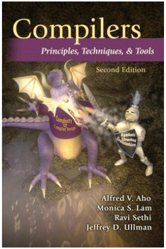
# Introduction

## The Structure of a Compiler

# Text Books


Compilers
Principles, Techniques, & Tools
Second Edition
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman


Maurizio Gabbrielli
Simone Martini
Linguaggi di programmazione
Principi e paradigmi
Seconda edizione
McGraw-Hill
web site

# What is a compiler?

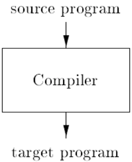



Figure 1.1: A compiler

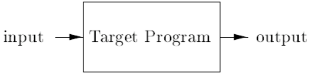



Figure 1.2: Running the target program

# Interpreter

Another kind of language processing

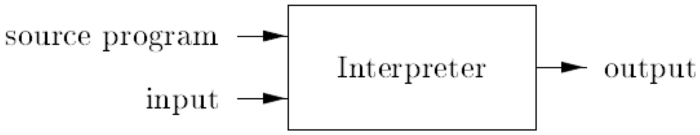



Figure 1.3: An interpreter

# Hybrid Approaches

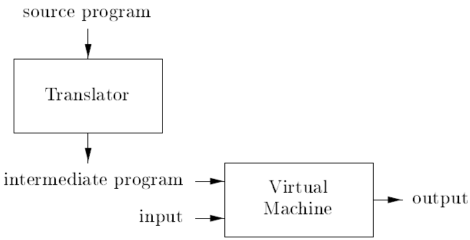



Figure 1.4: A hybrid compiler

- Combine compilation and interpretation (Java *bytecode* and *virtual machine*)
- Java *just-in-time* compilers.

# Producing a machine code

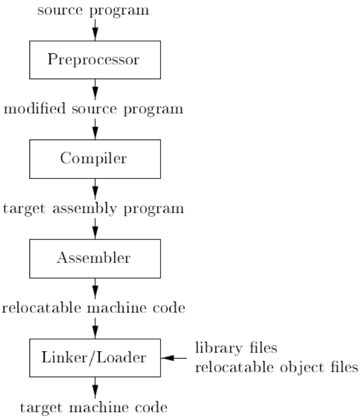



Figure 1.5: A language-processing system

# Phases of a Compiler

- Analysis or *front-end*
- Synthesis or *back-end*

The symbol table stores information about the entire source program.
Maps variables into attributes, i.e. type, name, dimension, address, etc.
This information helps us detecting inconsistencies and misuses during type checking.

# Compilation process

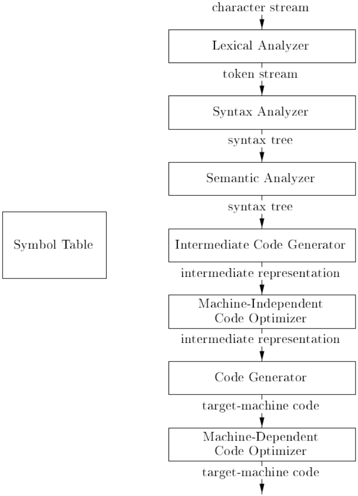



Figure 1.6: Phases of a compiler

# Compilation process

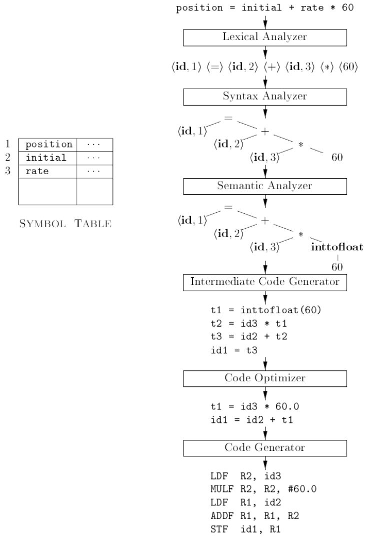



Figure 1.7: Translation of an assignment statement

# Analysis: A Simple Example

Consider the simple Java program:

```
{
     int i; int j; float[100] a; float v; float x;

     while ( true ) {
          do i = i+1; while ( a[i] < v );
          do j = j-1; while ( a[j] > v );
          if ( i >= j ) break;
          x = a[i]; a[i] = a[j]; a[j] = x;
     }
}
```

Figure 2.1: A code fragment to be translated

# A Simple Example (ctd.)

The compiler front end translates the program into the form:

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

Figure 2.2: Simplified intermediate code for the program fragment in Fig. 2.1

# A Quick Tour

For constructing a compiler front end we need first of all a

- *Syntax* (specified in BNF).

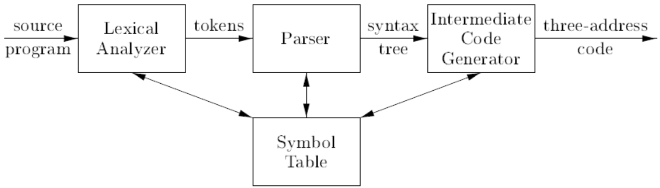



Figure 2.3: A model of a compiler front end

# Lexical Analysis (or Scanning)

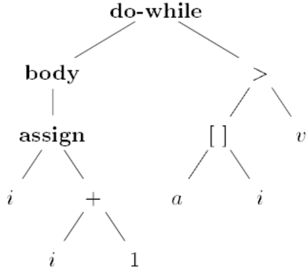
Input stings are split into symbol groups representing syntactic categories, called *lexemes*.

For each lexeme, the scanner produces as output a token:

$$(\texttt{token-name}, \texttt{attribute-value}),$$

- `token-name` is the abstract symbol used in the syntax analysis
- `attribute-value` points to an entry in the symbol table containing information for the semantic analysis and code generation.

# Intermediate Code




(a)

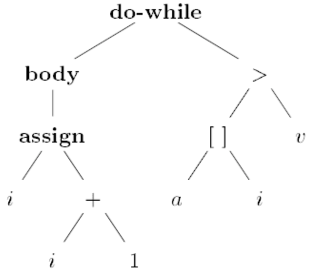
```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

Figure 2.4: Intermediate code for "`do i = i + 1; while ( a[i] < v);`"

# Syntax Analysis (or Parsing)

Problem: How to derive a given string of terminal from the start symbol of the grammar.
If the string (token stream) cannot be derived, then the parser must report syntax errors within the string.




(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

Figure 2.4: Intermediate code for "`do i = i + 1; while ( a[i] < v );`"

# Parse Trees

Consider the following grammar:

$$
\begin{array}{lcl}
\text{list} & ::= & \text{list} + \text{digit} \\
\text{list} & ::= & \text{list - digit} \\
\text{list} & ::= & \text{digit} \\
\text{digit} & ::= & 0|1|2|3|4|5|6|7|8|9
\end{array}
$$

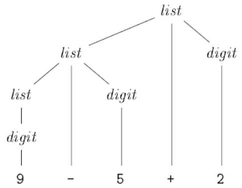



Figure 2.5: Parse tree for 9-5+2 according to the grammar in Example 2.1

# Ambiguity

If we do not distinguish between list and digit we get the grammar:

$$string ::= string + string | string - string | 0|1|2|3|4|5|6|7|8|9.$$

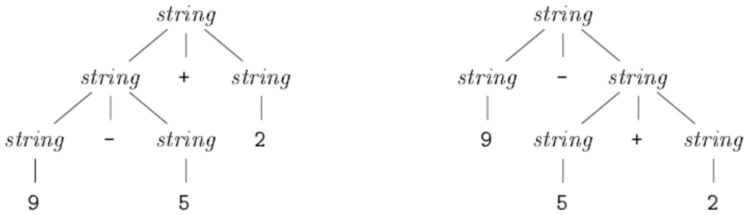



Figure 2.6: Two parse trees for 9-5+2

# Precedence of Operators

A grammar can be defined so as to reflect different associative rules. Operators on the same line have the same precedence.

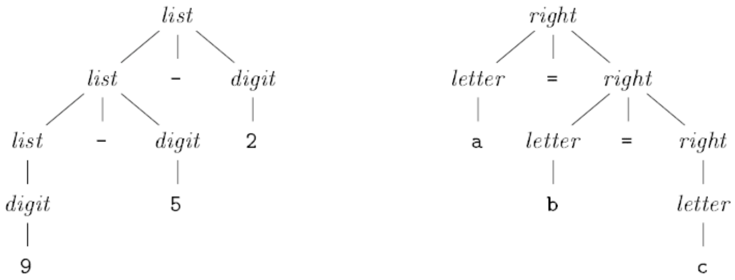



Figure 2.7: Parse trees for left- and right-associative grammars

# An (ambiguous) Grammar for Java

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{id} = \mathit{expression} \ ; \\
& | & \textbf{if} \ ( \ \mathit{expression} \ ) \ \mathit{stmt} \\
& | & \textbf{if} \ ( \ \mathit{expression} \ ) \ \mathit{stmt} \ \textbf{else} \ \mathit{stmt} \\
& | & \textbf{while} \ ( \ \mathit{expression} \ ) \ \mathit{stmt} \\
& | & \textbf{do} \ \mathit{stmt} \ \textbf{while} \ ( \ \mathit{expression} \ ) \ ; \\
& | & \{ \ \mathit{stmts} \ \} \\
\\
\mathit{stmts} & \rightarrow & \mathit{stmts} \ \mathit{stmt} \\
& | & \epsilon
\end{array}
$$

Figure 2.8: A grammar for a subset of Java statements

# Syntax-Directed Translation

Attaching rules to productions in a grammar.

Essential concepts:

- Attibutes: any quantity associated with a programming construct.
- Translation schemes: notations for attaching program fragments to the productions of a grammar.

Example:

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $expr \rightarrow expr_1 + term$ | $expr.t = expr_1.t \parallel term.t \parallel '+'$ |
| $expr \rightarrow expr_1 - term$ | $expr.t = expr_1.t \parallel term.t \parallel '-'$ |
| $expr \rightarrow term$ | $expr.t = term.t$ |
| $term \rightarrow 0$ | $term.t = '0'$ |
| $term \rightarrow 1$ | $term.t = '1'$ |
| $\cdots$ | $\cdots$ |
| $term \rightarrow 9$ | $term.t = '9'$ |

Figure 2.10: Syntax-directed definition for infix to postfix translation

# An Annotated Parse Tree

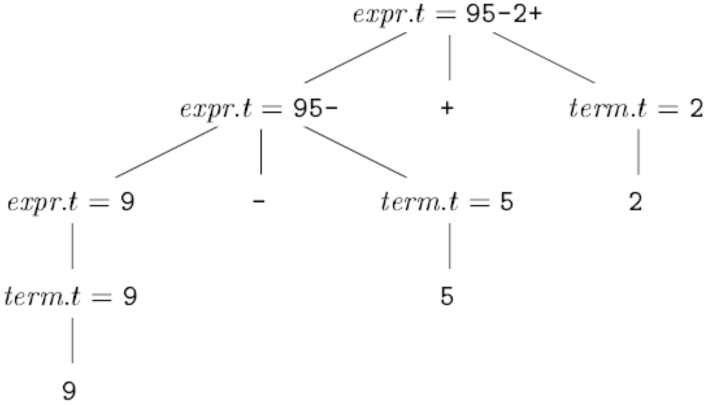



Figure 2.9: Attribute values at nodes in a parse tree

# Parsing

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{expr} \; ; \\
 & | & \textbf{if} \; ( \; \textbf{expr} \; ) \; \mathit{stmt} \\
 & | & \textbf{for} \; ( \; \mathit{optexpr} \; ; \; \mathit{optexpr} \; ; \; \mathit{optexpr} \; ) \; \mathit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\mathit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

Figure 2.16: A grammar for some statements in C and Java

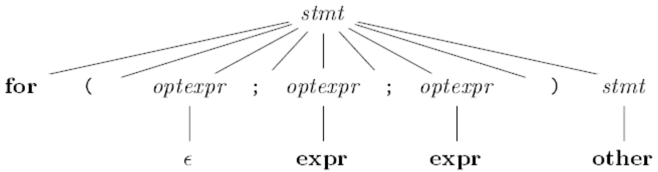



Figure 2.17: A parse tree according to the grammar in Fig. 2.16

# Top-down Parsing