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

COMPILER DESIGN LABORATORY

Course Code: CSE321

Semester: V

Lab Manual

2024

B.TECH CSBS

SHANMUGHA ARTS, SCIENCE, TECHNOLOGY AND RESEARCH ACADEMY
(SASTRA Deemed to be) University
Tirumalaisamudram, Thanjavur-613 401
School of Computing

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

Ex. No	Date	Particulars	Signature
1.		Implementation of Lexical Analyser using LEX.	
2.		Computation of FIRST & FOLLOW Sets in JAVA.	
3.		Implementation of LL (1) Parser	
4.		Implementation of LR (1) Parser	
5.		Implement a Parser for Branching Statements	
6.		Implement a Parser for Looping constructs in C	
7.		Implement a Parser for Complex Statements in C	
8.		Implementation of Front-End of the Compiler	
9.		Implement an Optimizer for the Intermediate ode	
10.		Implementation of Local List Scheduling Algorithm	
11.		Implementation of Global Register Allocation	
12.		Implementation of Code Generation Phase	

Signature of the Faculty

Course Objective:

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 using LEX and YACC tools.

Course Learning Outcomes:

1. Demonstrate the scanner construction from using Lex
2. Develop parser using Lex & YACC
3. Apply context sensitive analysis for type Inferencing
4. Construct intermediate Code representation for a given source code
5. Identify appropriate techniques for code optimization
6. Explain about the code generation and register allocation components in the backend phase of a compiler

List of Experiments:

1. Develop a scanner using LEX for recognizing the tokens in a given C program.
2. Develop a program to find the FIRST and FOLLOW sets for a given Context Free Grammar
3. Extend the outcome of experiment 2 to implement a LL(1) parser in C or Java to decide whether the input string is valid or not
4. 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)
5. Develop a parser for all branching statements of 'C' programming language using LEX& YACC
6. Develop a parser for all looping statements of 'C' programming language using LEX & YACC
7. Develop a parser for complex statements in 'C' programming language with procedure calls and array references using LEX & YACC
8. Use LEX and YACC to create two translators that would translate the given input (compound expression used in experiment 9) into three-address and postfix intermediate codes. The input and output of the translators should be a file
9. 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.
10. Implement Local List Scheduling Algorithm.
11. Implement Register Allocation
12. Use LEX & YACC to write a back end that traverses the three address intermediate code and generates x86 code.

Exercise No. 1 IMPLEMENTATION OF LEXICAL ANALYZER (SCANNER) USING LEX

Develop a lexical analyser (scanner) using LEX for recognizing the various token types in a given input C program.

Objective:

Learner will be able to design a lexical analyser for recognizing any type of tokens of a programming language.

Prerequisite:

Structure of a LEX program, Lex Specification for Tokens.

Pre-lab Exercises:

Lex program to recognize a number and string.

Lex program to count the number of words, lines and characters in the given input.

Procedure:

- Define the rules for tokenizing the Input.
- Construct the regular expressions for each token.
- Translate the regular expressions in accordance with the LEX specification.
- Create a LEX source file with a "`.l`" extension. The LEX source file will contain the definitions, rules and auxiliary code sections.
- Compile the LEX source file. This will generate the C source file named "lex.yy.c".
- Then compile the generated C source file, the scanner.
- Finally, run the compiled scanner with an input file containing the code to be tokenized.

Sample Lex program to recognize an Integer " num.l"	Sample Output
%{ #include **<stdio.h>** %} %% [0-9]+ { printf("**Integer: %s\n**", yytext); } . { printf("**Unrecognized character: %s\n**", yytext); } %% int main( ) { yylex( ); return 0; } int yywrap( ) { return 1; }	***Compilation*** ***$ flex num.l*** ***$ gcc lex.yy.c*** ***$ ./a.out*** ***Sample Input:*** *45* ***Output:*** *Integer: 45*

Sample Input / Output

Input	Output	

#include<stdio.h>
int main()
{
printf("Hello World");
}
	#include<stdio.h>
	int
	main(
	{
	printf(
	"Hello World"
	)
	}
		is a preprocessor directive
		is a keyword
		is a function
		block begins
		is a function
		is a string
		symbol
		end of the block.

Additional Exercises:

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

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 2 – COMPUTATION OF FIRST AND FOLLOW SETS

Write 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.

Software:

Java / Python

Prerequisite:

Definition and Structure of Context Free Grammar.
Handling Left Recursion, Left factor and Null Productions in the Grammar.
Definition and Rules for computing of FIRST and FOLLOW sets.

Pre-lab Exercise:

Substring extraction

Algorithm

Algorithm to Compute FIRST Sets

1. **Initialization**:
   - For each terminal symbol t, set **FIRST(t) = {t}.**
   - For each non-terminal symbol A, initialize FIRST(A) = {}.
2. **Repeat the Following Steps Until No Changes Occur**:
   - For each production A → α (where α is a sequence of grammar symbols):
     - For each symbol X in α from left to right:
       1. **Add {FIRST(X) - {ε}} to FIRST(A)**:
          - Add all non-ε elements of **FIRST(X) to FIRST(A).**
       2. **If ε is in FIRST(X)**, continue to the next symbol in α and repeat the process.
       3. **If ε is not in FIRST(X)**, stop here and do not include ε in FIRST(A) for this production.
     - If α is nullable (i.e., all symbols in α can derive ε), add ε to FIRST(A).

Algorithm to Compute FOLLOW Sets

1. **Initialization**:
   - For each non-terminal A, initialize FOLLOW(A) = {}.
   - Add $ (the input right end marker) to FOLLOW(S), where S is the start symbol.

2. **Repeat the Following Steps Until No Changes Occur**:
   - For each production A →αBβ:

     (a) For each symbol B in α:

- Add **(FIRST(β) - {ε}) to FOLLOW(B).**
- If β is nullable (can derive ε), add **FOLLOW(A) to FOLLOW(B).**

(b) For each production A → αB:

- Add **FOLLOW(A) to FOLLOW(B).**

Sample Input / Output

<table>
<tr><th>INPUT</th><th>OUTPUT</th></tr>
<tr><td>S → CC
C → aC | b</td><td>First (a) = a & First (b) = b
First (S) = {a, b}
First (C)= {a, b}
Follow (S) = {$}
Follow (C) = {$, a, b}</td></tr>
<tr><td>S → AB
A → a | ε
B → b</td><td>First (a) = a & First (b) = b
First (S) = {a, b} Follow(S) = {$}
First (A)= {a, ε } Follow(A) = {b}
First (B) = {b} Follow(B) = {$}</td></tr>
<tr><td>E → TA
A → +TA | ε
T → FB
B → *FB | ε
F → (E) | id</td><td>First (+) = +, First (*) = *, First (() = (, First ()) =), First (id) = id
<table>
<tr><th>Variable</th><th>FIRST</th><th>FOLLOW</th></tr>
<tr><td>E</td><td>{(, id }</td><td>{), $}</td></tr>
<tr><td>A</td><td>{+, ε }</td><td>{), $}</td></tr>
<tr><td>T</td><td>{(, id }</td><td>{+,), $}</td></tr>
<tr><td>B</td><td>{*, ε }</td><td>{+,), $}</td></tr>
<tr><td>F</td><td>{(, id }</td><td>{*, +,), $}</td></tr>
</table>
</td></tr>
</table>

Additional Exercises:

1. Program to find Augmented FIRST sets.

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 3 – IMPLEMENTATION OF LL (1) PARSER

Write a program to implement a LL (1) parser to decide whether the given 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 in either C or in Java.

Prerequisite:

Two-dimensional array manipulation, String searching and Substring extraction.

Software:

Java / C

Pre-lab exercise:

Programs on Array processing and string processing

Program to Compute the FIRST and FOLLOW sets for the given CFG.

Algorithm for LL (1) parsing

1. **Compute FIRST and FOLLOW sets** for the given Context Free Grammar. (Use the output of the previous exercise)

2. Construct the Parsing Table

1. Initialize the parsing table M with empty entries.
2. For each production A → α:
   - For each terminal t in FIRST(α), add A → α to M [A, t].
   - If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add A → α to M[A, b].
   - If ε is in FIRST(α) and $ is in FOLLOW(A), add A → α to M[A, $].

3. **Parsing Algorithm**

1. **Initialize:**
   - Push the end-of-input marker ($) onto the stack.
   - Push the start symbol S onto the stack.
   - Set the current input pointer to the first symbol of the input string.
2. **Parse:**
   - While the stack is not empty:
     - Let X be the top symbol on the stack.
     - Let 'a' be the current input symbol.
     - If X is a terminal or the end-of-input marker ($):
       - If X=a:
         - Pop X from the stack.

- Advance the input pointer to the next symbol.
- Else:
  - Report a syntax error.
- Else (if X is a non-terminal):
  - If M [X, a] is not empty:
    - Pop X from the stack.
    - Push the production in M[X,a] onto the stack in reverse order.
  - Else:
    - Report a syntax error.

3. **Accept or Reject:**
   - If the stack is empty and the input pointer is at the end of the input string ($):
     - Accept the input string.
   - Else:
     - Reject the input string (report a syntax error).

Sample Input / Output

INPUT

Grammar:

S → CC

C → aC | b

First & Follow Sets

First (a) = a & First (b) = b

First (S) = {a, b}

First (C)= {a, b}

Follow (S) = {$}

Follow (C) = {$, a, b}

Input string:

1. aab
2. abb

OUTPUT

LL (1) Parsing Table

NT / T	a	b	$
S	S → CC	S → CC	
C	C → aC	C → b	

Parsing the Input string "aab"

Stack	Input Buffer	Action taken
$S	aab$	M[S,a]: Expand by S →CC
$CC	aab$	M[C,a]: Expand by C →aC
$CCa	aab$	Match 'a'
$CC	ab$	M[C,a]: Expand by C →aC
$CCa	ab$	Match 'a'
$CC	b$	M[C,b]: Expand by C →b
$Cb	b$	Match 'b'
$C	$	M[C,$]: No Entry - Error

Parsing the Input string "abb"

Stack	Input Buffer	Action taken
$S	abb$	M[S,a]: Expand by S →CC
$CC	abb$	M[C,a]: Expand by C →aC
$CCa	abb$	Match 'a'
$CC	bb$	M[C,b]: Expand by C →b
$Cb	bb$	Match 'b'
$C	b$	M[C,b]: Expand by C →b
$b	b$	Match 'b'
$	$	Accept

Additional Exercises:

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

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 4 – IMPLEMENTATION OF LR (1) PARSER

Implement a CLR parser in C / 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) item set computation.

Pre-lab Exercise:

Programs on Closure and Move computation, LR (1) items creation.

Algorithm for Constructing the CLR Parsing Table

Step 1: Augment the Grammar

1. Add a new start symbol S' and a new production S' -> S, where S is the original start symbol of the grammar.

Step 2: Compute the First and Follow Sets

2. Compute the First and Follow sets for all non-terminals in the grammar. These sets are crucial for constructing the parsing table.

Step 3: Construct the Canonical Collection of LR(1) Items

3. Initialize the canonical collection of LR(1) items with the closure of the augmented start production:
    - C = { closure({[S' -> .S, $]}) }
4. Repeat until no new sets of LR(1) items can be added to C:
    - For each set of items I in C and each grammar symbol X:
        - Compute the goto function: goto(I, X)
        - If goto(I, X) is not empty and not already in C, add it to C.

Step 4: Construct the Parsing Table

5. Create an empty parsing table with action and goto sections.
6. For each set of LR(1) items I:
    - For each item [A -> α.β, a] in I:
        - If β is a terminal t, set action[I, t] to "shift goto(I, t)".
        - If β is empty and A is not S', set action[I, a] to "reduce A -> α".
        - If [S' -> S., $] is in I, set action[I, $] to "accept".
7. For each non-terminal A:
    - Set goto[I, A] to the state goto(I, A).

Step 5: Algorithm for Parsing Using the CLR Parsing Table

1. **Initialization**:
    - Push the start state (0) onto the stack.
    - Set the input pointer to the first symbol of the input string.
2. **Parsing Loop**:
    - Repeat until the parsing action is "accept":
        1. Let s be the state on top of the stack.
        2. Let a be the current input symbol.
        3. Look up the action for state s and input symbol a in the parsing table:
            - **Shift**: If action[s, a] is "shift t":
                - Push t onto the stack.
                - Move the input pointer to the next symbol.
            - **Reduce**: If action[s, a] is "reduce A -> β":
                - Pop |β| symbols off the stack.
                - Let s' be the new state on top of the stack.
                - Push goto[s', A] onto the stack.
                - Output the production A -> β.
            - **Accept**: If action[s, a] is "accept":
                - The input string is successfully parsed.
            - **Error**: If no action is defined for s and a:
                - Report a syntax error.

Sample Input / Output

INPUT

Grammar:

S → CC ..... (R1)
C → aC ..... (R2)
C → b ..... (R3)

CLR Parsing Table

State	ACTION			GOTO	
	a	b	$	S	C
0	S3	S4		1	2
1			ACC		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

Input string: abb

OUTPUT

Parsing the Input string "abb"

Stack	Input Buffer	Action taken
0	abb$	M[0,a] = S3 – Shift.
0a3	bb$	M[3,b] = S4 – Shift
0a3b4	b$	M[4,b] = Reduce (C → b)
0a3C8	b$	M[8,b] = Reduce (C→ aC)
0C2	b$	M[2,b] = S7 – Shift
0C2b7	$	M[7,$] = Reduce (C → b)
0C2C5	$	M[5,$] = Reduce (S → CC)
0S1	$	Accept

Additional Exercises:

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

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 5 – IMPLEMENT A PARSER FOR BRANCHING STATEMENT

Developing a parser for the conditional statements of 'C' Language using LEX and YACC tool.

Objective:

Develop parsers for a conditional / branching construct along with a supporting scanner.
Design and construct complex parsers by combining these simple parsers.
Develop the skill to design and construct context free grammar for a programming constructs.

Prerequisite:

Branching statements in C programming language with their syntax and semantics.
Augmenting a Grammar for a programming language construct in BNF and their corresponding YACC specification.
Structure of a LEX program, YACC program and the compilation procedure.

Pre-lab Exercise:

Any Parser using Lex and YACC.

Procedure:

1. LEX File ('.l' extension**)**: Defines the tokens for the C branching statements, identifiers, constants and operators.

2. YACC File ('.y' extension**)**: Defines the grammar rules for the branching statements, expressions and statements. It uses the tokens defined in the LEX file.

3. Building and Running: Generate the scanner and parser using FLEX and YACC. Like LEX, the YACC compiler will generate the C source file named "y.tab.c".

4. Then **compile and run the parser** with a sample input file containing C branching statements.

```
$ lex lexer.l
$ yacc -d par.y
$ cc lex.yy.c y.tab.c
$ ./a.out
```

LEX program for recognizing integer token	YACC program for validating an expression

%{
#include "y.tab.h"
%}
 	
%%
 	
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
. { }
 	
%%
 	
int yywrap() {
return 1;
}
	%{
	#include <stdio.h>
	#include <stdlib.h>
	%}
	%token NUMBER
	%%
	input:
	| input line
	;
	line:
	'\n'
	| expr '\n' { printf("Result: %d\n", $1); }
	;
	expr:
	NUMBER
	| expr '+' expr { $$ = $1 + $3; }
	;
	%%
	int main( ) {
	printf("Enter an expression: ");
	return yyparse();
	}
	void yyerror(const char *s) {
	fprintf(stderr, "Error: %s\n", s);
	}

Sample Input / Output

INPUT	OUTPUT
If (a>b) big=a; else big=b;	Valid Syntax
If (a<b) b=a;	Valid Syntax
If (a=b) a=a+10;	Error.

Additional Exercises:

1. Implement parser for validating the syntax of “else if” ladder statement in C
2. Implement parser for validating the syntax of “switch” statement in C
3. Implement parser for balanced parenthesis

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 6 – IMPLEMENT A PARSER FOR LOOPING STATEMENT IN C

Developing a parser for the looping statements of 'C' language using LEX & YACC

Objective:

Develop parsers for an iterative construct along with a supporting scanner.

Design and construct complex parsers by combining these simple parsers

Develop the skill to design and construct context free grammar for nested programming construct like nested for loops, etc.,

Prerequisite:

Looping statements in C programming language with their syntax and semantics.

Augmenting a Grammar for an iterative constructs of C language in BNF and their corresponding YACC specification. (while, do-while and for statement)

Structure of a LEX program, YACC program and the compilation procedure.

Pre-lab Exercise:

Design of Parser using LEX and YACC for the conditional and branching statements in C.

Procedure:

1. LEX File ('.l' extension**)**: Defines the tokens for the C looping statements, identifiers, constants and operators.

2. YACC File ('.y' extension**)**: Defines the grammar rules for the looping statements, expressions and statements. It uses the tokens defined in the LEX file.

3. Building and Running: Generate the scanner and parser using FLEX and YACC. Like LEX, the YACC compiler will generate the C source file named "y.tab.c".

```
$ flex lexer.l
$ yacc -d parser.y
$ gcc lex.yy.c y.tab.c -o
parser –ll
```

4. Then **compile and run the parser** with a sample input file containing C looping statements.

LEX program for recognizing few tokens	YACC program for validating an expression

%{
#include "y.tab.h"
%}
 	
%%
While | do | for {return KEYWORD;}
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
>|>=|
<|<=|==|!= {return RELOP;}
. ;
%%
 	
int yywrap() {
return 1;
}
	%{
	#include <stdio.h>
	#include <stdlib.h>
	%}
	%token KEYWORD, NUMBER, RELOP
	%%
	input:
	| input line
	;
	line:
	'\n'
	| expr '\n' { printf("Result: %d\n", $1); }
	;
	expr:
	NUMBER
	| expr '+' expr { $$ = $1 + $3; }
	;
	%%
	int main( ) {
	printf("Enter an expression: ");
	return yyparse();
	}
	void yyerror(const char *s) {
	fprintf(stderr, "Error: %s\n", s);
	}

Sample Input / Output

INPUT	OUTPUT
while(i<j) { sum+=i;i++;}	Valid Syntax
for(i=0;i<n;i++) printf("%d",i);	Valid Syntax
While a>b a=a+b;	Error.

Additional Exercises:

1. Implement a parser for looping statements in Python / JAVA.
2. Implement a parser for validating the nested looping statements in C.

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 7 – IMPLEMENT A PARSER FOR COMPLEX STATEMENT IN C

Developing a parser for validating the Syntactical correctness of the statements which includes Procedure definition, calls and Array references in 'C' language using LEX & YACC.

Objective:

The Learner will be able to design and construct a parser for the complex programming language statements involving procedure call or array references.

Prerequisite:

Develop the skill to construct a Context Free Grammar for procedure call statement in C

Develop the skill to construct a Context Free Grammar for expressions involving an array references in C

Pre-lab exercise:

Building a parser for a conditional / iterative constructs in C programming language.

Procedure:

1. LEX File ('.l' extension**)**: Defines the tokens for the complex statements, identifiers, constants, operators.

2. YACC File ('.y' extension**)**: Defines the grammar rules for the procedure call and Expressions involving one dimensional array references. It uses the tokens defined in the LEX file.

3. Building and Running: Generate the scanner and parser using FLEX and YACC. Like LEX, the YACC compiler will generate the C source file named "y.tab.c".

4. Then **Compile and run** the parser with a sample input file .

LEX program	YACC program for validating an array

%{
#include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z0-9_]* {return ID; }
[0-9]+ { return NUM; }
"[" { return '['; }
"]" { return ']'; }
[\t\n] { }
. { }
%%
int yywrap() {
return 1;
}
	%{
	#include <stdio.h>
	#include <stdlib.h>
	void yyerror(const char *s);
	%}
	%union { char *str; int num; }
	%token <str> ID
	%token <num> NUM
	%%
	input:
	| input line
	;
	line:
	'\n'
	| array_ref '\n'
	{ printf("Valid array reference: %s[%d]\n", $1.str, $2.num); }
	;
	array_ref:
	IDENTIFIER '[' NUMBER ']'
	{ $$ = (typeof($$)){ .str = $1.str, .num = $3.num }; }
	;
	%%
	int main() {
	printf("Enter an array reference: "); return yyparse(); }
	void yyerror(const char *s) { fprintf(stderr, "Error: %s\n", s); }

Execution

```
$ flex lexer.l

$ yacc -d parser.y

$ gcc lex.yy.c y.tab.c -o parser –ll

$ ./parser
```

Sample Input / Output

INPUT	OUTPUT

int sum(int a, int b) {
int c;
c=a+b;
return(c);
}
	Valid Syntax

x=a[5]+10;	Valid Syntax
a[i] = b[i] + c[i];	Valid Syntax

Additional Exercises:

1. Implement a parser for expressions involving multi-dimensional array references.
2. Implement a parser for Complex programming language statement in Python

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 8 – IMPLEMENTATION OF FRONT-END OF COMPILER

Generate Intermediate Code (TAC / POSTFIX) for an Expression using LEX & YACC

(Use LEX and YACC to create a translator that would translate a given input compound expression into Three-Address Code Statement or equivalent Postfix Expression. The input and output of the translators should be file.)

Objective:

The Learner will be able to design and create translators for generating Intermediate Code.

Prerequisite:

Knowledge about the various Intermediate code representations TAC, Postfix and Syntax tree.
Note: Refer Classic Expression Grammar from text.

Pre-lab exercise:

Parser to validate the syntax of an Arithmetic expression.
Parser to validate the given branching or looping constructs in C Language.

Algorithm

1. **Define Tokens in Lex File**
    1. **Token Definitions**: Define tokens for identifiers, numbers, operators and other necessary symbols.
    2. **Return Tokens**: Return the appropriate token type to the YACC parser.

2. **Define Grammar Rules in YACC File**
    1. **Define Tokens and Data Types:** Use %token to define tokens and %union to define the data types for semantic values.
    2. **Define Grammar Rules**: Create rules for expressions, terms, and factors.

3. **Generate Temporary Variables**
    - Use a counter to generate new temporary variables for intermediate results.
    - Implement a function to generate new temporary variable names.

4. **Create a Function to Add TAC Instructions**
    - In the actions for the grammar rules, generate TAC instructions and store them in a list.
    - Define a function to add TAC instructions to a list/file.

5. **Print TAC Instructions**
    - After parsing, print the generated TAC instructions.

Sample Input / Output

INPUT	OUTPUT
x=a+b*c;	t1=b*c t2=a+t1 x=t1
x=a+b*c;	abc*+x=
x=a*-b+c*-b;	t1=Uminus b t2=a*t1 t3=Uminus b t4=c*t3 t5=t2+t4 x=t5
x=a*-b+c*-b;	xab-*cb-*+=

Additional Exercises:

1. Implement quadruple / Triple translator
2. Implement prefix translator

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 9 – IMPLEMENT AN OPTIMIZER FOR INTERMEDIATE CODE

Write an Optimizer pass in C or Java that does Common Sub-Expression Elimination on the Three Address Code generated in the previous exercise.

Objective:

The learner will be able to identify the Common Sub Expression in the given input TAC sequence and Eliminate the common subexpression.

Prerequisite:

Knowledge of Intermediate representations – TAC, Postfix and Syntax tree.
Familiarity with using lists (e.g., ArrayList) and maps (e.g., HashMap) in Java.
Understanding of hash functions and their use in optimizing lookups.
Basics of Common Subexpression and other about Function – preserving transformations.

Pre-lab Exercise:

Implement string matching and substring replacements in C / JAVA.

Knowledge in Java, including understanding of object-oriented principles, data structures and standard libraries.

Algorithm

1. **Read TAC Instructions**:

   Read the TAC Instructions from the input file.

2. **Identify Common Subexpressions**:

   Use a hash table to identify and store common sub-expressions.

3. **Replace Common Subexpressions**:

   Replace the common subexpressions with previously computed results.

4. **Output Optimized TAC**:

   Print the optimized TAC instructions.

Sample Input / Output

INPUT	OUTPUT

a = b + c;
d = b + c;
f = a + d;
	a = b + c;
	d = a;
	f = a + d;

x = a + b * c;
y = a + b * c + d;
z = (a + b * c) * (e + f);
	t1 = a + b * c;
	t2 = e + f;

	x = t1;
	y = t1 + d;
	z = t1 * t2;

t1 = a + b t2 = t1 * c t3 = a + b t4 = t3 * d t5 = t2 + t4	t1 = a + b t2 = t1 * c t4 = t1 * d t5 = t2 + t4

Additional Exercises:

1. Write a C / Java Program to perform other transformations such as Copy propagation, Dead-code elimination, etc.
2. Implement loop-unrolling in Java.

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 10 – IMPLEMENTATION OF LIST SCHEDULING ALGORITHM

Implement a Local List Scheduling Algorithm in Java / C, which is used to improve the performance of generated code by reordering instructions to better utilize the processor's resources and reduce execution time.

Objective:

The learner shall design and implement local list scheduling algorithm using C or JAVA

Pre-requisite:

Basics of Instruction scheduling and dependencies.
Knowledge of Basic Blocks and Intermediate representations.
Proficiency in Java, including understanding of data structures and algorithms.

Pre-lab Exercise:

Generating Intermediate representation especially, TAC Instructions
Implementing transformation used for Code Optimization

Algorithm

1. **Build Dependency Graph:**

   Create nodes for each instruction and edges to represent dependencies between instructions.

2. **Calculate Priorities:**

   Assign a priority to each node based on its dependencies. Commonly, nodes with fewer dependencies or those that enable more instructions (i.e., longer chains) get higher priorities.

3. **List Scheduling:**

   Use a priority queue to select the highest-priority node that is ready to be scheduled (i.e., all its dependencies are met).

   Schedule the selected node and update the priority queue.

Sample Input / Output

INPUT	OUTPUT
1. t1 = a + b 2. t2 = c + d 3. t3 = t1 * e 4. t4 = t2 * f 5. t5 = t3 + t4	**Dependency Analysis:** t3 depends on t1 t4 depends on t2 t5 depends on t3 and t4 **Scheduling:** **Initial Ready List**: [1, 2] (both t1 = a + b and t2 = c + d are independent) Schedule t1 = a + b and t2 = c + d (can be done in parallel) **Updated Ready List**: [3, 4] (both t3 and t4 can be scheduled next) Schedule t3 = t1 * e and t4 = t2 * f (can be done in parallel) **Updated Ready List**: [5] (only t5 = t3 + t4 is left) Schedule t5 = t3 + t4 **List** 1. t1 = a + b 2. t2 = c + d 3. t3 = t1 * e 4. t4 = t2 * f 5. t5 = t3 + t4
1. t1 = a + b 2. t2 = t1 * c 3. t3 = d + e 4. t4 = t3 - f 5. t5 = t2 + t4	1. t1 = a + b 2. t3 = d + e 3. t2 = t1 * c 4. t4 = t3 - f 5. t5 = t2 + t4

Additional Exercises:

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

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 11 – IMPLEMENTATION OF REGISTER ALLOCATION ALGORITHM

Implement a Global register allocation algorithm in JAVA, aimed to efficiently assigning a limited number of machine registers to a larger number of variables used in the source program.

Objective:

The learner shall design and implement Global Register Allocation algorithm in JAVA.

Prerequisite:

Knowledge of Basic Blocks and Intermediate representations
Knowledge on graph coloring algorithm.

Pre-lab Exercise:

Generating Intermediate representation especially, TAC Instructions
Implementing Graphs and traversals algorithms in C / JAVA

Algorithm

1. **Build the Interference Graph:**

   Nodes represent variables.

   Edges represent interference between variables (i.e., they cannot be assigned to the same register).

2. **Simplify:**

   Repeatedly remove nodes with fewer edges than the number of available registers, pushing them onto a stack.

3. **Spill:**

   If all remaining nodes have more edges than registers, select a node to spill (i.e., store in memory) and remove it from the graph.

4. **Select:**

   Pop nodes from the stack and assign them registers, ensuring no two adjacent nodes share the same register.

5. **Assign Registers:**

   Map the variables to registers or memory based on the graph colouring results

Sample Input / Output

INPUT	OUTPUT
1. a = b + c 2. d = a + e 3. f = d + g 4. h = f + i	1. R1 = R2 + R3 // a = b + c, a in R1, b in R2, c in R3 2. R2 = R1 + R1 // d = a + e, d in R2, a in R1, e in R1 (spilled e to R1) 3. R3 = R2 + R2 // f = d + g, f in R3, d in R2, g in R2 (spilled g to R2) 4. R1 = R3 + R3 // h = f + i, h in R1, f in R3, i in R3 (spilled i to R3)

Additional Exercises:

1. Program to implement peephole optimization techniques on the given input.

DETAILS OF NOTES & OBSERVATIONS

Exercise No: 12 – IMPLEMENTATION OF CODE GENERATION PHASE

Write a Back-end of the Compiler that traverses the TAC and generates x86 Assembly code

Objective:

The learner will be able to design and implement back-end of the compiler that reads TAC Instruction and generates x86 machine code.

Prerequisite:

Knowledge of Intermediate representations – TAC, Postfix and Syntax tree
Knowledge in SIC / XE instruction Set
Familiarity in YACC Grammar rules and LEX patterns.

Pre-lab Exercise:

Generation of TAC Instructions
Implement file I/O operations

Algorithm

1. Define the Grammar (YACC):

Define grammar rules that recognize and handle TAC instructions.
TAC typically includes operations like assignment (=, +, -, *, /), conditional jumps (ifgoto), and labels (label).

2. Lexical Analysis (Lex):

Define a Lex file to specify patterns and actions for tokens recognized by YACC.

3. YACC Actions to Generate x86 Code:

Once the grammar rules are defined and tokens are recognized, define YACC actions to process these tokens and generate x86 code.

Sample Input / Output

INPUT	OUTPUT
1. a = b + c 2. d = a + e 3. f = d + g 4. h = f + i	ADD R7, R1, R2 // Compute a + b and store in R7 MUL R8, R7, R3 // Compute (a + b) * c and store in R8 ADD R9, R4, R5 // Compute d + e and store in R9 SUB R10, R9, R6 // Compute (d + e) - f and store in R10 ADD R11, R8, R10 // Compute ((a + b) * c) + ((d + e) - f) and store in R11

Additional Exercises:

1. Use Lex & YACC to write a back end that generates 8085 assembly code for the given TAC Statements.

DETAILS OF NOTES & OBSERVATIONS