**Introduction to Lex Tool**

**Experiment No. 1**  **Date:-** 05/09/24

**Aim :-**   
Write a Lex program to validate different tokens such as Identifier, Floating point number, Integer, Keywords etc.

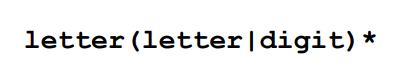
**Theory :-**

Lex is a Unix utility for lexical analysis that parses a file of characters and uses regular expression matching to ‘tokenize’ the contents. It is commonly used with the YACC utility for more complex parsing tasks.

Lex generates C code for a lexical analyzer, or scanner, which matches patterns in the input and converts them into tokens—numerical representations that simplify processing. Tokens are stored in a symbol table, potentially with additional details like data type and memory location.

In the context of compilers, Lex is typically used in the initial phase to convert input strings into tokens. It uses regular expressions to specify patterns that can match input strings. Each pattern has an associated action, usually returning a token for further processing by the parser.

Example:



This pattern matches a string of characters that begins with a single letter, and is followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

• repetition, expressed by the “\*” operator

• alternation, expressed by the “|” operator

• concatenation

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

* Type token : id, number, real, . . .
* Punctuation tokens : if, void, return, . . .
* Alphabetic tokens : keywords
* Keywords : for, while, if, etc.
* Identifier : Variable name, function name, etc.
* Operators : '+', '++', '-' etc.
* Separators : ',' ';' etc.

Example of Non-Tokens:

* Comments, preprocessor directive, macros, blanks, tabs, newline

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.

Example - “float”, “abs\_zero\_Kelvin”, “=”, “-”, “273”, “;” .

**Steps in Lexical Analysis:**

1. **Input preprocessing:** This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.
2. **Tokenization:** This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.
3. **Token classification:** In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.
4. **Token validation:** In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.
5. **Output generation:** In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.

The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.

Suppose we pass a statement through lexical analyzer – a = b + c ;

It will generate token sequence like this: id=id+id;

How to execute:

1. Type lex lexfile.l
2. Type gcc lex.yy.c
3. Type  ./a.out

**Algorithm :-**

1. Define regular expressions for each type of token you want to recognize, e.g., identifiers, keywords, numbers.
2. Write a Lex program that associates each regular expression with an action. The action typically prints or stores the token when it matches the regular expression.
3. Compile the Lex program using the Lex tool to generate a C source file.
4. Compile the generated C file using a C compiler.
5. Run the executable to test if the program correctly identifies and validates tokens in input strings.

**Program :-**

%{

#include<stdio.h>

%}

letters[a-zA-Z]

digit[0-9]

%%

{letters}({letters}|{digit})\* { printf("Variable: %s\n",yytext); }

{digit}+ { printf("Interger type %s\n",yytext); }

{digit}+"."{digit}+ { printf("Float type %s\n",yytext); }

%%

void main(){

yylex();

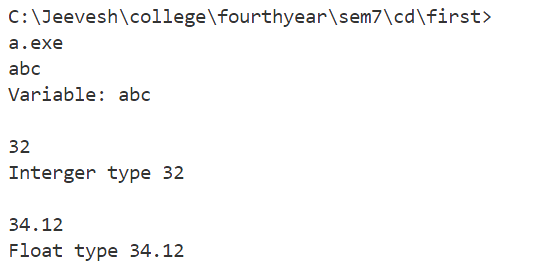
}

int yywrap() {

return 1;

}

**Output:-**



**Conclusion:-**

In this experiment, we learn the basics of LEX programming. The regular expressions for identifiers, keywords, operators, integers and floating points are found, to validate them using a LEX code.

**Lexical Analyzer for C Language**

**Experiment No. 2** **Date:-** 13/09/24

**Aim :-** Write a Lex program to validate all the tokens from C Language.

**Theory :-**

In the compilation process, lexical analysis is the initial step that breaks down the source code into manageable parts called tokens. Each token is a string that represents a basic, indivisible element of the programming language, such as keywords, identifiers, operators, punctuation, and literals.

**Tokens in the C Language:**

* **Keywords:** These are reserved words that have a specific meaning in the language and cannot be used as identifiers. Examples include int, float, if, else, return, while, etc.
* **Identifiers:** Identifiers are names given to variables, functions, arrays, etc., created by the programmer. An identifier must start with an alphabet or underscore and can be followed by any number of alphanumeric characters or underscores. Example: myVariable, sum1.
* **Operators:** Operators are symbols that perform operations on variables and values. Examples include:
  + Arithmetic operators: +, -, \*, /
  + Relational operators: ==, !=, >, <, <=, >=
  + Logical operators: &&, ||
* **Literals:** A literal represents a fixed value in the source code. There are different types of literals:
  + Integer Literals: Numbers without a decimal point, e.g., 42, 100.
  + Floating-point Literals: Numbers with a decimal point, e.g., 3.14, 2.71.
  + Character Literals: Single characters enclosed in single quotes, e.g., 'a', '1'.
  + String Literals: Sequences of characters enclosed in double quotes, e.g., "Hello, World!".
* **Punctuation and Symbols:** These are used to structure the C language syntax. They include {, }, ;, (, ), [, ].

Each of these token types has a specific regular expression associated with it that helps to identify and classify the token. Lexical analyzers like Lex use regular expressions to match these patterns in the source code.

**Sample Token Definitions:**

Let's define some regular expressions and their expected matches for common C tokens:

* **Keywords:** Recognized by exact match (e.g., if, else, int). For example, int main would match int as a keyword.

if { return IF; } // Matches the keyword "if"

* **Identifiers:** Matches any alphanumeric string starting with an alphabet or underscore, e.g., var1, main\_function.

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

* **Operators:** Matches characters like +, -, \*, /, or relational and logical operators like ==,!=, &&. For example, in a + b, it would match + as an arithmetic operator.

"+"|"-"|"\*"|"/" { printf("Operator: %s\n", yytext); }

* **Literals:**
  + Integer Literals: Matches numbers like 123, 456.

[0-9]+ { printf("Integer Literal: %s\n", yytext); }

* + String Literals: Matches sequences in double quotes, e.g., "Hello World".

\"[^\"]\*\" { printf("String Literal: %s\n", yytext); }

* Punctuation: Matches symbols like ;, {, }, (, ) to recognize the structure of the code.

";" { printf("Semicolon: %s\n", yytext); }

"{" { printf("Opening brace: %s\n", yytext); }

**Algorithm :-**

1. Define Token Patterns: Specify patterns for all token types in the C language using regular expressions in Lex. This includes keywords, identifiers, operators, literals, and punctuation symbols.
2. Initialize the Lex Program: Begin by including standard libraries and setting up rules in Lex.
   1. Write Regular Expressions:
   2. Keywords: Define reserved keywords in C using specific patterns.
   3. Identifiers: Define patterns for valid variable names.
   4. Operators: Define patterns for arithmetic, relational, and logical operators.
   5. Literals: Define patterns for numeric literals, character literals, and string literals.
   6. Punctuation: Define patterns for symbols like {, }, ;, (, ).
3. Categorize and Output Tokens: For each matched pattern, print the category and value of the token.
4. Compile and Run the Lex Program: Use flex and gcc to compile and test the program with different C code inputs.

**Program :-**

%{

#include <stdio.h>

%}

%%

[0-9]+ { printf("Integer: %s\n", yytext); }

\".\*\" { printf("String: %s\n", yytext); }

\'[A-Za-z]\' { printf("Charecter : %s\n", yytext); }

[0-9]\*\.[0-9]+ { printf("Float: %s\n", yytext); }

char|int|float|double|void|boolean { printf("Datatype: %s\n", yytext); }

if|else|switch|"else if" { printf("Conditional Statement : %s\n",yytext);}

for|while|do { printf("Loops: %s\n", yytext); }

\\*|\+|\-|\/ { printf("Arithmetic Operator: %s\n", yytext);}

==|<=|>=|<|> { printf("Comparison Operator: %s\n", yytext); }

= { printf("Assignment Operator: %s\n", yytext); }

case { printf("Case Statement : %s\n", yytext);}

return { printf("Return Statement : %s\n", yytext);}

break { printf("Break Statement : %s\n", yytext);}

exit { printf("Exit Statement : %s\n", yytext);}

[\_a-zA-Z][\_a-zA-Z0-9]\* { printf("Identifier : %s\n", yytext); }

\(|\{|\[ { printf("Opening Bracket : %s\n", yytext);}

\)|\}|\] { printf("Closing Bracket : %s\n", yytext);}

; { printf("Statemetn Terminator : %s\n", yytext);}

\\n { printf("Newline : %s\n", yytext);}

\\t { printf("Horizontal End : %s\n", yytext);}

&|\| { printf("Bitwise Operator : %s\n", yytext);}

&&|\|\| { printf("Logical Operator : %s\n", yytext);}

. { printf("Invalid Token\n");}

%%

int main() {

printf("Enter the expression - \n");

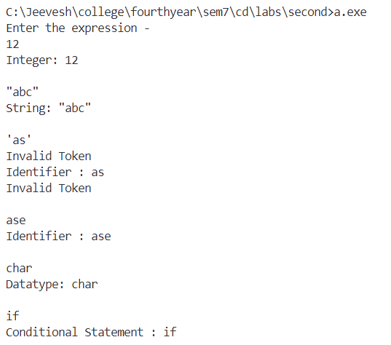
yylex();

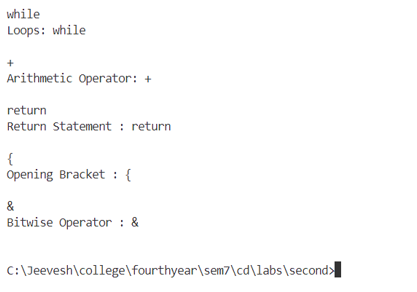
return 0;

}

int yywrap(){return 1;}

**Output :-**

****



**Conclusion :-**

A Lex program was successfully written and executed to validate all the tokens from C Language.

**Syntax Analyzer –Validate Simple For/While Statement**

**Experiment No. 3**  **Date:-** 19/09/24

**Aim :-** Write a YACC program to validate syntax of simple For/ While loop.

**Theory :-**

YACC (Yet Another Compiler Compiler) is a powerful tool for parsing structured text and is often used for building compilers and interpreters. Combined with Lex, YACC helps in validating, interpreting, or translating code by parsing tokens generated by a lexical analyzer.

In programming, loops are fundamental constructs allowing repetitive execution of code blocks. This YACC program focuses on validating two types of loops:

1. For Loop:
   * The for loop allows iterating a code block for a specific number of times. Its syntax generally has three main parts: initialization, condition, and increment.
   * A standard for loop structure:

for (initialization; condition; increment) {

// Code to execute in each iteration

}

1. While Loop:
   * The while loop executes a code block as long as a specified condition is true.
   * A standard while loop structure:

while (condition) {

// Code to execute while the condition is true

}

**Grammar Rules for Parsing Loops**

* For Loop Grammar:
  + Consists of an ID (identifier), assignment (=), initialization, condition with comparison operators, increment expressions, and body within {}.
  + Basic structure:

S ->: FOR '(' INIT ';' COND ';' INC ')' '{' STMT ‘;’ '}'

INIT -> ID '=' EXP

COND -> ID RELOP ID

RELOP -> ‘>’ | ‘<’ | ‘>=’ | ‘<=’

INC : ID OP

OP -> ‘++’ | ‘--'

STMT -> ID ‘=’ EXP

EXP -> EXP + EXP | EXP – EXP | ID | NO

* + Example: for ( i = 0 ; i< 2 ; i++ ) { x = x + 1 ; } (x < 10)
* While Loop Grammar:
  + Starts with the while keyword, followed by a condition enclosed in (), and a body enclosed in {}.
  + Basic structure:

S ->: WHILE '(' COND ')' '{' STMT ‘;’ '}'

COND -> ID RELOP ID

RELOP -> ‘>’ | ‘<’ | ‘>=’ | ‘<=’

STMT -> ID ‘=’ EXP

EXP -> EXP + EXP | EXP – EXP | ID | NO

* + Example: while (x < 10) { x = x + 1; }

**Algorithm :-**

1. Use Lex to define tokens for keywords (for, while), identifiers, operators, and literals (integers, parentheses, braces).
2. Define production rules in YACC for for and while loops that specify correct syntax structures.
3. For each rule, define actions in YACC to print confirmation of successful parsing or generate error messages for invalid syntax.
4. Compile Lex and YACC code to validate example loops.

**Program :-**

**Yacc program:**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern int yylex();

void yyerror(const char \*s);

%}

%token FOR LPAREN RPAREN LBRACE RBRACE SEMICOLON

%token ASSIGN NUMBER IDENTIFIER datatype WHILE

%token EQ NEQ LT LE GT GE

%token PLUS MINUS MUL DIV INCREMENT DECREMENT

%%

program : for\_loop { printf("The for loop is valid.\n"); }

| while\_loop { printf("The while loop is valid.\n"); } ;

for\_loop : FOR LPAREN init SEMICOLON condition SEMICOLON increment RPAREN LBRACE stmt\_list RBRACE

;

while\_loop : WHILE LPAREN condition RPAREN LBRACE stmt\_list RBRACE

init : DATATYPE IDENTIFIER ASSIGN NUMBER

;

DATATYPE : datatype;

condition : IDENTIFIER LT NUMBER

| IDENTIFIER LE NUMBER

| IDENTIFIER GT NUMBER

| IDENTIFIER GE NUMBER

;

increment : IDENTIFIER increment

;

stmt\_list : stmt\_list stmt

;

stmt: DATATYPE IDENTIFIER ASSIGN NUMBER SEMICOLON

;

%%

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

int main() {

if (yyparse() == 0) {

printf("Parsing completed successfully.\n");

} else {

printf("Parsing failed.\n");

}

return 0;

}

**Lex Program:**

%{

#include "y.tab.h"

#include <stdlib.h>

#include <string.h>

%}

%%

[ \t\n]+ ;

"for" { return FOR; }

"while" {return WHILE;}

"(" { return LPAREN; }

")" { return RPAREN; }

"{" { return LBRACE; }

"}" { return RBRACE; }

"int"|"char" {return datatype;}

";" { return SEMICOLON; }

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

[a-zA-Z\_][a-zA-Z0-9\_]\* { return IDENTIFIER; }

"=" { return ASSIGN; }

"==" { return EQ; }

"!=" { return NEQ; }

"<" { return LT; }

"<=" { return LE; }

">" { return GT; }

">=" { return GE; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MUL; }

"/" { return DIV; }

"++" { return INCREMENT; }

"--" { return DECREMENT; }

. { }

%%

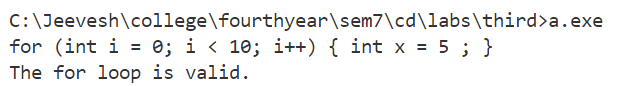
int yywrap()

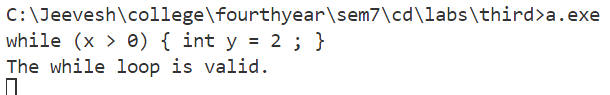
{

return 1;

}

**Output :-**

****

****

**Conclusion :-**

The YACC program to validate syntax of simple For/ While loop was successfully implemented and executed.

**Syntax Analyzer – Validate Input Expression**

**Experiment No. 4** **Date:-** 20/09/24

**Aim :-** Write a YACC program to validate the syntax of input expression and evaluate the expression if the string is valid.

**Theory :-**

**Mathematical Expressions**

Mathematical expressions consist of **operands** (numbers, variables) and **operators** (addition, subtraction, multiplication, etc.). The valid syntax of expressions includes the following components:

* **Operators**: +, -, \*, /, % (Addition, Subtraction, Multiplication, Division, Modulo).
* **Operands**: Numbers or identifiers (variables).
* **Parentheses**: Used for grouping sub-expressions, which dictate the order of evaluation.

A basic expression might look like:

3 + 5 \* (2 - 4)

This expression follows the order of operations: parentheses first, then multiplication, and finally addition.

**Ambiguity and Conflicts**

A **grammar** is ambiguous if there is an input string that can be structured in two or more different ways. For example, the rule: Expr: expr ‘-’ expr does not completely specify how complex inputs should be structured.

For an input like:

expr - expr – expr

YACC could reduce it in two ways:

* (expr - expr) - expr
* expr - (expr - expr)

This is called a **shift/reduce conflict**, where the parser is unsure whether to shift or reduce.

YACC resolves conflicts with the following disambiguating rules:

1. In a shift/reduce conflict, the default is to **shift**.
2. In a reduce/reduce conflict, the default is to **reduce by the earlier rule**.\

However, when there are conflicts due to errors in the input or the grammar, disambiguating rules may not be sufficient, and these conflicts must be addressed by rewriting the grammar or adjusting the rules.

**Precedence**

Precedence helps resolve parsing conflicts by defining the order in which operators are evaluated.

For example:

expr: expr OP expr

expr: UNARY expr

This grammar is ambiguous, as it doesn't specify operator precedence or associativity. To resolve this, **precedence levels** are defined for operators using YACC keywords: %left, %right, and %nonassoc.

For instance:

%left '+' '-'

%left '\*' '/'

This declares the precedence of the operators, with + and - having lower precedence than \* and /.

* **%left** indicates **left-associative** operators (evaluated from left to right).
* **%right** indicates **right-associative** operators (evaluated from right to left).
* **%nonassoc** indicates operators that should not associate (used for comparison operators like == or !=).

YACC resolves conflicts based on these precedence rules:

1. If there’s a shift/reduce conflict, YACC uses the operator with the higher precedence.
2. If the precedence levels are the same, associativity is used: left-associative implies a reduction, right-associative implies a shift.

The precedence rules help to disambiguate ambiguous grammar, allowing YACC to construct a parser that resolves conflicts in a manner consistent with common mathematical expression evaluation.

**Algorithm:-**

1. The first step is to tokenize the input expression using lex (or flex). This converts the input string into tokens (numbers, operators, parentheses).
2. Using YACC, define grammar rules that correspond to the syntax of valid expressions. The grammar will validate the structure of the input and detect any errors (such as misplaced operators or parentheses).
3. YACC parses the expression, we can compute the result. For example, whenever an operator is encountered, it applies the operation to the operands. This can be done during the parsing process itself.
4. If the input does not match the grammar, an error message is displayed, indicating the type of error (e.g., invalid operator, missing operand).

**Program :-**

**Yacc program:**

%{

#include<stdio.h>

#include<stdlib.h>

void yyerror(char \*);

int yylex(void);

int val=0;

%}

%token ID NUM

%start S

%%

S : ID '=' Exp '\n' {printf("Valid Expression, value is: %d", $3);}

Exp : Exp '-' Exp2 {$$ = $1 - $3;}

| Exp2 {$$ = $1;}

Exp2 : Exp2 '+' Exp3 {$$ = $1 + $3;}

| Exp3 {$$ = $1;}

Exp3 : Exp3 '\*' Exp4 {$$ = $1 \* $3;}

| Exp4 {$$ = $1;}

Exp4 : Exp4 '/' Exp5 {$$ = $1 / $3;}

| Exp5 {$$ = $1;}

Exp5 : '(' Exp ')' {$$ = $2;}

| NUM {$$ = $1;}

%%

void yyerror(char \*err)

{

fprintf(stderr, "Error: %s\n\n", err);

}

int main(void)

{

yyparse();

return 0;

}

**Lex program:**

%{

#include "expression.tab.h"

void yyerror(char\*);

%}

alpha [a-zA-Z\_]

digit [0-9]

%%

{alpha} {yylval = yytext[0]; return ID;}

{digit}+ {yylval = yytext[0] - '0'; return NUM;}

[\n\;()+-=\*/] {return yytext[0];}

[ \t] ;

. {yyerror(yytext);}

%%

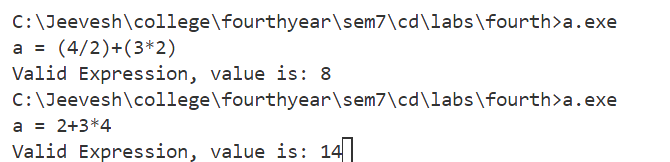
int yywrap(void)

{

return 1;

}

**Output :-**

****

**Conclusion:-**

The YACC program to validate the syntax of input expression and evaluate the expression if the string is valid was successfully implemented and executed.

**Syntax Analyzer – Validate For/ While Statement with multiple statements**

**Experiment No. 5**  **Date:-** 03/10/24

**Aim :-** Write a YACC program to validate syntax of for/ while loop with multiple statements inside the loop.

**Theory :-**

Syntax Analysis is a stage in the compilation process where a program’s syntax is analyzed to ensure that it conforms to the defined language grammar rules. The syntax analyzer, also known as a parser, takes tokens produced by the lexical analyzer and organizes them into a syntax tree based on the structure dictated by the grammar.

In this experiment, we use YACC (Yet Another Compiler Compiler) to create a parser that validates the syntax of for and while loops with multiple statements within them. YACC, combined with a lexical analyzer like Lex, allows us to define grammar rules for programming constructs and check whether an input conforms to those rules. Additionally, YACC can be used to evaluate and take actions when syntax rules are successfully matched.

**Components of a Loop Syntax Analyzer**

1. Loop Constructs:
   * Loops are fundamental constructs in programming used to repeat a set of statements. Common loop types include the for and while loops.
2. Grammar Rules for Loops:
   * The for loop typically has an initialization, a condition, an increment, and a body of statements.
   * For example:

for (int i = 0; i < 10; i++) {

statement1;

statement2;

...

}

* + The while loop has a condition and a body. For example:

while (i < 10) {

statement1;

statement2;

...

}

1. Multiple Statements Inside a Loop:
   * A valid loop syntax allows multiple statements inside the loop body, enclosed by { and } brackets. This setup forms a compound statement where each line is treated as an independent statement.
2. Syntax and Semantic Validation:
   * The syntax analyzer’s job is to ensure the structure of the loop is syntactically valid. For example, each component of the for loop (initialization, condition, increment, and statements) must follow the correct syntax. The while loop must have a condition and statements in the correct format.
   * YACC also allows semantic rules, such as validating that identifiers and operators are used correctly and that the loop has the proper closing brackets for multiple statements.

**Parsing Loops Using YACC**

Defining the Grammar in YACC:

* YACC grammar is written using production rules, where each rule specifies a valid pattern in the input language.
* For for and while loops, production rules define the structure of these constructs, capturing each required element and validating it.
* We define tokens and grammar rules for operators, expressions, and loops.

**Example Grammar Rules for for and while Loops:**

* A simplified grammar in YACC for parsing for and while loops might look like this:

program : for\_loop | while\_loop

for\_loop :

FOR LPAREN init SEMICOLON condition SEMICOLON increment RPAREN LBRACE stmt\_list RBRACE

while\_loop : WHILE LPAREN condition RPAREN LBRACE stmt\_list RBRACE

init : IDENTIFIER ASSIGN NUMBER

condition : IDENTIFIER RELOP NUMBER

increment : IDENTIFIER INCREMENT

stmt\_list : stmt\_list stmt | stmt

stmt : IDENTIFIER ASSIGN expression SEMICOLON

expression : expression ADD\_OP expression | expression MUL\_OP expression | NUMBER ;

**Explanation of Grammar Rules:**

* for\_loop: Matches a for loop with initialization (init), condition (condition), and increment (increment) statements, followed by multiple statements (stmt\_list) inside curly braces { }.
* while\_loop: Matches a while loop with a condition and multiple statements.
* stmt\_list: Allows for a series of statements within the loop body, where each statement assigns an expression to an identifier.
* expression: Supports arithmetic operations within statements, such as addition and multiplication.

**Ambiguity and Conflict Resolution**

Since YACC parsers can encounter shift/reduce and reduce/reduce conflicts, precedence rules help to resolve these conflicts in expressions and loop constructs:

* %left, %right, or %nonassoc directives define operator precedence for arithmetic and relational operators.
* Associativity rules for operators in YACC help to determine how expressions within statements are grouped, resolving potential conflicts.

**Algorithm :-**

1. Identify and declare the tokens required for parsing for and while loops, which include:
2. Define a starting rule, such as program, to handle either a for\_loop or while\_loop.
3. Define the structure of the for\_loop and while\_loop
4. Define sub-rules for each component
5. Set precedence for arithmetic and relational operators to resolve shift/reduce conflicts during parsing.
6. Implement the yyerror function to output error messages when syntax validation fails, indicating the position or type of syntax error.
7. Compile and run the YACC program to parse input containing for and while loops.

**Program :-**

**Yacc program:** %{

#include<stdio.h>

#include<stdlib.h>

void yyerror(char \*);

int yylex(void);

%}

%token ID RELOP FOR NUM WHILE IF

%start S

%%

S : ForLoop '{' Stmt ';' Stmt ';' Stmt ';' '}' {printf("\nValid For loop syntax\n\n");}

|WhileLoop '{' Stmt ';' Stmt ';' Stmt ';' '}' {printf("\nValid while loop syntax\n\n");}

ForLoop : FOR '(' Itr ';' Cond ';' Inc ')' {}

WhileLoop : WHILE '(' Cond ')' {}

Ifstmt : IF '(' Cond ')' Stmt {}

Stmt : ForLoop {}

| WhileLoop {}

| Ifstmt {}

| Itr {}

| ';' Stmt {}

Itr : ID '=' Exp {}

Cond : ID RELOP ID {}

| ID RELOP NUM {}

Inc : ID '+''+' {}

| ID '-''-' {}

| '+''+' ID {}

| '-''-' ID {}

Exp : Exp '+' Exp2 {}

| Exp2 {}

Exp2 : Exp2 '-' ID {}

| ID {}

| NUM {}

%%

void yyerror(char \*err)

{

fprintf(stderr, "Error: %s\n\n", err);

}

#ifdef YYDEBUG

yydebug = 0;

#endif

int main(void)

{

yyparse();

return 0;

}

**Lex program:  
 %{**

#include "exp5.tab.h"

void yyerror(char\*);

%}

alpha [a-zA-Z\_]

digit [0-9]

%%

"for" {return FOR;}

"while" {return WHILE;}

"if" {return IF;}

{alpha}({alpha}|{digit})\* {return ID;}

{digit}+ {return NUM;}

">="|"<="|"<"|">" {return RELOP;}

[\;(){}+-=] {return yytext[0];}

[ \t\n] ;

. {yyerror(yytext);}

%%

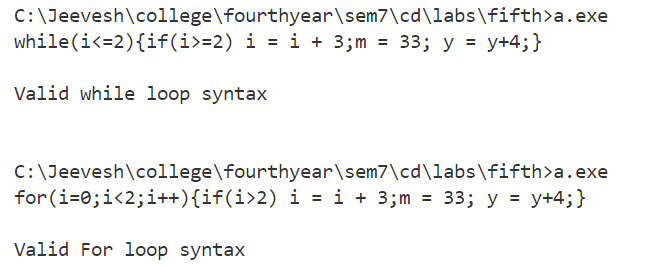
int yywrap(void)

{

return 1;

}

**Output :-**



**Conclusion:-**

The YACC program to validate syntax of for/ while loop with multiple statements inside the loop was successfully implemented and executed.

**Semantic Analyzer**

**Experiment No. 6** **Date:-** 04/10/24

**Aim :-** Write a YACC program to implement Semantic analyzer for the given input string.

**Theory :-**

Semantic analysis is a crucial phase in the compiler design process that follows syntax analysis (parsing). Its primary purpose is to ensure that the program's statements are meaningful and adhere to the rules of the programming language. Semantic analysis involves checking the program for semantic errors, which are related to the meaning of the constructs in the code rather than their syntactic structure. While syntax analysis ensures that the code is grammatically

correct (i.e., follows the rules of the language's grammar), semantic analysis verifies that the code makes sense in terms of the language's semantics.

Following are some semantic actions that the compiler might perform:

- Type Checking: This involves making sure that all the operands in an

operation are the correct type.

For eg.

int a, b;

float c, d;

a = 1;

b = 2;

a= a+c; // Error: a is int and c is float which is the wrong type.

b = 3.14159; // Error: Trying to store a float in an int

Type checking can also check if function calls have the same number of operands as function definition and they’re the correct type.

1. int func(int a, char b, float c);
2. int x = func(1, ‘c’, 4.2); // correct
3. float y = func(2.3, 31, ‘d’, 3.4); //Error: number of parameter and types don’t match

**Scope Management**

This involves making sure that variables have been efined in the scope that they’re being used in.

{

int x;

func(x); //Still in scope

}

func(x); //Error: x out of scope

**Symbol Table Management**

Semantic Analyser creates entries for identifiers in the symbol table. It checks that the name isn't already used in the scope, sets the types, checks if the variables used in an instruction are defined in the scope etc.

**Control Flow Analysis**

This step checks if control flow operations like loops and conditionals are used properly. It can check if a condition is never true for eg.

if(0){

printf(“Hello”);

}

It can also check for dead code like

void func(){

return;

printf(“Control will never flow here”);

}

**Importance of Semantic Analysis**

1. Semantic analysis ensures correctness.
2. It reduces runtime errors.
3. It helps programmers find bugs in the code.
4. It also helps in optimizations.

In YACC, we can perform the semantic analysis in the C Code section of the matched pattern.

For eg

Decl : D V {}

D : int { current\_type=type\_int;}

| float { current\_type=type\_float;}

| char { current\_type=type\_char;}

;

V : id { if(inSymbolTable($1) yyerror(“variable already defined”);

else defineVariable($1, current\_type);

}

| id, V { if(inSymbolTable($1) yyerror(“variable already defined”);

else defineVariable($1, current\_type);

}

;

In this example we first check if the variable has already been defined. If it is, we

generate an error. If it is not, we put it into the symbol table with the type that was

parsed in the D production.

**Algorithm :-**

1. Define the grammar of the language using YACC.
2. Identify different expressions and declarations in the input string.
3. Add semantic checks to the grammar rules in YACC.
4. Implement semantic actions for each rule. These actions include checking types, symbol tables, variable declarations, etc.
5. If a semantic error is detected (e.g., using an undeclared variable), print an error message.
6. Implement the YACC program to process the input string and apply the semantic checks.

**Program :-**

**Yacc program:**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void yyerror(const char \*s);

int yylex();

void InsertIntoTable(char \*, int);

void CHECK(char \*);

typedef struct {

char \*string\_value;

int num\_value;

} YYSTYPE;

#define YYSTYPE\_IS\_DECLARED 1

int datatype;

struct SYMBOL {

char name[32];

int dtype;

};

struct SYMBOL symbol\_table[10];

int n = 0;

%}

%union {

int num\_value;

char \*string\_value;

}

%token <num\_value> NUMBER

%token <string\_value> IDENTIFIER

%token INT FLOAT CHAR

%left '+' '-'

%left '\*' '/'

%%

program : declaration assignment {printf("valid expression")}

;

declaration : datatype variable ';';

datatype : INT { datatype = 1; }

| FLOAT { datatype = 2; }

| CHAR { datatype = 3; }

;

variable : IDENTIFIER ',' variable { InsertIntoTable($1, datatype); }

| IDENTIFIER { InsertIntoTable($1, datatype); };

assignment : IDENTIFIER '=' expr ';'

;

expr : expr '+' expr

| expr '-' expr

| expr '\*' expr

| expr '/' expr

| '(' expr ')'

| NUMBER

| IDENTIFIER { CHECK($1); };

%%

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

void InsertIntoTable(char \*variable, int dtype) {

for (int i = 0; i < n; i++) {

if (strcmp(variable, symbol\_table[i].name) == 0) {

printf("Variable already declared\n");

exit(0);

}

}

strcpy(symbol\_table[n].name, variable);

symbol\_table[n++].dtype = dtype;

}

void CHECK(char \*variable) {

int i;

for (i = 0; i < n; i++) {

if (strcmp(symbol\_table[i].name, variable) == 0) {

if (symbol\_table[i].dtype == 3) {

printf("Non-numeric datatype\n");

exit(0);

}

break;

}

}

if (i < n) {

printf("Valid Expression\n");

} else {

printf("Variable not declared\n");

exit(0);

}

}

int main() {

printf("Enter an expression: \n");

yyparse();

return 0;

}

**Lex program:**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "semantic.tab.h"

%}

%%

"char" { return CHAR; }

"int" { return INT; }

"float" { return FLOAT; }

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"=" { return '='; }

"," { return ','; }

"(" { return '('; }

")" { return ')'; }

";" { return ';'; }

[ \t\n]+ { /\* Ignore whitespace \*/ }

[0-9]+ { yylval.num\_value = atoi(yytext); return NUMBER; }

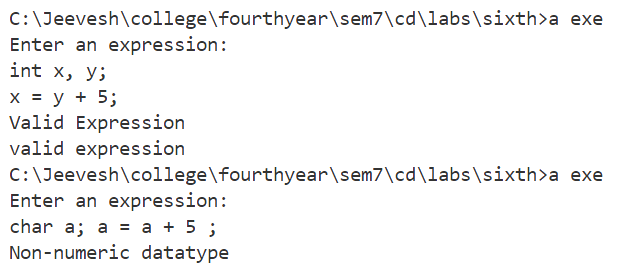
[a-zA-Z\_][a-zA-Z0-9\_]\* { yylval.string\_value = strdup(yytext); returnIDENTIFIER; }

. { printf("Unexpected character: %s\n", yytext); }

%%

int yywrap() { return 1; }

**Output :-**

****

**Conclusion:-**

The YACC program to implement Semantic analyzer for the given input string was successfully implemented and executed.

**Intermediate Code Generator**

**Experiment No. 7**  **Date:-** 10/10/24

**Aim:-** Write a YACC program to implement Intermediate Code Phase for the given input string.

**Theory :-**

In compiler design, intermediate code generation is a crucial phase where the compiler translates high-level source code into an intermediate representation (IR). This IR is an abstraction that lies between the source code and machine code, simplifying the compilation process by separating front-end (syntax analysis) and back-end (code generation) phases. Intermediate code generation allows for easier optimization and makes it more straightforward to retarget the compiler for different hardware architectures.

**Purpose of Intermediate Code Generation**

1. Portability: The IR is typically platform-independent, which means that the same intermediate code can be used to generate machine code for different architectures.
2. Optimization: IR provides an abstraction that enables code optimizations before the final machine code is generated.
3. Simplification: Translating to an IR simplifies handling complex source language constructs by breaking them down into smaller, more manageable operations.
4. Modularity: It separates the front-end of the compiler (parsing and analyzing source code) from the back-end (generating machine-specific code).

**Types of Intermediate Code Representations**

The IR comes in various forms, with the following being commonly used in compilers:

1. **Three-Address Code (TAC):**
   * Consists of instructions with at most three operands: a = b op c.
   * Uses temporary variables to store intermediate results.
   * Each operation is usually simplified into individual statements, making TAC suitable for analysis and optimization.
   * Example: For the expression a = b + c \* d, TAC might be:

t1 = c \* d

t2 = b + t1

a = t2

1. **Quadruples and Triples:**
   * Quadruples: These have four fields — operator, two operands, and a result.

(op, arg1, arg2, result)

* + Triples: These have three fields — operator, and two operands. They don’t use a result variable, but instead, use positional references.
  + Example for a = b + c \* d in Quadruple form:

(MULT, c, d, t1)

(ADD, b, t1, a)

1. **Syntax Trees and DAGs (Directed Acyclic Graphs):**
   * Syntax trees represent the hierarchical structure of expressions and statements in code.
   * DAGs represent common subexpressions and reuse computations where possible.

**Components of an Intermediate Code Generator**

1. Temporary Variables:
   * Used to store intermediate results in expressions.
   * Assigned names like t1, t2, etc., in the intermediate code.
2. Translation Rules:
   * These are grammar rules in the parser that define how each construct in the source language is converted into intermediate code.
   * For example, assignment statements and arithmetic expressions are broken down into smaller components using temporary variables.
3. Control Flow Statements:
   * Intermediate code must also support control flow, including loops (for, while) and conditional statements (if-else).
   * This often involves using labels and jumps for branching.

**Benefits of Intermediate Code**

1. Enhanced Optimization:
   * Compilers perform optimizations like constant folding, common subexpression elimination, and dead code elimination more easily on intermediate code.
   * Since the IR is platform-independent, these optimizations are applied once, rather than for each target architecture.
2. Platform Independence:
   * The IR allows compilers to generate code for multiple target architectures, simply by translating the IR into target-specific instructions.
3. Easier Error Detection and Debugging:
   * Intermediate code generation helps in identifying and isolating errors early in the compilation process, as each step is broken down.

**Example of Intermediate Code Generation**

Consider the following high-level code snippet:

x = (a + b) \* (c - d);

In intermediate code (e.g., TAC), this would be represented as:

t1 = a + b

t2 = c - d

x = t1 \* t2

Each subexpression is broken down, assigned to temporary variables, and evaluated independently. This breakdown allows compilers to optimize subexpressions separately.

**Ambiguities and Conflicts in Intermediate Code Generation**

During parsing, ambiguities may arise, especially with expressions that involve operators of different precedence or associativity (e.g., a - b - c could be parsed as (a - b) - c or a - (b - c)). These ambiguities are resolved by defining precedence and associativity rules for operators in the parser.

* **Shift/Reduce Conflict:** This occurs when the parser cannot decide whether to shift (read more input) or reduce (apply a rule).
* **Reduce/Reduce Conflict:** This occurs when the parser has multiple rules it could apply and cannot decide between them.

Using precedence rules, associativity, and temporary variables, YACC resolves these conflicts, ensuring consistent intermediate code output.

**Algorithm:-**

1. Declare tokens for operators (+, -, \*, /), identifiers, assignment (=), and constants.
2. Write YACC rules to recognize assignment and arithmetic expressions.
3. Break down complex expressions into smaller steps using temporary variables.
4. For each rule, generate intermediate code using temporary variables.
5. Print intermediate code for assignments and arithmetic expressions.
6. Use a counter for temporary variables (t1, t2, etc.) to track intermediate results.
7. Implement an error function to handle invalid syntax or expressions.
8. Parse the input string and print the generated intermediate code as output.

**Program :-**

**Yacc program:** %{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void yyerror(char\*);

void Gen(char\*);

char temp2[100];

int yylex(void);

int j = 0;

int i = 100;

%}

%union {

char \*string\_value;

}

%type <string\_value> exp

%type <string\_value> s

%token <string\_value> ID OP

%type <string\_value> COND

%type <string\_value> stmt stmt\_list

%token IF '(' ')' '{' '}' '=' SC

%left '+' '-'

%left '\*'

%%

s : IF '(' COND ')' '{' stmt\_list '}'

{ sprintf(temp2, "\nt%d=if(%s){%s}", i, $3, $6); sprintf($$, "t%d", i); i++; Gen(temp2); }

;

stmt\_list : stmt

| stmt stmt\_list { sprintf(temp2, "%s ; %s", $1, $2); $$ = strdup(temp2); }

;

stmt : ID '=' exp SC

{ sprintf(temp2, "\nt%d=%s=%s", j, $1, $3); sprintf($$, "t%d", j); j++; Gen(temp2); }

| s { $$ = strdup($1); }

;

exp : exp '+' exp

{ sprintf(temp2, "\nt%d=%s+%s", j, $1, $3); sprintf($$, "t%d", j); j++; Gen(temp2); }

| exp '-' exp

{ sprintf(temp2, "\nt%d=%s-%s", j, $1, $3); sprintf($$, "t%d", j); j++; Gen(temp2); }

| exp '\*' exp

{ sprintf(temp2, "\nt%d=%s\*%s", j, $1, $3); sprintf($$, "t%d", j); j++; Gen(temp2); }

| ID { $$ = strdup($1); }

;

COND : ID OP ID

{ sprintf(temp2, "\nt%d=%s %s %s", j, $1, $2, $3); sprintf($$, "t%d", j); j++; Gen(temp2); }

;

%%

void Gen(char \*val) {

FILE \*f = fopen("outputicg.txt", "a");

if (f) {

fputs(val, f);

fclose(f);

}

}

int yywrap() { return 1; }

void yyerror(char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

int main(int argc, char \*\*argv) {

yyparse();

return 0; // Return 0 for successful completion

}

**Lex program:**

%{

#include "seven.tab.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

%}

alpha [a-zA-Z\_]

digit [0-9]

%%

if { return IF; }

[\t\n ]+ { /\* Ignore whitespace \*/ }

[()] { return \*yytext; }

[{}] { return \*yytext; }

"=" { return '='; }

[-+\*] { return \*yytext; }

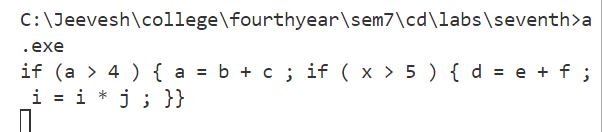
";" { return SC; }

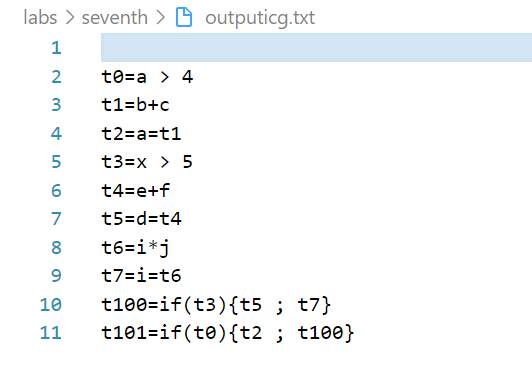
{digit}|{alpha}({alpha}|{digit})\* { yylval.string\_value = strdup(yytext); return ID; }

">"|"<"|">="|"<="|"!="|"==" { yylval.string\_value = strdup(yytext); return OP; }

%%

**Output :-**





**Conclusion:-**

The YACC program to implement Intermediate Code Phase for the given input string was successfully implemented and executed.

**Code Optimization**

**Experiment No. 8** **Date:-** 17/10/24

**Aim :-** Write a YACC program to implement Code optimization for given input.

**Theory :-**

Code optimization is a key phase in the compilation process where the compiler attempts to improve the code's performance and reduce resource usage without changing the program’s intended output. The primary objectives of optimization are to minimize the execution time, reduce memory usage, and enhance the overall efficiency of the program. Optimization is carried out on the intermediate code before generating the final machine code, ensuring that improvements are platform-independent.

**Goals of Code Optimization**

1. Increase Execution Speed: By minimizing the number of instructions and reducing redundant calculations, optimized code executes faster.
2. Reduce Memory Usage: Optimization minimizes the usage of registers and memory, leading to more efficient storage and retrieval.
3. Enhance Resource Utilization: By optimizing memory and processor resources, programs can perform better on limited hardware.
4. Improve Code Maintainability: Optimized code is often simpler and less redundant, making it easier to maintain.

**Types of Code Optimization**

1. Peephole Optimization:
   * A local optimization technique that inspects a small window of instructions, known as a "peephole," and optimizes this sequence by removing redundancies or simplifying instructions.
   * Common peephole optimizations include removing unnecessary load/store instructions and replacing expensive operations with cheaper ones.
2. Control Flow Optimization:
   * Analyzes and optimizes the control structures, including loops and branches.
   * Techniques like loop unrolling, loop invariant code motion, and branch prediction fall under this category.
3. Data Flow Optimization:
   * Improves the way data is handled and stored, focusing on data dependencies.
   * This includes constant propagation, common subexpression elimination, dead code elimination, and register allocation.
4. Machine-Specific Optimization:
   * These are low-level optimizations tailored to the target machine's architecture, often applied after intermediate code generation.
   * This includes instruction scheduling, instruction pipelining, and cache optimization.

**Common Optimization Techniques**

1. Constant Folding:
   * Simplifies expressions at compile-time if all operands are constants. For example, x = 2 + 3 can be optimized to x = 5.
2. Constant Propagation:
   * Replaces variables with their known constant values. For instance, if x = 5, and later we encounter y = x + 3, it can be transformed to y = 8.
3. Common Subexpression Elimination:
   * Detects and reuses the result of common expressions. For instance, in the expression a = b + c; d = b + c, d can reuse a instead of recalculating b + c.
4. Dead Code Elimination:
   * Removes code statements that do not affect the program outcome, such as unused variables and unreachable code segments.
5. Loop Optimization:
   * Loop Invariant Code Motion: Moves expressions within a loop that do not depend on the loop counter outside the loop.
   * Loop Unrolling: Reduces the overhead of branching by replicating the loop body multiple times.
6. Basic Block Partitioning :

Search header statements of all the basic blocks from where a basic block starts:

* + 1. First statement of a program.
    2. Statements that are target of any branch (conditional/unconditional).
    3. Statements that follow any branch statement.

Header statements and the statements following them form a basic block.

A basic block does not include any header statement of any other basic block

**Example of Code Optimization**

Consider the following unoptimized code:

x = 4 + 5;

y = x + 3;

z = y \* 2;

a = 4 + 5; // Redundant calculation

An optimized version would look like:

x = 9; // Constant folding

y = x + 3;

z = y \* 2;

a = x; // Common subexpression elimination

**Role of YACC in Code Optimization**

In a YACC (Yet Another Compiler Compiler) program, code optimization can be performed by defining rules that apply specific transformations to the intermediate code. By leveraging semantic actions associated with parsing rules, YACC can generate optimized intermediate code as it parses the source code.

For example, using YACC:

1. Constant Folding: Apply arithmetic calculations directly in the grammar rules to avoid generating redundant operations.
2. Dead Code Elimination: Use parsing rules to detect unused variables or unreachable code.
3. Common Subexpression Elimination: Identify and reuse temporary variables for repeated expressions within the rules.

**Algorithm :-**

1. Parse the input code and generate intermediate code for each statement.
2. Identify expressions involving only constants and compute their results during parsing.
3. Track assignments of constants to variables and replace variables with constant values where applicable.
4. Check for expressions that appear multiple times and reuse the result from the first calculation.
5. Remove statements or variables that do not impact the final result (e.g., unused assignments).
6. Output the optimized code based on the transformed intermediate representation.

**Program :-**

**Yacc program:**

%{

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_TEMP\_VARS 100

int yylex(void);

void yyerror(char\*);

void Gen(char\*);

int labelCounter = 100;

int tempCounter = 1;

typedef struct {

char expression[50];

char temp[10];

} TempVar;

// Array to store intermediate expressions and their associated temporary variables

TempVar tempVars[MAX\_TEMP\_VARS];

int tempVarCount = 0;

int findTemp(char\* expr);

int addTemp(char\* expr);

%}

%union {

char \*string\_value;

}

%token <string\_value> ID INTEGER SEMICOLON

%type <string\_value> expr stmt\_list stmt

%%

s : stmt\_list

stmt\_list : stmt stmt\_list { }

| stmt { }

;

stmt : ID '=' expr SEMICOLON

{

// Dead code elimination: Avoid redundant assignments like `x = x`.

if (strcmp($1, $3) != 0) {

char tempCode[100];

sprintf(tempCode, "%s = %s\n", $1, $3);

Gen(tempCode);

}

}

;

expr : expr '+' expr

{

char expr[50];

sprintf(expr, "%s + %s", $1, $3);

int tempIdx = findTemp(expr);

if (tempIdx == -1) {

// If not found, create a new temp variable

tempIdx = addTemp(expr);

sprintf(tempVars[tempIdx].temp, "t%d", tempCounter++);

sprintf(tempVars[tempIdx].expression, "%s", expr);

char tempCode[100];

sprintf(tempCode, "%s = %s + %s\n", tempVars[tempIdx].temp, $1, $3);

Gen(tempCode);

}

$$ = tempVars[tempIdx].temp;

}

| expr '\*' expr

{

char expr[50];

sprintf(expr, "%s \* %s", $1, $3);

int tempIdx = findTemp(expr);

if (tempIdx == -1) {

// If not found, create a new temp variable

tempIdx = addTemp(expr);

sprintf(tempVars[tempIdx].temp, "t%d", tempCounter++);

sprintf(tempVars[tempIdx].expression, "%s", expr);

char tempCode[100];

sprintf(tempCode, "%s = %s \* %s\n", tempVars[tempIdx].temp, $1, $3);

Gen(tempCode);

}

$$ = tempVars[tempIdx].temp;

}

| INTEGER { $$ = $1; }

| ID { $$ = $1; }

;

%%

// Check if expression already exists in tempVars array

int findTemp(char\* expr) {

for (int i = 0; i < tempVarCount; i++) {

if (strcmp(tempVars[i].expression, expr) == 0) {

return i;

}

}

return -1;

}

// Add a new temporary variable for a new expression

int addTemp(char\* expr) {

if (tempVarCount < MAX\_TEMP\_VARS) {

strcpy(tempVars[tempVarCount].expression, expr);

return tempVarCount++;

} else {

yyerror("Temporary variable limit exceeded.");

exit(1);

}

}

void Gen(char \*val) {

FILE \*f = fopen("optimized\_output.txt", "a");

fputs(val, f);

fclose(f);

}

void yyerror(char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

int main(int argc, char \*\*argv) {

yyparse();

return 1;

}

int yywrap() { return 1; }

**Lex program:**

%{

#include "eigth.tab.h"

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

%}

%%

[\t ]+ /\* ignore blank spaces \*/ ;

\n /\* ignore newline characters \*/;

";" { return SEMICOLON; }

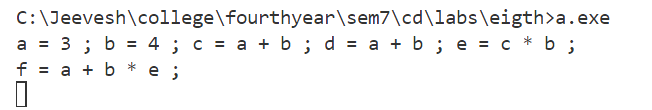
[0-9]+ { yylval.string\_value = strdup(yytext); return INTEGER; }

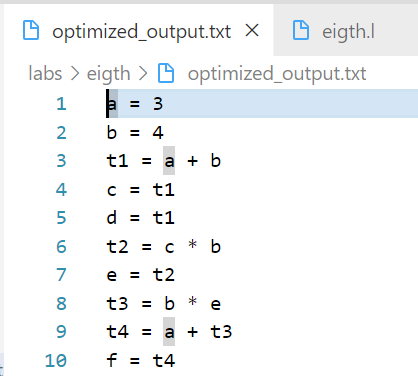
[a-zA-Z][a-zA-Z0-9]\* { yylval.string\_value = strdup(yytext); return ID; }

[-+\*=/] { return \*yytext; }

%%

**Output :-**

****

****

**Conclusion :-**

The YACC program to implement Code optimization for given input was successfully implemented and executed.

**Code Generation**

**Experiment No. 9**  **Date:-** 24/10/24

**Aim :-** Write a YACC program to implement Simple Code Generation for given input.

**Theory :-**

Code generation is a critical phase in compiler design where the compiler translates intermediate representations of a program into executable code (often assembly or machine code). This phase occurs after syntax and semantic analysis, where the program's structure and meaning are verified. The goal of code generation is to produce efficient, executable code that maintains the behavior specified in the source code.

1. **Intermediate Representation (IR):**
   * IR serves as a bridge between the high-level source code and the machine-level code.
   * It is a simplified, lower-level representation that retains the program's logical structure, making it easier for optimizations and code generation.
   * Common IR formats include three-address code (TAC), abstract syntax trees (AST), and control-flow graphs (CFG).
2. **Target Code:**
   * Target code is the final output of the code generation phase. It could be:
     + Assembly code, which is human-readable and maps closely to machine instructions.
     + Machine code, which is directly executable by the CPU.
   * For this experiment, we will focus on generating a simple assembly-like language with instructions for assignment, addition, multiplication, etc.
3. **Temporary Variables:**
   * During code generation, intermediate values need to be stored. For this, temporary variables are introduced.
   * These are unique placeholders (e.g., t1, t2, etc.) that hold intermediate results.
   * Temporary variables help manage complex expressions by breaking them down into simpler steps.
4. **Pseudo-Assembly Instructions:**
   * To make the code readable and close to machine code, we use pseudo-assembly instructions, such as:
     + MOV for assignments.
     + ADD, SUB, MUL, and DIV for arithmetic operations.
   * Each instruction operates on variables or temporary values and stores results in temporary variables or directly in target variables.
5. **Registers and Memory Allocation:**
   * In real-world compilers, the code generator also maps variables and temporaries to CPU registers and memory locations.
   * For simplicity, we assume an unlimited number of temporary variables and do not consider register allocation or memory constraints in this experiment.

**Approach to Code Generation Using YACC**

1. **Tokenization and Parsing:**
   * Lexical analysis breaks down the input program into tokens (e.g., ID, INTEGER, +, \*, etc.).
   * Parsing enforces the syntax rules and structures the tokens into an abstract syntax tree or parse tree.
2. **Semantic Actions in YACC:**
   * In YACC, semantic actions are added to the grammar rules to perform specific actions during parsing.
   * For code generation, each grammar rule is associated with an action that produces the corresponding target code instructions.
   * For example, when parsing an expression like a + b, the action for the + operation generates an ADD instruction.
3. **Translation of Expressions:**
   * Expressions are broken down into binary operations (like + and \*), each resulting in a temporary variable to store the intermediate result.
   * The generated code for an expression involves creating an ADD or MUL instruction for each operation and storing the result in a temporary.
4. **Translation of Assignments:**
   * Assignment statements (e.g., a = b + c) are handled by generating a MOV instruction.
   * First, the right-hand side (b + c) is evaluated, and the result is stored in a temporary variable.
   * Then, this temporary is assigned to the target variable (a in this example) with a MOV instruction.
5. **Handling Optimization:**
   * Even in a simple code generation phase, some basic optimizations can be applied, such as avoiding redundant computations and eliminating unnecessary temporaries.
   * However, in this experiment, we focus on generating straightforward code without additional optimization.

Example

a = b + c;

d = a \* e;

Generated Code:

ADD t1, b, c

MOV a, t1

MUL t2, a, e

MOV d, t2

This sequence represents a simple yet systematic translation from high-level expressions to low-level instructions.

Advantages of This Code Generation Approach

* **Simplicity:** The approach is straightforward and easy to implement for basic expressions and assignments.
* **Modularity:** Each operation (e.g., addition, multiplication) has a dedicated code generation rule, making the program modular and easy to expand.
* **Portability:** Since the generated code is in pseudo-assembly, it can be adapted for different architectures by modifying the instruction set.

**Algorithm :-**

1. Initialize temporary storage to store intermediate expressions and results.
2. Define YACC grammar rules for expressions and statements
3. Embed code generation actions in the rules
4. Output the generated code to the console or a file.

**Program :-**

**Yacc program:**

%{

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

int yylex(void);

void yyerror(char\*);

void Gen(char\*);

int i=1,c=0,j,f,count=0,prev,label=100;

char \*Array[10][10],\*temp,temp1[50],temp2[50];

int lineNo = 0;

yydebug = 0;

extern FILE \*yyin;

%}

%union {

char \*string\_value;

int intVal;

}

%type <string\_value> Exp S Cond

%type <intVal> Stmt Exp1

%token IF

%token <string\_value> ID

%token <string\_value> INTEGER RELOP

%start S

%%

S : IF '(' Cond ')' '{' Stmt '}'

{ $$="t";

Gen(temp2);

sprintf(temp2,"\n%d]if not %s, jmp %d\n", lineNo++,$3, $6);

Gen(temp2);

printf("%s\n", temp2); exit(0);

}

Cond : ID RELOP ID { sprintf($$, "%s %s %s", $1, $2, $3);}

Stmt : Stmt Exp1 ';' { $$ = $2 + 1;}

| { }

Exp1 : ID '=' Exp

{ $$=lineNo + 1;

Gen(temp2);

sprintf(temp2,"\n%d]mov %s, %s\n", lineNo++,$1,$3);

Gen(temp2);printf("%s", temp2);

}

Exp : Exp '+' Exp

{ $$="t";

sprintf(temp2,"\n%d]mov A, %s;add A, %s; mov t, A;", lineNo++, $1, $3);

Gen(temp2);

printf("%s", temp2);

}

| Exp '-' Exp

{ $$="t";

sprintf(temp2,"\n%d]mov A, %s;sub A, %s; mov t, A;", lineNo++, $1, $3);

Gen(temp2);

printf("%s", temp2);

}

| Exp '\*' Exp

{ $$="t";

sprintf(temp2,"\n%d]mov A, %s;mul A, %s; mov t, A;", lineNo++, $1, $3);

Gen(temp2);printf("%s", temp2);

}

| ID { $$=$1; }

| INTEGER { $$=$1; }

;

%%

void Gen(char \*val)

{

FILE \*f;

f=fopen("output.txt","a");

fputs(val,f);

fclose(f);

}

int yywrap() { return 1; }

void yyerror(char \*s)

{ }

int main( int argc, char \*\*argv ) {

yyparse();

return 1;

}

**Lex program:** %{

#include "nine.tab.h"

#include<stdio.h>

%}

%%

"if" {return IF;}

[\t ]+ /\* ignore the blank spaces \*/ ;

">"|"<"|">="|"<="|"!="|"==" { yylval.string\_value = strdup(yytext) ;return RELOP; }

[-+\*=\n,;(){}] { return \*yytext; }

[a-zA-Z]+ { // return valid tokens to yacc program

yylval.string\_value = strdup(yytext );

return ID;

};

[0-9]+ { // return valid tokens to yacc program

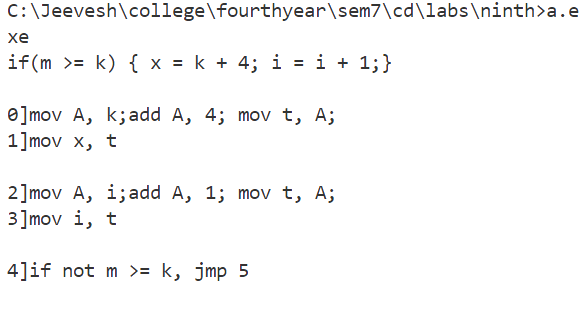
yylval.string\_value = strdup(yytext );

return INTEGER;

};

%

**Output :-**



**Conclusion:-**

The YACC program to implement Simple Code Generation for given input was successfully implemented and executed.

**Symbol Table**

**Experiment No. 10**  **Date:-** 31/11/24

**Aim :-** Write a YACC program to implement symbol table for given input.

**Theory :-**

In compiler design, a symbol table is a fundamental data structure that helps keep track of various identifiers in a program, such as variable names, function names, class names, etc. The symbol table stores essential information about each identifier, including its type, scope, memory address, and additional attributes needed during various stages of compilation (e.g., semantic analysis, code generation, and optimization).

The symbol table is generally implemented as a data structure that can be efficiently searched, such as a hash table, an array, or a binary search tree, depending on the requirements and scope of the compiler.

**Purpose of a Symbol Table**

1. **Identifier Management**: A symbol table manages all identifiers within a program, ensuring that each variable or function is uniquely identifiable within its scope.
2. **Semantic Analysis:** It helps in type checking by verifying data types, function parameters, and variable declarations. The compiler can quickly access an identifier’s type, scope, and other attributes to catch errors early in the compilation.
3. **Memory Management:** During code generation, the symbol table provides information on memory allocation for variables, objects, and functions.
4. **Optimization:** Symbol tables are used to optimize code, for example, by detecting unused variables, reducing redundant calculations, or identifying constants.

**Components of a Symbol Table**

A typical symbol table entry includes the following fields:

1. **Identifier Name:** The name of the variable, function, class, or constant.
2. **Data Type:** The data type associated with the identifier, such as int, float, or char.
3. **Scope:** Information on where the identifier is visible, whether globally or locally within a function or block.
4. **Memory Location:** The location in memory where the identifier’s value is stored, which is useful during code generation.
5. **Additional Attributes:**
   * Line Number: The line number where the identifier is declared, aiding in error reporting.
   * Attributes for Functions: Parameters, return type, and arity (number of parameters).
   * Attributes for Arrays: Size and dimensions.
   * **Modifiers:** Information like static, const, or volatile that modify the behavior of variables or functions.

**Structure and Operations in a Symbol Table**

The symbol table is often implemented using a structure that allows for efficient insertion, searching, and updating. Here are the essential operations:

1. **Insertion:** When a new identifier is declared, it is added to the symbol table with all necessary information. If an identifier is already in the symbol table within the same scope, an error is reported to avoid redeclaration.
2. **Lookup/Search:** When an identifier is used, the compiler looks it up in the symbol table to verify its existence, type, and scope. If an identifier is not found, it indicates that it was used before being declared, and an error is raised.
3. **Modification:** Sometimes, attributes of an identifier need updating, for instance, when a function is initially declared and later defined. The symbol table provides a way to modify entries efficiently.
4. **Display:** For debugging purposes, the contents of the symbol table can be printed out to show the identifiers and their attributes.

**Example of Symbol Table**

int x;

float y;

x = 5;

y = 3.14;

int x; // Error: redeclaration of 'x'

For this program, the symbol table will:

1. Add x with type int when it first encounters it on line 1.
2. Add y with type float on line 2.
3. Check if x already exists when it reaches line 5. Since x was declared earlier, it raises an error indicating redeclaration.

The symbol table after parsing the program above will look like:



**Algorithm :-**

1. Initialize the Symbol Table Structure:
   1. Use an array or a hash table to store each symbol’s information. In this case, we can use a structure to store identifier attributes (name, type, etc.).
2. Parse Input and Identify Declarations:
   1. Use a lexical analyzer (scanner) to tokenize input (e.g., int a;, float b;).
   2. Parse these tokens to extract identifier names and their types.
3. Insert Symbols into the Symbol Table:
   1. For each identifier, check if it already exists in the table:
      1. If not, add a new entry with attributes (name, type, scope).
      2. If it exists, handle accordingly (e.g., report redefinition errors).
4. Display or Access Symbol Table:
   1. Once parsing is complete, display the contents of the symbol table to verify the identifiers and attributes.

**Program :-**

**Yacc program:**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void yyerror(const char \*s);

int yylex();

void InsertIntoTable(char \*, int);

void printtable();

int datatype;

struct SYMBOL {

char name[32];

int dtype;

};

struct SYMBOL symbol\_table[10];

int n = 0;

%}

%union {

char \*string\_value;

};

%token <string\_value> IDENTIFIER

%token INT FLOAT CHAR

%left '+' '-'

%left '\*' '/'

%%

s : program { printtable(); }

;

program : declaration ';' '\n' {}

| declaration ';' program {}

;

declaration : datatype variable {}

;

datatype : INT { datatype = 1; }

| FLOAT { datatype = 2; }

| CHAR { datatype = 3; }

;

variable : IDENTIFIER ',' variable { InsertIntoTable($1, datatype); }

| IDENTIFIER { InsertIntoTable($1, datatype); }

;

%%

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

void InsertIntoTable(char \*variable, int dtype) {

int i;

for (i = 0; i < n; i++) {

if (strcmp(variable, symbol\_table[i].name) == 0) {

printf("Variable %s already declared.\n", variable);

exit(1);

}

}

strcpy(symbol\_table[n].name, variable);

symbol\_table[n++].dtype = dtype;

}

void printtable() {

printf("--symbolTable--\n");

printf("SYMBOL DTYPE \n");

int i;

for (i = 0; i < n; i++) {

printf("%s %d\n", symbol\_table[i].name,symbol\_table[i].dtype);

}

}

int main() {

printf("Enter an expression: \n");

yyparse();

return 0;

}

**Lex program:** %{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "symbolTable.tab.h"

%}

%%

"char" { return CHAR; }

"int" { return INT; }

"float" { return FLOAT; }

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"=" { return '='; }

"," { return ','; }

"(" { return '('; }

")" { return ')'; }

";" { return ';'; }

[\n] {return '\n';}

[ \t] {}

[a-zA-Z\_][a-zA-Z0-9\_]\* { yylval.string\_value = strdup(yytext); return IDENTIFIER; }

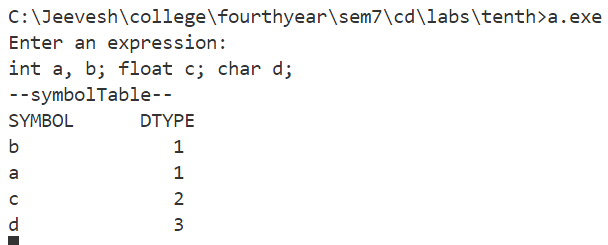
. { printf("Unexpected character: %s\n", yytext); }

%%

int yywrap()

{ return 1; }

**Output :-**

****

**Conclusion :-**

The YACC program to implement symbol table for given input was successfully implemented and executed.