Principles of Compiler Construction

Prof. Wen-jun LI
School of Computer Science and Engineering
lnslwj@mail.sysu.edu.cn

Lecture 1. Introduction

1. Computer Languages
2. Language: Definition and Processing
3. Structure of a Compiler
4. Compiler Construction
5. Course Description

Prologue

- Why learning compiler courses?
  - Excellent combination of theory and practice
  - More insights into programming languages
  - Classical instance of **Programming in the Large** and **Software Engineering**
- But for those students who almost never develop a compiler
  - We focus on: **language is an alternative approach to problem solving.**

Let's Play a Game

- Calculate the following with Windows GUI calculator (mouse only)
  - 5 + (8 – 2)
  - (286 + 8716) / (1973 + 348)
- What's the revelation?
  - How about solving this problem using a language?

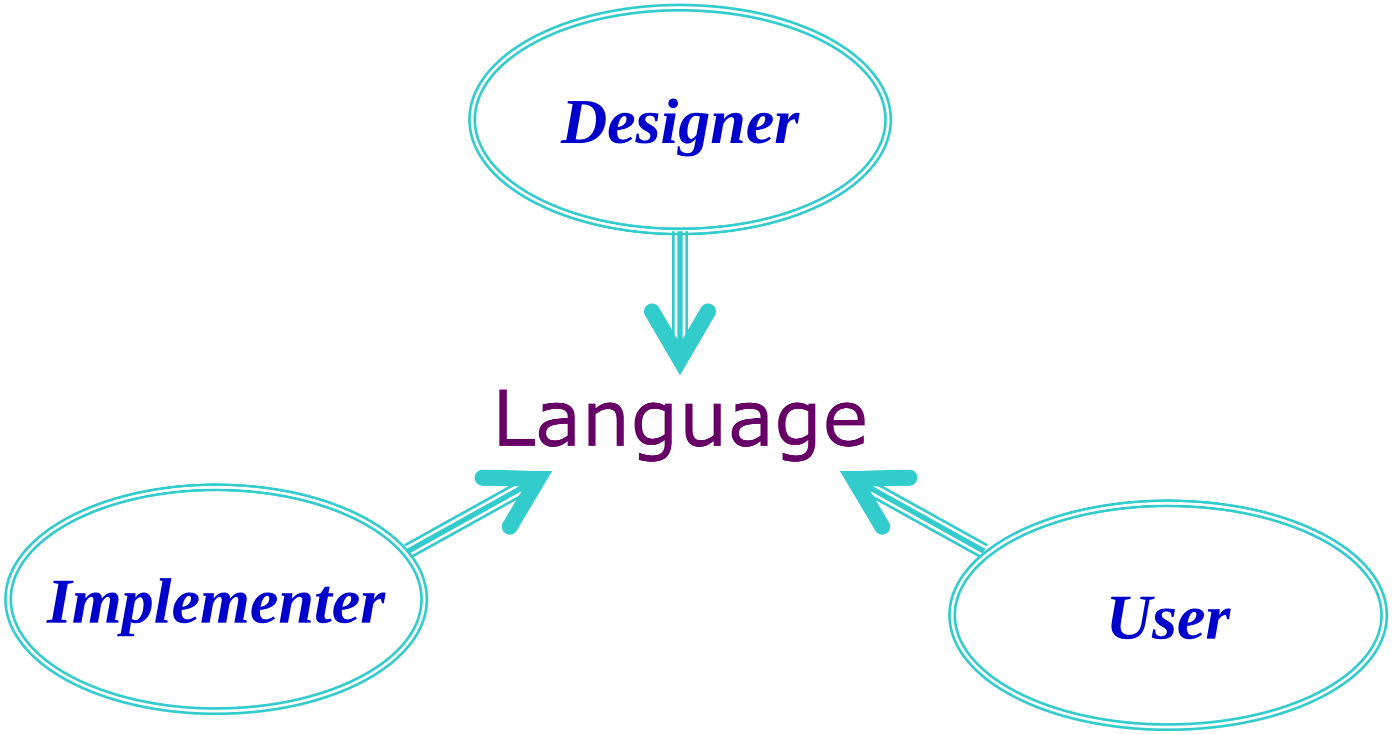
Language Processing

- Language
  - Programming languages
    - Including scripts
  - Domain-Specific Languages (DSL)
    - SQL, HTML, XML, PostScript/PDF/LaTex, etc.
    - Report, workflow, music, recitation, etc.
- Processing
  - Specific to languages
  - Even for programming languages
    - Not only compiling, but also …
    - Beautifier, complexity evaluation, structured editor, reverse engineering, etc.

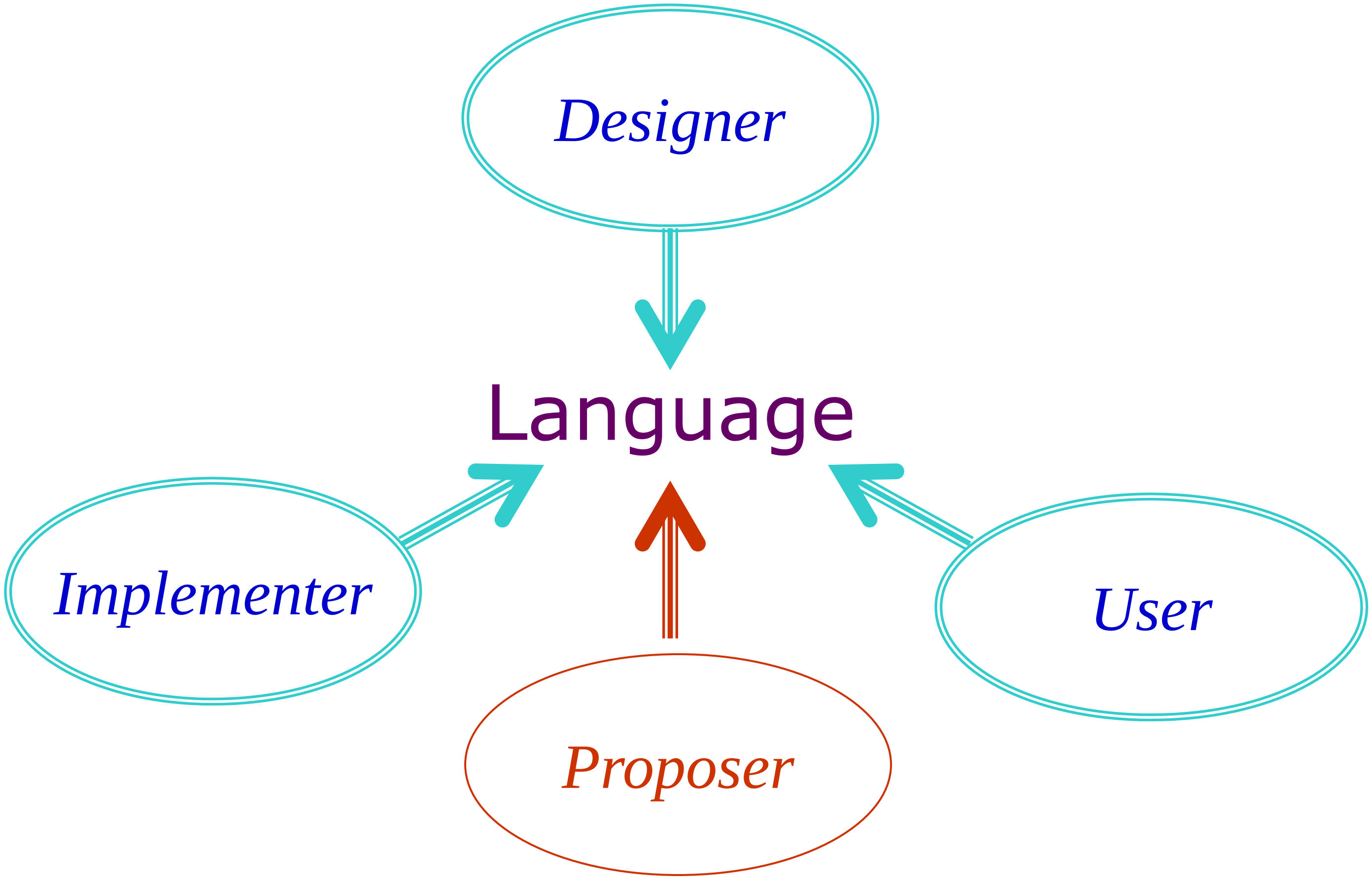
1. Computer Languages

- Language Participants and Courses
- Language Definition: How to Keep Consistency?
- Ambiguity
- Syntax, Semantics and Pragmatics

Participants of a Language

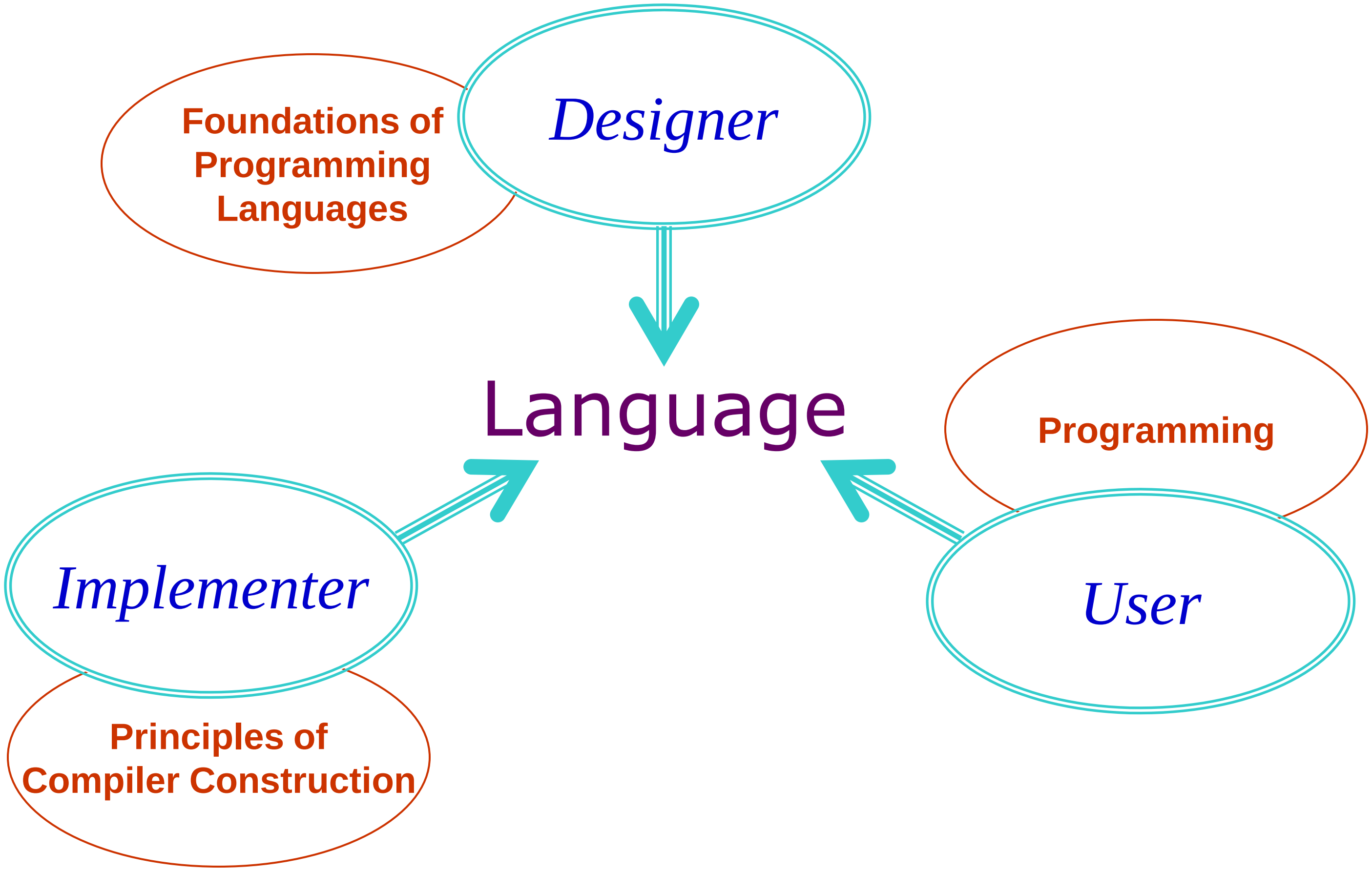


More Participants

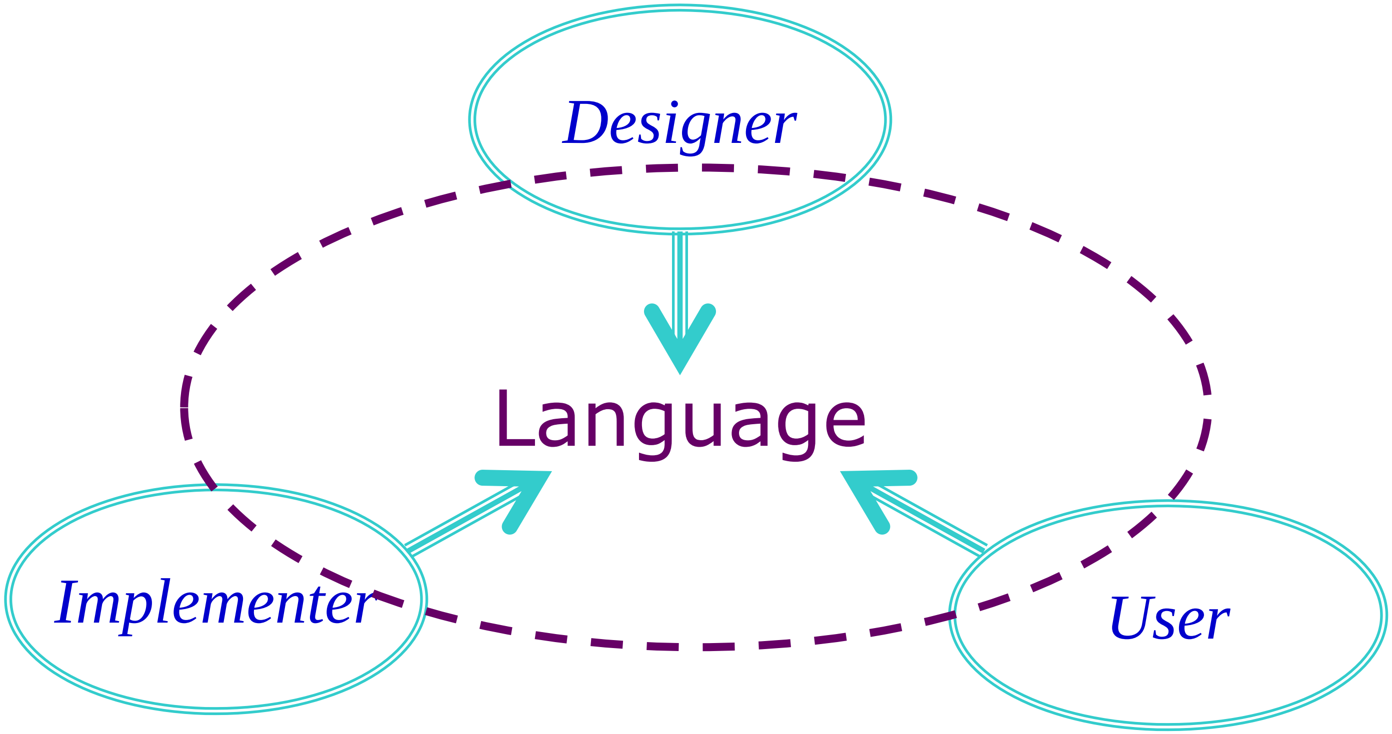


Ada, 1980'

Corresponding Courses



How to Keep Consistency



Natural vs. Formal Languages

- Natural languages lead to **ambiguity**
  - E.g. the **order of parameter evaluation** in programming languages such as Java and C++.
- Fatal weakness of natural languages
  - Ambiguities can be removed by updates or amendments.
  - But it's impossible to support **automatic language processing**.

Wide gap between formal and informal categories

Ambiguity

- 《论语·泰伯篇》
  - 民可使由之不可使知之
  - 民可使由之，不可使知之。
  - 民可使，由之；不可使，知之。
  - 民可使，由之不可，使知之。

Ambiguities in Computer Programs

- Two typical kinds of ambiguities
  - Precedence and associativity in expressions
    - **a + b * c** 加法为左结合，幂运算为右结合
  - Dangling else problem
    - **if x > 0 then if y > 0 then x := 0 else y := 0**
    - **if x > 0 then if y > 0 then x := 0 else y := 0** 最近匹配原则
- Trade-off
  - Unambiguity: *unambiguous rules* only
  - Simplicity: *ambiguous rules* + *additional constraints*

2. Language: Definition and Processing

- Syntax, Semantics and Pragmatics
- BNF and Syntax Graph
- Formal Approach to Syntax
- Formal Semantics
- Type System

Syntax, Semantics and Pragmatics

- Syntax 语法
  - The phrase structure of symbols.
  - A program must be **well-formed**.
- Semantics 语义
  - The meaning of programs, i.e. the connection between symbols and the meanings they denote.
- Pragmatics
  - The ways in which context contributes to meaning.
  - Not quite clear nowadays.

Syntax + Semantics + Pragmatics

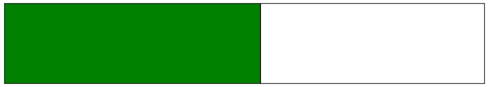
=

Semiotics

Abstraction of a Language

- **Thinking in abstraction**
  - Abstract the most important features that we take under consideration.
  - Ignore other subordinate details.
    - E.g. pronunciation of the language
- Do you believe it?
  - {a, ab, abb, abbb, ...} is a language.
  - $\varnothing$ is also a language.

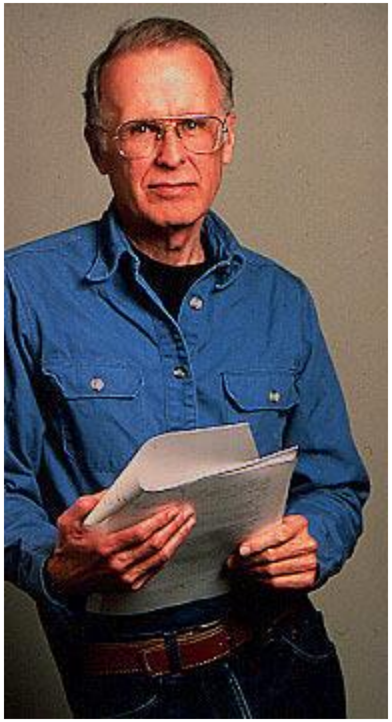
Achievements and Opportunities

Domain	Period	Achievements
Syntax	40's - 60's	
Semantics	70's - 90's	
Pragmatics	--	

Syntax Definition

1. Character set
   - Properties: **finite set**; order.
   - Examples:
     - Ada and C++: **ASCII**
     - APL: **EBCDIC**
     - Java: **?**
2. Syntax rules
   - **BNF**
   - **Syntax Graph**

BNF: Backus-Naur Form



John Backus (IBM)

(Dec. 3, 1924 - March 17, 2007)

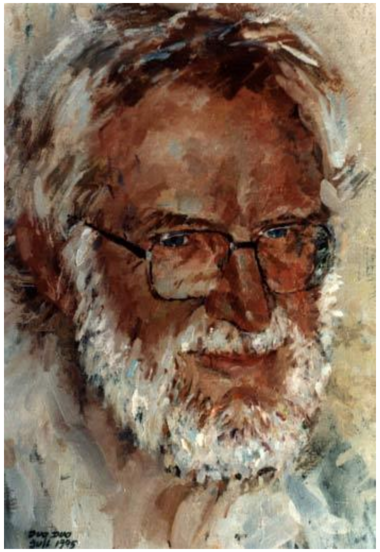
Father of Fortran

ACM Turing Award, 1977

"Much of my work has come from being lazy.

I didn't like writing programs ..."

-- IBM Think Magazine, 1979



Prof. Peter Naur

University of Copenhagen

The 17-page Algol 60 Report

ACM Turing Award, 2005

Examples of BNF

```
identifier   ::=   letter { letter | digit }  {}循环
letter       ::=   A | B | … | Z | a | b | … | z
digit        ::=   0 | 1 | … | 9
```

```
integer      ::=   [ symbol ] unsigned  []可选项
unsigned     ::=   digit { digit }
symbol       ::=   + | -
digit        ::=   0 | 1 | … | 9
```

```
for_stmt     ::=   for loop_var := init direction final
                   do stmt
loop_var     ::=   int_var
int_var      ::=   var_id
var_id       ::=   identifier
init         ::=   expr
final        ::=   expr
direction    ::=   to | downto
```

More Examples of BNF

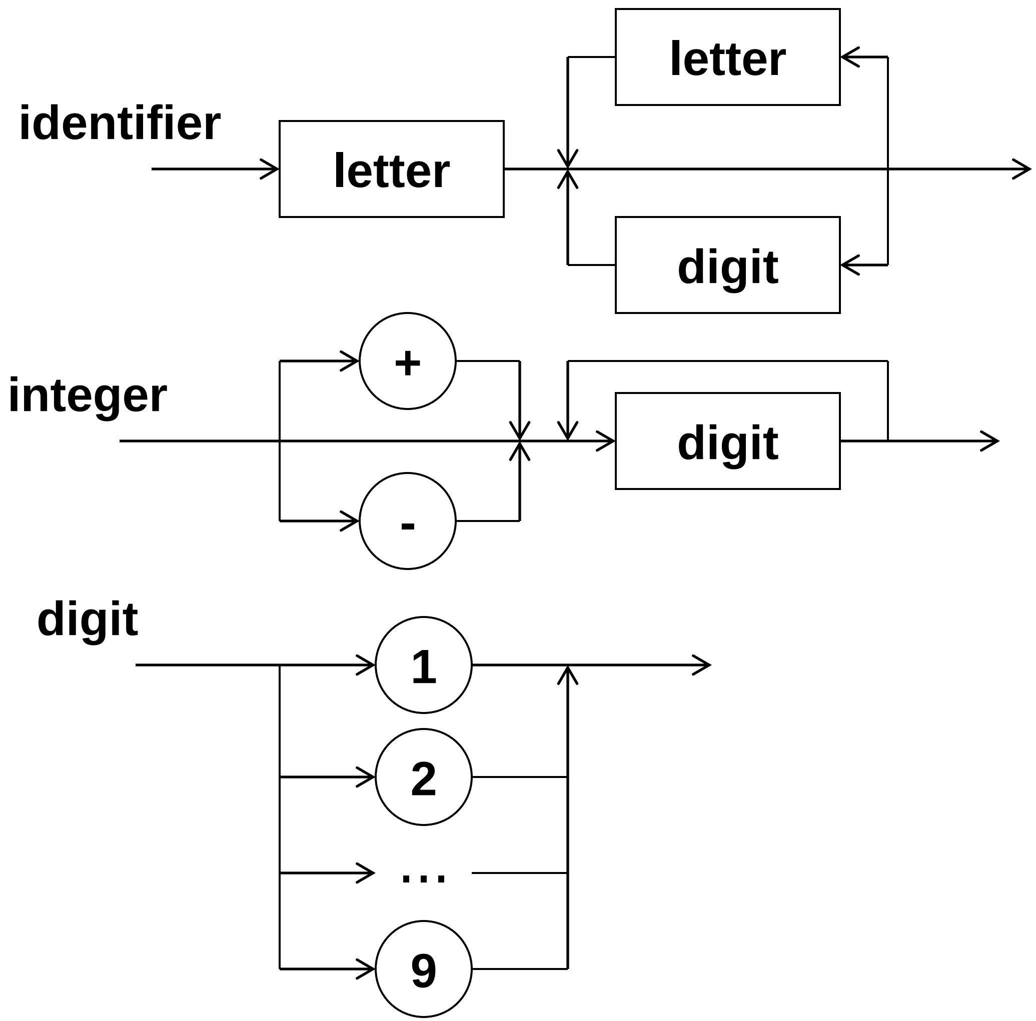
Rules:

```
bexpr       ::= bexpr or bterm | bterm
bterm       ::= bterm and bfactor | bfactor
bfactor     ::= not bfactor | ( bexpr ) | true | false
```

Instances:

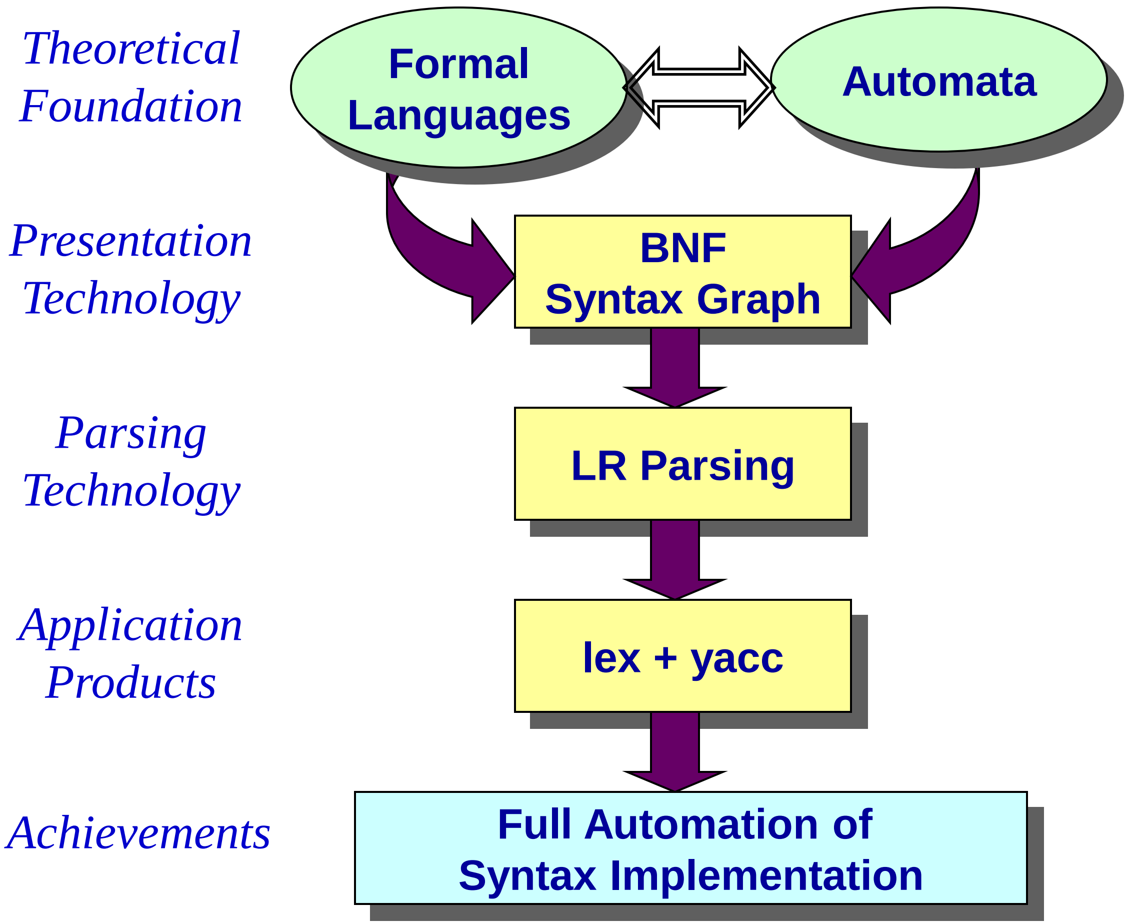
```
true and false or (not true)
false or true and not false
true and false and (not false and (true or false))
```

Syntax Graph



Prof. **Niklaus Wirth**
Father of Pascal
ACM Turing Award, 1984

Formal Approach to Syntax



Achievements

1. **A de facto standard** to define syntax of a new language.
2. Analyze **syntactic properties** of a language.
   - Is it (or the syntax defined) ambiguous?
   - Is the grammar LL(k) or LR(k)?
   - ...
3. **Automation** of syntax processing.
   - For automation, the input of the processor must be formal definitions.
     - Lexical rules as the input of **lex**
     - Syntax rules as the input of **yacc**

Difficulties in Formal Semantics

- Difficulties in nature
  - Must be based on formal syntax.
  - More complex than syntax.
  - More mathematical foundations required.
- Artificial difficulties
  - Different viewpoints lead to different approaches.
  - Notations: lack of standardization.

Mathematical Difficulties

- Mathematical foundations required
  - **Discrete mathematics**
    - *Set theory*, *mathematical logic*, *abstract algebra*, *category theory*, *type theory*, etc.
  - **Computational models**
    - *λ-calculus*, *formal languages and automata*, *process algebra*, *Petri nets*, etc.
  - **Proprietary theories**
    - *Domain theory*, *power domain theory*, *Hoare logic* and other logics, etc.

Approaches to Formal Semantics

- Different viewpoints lead to various approaches to formal semantics:
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics
  - Algebraic semantics
    - Semantics of ADTs based on category theory

Levels of Abstraction

Expectation of Formal Semantics

- A standard to define formal semantics of a language.
- Formal analysis of semantic properties.
  - Is the language strong typing?
  - Does the language support block structures?
  - Is the language single threading? ...
- Automation of the semantic processing of language processors.
  - Formal definition of semantics as the input.

Type System

- A lightweight formal semantics
  - Maybe the most successful application of formal semantics in practice.
- Type safeness
  - Compiler can discover all type related errors **statically**.
    - C++: `float x = 3.14;` (correct)
    - Java: `float x = 3.14;` (erroneous)
    - **Explicit** type conversion in Java

```
float x = (float) 3.14;
```

Important Terminology

- Static vs. dynamic
  - I.e. compile-time vs. run-time
- Explicitly vs. implicitly
  - I.e. manually vs. automatically
- Logical vs. physical
  - Two levels of abstraction
- Safeness vs. security
  - Type safeness
  - Thread safeness

3. Structure of a Compiler

- Basic Concepts of Translation
- Phases of a Compiler
- A Compiling Example
- Software Architecture

How Does a Compiler Work?

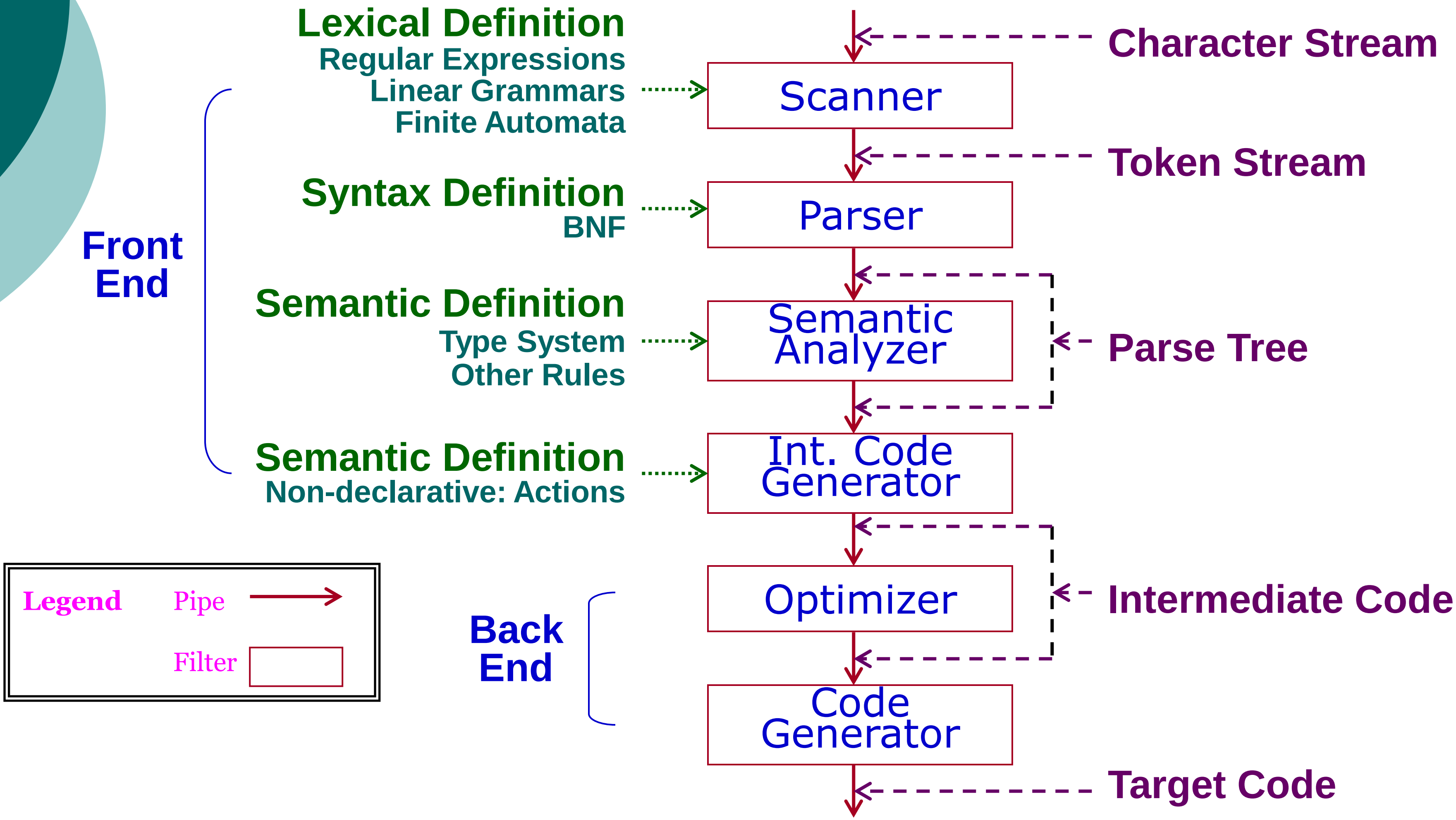
Basic Concepts

- Programming languages
  - High-level
    - SP, OOP, functional, logical, concurrent, etc.
  - Low-level
    - Assembly, machine
  - **Discussion**
    - What's the essential differences between high-level and low-level programming languages?

Basic Concepts (cont')

- Translation
  - **Discussion**
    - Compiler vs. (macro) assembler
    - Compiler vs. interpreter (advantages and disadvantages)
    - What is the execution model for Java and Microsoft .Net? And why?

Structure of a Compiler



Architecture Design

- Analysis vs. synthesis
  - Structure analysis
    - Lexical analysis and syntax analysis
  - Semantic analysis
- Front end vs. back end
  - Standard intermediate representation (IR/IL) supports substitution.
    - **GCC**: **G**NU (/'gnu:/) **C**ompiler **C**ollection.
- Error recovery and symbol table management

Cousins of a Compiler

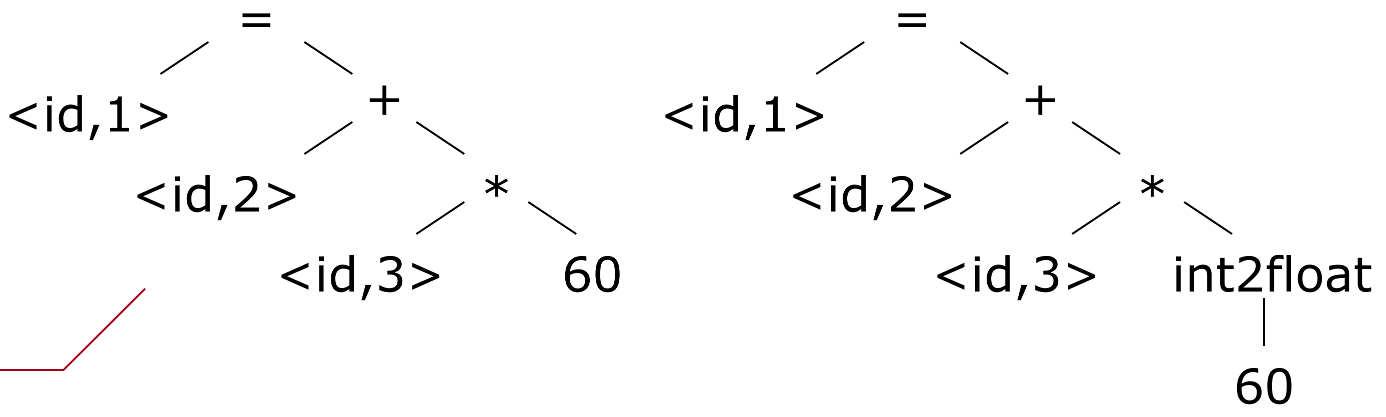
- Preprocessor
  - C/C++: `#include <...>`
- Assembler
- Linker
- Loader
- Debugger
- IDE: Integrated Development Environment
  - Editor + Compiler + Linker + Debugger + ...

A Compiling Example

- Source
  - position = initial + rate * 60
- Scanner
  - <id,1> <=> <id,2> <+> <id,3> <*> <60>

token stream

- Parser and semantic analyzer



parse tree

A Compiling Example (cont')

- Intermediate code generator
  - t1 = int2float(60)
  - t2 = id3 * t1
  - t3 = id2 + t2
  - id1 = t3
- Code optimizer
  - t1 = id3 * 60.0
  - id1 = id2 + t1
- Code generator
  - LDF R2, id3
  - MULF R2, R2, #60.0
  - LDF R1, id2
  - ADDF R1, R1, R2
  - STF id1, R1

Software Architecture

- What is Software Architecture?
- Typical SA styles
  - Layered (3-tier, n-tier)
  - Pipes and filters
  - Event-driven
  - Client-server
  - etc.
- What benefits from SA?
  - Passes
  - Logical vs. physical

4. Compiler Construction

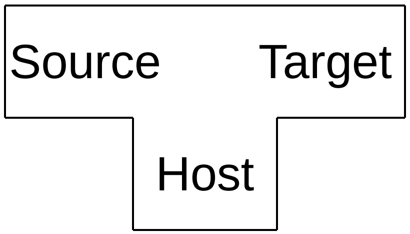
- Requirements for compiler design
- T-diagram
- Bootstrapping and porting
- Compiler generators
  - Scanner generators
  - Parser generators
  - Other generators

Requirements for Compiler Design

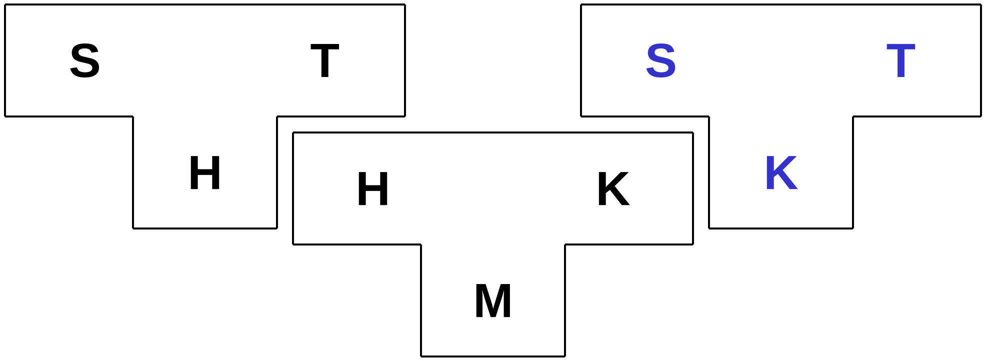
- Efficiency of a compiler
  - Time vs. space
- Efficiency of the target code
  - Time vs. space
- Ability to error recovery
- High reliability
- …

T-Diagram: A Formal Notation

- T-diagram

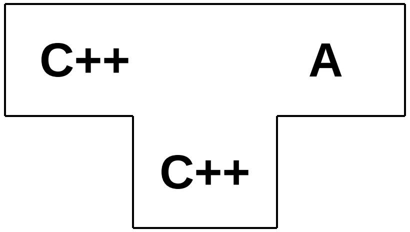


- T-diagram combination



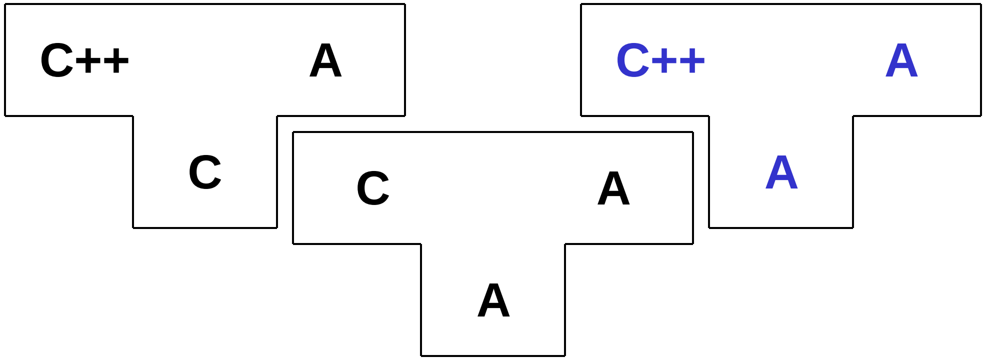
Self Compiling

- Write a compiler in the same language



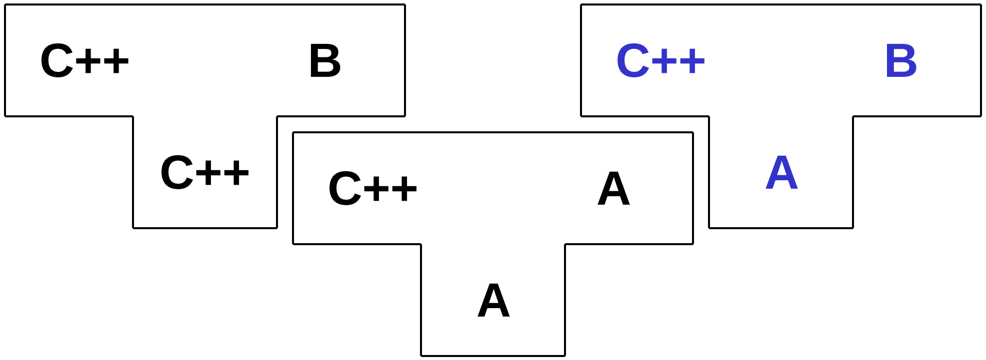
Bootstrapping

- The bootstrapping process may be repeated

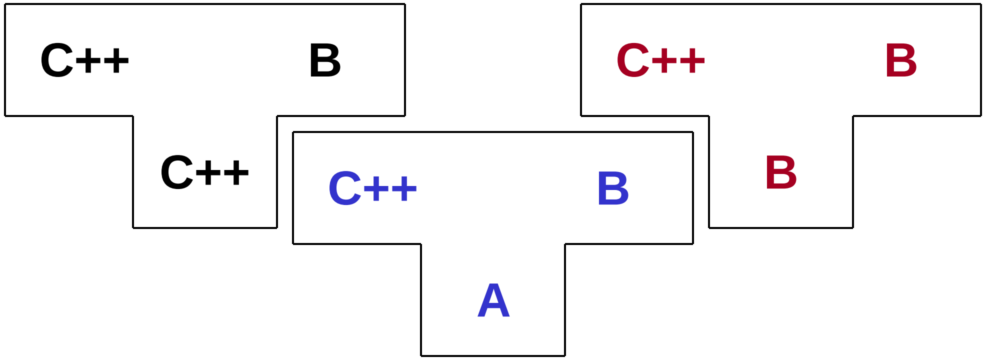


Porting

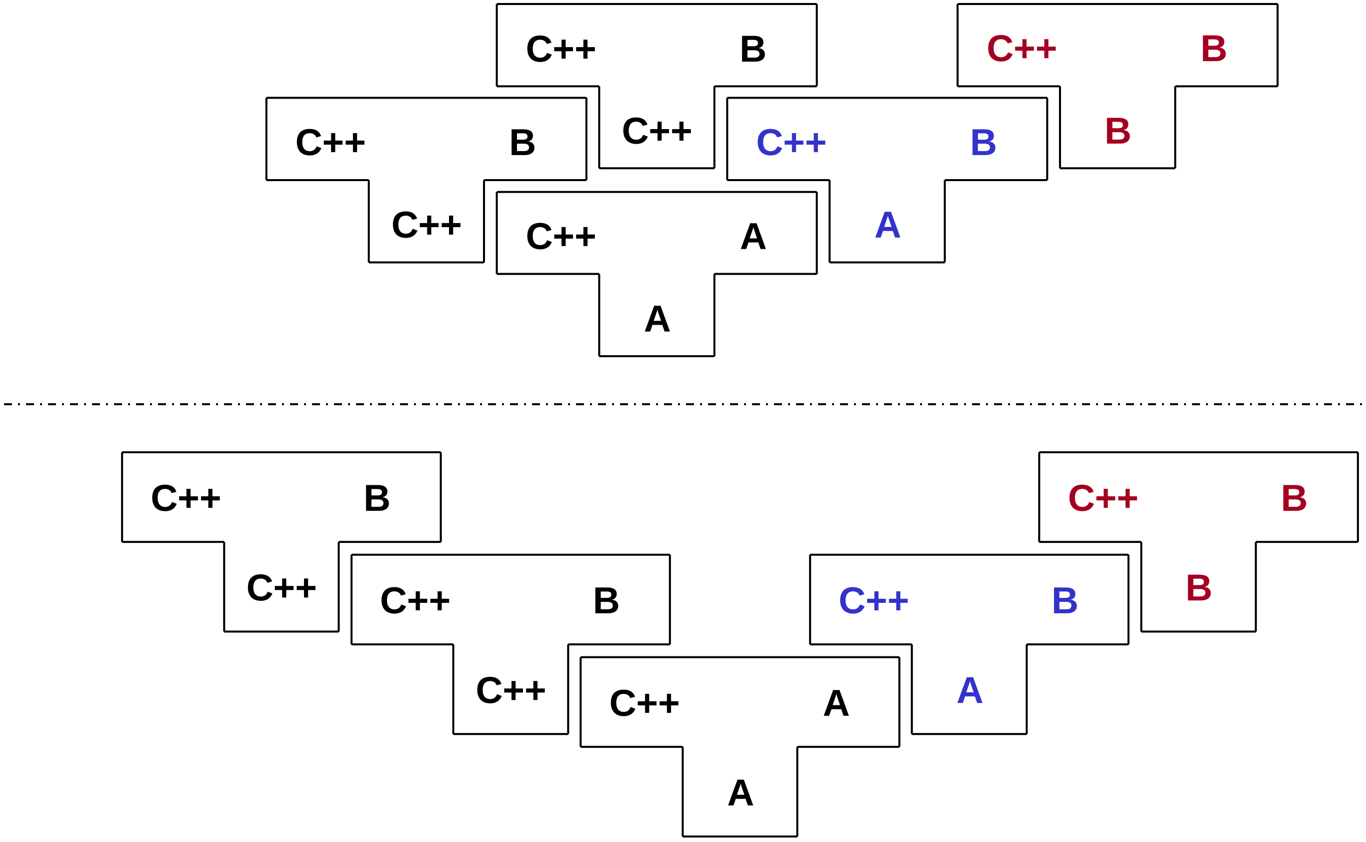
- Cross-compiler (2 stages)
  - Stage 1



  - Stage 2



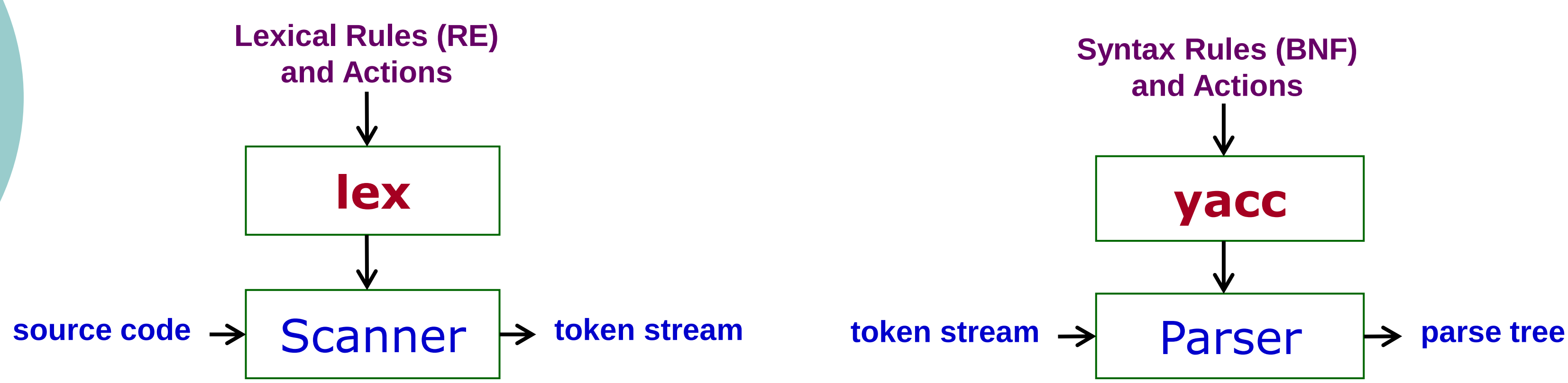
Other Notations of Combination

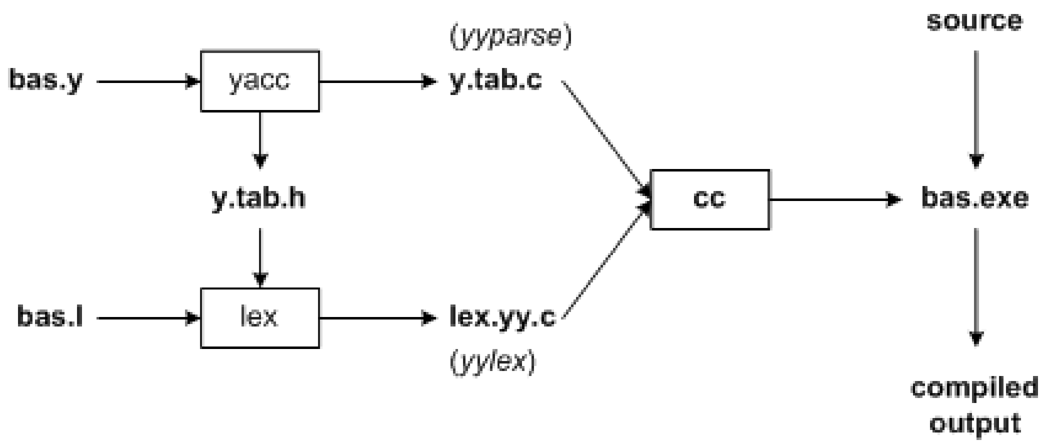


Compiler Construction Tools

- Formal definitions lead to automatic tools
  - **lex**: scanner generator
  - **yacc**: parser generator
    (**Y**et **A**nother **C**ompiler **C**ompiler)
- Popular tools
  - C/C++: GNU **Flex** and GNU **Bison**
  - Java: **JFlex** and **JavaCUP**, **ANTLR**

lex and yacc

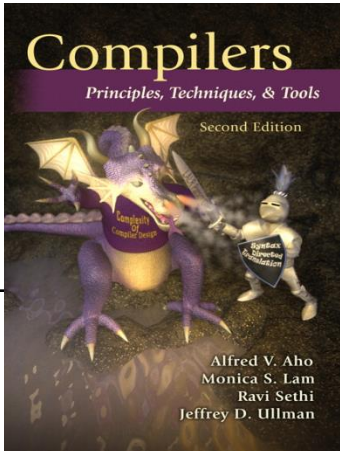

Lexical Rules (RE)
and Actions
lex
source code
Scanner
token stream
Syntax Rules (BNF)
and Actions
yacc
token stream
Parser
parse tree


bas.y
yacc
(yyparse)
y.tab.c
y.tab.h
bas.l
lex
lex.yy.c
(yylex)
cc
source
bas.exe
compiled
output

5. Course Description

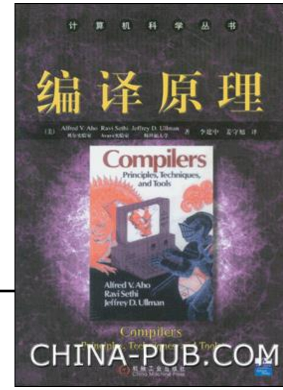
- Textbook
- References

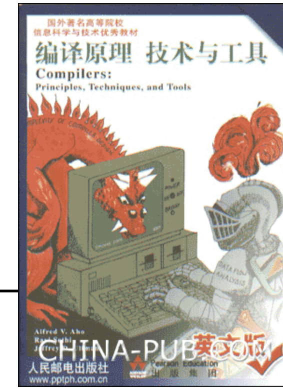
Textbook

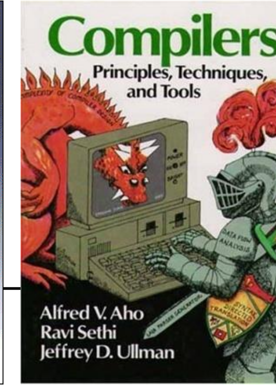


- The Dragon Book, 2nd Ed.
  - A. Aho, M. Lam, R. Sethi and J. Ullman. **Compilers: Principles, Techniques, and Tools, 2nd Ed.** Addison-Wesley, 2006, ISBN 0-321-48681-1
- We only use about 600 pages:
  - Chapter 1 – 8
  - Appendix A

References

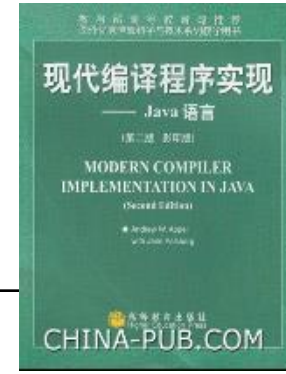


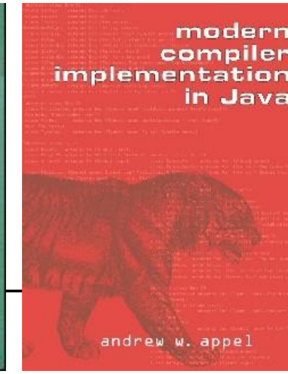




- The Dragon Book, 1st Ed.
  - A. Aho, R. Sethi and J. Ullman. **Compilers: Principles, Techniques, and Tools.** Addison-Wesley, 1988, ISBN 0-201-10088-6
  - 李建中、姜守旭译. **编译原理.** 北京: 机械工业出版社, 计算机科学丛书中文系列, 2003, ISBN 7-111-12349-2
- The ancestor of many textbooks compiled in Chinese.

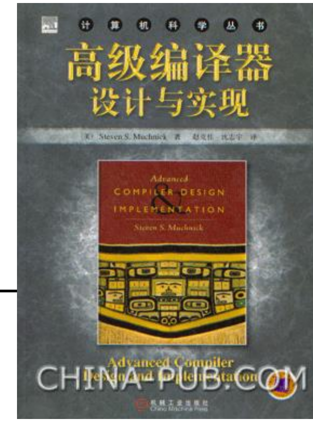
References

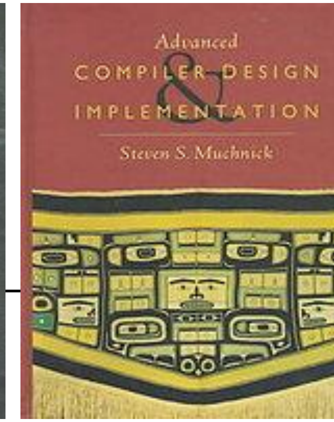




- The Tiger Book
  - A. Appel.
    Modern Compiler Implementation in Java.
    Cambridge University Press, 2002, ISBN 0-521-82060-X
  - 陈明等译.
    现代编译器的Java实现（第2版）.
    北京: 电子工业出版社, 国外计算机科学教材系列, 2004, ISBN 7-121-00270-1
- A book worth buying and reading.

References





- The Whale Book
  - S. Muchnick.
    Advanced Compiler Design and Implementation.
    Morgan Kaufmann, 1997, ISBN 1-558-60320-4
  - No Chinese version available. But there is a copyright transferred English version in Mainland China (published by CMP).
- Focus on compiler optimization.
- Not suitable for beginners.

References in Chinese

- 陈火旺、刘春林、谭庆平、赵克佳、刘越.
  程序设计语言编译原理（第3版）.
  国防工业出版社, 2000, ISBN 7-118-02207-1
- 杜淑敏、王永宁.
  编译程序设计原理.
  北京大学出版社, 1990, ISBN 7-301-01210-1
- 张素琴、吕映芝、蒋维杜、戴桂兰.
  编译原理（第2版）.
  清华大学出版社, 清华大学计算机系列教材, 2005, ISBN 7-302-08979-5

Exercise 1.1

- Imagine an artificial computer language, which can be utilized to solve a practical problem, i.e. the application of the language.
  - Tips 1. Language is an alternative approach to problem solving.
  - Tips 2. First find a proper problem, then design a language to solve the problem.
- Give an example of a complete piece written in the proposed language.
- Discuss how to define the new language and try your approach.
- Describe the process of changing the thinking of your language to a reality, i.e. how to make the artificial language usable.

Exercise 1.2

- Draw a T-diagram with two stages of bootstrappings.
  - Given a new programming language L++, we firstly implement L, a small subset of L++.
  - Then we use L to implement L+, a subset of L++ and a superset of L.
  - Finally, L++ is implemented using L+.

Further Reading

- Dragon Book, 2nd Edition (DBv2)
  - Comprehensive Reading: Section 1.1, 1.2, 1.6
  - Skip Reading: Section 1.3, 1.4, 1.5
- On domain-specific languages
  - http://en.wikipedia.org/wiki/Domain-specific_programming_language

Enjoy the Course!