Chapter1. Introduction

2019年11月26日 10:21

NOTE TAKING AREA

References

Textbook:

- Compilers: Principles, Techniques, & Tools

References:

- 编译原理实践与指导教程

Grading

- Programming assignments: 40%
- Final exam: 35%
- Written assignments: 20%
- Lecture attendance: 5%

Bonus: optional assignment requirements, and optional task in projects.

Course content

- Introduction to compilers
- Lexical analysis*
- Syntax analysis*
- Syntax-directed translation
- Intermediate-code generation*
- Run-time environments
- Code generation
- Machine-independent optimizations

Contents with star mark means it's hard.

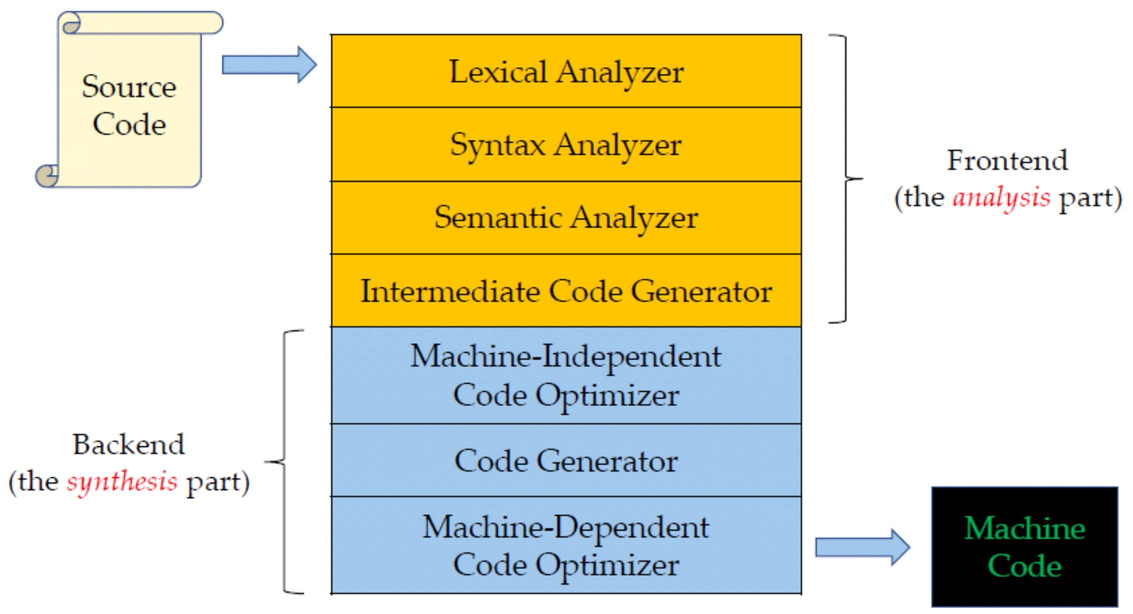
Programming languages

Low-level: directly understandable by a computer.
High-level: understandable by human beings, need a translator to be understood by a computer.
The first electronic computer: ENIAC (1946), **machine language**.

Assembly language (1950s)
Mnemonic names: for machine instructions.
Macro instructions: for frequently used sequences of machine instructions.
Still low-level and machine dependent.

High-level language
High-level language in second half of 1950s: Fortran (scientific computation), Cobol (business data processing), Lisp (symbolic computation).
Fortran: Formula translation, first high-level language (1957).
    Huge impact, modern compilers preserve the outline of Fortran I.

Compiler structure and phases



Source Code
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Machine-Independent Code Optimizer
Code Generator
Machine-Dependent Code Optimizer
Machine Code
Frontend (the *analysis* part)
Backend (the *synthesis* part)

Frontend of compiler:
- Breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- Uses the grammatical structure to create an <u>intermediate representation (IR)</u> of the source program.
- Collect information about the source program and stores it in <u>symbol table</u> (pass to backend through IR).

Backend of compiler:
- Constructs the target program from the IR and the information in the symbol table.
- Performs code optimizations during the process.

Lexical analysis (scanning)

$$\text{Character stream} \longrightarrow \boxed{\text{Lexical Analyzer}} \longrightarrow \text{Token stream}$$

Breaks down the source code into a sequence of <u>lexemes</u> or words.
For each lexeme, produce a <u>token</u> in the form: $< token - name, attribute - value >$.
- Token-name: an abstract symbol that is used during syntax analysis.
- Attribute-value: points to an entry in the symbol table.
  The information in the table entry is needed for semantic analysis and code generation.

<u>Lexeme</u> is a string of characters that is a lowest-level syntactic unit in the programming language.
A <u>token</u> is a syntactic category representing a class of lexemes.

Syntax analysis (parsing)

$$\text{Token stream} \longrightarrow \boxed{\text{Syntax Analyzer}} \longrightarrow \text{Syntax tree}$$

<u>Syntax analyzer (parser)</u> uses the token names produced by the lexer to create an intermediate representation that depicts the grammar structure of the token stream (<u>syntax tree</u>).
Interior node represents an operation;
Children of the node represent the arguments of the operation.

Semantic analysis

$$\text{Syntax tree} \longrightarrow \boxed{\text{Semantic Analyzer}} \longrightarrow \text{Syntax tree}$$

Semantic analyzer uses syntax tree and the information in symbol table to check the source program for semantic consistency with the language definition.

It also gathers type information for type checking, type conversion and intermediate code generation.

Syntax describes the proper form of programs.
Semantics describes the meaning of programs.

Intermediate code generation

$$\begin{tikzcd}\text{Syntax tree} \arrow[r] & \boxed{\text{Intermediate Code Generator}} \arrow[r] & \substack{\text{IR} \\ \text{(in three-address code)}}\end{tikzcd}$$

Compilers generate an intermediate representation, typically three-address-code.

- Assembly-like instructions with three operands per instruction.
- Each operand can act like a register.
- Each assignment instruction has at most one operator on the right side.
- Easy to translate into machine instructions of the target machine.

Machine-independent code optimization

$$\begin{tikzcd}\text{IR} \arrow[r] & \boxed{\text{Machine-Independent Code Optimizer}} \arrow[r] & \text{IR}\end{tikzcd}$$

Improve the intermediate code for better target code.

Code generation

$$\begin{tikzcd}\text{IR} \arrow[r] & \boxed{\text{Code Generator}} \arrow[r] & \text{Target-machine code}\end{tikzcd}$$

It's crucial to allocate register and memory to hold values.

Symbol table management

Performed by the frontend, and passed with IR to backend.
Record the variable names and various attributes.

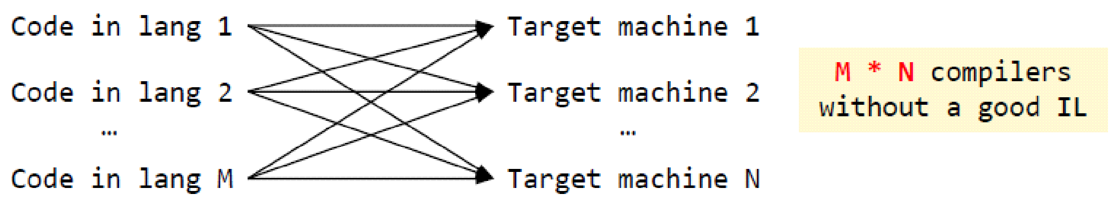
Storage allocated, type, scope.

Record the procedure names and various attributes.

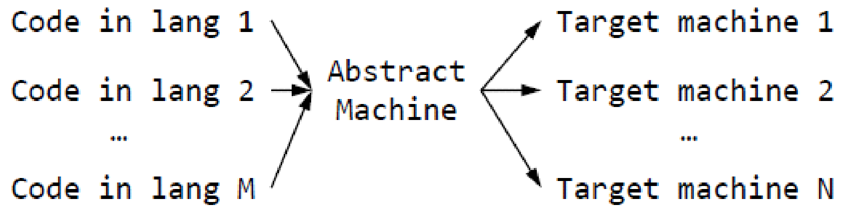
Number and type of arguments, method of passing arguments (value or reference), return type.

Intermediate language (IL)

Without IL, compiler implementation is complex.



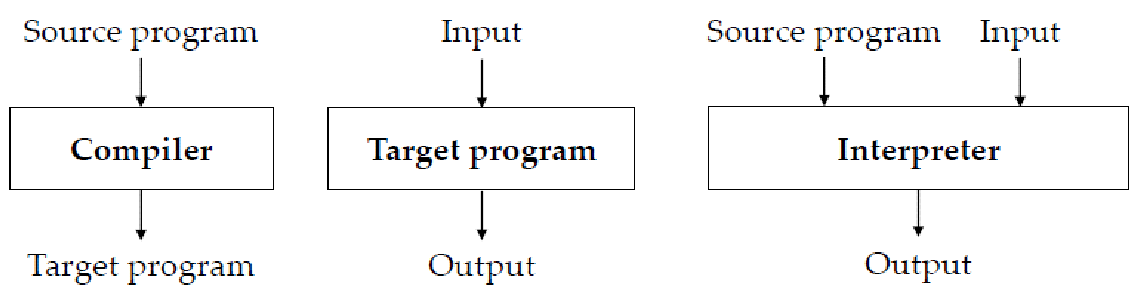
The number of compilers reduced to $M + N$ with a good IL.



Compilers vs Interpreters

- A compiler translates source programs (high-level language) into machine codes.

An interpreter directly executes each statement in the source code.

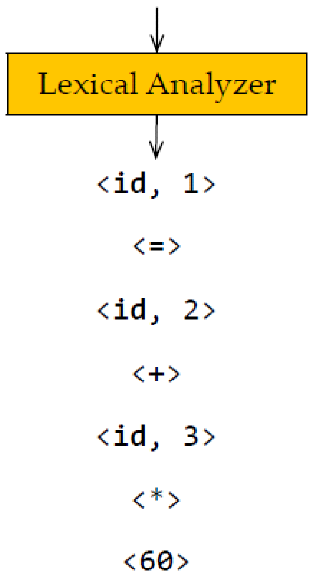


- Interpreters often take less time analyze the source code: simply parses one statement and executes it.
  Compilers analyze the relationships among statements (control and data flow) to enable optimizations.
- Interpreters continue executing a program until the first error is met.
  Compiled language programs are executable only after they are successfully compiled.

CUE COLUMN

Lexical analysis example

```
position = initial + rate * 60
```



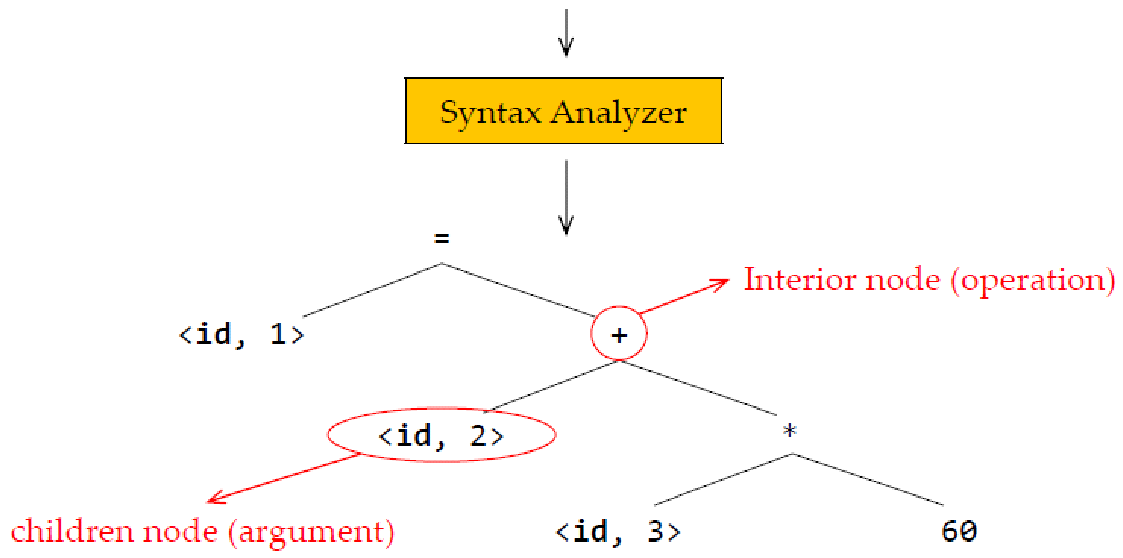
SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...

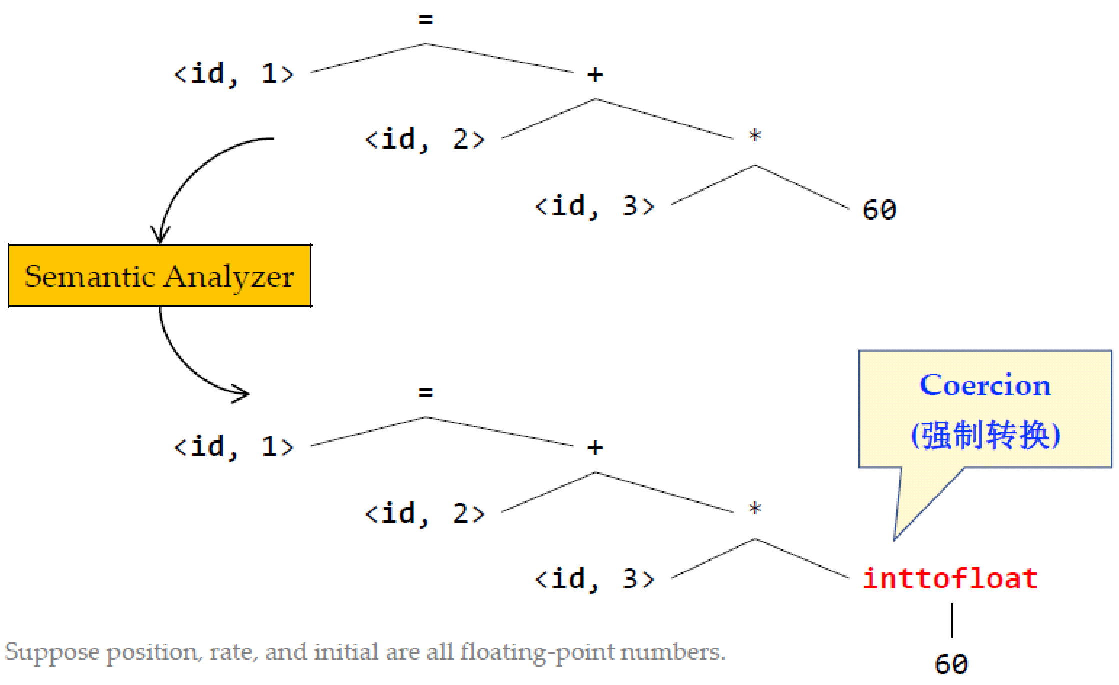
Note: <=>, <+>, <*>, <60> are not in the defined form. This is for notational convenience. <=> could have been <assign, -> and <60> could have been <number, 4>.

Syntax analysis example

```
<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>
```

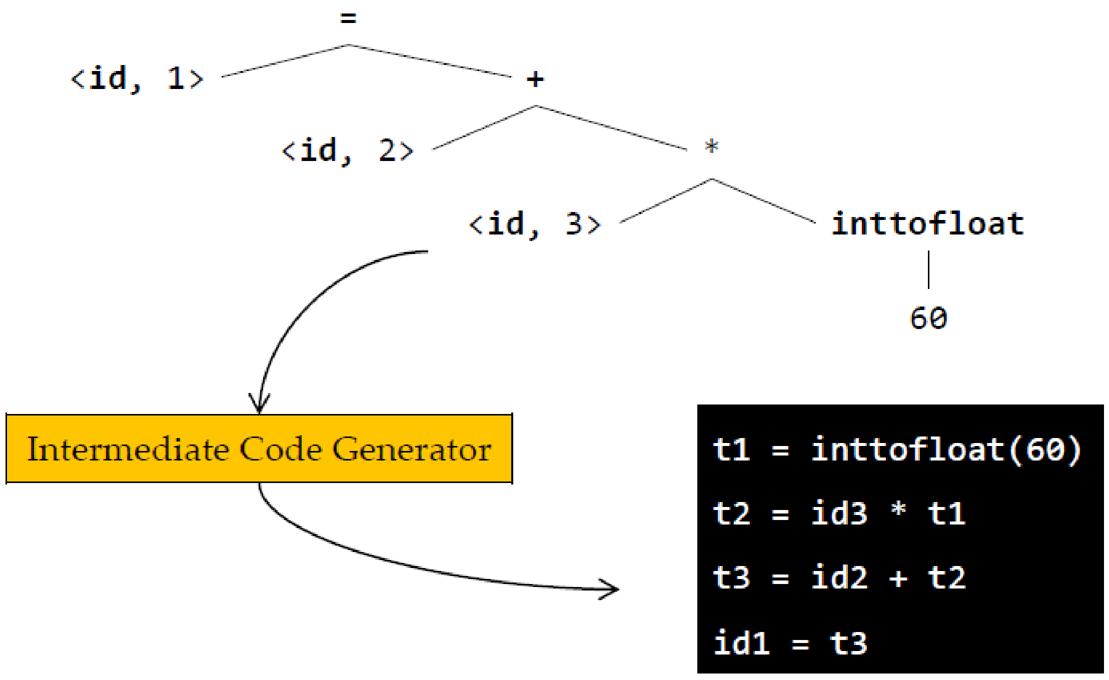


Semantic analysis example



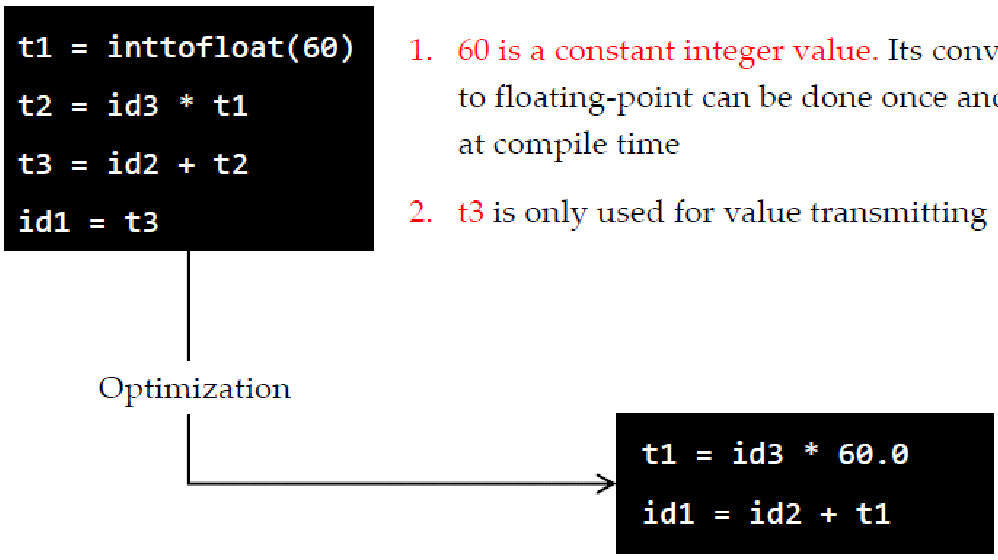
Suppose position, rate, and initial are all floating-point numbers.

Three-address code example



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code optimization example

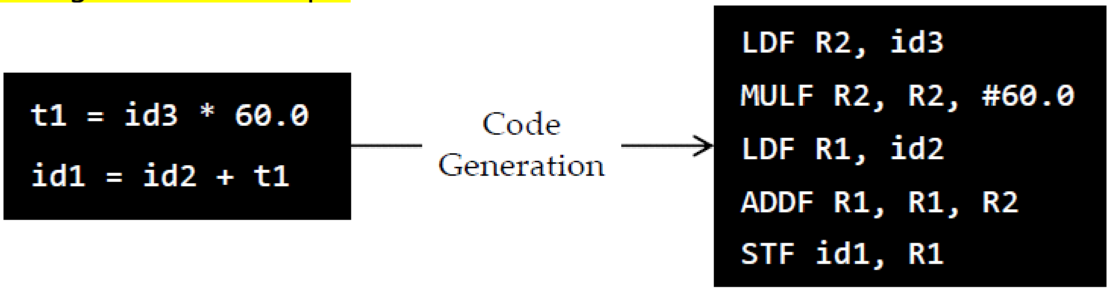


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

1. 60 is a constant integer value. Its conversion to floating-point can be done once and for all at compile time
2. t3 is only used for value transmitting

Code generation example



```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

SUMMARIES

1. References, grading rule, and course content;
2. Compiler structure and phases:
   Frontend and backend of total 7 layers;
   Lexeme, token, syntax tree, symbol table, IR, and IL.
3. Compilers vs Interpreters: 3 differences.