Compilers: Principles, Techniques, and Tools

Preliminaries Required

- ✓ Basic knowledge of CFG (Context Free Grammar).
- ✓ Knowledge of a high programming language (C/C++/Java/Python) for the programming assignments.

Textbook:

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
“*Compilers: Principles, Techniques, and Tools*” Second Edition
Pearson Addison-Wesley, 2007, ISBN 0-321-48681-1

Reference book and websites:

1. Kenneth C. Louden, *Compiler Construction Principles and Practice*，机械工业出版社，2002
2. http://gcc.gnu.org/
3. http://suif.stanford.edu/

Purposes of this course

- Theoretical Part: Basic knowledge of theoretical techniques, such as finite automaton theory, grammar (LL(1), LR(1)), code generation and so on.

- Practical experience: to design and program an actual compiler by hand-writing or using tools, such as FLEX, YACC， Bison etc.

Course Outline

- Introduction to Compiling
- Lexical Analysis
  - Regular expressions, regular definitions
  - NFA, DFA
  - Subset construction, Thompson’s construction
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation
- Assembler Code Generation

A brief history of compiler

- In the late1940s, the stored-program computer invented by John von Neumann， programs were written in machine language， c7 06 0000 0002

- assembly language （汇编语言）: numeric codes were replaced symbolic forms. Mov x, 2

  asssembler （汇编器）: translate the symbolic codes and memory location of assembly language into the corresponding numeric codes.

- Defects of the assembly language： difficult to read ,write and understanding ;dependent on the particular machine.

A brief history of compiler (cont.)

◆ FORTRAN language and its compiler: between 1954 and 1957 , developed by the team at IBM , John Backus.

(1) The structure of natural language studied by **Noam Chomsky**,

(2) The classification of languages according to the complexity of their grammars and the power of the algorithms needed to recognize them.

(3) Four levels of grammars: type 0 、 type 1、 type2 and type3 grammars

- ➢ Type 0: Turing machine
- ➢ Type 1: context-sensitive grammar
- ➢ **Type 2: context-free grammar（上下文无关文法）, the most useful of programming language**
- ➢ Type 3: right-linear grammar, regular expressions and finite automata

A brief history of compiler (cont.)

(4) parsing problem : studied in 1960s and 1970s

(5) Code improvement techniques (optimization techniques): improve compilers efficiency.

(6) Compiler-compilers (parser generator): only in one part of the compiler process.

YACC（语法自动生成器）written in 1975 by Steve Johnson for the UNIX system.

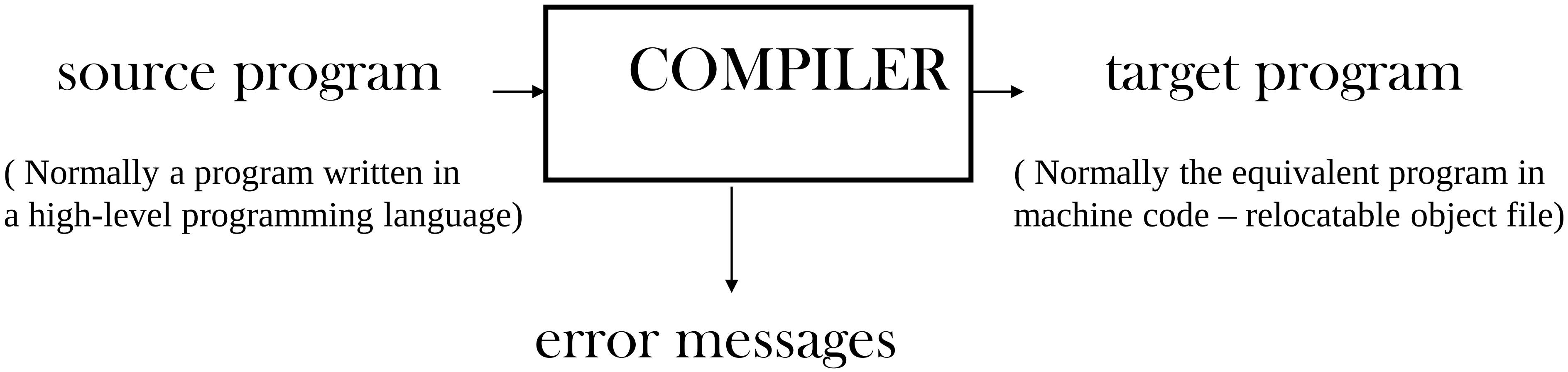
LEX （词法自动生成器） written in 1975 by Mike Lest.

(7) recent advances in compiler design:

- application of more sophisticated algorithms for inferring and /or simplifying the information contained in a program. (with the development of more sophisticated programming languages that allow this kind of analysis.)
- development of standard windowing environments. (interactive development environment. IDE)

Compilers

A **compiler**（编译器） is a program takes a program written in a source language and translates it into an equivalent program in a target language.



source program

(Normally a program written in
a high-level programming language)

COMPILER

target program

(Normally the equivalent program in
machine code – relocatable object file)

error messages

Fig. 1.1. a compiler

Other Applications

- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.

  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL (Structured Query Language,).
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Compiling system

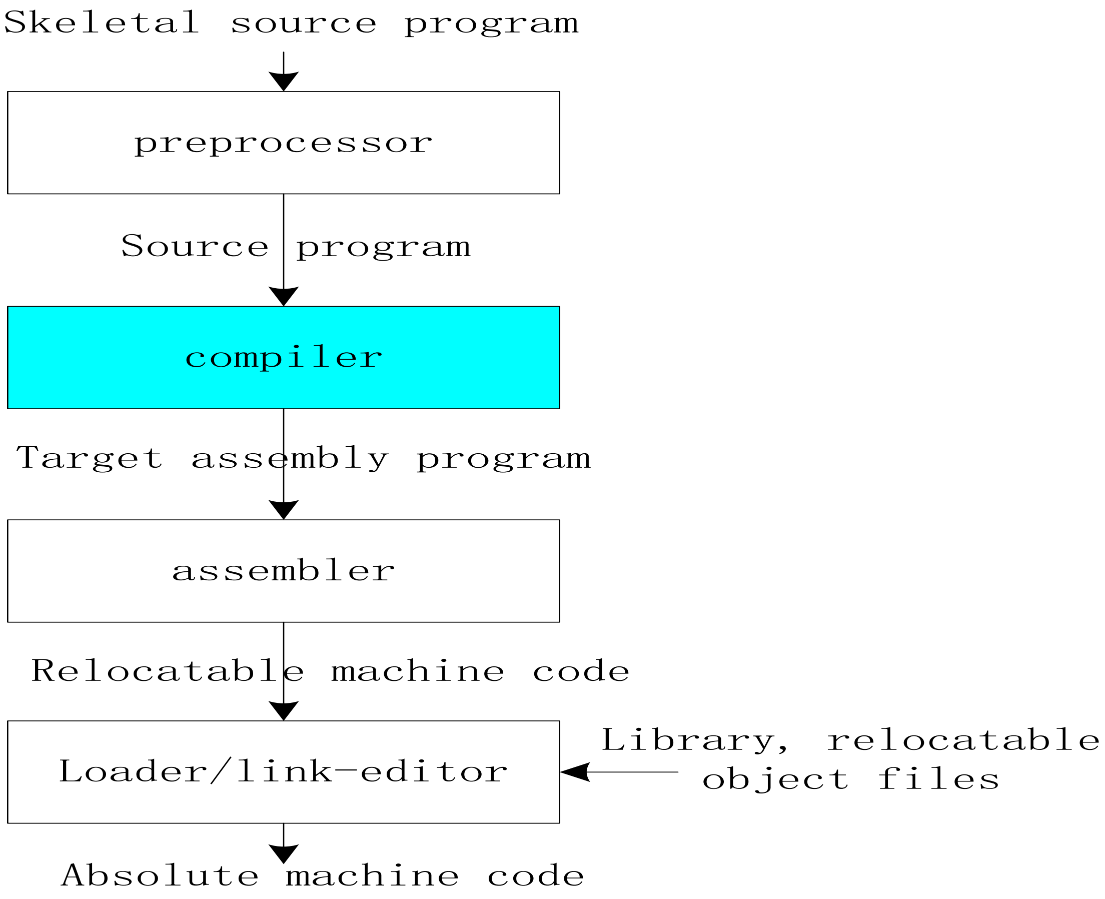


Fig. 1.3 a language-processing system

Cousins of the compiler

- Preprocessors

  delete comments, include other files, perform macro substitutions.

- compilers

  A language translator. It executes the source program immediately. Depending on the language in use and the situation

- Assemblers

  A translator translates assembly language into object code

- Linkers

  Collects code separately compiled or assembled in different object files into a file.

  Connects the code for standard library functions.

  Connects resources supplied by the operating system of the computer.

Cousins of the compiler

- ◆Loaders

  Relocatable : the code is not completely fixed .

  Loaders resolve all relocatable address relative to the starting address.

- ◆Editors

  Produce a standard file (structure based editors)

- ◆Debuggers

  Determine execution errors in a compiled program.

Major Parts of Compilers

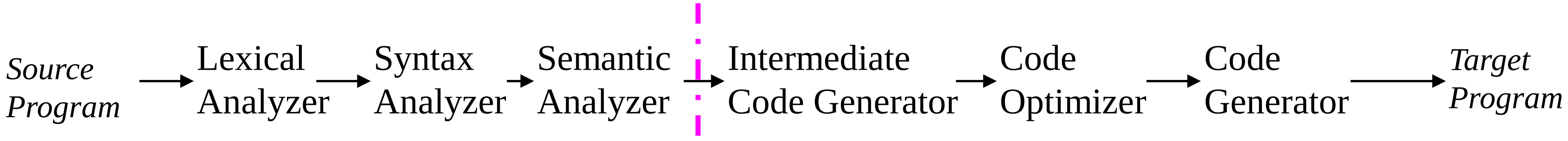
- There are two major parts of a compiler:

  Analysis and **Synthesis/Front-end** and **Back-end**

  (分析与综合/前端与后端)

- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

The Phases of a Compiler （pass）(P.10)



Source Program

Lexical Analyzer

Syntax Analyzer

Semantic Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Target Program

- Each phase transforms the source program from one representation into another representation.

- They communicate with the symbol table（符号表）.

- They communicate with error handlers.

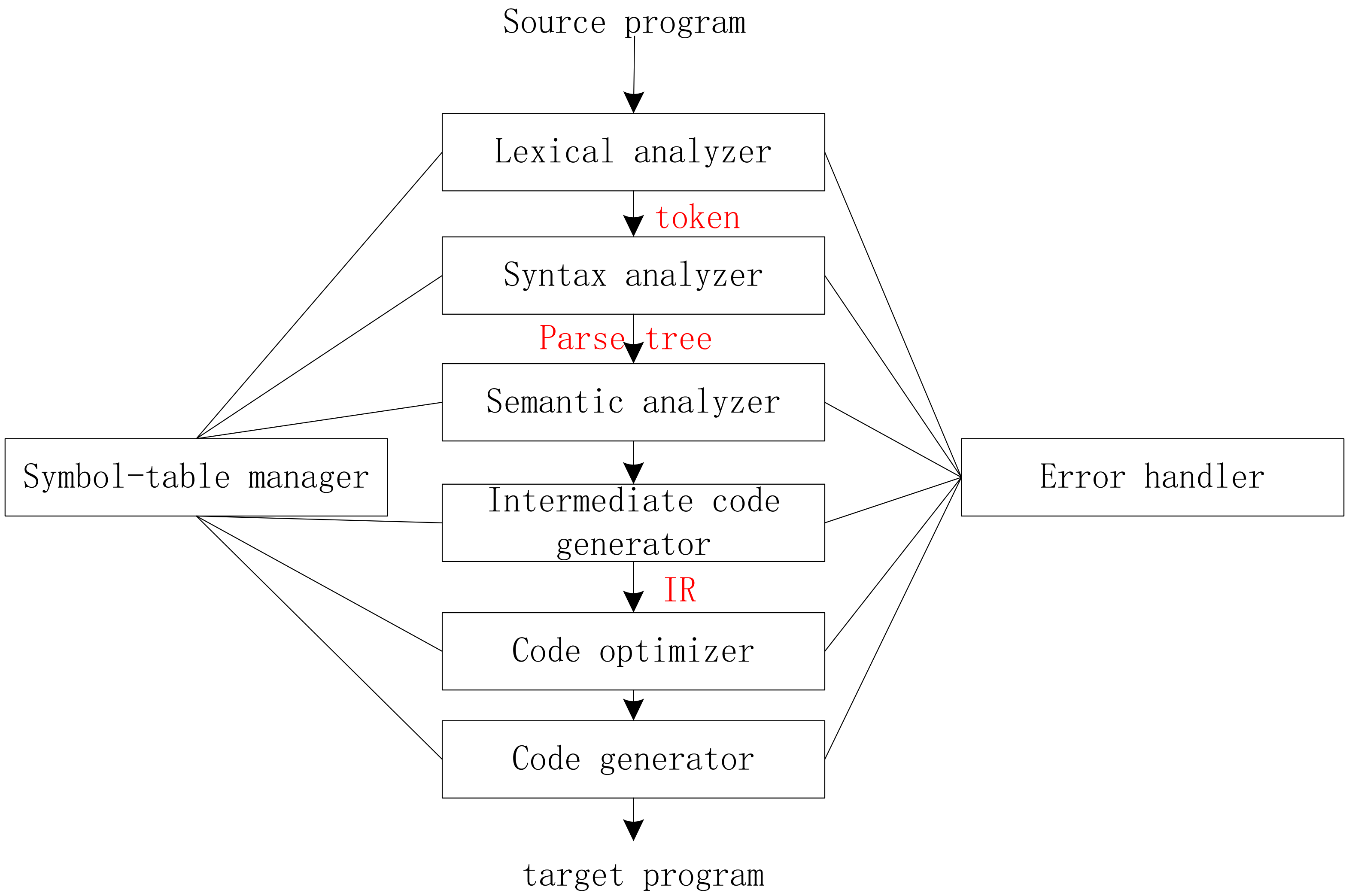


Fig. 1. 9. phases of a compiler

Lexical Analyzer

- **Lexical Analyzer**（词法分析器） reads the source program character by character and returns the *tokens* of the source program.
- A token （单词）describes a pattern of characters having some meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

Ex: newval := oldval + 12 => tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

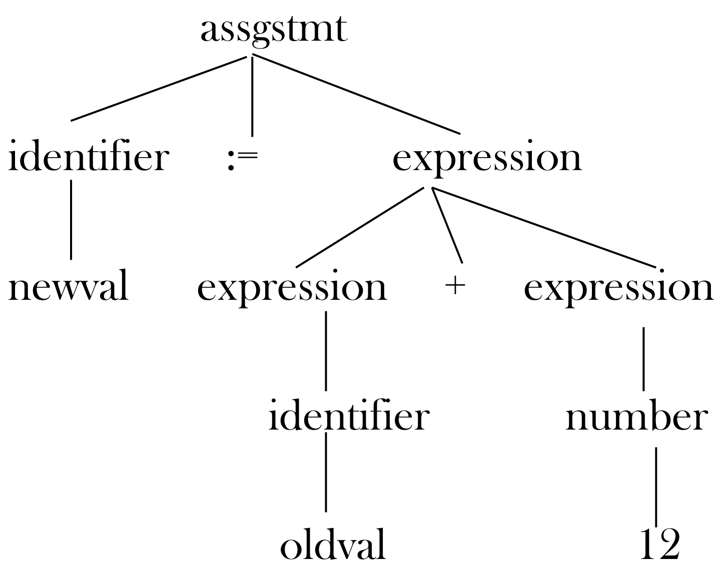
Lexical Analyzer

- Puts information about identifiers into the symbol table.
- Regular expressions （正规/正则表达式） are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton （有限状态自动机） can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** （语法分析器）creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser** （语法分析器）.
- A **parse tree** （语法树）describes a syntactic structure.

Ex: newval := oldval + 12



- In a parse tree, all terminals （终结符）are at leaves.
- All inner nodes are non-terminals in a context free grammar.

Fig.1.4 parse tree for newval := oldval + 12

Syntax Analyzer (cont.)

- The syntax of a language is specified by a **context free grammar** (**CFG**)（上下文无关文法）.
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.

  If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus-Naur Form巴科斯-诺尔范式) to specify a CFG

```
assgstmt     ->  identifier  := expression
expression  ->  identifier
expression  ->  number
expression  ->  expression  +  expression
```

Syntax Analyzer vs. Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures (sentences) in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.

- These parsing techniques are categorized into two groups:
  - *Top-Down Parsing* （自顶向下语法分析）
  - *Bottom-Up Parsing*（自底向上语法分析）

Parsing Techniques (cont.)

◆Top-Down Parsing:

- ➢Construction of the parse tree starts at the root, and proceeds towards the leaves.
- ➢Efficient top-down parsers can be easily constructed by hand.
- ➢Recursive Predictive （递归预测）Parsing,
- ➢Non-Recursive Predictive Parsing (**LL Parsing**).
- ➢ （**L**-left to right; **L**-leftmost derivation（最左推导））

Parsing Techniques (cont.)

◆Bottom-Up Parsing:

- Construction of the parse tree starts at the leaves, and proceeds towards the root.
- Normally efficient bottom-up parsers are created with the help of some software tools.
- Bottom-up parsing is also known as shift-reduce parsing (移进-规约).
- Operator-Precedence （操作符优先）Parsing – simple, restrictive, easy to implement
- LR Parsing – much general form of shift-reduce parsing: LR, SLR, LALR （L-left to right; R-rightmost derivation（最右推导））

Semantic Analyzer

- A semantic analyzer（语义分析器） checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking（类型检查） is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language （上下文无关语言）used in syntax analyzers.

Semantic Analyzer (cont.)

- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules 语义规则)
  - the result is a syntax-directed translation（语法制导翻译）
  - Attribute grammars（属性文法）
- Ex:

```
newval := oldval + 12
```

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Intermediate Code Generation

- ◆A compiler may produce an explicit intermediate codes representing the source program.
- ◆These intermediate codes are generally machine (architecture) independent (except GCC). But the level of intermediate codes is close to the level of machine codes.

Intermediate Code Generation (example)

- Ex:

newval := oldval * fact + 1

id1 := id2 * id3 + 1

```
MULT temp1, id2, id3;
ADD  temp2, temp1, #1
MOV  id1, temp2
```

Intermediates Codes
(Quadraples or three address code)

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

```
MULT  temp1,  id2,  id3
ADD   temp2,  temp1,  #1
MOV   id1,  temp2
                 ↓
MULT  temp1,  id2,  id3
ADD   id1,  temp1,  #1
```

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.
- Ex:

id1 := id2 * id3 + 1

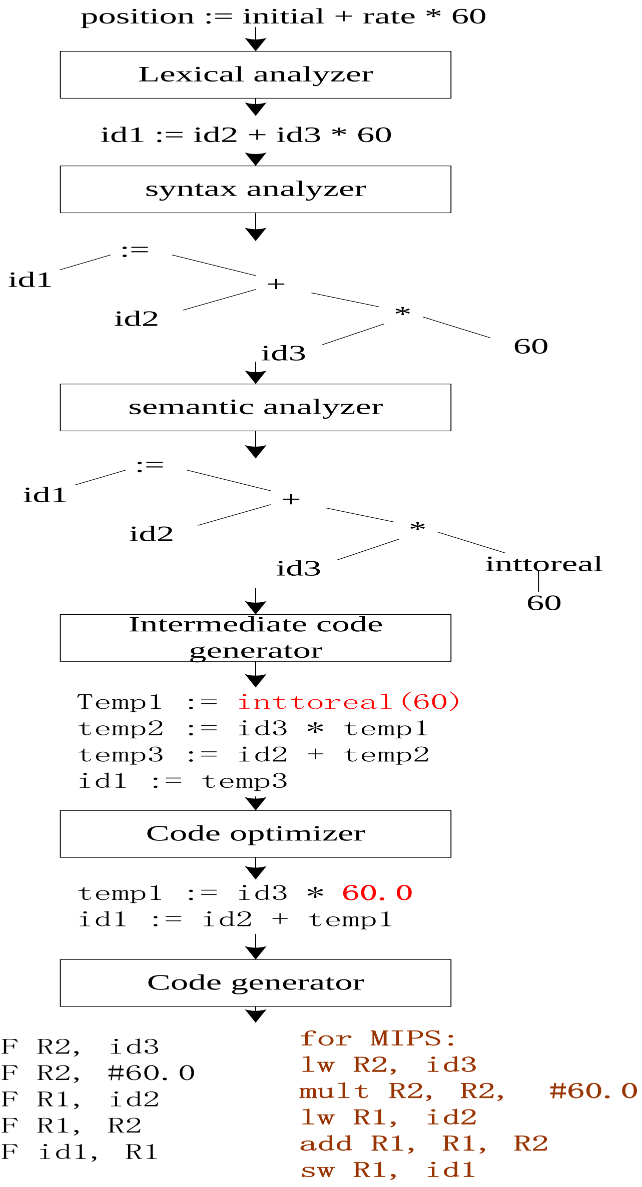
(assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
ARM:
LDR     R1, [id2]
LDR     R2, [id3]
MULT    R3,  R1,  R2
ADD     R1,  R3,  #1
STR     R1, [id1]
```

```
MOV    R1,  id2
MULT   R1,  id3
ADD    R1,  #1
MOV    id1,  R1
MULT   temp1,  id2,  id3
ADD    id1,  temp1,  #1
```

Symbol table

position	…
initial	…
rate	…



```
Temp1  := inttoreal(60)
temp2  := id3 * temp1
temp3  := id2 + temp2
id1  := temp3
```

```
temp1  := id3 * 60.0
id1  := id2 + temp1
```

```
MOVF  R2,  id3
MULF  R2,  #60.0
MOVF  R1,  id2
ADDF  R1,  R2
MOVF  id1,  R1
```

```
for MIPS:
lw R2,  id3
mult R2,  R2,   #60.0
lw R1,  id2
add R1,  R1,  R2
sw R1,  id1
```

Fig.1.10. translation of a statement

Symbol-table management

- Record the identifiers used in the source program and
- Collect information about various attributes of each identifiers
- Provide information about the storage allocated for an identifier, its type, its scope, and so on.

Symbol-table management (cont.)

- ◆A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier
- ◆In lexical analyzer the identifier is entered into the symbol table
- ◆The attributes of an identifier cannot normally be determined during lexical analysis.

Error detection and reporting

- Each phase can encounter errors
- A phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- It is very difficult to proceed to analyze program or to correct errors.

Major data structures in a compiler

- **tokens:** a value of an enumerated data type the sets of tokens.

- the **syntax tree:** each node is a record whose fields represent the information collected by the parser and semantic analyzer.

- the **symbol table:** information associated with identifiers, functions, variables, constants, and data types.

Major data structures in a compiler (cont.)

- the literal table:

store: constants and strings need quick insertion and lookup, need not allow deletions

- intermediate code :

this code kept as an array of text strings, a temporary text file, or as a linked list of structures.

- temporary files

using temporary files to hold the products of intermediate steps

The grouping of phases

- analysis and synthesis

  analysis: lexical analysis , syntax analysis, semantic analysis (optimization)

  synthesis: code generation (optimization)

- front end and back end

  separation depend on the source language or the target language

  the front end: the scanner（扫描器）, parser, semantic analyzer, intermediate code synthesis

  the back end: the code generator, some optimization

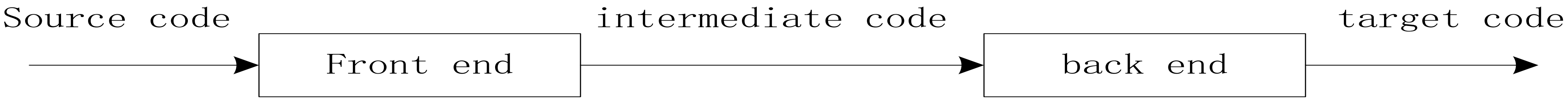


Fig. a. 1. 1

The grouping of phases (cont.)

- Passes（遍）

passes: process the entire source program several times

a pass consist of several phases

one pass or three passes (scanning and parsing, semantic analysis and source-level optimization , code generation and target-level optimization)

- language definition and compilers

relation between the language definition and compiler

the structure and behavior of the runtime environment of the language affect compiler construction.

Other issues in compiler structure (cont.)

- compiler options and interfaces
- interfaces with the operating system
- provide options to the user for various purposes
- ABI (application binary interface): memory, register, assemble code…
- error handling: static error, execution error
- Portability: retargetable compiler, CGG (code generator generator)

Summary

- the constitute of compiler and compiling system
- the concept of lexical analyzer, syntax analyzer, semantic analyzer, code generation, symbol table, intermediate representation, parse tree, token and so on.
- Context free grammar (CFG)
- Regular expression and regular definition
- the concept of retargetable compiler and some typical ones