Compiler construction

Martin Steffen

January 19, 2016

Contents

1 Introduction

19. 01. 2016

1.1 Introduction

1. Course info

   (a) Course presenters:

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

   (b) 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.

2. Course material and plan

   - The material is based largely on [Louden, 1997], but also other sources will play a role. A classic is "the dragon book" [Aho et al., 1986]
   - 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**: *8th June, 14:30*, 4 hours.

3. 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".

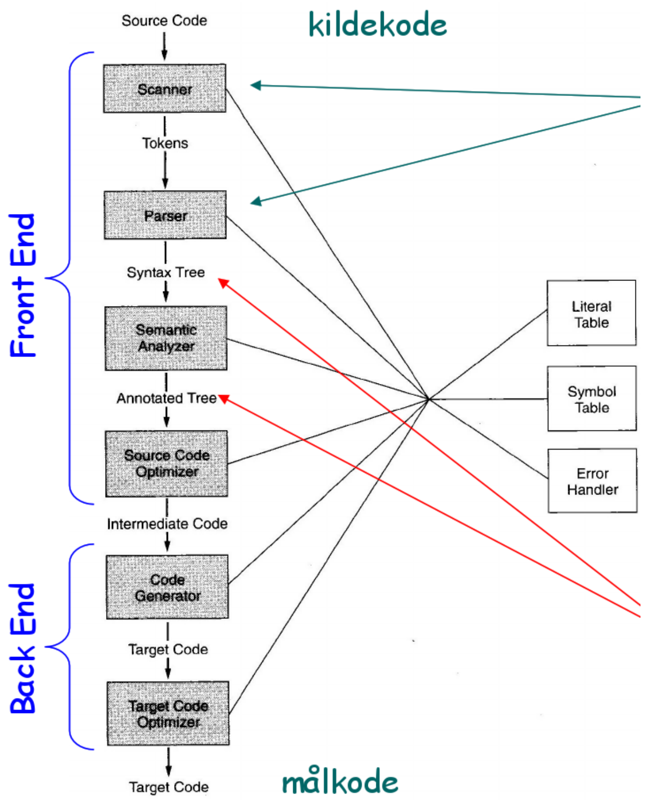
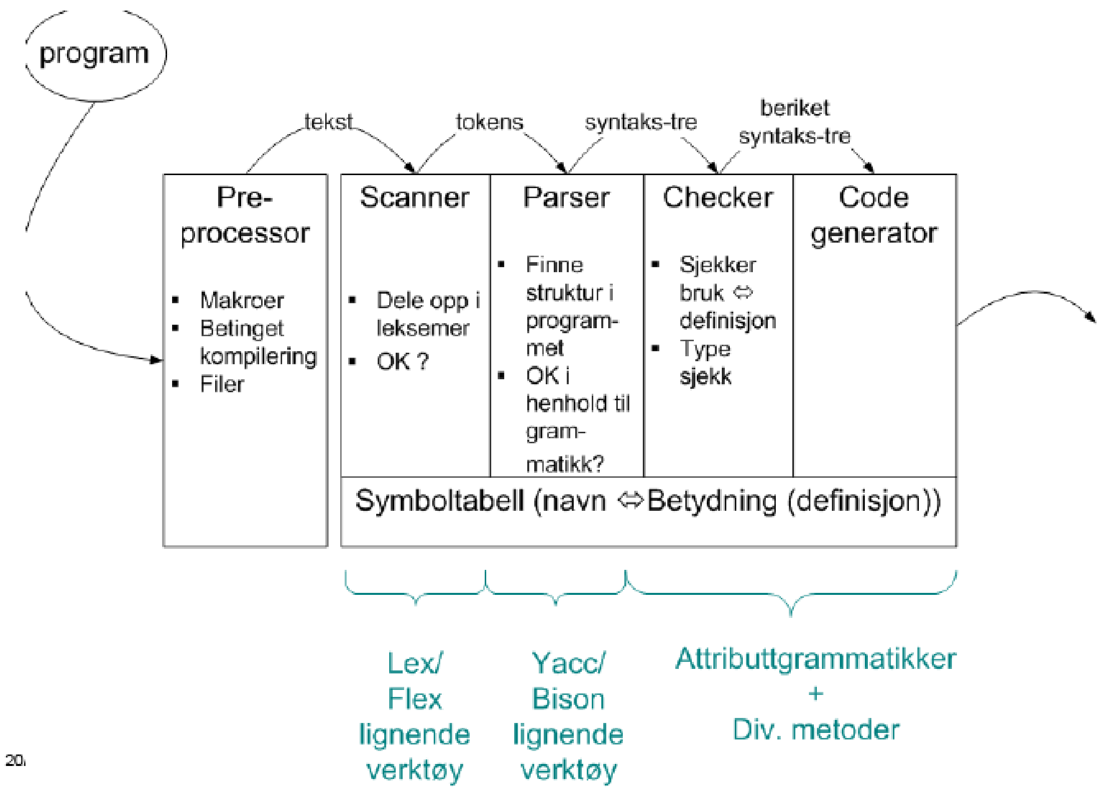


Figure 1: Structure of a typical compiler

1.2 Compiler architecture & phases

1. Architecture of a typical compiler
2. Anatomy of a compiler



program
tekst
tokens
syntaks-tre
beriket syntaks-tre
Pre-processor
▪ Makroer
▪ Betinget kompilering
▪ Filer
Scanner
▪ Dele opp i leksemer
▪ OK ?
Parser
▪ Finne struktur i program-met
▪ OK i henhold til gram-matikk?
Checker
▪ Sjekker bruk ⇔ definisjon
▪ Type sjekk
Code generator
Symboltabell (navn ⇔Betydning (definisjon))
Lex/ Flex lignende verktøy
Yacc/ Bison lignende verktøy
Attributtgrammatikker + Div. metoder
20

3. 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

4. C-style preprocessor examples

```
#include <filename>
```

Listing 1: file inclusion

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

Listing 2: Conditional compilation

5. C-style preprocessor: macros

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

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

Listing 3: Macros

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

6. 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

7. Scanner: illustration

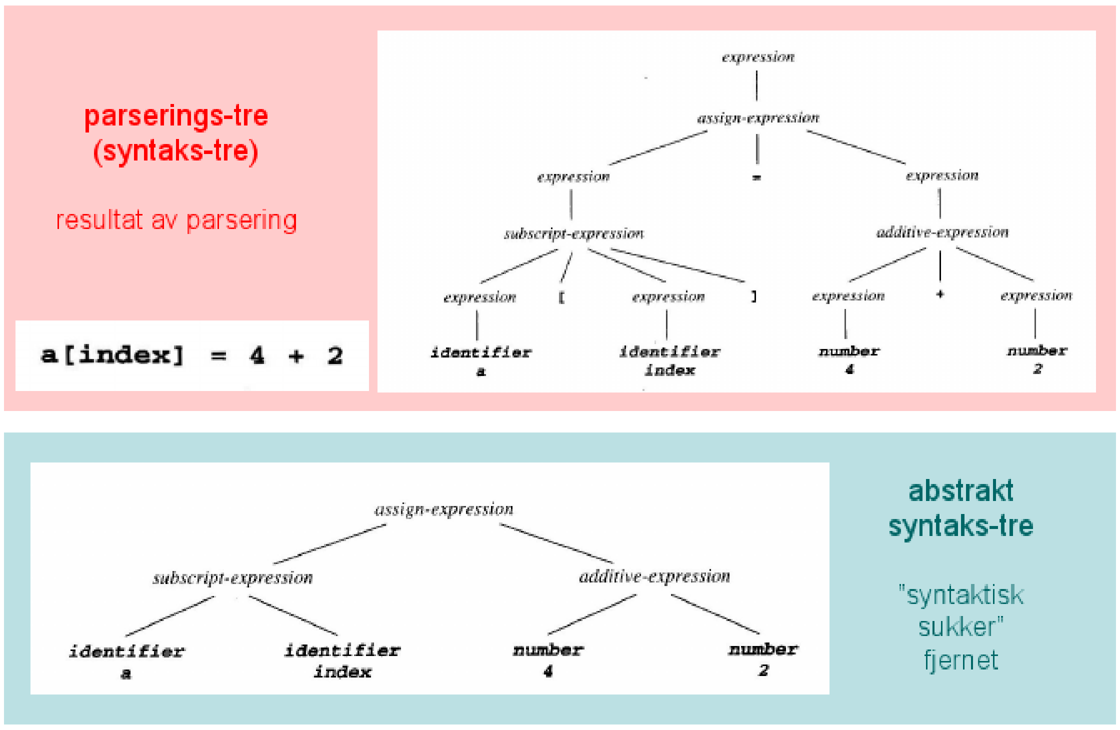
```
a[index]␣=␣4␣+␣2
```

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

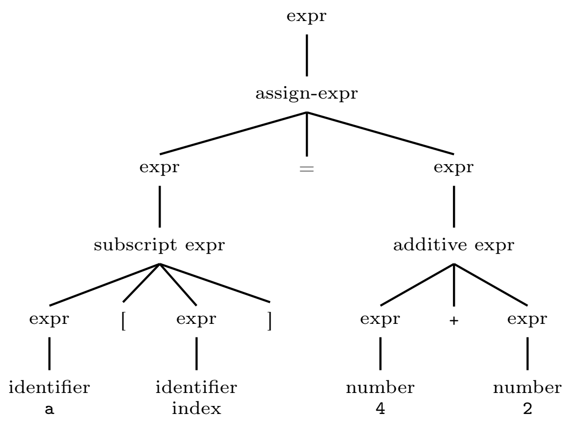
0	
1	
2	`"a"`
	⋮
21	`"index"`
22	
	⋮

[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.

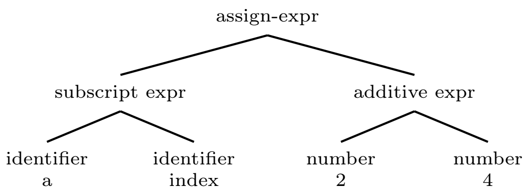
8. Parser



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

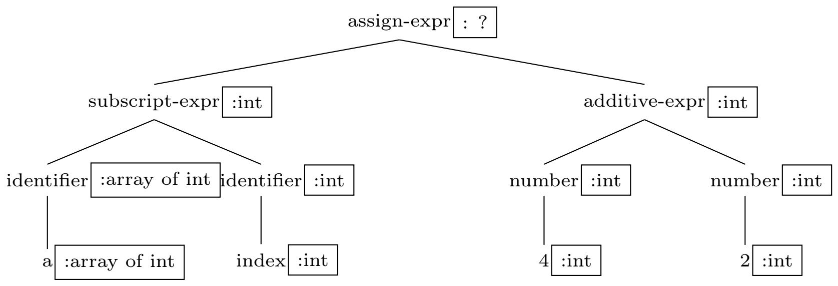


10. `a[index] = 4 + 2`: abstract syntax tree



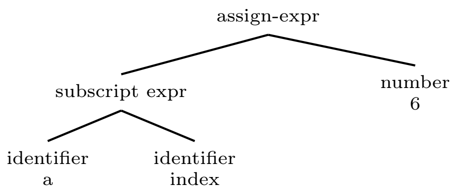
11. (One typical) Result of semantic analysis

- 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.

12. Optimization at source-code level



(a) 1

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

(b) 2

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

(c) 3

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

13. Code generation & optimization

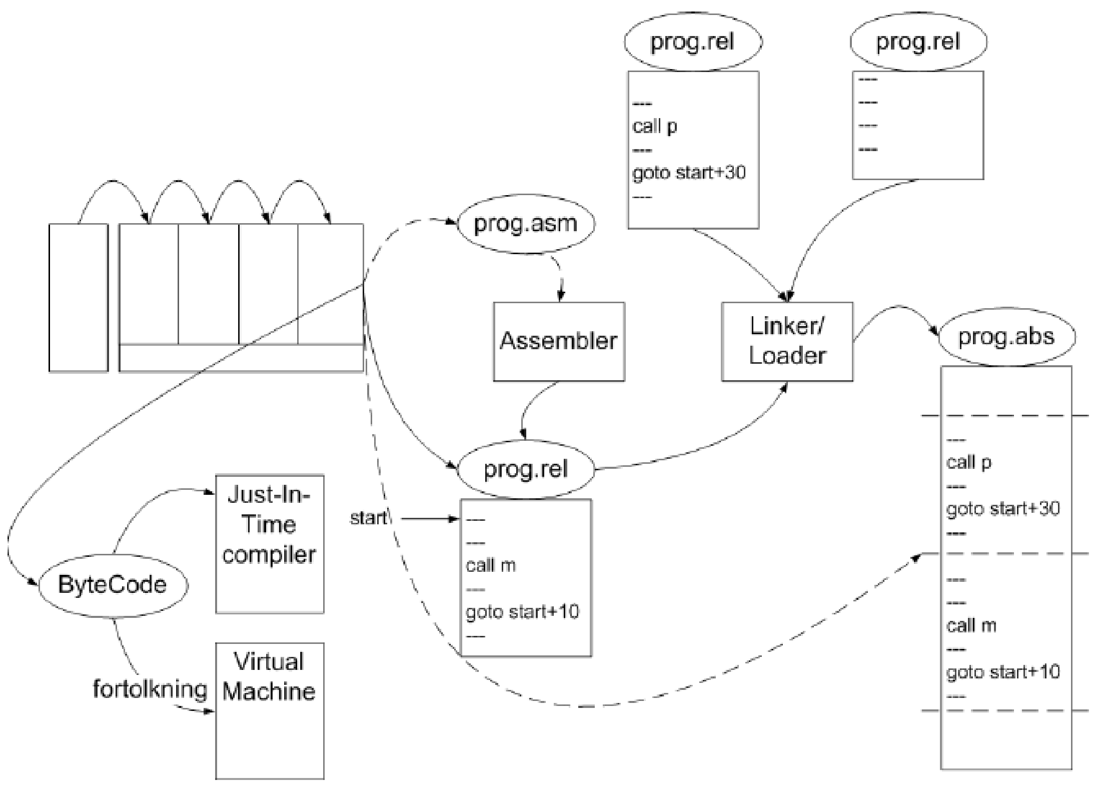
```
MOV  R0, index  ;;  value of index -> R0
MUL  R0, 2      ;;  double value of R0
MOV  R1, &a     ;;  address of a -> R1
ADD  R1, R0     ;;  add R0 to R1
MOV *R1, 6      ;;  const 6 -> address in R1
```

```
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

14. Anatomy of a compiler (2)

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



15. 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 tool and infra structure, e.g.
  - debuggers
  - profiling
  - project management, editors
  - build support
  - ...

16. Compiler vs. interpeter

(a) **Compilation**

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

(b) full **interpretation**

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

(c) 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)

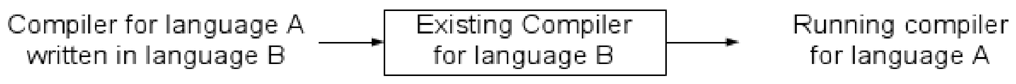
17. More recent compiler technologies

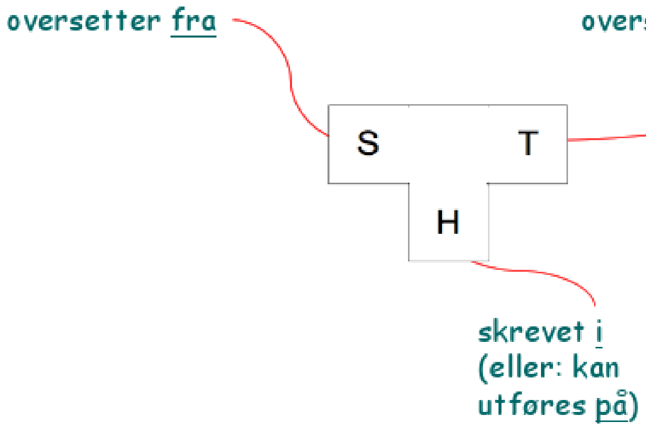
    - *Memory* has become cheap (thus comparatively large)
        – keep whole program in main memory, while compiling
    - OO has become rather popular
        – special challenges & optimizations
    - Java
        – "compiler" generates byte code
        – part of the program can be *dynamically* loaded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

1.3 Bootstrapping and cross-compilation

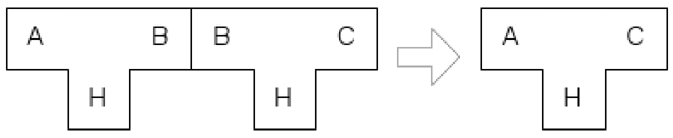
1. Compiling from source to target on host

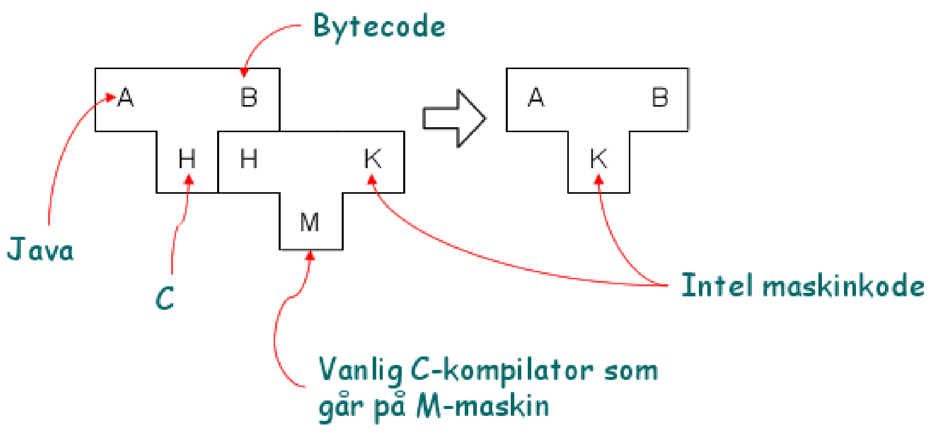
   "tombstone diagrams" (or T-diagrams). . . .


Compiler for language A written in language B
Existing Compiler for language B
Running compiler for language A

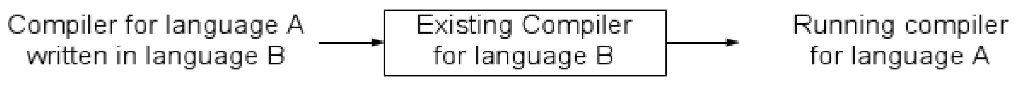


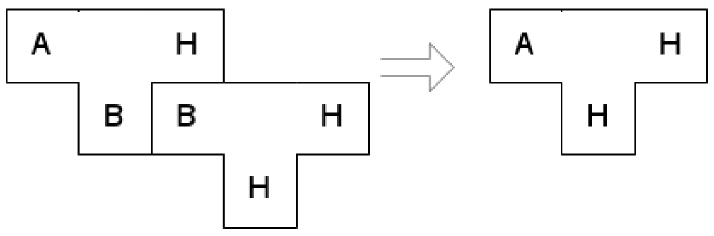
2. Two ways to compose "T-diagrams"





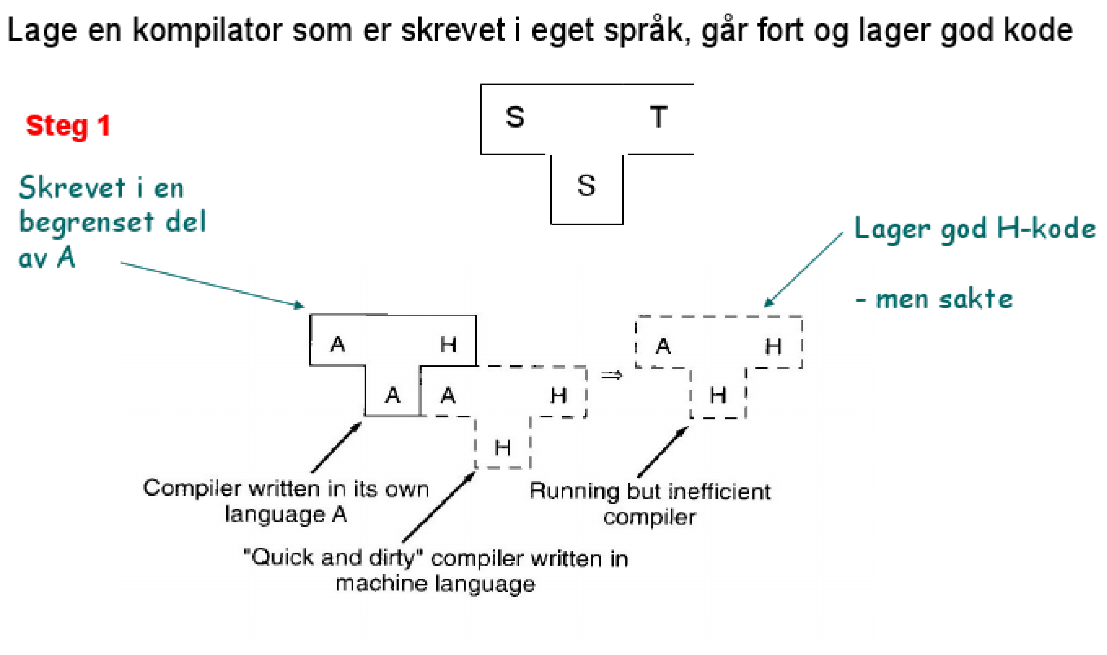
3. Using an “old” language and its compiler for write a compiler for a “new” one





4. 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
Steg 1
S T S
Skrevet i en begrenset del av A
Lager god H-kode
- men sakte
A H A A H H
A H H
Compiler written in its own language A
Running but inefficient compiler
"Quick and dirty" compiler written in machine language

(a) 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.

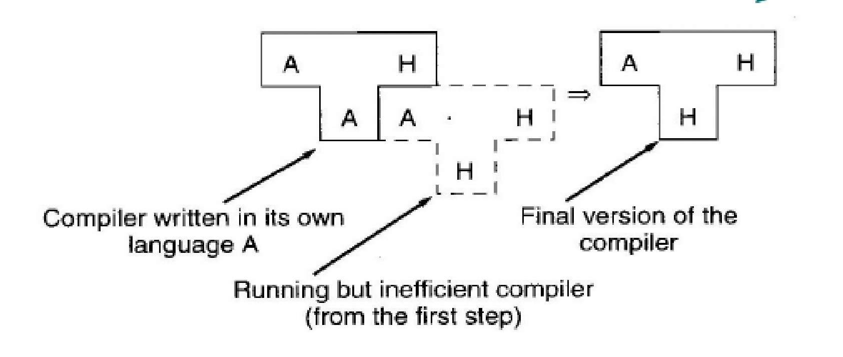
Furthermore: when talking about *efficiency* of a compiler, we mean here exactly that here: it’s the compilation process itself which is ineffcient! As far as efficency goes, one the one hand the compilation process can be efficient or not, and on the other the generated code can be (on average and given 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).

5. Bootstrapping 2

Steg 2

Lager god H-kode

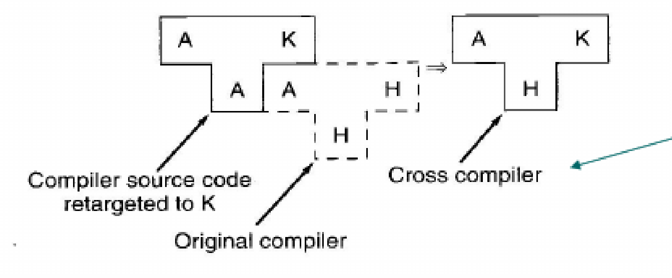
- og fort


A
H
A
A
H
H
⇒
A
H
H
Compiler written in its own language A
Running but inefficient compiler (from the first step)
Final version of the compiler

6. Porting & cross compilation

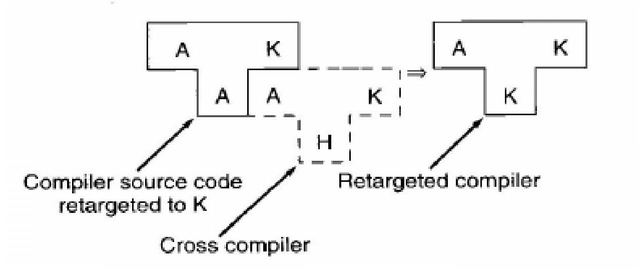
- Har: A kompilator som oversetter til H-maskinkode
- Ønsker: A-kompilator som oversetter til K-maskin kode

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



går på H-maskin, produserer K-kode (kryss-kompilator)

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



20/01/15

(a) 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 *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*!

References

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

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

Index