullable productions if no other choice is possible.

2.9 *FOLLOW*

To determine when we can select nullable productions during predictive parsing, we introduce *FOLLOW* sets for nonterminals.

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

In other words, a terminal c is in $\mathit{FOLLOW}(N)$ if c may follow N at some point in a derivation. Unlike $\mathit{FIRST}(N)$, 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 want to detect if $S \Rightarrow \alpha N$, i.e., if there are derivations where N can be followed by the end of input. It turns out to be easy to do this by adding an extra production to the grammar:

$$S' \to S\$$$

where S' is a new nonterminal that replaces S as start symbol and \$ is a new terminal symbol representing the end of input. Hence, in the new grammar, \$ will be in $\mathit{FOLLOW}(N)$ 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 least sets that obey the constraints.

A production

$$M \to \alpha N \beta$$

generates the constraint $\mathit{FIRST}(\beta) \subseteq \mathit{FOLLOW}(N)$, since β, obviously, can follow N. Furthermore, if $\mathit{Nullable}(\beta)$ the production also generates the constraint $\mathit{FOLLOW}(M) \subseteq \mathit{FOLLOW}(N)$ (note the direction of the inclusion). The reason

is that, if a symbol c is in $FOLLOW(M)$, then there (by definition) is a derivation $S' \Rightarrow \gamma M c \delta$. But since $M \rightarrow \alpha N \beta$ and β is nullable, we can continue this by $\gamma M c \delta \Rightarrow \gamma \alpha N c \delta$, so c is also in $FOLLOW(N)$.

If a right-hand side contains several occurrences of nonterminals, we add constraints for all occurrences, i.e., splitting the right-hand side into different αs, Ns and βs. 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 constraints in the following fashion:

We start by assuming empty *FOLLOW* sets for all nonterminals. We then handle the constraints of the form $FIRST(\beta) \subseteq FOLLOW(N)$: We compute $FIRST(\beta)$ and add this to $FOLLOW(N)$. Then, 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.

The steps taken to calculate the follow sets of a grammar are, hence:

1. Add 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 each nonterminal N, locate all occurrences of N on the right-hand sides of productions. For each occurrence do the following:
   2.1. Let β be the rest of the right-hand side after the occurrence of N. Note that β may be empty. In other words, the production is of the form $M \rightarrow \alpha N \beta$, where M is a nonterminal (possibly equal to N) and α and β are (possibly empty) sequences of grammar symbols. Note that if a right-hand-side contains several occurrences of N, we make a split for each occurrence.
   2.2. Let $m = FIRST(\beta)$. Add the constraint $m \subseteq FOLLOW(N)$ to the set of constraints. If β is empty, you can omit this constraint, as it does not add anything.
   2.3. If $Nullable(\beta)$, find the nonterminal M at the left-hand side of the production and add the constraint $FOLLOW(M) \subseteq FOLLOW(N)$. If $M = N$, you can omit the constraint, as it does not add anything. Note that if β is empty, $Nullable(\beta)$ is true.
3. Solve the constraints using the following steps:
   3.1. Start with empty sets for $FOLLOW(N)$ for all nonterminals N (not including S').
   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
$T' \to T\$$	$\{\$\} \subseteq FOLLOW(T)$
$T \to R$	$FOLLOW(T) \subseteq FOLLOW(R)$
$T \to \mathtt{a}T\mathtt{c}$	$\{\mathtt{c}\} \subseteq FOLLOW(T)$
$R \to$	
$R \to R\mathtt{b}R$	$\{\mathtt{b}\} \subseteq FOLLOW(R),\ FOLLOW(R) \subseteq FOLLOW(R)$

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:

$$FOLLOW(T) \supseteq \{\$, \mathtt{c}\}$$
$$FOLLOW(R) \supseteq \{\mathtt{b}\}$$

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

$$FOLLOW(T) \supseteq \{\$, \mathtt{c}\}$$
$$FOLLOW(R) \supseteq \{\$, \mathtt{c}, \mathtt{b}\}$$

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

$$FOLLOW(T) = \{\$, \mathtt{c}\}$$
$$FOLLOW(R) = \{\$, \mathtt{c}, \mathtt{b}\}$$

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., $\{\$, \mathtt{c}, \mathtt{b}\}$ and choose the non-empty production on the symbols in $FIRST(R\mathtt{b}R)$, i.e., $\{\mathtt{b}\}$. 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 as and bs that are not of the form ww where w is any string of as and bs. In other words, the strings can not consist of two identical halves.

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

The idea is that if the string does not consist of two identical halves, there must be a point in the first string that has an `a` where the equivalent point in the second string has a `b` or vice-versa. The grammar states that one of these is the case.

We first 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* for each nonterminal and right-hand side:

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

$$
\begin{aligned}
\mathit{FIRST}(A\ B) &= \mathit{FIRST}(A) \\
\mathit{FIRST}(B\ A) &= \mathit{FIRST}(B) \\
\mathit{FIRST}(\texttt{a}) &= \{\texttt{a}\} \\
\mathit{FIRST}(C\ A\ C) &= \mathit{FIRST}(C) \\
\mathit{FIRST}(\texttt{b}) &= \{\texttt{b}\} \\
\mathit{FIRST}(C\ B\ C) &= \mathit{FIRST}(C)
\end{aligned}
$$

which we solve by fixed-point iteration. We initially set the *FIRST* sets for the nonterminals to the empty sets, calculate the *FIRST* sets for right-hand sides and then nonterminals, repeating the last two steps until no changes occur:

RHS	Iteration 1	Iteration 2	Iteration 3
$A\ B$	$\emptyset$	{a}	{a, b}
$B\ A$	$\emptyset$	{b}	{a, b}
a	{a}	{a}	{a}
$C\ A\ C$	$\emptyset$	{a, b}	{a, b}
b	{b}	{b}	{b}
$C\ B\ C$	$\emptyset$	{a, b}	{a, b}
Nonterminal	**Iteration 1**	**Iteration 2**	**Iteration 3**
N	$\emptyset$	{a, b}	{a, b}
A	{a}	{a, b}	{a, b}
B	{b}	{a, b}	{a, b}
C	{a, b}	{a, b}	{a, b}

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

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

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

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

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

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

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

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

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 $N \to \alpha$ on input symbol c if

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

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. A grammar that can be parsed 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 structure of a program, and a table-based approach that encodes the decision process in a table.

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. 2.16 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 $T' \to T\$$ 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 next input symbol anyway, so we can report an error if it is not in $\mathit{FIRST}(T')$. This is shown in the function `parseT'`.

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

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

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

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

function parseR() =
  if next = 'c' or next = '$' then
    (* do nothing *)
  else if next = 'b' then
    match('b') ; parseR()
  else reportError()
```

Fig. 2.16 Recursive descent parser for Grammar 2.9

The function `match` takes as argument a symbol, which it tests for equality with the next input symbol. If they are equal, the following symbol is read into the variable `next`. We assume `next` is initialised to the first input symbol before `parseT'` is called.

The program in Fig. 2.16 only checks if the input is valid. It can easily be extended to construct a syntax tree by letting the parse functions return the sub-trees for the parts of input that they parse.

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.

The table is cross-indexed by nonterminal and terminal and contains for each such pair the production (if any) that is chosen for that nonterminal when that terminal is the next input symbol. This decision is made just as for recursive descent parsing: The production $N \to \alpha$ is in the table at (N,`a`) if `a` is in $\mathit{FIRST}(\alpha)$ or if both $\mathit{Nullable}(\alpha)$ and `a` is in $\mathit{FOLLOW}(N)$.

For Grammar 2.9 we get the table shown in Fig. 2.17.

The program that uses this table is shown in Fig. 2.18. It uses a stack, which at any time (read from top to bottom) contains the part of the current derivation that has

Fig. 2.17 LL(1) table for Grammar 2.9

	a	b	c	$
T'	$T' \to T\$$	$T' \to T\$$		$T' \to T\$$
T	$T \to \mathtt{a}T\mathtt{c}$	$T \to R$	$T \to R$	$T \to R$
R		$R \to \mathtt{b}R$	$R \to$	$R \to$

```
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),next) = empty then
    reportError
  else
    rhs := rightHandSide(table(top(stack),next)) ;
    pop(stack) ;
    pushList(rhs,stack)
```

Fig. 2.18 Program for table-driven LL(1) parsing

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 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 of these will be at the top of the stack.

As an example, Fig. 2.19 shows the input and stack at each step during parsing of the string `aabbbcc$` using the table in Fig. 2.17. The top of the stack is to the left.

The program in Fig. 2.18, like the one in Fig. 2.16, only checks if the input is valid. It, too, can be extended to build a syntax tree. This can be done by letting each nonterminal on the stack point to its node in the partially built syntax tree. When the nonterminal is replaced by one of its right-hand sides, nodes for the symbols on the right-hand side are added as children to the node.

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

Fig. 2.19 Input and stack during 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$
$	$

there are also unambiguous grammars that cause conflicts. An example of this is the unambiguous expression grammar (Grammar 2.11). 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 the grammar used to describe it is not.

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 grammars can be rewritten to allow LL(1) parsing. In these cases stronger parsing techniques must be used.

2.12.1 Eliminating Left-Recursion

As mentioned above, the unambiguous expression grammar (Grammar 2.11) is not LL(1). The reason is that all productions in *Exp* and *Exp*2 have the same *FIRST*

sets. Overlap like this will always happen when there are 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 direct left-recursion.

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

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

where the β_i do not start with N, we observe that this generates all sequences that start with one of the β_i and continues with any number (including 0) of the α_j. In other words, the grammar is equivalent to the regular expression $(\beta_1 \,|\, \ldots \,|\, \beta_n)(\alpha_1 \,|\, \ldots \,|\, \alpha_m)^*$.

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:

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

where N_* is a new nonterminal that generates a sequence of αs.

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, the α_j can not start with this, so the last m productions can't have direct left-recursion either.

There may, however, still be *indirect* left-recursion if any of the α_j are nullable or the β_i can derive something starting with N. 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 2.11. This has left recursion in both *Exp* and *Exp*2, so we apply the trans-

Grammar 2.20 Removing left-recursion from Grammar 2.11

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

formation to both of these to obtain Grammar 2.20. The resulting Grammar 2.20 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 is no *indirect left-recursion*. Indirect left-recursion can have several faces:

1. There are mutually left-recursive productions

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

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. 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. We will not go into this here, as in practice most cases of left-recursion are direct left-recursion. Details can be found in [2].

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 Gram-

Grammar 2.21
Left-factorised grammar for conditionals

Stat → **id** := *Exp*
Stat → `if` *Exp* `then` *Stat Elsepart*

Elsepart → `else` *Stat*
Elsepart →

mar 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 2.21. 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 2.13) can not easily be rewritten to a form suitable for LL(1), so in practice Grammar 2.21 is used anyway and the conflict is handled by choosing the non-empty production for *Elsepart* whenever the symbol `else` is encountered, as this gives the desired behaviour of letting an `else` match the nearest `if`. We can achieve this by removing the empty production from the table entry for *Elsepart* / `else`, so only the non-empty production *Elsepart* → `else` *Stat* remains.

Very few LL(1) conflicts caused by ambiguity can be removed in this way, however, without also changing the language recognized by the grammar. For example, operator precedence ambiguity can not be resolved by deleting conflicting entries in the LL(1) table.

2.12.3 Construction of LL(1) Parsers Summarized

1. Eliminate ambiguity
2. Eliminate left-recursion
3. Perform left factorisation where required
4. Add an extra start production $S' \rightarrow S\$$ to the grammar.

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

5. Calculate *FIRST* for every production and *FOLLOW* for every nonterminal.
6. For nonterminal N and input symbol c, choose production $N \to \alpha$ when:

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

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

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 accept 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 declaration of operator precedences for resolving ambiguity, instead of requiring the grammars themselves to be unambiguous.

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 table-driven bottom-up parsers and use two kinds of "actions" involving the input stream and a stack:

shift: A 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 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.

LR parsers are also called *shift-reduce parsers*. 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 achieve this, we construct a DFA. Conceptually, this DFA reads the contents of the stack, starting from the bottom. If the DFA is in an accepting state when it reaches the top of the stack, the correct action is reduction by a production that is determined by the next input symbol and a mark on the accepting

DFA state. If the DFA is not in an accepting state when it reaches the stack top, the correct action is a shift on one of the symbols for which there is an outgoing edge from the DFA state. Hence, at every step, the DFA reads the stack from bottom to top and the action is determined by looking at the DFA state and the next input symbol.

Letting the DFA read the entire stack at every action is, however, not very efficient so, instead, we store with each stack element the state of the DFA when it reads this element. This way, we do not need to start from the bottom of the stack, but can start from the current stack top starting the DFA in the stored state instead of in its initial state.

When the DFA has indicated a shift, the course of action is easy: We get the state from the top of the stack and follow the transition marked with the next input symbol to find the next DFA state, and we store both the symbol and the new state on the stack.

If the DFA indicated a reduce, we pop the symbols corresponding to the right-hand side of the production off the stack. We then read the DFA state from the new stack top. This DFA state should have a transition on the nonterminal that is the left-hand side of the production, so we store both the nonterminal and the state at the end of the transition on the stack.

With these optimisations, the DFA only has to inspect a terminal or nonterminal once, at the time it is pushed on the stack. At all other times, it just needs to read the DFA state that is stored on the top of the stack. We, actually, do not need to store the current input symbol or nonterminal once we have made a transition on it, as no future transitions will depend on it—the stored DFA state is enough. So we can let each stack element just contain the DFA state instead of both the symbol and the state. We still use the DFA to determine the next action, but it now only needs to look at the current state (stored at the top of the stack) and the next input symbol (at a shift action) or nonterminal (at a reduce action).

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:	Read next input symbol and push state n on the stack.
go n:	Push state n on the stack.
reduce p:	Reduce with the production numbered p.
accept:	Parsing has completed successfully.
error:	A syntax error has been detected.

Note that the current state is always found at the top of the stack. *Shift* and *reduce* actions are used when a state is cross-indexed with a terminal symbol. *Go* actions are used when a state is cross-indexed with a nonterminal. A *Go* action can occur only immediately after a reduce, but we can not in the table combine the *go* actions with the *reduce* actions, as the destination state of a *go* action depends on the state at the top of the stack *after* the right-hand side of the reduced production is popped off: A *reduce* in the current state is immediately followed by a *go* in the state that is found when the stack is popped.

Fig. 2.22 SLR table for Grammar 2.9

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

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

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

    accept:   terminate with success

    error:    reportError
endloop
```

Fig. 2.23 Algorithm for SLR parsing

An example SLR table is shown in Fig. 2.22. 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 algorithm for parsing a string using the table is shown in Fig. 2.23. 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: Each stack element holds (in addition to the state number) a portion of a syntax tree. When performing a *reduce* action, a new (partial) 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 and the new state are then pushed (as a single stack element).

Figure 2.24 shows an example of parsing the string `aabbbcc` using the table in Fig. 2.22. The sequences of numbers in the "stack" column represent the stack contents with the stack bottom shown to the left and the stack top to the right. At each step, we look at the next input symbol (at the left end of the string in the input

Fig. 2.24 Example SLR parsing

input	stack	action
aabbbcc$	0	s3
abbbcc$	03	s3
bbbcc$	033	s4
bbcc$	0334	s4
bcc$	03344	s4
cc$	033444	r3 ($R \rightarrow$) ; g6
cc$	0334446	r4 ($R \rightarrow$ bR) ; g6
cc$	033446	r4 ($R \rightarrow$ bR) ; g6
cc$	03346	r4 ($R \rightarrow$ bR) ; g2
cc$	0332	r1 ($T \rightarrow R$) ; g5
cc$	0335	s7
c$	03357	r2 ($T \rightarrow$ aTc) ; g5
c$	035	s7
$	0357	r2 ($T \rightarrow$ aTc) ; g1
$	01	accept

Grammar 2.25 Example grammar for SLR-table construction

0: $T' \rightarrow T$
1: $T \rightarrow R$
2: $T \rightarrow \mathtt{a}T\mathtt{c}$
3: $R \rightarrow$
4: $R \rightarrow \mathtt{b}R$

column) and the state at the top of the stack (at the right end of the sequence in the stack column). We look up the pair of input symbol and state in the table and find an action, which is shown in the action column. When the shown 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.

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 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 2.25.

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

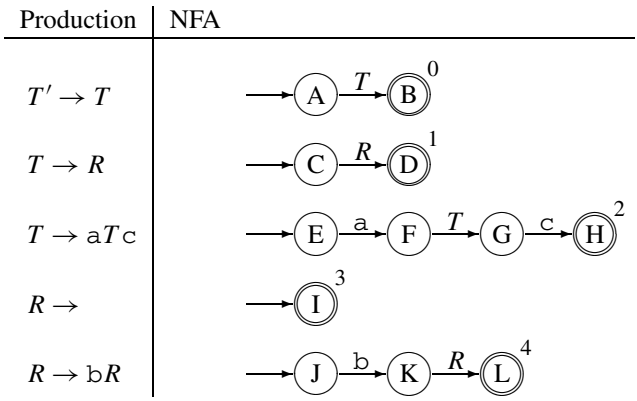


Fig. 2.26 NFAs for the productions in Grammar 2.25

is shown in Fig. 2.26. Note that we have used the optimised construction for ε (the empty production) as shown in Fig. 1.6.

The NFAs in Fig. 2.26 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, whereby some elements of the stack (corresponding to the right-hand side of a production) are replaced by a nonterminal (corresponding to the left-hand side of that production). However, before we can reduce on a nonterminal, the symbols that form the right-hand side must be on the stack. So we prepare for a later transition on a nonterminal by allowing transitions that, eventually, will leave the symbols forming right-hand side of the production on the stack, so we can reduce these to the nonterminal and then make a transition in this.

So, whenever a transition by a nonterminal is possible, we also allow transitions on the sequences of symbols on the right-hand sides of the productions for that nonterminal. We achieve this by adding epsilon transitions to the NFAs in Fig. 2.26: Whenever there is a transition from state s to state t on a nonterminal N, we add a epsilon transitions from s to the initial states of the NFAs for the productions with N on the left-hand side. We also 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. 2.27.

We must now convert this NFA into a DFA using the subset construction shown in Sect. 1.4. Instead of showing the resulting DFA graphically, we construct a table where transitions on terminals are shown as *shift* actions and transitions on nonterminals as *go* actions. This will make the table look similar to Fig. 2.22, except that no *reduce* or *accept* actions are present yet. Figure 2.28 shows the DFA constructed from the NFA in Fig. 2.27. The sets of NFA states that form each DFA state is shown in the second column of the table in Fig. 2.28. 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.

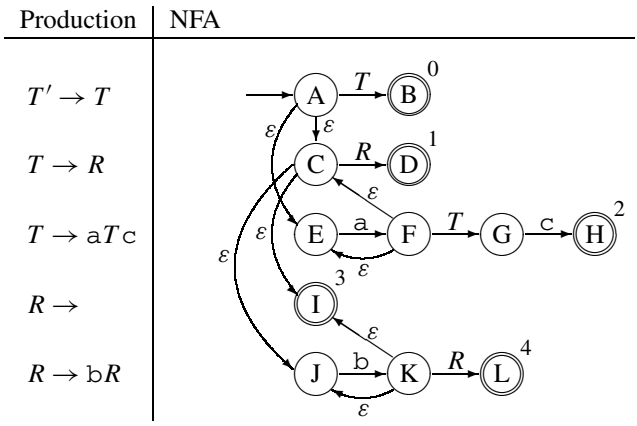


Fig. 2.27 NFAs for the productions in Grammar 2.25 with epsilon transitions added

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

Fig. 2.28 SLR DFA for Grammar 2.9

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:

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

We then 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 on production 0 (the extra start production that was added before constructing the NFA) is written as *accept*.

In Fig. 2.28, 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. We add this at the symbol $ ($FOLLOW(T')$). As noted above, this is written as *accept* (abbreviated to "a"). In

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. Convert the combined NFA to a DFA. Use the starting state of the NFA for the production added in step 1 as the starting state for the combined NFA.
5. 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.
6. Calculate *FOLLOW* for each nonterminal. For this purpose, we add one more start production: $S'' \rightarrow S'\$$.
7. When a DFA state contains an accepting NFA state marked with production number p, where the 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 production p is the production added in step 1, add an *accept* action instead of a *reduce p* action.

Fig. 2.29 Summary of SLR parse-table construction

the same way, we add reduce actions to state 3, 4, 6 and 7. The result is shown in Fig. 2.22.

Figure 2.29 summarises the SLR construction.

2.14.1 Conflicts in SLR Parse-Tables

When *reduce* actions are added to SLR parse-tables, we might add one to a place where there is already a *shift* action, or we may add *reduce* actions for several different productions to the same place. When either of this happens, 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 may occur in the same place.

Conflicts are often caused by ambiguous grammars, but (as is the case for LL-parsers) even some non-ambiguous grammars may 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. Alternatively, operator precedence declarations may be used to disambiguate an ambiguous grammar, as we shall see in Sect. 2.15.

But even unambiguous grammars may in some cases generate conflicts in SLR-tables. In some cases, it is still possible to rewrite the grammar to get around the problem, but in a few cases the language simply is not SLR. Rewriting an unambiguous grammar to eliminate conflicts is somewhat of an art. Investigation 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.

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 rules 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 a +. 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 rules give that + is left-associative, we prefer the first of these options and, hence, eliminate the shift-action from the table and keep the reduce-action.

2) The offending expressions here have the form `a*b+c`. Since the rules make multiplication bind stronger than addition, we, again, prefer reduction over shifting.
3) In expressions of the form `a+b*c`, the rules, again, make multiplication bind stronger, so we do 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 the rules is left-associative conflicts with itself. We, like 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. The operator associated with a reduce-action is the operator used in the production that is reduced.
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. 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.
c) If there are several operators with declared precedence in the production that is used in a reduce-action, the last of these is used to determine the precedence of the reduce-action.[4]

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`. If we, instead, reduced on the production *Stat* → `if` *Exp* `then` *Stat*, we would lose this association. Giving `else` a higher precedence than `then` or giving them the same precedence and making them right-associative will ensure that a shift is made on `else` when we need it.

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

Not all conflicts should be eliminated by precedence rules. If you blindly add precedence rules until no conflicts are reported, you risk disallowing also legal 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.

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

- It is simpler.
- In practice, LALR(1) handles only a few more grammars than SLR.
- 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.
- Understanding of SLR principles is sufficient to know how to handle a grammar rejected by a LALR(1) parser generator by adding precedence declarations or by rewriting the grammar.

In short, 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 Declarations and Actions

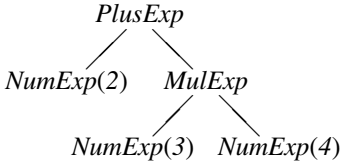
Each nonterminal and terminal is declared and associated with a data-type. For a terminal, the data-type is used to hold the values that are associated with the tokens that come from the lexer, *e.g.*, the values of numbers or names of identifiers. For a nonterminal, the type is used for the values that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

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 some kind of syntax representation during parsing and then later operate on this representation.

2.16.2 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, *abstract syntax* is commonly used.

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. 2.12 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 keep the subtrees that are implied by the choice of production, such as the subtree from Fig. 2.12 that holds the symbol +. Likewise, the sequence of nodes *Exp*, *Exp*2, *Exp*3, 2 at the left of Fig. 2.12 are combined to a single node *NumExp*(2) that includes both the choice of productions for *Exp*, *Exp*2 and *Exp*3 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, each 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.

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

$$E \rightarrow E + \textbf{num}$$
$$E \rightarrow \textbf{num}$$

gets transformed by left-recursion elimination into

$$E \rightarrow \textbf{num}\ E'$$
$$E' \rightarrow + \textbf{num}\ E'$$
$$E' \rightarrow$$

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' should 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' returns just a hole.
- In the first production for E', the $+$ and **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 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 **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.

$$\begin{array}{ll} E \to E + \textbf{num} & \{\ \texttt{PlusExp(\$1,NumExp(\$3))}\ \} \\ E \to \textbf{num} & \{\ \texttt{NumExp(\$1)}\ \} \end{array}$$

We now want to make actions for the transformed grammar that will produce the same abstract syntax trees as this 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:

$$\begin{array}{lll} E \to \textbf{num}\, E' & \{\ \texttt{\$2(NumExp(\$1))}\ \} \\ E' \to +\,\textbf{num}\, E' & \{\ \texttt{λx.\$3(PlusExp(x,NumExp(\$2)))}\ \} \\ E' \to & \{\ \texttt{λx.x}\ \} \end{array}$$

where $\lambda x.e$ is a nameless 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 Haskell as `\x -> e` and in Scheme as `(lambda (x) e)`.

The imperative version of the actions in the original grammar is

$$\begin{array}{ll} E \to E + \textbf{num} & \{\ \texttt{\$0 = PlusExp(\$1,NumExp(\$3))}\ \} \\ E \to \textbf{num} & \{\ \texttt{\$0 = NumExp(\$1)}\ \} \end{array}$$

In this setting, `NumExp` and `PlusExp` are not constructors but functions that allocate and build node and return pointers to these. Unnamed functions of the kind used in the above solution for functional languages can not be built in most imperative languages, 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

```
E  → num E'   { $2->hole = NumExp($1);
                $0 = $2.tree }
E' → + num E' { $0.hole = makeHole();
                $3->hole = PlusExp($0.hole,NumExp($2));
                $0.tree = $3.tree }
E' →          { $0.hole = makeHole();
                $0.tree = $0.hole }
```

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.

2.16.3 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 conflicts, it will tell in which state in the table these occur. 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 the NFA corresponds to a position in a production in the grammar. If we, for example, look at the NFA states in Fig. 2.26, these would be written as shown in Fig. 2.30. Note that a '.' is used to indicate the position of the state in the production. State 4 of the table in Fig. 2.28 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.

If all efforts to get a grammar through a parser generator fails, a practical solution may be to change the grammar so it 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).

Some languages allow programs to declare precedence and associativity for user-defined operators. This can make it difficult to handle precedence during parsing, as

Fig. 2.30 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 .`

the precedences are not known when the parser is generated. A typical solution is to parse all operators using the same precedence and then restructure the syntax tree afterwards. See Exercise 2.20 for other approaches.

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.

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. 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 $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ by a context-free grammar, the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ 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 repeating a string twice 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, the language consisting of all subsequences of a context-free language is actually regular. However, context-free languages are *not* closed under intersection or complement. For example, the languages $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m \mid m, n \geq 0\}$ and $\{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n \mid m, n \geq 0\}$ are both context-free while their intersection $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ 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 [3], 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 these represent all languages with decidable membership.

Context-free grammars are too weak to describe natural languages, but were adopted for defining the Algol60 programming language [4]. 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, but 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 a few 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].

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 sequences

```
  ε      (i.e., the empty string)
  ()
 (())
 ()()
((()()))
```

but none of the following

```
(
)
)(
(()
()())
```

Exercise 2.4 Write grammars for each of the following languages:

a) All sequences of `a`s and `b`s that contain the same number of `a`s and `b`s (in any order).
b) All sequences of `a`s and `b`s that contain strictly more `a`s than `b`s.
c) All sequences of `a`s and `b`s that contain a different number of `a`s and `b`s.
d) All sequences of `a`s and `b`s 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

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

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 the syntax trees using the new grammars for the string you used to prove the original grammar ambiguous.

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 for $2 : 3 < 4 + 5 : 6 * 7$ using the unambiguous grammar.

Exercise 2.8 Extend Grammar 2.13 with the productions

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

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

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

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 2.20.

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:

$$E \rightarrow E\ E\ +$$
$$E \rightarrow E\ E\ *$$
$$E \rightarrow \textbf{num}$$

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 2.11 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. 2.26) for the productions in Grammar 2.11 (after extending it as shown in Sect. 2.14) and add epsilon-transitions as in Fig. 2.27. Convert the combined NFA into an SLR DFA like the one in Fig. 2.28. 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. 2.26) for the productions in Grammar 2.2 (after extending it as shown in Sect. 2.14) and add epsilon-transitions as in Fig. 2.27. Convert the combined NFA into an SLR DFA like the one in Fig. 2.28. 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

$$T \rightarrow T\ \texttt{->}\ T$$
$$T \rightarrow T\ *\ T$$
$$T \rightarrow \textbf{int}$$

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. 2.16.3 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 a SLR parse table for the grammar.
d) Can the conflicts in the table be eliminated?

Exercise 2.22 The following ambiguous grammar describes boolean expressions:

$$
\begin{aligned}
B &\rightarrow \textbf{true} \\
B &\rightarrow \textbf{false} \\
B &\rightarrow B\ \vee\ B \\
B &\rightarrow B\ \wedge\ B \\
B &\rightarrow \neg\ B \\
B &\rightarrow (\ B\ )
\end{aligned}
$$

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 specification and implementations of programming languages. In: Maluszyński, J., Wirsing, M. (eds.) Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming. LNCS, vol. 528, pp. 183–194. Springer, Berlin (1991). citeseer.ist.psu.edu/article/aasa91precedences.html
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers; Principles, Techniques and Tools. Addison-Wesley, Reading (2007)
3. Chomsky, N.: Three models for the description of language. IRE Trans. Inf. Theory **2**(3), 113–124 (1956)
4. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language Algol 60. Commun. ACM **6**(1), 1–17 (1963)
5. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation, 2nd edn. Addison-Wesley, Reading (2001)
6. Jensen, K., Wirth, N.: Pascal User Manual and Report, 2nd edn. Springer, Berlin (1975)
7. Kerninghan, B.W., Ritchie, D.M.: The C Programming Language. Prentice-Hall, New York (1978)
8. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages, 1st edn. Pragmatic Programmers. The Pragmatic Bookshelf, Lewisville (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* objects such as variables, functions and types. Each such named object will have a *declaration*, where the name is defined as a synonym for the object. This is called *binding*. Each name will also have a number of *uses*, where the name is used as a reference to the object 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

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_3, © Springer-Verlag London Limited 2011

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 += (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 objects 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 information. 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 a piece of information. 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 information the name is bound to. 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 is copied when a symbol table is updated, most 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 how more advanced approaches can overcome some of the efficiency problems with the simple approaches.

3.1.2 Simple Persistent Symbol Tables

In functional languages like SML, 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 information, 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/information 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 information 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 information and this special value, i.e., a sum-type.

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 information `i` 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 information `i`. 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,i,t)
  = fn n1 => if n1=n then i 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/information pair) is pushed on top of the stack.

lookup: The stack is searched top-to-bottom until a matching name is found. The information 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, 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. In other languages (e.g. C or SML) the context can not (easily) distinguish variables from functions. Hence, declaring a local variable might hide a function 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 the name and the name-space indicator of the symbol that is looked up with the name and the 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 objects share name spaces: Variables, functions, procedures and types. If there are more kinds of named objects (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.

References

1. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (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

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_4,

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 this.

4.1 The Structure of an Interpreter

An intepreter 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 may be the value of an expression or an updated symbol table.

The functions can be implemented in any language that we already have an implementation of. 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 the interpreter functions in a notation reminiscent of a high-level programming language and assume an implementation 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. 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 and integers, but addition only on integers.

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 is the output of the program.

Program → *Funs*

Funs → *Fun*
Funs → *Fun Funs*

Fun → *TypeId* (*TypeIds*) = *Exp*

TypeId → `int` **id**
TypeId → `bool` **id**

TypeIds → *TypeId*
TypeIds → *TypeId* , *TypeIds*

Exp → **num**
Exp → **id**
Exp → *Exp* + *Exp*
Exp → *Exp* = *Exp*
Exp → `if` *Exp* `then` *Exp* `else` *Exp*
Exp → **id** (*Exps*)
Exp → `let` **id** = *Exp* `in` *Exp*

Exps → *Exp*
Exps → *Exp* , *Exps*

Grammar 4.1 Example language for interpretation

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 have 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. 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 return to this later.

The main part of $Eval_{Exp}$ is a case-expression 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 box to the right of a pattern shows the actions needed to evaluate the expression. 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 call $Call_{Fun}$ to find the result of the call.

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
num	*getvalue*(**num**)
id	$v = lookup(vtable, getname(\textbf{id}))$ *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 *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 *are both integers or both booleans* *then if* $v_1 = v_2$ *then* ***true*** *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* $v_1 =$ ***true*** *then* $Eval_{Exp}(Exp_2, vtable, ftable)$ *else* $Eval_{Exp}(Exp_3, vtable, ftable)$ *else* ***error***()
id (*Exps*)	$def = lookup(ftable, getname(\textbf{id}))$ *if* def $=$ *unbound* *then* ***error***() *else* $args = Eval_{Exps}(Exps, vtable, ftable)$ $Call_{Fun}(def, args, ftable)$
`let` **id** = Exp_1 `in` Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\textbf{id}), v_1)$ $Eval_{Exp}(Exp_2, vtable', ftable)$

$Eval_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`	
Exp	$[Eval_{Exp}(Exp, vtable, ftable)]$
Exp , $Exps$	$Eval_{Exp}(Exp, vtable, ftable)$ :: $Eval_{Exps}(Exps, vtable, ftable)$

Fig. 4.2 Evaluating expressions

- 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.

$Eval_{Exps}$ builds a list of the values of the expressions in the expression list. The notation is taken from SML: A list is written in square brackets with commas between the elements. The operator :: adds an element to the front of a 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, it must be checked 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 Get_{TypeId} just returns a pair of the declared name and type, and $Bind_{TypeIds}$ checks the declared type against a value and builds a symbol table that binds the name to the value if they match (and report an error if they do not). $Binds_{TypeIds}$ 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 $Bind_{TypeIds}$ 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 $Build_{ftable}$ 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.

Suggested exercises: 4.5.

4.4 Advantages and Disadvantages of Interpretation

Interpretation is often the simplest way of executing a program once you have its abstract syntax tree. However, it is also a relatively slow way to do so. When we

$Call_{Fun}(Fun,\ args,\ ftable) =$ `case` Fun `of`	
$TypeId\ (\ TypeIds\ )\ =\ Exp$	$(f, t_0) = Get_{TypeId}(TypeId)$ $vtable = Bind_{TypeIds}(TypeIds,\ args)$ $v_1 = Eval_{Exp}(Exp, vtable, ftable)$ *if* v_1 is of type t_0 *then* v_1 *else* **error**()

$Get_{TypeId}(TypeId) =$ `case` $TypeId$ `of`	
`int` **id**	(*getname*(**id**), `int`)
`bool` **id**	(*getname*(**id**), `bool`)

$Bind_{TypeIds}(TypeIds,\ args) =$ `case` $(TypeIds,\ args)$ `of`	
$(TypeId,$ $[v])$	$(x, t) = Get_{TypeId}(TypeId)$ *if* v is of type t *then* $bind(emptytable, x, v)$ *else* **error**()
$(TypeId\ ,\ TypeIds,$ $(v :: vs))$	$(x, t) = Get_{TypeId}(TypeId)$ $vtable = Bind_{TypeIds}(TypeIds,\ vs)$ *if* $lookup(vtable, x) = unbound$ *and* v is of type t *then* $bind(vtable, x, v)$ *else* **error**()
_	**error**()

Fig. 4.3 Evaluating a function call

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 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 if it is O.K. to do it than on actually doing it.

To get faster execution, we use the observation that a program that only executes each part of the program 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 once for each 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, 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 for potential mismatches between the types of values and the requirements of operations. It does so for the whole program regardless of whether all parts will actually be executed, so it may report errors even

$Run_{Program}(Program,\ input) =$ `case` $Program$ `of`	
$Funs$	$ftable = Build_{ftable}(Funs)$ $def = lookup(ftable,$ `main`$)$ *if* def $= unbound$ *then* ***error***() *else* $Call_{Fun}(def,\ [input],\ ftable)$

$Build_{ftable}(Funs) =$ `case` $Funs$ `of`	
Fun	$f = Get_{fname}(Fun)$ $bind(emptytable,\ f, Fun)$
$Fun\ Funs$	$f = Get_{fname}(Fun)$ $ftable = Build_{ftable}(Funs)$ *if* $lookup(ftable,\ f) = unbound$ *then* $bind(ftable,\ f, Fun)$ *else* **error**()

$Get_{fname}(Fun) =$ `case` Fun `of`	
$TypeId$ ($TypeIds$) = Exp	$(f, t_0) = Get_{TypeId}(TypeId)$ f

Fig. 4.4 Interpreting a program

if an interpretation of the program would finish without errors. So static type checking puts extra limitations on programs but reduces the time needed to check 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. It 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.

Compilation gets rid of the abstract syntax tree of the source program by translating it into a target program (in a language that we already have an implementation of) that only performs the operations in the source program. Usually, the target language is a low-level language such as machine code. Like static checking, compilation must 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 there are compilers for languages with dynamic (run-time) type checking and 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 the first time it is called, hence reducing the delay from editing a program to its execution but at the cost of adding smaller delays during execution.

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:

$$\begin{aligned} \mathit{Exp} &\rightarrow \texttt{not}\ \mathit{Exp} \\ \mathit{Exp} &\rightarrow \mathit{Exp}\ \texttt{and}\ \mathit{Exp} \end{aligned}$$

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

$$\begin{aligned} \mathit{Exp} &\rightarrow \textbf{floatconst} \\ \mathit{TypeId} &\rightarrow \texttt{float}\ \textbf{id} \end{aligned}$$

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 floating point numbers for equality.

(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 the same `let`-bound variable 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, so we assume symbol tables are fully persistent. We add the following productions to Grammar 4.1:

$$\begin{aligned} \mathit{TypeId} &\rightarrow \texttt{fun}\ \textbf{id} \\ \\ \mathit{Exp} &\rightarrow \mathit{Exp}\ \mathit{Exp} \\ \mathit{Exp} &\rightarrow \texttt{fn}\ \textbf{id}\ \texttt{=>}\ \mathit{Exp} \end{aligned}$$

The notation is taken from Standard ML.

Evaluating `fn` x `=>` e in an environment *vtable* produces a functional value f. 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. Represent a lexical closures as a pair of (the abstract syntax of) an expression and an environment.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. MIT Press, Cambridge (1996). Also downloadable from http://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, the translator may exploit or depend on type information, which makes it natural to combine calculation of types with the actual translation. 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 Types

We have already discussed the difference between static and dynamic typings, 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 a type that the

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_5, © Springer-Verlag London Limited 2011

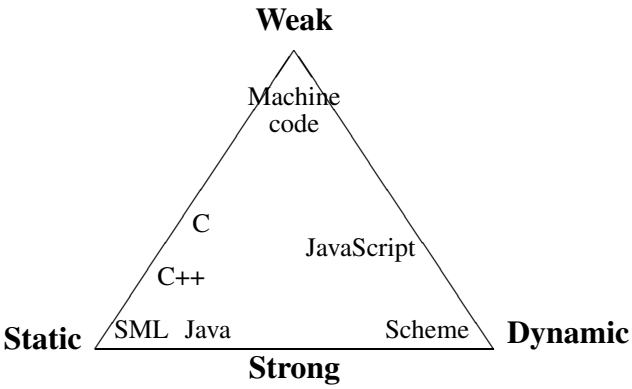


Fig. 5.1 The design space of types

operation is defined for, 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 just performed with no 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. Weakly typed languages are mostly used for system programming, where you need to manipulate move, copy, encrypt or compress data without regard to what the data represents.

Many languages combine both strong and weak typing or both static and dynamic typing: Some types are checked before execution and other 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. For example, you can store an integer in a union-typed variable and read it back as a pointer or floating-point number. Another 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 kind of weak typing, as the multiplication operation is applied to arguments (strings) where multiplication 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 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 probably makes no sense.

Figure 5.1 shows a diagram of the design space of static vs. dynamic and weak vs. strong typing, 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 vs. strong spectrum, the distinction between static and dynamic is meaningless.

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 same syntax tree as 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 syntactical 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 in 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 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, `(int,bool)` $\rightarrow$ `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 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 function use a notation similar to the concrete syntax for pattern-matching purposes. 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. 6.

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.

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.

[1] Unlike in Chap. 4, where the **error** function stops execution.

$Check_{Exp}(Exp, vtable, ftable) =$ `case` *Exp* `of`	
num	`int`
id	$t = lookup(vtable, getname(\textbf{id}))$ *if* $t =$ *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* $t_1 =$ `int` *and* $t_2 =$ `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* $t_1 =$ `bool` *and* $t_2 = t_3$ *then* t_2 *else* ***error***(); t_2
id (*Exps*)	$t = lookup(ftable, getname(\textbf{id}))$ *if* $t =$ *unbound* *then* ***error***(); `int` *else* $((t_1, \ldots, t_n) \rightarrow t_0) = t$ $[t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ *if* $m = n$ *and* $t_1 = t'_1, \ldots, t_n = t'_n$ *then* t_0 *else* ***error***(); t_0
`let` **id** $= Exp_1$ `in` Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\textbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$

$Check_{Exps}(Exps, vtable, ftable) =$ `case` *Exps* `of`	
Exp	$[Check_{Exp}(Exp, vtable, ftable)]$
Exp , *Exps*	$Check_{Exp}(Exp, vtable, ftable)$ :: $Check_{Exps}(Exps, vtable, ftable)$

Fig. 5.2 Type checking of expressions

- 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 num-

ber 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 *ftable*, 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 parts 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 commas between the elements. 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. 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 internal 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, and $Check_{TypeIds}$ builds a symbol table from such pairs. $Check_{TypeIds}$ also checks if all parameters have different names. *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 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

$Check_{Fun}(Fun, ftable) =$ `case` Fun `of`	
$TypeId\ (TypeIds)\ =\ Exp$	$(f, t_0) = Get_{TypeId}(TypeId)$ $vtable = Check_{TypeIds}(TypeIds)$ $t_1 = Check_{Exp}(Exp, vtable, ftable)$ $if\ \ t_0 \neq t_1$ $then$ **error**()

$Get_{TypeId}(TypeId) =$ `case` $TypeId$ `of`	
`int` **id**	($getname$(**id**), `int`)
`bool` **id**	($getname$(**id**), `bool`)

$Check_{TypeIds}(TypeIds) =$ `case` $TypeIds$ `of`	
$TypeId$	$(x, t) = Get_{TypeId}(TypeId)$ $bind(emptytable, x, t)$
$TypeId\ ,\ TypeIds$	$(x, t) = Get_{TypeId}(TypeId)$ $vtable = Check_{TypeIds}(TypeIds)$ $if\ \ lookup(vtable, x) = unbound$ $then\ \ bind(vtable, x, t)$ $else$ **error**(); $vtable$

Fig. 5.3 Type checking a function declaration

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 *Funs* are Get_{Funs}, which builds the symbol table and checks for duplicate definitions, and $Check_{Funs}$, 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.

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 check if a variable is potentially used before it is given a value, or

$Check_{Program}(Program) = \texttt{case}\ Program\ \texttt{of}$	
Funs	$ftable = Get_{Funs}(Funs)$ $Check_{Funs}(Funs, ftable)$ *if* $lookup(ftable, \texttt{main}) \neq (\texttt{int}) \rightarrow \texttt{int}$ *then* ***error***()

$Get_{Funs}(Funs) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$(f, t) = Get_{Fun}(Fun)$ $bind(emptytable, f, t)$
Fun Funs	$(f, t) = Get_{Fun}(Fun)$ $ftable = Get_{Funs}(Funs)$ *if* $lookup(ftable, f) = unbound$ *then* $bind(ftable, f, t)$ *else* **error**(); *ftable*

$Get_{Fun}(Fun) = \texttt{case}\ Fun\ \texttt{of}$	
TypeId (TypeIds) = Exp	$(f, t_0) = Get_{TypeId}(TypeId)$ $[t_1, \ldots, t_n] = Get_{Types}(TypeIds)$ $(f, (t_1, \ldots, t_n) \rightarrow t_0)$

$Get_{Types}(TypeIds) = \texttt{case}\ TypeIds\ \texttt{of}$	
TypeId	$(x, t) = Get_{TypeId}(TypeId)$ $[t]$
TypeId TypeIds	$(x_1, t_1) = Get_{TypeId}(TypeId)$ $[t_2, \ldots, t_n] = Get_{Types}(TypeIds)$ $[t_1, t_2, \ldots, t_n]$

$Check_{Funs}(Funs, ftable) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$Check_{Fun}(Fun, ftable)$
Fun Funs	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

Fig. 5.4 Type checking a program

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.

Data Structures A data structure may define a value with several components (e.g., a *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 declarations. Operations that build or take apart structured data need to be tested for correctness. If each operation on structured data has well-defined types for its arguments and a type for its result, this can be done in a way similar to how function calls are tested.

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 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 pick 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 any type, not just by a limited list of declared alternatives as is the case with overloading.

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. If there are inconsistent uses of a variable, a type error is reported.

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].

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. 5.2–5.4 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:

$$S \to \texttt{throw}\ \textbf{id}$$
$$S \to S\ \texttt{catch}\ \textbf{id} \Rightarrow S$$
$$S \to S\ \texttt{or}\ S$$
$$S \to \texttt{other}$$

A `throw` statement throws a named exception. This is caught by the nearest enclosing `catch` statement (i.e., where the `throw` statement is in the left sub-statement of the `catch` statement) using the same name, whereby the statement after the arrow in the `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 do not throw any exceptions.

We want the type checker to ensure that all possible exceptions are caught and that no `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 closures and implemented these in the interpreter.

Extend the type-checking functions in Figs. 5.2–5.4 to statically type check the same extensions.

Hint: Check a function definition when it is declared.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers; Principles, Techniques and Tools. Addison-Wesley, Reading (2007)
2. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978)

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 final 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 which 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 a language for which a compiler or interpreter already exists.

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

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_6, © Springer-Verlag London Limited 2011

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.

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 a lot slower than executing translated code directly. Nevertheless, the approach has seen some success, e.g., with Java.

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 for executing the intermediate code for Java.

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 Choosing an Intermediate Language

An intermediate language should, ideally, have the following properties:

- It should be easy to translate from a high-level language to the intermediate language. This should be the case for a wide range of different source languages.
- It should be easy to translate from 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 close to this. However, this may be hard to achieve for more than a small number of similar 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 machines 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 operation 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 operations.

The opposite approach is to let each intermediate-code operation be as small as possible. This means that each intermediate-code operation is typically translated into a single machine-code instruction or that several intermediate-code operations can be combined into one machine-code operation. 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 operations. By assigning cost to each machine-code operation, 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

In 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:

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*

Grammar 6.1 The intermediate language

- 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.
- 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.

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 is not difficult, though.

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

$$
\begin{aligned}
\mathit{Exp} &\rightarrow \textbf{num} \\
\mathit{Exp} &\rightarrow \textbf{id} \\
\mathit{Exp} &\rightarrow \textbf{unop}\ \mathit{Exp} \\
\mathit{Exp} &\rightarrow \mathit{Exp}\ \textbf{binop}\ \mathit{Exp} \\
\mathit{Exp} &\rightarrow \textbf{id}(\mathit{Exps}) \\
\mathit{Exps} &\rightarrow \mathit{Exp} \\
\mathit{Exps} &\rightarrow \mathit{Exp}\ ,\ \mathit{Exps}
\end{aligned}
$$

Grammar 6.2 A simple expression language

be done at compile-time, but some things need to be postponed until run-time, as they need 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 happens at which time, but the notation used in the translation functions makes this clear: Intermediate-language code 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 one. 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 this 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 decide where to 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 would be 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) "$\mathtt{x} = e$", 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.

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.

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 the *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*.

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
num	$v = getvalue(\textbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\textbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = transop(getopname(\textbf{unop}))$ $code_1 ++ [place := op\ place_1]$
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = transop(getopname(\textbf{binop}))$ $code_1 ++ code_2 ++ [place := place_1\ op\ place_2]$
id$(Exps)$	$(code_1, [a_1, \ldots, a_n])$ $= Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\textbf{id}))$ $code_1 ++ [place :=$ `CALL` $fname(a_1, \ldots, a_n)]$

$Trans_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`	
Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
Exp , $Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 ++ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Fig. 6.3 Translating an expression

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 *ftable* 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.

6.4.1 Examples of 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 `v0`, `v1` and `v2`, respectively and a function table that binds `f` to `_f`. The place for the result is `t0` and we assume that calls to *newvar*() return, in sequence, the variables `t1`, `t2`, `t3`, ...

We start by the simple expression `x-3`. This is a binop-expression, so the first we do is to call *newvar*() twice, giving $place_1$ the value `t1` and $place_2$ the value `t2`. We then call $Trans_{Exp}$ recursively with the expression `x`. When translating this, we first look up `x` in the variable symbol table, yielding `v0`, and then return the code [`t1` := `v0`]. 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 [`t2` := 3], which we assign to $code_2$. The final result is produced by $code_1 ++ code_2 ++ [\texttt{t0} := \texttt{t1} - \texttt{t2}]$ which yields $[\texttt{t1} := \texttt{v0},\ \texttt{t2} := 3,\ \texttt{t0} := \texttt{t1} - \texttt{t2}]$. We have translated the source-language operator `-` to the intermediate-language operator `-`.

The resulting code looks quite suboptimal, and could, indeed, be shortened to $[\texttt{t0} := \texttt{v0} - 3]$. When we generate intermediate code, we want, for simplicity, to treat each subexpression independently of its context. This may lead to superfluous assignments. We will look at ways of getting rid of these when we treat machine code generation and register allocation in Chaps. 7 and 8.

A more complex expression is `3+f(x-y,z)`. Using the same assumptions as above, this yields the code

$$
\begin{array}{l}
\texttt{t1} := 3 \\
\quad \texttt{t4} := \texttt{v0} \\
\quad \texttt{t5} := \texttt{v1} \\
\texttt{t3} := \texttt{t4} - \texttt{t5} \\
\texttt{t6} := \texttt{v2} \\
\texttt{t2} := \texttt{CALL\ \_f(t3,t6)} \\
\texttt{t0} := \texttt{t1} + \texttt{t2}
\end{array}
$$

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 Fig. 6.2 with statements. The extensions are shown in Grammar 6.4. Note that we use := for assignment to differentiate from the comparison operator.

$$
\begin{aligned}
\mathit{Stat} &\rightarrow \mathit{Stat}\ ;\ \mathit{Stat}\\
\mathit{Stat} &\rightarrow \textbf{id} := \mathit{Exp}\\
\mathit{Stat} &\rightarrow \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \mathit{Stat}\\
\mathit{Stat} &\rightarrow \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \mathit{Stat}\ \texttt{else}\ \mathit{Stat}\\
\mathit{Stat} &\rightarrow \texttt{while}\ \mathit{Cond}\ \texttt{do}\ \mathit{Stat}\\
\mathit{Stat} &\rightarrow \texttt{repeat}\ \mathit{Stat}\ \texttt{until}\ \mathit{Cond}\\
\mathit{Cond} &\rightarrow \mathit{Exp}\ \textbf{relop}\ \mathit{Exp}
\end{aligned}
$$

Grammar 6.4 Statement language

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 it on to Trans_{Exp}.

Just like we use *newvar* to generate new unused variables, we use a similar function *newlabel* to generate new unused labels. The translation function for statements is shown in Fig. 6.5. It uses an auxiliary translation function for conditions shown in Fig. 6.6.

A sequence of two statements is translated by putting the code for these in sequence.

An assignment is translated by translating the right-hand-side expression and then copying its result 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. 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,

$Trans_{Stat}(Stat, vtable, ftable)$ = `case` *Stat* `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*	$place = newvar()$ $x = lookup(vtable, getname(\textbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place) ++ [x := place]$
`if` *Cond* `then` $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_1 ++ [\texttt{LABEL}\ label_1] ++ code_2$ $++ [\texttt{LABEL}\ label_2]$
`if` *Cond* `then` $Stat_1$ `else` $Stat_2$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_3 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 ++ [\texttt{LABEL}\ label_1] ++ code_2$ $++ [\texttt{GOTO}\ label_3,\ \texttt{LABEL}\ label_2]$ $++ code_3 ++ [\texttt{LABEL}\ label_3]$
`while` *Cond* `do` $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_2, label_3, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] ++ code_1$ $++ [\texttt{LABEL}\ label_2] ++ code_2$ $++ [\texttt{GOTO}\ label_1,\ \texttt{LABEL}\ label_3]$
`repeat` $Stat_1$ `until` *Cond*	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond, label_2, label_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] ++ code_1$ $++ code_2 ++ [\texttt{LABEL}\ label_2]$

Fig. 6.5 Translation of statements

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

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) =$ `case` $Cond$ `of`	
Exp_1 **relop** Exp_2	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(getopname(\textbf{relop}))$ $code_1 ++ code_2 ++ [\texttt{IF}\ t_1\ op\ t_2\ \texttt{THEN}\ label_t\ \texttt{ELSE}\ label_f]$

Fig. 6.6 Translation of simple conditions

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.

Suggested exercises: 6.2.

6.6 Logical Operators

Logical conjunction, disjunction and negation are often available for conditions, so we can write, e.g., ($x = y$ **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 not evaluated if the first operand is sufficient to determine the result. This means that a logical **and** will not evaluate its second operand if its first argument evaluates to **false**, and a logical **or** will not evaluate its second operand if its first argument is **true**. 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 include the required relational operators, bitwise logical operations and logical negation. We can now simply allow any expression 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, the is no separate boolean type, the expression must be of integer type.

We then extend the translation function for conditions as follows:

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) =$ `case` $Cond$ `of`	
Exp_1 **relop** Exp_2	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(getopname(\textbf{relop}))$ $code_1 ++ code_2 ++ [$`IF` t_1 op t_2 `THEN` $label_t$ `ELSE` $label_f]$
Exp	$t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1 ++ [$`IF` $t \neq 0$ `THEN` $label_t$ `ELSE` $label_f]$

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 it separate 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 Fig. 6.7. 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 as little as possible, as this will yield the best code.

The revised translation functions for *Exp* and *Cond* are shown in Fig. 6.8. 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

$$
\begin{array}{lll}
\mathit{Exp} & \rightarrow & \textbf{num} \\
\mathit{Exp} & \rightarrow & \textbf{id} \\
\mathit{Exp} & \rightarrow & \textbf{unop}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \textbf{binop}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \textbf{id}(\mathit{Exps}) \\
\mathit{Exp} & \rightarrow & \texttt{true} \\
\mathit{Exp} & \rightarrow & \texttt{false} \\
\mathit{Exp} & \rightarrow & \mathit{Cond} \\
\\
\mathit{Exps} & \rightarrow & \mathit{Exp} \\
\mathit{Exps} & \rightarrow & \mathit{Exp}\ \texttt{,}\ \mathit{Exps} \\
\\
\mathit{Cond} & \rightarrow & \mathit{Exp}\ \textbf{relop}\ \mathit{Exp} \\
\mathit{Cond} & \rightarrow & \texttt{true} \\
\mathit{Cond} & \rightarrow & \texttt{false} \\
\mathit{Cond} & \rightarrow & \texttt{!}\ \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Cond}\ \texttt{\&\&}\ \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Cond}\ \texttt{||}\ \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Exp}
\end{array}
$$

Grammar 6.7 Example language with logical operators

number. We do this by first assuming that the condition is false and hence assign 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 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 $\mathit{Trans}_{\mathit{Cond}}$, 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 negates the effect of 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 $\mathit{Trans}_{\mathit{Cond}}$. The call to $\mathit{Trans}_{\mathit{Cond}}$ 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.

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
$\vdots$	
`true`	$[place := 1]$
`false`	$[place := 0]$
$Cond$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $[place := 0]$++$code_1$ ++[`LABEL` $label_1$, $place := 1$] ++[`LABEL` $label_2$]

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) =$ `case` $Cond$ `of`	
Exp_1 **relop** Exp_2	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(getopname(\textbf{relop}))$ $code_1$++$code_2$++[`IF` t_1 op t_2 `THEN` $label_t$ `ELSE` $label_f$]
`true`	[`GOTO` $label_t$]
`false`	[`GOTO` $label_f$]
`!` $Cond_1$	$Trans_{Cond}(Cond_1, label_f, label_t, vtable, ftable)$
$Cond_1$ `&&` $Cond_2$	$arg_2 = newlabel()$ $code_1$=$Trans_{Cond}(Cond_1, arg_2, label_f, vtable, ftable)$ $code_2$=$Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$++[`LABEL` arg_2]++$code_2$
$Cond_1$ `||` $Cond_2$	$arg_2 = newlabel()$ $code_1$=$Trans_{Cond}(Cond_1, label_t, arg_2, vtable, ftable)$ $code_2$=$Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$++[`LABEL` arg_2]++$code_2$
Exp	$t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1$++[`IF` $t \neq 0$ `THEN` $label_t$ `ELSE` $label_f$]

Fig. 6.8 Translation of sequential logical operators

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 non-sequential (e.g., bitwise) **and** and **or**, as these have a different effect (i.e., they always evaluate both arguments).

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 $Trans_{Cond}$ while assignment will choose a one similar to the current $Trans_{Exp}$.

Suggested exercises: 6.3.

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.

Goto and Labels Labels are stored in a symbol table that binds each 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.

The scope of labels can be controlled by the symbol table, so labels can be local to a procedure or block.

Break/Exit 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 parameter which is the label that a `break` or `exit` statement must jump to. 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 (see Chap. 5), though.

C's `continue` statement, which jumps to the start of the current loop, can be handled similarly.

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 is 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.

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 binary 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:

$$\begin{aligned} \mathit{Exp} &\rightarrow \mathit{Index} \\ \mathit{Stat} &\rightarrow \mathit{Index} := \mathit{Exp} \\ \mathit{Index} &\rightarrow \mathbf{id}[\mathit{Exp}] \end{aligned}$$

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
$Index$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $code_1 ++ [place := M[address]]$

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Index := Exp$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $t = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1 ++ code_2 ++ [M[address] := t]$

$Trans_{Index}(Index, vtable, ftable) =$ `case` $Index$ `of`	
id$[Exp]$	$base = lookup(vtable, getname(\textbf{id}))$ $t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_2 = code_1 ++ [t := t * 4, t := t + base]$ $(code_2, t)$

Fig. 6.9 Translation for one-dimensional arrays

Index 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. 6.9, we use 4 for the size of integers. We show only the new parts of the translation functions for *Exp* and *Stat*.

We use a translation function $Trans_{Index}$ 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 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.

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 from this at every array-lookup. However, this does not work if the sizes of these arrays are given at run-time. In this case, we need to use a variable to hold the base address of each array. The address is calculated when the array is allocated and then stored in the corresponding variable. This can subsequently be used as described in $Trans_{Index}$ 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 arrays that are no longer in use.

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. 6.10. 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]

[1]Note that the coordinate system, following computer-science tradition, is rotated 90° clockwise compared to mathematical tradition.

	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]`	⋯
⋮	⋮	⋮	⋮	⋱

Fig. 6.10 A two-dimensional array

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]`

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 $dim_0, dim_1, \ldots, dim_{n-2}, dim_{n-1}$, then in row-major format an element at index $[i_0][i_1]\ldots[i_{n-2}][i_{n-1}]$ has the address

$$base + ((\ldots(i_0 * dim_1 + i_1) * dim_2 \ldots + i_{n-2}) * dim_{n-1} + i_{n-1}) * size$$

In column-major format the address is

$$base + ((\ldots(i_{n-1} * dim_{n-2} + i_{n-2}) * dim_{n-3} \ldots + i_1) * dim_0 + i_0) * size$$

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:

$$\begin{aligned} Index &\rightarrow \mathbf{id}[Exp] \\ Index &\rightarrow Index[Exp] \end{aligned}$$

and extend the translation functions as shown in Fig. 6.11. 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 $Calc_{Index}$ to calculate the position of an element. In $Trans_{Index}$ we multiply this position by the element size and add the base address. As before, we assume the size of elements is 4.

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

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
$Index$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $code_1 ++ [place := M[address]]$

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Index := Exp$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $t = newvar()$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t)$ $code_1 ++ code_2 ++ [M[address] := t]$

$Trans_{Index}(Index, vtable, ftable) =$
$(code_1, t, vase, []) = Calc_{Index}(Index, vtable, ftable)$ $code_2 = code_1 ++ [t := t * 4, t := t + base]$ $(code_2, t)$

$Calc_{Index}(Index, vtable, ftable) =$ `case` $Index$ `of`	
id$[Exp]$	$(base, dims) = lookup(vtable, getname(\textbf{id}))$ $t = newvar()$ $code = Trans_{Exp}(Exp, vtable, ftable, t)$ $(code, t, base, tail(dims))$
$Index[Exp]$	$(code_1, t_1, vase, dims) = Calc_{Index}(Index, vtable, ftable)$ $dim_1 = head(dims)$ $t_2 = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t_2)$ $code_3 = code_1 ++ code_2 ++ [t_1 := t_1 * dim_1, t_1 := t_1 + t_2]$ $(code_3, t_1, base, tail(dims))$

Fig. 6.11 Translation of multi-dimensional arrays

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*—4, the size of the second dimension at *base*—8 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.

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 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. In general, 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.

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 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 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.

6.8.3 Strings

Strings are usually 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.

However, strings often differ from arrays in that the length is not static, but can vary at run-time. This leads to an implementation similar to the kind of arrays where the length is stored in memory, as explained in Sect. 6.8.2.2. Another difference is that the size of a character is typically one byte (unless 16-bit Unicode characters are used), so the index calculation does not multiply the index by the size (as this is 1). Operations on strings, e.g., concatenation and substring extraction, are typically implemented by calling 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 the base-address and the offsets for each field in the record. The symbol table for types must hold the offsets for every record type, such that these can be inserted into the symbol table for variables when a record of this type is declared.

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 an 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 Translating 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 keep in the symbol-table entry for that name 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 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.

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Decl\ ;\ Stat_1$	$(code_1, vtable_1) = Trans_{Decl}(Decl, vtable)$ $code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable)$ $code_1 ++ code_2$

$Trans_{Decl}(Decl, vtable) =$ `case` $Decl$ `of`	
`int` **id**	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\textbf{id}), t_1)$ $([],\ vtable_1)$
`int` **id**[**num**]	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\textbf{id}), t_1)$ $([t_1 := HP,\ HP := HP + (4 * getvalue(\textbf{num}))],\ vtable_1)$

Fig. 6.12 Translation of simple declarations

6.9.1 Simple Local Declarations

We extend the statement language by the following productions:

$$Stat \rightarrow Decl\ ;\ Stat$$

$$Decl \rightarrow \texttt{int}\ \textbf{id}$$
$$Decl \rightarrow \texttt{int}\ \textbf{id}[\textbf{num}]$$

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 6.12 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.

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 the value. Local variable declarations are translated like the local declarations above.

If functions can call functions that are declared later, we use two passes: One to build *vtable* and another to translate the function definitions using this *vtable*. 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 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 the complex operations also helped 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.

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 *ftable* 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 6.8 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 `||` q`)` is equivalent to `(!`p`)` `&&` `(!`q`)`. Show that these generate identical code when compiled with $Trans_{Cond}$ from Fig. 6.8.

Exercise 6.5 Show that, in any code generated by the functions in Figs. 6.5 and 6.8, 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

$$Stat \rightarrow \texttt{break}$$

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:

$$Stat \rightarrow \textbf{labelid} :$$
$$Stat \rightarrow \texttt{goto}\ \textbf{labelid}$$

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. 6.11, 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:

$$\begin{array}{l}\texttt{LABEL}\ label_1\\ \texttt{GOTO}\ \ label_2\end{array}$$

It is clear that any jump to $label_1$ can be replaced by a jump to $label_2$, 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 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. 6.8, i.e., that p && (q && r) generates the same code (up to renaming of labels) as (p && q) && r.

Exercise 6.12 Figure 6.11 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 $dim_0, dim_1, \ldots, dim_n$ is implemented as a one-dimensional array of size dim_0 with pointers to dim_0 different arrays each of dimension $dim_1, \ldots, dim_n$, 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.

a) Assuming pointers and array elements need four bytes each, what is the total number of bytes required to store an array of dimensions $dim_0, dim_1, \ldots, dim_n$?
b) Write translation functions for array-access in the style of Fig. 6.11 using this representation of arrays. Use addition to multiply numbers by 4 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

$$\begin{aligned} \mathit{FunDec} &\rightarrow \textbf{id}\ (\ \mathit{Params}\ )\ \mathit{Stat} \\ \mathit{Params} &\rightarrow \textbf{id} \\ \mathit{Params} &\rightarrow \textbf{id}\ ,\ \mathit{Params} \\ \mathit{Stat} &\rightarrow \texttt{return}\ \mathit{Exp} \end{aligned}$$

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 *ftable* 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

1. Aït-Kaci, H.: Warren's Abstract Machine—A Tutorial Reconstruction. MIT Press, Cambridge (1991)
2. Appel, A.W.: Compiling with Continuations. Cambridge University Press, Cambridge (1992)
3. Jones, S.L.P., Lester, D.: Implementing Functional Languages—A Tutorial. Prentice Hall, Reading (1992)
4. Lattner, C.: Llvm language reference manual (2011). http://llvm.org/docs/LangRef.html
5. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
6. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Mateo (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 reg-

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_7, © Springer-Verlag London Limited 2011

ister 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   l_t
jump          l_f
```

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   l_f
```

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 test 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.

If the conditional jump instructions in the target machine language do not allow conditions as complex as those used in the intermediate language, code must be generated to first calculate the condition and put the result somewhere where it can be tested by a subsequent conditional jump instruction. In some machine architectures, e.g., MIPS and Alpha, this "somewhere" can be a general-purpose register. Other machines, e.g., PowerPC or IA-64 (also known as Itanium) use special condition registers, while yet others, e.g., IA-32 (also known as x86), Sparc, PA-RISC and ARM use a single set of arithmetic flags that can be set by comparison or arithmetic instructions. A conditional jump may test various combinations of the flags, so the same comparison instruction can, depending on the subsequent condition, be used for testing equality, signed or unsigned less-than, overflow and several other properties. 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.

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 the ARM, 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.

When an intermediate-language instruction uses a constant, the code generator must check if it 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, followed by an instruction that uses this register 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 this 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.

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`, which will be detailed in Chap. 9.

CISC (Complex Instruction Set Computer) processors like IA-32 have composite (i.e., non-atomic) instructions in abundance. And while the philosophy behind RISC (Reduced Instruction Set Computer) processors like MIPS and ARM advocates that machine-code instructions should be simple, most RISC processors include at least a few non-atomic instructions, typically for memory-access instructions.

We will in this chapter use a subset of the MIPS instruction set as an example. A description of the MIPS instruction set can be found Appendix A of [5], which is available online [3]. If you are not already familiar with the MIPS 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

$$t_2 := t_1 + 116$$
$$t_3 := M[t_2]$$

can be translated into the single MIPS instruction

```
lw r3, 116(r1)
```

where `r1` and `r3` are the registers chosen for t_1 and t_3, respectively. However, it is only possible to combine the two instructions if the value of the intermediate value 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 it is *dead* after a particular use. When generating intermediate code, most of the temporary variables introduced by the compiler will be single-use and can be marked as such. Any use of a single-use variable will, by definition, be the last 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. We assume this is done, and the last use of any variable in the intermediate code is marked by *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 MIPS `lw` (load word) instruction can be described by the pattern/replacement pair

$t := r_s + k$ $r_t := M[t^{last}]$	`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`, which we handle separately in Chap. 9) must be covered by at least one pattern. This means that we must include the MIPS 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 replacement 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 so 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 MIPS microprocessor architecture in terms of the intermediate language as a set of pattern/replacement pairs. Note that we exploit the fact that register 0 is hardwired to be the value 0 to, e.g., use the `addi` instruction to generate a constant. 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 a `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 MIPS 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 MIPS 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` and `slt`.

$t := r_s + k$, $r_t := M[t^{last}]$		`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(`R0`)
$t := r_s + k$, $M[t^{last}] := r_t$		`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(`R0`)
$r_d := r_s + r_t$		`add`	r_d, r_s, r_t
$r_d := r_t$		`add`	r_d, `R0`, r_t
$r_d := r_s + k$		`addi`	r_d, r_s, k
$r_d := k$		`addi`	r_d, `R0`, k
`GOTO` *label*		`j`	*label*
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_f$		`beq` $label_f$:	$r_s, r_t, label_t$
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_t$		`bne` $label_t$:	$r_s, r_t, label_f$
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$		`beq` `j`	$r_s, r_t, label_t$ $label_f$

`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$,
`LABEL` $label_f$
		`slt`
		`bne`
		$label_f$:
			r_d, r_s, r_t
			r_d, `R0`, $label_t$

`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$,
`LABEL` $label_t$
		`slt`
		`beq`
		$label_t$:
			r_d, r_s, r_t
			r_d, `R0`, $label_f$

`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$		`slt`
		`bne`
		`j`
			r_d, r_s, r_t
			r_d, `R0`, $label_t$
			$label_f$

`LABEL` *label*	*label*:		

Fig. 7.1 Pattern/replacement pairs for a subset of the MIPS instruction set

We can try to use Fig. 7.1 to select MIPS instructions for the following sequence of intermediate-language instructions:

$a := a + b^{last}$
$d := c + 8$
$M[d^{last}] := a$
`IF` $a = c$ `THEN` $label_1$ `ELSE` $label_2$
`LABEL` $label_2$

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 $label_2$ even though the resulting sequence does not refer to it, as 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:

$r_t := r_s$	`mov` r_t, r_s
$r_t := r_t + r_s$	`add` r_t, r_s
$r_d := r_s + r_t$	`move` r_d, r_s `add` r_d, r_t

that 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_d 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 to new registers before operations and let register allocation allocate these to the right type of registers (and eliminate as many of the moves as possible).

Suggested exercises: 7.2.

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

[1] This is, strictly speaking, not entirely true, as superfluous labels might inhibit later optimisations.

it makes sense to perform optimisations mainly in the intermediate language, as the optimisations hence 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.

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.

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. Again, this optimisation is not shown.

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 reduce calculations. 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` × `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.

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 [4] or [2] for more details.

Descriptions of and methods for implementation of a large number of different optimisations can be found in [1, 4] and [2].

The instruction set of (one version of) the MIPS microprocessor architecture is described in [5]. This description is also available online [3].

7.6 Exercises

Exercise 7.1 Add extra inherited attributes to $Trans_{Cond}$ in Fig. 6.8 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:

$$
\begin{array}{l}
d := c + 8 \\
:= a + b^{last} \\
M[d^{last}] := a \\
\texttt{IF } a < c \texttt{ THEN } label_1 \texttt{ ELSE } label_2 \\
\texttt{LABEL } label_1
\end{array}
$$

Compare this to the example in Section 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 MIPS) 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 such a 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 MIPS, register 0 (`R0`) is hardwired to the value 0.

Add to the above table pattern/replacement pairs sufficient to translate the following intermediate-code instructions to sequences of machine-code instructions using only `MOV`, `ADD` and `ADDI` instructions in the replacement sequences:

$$
\begin{array}{l}
r_d := k \\
r_d := r_s + r_t \\
r_d := r_s + k
\end{array}
$$

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, Techniques and Tools. Addison-Wesley, Reading (2007)
2. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge (1998)
3. Larus, J.: Assembler, linkers and the spim simulator (1998). http://pages.cs.wisc.edu/~larus/HP_AppA.pdf
4. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Mateo (1997)
5. Patterson, D.A., Hennessy, J.L.: Computer Organization & Design, the Hardware/Software Interface. Morgan Kaufmann, San Mateo (1998)

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 $<$ to MIPS code. Hence, the register allocator must make sure that there are enough spare registers for temporary storage. Usually, this is only one or two registers, but on a processor with a small number of registers, this can be significant.

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_8, © Springer-Verlag London Limited 2011

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.

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 can be used in the future.

We have already hinted at this concept in Chap. 7, when we talked about last-uses of variables.

Loosely speaking, two variables may share a register 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 as in Fig. 8.2.

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 $succ[i]$. We use the following rules to find $succ[i]$:

1) The instruction numbered j (if any) that is listed just after instruction number i is in $succ[i]$, unless i is a `GOTO`, `IF-THEN-ELSE` or `RETURN` instruction. If instructions are numbered consecutively, $j = i + 1$.
2) If instruction number i is of the form `GOTO` l, (the number of) the instruction `LABEL` l is in $succ[i]$. Note that there in a correct program will be exactly one `LABEL` instruction with the label used by the `GOTO` instruction.
3) If instruction i is `IF` c `THEN` l_t `ELSE` l_f, (the numbers of) the instructions `LABEL` l_t and `LABEL` l_f are in $succ[i]$.
4) If instruction i is of the form `RETURN` x, $succ[i]$ is empty.

Note that we assume that both outcomes of an `IF-THEN-ELSE` instruction are possible. If this happens not to be the case (i.e., if the condition is always true or always false), our liveness 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 for a variable that is not going to be used after all. 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 may be assigned a value by the instruction. Figure 8.1 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.

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:

$$in[i] = gen[i] \cup (out[i] \setminus kill[i]) \tag{8.1}$$

$$out[i] = \bigcup_{j \in succ[i]} in[j] \tag{8.2}$$

Fig. 8.1 Gen and kill sets

Instruction i	$gen[i]$	$kill[i]$
`LABEL` l	$\emptyset$	$\emptyset$
$x := y$	$\{y\}$	$\{x\}$
$x := k$	$\emptyset$	$\{x\}$
$x :=$ **unop** y	$\{y\}$	$\{x\}$
$x :=$ **unop** k	$\emptyset$	$\{x\}$
$x := y$ **binop** z	$\{y, z\}$	$\{x\}$
$x := y$ **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 **relop** y `THEN` l_t `ELSE` l_f	$\{x, y\}$	$\emptyset$
$x :=$ `CALL` $f(args)$	$args$	$\{x\}$
`RETURN` x	$\{x\}$	$\emptyset$

These equations are recursive. We solve these by fixed-point iteration, as shown in Appendix: 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 sets with finite support (i.e., a finite number of possible values) 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. 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.

Figure 8.2 shows a small program that we will calculate liveness for. Figure 8.3 shows *succ*, *gen* and *kill* sets for the instructions in the program.

The program in Fig. 8.2 is a function that calculates the Nth Fibonacci number. The *out* sets are defined by (8.2) and all *in* sets are defined by (8.1). We initialise all *in* and *out* sets to the empty set and iterate 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

$$out[14],\ in[14],\ out[13],\ in[13], \ldots, out[1],\ in[1]$$

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

Fig. 8.2 Example program for liveness analysis and register allocation

```
   fib(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
   ]
```

Fig. 8.3 *succ*, *gen* and *kill* for the program in Fig. 8.2

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	

(8.1) 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.

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

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

initialised, which is generally considered an error (since it can lead to unpredictable results and even security holes). Some compilers issue warnings about uninitialised variables and some compilers add instructions to initialise 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 $x \in kill[i]$, $y \in out[i]$ *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. 8.2:

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

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 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. Also, 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.

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. 8.2 is shown in Fig. 8.5.

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 share an edge have different register numbers.
2) The total number of different register numbers is no higher than the number of available registers.

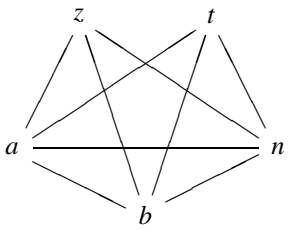


Fig. 8.5 Interference graph for the program in Fig. 8.2

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-complete, 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 remaining colours.

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 it is connected to, and remove it 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 more nodes left in the graph, continue with **simplify**, otherwise go to **select**.

select: Take a node and its list of connected nodes from 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 more nodes on the stack, continue with **select**.

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).

8.5 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, hence, select some variables that will reside in memory (except for brief periods). This process is called *spilling*. Obvious candidates for spilling are variables at 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.
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. 8.2, 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 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. Note that, since a is live at the end of the program, we have inserted a load instruction at the end of the program. Figure 8.8 shows the interference graph for the program in Fig. 8.7 and Fig. 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, some other node is chosen. So far, we have chosen arbitrarily, but we may apply some heuristics (qualified guessing) to the choice in order to make colouring more likely or reduce the number of spilled variables:

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. 8.2 after spilling variable a

$fib(n)$ [
1: $a_1 := 0$
$M[address_a] := a_1$
2: $b := 1$
3: $z := 0$
4: `LABEL` *loop*
5: `IF` $n = z$ `THEN` *end* `ELSE` *body*
6: `LABEL` *body*
$a_7 := M[address_a]$
7: $t := a_7 + b$
8: $a_8 := b$
$M[address_a] := a_8$
9: $b := t$
10: $n := n - 1$
11: $z := 0$
12: `GOTO` *loop*
13: `LABEL` *end*
$a_{14} := M[address_a]$
14: `RETURN` a_{14}
]

- 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 maybe others as well) may be combined into a single heuristic by giving numeric values describing how well a variable fits each criterion,

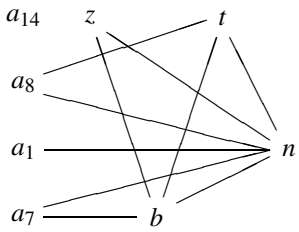


Fig. 8.8 Interference graph for the program in Fig. 8.7

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

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 neigbours, 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.

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 will have no effect). Most register allocators remove such redundant move instructions, and some even 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. This is not quite as effective at increasing the chance of colouring as spilling, but the cost of the extra assignments is likely to be less than the cost of the loads and stores inserted by spilling.

8.6.2 Using Explicit Register Numbers

Some operations 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 results to such operations must, hence, be assigned to these registers *a priori*, before the register allocation begins. We call these precoloured nodes in the interference graph. If two nodes that are precoloured 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 to insert move instructions that move the variables to and from the required registers before and after an instruction that requires specific registers. The specific registers must still be included as precoloured nodes in the interference graph, but are not removed from it in the **simplify** phase. Once only precoloured 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 precoloured 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].

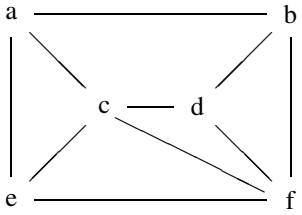
8.8 Exercises

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 a higher score implies a stronger candidate for spill.

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 between address and data registers.

By adding the registers as nodes in the interference graph, a variable can be prevented from being allocated in a specific register by making it interfere with it.

a) Describe how instructions that require argument or result variables to be in a specific type of register can ensure this by adding interference for its argument and result variables.
b) The answer above is likely to cause spilling of variables that are used as both address and data (as they interfere with all registers). Describe how this can be avoided by taking care of this situation in the spill phase. Hint: Add register-to-register move instructions to the program.
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 the instructions only need 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.

References

1. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge (1998)
2. Briggs, P.: Register allocation via graph coloring. Ph.D. thesis, Rice University, Center for Research on Parallel Computation (1992). Tech. rept. cpc-tr94517-s
3. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Mateo (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.

We will, in this chapter, remedy these omissions. We will initially assume that all variables are local to the function that access 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

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4_9, © Springer-Verlag London Limited 2011

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 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

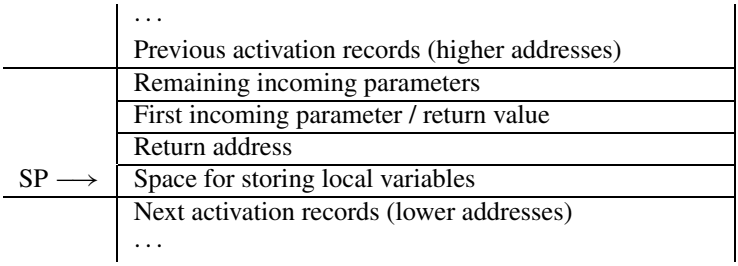


Fig. 9.1 Simple activation record layout

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 and position of stack pointer relative to the stack top can differ from system to system.

In this layout, the first (top) words of the activation record holds the incoming parameters. 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, local arrays or for preserving variables across later function calls.

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 intermediate-code, named 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 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 jump 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 $f(p_1, \ldots, p_m)$ 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 parameters and return address.

$$
\begin{array}{l}
\texttt{LABEL}\ f \\
p_1 := M[SP + 4] \\
\ldots \\
p_m := M[SP + 4 * m] \\
SP := SP - framesize_f
\end{array}
$$

Fig. 9.2 Prologue for the header $f(p_1, \ldots, p_m)$ using the frame layout shown in Fig. 9.1

$$
\begin{array}{l}
SP := SP + framesize_f \\
M[SP + 4] := result \\
\texttt{GOTO}\ M[SP]
\end{array}
$$

Fig. 9.3 Epilogue for the instruction RETURN *result* using the frame layout shown in Fig. 9.1

Note that, though we have used a notation similar to the intermediate language introduced in Chap. 6, we have extended this a bit: We have used $M[]$ and GOTO with general expressions as arguments.

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 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 $x :=$ CALL $g(a_1, \ldots, a_n)$ called from a function f.

First, all registers that can be used to hold variables are stored in the caller's (i.e, f's) frame. In Fig. 9.4, $R0$–Rk are assumed to hold variables. These are stored in the space for storing local variables.

Before storing the parameters and the return address in g's frame, we must move *SP*, as otherwise a system interrupt (that uses the same stack) might overwrite the values. For the same reason, we don't restore *SP* to its former 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 former value and the saved registers are read back from the old frame.

$$
\begin{array}{l}
M[SP] := R0 \\
\ldots \\
M[SP + 4 * k] := Rk \\
SP := SP - 4 * (n + 1) \\
M[SP + 4] := a_1 \\
\ldots \\
M[SP + 4 * n] := a_n \\
M[SP] := \mathit{returnaddress} \\
\texttt{GOTO}\ g \\
\texttt{LABEL}\ \mathit{returnaddress} \\
x := M[SP + 4] \\
SP := SP + 4 * (n + 1) \\
R0 := M[SP] \\
\ldots \\
Rk := M[SP + 4 * k]
\end{array}
$$

Fig. 9.4 Call sequence for $x := \texttt{CALL}\ g(a_1, \ldots, a_n)$ using the frame layout shown in Fig. 9.1

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 $a_1, \ldots, a_n$ 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 $a_1, \ldots, a_n$ 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 frame all registers that can hold register-allocated variables, 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

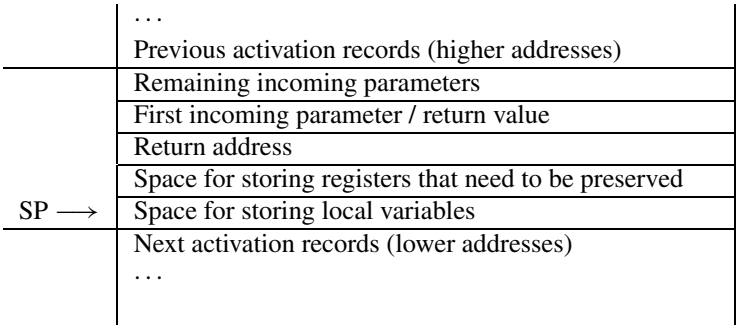


Fig. 9.5 Activation record layout for callee-saves

$$
\begin{array}{l}
\texttt{LABEL } f \\
SP := SP - framesize_f - 4 * (k+1) \\
M[SP + framesize_f] := R0 \\
\ldots \\
M[SP + framesize_f + 4 * k] := Rk \\
p_1 := M[SP + framesize_f + 4 * (k+1)] \\
\ldots \\
p_m := M[SP + framesize_f + 4 * (k+m)]
\end{array}
$$

Fig. 9.6 Prologue for the header $f(p_1, \ldots, p_m)$ using callee-saves

(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), only the space for local variables. If $framesize_f$ is known at compile time, offsets like $framesize_f + 4 * (k+m)$ can be calculated at compile time, so all offsets to SP are constants.

Note that it may not be necessary to store *all* registers that can 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.

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 may potentially hold variables, other differences emerge: Caller-saves need only save the registers that hold *live* variables and callee-saves need only save the

$$
\begin{array}{l}
M[SP + framesize_f + 4 * (k+1)] := result \\
R0 := M[SP + framesize_f] \\
\ldots \\
Rk := M[SP + framesize_f + 4 * (k+m)]] \\
SP := SP + framesize_f + 4 * (k+1) \\
\texttt{GOTO}\ M[SP]
\end{array}
$$

Fig. 9.7 Epilogue for the instruction `RETURN` *result* using callee-saves

$$
\begin{array}{l}
SP := SP - 4 * (n+1) \\
M[SP+4] := a_1 \\
\ldots \\
M[SP + 4 * n] := a_n \\
M[SP] := returnaddress \\
\texttt{GOTO}\ g \\
\texttt{LABEL}\ returnaddress \\
x := M[SP+4] \\
SP := SP + 4 * (n+1)
\end{array}
$$

Fig. 9.8 Call sequence for $x :=$ `CALL` $g(a_1, \ldots, a_n)$ using callee-saves

registers that the function actually uses. 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/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 function 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.

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

	... Previous activation records (higher addresses)
	Incoming parameters in excess of four
	Return address
	Space for storing callee-saves registers that need to be preserved
SP ⟶	Space for storing local variables
	Next activation records (lower addresses) ...

Fig. 9.10 Activation record layout for the register division shown in Fig. 9.9

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 few 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. 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 if it is used in the body.

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.

```
LABEL f
SP := SP − framesize_f − 4 ∗ U
M[SP + framesize_f + offset_R4] := R4      (if used in body)
...
M[SP + framesize_f + offset_R12] := R12    (if used in body)
M[SP + framesize_f] := R15                 (if used in body)
p_1 := R0
p_2 := R1
p_3 := R2
p_4 := R3
p_5 := M[SP + framesize_f + 4 ∗ U]
...
p_m := M[SP + framesize_f + 4 ∗ (U + m − 5)]
```

Fig. 9.11 Prologue for the header $f(p_1, \ldots, p_m)$ using the register division shown in Fig. 9.9

```
R0 := result
R4 := M[SP + framesize_f + offset_R4]      (if used in body)
...
R12 := M[SP + framesize_f + offset_R12]    (if used in body)
R15 := M[SP + framesize_f]                 (if used in body)
SP := SP + framesize_f + 4 ∗ U
GOTO R15
```

Fig. 9.12 Epilogue for the instruction RETURN *result* using the register division shown in Fig. 9.9

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.

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.

$M[SP + offset_{live_1}] := live_1$ (if allocated to a caller-saves register)
...
$M[SP + offset_{live_k}] := live_k$ (if allocated to a caller-saves register)
$SP := SP - max(0, 4 * (n - 4))$
$R0 := a_1$
...
$R3 := a_4$
$M[SP + 4] := a_5$
...
$M[SP + 4 * (n - 4)] := a_n$
$R15 := returnaddress$
GOTO g
LABEL *returnaddress*
$x := R0$
$SP := SP + max(0, 4 * (n - 4))$
$live_1 := M[SP + offset_{live_1}]$ (if allocated to a caller-saves register)
...
$live_k := M[SP + offset_{live_k}]$ (if allocated to a caller-saves register)

Fig. 9.13 Call sequence for $x :=$ CALL $g(a_1, \ldots, a_n)$ 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 such variables need to be saved before the function call.

Likewise, the register allocator can return information about which registers are used by the function body, 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 (if the callee does 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 caller-saves registers, this is all done by the register allocator.

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 vari-

ables that are live across function calls. 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 the 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 is called on (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 back again. This overhead is, however, usually eliminated by later optimisations.

It may seem odd that we move register-passed parameters to named registers instead of just letting them stay in the registers they are passed in. 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. 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 precoloured temporary variables used for parameter-passing in call sequences or for instructions that require specific registers, see Sect. 8.6.2).
2) Add code for moving parameters from numbered registers and stack locations into the named variables used for accessing the parameters in the body of the function, and for moving the function-result from a named register to the register used for function results.
3) Call the register allocator with this extended function body. 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, and should return both the set of used callee-saves registers and the set of spilled variables.
4) To the register-allocated code, add code for saving and restoring the callee-saves registers that the register allocator said have been used in the extended function

body (including the return-address register) 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.

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. For arrays and records, this means that the address of any element or field is a compile-time constant from the stack pointer, so we can use the frame layouts, prologues, epilogues and call sequences above without modification.

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.

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 or local to a function. Local variables are treated exactly as we have described, i.e, typically stored in a register. Global variables will, on the other hand, be stored in memory. The location of each global variable

will be known 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 global variables are treated almost like spilled variables: Their 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. Hence, it is best to allocate local copies of global variables 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 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.

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, e.g., Pascal (which uses the term var-parameters), allow parameters to be passed by *call-by-reference*. 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 in that these update only a local copy.

Call-by-reference is implemented by passing the address of the variable, array element or whatever that is given as parameter. 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. When the value of the variable is used or updated, this pointer must be explicitly followed, using the `*` (dereference) operator. So, apart from notation and a higher potential for programming errors, this is not significantly different from "real" call-by-reference parameters.

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. This means that it must be spilled at the time of the call or allocated to a caller-saves register, so it will be stored before the call and restored afterwards.

It also means that passing a result back to the caller by call-by-reference or pointer parameters can be slower than using the function's 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.

Either 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 can't 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 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, point to the opposite end of the frame than the stack pointer does. 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 the return address. 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, there is a fixed number of parameters known at compile time and the number of extra parameters can be calculated from these.

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. 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–$+127$), 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., Sun's Sparc and Intel's IA-64 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 signaled 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.

Suggested exercises: 9.3.

9.12 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 [3]. Additionally, the calling convention for the MIPS microprocessor is shown in [4].

Functions declared inside other functions require more complex mechanisms than described above, especially if they can be passed as arguments or returned as function values. See [1, 2] for how this can be done.

9.13 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 Sometimes, a function call is immediately followed by a return, such as in the C statement `return g(p,q)`. This means that a call sequence is immediately followed by an epilogue, which can allow for optimisations.

Combine the call sequence in Fig. 9.13 with the epilogue in Fig. 9.12 and try to optimise as much as possible. You can assume that all parameters are passed in registers (i.e., that there are no more than four parameters). Exploit that *x* and *result* are the same variable and that no other variables are live after the call. If you reorder operations that can be executed out of order, you can combine more operations than if you keep the order strictly. When you reorder operations, argue why they can be reordered.

If you are careful, you can even reuse the caller's frame for the callee's frame. This is called *tail-call optimisation*.

Exercise 9.4 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 $label_1$ ELSE $label_2$
LABEL $label_1$
$code_1$
GOTO $label_3$
LABEL $label_2$
$code_2$
LABEL $label_3$

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, get information about variable use in the different parts of the body (i.e., *cond*, $code_1$ and $code_2$) from the register allocator.

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 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 $code_1$".

References

1. Appel, A.W.: Compiling with Continuations. Cambridge University Press, Cambridge (1992)
2. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge (1998)
3. ARM Limited: Procedure call standard for the arm architecture (2009). http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf
4. Patterson, D.A., Hennessy, J.L.: Computer Organization & Design, the Hardware/Software Interface. Morgan Kaufmann, San Mateo (1998)

Appendix
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}. The order of elements do not matter in a set, 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".

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4,

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: "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:

$$x \in (M \cup N) \quad \Leftrightarrow \quad x \in M \vee x \in N$$

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:

$$x \in (M \cap N) \quad \Leftrightarrow \quad x \in M \wedge x \in N$$

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 or M but not members of N. For example, $\{1, 2\} \setminus \{5, 1\} = \{2\}$. The following statement holds for membership and set difference:

$$x \in (M \setminus N) \quad \Leftrightarrow \quad x \in M \wedge x \notin N$$

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

$$M \subseteq N \quad \Leftrightarrow \quad (x \in M \Rightarrow x \in N)$$

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

$$
\begin{aligned}
A \cup B &= B \cup A && \text{union is commutative}\\
A \cap B &= B \cap A && \text{intersection is commutative}\\
A \cup (B \cup C) &= (A \cup B) \cup C && \text{union is associative}\\
A \cap (B \cap C) &= (A \cap B) \cap C && \text{intersection is associative}\\
A \cup (B \cap C) &= (A \cup B) \cap (A \cup C) && \text{union distributes over intersection}\\
A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) && \text{intersection distributes over union}\\
A \cup \emptyset &= A && \text{the empty set is a unit element of union}\\
A \cap \emptyset &= \emptyset && \text{the empty set is a zero element of intersection}\\
A \subseteq B &\Leftrightarrow A \cup B = B && \text{subset related to union}\\
A \subseteq B &\Leftrightarrow A \cap B = A && \text{subset related to intersection}\\
A \subseteq B &\Leftrightarrow A \setminus B = \emptyset && \text{subset related to set difference}\\
A \subseteq B \wedge B \subseteq A &\Leftrightarrow A = B && \text{subset is antisymmetric}\\
A \subseteq B \wedge B \subseteq C &\Rightarrow A \subseteq C && \text{subset is transitive}\\
A \setminus (B \cup C) &= (A \setminus B) \setminus C && \text{corresponds to } x - (y + z) = (x - y) - z
\end{aligned}
$$

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 fulfills 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 $\{1, 2, 3, 4\}$ fulfill 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 fulfill 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 $\bigcap\{\{1\}, \{1, 2, 4\}, \{1, 3, 9\}\} = \{1\}$.

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

$$X = \{x^2 \mid x \in X\}$$

This particular equation has several solutions, including $X = \{0\}$, $X = \emptyset$ and $X = \{0, 1\}$ or 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 $X = F(X)$.

We say that a set function F is *monotonic* if $X \subset Y \Rightarrow F(X) \subseteq F(Y)$.

Theorem A.1 *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

$$\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq F(F(F(\emptyset))) \subseteq \cdots$$

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 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 $F^m(\emptyset)$. □

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 := Ø;
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 := Ø;
W := F(Ø);
while W ≠ Ø do
   pick x ∈ W;
   W := W\{x};
   X := X ∪ {x};
   W := W ∪ (F({x})\X);
return X
```

We keep a work set `W` that by invariant is equal to $F(\mathtt{X}) \setminus \mathtt{X}$. A solution must include any $x \in \mathtt{W}$, so we move this from `W` to `X` while keeping the invariant by adding $F(x) \setminus \mathtt{X}$ to `W`. When `W` becomes empty, we have $F(\mathtt{X}) = \mathtt{X}$ 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:

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

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 $i = 1 \ldots n$ 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.

Exercises

Exercise A.2 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.3 What set is built by the set expression

$$\bigcup_{x \in \{1, 2, 3\}} \{x^n \mid n \in \{0, 1, 2\}\} \ ?$$

Exercise A.4 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 X and see for which the equation holds.

Exercise A.5 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

T.Æ. Mogensen, *Introduction to Compiler Design*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-829-4, © Springer-Verlag London Limited 2011