Course Script

INF 5110: Compiler construction

INF5110, spring 2024

Martin Steffen

Contents

Chapter 1 Introduction

What is it about?

Learning Targets of this Chapter

The chapter gives an overview over different phases of a compiler and their tasks. It also mentions *organizational* things related to the course.

Contents

This is the script version of the material of the lecture. In previous iterations of the document, it consisted of basically all the slides in the order presented[1] plus additional remarks. Normally, I like not to overload slides with written information, but rather rely on speaking and telling a story, with the slides as guidance. So the "script" was previously an annotated version of the slides, each slide augmented by more details and explanations.

In the meantime, however, the script has emancipated from the slides to some degree. No longer does each individual slide appear here as subsub-...-section with some surrounding text and remarks added. Instead it has become more like a script rather than an annotated handout version of the slides. The document also covers background information, hints to additional sources, and bibliographic references. Some of the links or other information in the PDF version are clickable hyperrefs.

Course info

One recommended book the course is Cooper and Torczon [1] besides also, as in previous years, Louden [2].[2] We won't be able to cover the whole book anyway (neither the full Louden [2] book). In addition the material will take into account other sources, as well. At any rate, especially in the first chapters, for the so called compiler fronted, the material is so "standard" and established, that it almost does not matter, which book to take.

As far as the **exam** is concerned: Some time ago, it has mostly been a written exam, and it was "open book".

[1] Only the figures maye have been laid out differently and more compactly and except that gradual overlays were not reproduced in a step-by-step manner.

[2] see also *errata* list at `http://www.cs.sjsu.edu/~louden/cmptext/`.

Exam

For **spring 2024**, we do an **oral** exam, as in the previous 4 or 5 semesters. □

Course material

A master-level compiler construction lecture has been given for quite some time at IFI. The slides started off inspired by earlier editions of the lecture. Some parts are reused, and reworked over the semesters, and some graphics have just been clipped in and not (yet) been ported. The following list contains people designing and/or giving earlier versions of the lecture over the years, though more probably have been involved, as well.

- Martin Steffen (`msteffen@ifi.uio.no`)
- Stein Krogdahl (`stein@ifi.uio.no`)
- Birger Møller-Pedersen (`birger@ifi.uio.no`)
- Eyvind Wærstad Axelsen (`eyvinda@ifi.uio.no`)

Course's web-page

`http://www.uio.no/studier/emner/matnat/ifi/INF5110`

As usual, the page contains overview over the course, slides, material various announcements, beskjeder, etc. Watch for updates.

1.1 Motivation: What is compiler construction good for?

Compiler construction is an established, classical field in computer science and a core ingredient of most, if not all, computer science curricula.

Obviously, not everyone working in IT or in computer science is actually building a full-blown compiler. Very few people will ever do a significant, industrial-strength compiler, if only for the reason that a mature compiler (and accompanying technologies) for a general purpose programming language is a complex piece of software that typically involves many people, maybe evolving and maturing over years.

But compiler construction builds on many underlying fundamental concepts and techniques. Besides, most, if not basically all, software reads, processes and transforms input into output. The handled data is always presented in some format, except perhaps when inputting some raw sensor data, where the first step would be to render it into some structured format that can be processed further. Those formats are not necessary program languages, but anyway, processing them also uses involves techniques central in compiler construction. Furthermore, understanding the inner workings of compilers gives a deeper understanding of programming language(s). And there are constantly new languages (domain specific languages, graphical ones, new language paradigms and constructs...), which means compilers and their principles will *never* be "out-of-fashion".

Full employment for compiler writers

There is also something known as *full employment theorems* (FET), for instance for compiler writers. That result is basically a consequence of the fact that interesting properties of programs (in a full-scale programming language) are *undecidable* in general. "In general" means, when considered for *all* programs, and "interesting" means, semantic properties. For one particular program or some restricted class of programs, semantical properties may well be decidable. And of course syntactic properties, like whether the letter `x` occurs in the source code or not, are decidable.

The most well-known undecidable question is the so-called **halting problem**: can one decide generally if a program terminates or not (and the answer is: provably no). But that's only one particular and well-known instance of the fact, that (basically) all interesting (= semantical) properties of programs are undecidable (that's Rice's theorem). That puts some limitations on what compilers can do and what not. Still, compilation of general programming languages is of course possible, and it's also possible to prove the compilation generally correct; a compiler is just *one* particular program itself, though typically a complicated one. What is not possible is to *generally* prove a property about *all* programs (like whether a given program halts or not).

What does that imply for compilers? The limitations concern, among other aspects, in particular *optimizations*. An important part of compilers is to "optimize" the resulting code (machine code or otherwise). That means to improve the program's performance without changing its meaning otherwise (improvements like using less memory or running faster, etc.) The full employment theorem does not refer to the fact that targets for optimization are often contradictory; for example, there is often a trade-off between memory efficiency and speed. The full employment theorem rests on the fact that it's provably *undecidable* how much memory a program uses or how fast it is: all of those questions are undecidable. Without being able to (generally) determine such performance indicators, it should be clear that a *fully optimizing* compiler is unobtainable. Fully optimizing is a technical term in that context, and when speaking about optimizing compilers or *optimization* in a compiler, one means: do some effort to get better performance than you would get without that effort (and the improvement could be "always" or "on the average"). An "optimal" compiler is not possible anyway, but efforts to improve the compilation results are an important part of any compiler.

More specifically, the FET for compiler writers is often phrased in a slightly refined manner, namely:

> It can be proven that for each "optimizing compiler" there is another one that beats it (which is therefore "more optimal").

Since it's a mathematical fact that there's always room for improvement for *any* compiler no matter how "optimized" and tuned it already is, compiler writers will never be out of work (even in the unlikely event that no new programming languages or hardwares will be developed in the future...).

The proof of that fact is rather simple (if one assumes the undecidability of the halting problem as given, whose proof is more involved). However, the proof is not *constructive* in

that it does not give a concrete construction or *algorithm* how to *actually optimize* a given compiler. Well, of course, if that could be automated, then compiler writers would again face unemployment... But the FET assures: don't worry, it cannot be automated.

1.2 Compiler architecture & phases

Central for the architecture of a compiler is its "layered" structure, consisting of *phases*. It basically a "pipeline" of transformations, with a sequence of characters as input (the source code) and a sequence of bits or bytes as ultimate output at the very end (assuming "full compilation"). Conceptually, each phase analyzes, enriches, transforms, etc. the representations, and afterwards hands over the result to the next phase.

This section give just a taste of general, typical phases of a full-scale compiler. Of course, there may be compilers in the broad sense, that don't realize all phases. For instance, if one chooses to consider a source-to-source transformation as a compiler (known, not surprisingly as S2S or source-to-source compiler), there would be no machine code generation (unless of course, it's a machine code to machine code transformation...). Also *domain specific languages* may be unconventional compared to classical general purpose languages and may have consequently an unconventional architecture. Also, the phases in a compiler may be more fine-grained, i.e., some of the phases from the picture may be sub-divided further. Still, the picture gives a fairly standard view on the architecture of a typical compiler for a typical programming language, and similar pictures can be found in all text books.

Each phase can be seen as one particular *module* of the compiler with a clearly defined *interface*. The phases of the compiler naturally will be used to *structure* the lecture into *chapters* or sections, proceeding "top-down" during the semester. In the introduction here, we shortly mention some of the phases and their typical functionality.

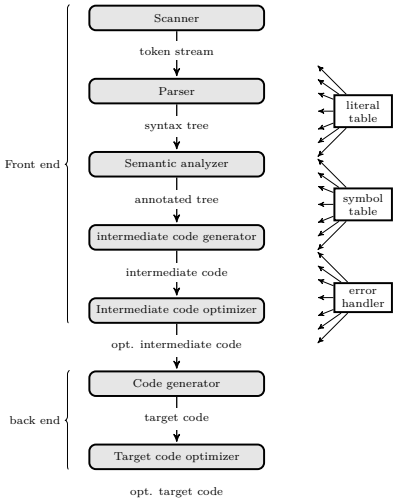


Figure 1.1: Structure of a typical compiler

1.2.1 Preprocessor

The architecture in Figure 1.1 does not mention a preprocessor and one may debate whether a preprocessor is part of a compiler or not. The word already indicates that a preprocessor does whatever it's doing *before* the real compiler does its job. So, the preprocessor can be a separate program, independent from the compiler, but invoked before the compilation start for earnest. So, one may well consider the preprocessors as a very first phase of the overall compilation. It's also possible that the preprocessor resp. preprocessor functionalities are not covered by a separate program, but integrated into compiler.

Either way, core compilers often ship with very many accompanying tools, the preprocessor may be one of them. There may be a formatting facility, a debugger, decompilers, refactoring and very many others. Some could be seen as *more* part of the compiler, some less. Is a specific editor part of the programming language? Most would say *no!* in most cases. But for instance the *Visual Basic* languages state, that it's "not a standalone product".

Anyway, it's not really important and in this lecture we won't even cover preprocessing except mentioning it here. Likewise, the oblig will not involve programming a preprocessor.

But then, what *is* typically the task of a preprocessor? Let's have a look what for instance can be done with a C-style preprocessor.

> Typical tasks there are *file inclusion*, *macro definition and expansion*, and *conditional code* or *conditional compilation*.

The C-prepocessor is sometimes seen a "hack" grafted on top of a compiler. But it probably is considered a *useful* hack, otherwise it would not be around ... But it does not naturally encourage elegant and well-structured code, it just offers fixes for some situations. The C-style preprocessor has been criticized variously, as it can easily lead to brittle, confusing, and hard-to-maintain code. By definition, the *pre*-processor does its work before the real compiler kicks in: it massages the source code before it hands it over to the compiler. The compiler is a complicated program and it involves complicated phases that try to "make sense" of the input source code string. It classifies and segments the input, cuts it into pieces, builds up intermediate representations like graphs and trees which may be enriched by "semantical information". However, not on the original source code but on the code after the preprocessor has made its rearrangements. Already simple **debugging** and **error localization** ("in which line did the error occur") may be tricky, as the compiler can make its analyses and checks only on the massaged input, it never even sees the "original" code.

In this lecture and this introductory remarks we talk about C-style preprocessing and C-style macros. That's because the C-preprocessor is a prominent example of preprocessors. As said, it has it's limitations, and has been accordingly criticized.

Most other languages don't even have preprocessors, for example Java. That latter statement is not 100% correct. Officially, there may be none. Though if you google around you will find (not surprisingly) that the lack of a preprocessor let some people not sleep

until they made one. But actually, if one likes "C-style preprocessing", why would one actually need one? Preprocessing is massaging a textual file. The behavior is governed by the preprocessor syntax (not C-syntax or C^{++}-syntax, Java-syntax or whatever). So if one has a file with Java code, and one for some reason misses directives like `#ifdefs` very much, one can just sprinkles one's code with those macro-syntax and pump it through the C-preprocessor (using for example flags like `gcc -P -E ...`). Then that results in a different file. The (gcc) preprocessor may be written in C, but the language it handles is not C, it's the particular preprocessor syntax. Of course, it may be a bad idea, to do C preprocessor directives in Java programs, it will interfere in not so nice ways if one programs under an IDE (like Eclipse), but there is no real reason why it's not possible.

Another remarks concerns **macros**, not to leave the impression that macro programming is synonymous with preprocessing. Languages, for instance Rust, may offer more "integrated" macro facilities. More integrated in that they don't work on the input file, but maybe the *token stream*. If you at that point don't know what the token streams is, at the end of the lecture you will. Actually, already half-way through the lecture, as the token stream is the output of the parsing phase.

Other aspects of C-style preprocessing concerns **file inclusion**, using, say, `#input`. See Listing 1.1.

```
#include <filename>
```

Listing 1.1: file inclusion

It's the single most primitive way of "composing" programs split into separate pieces into one program. It's basically that instead of **copy-and-paste** some code contained in a file literally, it simply "imports" it via the preprocessor. It's easy, understandable (and thereby useful), completely transparent even for a beginner, and is a **trivial** mechanism as far as compiler technology is concerned. If used in a disciplined way, it's helpful, but it's not really a decent modularization concept. Or to say it differently: it "moduralizes" the program on the "character string" level (with support of the operating system's concept of *file* abstraction) but not on any more decent, program language level.

Conditional compilation is illustrated in Listing 1.2

```
#vardef #a = 5; #c = #a+1
 ...
#if (#a < #b)
     ..
#else
   ...
#endif
```

Listing 1.2: Conditional compilation

Note; `#if` is *not* the same as the `if`-programming-language construct.

Also languages like TEX, LATEX etc, certainly domain-specific languages but with a supporting compiler nonetheless. support conditional compilation. E.g., in TEX (and thus in LATEX as well), one can write `\if<condition> ... \else ... \fi`. It's not, however, done by a preprocessor outside the TEX-compiler. As a side remark: In previous semesters, the sources, for the slides and this script (written in LATEX, sometimes LATEXin

combination with `org` as "pre-processor") make quite some use of conditional compilation, compiling from the source code to the target code, for instance PDF: some text shows up only in the script-version but not the slides-version, pictures are scaled differently on the slides compared to the script ....

```
#macrodef hentdata(#1,#2)
    ---- #1-----
     #2---(#1)---
#enddef

...
#hentdata(kari,per)
```

Listing 1.3: Macros

```
 ---- kari-----
per---(kari)---
```

Note: the code is not really C, it's used to illustrate macros similar to what can be done in C. For real C, see `https://gcc.gnu.org/onlinedocs/cpp/Macros.html`. There, conditional compilation is done with `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`. and `#endif`. Definitions are done with `#define`.

1.2.2 Scanner or lexer

Ignoring the preprocessor, the lexer is the first compiler phase. It will be covered in the first chapter after the introduction. The words lexer or scanner are synonymous, and the lexer performs what is called *lexicographic* analysis (hence the name). That's different from *syntactic* analysis which comes afterwards and which is done by the *parser*. The lecture will cover both phases to quite some extent, in particular parsing.

> The input of the lexer is the "program text". That can be in the form of a string or a character stream, or similar. And the task is to **divide** and **classify** that input into a stream of **tokens** handed over to parser. In doing so, it typically also removes **whitespace** (blanks, newlines, tabs ...) and **comments**. Conceptually, the lexer is based on **finite state automata** and **regular languages.**

Let's look at the simple code snippet from Listing 1.4. The input code snippet is supposed to be a sequence of characters (or a string). The blanks (space characters, or white spaces) are made specially visible in the listing, they are not just displayed as empty spaces.

```
a[index]␣=␣4␣+␣2
```

Listing 1.4: Simple code snippet

The table in Figures 1.2 shows the individual pieces of that string in the left column of the table. Those pieces are called **lexemes** Note that the white spaces are ignored in some way: there is no white-space lexeme (which is typical when do scanning). The column on the right shows a possible (and typical) classification The tokens consists of a token class and of a token value. Sometimes, there is no token value, only a token class. Note that

in the picture, the token values are uniformly strings, i.e., the lexeme (but there will be some comments on that soon).

	token	
lexeme	token class	value
`a`	*identifier*	`"a"`
`[`	*left bracket*	
`index`	*identifier*	`"index"`
`]`	*right bracket*	
`=`	*assignment*	
`4`	*number*	`"4"`
`+`	*plus sign*	
`2`	*number*	`"2"`

Figure 1.2: Sample lexemes and tokens

That's a possible way, but the lexer does a bit more. Take the lexemes `2` or `4` representing numbers. One can store the corresponding string, as shown, but, having classified them as numbers, one can already transform them into numbers. All languages have conversion functions that can do that, for instance in Java there is `Integer.parseInt(mystring)` and other methods that can do the job. That is shown in the table in Figure 1.3a. The alternative treatment does something else on top, concerning the token class of **identifiers.** String constants or string literals, as they are also called, are not ideal to work with. Basically, because they are compound data structures. That would make it costly to compare strings, maybe representing variable names or other named entities, and the compiler will certainly have to compare those, probably more than once. So it's worthwhile to think of a better representing those, for instance storing them in a separate table and use an index to the table as value in the token instead. That is shown in Figure 1.3.

	token	
lexeme	token class	value
`a`	*identifier*	2
`[`	*left bracket*	
`index`	*identifier*	21
`]`	*right bracket*	
`=`	*assignment*	
`4`	*number*	4
`+`	*plus sign*	
`2`	*number*	2

(a) alternative

0	
1	
2	`"a"`
	⋮
21	`"index"`
22	
	⋮

(b) separate table

Figure 1.3: Sample lexemes and tokens (2)

We said, that white-space characters are typically filtered out by the lexer. That does not mean white-space is completely "meaningless" in the sense that one could add and remove white space arbitrarily. Also that is common for most programming languages nowadays (and most written languages[3] based on an lettered alphabet, like Western languages). We

[3] Noonelikestoreadsentenceslikethisnoteveninfootnotes

will see later, in the chapter about lexing, that there had been for instance versions of Fortran, which treated white-space as completely meaningless.[4] In the example here, as in basically all programming languages, white space is not completely meaningless: it serves as a form of **separator**. Like the string `index` in the example counts as one lexeme, one unit of the overall string, which is classified as identifier in the table (the so-called token class). If it had been written with one white space as `in dex`, then the scanner would have returned two identifiers. Presumably that would make the overall string syntactically wrong, but that's a question for the parser to decide, not the lexer.

Note also, that `index`, without the white space, is marked as one identifier, not as two or maybe 5 individual ones. That implies, that the lexer tries to find the **longest** stretch of characters that can be interpreted (e.g.) as identifier, uninterrupted by white space or other characters that are disallowed for identifiers. All that sounds obvious (because one is so much used to it), but, as mentioned, there are different ways to interpret white spaces (meaningless, or as separator, one may even interpret indentation, which is a sequence of white spaces or tabs to have some grouping meaning), to have some meaning beyond looking nice for the programmer). Rules governing lexical aspects of the language cover all that: what are allowed characters for identifiers, actually; what are overall the allowed reservoir of characters (called the **alphabet**), what are white spaces (blanks, tabs, newlines, carriage returns, others?), what's a comment?

One may ask: what are then exactly lexical aspects of a language? A non-helpful and tautological answer is: those that are dealt with by the lexer. A better answer is: those aspects that can be captured by **regular expressions**. Lexer generator tools (like `lex` and similar ones) are tools which allow to specify lexical aspects of a language by regular expressions, and they use that specification to generate a lexer or scanner program. Basically, realizing a finite state automaton that performs the lexing task. What cannot be covered by regular expressions resp. finite state automata, is handed over to the next phase(s), the next one being the parser, which is responsible for syntactic aspects. Those are aspects that can be covered by some more expressive formalism, known as *context-free grammars*.

1.2.3 The parser

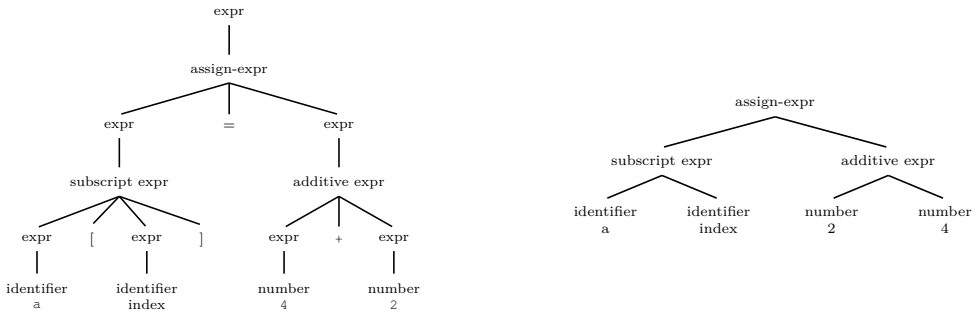
The *parser* is the phase after the lexer. It is responsible for checking *syntactic* aspects of the language and hand over to the next phases a intermediate representation that captures the syntax of a syntactically correct program. This representation is called the "syntax tree". Actually, there are two kinds of trees involved when parsing a program, more precisely parsing a token stream of a lexically correct program generated by the lexer. The two forms of syntax trees are known as **concrete syntax tree** or **parse tree** on the one hand and **abstract syntax tree** on the other. We we discuss these extensively in the corresponding parts of the lecture.

Take as example the expression

$$\texttt{a[index]} = 4 + 2$$

[4] And then there is of course the "programming language" called *Whitespace* ...

again. We have use it earlier to illustrate the working of a scanner and lexemes etc. for instance in Figure 1.2. Corresponding trees are shown in Figure 1.4. In the *leaves* of the parse tree are the **token** the lexer has produced. So when writing, for instance [in one of the leaves, it's not the character or string `'['` from the corresponding lexeme, it's meant as the corresponding token (called *right bracket* in the table of Figure 1.2).



(a) Parse tree

(b) Abstract syntax tree

Figure 1.4: Syntax trees

The trees here are mainly for illustration. It's not meant as "this is how *the* abstract syntax tree looks like" for the example. In general, abstract syntax trees are less verbose that parse trees. The latter are sometimes also called *concrete* syntax trees. The parse tree(s) for a given word are fixed by the **grammar**. One should more precisely say "context-free grammar" as there are also more expressive grammars, but without further qualification, the word "grammar" often just means context-free grammar. The abstract syntax tree is a bit a matter of design. Of course, the grammar is also a matter of design, but once the grammar is fixed, the form of parse trees is fixed, as well. What *is* typical in the illustrative example is: an abstract syntax tree would not bother to add nodes representing brackets (or parentheses etc.), so those are omitted. In general, ASTs are more compact, leaving out superfluous information without omitting *relevant* information.

When saying the grammar *fixes* the form of the parse-trees, it is *not* meant that, given one sequence of tokens, then there is exactly one parse tree. A grammar, where for each input token stream, there is at most one parse tree is is called **unambiguous**, some grammars are and some not. At any rate, ambiguous grammars are unwelcome, and parsers realize typically unambiguous grammars. Parser generators (like `yacc` and similar), when fed with an ambiguous grammar as specification, will indicate so-called *conflicts*. That are points where the parser has different options as reaction to an input, which is not a good thing. The parser would typically make some form of decision (like taking just the first option and ignoring the alternatives), but, as said, it's not a good sign. It typically indicates troubles with the grammar.

To avoid misconceptions: an ambiguous grammar will lead to conflicts in such tools, but the other way around is not true: a parser tool may indicate conflicting situations even if the grammar is unambiguous. The reason is that parsers typically are not expressive enough to cover all kinds of context-free grammars, not even all unambiguous ones. They focus on more restricted classes of context-free grammars (and which class it is depends

on the chosen parser technology and how much one wants to "invest" in so-called **look-aheads**. We will encounter different conflicts in the corresponding chapter.

1.2.4 Semantical analysis

The *semantical analysis* deals with properties more complex than the language's syntax. There are **very many** ingredients to be dealt with beyond syntax, which means, the part(s) that comes after parsing are often big and complicated, and cover different things. Also the underlying principles and theories is less "uniform", it's more that various different concepts come into play. One typical phase that often comes directly after parsing and thus works directly with the AST is **type checking**. It can be understood as "decorating" the AST with type information, as illustrated in the following pictures. It may not be that it's concretely implemented that one adds information directly into the AST structure. Often, especially the semantic analysis phase work with some structure called **symbol table** that maintains information about syntactic entities for easy consultation during the analysis.

- additional info (non context-free):
  - *bindings* for declarations
  - (static) *type* information

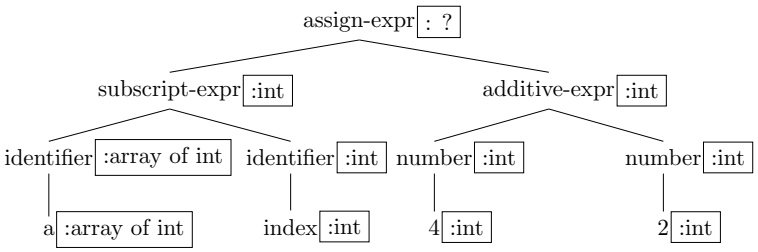


Figure 1.5: (One typical) result of semantic analysis

- here: *identifiers* looked up wrt. declaration
- 4, 2: due to their form, basic types.

As far as the assignment statement in the *root* of the syntax tree is concerned: the question mark is added there, to indicate, that there is no unform correct answer, it depends a bit on the programming language, resp. how the assignment statement *behave*. To say it differently, what the *semantics* of the assignment, and there are two plausible choices for that.

Non-negotiable is, of course, to generate **correct** code, i.e., code that correctly and for all program reflects the language's intended semantics. In particular, it needs to realize all the fancy programming abstractions the language may offer. Even variables are abstractions, they may "feel" like changing directly the "memory" of the machines one runs the program on, but they offer typically are already quite some level of abstraction. Ultimately, from the perspective of the compiler and machine code, one has to operate with addresses and perhaps the value is stored temporarily in registers. Not only have variables symbolic names chosen by the programmer, they are also organized in scopes, there may be local or

global variable etc. Variables may be formal parameters of a procedure. All those are very convenient **abstractions**, which need to be realized (by the compiler) by managing the memory properly. Each variable access must ultimately translated in perhaps a sequence of machine instructions, which ultimately access the current corresponding location which holds the value of the variable. All that invisible for the programmer, who thinks in terms of variables and has an intuitive feeling of scopes and locality of the variable, like: "x is a variable local to procedure p". Of course, if p is called multiple times, perhaps recursively, there are multiple instances of x to be managed at run-time. The corresponding arrangements realized by the compiler is called the **run-time environment**. Also, parameter passing needs to be arranged by the compiler, since at the lowest level of machine code, there's no such things as variables or "passing them", it's just sequences of cleverly designed machine instructions that realize parameter passing, scoping etc.

So, the non-negotiable correctness requirement for a compiler basically means to maintain those abstractions: the programmer thinks in terms of parameter passing: the formal parameters are "replaced" by the actual parameters, but this is broken down to perhaps many individual, very small steps, perhaps even shuffling around values in registers etc. which behave, when thinking about a higher level of abstraction, like parameter passing.

Besides correctness of the generated code, there is the question, how efficient the generated code maintains the abstractions. Optimization addresses efficiency without of course compromising correctness. Optimization can be done in various phases of the compiler, and also repeatedly. We don't go too much into corresponding issues in connection with compilers. The examples on the slide illustrate different versions of a code snippet, some presumably more efficient than others (thus "optimized"). The word, "optimizing" is anyway a bit of a misnomer, as a compiler that guarantees genuinely optimal code is unobtainable (even if one could agree on criteria to measure the quality). Independent from that, there are influences outside the control of the compiler, which influence the efficiency of the result. The examples shown here are on the level of source code, but often similar "optimizations" are done (also) on lower levels, a for instance a a so-called *intermediate code* level or at machine code level (or both). The improvements illustrated on the slides here can be made systematic with techniques called *data-flow* analyses. We don't do too much systematic data flow analysis or aggressive optimization in this lecture, but we will cover one important such analysis called *liveness* analysis.

1.2.5 Optimization

Like semantic analysis, optimization is a broad and diverse topic. Both also overlap, resp. static or semantic analysis can provide necessary information used for optimization. Optimization refers to efforts in the compiler to produce not just correct code, but "good" correct code (fast, ...), and, as said, it often goes hand in hand with some semantic analysis.

The following example shows a transformation done a the **source-code level**.

```
t = 4+2;
a[index] = t;
```

```
t = 6;
a[index] = t;
```

```
a[index] = 6;
```

The code examples show three different "variants" of semantically the same program. The optimizations are not very radical and complicated, but doing corresponding steps in more complex situations can be challenging. For instance, it's not always so trivial to figure out that a value or variable is actually constant (in the example it's obvious).

The lecture will not dive too much into optimizations. The ones illustrated here are known as *constant folding* and *constant propagation*. Optimizations can be done and are done in various phases on the compiler. Here we said, optimization at "source-code level", and what is typically meant by that is optimization on the abstract syntax tree (presumably at the AST after type checking and some semantic analysis). The AST is considered so close to the actual input that one still thinks of it as "source code" and no one tries seriously to optimize code a the input-string level. If the compiler (or a compiler-related tool) "massages" the input string, it's mostly not seen as optimization, it's rather (re-)formatting or preprocessing. There are indeed format-tool that assist the user to have the program is a certain "standardized" format (standard indentation, new-lines appropriately, etc.) Also C-style preprocessing, we mentioned before, falls into that category of massaging the input string.

Concerning optimization, what is also typical is, that there are many different optimizations building upon each other. First, optimization A is done, then, taking the result, optimization B, etc. Sometimes even doing A again, and then B again, etc.

1.2.6 Code generation & optimization

```
MOV R0, index  ;; value of index -> R0
MUL R0, 2      ;; double value of R0
MOV R1, &a     ;; address of a -> R1
ADD R1, R0     ;; add R0 to R1
MOV *R1, 6     ;; const 6 -> address in R1
```

```
MOV R0, index  ;; value of index -> R0
SHL R0         ;; double value in R0
MOV &a[R0], 6  ;; const 6 -> address a+R0
```

- potentially difficult to automatize[5], based on a formal description of language and machine
- platform dependent

For now it's not too important what the code snippets do. It should be said, though, that it's not a priori always clear in which way a transformation such as the one shown is an *improvement*. One transformation that most probably is an improvement, that's the "shift left" for doubling. Another one is that the program is *shorter*. Program size is something that one might like to "optimize" in itself. Also: ultimately each machine operation needs to be *loaded* to the processor (and that costs time in itself). Note, however, that it's generally not the case that "one assembler line costs one unit of time". Especially, the last line in the second program could costs more than other simpler operations. In general, operations on *registers* are quite faster anyway than those referring to main memory. In order to make a meaningful statement of the effect of a program transformation, one would need to have a *"cost model"* taking register access vs. memory access and other aspects into account.

[5] Not that one has much of a choice. Difficult or not, (almost) *no one* wants to optimize generated machine code by hand ....

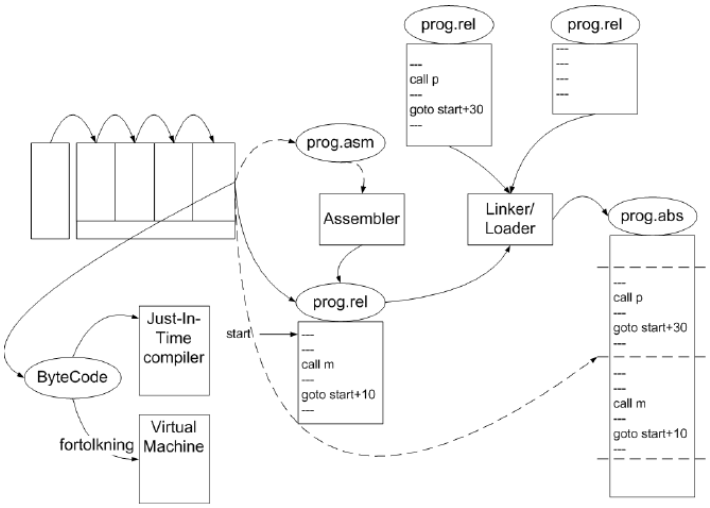


Figure 1.6: Anatomy of a compiler

The picture illustrates that there are, at the lower end of the compiler or "after" the compiler, different low-level representation. For instance, different flavors of "machine code" or "assembly code". There is also the notion of *relocatable code*. Relocatable it not (yet) *absolute* code. The difference refers to the addresses. The addresses in relocatable code are seen as relative. Only in absolute machine code, the addresses refer to actually addresses in (virtual) memory. The compiler typically does not work with absolute addresses, but relative. That's keep open till the very end. Of course, whether using relative addresses (in relocatable code) or absolute addresses, it's the same program. Fixing the addresses does not involve changing a level of abstraction. So it can be seen as a last finishing touch or deployment *after* the compiler has done its work. Of course, it's also completely independent from the input language and the compiler. So, it's often done by a separate tool, known as (linker)/loader. So often, as shown in the picture, the very final stages don't just involves fixing absolute addresses, but also tying different pieces of code together (linking). The story here is more for *static* linkers and loaders. There are also dynamic versions and *dll's* (dynamically linked libraries) etc.

1.2.7 Diverse notions in connection with compilers

Computers a complicated (and diverse) machines. Likewise, programming languages or languages and notations instructing computer systems are very diverse. Consequently, compilers and related technologies can be complicated and very diverse. Without being overly systematic, let's here just mention some concept in connection with compilers and their "eco-system".

In a compiler, one often separates **front-end** and the **back-end**. Not everyone might agree where the dividing line between the front- and the back-end exactly is. One common separation is that everything that is platform-independent is the front-end, and the platform-dependent part is the back-end.

One could also align the separation in front- and back-end based on the following observation. Roughly speaking, a compiler does first **analysis** and afterwards **synthesis**.

In a standard, full-blown compiler, the input is rather *unstructured*, simply a stream of characters. So, in the first phases, by doing analysis, the compiler builds up more and more complex data structures (syntax **trees**, control-flow **graphs**, dependency and conflict **graphs** and what not, enriched with all kinds of information (**types**). So the representation gets more and more structured. In the synthesis part, all the additional structure get's through away, until at the end, again a very **unstructured** stream of bits or words come out, as result of the synthesis.

Each mature compiler comes along with many assisting tools and infrastructure, e.g. debuggers, formatting tools, profiling, project management, editors and integrated development environments, build support, etc.

One important distinction is that of **compilers vs. interpreters**. Some languages are compiled, some are interpreted, for some one can do both, and often it's not pure interpretation anyway.

When speaking about **compilation**, one classically means translating *source code* to *machine code* for given machine. As mentioned, there are different "forms" of machine code for one machine): executable code, relocatable code, or textual assembler code, which is (more or less) humanly readable.

In **full interpretation**, on the other hand, the source code is directly executed. Since that is rather unpractical, it's more the syntax tree that are executed (or interpreted); syntax-trees still count loosely as source code, since they still represent the syntax of the program. This is often done for command languages, interacting with the OS. Normally full interpretation is quite slower than machine code produced by a good compiler. Examples of languages that commonly counted among the interpreted ones are Scheme/Lisp, PHP, Ruby, Python, and JavaScript.

More often than full interpreted ones, one sees **compilation to intermediate code which is interpreted**. Instead of saying the intermediate code is interpreted, one also say, the **intermediate code** is executed on a **virtual machine**. Java and the Java Virtual Machine is one well-known example for that, but there are many other interpreted language in that sense. The intermediate code is ideally designed for efficient execution. The interpreted intermediate code in Java is **byte code,** i.e., virtual machine is a **byte-code interpreter**. To some extent, the remark about a speed applies here as well, intermediate code execution is slower that direct compilation, but typically faster than full interpretation.

1.3 Bootstrapping and cross-compilation (no pensum)

Let's just glance over this section, we won't discuss it much in class. It's not part of the pensum, but may be interesting nonetheless.

Bootstrapping refers to a process of "building something out of nothing", like in the tale from the guy that used his own bootstraps to pull himself out of a swamp. Of course one has to "start somewhere", dragging oneself out of the swamp by one owns bootstrap in this way without some place to stand on is possible in some funny tale only. Bootstrapping is also the origin of the term "to boot", which refers to firing up a computer system by

starting its OS. That's a multi-stage process, which gradually "escalates" from hardware or firmware, the master boot record, boot loader etc., until the whole OS is up and running. So it starts with a very "thin thread", some "commands" in silicon, pulling up a thicker one, commands in the master boot record etc. until the whole complex system is showing the login screen or whatever the computer it is supposed to do when operational.

That sketches the process of booting a computer system. For writing a *compiler*, one faces (or maybe historically faced) the task: how can I write a compiler from scratch? Well, one can of course implement the whole thing in *assembler*; the hardware certainly has some instruction set, and one can use that to implement the desired compiler.

Now, that's a tall order; one would rather avoid using assembler (except perhaps for carefully selected special tiny sub-tasks) and make use of a high-level language, with all its abstractions and other infrastructure, like libraries, editors, configuration and version management etc.

If such a language is not around, well: That's the chicken-and-egg problem of *bootstrapping a compiler*:

> If one had a *compiler executable*, one could (more) easily write the compiler program and compile that source code to an executable compiler.

Nowadays, the problem is perhaps not so pressing insofar that there are enough high-level languages around to choose from. Assuming one is happy with C as high-level language and intends to invent C++, one can write the first C++ compiler in C, of course, and that's quite more easy compared to write it in assembler.

But there had been a time, before the era of the PC and the mass-market for electronic computers, where ordering a computer means ordering a room or cabinet of hardware, with an instruction set (and no internet to quickly download something useful). Perhaps the HW came with some operating system, but maybe also not, or it was kind of rudimentary, barely able to process "jobs" (by reading punch-cards perhaps) and controlling other peripherals.

In such a situation, no compiler for a high-level language may be available, or perhaps not for the particular machine (or maybe not even any high-level language yet, because one is the first one, who not only proposes to use high-level languages, but who actually tries to implement one). That's where bootstrapping for compiler comes in: instead of writing the production-quality compiler from scratch in assembler (which is too tough) and instead of writing the compiler for the newly design language in the language itself (which makes no practical sense), one goes gradually. One starts with a simple version of some relevant aspects of the planned language, leaving out optimizations, etc., until that rudimentary compiler exists. One then starts writing in that new language better or more comprehensive versions of that language etc., until one has a decent stable version that is strong enough to compile *itself* without much reliance on assembler to rely on.

Historically, the development of the language C went hand-in-hand with the development of Unix, insofar it was a larger bootstrapping problem than just a compiler. How to develop a modern and at that time revolutionary operating system together with a compiler that can compile C programs and can compile the operating system itself, on which then the C

programs run.... Note that the fact that the OS is written (for most part) in a high-level language is enormously *important*, as it allows *portability* (!). If every OS had to be written totally from scratch, there would be no portability across different hardware platforms. And, as with compilers and languages, there had been a time where non-portability was the norm.

1.3.1 Tombstone diagrams

The compilation process is here illustrated with so-called *T-diagrams* or *tombstone diagrams* which is some "graphical" representation of the compilation process, mentioning the input language, the output language, and the language in which the compiler is represented as the three arms of the "T" (see Figure 1.7).

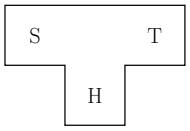
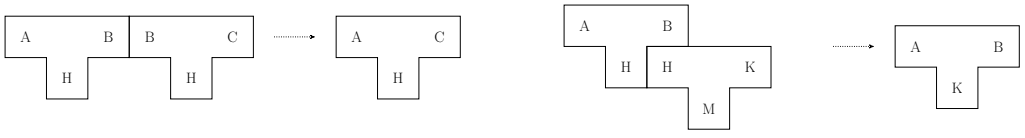


Figure 1.7: T-diagram: translating source to target language in a host language

To apply such a setting, the host "H" should be executable, i.e., it represents a compiler executable for which one has a platform to execute on, either in native code, or a interpreter or byte code etc. The "S", the source of the translation is typically given as source code, i.e., in a format not directly executable. The target format "T" then is then typically an executable format, like machine code or a format that can be run on an interpreter or virtual machine, like byte code. But there also exists source-code compilation, translating perhaps code in one high-level language into source code of another one.

Assuming one has written compiler for a given language A, where the compiler is written in B, for which one has a executable compiler on some platform H, then that compiler yields an executable compiler for A, as well. For instance, writing the first C^{++} compiler in C, and using an existing C compiler executable to get a executable C^{++} compiler.

There are two ways to *compose* those T-diagrams (see Figure 1.8).



(a) (b)

Figure 1.8: Composition of T-diagrams

Basically, one can chain the *results* of a compiler, or compile the compiler code itself. The first notion of composition is fairly obvious, it's just functional composition: if one can translate from A to B and from B to C, all using the host language H or host machine/platform), then one also can translate from A to C, still written in H (see Figure 1.8a).

The second form of composition is not much more complex (see Figure 1.8b). One can use the second compiler to compile H; in this scenario, H is typically not yet an executable, but the compiler is given in some more high-level format, So one compiles the compiler source code H. So, given a compiler from A to B in H, obviously, if one can translate H to K, provided one has an appropriate compiler for that, say in host-language or platform M.

For instance, if one has a compiler that translates Java to Java byte-code, and that compiler is written in C, then obviously one can a executable byte-code compiler by using a standard C-compiler and

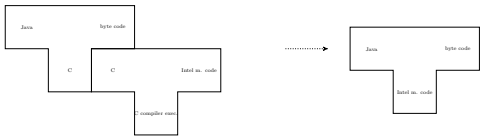


Figure 1.9: Composition of T-diagrams: compiling compiler code

As mentioned earlier, one very plausible scenario is also to use an existing "old" language to compiler for a new language and to use the existing language's compiler executable to get an executable for the new language (see Figure 1.10).

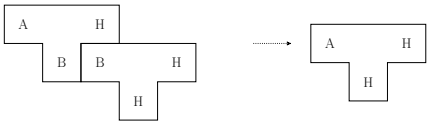


Figure 1.10: Compiler executable for a "new" language

That's the same diagram as form Figure 1.8b, ("compile-a-compiler"), more precisely a special case, with the letters shuffled around a bit. The interpretation is that one is given some platform or language H, for which one likes to implement a compiler of a "new" language A. To do so, i.e., as language in which the compiler is written, one chooses an "old" language B. "Old" in the sense that there exists already a compiler *on* H and translating *to* H. Not surprisingly, if you use that compiler (the right-hand "T") to compile the newly written one, one get a compiler for the new language *on* the host H and compiling *to* H.

1.3.2 Pulling oneself up on one's own bootstraps

Now it's time to look at bootstrapping again. That can be seen as situation from before in Figure 1.10, except that we replace B by A. In other words, we try to use the "new" language to write a compiler for itself. That's some kind of circularity which sounds suspicious. If you have already a working compiler (on H) for the new language, why would you use it to compile the compiler for A (and written in A)? And where does the compiler *on* H come from in the first place? The whole process however, is very common and is known as *bootstrapping*.

> bootstrap (verb, trans.): to promote or develop ... with little or no assistance
>
> — Merriam-Webster

(a) Compiler for A in A + compiler for A in machine code

(b) Compiler for A in A + compiler for A produced in the first stage

Figure 1.11: Bootstrapping

There is no magic here (see Figure 1.11). Let's assume we have designed a new language A and we have written a compiler for A in A itself (see the leftmost tombstone in Figure 1.11a). Now the problem is, how to get that compiler up and running on a some given platform, say H?

We cannot create a compiler out of nothing, we have to start from somewhere. So let's assume we have some simplistic, quick-and-dirty (Q&D) compiler for A running on H. So we need that machine-code compiler (the second tombstone) but it does not mean that we have to write that Q&D compiler in machine code from scratch. Of course we can use the approach explained before that we use an existing language with an existing compiler to create that machine-code version of the Q&D compiler. Only if we are really the pioneers doing the first high-level language ever, there's no way around and we need to write it assembler. But it does not need to be efficient, maybe not even covering all features of A, only the ones we used in writing our compiler for A.

At any rate, the outcome (the third, light-gray tombstone in Figure 1.11a, is a working compiler for A, only that it's not optimized, since the code we used to compile the compiler was just good enough to produce a correct outcome, without putting much effort on optimization.

Now, with a working executable compiler available, one can continue with stage 2, see Figure 1.11b. The outcome of that stage, the third, darker tombstone in Figure 1.11b, is another compiler executable for A. The compilation process resulting in that version may be slower than one would expect, but one need to do it only once. The resulting compiler is now produces fast code and is perhaps itself fact (assuming that when programming the compiler for A in A itself, one took care to include fancy optimization into the compiler). Of course there is always a balance: if one includes many aggressive the optimizations in a compiler to produce fast code and/or code with a small memory footprint, then the compilation itself typically takes longer.

1.3.3 Porting and cross-compilation

Assume there is a new platform H_1 and we want to get a compiler for our new language A for H_2 (assuming we have one already for the old platform H_1). It means that not only we want to compile *onto* H_1, but also, of course, that our compiler has run on H_2. These

are two requirements: (1) a compiler *to* H_2 and (2) a compiler to run *on* H_2. That leads to two stages.

(a) Stage 1 (b) Stage 2

Figure 1.12: Cross-compilation

In a first stage, we "rewrite" our compiler for A, currently targeted towards H_1, to the new platform H_2. If structured properly, it will only require to *port* or *re-target* the so-called *back-end* from the old platform to the new platform. If we have done that, we can use our executable compiler on H_1 to generate code for the new platform H_2. That's known as *cross-compilation*: use platform H_1 to generate code for platform H_2 (see Figure 1.12a). But now, that we have a (so-called cross-)compiler from A to H_2, running on the old platform H_2, we can use it to compile the retargeted compiler *again* (see Figure 1.12b)!

Not always is one interested in the second stage, porting the executable new compiler to the new platform. It may be that the new platform is resource restricted like a small embedded systems chip or part of a special purpose system. In that case it may not be possible or not part of the intended usage to run a compiler on it. So one simply uses the original platform H_1 to generate code for H_2, say some control software the embedded system. Similarly, if such special purpose settings, the source language may likewise be special, maybe a domain-specific language for control software running on embedded systems, to stick to the example. In that case, the language may not be ideal to program a compiler, i.e., instead of implementing the language A in A itself, one would implement A in some other language B. Also that would be cross-compilation.

Bibliography

[1] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

[2] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

Index