Course Script

INF 5110: Compiler construction

INF5110/ spring 2018

Martin Steffen

Contents

Chapter 1 Introduction

What is it about?

Learning Targets of this Chapter

The chapter gives basically an overview over different phases of a compiler and their tasks.

Contents

1.1 Introduction

Course info

Sources

Different from previous semesters, one "official" recommended book the course is based upon is [2] (in previous years it was mostly [3]. We will not be able to cover the whole book (neither the full [3] book). In addition the slides will draw on other sources, as well. Especially in the first chapters (the front-end), the material is so "standard" and established, that it almost does not matter, which book to take.

Course material from:

- 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`

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

Course material and plan

- Material: based largely on [2] (previously [3] which also is fine)), but also other sources will play a role. A classic is "the dragon book" [**?**], 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
- mandatory assignments (= "obligs")
  - O1 published mid-February, deadline mid-March
  - O2 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**: (if written one) *12th June, 09:00*, 4 hours.

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 language (domain specific, graphical, new language paradigms and constructs. . .)

  ⇒ CC & their principles will *never* be "out-of-fashion".

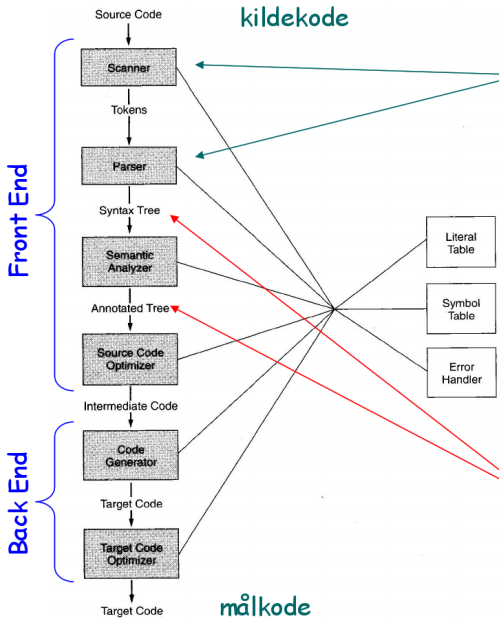
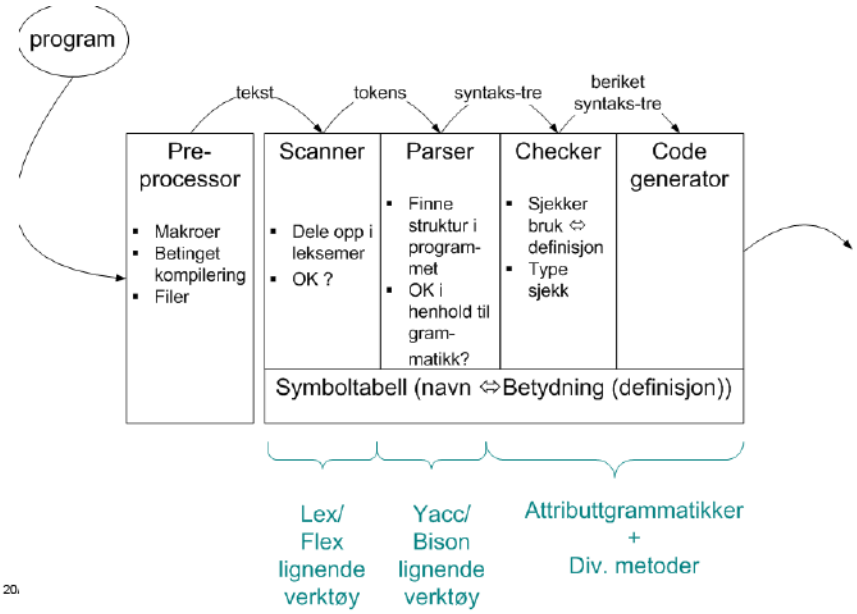


Figure 1.1: Structure of a typical compiler

1.2 Compiler architecture & phases

Architecture of a typical compiler

Anatomy of a compiler



Pre-processor

- either separate program or integrated into compiler
- nowadays: C-style preprocessing mostly seen as "hack" grafted on top of a compiler.[1]
- examples (see next slide):
  - file inclusion[2]
  - macro definition and expansion[3]
  - conditional code/compilation: Note: `#if` is *not* the same as the `if`-programming-language construct.
- problem: often messes up the line numbers

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, LATEXetc. support conditional complication (e.g., `if<condition> ... else ... fi` in TEX). These slides and this script makes quite some use of it: some text shows up only in the handout-version, etc.

C-style preprocessor: macros

[1] C-preprocessing is still considered sometimes a *useful* hack, otherwise it would not be around ... But it does not naturally encourage elegant and well-structured code, just quick fixes for some situations.

[2] the single most primitive way of "composing" programs split into separate pieces into one program.

[3] Compare also to the `\newcommand`-mechanism in LATEX or the analogous `\def`-command in the more primitive TEX-language.

```
#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`. Comditional 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

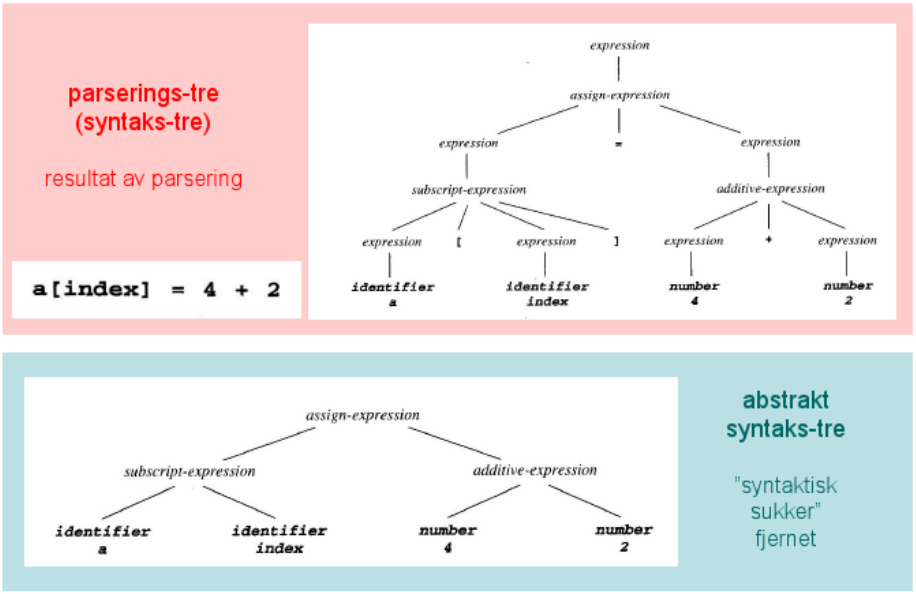
Scanner: illustration

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

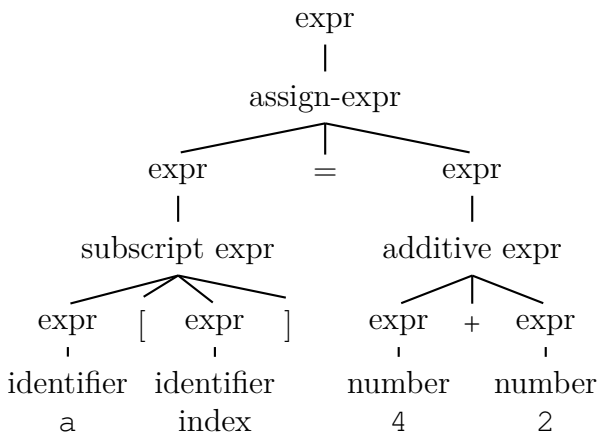
lexeme	token class	value
a	*identifier*	"a" 2
[	*left bracket*	
index	*identifier*	"index" 21
]	*right bracket*	
=	*assignment*	
4	*number*	"4" 4
+	*plus sign*	
2	*number*	"2" 2

0	
1	
2	"a"
	⋮
21	"index"
22	
	⋮

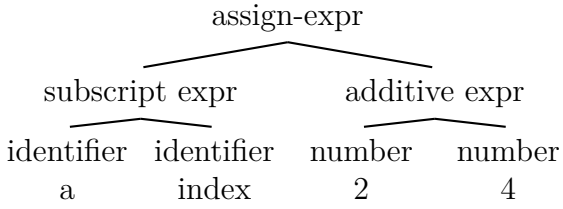
Parser



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



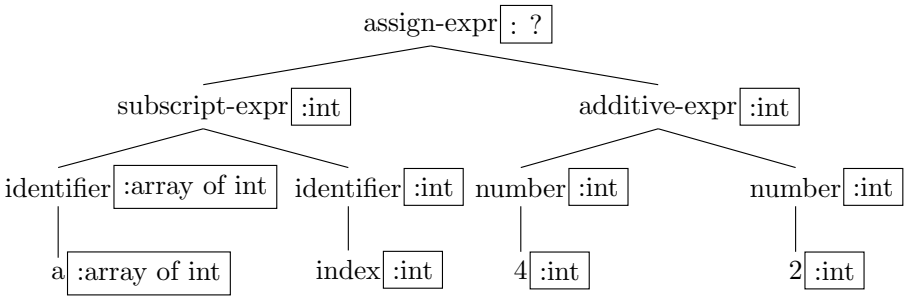
`a[index] = 4 + 2`: abstract syntax tree



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 tree is less verbose that the parse three which is sometimes also called concrete syntax tree. The parse tree(s) for a given word are fixed by the *grammar*. The abstract syntax tree is a bit a matter of design (but of course, the grammar is also a matter of design, but once the grammar is fixed the 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).

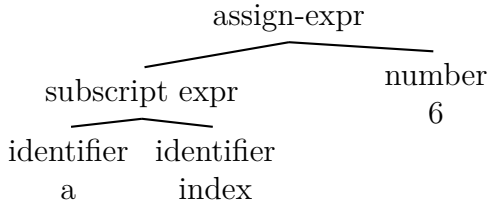
(One typical) Result of semantic analysis

- 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.

Optimization at source-code level



1

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

2

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

3

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

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) at various phases on the compiler. What is also typical is, that there are many different optimizations building upon each other. First, optmization A is done, then, taking the result, optimization B is done 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
```

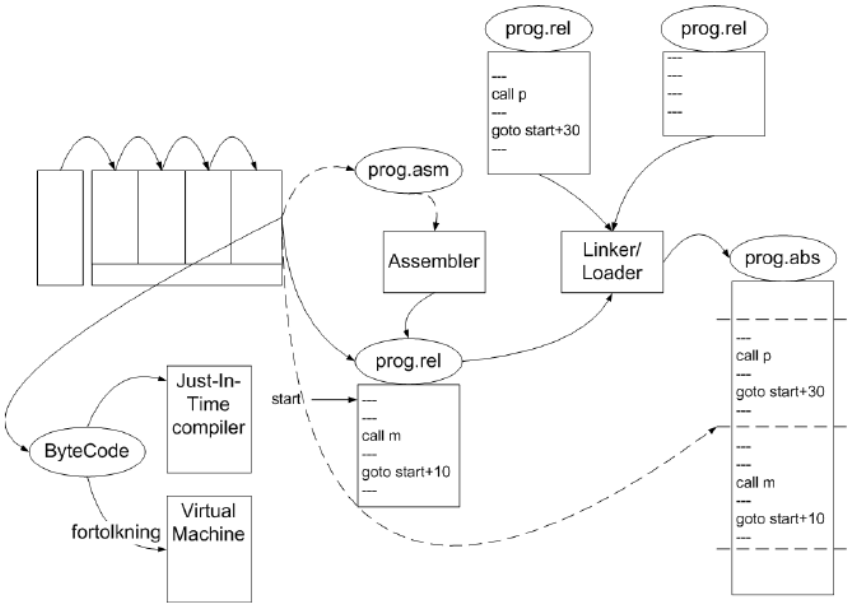
```
MOV␣R0,␣index␣␣␣␣␣␣;;␣value␣of␣index␣->␣R0
SHL␣R0␣␣␣␣␣␣␣␣␣␣␣␣;;␣double␣value␣in␣R0
MOV␣&a[R0],␣6␣␣␣␣␣␣;;␣const␣6␣->␣address␣a+R0
```

- *many* optimizations possible
- potentially difficult to automatize[4], based on a formal description of language and machine
- platform dependent

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

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 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 assmbler 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.

Anatomy of a compiler (2)



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 OS etc.
- 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)

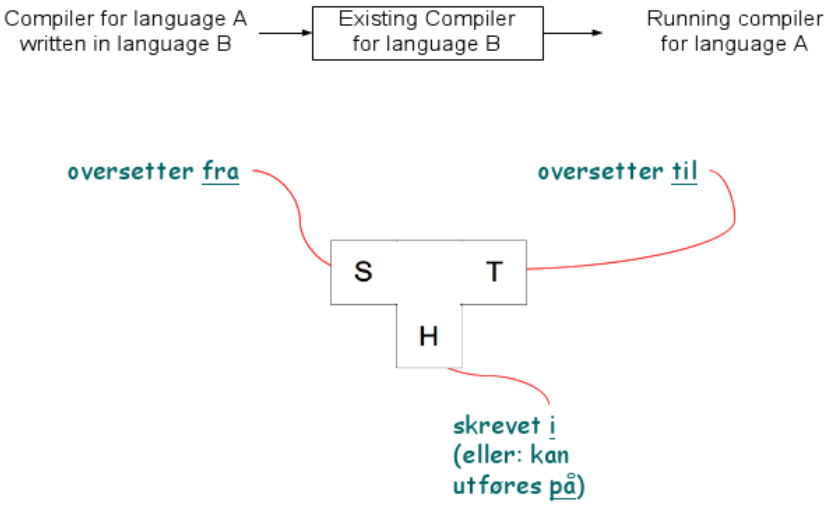
More recent compiler technologies

- *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
- graphical languages (UML, etc), "meta-models" besides grammars

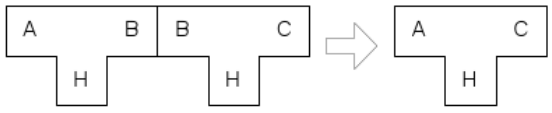
1.3 Bootstrapping and cross-compilation

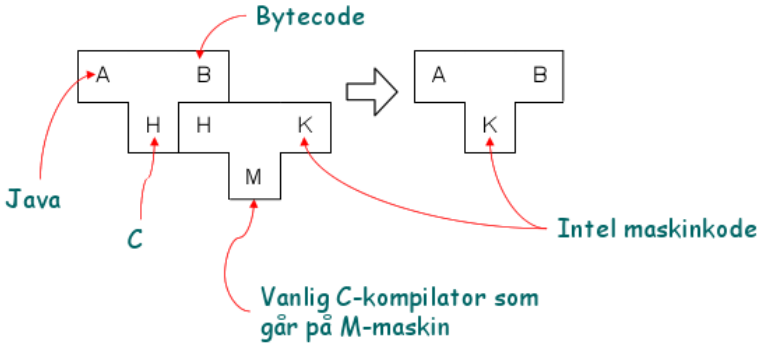
Compiling from source to target on host

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

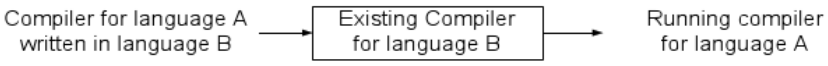


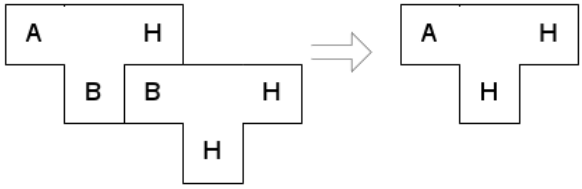
Two ways to compose "T-diagrams"





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

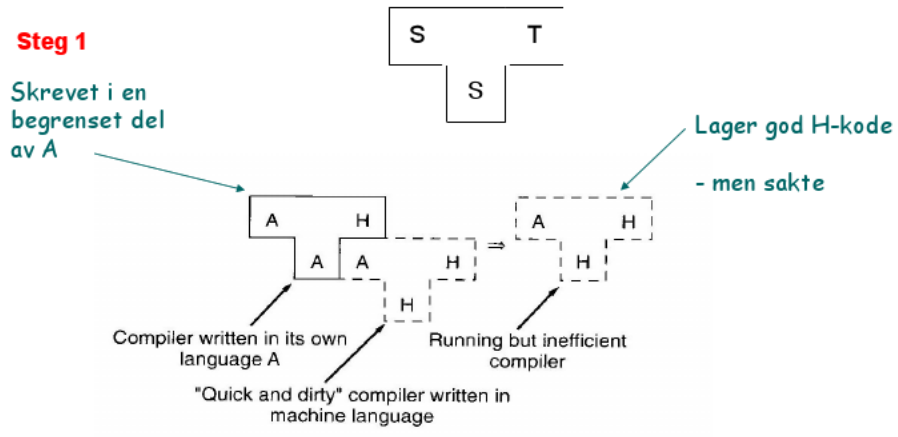




Pulling oneself up on one’s own bootstraps

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

Lage en kompilator som er skrevet i eget språk, går fort og lager god kode

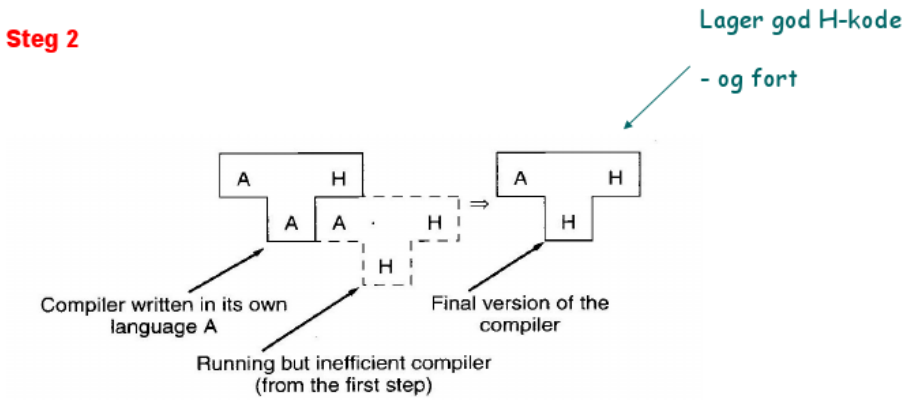


Explanation

There is no magic here. The first thing is: the "Q&D" compiler in the diagram is said do 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 here exactly that here: 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. In 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).

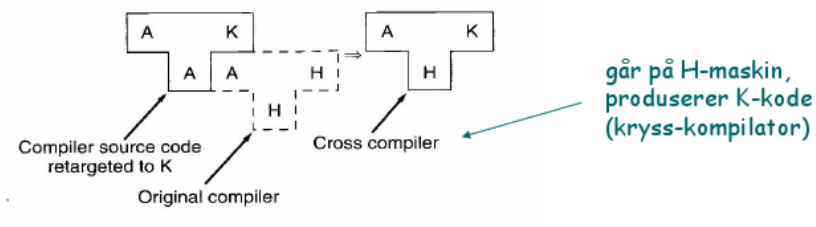
Bootstrapping 2



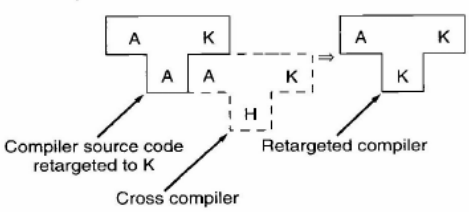
Porting & cross compilation

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

Steg 1: Skriv kompilator slik at den produserer K-kode (f.eks. vha ny back-end)



Steg 2: Oversetter den nye kompilatoren til K-kode. Gjøres på en H-maskin vha krysskompilatoren



20/01/15

Explanation

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 *retarget* 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*!

Chapter 4
References

Bibliography

[] 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