Basics of Compiler Design

Extended edition

Torben Ægidius Mogensen

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF COPENHAGEN

Published through lulu.com.

torbenm@diku.dk

Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen
DENMARK

Book homepage:
http://www.diku.dk/∼torbenm/Basics

First published 2000
This edition: July 25, 2008

Contents

List of Figures

Chapter 1

Introduction

1.1 What is a compiler?

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 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.

1.2 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 doesn't 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.

1.3 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 (see chapter 11), 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 not discuss interpreters in any detail in this book, except in relation to bootstrapping in chapter 11. A good introduction to interpreters can be found in [2].

1.4 Why learn about compilers?

Few people will ever be required to write a compiler for a general-purpose language like C, Pascal 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.

1.5 The structure of this book

The first part 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 chapter 2 and syntactical analysis in chapter 3.

The second part of the book (chapters 4 – 9) covers the middle part and back-end of the compiler, where the program is converted into machine language. Chapter 4 covers how definitions and uses of names (*identifiers*) are connected through *symbol tables*. In chapter 5, this is used to type-check the program. In chapter 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 chapter 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 chapter 8. Up to this point, a "program" has been what corresponds to the body of a single procedure. Procedure calls and nested procedure declarations add some issues, which are discussed in chapter 9. Chapter 10 deals with analysis and optimisation.

Finally, chapter 11 will discuss the process of *bootstrapping* a compiler, *i.e.*, using a compiler to compile itself.

Chapter 10 (on analysis and optimisation) was not found in editions before April 2008, which is why the latest editions are called "extended"

1.6 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 DIKU, the compiler course was previously taught right after the introductory programming course[1], which is earlier than in most other universities. Hence, existing textbooks tended either to be too advanced for the level of the course or be too simplistic in their approach, for example only describing a single very simple compiler without bothering too much with the general picture.

This book was written as a response to this and aims at bridging the gap: It is intended to convey the general picture without going into extreme detail about such things as efficient implementation or the newest techniques. It should give the students an understanding of how compilers work and the ability to make simple (but not simplistic) compilers

[1] It is now in the second year.

for simple languages. It will also lay a foundation that can be used for studying more advanced compilation techniques, as found *e.g.* in [30].

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 set of exercises. Few of these require access to a computer, but can be solved on paper or black-board. In fact, many of the exercises are based on exercises that have been used in written exams at DIKU.

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. Suitable credit for a course that uses this book is from 5 to 10 ECTS points, depending on the amount of project work.

1.7 Acknowledgements

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 draft 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. I am extremely grateful to the people who in 2000 read parts of or all of the first draft and made helpful suggestions.

1.8 Permission to use

Chapter 2

Lexical Analysis

2.1 Introduction

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 words are traditionally called *tokens*.

A *lexical analyser*, or *lexer* for short, 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 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.

2.2 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 using these letters) we will use typewriter font, *e.g.*, `a` or `b`. Hence, when

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.
$s\|t$	$\mathrm{L}(s) \cup \mathrm{L}(t)$	Strings from both languages
st	$\{vw \mid v \in \mathrm{L}(s), w \in \mathrm{L}(t)\}$	Strings constructed by concatenating a string from the first language with a string from the second language. Note: In set-formulas, "|" isn't a part of a regular expression, but part of the set-builder notation and reads as "where".
s^*	$\{\text{""}\} \cup \{vw \mid v \in \mathrm{L}(s),\ w \in \mathrm{L}(s^*)\}$	Each string in the language is a concatenation of any number of strings in the language of s.

Figure 2.1: Regular expressions

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 $\mathrm{L}(s)$ to denote the language (*i.e.*, set of strings) described by the regular expression s. For example, $\mathrm{L}(\texttt{a})$ is the set {"`a`"}.

Figure 2.1 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 2.10).
- $s|t$ (pronounced "s or t") describes the union of the languages described by s and t.
- 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 as and bs.

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 isn't *a priori* clear how the different subexpressions are grouped. We can use parentheses to make the grouping of symbols clear. 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 is based on set union, which has these properties). Concatenation is associative (but obviously not commutative) and distributes over |. Figure 2.2 shows these and other algebraic properties of regular expressions, including definitions of some shorthands introduced below.

2.2.1 Shorthands

While the constructions in figure 2.1 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 occurrences of s. With this notation, we can abbreviate our description of integers to $[0\text{-}9]^+$. On a similar note, it is common that we can have zero or one occurrence of something (*e.g.*, an optional sign to a number). Hence we introduce the shorthand $s?$ for $s|\epsilon$. $^+$ and ? bind with the same precedence as *.

$$
\begin{array}{c}
(r|s)|t \;=\; r|s|t \;=\; r|(s|t) \\
s|t \;=\; t|s \\
s|s \;=\; s \\
s? \;=\; s|\epsilon \\
(rs)t \;=\; rst \;=\; r(st) \\
s\epsilon \;=\; s \;=\; \epsilon s \\
r(s|t) \;=\; rs|rt \\
(r|s)t \;=\; rt|st \\
(s^*)^* \;=\; s^* \\
s^*s^* \;=\; s^* \\
ss^* \;=\; s^+ \;=\; s^*s
\end{array}
$$

Figure 2.2: Some algebraic properties of regular expressions

We must stress that these shorthands are just that. They don't 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.

2.2.2 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 doesn't (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 section 2.9 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]|\[a-zA-Z])*"
```

2.3 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 2.1 *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 2.3 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 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.

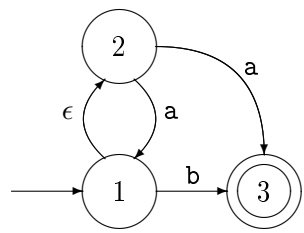


Figure 2.3: Example of an NFA

2.4 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 figure 2.4. 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 figure 2.5. We have numbered the states for future reference.

2.4.1 Optimisations

We can use the construction in figure 2.4 for any regular expression by expanding out all shorthand, *e.g.* converting s^+ to ss^*, `[0-9]` to `0|1|2|`$\cdots$`|9` and s? to $s|\epsilon$, *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 doesn't quite follow the

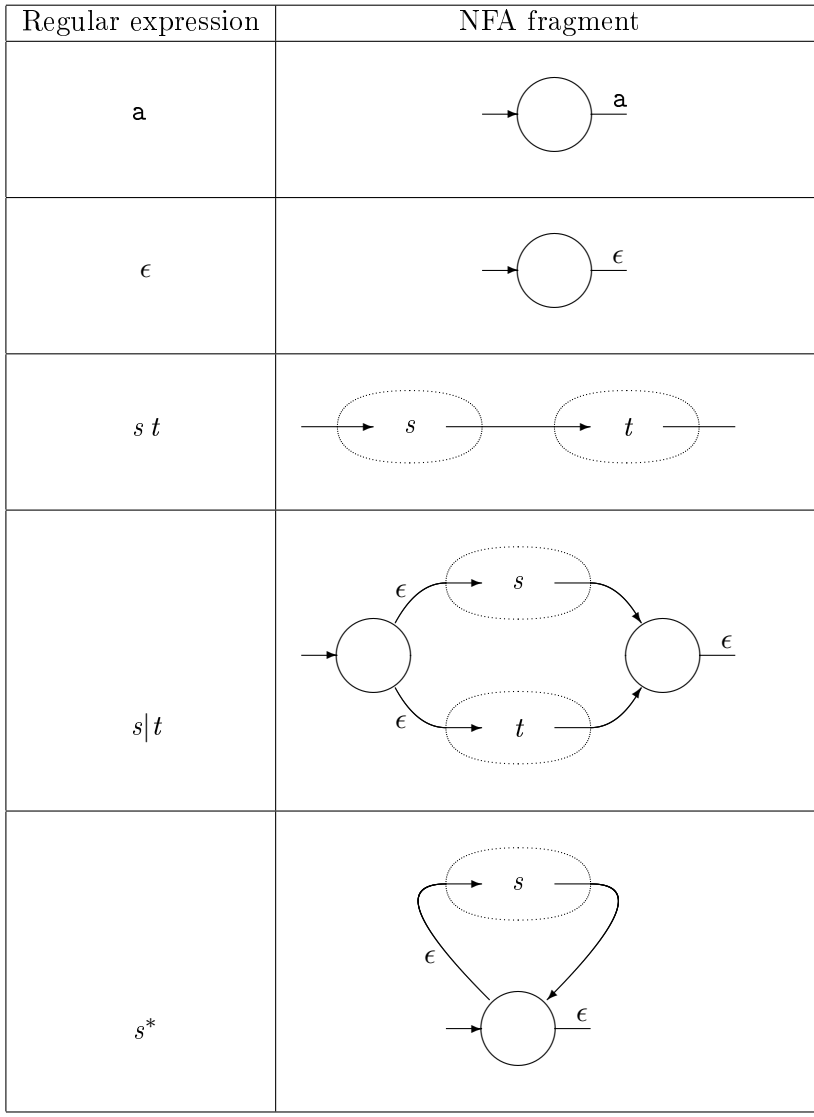


Figure 2.4: Constructing NFA fragments from regular expressions

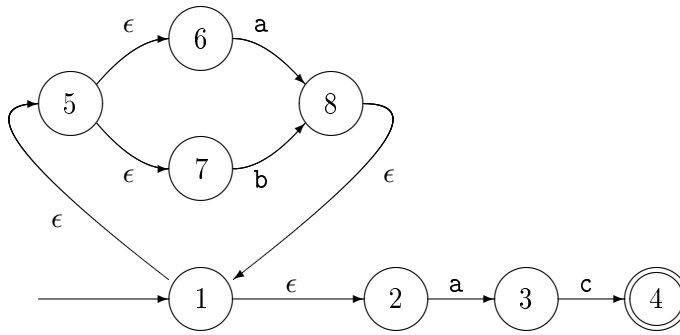


Figure 2.5: NFA for the regular expression (a|b)*ac

formula used in figure 2.4, as it doesn't have two half-transitions. Rather, the line-segment notation is intended to indicate that the NFA fragment for ϵ just connects the half-transitions of the NFA fragments that it is combined with. In the construction for [0-9], the vertical ellipsis is meant to indicate that there is a transition for each of the digits in [0-9]. This construction generalises in the obvious way to other sets of characters, *e.g.,* [a-zA-Z0-9]. We have not shown a special construction for $s?$ as $s|\epsilon$ will do fine if we use the optimised construction for ϵ.

The optimised constructions are shown in figure 2.6. As an example, an NFA for $[0\text{-}9]^+$ is shown in figure 2.7. Note that while this is *optimised*, it is not *optimal*. You can make an NFA for this language using only two states.

2.5 Deterministic finite automata

Nondeterministic automata are, as mentioned earlier, not quite as close to "the machine" as we would like. Hence, we now introduce a more restricted form of finite automaton: The deterministic finite automaton, or DFA for short. DFAs are 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.

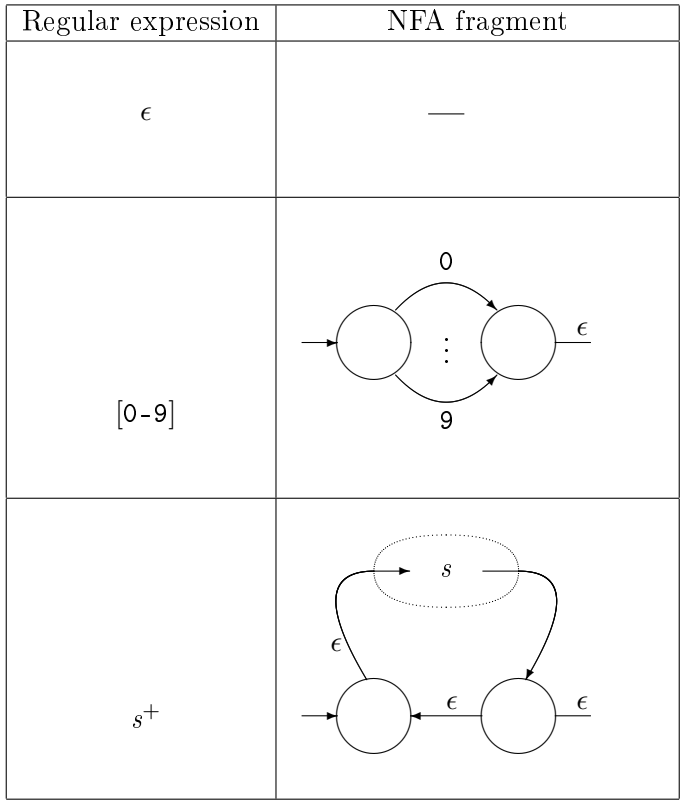


Figure 2.6: Optimised NFA construction for regular expression shorthands

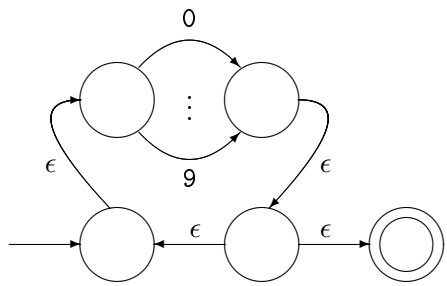


Figure 2.7: Optimised NFA for $[0\text{-}9]^+$

This means that we never have a choice of several next-states: The state and the next input symbol uniquely determines the transition (or lack of same). This is why these automata are called *deterministic*.

The transition relation is now 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 section 2.10). 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.

2.6 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 section 2.4, 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 closing the set of NFA states under epsilon-transitions and then following transitions with input symbols. 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 2.2 *Given a set M of NFA states, we define ϵ-closure(M) to be the least (in terms of the subset relation) solution to the set equation*

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

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.

2.6.1 Solving set equations

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 2.2, ϵ-*closure(M)* is the value we have to find, so we replace this by X and get the equation:

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

and hence

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

This function has a property that is essential to our solution method: If $X \subseteq Y$ then $F(X) \subseteq F(Y)$. We say that F is *monotonic*. Note that $F(X)$ is not ϵ-*closure(X)* and that F depends on M, so a new F is required for each M that we want to find the epsilon-closure of.

There may be several solutions to this equation. For example, if the NFA has a pair of states that connect to each other by epsilon transitions,

adding this pair to a solution that does not already include the pair will create a new solution. The epsilon-closure of M is the *least* solution to the equation (*i.e.*, the smallest set that satistifes 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 isn't 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 figure 2.5. We use a version of F where $M = \{1\}$, so we start by calculating

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

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

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

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

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

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

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

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

We can use this principle to formulate a *work-list algorithm* for finding the least fixed-points for distributive functions. 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 $F(\emptyset) = \{1\}$. The element 1 is unmarked, so we pick this, mark it and calculate $F(\{1\})$ 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.

2.6.2 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 2.3 (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:*

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

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 section 2.6.1.

- 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 figure 2.5 to a DFA.

The initial state in the DFA is ϵ-*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 section 2.6.1).

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

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

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

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

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

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

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

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

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

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:

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

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

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

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

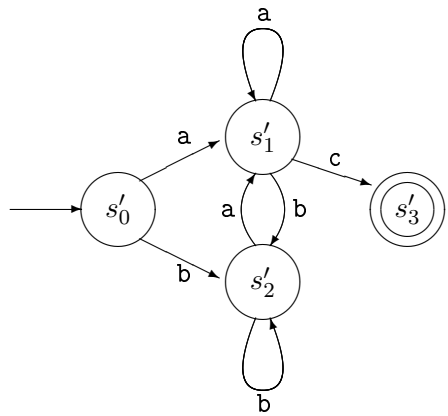


Figure 2.8: DFA constructed from the NFA in figure 2.5

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

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

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

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 2.8 shows the completed DFA.

2.7 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 section 2.10.1. Since we are mainly interested in NFAs that are constructed from regular expressions as in section 2.4, we might ask ourselves if these might not be in a suitably simple class that do not risk exponential-sized DFAs. Alas, this is not the case. Just as we can construct a class of NFAs that expand exponentially, we can construct a class of regular expressions where the smallest equivalent DFAs are exponentially larger. This happens rarely when we use regular expressions or NFAs to describe tokens in programming languages, though.

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

2.8 Minimisation of DFAs

Even though the DFA in figure 2.8 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. 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

[1] If we don't consider the effects of cache-misses *etc.*

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 separate 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 2.4 (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' 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 and any symbol c, $move(s_1, c)$ is in the same group G' as $move(s_2, c)$ or both are undefined.*

1) We start with two groups: F and $S \setminus F$. 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.*

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 2.4 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

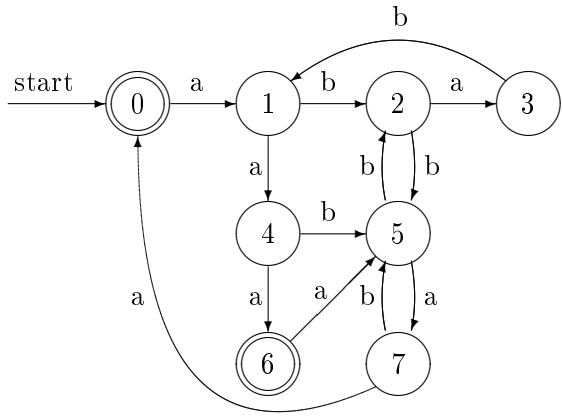


Figure 2.9: Non-minimal DFA

implemented so it uses little more than linear time to do minimisation. We will not here go into further detail but just refer to [3] for the optimal algorithm.

We will, however, note that we can make a slight optimisation to algorithm 2.4: A group that consists of a single state need never be split, so we need never select such in step 2, and we can stop when all unmarked groups are singletons.

2.8.1 Example

As an example of minimisation, take the DFA in figure 2.9.

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

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

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):

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

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 isn't consistent either, so we split again and get

$$\begin{aligned} G_1 &= \{0, 6\} \\ G_4 &= \{3\} \\ G_5 &= \{4, 7\} \\ G_6 &= \{1, 5\} \\ G_7 &= \{2\} \end{aligned}$$

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

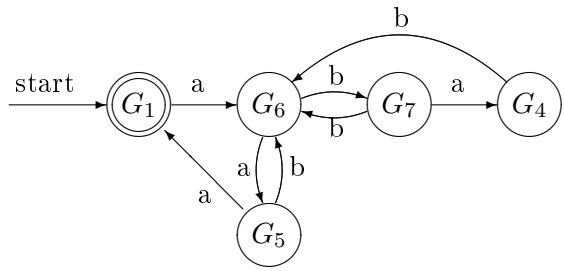


Figure 2.10: Minimal DFA

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 figure 2.9. The minimised DFA is shown in figure 2.10.

2.8.2 Dead states

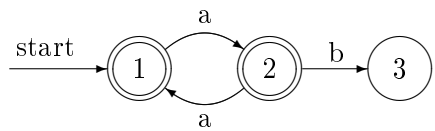
Algorithm 2.4 works under some, as yet, unstated assumptions:

- 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 section 2.4. 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 2.4 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 doesn't 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 2.4 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 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. 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.

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 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 and state 3 has a transition to state 4 on a. 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 minimized automata.

2.9 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). This is the heart of a lexer generator, but not the full story. There are several additional issues, which we address below:

- 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.

We do not wish to scan the input repeatedly, once for every type of token, as this can be quite slow. Hence, we wish to generate a DFA that tests for all the token types simultaneously. This isn't too difficult: 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 and the DFA constructed from it 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. The easiest way to do this is:

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, some of which are accepting states 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 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, figure 2.11 shows an NFA made by combining NFAs for variable names, the keyword `if`, integers and

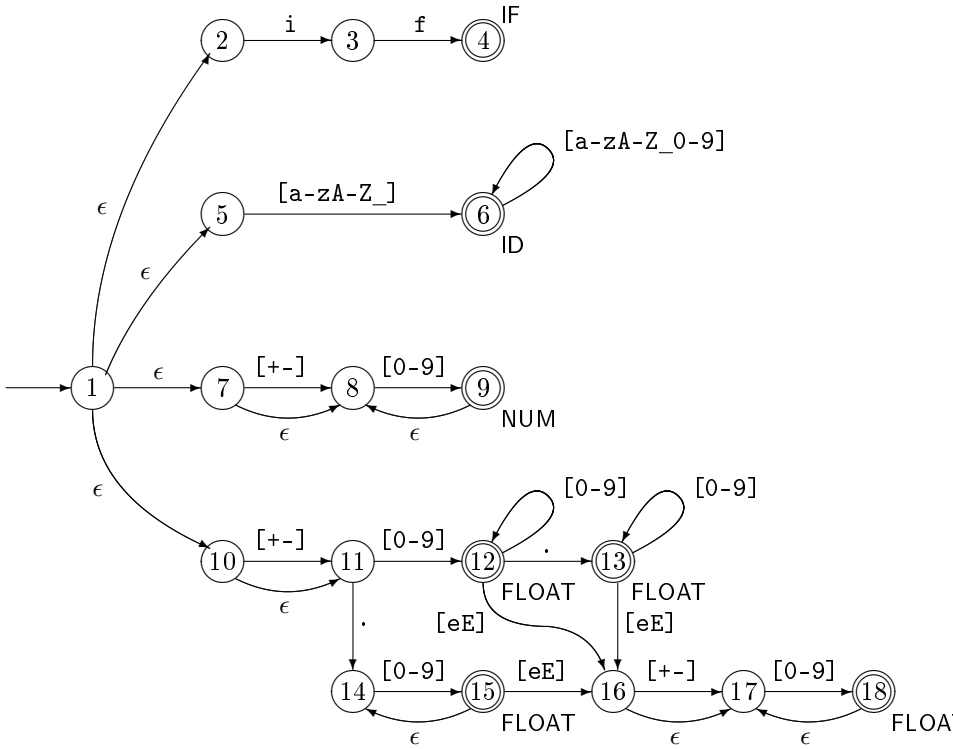


Figure 2.11: Combined NFA for several tokens

floats, as described by the regular expressions in section 2.2.2. The individual NFAs are (simplified versions of) what you get from the method described in section 2.4. 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 figure 2.12. 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:

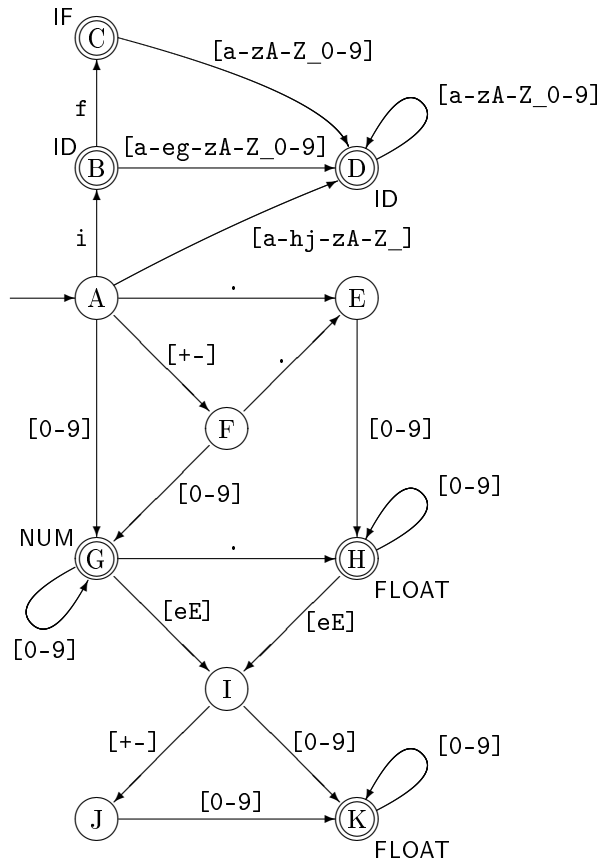


Figure 2.12: Combined DFA for several tokens

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

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

Modern languages like C, Java or 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 it sinitial state) on the remaining input to find the next tokens.

As an example, consider lexing of the string `3e-y` with the DFA in figure 2.12. 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 isn't 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 continuig, 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.

2.9.1 Lexer generators

A lexer generator will typically use a notation for regular expressions similar to the one described in section 2.1, 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 section 2.10), 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 figures 2.11 and 2.12, 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.

2.10 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.

2.10.1 Relative expressive power

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

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

If we are only interested in *if* a language is regular rather than the size of its description, however, it doesn't 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 2.9), 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 2.3) 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 isn't 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`.

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 or 1), using bit-wise or.

This isn't 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 subset.

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.

2.10.2 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 isn't 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 Pascal programs of less than a million pages 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 pages, it isn't 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 2.9).

2.10.3 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 section 2.8.2). 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.

2.11 Further reading

There are many variants of the method shown in section 2.4. 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 [5] and [9].

It is possible to convert a regular expression to a DFA directly without going through an NFA. One such method [26] [5] 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 [12]. 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 [18]. An efficient DFA minimization algorithm can be found in [22].

Lexer generators can be found for most programming languages. For C, the most common are Lex [24] and Flex [34]. 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 [28]. 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.

Exercises

Exercise 2.1

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 $[0\text{-}9]^+$. 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 2.2

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

a) Construct an equivalent NFA using the method in section 2.4.

b) convert this NFA to a DFA using algorithm 2.3.

Exercise 2.3

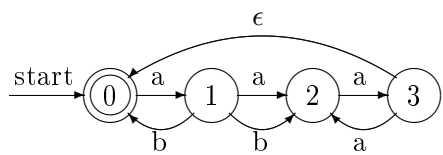
Given the regular expression `((a|b)(a|bb))*`:

a) Construct an equivalent NFA using the method in section 2.4.

b) convert this NFA to a DFA using algorithm 2.3.

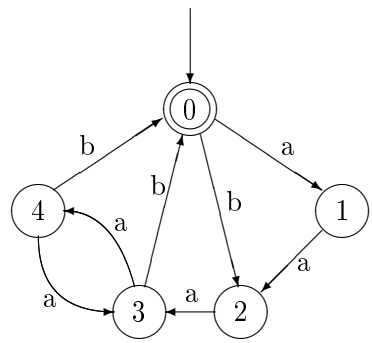
Exercise 2.4

Make a DFA equivalent to the following NFA:



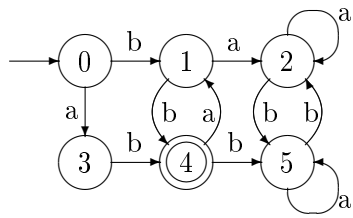
Exercise 2.5

Minimise the following DFA:



Exercise 2.6

Minimise the following DFA:



Exercise 2.7

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 bs (and any number of as).

b) The set of strings where the number of bs is a multiple of 3 (and there can be any number of as).

c) The set of strings where the difference between the number of as and the number of bs is a multiple of 3.

Exercise 2.8

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

Exercise 2.9

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 2.10

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 section 2.2. Hence, the equivalence between DFAs and regular expressions is not complete. To remedy this, a new regular expression ϕ is introduced such that $L(\phi) = \emptyset$.

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

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

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 2.11

Show that regular languages are closed under prefix, suffix, subsequence and reversal, as postulated in section 2.10. 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 2.12

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 2.13

In figures 2.11 and 2.12 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 2.3 must be modified to handle transitions with sets in such a way that the disjointedness requirement for DFAs are ensured.

b) Sketch how algorithm 2.4 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 2.14

As mentioned in section 2.5, 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 UNI-code 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 64KB of memory, which may be a problem.

A possible solution is to split each 16-bit UNI-code 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 2.15

If L is a regular language, so is $L \setminus \{\epsilon\}$, *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 $L \setminus \{\epsilon\}$. 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:

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

where ϕ is the regular expression for the empty language (see exercise 2.10).

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 2.16

If L is a regular language, so is the set of all prefixes of strings in L (see section 2.10.3).

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:

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

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 and $s|t$.

b) Use this definition to find $\mathit{prefixes}(\mathtt{ab}^*\mathtt{c})$.

Chapter 3

Syntax Analysis

3.1 Introduction

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.

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

We shall see two ways of generating such automata. The first of these, LL(1), is relatively simple, but works only for a somewhat restricted class of grammars. The SLR construction, which we present later, is more complex but 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.

3.2 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 as 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

$$\begin{array}{lcl} B & \rightarrow & \\ B & \rightarrow & \mathtt{a}B \end{array}$$

where the first production indicates that the empty string is part of the set B. Compare this grammar with the definition of s^* in figure 2.1.

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 section 2.10, 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

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

The second production ensures that the as and bs are paired symmetrically around the middle of the string, ensuring that 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

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

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.

Form of s_i	Productions for N_i
ϵ	$N_i \rightarrow$
a	$N_i \rightarrow$ a
$s_j s_k$	$N_i \rightarrow N_j N_k$
$s_j \vert s_k$	$N_i \rightarrow N_j$ $N_i \rightarrow N_k$
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.

Figure 3.1: From regular expressions to context free grammars

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

$$\begin{array}{lcl} T & \rightarrow & R \mid \texttt{a}T\texttt{a} \\ R & \rightarrow & \texttt{b} \mid \texttt{b}R \end{array}$$

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

3.2.1 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 figure 3.1. 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 expressions

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

Grammar 3.2: Simple expression grammar

that are part of most programming languages (and also found on electronic calculators). Such expressions can be described by grammar 3.2. Note that, as mentioned in section 2.10, the matching parentheses can't be described by regular expressions, as these can't "count" the number of unmatched opening parentheses at a particular point in the string. However, if we didn't have parentheses in the language, it could be described by the regular expression

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

Even so, the regular description isn't 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 sections 3.3.1 and 3.4.

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:

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 3.2. More nonterminals might be needed to describe a

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \textbf{id} := \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \mathit{Stat}\ ;\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat}\ \textbf{else}\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Stat}
\end{array}
$$

Grammar 3.3: Simple statement grammar

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 3.3, which refers to the *Exp* nonterminal from grammar 3.2.

3.3 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 section 2.6.1 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 $N \rightarrow \gamma$
2. $\alpha \Rightarrow \alpha$
3. $\alpha \Rightarrow \gamma$ if there is a β such that $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$

where α, β and γ are (possibly empty) sequences of grammar symbols (terminals and nonterminals). The first rule states that using a pro-

$$\begin{array}{lcl} T & \rightarrow & R \\ T & \rightarrow & \mathtt{a}T\mathtt{c} \\ R & \rightarrow & \\ R & \rightarrow & R\mathtt{b}R \end{array}$$

Grammar 3.4: Example grammar

duction as a rewrite rule (anywhere in a sequence of grammar symbols) is a derivation step. The second states that the derivation relation is reflexive, *i.e.*, that a sequence derives itself. The third rule describes transitivity, *i.e.*, that a sequence of derivations is in itself a derivation[2].

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

Definition 3.1 *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 3.4 generates the string `aabbbcc` by the derivation shown in figure 3.5. 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 doesn't matter. So, we might as well decide to always rewrite the leftmost nonterminal, as shown in figure 3.6.

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*.

3.3.1 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

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

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

Figure 3.5: Derivation of the string `aabbbcc` using grammar 3.4

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

Figure 3.6: Leftmost derivation of the string `aabbbcc` using grammar 3.4

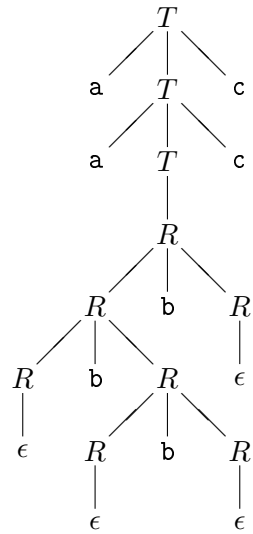


Figure 3.7: Syntax tree for the string `aabbbcc` using grammar 3.4

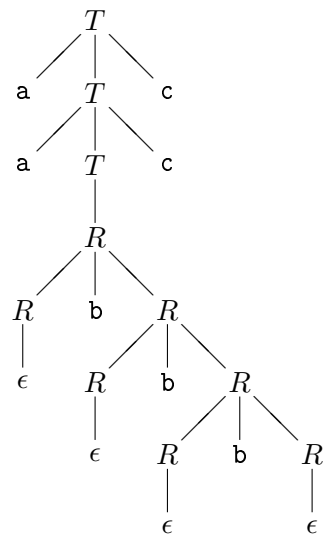


Figure 3.8: Alternative syntax tree for the string `aabbbcc` using grammar 3.4

$$
\begin{array}{lcl}
T & \rightarrow & R \\
T & \rightarrow & \texttt{a}T\texttt{c} \\
R & \rightarrow & \\
R & \rightarrow & \texttt{b}R
\end{array}
$$

Grammar 3.9: Unambiguous version of grammar 3.4

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 figures 3.5 and 3.6 yield the same syntax tree, which is shown in figure 3.7.

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 doesn't 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, figure 3.8 shows an alternative syntax tree for the same string that was derived in figure 3.7.

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 isn't 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 desireable 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 3.4 is shown in figure 3.9.

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 3.2 and 3.4.

We will, in sections 3.11 and 3.13, 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.

3.4 Operator precedence

As mentioned in section 3.2.1, we can describe traditional arithmetic expressions by grammar 3.2. Note that **num** is a terminal that denotes all integer constants and that, here, the parentheses are terminal symbols

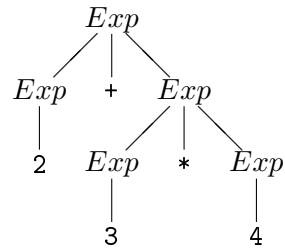


Figure 3.10: Preferred syntax tree for `2+3*4` using grammar 3.2

(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

$$Exp \rightarrow Exp\,\texttt{+}\,Exp$$

which has the form that in section 3.3.1 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 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 figure 3.10, 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 section 3.15. 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 doesn't 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 $2 < 3 < 4$.

3.4.1 Rewriting ambiguous expression grammars

If we have an ambiguous grammar

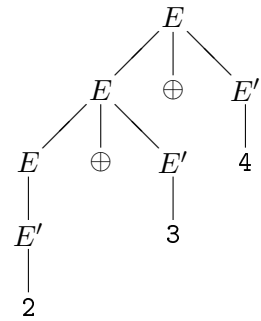
$$\begin{array}{lcl} E & \rightarrow & E \oplus E \\ E & \rightarrow & \textbf{num} \end{array}$$

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:

$$\begin{array}{lcl} E & \rightarrow & E \oplus E' \\ E & \rightarrow & E' \\ E' & \rightarrow & \textbf{num} \end{array}$$

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



We get a slightly more complex syntax tree than in figure 3.10, but not enormously so.

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

$$\begin{array}{lcl} E & \rightarrow & E' \oplus E \\ E & \rightarrow & E' \\ E' & \rightarrow & \textbf{num} \end{array}$$

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

$$\begin{array}{lcl} E & \rightarrow & E' \oplus E' \\ E & \rightarrow & E' \\ E' & \rightarrow & \textbf{num} \end{array}$$

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.

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 -:

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

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

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

Grammar 3.11: Unambiguous expression grammar

$$
\begin{array}{lcl}
E & \rightarrow & E \texttt{ + } E' \\
E & \rightarrow & E' \oplus E \\
E & \rightarrow & E' \\
E' & \rightarrow & \textbf{num}
\end{array}
$$

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

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

We also need to handle operators with different precedences. This is done by using a nonterminal for each precedence level. The idea is that if an expression uses an operator of a certain precedence level, then its subexpressions cannot use operators of lower precedence (unless these are inside parentheses). Hence, the productions for a nonterminal corresponding to a particular precedence level refers only to nonterminals that correspond to the same or higher precedence levels, unless parentheses or similar bracketing constructs disambiguate the use of these. Grammar 3.11 shows how these rules are used to make an unambiguous version of grammar 3.2. Figure 3.12 show the syntax tree for `2+3*4` using this grammar.

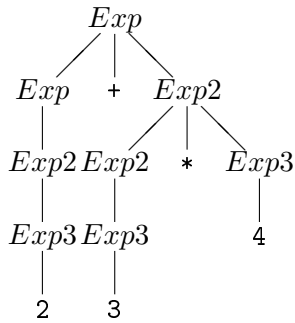


Figure 3.12: Syntax tree for `2+3*4` using grammar 3.11

3.5 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 section 3.4. 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 3.3. The problem is that it isn't 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 section 3.4 can't directly be applied to grammar 3.3, as the productions for conditionals don't 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 3.13. This grammar also

Stat	→	*Stat*2 ; *Stat*
Stat	→	*Stat*2
*Stat*2	→	*Matched*
*Stat*2	→	*Unmatched*
Matched	→	`if` *Exp* `then` *Matched* `else` *Matched*
Matched	→	**id** := *Exp*
Unmatched	→	`if` *Exp* `then` *Matched* `else` *Unmatched*
Unmatched	→	`if` *Exp* `then` *Stat*2

Grammar 3.13: Unambiguous grammar for statements

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 section 3.15.

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.

3.6 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.

3.7 Predictive parsing

If we look at the left-derivation in figure 3.6, 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 figure 3.6, 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

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

starts with an `a`, we obviously can't 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't immediately choose a production based on this. As the grammar (grammar 3.4) is ambiguous, it should not be a surprise that we can't always choose uniquely. If we instead use the unambiguous grammar (grammar 3.9) 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 this kind of predictive parsing.

3.8 *Nullable* and *FIRST*

In simple cases, like the above, all but one of the productions for a nonterminal start with distinct terminals and the remaining production does not start with a terminal. However, the method can be applied also for grammers that don't have this property: Even if several productions start with nonterminals, we can choose among these if the strings these productions can derive begin with symbols from known disjoint sets. 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 3.2 *A symbol c is in FIRST(α) if and only if $\alpha \Rightarrow c\beta$ for some 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 3.3 *A sequence α of grammar symbols is* Nullable *(we write this as Nullable(α)) if and only if $\alpha \Rightarrow \epsilon$.*

A production $N \to \alpha$ is called nullable if *Nullable(α)*. We describe calculation of *Nullable* by case analysis over the possible forms of sequences of grammar symbols:

Algorithm 3.4

$$
\begin{array}{lcl}
\mathit{Nullable}(\epsilon) & = & \mathit{true} \\
\mathit{Nullable}(\mathtt{a}) & = & \mathit{false} \\
\mathit{Nullable}(\alpha\,\beta) & = & \mathit{Nullable}(\alpha) \wedge \mathit{Nullable}(\beta) \\
\mathit{Nullable}(N) & = & \mathit{Nullable}(\alpha_1) \vee \ldots \vee \mathit{Nullable}(\alpha_n), \\
 & & \textit{where the productions for } N \textit{ are} \\
 & & N \to \alpha_1, \quad \ldots \quad , N \to \alpha_n
\end{array}
$$

where 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.

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 section 2.6.1. 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 section 2.6.1, 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 3.9, we get these equations for nonterminals and right-hand sides:

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

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

In a fixed-point calculation, we initially assume that *Nullable* is false for all nonterminals and use this as a basis for calculating *Nullable* for first the right-hand sides and then the nonterminals. We repeat recalculating these until there is no change between two iterations. Figure 3.14 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*:

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*

Figure 3.14: Fixed-point iteration for calculation of *Nullable*

Algorithm 3.5

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

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 initailly assume every set to be empty. For grammar 3.9, we get the following equations:

$$
\begin{array}{lcl}
\mathit{FIRST}(T) & = & \mathit{FIRST}(R) \cup \mathit{FIRST}(\mathtt{a}T\mathtt{c}) \\
\mathit{FIRST}(R) & = & \mathit{FIRST}(\epsilon) \cup \mathit{FIRST}(\mathtt{b}R)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{FIRST}(R) & = & \mathit{FIRST}(R) \\
\mathit{FIRST}(\mathtt{a}T\mathtt{c}) & = & \mathit{FIRST}(\mathtt{a}) \\
\mathit{FIRST}(\epsilon) & = & \emptyset \\
\mathit{FIRST}(\mathtt{b}R) & = & \mathit{FIRST}(\mathtt{b})
\end{array}
$$

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}

Figure 3.15: Fixed-point iteration for calculation of *FIRST*

The fixed-point iteration is shown in figure 3.15.

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 *FIRST*(aTc) to {a}.

3.9 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 section 3.7, 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 must actually do this for any production that is *Nullable*. Hence, at most one production for a nonterminal may be nullable, as otherwise we would not be able to choose deterministically between the two.

We said in section 3.3.1 that our syntax analysis methods will detect ambiguous grammars. However, this isn't true with the method as stated above: We will get unique choice of production even for some ambiguous grammars, including grammar 3.4. 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 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 3.9 to

$$T \rightarrow \texttt{a}T\texttt{b}$$

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`.

Hence, we add to our construction of predictive parsers a test that will reject 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. However, we should extend this to say that we choose a production $N \rightarrow \alpha$ on symbol c if one of the two conditions below are satisfied:

1) $c \in \mathit{FIRST}(\alpha)$.

2) *Nullable(α)* and c can validly follow N in a derivation.

This makes us choose nullable productions more often than before. This, in turn, leads to more cases where we can not choose uniquely, including the example above with the modified grammar 3.9 (since `b` can follow R in valid derivations) and all ambiguous grammars that are not caught by the original method.

3.10 *FOLLOW*

For the purpose of rejecting grammars that are problematical for predictive parsing, we introduce *FOLLOW* sets for nonterminals.

Definition 3.6 *A terminal symbol* `a` *is in FOLLOW(N) if and only if there is a derivation from the start symbol S of the grammar such that $S \Rightarrow \alpha N\texttt{a}\beta$, where α and β are (possibly empty) sequences of grammar symbols.*

In other words, a terminal c is in $FOLLOW(N)$ if c may follow N at some point in a derivation.

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:

$$S' \rightarrow S\$$$

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 $FOLLOW(N)$ 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. A production

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

generates the constraint $FIRST(\beta) \subseteq FOLLOW(N)$, since β, obviously, can follow N. Furthermore, if $Nullable(\beta)$ the production also generates the constraint
$FOLLOW(M) \subseteq FOLLOW(N)$ (note the direction of the inclusion). The reason is that, if 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 the constraint $FOLLOW(A) \subseteq FOLLOW(B)$ by "splitting" after the last 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). Thereafter, we handle the second type of constraints: For each constraint $FOLLOW(M) \subseteq FOLLOW(N)$, we add $FOLLOW(M)$ 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.

   2.2 Let $m = \mathit{FIRST}(\beta)$. Add the constraint $m \subseteq \mathit{FOLLOW}(N)$ to the set of constraints. If β is empty, you can omit the constraint, as it doesn't add anything.

   2.3 If $\mathit{Nullable}(\beta)$, find the nonterminal M at the left-hand side of the production and add the constraint $\mathit{FOLLOW}(M) \subseteq \mathit{FOLLOW}(N)$. If $M = N$, you can omit the constraint, as it doesn't add anything. Note that if β is empty, $\mathit{Nullable}(\beta)$ is true.

3. Solve the constraints using the following steps:

   3.1 Start with empty sets for $\mathit{FOLLOW}(N)$ for all nonterminals N (not including S').

   3.2 For each constraint of the form $m \subseteq \mathit{FOLLOW}(N)$ constructed in step 2.1, add the contents of m to $\mathit{FOLLOW}(N)$.

   3.3 Iterating until a fixed-point is reached, for each constraint of the form $\mathit{FOLLOW}(M) \subseteq \mathit{FOLLOW}(N)$, add the contents of $\mathit{FOLLOW}(M)$ to $\mathit{FOLLOW}(N)$.

We can take grammar 3.4 as an example of this. We first add the production

$$T' \rightarrow T\$$$

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

Production	Constraints
$T' \rightarrow T\$$	$\{\$\} \subseteq \mathit{FOLLOW}(T)$
$T \rightarrow R$	$\mathit{FOLLOW}(T) \subseteq \mathit{FOLLOW}(R)$
$T \rightarrow \mathtt{a}T\mathtt{c}$	$\{\mathtt{c}\} \subseteq \mathit{FOLLOW}(T)$
$R \rightarrow$	
$R \rightarrow R\mathtt{b}R$	$\{\mathtt{b}\} \subseteq \mathit{FOLLOW}(R),\ \mathit{FOLLOW}(R) \subseteq \mathit{FOLLOW}(R)$

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 use the constraints that involve these *FIRST* sets:

$$\begin{array}{rcl} \mathit{FOLLOW}(T) & = & \{\$, \mathtt{c}\} \\ \mathit{FOLLOW}(R) & = & \{\mathtt{b}\} \end{array}$$

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

$$\begin{array}{rcl} \mathit{FOLLOW}(T) & = & \{\$, \mathtt{c}\} \\ \mathit{FOLLOW}(R) & = & \{\$, \mathtt{c}, \mathtt{b}\} \end{array}$$

Which is the final values for the *FOLLOW* sets.

If we return to the question of predictive parsing of grammar 3.4, we see that for the nonterminal R we should choose the empty production on the symbols 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 $\mathtt{b}$), 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.

3.11 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
- *Nullable*(α) and $c \in \mathit{FOLLOW}(N)$.

If we can always choose a production uniquely by using these rules, this is called 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.

3.11.1 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 section 3.11. 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, figure 3.16 shows pseudo-code for a recursive descent parser for grammar 3.9. We have constructed this program by the following process:

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

T' has only one production, so the choice is trivial. However, we have added a check on the next input symbol anyway, so we can report an error if it isn't in *FIRST(T')*. This is shown in the function `parseT'`.

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

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

The 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 figure 3.16 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.

```
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()
```

Figure 3.16: Recursive descent parser for grammar 3.9

3.11.2 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 *FIRST(α)* or if both *Nullable(α)* and a is in *FOLLOW(N)*.

For grammar 3.9 we get the table shown in figure 3.17.

The program that uses this table is shown in figure 3.18. It uses a stack, which at any time (read from top to bottom) contains the part of the current derivation that has not yet been matched to the input. When this eventually becomes empty, the parse is finished. If the stack is non-empty, and the top of the stack contains a terminal, that terminal is matched against the input and popped from the stack. Otherwise, the top of the stack must be a nonterminal, which we cross-index in the table

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

Figure 3.17: LL(1) table for grammar 3.9

```
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)
```

Figure 3.18: Program for table-driven LL(1) parsing

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, figure 3.19 shows the input and stack at each step during parsing of the string `aabbbcc$` using the table in figure 3.17. The top of the stack is to the left.

The program in figure 3.18, like the one in figure 3.16, 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.

3.11.3 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.

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$
$	$

Figure 3.19: Input and stack during table-driven LL(1) parsing

Conflicts may be caused by ambiguous grammars (indeed all ambiguous grammars will cause conflicts) but there are also unambiguous grammars that cause conflicts. An example of this is the unambiguous expression grammar (grammar 3.11). 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 isn't.

3.12 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.

3.12.1 Eliminating left-recursion

As mentioned above, the unambiguous expression grammar (grammar 3.11) is not LL(1). The reason is that all productions in Exp and $Exp2$ have the same $FIRST$ sets. Overlap like this will always happen when there are 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 non-left-recursive productions, *i.e.*,

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

where the β_i do not start with N, we observe that this is equivalent to the regular expression $(\beta_1 \mid \ldots \mid \beta_n)(\alpha_1 \mid \ldots \mid \alpha_m)^*$. We can generate the same set of strings by the grammar

$$\begin{array}{lcl} N & \rightarrow & \beta_1 N' \\ & \vdots & \\ N & \rightarrow & \beta_n N' \\ N' & \rightarrow & \alpha_1 N' \\ & \vdots & \\ N' & \rightarrow & \alpha_m N' \\ \\ N' & \rightarrow & \end{array}$$

where N' is a new nonterminal.

Note that, since the β_i do not start with N, there is no direct left-recursion in this grammar. There may, however, still be instances of indirect left-recursion. We will briefly look at indirect left-recursion in section 3.12.1.

Rewriting the grammar like above will change 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 describe. We will return to this in section 3.16.

As an example of left-recursion removal, we take the unambiguous expression grammar 3.11. This has left recursion in both Exp and $Exp2$, so we apply the transformation to both of these to obtain grammar 3.20. The resulting grammar 3.20 is now LL(1).

Indirect left-recursion

The transformation shown in section 3.12.1 only serves in the simple case where there is no *indirect left-recursion*. Indirect left-recursion can have several faces:

1. There are productions

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

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp2\ Exp' \\
Exp' & \rightarrow & \texttt{+}\ Exp2\ Exp' \\
Exp' & \rightarrow & \texttt{-}\ Exp2\ Exp' \\
Exp' & \rightarrow & \\
Exp2 & \rightarrow & Exp3\ Exp2' \\
Exp2' & \rightarrow & \texttt{*}\ Exp3\ Exp2' \\
Exp2' & \rightarrow & \texttt{/}\ Exp3\ Exp2' \\
Exp2' & \rightarrow & \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & \texttt{(}\ Exp\ \texttt{)}
\end{array}
$$

Grammar 3.20: Removing left-recursion from grammar 3.11

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 practise most cases of left-recursion are direct left-recursion. Details can be found in [4].

3.12.2 Left-factorisation

If two productions for the same nonterminal begin with the same sequence of symbols, they obviously have overlapping *FIRST* sets. As an example, in grammar 3.3 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. Seeleft-recursion-elimination grammar 3.21. In this grammar[3], we can uniquely choose one of the productions for *Stat* based on one input token.

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

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \textbf{id} := \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Stat}\ \mathit{Elsepart} \\
 & & \\
\mathit{Elsepart} & \rightarrow & \texttt{else}\ \mathit{Stat} \\
\mathit{Elsepart} & \rightarrow &
\end{array}
$$

Grammar 3.21: Left-factorised grammar for conditionals

For most grammars, combining productions with common prefix will solve the problem. However, in this particular example the grammar still isn't LL(1): We can't 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 shouldn't be a surprise to us, since, after all, the grammar is ambiguous and ambiguous grammars can't be LL(1). The equivalent unambiguous grammar (grammar 3.13) can't easily be rewritten to a form suitable for LL(1), so in practice grammar 3.21 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`. 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.

3.12.3 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.
5. Calculate *FIRST* for every production and *FOLLOW* for every nonterminal.
6. For nonterminal N and input symbol c, choose production $N \rightarrow \alpha$ when:

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

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

3.13 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.

A class of bottom-up methods for parsing called *LR parsers* exist which accept a much larger class of grammars (though still not all grammars). The main advantage of LR parsing is that less rewriting is required to get a grammar in acceptable form, but there are also languages for which there exist LR-acceptable grammars but no LL(1) grammars. Furthermore, as we shall see in section 3.15, LR parsers allow external declaration of operator precedences for resolving ambiguity instead of requiring the grammar itself to be unambiguous.

We will look at a simple form of LR-parsing called SLR parsing. While most parser generators use a somewhat more complex method called LALR(1) parsing, we limit the discussion to SLR for the following reasons:

- It is simpler.
- In practice, LALR(1) handles few grammars that are not also handled by SLR.
- When a grammar is in the SLR class, the parse-tables produced by SLR are identical to those produced by LALR(1).
- Understanding of SLR principles is sufficient to know how a grammar should be rewritten when a LALR(1) parser generator rejects it.

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 right-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: On the stack, a number of symbols that are identical to the right-hand side of a production are replaced by the left-hand side of that 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.

When all of the input is read, the stack will have a single element, which will be the start symbol of the grammar.

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, it will cause reduction by a production that is determined by the state and the next input symbol. If the DFA is not in an accepting state, it will cause a shift. Hence, at every step, the action can be determined by letting the DFA read the stack from bottom to top.

Letting the DFA read the entire stack at every action is not very efficient, so, instead, we keep track of the DFA state every time we push an element on the stack, storing the state as part of the stack element.

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.

If the DFA indicated a reduce, we pop the right-hand side of the production off the stack. We then read the DFA state from the new stack top. When we push the nonterminal that is the left-hand side of the production, we make a transition from this DFA state on the nonterminal.

With these optimisations, the DFA only has to inspect a terminal or nonterminal at the time it is pushed on the stack. At all other times, it just need to read the DFA state that is stored with the stack element. Hence, we can forget about what the actual symbols are as soon as the DFA has made the transition. There is, thus, no reason to keep the symbols on the stack, so we let a stack element just contain the DFA 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, 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 found when a state is cross-indexed with a terminal symbol. *Go* actions are found when a state is cross-indexed with a nonterminal. *Go* actions are only used immediately after a reduce, but we can't put them next to the *reduce* actions in the table, as the destination state of a *go* depends on the state on 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.

An example SLR table is shown in figure 3.22. The table has been produced from grammar 3.9 by the method shown below in section 3.14. 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 figure 3.23. As written, the 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 doing a *reduce* action, a new (partial) syntax tree is built by using the nonterminal from the reduced production as root and the syntax trees attached to the popped-off stack elements as children. The new tree is then attached to the stack element that is pushed.

Figure 3.24 shows an example of parsing the string `aabbbcc` using the table in figure 3.22. The stack grows from left to right.

3.14 Constructing SLR parse tables

An SLR parse table has as its core a DFA. Constructing this DFA from the grammar is not much different from constructing a DFA from a regular expression as shown in chapter 2: We first construct an NFA using techniques similar to those in section 2.4 and then convert this into a DFA using the construction shown in section 2.5.

	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		

Figure 3.22: SLR table for grammar 3.9

```
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
```

Figure 3.23: Algorithm for 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

Figure 3.24: Example SLR parsing

$$
\begin{array}{lrcl}
0: & T' & \rightarrow & T \\
1: & T & \rightarrow & R \\
2: & T & \rightarrow & \mathtt{a}T\mathtt{c} \\
3: & R & \rightarrow & \\
4: & R & \rightarrow & \mathtt{b}R
\end{array}
$$

Grammar 3.25: Example grammar for SLR-table construction

However, before we do this, we extend the grammar with a new starting production. Doing this to grammar 3.9 yields grammar 3.25.

The next step is to make an NFA for each production. This is done exactly like in section 2.4, treating both terminals and nonterminals as alphabet symbols. The accepting state of each NFA is labelled with the number of the corresponding production. The result is shown in figure 3.26. Note that we have used the optimised construction for ϵ (the empty production) as shown in figure 2.6.

The NFAs in figure 3.26 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

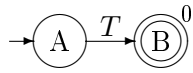
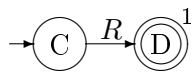
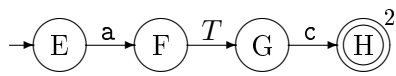
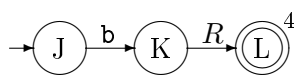
Production	NFA
$T' \to T$	
$T \to R$	
$T \to \texttt{a}T\texttt{c}$	
$R \to$	
$R \to \texttt{b}R$	

Figure 3.26: NFAs for the productions in grammar 3.25

state	epsilon-transitions
A	C, E
C	I, J
F	C, E
K	I, J

Figure 3.27: Epsilon-transitions added to figure 3.26

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 do this, the symbols that form the right-hand side must be on the stack.

To achieve this we must, whenever a transition by a nonterminal is possible, also allow transitions on the symbols on the right-hand side of a production for that nonterminal so these eventually can be reduced to the nonterminal. We do this by adding epsilon-transitions to the NFAs in figure 3.26: Whenever there is a transition from state s to state t on a nonterminal N, we add epsilon-transitions from s to the initial states of all the NFAs for productions with N on the left-hand side. Adding these graphically to figure 3.26 would make a very cluttered picture, so instead we simply note the transitions in a table, shown in figure 3.27.

Together with these epsilon-transitions, the NFAs in figure 3.26 form

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

Figure 3.28: SLR DFA for grammar 3.9

a single, combined NFA. This NFA has the starting state A (the starting state of the NFA for the added start production) and an accepting state for each production in the grammar. We must now convert this NFA into a DFA using the subset construction shown in section 2.5. 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 figure 3.22, except that no *reduce* or *accept* actions are present yet. Figure 3.28 shows the DFA constructed from the NFA made by adding epsilon-transitions in 3.27 to figure 3.26. The set of NFA states that forms each DFA state is shown in the second column of the table in figure 3.28. We will need these below for adding *reduce* and *accept* actions, but once this is done we will not need then anymore, and we can remove then from the final table.

To add *reduce* and *accept* actions, we first need to compute the *FOLLOW* sets for each nonterminal, as described in section 3.10. For purpose of calculating *FOLLOW*, we add yet another extra start production: $T'' \rightarrow T'\$$, to handle end-of-text conditions as described in section 3.10. 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)*. Reduc-

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

In figure 3.28, 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 the same way, we add reduce actions to state 3, 4, 6 and 7. The result is shown in figure 3.22.

Figure 3.29 summarises the SLR construction.

1. Add the production $S' \to S$, where S is the start symbol of the grammar.
2. Make an NFA for the right-hand side of each production.
3. For each state s that 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 at transitions on terminals and *go* actions on transitions on nonterminals.
6. Calculate *FOLLOW* for each nonterminal. For this purpose, we add one more start production: $S'' \to S'\$$.
7. When a DFA state contains an NFA state that accepts the right-hand side of the production numbered p, add *reduce* p at all symbols in *FOLLOW(N)*, where N is the nonterminal on the left of production p. If production p is the production added in step 1, the action is *accept* instead of *reduce* p.

Figure 3.29: Summary of SLR parse-table construction

3.14.1 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 sections 3.4 and 3.5. Alternatively, operator precedence declarations may be used to disambiguate an ambiguous grammar, as we shall see in section 3.15.

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 isn't 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 isn't a problem for SLR-parsers, as it is for LL(1)-parsers.

3.15 Using precedence rules in LR parse tables

We saw in section 3.12.2, 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 3.2 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 section 3.4.1.
- 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 3.2 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.

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 `+` 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 we want multiplication to bind stronger than addition, we, again, prefer reduction over shifting.
3) In expressions of the form `a+b*c`, we, as before, want multiplication to group 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 a left-associative operator conflicts with itself and is likewise handled 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 section 3.4.1) 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 (section 3.5) can also be eliminated using precedence rules: Giving `else` a higher precedence than `then` or giving them the same precedence and making them right-associative will handle the problem, as either of these will make the parser shift on `else` instead of reducing $Stat \rightarrow$ `if` Exp `then` $Stat$ when this is followed by `else`.

Not all conflicts should be eliminated by precedence rules. Excessive use of precedence rules may cause the parser to accept only a subset

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

of the intended language (*i.e.*, if a necessary action is eliminated by a precedence rule). So, unless you know what you are doing, you should limit the use of precedence declarations to operators in expressions.

3.16 Using LR-parser generators

Most LR-parser generators use an extended version of the SLR construction called LALR(1). In practice, however, there is little difference between these, so a LALR(1) parser generator can be used with knowledge of SLR only.

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).

3.16.1 Declarations and actions

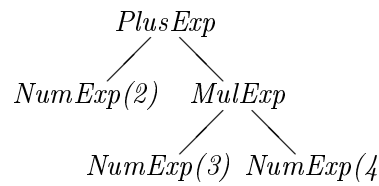
Each nonterminal and terminal is declared and associated with a data-type. For a terminal, the data-type is used to hold the values that are associated with the tokens that come from the lexer, *e.g.*, the values of numbers or names of identifiers. For a nonterminal, the type is used for the values that 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 reduced production 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.

3.16.2 Abstract syntax

The syntax trees described in section 3.3.1 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 figure 3.12 may be represented by the following abstract syntax tree:



Here the names *PlusExp*, *MulExp* and *NumExp* may be constructors in a data-type or they may be elements from an enumerated type used as tags in a union-type. 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 figure refexpression-tree2 that holds the symbol +. Likewise, the sequence of nodes *Exp*, *Exp*2, *Exp*3, 2 at the left of figure refexpression-tree2 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.

A compiler designer has much freedom in the choice of abstract syntax. Some use abstract syntax that retain alls 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, skipping parenthesis and other (for this purpose) irrelevant structure.

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 return value of the action.

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 non-trivially from the concrete syntax tree.

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

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

gets transformed by left-recursion elimination into

$$\begin{array}{lcl} E & \rightarrow & \mathbf{num}\, E' \\ E' & \rightarrow & +\,\mathbf{num}\, E' \\ E' & \rightarrow & \end{array}$$

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.

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

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:

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

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

$$\begin{array}{lcll} E & \rightarrow & E + \mathbf{num} & \{\ \texttt{\$0 = PlusExp(\$1,NumExp(\$3))}\ \} \\ E & \rightarrow & \mathbf{num} & \{\ \texttt{\$0 = NumExp(\$1)}\ \} \end{array}$$

In this setting, `NumExp` and `PlusExp` aren't 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

$$\begin{array}{lcll} E & \rightarrow & \mathbf{num}\ E' & \{\ \texttt{\$2->hole = NumExp(\$1);} \\ & & & \quad \texttt{\$0 = \$2.tree}\ \} \\ E' & \rightarrow & +\,\mathbf{num}\ E' & \{\ \texttt{\$0.hole = makeHole();} \\ & & & \quad \texttt{\$3->hole = PlusExp(\$0.hole,NumExp(\$2));} \\ & & & \quad \texttt{\$0.tree = \$3.tree}\ \} \\ E' & \rightarrow & & \{\ \texttt{\$0.hole = makeHole();} \\ & & & \quad \texttt{\$0.tree = \$0.hole}\ \} \end{array}$$

This may look bad, but when using LR-parser generators, left-recursion removal is rarely needed, and parser generators based on LL(1) often do left-recursion removal automatically and transform the actions appropriately.

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.

3.16.3 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

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 .`

Figure 3.30: Textual representation of NFA states

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 figure 3.26, these would be written as shown in figure 3.30. Note that a '.' is used to indicate the position of the state in the production. State 4 of the table in figure 3.28 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 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, but see also exercise 3.20.

3.17 Properties of context-free languages

In section 2.10, 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 con-

sisting 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.

3.18 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 [13], 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 computer, so these represent all computable languages.

Context-free grammars are actually too weak to describe natural languages, but were adopted for defining the Algol60 programming language [15]. Since then, variants of this notation has been used for defining or describing almost all programming languages.

Some languages have been designed with specific parsing methods in mind: Pascal [19] has been designed for LL(1) parsing while C [23] was originally designed to fit LALR(1) parsing, but this property was lost in subsequent versions of the language.

Most parser generators are based on LALR(1) parsing, but a few use LL(1) parsing. An example of this is ANTLR (`http://www.antlr.org/`).

"The Dragon Book" [4] tells more about parsing methods than the present book.

Several textbooks exist that describe properties of context-free languages, e.g., [18].

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 handling arbitrary precedences of infix, prefix and postfix operators is presented in [1].

Exercises

Exercise 3.1

Figures 3.7 and 3.8 show two different syntax trees for the string `aabbbcc` using grammar 3.4. Draw a third, different syntax tree for `aabbbcc` using the same grammar and show the left-derivation that corresponds to this syntax tree.

Exercise 3.2

Draw the syntax tree for the string `aabbbcc` using grammar 3.9.

Exercise 3.3

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 3.4

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 3.5

We extend the language of balanced parentheses from exercise 3.3 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 3.6

Show that the grammar

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

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 3.7

In grammar 3.2, 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 section 3.4.1. Show the syntax tree for $2 : 3 < 4 + 5 : 6 * 7$ using the unambiguous grammar.

Exercise 3.8

Extend grammar 3.13 with the productions

$$\begin{array}{lcl} \mathit{Exp} & \rightarrow & \textbf{id} \\ \mathit{Matched} & \rightarrow & \end{array}$$

then calculate *Nullable* and *FIRST* for every production in the grammar.

Add an extra start production as described in section 3.10 and calculate *FOLLOW* for every nonterminal in the grammar.

Exercise 3.9

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

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

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

Exercise 3.10

Eliminate left-recursion from grammar 3.2.

Exercise 3.11

Calculate *Nullable* and *FIRST* for every production in grammar 3.20.

Exercise 3.12

Add a new start production $Exp' \rightarrow Exp\,\$$ to the grammar produced in exercise 3.10 and calculate *FOLLOW* for all nonterminals in the resulting grammar.

Exercise 3.13

Make a LL(1) parser-table for the grammar produced in exercise 3.12.

Exercise 3.14

Consider the following grammar for postfix expressions:

$$\begin{array}{lcl} E & \rightarrow & E\,E+ \\ E & \rightarrow & E\,E* \\ E & \rightarrow & \textbf{num} \end{array}$$

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 3.15

Extend grammar 3.11 with a new start production as shown in section 3.14 and calculate *FOLLOW* for every nonterminal. Remember to add an extra start production for the purpose of calculating *FOLLOW* as described in section 3.10.

Exercise 3.16

Make NFAs (as in figure 3.26) for the productions in grammar 3.11 (after extending it as shown in section 3.14) and show the epsilon-transitions as in figure 3.27. Convert the combined NFA into an SLR DFA like the one in figure 3.28. Finally, add reduce and accept actions based on the *FOLLOW* sets calculated in exercise 3.15.

Exercise 3.17

Extend grammar 3.2 with a new start production as shown in section 3.14 and calculate *FOLLOW* for every nonterminal. Remember to add an extra start production for the purpose of calculating *FOLLOW* as described in section 3.10.

Exercise 3.18

Make NFAs (as in figure 3.26) for the productions in grammar 3.2 (after extending it as shown in section 3.14) and show the epsilon-transitions as in figure 3.27. Convert the combined NFA into an SLR DFA like the one in figure 3.28. Add reduce actions based on the *FOLLOW* sets calculated in exercise 3.17. Eliminate the conflicts in the table by using operator precedence rules as described in section 3.15. Compare the size of the table to that from exercise 3.16.

Exercise 3.19

Consider the grammar

$$\begin{array}{lcl} T & \rightarrow & T \texttt{ -> } T \\ T & \rightarrow & T * T \\ T & \rightarrow & \textbf{int} \end{array}$$

where `->` is considered a single terminal symbol.

a) Add a new start production as shown in section 3.14.

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 3.20

In section 3.16.3 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 3.21

Consider the grammar

$$\begin{array}{lcl} A & \rightarrow & a\ A\ a \\ A & \rightarrow & b\ A\ b \\ A & \rightarrow & \end{array}$$

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 3.22

The following ambiguous grammar describes boolean expressions:

$$
\begin{array}{lcl}
B & \rightarrow & \textbf{true} \\
B & \rightarrow & \textbf{false} \\
B & \rightarrow & B \vee B \\
B & \rightarrow & B \wedge B \\
B & \rightarrow & \neg B
\end{array}
$$

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 generates only true boolean expressions. Hint: Use the answer from question a) and add an additional nonterminal for false boolean expressions.

Chapter 4

Symbol Tables

4.1 Introduction

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.

Scoping based in this way on the structure of the syntax tree 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).

4.2 Symbol tables

A symbol table is a table that binds names to objects. 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 an object. 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 object 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.

4.2.1 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 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, 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 a stack. When an update is made, the old binding of a name that is overwritten is recorded (pushed) on the stack. When a new scope is entered, a marker is pushed on the stack. When the scope is exited, the bindings on the stack (down to the marker) are used to reestablish the old symbol table. The bindings and the marker are popped off the stack in the process, returning the 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.

4.2.2 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*). For example, when a new element is added to a list or an element is taken off the head 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 object, 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.

binding: A new binding (name/object pair) is added (cons'ed) to the front of the list.

lookup: The list is searched until a matching name is found. The object 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 type that can hold both objects and error-indications, *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. Entering and exiting a scope is done by binding a symbol table to a name before entering a new scope and then referring to this name again after the scope is exited.

As new bindings are added to the front of the list, these will automatically take precedence over old bindings as the list is searched from the front to the back.

Another functional approach to symbol tables is using functions: A symbol table is quite naturally seen as a function from names to objects. 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.

binding: Adding a binding of the name `n` to the object `o` 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 object `o`. Otherwise, t' calls t with `n1` as argument and returns the result that this call yields.

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.

4.2.3 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.

binding: A new binding (name/object pair) is pushed on top of the stack.

lookup: The stack is searched top-to-bottom until a matching name is found. The object paired with the name is then returned. If the bottom of the stack is reached, we instead return an error-indication.

enter: The top-of-stack pointer is remembered.

exit: The old top-of-stack pointer is recalled and becomes the current.

This is not quite a persistent data structure, as leaving a scope will destroy its symbol table. For most languages, this won't matter, as a scope isn't needed again after it is exited. If this is not the case, a real persistent symbol table must be used, or the needed parts of the symbol table must be stored for later retrieval before exiting a scope.

4.2.4 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 stack to record all updates to the table such that 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.

4.2.5 Shared or separate name spaces

In some languages (like C) 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 other. In other languages (*e.g.* Pascal 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, excep-

tions, constructors, classes, field selectors *etc.* Sharing can exist between any subsets of these name spaces, and which name spaces are shared is language-dependent.

Separate name spaces are easily implemented using a symbol table per name space, whereas shared name spaces naturally share a single symbol table. However, it is sometimes convenient to use a single symbol table even if there are separate name spaces. This can be done fairly easily by adding name-space indicators to the names. A name-space indicator can be a textual prefix to the name or it may be a tag that is paired with the name. In either case, a lookup in the symbol table must match both name and name-space indicator of the symbol that is looked up with the entry in the table.

4.3 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 [32].

Exercises

Exercise 4.1

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 4.2

Implement, in a programming language of your choice, data structures and operations (empty, binding, lookup, enter and exit) for simple symbol tables.

Exercise 4.3

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.

Chapter 5

Type Checking

5.1 Introduction

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 translation. Often, the translator may exploit or depend on type information, which makes it natural to combine calculation of types with the actual translation. We will here, for the sake of exposition, assume that a separate phase is used for type checking and related checks, and similarly assume that any information gathered by this phase is available in subsequent phases.

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 *syntactical 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.

5.3 A small example language

We will use a small (somewhat contrived) language to show the principles of type checking. The language is a first-order functional language with recursive definitions. The syntax is given in grammar 5.1. The shown grammar is clearly ambiguous, but that doesn't matter since we operate on the abstract syntax, where such ambiguities have been resolved.

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.

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Funs} \\
 & & \\
\mathit{Funs} & \rightarrow & \mathit{Fun} \\
\mathit{Funs} & \rightarrow & \mathit{Fun}\ \mathit{Funs} \\
 & & \\
\mathit{Fun} & \rightarrow & \mathit{TypeId}\ (\ \mathit{TypeIds}\ )\ \texttt{=}\ \mathit{Exp} \\
 & & \\
\mathit{TypeId} & \rightarrow & \texttt{int}\ \mathbf{id} \\
\mathit{TypeId} & \rightarrow & \texttt{bool}\ \mathbf{id} \\
 & & \\
\mathit{TypeIds} & \rightarrow & \mathit{TypeId} \\
\mathit{TypeIds} & \rightarrow & \mathit{TypeId}\ \texttt{,}\ \mathit{TypeIds} \\
 & & \\
\mathit{Exp} & \rightarrow & \mathbf{num} \\
\mathit{Exp} & \rightarrow & \mathbf{id} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \texttt{=}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathbf{id}\ (\ \mathit{Exps}\ ) \\
\mathit{Exp} & \rightarrow & \texttt{let}\ \mathbf{id}\ \texttt{=}\ \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \\
 & & \\
\mathit{Exps} & \rightarrow & \mathit{Exp} \\
\mathit{Exps} & \rightarrow & \mathit{Exp}\ \texttt{,}\ \mathit{Exps}
\end{array}
$$

Grammar 5.1: Example language for type checking

5.4 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.

5.5 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 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 *name*, that extracts the name of the identifier. Similarly, **num** has a function *value*, that returns the value of the number. The latter is not required for type checking, though, but we will use it in chapter 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 figure 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. After reporting a type error, the type checker can make a guess at what the type should have been and return this guess, allowing the type checking to continue. 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

$Check_{Exp}(Exp, vtable, ftable)$ = `case` Exp `of`	
num	`int`
id	$t = lookup(vtable, name(\mathbf{id}))$ *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 ($Exps$)	$t = lookup(ftable, name(\mathbf{id}))$ *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	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, name(\mathbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$

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

Figure 5.2: Type checking of expressions

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.
- At a function call, the function name is looked up in the function environment to find the number and types of the arguments as well as the return type. The number of arguments to the call must coincide with the expected number and their types must match the declared types. The resulting type is the return-type of the function. If the function name isn't 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 figure 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.

5.6 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 figure 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*.

5.7 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. Since all functions are mutually recursive, each of these must be type checked using a symbol table where all functions are bound to their type. This requires two passes over the list of functions: One to build the symbol table and one to check the function definitions using this table. Hence, we need two functions operating over $Funs$ and two functions operating over Fun. We have already seen one of the latter, $Check_{Fun}$. The other, Get_{Fun}, returns the pair of the function's declared name and type, which consists of the types of the arguments and the type of the result. It uses an auxiliary function Get_{Types} to find the types of the arguments. The two functions for the syntactic category $Funs$ 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 figure 5.4.

This completes type checking of our small example language.

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

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

Figure 5.3: Type checking a function declaration

$Check_{Program}(Program) =$ `case` $Program$ `of`	
$Funs$	$ftable = Get_{Funs}(Funs)$ $Check_{Funs}(Funs, ftable)$

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

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

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

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

Figure 5.4: Type checking a program

5.8 Advanced type checking

Our example language is very simple and obviously doesn't 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 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.

5.9 Further reading

Overloading of operators and functions is described in section 6.5 of [5]. 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 [27].

Exercises

Exercise 5.1

Add the productions

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

to grammar 5.1. 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 type checking functions in figures 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 an `if-then-else` expression 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 5.2

The type check function in figure 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 figure 5.2 to use the `unknown` type as described above.

Exercise 5.3

We look at a simple language with an exception mechanism:

$$\begin{array}{lcl} S & \rightarrow & \texttt{throw}\ \textbf{id} \\ S & \rightarrow & S\ \texttt{catch}\ \textbf{id} \Rightarrow S \\ S & \rightarrow & S\ \texttt{or}\ S \\ S & \rightarrow & \texttt{other} \end{array}$$

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 don't 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.

Chapter 6

Intermediate-Code Generation

6.1 Introduction

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 doesn't 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 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 is needed for each machine, making the total equal to the sum of the number of languages and the number of machines.

If an interpreter for the intermediate language is written in a language for which there already exist compilers on the target machines, the interpreter can be compiled on each of these. 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.
- 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 a the back-end of the compiler.

6.2 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 doesn't 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-

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & [\ \mathit{Instructions}\ ] \\
 & & \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction} \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction}\ ,\ \mathit{Instructions} \\
 & & \\
\mathit{Instruction} & \rightarrow & \texttt{LABEL}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \textbf{unop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \textbf{id}\ \textbf{binop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := M[\mathit{Atom}] \\
\mathit{Instruction} & \rightarrow & M[\mathit{Atom}] := \textbf{id} \\
\mathit{Instruction} & \rightarrow & \texttt{GOTO}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \texttt{IF}\ \textbf{id}\ \textbf{relop}\ \mathit{Atom}\ \texttt{THEN}\ \textbf{labelid}\ \texttt{ELSE}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \texttt{CALL}\ \textbf{functionid}(\mathit{Args}) \\
 & & \\
\mathit{Atom} & \rightarrow & \textbf{id} \\
\mathit{Atom} & \rightarrow & \textbf{num} \\
 & & \\
\mathit{Args} & \rightarrow & \textbf{id} \\
\mathit{Args} & \rightarrow & \textbf{id}\ ,\ \mathit{Args}
\end{array}
$$

Grammar 6.1: The intermediate language

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 chapter 7.

6.3 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 chapter 9, so a "program" in our intermediate language will, for the time being, correspond to the body of a function or procedure in a real program. For the same reason, function calls are initially treated as primitive operations in the intermediate language.

The grammar for the intermediate language is shown in grammar 6.1. A program is a sequence of instructions. The instructions are:

- A label. This has no effect but serves only to mark the position in the program as a target for jumps.
- An assignment of an atomic expression (constant or variable) to a variable.
- 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.

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.

We assume that all values are integers. Adding floating-point numbers and other primitive types isn't difficult, though.

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

Grammar 6.2: A simple expression language

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 *opname*.

When writing a compiler, we must decide what needs to be done at compile-time and what needs to be done at run-time. Ideally, as much as possible should be done at compile-time, but some things need to be postponed until run-time, as they need the actual values of variables, *etc.*, which aren't 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 make this clear: The code that is written between the square brackets is executed at run-time, the rest is done at compile-time.

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 in one. We use a function *newvar* to generate new variables in the intermediate language. Whenever *newvar* is called, it returns a previously unused variable name.

We will describe translation of expressions by a translation function using a notation similar to the notation we used for type-checking functions in chapter 5.

Some attributes for the translation function are obvious: It must return the code as a synthesised attribute. Furthermore, it must translate 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 sub-expressions. This can be done in two ways:

1) The location of the values of a sub-expression can be passed up as a synthesised attribute to the parent expression, which decides on a position for its own value.

2) The parent expression can decide where it wants to find the values of its sub-expressions and pass this information down to these as inherited attributes.

Neither of these is obviously superior to the other. Method 1 has a slight advantage when generating code for a variable access, as it doesn't have to generate any code, but can simply return the name of the variable that holds the value. This, however, only works under the assumption that the variable isn't updated before the value is used by the parent expression. If expressions can have side effects, this isn't always the case, as the C expression "`x+(x=3)`" shows. Our expression language doesn't have assignment, but it does have function calls, which may have side effects.

Method 2 doesn't have this problem: Since the value of the expression is created immediately before the assignment is executed, there is no risk of other side effects between these two points in time. Method 2 also has a slight advantage when we later extend the language to have assignment statements, as we can then generate code that calculates the expression result directly into the desired variable instead of having to copy it from a temporary variable.

Hence, we will choose method 2 for our translation function $Trans_{Exp}$, which is shown in figure 6.3.

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*.

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

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
num	$v = value(\textbf{num})$ $[place := v]$
id	$x = lookup(vtable, name(\textbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = transop(opname(\textbf{unop}))$ $code_1 +\!\!+ [place := op\ place_1]$
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = transop(opname(\textbf{binop}))$ $code_1 +\!\!+ code_2 +\!\!+ [place := place_1\ op\ place_2]$
id$(Exps)$	$(code_1, [a_1, \ldots, a_n])$ $= Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, name(\textbf{id}))$ $code_1 +\!\!+ [place :=$ `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 , $Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 +\!\!+ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Figure 6.3: Translating an expression

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 $code_1$+$+code_2$+$+$[`t0` := `t1` − `t2`] which yields [`t1` := `v0`, `t2` := 3, `t0` := `t1` − `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 [`t0` := `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 chapters 7 and 8.

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

```
t1 := 3
   t4 := v0
   t5 := v1
 t3 := t4 − t5
 t6 := v2
t2 := CALL _f(t3,t6)
t0 := t1 + t2
```

We have, for readability, laid the code out on separate lines rather than using a comma-separated list. The indentation indicates the depth of calls to $Trans_{Exp}$ that produced the code in each line.

6.5 Translating statements

We now extend the expression language in figure 6.2 with statements. The extensions are shown in grammar 6.4.

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 figure 6.5. It uses an auxiliary translation function for conditions shown in figure 6.6.

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

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

Grammar 6.4: Statement language

An assignment is translated by translating the right-hand-side expression using the left-hand-side variable as target location (*place*).

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, a jump is

$Trans_{Stat}(Stat, vtable, ftable)$ = `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	$place = lookup(vtable, name(\textbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place)$
`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` $Cond$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond, label_2, label_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] ++ code_1$ $++ code_2 ++ [\texttt{LABEL}\ label_2]$

Figure 6.5: Translation of statements

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

Figure 6.6: Translation of simple conditions

made to the start of the code for the condition. A total of three labels are thus required: One for the start of the loop, one for the loop body and one for the end of the loop.

A `repeat-until` loop is slightly simpler. The body precedes the condition, so there is always at least one pass through the loop. If the condition is true, the loop is terminated and we continue with the code after the loop. If the condition is false, we jump to the start of the loop. Hence, only two labels are needed: One for the start of the loop and one for the code after the loop.

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 the first evaluates to **false**, and a logical **or** will not evaluate the second operand if the first is **true**.

The first variant is typically implemented by using bitwise logical operators and uses 0 to represent **false** and a nonzero value (typically 1 or -1) to represent **true**. In C, there is no separate boolean type. The integer 1 is used for logical truth[1] and 0 for falsehood. Bitwise logical

[1] Actually, any non-zero value is treated as logical truth.

operators `&` (bitwise **and**) and `|` (bitwise **or**) are used to implement the corresponding logical operations. Logical negation is *not* handled by bitwise negation, as the bitwise negation of 1 isn't 0. Instead, a special logical negation operator `!` is used. This maps any non-zero value to 0 and 0 to 1.

The second variant is called *sequential logical operators*. In C, these are called `&&` (logical **and**) and `||` (logical **or**).

Adding non-sequential logical operators to our language isn't too difficult. Since we haven't said exactly which binary and unary operators exist in the intermediate language, we can simply assume these include relational operators, bitwise logical operations and logical negation. We can now simply allow any expression[2] as a condition by adding the production

$$Cond \rightarrow Exp$$

to grammar 6.4. 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	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(opname(\textbf{relop}))$ $code_1 ++ code_2 ++ [$`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). 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.

[2] If it of boolean type, which we assume has been checked by the type checker.

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

Grammar 6.7: Example language with logical operators

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 figure 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 figure 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 number. We do this by first assuming that the condition is false and hence assign 0 to the target location. We

$Trans_{Exp}(Exp, vtable, ftable, place)$ = `case` Exp `of`	
⋮	
`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	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(opname(\textbf{relop}))$ $code_1$++$code_2$++[`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` $label_f$]

Figure 6.8: Translation of sequential logical operators

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.

It gets a bit more interesting in $Trans_{Cond}$, where we translate conditions. We have already seen how comparisons and expressions are translated, so we move directly to the new cases.

The constant `true` condition just generates a jump to the label for true conditions, and, similarly, `false` generates a jump to the label for false conditions.

Logical negation generates no code by itself, it just swaps the attribute-labels for true and false when translating its argument. This 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 $Trans_{Cond}$. The call to $Trans_{Cond}$ 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.

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 don't 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: 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 $Trans_{Exp}$.

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 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 chapter 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:

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `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`	
$\mathbf{id}[Exp]$	$base = lookup(vtable, name(\mathbf{id}))$ $t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_2 = code_1 +\!\!+ [t := t * 4, t := t + base]$ $(code_2, t)$

Figure 6.9: Translation for one-dimensional arrays

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

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 figure 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.

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 chapter 9) and can be calculated from this at every array-lookup. However, this doesn't 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 han-

	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]`	⋯
⋮	⋮	⋮	⋮	⋱

Figure 6.10: A two-dimensional array

dles 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. Garbage collection is, however, beyond the scope of this book.

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 i Figure 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]`, ...[3]

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]
```

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

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:

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

and extend the translation functions as shown in figure 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 chapter 8) and makes it easier to return an array as the result of an expression or function, as only the base-address needs to be returned. The size information is normally stored just before the base-address so, for example, the size of the first dimension can be at address $base - 4$, 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

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

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

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

Figure 6.11: Translation of multi-dimensional arrays

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.

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. 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.

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 section 6.8.2. 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.

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.

Translation of function declarations will be treated in chapter 9.

6.9.1 Example: Simple local declarations

We extend the statement language by the following productions:

$$\begin{array}{lcl} \mathit{Stat} & \rightarrow & \mathit{Decl}\ ;\ \mathit{Stat} \\ \mathit{Decl} & \rightarrow & \texttt{int}\ \mathbf{id} \\ \mathit{Decl} & \rightarrow & \texttt{int}\ \mathbf{id}[\mathbf{num}] \end{array}$$

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. When allocating arrays, no check for heap overflow is done.

6.10 Further reading

A comprehensive discussion about intermediate languages can be found in [30].

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 [21], [8] and [6].

$Trans_{Stat}(Stat, vtable, ftable) =$ `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**	$t_1 = newvar()$ $vtable_1 = bind(vtable, name(\mathbf{id}), t_1)$ $([],\ vtable_1)$
`int` **id**[**num**]	$t_1 = newvar()$ $vtable_1 = bind(vtable, name(\mathbf{id}), t_1)$ $([t_1 := HP,\ HP := HP + (4 * value(\mathbf{num}))],\ vtable_1)$

Figure 6.12: Translation of simple declarations

Another high-level intermediate language is the Java Virtual Machine [25]. This language has single instructions for such complex things as calling virtual methods 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.

Exercises

Exercise 6.1

Use the translation functions in figure 6.3 to generate code for the expression `2+g(x+y,x*y)`. Use a *vtable* that binds `x` to `t0` and `y` to `t1` 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 figures 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 figures 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 figure 6.8.

Exercise 6.5

Show that, in any code generated by the functions in figures 6.5 and 6.8, every `IF-THEN-ELSE` instruction will be followed by one of the target labels.

Exercise 6.6

Extend figure 6.5 to include a `break`-statement for exiting loops, as described in section 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:

$$\begin{array}{lcl} Stat & \rightarrow & \textbf{labelid} : \\ Stat & \rightarrow & \texttt{goto}\ \textbf{labelid} \end{array}$$

for defining and jumping to labels.

Extend figure 6.5 to handle these as described in section 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 figure 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 figure 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:

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

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 figure 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 curly braces are used as disambiguators, though they are not part of grammar 6.4.

Use the same *vtable* as exercise 6.1 and use `endlab` as the *endlabel* for the whole statement.

Exercise 6.10

In figure 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 figure 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 both 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 figure 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.

Chapter 7

Machine-Code Generation

7.1 Introduction

The intermediate language we have used in chapter 6 is quite low-level and not unlike 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 a complex `CALL` instruction for function calls.
- 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 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 chapter 8. Function calls are treated in chapter 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 see how we can exploit the instruction set this way.

Additionally, we will briefly discuss other optimisations.

7.2 Conditional jumps

Conditional jumps come in many shapes on different machines. Some conditional jump instructions embody a relational comparison between two registers (or a register and a constant) and are, hence, similar to the `IF-THEN-ELSE` instruction in our intermediate language. Other types of conditional jump instruction 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.

This isn't terribly difficult to handle: `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 jump instruction on the condition c.

It will, however, often be the case that 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 chapter 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
```

Hence, the code generator should test which (if any) of the target labels follow an `IF-THEN-ELSE` instruction and translate it accordingly. Alternatively, a pass can be made over the generated machine-code to remove superfluous jumps.

It is possible to extend the translation function for conditionals to use extra inherited attributes that tell which of the target labels (if any) immediately follow the condition code and use this to generate code such that the false-labels of `IF-THEN-ELSE` instructions immediately follow these (inserting `GOTO` instructions if necessary).

If the conditional jump instructions in the target machine do not

allow conditions as complex as those used in the intermediate language, code must be generated to calculate the condition and put the result somewhere where it can be tested by the 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 Intel's IA-64) use special condition registers, while yet others (*e.g.* IA-32, 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, in combination with different conditions, be used for testing equality, signed or unsigned less-than, overflow and several other properties. Usually, an `IF-THEN-ELSE` instruction can be translated to two instructions: One that does the comparison and one that does the conditional jump.

7.3 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 portions (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 any 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, a single machine-code instruction is generated. If not, a sequence of instructions are generated 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 section 7.5.

7.4 Exploiting complex instructions

Most instructions in our intermediate language are *atomic*, in the sense that they each correspond to a single operation and can not sensibly be split into several smaller steps. The exceptions to this rule are `IF-THEN-ELSE`, which is described above, and `CALL`, which will be detailed in chapter 9.

While the philosophy behind RISC (Reduced Instruction Set Computer) processors advocates that machine-code instructions should be atomic, most RISC processors include a few non-atomic instructions. CISC (Complex Instruction Set Computer) processors have composite (*i.e.*, non-atomic) instructions in abundance.

To exploit these composite instructions, several intermediate-language instructions must be grouped together and translated into a single machine-code instruction. For example, the 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, this is only possible if the value of t_2 isn't required later, as the combined instruction doesn't store this intermediate 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, as we shall see in chapter 8. For now, we will just assume that the last use of any variable is marked in the intermediate code.

Our next step is to describe each machine-code instruction in terms of one or more intermediate-language instructions. For example, the MIPS instruction `lw` r_t, $k(r_s)$ can be described by the *pattern*

$$[t := r_s + k,\ r_t := M[t^{last}]]$$

where t^{last} indicates that t can't be used afterwards. A pattern can only be used to replace a piece of intermediate code if all *last* annotations in the pattern are matched by *last* annotations in the intermediate code.

The converse, however, isn't true: It is not harmful to store a value even if it isn't used later, so a *last* annotation in the intermediate language need not be matched by a *last* annotation in the pattern.

The list of patterns that describe the machine-code instruction set must cover all of the intermediate language. In particular, each single intermediate-language instruction must be covered by a pattern. This means that we must include the MIPS instruction `lw` r_t, $0(r_s)$ to cover the intermediate-code sequence $[r_t := M[r_s]]$, even though we have already listed a more general form for `lw`. If there are intermediate-language instructions for which there are no equivalent machine-code instruction, a sequence of machine-code instructions must be given for these. 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 achieving this is to list the pairs in order of preference (*e.g.*, longest pattern first) and pick the first pattern that matches a prefix of the intermediate code.

This kind of algorithm is called *greedy*, because it always picks the choice that is best for immediate profit. It will, however, not always yield the optimal solution for the total sequence of intermediate-language instructions. If costs are given for each machine-code instruction sequence in the code-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. We will not go into detail about optimal solutions, though. For RISC processors, a greedy algorithm will typically get close to optimal solutions, so the gain by using a better algorithm is small.

As an example, figure 7.1 describes a subset of the instructions for the MIPS microprocessor architecture in terms of the intermediate language. Note that we exploit the fact that register 0 is hard-wired to be the value 0 to, *e.g.*, get the `addi` instruction to generate a constant. We assume we, at this point, have already handled the problem of too-large constants, so any constant remaining in the intermediate code can be used as an immediate constant in an instruction. Note that we make special cases for `IF-THEN-ELSE` when one of the labels follow the test. Note, also, that we need (at least) two instructions from our MIPS subset to implement an `IF-THEN-ELSE` instruction that uses less-than as the

relational operator, while we need only one for comparison by equal. Figure 7.1 does not cover all of the intermediate language, nor does it cover the full MIPS instruction set, but it can be extended to do either or both.

The instructions in figure 7.1 are listed in order of priority. This is only important when the pattern for one instruction sequence is a prefix of a pattern for another instruction sequence, as is the case with `addi` and `lw`/`sw` and for the different instances of `beq`/`bne` and `slt`.

We can try to use figure 7.1 to select instructions for the following code sequence:

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

Only one pattern (for the `add` instruction) in figure 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: `sw` and `addi`. Since `sw` is listed first, we choose this to replace the next two intermediate-language instructions. Finally, `beq` match the last two instructions. Hence, we generate the code

$$
\begin{array}{rl}
 & \texttt{add} \quad a,\ a,\ b \\
 & \texttt{sw} \quad\ a,\ 8(c) \\
 & \texttt{beq} \quad a,\ c,\ label_1 \\
label_2 : &
\end{array}
$$

Note that we retain $label_2$ even though the resulting sequence doesn't 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 don't generate actual code and hence cost nothing[1].

7.4.1 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

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

	`lw`	$r_t,\ k(r_s)$	$t := r_s + k,$ $r_t := M[t^{last}]$
	`lw`	$r_t,\ 0(r_s)$	$r_t := M[r_s]$
	`lw`	$r_t,\ k($`R0`$)$	$r_t := M[k]$
	`sw`	$r_t,\ k(r_s)$	$t := r_s + k,$ $M[t^{last}] := r_t$
	`sw`	$r_t,\ 0(r_s)$	$M[r_s] := r_t$
	`sw`	$r_t,\ k($`R0`$)$	$M[k] := r_t$
	`add`	r_d, r_s, r_t	$r_d := r_s + r_t$
	`add`	r_d, `R0`, r_t	$r_d := r_t$
	`addi`	r_d, r_s, k	$r_d := r_s + k$
	`addi`	r_d, `R0`, k	$r_d := k$
	`j`	*label*	`GOTO` *label*
	`beq`	$r_s, r_t, label_t$	`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$,
$label_f$:			`LABEL` $label_f$
	`bne`	$r_s, r_t, label_f$	`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$,
$label_t$:			`LABEL` $label_t$
	`beq`	$r_s, r_t, label_t$	`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$
	`j`	$label_f$	
	`slt`	r_d, r_s, r_t	`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$,
	`bne`	r_d, `R0`, $label_t$	`LABEL` $label_f$
$label_f$:			
	`slt`	r_d, r_s, r_t	`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$,
	`beq`	r_d, `R0`, $label_f$	`LABEL` $label_t$
$label_t$:			
	`slt`	r_d, r_s, r_t	`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$
	`bne`	r_d, `R0`, $label_t$	
	`j`	$label_f$	
label:			`LABEL` *label*

Figure 7.1: A subset of the MIPS instruction set

processors use two-address code, where the destination register is the same as the first operand register. To handle this, we use patterns like

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

that use an extra copy-instruction in the case where the destination register is not the same as the first operand. As we will see in chapter 8, the register allocator will often be able to remove the extra copy-instruction by allocating r_d and r_s in the same register.

7.5 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 it makes sense to perform optimisations mainly in the intermediate language, as the optimisations hence can be shared among all the 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 out to before the loop so it will only be calculated once. Note that this optimisation can't be done on source-code level, as the address calculations aren't visible there. For the same reason, the optimised version isn't shown here.

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 isn't shown.

Constant propagation. Some variables may, at some points in the program, have values that are always equal to some constant. If these variables are used in calculations, these calculations may be done at compile-time. Furthermore, the variables that hold the results of these computations may now also become constant, which may enable even more compile-time calculations. Constant propagation algorithms first trace the flow of constant values through the program and then eliminate calculations. The more advanced methods 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 chapter 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 don't need to check the lower bound either.

7.6 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 section 7.4 are interchanged, our simple prefix-matching algorithm can not include the address calculation in the `sw` instruction, and would hence need 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 [30] or [9] for more details.

Descriptions of and methods for a large number of different optimisations can be found in [5], [30] and [9].

The instruction set of one version of the MIPS microprocessor architecture is described in [33].

Exercises

Exercise 7.1

Add extra inherited attributes to $Trans_{Cond}$ in figure 6.8 that, for each of the target labels, indicate if this immediately follows the code for the condition. Use this to make sure that the false-labels of `IF-THEN-ELSE` instructions immediately follow these. You can use the function *negate* to negate relational operators. Make sure the new attributes are maintained in recursive calls and modify $Trans_{Stat}$ in figure 6.5 so it sets these attributes when calling $Trans_{Cond}$.

Exercise 7.2

Use figure 7.1 and the method described in section 7.4 to generate code for the following intermediate code sequence:

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

Compare this to the example in section 7.4.

Exercise 7.3

In figures 6.3 and 6.5, identify guaranteed last-uses of temporary variables, *i.e.*, places where *last*-annotations can safely be inserted.

Exercise 7.4

Choose an instruction set (other than MIPS) and make patterns for the same subset of the intermediate language as covered by figure 7.1. Use this to translate the intermediate-code example from section 7.4.

Exercise 7.5

In some microprocessors, aritmetic 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 as follows (using notation like in figure 7.1):

`MOV` r_d, r_t	$r_d := r_t$
`ADD` r_d, r_t	$r_d := r_d + r_t$
`ADDI` r_d, k	$r_d := r_d + k$

As in MIPS, register 0 (`R0`) is hardwired to the value 0.

Add patterns to the above table for the following intermediate code instructions:

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

Use only sequences of the `MOV`, `ADD` and `ADDI` instructions to implement the intermediate code instructions. Note that neither r_s nor r_t have the *last* annotation, so their values must be preserved.

Chapter 8

Register Allocation

8.1 Introduction

When generating intermediate code in chapter 6, we have freely used as many variables as we found convenient. In chapter 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 something is missing in this picture. That thing is *register allocation*. Register allocation must 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 simply aren't registers enough 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 chapter 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 to a sequence of machine-code in-

structions, the machine code may need an extra register (or two) for storing temporary values. Hence, the register allocator must make sure that there is always a spare register for temporary storage.

The techniques are, however, the same regardless of when register allocation is done, so we will just describe the register allocation in terms of the intermediate language introduced in chapter 6.

As in chapter 6, we operate on the body of a single procedure or function, so when we below use the term "program", we mean it to be such a body.

8.2 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 chapter 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 that time isn't used.

3) If a variable is live at the start of an instruction, it is alive at the end of the immediately preceding instructions.

4) If a variable is live at the end of an instruction and that instruction doesn't assign a value to the variable, then the variable is also live at the start of the instruction.

Rule 1 tells how liveness is *generated*, rule 2 how liveness is *killed* and rules 3 and 4 how it is *propagated*.

8.3 Liveness analysis

We can formalise the above rules into equations. The process of solving these equations is called *liveness analysis*, and will for all points 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 figure 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` or `IF-THEN-ELSE` instruction. If instructions are numbered consecutively, $j = i + 1$.

2) If the instruction numbered i is `GOTO` l, (the number of) the instruction `LABEL` l is in $succ[i]$.

3) If instruction i is `IF` p `THEN` l_t `ELSE` l_f, (the numbers of) the instructions `LABEL` l_t and `LABEL` l_f are in $succ[i]$.

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 isn't going to be used. The converse (claiming a variable dead when it is in fact live) is worse, as we may overwrite a value that may actually be used later, and hence get wrong results from the program. Precise liveness is not computable, so it is quite reasonable to allow imprecise results, as long as we err on the side of safety.

For every instruction i, we have a set $gen[i]$. $gen[i]$ lists the variables that may be read by instruction i and hence are live at the start of the instruction, *i.e.*, the variables that i *generate* 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:

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` $f(args)$	$args$	$\{x\}$

Figure 8.1: Gen and kill sets

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

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

These equations are recursive. We solve these by fixed-point iteration, as shown in section 2.6.1: We initialise all $in[i]$ and $out[i]$ to the empty set and repeatedly calculate new values for these until no changes occur.

This works under the assumption that all variables are dead at the end of the program. If a variable contains, *e.g.*, the output of the program, it isn't dead at the end of the program, so we must ensure that the analysis knows this. This can be done by letting $out[i]$, where i is the last instruction in the program, contain all variables that are live at the end of the program. This definition of $out[i]$ replaces (for the last instruction only) equation 8.2.

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.

We assume that a contains the result of the program (*i.e.*, is live at the end of it), so we set $out[13] = \{a\}$. The other *out* sets are defined

```
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
```

Figure 8.2: Example program for liveness analysis

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			

Figure 8.3: *succ*, *gen* and *kill* for the program in figure 8.2

	Initial		Iteration 1		Iteration 2		Iteration 3	
i	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$
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

Figure 8.4: Fixed-point iteration for liveness analysis

by equation 8.2 and all *in* sets are defined by equation 8.1. We initialise all *in* and *out* sets to the empty set and iterate.

The order in which we treat the instructions doesn't matter for the final result of the iteration, but it may influence how quickly we reach the fixed-point. Since the information in equations 8.1 and 8.2 flow 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 calculate in the order

$$out[13],\ in[13],\ out[12],\ in[12], \ldots, out[1],\ in[1]$$

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 equations 8.1 and 8.2, so, when a value comes from a higher instruction number, the value from the same column in figure 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 at the start of the program, which means that n may be used before it is given a value. In this example, n is a parameter to the program (which calculates the n'th Fibonacci number) so it will be initialised before running the

program. In other cases, a variable that is live at the start of a program may be used before it is initialised, which may lead to unpredictable results. Some compilers issue warnings in such situations.

8.4 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 *the instruction isn't of the form $x := y$.*

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 isn't used after an instruction that kills x. In this case x is not technically live afterwards, but it still interferes with any y that is live after the instruction, as the instruction will overwrite the register that contains x.

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't be eliminated.

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.

We can draw interference as a 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 y interferes with x. The *interference graph* for the program in figure 8.2 is shown in figure 8.5. This interference is generated by the assignments in figure 8.2 as follows:

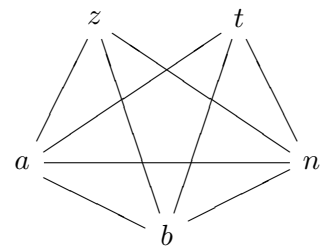


Figure 8.5: Interference graph for the program in figure 8.2

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

8.5 Register allocation by graph colouring

In the interference graph, two variables can share a register if they aren't connected by an edge. Hence, we must assign to each node in the 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.

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, nor is one likely to exist. 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, so at worst we get slightly slower programs than we would get if we could colour the graph optimally.

The basic idea of the heuristic method we use is simple: If a node in the graph has strictly fewer than N neighbours, 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 less-than-N neighbours of the selected node can't 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 figure 8.5:

1) z has three neighbours, which is strictly less than four. Hence, we remove z from the graph.

2) Now, a has less than four neighbours, so we remove this.

3) Only three nodes are now left (b, t and n), so we give each of these a number, *e.g.*, $b = 1$, $t = 2$ and $n = 3$.

4) Since a neighbours b, t and n, we must choose a fourth colour for a, *i.e.*, $a = 4$.

5) z has a, b and n as neighbours, so we choose a colour that is different from 4, 1 and 3, *i.e.*, $z = 2$.

this is easy enough. The problem comes if there are no nodes that have less than N neighbours. This in itself is no guarantee that the graph isn't colour-able. 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 neighbours, put this along with a list of its neighbours on the stack and remove it and its edges from the graph.*

If there is no node with less than N neighbours, 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 neighbour-list from the stack. If possible, give the node a colour that is different from the colours of its neighbours. 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 point of this heuristics is that, even though a node has N or more neighbours, some of these may be given identical colours, so it may, in **select**, be possible to find a colour for the node anyway.

There are several things left unspecified by algorithm 8.3: Which node to choose in **simplify** when none have less than N neighbours, and which colour to choose in **select** if there are several choices. If an oracle chooses perfectly in both cases, algorithm 8.3 will do optimal colouring. In practice, we will have to make do with qualified guesses. We will, in section 8.7, look at some possibilities for doing this. For now, we just make arbitrary choices.

8.6 Spilling

If **select** can not find a colour for a node, algorithm 8.3 can not colour the graph. If this is the case, we must give up on keeping all variables in registers at all times. We must hence select some variables that (most of the time) reside in memory. This is called *spilling*. Obvious candidates for spilling are variables at nodes that can not be given colours by **select**. We simply mark these as *spilled* and continue doing **select** on the rest of the stack, ignoring spilled neighbours 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 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 live at the start of the program, add an instruction $M[address_x] := x$ to the start of the program. Note that we use the original name for x here.

6) If x is live at the end of the program, add an instruction $x := M[address_x]$ to the end of the program. Note that we use the original name for x here.

After this rewrite of the program, we do register allocation again. This includes re-doing the liveness analysis, since the x_i have different liveness than the x they replace. We may optimise this a bit by doing liveness analysis only on the spilled variables, as the other variables have unchanged liveness.

It may happen that the new register allocation too will fail and generate more spill. There are several reasons why this may be:

- We have ignored spilled variables when selecting colours for the other nodes, but the spilled variables are replaced by new ones that may still interfere with some of these nodes and cause colouring of these to fail.

- The order in which we select nodes for simplification and colouring has changed, 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 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 needs special treatment. We will look at this in chapter 9.

If we take our example from figure 8.2, we can attempt to colour its interference graph (figure 8.5) with only three colours. The stack built by the **simplify** phase of algorithm 8.3 and the colours chosen

Node	Neighbours	Colour
n		1
t	n	2
b	t, n	3
a	b, n, t	*spill*
z	a, b, n	2

Figure 8.6: Algorithm 8.3 applied to the graph in figure 8.5

for these nodes in the **select** phase are shown in figure 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 new interference graph and figure 8.9 shows the stack used by algorithm 8.3 for colouring this graph.

8.7 Heuristics

When the **simplify** phase of algorithm 8.3 can't find a node with less than N neighbours, 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:

- We may choose a node with close to N neighbours, as this is likely to be colour-able in the **select** phase anyway. For example, if a node has exactly N neighbours, it will be colour-able 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 doesn't cost so much to spill, *e.g.*, a variable that is not used inside a loop.

```
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_2 := M[address_a]
7:  t := a_2 + b
8:  a_3 := b
    M[address_a] := a_3
9:  b := t
10: n := n - 1
11: z := 0
12: GOTO loop
13: LABEL end
    a := M[address_a]
```

Figure 8.7: Program from figure 8.2 after spilling variable a

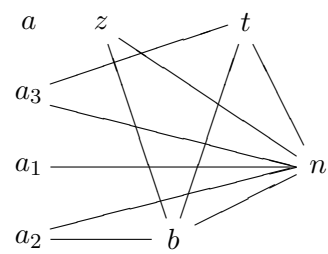


Figure 8.8: Interference graph for the program in figure 8.7

Node	Neighbours	Colour
n		1
t	n	2
a_3	t, n	3
b	t, n	3
a_2	b, n	2
z	b, n	2
a_1	n	2
a		1

Figure 8.9: Colouring of the graph in figure 8.8

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 criteria, and then add these values to give a weighted sum.

The other place where we have made arbitrary choices, is 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 new colour. This will use a smaller total number of colours and, hence, make it more likely that the neighbours of an as yet uncoloured node will share colours. We can refine this a bit by looking at the uncoloured neighbours of the selected node and for each of these look at their already coloured neighbours. If we can pick a colour that occurs often among these, this increases the likelihood that we will be able to colour the uncoloured neighbours.

8.7.1 Removing redundant moves

An assignment of the form $x := y$ can be removed if x and y use the same register. Most register allocators do this, and some even try to increase the number of such removed assignments by increasing the chance that x and y use the same register.

If x has already been given a colour by the time we need to select a colour for y (or *vice versa*), we can choose the same colour for y, as long as it isn't used by any of y's neighbours (including, possibly, x). This is called *biased colouring*.

Another method of achieving the same goal is to combine x and y (if they don't interfere) into a single node before colouring the graph, and

only split the combined node if the **simplify** phase can't otherwise find a node with less than N neighbours. 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.

8.8 Further reading

Preston Briggs' Ph.D. thesis [11] 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 [30] and [9] show some other variants and a few newer developments. A completely different approach that exploits the structure of a program is suggested in [35].

Exercises

Exercise 8.1

Given the following program:

```
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
```

a) Show *succ*, *gen* and *kill* for every instruction in the program.

b) Assuming a is live at the end of the program, *i.e.*, $out[11] = \{a\}$, calculate *in* and *out* for every instruction in the program. Show the iteration as in figure 8.4.

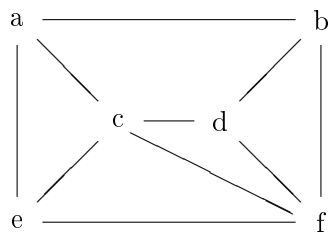
c) Draw the interference graph for a, b, t and z.

d) Make a three-colouring of the interference graph. Show the stack as in figure 8.6.

e) Attempt, instead, a two-colouring of the graph. Select variables for spill, do the spill-transformation as shown in section 8.6 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 figure 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 section 8.7 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 address 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 alloacetd in register pairs. We note that live-ness 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.

Chapter 9

Function calls

9.1 Introduction

In chapter 6 we have shown how to translate the body of a single function. Function calls were left (mostly) untranslated by using the `CALL` instruction in the intermediate code. Nor did we in chapter 7 show how the `CALL` instruction should be translated.

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

9.1.1 The call stack

A single procedure body uses (in most languages) a finite number of variables. We have seen in chapter 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't 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*) will be stored in memory so the

registers will be free for use by the called function (the *callee*). When the callee returns, the stored variables are loaded back into registers. It is convenient to use a stack for this storing and loading, pushing register contents on the stack when they must be saved and popping them back into registers when they must be restored. Since a stack is (in principle) unbounded, this fits well with the idea of unbounded recursion.

The stack can also be used for other purposes:

- Space can be set aside on the stack for variables that need to be spilled to memory. In chapter 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 stack-pointer. This makes the spill-code slightly more complex, but has the advantage that spilled registers are already saved on the stack when or if a function is called, so they don't 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 therefrom 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, variables that are in scope in a function but not declared locally in that function must reside in memory. It is convenient to access these through the stack.

- Arrays and records that are allocated locally in a function can be allocated on the stack, as hinted in section 6.8.2.

We shall look at each of these in more detail later on.

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,

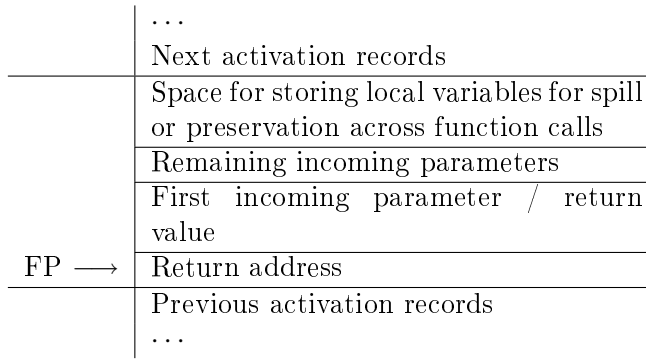


Figure 9.1: Simple activation record layout

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 another compiler or even written in a different language, as long as both compilers follow the same calling convention.

We will start by defining very simple activation records and then extend and refine these later on. Our first model uses the assumption that all information is stored in memory when a function is called. This includes parameters, return address and the contents of registers that need to be preserved. A possible layout for such an activation record is shown in figure 9.1.

FP is shorthand for "Frame pointer" and points to the first word of the activation record. In this layout, the first word holds the return address. Above this, the incoming parameters are 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). Above the incoming parameters, the activation record has space for storing other local variables, *e.g.*, for spilling or for preserving across later function calls.

9.3 Prologues, epilogues and call-sequences

In front of the code generated (as shown in chapter 6) for a function body, we need to put some code that reads parameters from the activation

$$\text{Prologue} \begin{cases} \texttt{LABEL}\ \textit{function-name} \\ \textit{parameter}_1 := M[FP + 4] \\ \cdots \\ \textit{parameter}_n := M[FP + 4 * n] \end{cases}$$

code for the function body

$$\text{Epilogue} \begin{cases} M[FP + 4] := \textit{result} \\ \texttt{GOTO}\ M[FP] \end{cases}$$

Figure 9.2: Prologue and epilogue for the frame layout shown in figure 9.1

record into registers. This code is called the *prologue* of the function. Likewise, after the function body, we need code to store the calculated return value in the activation record and jump to the return address that was stored in the activation record by the caller. For the activation record layout shown in figure 9.1, a suitable prologue and epilogue is shown in figure 9.2. Note that, though we have used a notation similar to the intermediate language introduced in chapter 6, we have extended this a bit: We have used $M[]$ and `GOTO` with general expressions as arguments.

We use the names $\textit{parameter}_1, \ldots, \textit{parameter}_n$ for the intermediate-language variables used in the function body for the n parameters. *result* is the intermediate-language variable that holds the result of the function after the body have been executed.

A function call is translated into a *call-sequence* of instructions that will save registers, set up parameters, *etc.* A call-sequence suitable for the activation record layout shown in figure 9.1 is shown in figure 9.3. The code is an elaboration of the intermediate-language instruction $x :=$ `CALL` $f(a_1, \ldots, a_n)$. First, all registers that can be used to hold variables are stored in the frame. In figure 9.3, *R*0-*Rk* are assumed to hold variables. These are stored in the activation record just above the calling functions own m incoming parameters. Then, the frame-pointer is advanced to point to the new frame and the parameters and the return address are stored in the prescribed locations in the new frame. Finally, a jump to the function is made. When the function call returns, the result is read from the frame into the variable x, FP is restored to its former value and the saved registers are read back from the old frame.

$$
\begin{array}{l}
M[FP + 4 * m + 4] := R0 \\
\cdots \\
M[FP + 4 * m + 4 * (k + 1)] := Rk \\
FP := FP + framesize \\
M[FP + 4] := a_1 \\
\cdots \\
M[FP + 4 * n] := a_n \\
M[FP] := returnaddress \\
\texttt{GOTO}\ f \\
\texttt{LABEL}\ returnaddress \\
x := M[FP + 4] \\
FP := FP - framesize \\
R0 := M[FP + 4 * m + 4] \\
\cdots \\
Rk := M[FP + 4 * m + 4 * (k + 1)]
\end{array}
$$

Figure 9.3: Call sequence for $x :=$ `CALL` $f(a_1, \ldots, a_n)$ using the frame layout shown in figure 9.1

Keeping all the parameters in register-allocated variables until just before the call, and only then storing them in the frame can require a lot of registers to hold the parameters (as these are all live up to the point where they are stored). An alternative is to store each parameter in the frame as soon as it is evaluated. This way, only one of the variables $a_1, \ldots, a_n$ will be live at any one time. However, this can go wrong if a later parameter-expression contains a function call, as the parameters to this call will overwrite the parameters of the outer call. Hence, this optimisation must only be used if no parameter-expressions contain function calls or if nested calls use stack-locations different from those used by the outer call.

In this simple call-sequence, we save on the stack all registers that can potentially hold 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 section 9.6.

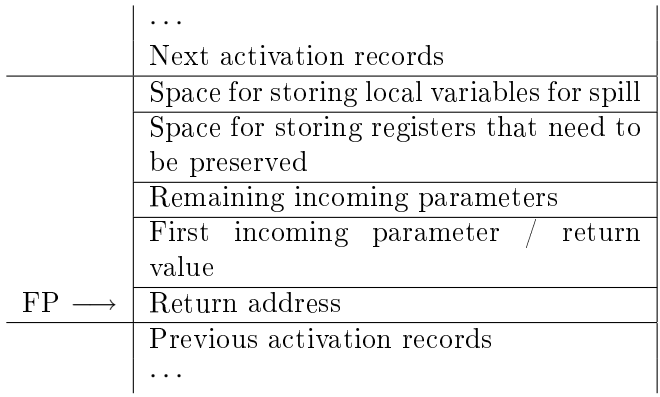


Figure 9.4: Activation record layout for callee-saves

9.4 Caller-saves versus callee-saves

The convention used by the activation record layout in figure 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 that the called function saves the contents of the registers that need to be preserved and restores 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 figures 9.4, 9.5 and 9.6.

Note that it may not be necessary to store *all* registers that may potentially be used to hold variables, only those that the function actually uses to hold its local variables. We will return to this issue in section 9.6.

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 registers that the function actually uses. We will in section 9.6 return to how this can be achieved, 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 doesn't use the register that holds this variable. On the other hand, with callee-saves we might save some registers that don't actually hold live values. We can't avoid these unnecessary saves, as each function is compiled independently and

$$
\text{Prologue}\ \begin{cases}
\texttt{LABEL}\ \textit{function-name} \\
M[FP + 4 * n + 4] := R0 \\
\cdots \\
M[FP + 4 * n + 4 * (k + 1)] := Rk \\
\textit{parameter}_1 := M[FP + 4] \\
\cdots \\
\textit{parameter}_n := M[FP + 4 * n]
\end{cases}
$$

code for the function body

$$
\text{Epilogue}\ \begin{cases}
M[FP + 4] := \textit{result} \\
R0 := M[FP + 4 * n + 4] \\
\cdots \\
Rk := M[FP + 4 * n + 4 * (k + 1)] \\
\texttt{GOTO}\ M[FP]
\end{cases}
$$

Figure 9.5: Prologue and epilogue for callee-saves

$$
\begin{array}{l}
FP := FP + \textit{framesize} \\
M[FP + 4] := a_1 \\
\cdots \\
M[FP + 4 * n] := a_n \\
M[FP] := \textit{returnaddress} \\
\texttt{GOTO}\ f \\
\texttt{LABEL}\ \textit{returnaddress} \\
x := M[FP + 4] \\
FP := FP - \textit{framesize}
\end{array}
$$

Figure 9.6: Call sequence for $x :=$ `CALL` $f(a_1, \ldots, a_n)$ for callee-saves

hence don't 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 figure 9.3, 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 first, as in figure 9.5 (though only callee-saves registers that are actually used in the body are saved).

Calling conventions typically specify which registers are caller-saves and which are callee-saves, as well as the layout of the activation records.

9.5 Using registers to pass parameters

In both call sequences shown (in figures 9.3 and 9.6), parameters are stored in the frame, and in both prologues (figures 9.2 and 9.5) most of these are immediately loaded back into registers. It will save a good deal of memory traffic if we pass the parameters in registers instead of memory.

Normally, only a few (4-8) registers are used for parameter passing. These are used for the first parameters of a function, while the remaining parameters are passed on the stack, as we have done above. Since most functions have fairly short parameter lists, most parameters will normally be passed in registers. The registers used for parameter passing are typically a subset of the caller-saves registers, as parameters aren't live after the call and hence don't have to be preserved.

A possible division of registers for a 16-register architecture is shown in figure 9.7. Note that the return address is also passed in a register. Most RISC architectures have jump-and-link (function-call) instructions, which leaves the return address in a register, so this is only natural. However, if a further call is made, this register is overwritten, so the return address must be saved in the activation record before this happens. The return-address register is marked as callee-saves in figure 9.7. In this manner, the return-address register is just like any other variable that must be preserved in the frame if it is used in the body (which it is if a function call is made). Strictly speaking, we don't need the return address after the call has returned, so we can also argue that R15 is a caller-saves register. If so, the caller must save R15 prior to any call, *e.g.*, by spilling it.

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	FP
15	callee	return address

Figure 9.7: Possible division of registers for 16-register architecture

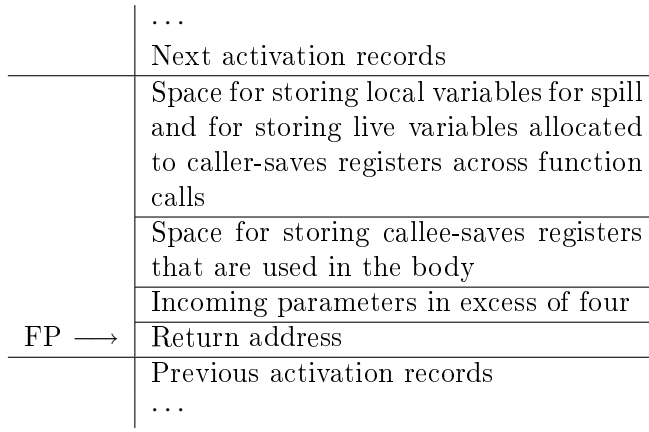


Figure 9.8: Activation record layout for the register division shown in figure 9.7

Prologue

```
LABEL function-name
M[FP + offset_R4] := R4          (if used in body)
...
M[FP + offset_R12] := R12        (if used in body)
M[FP] := R15       (if used in body)
parameter_1 := R0
parameter_2 := R1
parameter_3 := R2
parameter_4 := R3
parameter_5 := M[FP + 4]
...
parameter_n := M[FP + 4 * (n - 4)]
```

code for the function body

Epilogue

```
R0 := result
R4 := M[FP + offset_R4]          (if used in body)
...
R12 := M[FP + offset_R12]        (if used in body)
R15 := M[FP]       (if used in body)
GOTO R15
```

Figure 9.9: Prologue and epilogue for the register division shown in figure 9.7

$M[FP + \mathit{offset}_{live_1}] := live_1$ (if allocated to a caller-saves register)
$\ldots$
$M[FP + \mathit{offset}_{live_k}] := live_k$ (if allocated to a caller-saves register)
$FP := FP + framesize$
$R0 := a_1$
$\ldots$
$R3 := a_4$
$M[FP + 4] := a_5$
$\ldots$
$M[FP + 4 * (n - 4)] := a_n$
$R15 := returnaddress$
`GOTO` f
`LABEL` $returnaddress$
$x := R0$
$FP := FP - framesize$
$live_1 := M[FP + \mathit{offset}_{live_1}]$ (if allocated to a caller-saves register)
$\ldots$
$live_k := M[FP + \mathit{offset}_{live_k}]$ (if allocated to a caller-saves register)

Figure 9.10: Call sequence for $x :=$ `CALL` $f(a_1, \ldots, a_n)$ for the register division shown in figure 9.7

Activation record layout, prologue/epilogue and call sequence for a calling convention using the register division in figure 9.7 are shown in figures 9.8, 9.9 and 9.10.

Note that the offsets for storing registers are not simple functions of their register numbers, as only a subset of the registers need to be saved. R15 (which holds the return address) is treated as any other callee-saves register. Its offset is 0, as the return address is stored at offset 0 in the frame.

In a call-sequence, the instructions

$$
\begin{array}{l}
R15 := returnaddress \\
\texttt{GOTO}\ f \\
\texttt{LABEL}\ returnaddress
\end{array}
$$

can on most RISC processors be implemented by a jump-and-link instruction.

9.6 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 variables 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 doesn't have to save these and, with luck, they don't have to be saved by the callee either (if the callee doesn't 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 variables 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 figures 9.2, 9.5 and 9.9 load all stack-passed parameters into registers, this should actually only be done for parameters that aren't spilled. Likewise, a register-passed parameter that needs to be spilled should in the prologue be transferred to a stack location instead of to a symbolic register ($parameter_i$).

In figures 9.2, 9.5 and 9.9, 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 body when the register allocator is called (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 problem is, however, usually handled 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 these 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 shouldn't 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 same register as the parameter was passed in and eliminate the (superfluous) register-to-register move. As mentioned in section 8.7, 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 (except for parameter-passing in call sequences).

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.

4) To the register-allocated code, add code for saving and restoring the callee-saves registers that the register allocator says have been used in the extended function body and for updating the frame pointer with the size of the frame (including the space for saved registers).

5) Add a function label at the beginning of the code and a return jump at the end.

9.7 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.

9.7.1 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

$$t := M[address_x]$$
instruction that uses t

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

$$t := \text{the value to be stored in } x$$
$$M[address_x] := t$$

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.

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 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.

9.7.2 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 `*` (de-reference) operator. So, apart from notation and a higher potential for programming errors, this isn't 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. registers. 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.

9.7.3 Nested scopes

Some languages, *e.g.*, Pascal and SML, allow functions to be declared locally within other functions. A local function typically has access to variables declared in the function in which it itself is declared. For example, figure 9.11 shows a fragment of a Pascal program. In this program, `g` can access `x` and `y` (which are declared in `f`) as well as its own local variables `p` and `q`.

Note that, since `f` and `g` are recursive, there can be many instances of their variables in different activation records at any one time.

When `g` is called, its own local variables (`p` and `q`) are held in registers, as we have described above. All other variables (*i.e.*, `x` and `y`) reside in the activation records of the procedures/functions in which they are declared (in this case `f`). It is no problem for `g` to know the offsets for

```
procedure f (x : integer);
   var y : integer;
   function g(p : integer);
   var q : integer;
   begin
      if p<10 then y := g(p+y)
      else q := p+y;
      if (y<20) then f(y);
      g := q;
   end;
begin
   y := x+x;
   writeln(g(y),y)
end;
```

Figure 9.11: Example of nested scopes in Pascal

`x` and `y` in the activation record for `f`, as `f` can be compiled before `g`, so full information about `f`'s activation record layout is available for the compiler when it compiles `g`. However, we will not at compile-time know the position of `f`'s activation record on the stack. `f`'s activation record will not always be directly below that of `g`, since there may be several recursive invocations of `g` (each with its own activation record) above the last activation record for `f`. Hence, a pointer to `f`'s activation record will be given as parameter to `g` when it is called. When `f` calls `g`, this pointer is just the contents of FP, as this, by definition, points to the activation record of the active function (*i.e.*, `f`). When `g` is called recursively from `g` itself, the incoming parameter that points to `f`'s activation record is passed on as a parameter to the new call, so every instance of `g` will have its own copy of this pointer.

To illustrate this, we have in figure 9.12 added this extra parameter explicitly to the program from figure 9.11. Now, `g` accesses all non-local variables through the `fFrame` parameter, so it no longer needs to be declared locally inside `f`. Hence, we have moved it out. We have used record-field-selection syntax in `g` for accessing `f`'s variables through `fFrame`. Note that `fFrame` is a call-by-reference parameter (indicated by the `var` keyword), as `g` can update `f`'s variables (*i.e.*, `y`). In `f`, we have used `FP` to refer to the current activation record. Normally, a function in a Pascal program will not have access to its own frame, so this is not quite standard Pascal.

```
function g(var fFrame : fRecord, p : integer);
var q : integer;
begin
   if p<10 then fFrame.y := g(fFrame,p+fFrame.y)
   else q := p+fFrame.y;
   if (fFrame.y<20) then f(fFrame.y);
   g := q;
end;

procedure f (x : integer);
   var y : integer;
begin
   y := x+x;
   writeln(g(FP,y),y)
end;
```

Figure 9.12: Adding an explicit frame-pointer to the program from figure 9.11

It is sometimes possible to make the transformation entirely in the source language (*e.g.*, Pascal), but the extra parameters are usually not added until the intermediate code, where FP is made explicit, has been generated. Hence, figure 9.12 mainly serves to illustrate the idea, not as a suggestion for implementation.

Note that all variables that can be accessed in inner scopes need to be stored in memory when a function is called. This is the same requirement as was made for call-by-reference parameters, and for the same reason. This can, in the same way, be handled by allocating such variables in caller-saves registers.

Static links

If there are more than two nested scopes, pointers to all outer scopes need to be passed as parameters to locally declared functions. If, for example, `g` declared a local function `h`, `h` would need pointers to both `f`'s and `g`'s activation records. If there are many nested scopes, this list of extra parameters can be quite long. Typically, a single parameter is instead used to hold a linked list of the frame pointers for the outer scopes. This is normally implemented by putting the links in the activation records themselves. Hence, the first field of an activation record (the field that

	... Next activation records
	Space for storing local variables for spill and for storing live variables allocated to caller-saves registers across function calls
	Space for storing callee-saves registers that are used in the body
	Incoming parameters in excess of four
	Return address
FP ⟶	Static link (SL)
	Previous activation records ...

Figure 9.13: Activation record with static link

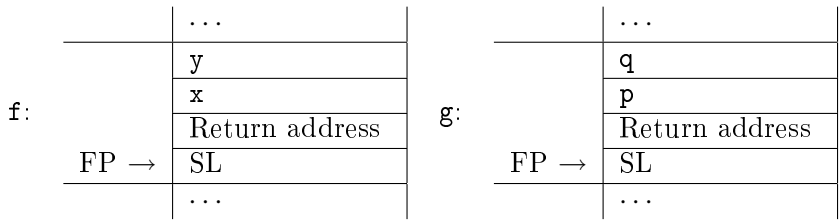


Figure 9.14: Activation records for `f` and `g` from figure 9.11

FP points to) will point to the activation record of the next outer scope. This is shown in figure 9.13. The pointer to the next outer scope is called the *static link*, as the scope-nesting is static as opposed to the actual sequence of run-time calls that determine the stacking-order of activation records[1]. The layout of the activation records for `f` and `g` from figure 9.11 is shown in figure 9.14.

`g`'s static link will point to the most recent activation record for `f`. To read `y`, `g` will use the code

$FP_f := M[FP]$	Follow `g`'s static link
$address := FP_f + 12$	Calculate address of `y`
$y := M[address]$	Get `y`'s value

where y afterwards holds the value of `y`. To write `y`, `g` will use the code

[1] Sometimes, the return address is referred to as the *dynamic link*.

$FP_f := M[FP]$	Follow `g`'s static link
$address := FP_f + 12$	Calculate address of `y`
$M[address] := y$	Write to `y`

where y holds the value that is written to `y`. If a function `h` was declared locally inside `g`, it would need to follow two links to find `y`:

$FP_g := M[FP]$	Follow `h`'s static link
$FP_f := M[FP_g]$	Follow `g`'s static link
$address := FP_f + 12$	Calculate address of `y`
$y := M[address]$	Get `y`'s value

This example shows why the static link is put in the first element of the activation record: It makes following a chain of links easier, as no offsets have to be added in each step.

Again, we can see that a programmer should keep variables as local as possible, as non-local variables take more time to access.

9.8 Variants

We have so far seen fixed-size activation records on stacks that grow upwards in memory, and where FP points to the first element of the frame. There are, however, reasons why you sometimes may want to change this.

9.8.1 Variable-sized frames

If 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 point to the end of the frame. This is typically called the *stack pointer*, because the end of the frame is also the top of the stack. When setting up parameters to a new call, these are put at places relative to SP rather than relative to FP. When a function is called, the new FP takes the value of the old SP, but we now need to store the old value of FP, as we no longer can restore it by subtracting a constant from the current FP. Hence, the old FP is passed as a parameter (in a register or in the frame) to the new function, which restores FP to this value just before returning.

If arrays are allocated on a separate stack, frames can be of fixed size, but a separate stack-pointer is now needed for allocating/deallocating arrays.

If two stacks are used, it is customary to let one grow upwards and the other downwards, such that they grow towards each other. This way, stack-overflow tests on both stacks can be replaced by a single test on whether the stack-tops meet. It also gives a more flexible division of memory between the two stacks than if each stack is allocated its own fixed-size memory segment.

9.8.2 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.

The layouts we have shown in this chapter all assume that there is a fixed number of arguments, so the offsets to, *e.g.*, local variables are known. If the number of parameters can vary, this is no longer true.

One possible solution is to have two frame pointers: One that shows the position of the first parameter and one that points to the part of the frame that comes after the parameters. However, manipulating two FP's is somewhat costly, so normally another trick is used: The FP points to the part of the frame that comes after the parameters, Below this, the parameters are stored at negative offsets from FP, while the other parts of the frame are accessed with (fixed) positive offsets. The parameters are stored such that the first parameter is closest to FP and later parameters further down the stack. This way, parameter number k will be a fixed offset $(-4 * k)$ from FP.

When a function call is made, the number of arguments to the call is known to the caller, so the offsets (from the old FP) needed to store the parameters in the new frame will be fixed at this point.

Alternatively, FP can point to the top of the frame and all fields can be accessed by fixed negative offsets. If this is the case, FP is sometimes called SP, as it points to the top of the stack.

9.8.3 Direction of stack-growth and position of FP

There is no particular reason why a stack has to grow upwards in memory. It is, in fact, more common that call stacks grow downwards in memory. 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. This means that either FP must point to the bottom of the frame and the stack grow upwards, or FP must point to the top of the frame and the stack grow downwards.

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, as suggested in section 9.8.2, by placing FP after the parameters but before the rest of the frame, so parameters are addressed by negative offsets and the rest by positive. Alternatively, FP can be positioned k bytes above the bottom of the frame, where $-k$ is the largest negative offset.

9.8.4 Register stacks

Some processors, *e.g.*, Suns Sparc and Intels IA-64 have on-chip stacks of registers. The intention is that frames are kept in registers rather than on a stack in memory. At call or return of a function, the register stack 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 trapped by the operating system which stores part of the stack in memory and shifts the rest down (or up) to make room for new elements. If the stack underflows (at a pop from an empty register stack), the OS will restore earlier saved parts of the stack.

9.9 Further reading

Calling conventions for various architectures are usually documented in the manuals provided by the vendors of these architectures. Additionally, the calling convention for the MIPS microprocessor is shown in [33].

In figure 9.12, we showed in source-language terms how an extra parameter can be added for accessing non-local parameters, but stated that this was for illustrative purposes only, and that the extra parameters aren't normally added at source-level. However, [8] argues that it *is*, actually, a good idea to do this, and goes on to show how many advanced features regarding nested scopes, higher-order functions and even register allocation can be implemented mostly by source-level transformations.

Exercises

Exercise 9.1

In section 9.3 an optimisation is mentioned whereby parameters are stored in the new frame as soon as they are evaluated instead of just before the call. It is warned that this will go wrong if any of the parameter-expressions themselves contain function calls. Argue that the *first* parameter-expression of a function call can contain other function calls without causing the described problem.

Exercise 9.2

Section 9.8.2 suggests that a variable number of arguments can be handled by storing parameters at negative offsets from FP and the rest of the frame at non-negative offsets from FP. Modify figures 9.8, 9.9 and 9.10 to follow this convention.

Exercise 9.3

Find documentation for the calling convention of a processor of your choice and modify figures 9.7, 9.8, 9.9 and 9.10 to follow this convention.

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.

Vi vil nu prøve at kombinere prolog og epilog med en krop af form som et if-then-else udtryk, sådan at man mindsker antallet af gemte *callee-saves* registre.

Replace in figure 9.9 the text "*code for function body*" by 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 $code_1$".

Chapter 10

Analysis and optimisation

In chapter 7, we briefly mentioned optimisations without going into detail about how they are done. We will remedy this in this chapter.

An optimisation is, generally, is about recognising instructions that form a specific pattern that can be replaced by a smaller or faster pattern of new instructions. In the simplest case, the pattern is just a short sequence of instructions that can be replaced by a another short sequence of instructions. In chapter 7, we replaced sequences of intermediate-code instructions by sequences of machine-code instructions, but the same idea can be applied to replacing sequences of machine-code instructions by sequences of machine-code instructions or sequences of intermediate-code instructions by other sequences of intermediate-code instructions. This kind of optimisation is called *peephole optimisation*, because we look at the code through a small hole that just allows is to see short sequences of instructions.

Another thing to note about the patterns we used in chapter 7 is that they sometimes required some of the variables involved to have no subsequent uses. This is a non-local property that requires looking at an arbitrarily large context of the instructions, sometimes the entire procedure or function in which the instructions appear. A variable having possible subsequent uses is called *live*. In chapter 8 we looked at how *liveness analysis* could determine which variables are live at each instructions. We used this to determine interference between variables, but the same information can also be used to enable optimisations, such as replacing a sequence of instructions by a simpler sequence.

Many optimisations are like this: We have a pattern of instructions and some requirements about the context in which the pattern appears. These contextual requirements are often find by analysis similar to live-

ness analysis, collectively called *data-flow analyses*.

We will now look at optimisations that can be enabled by such analyses. We will later present a few other types of optimisations.

10.1 Data-flow analysis

As the name indicates, data-flow analysis attempts to discover how information flows through a program. We already discussed liveness analysis, where the information that a variable is live flows through the program in the opposite order of the flow of values: A value flows from an assignment to a variable to its uses, but liveness information flows from a use of a variable back to its assignments. Liveness analysis is, hence, called a backwards analysis. In other analyses information flows in the same order as values. For example, an analysis might try to approximate the set of possible values that a variable can hold, and here the flow is naturally from assignments to uses.

The liveness analysis presented in chapter 8 consisted of four things:

1. Information about which instructions can follow others, *i.e.*, the successors of each instruction.

2. For each instruction *gen* and *kill* sets that describe how data-flow information is created and destroyed by the instruction.

3. Equations that define *in* and *out* sets by describing how data-flow information flow between instructions.

4. Initialisation of the *in* and *out* sets for a fixed-point iteration that solve the data-flow equations.

We will use the same template for other data-flow analyses, but the details might differ. For example:

1. Forwards analyses require information about the predecessors of an instruction instead of its successors.

2. Where liveness analysis uses sets of variables, other analyses might use sets of instructions or sets of variable/value pairs.

3. The equations for *in* and *out* sets might differ. For example, they may use intersection instead of union to combine information from several successors or predecessors of an instruction.

4. Where liveness analysis initialises all *in* and *out* sets to empty sets (except for the *out* set of the last instruction in the function, which is initialised to the set of variables live at the exit of the function), other analyses might initialise the sets to, for example, the set of all instructions in the function.

We will in the following sections show some examples of optimisations and the data-flow analyses required to find the contextual information required to determine if the optimisation is applicable. As the optimisations we show are not specific to any particular processor, we will show them in terms of intermediate code, but the same principles can be applied to analysis and optimisations of machine-code.

10.2 Common subexpression elimination

After translation to intermediate code, there might be several occurrences of the same calculations, even when this is not the case in the source program. For example, the assignment `a[i] := a[i]+1` can in many languages not be simplified at the source level, but the assignment might be translated to the following intermediate-code sequence:

```
t1 := 4*i
t2 := a+t1
t3 := M[t2]
t4 := t3+1
t5 := 4*i
t6 := a+t5
M[t6] := t4
```

Note that both the multiplication by 4 and the addition of `a` is repeated, but that while we can see that the expressions `a+t1` and `a+t5` have the same value, they are not textually identical.

Our ultimate goal is to eliminate both of these redundancies, but we will start by a simpler analysis that only catches the second occurrence of `4*i` and then discuss how it can be extended to also eliminate `a+t5`.

10.2.1 Available assignments

If we want to replace an expression by a variable that holds the value of the expression, we need to keep track of which expressions we have available and which variables hold their values. So we want at each

program point a set of pairs of variables and expressions. Since each such pair originates in an assignment of the expression to the variable, we can, equivalently, use a set of assignment instructions that occur in the program. This makes things slightly simpler later on. We call this analysis *available assignments*

It is clear that information flows from assignment forwards, so for each instruction in the program, the *in* set is the set of available assignments before the instruction is executed and the *out* set is the set of assignments available afterwards. The *gen* and *kill* sets should, hence, describe which new assignments become available and which are no longer available. An assignment makes itself available, *unless* the variable on the left-hand side also occurs on the right-hand side (because the assignment would make a new occurrence of the expression have a different value). All other assignments where the variable on the left-hand side of the instruction occurs (on either side) are invalidated: If the variable occurs on the left-hand side on another assignment, the variable no longer holds the value of the expression on the right-hand side, and if the variable occurs in an expression, the expression changes value, so the left-hand-side variable no longer holds the current value of the expression. Figure 10.1 shows the *gen* and *kill* sets for each kind of instruction in the intermediate language.

Note that a copy instruction $x := y$ doesn't generate any available assignment, as nothing is gained by replacing an occurrence of y by x. Note, also, that any store of a value to memory kills all instructions that load from memory. We need to make this rather conservative assumption because we don't know where in memory loads and stores go.

The next step is to define the equations for *in* and *out* sets:

$$out[i] = gen[i] \cup (in[i] \setminus kill[i]) \tag{10.1}$$

$$in[i] = \bigcap_{j \in pred[i]} out[j] \tag{10.2}$$

As mentioned above, the assignments that are available after an instruction (*i.e.*, in the *out* set) are those that are generated by the instruction and those that were available before, except for those that are killed by the instruction. The available assignments before an instruction (*i.e.*, in the *in* set) are those that are available at all predecessors, so we take the intersection of the sets available at the predecessors.

The predecessors $pred[i]$ of instruction i is $\{i-1\}$, except when i is a `LABEL` instruction, in which case we also add all `GOTO` and `IF` instructions

Instruction i	$gen[i]$	$kill[i]$
`LABEL` l	$\emptyset$	$\emptyset$
$x := y$	$\emptyset$	$assg(x)$
$x := k$	$\{x := k\}$	$assg(x)$
$x :=$ **unop** y where $x \neq y$	$\{x :=$ **unop** $y\}$	$assg(x)$
$x :=$ **unop** x	$\emptyset$	$assg(x)$
$x :=$ **unop** k	$\{x :=$ **unop** $k\}$	$assg(x)$
$x := y$ **binop** z where $x \neq y$ and $x \neq z$	$\{x := y$ **binop** $z\}$	$assg(x)$
$x := y$ **binop** z where $x = y$ or $x = z$	$\emptyset$	$assg(x)$
$x := y$ **binop** k where $x \neq y$	$\{x := y$ **binop** $k\}$	$assg(x)$
$x := x$ **binop** k	$\emptyset$	$assg(x)$
$x := M[y]$	$\{x := M[y]\}$	$assg(x)$
$x := M[k]$	$\{x := M[k]\}$	$assg(x)$
$M[x]$:= y	$\emptyset$	$loads$
$M[k]$:= y	$\emptyset$	$loads$
`GOTO` l	$\emptyset$	$\emptyset$
`IF` x **relop** y `THEN` l_t `ELSE` l_f	$\emptyset$	$\emptyset$
$x :=$ `CALL` $f(args)$	$\emptyset$	$\emptyset$

where $assg(x)$ is the set of all assignments in which x occurs on either left-hand or right-hand side, and $loads$ is the set of all assignments of the form $x := M[\cdot]$.

Figure 10.1: Gen and kill sets for available assignments

that can jump to i, or when i is the first instruction in the program (*i.e.*, when $i = 1$), in which case $i - 1$ is not in the predecessors of i.

We need to initialise the *in* and *out* sets for the fixed-point iteration. The most immediate choice would be to initialise all to the empty set, but that would not give a useful result. Consider a loop, where an assignment is available before the loop and no variable in the assignment is changed inside the loop. We would want the assignment to be available inside the loop, but if we initialise the *out* set of the jump from the end of the loop to its beginning to the empty set, the intersection of the assignments available before the loop and those available at the end of the loop will be empty, and remain that way throughout the iteration. So, instead we initialise the *in* and *out* sets for all instructions except the first to the set of all assignments in the program. The *in* set for the first instruction remains empty, as no assignments is available at the beginning of the program.

When an analysis takes the intersection of the values for the predecessors, we will always initialise sets (except for the first instruction) to the largest possible, so we don't get overly conservative results for loops.

10.2.2 Example of available-assignments analysis

Figure 10.2 shows a program that doubles elements of an array p

Figure 10.3 shows *pred*, *gen* and *kill* sets for each instruction in this program. We represent an assignment by the number of the assignment instruction, so *gen* and *kill* sets are sets of numbers. We will, however, identity identical assignments with the same number, so both the assignment in instruction 2 and the assignment in instruction 13 are identified with the number 2, as can be seen in the *gen* set of instruction 13.

Note that each assignment kills itself, but since it also (in most cases) generates itself, the net effect is to remove all conflicting assignments (including itself) and then adding itself. Assignment 12 ($i := i+1$) does not generate itself, since i also occurs on the right-hand side. Note, also, that the write to memory in instruction 11 kills instruction 7, as this loads from memory.

For the fixed-point iteration we initialise the *in* set of instruction 1 to the empty set and all other *in* and *out* sets to the set of all assignments. We need only the assignments that area actually generated by some instructions, *i.e.*, $\{1, 2, 5, 6, 7, 8, 9, 10\}$. We then iterate equations 10.2 and 10.1 as assignments until we reach a fixed-point. Since information flow is forwards, we process the instructions by increasing number and calculate $in[i]$ before $out[i]$. The iteration is shown in figure 10.4. For

```
1:  i := 0
2:  a := n * 3
3:  IF i < a THEN loop ELSE end
4:  LABEL loop
5:  b := i * 4
6:  c := p + b
7:  d := M[c]
8:  e := d * 2
9:  f := i * 4
10: g := p + f
11: M[g] := e
12: i := i + 1
13: a := n * 3
14: IF i < a THEN loop ELSE end
15: LABEL end
```

Figure 10.2: Example program for available-assignments analysis

i	$pred[i]$	$gen[i]$	$kill[i]$
1		1	1, 5, 8, 11
2	1	2	2
3	2		
4	3, 14		
5	4	5	5, 6
6	5	6	6, 7
7	6	7	7, 8
8	7	8	8
9	8	9	9, 10
10	9	10	10
11	10		7
12	11		1, 5, 9
13	12	2	2
14	13		
15	3, 14		

Figure 10.3: *pred*, *gen* and *kill* for the program in figure 10.2

space reasons, the table is shown sideways and the final iteration (which is identical to iteration 2) is not shown.

10.2.3 Using available assignments for common subexpression elimination

If instruction i is of the form $x := e$ for some expression e and $in[i]$ contains an assignment $y := e$, then we can replace $x := e$ by $x := y$.

If we apply this idea to the program in figure 10.2, we see that at instruction 9 ($f := i * 4$), we have assignment 5 ($b := i * 4$) available, so we can replace instruction 9 by ($f := b$). At instruction 13 ($a := n * 3$), we have assignment 2 ($a := n*3$) available, so we can replace instruction 13 by $a := a$, which we can omit entirely as it is a no-operation. The optimised program is shown in figure 10.5.
Note that while we could eliminate identical expressions, we could not eliminate the expression $p + f$ in instruction 10 even though it has the same value as the right-hand side of the available assignment 6 ($c := p+b$), since $b = f$. A way of achieving this is to first replace all uses of f by b (which we can do since the only assignment to f is $f := b$) and then repeat common subexpression elimination on the resulting program. If a large expression has two occurrences, we might have to repeat this a large number of times to get the optimal result. An alternative is to keep track of sets of variables that have the same value (a technique called *value numbering*), which allows large common subexpressions to be eliminated in one pass.

Another limitation of the available assignment analysis is when two different predecessors to an instruction have the same expression available but in different variables, *e.g.*, if one predecessor of instruction i has the available assignments $\{x := a+b\}$ and the other predecessor has the available assignments $\{y := a+b\}$. These have an empty intersection, so the analysis would have no available assignments at the entry of instruction i. It is possible to make common subexpression elimination make the expression $a + b$ available in this situation, but this makes analysis and transformation more complex.

10.3 Jump-to-jump elimination

When we have an instruction sequence like

```
LABEL l1
GOTO l2
```

i	Initialisation		Iteration 1		Iteration 2	
	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$
1		$1,2,5,6,7,8,9,10$		1		1
2	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	1	$1,2$	1	$1,2$
3	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2$	$1,2$	$1,2$	$1,2$
4	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2$	$1,2$	2	2
5	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2$	$1,2,5$	2	$2,5$
6	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5$	$1,2,5,6$	$2,5$	$2,5,6$
7	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6$	$1,2,5,6,7$	$2,5,6$	$2,5,6,7$
8	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,7$	$1,2,5,6,7,8$	$2,5,6,7$	$2,5,6,7,8$
9	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8$	$1,2,5,6,7,8,9$	$2,5,6,7,8$	$2,5,6,7,8,9$
10	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9$	$1,2,5,6,7,8,9,10$	$2,5,6,7,8,9$	$2,5,6,7,8,9,10$
11	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,8,9,10$	$2,5,6,7,8,9,10$	$2,5,6,8,9,10$
12	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$1,2,5,6,8,9,10$	$2,6,8,10$	$2,5,6,8,9,10$	$2,6,8,10$
13	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$2,6,8,10$	$2,6,8,10$	$2,6,8,10$	$2,6,8,10$
14	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	$2,6,8,10$	$2,6,8,10$	$2,6,8,10$	$2,6,8,10$
15	$1,2,5,6,7,8,9,10$	$1,2,5,6,7,8,9,10$	2	2	2	2

Figure 10.4: Fixed-point iteration for available-assignment analysis

```
 1: i := 0
 2: a := n * 3
 3: IF i < a THEN loop ELSE end
 4: LABEL loop
 5: b := i * 4
 6: c := p + b
 7: d := M[c]
 8: e := d * 2
 9: f := b
10: g := p + f
11: M[g] := e
12: i := i + 1
14: IF i < a THEN loop ELSE end
15: LABEL end
```

Figure 10.5: The program in figure 10.2 after common subexpression elimination.

we would like to replace all jumps to l_1 by jumps to l_2. However, there might be chains of such jumps, *e.g.*,

```
LABEL l1
GOTO l2
...
LABEL l2
GOTO l3
...
LABEL l3
GOTO l4
```

We want, in this example, to replace a jump to l_1 with a call to l_4 directly. To do this, we make a data-flow analysis that for each jump finds its ultimate destination. The analysis is a backwards analysis, as information flows from a label back to the instruction that jumps to it. The *gen* and *kill* sets are quite simple:

instruction	*gen*	*kill*
`LABEL` l	$\{l\}$	$\emptyset$
`GOTO` l	$\emptyset$	$\emptyset$
`IF` c `THEN` l_1 `ELSE` l_2	$\emptyset$	$\emptyset$
any other	$\emptyset$	the set of all labels

The equations for *in* and *out* are:

$$in[i] = \begin{cases} gen[i] \setminus kill[i] & \text{if } out[i] \text{ is empty} \\ out[i] \setminus kill[i] & \text{if } out[i] \text{ is non-empty} \end{cases} \tag{10.3}$$

$$out[i] = \bigcap_{j \in succ[i]} in[j] \tag{10.4}$$

We initialise all *out* sets to the set of all labels, except the *out* set of the last instruction (which has no successors), which is initialised to empty.

Note that, after the fixed-point iteration, $in[i]$ can only have more than one element if i is part of or jumps to a loop consisting entirely of `GOTO` instructions and labels. Such infinite loops can occur in valid programs (*e.g.*, as an idle-loop that waits for an interrupt), so we allow this.

When we have reached a fixed-point, we can do the following optimisations:

- At `GOTO` i, we can replace i by any element in $in[i]$.
- If instruction i is `LABEL` l and $l \notin in[i]$, there will be no jumps to l after the optimisation, so we can remove instruction i and all following instructions up to just before the next label. This is called *dead code elimination*.
- At `IF` c `THEN` i `ELSE` j, we can replace i by any element in $in[i]$ and j by any element in $in[j]$. If $in[i] = in[j]$, the test is redundant, and we can replace `IF` c `THEN` i `ELSE` j by `GOTO` l, where l is any element in $in[i]$.

10.4 Index-check elimination

When a language requires bounds-checking of array accesses, the compiler must insert tests before each array access to verify that the index is in range. Index-check elimination aims to remove these checks when they are redundant.

To find this, we collect for each program point a set of inequalities that hold at this point. If a bounds check is implied by these inequalities, we can eliminate it.

We will use the conditions in `IF-THEN-ELSE` instructions as our source for inequalities. Note that, since bounds checks are translated into such instructions, this set of conditions includes all bounds checks.

Conditions all have the form x **relop** p, where x is a variable and p is either a constant or a variable. We only care about inequalities, *i.e.*, conditions of the form $p < q$ or $p \leq q$, where p and q are either variables or constants (but they can not both be constants). Each condition that occurs in the program is translated into a set of inequalities of this form. For example $x = y$ is translated into $x \leq y$ and $y \leq x$. We also generate inequalities for the negation of each condition in the program, so a condition like $x < 10$ generates the inequalities $x < 10$ and $10 \leq x$. A condition $x = y$ generates no inequalities for its negation, as $x \neq y$ can not be expressed as a conjunction of inequalities. This gives us a universe Q of inequalities that we work on. At each point in the program, we want to find which of these inequalities are guaranteed to hold at this point, *i.e.*, a subset of Q.

The idea is that, after executing the instruction `IF` c `THEN` l_1 `ELSE` l_2, the conditions derived from c will be true at l_1 and the conditions derived from the negation of c will be true at l_2, *assuming* there are no other jumps to l_1 or l_2. To ensure this assumption, we insert extra labels and jumps in the program: each instruction of the form `IF` c `THEN` l_1 `ELSE` l_2 is replaced by the sequence

```
IF c THEN t ELSE f
LABEL t
GOTO l1
LABEL f
GOTO l2
```

where t and f are new labels that do not occur anywhere else.

This way, if a label has more than one predecessor, these will all be `GOTO` statements (that do not alter the set of valid inequalities). We can later remove the added code by jump-to-jump elimination as described in section 10.3.

We don't use *out* sets, but write the equation for *in* sets in terms of the *in* sets of the predecessors and the type of instruction that the predecessor is shown in figure 10.6. We use the following auxiliary definitions:

- $when(c)$ is the set of inequalities implied by the condition c. These will be elements of Q (the universe of inequalities), since Q was constructed to include all these.

- $whennot(c)$ is the set of inequalities implied by the negation of the condition c. These will, likewise, be elements of Q.

$$in[i] = \begin{cases} \bigcap_{j \in pred[i]} in[j] \\ \qquad \text{if } pred[i] \text{ has more than one element} \\ in[pred[i]] \cup when(c) \\ \qquad \text{if } pred[i] \text{ is } \texttt{IF}\, c\, \texttt{THEN}\, i\, \texttt{ELSE}\, j \\ in[pred[i]] \cup whennot(c) \\ \qquad \text{if } pred[i] \text{ is } \texttt{IF}\, c\, \texttt{THEN}\, j\, \texttt{ELSE}\, i \\ (in[pred[i]] \setminus conds(Q,x)) \cup equal(Q,x,p) \\ \qquad \text{if } pred[i] \text{ is of the form } x := p \\ in[pred[i]] \setminus upper(Q,x) \\ \qquad \text{if } pred[i] \text{ is of the form } x := x + k \text{ where } k \geq 0 \\ in[pred[i]] \setminus lower(Q,x) \\ \qquad \text{if } pred[i] \text{ is of the form } x := x - k \text{ where } k \geq 0 \\ in[pred[i]] \setminus conds(Q,x) \\ \qquad \text{if } pred[i] \text{ is of a form } x := e \text{ not covered above} \\ in[pred[i]] \\ \qquad otherwise \end{cases}$$

Figure 10.6: Equations for index-check elimination

- $conds(Q,x)$ is the set of inequalities from Q that involve x.
- $equal(Q,x,p)$, where p is a variable or a constant, is the set of inequalities from Q that are implied by the equality $x = p$. For example, if $Q = \{x < 10,\ 10 \leq x,\ 0 < x, x \leq 0\}$ then $equal(Q,x,7) = \{x < 10,\ 0 < x\}$.
- $upper(Q,x)$ is the set of inequalities from Q that have the form $p < x$ or $p \leq x$, where p is a variable or a constant.
- $lower(Q,x)$ is the set of inequalities from Q have the form $x < p$ or $x \leq p$, where p is a variable or a constant.

Normally, any assignment to a variable invalidates all inequalities involving that variable, but we have made some exceptions: If we assign a constant or variable to a variable, we check all the possible inequalities and add those that are implied by the assignment. Also, if x increases, we invalidate all inequalities that bound x from above but keep those that bound x from below, and if x decreases, we invalidate the inequalities that bound x from below but keep those that bound x from above. We can add more special cases to make the analysis more precise, but the above are sufficient for the most common cases.

1: $i := 0$
2: `LABEL` $for1$
3: `IF` $i \leq 9$ `THEN` $for2$ `ELSE` $for3$
4: `LABEL` $for2$
5: `IF` $i < 0$ `THEN` $error$ `ELSE` $ok1$
6: `LABEL` $ok1$
7: `IF` $i > 10$ `THEN` $error$ `ELSE` $ok2$
8: `LABEL` $ok2$
9: $t := i * 4$
10: $M[t] := 0$
11: $i := i + 1$
12: `GOTO` $for1$
13: `LABEL` $for3$

Figure 10.7: Intermediate code for for-loop with index check

We initialise all in sets to Q, except the in set for the first instruction, which is initialised to the empty set.

After the data-flow analysis reaches a fixed-point, the inequalities in $in[i]$ are guaranteed to hold at instruction i. So, if we have an instruction i of the form `IF` c `THEN` l_t `ELSE` l_f and c is implied by the inequalities in $in[i]$, we can replace the instruction by `GOTO` l_t. If the negation of c is implied by the inequalities in $in[i]$, we can replace the instruction by `GOTO` l_f.

We illustrate the analysis by an example. Consider the following for-loop and assume that the array `a` is declared to go from 0 to 10.

```
for i:=0 to 9 do
  a[i] := 0;
```

This loop can be translated (with index check) into the intermediate code shown in figure 10.7.

The set Q of possible inequalities in the program are derived from the conditions in the three `IF-THEN-ELSE` instructions and their negations, *i.e.*, $Q = \{i \leq 9, i > 9, i < 0, i \geq 0, i > 10, i \leq 10\}$.

We leave the fixed-point iteration and check elimination as an exercise to the reader, but note that the assignment $i := 0$ in instruction 1 implies the inequalities $\{i \leq 9, i \geq 0, i \leq 10\}$ and that the assignment $i := i + 1$ in instruction 11 preserves $i \geq 0$ but invalidates $i \leq 9$ and $i \leq 10$.

10.5 Limitations of data-flow analyses

All of the data-flow analyses we have seen above are approximations: They will not always accurately reflect what happens at runtime: The index-check analysis may fail to remove a redundant index check, and the available assignment analysis may say an assignment is unavailable when, in fact, it is available.

In all cases, the approximations err on the safe side: It is better to miss an opportunity for optimisation than to make an incorrect optimisation. For liveness analysis, this means that if you are in doubt about a variable being live, you had better say that it is, as assuming it dead might cause its value to be overwritten. When available assignment analysis is used for common subexpression elimination, saying that an assigning is available when it isn't may make the optimisation replace an expression by a variable that does not always hold the same value as the expression, so it is better to leave an assignment out of the set if you are in doubt.

It can be shown that no compile-time analysis that seeks to uncover nontrivial information about the run-time behaviour of programs can ever be completely exact. You can make more and more complex analyses that get closer and closer to the exact result, but there will always be programs where the analysis isn't precise. So a compiler writer will have to be satisfied with analyses that find most cases where an optimisation can be applied, but misses some.

10.6 Loop optimisations

Since many programs spend most of their time in loops, it is worthwhile to study optimisations specific for loops.

10.6.1 code hoisting

One such optimisation is recognising computations that are repeated in every iteration of the loop without changing the values involved, *i.e.,* loop-invariant computations. We want to lift such computations outside the loop, so they are performed only once. This is called *code hoisting.*

We saw an example of this in section 10.2.3, where calculation of $n * 3$ was done once before the loop and subsequent re-computations were replaced by a reference to the variable a that holds the value of $n * 3$ computed before the loop. However, it is only because there was an explicit computation of $n * 3$ before the loop, that we could avoid

re-computation inside the loop: Otherwise, the occurrence of $n*3$ inside the loop would not have any available assignment that can replace the calculation.

So our aim is to move or copy loop-invariant assignments to before the loop, so their result can be reused inside the loop. Moving a computation to before the loop may, however, cause it to be computed even when the loop is not entered. In addition to causing unnecessary computation (which goes against the wish for optimisation), such computations can cause errors when the precondition (the loop condition) is not satisfied. For example, if the invariant computation is a memory access, the address may be valid only if the loop is entered.

A common solution to this problem is to unroll the loop once: A loop of the form (using C-like syntax):

```
while (cond) {
  body
}
```

is transformed to

```
if (cond) then {
  body
  while (cond) {
    body
  }
}
```

Similarly, a test-at-bottom loop of the form

```
do
  body
while (cond)
```

can be unrolled to

```
body
while (cond) {
  body
}
```

Now, we can safely calculate the invariant parts in the first copy of the body and reuse the results in the loop. If the compiler does common subexpression elimination, this unrolling is all that is required to do code

hoisting – assuming the unrolling is done before common-subexpression elimination. Unrolling of loops is most easily done at source-code level (*i.e.*, on the abstract syntax tree), so this is no problem. This unrolling will, of course, increase the size of the compiled program, so it should be done with care if the loop body is large.

10.6.2 Memory prefetching

If a loop goes through a large array, it is likely that parts of the array will not be in the cache of the processor. Since access to non-cached memory is *much* slower than access to cached memory, we would like to avoid this.

Many modern processors have *memory prefetch instructions* that tell the processor to load the contents of an address into cache, but unlike a normal load, a memory prefetch doesn't cause errors if the address is invalid, and it returns immediately without waiting for the load to complete. So a way to ensure that an array element is in the cache is to issue a prefetch of the array element well in advance of its use, but not so well in advance that it is likely that it will be evicted from the cache between the prefetch and the use. Given modern cache sizes and timings, 25 to 10000 cycles ahead of the use is a reasonable time for prefetching – less than 25 increases the risk that the prefects is not completed before the use, and more than 10000 increases the chance that the value will be evicted from the cache before use.

A prefetch instruction usually loads an entire cache line, which is typically four or eight words, so we don't have to explicitly prefetch every array element – every fourth element is enough.

So, assume we have a loop that adds the elements of an array:

```
sum = 0;
for (i=0; i<100000; i++)
  sum += a[i];
}
```

we can rewrite this to

```
sum = 0;
for (i=0; i<100000; i++) {
  if (i&3 == 0) prefetch a[i+32];
  sum += a[i];
}
```

where `prefetch a[i+32]` prefetches the element of `a` that is 32 places after the current element. The number 32 is rather arbitrary, but makes the number of cycles between prefetch and use lie in the interval mentioned above. Note that we used the test `i&3==0`, which is equivalent to `i%4==0`, but somewhat faster.

We don't have to worry about prefetching past the end of the array – prefetching will never cause runtime errors, so at worst we prefetch something that we don't need.

While this transformation adds a test (that takes time), the potential savings by having all array elements in cache before use are much larger. The overhead of testing can be reduced by unrolling the loop body:

```
sum = 0;
for (i=0; i<100000; i++) {
  prefetch a[i+32];
  sum += a[i];
  i++;
  sum += a[i];
  i++;
  sum += a[i];
  i++;
  sum += a[i];
}
```

This should, of course, only be done if the loop body is small. We have exploited that the number of iterations is a multiple of 4, so the exit test is not needed at every increment of `i`. If we don't know this, the exit test must be replicated after each increase of `i`, like shown here:

```
sum = 0;
for (i=0; i<n; i++) {
  prefetch a[i+32];
  sum += a[i];
  if (++i < n) {
    sum += a[i];
    if (++i < n) {
      sum += a[i];
      if (++i < n) {
        sum += a[i];
      }
    }
  }
}
```

In a nested loop that accesses a multi-dimensional array, you can prefetch the next row while processing the current. For example, the loop

```
sum = 0;
for (i=0; i<1000; i++)
  for (j=0; j<1000; i++)
    sum += a[i][j];
  }
}
```

can be transformed to

```
sum = 0;
for (i=0; i<1000; i++)
  for (j=0; j<1000; i++)
    if (j&3 == 0) prefetch a[i+1][j];
    sum += a[i][j];
  }
}
```

Again, we can unroll the body of the inner loop to reduce the overhead.

10.7 Optimisations for function calls

Modern coding styles use frequent function (or method) calls, so optimising function calls is as worthwhile as optimising loops.

Basically, optimisation of function calls attempt to reduce the overhead associated with call sequences, prologues and epilogues (see chapter 9). We will see a few ways of doing this below.

10.7.1 Inlining

Inlining means replacing a function call by a copy of the body of the function, with some glue code to replace parameter and result passing.

If we have a call (using C-style syntax)

$$x \texttt{ = f(}exp_1\texttt{,...,}exp_n\texttt{);}$$

and the function `f` is defined as

```
type0 f(type1 x1,...,typen xn)
{
  body
  return(exp);
}
```

where *body* represents the body of the function (apart from the `return` statement) and the shown `return` statement is the only exit point of the function, we can replace the call by the block

```
{
  type1 x1 = exp1;
  ...
  typen xn = expn;
  body
  x = exp;
}
```

Note that if, say, exp_n refers to a variable with the same name as x_1, it will refer to a different instance of x_1 after the transformation. So we need to avoid variables in the inlined function shadowing variables used in the function call statement. We can achieve this by renaming the variables in the inlined function to new, previously unused, names before it is inlined.

Note that, unless the body of the inlined function is very small, inlining causes the program to grow in size. Hence, it is common to put a limit on the size of functions that are inlined. What the limit is depends on the desired balance between speed and size.

Care must be taken if you inline calls recursively: If the inlined body contains a call that is also inlined, and this again contains a call that is inlined, and so on, we might continue inlining forever. So it is common to limit inlining to only one or two levels deep or treat (mutually) recursive functions as special cases.

10.7.2 Tail call optimisation

A *tail call* is a call that happens just before a return.

As an example, assume we in a function `f` have (using C-style notation) a statement `return(g(x,y));`. Clearly, `f` returns just after `g` returns, and the result of `f` is the result of `g`. We want to combine the call sequence for the call to `g` with the epilogue of `f`. We call this *tail call optimisation*.

We will exploit the following observations:

- No variables in `f` are live after the call to `g` (since there aren't any uses of variables after the call), so there will be no need to save and restore caller-saves variables around the call.

- If we can eliminate all of the epilogue of `f` except for the return-jump to `f`'s caller, we can instead make `g` return directly to `f`'s caller, hence skipping `f`'s epilogue entirely.

- If `f`'s frame holds no useful information at the time we call `g` (or we can be sure that `g` does not overwrite any useful information in the frame), we can reuse the frame for `f` as the frame for `g`.

We will look at more detail at this below.

If we assume a simple stack-based caller-saves strategy like the one shown in figures 9.2 and 9.3, the combined call sequence of the call to `g` and the epilogue of `f` becomes:

$$
\begin{array}{l}
M[FP + 4 * m + 4] := R0 \\
\cdots \\
M[FP + 4 * m + 4 * (k + 1)] := Rk \\
FP := FP + framesize \\
M[FP + 4] := a_1 \\
\cdots \\
M[FP + 4 * n] := a_n \\
M[FP] := returnaddress \\
\texttt{GOTO g} \\
\texttt{LABEL } returnaddress \\
result := M[FP + 4] \\
FP := FP - framesize \\
R0 := M[FP + 4 * m + 4] \\
\cdots \\
Rk := M[FP + 4 * m + 4 * (k + 1)] \\
M[FP + 4] := result \\
\texttt{GOTO } M[FP]
\end{array}
$$

Since there are no live variables after the call to `g`, we can eliminate the saving and restoring of $R0, \ldots, Rk$, yielding the following simplified code:

$FP := FP + framesize$
$M[FP + 4] := a_1$
$\ldots$
$M[FP + 4 * n] := a_n$
$M[FP] := returnaddress$
`GOTO g`
`LABEL` $returnaddress$
$result := M[FP + 4]$
$FP := FP - framesize$
$M[FP + 4] := result$
`GOTO` $M[FP]$

We now see that all that happens after we return is an adjustment of FP, a copy of the result from `g`'s frame to `f`'s frame and a jump to the return address stored in `f`'s frame.

What we now want is to reuse `f`'s frame for `g`'s frame. We do this by not adding and subtracting $framesize$ from FP, so we get the following simplified code:

$M[FP + 4] := a_1$
$\ldots$
$M[FP + 4 * n] := a_n$
$M[FP] := returnaddress$
`GOTO g`
`LABEL` $returnaddress$
$result := M[FP + 4]$
$M[FP + 4] := result$
`GOTO` $M[FP]$

It is immediately evident that the two instructions that copy the result from and to the frame cancel out, so we can simplify further to

$M[FP + 4] := a_1$
$\ldots$
$M[FP + 4 * n] := a_n$
$M[FP] := returnaddress$
`GOTO g`
`LABEL` $returnaddress$
`GOTO` $M[FP]$

We also see an unfortunate problem: Just before the call to `g`, we overwrite `f`'s return address in $M[FP]$ by `g`'s return address, so we won't

correctly return to `f`'s caller (we will, instead, get an infinite loop). However, since all that happens after we return from `g` is a return from `f`, we can make `g` return directly to `f`'s caller. We do this simply by not overwriting `f`'s return address. This makes the instructions after the jump to `g` unreachable, so we can just delete them. This results in the following code:

$$
\begin{aligned}
&M[FP + 4] := a_1 \\
&\cdots \\
&M[FP + 4 * n] := a_n \\
&\texttt{GOTO g}
\end{aligned}
$$

By doing this, we have eliminated the potential ill effects of reusing `f`'s frame for `g`. Not only have we shortened the combined call sequence for `g` and epilogue for `f` considerably, we have also saved stack space. Functional programming languages rely on this space saving, as they often use recursive tail-calls where imperative languages use loops. By doing tail-call optimisation, an arbitrarily long sequence of recursive tail calls can share the same stack frame, so only a constant amount of stack space is used.

In the above, we relied on a pure caller-saves strategy, since the absence of live variables after the call meant that there would be no saving and restoring of caller-saves registers around the call. If a callee-saves strategy or a mixed caller-saves/callee-saves strategy is used, there will still be no saving and restoring around the call, but `f`'s epilogue would restore the callee-saves registers that `f` saved in its own prologue. This makes tail-call optimisation a bit more complicated, but it is normally possible (with a bit of care) to move the restoring of callee-saves registers to *before* the call to `g` instead of waiting until after `g` returns. This allows `g` to overwrite `f`'s frame, which no longer holds any useful information (except the return address, which we explicitly avoid overwriting).

Exercise 9.4 asks you to do such tail-call optimisation for a mixed caller-saves/callee-saves strategy.

10.8 Specialisation

Modern programs consist mainly of calls to predefined library functions (or procedures or methods). Calls to such library functions often fix some of the parameters to constants, as the function is written more generally than needed for the particular call. For example, a function that raises a number to a power is often called with a constant as the

power, say, `power(x,5)` which raises `x` to its fifth power. In such cases, the generality of the library function is wasted, and speed can be gained by using a more specific function, say, one that raises its argument to the fifth power.

A possible implementation of the general `power` function (in C) is:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

If we have a call `power(x,5)`, we can replace this by a call `power5(x)` to a specialised function. We now need to add a definition of this specialised function to the program. The most obvious idea would be to take the above code for the `power` function and replace all occurrences of `n` by 5, but this won't work, as `n` changes value inside the body of `power`. What we do instead is to observe the following:

1. The loop condition depends only on `n`.
2. Every change to `n` depends only on `n`.

So it is safe to unroll the loop at compile-time, doing all the computations on `n`, but leaving the computations on `p` and `x` for run-time. This yields the following specialised definition:

```
double power5(double x)
{
  double p=1.0;
  p = p*x;
  x = x*x;
  x = x*x;
  p = p*x;
  return(p);
}
```

Executing `power5(x)` is, obviously, a lot faster than executing `power(x,5)`. Since `power5` is fairly small, we can additionally inline the call, as described in section 10.7.1.

This kind of specialisation may not always be applicable, even if a function call has constant parameters, for example if the call was `power(3.14159,p)`, where `p` is not a constant, but when the method is applicable, the speedup can be dramatic.

Similar specialisation techniques are used in C++ compilers for compiling templates: When a call specifies template parameters, the definition of the template is specialised with respect to the actual template parameters. Since templates in C++ can be recursively defined, an infinite number of specialised versions might be required. Most C++ compilers put a limit on the recursion level of template instantiations and stop with an error message when this limit is exceeded.

10.9 Further reading

We have covered only a small portion of the optimisations that are found in optimising compilers. More examples of optimisations (including value numbering) can be found in advanced compiler textbooks, such as [4, 7, 9, 30].

A detailed treatment of program analysis can be found in [31]. Specialisation techniques like those mentioned in section 10.8 are covered in more detail in [16, 20]. The book [29] has good articles on both program analysis and transformation.

Additionally, the conferences "Compiler Construction" (CC), "Programming Language Design and Implementation" (PLDI) and other programming-language-oriented conferences often present new optimisation techniques, so past proceedings from these is a good source for advanced optimisation methods.

Exercises

Exercise 10.1

In the program in figure 10.2, replace instructions 13 and 14 by

13: $h := n * 3$
14: `IF` $i < h$ `THEN` *loop* `ELSE` *end*

a) Repeat common subexpression elimination on this modified program.

b) Repeat, again, common subexpression elimination on the modified program, but, prior to the fixed-point iteration, initialise all sets to the empty set instead of the set of all assignments.

   What differences does this make to the final result of fixed-point iteration, and what consequences do these differences have for the optimisation?

Exercise 10.2

Write a program that has jumps to jumps and perform jump-to-jump optimisation of it as described in section 10.3. Try to make the program cover all the three optimisation cases described at the end of section 10.3.

Exercise 10.3

a) As described in the beginning of section 10.4, add extra labels and gotos for each `IF-THEN-ELSE` in the program in figure 10.7.

b) Do the fixed-point iteration for index-check elimination on the result.

c) Eliminate the redundant tests.

d) Do jump-to-jump elimination as described in section 10.3 on the result to remove the extra labels and gotos introduced in question a.

Exercise 10.4

Section 10.7.2 describes tail-call optimisation for a pure caller-saves strategy. Things become somewhat more complicated when you use a mixed caller-saves/callee-saves strategy.

Using the call sequence from figure 9.10 and the epilogue from figure 9.9, describe how the combined sequence can be rewritten to get some degree of tail-call optimisation. Your main focus should be on reusing stack space, and secondarily on saving time.

Exercise 10.5

Specialise the `power` function in section 10.8 to `n` $= 12$.

Chapter 11

Bootstrapping a compiler

11.1 Introduction

When writing a compiler, one will usually prefer to write it in a high-level language. A possible choice is to use a language that is already available on the machine where the compiler should eventually run. It is, however, quite common to be in the following situation:

You have a completely new processor for which no compilers exist yet. Nevertheless, you want to have a compiler that not only targets this processor, but also runs on it. In other words, you want to write a compiler for a language A, targeting language B (the machine language) and written in language B.

The most obvious approach is to write the compiler in language B. But if B is machine language, it is a horrible job to write any non-trivial compiler in this language. Instead, it is customary to use a process called "bootstrapping", referring to the seemingly impossible task of pulling oneself up by the bootstraps.

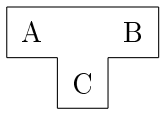
The idea of bootstrapping is simple: You write your compiler in language A (but still let it target B) and then let it compile itself. The result is a compiler from A to B written in B.

It may sound a bit paradoxical to let the compiler compile itself: In order to use the compiler to compile a program, we must already have compiled it, and to do this we must use the compiler. In a way, it is a bit like the chicken-and-egg paradox. We shall shortly see how this apparent paradox is resolved, but first we will introduce some useful notation.

11.2 Notation

We will use a notation designed by H. Bratman [10]. The notation is hence called "Bratman diagrams" or, because of their shape, "T-diagrams".

In this notation, a compiler written in language C, compiling from the language A and targeting the language B is represented by the diagram



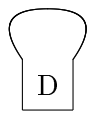
In order to use this compiler, it must "stand" on a solid foundation, *i.e.*, something capable of executing programs written in the language C. This "something" can be a machine that executes C as machine-code or an interpreter for C running on some other machine or interpreter. Any number of interpreters can be put on top of each other, but at the bottom of it all, we need a "real" machine.

An interpreter written in the language D and interpreting the language C, is represented by the diagram

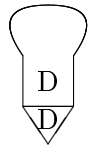
A machine that directly executes language D is written as

The pointed bottom indicates that a machine need not stand on anything; it is itself the foundation that other things must stand on.

When we want to represent an unspecified program (which can be a compiler, an interpreter or something else entirely) written in language D, we write it as

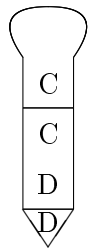


These figures can be combined to represent executions of programs. For example, running a program on a machine D is written as



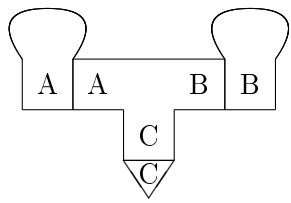
Note that the languages must match: The program must be written in the language that the machine executes.

We can insert an interpreter into this picture:



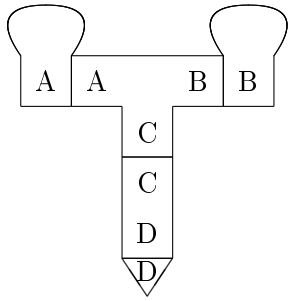
Note that, also here, the languages must match: The interpreter can only interpret programs written in the language that it interprets.

We can run a compiler and use this to compile a program:



The input to the compiler (*i.e.*, the source program) is shown at the left of the compiler, and the resulting output (*i.e.*, the target program) is shown on the right. Note that the languages match at every connection and that the source and target program aren't "standing" on anything, as they aren't executed in this diagram.

We can insert an interpreter in the above diagram:



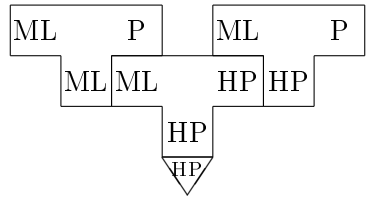
11.3 Compiling compilers

The basic idea in bootstrapping is to use compilers to compile themselves or other compilers. We do, however, need a solid foundation in form of a machine to run the compilers on.

It often happens that a compiler does exist for the desired source language, it just doesn't run on the desired machine. Let us, for example, assume we want a compiler for ML to Pentium machine code and want this to run on a Pentium. We have access to an ML-compiler that generates HP PA-RISC machine code and runs on an HP machine, which we also have access to. One way of obtaining the desired compiler would be to do *binary translation*, *i.e.*, to write a compiler from HP machine code to Pentium machine code. This will allow the translated compiler to run on a Pentium, but it will still generate HP code. We can use the HP-to-Pentium compiler to translate this into Pentium code afterwards, but this introduces several problems:

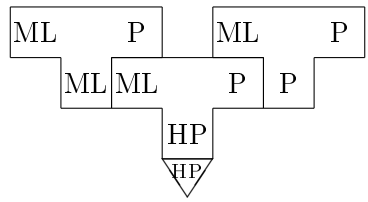
- Adding an extra pass makes the compilation process take longer.
- Some efficiency will be lost in the translation.
- We still need to make the HP-to-Pentium compiler run on the Pentium machine.

A better solution is to write an ML-to-Pentium compiler in ML. We can compile this using the ML compiler on the HP:



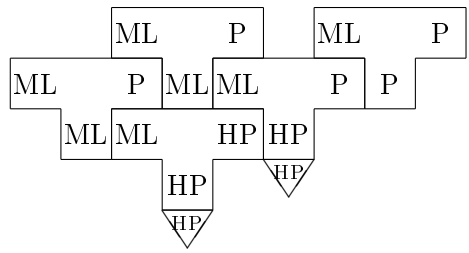
where "P" is short for Pentium.

Now, we can run the ML-to-Pentium compiler on the HP and let it compile itself[1]:



We have now obtained the desired compiler. Note that the compiler can now be used to compile itself directly on the Pentium platform. This can be useful if the compiler is later extended or, simply, as a partial test of correctness: If the compiler, when compiling itself, yields a different object code than the one obtained with the above process, it must contain an error. The converse isn't true: Even if the same target is obtained, there may still be errors in the compiler.

It is possible to combine the two above diagrams to a single diagram that covers both executions:



In this diagram, the ML-to-Pentium compiler written in HP has two roles: It is the output of the first compilation and the compiler that runs the second compilation. Such combinations can, however, be a bit confusing: The compiler that is the input to the second compilation step looks like it is also the output of the leftmost compiler. In this case,

[1]When a program is compiled and hence, strictly speaking, isn't textually the same, we still regard it as the same program.

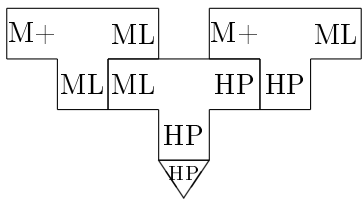
the confusion is avoided because the leftmost compiler isn't running and because the languages doesn't match. Still, diagrams that combine several executions should be used with care.

11.3.1 Full bootstrap

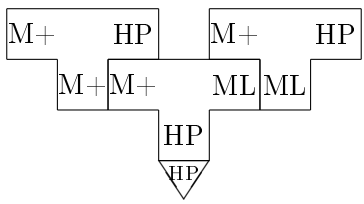
The above bootstrapping process relies on an existing compiler for the desired language, albeit running on a different machine. It is, hence, often called "half bootstrapping". When no existing compiler is available, *e.g.*, when a new language has been designed, we need to use a more complicated process called "full bootstrapping".

A common method is to write a QAD ("quick and dirty") compiler using an existing language. This compiler needs not generate code for the desired target machine (as long as the generated code can be made to run on some existing platform), nor does it have to generate good code. The important thing is that it allows programs in the new language to be executed. Additionally, the "real" compiler is written in the new language and will be bootstrapped using the QAD compiler.

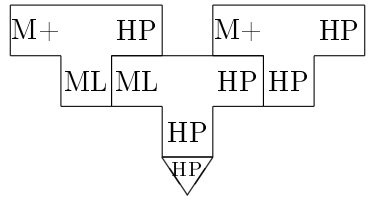
As an example, let us assume we design a new language "M+". We, initially, write a compiler from M+ to ML in ML. The first step is to compile this, so it can run on some machine:



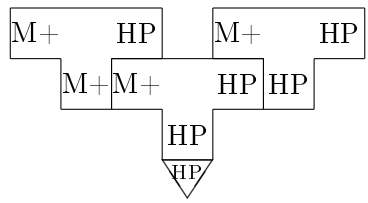
The QAD compiler can now be used to compile the "real" compiler:



The result is an ML program, which we need to compile:



The result of this is a compiler with the desired functionality, but it will probably run slowly. The reason is that it has been compiled by using the QAD compiler (in combination with the ML compiler). A better result can be obtained by letting the generated compiler compile itself:

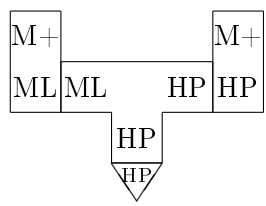


This yields a compiler with the same functionality as the above, *i.e.*, it will generate the same code, but, since the "real" compiler has been used to compile it, it will run faster.

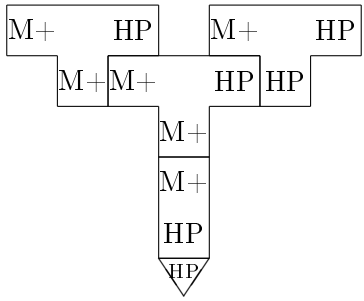
The need for this extra step might be a bit clearer if we had let the "real" compiler generate Pentium code instead, as it would then be obvious that the last step is required to get the compiler to run on the same machine that it targets. We chose the target language to make a point: Bootstrapping might not be complete even if a compiler with the right functionality has been obtained.

Using an interpreter

Instead of writing a QAD compiler, we can write a QAD interpreter. In our example, we could write an M+ interpreter in ML. We would first need to compile this:



We can then use this to run the M+ compiler directly:



Since the "real" compiler has been used to do the compilation, nothing will be gained by using the generated compiler to compile itself, though this step can still be used as a test and for extensions.

Though using an interpreter requires fewer steps, this shouldn't really be a consideration, as the computer(s) will do all the work in these steps. What is important is the amount of code that needs to be written by hand. For some languages, a QAD compiler will be easier to write than an interpreter, and for other languages an interpreter is easier. The relative ease/difficulty may also depend on the language used to implement the QAD interpreter/compiler.

Incremental bootstrapping

It is also possible to build the new language and its compiler incrementally. The first step is to write a compiler for a small subset of the language, using that same subset to write it. This first compiler must be bootstrapped in one of the ways described earlier, but thereafter the following process is done repeatedly:

1) Extend the language subset slightly.

2) Extend the compiler so it compiles the extended subset, but without using the new features.

3) Use the previous compiler to compile the new.

In each step, the features introduced in the previous step can be used in the compiler. Even when the full language is compiled, the process can be continued to improve the quality of the compiler.

11.4 Further reading

Bratman's original article, [10], only describes the T-shaped diagrams. The notation for interpreters, machines and unspecified programs was added later in [14].

The first Pascal compiler [36] was made using incremental bootstrapping.

Though we in section 11.3 dismissed binary translation as unsuitable for porting a compiler to a new machine, it is occasionally used. The advantage of this approach is that a single binary translator can port any number of programs, not just compilers. It was used by Digital Equipment Corporation in their FX!32 software [17] to enable programs compiled for Windows on a Pentium-platform to run on their Alpha RISC processor.

Exercises

Exercise 11.1

You have a machine that can execute *Alpha* machine code and the following programs:

1: A compiler from C to *Alpha* machine code written in *Alpha* machine code.

2: An interpreter for ML written in C.

3: A compiler from ML to C written in ML.

Now do the following:

a) Describe the above programs and machine as diagrams.

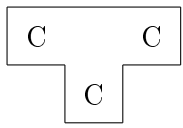
b) Show how a compiler from ML to C written in *Alpha* machine code can be generated from the above components. The generated program must be stand-alone, *i.e.*, it may not consist of an interpreter and an interpreted program.

c) Show how the compiler generated in question b can be used in a process that compiles ML programs to *Alpha* machine code.

Exercise 11.2

A source-code optimiser is a program that can optimise programs at source-code level, *i.e.*, a program O that reads a program P and outputs another program P', which is equivalent to P, but may be faster.

A source-code optimiser is like a compiler, except the source and target languages are the same. Hence, we can describe a source-code optimizer for C written in C with the diagram



Assume that you additionally have the following components:

- A compiler, written in ARM code, from C to ARM code.
- A machine that can execute ARM code.
- Some unspecified program P written in C.

Now do the following:

a) Describe the above components as diagrams.

b) Show, using Bratman diagrams, the steps required to optimise P to P' and then execute P'.

Bibliography

[1] A. Aasa. Precedences in specification and implementations of programming languages. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 183–194. Springer Verlag, 1991.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996. Also downloadable from `http://mitpress.mit.edu/sicp/full-text/sicp/book/`.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers; Principles, Techniques and Tools.* Addison-Wesley, 2007. Newer edition of [5].

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers; Principles, Techniques and Tools.* Addison-Wesley, 1986. Rereleased in extended form as [4].

[6] Hassan Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction.* MIT Press, 1991.

[7] John R. Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann, 2001.

[8] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[9] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[10] H. Bratman. An alternative form of the 'uncol' diagram. *Communications of the ACM*, 4(3):142, 1961.

[11] Preston Briggs. *Register Allocation via Graph Coloring, Tech. Rept. CPC-TR94517-S.* PhD thesis, Rice University, Center for Research on Parallel Computation, Apr. 1992.

[12] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 1(4):481–494, 1964.

[13] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, IT-2(3):113–124, 1956.

[14] J. Earley and H. Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13:607–617, 1970.

[15] Peter Naur (ed.). Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[16] John Hatcliff, Torben Mogensen, and Peter Thiemann (Eds.). *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[17] Raymond J. Hookway and Mark A. Herdeg. Digital fx!32: Combining emulation and binary translation. `http://www.cs.tufts.edu/comp/150PAT/optimization/DTJP01PF.pdf`, 1997.

[18] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation, 2nd ed.* Addison-Wesley, 2001.

[19] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report (2nd ed.).* Springer-Verlag, 1975.

[20] Neil D. Peyton Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[21] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages – A Tutorial.* Prentice Hall, 1992.

[22] J. P. Keller and R. Paige. Program derivation with verified transformations – a case study. *Communications in Pure and Applied Mathematics*, 48(9–10), 1996.

[23] B. W. Kerninghan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[24] M. E. Lesk. Lex: a Lexical Analyzer Generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[25] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd ed.* Addison-Wesley, Reading, Massachusetts, 1999.

[26] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[27] Robin Milner. A theory of type polymorphism in programming. *Journal of Computational Systems Science*, 17(3):348–375, 1978.

[28] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[29] Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors. *The essence of computation: complexity, analysis, transformation*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[30] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[31] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[32] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[33] David A. Patterson and John L. Hennessy. *Computer Organization & Design, the Hardware/Software Interface*. Morgan Kaufmann, 1998.

[34] Vern Paxson. Flex, version 2.5, a fast scanner generator. `http://www.gnu.org/software/flex/manual/html_mono/flex.html`, 1995.

[35] Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[36] Niklaus Wirth. The design of a pascal compiler. *Software - Practice and Experience*, 1(4):309–333, 1971.

Index