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 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.

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 13), 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 chapters 5 and 13, but they are not the main focus of this book.

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 – 10) covers the middle part and back-end of interpreters and compilers. Chapter 4 covers how definitions and uses of names (*identifiers*) are connected through *symbol tables*. Chapter 5 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 (chapter 6) and compilation (chapters 7 – 10. In chapter 7, 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 8, 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 9. 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 10. Chapter 11 deals with analysis and optimisation and chapter 12 is about allocating and freeing memory. Finally, chapter 13 will discuss the process of *bootstrapping* a compiler, *i.e.*, using a compiler to compile itself.

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

Chapter 11 (on analysis and optimisation) was added in 2008 and chapter 5

(about interpreters) was added in 2009, which is why editions after April 2008 are called "extended". In the 2010 edition, further additions (including chapter 12 and appendix A) were made. Since ten years have passed since the first edition was printed as lecture notes, the 2010 edition is labeled "anniversary edition".

1.6 To the lecturer

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

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

This book was written as a response to this and aims at bridging the gap: It is intended to convey the general picture without going into extreme detail about such things as efficient implementation or the newest techniques. It should give the students an understanding of how compilers work and the ability to make simple (but not simplistic) compilers for simple languages. It will also lay a foundation that can be used for studying more advanced compilation techniques, as found *e.g.* in [35]. The compiler course at DIKU was later moved to the second year, so additions to the original text has been made.

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 (except this) has a section on further reading, which suggests additional reading material for interested students. All chapters (also except this) has a set of exercises. Few of these require access to a computer, but can be solved on paper or black-board. In fact, many of the exercises are based on exercises that

have been used in written exams at DIKU. After some of the sections in the book, a few easy exercises are listed. It is suggested 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 how much of the book is used and the amount of project work, the book can support course sizes ranging from 5 to 15 ECTS points.

1.7 Acknowledgements

The author wishes to thank all people who have been helpful in making this book a reality. This includes the students who have been exposed to draft versions of the book at the compiler courses “Dat 1E” and “Oversættere” at DIKU, and who have found numerous typos and other errors in the earlier versions. I would also like to thank the instructors at Dat 1E and Oversættere, who have pointed out places where things were not as clear as they could be. I am extremely grateful to the people who in 2000 read parts of or all of the first draft and made helpful suggestions.

1.8 Permission to use