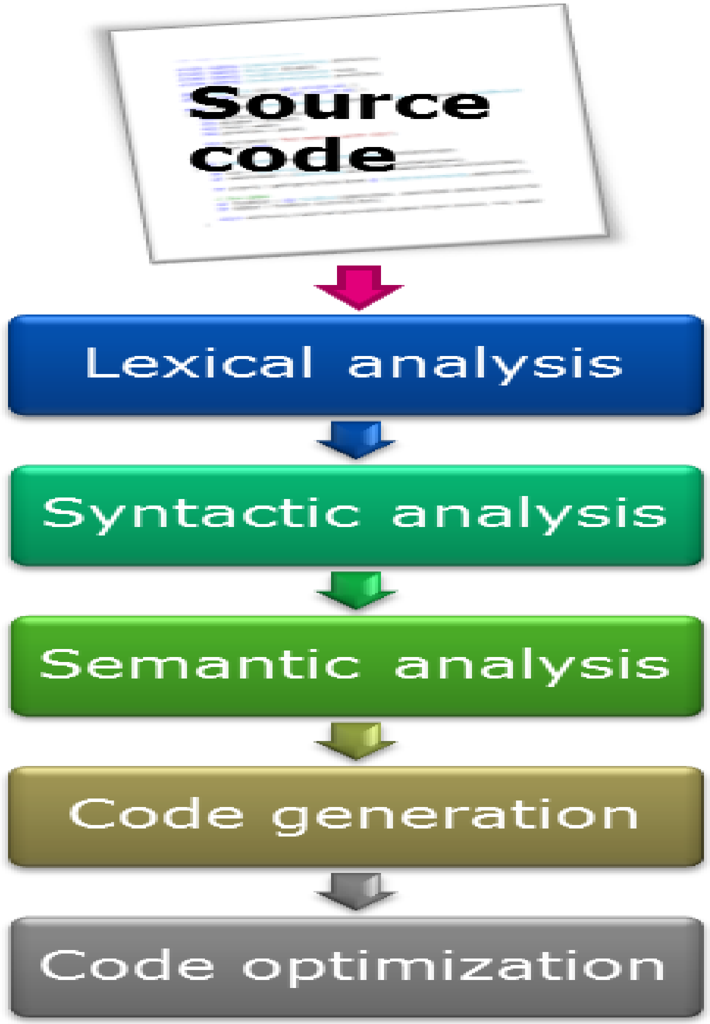
Compiler Design Laboratory (CS 653)

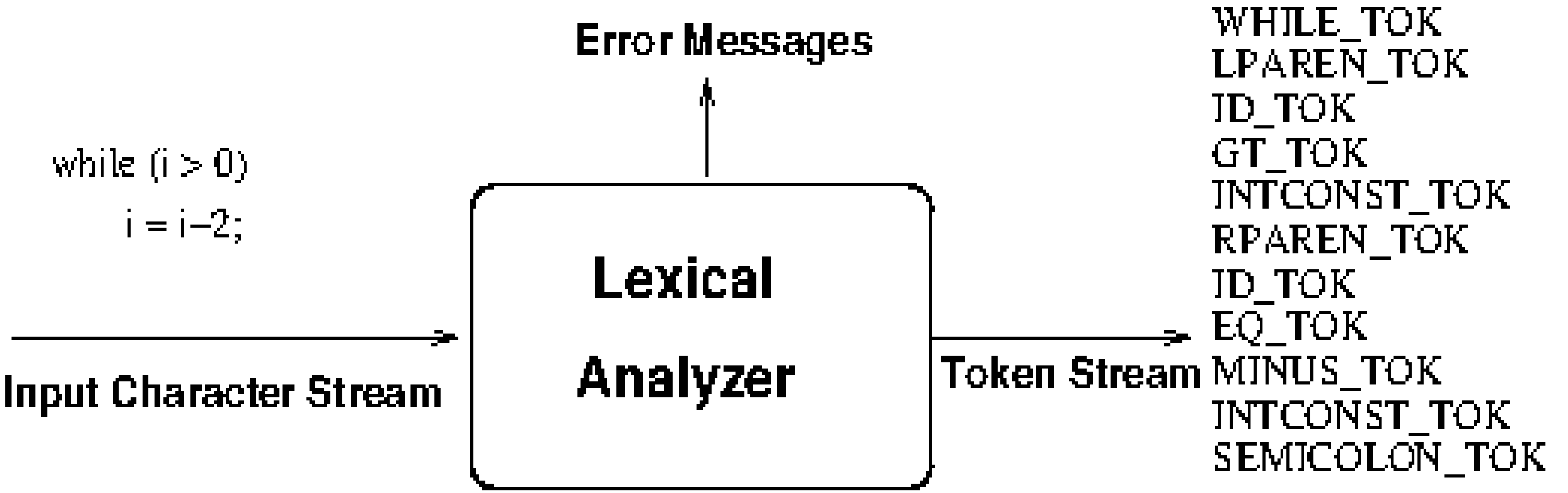
Sessional Study Materials

Manas Hira

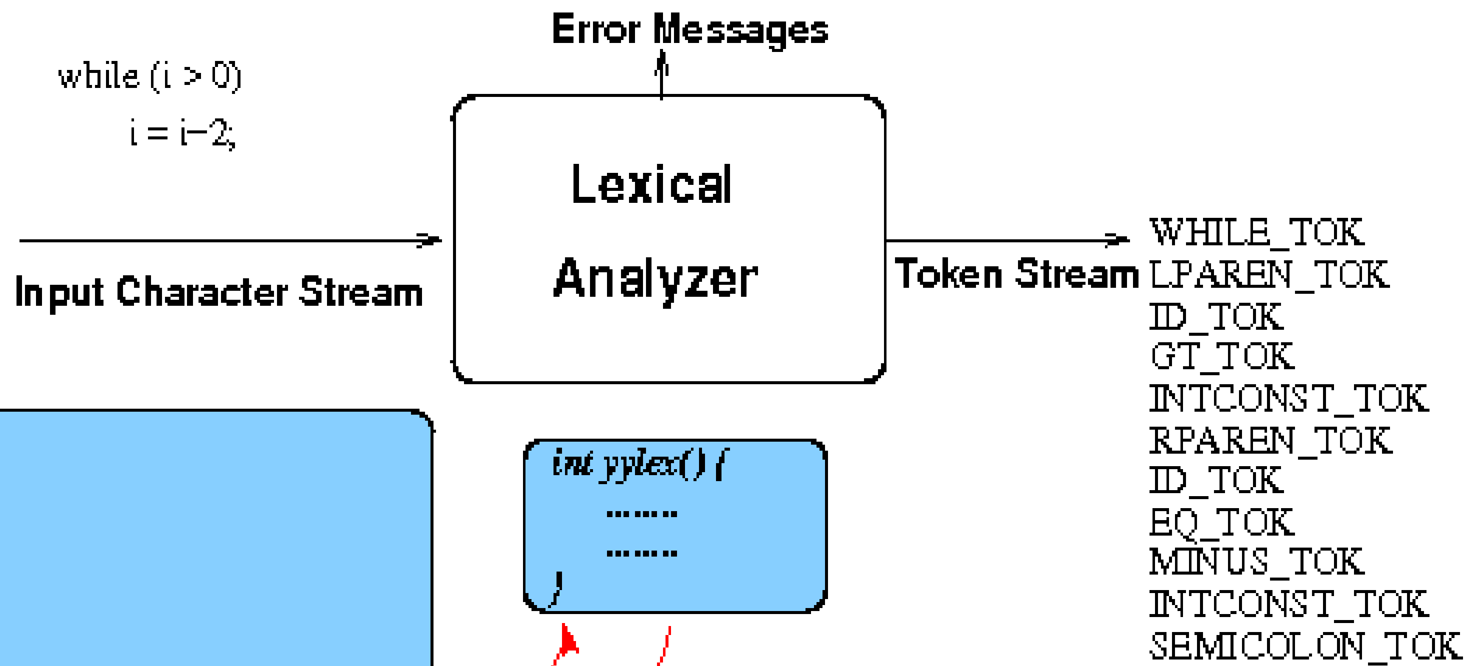
Phases of Compilation



What Lexical Analyzer does



Programmer's View




```
FILE *yyin;
char *yytext;
int yyleng;

main(int argc, char *argv[]) {
        int token;
        if(argc != 2) {
                fprintf(stderr, "Usage: %s <inputfile>\n");
                exit(1);
        } else {
                yyin = fopen(argv[1], "r");
                while (!feof(yyin)) {
                        token = yylex();
                        printf("%d\n", token);
                }
        fclose(yyin);
        }
}
```

Approaches for Lexical Analysis

- Hardcoded (ad-hoc) lexical analysis – Loop and Switch approach.
- Lexical analysis based on theory of Finite Automata

Loop and switch Approach

```
/* Single caharacter lexemes */
#define LPAREN_TOK '('
#define GT_TOK '>'
#define RPAREN_TOK ')'
#define EQ_TOK '='
#define MINUS_TOK '-'
#define SEMICOLON_TOK ';'
/*
.
.
.*/
/* Reserved words */
#define WHILE_TOK 256
/*
.
.
.*/
/* Identifier, constants..*/
#define ID_TOK 350
#define INTCONST 351
/*
.
.
.*/
```

Loop and switch Approach (contd.)

```
int yylex() {
        char ch;
        If (yyin == null) {
                yyin = stdin;
        }
        ch = getc(fp); // read next char from input stream
        while (isspace(ch)) // if necessary, keep reading til non-space char
                ch = getc(fp);
                // (discard any white space)

        switch(ch) {
                case ';': case ',': case '=': // ... and other single char tokens
                yytext[0] = ch;
                yyleng = 1;
                return ch;  // ASCII value is used as token value

                case 'A': case 'B': case 'C': // ... and other upper letters
                .
                .
                case 'a': case 'b': case 'c': // ... and other lower letters
                .
                .
}
```

Assignment Statement

Implement a hardcoded lexical analyzer for **exactly** the following types of tokens

- Arithmetic, Relational, Logical, Bitwise and Assignment Operators of C
- Reserved words: for, switch-case, if-else
- Identifier
- Integer Constants
- Parentheses, Curly braces

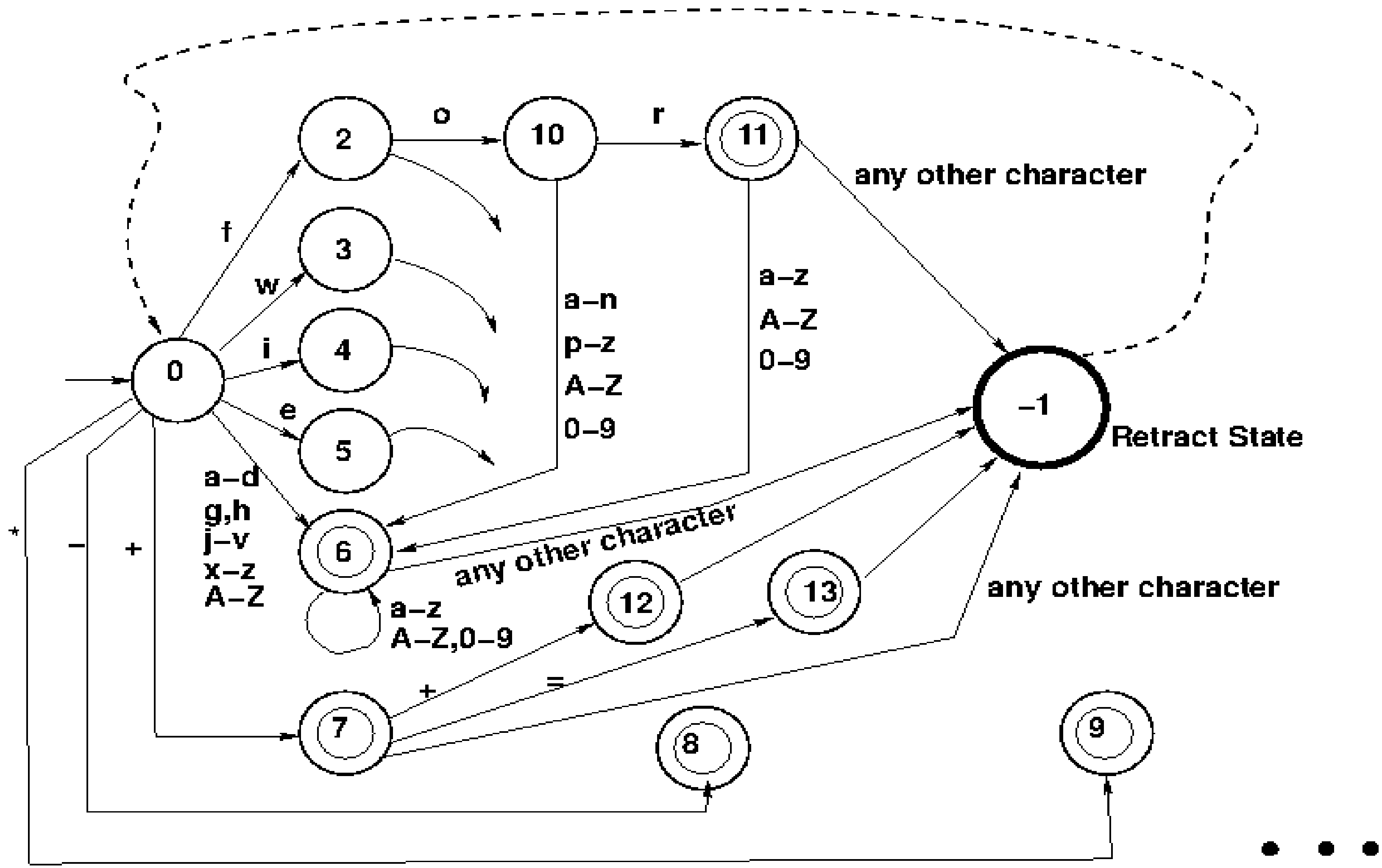
Follow the ideas of yytext, yyleng, etc as stated in the study material.
Preferably use C++ for implementation.

Lexical Analyzer based on Automata

Steps:

- Construct the Deterministic Finite Automaton (DFA) considering all the tokens of the Language (If the tokens are specified as regular expressions, then first construct the Non-Deterministic Finite Automata (NDFA) from the Regular Expressions and then convert the NDFA to an equivalent DFA.)
- Mechanically capture the DFA within the Lexical Analyzer.

Sample Finite Automata



Lexical Analyzer - Sample Code

```
int yylex() {
    int token;
    char c;
    int state;

    state = 0;
    while(1) {
        switch (state) {
            case 0:
                c = nextchar();
                if (c == 'f') state = 2;
                else if (c == 'w') state = 3;
                else if (c== 'i') state = 4;
                else if (c=='e') state = 5;
                else if ( (c >= 'a' && c <= 'd') || (c == 'g') || (c == 'h') || ...)
                    state = 6;
                else if (....
                break;
```

Lexical Analyzer - Sample Code

```
        case 2:
            c = nextchar();
            if (c == 'o') state = 10;
            else if....
            .
            .
            break;
        Case -1:
            retract();
        }
    }
}
```

Assignment Statement

Implement a lexical analyzer based on the theory of Finite Automata for **exactly** the following types of tokens

- Arithmetic, Relational, Logical, Bitwise and Assignment Operators of C
- Reserved words: for, while, if-else
- Identifier
- Integer Constants
- Parentheses, Curly braces

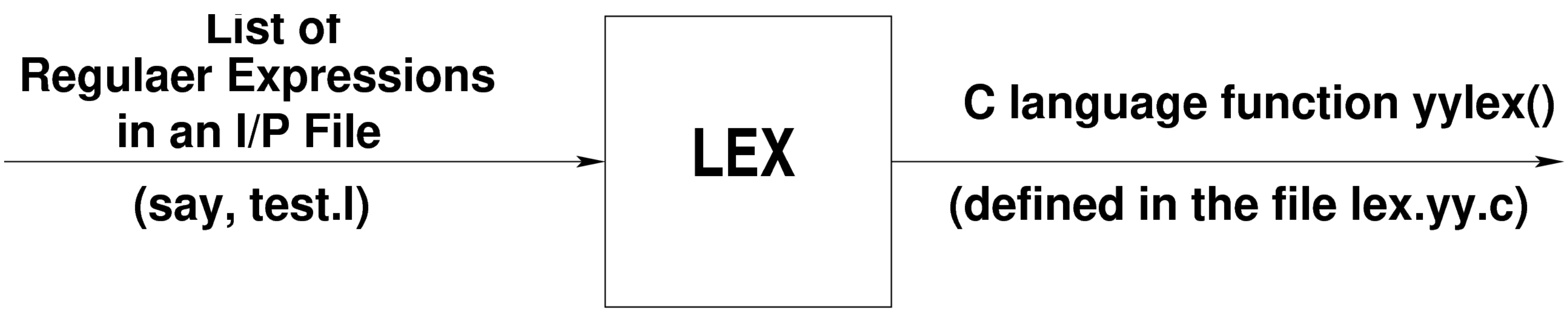
Follow the ideas of yytext, yyleng, etc as stated in the study material.
Preferably use C++ for implementation.

Lexical Analyzer using LEX

Observations:

- Specifying the tokens in Regular Expressions (RE) is easier than using DFA or NDFA.
- Construction of an NDFA from an RE and thereof converting the NDFA to an equivalent DFA is mechanical (algorithmic).
- Construction of a Lexical Analyzer for a given DFA is again mechanical (algorithmic).
- As a result, given RE's for tokens, construction of corresponding Lexical Analyzer too is mechanical and can be done by a program.
- **LEX is exactly one such a program that produces a Lexical Analyzer from the given RE's.**

LEX in a nutshell



- ***yylex()*** reads from the I/P stream ***FILE *yyin*** to match strings against RE's of test.l
- When ***yylex()*** finds a match for an RE, it performs the "***desired task***" as specied in test.l for that RE.
- ***yylex()*** produces output, if any, to ***FILE * yyout***

What ***yylex()*** is to do

Expected Behaviour	
String in I/P Stream (Lexeme)	**To be done (action) by *yylex()***
int	Returns INT_TOK
while	Returns WHILE_TOK
no_f_students1	Returns ID_TOK
. . .	. . .

Specifying the Behaviour (in test.l)	
Token Specifier (Regular Expression)	**Corresponding Action**
int	{ return (INT_TOK); }
while	{ return (WHILE_TOK); }
[a-zA-Z_][a-zA-Z0-9_]*	{ return (ID_TOK) }
. . .	. . .

test.l should contain...

Structure of *test.l*	**Note**
Definition Section %% Rule Section %% User Defined Functions	• Blue Section is Optional • Red Section is Required • Substitutions, Code, and Start States are kept in Definition Section • %{ <*some text*> • %}, if in Definition Section, will put *some text* at the beginning of *lex.yy.c* • *digit [0-9]*, when in Definition Section, {digit}+ can be used in Rule Section. • Each Rule should be in a separate line • A Rule starts at the beginning of a line • A Rule: <*RE*> <*White Space*> <*Action*>

How ***yylex()*** matches the I/P stream

- When yylex() runs, it scans the input stream looking for strings matching any of the REs
- If the current input can be matched by several REs, then the ambiguity is resolved by choosing the RE -

  → making the longest match

  → occurring first in the LEX source file (test.l)
- Once the match is determined

  → the corresponding text is available in the global character pointer ***yytext***

  → its length in ***yyleng*** and

  → the current line number in ***yylineno***

  → the action corresponding to the RE is then executed

  → and then, if the action is not return or exit,
  the remaining input is scanned for another match
- Unmatched input characters are copied to ***yyout*** (***stdout*** by default)

How to write an RE for LEX

Operators: " \ ] ^ - ? . * | ( ) $ / { } % < >

Regular Expression (RE) for LEX	Matching String
α ∈ {a, b, ..., z, A, B, ..., Z, 0, 1, ..., 9}	The symbol itself
.	Any character other than newline (\n)
[character sequence] Eg., • [acbg] • [a-p] • [A-Z0-5]	Character Set/Class - Any character from the sequence. Eg., • 'a' / 'c' / 'b' / 'g' • any small letter in the range a to p • for what?
[^character set] • [^a-zA0-9]	Any character not in the set • Any non-alpha non-digit character
“α” where α is a stringr	The string α
\α where α is an operator (escaping)	The operator character α itself
\n \t	Newline character Tab character
αß where α and ß are REs	Concatenation of strings matching α and ß
(α) where α is an RE	String matching α

How to write an RE for LEX(contd.)

Operators: " \ ] ^ - ? . * | ( ) $ / { } % < >

Regular Expression (RE) for LEX	**Matching String**
α|ß where α and ß are RE's	String matching α and/or ß
α* where α is an RE	Zero or more occurrences of the string matching α
α+ where α is an RE	One or more occurrences of the string matching α
α? where α is an RE	Zero or one occurrence of the string matching α
α{n1,n2} where α is an RE, n1 ≤ n2	n1 to n2 occurrences of the string matching α
α$ where α is an RE	String matching α at end of line (before \n)
α/ß where α and ß are REs	String matching α before string matching ß

Examples of REs for LEX

Regular Expression (RE) for LEX	**Matching String**
xyz"++"	xyz++
[ab]	a or b
[^abc]+	String of length 1 or more containing characters other than a or b or c
[a^b]	a, ^, b
ab?c	ac or abc
[A-Za-z_][A-Za-z0-9_]*	1 alpha or underscore followed by 0 or more alphanumeric or underscore (identifier?)
[\t\n]+	whitespace

LEX defines a few macros in lex.yy.c

ECHO	Copies yytext to the scanner's output
REJECT	Scanner proceeds on to the "second best" rule which matched the input (or a prefix of the input)
yymore()	Next time scanner matches a rule, the corresponding lexeme should be appended onto the current value of yytext rather than replacing it.
yyless()	
yyterminate()	
input()	
output()	
unput()	

Assignment Statement

Using LEX implement a Lexical Analyzer that returns tokens for the following subset of C language.

- main and user-defined function definitions
- int, char and float data types
- Variable definition
- Arithmetic, Relational and Logical Expressions
- Assignment Statement
- if-else and switch-case constructs
- while and for constructs
- Function Call

The Lexical Analyzer should manage a symbol table as discussed in the theory class.