Undergraduate Topics in Computer Science

Torben Ægidius Mogensen

Introduction to Compiler Design

UTiCS

Springer

Undergraduate Topics in Computer Science

Torben Ægidius Mogensen

Introduction to Compiler Design

Springer

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

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

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

Library of Congress Control Number: 2011933601

Cover design: VTeX UAB, Lithuania

Printed on acid-free paper

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

Preface

"Language is a process of free creation; its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied. Even the interpretation and use of words involves a process of free creation."
Noam Chomsky (1928–)

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

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

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

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

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

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

The Phases of a Compiler

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

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

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

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

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

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

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

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

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

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

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

Assembly and linking are typically done by programs supplied by the machine or operating system vendor, and are hence not part of the compiler itself, so we will not further discuss these phases in this book.

Interpreters

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

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

Using an interpreter is also useful during program development, where it is more important to be able to test a program modification quickly rather than run the program efficiently. And since interpreters do less work on the program before execution starts, they are able to start running the program more quickly. Furthermore, since an interpreter works on a representation that is closer to the source code than is compiled code, error messages can be more precise and informative.

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

Why Learn About Compilers?

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

Some typical reasons are:

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

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

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

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

Reason “d” is becoming more and more important as domain specific languages (DSLs) are gaining in popularity. A DSL is a (typically small) language designed for a narrow class of problems. Examples are data-base query languages, text-formatting languages, scene description languages for ray-tracers and languages for setting up economic simulations. The target language for a compiler for a DSL may be traditional machine code, but it can also be another high-level language for which compilers already exist, a sequence of control signals for a machine, or formatted text and graphics in some printer-control language (e.g. PostScript). Even so, all DSL compilers will share similar front-ends for reading and analysing the program text.

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

The Structure of This Book

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

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

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

To the Lecturer

This book was written for use in the introductory compiler course at DIKU, the department of computer science at the University of Copenhagen, Denmark.

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

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

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

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

Teaching with this book can be supplemented with project work, where students write simple compilers. Since the book is language neutral, no specific project is given. Instead, the teacher must choose relevant tools and select a project that fits the level of the students and the time available. Depending on the amount of project work and supplementary material, the book can support course sizes ranging from 5 to 7.5 ECTS points.

Acknowledgements

"Most people return small favors, acknowledge medium ones and repay greater ones—with ingratitude."
Benjamin Franklin (1705–1790)

The author wishes to thank all people who have been helpful in making this book a reality. This includes the students who have been exposed to earlier versions of the book at the compiler courses "Dat 1E" and "Oversættere" at DIKU, and who have found numerous typos and other errors in the earlier versions. I would also like to thank the instructors at Dat 1E and Oversættere, who have pointed out places where things were not as clear as they could be.

Copenhagen, Denmark Torben Ægidius Mogensen

Contents

List of Figures

Chapter 1
Lexical Analysis

"I am not yet so lost in lexicography as to forget that words are the daughters of earth, and that things are the sons of heaven. Language is only the instrument of science, and words are but the signs of ideas."
Samuel Johnson (1709–1784)

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

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

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

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

It is usually not terribly difficult to write a lexer by hand: You first read past initial white-space, then you, in sequence, test to see if the next token is a keyword, a number, a variable or whatnot. However, this is not a very good way of handling

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

the problem: You may read the same part of the input repeatedly while testing each possible token and in some cases it may not be clear where the next token ends. Furthermore, a handwritten lexer may be complex and difficult to maintain. Hence, lexers are normally constructed by *lexer generators*, which transform human-readable specifications of tokens and white-space into efficient programs.

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

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

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

1.1 Regular Expressions

The set of all integer constants or the set of all variable names are sets of strings, where the individual letters are taken from a particular alphabet. Such a set of strings is called a *language*. For integers, the alphabet consists of the digits 0–9 and for variable names the alphabet contains both letters and digits (and perhaps a few other characters, such as underscore).

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

When talking about regular expressions, we will use the letters (r, s and t) in italics to denote unspecified regular expressions. When letters stand for themselves (i.e., in regular expressions that describe strings that use these letters) we will use typewriter font, e.g., `a` or `b`. Hence, when we say, e.g., "The regular expression `s`" we mean the regular expression that describes a single one-letter string "`s`", but when we say "The regular expression s", we mean a regular expression of any form which we just happen to call s. We use the notation L(s) to denote the language (i.e., set of strings) described by the regular expression s. For example, L(`a`) is the set {"`a`"}.

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

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

Regular expression	Language (set of strings)	Informal description
`a`	{"`a`"}	The set consisting of the one-letter string "`a`".
ε	{""}	The set containing the empty string.
$s\|t$	$\mathrm{L}(s) \cup \mathrm{L}(t)$	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, "|" is not 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.

Fig. 1.1 Regular expressions

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

Note that while we use the same notation for concrete strings and regular expressions denoting one-string languages, the context will make it clear which is meant. We will often show strings and sets of strings without using quotation marks, e.g., write {`a`, `bb`} instead of {"`a`", "`bb`"}. When doing so, we will use ε to denote the empty string, so the example from L(s^*) above is written as {ε, `a`, `b`, `aa`, `ab`, `ba`, `bb`, `aaa`, ...}. The letters u, v and w in italics will be used to denote unspecified single strings, i.e., members of some language. As an example, `ab`w denotes any string starting with `ab`.

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

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

1.1.1 Shorthands

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

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

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

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

When using intervals, we must be aware of the ordering for the symbols involved. For the digits and letters used above, there is usually no confusion. However, if we write, e.g., [`0`–`z`] it is not immediately clear what is meant. When using such notation in lexer generators, standard ASCII or ISO 8859-1 character sets are usually used, with the hereby implied ordering of symbols. To avoid confusion, we will use the interval notation only for intervals of digits or alphabetic letters.

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

Since s^* denotes *zero or more* occurrences of s, we needed to write the set of digits twice to describe that *one or more* digits are allowed. Such non-zero repetition is quite common, so we introduce another shorthand, s^+, to denote one or more