4BCS604: COMPILER DESIGN

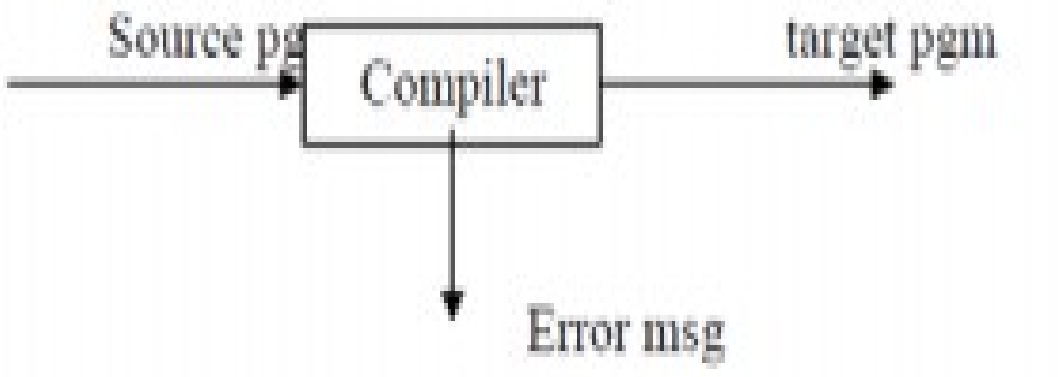
Module - I

Introduction to Lexical Analysis : -

Language processors, the structure of a compiler, the science of building a compiler, programming language basics. Lexical Analysis: the role of the lexical analyser, input buffering, recognition of tokens

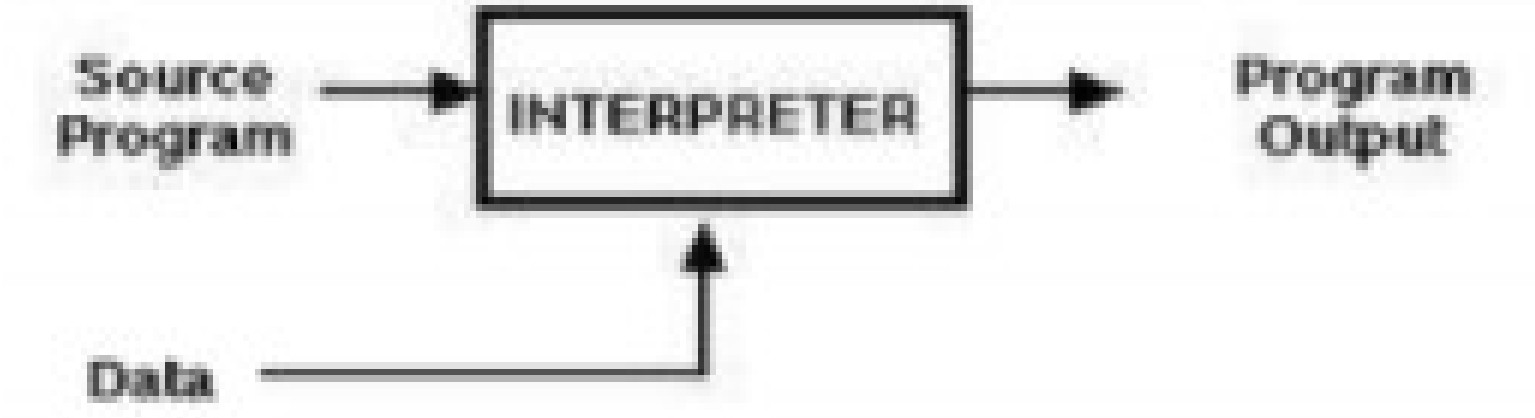
Compiler

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



Interpreter

An interpreter is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages.



Difference between Compiler and Interpreter

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

Analysis-Synthesis model of compilation

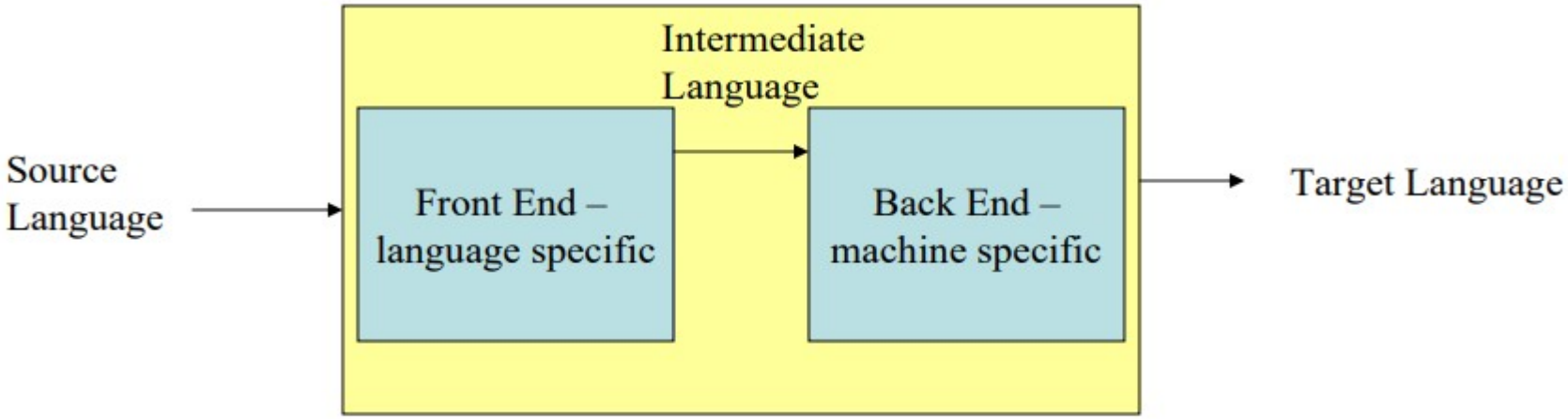
Two parts

– Analysis

- Breaks up the source program into constituents

– Synthesis

- Constructs the target program



Other Tools that Use the Analysis-Synthesis Model

- *Structure Editors* (syntax highlighting)
- *Pretty printers* (e.g. Doxygen)
- *Static checkers* (e.g. Lint and Splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)

Analysis of the Source Program:

- In compiling, analysis consists of three phases:
    - Linear Analysis:
    - Hierarchical Analysis:
    - Semantic Analysis:

Linear Analysis:

- In which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of

characters having a collective meaning.

- In a compiler, linear analysis is called lexical analysis or scanning.
- For example, in lexical analysis the characters in the assignment statement

  Position: = initial + rate * 60

- The blanks separating the characters of these tokens would normally be eliminated during the lexical analysis.

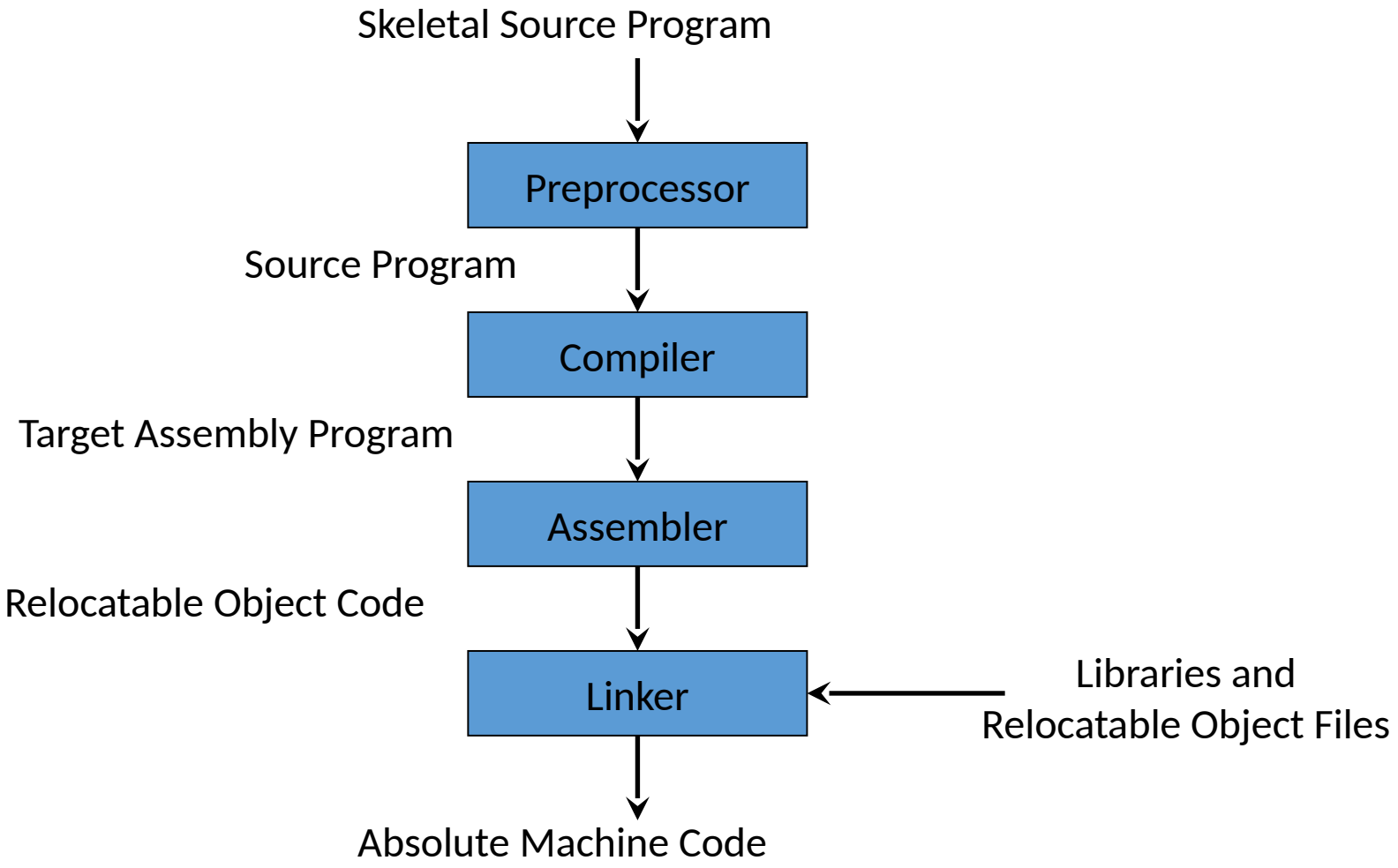
Hierarchical Analysis:

- In which characters or tokens are grouped hierarchically into nested collections with collective meaning.
- Hierarchical analysis is called parsing or syntax analysis.
- It involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output.
- The grammatical phrases of the source program are represented by a parse tree.

Semantic Analysis:

- In which certain checks are performed to ensure that the components of a program fit together meaningfully.
- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent codegeneration phase.
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operand of expressions and statements.
- An important component of semantic analysis is type checking.

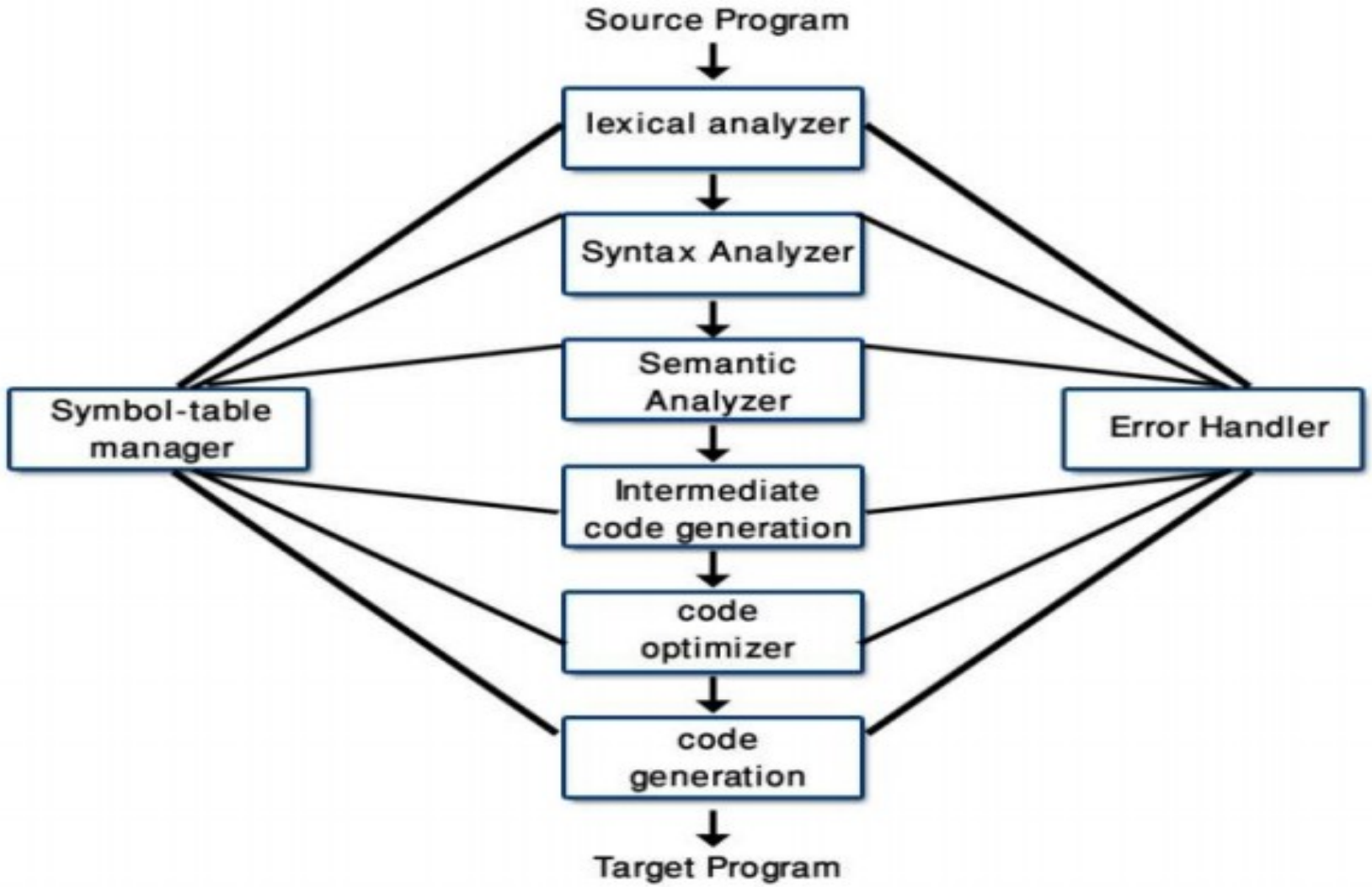
Language Processors



The Phases of a Compiler

- A compiler operates in phases.

- Each of which transforms the source program from one representation to another.


Source Program
lexical analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate code generation
code optimizer
code generation
Target Program
Symbol-table manager
Error Handler

Lexical Analysis:

- In which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of

characters having a collective meaning.

- In a compiler, linear analysis is called lexical analysis or scanning.
- For example, in lexical analysis the characters in the assignment statement

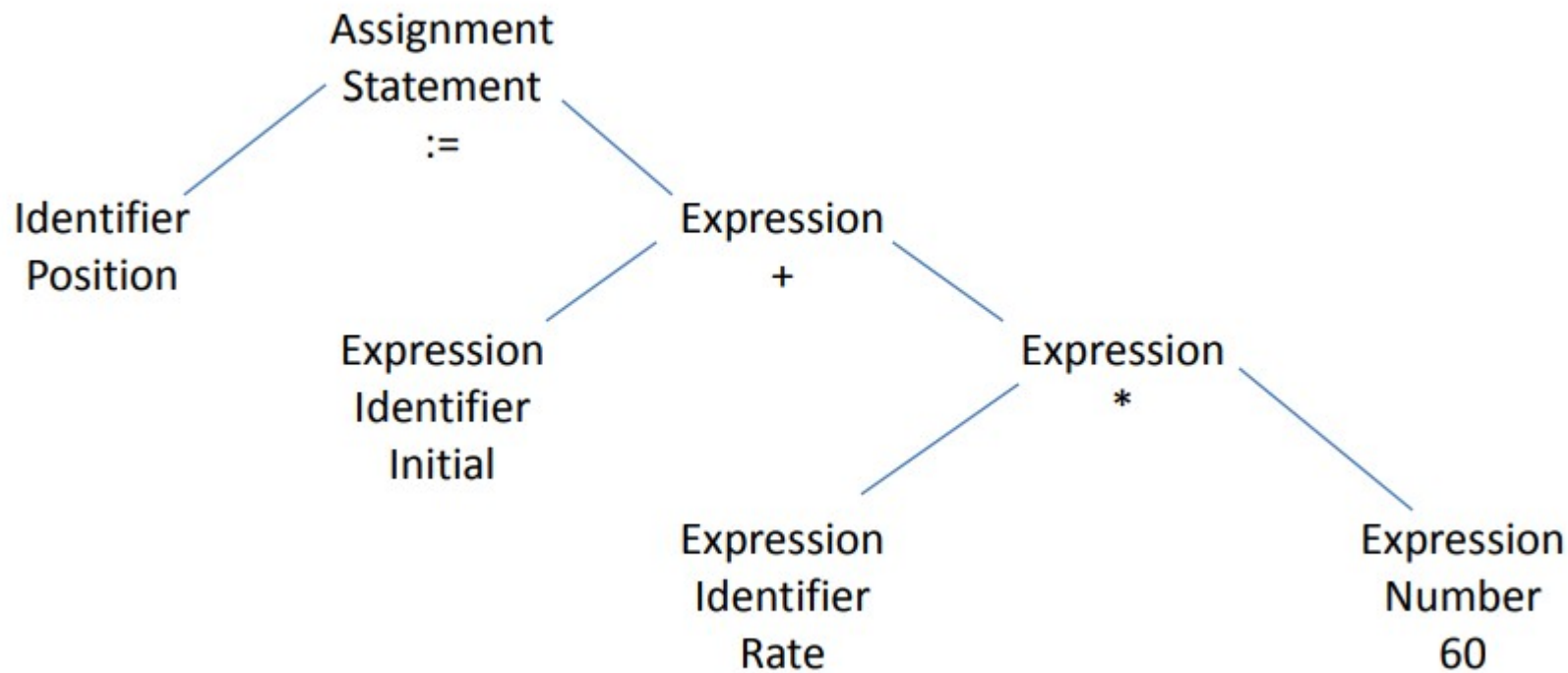
Position: = initial + rate * 60

- The identifier, position.
- The assignment symbol :=
- The identifier initial.
- The plus sign.
- The identifier rate.
- The multiplication sign.
- The number 60.

The blanks separating the characters of these tokens would normally be eliminated during the lexical analysis.

Syntax Analysis:

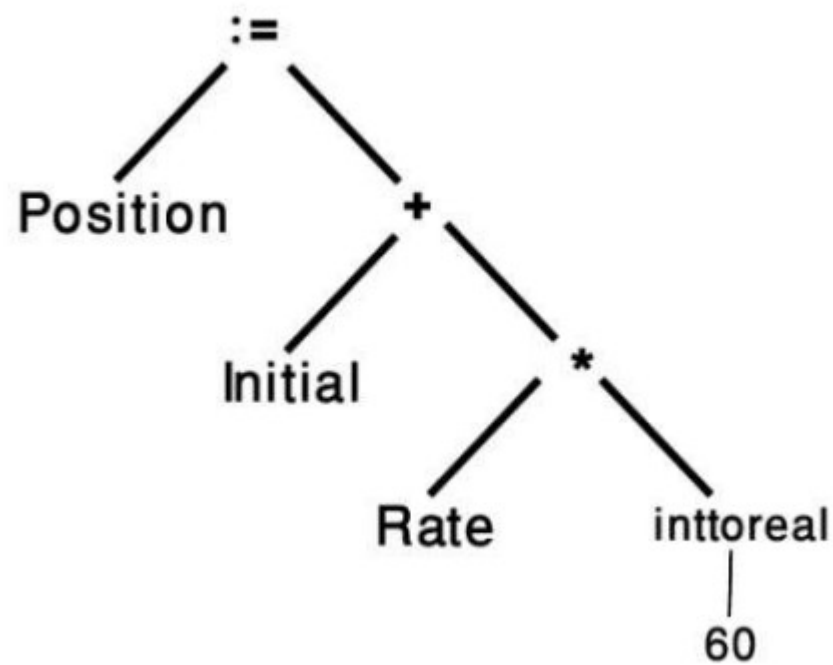
- In which characters or tokens are grouped hierarchically into nested collections with collective meaning.
- Hierarchical analysis is called parsing or syntax analysis
- It involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output
- The grammatical phrases of the source program are represented by a parse tree.


Assignment
Statement
:=
Identifier
Position
Expression
+
Expression
Identifier
Initial
Expression
*
Expression
Identifier
Rate
Expression
Number
60

- The hierarchical structure of a program is usually expressed by recursive rules.
- For example, we might have the following rules, as part of the definition of expression:
- Any identifier is an expression.
- Any number is an expression
- If expression1 and expression2 are expressions,
- then so are
  - Expression1 + expression2
  - Expression1 * expression2
  - (Expression1)

Semantic Analysis:

- In which certain checks are performed to ensure that the components of a program fit together meaningfully.
- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent codegeneration phase.
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the perators and operand of expressions and statements.
- An important component of semantic analysis is type checking.


:=
Position
+
Initial
*
Rate
inttoreal
60

Intermediate Code Generation:

- After Syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program.

- This intermediate representation should have
- two important properties;
- – it should be easy to produce,
- – easy to translate into the target program.

- We consider an intermediate form called “three-address code,”
- which is like the assembly language for a machine in which every memory location can act like a register.
- Three-address code consists of a sequence of instructions, each of which has at most three operands.

- Temp1 := inttoreal (60)
- Temp2 := id3 * temp1
- Temp3 := id2 + temp2
- id1 := temp3

Code Optimization:

- The code optimization phase attempts to improve the intermediate code, so that fasterrunning machine code will result.

- Temp1 := id3 * 60.0
- id := id2 + temp1

Code Generation

- The final phase of the compiler is the generation of target code
- consisting normally of relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program.
- Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

- MOVF id3, r2
- MULF #60.0, r2
- MOVF id2, r1
- ADDF r2, r1
- MOVF r1, id1

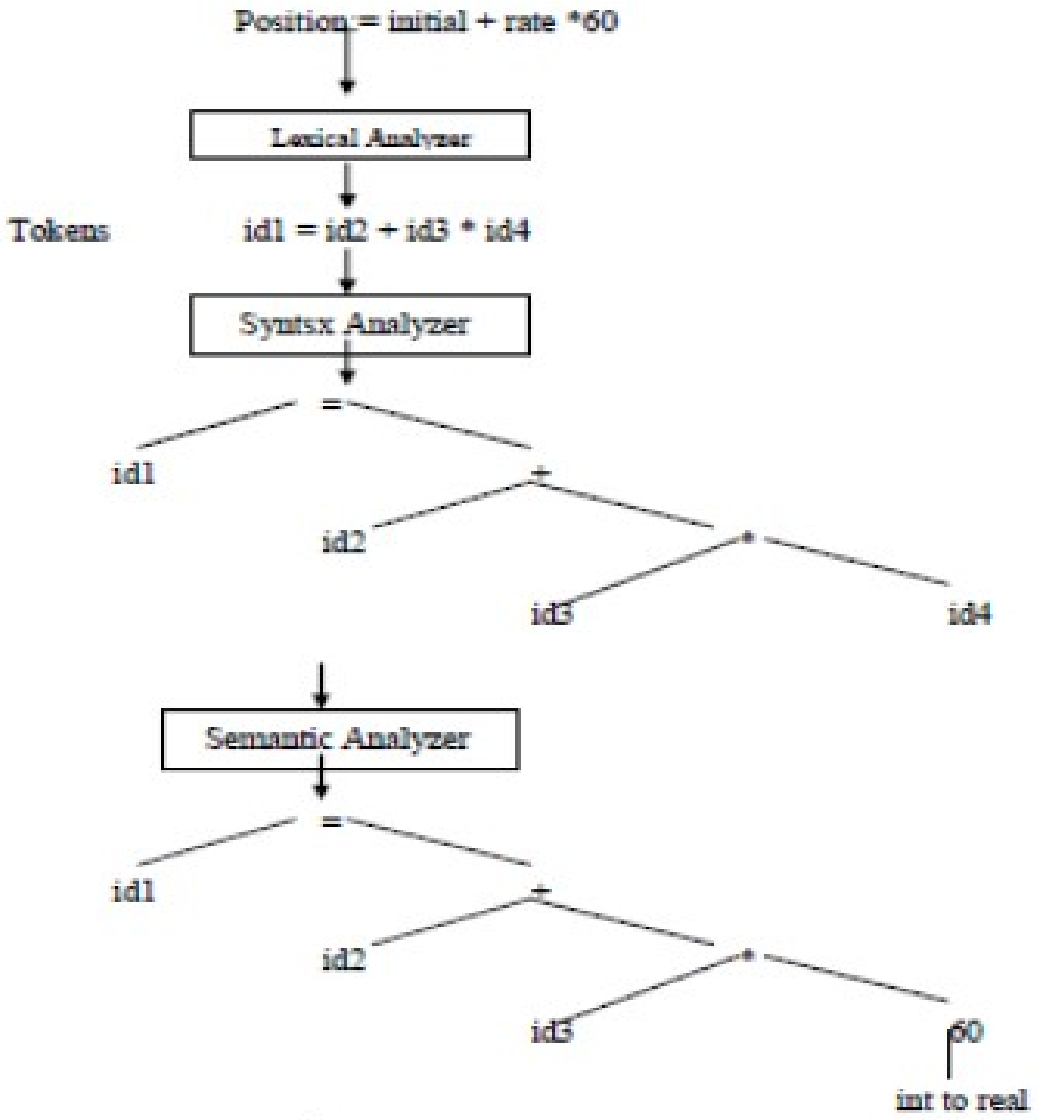
Symbol Table Management:

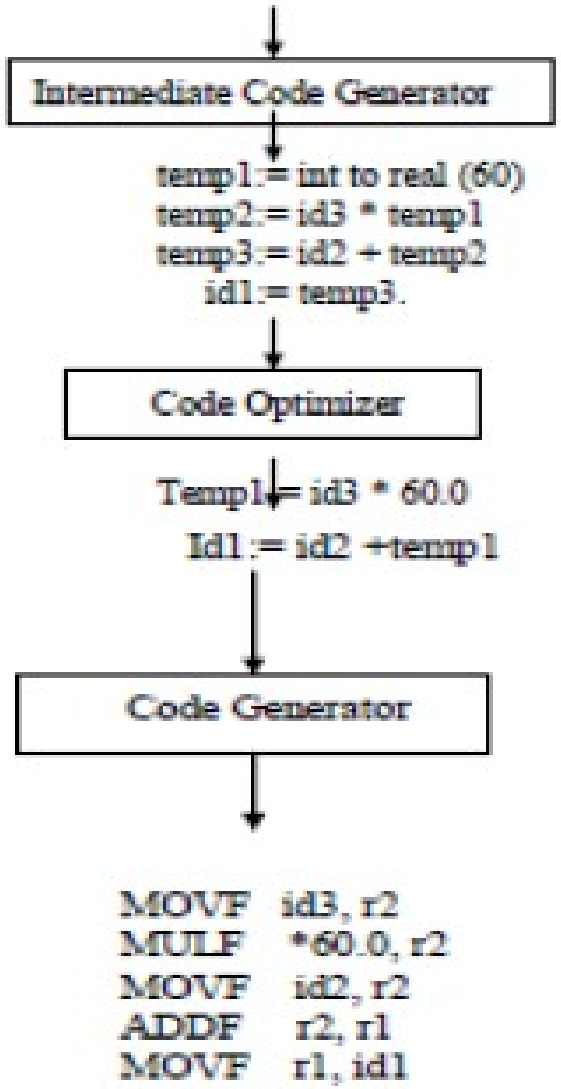
- An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier.
- These attributes may provide information about the storage allocated for an identifier, its type, its scope.
- The symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier.
- When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table

Error Detection and Reporting:

- Each phase can encounter errors.

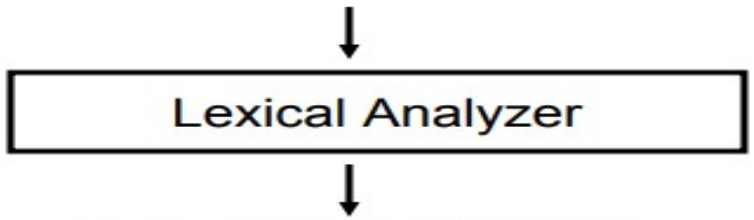
- However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.


Position = initial + rate *60
Lexical Analyzer
Tokens
id1 = id2 + id3 * id4
Syntsx Analyzer
=
id1
+
id2
*
id3
id4
Semantic Analyzer
=
id1
+
id2
*
id3
60
int to real

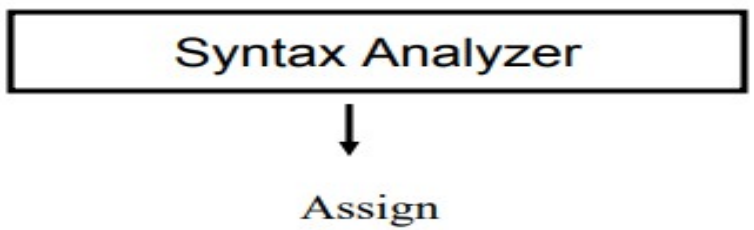

Intermediate Code Generator
temp1:= int to real (60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1:= temp3.
Code Optimizer
Temp1:= id3 * 60.0
Id1 := id2 +temp1
Code Generator
MOVF id3, r2
MULF *60.0, r2
MOVF id2, r2
ADDF r2, r1
MOVF r1, id1

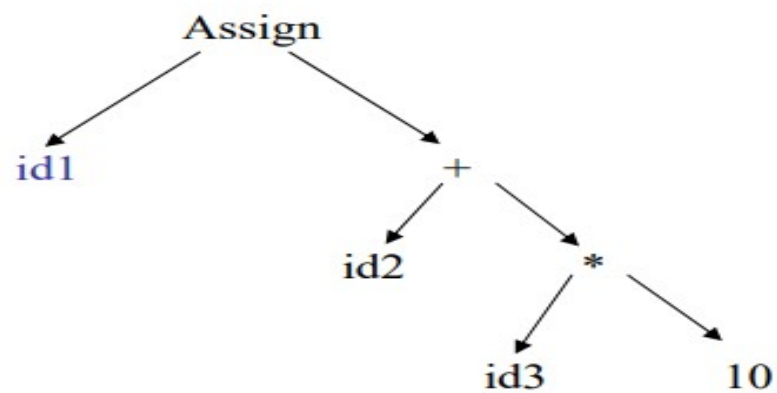
Translation of a statement

result = a + b * 10



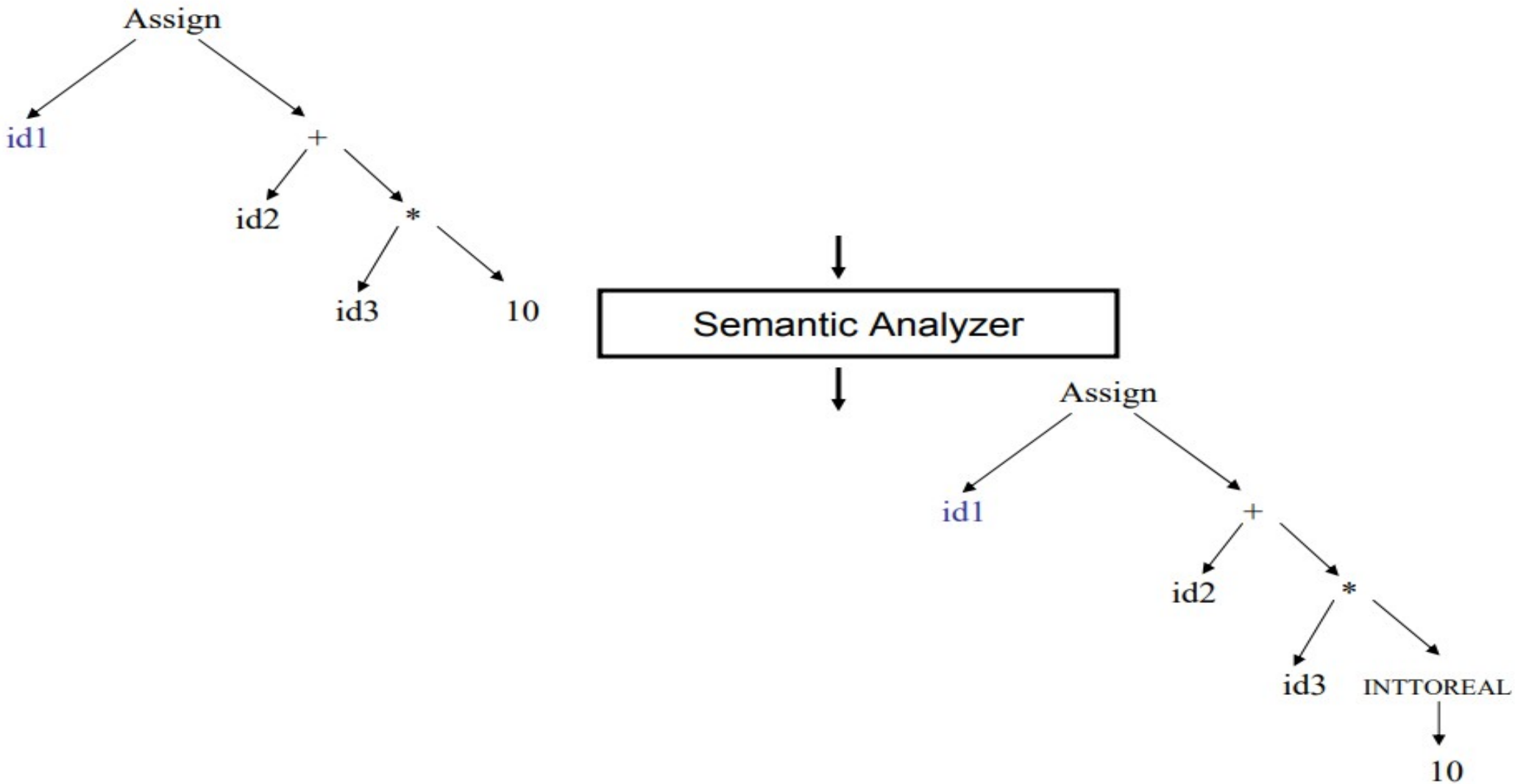
id1 = id2 + id3 * 10

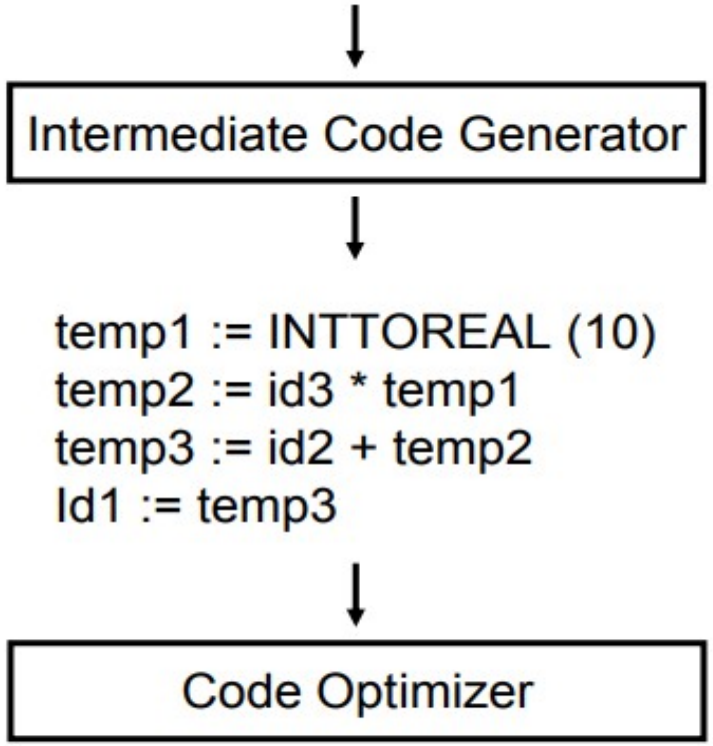


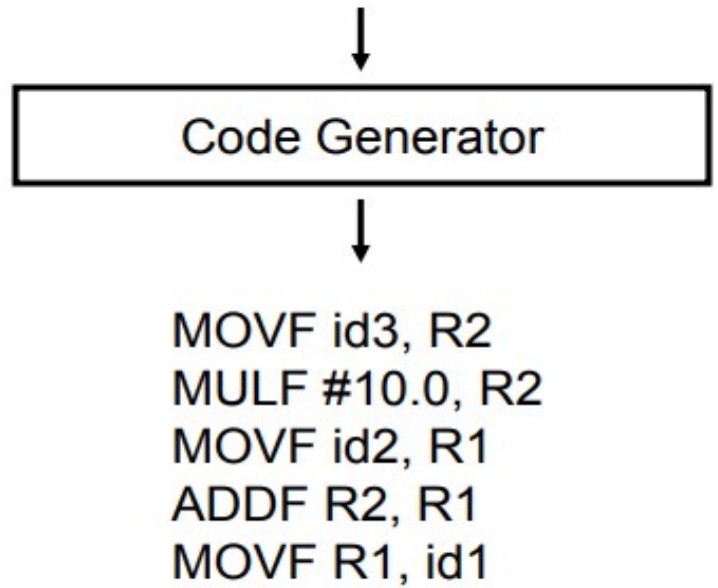


Symbol Table

result	.....
a	.....
b	.....


Assign
id1
+
id2
*
id3
10
Semantic Analyzer
Assign
id1
+
id2
*
id3
INTTOREAL
10


Intermediate Code Generator
temp1 := INTTOREAL (10)
temp2 := id3 * temp1
temp3 := id2 + temp2
Id1 := temp3
Code Optimizer
temp1 := id3 * 10.0
Id1 := id2 + temp1


Code Generator
MOVF id3, R2
MULF #10.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

The Grouping of Phases

Front and Back Ends

- The phases are collected into a front end and a back end.
- The front end consists of those phases that depend primarily on the source language and are largely independent of the target machine.
- These normally include lexical and syntactic analysis, the creating of the symbol table, semantic analysis, and the generation of intermediate code.
- A certain among of code optimization can be done by the front end as well.
- The front end also includes the error handling that goes along with each of these phases.

- The back end includes those portions of the compiler that depend on the target machine.
- And generally, these portions do not depend on the source language, depend on just the intermediate language.
- In the back end, we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling and symbol table operations

Compiler-Construction Tools

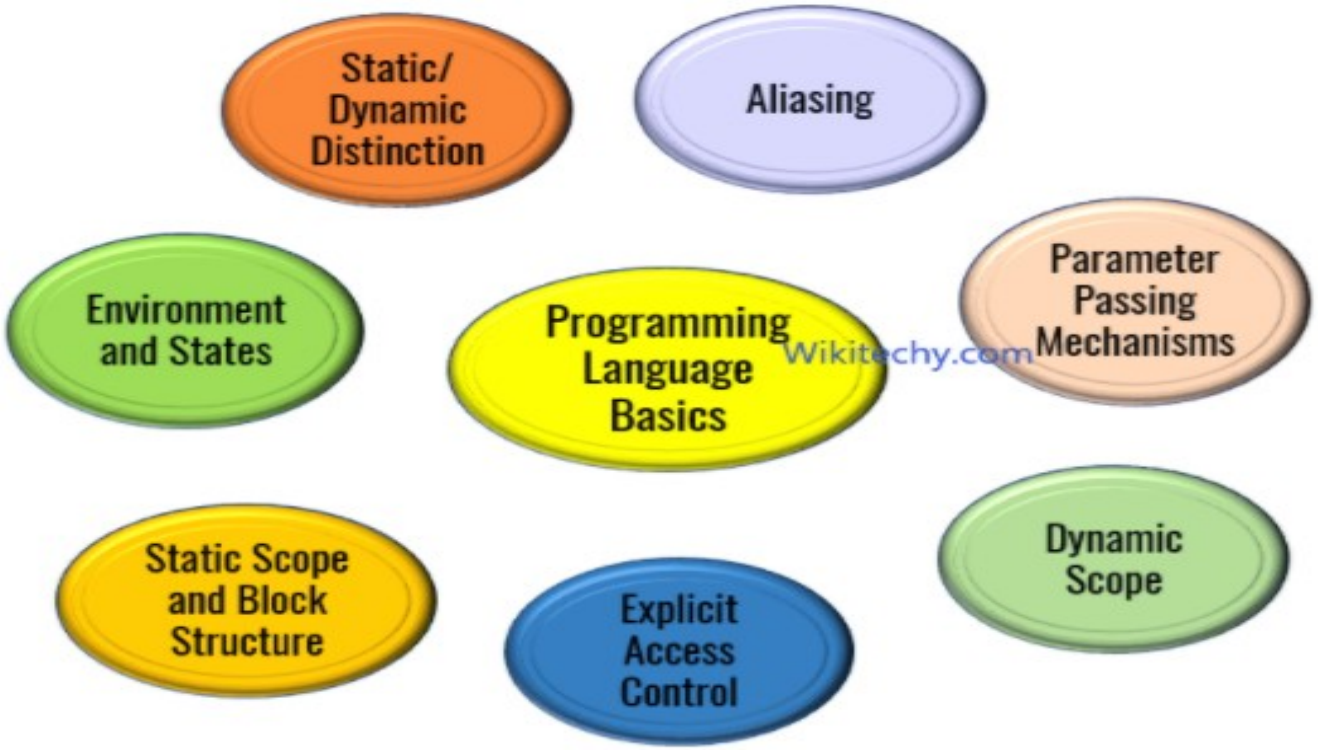
The following is a list of some useful compilerconstruction tools:

- Parser generators
- Scanner generators
- Syntax directed translation engines
- Automatic code generators
- Data-flow engines

The Science of Building a Compiler

- Modeling in Compiler Design and Implementation
- The Science of code Otimization
- compiler optimizations must meet the following design objectives:
  - The optimization must improve the prefomance of many programs,
  - The compilation Time must be kept reasonable,
  - The Engineering effort required must be manageable

Programming Language Basics



Programming Language Basics

Static and Dynamic Distinction

- Static - Events occur at compile time.
- Dynamic - Events occur at run time.

Example

- The scope of a declaration of x is the region of the program in which uses of x refer to this declaration.
- A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program.
- Otherwise, the language uses dynamic scope. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x.

Environment and States

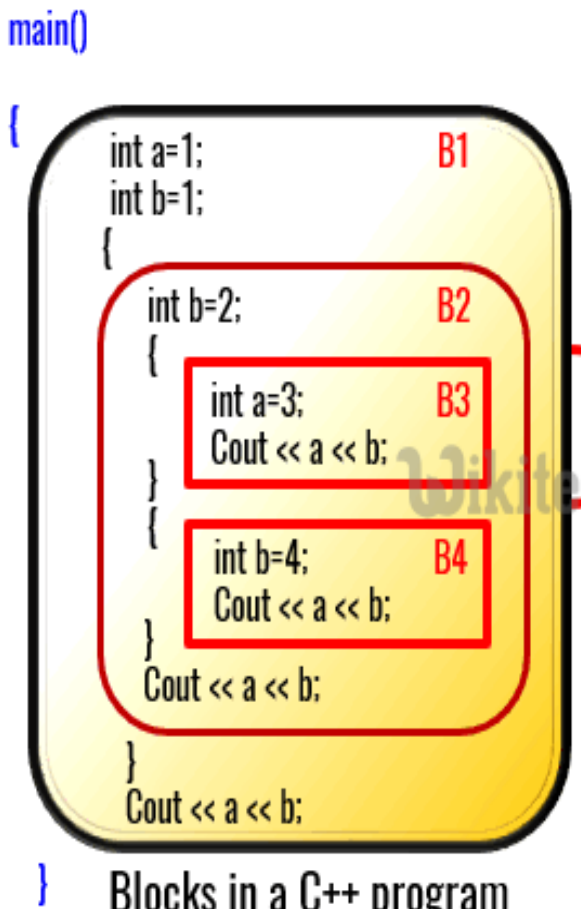
The environment is mapping from names to locations in the store. Since variables refer to locations, we could alternatively define an environment as a mapping from names to variables.

The state is a mapping from locations in store to their values.

Environment and States

Static Scope and Block Structure

- The scope rules for C are based on program structure. The scope of a declaration is determined implicitly by where the declaration appears in the program.
- Programming languages such as C++, Java, and C#, also provide explicit control over scopes through the use of keywords like public, private, and protected.
- A block is a grouping of declarations and statements. C uses braces { and } to delimit a block, the alternative use of begin and end in some languages.



Blocks in a C++ program

Scopes of declarations

BLOCK

Declaration D "belongs" to block B if B is the most closely nested block containing D

Scope of Declaration D is the block containing D and all sub-blocks That don't redeclare D.

DECLARATION	SCOPE
Int a=1;	B1 – B3
Int b=1;	B1 – B2
Int b=2;	B2 – B4
Int a=3;	B3
Int b=4;	B4

Explict Access Control

- Classes and structures introduce a new scope for their members.
- If p is an object of a class with a field (member) x, then the use of x in p.x refers to field x in the class definition.
- Through the use of keywords like public, private, and protected, object oriented languages such as C++ or Java provide explicit control over access to member names in a super class. These keywords support encapsulation by restricting access.
  - Public - Public names are accessible from outside the class
  - Private - Private names include method declarations and definitions associated with that class and any "friend" classes.
  - Protected - Protected names are accessible to subclasses.

Dynamic Scope

- The term dynamic scope, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration.
- Dynamic scoping of this type appears only in special situations. The two dynamic policies are:
  - Macro expansion in the C preprocessor
  - Method resolution in object-oriented programming.

```
int x = 10;
// Called by g()
int f()
{
 return x;
}
// g() has its own variable
// named as x and calls f()
int g()
{
 int x = 20;
 return f();
}
main()
{
 printf(g());
}
```

20

Parameter Passing Mechanisms

- Every language has some method for passing parameters to functions and procedures.
- Formal Parameters: The identifier used in a method to stand for the value that is passed into the method by a caller.
- Actual Parameters: The actual value that is passed into the method by a caller.
  - Call by Value - The actual parameter is evaluated (if it is an expression) or copied (if it is a variable) in a formal parameter.
  - Call by Reference - The address of the actual parameter is passed as value of the corresponding formal parameter.
  - Call by Name - The Called object execute as if the actual parameter were substituted literally for the formal parameter.

CALL BY VALUE

```
#include <stdio.h>
int sum(int a, int b)
{
    int c=a+b;
    return c;
}

int main(
{
   int var1 =10;
   int var2 = 20;
   int var3 = sum(var1, var2);
   printf("%d", var3);
}
```

Formal parameter

Actual parameter

CALL BY REFERENCE

```
#include
void swapnum ( int *var1, int *var2 )
{
   int tempnum ;
   tempnum = *var1 ;
   *var1 = *var2 ;
   *var2 = tempnum ;
}
int main( )
{
   int num1 = 35, num2 = 45 ;
   printf("Before swapping:");
   printf("%d", num1);
   printf("%d", num2);

   /*calling swap function*/
   swapnum( &num1, &num2 );

   printf("\nAfter swapping: ");
   printf("%d", num1);
   printf("%d", num2);
}
```


passing value to a function copies the address of an argument

CALL BY NAME

```
Void sayHello (void)
{
Printf(" Hello
World!\n");
}
Int main()
{
sayHello();
Return 0;
}
```

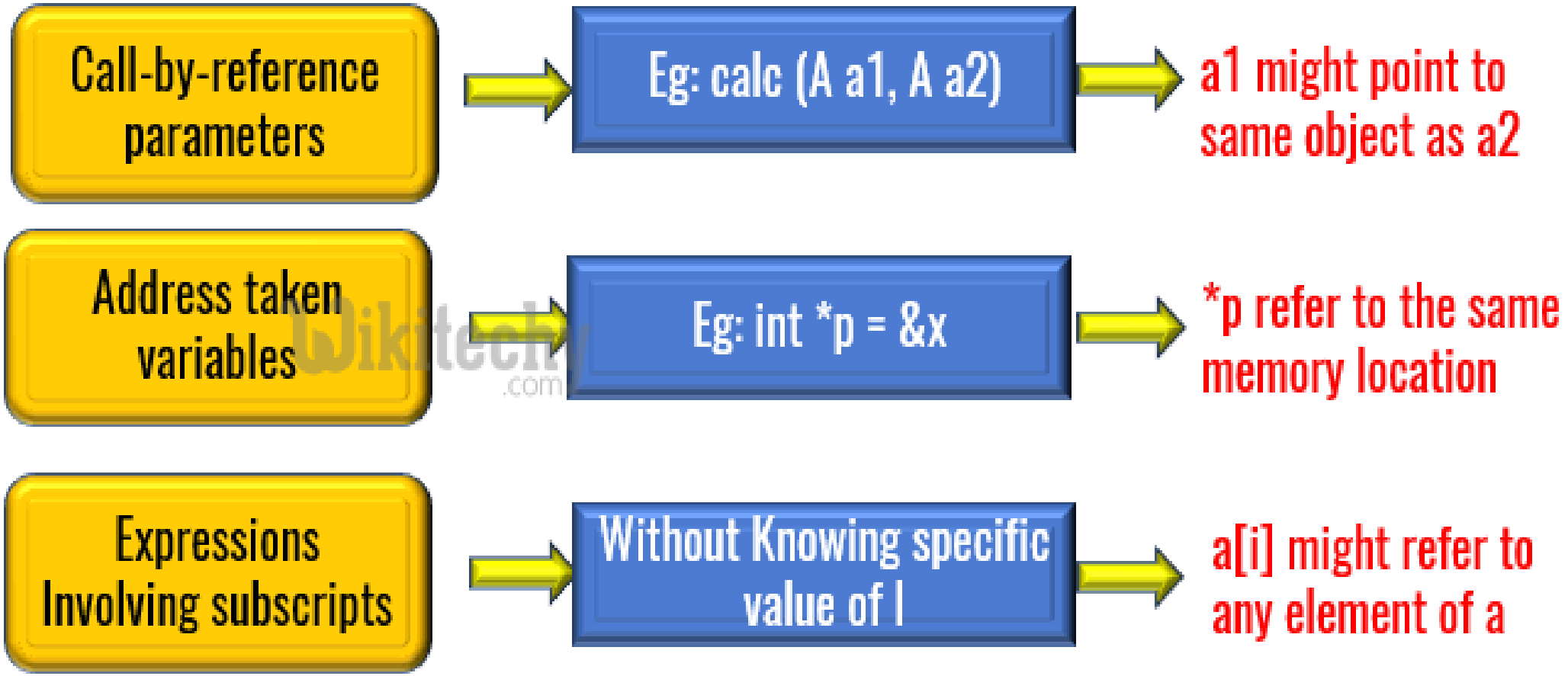
Function name

Function Body

Formal and Actual Parameter

Aliasing

- When two names refer to the same location in memory.
- There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value.
- It is possible that two formal parameters can refer to the same location; such variables are said to be aliases of one another.
- As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other.



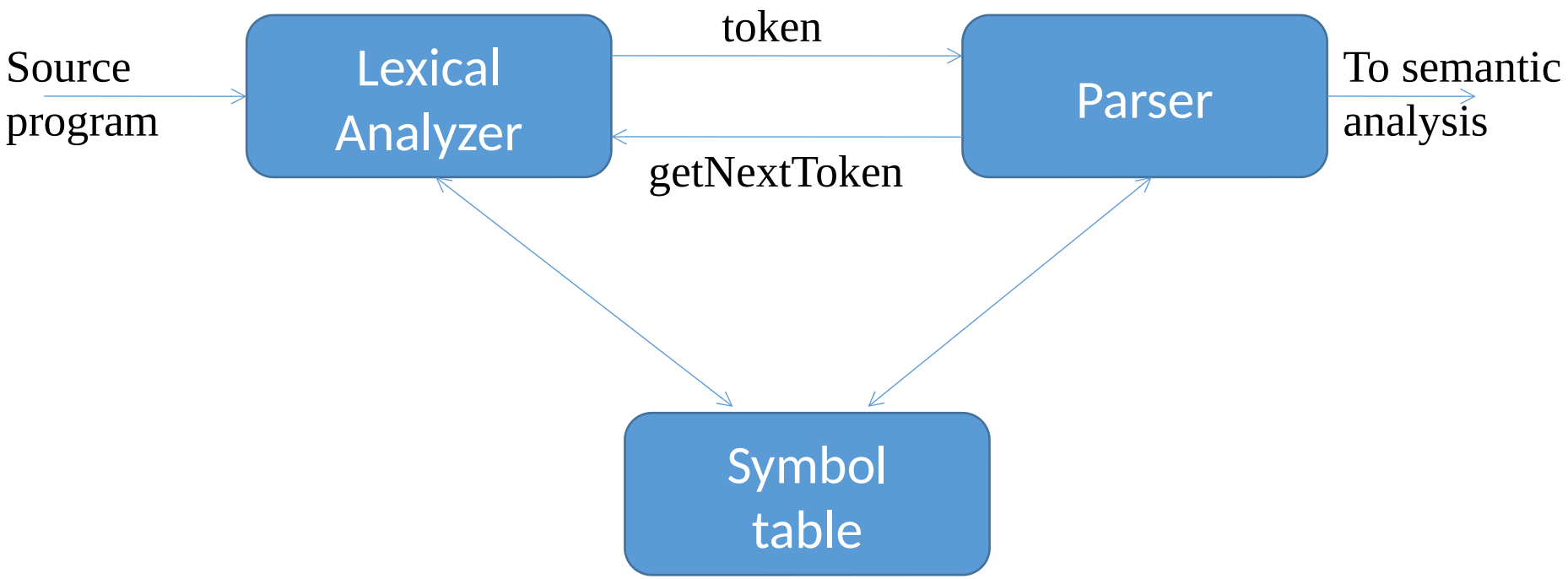
Aliasing

lexical analyzer

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

The role of lexical analyzer



Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

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”

printf(“total = %d\n”, score);

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>

Lexical errors

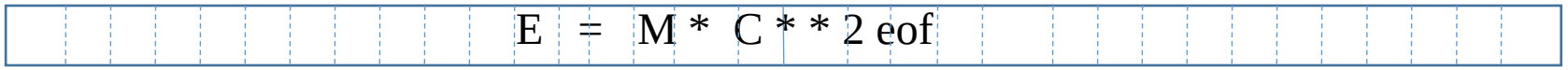
- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

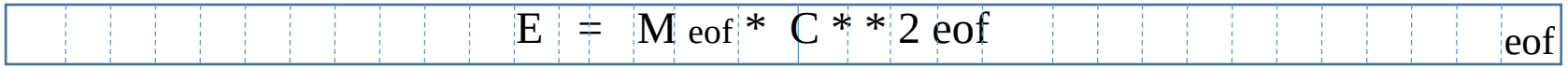
- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

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



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;
}
```

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
    - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in ∑then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L9r))*
- (r) is a regular expression denting L(r)

Regular definitions

d1 -> r1

d2 -> r2

…

dn -> rn

- Example:

  letter_ -> A | B | … | Z | a | b | … | Z | _
  digit -> 0 | 1 | … | 9
  id -> letter_ (letter_ | digit)*

Extensions

- One or more instances: (r)+
- Zero of one instances: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit   -> [0-9]
  - id   -> letter_(letter|digit)*

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ε
expr -> term **relop** term
| term
term -> **id**
| **number**

Recognition of tokens (cont.)

- 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

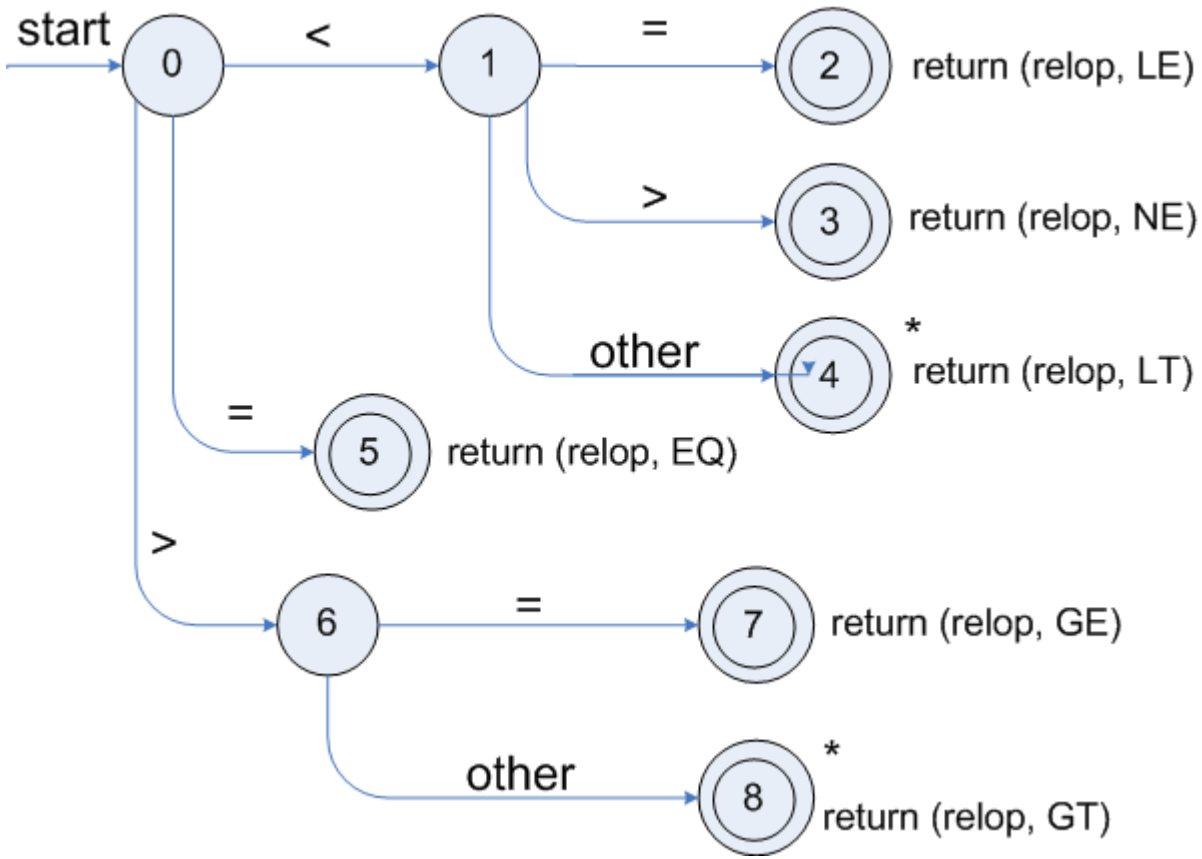
  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

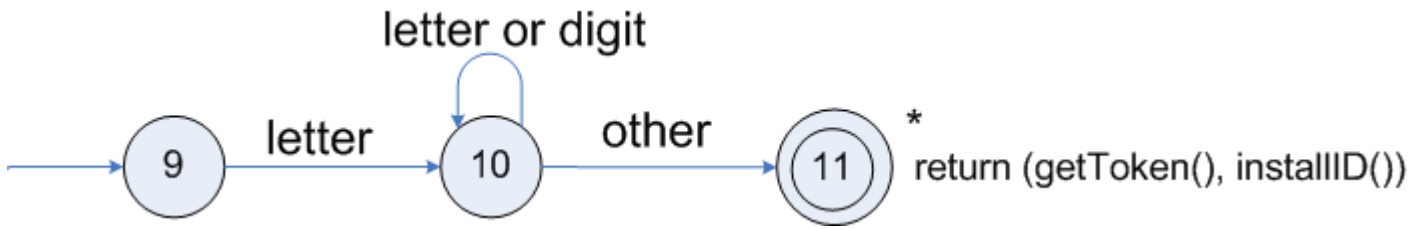
Transition diagrams

- Transition diagram for relop



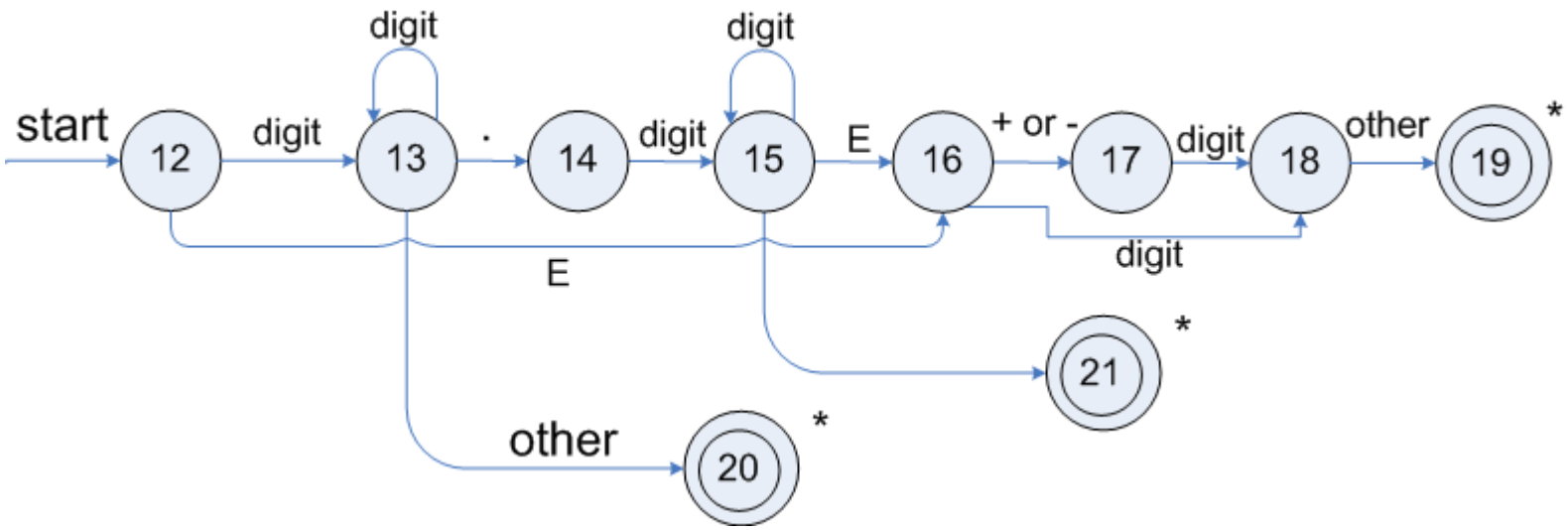
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



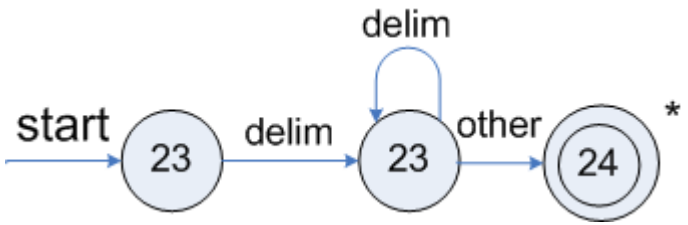
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

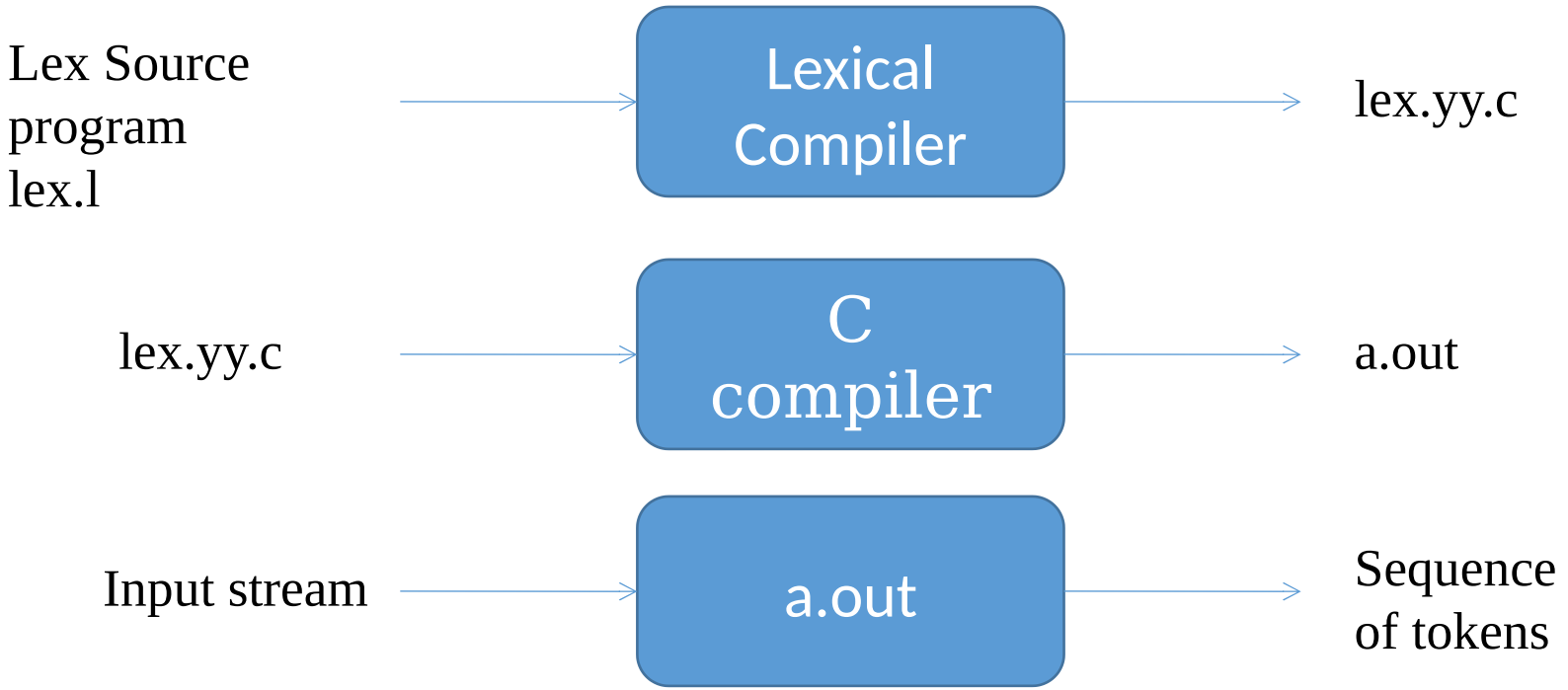
- Transition diagram for whitespace



Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) {    /* repeat character processing until a
    return or failure occurs    */
    switch(state) {
    case 0: c= nextchar();
      if (c == '<') state = 1;
      else if (c == '=') state = 5;
      else if (c == '>') state = 6;
      else fail();   /* lexeme is not a relop */
      break;
    case 1: …
    …
    case 8: retract();
     retToken.attribute = GT;
     return(retToken);
    }
```

Lexical Analyzer Generator - Lex



Structure of Lex programs

declarations
%%
translation rules
%%
auxiliary functions

Pattern {Action}

Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim      [ \t\n]
ws {delim}+
letter     [A-Za-z]
digit      [0-9]
id  {letter}({letter}|{digit})*
number  {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}       {/* no action and no return */}
if   {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID); }
{number}{yylval = (int) installNum(); return(NUMBER);}
…
```

```
Int installID() {/* funtion to
    install the lexeme, whose first
    character is pointed to by
    yytext, and whose length is
    yyleng, into the symbol table
    and return a pointer thereto
    */
}

Int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate
    table */
}
```