Chapter 1

Introduction

Course “Compiler Construction”
Martin Steffen
Spring 2024

Chapter 1

Learning Targets of Chapter “Introduction”.

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

Chapter 1

Outline of Chapter “Introduction”.

Section

Introduction

Chapter 1 “Introduction”
Course “Compiler Construction”
Martin Steffen
Spring 2024

Course info

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.

- `astro-discourse` (some discussion platform)
- (mattermost)

Course material and plan

- see script
- screencasts from 2021 (a corona-semester)
- 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)
- slides: see updates on the net

Obligs and exam

Obligs

- mandatory assignments (= "obligs")
  - O_1 published mid-February, deadline mid-March
  - O_2 published beginning of April, deadline beginning of May
- group work 2 (evtl. 3) people recommended. Please inform us about such planned group collaboration

Exam

We will go for an **oral** exam.

Motivation: What is CC good for?

Good news (?)

Full employment (theorem) for compiler writers.

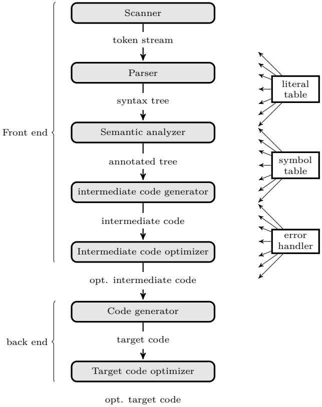
- not everyone builds a full-blown compiler, but
  - fundamental concepts and techniques
  - most, if not basically all, software reads, processes/transforms and outputs "data"
  - ⇒ often techniques central to CC
  - understanding compilers ⇒ deeper understanding of programming language(s)
  - new languages (domain specific, graphical, new language paradigms and constructs. . .)
  - ⇒ CC & principles **never** out-of-fashion

Section

Compiler architecture & phases

Chapter 1 “Introduction”
Course “Compiler Construction”
Martin Steffen
Spring 2024

Architecture of a typical compiler



Pre-processor

- either separate program or integrated into 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)

C-style preprocessor examples

```
#include <filename>
```

Listing: file inclusion

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

Listing: Conditional compilation

C-style preprocessor: macros

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

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

```
 —— 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"

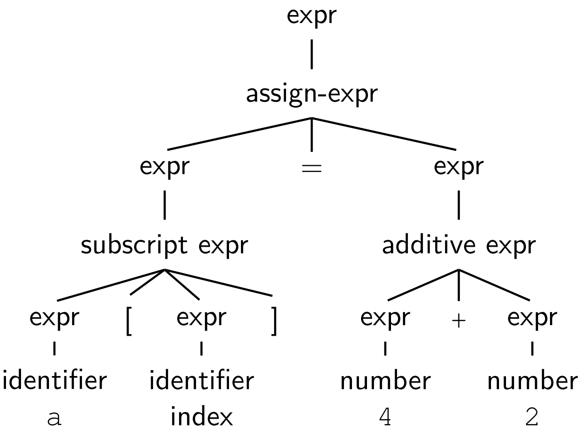
Scanner: illustration

```
␣␣a[index]␣=␣4␣+␣2
```

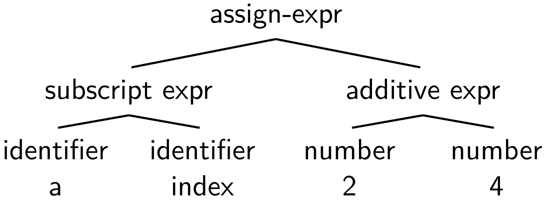
lexeme	token class	value
a	*identifier*	2
[	*left bracket*	
index	*identifier*	21
]	*right bracket*	
=	*assignment*	
4	*number*	4
+	*plus sign*	
2	*number*	2

0	
1	
2	"a"
	⋮
21	"index"
22	
	⋮

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

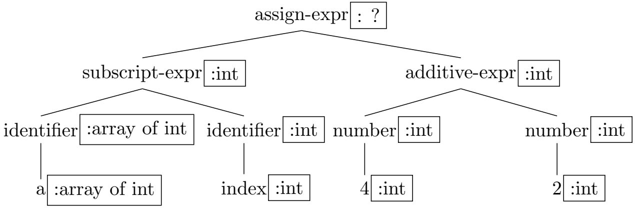


`a[index] = 4 + 2`: abstract syntax tree



(One typical) Result of semantic analysis

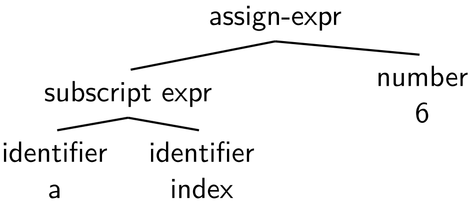
- one standard, general outcome of semantic analysis: "annotated" or "decorated" AST
- additional info (non context-free):
    - *bindings* for declarations
    - (static) *type* information



xs

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

Optimization at source-code level




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

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

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

Code generation & optimization

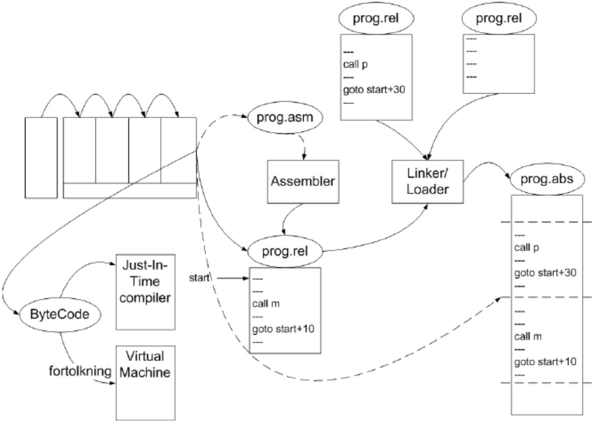
```
MOV R0, index   ;; value of index -> R0
MUL R0, 2       ;; double value of R0
MOV R1, &a      ;; address of a -> R1
ADD R1, R0      ;; add R0 to R1
MOV *R1, 6      ;; const 6 -> address in R1
```

```
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[1], based on a formal description of language and machine
- platform dependent

[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)



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)

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

Section

Bootstrapping and cross-compilation

Chapter 1 “Introduction”
Course “Compiler Construction”
Martin Steffen
Spring 2024

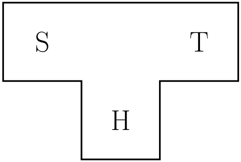
Compiling from source to target on host

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

compilation

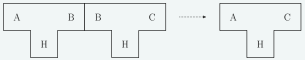
- from **source** language H
- to **target** language T
- on **host** "H" (executed on H or written in H)

E.g.: compiler written for (new) language A written in B

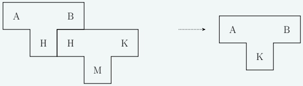


Two ways to compose “T-diagrams”

“transitive”

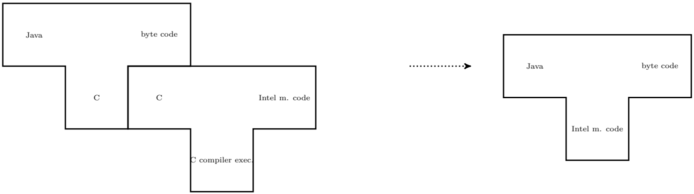


compiling a compiler



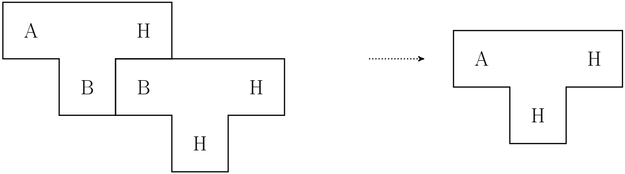
the second one perhaps: compiler written in C (H) translated to a K-executable compiler with the help of an M-executable C-compiler

For example



Java
byte code
C
C
Intel m. code
C compiler exec.
Java
byte code
Intel m. code

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



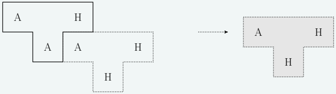
compiler for new language A written in B.

Pulling oneself up on one's own bootstraps

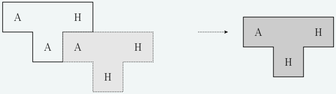
bootstrap (verb, trans.)

to promote or develop . . . with little or no assistance
— Merriam-Webster

step 1

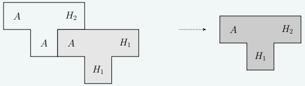


step 2

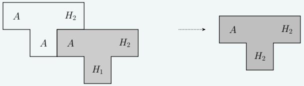


Porting & cross compilation

step 1



step 2



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.