Undergraduate Topics in Computer Science

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

More information about this series at http://www.springer.com/series/7592

Torben Ægidius Mogensen

Introduction to Compiler Design

Second Edition

Springer

Torben Ægidius Mogensen
Datalogisk Institut
Københavns Universitet
Copenhagen
Denmark

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-66965-6 ISBN 978-3-319-66966-3 (eBook)
https://doi.org/10.1007/978-3-319-66966-3

Library of Congress Control Number: 2017954288

Printed on acid-free paper

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

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 detect and report obvious programmer mistakes.

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

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

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

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

The Phases of a Compiler

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

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

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

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

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

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

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

Interpreters

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

Compilation and interpretation may be combined to implement a programming language. For example, the compiler may produce intermediate-level code which is then interpreted rather than compiled to machine code. In some systems, there may even be parts of a program that are compiled to machine code, some parts that are compiled to intermediate code that is interpreted at runtime, while other parts may be interpreted directly from the syntax tree. Each choice is a compromise between speed and space: Compiled code tends to be bigger than intermediate code, which 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 program 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 database 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), and DSLs are often interpreted instead of compiled. Even so, all DSL compilers and interpreters will have front-ends for reading and analyzing the program text that are similar to those used in compilers and interpreters for general-purpose languages.

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

About the Second Edition of the Book

The second edition has been extended with material about optimisations for function calls and loops, and about dataflow analysis, which can be used for various optimisations. This extra material is aimed at advanced BSc-level courses or MSc-level courses.

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 blackboard. After some of the sections in the book, a few easy exercises are listed as suggested exercises. It is recommended that the student attempts to solve these exercises before continuing reading, as the exercises support understanding of the previous sections.

Teaching with this book can be supplemented with project work, where students write simple compilers. Since the book is language neutral, no specific project is given. Instead, the teacher must choose relevant tools and select a project that fits the level of the students and the time available. Depending on the amount of project work and on how much of the advanced material added in the second edition is used, the book can support course sizes ranging from 5–10 ECTS points.

The following link contains extra material for the book, including solutions to selected exercises—http://www.diku.dk/~torbenm/ICD/.

Copenhagen, Denmark Torben Ægidius Mogensen

Acknowledgements

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

The author wishes to thank all 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 "Dat1E", "Oversættere", "Implementering af programmeringssprog" and "Advanced Language Processing" at DIKU, and who have found numerous typos and other errors in the earlier versions. I would also like to thank co-teachers and instructors at these courses, who have pointed out places where things were not as clear as they could be.

Copenhagen, Denmark
August 2017

Torben Ægidius Mogensen

Contents

List of Figures