SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

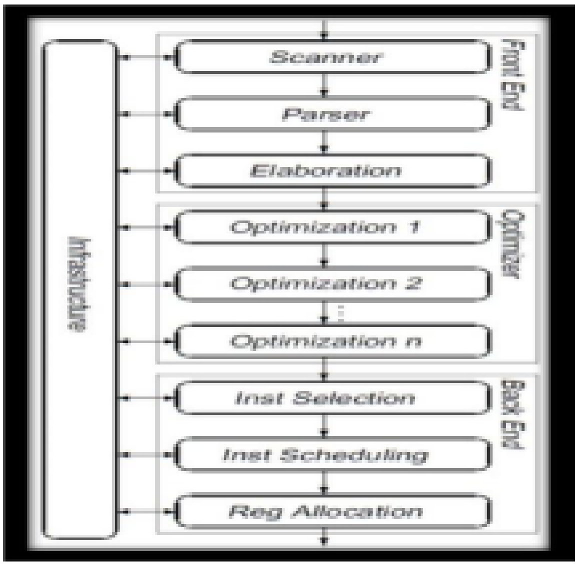
(U/S 3 of the UGC Act,1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

SCHOOL OF COMPUTING

Course Code : CSE403

Course Name : COMPILER ENGINEERING LABORATORY - MANUAL



Name :

Register No :

Year :

Section :

CourseObjective:

The learners will be able to design and implement the following phases of compiler like scanning and parsing, ad-hoc syntax directed translation, code generation and code optimization for any formal language usingLex andYACC tools.

List of Experiments:

1. Develop a scanner using LEX for recognizing the tokens in a given C program.
2. Develop a parser for all branching statements of 'C' programming language using LEX& YACC. 3.

Develop a parserfor alllooping statements of 'C' programming language using LEX & YACC.

4. Develop a parser for complex statements in 'C' programming language with procedure calls and array references using LEX & YACC.

5. Develop a program to find the FIRST and FOLLOW sets for a given Context Free Grammar.

6. Extend the outcome of experiment 5 to implement a LL(1) parser in C or Java to decide whether the input string is valid or not.

7. Implement a LR(1) bottom up parser in C or Java to decide whether the input string is valid or not (Context- Free Grammar, Action andGOTO tables are suppliedas inputs).

8. Identify the structure of if-else statements in 'C' programming language that can be transformed into their equivalent switch-case statements. Design and implement the Adhoc syntax directed translation systemforthe identified case.

9. Use LEX and YACC to create two translators that would translate the given input (compound expression) into three-address and postfix intermediate codes. The input and output of the translators should be a file.

10. Write an optimizer pass in C or Java that does common-sub expression elimination on the three address intermediate code generated in the previous exercise.

11. Use LEX & YACC to write a back end that traverses the three address intermediate code and generates x86 code.

12. Implement Local List Scheduling Algorithm.

Software/ Tools used: C, Java, Python, Lex, Yacc and JFLAP

1. Develop a scanner using LEX for recognizing the token types in a given C program.

Objective: learner will be able to design regular expressions for each token type

Prerequisite: structure of a lex program

Pre-lab exercise: Lex program to count the number of words, lines and characters in the input

Procedure:

1. Construct the regular expressions for each token type- identifiers, keywords, operators, constant, separators, and special symbols
2. Incorporate these regular expressions as per the priority in the lex module.

Additional Exercises:

1. Construct a lexical analyzer for Java Constructs
2. Construct a lexical analyzer for Python Constructs

2. Develop a parser for all branching statements of ‘C’ programming language using LEX & YACC

Objective: learner will be able to

1. Develop parsers to a programming construct along with a supporting scanner.

2. Design and construct complex parsers by combining these simple parsers
3. Develop the skill to design and construct context free grammar for a programming construct

Prerequisite: Constructing Grammar for a programming construct in Backus Naur Form and YACC structure

Pre-lab exercise: Compiling a Lex & YACC program

Procedure:

1. Consider a branching construct and 5 to 10 example statements of the same.
2. Design the grammar and derive the above statements manually.
3. Write the appropriate Lex and YACC specification in rule section of Lex and YACC based on the grammar.
4. Test rigorously with the sample statements Lex alone first and then Lex & YACC
5. Adjust the Lex & YACC specifications according to testing results.
6. Repeat the steps 1 to 5 to include other branching statements and styles

Additional Exercises:

1. Implement parser for if else ladder
2. Implement parser for matching parenthesis

3. Develop a parser for all looping statements of 'C' programming language using LEX & YACC

Objective: learner will be able to

1. Develop parsers to a programming construct along with a supporting scanner.
2. Design and construct complex parsers by combining these simple parsers

Prerequisite: Develop the skill to design and construct context free grammar for a programming construct

Pre-lab exercise: Constructing Grammar for a programming construct in Backus Naur Form. **Procedure:**

1. Consider a looping construct and 5 to 10 example statements of the same.
2. Design the grammar and derive the above statements manually.
3. Write the appropriate Lex and YACC specification in rule section of Lex and YACC based on the grammar.
4. Test rigorously with the sample statements Lex alone first and then Lex & YACC
5. Adjust the Lex & YACC specifications according to testing results.
6. Repeat the steps 1 to 5 to include other branching statements and styles

Additional Exercises:

1. Implement parser for nested loops
2. Implement parser for class and structure statements

4. Develop a parser for complex statements in 'C' programming language with procedure calls and array references using LEX & YACC

Objective: learner will be able to design and construct complex parsers

Prerequisite: Develop the skill to design and construct context free grammar for a programming construct

Pre-lab exercise: Constructing Grammar for a programming construct in Backus Naur Form. **Procedure:**

1. Consider 5 to 10 complex statements with procedure calls and array references
2. Design the grammar and derive the statements manually.
3. Write the appropriate Lex and YACC specification in rule section of Lex and YACC based on the grammar.
4. Test rigorously with the sample statements Lex alone first and then Lex & YACC
5. Adjust the Lex & YACC specifications according to testing results.
6. Repeat the steps 1 to 5 to include other complex statements and styles.

5. Develop a program to find the FIRST and FOLLOW sets for a given Context Free Grammar.

Objective: learner will be able to find the FIRST and FOLLOW sets for a given Context Free Grammar
Prerequisite: Definition of FIRST and FOLLOW
Pre-lab exercise: substring extraction
Procedure:
Algorithm for computing FIRST sets:

```
for each α ϵ (T U eof Uε) do;
        FIRST(α) ← ϕ;
end;
for each A ϵ NT do;
        FIRST(A) ← ϕ;
end;
while (FIRST sets are still changing) do;
        for each p ϵ P, where p has the form A→β do;
                if β is β1 β 2 …β k, where βi ϵ T U NT , then begin;
                        rhs ← FIRST(β1) – {ε};
                        i ← 1;
                        while (ε ϵ FIRST(βi) and i ≤ k-1) do;
                                rhs ← rhs U (FIRST(β(i+1)-{ε}) ;
                                i ← i + 1;
                        end;

                end;
                if i = k and ε ϵ FIRST(βk)
                        then rhs ← rhs U {ε};
                FIRST(A) ← FIRST(A) U rhs;

        end;

end;
```

Algorithm for computing FOLLOW sets:

```
for each A є NT do;
        FOLLOW(A) ← ϕ ;
end;
FOLLOW(S) ← {eof};
while (FOLLOW sets are still changing) do;
        for each p є P of the form A→β_1 β_2 …β_k do;
                TRAILER ← FOLLOW(A);
                for i ← k down to 1 do;
                        if β_i є NT then begin;
                                FOLLOW(β_i) ← FOLLOW(β_i) ∪ TRAILER;
                                if ε є FIRST(β_i )
                                        then TRAILER ← TRAILER ∪ (FIRST(β_i )- ε)
                                        else TRAILER ← FIRST(β_i);
                        end;
                        else TRAILER ← FIRST(β_i);


                end;
        end;

end;
```

Additional Exercises:

1. Program to find Augmented FIRST sets.

6. Extend the outcome of Experiment 5 to implement a LL(1) parser in C or Java to decide whether the input string is valid or not.

Objective: The learner will be able to develop a LL(1) parser to decide if the given input string is valid or not.

Prerequisite: two-dimensional array manipulation, string searching and substring extraction

Pre-lab exercise: Programs on Array processing, string processing, $FIRST^{+}$ computation.

$$FIRST^{+}(A \rightarrow \beta) = \begin{cases} FIRST(\beta), & \text{if } \varepsilon \notin FIRST(\beta) \\ FIRST(\beta) \cup FOLLOW(A), & \text{otherwise} \end{cases}$$

Procedure:

Algorithm for LL(1) table construction:

build FIRST, FOLLOW, and $FIRST^{+}$ sets;

```
for each nonterminal A do;
    for each terminal w do;
        Table[A ,w] ← error;
    end;
    for each production p of the form A→β do;
        for each terminal w ϵ FIRST+( A→β) do;
            Table[A ,w] ← p;
        end;
        if eof ϵ FIRST+( A→β)
            then Table[A ,eof] p;
    end;
end;
```

Algorithm for LL(1) parsing:

```
word ← NextWord( );
push eof onto Stack;
push the start symbol, S, onto Stack;
focus ← top of Stack;
loop forever;
    if (focus = eof and word = eof)
        then report success and exit the loop;
    else if (focus ϵT or focus = eof) then begin;
        if focus matches word then begin;
            pop Stack;
            word ← NextWord( );
        end;
    else report an error looking for symbol at top of stack;
    end;
    else begin; /* focus is a nonterminal */
        if Table[focus,word] is A→ B1B2… Bk then begin;
            pop Stack;
            for i ← k to 1 by -1 do;
                if (Bi ≠ ε)
                    then push Bi onto Stack;
            end;
        end;
```

else report an error expanding focus;

end;

focus ← top of Stack;

end;

Additional Exercises:

1. Program to find if the given grammar is LL(1) or not.

7. Implement a LR(1) bottom up parser in C or Java to decide whether the input string is valid or not (Context- Free Grammar, Action and GOTO tables are supplied as inputs)

Objective: The learner will be able to develop a LR(1) parser to decide if the given input string is valid or not.

Prerequisite: Closure and Move algorithms, LR(1) items

Pre-lab exercise: Programs on Closure and Move computation, LR(1) items creation.

Procedure:

Algorithm for LR(1) Bottom-up parser:

```
push $;
push start state, s0;
word ← NextWord( );
while (true) do;
    state ← top of stack;
    if Action[state,word] = "reduce A→β" then begin;
        pop 2x|β| symbols;
        state ← top of stack;
        push A;
        push Goto[state, A];
    end;
    else if Action[state,word] = "shift si" then begin;
        push word;
        push si ;
        word ← NextWord( );
    end;
    else if Action[state,word] = "accept"
        then break;
    else Fail( );
end;
```

report success; /* executed break on “accept” case */

Additional Exercises:

1. Program to find if the given grammar is LR(1) or not.
2. Program to implement LALR(1) parser.
3. Program to implement SLR(1) parser.

8. Identify the structure of if-else statements in ‘C’ programming language that can be transformed into their equivalent switch-case statements. Design and implement the Ad-hoc syntax directed translation system for the identified case.

Objective: learner will be able to

1. Develop parsers to a programming construct along with a supporting scanner.
2. Design and construct complex parsers by combining these simple parsers
3. Develop the skill to design and construct context free grammar for a programming construct andperform ad-hoc SDT

Prerequisite: Constructing Grammar for a programming construct in Backus Naur Form and YACC structure

Pre-lab exercise: Compiling a Lex & YACC program

Procedure:

1. Consider if-else and switch-case branching constructs and 5 to 10 example statements of the same.
2. Design the grammar and derive the above statements manually.
3. Write the appropriate Lex and YACC specification in rule section of Lex and YACC based on the grammar.
4. Test rigorously with the sample statements Lex alone first and then Lex & YACC 5. Adjust the Lex & YACC specifications according to testing results. Evaluate the expression E iteratively using Lex & YACC by associating each production with an evaluation rule which evaluates the reduced term value

Example:

• Consider for example the case statement is handled by the production E->E+T { $$=$1 + $3 }

i.e., E->E+T is associated with $$=$1 + $3, where $$ is the value of LHS i.e., E and $1,$2,$3 are value of E, + and value of T respectively

• when E+T is reduced to E the value of E+T is stored into value of E

6. Repeat the steps 1 to 5 to include other branching statements and styles.

Additional Exercises:

1. Implement binary to decimal conversion

2. Calculate the evaluation cost of given expression

9. Use LEX and YACC to create two translators that would translate a given input (compound expression) into three-address and postfix intermediate codes. The input and output of the translators should be file.

Objective: learner will be able to design and create translators for intermediate code

Prerequisite: knowledge about the required intermediate code

Pre-lab exercise: Convert a prefix expression into equivalent postfix expression

Note: Refer Classic Expression Grammar from text.

Procedure (three-address code):

1. Get an input expression.
2. Using YACC, for every valid expression of an operation on two operands, create a temporary variable and assign it with the specified operation.
3. Print the intermediate code to output file.
4. If it is invalid return the error message.

Procedure (postfix intermediate code):

1. Get the input expression.
2. Using YACC, Upon recognizing a valid expression with two operands and an operator will replace it to postfix i.e., Each sub expression of type E + T is converted to E T +
3. Print the intermediate code to output file.
4. If invalid expression return the error message.

Additional Exercises:

1. Implement quadruple translator
2. Implement prefix translator

10. Write an optimizer pass in C or Java that does common-sub expression elimination on the three address intermediate code generated in the previous exercise.

Objective: learner will be able to analyze the possibility of Common Sub Expression elimination and implement.

Prerequisite: knowledge about optimization, Common Sub Expression and its elimination

Pre-lab exercise: implement string matching and substring replacement

Procedure:

1. Traverse three address code from top to bottom
2. If RHS is equal for two temporary variables
3. One of the variable is replaced with other variable and eliminated

Additional Exercises:

1. Write a C or Java Program to remove loop invariants in a piece of code.
2. Implement loop unrolling

11. Use Lex & YACC to write a back end that traverses the three address intermediate code and generates x86 code

Objective: learner will be able to design and implement back end machine code generator

Prerequisite: learner should know SIC / XE instruction Set

Pre-lab exercise: Implement file I/O operations

Procedure:

For each line in three address code based on the operation generate the SIC / XE code for example SIC / XE code for t = a+ b is as follows

```
LDA a
LDT b
ADDR A,T
STA t
```

Additional Exercises:

1. Use Lex & YACC to write a back end that generates 8085 assembly code for the three address code.
2. Use Lex & YACC to write a back end that generates ILOC intermediate code for the three address code.

12. Implement Local List Scheduling Algorithm

Objective: learner shall design and implement local list scheduling **Pre-requisite:** Knowledge of precedence graph, anti-dependence and priority function **Algorithm:**

Cycle ← 1

Ready ← leaves of *P*

Active ← Ø

while (Ready ∪ Active ≠ Ø)

  if (Ready ≠ Ø) then

    remove an *op* from Ready

    S(*op*) ← Cycle

Active ¬ Active ∪ *op*

Cycle ← Cycle + 1

for each *op* ∈ Active

if (S(*op*) + delay(*op*) ≤ Cycle) then

remove *op* from Active

for each successor s of *op* in P

if (s is ready) then

Ready ← Ready ∪ s

Additional Exercises:

1. Program to implement Global Scheduling.
2. Program to implement Resource allocation.

Additional Exercise:

An interactive decompiler, disassembler, debugger, and binary analysis platform built by reverse engineers, for reverse engineering API - **Binary Ninja**

(or)

A powerful disassembler and a versatile debugger **IDA Pro** can be used to create maps of their execution to show the binary instructions that are actually executed by the processor in a symbolic representation (assembly language).