Introduction to Compiler Construction

ALSU Textbook Chapter 1.1–1.5

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

`http://www.iis.sinica.edu.tw/~tshsu`

What is a compiler?

- Definitions:
  - a recognizer ;
  - a translator .

$$\boxed{\text{source program}} \Rightarrow \text{compiler} \Rightarrow \boxed{\text{target program}}$$

  - Source and target must be equivalent!
- Compiler writing spans:
  - programming languages;
  - machine architecture;
  - language theory;
  - algorithms and data structures;
  - software engineering.
- History:
  - 1950: the first FORTRAN compiler took 18 man-years;
  - now: using software tools, can be done in a few months as a student's project.

Applications

- **High-level programming language compilers.**
- **Optimizations for computer architectures.**
- **Design of new computer architectures.**
- **Translator: from one format to another.**
  - query interpreter
  - text formatter
  - silicon compiler
  - infix notation $\rightarrow$ postfix notation:

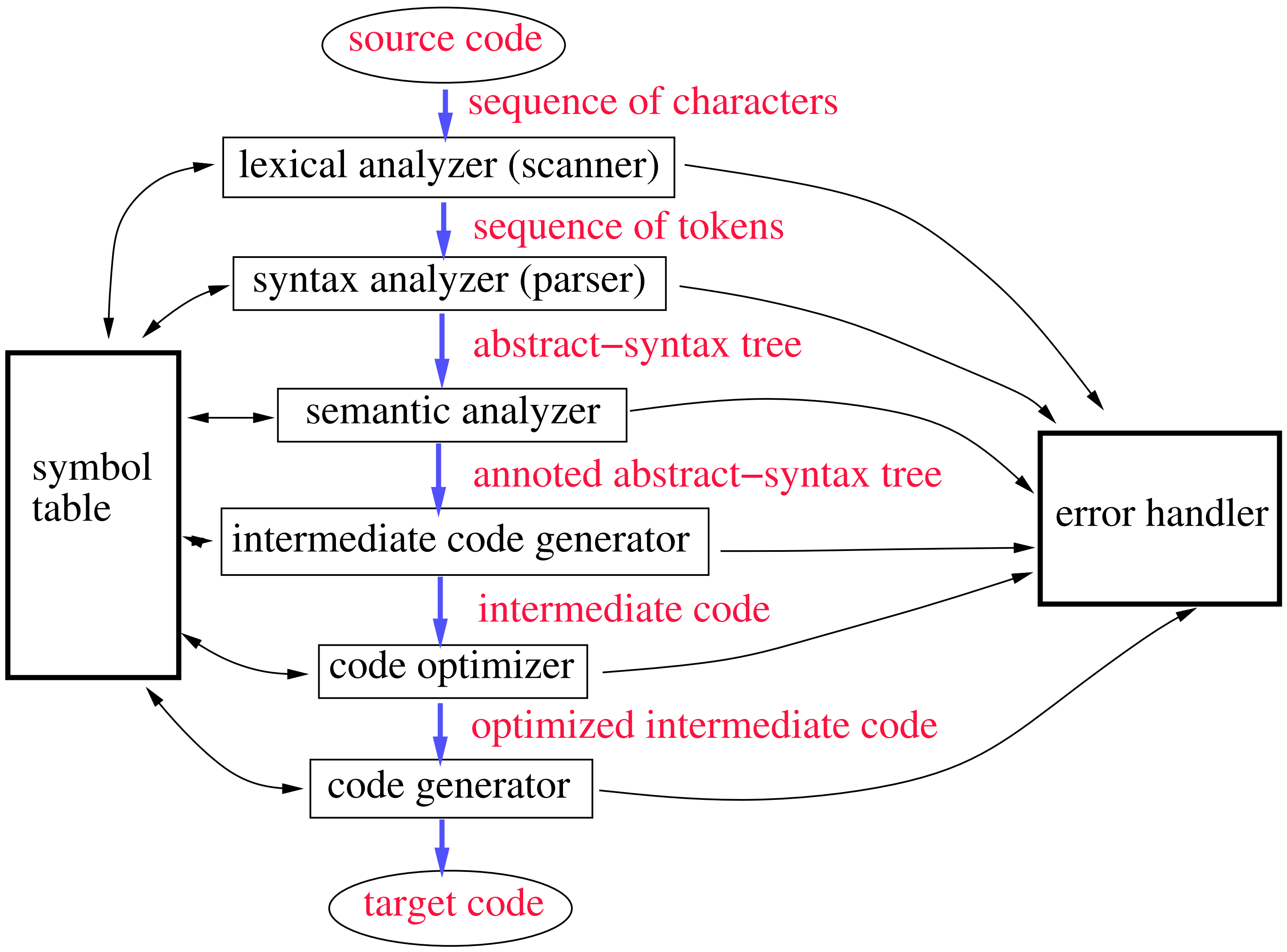
$$\boxed{3+5-6*6} \Rightarrow \boxed{3 \quad 5 \quad + \quad 6 \quad 6 \quad * \quad -}$$

  - pretty printers
  - $\cdots$
- **Software productivity tools.**

Relations with computational theory

- a set of grammar rules $\equiv$ the definition of a particular machine.
  - also equivalent to a set of languages recognized by this machine.
- a type of machines: a family of machines with a given set of operations, or capabilities;
- power of a type of machines
  $\equiv$ the set of languages that can be recognized by this type of machines.

Flow chart of a typical compiler



Scanner

- **Actions:**
  - **Reads characters from the source program;**
  - **Groups characters into lexemes, i.e., sequences of characters that “go together”, following a given pattern;**
  - **Each lexeme corresponds to a token.**
    - ▷ *the scanner returns the next token, plus maybe some additional information, to the parser;*
  - **The scanner may also discover lexical errors, i.e., erroneous characters.**
- **The definitions of what a lexeme, token or bad character is depend on the definition of the source language.**

Scanner example for C

- **Lexeme: C sentence**

$$\text{L1: } x = y2 + 12;$$

(Lexeme)	**L1**	**:**	**x**	**=**	**y2**	**+**	**12**	**;**
(Token)	**ID**	**COLON**	**ID**	**ASSIGN**	**ID**	**PLUS**	**INT**	**SEMI-COL**

- **Arbitrary number of blanks between lexemes.**
- **Erroneous sequence of characters, that are not parts of comments, for the C language:**
  - **control characters**
  - **@**
  - **2abc**

Parser

- **Actions:**
  - Group tokens into grammatical phrases, to discover the underlying structure of the source
  - Find syntax errors, e.g., the following C source line:

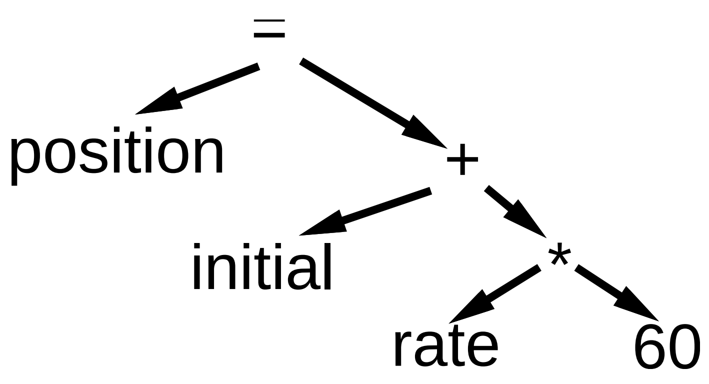
(Lexeme)	index	=	12	*	;
(Token)	ID	ASSIGN	INT	TIMES	SEMI-COL

  Every token is legal, but the sequence is erroneous!

- **May find some static semantic errors, e.g., use of undeclared variables or multiple declared variables.**
- **May generate code, or build some intermediate representation of the source program, such as an abstract-syntax tree.**

Parser example for C

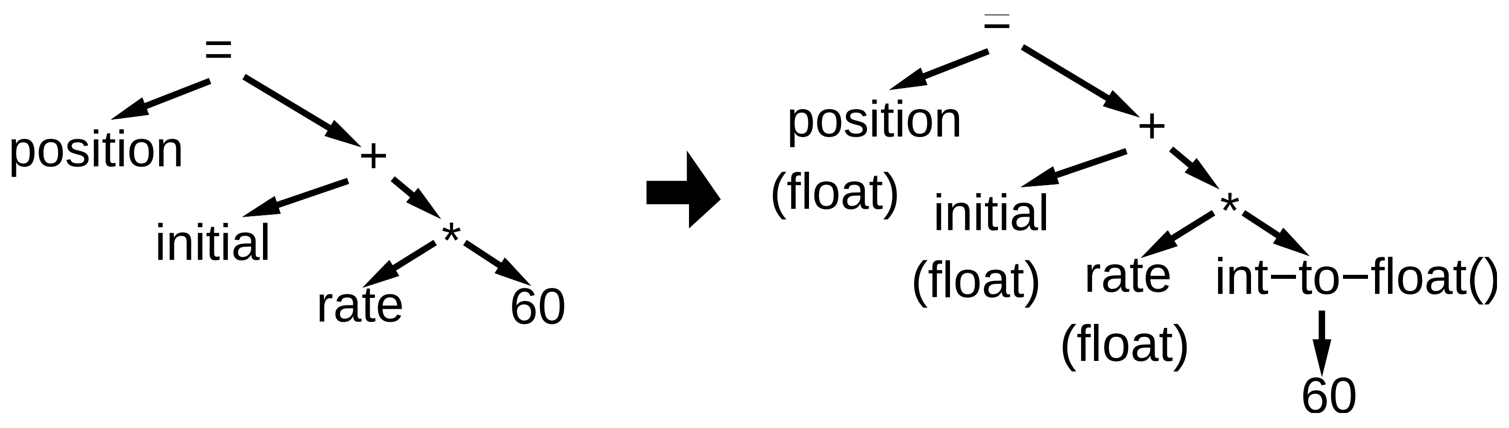
- **Source code:** $position = initial + rate * 60;$
- **Abstract-syntax tree:**



  - **interior nodes of the tree are OPERATORS;**
  - **a node's children are its OPERANDS;**
  - **each subtree forms a logical unit .**
  - **the subtree with $*$ at its root shows that $*$ has higher precedence than $+$, the operation "$rate * 60$" must be performed as a unit, not "$initial + rate$".**

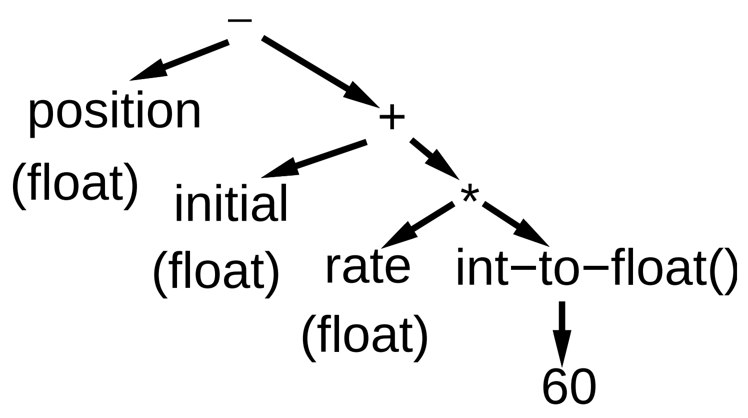
Semantic analyzer

- **Actions:**
  - **Check for more static semantic errors, e.g., type errors .**
  - **May annotate and/or change the abstract syntax tree.**



Intermediate code generator

- **Actions: translate from abstract-syntax trees to intermediate codes.**
- **One choice for intermediate code is 3-address code:**
  - **Each statement contains**
    - ▷ *at most 3 operands;*
    - ▷ *in addition to ":=", i.e., assignment, at most one operator.*
  - **An"easy" and "universal" format that can be translated into most assembly languages.**
- **Example:**



```
temp1 := int-to-float(60)
temp2 := rate * temp1
temp3 := initial + temp2
position := temp3
```

Optimizer

- **Improve the efficiency of intermediate code.**
- **Goal may be to make code run faster, and/or to use least number of registers ···**
- **Example:**

```
temp1 := int-to-float(60)
temp2 := rate * temp1
temp3 := initial + temp2
position := temp3
```

$\Rightarrow$

```
temp2 := rate * 60.0
position := initial + temp2
```

- **Current trends:**
  - to obtain smaller, but maybe slower, equivalent code for embedded systems;
  - to reduce power consumption;
  - to enable parallelism;
  - ···

Code generation

- **A compiler may generate**
  - **pure machine codes (machine dependent assembly language) directly, which is rare now;**
  - **virtual machine code.**
- **Example:**
  - **PASCAL** $\rightarrow$ $\boxed{\textbf{compiler}}$ $\rightarrow$ **P-code** $\rightarrow$ $\boxed{\textbf{interpreter}}$ $\rightarrow$ **execution**
  - **Speed is roughly 4 times slower than running directly generated machine codes.**
- **Advantages:**
  - **simplify the job of a compiler;**
  - **decrease the size of the generated code: 1/3 for P-code;**
  - **can be run easily on a variety of platforms**
    - ▷ *P-machine is an ideal general machine whose interpreter can be written easily;*
    - ▷ *divide and conquer;*
    - ▷ *recent example: JAVA and Byte-code.*

Code generation example

```
temp2 := rate * 60.0
position := initial + temp2
```

$\Longrightarrow$

LOADF	rate, R_1
MULF	#60.0, R_1
LOADF	initial, R_2
ADDF	R_2, R_1
STOREF	R_1, position

Practical considerations (1/2)

- **Preprocessing phase:**
  - **macro substitution:**
    - ▷ *#define MAXC 10*
  - **rational preprocessing: add new features for old languages.**
    - ▷ *BASIC*
    - ▷ $C \rightarrow C++$
  - **compiler directives:**
    - ▷ *#include <stdio.h>*
  - **non-standard language extensions.**
    - ▷ *adding parallel primitives*

Practical considerations (2/2)

- **Passes of compiling**
  - First pass reads the text file once.
  - May need to read the text one more time for any **forward addressed** objects, i.e., anything that is used before its declaration.
  - Example: C language

```
goto error_handling;
...
error_handling:
...
```

Reduce number of passes

- **Each pass takes I/O time.**
- **Back-patching : leave a blank slot for missing information, and fill in the empty slot when the information becomes available.**
- **Example: C language**
  when a label is used
  - **if it is not defined before, save a trace into the to-be-processed table**
    - ▷ *label_name corresponds to LABEL_TABLE[i]*
  - **code generated: GOTO LABEL_TABLE[i]**

  when a label is defined
  - **check known labels for redefined labels**
  - **if it is not used before, save a trace into the to-be-processed table**
  - **if it is used before, then find its trace and fill the current address into the trace**
- **Time and space trade-off !**