PUNE VIDYARTHI GRIHA's
COLLEGE OF ENGINEERING, NASHIK.

- ***"INTRODUCTION OF COMPILER AND L1EXICAL ANALYSIS "***

PREPARED BY :

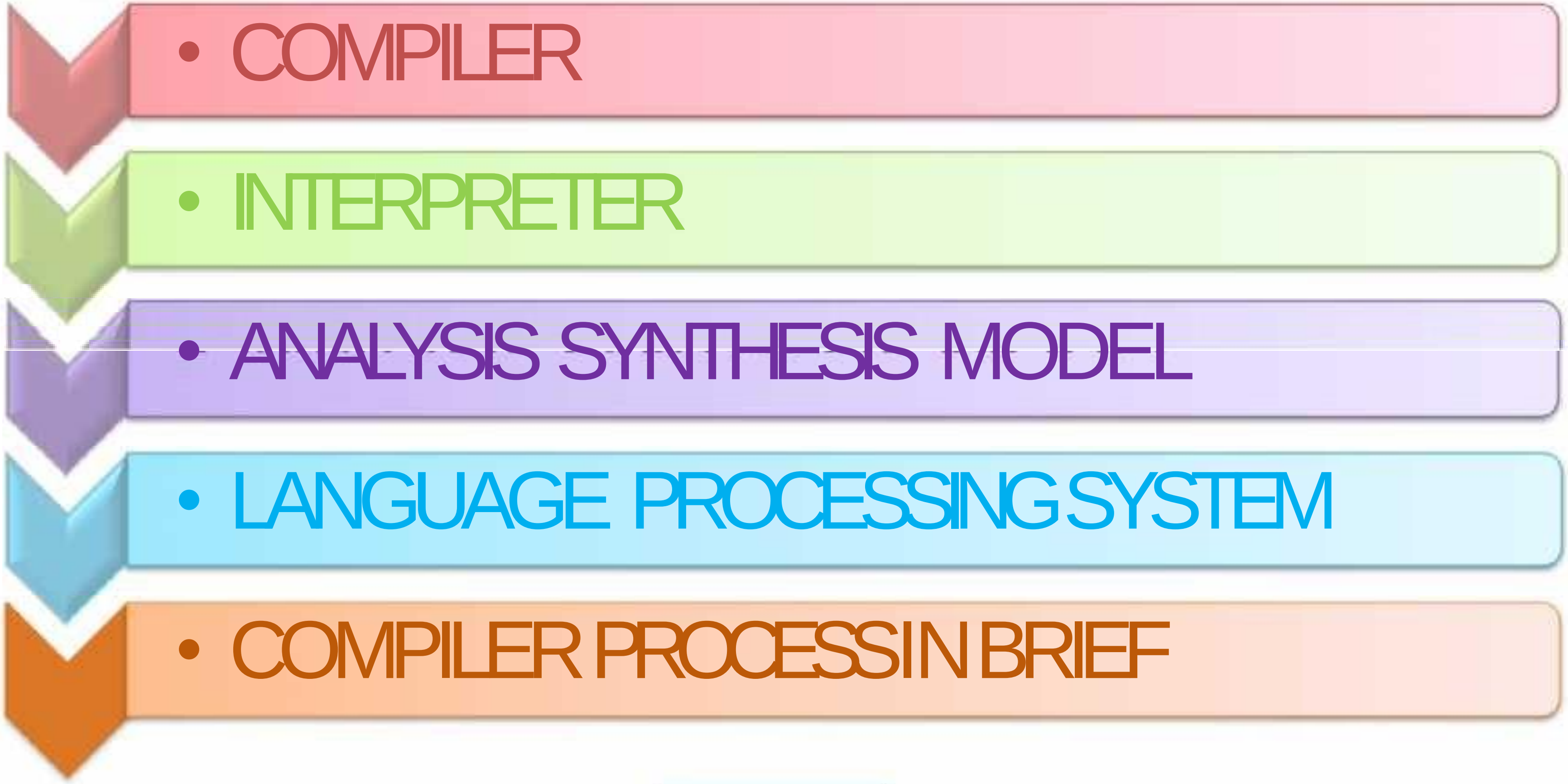
PROF. ANAND N. GHARU

ASSISTANT PROFESSOR

COMPUTER DEPARTMENT

SUBJECT – COMPILER (BE COMPUTER SPPU-2019)

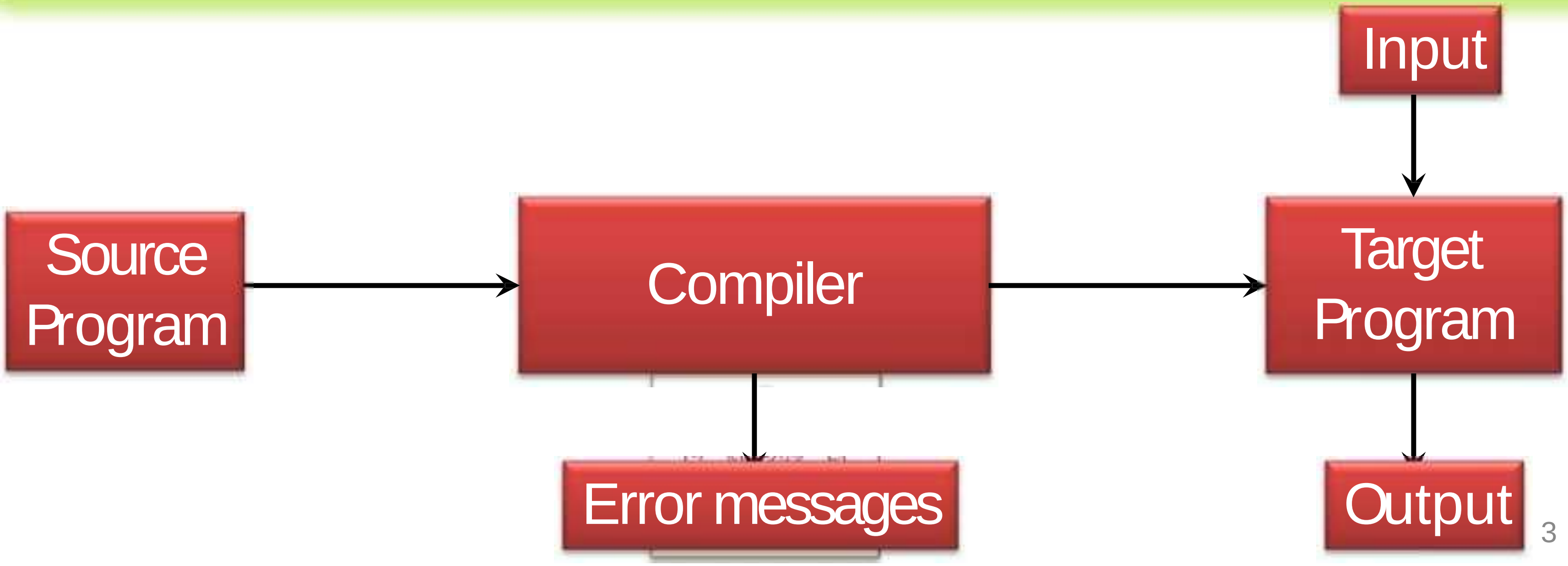
CONTENTS



- COMPILER
- INTERPRETER
- ANALYSIS SYNTHESIS MODEL
- LANGUAGE PROCESSING SYSTEM
- COMPILER PROCESSIN BRIEF

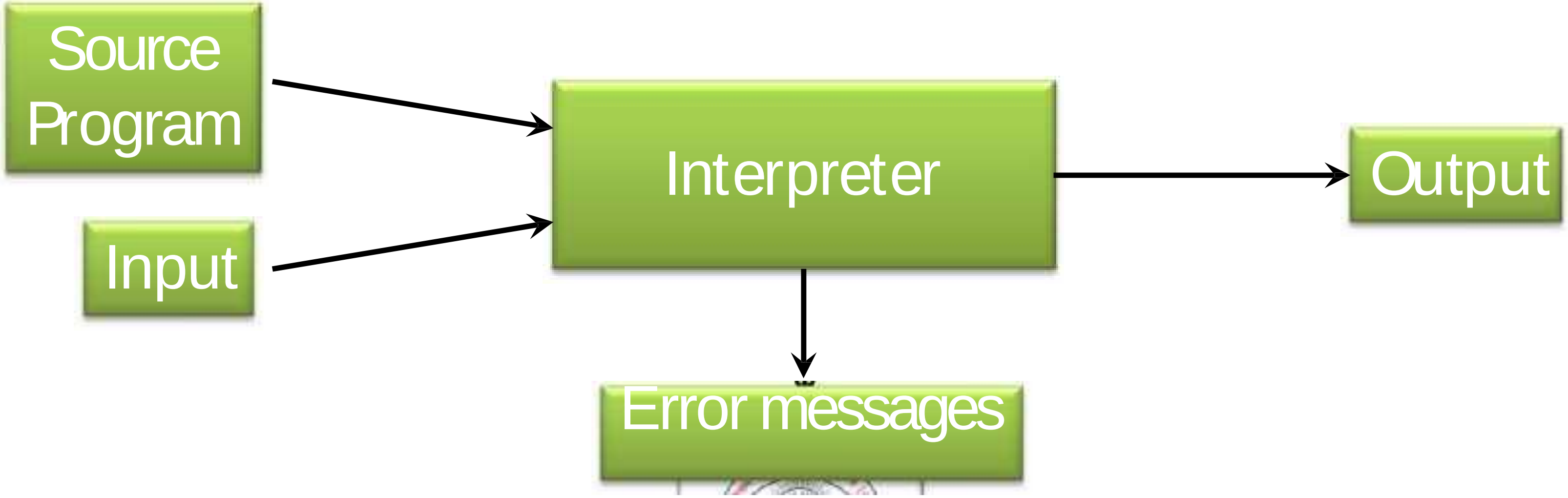
COMPILERS

- "*Compilation*"
  - – Translation of a program written in a source language into a semantically equivalent program written in a target language



INTERPRTERS

- *"Interpretation"*
  - – Performing the operations implied by the source program



ANALYSIS – SYNTHESIS MODEL

- There are two parts to compilation:

Analysis determines the operations implied by the source program which are recorded in a tree structure

Synthesis takes the tree structure and translates the operations therein into the target program

ANALYSIS

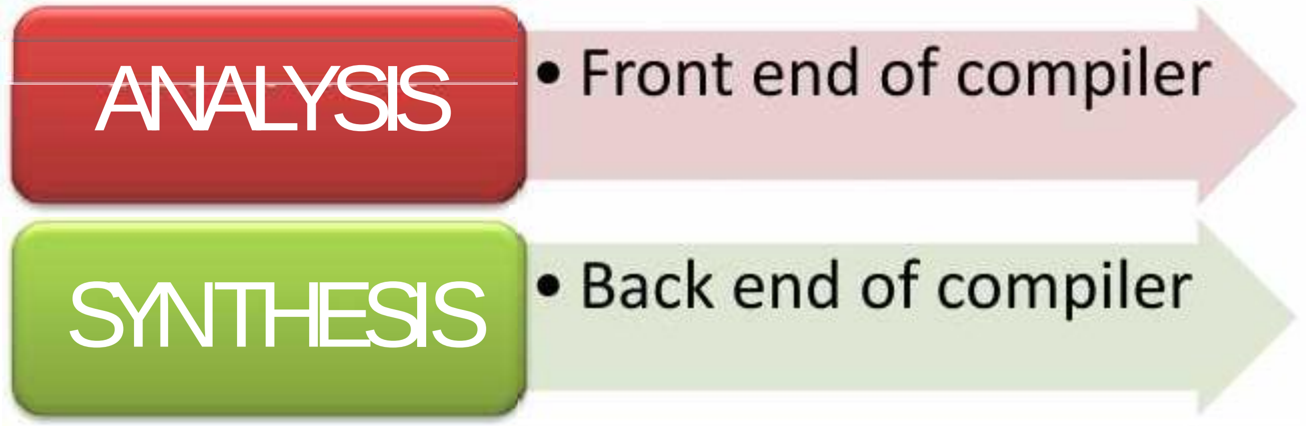
Breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.

If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.

The analysis part also collects information about the source program and stores it in a data structure called a ***symbol table,*** *which is passed along with the intermediate representation to the synthesis part.*

SYNTHESIS

- The synthesis part constructs the desired target program from the intermediate representation and the information in the ***symbol table***



ANALYSIS

- Front end of compiler

SYNTHESIS

- Back end of compiler

COMPILERS ROLE

- An essential function of a compiler –

Record the variable names used in the source program and collect information about various attributes of each name.

- These attributes may provide information about the storage allocated for a name , its type and its scope , procedure names ,number and types of its arguments, the method of passing each argument and the type returned

ISSUES IN COMPILATION

Hierarchy of operations need to be maintained to determine correct order of expression evaluation

Maintain **data type integrity** with automatic type conversions

Handle **user defineddata types.**

Develop appropriate **storage mappings**

ISSUES IN COMPILATION

Resolve occurrence of each variable name in a program i.e **construct separate symbol tables for different namespaces.**

Handle different control structures.

Perform **optimization**

ISSUES IN COMPILATION

	Single Pass	Multi Pass
Speed	better	worse
Memory	better for large programs	(potentially) better for small programs
Modularity	worse	better
Flexibility	worse	better
“Global” optimization	impossible	possible
Source Language	single pass compilers are not possible for many programming languages	

COMPILER PASSES

A pass is a complete traversal of the source program, or a complete traversal of some internal representation of the source program.

A pass can correspond to a “phase” but it does not have to!

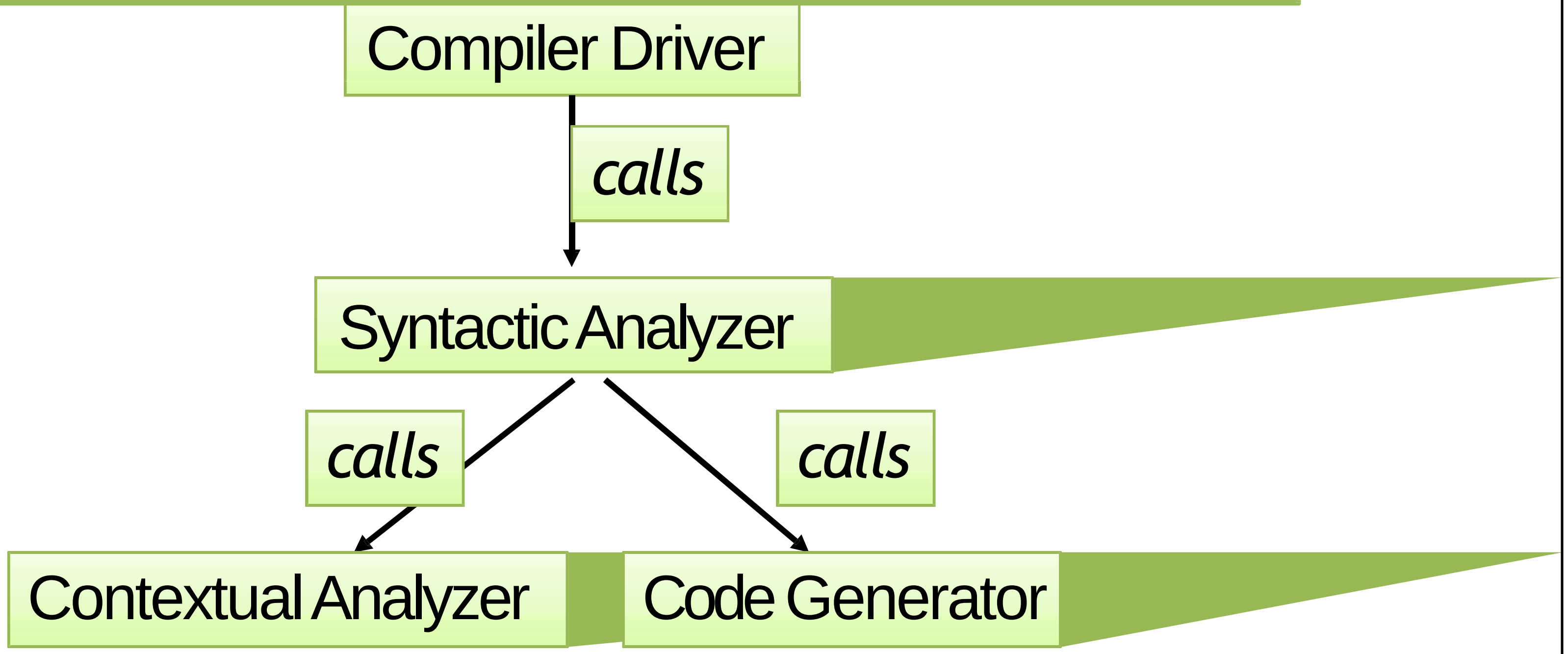
Sometimes a single “pass” corresponds to several phases that are interleaved in time.

What and how many passes a compiler does over the source program is an important designdecision.

SINGLE PASS COMPILER

A single pass compiler makes a single pass over the source text, parsing, analyzing and generating code all at once.

Dependency diagram of a typical Single Pass Compiler:



Compiler Driver

calls

Syntactic Analyzer

calls

calls

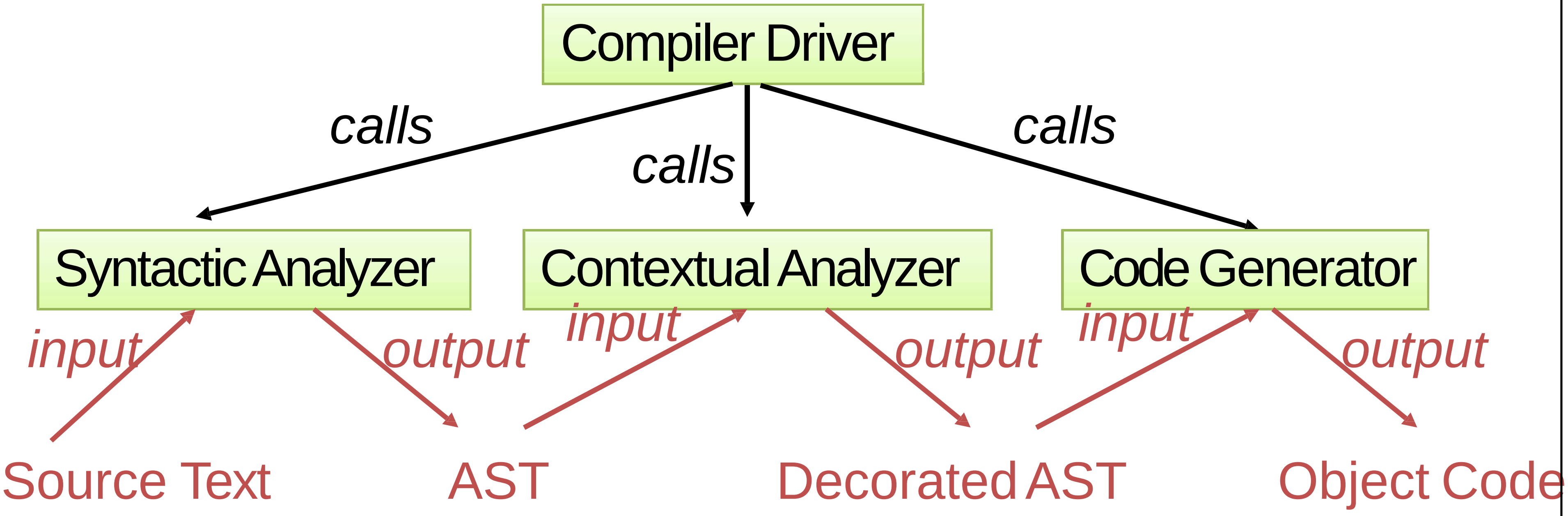
Contextual Analyzer

Code Generator

MULTI PASS COMPILER

A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.

Dependency diagram of a typical Multi Pass Compiler:



Compiler Driver

Syntactic Analyzer

Contextual Analyzer

Code Generator

Source Text

AST

Decorated AST

Object Code

SYMBOL TABLE MEANS

*Symbol tables are **data structures** that are used by compilers to hold information about source-program constructs.*

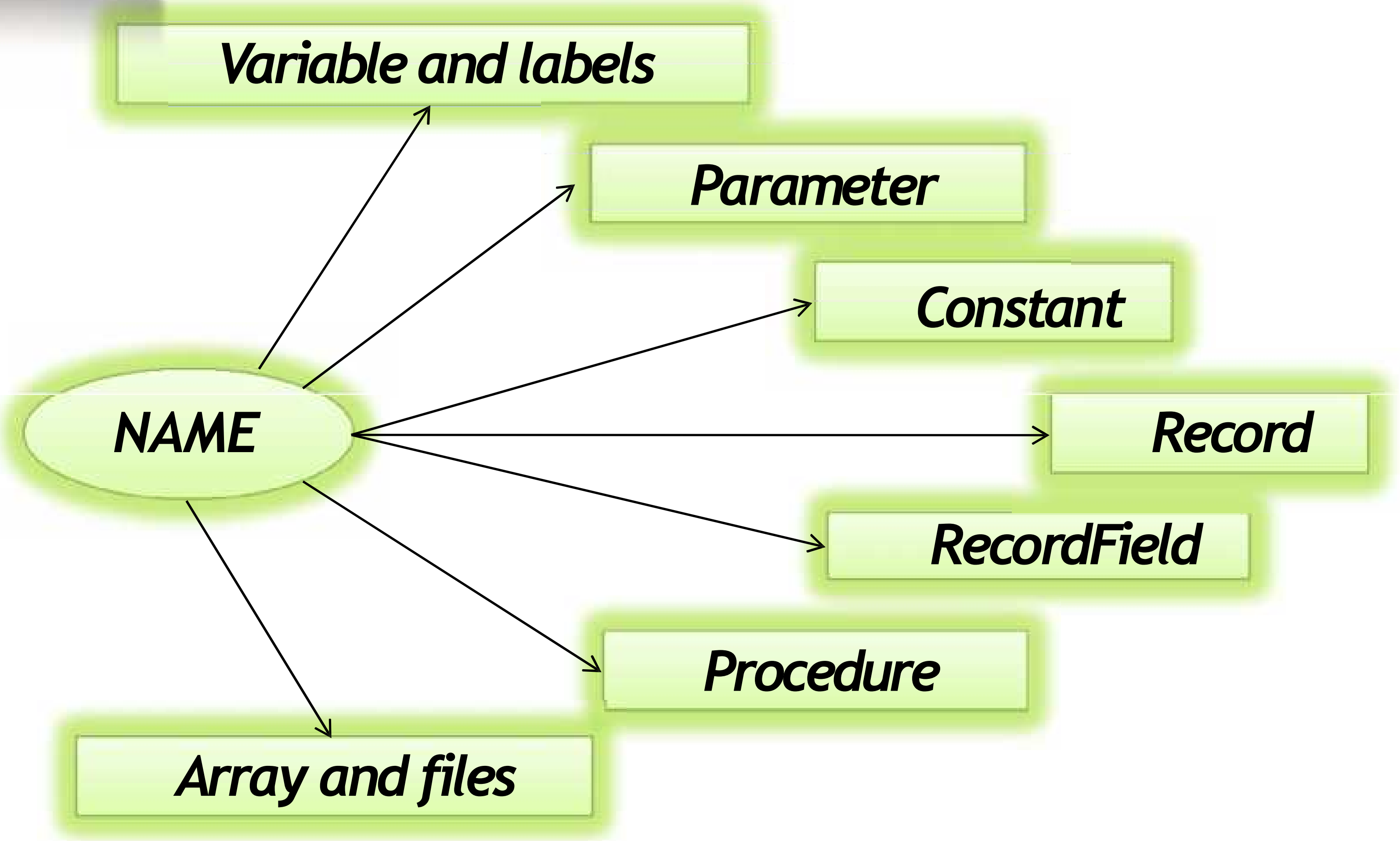
Asymbol table is a necessary component because

- Declaration of identifiers appears once in a program
- Use of identifiers may appear in many places of the program text

N FORMATION PROVIDED BY SYMBOL TABLE

- *Given an Identifier which name isit?*
- *What information is to be associated with a name?*
- *How do we access thisinformation?*

SYMBOL TABLE NAMES



WHO CREATES SYMBOL TABLE ?

- Identifiers and attributes are entered by the analysis phases when processing a definition (declaration) of an identifier
- In simple languages with only global variables and implicit declarations:
  - ✓The scanner can enter an identifier into a symbol table if it is not already there
- In block-structured languages with scopes and explicit declarations:
  - ✓ The parser and/or semantic analyzer enter identifiers and corresponding attributes

USE OF SYMBOL TABLE

- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – type checking
- To generate intermediate or target code

MEMORY MANAGEMENT

What has a compiler to do with memory management?

- compiler uses heap-allocated data structures
- modern languages have automatic data (de)allocation
  - garbage collection part of runtime support system
  - compiler usually assists in identifying pointers

GARBAGE COLLECTION

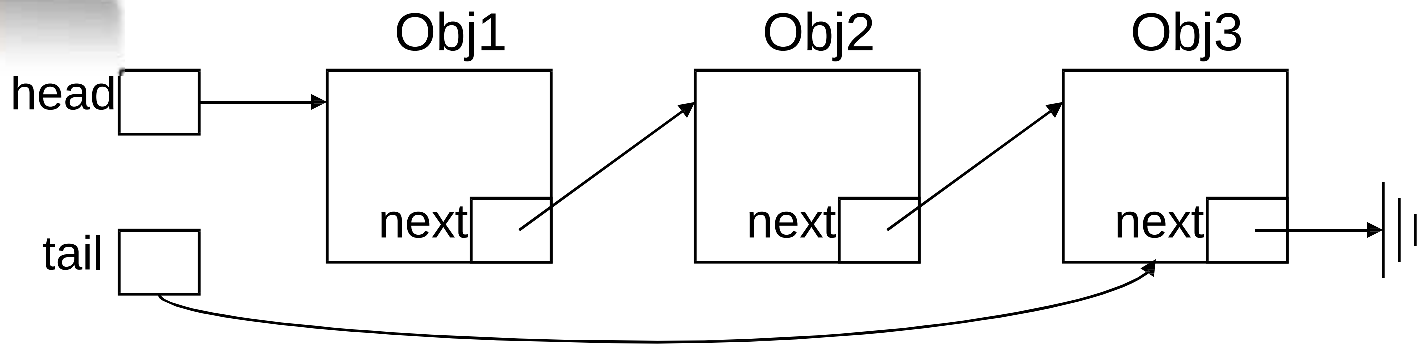
- Some systems require user to call *free* when finished with memory
  - C/ C++

    reason for destructors in C++
- Other systems detect unused memory and reclaim it
  - Garbage Collection
  - this is what Java does

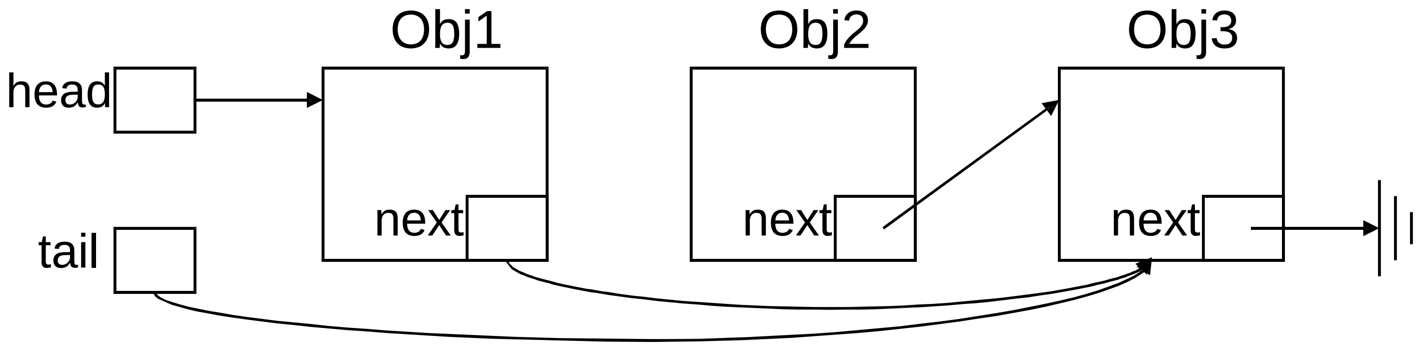
GARBAGE COLLECTION

- Basic idea
  - keep track of what memory is referenced and when it is no longer accessible, reclaim the memory
- Example
  - linked list

EXAMPLE



- Assume programmer does the following
  - obj1.next = obj2.next;

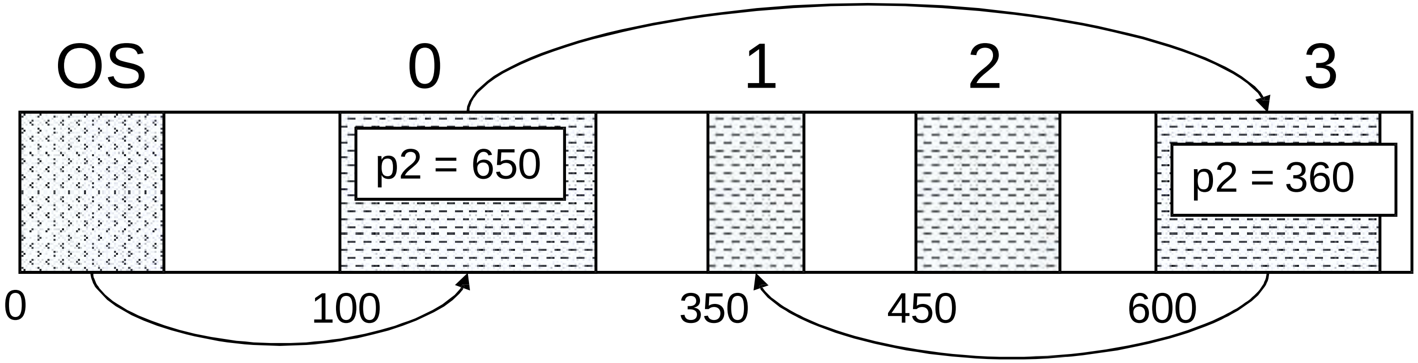


EXAMPLE

- Now there is no way for programmer to reference obj2
  - – it's garbage
- In system without garbage collection this is called a *memory leak*
  - – location can't be used but can't be reallocated
  - – waste of memory and can eventually crash a program
- In system with garbage collection this chunk will be found and reclaimed

MARK AND SWEEP

- Basic idea
  - go through all memory and mark every chunk that is referenced
  - make a second pass through memory and remove all chunks not marked



•Mark chunks 0, 1, and 3 as marked
•Place chunk 2 on the free list (turn it into a hole)

MARK AND SWEEP ISSUES

- Have to be able to identify all references
  - – this is difficult in some languages
  - – similar to compaction
- Requires jumping all over memory
  - – terrible for performance
    - cache hits
    - virtual memory
- Have to stop everything else to do
- Search time proportional to non-garbage
  - – may require lots of work for little reward

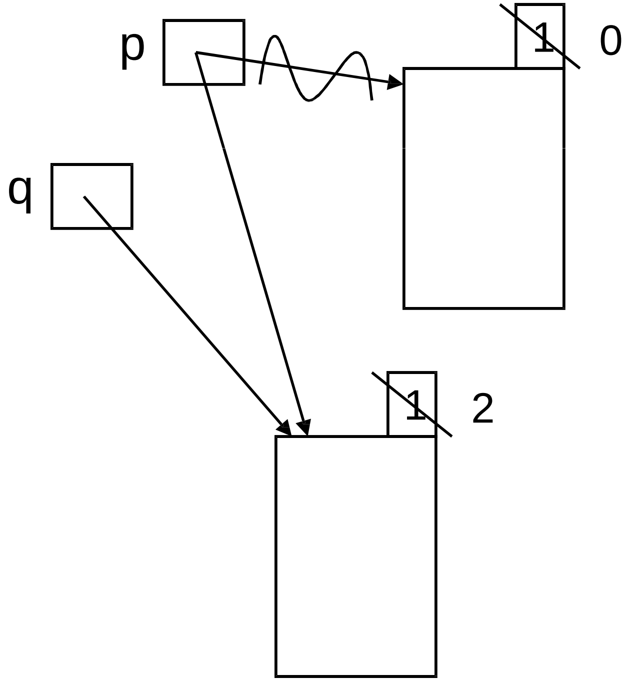
REFERENCE COUNTING

- Basic idea
  - give each chunk a special field that is the number of references to chunk
  - whenever a new reference is made, increment field by 1
  - whenever a reference is removed, decrement field by 1
  - when reference count goes to zero, collect chunk
- Requires compiler support

REFERENCE COUNTING

- Example
  - – everything in italics is added by compiler

```
Object p = new Object;
p.count++;
Object q = new Object;
q.count++;
p.count--;
if(p.count == 0)
            collect p
p = q;
p.count++;
```



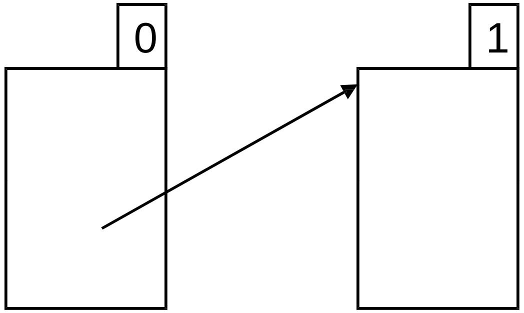
REFERENCE COUNTING

- Above example does not check for NULL reference

```
Object p = new Object
p.count++;
p.count--;
p = NULL;
if(p != NULL)
   p.count++;
```

REFERENCE COUNTING ISSUES

- What about pointers inside 0 referenced page?



  - both of these are garbage
  - before reclaiming a chunk, must go through all references in the chunk
    - decrement the chunk they reference by 1

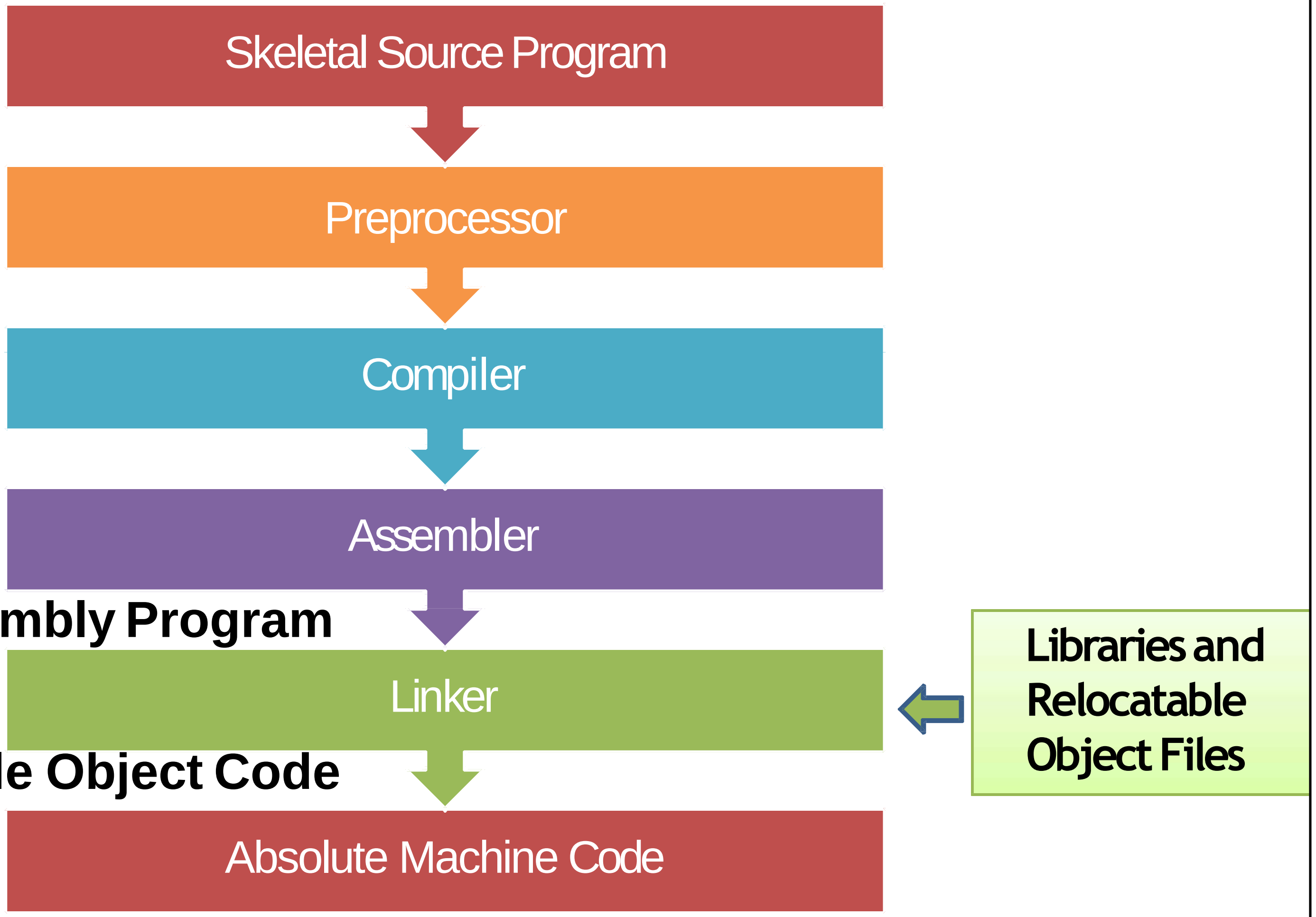
TOOLS USING ANALYSIS – SYNTHESIS MODEL

- Editors (syntax highlighting)
- Pretty printers (e.g. Doxygen)
- Static checkers (e.g. Lint and Splint)
- Interpreters

TOOLS USING ANALYSIS – SYNTHESIS MODEL

- Text formatters (e.g. TeX and LaTeX)
- Silicon compilers (e.g. VHDL)
- Query interpreters/compilers (Databases)

PREPROCESSORS, COMPILERS, ASSEMBLERS, AND LINKERS



Skeletal Source Program

Preprocessor

Compiler

Assembler

Target Assembly Program

Linker

Relocatable Object Code

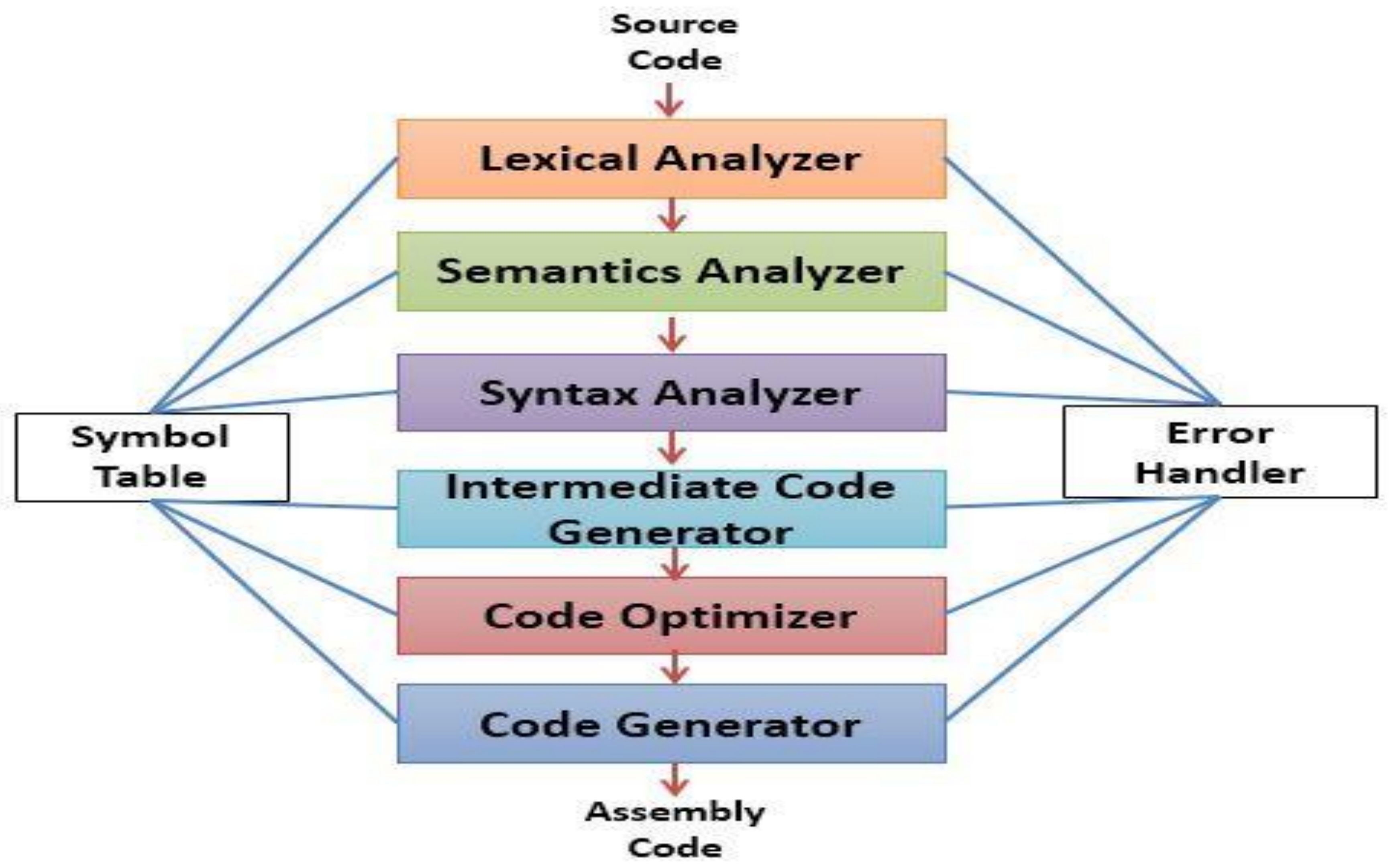
Absolute Machine Code

Libraries and Relocatable Object Files

Try for example:

`gcc myprog.c`

PHASES OF COMPILER



Source Code

Lexical Analyzer

Semantics Analyzer

Syntax Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Assembly Code

Symbol Table

Error Handler

Compiler

THE PHASES OF A COMPILER

Phases	Output	Sample
Programmer (source code producer)	Source string	A=B+C;
Scanner (performs lexical analysis)	Token string	‘A’, ‘=’, ‘B’, ‘+’, ‘C’, ‘;’ Andsymbol table with names
Parser (performs syntax analysisbased on the grammar of the programming language)	Parse tree or abstract syntax tree	; | = / \\ A + / \\ B C
Semantic analyzer (type checking, etc)	Annotated parse tree or abstract syntax tree	
Intermediate codegenerator	Three-address code, quads, orRTL	int2fp B t1 + t1 C t2 := t2 A
Optimizer	Three-address code, quads, orRTL	int2fp B t1 + t1 #2.3 A
Code generator	Assembly code	MOVF #2.3,r1 ADDF2 r1,r2 MOVF r2,A
Peephole optimizer	Assembly code	ADDF2 #2.3,r2 MOVF r2,A

THE GROUPING OF PHASES

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

Compiler *front* and *back ends*

- Acollection 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

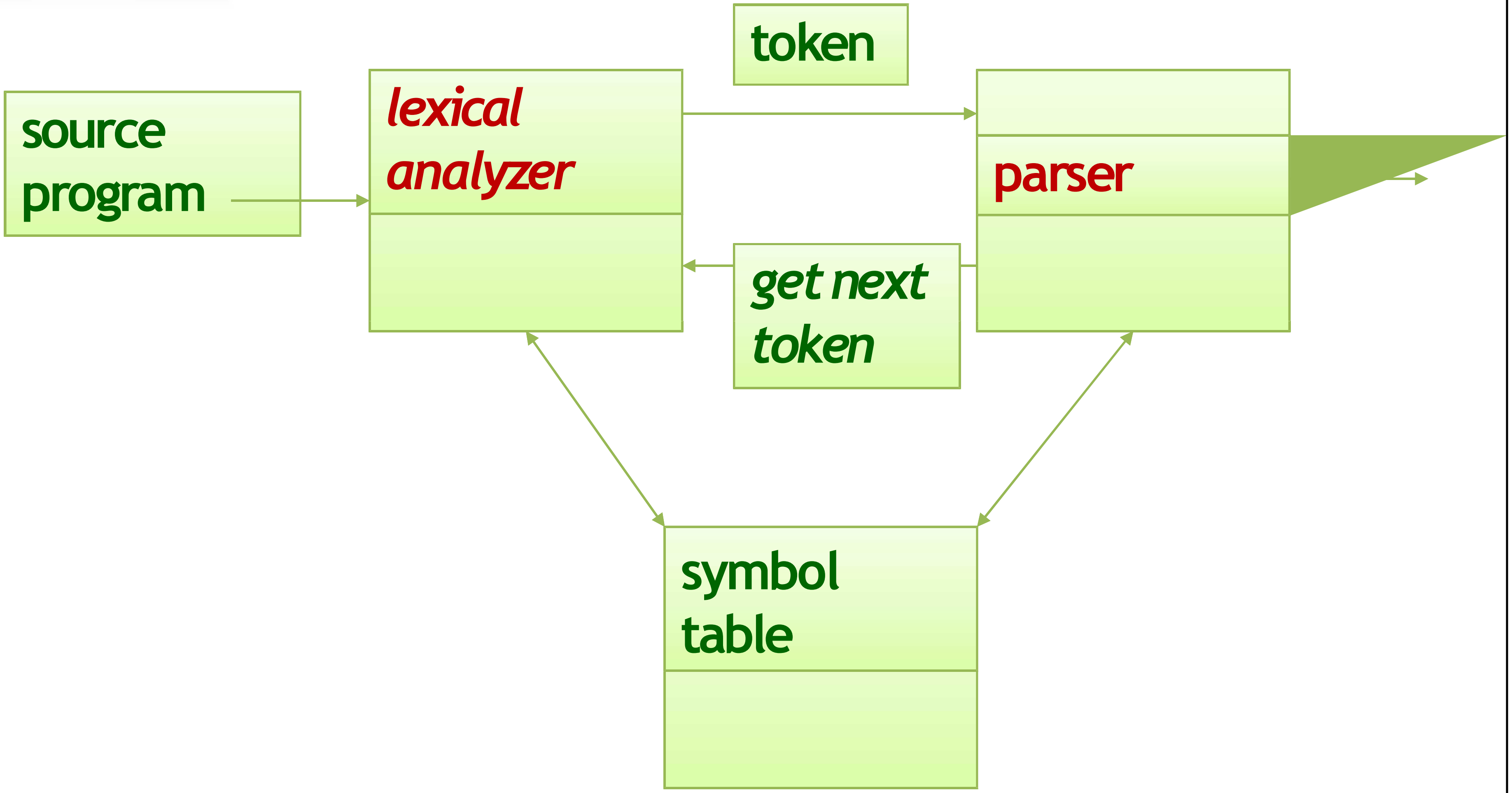
Compiler *passes*

COMPILER CONSTRUCTION TOOLS

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

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

BLOCK SCHEMATIC OF LEXICAL ANALYZER



LEXICAL ANALYZER PERSPECTIVE

LEXICAL ANALYZER

- Scan input
- Remove WS, NL, … Identify

Tok

Tok

- S

PARSER

- Perform Syntax Analysis
- Actions Dictated by Token Order
- Update Symbol Table Entries
- Create Abstract Rep. of Source
- Generate Errors

SEPERATION OF LEXICAL ANALYSIS FROM SYNTAX ANALYSIS

- Separation of Lexical Analysis From Parsing Presents a Simpler Conceptual Model
  - From a Software Engineering Perspective Division Emphasizes
    - High Cohesion and Low Coupling
    - Implies Well Specified ⇒ Parallel Implementation
- Separation Increases Compiler Efficiency (I/O Techniques to Enhance Lexical Analysis)
- Separation Promotes Portability.
  - This is critical today, when platforms (OSs and Hardware) are numerous and varied!
  - Emergence of Platform Independence - Java

BASIC TERMINOLOGIES OF LEXICAL ANALYSIS

- Major Terms for Lexical Analysis?
  - ❑ **TOKEN**
    - ➢ A classification for a common set of strings
    - ➢ Examples Include <Identifier>, <number>, etc.
  - ❑ **PATTERN**
    - ➢ The rules which characterize the set of strings for a token
    - ➢ Recall File and OS Wildcards ([A-Z]*.*)
  - ❑ **LEXEME**
    - ➢ Actual sequence of characters that matches pattern and is classified by a token
    - ➢ Identifiers: x, count, name, etc…

INTRODUCING BASIC TERMINOLOGY

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Classifies Pattern

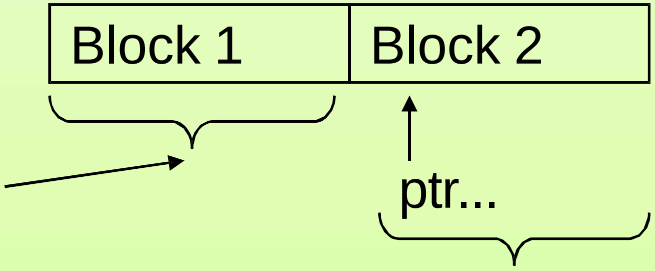
Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

I/O - KEY FOR SUCCESSFUL LEXICAL ANALYSIS

- Character-at-a-time I/O
- Block / Buffered I/O

- Block/Buffered I/O
  - ❑ Utilize Block of memory
  - ❑ Stage data from source to buffer block at a time
  - ❑ Maintain two blocks - Why (Recall OS)?
    - ➢ Asynchronous I/O - for 1 block
    - ➢ While Lexical Analysis on 2nd block



Block 1

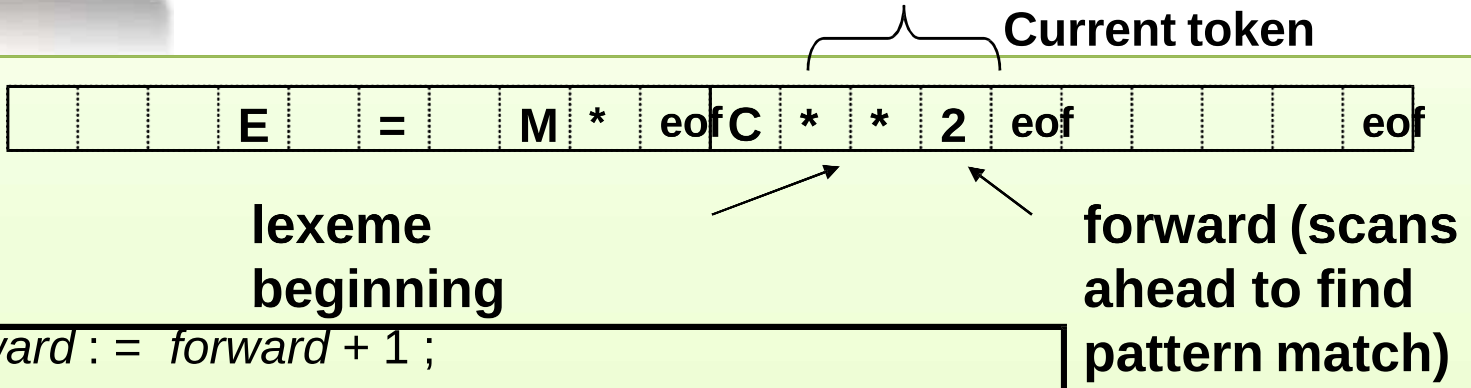
Block 2

When done, issue I/O

ptr...

Still Process token in 2nd block

Algorithm
Buffered I/O with Sentinels



Current token

			E		=		M	*	eof	C	*	*	2	eof					eof

lexeme beginning

forward (scans ahead to find pattern match)

```
forward : =  forward + 1 ;
if  forward  is at  eof  then  begin
    if  forward  at end of first half  then  begin
          reload second half ;  ← Block I/O
          forward  : =  forward  + 1
    end
    else if  forward  at end of second half  then  begin
          reload first half ; ←Block I/O
          move  forward to biginning of first half
    end
    else  / * eof within buffer signifying end of input  * /
         terminate lexical analysis    end
```

2nd eof ⇒ no more input !

HANDLING LEXICAL ERRORS

- Error Handling is very localized, with Respect to Input Source
- For example: whil (x := 0) do generates **no** lexical errors in PASCAL
- In what Situations do Errors Occur?
  - Prefix of remaining input doesn’tmatch any defined token
- Possible error recovery actions:
  - Deleting or Inserting Input Characters
  - Replacing or Transposing Characters
- Or, skip over to next separator to “ignore” problem

Lex

Lex

Tool that helps to take set of descriptions of possible tokens and produce Croutine

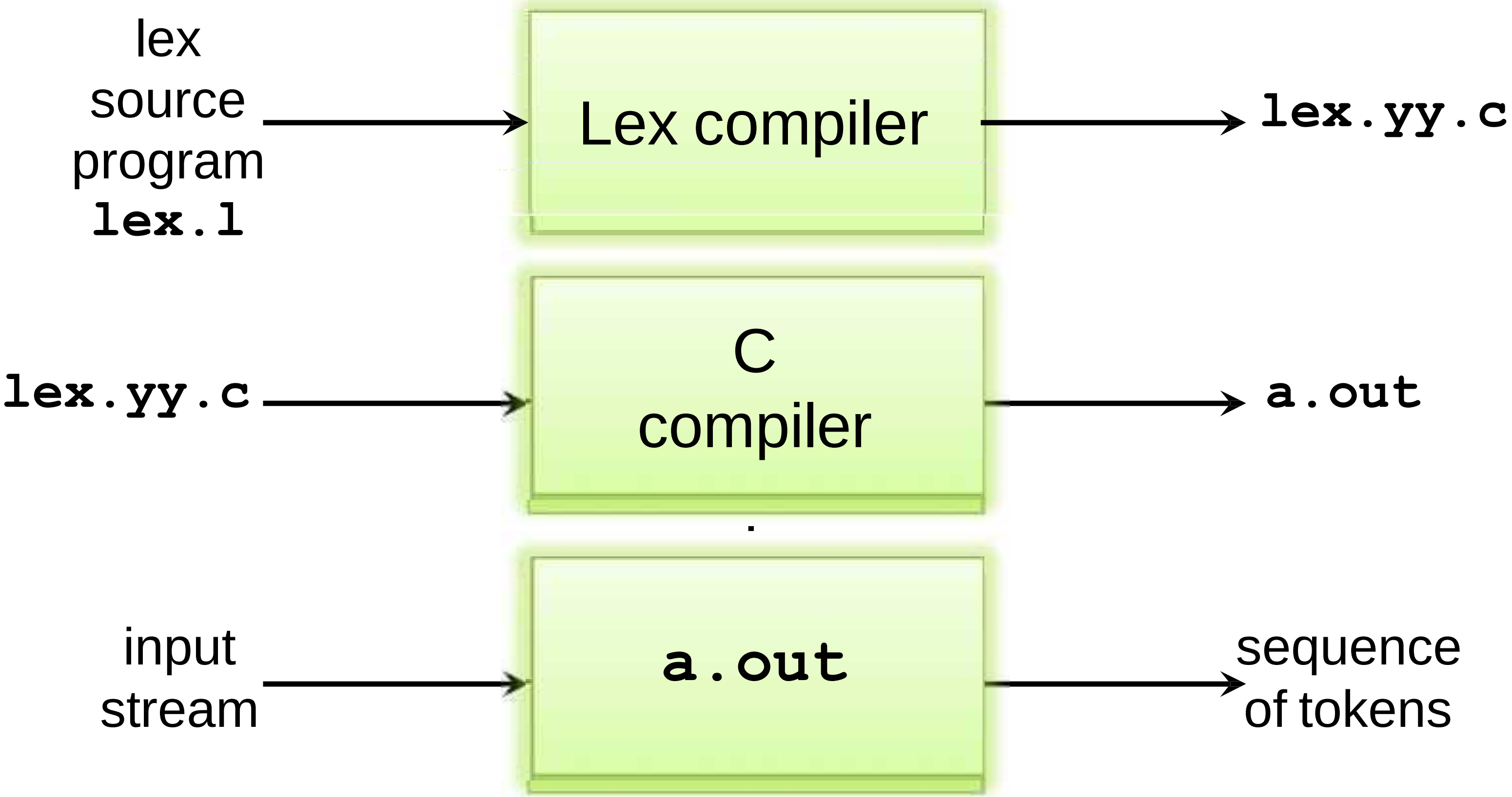
The set of descriptions is called lex specification

The token description are known as regular expressions

AUTOMATIC CONSTRUCTION OF EXICAL ANALYZER …….LEX

- Lex is a tool for creating lexical analyzers.
- Lexical analyzers *tokenize input streams.*
- Tokens are the *terminals of a language.*
- Regular expressions define *tokens* .

AUTOMATIC CONSTRUCTION OF EXICAL ANALYZER ……LEX



LEX SPECIFICATION

Lex Program Structure:

```
declarations
%%
translation rules
%%
auxiliary procedures
```

Name the file e.g. test.lex
Then, “`lex test.lex`” produces the file “`lex.yy.c`” (a C-program)

LEX SPECIFICATION

C declarations

```
%{
        /* definitions of all constants
        LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ... */
%}
```

declarations

```
......
letter  [A-Za-z]
digit   [0-9]
id      {letter}({letter}|{digit})*
......
%%
```

Rules

```
if      { return(IF);}
then    { return(THEN);}
{id}    { yylval = install_id(); return(ID); }
......
%%
```

Auxiliary

```
install_id()
{       /* procedure to install the lexeme to the ST */
```

AUTOMATIC CONSTRUCTION OF EXICAL ANALYZER ……LEX

- To run lex on a source file, use the command: *lex source.l*
- This produces the file lex.yy.c which is the C source for the

lexical analyzer.

- To compile this, use: *cc -o prog -O lex.yy.c -ll*

EXAMPLE OF LEX SPECIFICATION

```
%{
#include <stdio.h>
%}
%%
[0-9]+   { printf(“%s\n”, yytext); }
.|\n     { }
%%
main()
{ yylex();
}
```

Translation rules

Contains the matching lexeme

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```

EXAMPLE OF LEX SPECIFICATION

Translation rules

Regular definition

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim      [ \t]+
%%
\n         { ch++; wd++; nl++; }
^{delim}   { ch+=yyleng; }
{delim}    { ch+=yyleng; wd++; }
.          { ch++; }
%%
main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
```

EXAMPLE OF LEX SPECIFICATION

Translation rules

Regular definition

```
%{
#include <stdio.h>
%}
digit      [0-9]
letter     [A-Za-z]
id
{letter}({letter}|{digit})*
%%
{digit}+   { printf(“number: %s\n”,
yytext); }
{id}       { printf(“ident: %s\n”,
yytext); }
.          { printf(“other: %s\n”,
yytext); }
```

REGULAR EXPRESSIONS

- **`[xyz]`** match one character **`x`**, **`y`**, or **`z`** (use \ to escape –) **`[^xyz]`** match any character except **`x`**, **`y`**, and **`z`**
- **`[a-z]`** match one of **`a`** to **`z`**
- *r** closure (match zero or more occurrences)
- *r*+ positive closure (match one or more occurrences)
- *r*? optional (match zero or one occurrence)
- r_1r_2 match r_1 then r_2 (concatenation)

EXAMPLE OF LEX PROGRAM

```
 int num_lines = 0, num_chars = 0;

%%

\n     {++num_lines; ++num_chars;}
.      {++num_chars;}

%%

main( argc, argv )
int argc; char **argv;

    {
    ++argv, --argc;  /* skip over program name
```

EXAMPLE OF LEX PROGRAM

```
*/
    if ( argc > 0 )
       yyin = fopen( argv[0], "r" );
    else  yyin = stdin;
    yylex();
    printf( "# of lines = %d, # of chars =
%d\n",
            num_lines, num_chars );    }
```

EXAMPLE OF LEX PROGRAM

```
 %{ #include <stdio.h> %}
WS    [ \t\n]*

%%

[0123456789]+                printf("NUMBER\n");
[a-zA-Z][a-zA-Z0-9]*            printf("WORD\n");
{WS}                        /* do nothing */
.                           printf(“UNKNOWN\n“);
%%
```

EXAMPLE OF LEX PROGRAM

```
main( argc, argv )
int argc; char **argv;
    { ++argv, --argc;
     if ( argc > 0 ) yyin = fopen( argv[0], "r“);
     else  yyin = stdin;
    yylex();     }
```

- THANK YOU!!!!!!!!!!

My Blog : anandgharu.wordpress.com