Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1403-1404

References

1. **Compilers: Principles, Techniques, and Tools** (Second Edition), Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007.
2. **Modern Compiler Design** (Second Edition), D. Grune, H. Bal, C. Jacobs, and K. Langendoen, John Wiley, 2012.

Syllabus

- Introduction to Compilers
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate-Code Generation
- Run-Time Environments
- Code Generation
- Machine-Independent Optimizations

Introduction

Introduction

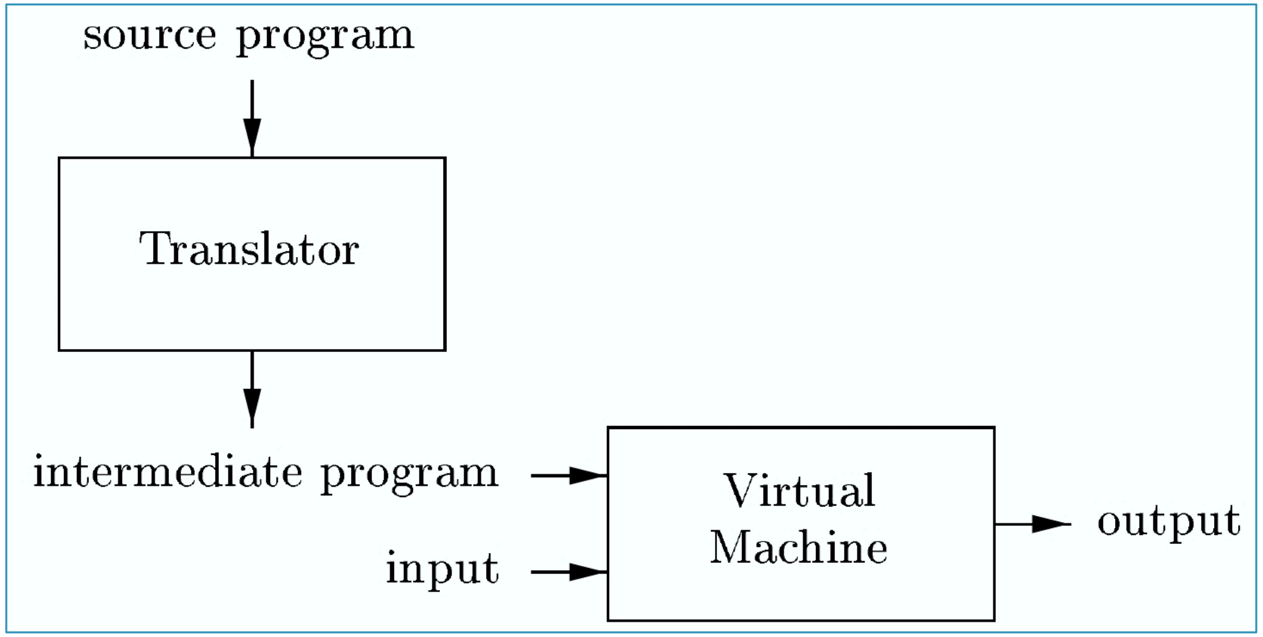
- Programming languages are notations for describing computations to people and to machines
- Before a program can be run, it first must be **translated** into a form in which it can be executed by a computer
  - **The software systems that do this translation are called compilers**
- **This course is about how to design and implement compilers**

Compiler vs. Interpreter

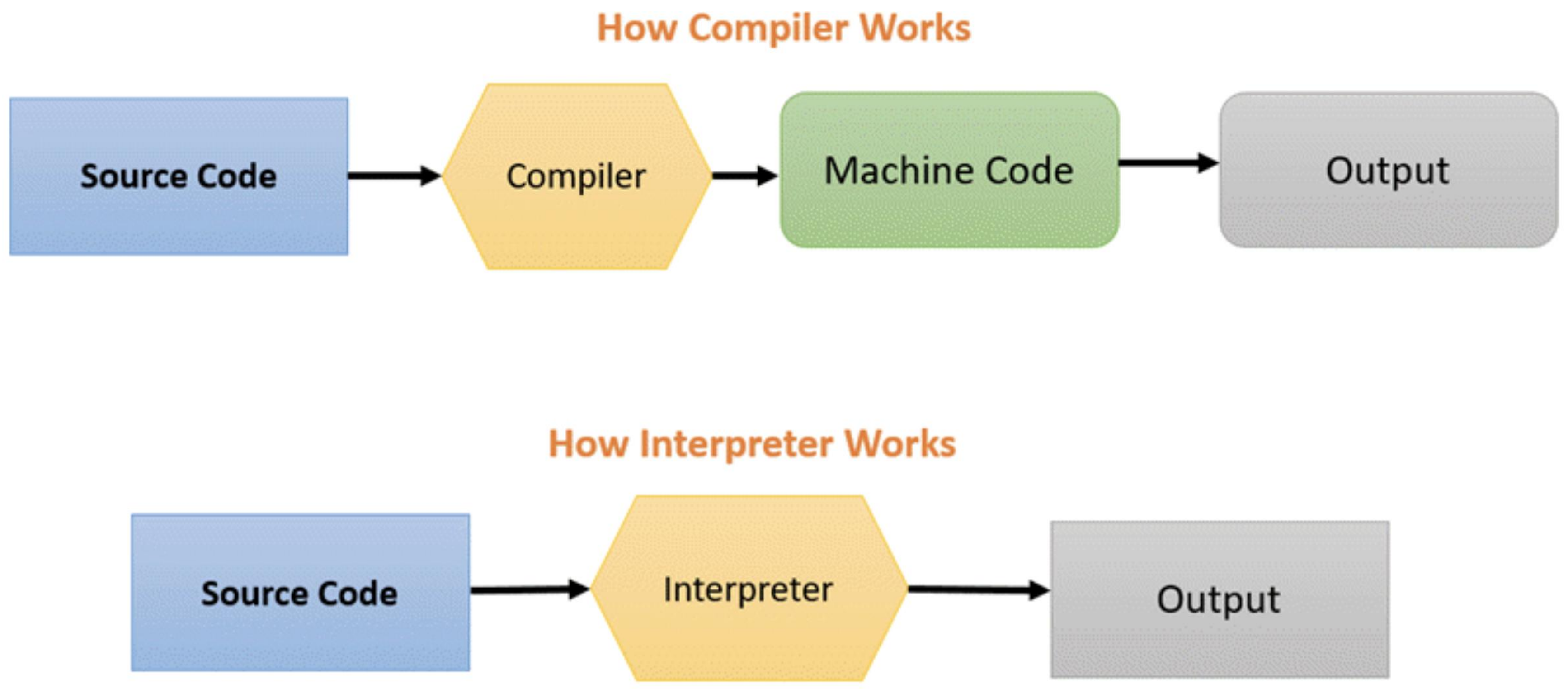
- **A compiler** is a program that can read a program in one language (**the source language**) and translate it into an equivalent program in another language (**the target language**)
- **An interpreter** directly executes the operations specified in the source program on inputs supplied by the user
- ***The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs***
- ***An interpreter can usually give better error diagnostics than a compiler, because it executes the source program statement by statement***

Compiler vs. Interpreter

- **Example**
  - Java language processors combine compilation and interpretation
    - A Java source program **first be compiled** into an intermediate form called bytecodes
    - The bytecodes are **then interpreted** by a virtual machine

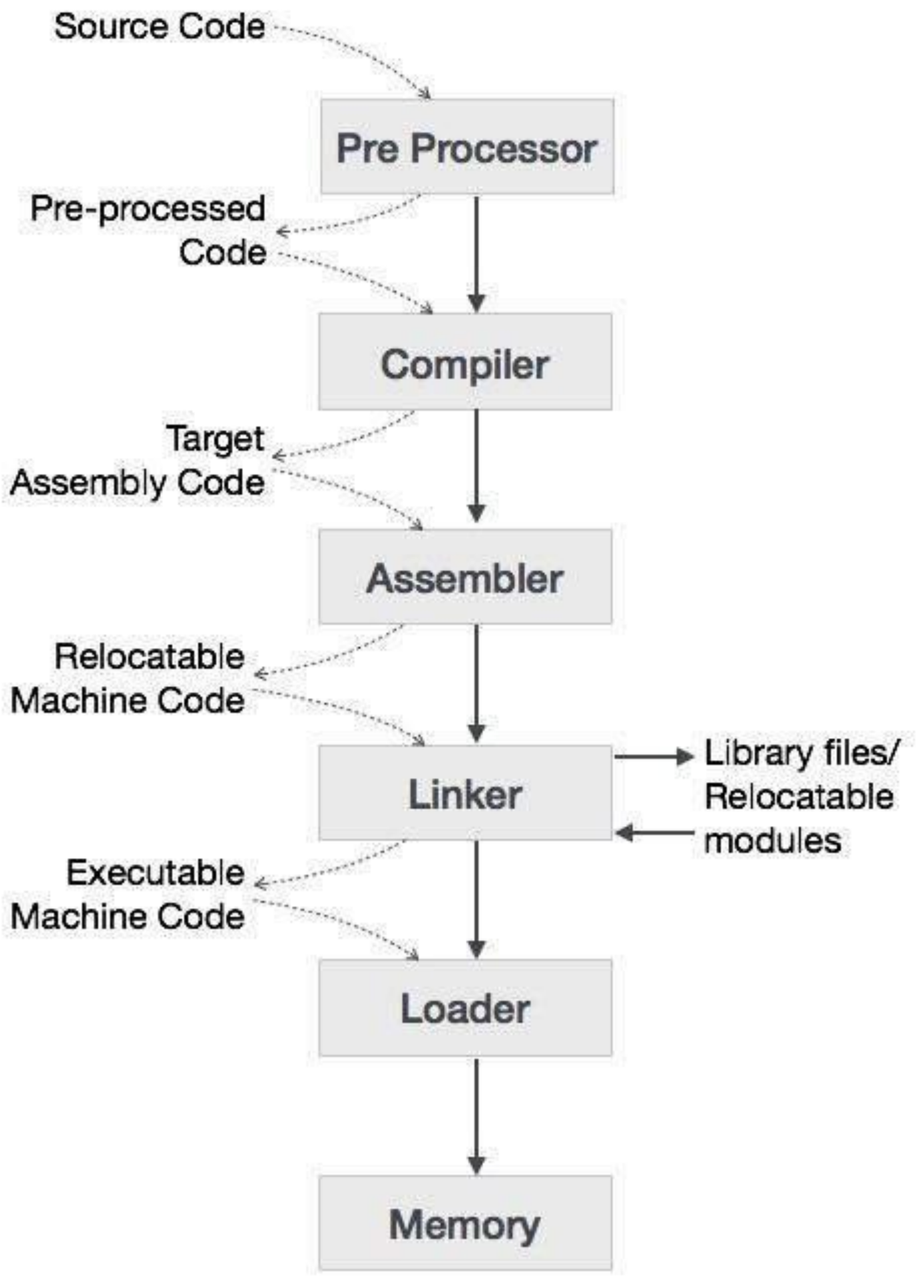


Compiler vs. Interpreter



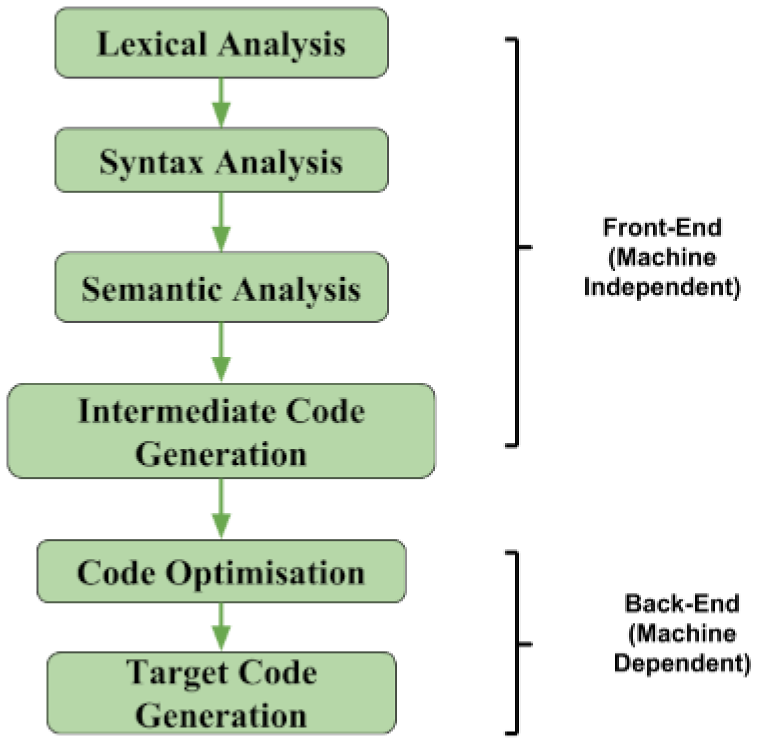
A Language-Processing System

- **Preprocessor**
  - Collecting the source program
- **Compiler**
  - Producing an assembly-language program
- **Assembler**
  - Producing relocatable machine code
- **Linker**
  - Resolving external memory addresses, where the code in one file may refer to a location in another file
- **Loader**
  - Putting together all of the executable object files into memory for execution



The Structure of a Compiler

- Two general parts of a compiler
  - The analysis part (Front-end)
  - The synthesis part (Back-end)


Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimisation
Target Code Generation
Front-End (Machine Independent)
Back-End (Machine Dependent)

The Structure of a Compiler

- Two general parts of a compiler
  - **The analysis part**
    - This part **breaks up the source program** into constituent pieces and **imposes a grammatical structure** on them
    - It then uses this structure to **create an intermediate representation of the source program**
    - The analysis part also collects information about the source program and stores it in a data structure called a **symbol table**
  - **The synthesis part**
    - The synthesis part **constructs the desired target program** from the intermediate representation and the information in the symbol table

The Structure of a Compiler

- **Lexical Analysis**
  - The first phase of a compiler is called lexical analysis or scanning
  - The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called ***lexemes***
  - For each lexeme, the lexical analyzer produces as output a token of the form:

<token-name, attribute-value>

An abstract symbol that is used during syntax analysis

Points to an entry in the symbol table for this token

The Structure of a Compiler

- Lexical Analysis
  - **Example**

position = initial + rate * 60

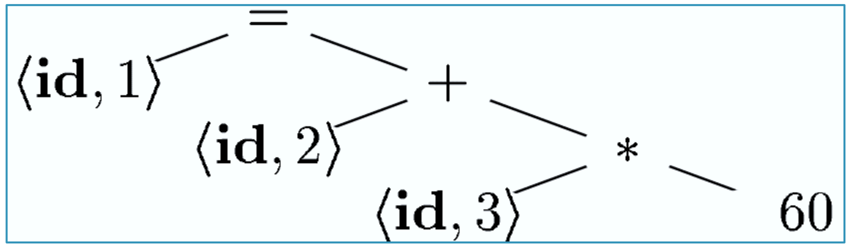
$$\Downarrow$$

$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$$

  - **Example: Lexical errors**
    - int 5temp;
    - a = 123.23.45;
    - char s[10] = “ali;

The Structure of a Compiler

- **Syntax Analysis**
  - The second phase of the compiler is syntax analysis or parsing
  - The parser uses the first components of the tokens produced by the lexical analyzer **to create a tree-like intermediate representation that depicts the grammatical structure of the token stream**
  - A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation

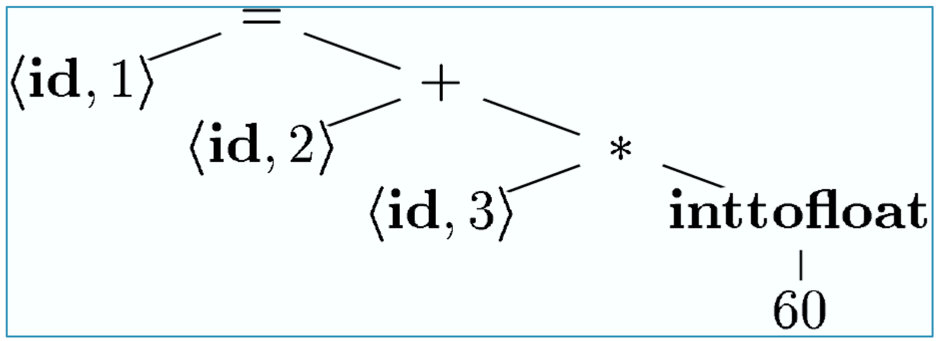


The Structure of a Compiler

- Syntax Analysis
  - **Example: Syntax errors**
    - a b =;
    - int q = 6
    - if (a > 1

The Structure of a Compiler

- Semantic Analysis
  - The semantic analyzer uses the syntax tree and the information in the symbol table to **check the source program for semantic consistency with the language definition**
  - It also **gathers type information** and saves it in either the syntax tree or the symbol table



The Structure of a Compiler

- Semantic Analysis
  1. **Type checking**
     - **Example**
       - The compiler must report an error if a floating-point number is used to index an array
  2. **Type casting**
     - **Example**
       - The compiler may convert the integer into a floating-point number
  3. **Redefine variable**
  4. **Check function parameters**

The Structure of a Compiler

- **Intermediate Code Generation**
  - Many compilers **generate an explicit low-level or machine-like intermediate representation**
  - This intermediate representation should have two important properties
    - It should be easy to produce
    - It should be easy to translate into the target machine
  - Three-address code consists of a sequence of assembly-like instructions with three operands per instruction

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

The Structure of a Compiler

- **Code Optimization**
  - The machine-independent code-optimization phase attempts to **improve the intermediate code** so that better target code will result
  - There is a great variation in the amount of code optimization different compilers perform

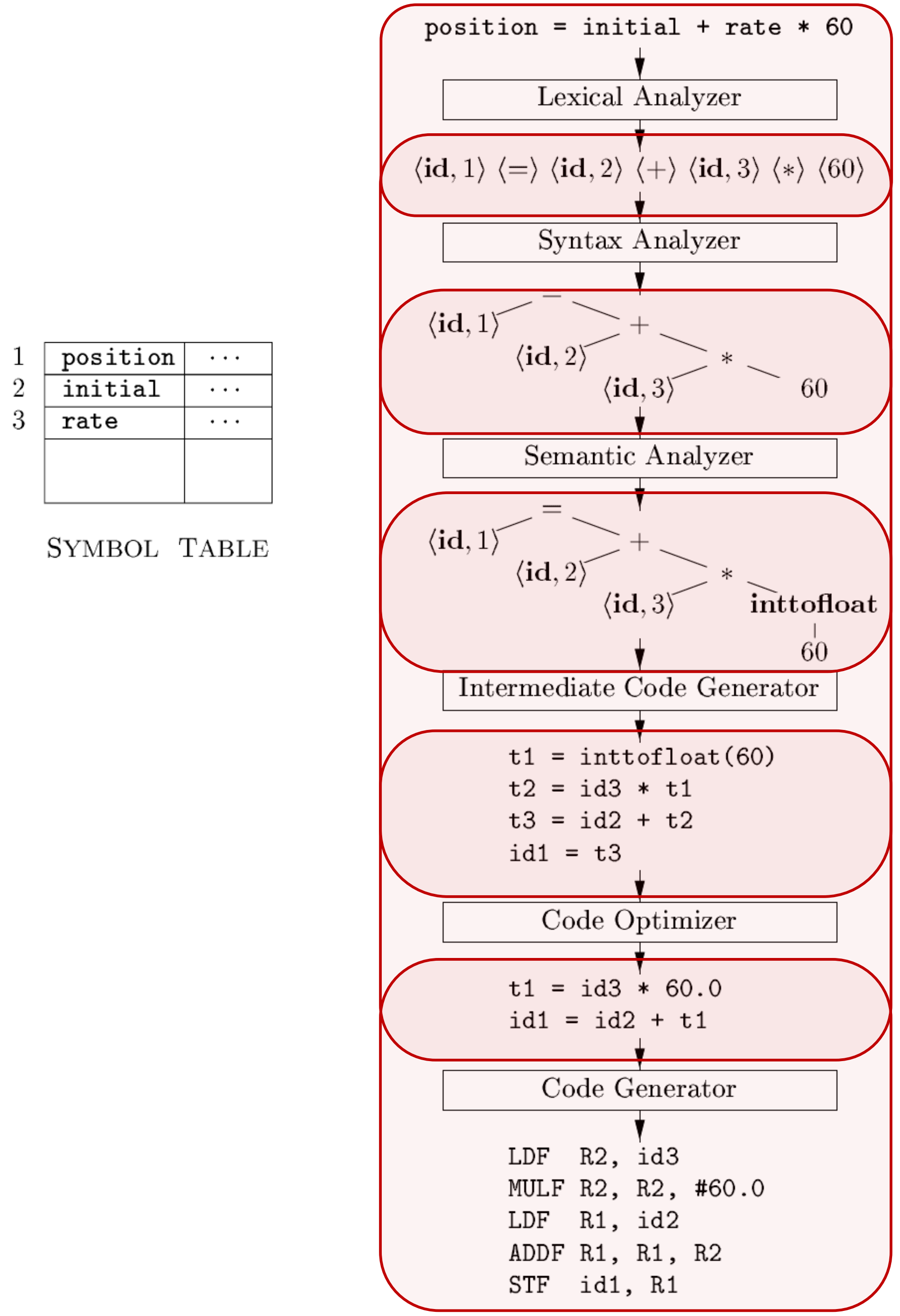
```
t1 = id3 * 60.0
id1 = id2 + t1
```

The Structure of a Compiler

- **Code Generation**
  - The code generator takes as input an intermediate representation of the source program and maps it into the target language

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

The Structure of a Compiler


position = initial + rate * 60
Lexical Analyzer
$\langle \mathbf{id}, 1\rangle$ $\langle = \rangle$ $\langle \mathbf{id}, 2\rangle$ $\langle + \rangle$ $\langle \mathbf{id}, 3\rangle$ $\langle * \rangle$ $\langle 60 \rangle$
Syntax Analyzer
=
$\langle \mathbf{id}, 1\rangle$
+
$\langle \mathbf{id}, 2\rangle$
*
$\langle \mathbf{id}, 3\rangle$
60
Semantic Analyzer
=
$\langle \mathbf{id}, 1\rangle$
+
$\langle \mathbf{id}, 2\rangle$
*
$\langle \mathbf{id}, 3\rangle$
inttofloat
60
Intermediate Code Generator
Code Optimizer
Code Generator
SYMBOL TABLE

1	position	···
2	initial	···
3	rate	···

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

```
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
```

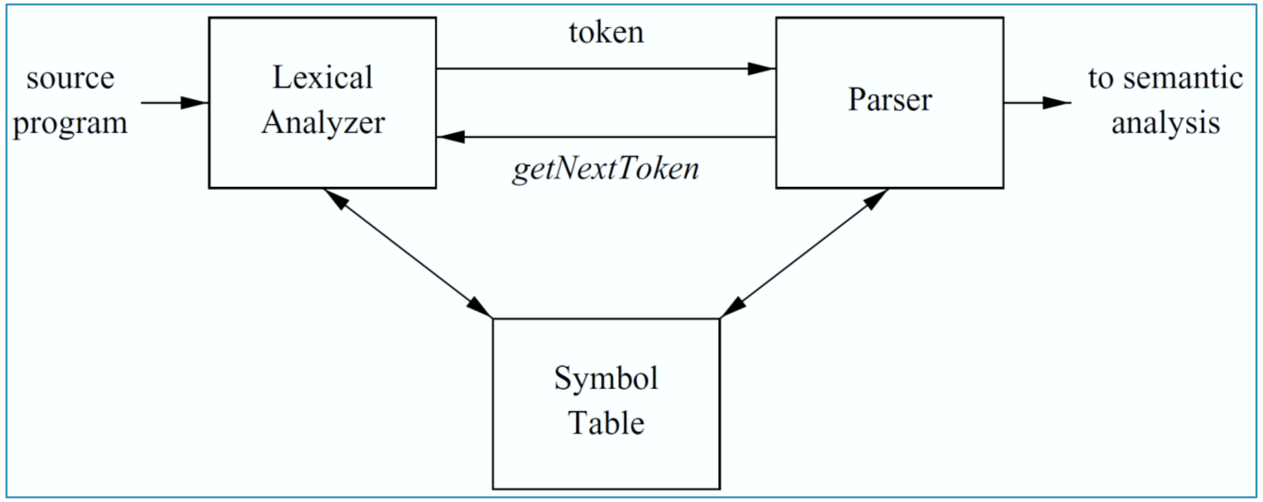
Lexical Analysis

Lexical Analysis

- The main task of the lexical analyzer:
  1. **Read the input characters of the source program**
  2. **Group them into lexemes**
  3. **Produce as output a sequence of tokens**

- **Lexical errors**
  - It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error
  - **Example**

    fi (a == f(x)) …



Lexical Analysis

- Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes
  1. **Stripping out comments and whitespace**
  2. **Correlating error messages generated by the compiler with the source program**
     - For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message

Lexical Analysis

- Sometimes, lexical analyzers are divided into a cascade of two processes:
  1. **Scanning** consists of the simple processes that do not require tokenization of the input, such as:
     - **Deletion of comments**
     - **Compaction of consecutive whitespace characters into one**
  2. **Lexical analysis** produces tokens from the output of the scanner

Lexical Analysis

- What does the lexical analyzer want to do?
- **Example**

```
if (i == j)
  Z = 0;
else
  Z = 1;
```

- The input is just a string of characters:
  - **\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;**
- **Goal:** Partition input string into substrings where the substrings are called **tokens**

Tokens

- **What's a Token?**
  - A syntactic category
    - **In English:**
      - **noun, verb, adjective, …**
    - **In a programming language:**
      - **Identifier, Integer, Keyword, Whitespace, …**
- A token class corresponds to a set of strings
  - **Examples**
    - **Identifier**: Strings of letters or digits, starting with a letter
    - **Integer**: A non-empty string of digits
    - **Keyword**: "else" or "if" or "for" or …
    - **Whitespace**: A non-empty sequence of blanks, newlines, and tabs
- **What are Tokens For?**
  - Classify program substrings according to role
    - An identifier is treated differently than a keyword

Tokens, Patterns, and Lexemes

- **A token** is a pair consisting of a **token name** and **an optional attribute value**
  - The token names are the input symbols that the parser processes
- **A pattern** is a description of the form that the lexemes of a token may take
  - **Example**
    - **In the case of a keyword as a token**, the pattern is the sequence of characters that form the keyword
    - **In the case of an identifier as a token**, the pattern is a more complex structure that is matched by many strings
- **A lexeme** is a sequence of characters in the source program that matches the pattern for a token

Tokens, Patterns, and Lexemes

- In many programming languages, the following classes cover most or all of the tokens:
  1. **One token for each keyword**
     - The pattern for a keyword is the same as the keyword itself
  2. **Tokens for the operators**, such as the token comparison
  3. **One token representing all identifiers**
  4. **One or more tokens representing constants**, such as numbers and literal strings
  5. **Tokens for each punctuation symbol**, such as left and right parentheses, comma, and semicolon

Tokens, Patterns, and Lexemes

- **Example**

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Attributes for Tokens

- We can assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information
- **Example**
  - Information about an **identifier** is kept in the symbol table, including:
    - Its lexeme
    - Its type
    - The location at which it is first found
  - **The appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier**

Attributes for Tokens

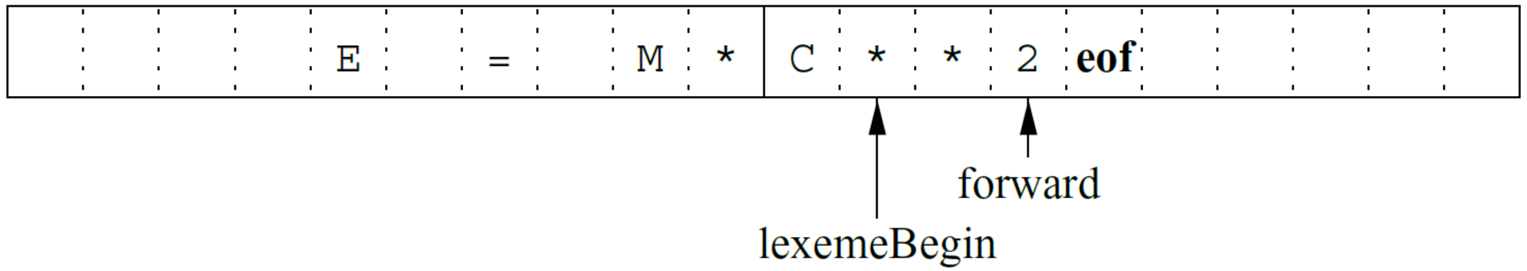
- **Example**
  - The token names and associated attribute values for the Fortran statement: E = M * C ** 2

<**id**, pointer to symbol-table entry for `E`>
<**assign_op**>
<**id**, pointer to symbol-table entry for `M`>
<**mult_op**>
<**id**, pointer to symbol-table entry for `C`>
<**exp_op**>
<**number**, integer value 2>

  - *Note that in certain pairs, especially operators, punctuation, and keywords,* ***there is no need for an attribute value***

Input Buffering

- Some buffering techniques have been developed to reduce the amount of overhead required to process a single input character
- Buffer Pairs
  - Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes
  - Using one system read command we can read N characters into a buffer, rather than using one system call per character

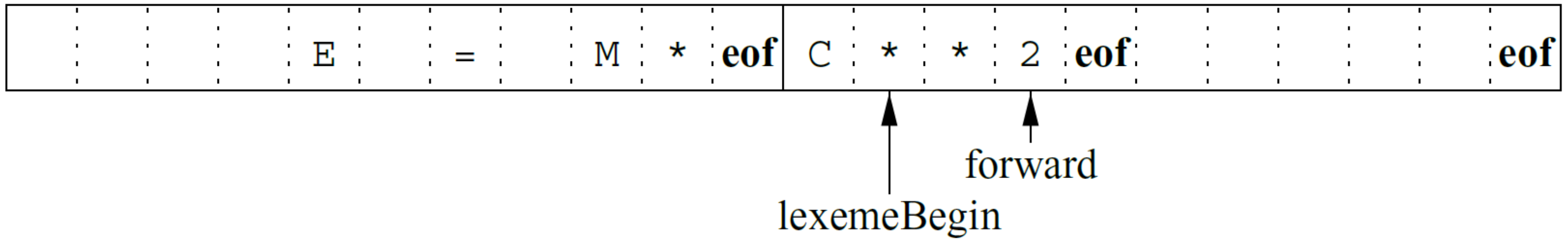


Input Buffering

- Buffer Pairs
  - Two pointers to the input are maintained:
    1. Pointer lexemeBegin, marks the beginning of the current lexeme
    2. Pointer forward scans ahead until a pattern match is found
       - Once the next lexeme is determined, forward is set to the character at its right end
       - After the lexeme is recorded to the parser, lexemeBegin is set to the character immediately after the lexeme just found
  - We first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer

Input Buffering

- Buffer Pairs
  - Sentinels
    - Each time we advance forward , we must check that we have not moved off one of the buffers; if we do, then we must also reload the other buffer
    - The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof


E
=
M
*
eof
C
*
*
2
eof
eof
forward
lexemeBegin