Course Script

INF 5110: Compiler construction

INF5110, spring 2021

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

1.1 Introduction

This is the script version of the slides shown in the lecture. It contains basically all the slides in the order presented (except that gradual overlays are not reproduced in a step-by-step manner). Normally, I try not to overload the slides with written information and rely on speaking and telling a story, with the slides as guidance. Such additional information, however, is presented in this script-version, so the document can be seen as an annotated version of the slides. Many explanations (hopefully) given during the lecture are written down here, but 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

Sources

Different from some previous semesters, one recommended book the course is Cooper and Torczon [2] besides also, as in previous years, Louden [3]. We will not be able to cover the whole book anyway (neither the full Louden [3] book). In addition the slides will take into account other sources, as well. Especially in the first chapters, for the so-called compiler front-end, the material is so "standard" and established, that it almost does not matter, which book to take.

As far as the exam is concerned: **Traditionally**, it has always been a written exam, and was "open book". This influences the style of the exam questions. In particular, there will be no focus on things one has "read" in one or the other pensum book; after all, one can bring along as many books as one can carry and look it up. Instead, the exam will require

to *do* certain constructions (analyzing a grammar, writing a regular expressions etc), so, besides reading background information, the best preparation is doing the exercises as well as working through previous exams.

For **spring 2021**: the exam form is not yet clear. On the web-page it's stated written home exam, but that will be discussed. Last semester (under corona), it was an **oral** exam, and that is definitely an option this semester as well.

Course material from:

A master-level compiler construction lecture has been given for quite some time at IFI. The slides are inspired by earlier editions of the lecture, and some graphics have just been clipped in and not (yet) been ported. The following list contains people designing and/or giving 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`)

This semester, **Gianluca Turin** (`gianlutu@ifi.uio.no`) will assist, for instance, with the exercises.

Course's web-page

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

- overview over the course, pensum (watch for updates)
- various announcements, beskjeder, etc.

Course material and plan

- based roughly on [2] and [3], but also other sources will play a role. A classic is "the dragon book" [1], we might use part of code generation from there
- see also *errata* list at `http://www.cs.sjsu.edu/~louden/cmptext/`
- approx. 3 hours teaching per week (+ exercises)
- mandatory assignments (= "obligs")
  - O_1 published mid-February, deadline mid-March
  - O_2 published beginning of April, deadline beginning of May
- group work up-to 3 people recommended. Please inform us about such planned group collaboration
- slides: see updates on the net

Exam

As mentioned, the status is unclear right now (at the beginning of the semester). The announcement on the net (home exam) **is not final**. It may be changed to **oral**.

Motivation: What is CC good for?

- not everyone is actually building a full-blown compiler, **but**
  - fundamental concepts and techniques in CC
  - most, if not basically all, software reads, processes/transforms and outputs "data"

  ⇒ often involves techniques central to CC
  - understanding compilers ⇒ deeper understanding of programming language(s)
  - new languages (domain specific, graphical, new language paradigms and constructs...)

  ⇒ CC & 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 properties of programs (in a full-scale programming language) are *undecidable* in general. "In general" means: for *all* programs. For one particular program or some restricted class of programs, semantical properties may well be 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 probably a complicated one. What is not possible is to *generally* prove a property about *all* programs (like whether it halts or not).

What limitations does that imply for compilers? The limitations concern 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 often may be 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 (it's a banality, since 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 unemployement... But the FET says: 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. Conceptually, each phase analyzes, enriches, transforms, etc. and afterwards hands the result over to the next phase.

This section is 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 not 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 an 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.

Architecture of a typical compiler

Anatomy of a compiler (removed)

Pre-processor

- either separate program or integrated into compiler

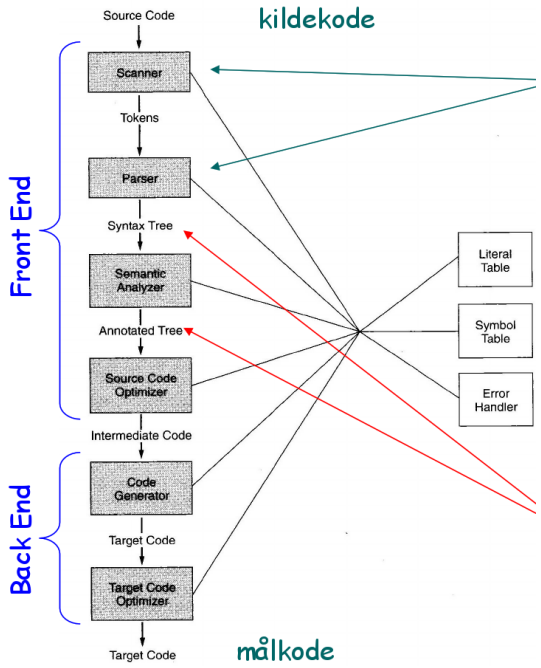


Figure 1.1: Structure of a typical compiler

- nowadays: C-style preprocessing sometimes seen as "hack" grafted on top of a compiler.
- examples (see next slide):
  - file inclusion
  - macro definition and expansion
  - conditional code/compilation: Note: `#if` is *not* the same as the `if`-programming-language construct.
- problem: often messes up the line numbers (among other things)

The C-prepocessor is called a "hack" on the slides. C-preprocessing is still 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 questions like "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 seens the "original" code.

In this lecture, more precisely, in this introductory remarks we talk about C-style preprocessing and C-style macros. That's because the C-preprocessor is the most prominent

example of preprocessors. As said, it has it's limitations, and has been accordingly criticized. Other languages don't have prerpocessors, for example Java. That latter statement is not 100% correct. Officially, there may not be one. Though if you google around you will find (not surprisingly) that the lack of a pre-processor let some people not sleep until they made one. But actually, if one like "C-style preprocessing", why would one actually need one? Preprocessing is massaging a textual file. The behavior is goverend by the pre-processor syntax (not C-syntax or C++-syntax or whatever). So if one has a file with Java code, and one misses directives like `#ifdefs` very much for some reason, one can just add those to the file and pump it through the pre-processor (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 (or C++ etc). 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 fundamental reason why it's not possible.

Another remarks concerns macros, not to leave the impression that macro programming is synonymous with pre-processing. Languages, for instance Rust, may offer more "integrated" macro facilites. More integrated in that they don't work on the ascii file, by 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 pre-processing concerns file inclusion using `#input`. 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: 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).

The lecture overall will *not* talk about preprocessing but focuses on the compiler itself.

C-style preprocessor examples

```
#include <filename>
```

Listing 1.1: file inclusion

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

Listing 1.2: Conditional compilation

Also languages like TEX, LATEX etc. support conditional compilation (e.g., `if<condition> ... else ... fi` in TEX). As a side remark: The sources for these slides and this script 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 ...

C-style preprocessor: macros

```
#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`. Conditional compilation is done with

`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`. and `#endif`. Definitions are done with `#define`.

Scanner (lexer ...)

- input: "the program text" (= string, char stream, or similar)
- task
  - *divide* and *classify* into *tokens*, and
  - remove blanks, newlines, comments ...
- theory: finite state automata, regular languages

The words lexer or scanner are synonymous. The task of the lexer is 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.

Scanner: illustration

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

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

0	
1	
2	`"a"`
	⋮
21	`"index"`
22	
	⋮

The terminology of tokens, token classes, lexemes, etc. will be made more clear in the chapters about lexing and parsing.

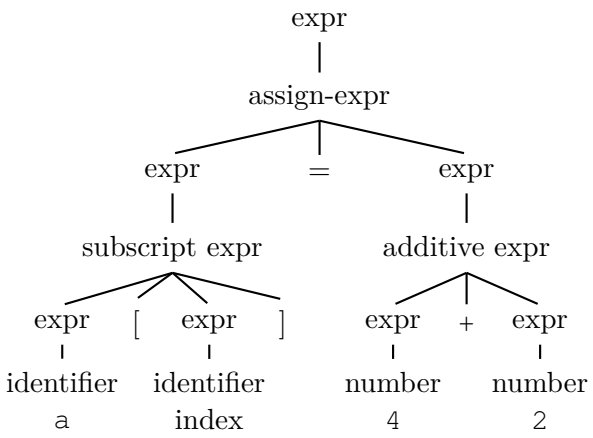
The input code snippet is supposed to be a sequence of characters (or a string). The blanks (space characters, or white spaces) are made specially visible (not just as blank spaces). The table afterwards shows the individual pieces of that string. Those pieces are called lexemes. Note that the white spaces are ignored in some way: there is no white-space lexeme (which is typical when do scanning). That does not mean that 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 written languages based on an lettered alphabet, like Western languages). We will see in the chapter about lexing, that there had for instance versions of Fortran, which treated white-space as completely meaningless, in that sense that it was treated as if not there at all. Here, in the example, 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). Note: 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 identifer, 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 (for instance here) as identifiers (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 alphabe), 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 aspects 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) can be seen as 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

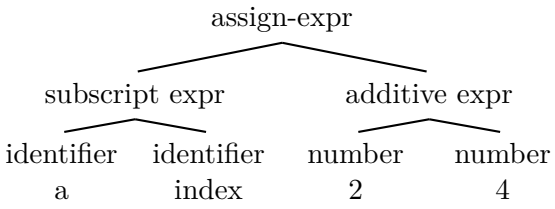
be covered by regular expressions resp. finite state automata, is handed over to the next phase(s), the next one being the parser, which is responsible for syntactic aspects. Those are ascpects that can be covered by some more expressive formalism, known as *context-free grammars*.

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 *conctrete 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.

`a[index] = 4 + 2`: parse tree/syntax tree



`a[index] = 4 + 2`: abstract syntax tree



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 are 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, ommitting 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. That kind of "fixing" is not meant, what is fixed is the general format of allowed parse trees. 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, ambigous grammars are unwelcome, and parsers realize typically unambigous grammars. Parser generators (like `yacc` and similar), when fed with an ambigous 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 it's not a good sign. It typically indicates troubles with the grammar.

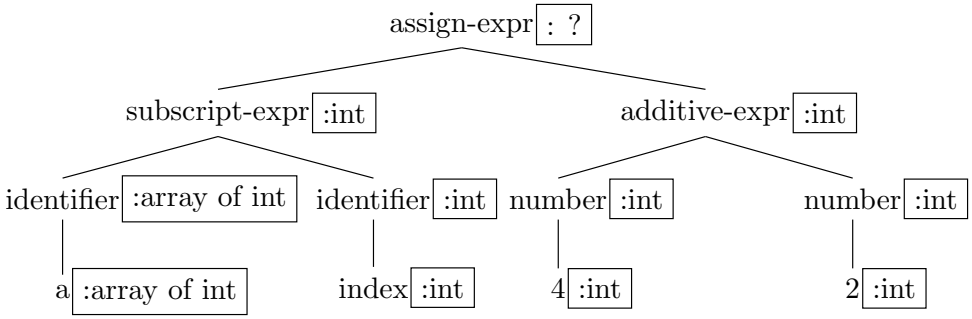
To avoid misconceptions: an ambiguous grammar will lead to conflicts in such tools, but the other way around is not true: a parser may indicate conflicting situations even if the grammar is unambigous. The reason is that parsers typically are not expressive enough to cover all kinds of context-free grammars, not even all unambiguious 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.

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 that comes after parsing is 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.

(One typical) Result of semantic analysis

- one standard, general outcome of semantic analysis: "annotated" or "decorated" AST
- additional info (non context-free):
  - *bindings* for declarations
  - (static) *type* information



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

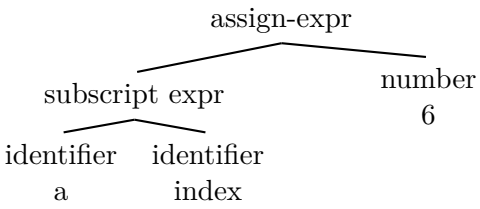
Non-negotionable is, of course, to generate *correct* code, i.e., code that correctly and for all program reflects the language's intended semantics. In particular, it needs to realize all the fancy programming abstractions the language may offer. Even variables are abstractions, they may "feel" like 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 propery. 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 inutive 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 compromizing 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 systemantic with techniques called *data-flow* analyses. We don't do too much systematic data flow analysis or agressive optimization in this lecture, but we will cover one important such analysis called *liveness* analysis.

Optimization at source-code level



1

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

2

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

3

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

The code examples show 3 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, in the steps here, 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 actually 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 considers it as "source code" and no

one tries seriously 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. 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 pre-processing, 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.

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
```

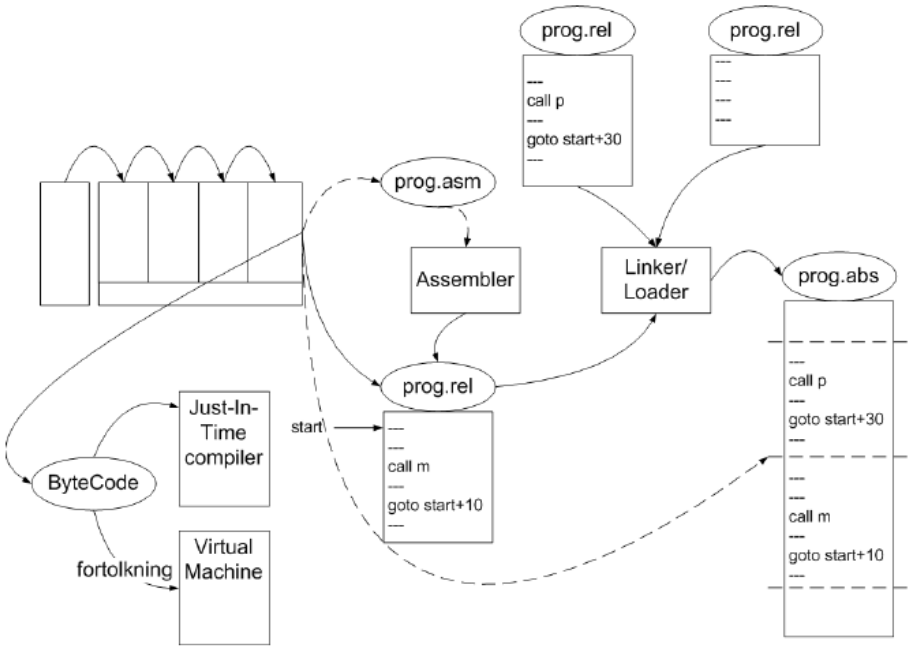
```
MOV␣R0,␣index␣␣␣␣␣␣;;␣value␣of␣index␣->␣R0
SHL␣R0␣␣␣␣␣␣␣␣␣␣␣␣;;␣double␣value␣in␣R0
MOV␣&a[R0],␣6␣␣␣␣␣␣;;␣const␣6␣->␣address␣a+R0
```

- *many* optimizations possible
- potentially difficult to automatize[1], 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 "optmize" 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.

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

Anatomy of a compiler (2)



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 adresses, 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.

Misc. notions

- front-end vs. back-end, analysis vs. synthesis
- separate compilation
- how to handle *errors*?
- "data" handling and management at run-time (static, stack, heap), garbage collection?
- language can be compiled in *one pass*?
  - E.g. C and Pascal: declarations must *precede* use

  - no longer too crucial, enough memory available
- compiler assisting tools and infrastructure, e.g.
  - debuggers
  - profiling
  - project management, editors
  - build support
  - ...

Compiler vs. interpeter

compilation

- classical: source ⇒ machine code for given machine
- different "forms" of machine code (for 1 machine):
  - executable ⇔ relocatable ⇔ textual assembler code

full interpretation

- directly executed from program code/syntax tree
- often for command languages, interacting with the OS.
- speed typically 10–100 slower than compilation

compilation to intermediate code which is interpreted

- used in e.g. Java, Smalltalk, ...
- intermediate code: designed for efficient execution (byte code in Java)
- executed on a simple interpreter (JVM in Java)
- typically 3–30 times slower than direct compilation
- in Java: byte-code ⇒ machine code in a just-in time manner (JIT)

More recent compiler technologies

- *Memory* has become cheap (thus comparatively large)
  - keep whole program in main memory, while compiling
- OO has become rather popular
  - special challenges & optimizations
- Java
  - "compiler" generates byte code
  - part of the program can be *dynamically* loaded during run-time
- concurrency, multi-core
- virtualization
- graphical languages (UML, etc), "meta-models" besides grammars

1.3 Bootstrapping and cross-compilation

Let's just glance over this section, we will not discuss in much in class. It's not part of the pensum for the written exam (and also not for the oral), but may be interesting. 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, 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 out 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 described 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. That's a tall order; one would rather avoid using assembler (except perhaps for carefully selected special tiny subtasks) and make use of a high-level language, with all its abstractions and other infrastructure, like libraries, editors, configuration and version management etc. and perhaps even textbooks, lectures, tutorials and training.

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. 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-marked for electronic computers, where ordering a computer means ordering a 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 it was kind of rudimentary compared to modern situation, barely able to process "jobs" (by reading punchcards perhaps) and controling other peripherals.

In such a situation, no compiler for a high-level language may be available (maybe not even any high-level language yet, bevause one is the first one, who not only proposes to use high-level languages, but who actually implements 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 sense), one goes gradually. One starts with a simple version of some relevant aspects of the planned language, not optimized, 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

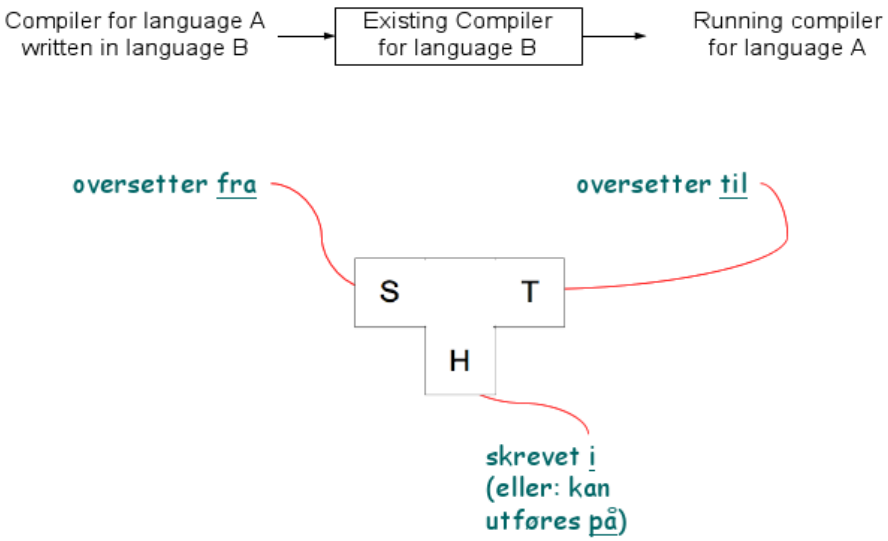
version that is strong enough to compiler *itself* without much reliance on assembler as "source code".

Historically, the development of the language C went hand-in-hand with the development of Unix, insofar it was a larger "bootstrapping problem". How to develop a (at that time) modern 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 basically the norm.

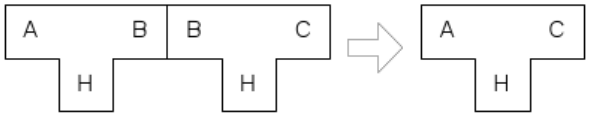
The compilation process is here illustrated with so-called *T-diagrams* which is some "graphical" representation of the compilation process, mentioning the input language, the output language, and the language in which the compiler is represented as the three arms of the "T".

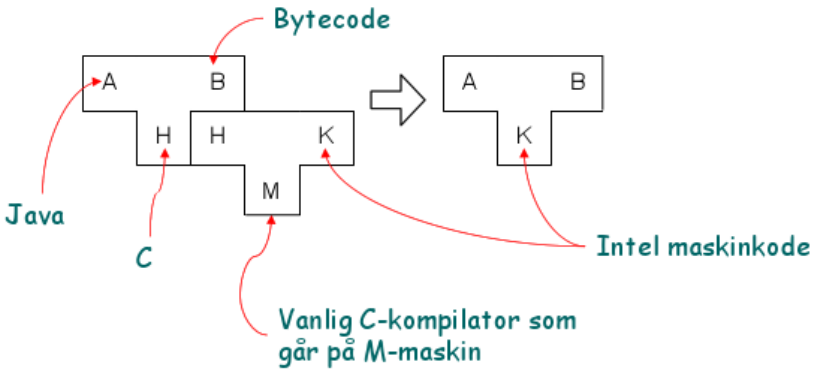
Compiling from source to target on host

"tombstone diagrams" (or T-diagrams)....

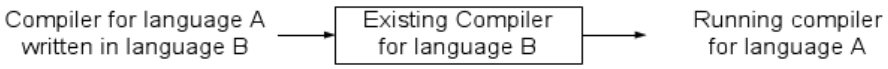


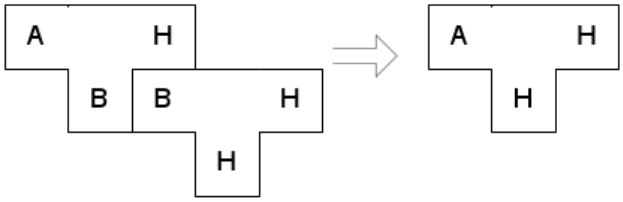
Two ways to compose “T-diagrams”





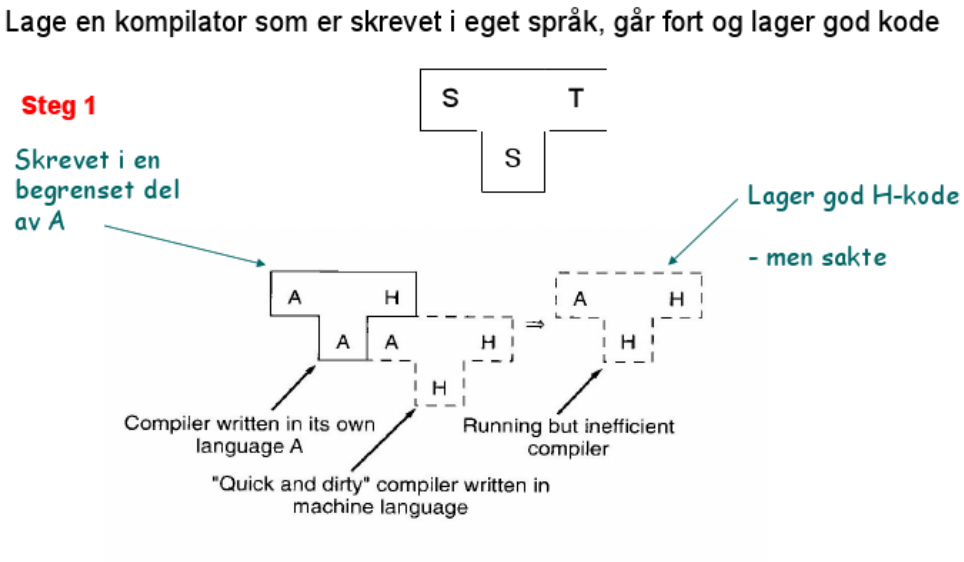
Using an “old” language and its compiler for write a compiler for a “new” one





Pulling oneself up on one’s own bootstraps

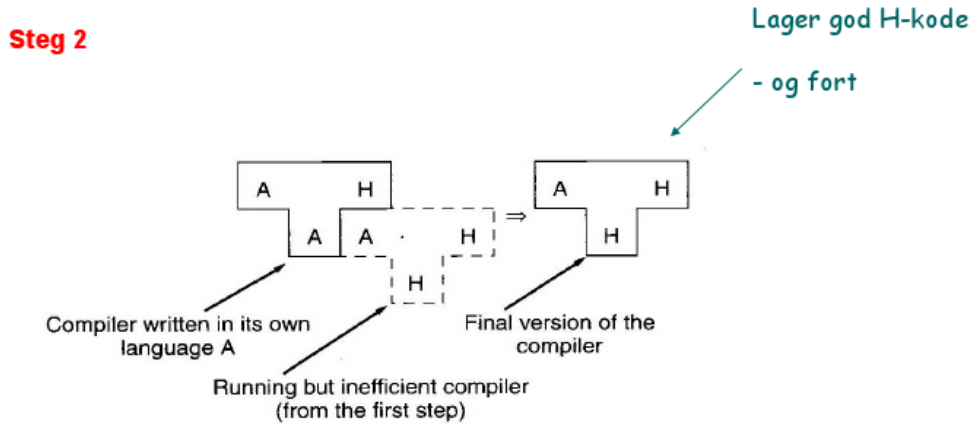
> bootstrap (verb, trans.): to promote or develop . . . with little or no assistance
>
> — Merriam-Webster



There is no magic here. The first thing is: the "Q&D" compiler in the diagram is said to be in machine code. If we want to run that compiler as executable (as opposed to being interpreted, which is ok too), of course we need machine code, but it does not mean that *we* have to write that Q&D compiler in machine code. 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.

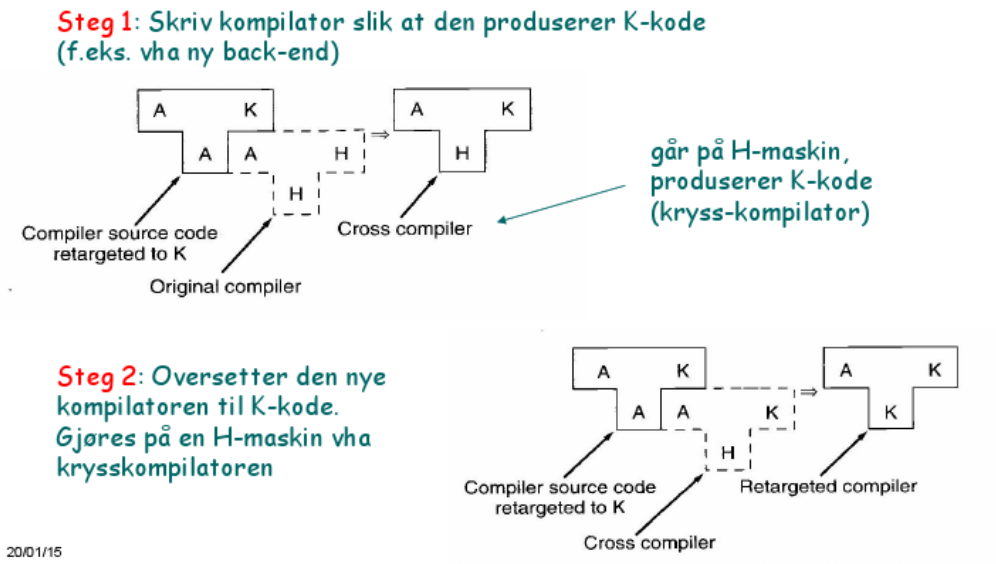
Furthermore: when talking about *efficiency* of a compiler, we mean (at least here) exactly that: it's the compilation process itself which is inefficent! As far as efficency goes, one the one hand the compilation process can be efficient or not, and on the other the generated code can be (on average and given competen programmers) be efficent not. Both aspects are not independent, though: to generate very efficient code, a compiler might use many and aggressive optimizations. Those may produce efficient code but cost time to do. At the first stage, we don't care how long it takes to compile, and *also* not how efficient is the *code it produces!* Note the that code that it produces is a compiler, it's actually a second version of "same" compiler, namely for the new language A to H and on H. We don't care how efficient the generated code, i.e., the compiler is, because we use it just in the next step, to generate the final version of compiler (or perhaps one step further to the final compiler).

Bootstrapping 2



Porting & cross compilation

- Har: A kompilator som oversetter til H-maskinkode
- Ønsker: A-kompilator som oversetter til K-maskin kode



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

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

In a first stage, we "rewrite" our compiler for A, targeted towards H, to the new platform K. 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 to generate code for the new platform K. That's known as *cross-compilation*: use platform H to generate code for platform K.

But now, that we have a (so-called cross-)compiler from A to K, running on the old platform H, we use it to compile the retargeted compiler *again*!

Bibliography

[1] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

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

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

Index