Basics of Compiler Design

Extended edition

Torben Ægidius Mogensen

Published through lulu.com.

© Torben Ægidius Mogensen 2000 – 2008

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

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 analysisand optimisation)was not found in editionsbefore April 2008, which is why the latesteditionsare called"extended"

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 " Over-sættere"at DIKU, and who have found numerous typos and other errors in the earlier versions. I would also like to thank the instructorsat Dat 1E and Oversættere, who have pointed out places where things were not as clear as they could be. I am extremelygratefulto the people who in 2000 read partsof or all of the first draft and made helpful suggestions.

1.8 Permission to use

Permission to copy and print for personal use is granted. If you, as a lecturer, want to print the book and sell it to your students, you can do so if you only charge the printing cost. If you want to print the book and sell it at profit, please contact the author at `torbenm@diku.dk` and we will find a suitable arrangement.

In all cases, if you find any misprints or other errors, please contact the author at `torbenm@diku.dk`.

See also the book homepage: `http://www.diku.dk/~torbenm/Basics`.

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^*	$\{``"\} \cup \{vw \mid v \in \mathrm{L}(s),\ w \in \mathrm{L}(s^*)\}$	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, $\mathrm{L}(\mathtt{a})$ 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

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

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 $[\texttt{0-9}][\texttt{0-9}]^*$.

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 $[\texttt{0-9}]^+$. On a similar note, it is common that we can have zero or one occurrence of something (*e.g.*, an optional sign to a number). Hence we introduce the shorthand $s?$ for $s|\epsilon$. $^+$ and ? bind with the same precedence as *.

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

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: $[\texttt{+-}]?[\texttt{0-9}]^+$. 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:

$$[\texttt{+-}]?((([\texttt{0-9}]^+\texttt{.}[\texttt{0-9}]^*|\texttt{.}[\texttt{0-9}]^+)([\texttt{eE}][\texttt{+-}]?[\texttt{0-9}]^+)?)|[\texttt{0-9}]^+[\texttt{eE}][\texttt{+-}]?[\texttt{0-9}]^+)$$

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

$$[\texttt{+-}]?(([\texttt{0-9}]^+(\texttt{.}[\texttt{0-9}]^*)?|\texttt{.}[\texttt{0-9}]^+)([\texttt{eE}][\texttt{+-}]?[\texttt{0-9}]^+)?)$$

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.

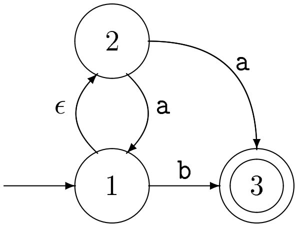


Figure 2.3: Example of an NFA

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 $(\mathtt{a}|\mathtt{b})^*\mathtt{ac}$ 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 $\mathtt{0}|\mathtt{1}|\mathtt{2}|\cdots|\mathtt{9}$ 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

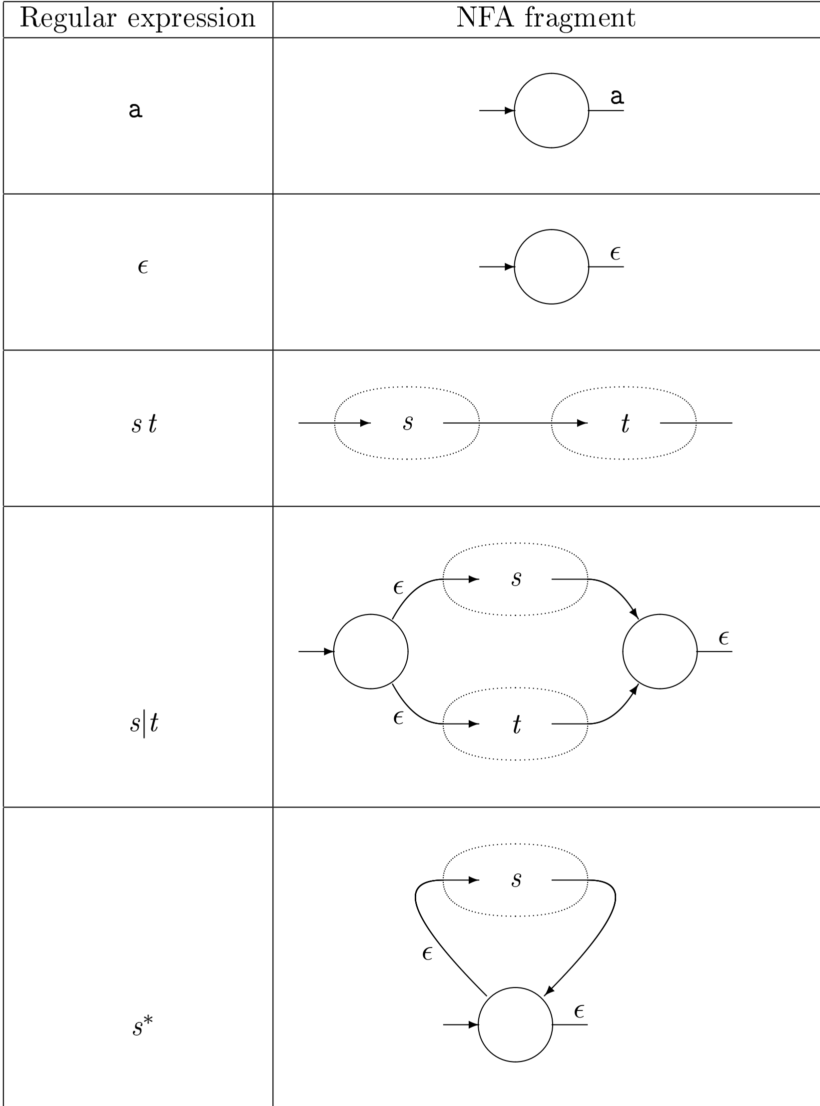


Figure 2.4: Constructing NFA fragments from regular expressions

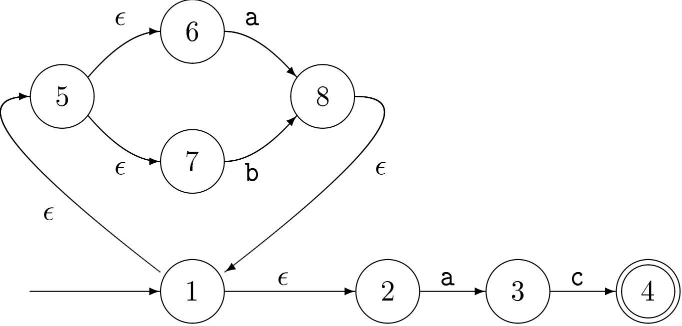


Figure 2.5: NFA for the regular expression (a|b)*ac

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 [0-9]$^+$ 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.

Regular expression	NFA fragment
ϵ	—
[0-9]	0 ⋮ 9 ϵ
s^+	s ϵ ϵ ϵ

Figure 2.6: Optimised NFA construction for regular expression shorthands

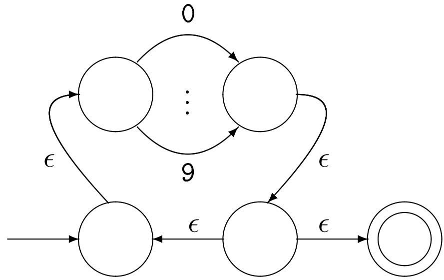


Figure 2.7: Optimised NFA for $[0\text{-}9]^+$

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*

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

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

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

If at any point an element in the sequence is identical to the previous, we have a fixed-point, *i.e.*, a set S such that $S = F(S)$. This fixed-point of the sequence will be the least (in terms of set inclusion) solution to the equation. This 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

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

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

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

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:

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

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:*

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

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

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

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

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

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 $\{\overset{\surd}{s'_0}, s'_1, s'_2\}$. We now pick s'_1, mark it and calculate its transitions:

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

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

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

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:

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

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

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

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

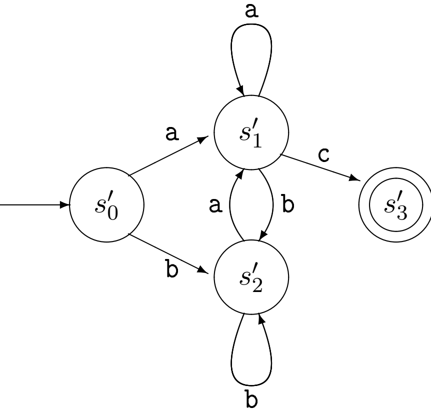


Figure 2.8: DFA constructed from the NFA in figure 2.5

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

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

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

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

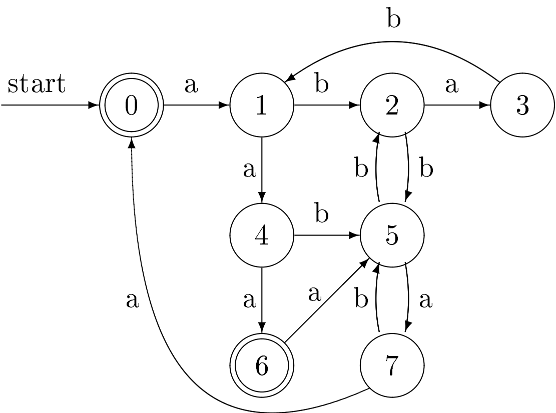


Figure 2.9: Non-minimal DFA

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.

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

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):

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

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

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

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

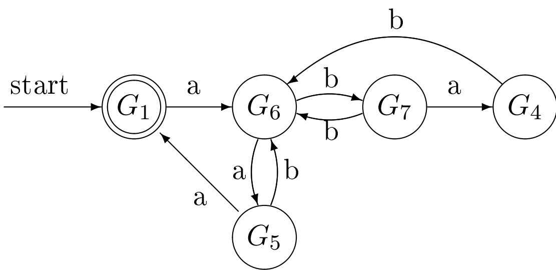


Figure 2.10: Minimal DFA

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

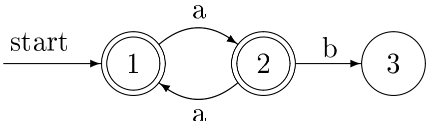
Algorithm 2.4 works under some, as yet, unstated assumptions:

- 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

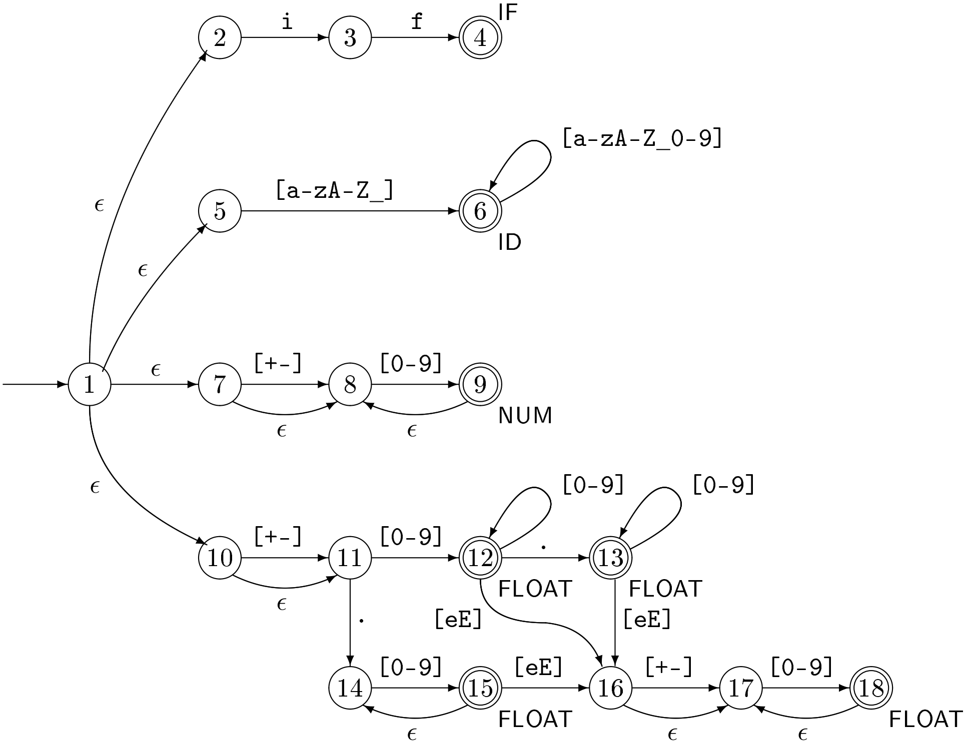


Figure 2.11: Combined NFA for several tokens

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:

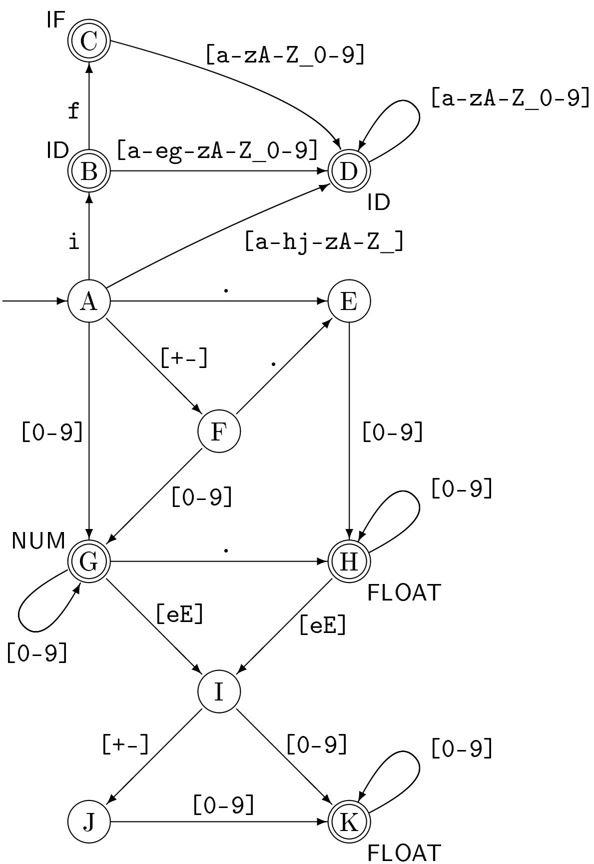


Figure 2.12: Combined DFA for several tokens

- 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 $(2x \text{ mod } (2^n))$.
- A transition on `b` maps the number x to $(x \text{ or } 1)$, 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 $\texttt{[0-9]}^+$. The value of a number-string is the usual interpretation of a number-string as an integer number. Note that leading zeroes are allowed.

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

a) All number-strings that have the value 42.

b) All number-strings that *do not* have the value 42.

c) All number-strings that have a value that is strictly greater than 42.

Exercise 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

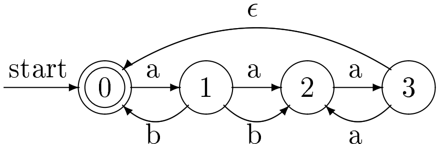
Given the regular expression `((a|b)(a|bb))*`:

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

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

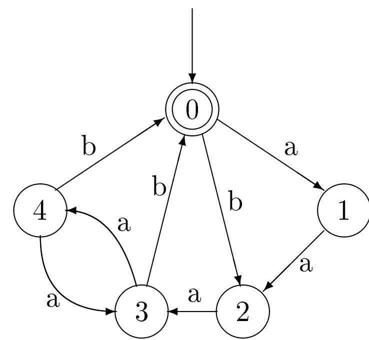
Exercise 2.4

Make a DFA equivalent to the following NFA:



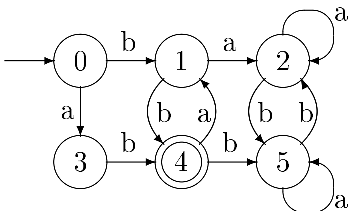
Exercise 2.5

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 `b`s (and any number of `a`s).

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

c) The set of strings where the difference between the number of `a`s and the number of `b`s is a multiple of 3.

Exercise 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:

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

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:

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

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 *prefixes*($\mathtt{ab}^*\mathtt{c}$).

Index

abstract syntax, **101**, 122
accept, 91, 95, 96
action, 43, 100, 101
activation record, 198
alias, 211, 212
allocation, 155, 216
Alpha, 169, 255
alphabet, 10
ARM, 169
array, 237
assembly, 3
assignment, 139
associative, 67, 68
attribute, 121
    inherited, 122
    synthesised, 121
available assignments, 223

back-end, 135
biased colouring, 192
binary translation, 255
binding
    dynamic, 115
    static, 115
bootstrapping, 247, 250
    full, 252
    half, 252
    incremental, 254
Bratman diagram, 248

C, 4, 41, 67, 70, 102, 104, 107, 119, 141, 147, 149, 152, 156, 210, 212, 217, 236, 244
C++, 245
cache, 237
cache line, 237
call sequence, 239
call stack, 197
call-by-reference, 211
call-by-value, 197
call-sequence, 199
callee-saves, 202, 204
caller-saves, 202, 204
caller/callee, 197
calling convention, 199
CISC, 170
coalescing, 193
code generator, 168, 171
code hoisting, 174, **235**
column-major, 156
comments
    nested, 43
common subexpression elimination, 174, 223, **228**, 236
compile-time, 140
compiling compilers, 250
conflict, 83, 88, 97, 99, 104
    reduce-reduce, 97, 99
    shift-reduce, 97, 99
consistent, 32
constant in operand, 169
constant propagation, 175
context-free, 121
    grammar, 55, **56**, 61
    language, 106

dangling-else, 70, 97, 99
data-flow analysis, 222, 235
dead code elimination, 231
dead variable, 170, 180
declaration, 115
 global, 115
 local, 115
derivation, 60, **60**, 61, 71, 82
 left, 64, 80
 leftmost, 61
 right, 64, 89
 rightmost, 61
DFA, 17, **21**, 46, 90, 91
 combined, 39
 converting NFA to, 23, 27
 equivalence of, 31
 minimisation, **31**, 32, 38
 unique minimal, 31
Digital Vax, 218
distributive, 26
domain specific language, 5
dynamic programming, 171

environment, 116, 124
epilogue, 199, 239
epsilon transition, 16
epsilon-closure, **24**

FA, 17
finite automaton
 graphical notation, 17
finite automaton, 10, **16**
 deterministic, **21**
 nondeterministic, **17**
FIRST, 73, 76
fixed-point, 25, 74, 76, 182, 184
flag, 168
 arithmetic, 169
floating-point constant, 15
floating-point numbers, 139
FOLLOW, 77
FORTRAN, 41
frame, 198
frame pointer, 199
front-end, 136
function call, 139, 239
function calls, 167, 197
functional, 116

gen and kill sets, 181
generic types, 131
global variable, 210
go, 91, 93
grammar, 71
 ambiguous, 64–66, 68, 72, 76, 85, 97
 equivalent, 65
graph colouring, 186, **187**
greedy algorithm, 171

hashing, 119
Haskell, 104, 117
heuristics, 186, **190**

IA-32, 169
IA-64, 169
IBM System/370, 218
imperative, 116
implicit types, 132
in and out sets, 181
index check, 159
 translation of, 159
index-check
 elimination, 175
index-check elimination, 231
inlining, 239
instruction set description, 171
integer, 15, 139
interference, 185
interference graph, 185
intermediate code, 3, 135, 179
intermediate language, 3, 136, 167, 174

tree-structured, 176
interpreter, 3, 135, 137, 248

Java, 41, 102, 136
jump, 139
conditional, 139, 168
jump-to-jump optimisation, 165, 228
just-in-time compilation, 136

keyword, 14

label, 139
LALR(1), 89, 100, 107
language, 10, 61
context-free, 106
high-level, 135, 247
left-associative, 66, 99
left-derivation, 72
left-factorisation, 87
left-recursion, 67, 68, 87, 102
elimination of, 85
indirect, 86
lexer, 9, **37**, 71
lexer generator, 37, 42
lexical, 9
analysis, 9
error, 42
lexical analysis, 2
lexing, 121
linking, 3
LISP, 217
live variable, 180, 197
at end of procedure, 182
live-range splitting, 193
liveness, **180**
liveness analysis, 181
LL(1), 56, 80, **82**, 85, 89, 97, 102, 107
local variables, 197
longest prefix, 41
lookahead, 80
LR, 89

machine code, 3, 135, 137, 167
machine language, 179
memory transfer, 139
MIPS, 169, 170, **171**, 176, 218
monotonic, 24

name space, 119, 122
nested scopes, 212, 214
NFA, 17, 91, 93, 105
combined, 38
converting to DFA, 23, 27
fragment, 19
non-associative, 67, 99
non-local variable, 210
non-recursive, 68
nonterminal, 56
Nullable, 73, 76

operator, 139
operator hierarchy, 66
optimisations, 174
overloading, 130

PA-RISC, 169
parser, 65
generator, 66, 100, 104
predictive, 71, 72, 77
shift-reduce, 90
table-driven, 89
top-down, 71
parsing, 55, 64, 121
bottom-up, 72
predictive, 76, 77, 80, 81
table-driven, 82
Pascal, 4, 67, 70, 102, 107, 119, 211, 212
pattern, 170
Pentium, 255
persistent, 116, 117
pointer, 211, 212
polymorphism, 131
PowerPC, 169

precedence, 59, 65, 66, 68, 69, 89, 97
  declaration, 97, 99, 106
  rules, 66
prefetch, 237
processor, 247
production, 56, 58
  empty, 57, 76
  nullable, 73, 77
prologue, 199, 239

recursive descent, **81**
reduce, 90, 91, 95
register, 179
  for passing function parameters, 204
register allocation, 3, 167, 179
  by graph colouring, 186
  global, 185
register allocator, 208
regular expression, **10**, 43
  converting to NFA, 19
  equivalence of, 31
regular language, 31, 44
return address, 198, 204
right-associative, 67, 99
right-recursion, 68
RISC, 167, 170, 204
row-major, 156
run-time, 140

Scheme, 104, 117
scope, 115
  nested, 212, 214
select, 188
sequential logical operators, 148, 149
set constraints, 78
set equation, 24, **24**
shift, 90, 91, 93
simplify, 187
SLR, 56, 89, 97
  algorithm, 92
  construction of table, 91, 96
SML, 4, 41, 67, 104, 117, 119, 212
source program, 249
Sparc, 169
spill, 199
spill-code, 190
spilling, 179, **188**
stack automaton, 55
stack automaton, 106
stack pointer, 216
start symbol, 56, 71
starting state, 16
state, 16, 17
  accepting, 17, 19, 28, 32, 37
  dead, 35
  final, 17
  initial, 17
  starting, 16, 17, 19, 27, 38
static links, 214
subset construction, 27
symbol table, 116, **116**, 124
  implemented as function, 118
  implemented as list, 117
  implemented as stack, 118
syntactical category, 122
syntax analysis, 2, 9, 55, 60, 64, **71**
syntax tree, 55, **61**, 71, 86

T-diagram, 248
tail call, 240
tail call optimisation, 240
target program, 249
templates, 245
terminal, 56
token, 9, 37, 39, 43, 71
transition, 16, 17, 28, 32
  epsilon, 16, 94
translation
  of arrays, 153
  of case-statements, 152

of declarations, 160
of expressions, 140
of function, 209
of index checks, 159
of logical operators, 147, 149
of multi-dimensional arrays, 156
of non-zero-based arrays, 159
of records/structs, 160
of statements, 144
of strings, 159
of `break/exit/continue`, 152
of `goto`, 152
type checking, 2, 121, 124
of assignments, 130
of data structures, 130
of expressions, 124
of function declarations, 127
of programs, 127
type conversion, 131
type error, 124

undecidable, 65

value numbering, 228
value numbering, 245
variable
global, 210
non-local, 210
variable name, 14

white-space, 9, 43
word length, 154
work-list algorithm, 26