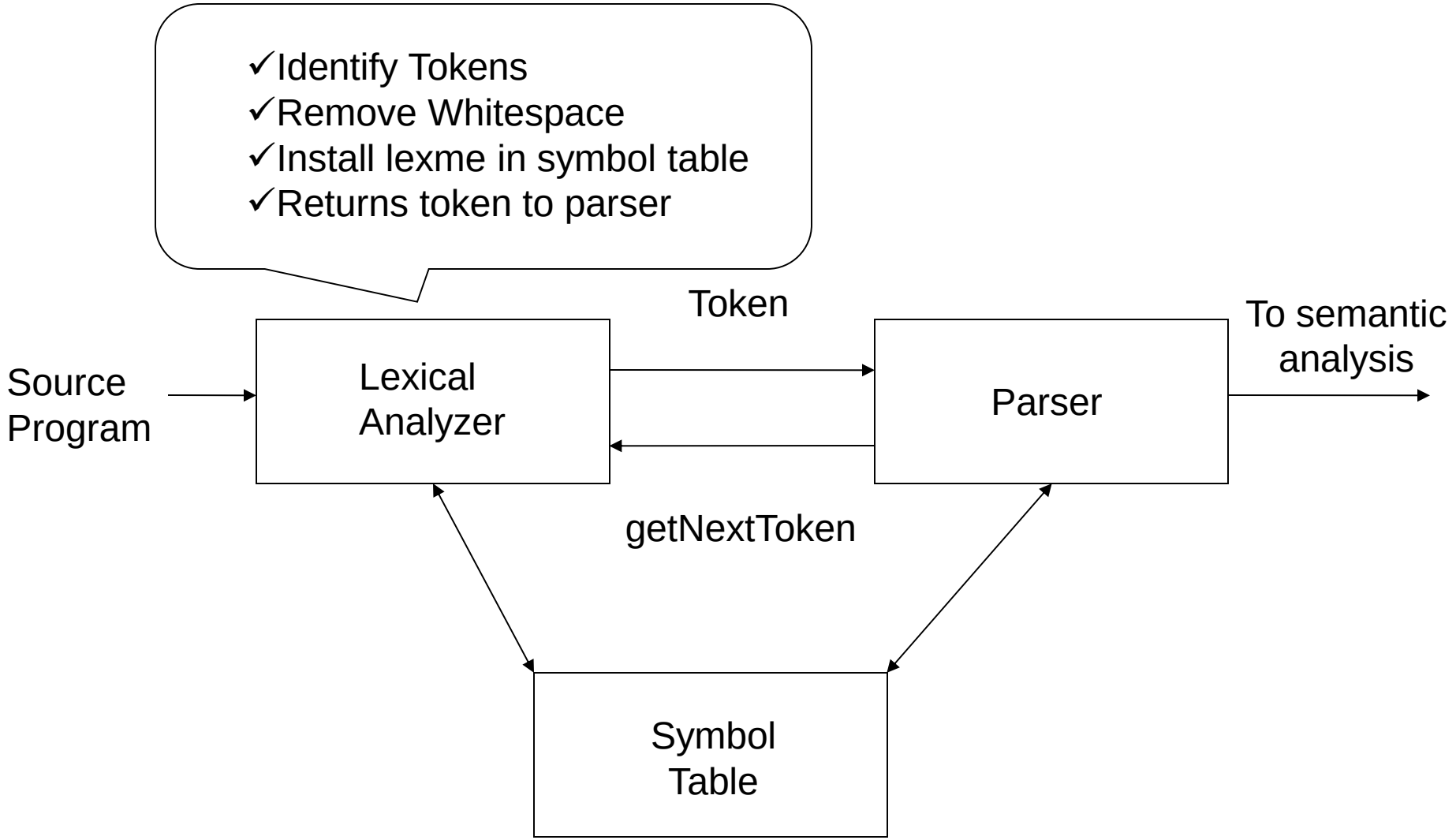
Lexical Analyzer

Using Lex

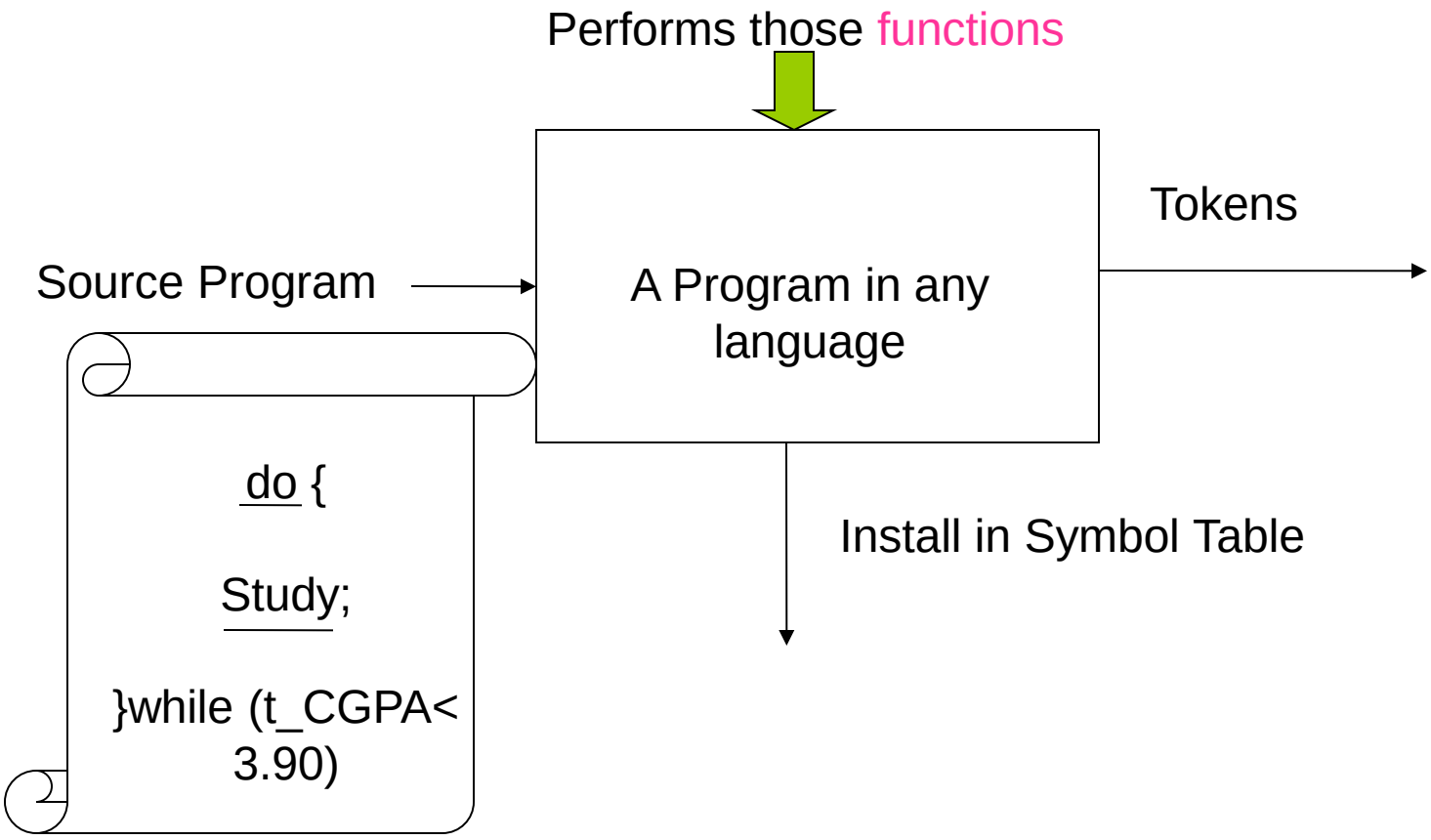
Lexical Analysis

- First phase of a Compiler
- Also called Scanning
- Scans the character stream of the Source program
- Groups them into meaningful sequences
  - Output: A sequence of token

Role of Lexical Analyzer



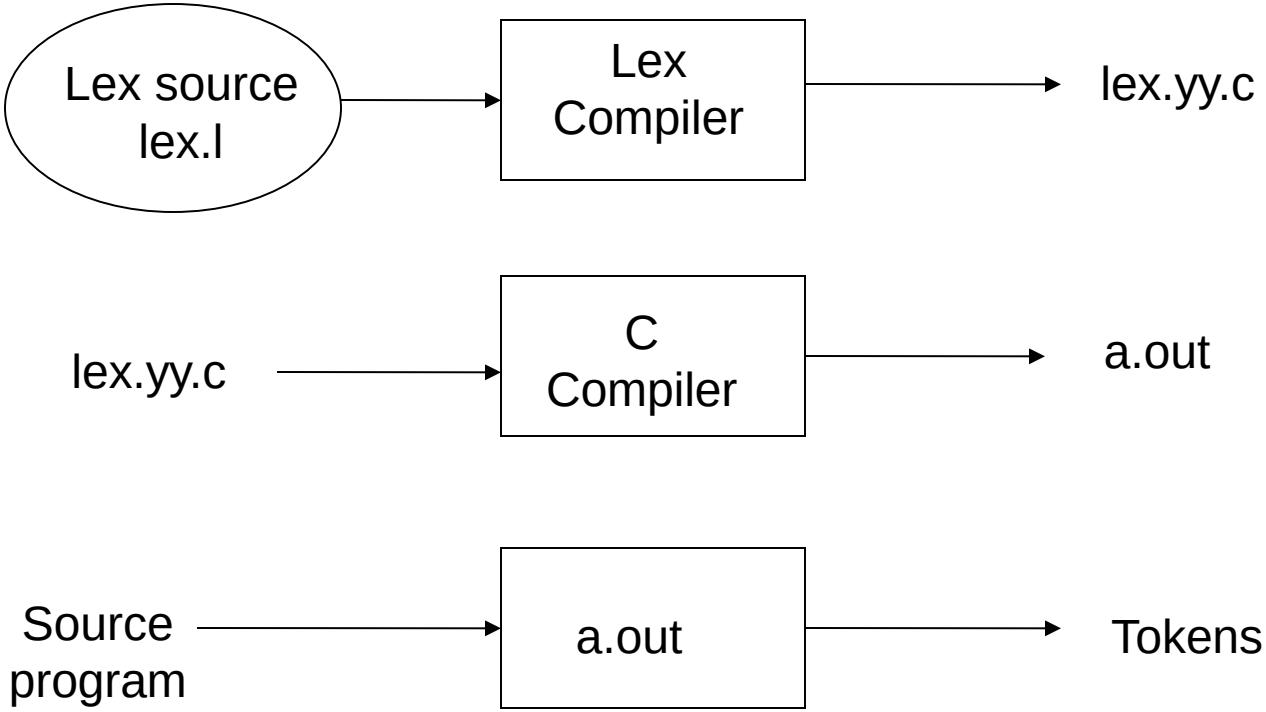
Lexical Analyzer



Lexical Analyzer

- No need to write the code
- Tools that produce the analyzer
  - Lex

Lex Tool


Lex source
lex.l
Lex
Compiler
lex.yy.c
lex.yy.c
C
Compiler
a.out
Source
program
a.out
Tokens

Token, Pattern, Lexeme

- Token: Set of strings that represent a particular construct in source language.
- Pattern: Rules that describe that string set
  - It matches each string in the set
- Lexeme: sequence of characters that is matched by a pattern for a token

Example

Token	Sample Lexemes	Pattern Description
WHILE	while	while
RELOP	<, <=, >, >=, <>, ==	< or <= or > or >= or <> or ==
ID	count, account, flag2	letter followed by letters and digits
C comment	/* hubi jabi/* aro habi jabi */	anything between /* and */
NUM	3.14, 3.2E+5, 5.9E-2	sequence of digits having fraction and exponent

Structure of Lex Programs

```
%{    #include<stdio.h>
       int Word_count;
// anything here is directly copied to lex.yy.c
%}                                  // regular definitions

Declarations                  //       token matching & actions
%%
Transition rules              // any other functions
%%
auxiliary functions
```

Transition rules

- Pattern { Action }

Regular expressions
to
Match the **token**

C code
to
Do the **functions**

Regular Expressions

- Specifies a set of strings to match
- One expression for each token pattern
- Some expression
  - [\t\n]   //for delimiter
  - [\t\n]+ // for white space
  - a(b)*   //a followed by zero or more occurrence of b

    //a, ab, abb, abbb

Actions

- Specify what to do if a rule matches a token
- Basically C code
- Examples

```
%%
[a-zA-z]        {
                    printf(“I found a letter”);
                }
[0-9]           {
                    printf(“I found a digit”)
                }
[  \t\n]        {
                    // actually I do nothing
                }
%%
```

Structure of Lex Programs

```
%{      #include<stdio.h>
        int Word_count;
%}

Declarations                    // regular definitions

%%

    [0-9]       {
                    printf("I found a digit");
                }
%%
auxiliary functions         // any other functions
```

regular definitions

- Give symbolic name to regular expressions
- Examples

```
delim       [  \t\n]
ws          {delim}+
digit       [0-9]
number      {digit}+
```

Complete Lex Source

```
%{
    #include<stdio.h>
    int digit_count = 0;
%}
delim           [  \t\n]
digit           [0-9]
%%
    {delim}+    {    }          //no action
    {digit}+    {  printf(“Here I found a digit”);  digit_count ++}
%%
Printf(“Total Count: %d”,word_count);
```

Assignment

- Write a lexical analyzer.
  - Ignore white space
  - Match all identifiers (keywords, variables etc)
    - Insert variables in symbol table
    - No need to insert keywords just show it in console
    - Exp: integer {printf("key: %s at line:%d\n","INTEGER",linenumber+1);}
  - Match all numbers and insert it in symbol table
  - Count line numbers

Assignment

Variables start with a letter or underscore (_)

- Ex : a, a9bc, _abc but not 8cde.

Relational operators are =, <>, < , <=, >=, >

- Insert the lexeme in symbol table and print the token RELOP
- Exp: {Relop} {printf("RELOP: %s at line%d\n",yytext,linenumber+1);Table.insert(yytext,"Relop");}

Assignment

- Addition operators are + - or
- Multiplier operators are * / div mod and
  - Insert lexeme and print token MULOP

Assignment

- Keywords to match
  - program
  - if
  - not
  - end
  - begin
  - else
  - then
  - do
  - while
  - function
  - Procedure
  - integer
  - real
  - var
  - oh
  - array
  - write

Print the corresponding token name and line no of occurring

Token name for parser is keyword name with capital letter