Chapter 1

Introduction

Course “Compiler Construction”
Martin Steffen
Spring 2018

Chapter 1

Learning Targets of Chapter “Introduction”.

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

Chapter 1

Outline of Chapter “Introduction”.

Introduction

Compiler architecture & phases

Bootstrapping and cross-compilation

Section

Introduction

Chapter 1 “Introduction”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Course info

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” [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
- 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".

Section

Compiler architecture & phases

Chapter 1 “Introduction”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Architecture of a typical compiler

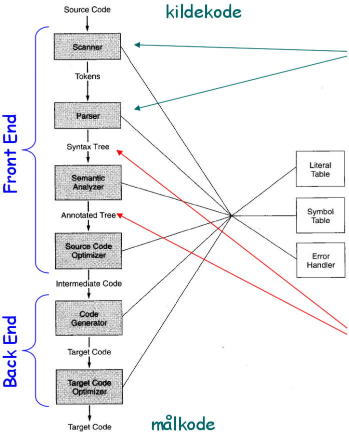
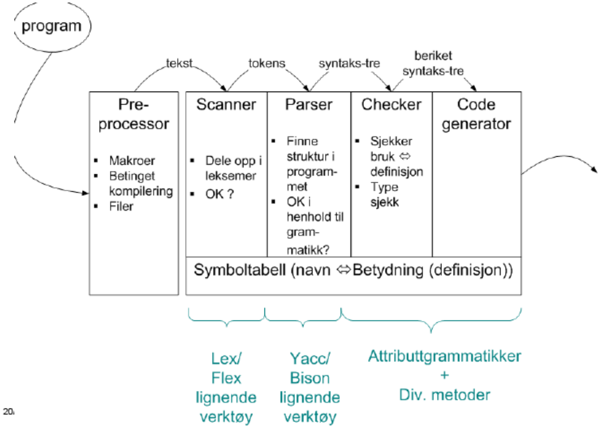


Figure: Structure of a typical compiler

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 programmet
- OK i henhold til grammatikk?

Checker

- Sjekker bruk ⇔ definisjon
- Type sjekk

Code generator

Symboltabell (navn ⇔Betydning (definisjon))

Lex/ Flex lignende verktøy

Yacc/ Bison lignende verktøy

Attributtgrammatikker + Div. metoder

20

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

[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.

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

C-style preprocessor: macros

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

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

Listing 3: Macros

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

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"
[	*left bracket*	
index	*identifier*	"index"
]	*right bracket*	
=	*assignment*	
4	*number*	"4"
+	*plus sign*	
2	*number*	"2"

Scanner: illustration

```
a[index]␣=␣4␣+␣2
```

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

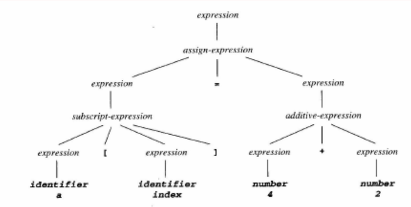
0	
1	
2	"a"
	⋮
21	"index"
22	
	⋮

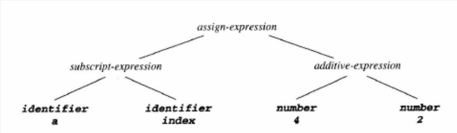
Parser

parserings-tre
(syntaks-tre)

resultat av parsering

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

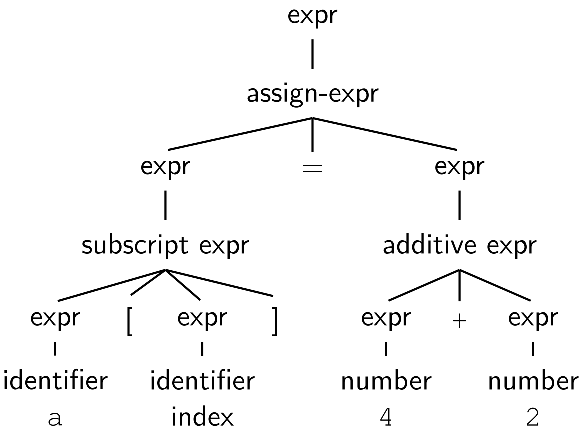




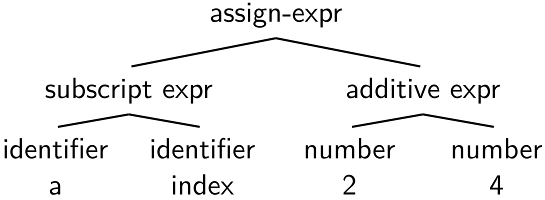
abstrakt
syntaks-tre

"syntaktisk
sukker"
fjernet

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

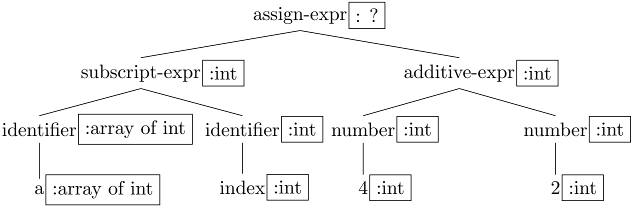


a[index] = 4 + 2: abstract syntax tree



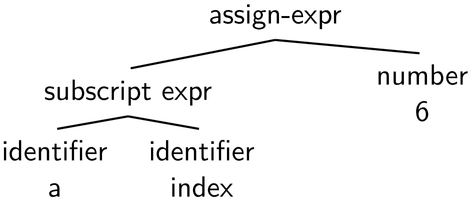
(One typical) Result of semantic analysis

- 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



```
t = 4+2;        t = 6;
a[index] = t;  a[index] = t;  a[index] = 6;
```

Code generation & optimization

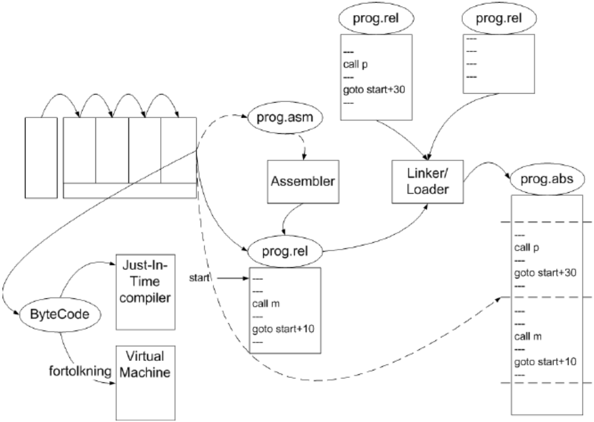
```
MOV  R0, index  ;;  value of index -> R0
MUL  R0, 2      ;;  double value of R0
MOV  R1, &a     ;;  address of a -> R1
ADD  R1, R0     ;;  add R0 to R1
MOV *R1, 6      ;;  const 6 -> address in R1
```

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

- *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 . . . .

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)

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

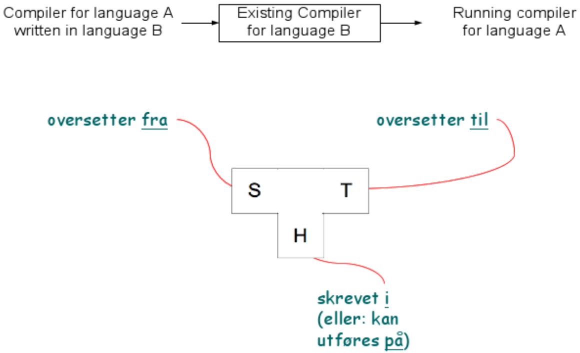
Section

Bootstrapping and cross-compilation

Chapter 1 “Introduction”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Compiling from source to target on host

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



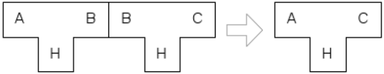
Compiler for language A written in language B → Existing Compiler for language B → Running compiler for language A

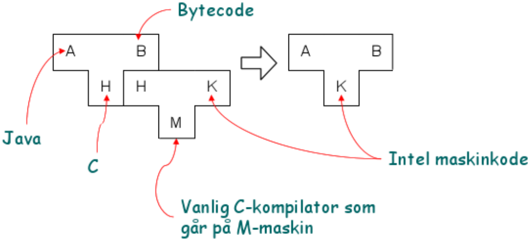
oversetter fra

oversetter til

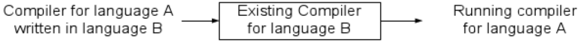
skrevet i (eller: kan utføres på)

Two ways to compose “T-diagrams”





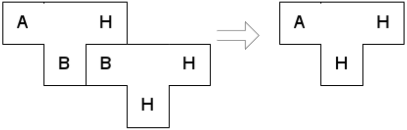
Using an “old” language and its compiler for write a compiler for a “new” one



Compiler for language A written in language B

Existing Compiler for language B

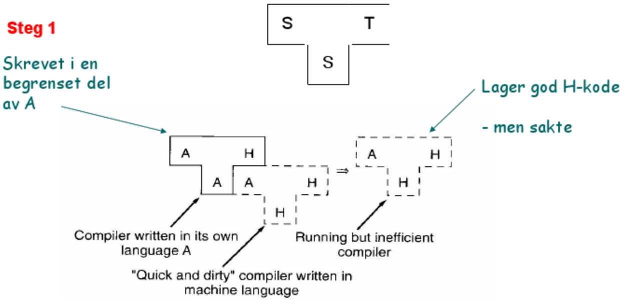
Running compiler for language A



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

Skrevet i en begrenset del av A

Lager god H-kode

- men sakte

Compiler written in its own language A

"Quick and dirty" compiler written in machine language

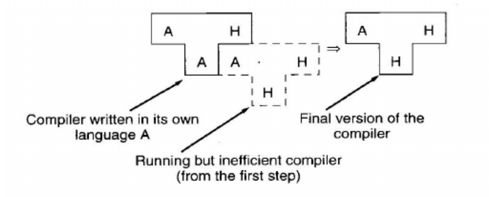
Running but inefficient compiler

Bootstrapping 2

Steg 2

Lager god H-kode

- og fort



Compiler written in its own language A

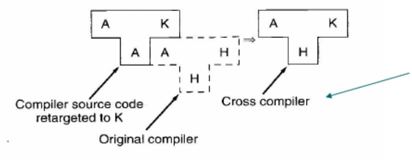
Running but inefficient compiler (from the first step)

Final version of the compiler

Porting & cross compilation

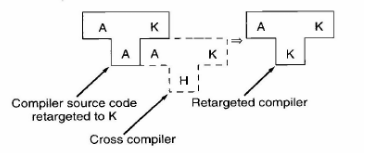
- Har: A kompilator som oversetter til H-maskinkode
- Ønsker: A-kompilator som oversetter til K-maskin kode

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

References I

*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.