BCSE307L – COMPILER DESIGN

Text Book(s)	
1.	A. V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, Compilers: Principles, techniques, & tools, 2007, Second Edition, Pearson Education, Boston.
Reference Books	
1.	Watson, Des. A Practical Approach to Compiler Construction. Germany, Springer International Publishing, 2017.
Mode of Evaluation: CAT, Quiz, Written assignment and FAT	

Course Objectives
1. To provide fundamental knowledge of various language translators. 2. To make students familiar with lexical analysis and parsing techniques. 3. To understand the various actions carried out in semantic analysis. 4. To make the students get familiar with how the intermediate code is generated. 5. To understand the principles of code optimization techniques and code generation. 6. To provide foundation for study of high-performance compiler design.

Module:1	**INTRODUCTION TO COMPILATION AND LEXICAL ANALYSIS**	**7 hours**
Introduction to LLVM - Structure and Phases of a Compiler-Design Issues-Patterns-Lexemes-Tokens-Attributes-Specification of Tokens-Extended Regular Expression- Regular expression to Deterministic Finite Automata (Direct method) - Lex - A Lexical Analyzer Generator.		
Module:2	**SYNTAX ANALYSIS**	**8 hours**
Role of Parser- Parse Tree - Elimination of Ambiguity – Top Down Parsing - Recursive Descent Parsing - LL (1) Grammars – Shift Reduce Parsers- Operator Precedence Parsing - LR Parsers, Construction of SLR Parser Tables and Parsing- CLR Parsing- LALR Parsing.		
Module:3	**SEMANTICS ANALYSIS**	**5 hours**
Syntax Directed Definition – Evaluation Order - Applications of Syntax Directed Translation - Syntax Directed Translation Schemes - Implementation of L-attributed Syntax Directed Definition.		
Module:4	**INTERMEDIATE CODE GENERATION**	**5 hours**
Variants of Syntax trees - Three Address Code- Types – Declarations - Procedures - Assignment Statements - Translation of Expressions - Control Flow - Back Patching- Switch Case Statements.		
Module:5	**CODE OPTIMIZATION**	**6 hours**
Loop optimizations- Principal Sources of Optimization -Introduction to Data Flow Analysis - Basic Blocks - Optimization of Basic Blocks - Peephole Optimization- The DAG Representation of Basic Blocks -Loops in Flow Graphs - Machine Independent Optimization-Implementation of a naïve code generator for a virtual Machine- Security checking of virtual machine code.		
Module:6	**CODE GENERATION**	**5 hours**
Issues in the design of a code generator- Target Machine- Next-Use Information - Register Allocation and Assignment- Runtime Organization- Activation Records.		
Module:7	**PARALLELISM**	**7 hours**
Parallelization- Automatic Parallelization- Optimizations for Cache Locality and Vectorization- Domain Specific Languages-Compilation- Instruction Scheduling and Software Pipelining- Impact of Language Design and Architecture Evolution on Compilers-Static Single Assignment		
Module:8	**Contemporary Issues**	**2 hours**

Course Outcomes
1. Apply the skills on devising, selecting, and using tools and techniques towards compiler design 2. Develop language specifications using context free grammars (CFG). 3. Apply the ideas, the techniques, and the knowledge acquired for the purpose of developing software systems. 4. Constructing symbol tables and generating intermediate code. 5. Obtain insights on compiler optimization and code generation.

Module:1	INTRODUCTION TO COMPILATION AND LEXICAL ANALYSIS	7 hours
Introduction to LLVM - Structure and Phases of a Compiler-Design Issues-Patterns-Lexemes-Tokens-Attributes-Specification of Tokens-Extended Regular Expression- Regular expression to Deterministic Finite Automata (Direct method) - Lex - A Lexical Analyzer Generator.		

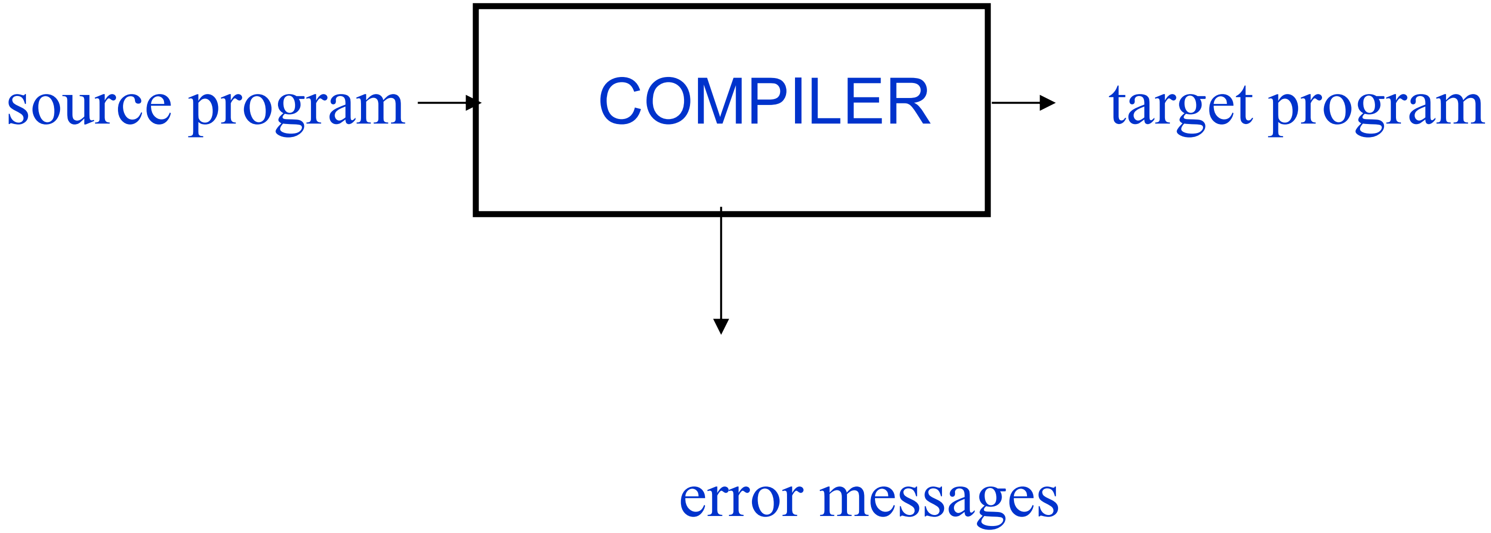
Introduction to Compiler

Language Processors

- Compiler
- Interpreter

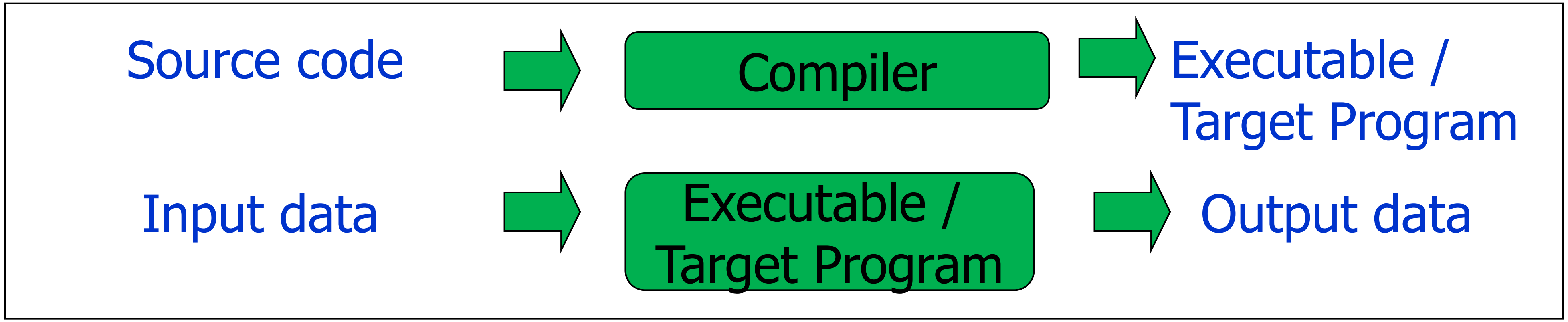
Compiler

- **A compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Compiler

Compilers: Translate a source (human-writable) program to an executable (machine-readable) program



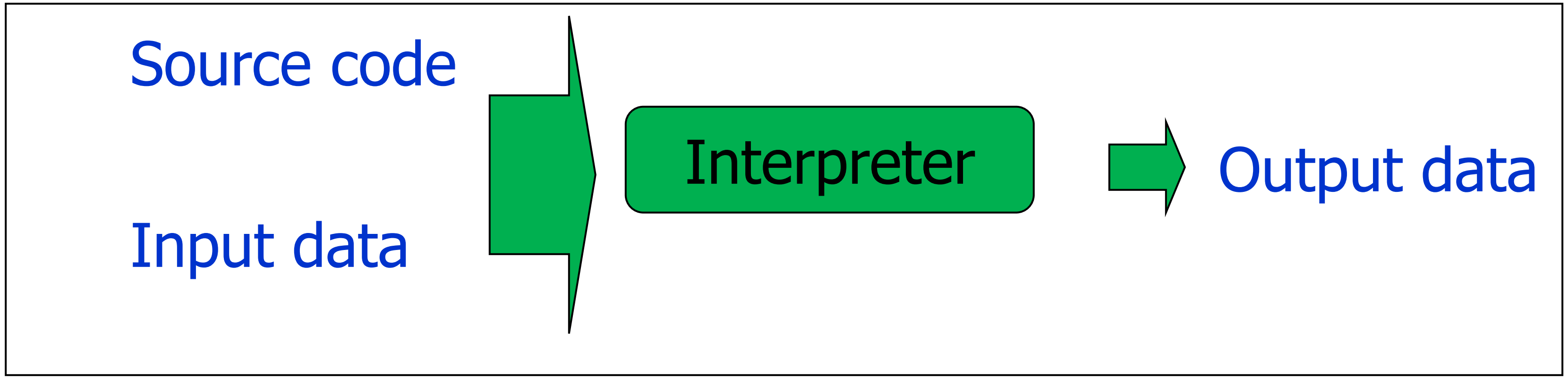
Example: FORTRAN, COBOL, C, C++, Pascal

Interpreter

An Interpreter Run programs "as is" without preliminary translation:

Successive phases of translation (to machine/intermediate code) and execution.

Interpreters: Convert a source program and execute it at the same time.(Line by line execution)



Example: Lisp, BASIC, APL, Perl, Python

Compiler VS. Interpreter

Compiler

– Efficient for production applications

– Order of magnitude faster

Interpreter

– Efficient and rapid for prototyping

– Efficient error reporting

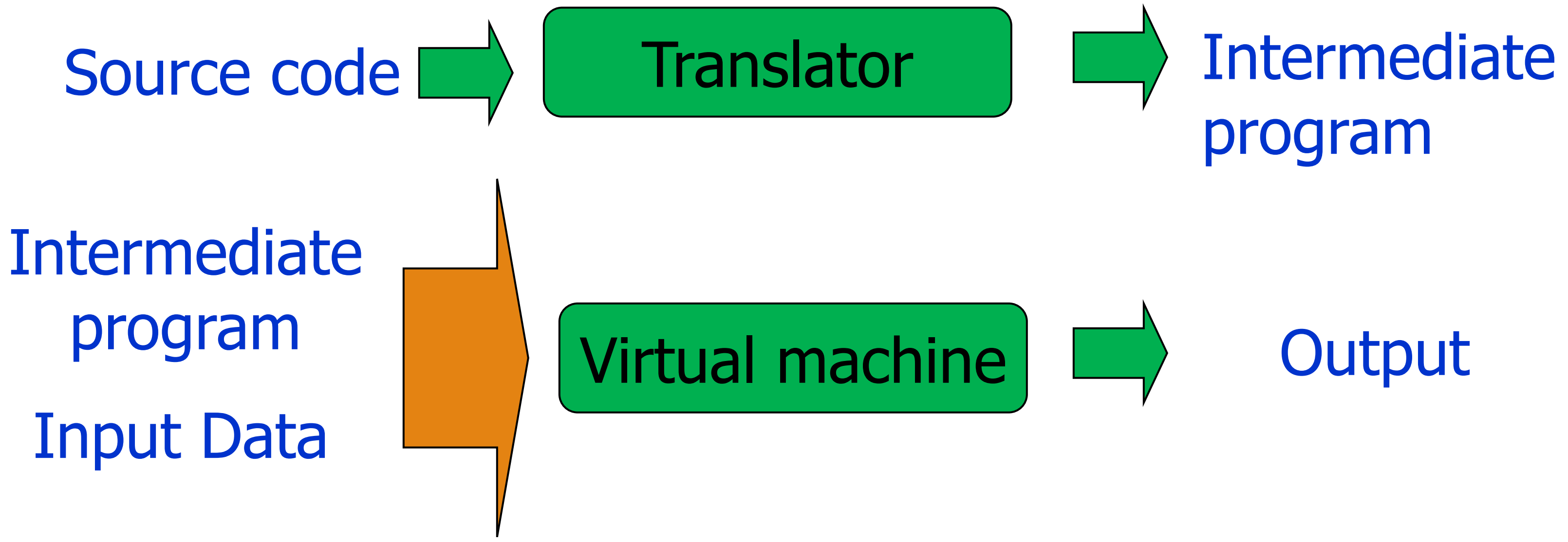
Compiler vs. Interpreter

Compilers: Translate a source (human-writable) program to an executable (machine-readable) program

Interpreters: Convert a source program and execute it at the same time.(Line by line execution)

Hybrid Compiler

- Virtual Machines (e.g., Java)
- Linking executable at runtime
- Java compiler (Just-in-time compiler)



Types of Compiler

Cross- Compiler-runs on a Windows 7 PC but generates code that runs on Android smartphone .

De-Compiler

- **LL to HLL**

Tanscompiler / Source-Source compiler

- **Pascal to C**

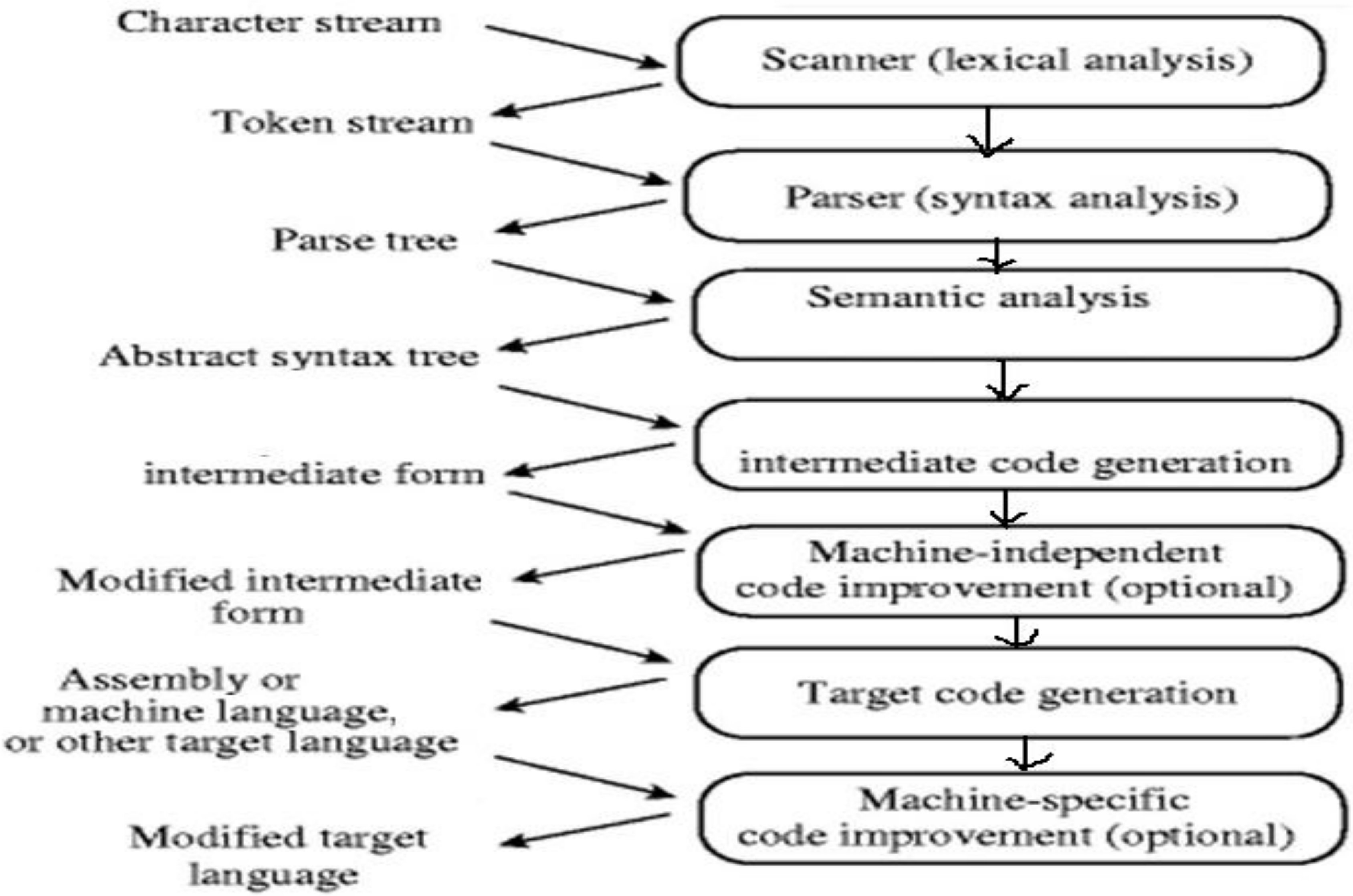
Incremental Compiler

- **Recompile only the portions modified**

Phases of Compiler

The Structure of a Compiler

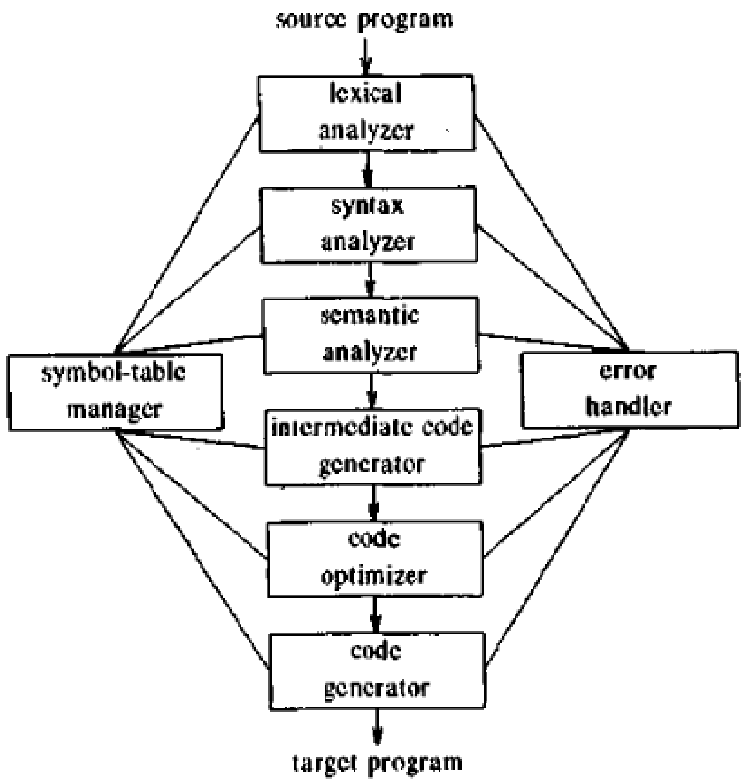
- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate Code Generator
- Machine –Independent Code Optimizer
- Code Generator
- Machine – Dependent Code Optimizer


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Abstract syntax tree
intermediate code generation
intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Assembly or machine language, or other target language
Machine-specific code improvement (optional)
Modified target language

The Structure of a Compiler / Phases of Compiler

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate Code Generator
- Code Optimizer
- Code Generator

Phases of a Compiler



Major Parts of Compiler Operation

Compiler consists of two Parts

Analysis

(Front End)

Synthesis

(Back End)

- **Analysis :** Breaks the source program into constituent pieces and creates intermediate representation.

The analysis part can be divided into:

- Lexical Analysis / Linear analysis / Scanning
- Syntax Analysis / Hierarchical analysis
- Semantic Analysis / Type checker
- Intermediate Code Generator

- Synthesis : Generates the target program from the intermediate representation.

The synthesis part can be divided along the following phases:

- Code Optimizer
- Code Generator

Lexical Analysis

- The **Lexical Analyzer** reads the program from left-to-right and sequence of characters are grouped into **tokens**–lexical units with a collective meaning.

- The sequence of characters that gives rise to a token is called **lexeme**.

  < token-name, attribute-value >

Input :

position = initial + rate * 60

Then, the lexical analyzer will group the characters in the following tokens:

Lexical Analysis

Lexeme	Token	Attribute-value
position	ID	1
=	=	=
initial	ID	2
+	+	+
rate	ID	3
*	*	*
60	NUM	60

Lexical Analysis

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

Output:

$$id_1 = id_2 + id_3 * 60$$

Lexical Analysis

- Stream of characters is grouped into tokens
- Examples of tokens are identifiers, reserved words, integers, doubles or floats, delimiters, operators and special symbols

```
int a;
a = a + 2;
```

Lexical Analysis

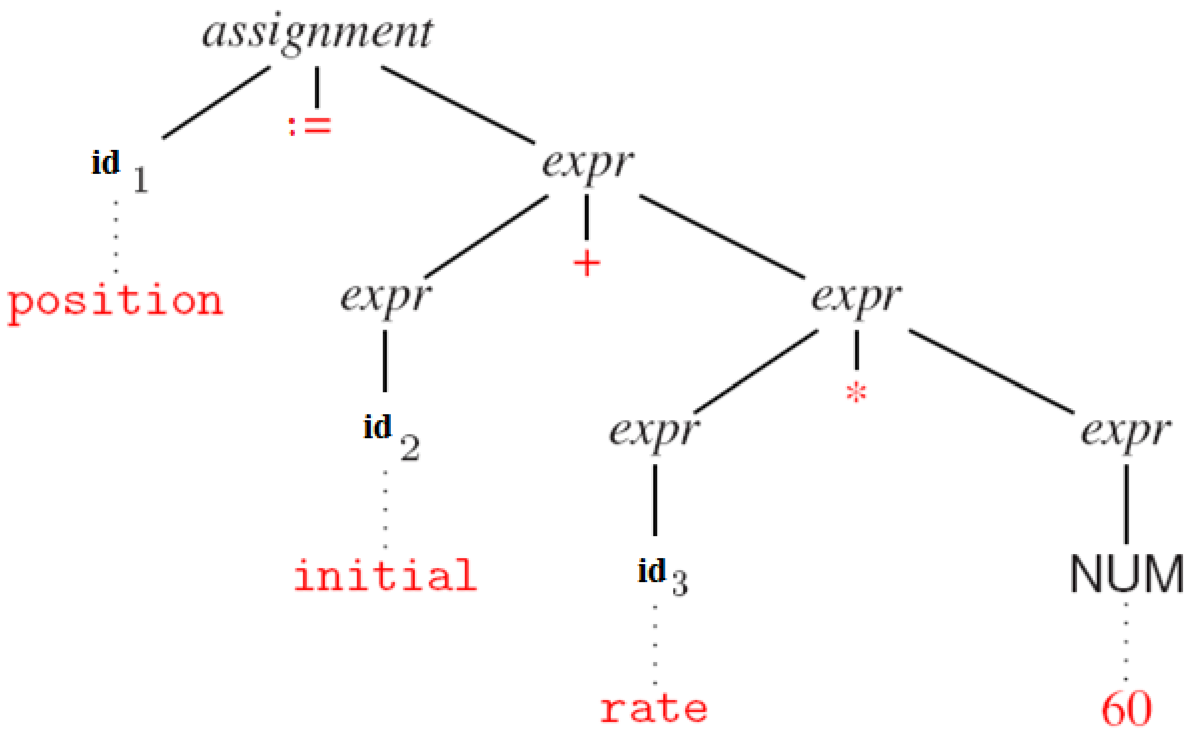
int	Keyword
a	identifier
;	special symbol
a	identifier
=	operator
a	identifier
+	operator
2	integer constant
;	special symbol

Syntax Analysis

- The **Syntactic Analysis** is also called **Parsing**.
  - (Determination of structure of source string)
- Tokens are grouped into grammatical phrases represented by a **Parse Tree** which gives a hierarchical structure to the source program.

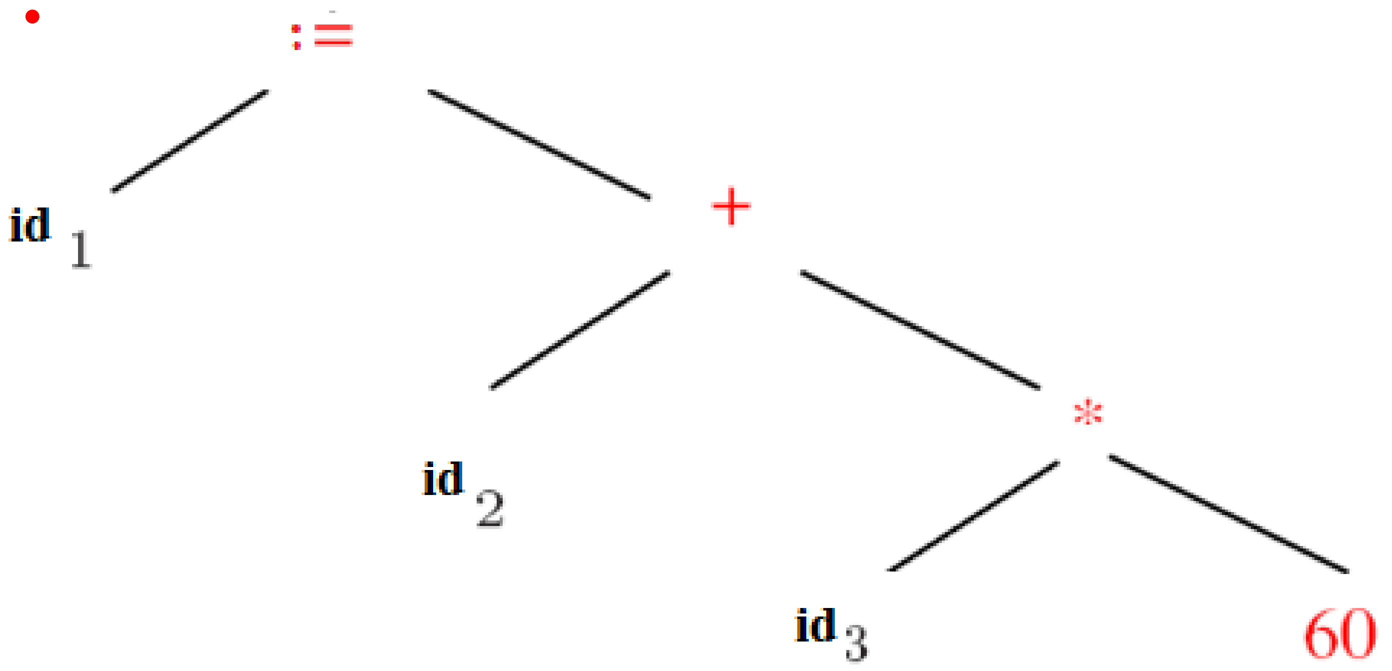
Syntax Analysis

Parse Tree :



Syntax Analysis

Syntax Tree :



Semantic Analysis

- The **Semantic Analysis** – Determination of meaning of source program
- Checks the program for semantic errors (**Type Checking**) and gathers type information for the successive phases.
- **Type Checking:** Compiler checks that each operator has matching operands

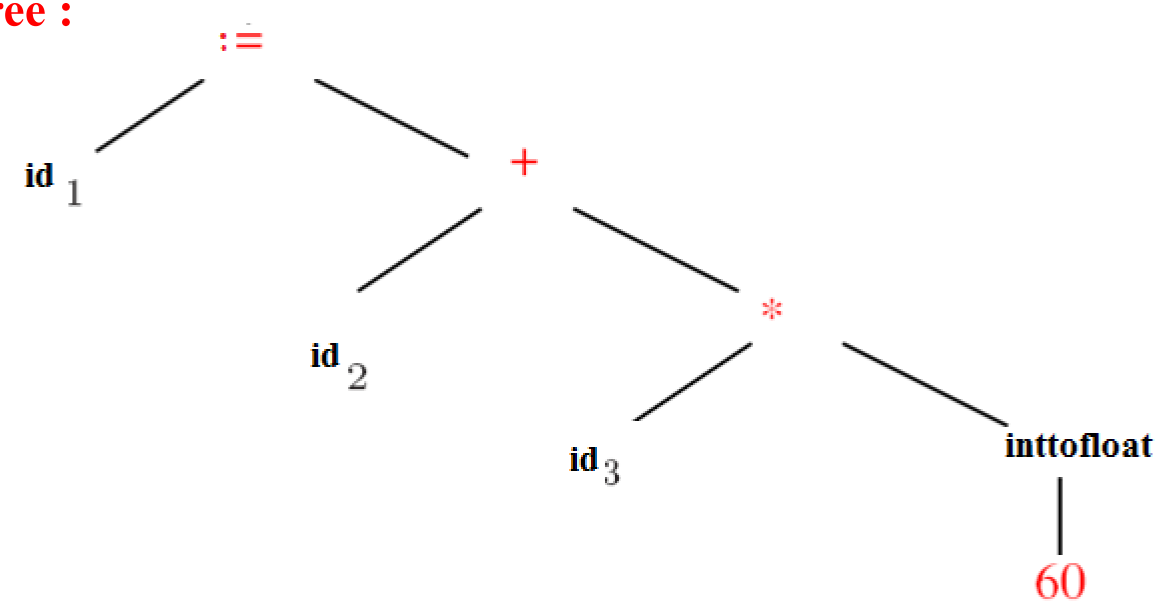
Semantic Analysis

Type Conversion (coercions)

- Eg: binary operator applied to either pair of integer or floating point
- Compiler may convert or coerce the integer into a floating point

Semantic Analysis

Abstract Syntax Tree :



Intermediate Code Generation

- The intermediate code should be easy to translate

  into the target program.
- Typical choices of intermediate code representation:
  - Annotated parse trees
  - Three Address Code (TAC)
  - Post fix
  - Bytecode, as in Java bytecode.

Intermediate Code Generation

- ***Three-address code*: Sequence of instructions with at most *three* operands such that:**
  - **There is at most one operator, in addition to the assignment.**
  - **Temporary names must be generated to compute intermediate operations.**

Output:

t_1 = inttofloat(60)

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

Intermediate Code Generation

Another example:

Input:

```
if (a <= b)

    { a = a – c; }

    c = b * c
```

Resulting TAC:

```
    t1 = a <= b
    if  t1 goto L0
     t2 = a – c
     a = t2
L0: t3 = b * c
     C = t3
```

Code Optimization

- Compiler converts the intermediate representation to another one that attempts to be smaller and faster.
- Typical optimizations:
  - Inhibit code generation for unreachable segments
  - Getting rid of unused variables
  - Eliminating multiplication by 1 and addition by 0
  - Loop optimization: e.g. removing statements not modified in the loop
  - Common sub-expression elimination

Code Optimization

Output:

$t_1 = id_3 * 60.0$

$t_2 = id_2 + t_1$

$id_1 = t_2$

Code Generation

- This phase generates the target code consisting of assembly code.

1. Memory locations are selected for each variable
2. Instructions are translated into a sequence of assembly instructions
3. Variables and intermediate results are assigned to memory registers

Code Generation

Output:

LDF R_2 , id_3

MULF R_2 , R_2 , #60.0

LDF R_1 , id_2

ADDF R_1 , R_1 , R_2

STF id_1 , R_1

Symbol Table

- An essential function of a compiler is to build the **Symbol Table** where the identifiers used in the program are recorded along with various **Attributes**.
  - Identifiers are found in lexical analysis and placed in the symbol table
  - During syntactical and semantical analysis, type and scope information are added
  - During code generation, type information is used to determine what instructions to use
  - During optimization, the "live analysis" may be kept in the symbol table

Symbol Table

SYMBOL TABLE

1	position	id
2	initial	id
3	rate	id
4		

Error Handling

- Error handling and reporting also occurs across many phases
  - Lexical analyzer reports invalid character sequences
  - Syntactic analyzer reports invalid token sequences
  - Semantic analyzer reports type and scope errors
- The compiler may be able to continue with some errors, but other errors may stop the process

Analysis of Source program

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

Lexical Analyzer
[Scanner]

Tokens

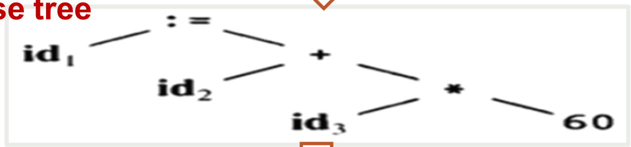
id_1 := id_2 + id_3 * 60

Syntax Analyzer
[Parser]

SYMBOL TABLE

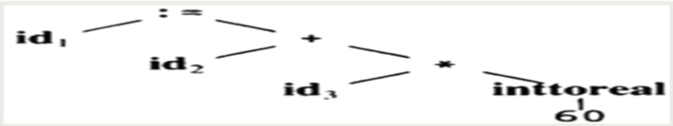
1	position	...
2	initial	...
3	rate	...
4		

Parse tree



Semantic Analyzer
[Semantic Process]

Abstract Syntax Tree



Intermediate Code Generator

Non-optimized Intermediate Code

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

Code Optimizer

Optimized Intermediate Code

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

Code Generator

Target machine code

LDF R_2 , id_3
MULF R_2 , R_2 , #60.0
LDF R_1 , id_2
ADDF R_1 , R_1 , R_2
STF id_1 , R_1

The Grouping of Phases

Compiler *front* and *back ends*:

- Front end: *analysis* (*machine independent and Source language dependent*)
- Back end: *synthesis* (*machine dependent and Source language independent*)

Compiler *passes:*

- A collection of phases is done only once (*single pass*) or multiple times (*multi pass*)
  - Single pass: usually requires everything to be defined before being used in source program
  - Multi pass: compiler may have to keep entire program representation in memory

Other Tools that Use the Analysis-Synthesis Model

Applications Related to Compilers

- Compiler Relatives
  - Interpreters
  - Structure Editors
  - Pretty Printers
  - Static Checkers
  - Debuggers
- Other Applications
  - Text Formatters
  - Silicon Compilers
  - Query Interpreters

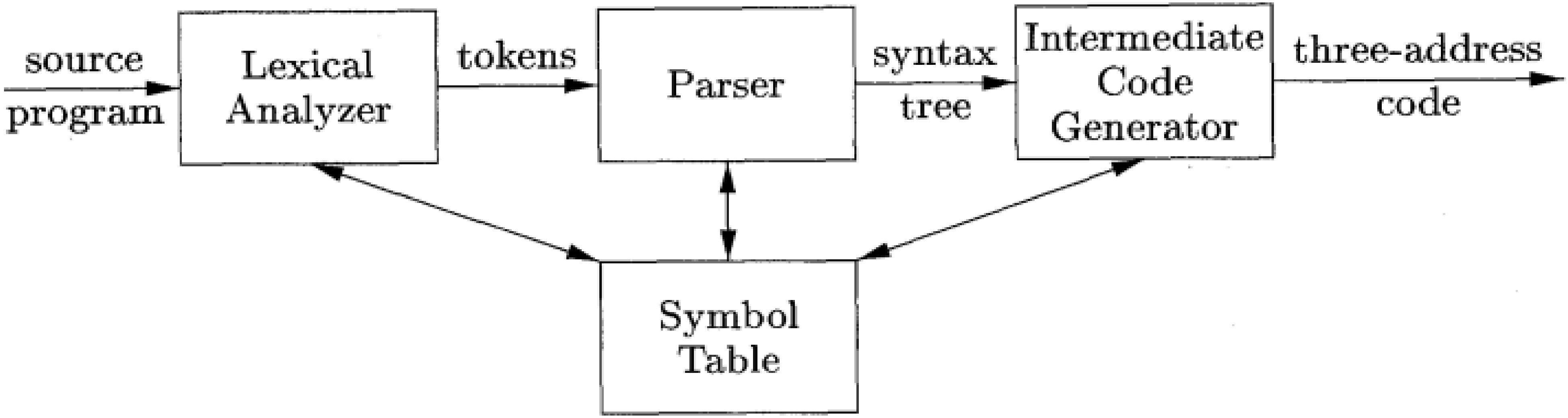
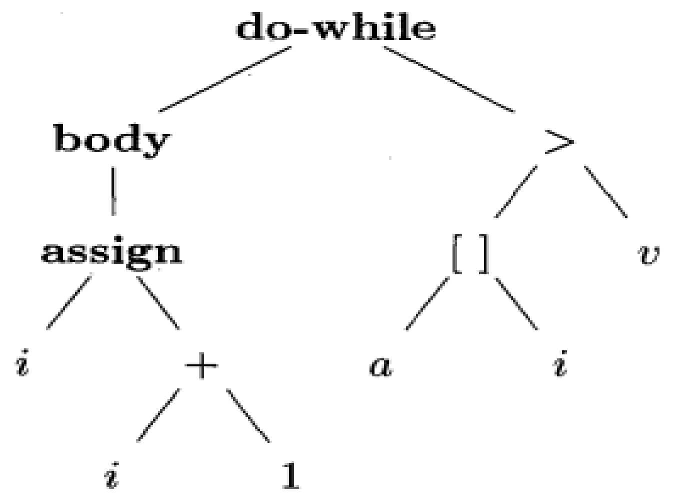


Figure 2.3: A model of a compiler front end



(a)

abstract syntax tree in Fig. 2.4(a) represents an entire do-while loop.

```
do i = i + 1; while ( a[i] < v);
```

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

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

Compiler-Construction Tools

Software development tools are available to implement one or more compiler phases

- **Scanner generators** - Generate Lexical Analysis
- **Parser generators** - Generate Syntax Analysis
- **Syntax-directed translation engines** - Intermediate Code generation
- **Automatic code generators** - Code Generation
- **Data-flow engines** - Code Optimization

Compiler Construction Tools

- Front End (Analysis)
  - Scanner Generators: Lex
  - Parser Generators: Yacc
  - Syntax-Directed Translation Engines
- Back End (Synthesis)
  - Automatic Code Generators
  - Peephole Optimizer Construction Tools

Cousins of Compilers

- Preprocessors
- Compiler
- Assemblers
- Linkers
- Loaders

Cousins of Compilers

- **Preprocessors**
  - It converts HLL into pure HLL
  - It includes all the header files
  - It also expand shorthands, called macros, into source language statements
  - It deals with macro-processing, augmentation, file inclusion, language extension, etc.
- **Compiler**
  - It produces an assembly-language program

Preprocessors

- Perform some preliminary processing on a source module.
  - definitions and macros
    - #define
  - file inclusion
    - #include
  - conditional compilation
    - #ifdef
  - line numbering
    - #line

Cousins of Compilers

- **Assembler**
  - **It is software which converts assembly code into object code, is called assembler.**

Assemblers

- Typically accomplished in 2 passes.
  - Pass 1: Stores all of the identifiers representing tokens in a table.
  - Pass 2: Translates the instructions and data into bits for the machine code.
- Produces relocatable code.

Cousins of Compilers

- **Linker**
  - **It resolves external memory addresses, where the code in one file may refer to a location in another file.**
- **Loader**
  - **It puts the executable object files into memory for execution**

Linkers and Loaders

- Linker
  - Produces an executable file.
  - Resolves external references.
  - Includes appropriate libraries.
- Loader
  - Creates a process from the executable.
  - Loads the process (or a portion of it) into main memory.
  - Produces absolute machine code.

Cousins of Compilers

Linking and loading

It has four functions

- **Allocation:**

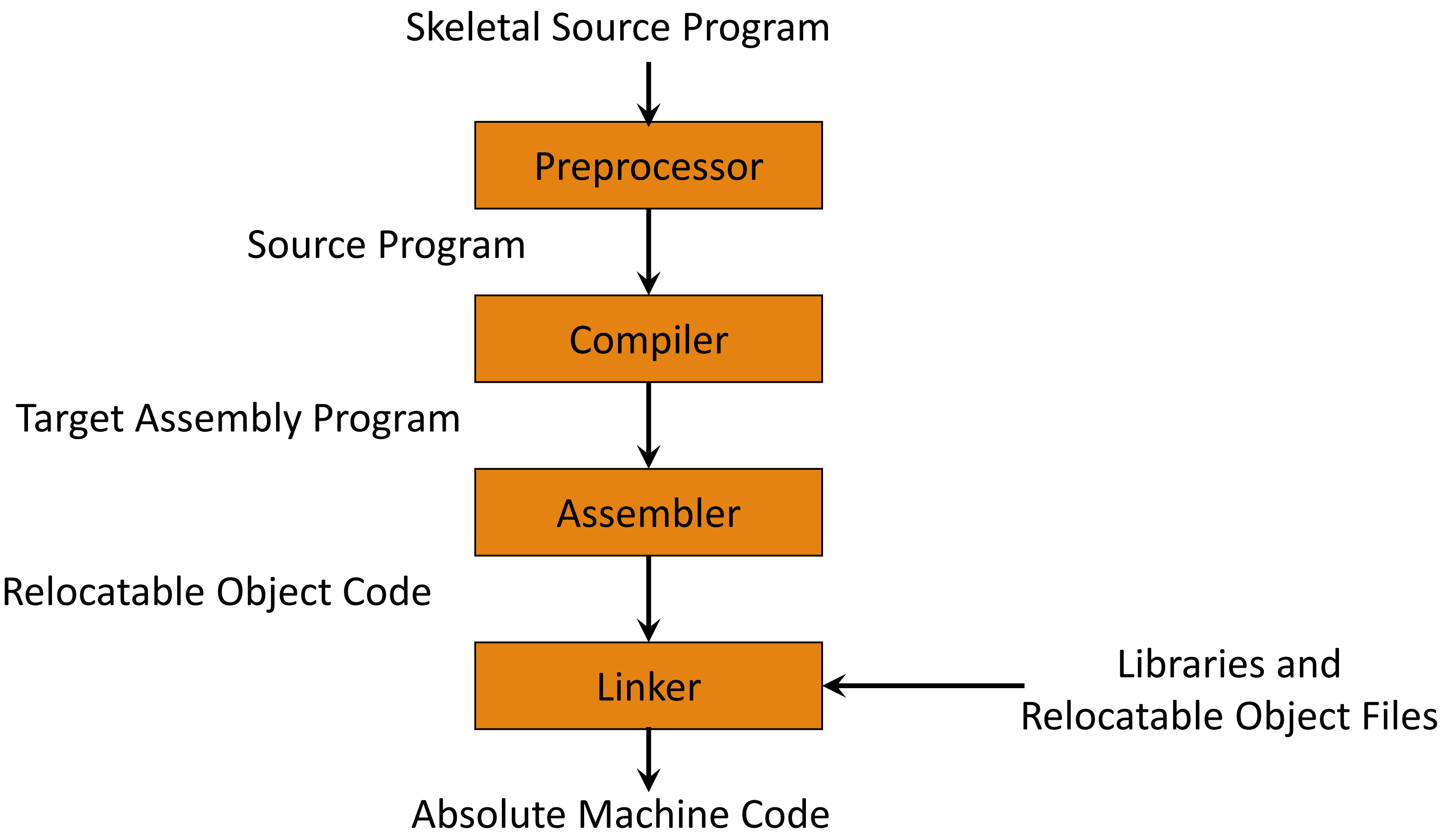
  It means get the memory portions from operating system and storing the object data.
- **Relocation:**

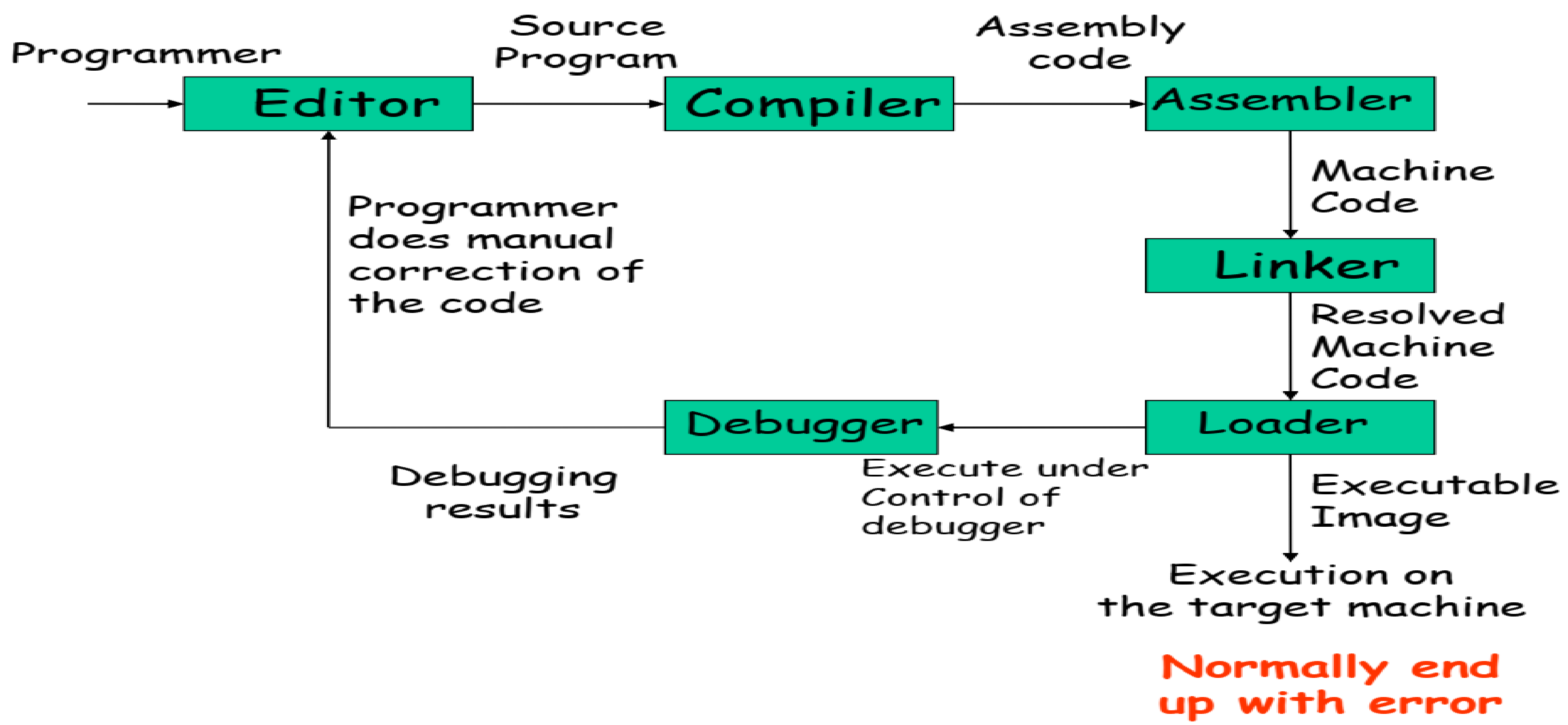
  It maps the relative address to the physical address and relocating the object code.
- **Linker:**

  It combines all the executable object module to pre single executable file.
- **Loader:**

  It loads the executable file into permanent storage.

A Language processing System




Programmer
Editor
Source
Program
Compiler
Assembly
code
Assembler
Machine
Code
Linker
Resolved
Machine
Code
Loader
Executable
Image
Execution on
the target machine
Normally end
up with error
Debugger
Execute under
Control of
debugger
Debugging
results
Programmer
does manual
correction of
the code

Lexical Analysis

Role of Lexical analyzer

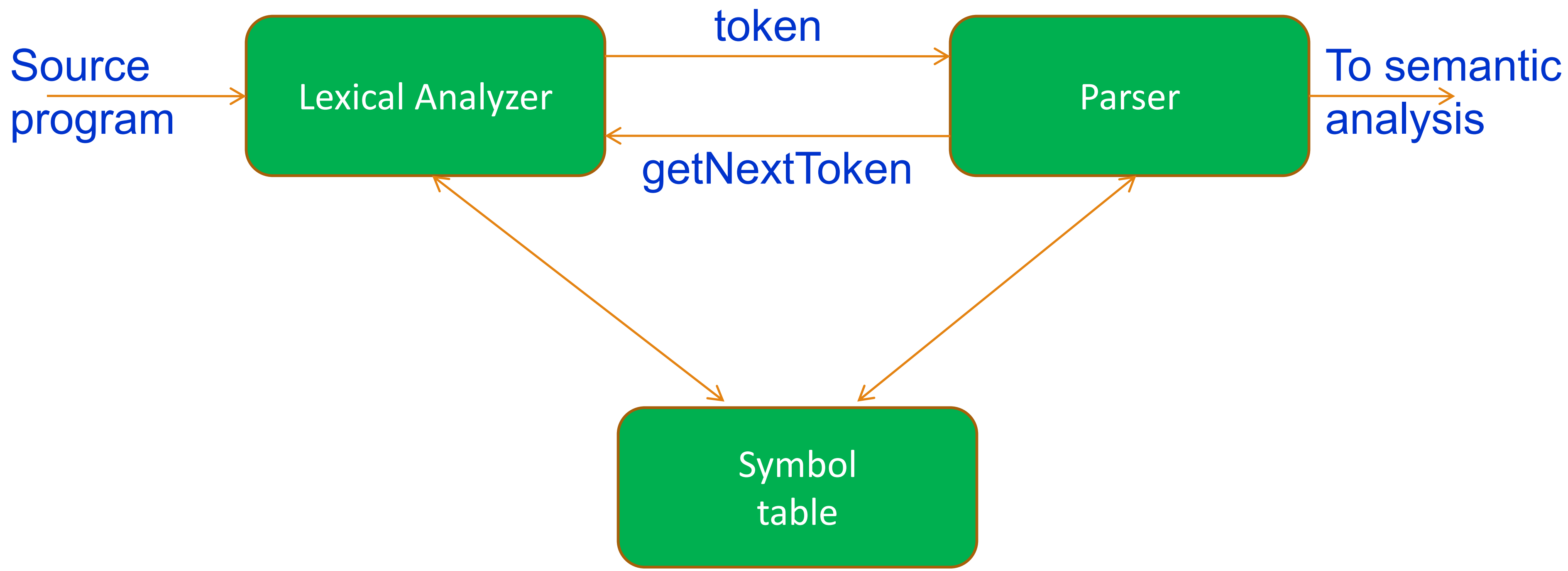
Tokens, Patterns and Lexemes

Input Buffering

Specification of Token

Recognition of Token

The role of lexical analyzer



Tokens, Patterns and Lexemes

A **token** is a pair a token name and an optional token value

A **pattern** is a description of the form that the lexemes of a token may take

A **lexeme** is a sequence of characters in the source program that matches the pattern for a token

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 letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “ sorrounded by “	“core dumped”

Attributes for tokens

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>

Example : E=M*C**2

Lexemes	Tokens
E	<id, 1> or id1
=	<assignment>
M	<id, 2> or id2
*	<*>
C	<id, 3> or id3
**	<**> or <exp-op>
2	<2>

SYMBOL TABLE			
1	Id	E	……… …
2	Id	M	
3	id	C	

Lexical errors

Some errors are out of power of lexical analyzer to recognize:

- fi (a == f(x)) …

However it may be able to recognize errors like:

- d = 2r

Such errors are recognized when no pattern for tokens matches a character sequence

Lexical errors

Commonly generated lexical errors are

- Spelling error
- Unmatched Error
- Appearance of illegal character
- Exceeding length of the identifier

Error recovery

Panic mode: successive characters are ignored until we reach to a well formed token

Delete one character from the remaining input

Insert a missing character into the remaining input

Replace a character by another character

Transpose two adjacent characters

Input buffering

Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return

- In C language: we need to look after -, = or < to decide what token to return
- In Fortran: DO 5 I = 1.25

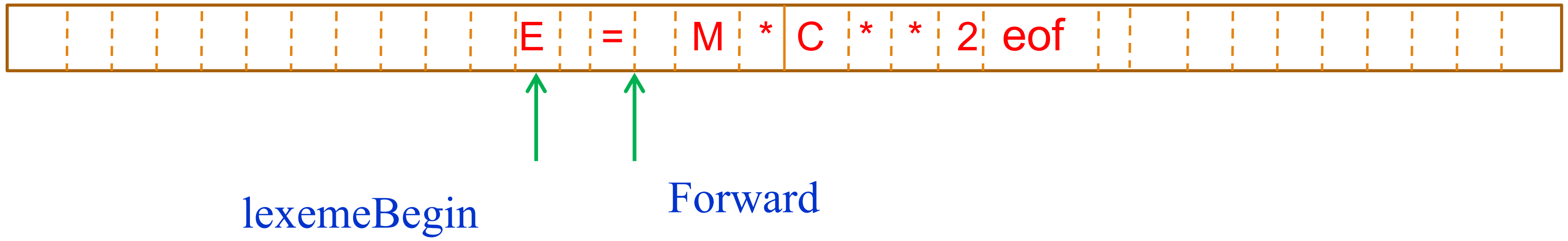
We need to introduce a two buffer scheme to handle large look-aheads safely

Input buffering

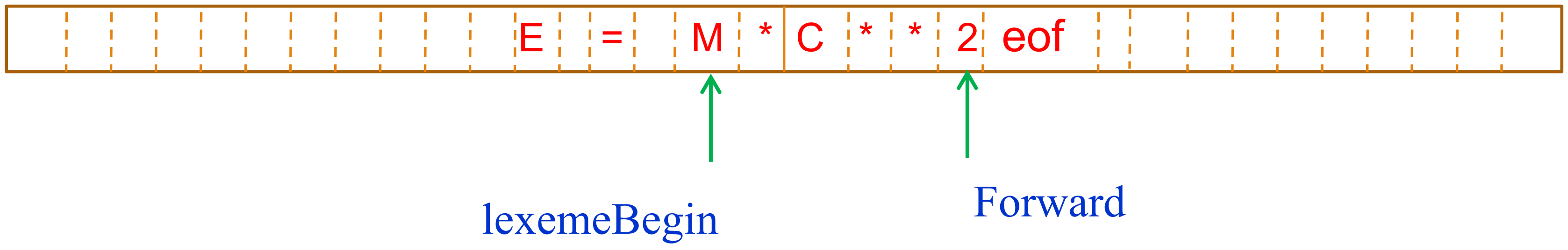
- Two buffer schemes
  - Buffer pair
  - Sentinels
- Two pointers to the input are maintained:
  - LexemeBegin pointer
    - Marks the beginning of the current lexeme
  - Forward pointer
    - Scans ahead until a pattern match is found

Buffer Pair

- Each buffer is of the same size N
- N is the size of a disk block, eg. 4096 bytes
- eof marks the end of the source file



Buffer Pair



Input buffering

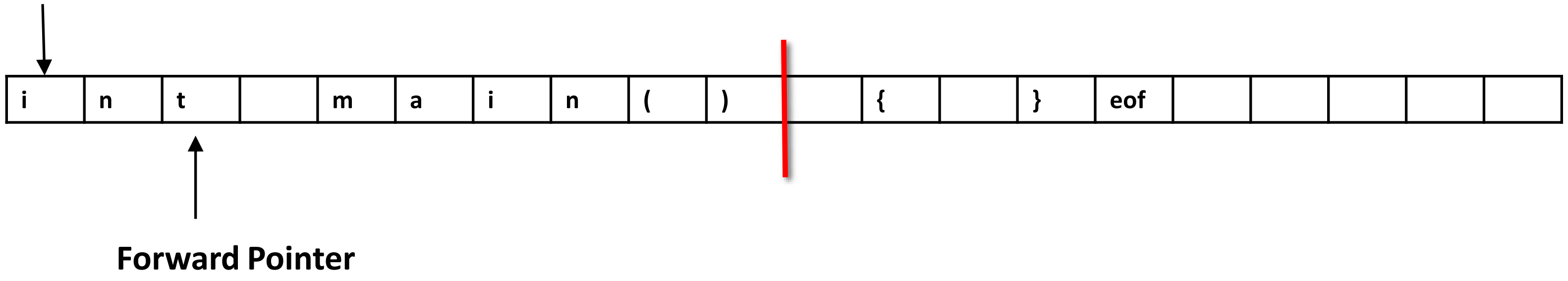
Lexical Analyzer there are two pointers are used:

- Lexeme begin Pointer
- Forward Pointer
- Example: int main()

```
int main()
{  …….
   ……..
}
```

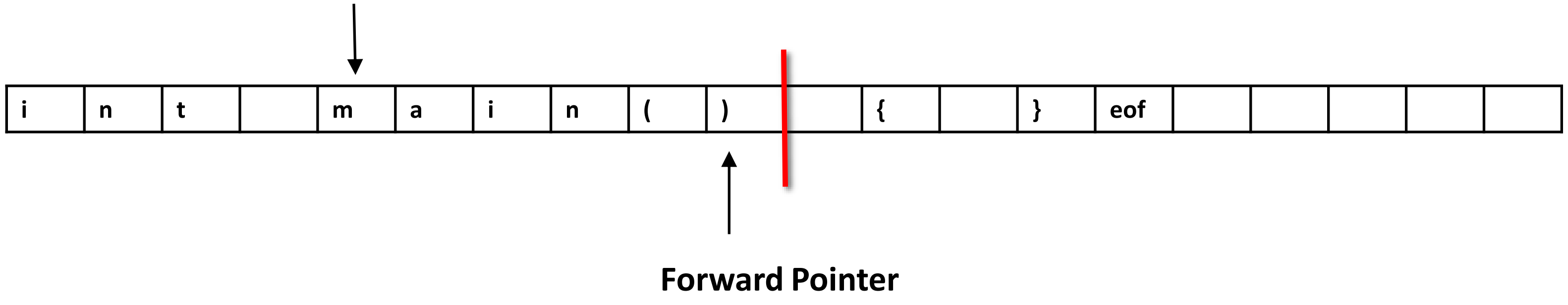
Buffer Pairs

Lexeme Begin Pointer

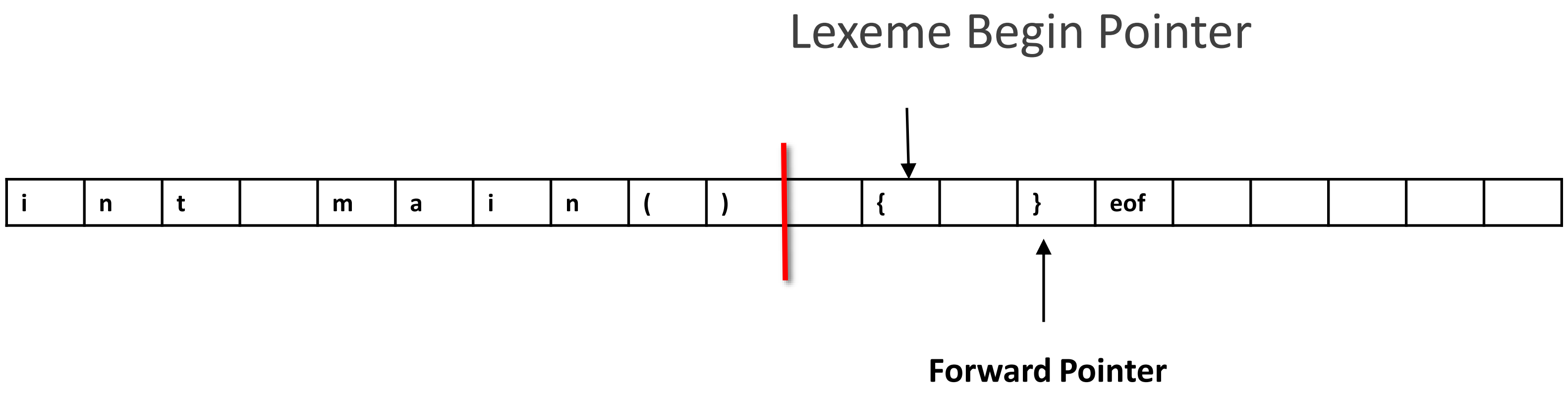


Buffer Pairs

Lexeme Begin Pointer



Buffer Pairs

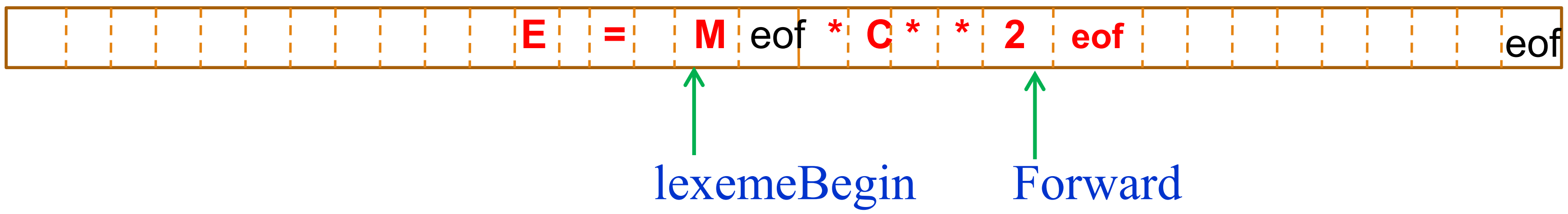


Sentinels

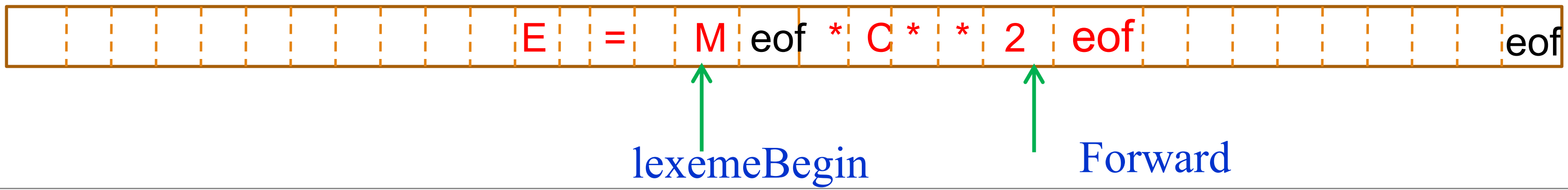
Buffer test

- one for the end of the buffer
- One to determine what character is read

Sentinels is a special character that cannot be part of the source program and natural choice is the character eof.



Sentinels




```
Switch (*forward++) {
 case eof:
        if (forward is at end of first buffer) {
                reload second buffer;
                forward = beginning of second buffer;
        }
        else if {forward is at end of second buffer) {
                reload first buffer;\
                forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
                terminate lexical analysis;
        break;
 cases for the other characters;
}
```

Specifications of Tokens

- Strings and Languages
- Operations on Languages
- Regular Expressions
- Regular Definitions
- Extensions of Regular Expressions

Strings and languages

- Alphabet
  - Is any finite set of symbols
    - Eg: symbols are letters, digits and punctuation
    - Eg: binary alphabet {0, 1}
- String
  - Is a finite sequence of symbols formed from that alphabet
  - The length of a string s, |s|, is the number of occurrences of symbols in s
    - Eg: banana, is a string of length six.
    - Eg: **ε** , the string of length zero
- Languages
  - Language is any countable set of strings over some fixed alphabet.
    - Eg: ∅ , the empty set or {**ε** }

Terms of string

1. Pre-fix of string s – removing zero or more symbols from the end of s
2. Suffix of string s – removing zero or more symbols from the beginning of s
3. Substring of s – deleting any prefix and any suffix from s
4. Proper prefixes, suffixes and substring of s – string that are not **ε** or not equal to s itself.
5. Subsequence of s – deleting zero or more not necessarily consecutive positions of s

Operations on Languages

Operations on languages

- Union
- Concatenation
- Closure (kleene)

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions of operations on languages

Example:

- L is Letter
- D is Digit
- L = { a, B, C ,…. , Z, a, b , c, …..z }
- D = { 0, 1, 2, …. 9 }

L U D

LD

L4

L*

L(LUD)*

D^+

Regular Expressions

Regular expressions used to specify tokens of a programming language.

Example: RE for Identifier

- Letter_(letter_ | digit)*

Each regular expression denotes a language.

A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
Basis	
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
Induction:	
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$
$(r)^+ = (r)(r)^*$	
$(r)? = (r) \mid \varepsilon$	

Regular Expressions

We may remove parentheses by using precedence rules.

- * highest
- concatenation next
- | lowest

$ab^*|c$ means $(a(b)^*)|(c)$

Ex:

- $\Sigma = \{0,1\}$
- $0|1 \Rightarrow \{0,1\}$
- $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
- $0^* \Rightarrow \{\varepsilon, 0, 00, 000, 0000, ...\}$
- $(0|1)^*$ => all strings with 0 and 1, including the empty string
- $0|0^*1 \Rightarrow \{0, 1, 01, 001, 0001, ...\}$

Regular Expressions

LAW	DESCRIPTION
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s \mid t) = rs \mid rt;\ (s \mid t)r = sr \mid tr$	Concatenation distributes over $\mid$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r \mid \epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Algebraic laws for regular expressions

Regular Definitions

Regular definition Give names to regular expressions - Use these names as symbols to define other regular expressions.

$$\begin{array}{ccc} d_1 & \to & r_1 \\ d_2 & \to & r_2 \\ & \cdots & \\ d_n & \to & r_n \end{array}$$

Ex: Identifiers in Pascal

letter → A | B | … | Z | a | b | … | z

digit → 0 | 1 | … | 9

id → letter (letter | digit) *

Regular Definitions

Ex: Unsigned numbers in Pascal

$$
\begin{aligned}
\text{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \\
\text{digits} &\rightarrow \text{digit}^{+} \\
\text{opt-fraction} &\rightarrow (\ .\ \text{digits}\ )\ ? \\
\text{opt-exponent} &\rightarrow (\ \text{E}\ (+|-)?\ \text{digits}\ )\ ? \\
\text{unsigned-num} &\rightarrow \text{digits opt-fraction opt-exponent}
\end{aligned}
$$

Eg: 5280, 0.01234, 6.336E4 or 1.89E-4

Extensions of Regular Expressions

- Zero or more instances (*)
- One or more instances (+)
- Zero or one instances (?)
- Character classes
  - Eg: [abc] - a|b|c
  - [a-z] - a|b|c|…|z

Recognition of tokens

Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| Ɛ

expr -> term **relop** term
| term

term -> **id**
| **number**

Recognition of tokens

The next step is to formalize the patterns:

digit -> [0-9]

Digits -> digit+

number -> digit(.digits)? (E[+-]? Digit)?

letter -> [A-Za-z]

id -> letter (letter|digit)*

If -> if

Then -> then

Else -> else

Relop -> < | > | <= | >= | = | <>

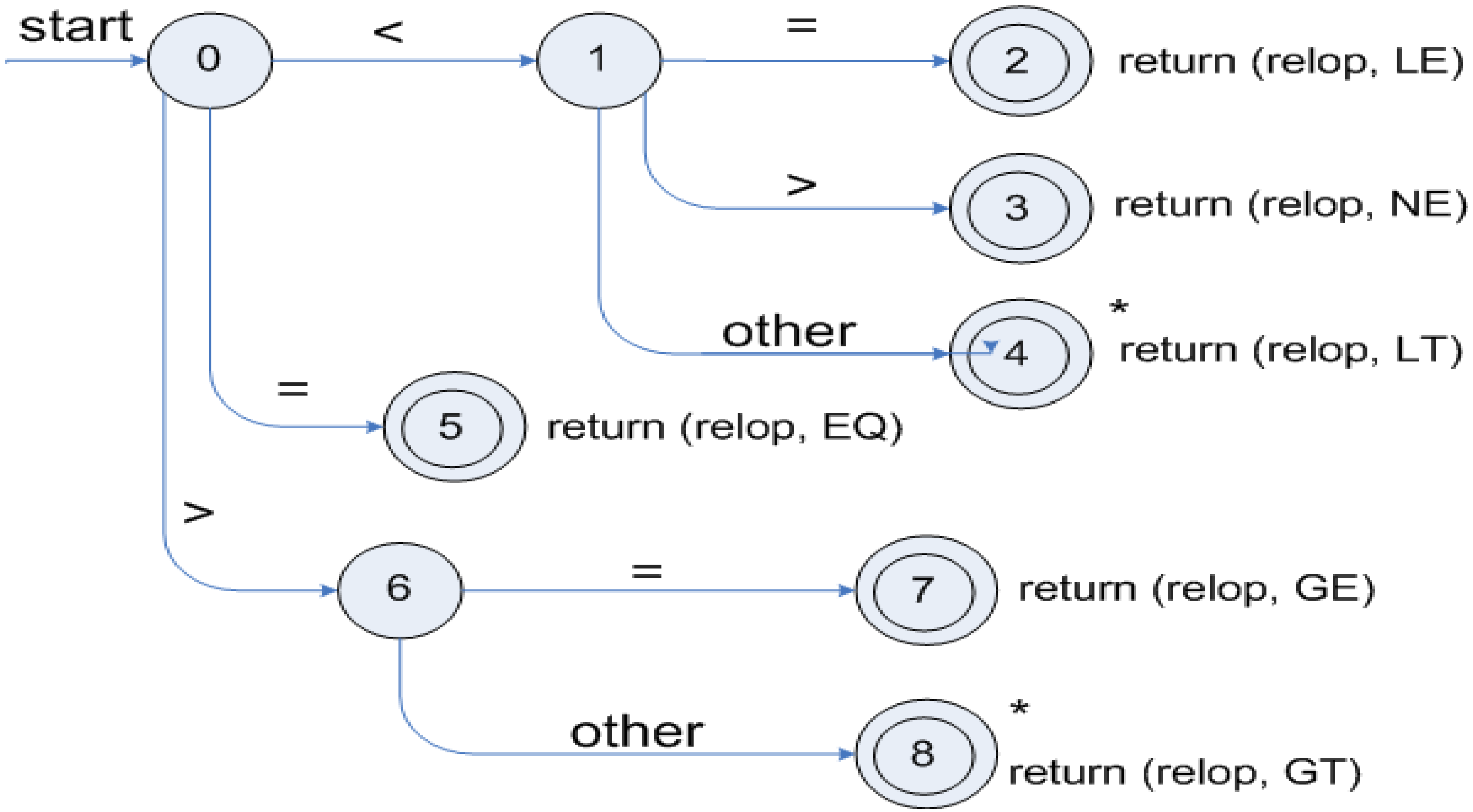
We also need to handle whitespaces:

ws -> (blank | tab | newline)+

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

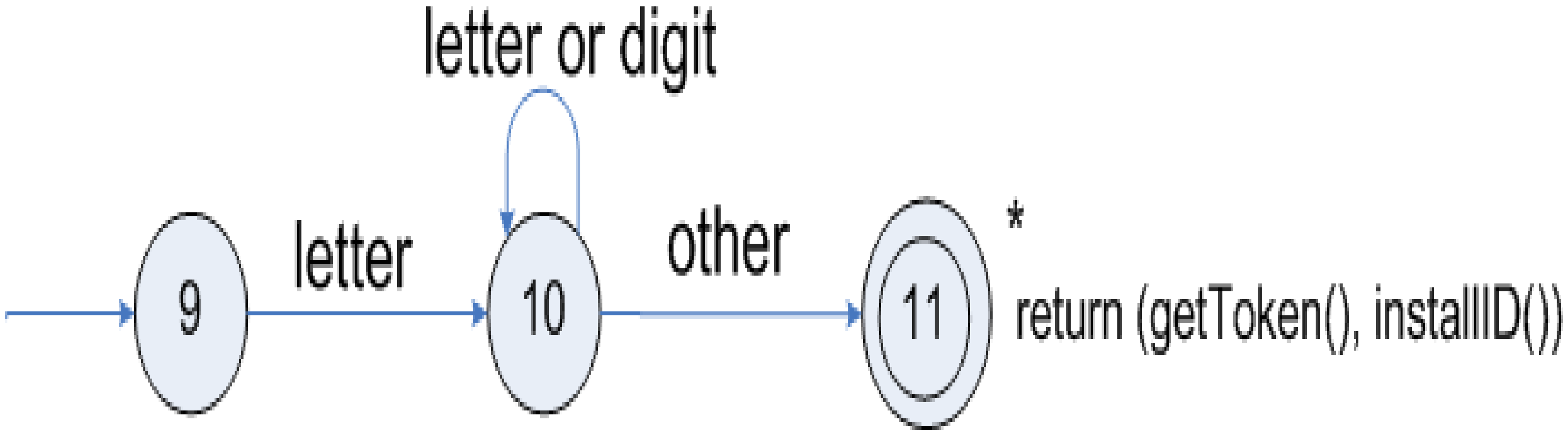
Tokens, their patterns, and attribute values

Transition diagram for relop



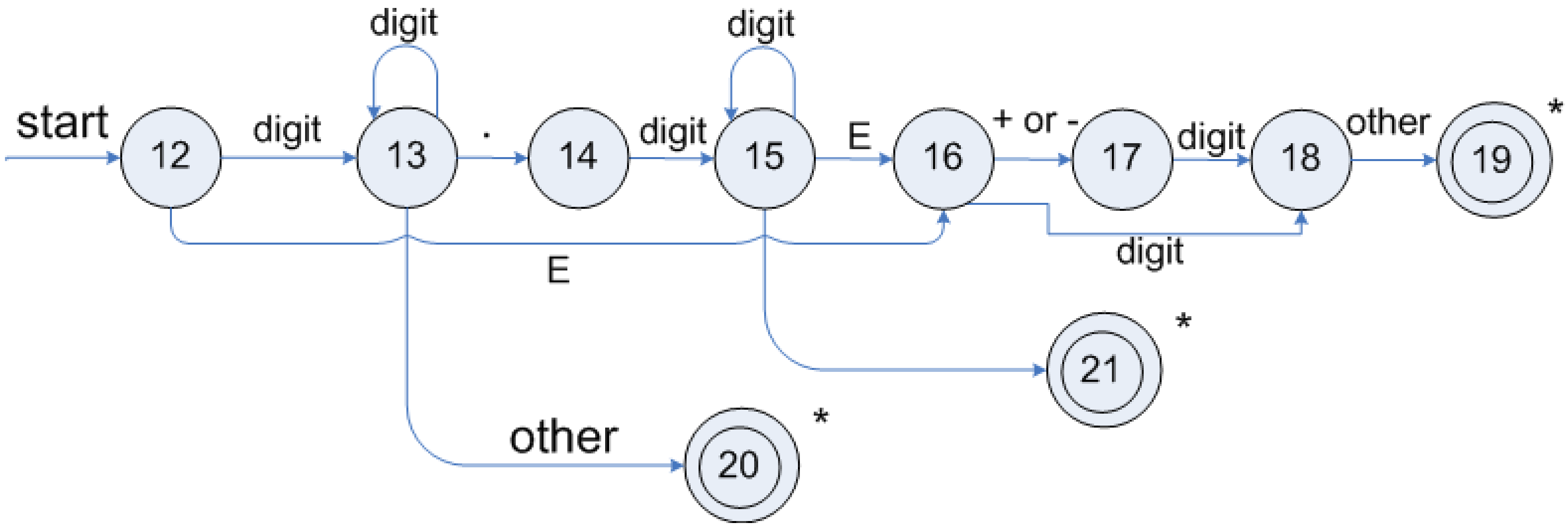
Reserved words and identifiers

id → letter (letter | digit) *



Unsigned numbers

- ***number*** **-> digit(.digits)? (E[+-]? Digit)?**



Transition diagram for whitespace

- ***delimit*** **-> (blank | tab | newline)**
- ***ws*** **-> delimit +**