

**Department of Computer Science and Engineering**

**Compiler Design Lab (CS 306)**

**Tools used in lab**

* Some programs will be implemented using C and some using LEX and YACC tools.
* **Execution of C programs**: You can use any editor and compiler which you are comfortable. I use Code::Blocks (<http://www.codeblocks.org/downloads>) for explaining few codes in C.
* **LEX/YACC** 
  + If you are using **Linux** install the packages using the following commands.
    - $sudo apt-get install flex
    - $sudo apt-get install bison
    - Refer the YouTube link for installation guidance

https://www.youtube.com/watch?v=FTFYA\_TOGXs

* + If you are using **Windows**: The following YouTube link has the instructions of installation. The software download link is available in the description.

https://www.youtube.com/watch?v=AI-jwky\_mqM

**List of lab exercises**

**Week 1: Language recognizer**

1. Write a program in C that recognizes the following languages.
   1. Set of all strings over binary alphabet containing even number of 0’s and even number of 1’s.
   2. **Lab Assignment:** Set of all strings ending with two symbols of same type.

**Week 2: Implementation of Lexical analyzer using C**

1. Implement lexical analyzer using C for recognizing the following tokens:

* A minimum of 10 keywords of your choice
* Identifiers with the regular expression : letter(letter | digit)\*
* Integers with the regular expression: digit+
* Relational operators: <, >, <=, >=, ==, !=
* Storing identifiers in symbol table.
* Using files for input and output.

**Week 3: Introduction to LEX tool**

1. Implement the following programs using Lex tool
   1. Identification of Vowels and Consonants
   2. count number of vowels and consonants
   3. Count the number of Lines in given input
   4. Recognize strings ending with 00
   5. Recognize a string with three consecutive 0’s

**Week 4: Implementation of lexical analyzer using LEX**

1. Implement lexical analyzer using LEX for recognizing the following tokens:

* A minimum of 10 keywords of your choice
* Identifiers with the regular expression : letter(letter | digit)\*
* Integers with the regular expression: digit+
* Relational operators: <, >, <=, >=, ==, !=
* Ignores everything between multi line comments (/\* …. \*/)
* Storing identifiers in symbol table
* Using files for input and output.

**Week 5: Lexical Analyzer**

1. **Lab Assignment:**

Consider the following mini Language, a simple procedural high-level language, only operating on integer data, with a syntax looking vaguely like a simple C crossed with Pascal. The syntax of the language is definedby the following BNF grammar:

<program> ::= <block>

<block> ::= { <variabledefinition> <slist> } | { <slist> }

<variabledefinition> ::= int<vardeflist>;

<vardeflist> ::= <vardec> | <vardec>, <vardeflist>

<vardec> ::= <identifier> | <identifier> [ <constant> ]

<slist> ::= <statement> | <statement>; <slist>

<statement> ::= <assignment> | <ifstatement> | <whilestatement> | <block> | <printstatement> | <empty>

<assignment> ::= <identifier> = <expression> | <identifier> [ <expression> ] = <expression>

<ifstatement> ::= <bexpression> then <slist> else <slist> endif | if <bexpression> then <slist> endif

<whilestatement> ::= while <bexpression> do <slist> enddo

<printstatement> ::= print ( <expression> )

<expression> ::= <expression> <additionop> <term> | <term> | addingop> <term>

<bexpression> ::= <expression> <relop> <expression>

<relop> ::= < | <= | == | >= | > | !=

<addingop> ::= + | -

<term> ::= <term><mulitop> <factor> | <factor>

<multop> ::= \* | /

<factor> ::= <constant> | <identifier> | <identifier> [ <expression> ] | ( <expression> )

<constant> ::= <digit> | <digit> <constant>

<identifier> ::= <identifier> <letterordigit> | <letter>

<letterordigit> ::= <letter> | <digit>

<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<empty> has the obvious meaning

Comments (zero or more characters enclosed between the standard C / Java style comment brackets /\*...\*/) can be inserted. The language has rudimentary support for 1-dimensional arrays. The declaration int a[3] declares an array of three elements, referenced as a[0], a[1] and a[2]. Note also that you should worry about the scoping of names.

A simple program written in this language is:

{ int a[3], t1, t2;

t1 = 2; a[0] = 1; a[1] = 2; a[t1] = 3;

t2 = -(a[2] + t1 \* 6)/ a[2] -t1);

if t2 > 5 then

print(t2);

else {

int t3;

t3 = 99;

t2 = -25;

print(-t1 + t2 \* t3); /\* this is a comment on 2 lines \*/

} endif

}

Design a Lexical analyser for the above language. The lexical analyser should ignore redundant spaces, tabs, and newlines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.

**Week 6: Recursive Descent Parser**

1. Implement Recursive Descent Parser for the Expression Grammar given below.

E 🡪 TE’

E’🡪 +TE’ | ͼ

T 🡪 FT’

T’🡪 \*FT’ | ͼ

F 🡪 (E) | i

1. **Lab Assignment:** Construct Recursive Descent Parser for the grammar

G = ({S, L}, {(, ), a, ,}, {S 🡪 (L) | a ; L🡪 L, S | S}, S) and verify the acceptability of the following strings:

1. (a,(a,a))
2. (a,((a,a),(a,a)))

You can manually eliminate Left Recursion if any in the grammar.

**Week 7: Predictive parser**

1. Write a C program for the computation of FIRST and FOLLOW for a given CFG

**Week 8: Predictive Parser**

1. Implement non-recursive Predictive Parser for the grammar

S -> aBa

B -> bB | ε

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | b | $ |
| S | S🡪aBa |  |  |
| B | B🡪ε | B🡪bB |  |

1. **Lab Assignment:** Implement Predictive Parser using C for the Expression Grammar

E 🡪 TE’

E’🡪 +TE’ | ε

T 🡪 FT’

T’🡪 \*FT’ | ε

F 🡪 (E) | d

**Week 9: Shift Reduce Parser**

1. Implementation of Shift Reduce parser using C for the following grammar and illustrate the parser’s actions for a valid and an invalid string.

E🡪E+E

E🡪E\*E

E🡪(E)

E🡪d

1. **Lab Assignment:** Implementation of Shift Reduce parser using C for the following grammar and illustrate the parser’s actions for a valid and an invalid string.

S –> 0S0 | 1S1 | 2

**Week 10: LALR Parser**

1. Implement LALR parser using LEX and YACC for the following Grammar:

E 🡪 E+T |T

E’🡪 T\*F | F

F 🡪 (E) | d

1. **Lab Assignment:** Implement LALR parser using LEX and YACC for the following Grammar by specifying proper precedence for operators:

E 🡪 E+E | E-E | E\*E | E/E | -E | (E) | digit

**Week 11:Intermediate code generation**

1. Generate quadruples for given arithmetic expression using LEX and YACC.

**Week 12:** **Intermediate code generation**

1. Generate 3-address code for if statement using LEX and YACC.
2. **Lab Assignment:** Generate 3-address code for while statement using LEX and YACC.

**Week 13: Code optimization**

1. Implement constant propagation and folding using C for a given set of intermediate instructions.
2. **Lab Assignment:** Write a program to eliminate dead code

**Week 14: Code optimization**

1. Write a program to eliminate common sub expressions
2. **Lab Assignment:** Write a program to perform loop unrolling

**Week 15: Code Generation**

1. Generate machine code from the abstract syntax tree generated by the parser. The following instruction set may be considered as target code. The following is a simple register-based machine, supporting a total of 17 instructions. It has three distinct internal storage areas. The first is the set of 8 registers, used by the individual instructions as detailed below, the second is an area used for the storage of variables and the third is an area used for the storage of program. The instructions can be preceded by a label. This consists of an integer in the range 1 to 9999 and the label is followed by a colon to separate it from the rest of the instruction. The numerical label can be used as the argument to a jump instruction, as detailed below. In the description of the individual instructions below, instruction argument types are specified as follows:

R specifies a register in the form R0, R1, R2, R3, R4, R5, R6 or R7 (or r0, r1, etc.).

L Specifies a numerical label (in the range 1 to 9999).

V Specifies a “variable location” (a variable number, or a variable location pointed to by a register -see below).

A Specifies a constant value, a variable location, a register, or a variable location pointed to by a register (an indirect address). Constant values are specified as an integer value, optionally preceded by a minus sign, preceded by a #symbol. An indirect address is specified by an @followed by a register. So, for example, an A-type argument could have the form 4 (variable number 4), #4 (the constant value 4), r4 (register 4) or @r4 (the contents of register 4 identifies the variable location to be accessed).

The instruction set is defined as follows:

LOAD A,R

loads the integer value specified by A into register R.

STORE R,V

stores the value in register R to variable V.

OUT R

outputs the value in register R.

NEG R

negates the value in register R.

ADD A,R

adds the value specified by A to register R, leaving the result in register R.

SUB A,R

subtracts the value specified by A from register R, leaving the result in register R.

MUL A,R

multiplies the value specified by Aby register R, leaving the result in register R.

DIV A,R

divides register R by the value specified by A, leaving the result in register R.

JMP L

causes an unconditional jump to the instruction with the label L.

JEQ R,L

jumps to the instruction with the label L if the value in register R is zero.

JNE R,L

jumps to the instruction with the label L if the value in register R is not zero.

JGE R,L

jumps to the instruction with the label L if the value in register R is greater than or equal to zero.

JGT R,L

jumps to the instruction with the label L if the value in register R is greater than zero.

JLE R,L

jumps to the instruction with the label L if the value in register R is less than or equal to zero.

JLT R,L

jumps to the instruction with the label L if the value in register R is less than zero.

NOP

is an instruction with no effect. It can be tagged by a label.

STOP

stops execution of the machine. All programs should terminate by executing a STOP instruction