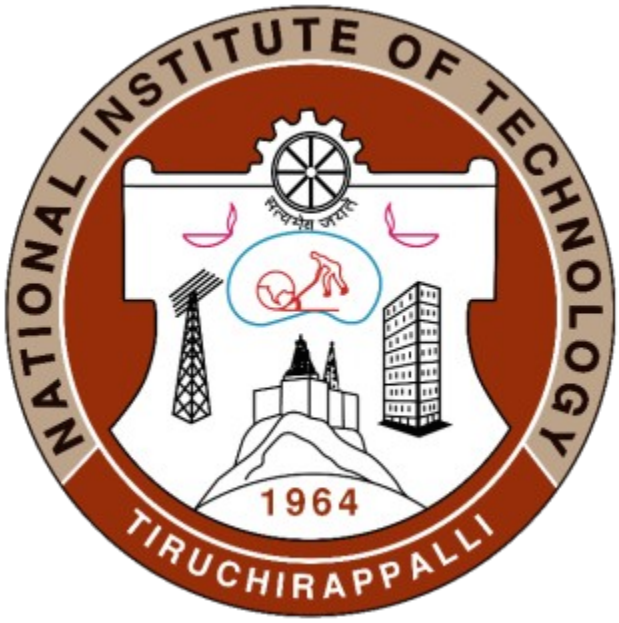


COMPILER DESIGN PROJECT REPORT

TOPIC: C COMPILER OF NESTED FOR LOOP WITH IF-ELSE CONSTRUCTS.

Course code: CSPC62
Professor: Dr. Sitara K
Class: CSE-B

Team members:

Kumar Swapnil – 106120056
Nitin Kanan – 106120078
Sanket Borkar – 106120102
Vikash Kumar Mishra – 106120140
Vipul Patel – 106120142

Introduction:

This project aims to undertake a sequence of experiments to design and implement various phases of a compiler for detecting nested ‘for loop’ using if-else constructs. Along with it the following constructs also can be handled by the compiler: -

1. Data Types: int, char data types with all its sub-types. Syntax: int a=3;
2. Comments: Single line and multiline comments,
3. Keywords: char, else, for, if, int, long, return, short, signed,
struct, unsigned, void, while, main
4. Identification of valid identifiers used in the language,
5. Looping Constructs: It will support nested for and while loops.
6. Conditional Constructs: if...else-if...else statements,
7. Operators: ADD(+), MULTIPLY(*), DIVIDE(/), MODULO(%), AND(&), OR(|)
8. Delimiters: SEMICOLON(;), COMMA(,)
9. Structure construct of the language, Syntax: struct pair{ int a; int b};
10. Function construct of the language, Syntax: int func(int x)
11. Support of nested conditional statement,
12. Support for a 1-Dimensional array. Syntax: char s[20];

Concept:

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.

The phases are as below:

Analysis

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Intermediate Code Generation

Synthesis

1. Code Optimization
2. Code Generation

Lexical Analysis:

The Lexical Analyzer is the first phase of the Analysis (front end) stage of a compiler. In layman’s terms, the Lexical Analyzer (or Scanner) scans through the input source program character by character, and identifies ‘Lexemes’ and categorizes them into ‘Tokens’. These ‘tokens’ are represented as a symbol table, and is given as input to the Parser (second phase of the front end of a compiler).

Syntactic Analysis and Parser:

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. We expect the parser to report any syntax errors in an intelligible manner and to recover from the commonly occurring errors to continue processing the remainder of the program. Parser detects the following types of errors:

1. Errors in structure
2. Missing operator
3. Misspelt keywords
4. Unbalanced parenthesis

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

There are generally 3 types of parsers for grammars:

1. Universal
2. Top-down
3. Bottom-up

The methods commonly used in compilers can be classified as being either top-down (parse from root to leaves) or bottom-up (parse from leaves to root).

Semantic Analysis:

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. We expect the parser to check the structure of the input program and report any syntax errors. Semantic analysis phase checks the semantics of the language.

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

Semantic analysis typically involves in following tasks:

1. Type Checking – Data types are used in a manner that is consistent with their definition (i.e., only with compatible data types, only with operations that are defined for them, etc.)
2. Label Checking – Labels references in a program must exist.
3. Array Bound Checking – When declaring an array, subscript should be defined properly.

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

1. Type mismatch
    a. Return type mismatch.
    b. Operations on mismatching variable types.

2. Undeclared variable
    a. Check if variable is undeclared globally.
    b. Check if variable is visible in current scope.
3. Reserved identifier misuse.
    a. Function name and variable name cannot be same.
    b. Declaration of keyword as variable name.
4. Multiple declaration of variable in a scope.
5. Accessing an out of scope variable.
6. Actual and formal parameter mismatch.

Intermediate Code Generation:

Intermediate Code Generation phase is the glue between frontend and backend of the compiler design stages. The final goal of a compiler is to get programs written in a high-level language to run on a computer. This means that, eventually, the program will have to be expressed as machine code which can run on the computer. Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called intermediate code. It provides lower abstraction from source level and maintain some high level information.

Intermediate Code can be represented in many different formats depending whether it is language-specific (e.g. Bytecode for Java) or language-independent (three-address-code). We have used three-address-code to make it independent.

Most common independent intermediate representations are:

1. Postfix notation
2. Three Address Code
3. Syntax tree

Three Address Code:

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

General representation: x = y op z

An address can be a name, constant or temporary.

- Assignments x = y op z; x = op y.
- Copy x = y.
- Unconditional jump goto L.
- Conditional jumps if x relop y goto L.
- Parameters param x.
- Function call y = call p

Code Optimization:

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives:

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Loop Optimization Techniques:

1. Code Motion or Frequency Reduction – The loop invariant statements are brought out of the loop.
2. Loop Jamming – Two or more loops are combined in a single loop. It helps in reducing the compile time.
3. Loop Unrolling – It increases the program's speed by eliminating the loop control and test instructions.

Code generation:

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- Target language.
- IR Type.
- Selection of instruction.
- Register allocation.
- Ordering of instructions.

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

- Register descriptor.
- Address descriptor.

Implementation:

Lexical Analysis:

The Regular Expressions for most of the features of C are fairly straightforward. However, a few features require a significant amount of thought, such as:

- The Regex for Identifiers: The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.
- Multiline comments should be supported: To implement it a proper regular expression was written along with that lookahead character set for operators were thought so to resolve conflict with the division operator.
- Literals: Different regular expressions have been implemented in the code to support all kinds of literals, i.e., integers, floats, strings, etc.
- Error Handling for Incomplete String: Open and close quote missing, both kind of errors have been handled in the rules written in the script.
- Error Handling for Unmatched Comments: This has been handled by adding lookahead characters to operator regular expression. If there is an unmatched comment, then it does not match with any of the patterns in the rule. Hence it goes to default state which in turn throws an error.
- Error Handling for unclean integer constant: This has been handled by adding appropriate lookahead characters for integer constant. E.g., int a = 786rt, is rejected as the integer constant should never follow an alphabet.

At the end of the token recognition, the lexer prints a list of all the identifiers and constants present in the program. We use the following technique to implement this:

- We maintain two structures one for symbol table and other for constant tableone corresponding to identifiers and other to constants.
- Four functions have been implemented lookupST( ), lookupCT( ), these functions return true if the identifier and constant respectively are already present in the table. InsertST( ), InsertCT( ) help to insert identifier/constant in the appropriate table.
- Whenever we encounter an identifier/constant, we call the insertST() or insertCT() function which in turns call lookupST( ) or lookupCT( ) and adds it to the corresponding structure.
- In the end, in main( ) function, after yylex returns, we call printST( ) and printCT( ), which in turn prints the list of identifier and constants in a proper format.

Syntax Analysis or Parser:

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex. These were:

A. The Regex for Identifiers
B. Multiline comments should be supported
C. Literals
D. Error Handling for Incomplete String
E. Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules. The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type, value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error.

The following functions were written in order to maintain symbol table:

1. LookupST() - This function checks whether the token is already present in the symbol table or not. If yes it returns 1 else 0.(Called by Scanner)
2. InsertST() - This function installs the token in the symbol table if it is not already present along with the token class.(Called by Scanner)
3. InsertSTtype() - This function appends the datatype of the identifier in the symbol table. (Called by Parser).
4. InsertSTvalue()- This function appends the value of the identifier in the symbol table. (Called by Parser).
5. InsertSTline()- This function appends the line of declaration of the identifier in the symbol table. (Called by Parser).
6. LookupCT() - This function checks whether the token is already present in the constant table or not. If yes it returns 1 else 0.(Called by Scanner).
7. InsertST() - This function installs the token in the constant table if it is not already present along with the token class.(Called by Scanner)
8. PrintST() - This function displays the entire content of the symbol table.
9. PrintCT() - This function displays the entire content of the constant table.

Semantic Analysis:

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex. These were:

A. The Regex for Identifiers
B. Multiline comments should be supported
C. Literals
D. Error Handling for Incomplete String
E. Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.

The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type , value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error. Along with this semantic actions were also added to each production rule to check if the structure created has some meaning or not.

The following functions were written in order to check semantics:

1. insertSTnest() - This function was used to insert the nesting value of an identifier to the symbol table.
2. insertSTparamscount() - Inserts the count of number of parameters for a function
3. getSTparamscount()- Get the number of parameters in a function
4. deletedata() - This function deletes the data when its scope is over.
5. checkscope() - It checks whether the identifier is declared in the current scope or not.
6. check_id_is_func()- Check if the identifier is declared as a function or not.
7. checkarray() - It checks whether the identifier is of array data type or not. If yes it returns true else false.
8. duplicate()- It checks if the identifier was already declared or not.
9. check_duplicate() - It checks if the function is re-declared or not.
10. check_declaration() - It checks if the function is declared or not,
11. check_params()- it checks whether the parameters used in function definition are not of type void.
12. char gettype() - it returns the first char of the data type of identifier.

Intermediate Code Generation:

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex. These were:

A. The Regex for Identifiers
B. Multiline comments should be supported
C. Literals
D. Error Handling for Incomplete String
E. Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.

The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type , value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error. Along with this semantic actions were also added to each production rule to check if the structure created has some meaning or not. Then we added the function to generate the 3 address code with production so that we can generate the desired intermediate code. In order to generate 3 address code we made use of explicit stack. Whenever we came across an operator, operand or constant we pushed it to stack. Whenever reduction occurred (Since LALR(1) parser is bottom up parser it evaluates SDT when reduction occurs) codegen( ) function generated the 3 address code by creating a new temporary variable and by making use of the entries in the stack, after that it popped those entries from the stack and pushed the temporary variable to the stack so that it gets used in further computation. Similarly functions like labels were used to assign appropriate labels while using conditional statements or iterative statements. All the functions used are described below :

1. codegen( ) : This function is called whenever a reduction of an expression takes place. It creates the temporary variable and displays the desired 3 address code i.e x = y op z.
2. codegencon( ) : This function is especially written for reductions of expression involving constants since its 3 address code is x op z.
3. isunary( ) : This function checks if the operator is an unary operator like '++'. If so it returns true else false.
4. genunary( ) : This function is specifically designed to generate 3 address code for unary operations. It makes use of isuanary function mentioned above. E.g. if a = i++ then it converts into t0 = i + 1, a = t0.
5. codeassign( ) : This function is specifically designed for assignment operator. It assigns the final temp variable value (after all the evaluation) to the desired variable.
6. label1( ) : It is used while evaluating conditions of loops or if statement. If the condition is not satisfied then it states where to jump to i.e. on which label the control should go.
7. label2( ) : It is used when the statement block pertaining to if statement is over. It tells where the control flow should go once that block is over i.e. it jumps the else statement block.
8. label3( ) : it is used after the whole if else construct is over. It gives label that tells where to jump after the if block is executed.
9. label4( ) : it is used to give labels to starting of loops.
10. label5( ) : it is used after the statement block of the loop. It indicates the label to jump to and also generates the label where the control should go once the loop is terminated.
11. funcgen( ) : it indicates beginning of a function.
12. funcgenend( ) : it indicates the ending of a function.
13. arggen( ) : it displays all the reference parameters that are used in a function call.
14. callgen( ) : it calls the function i.e. displays the appropriate function call according to 3 address code.
15. Itoa( ) : it worked as utility function since we had to name temporary variables and labels it was used to convert int to string and used several functions like reverse , swap to do it.

Result:

The C compiler of nested for loop using if-else constructs is implemented and tested successfully on each phase with the test cases and the output is verified.

We were able to successfully parse the tokens recognized by the flex script for C. The output displays the set of identifiers and constants present in the program with their types, values, and line of declaration. Also nesting values changes dynamically as the program ends its made infinite. The parser generates error messages in case of any syntactical errors in the test program or any semantic error. Also we are displaying the 3 address code generated by our yacc script.

Test Case 1:-

```c
#include <stdio.h>

void main()
{
        int i,n,a=0;
    for(i=0;i<n;i++)
        {
              if(i<n)
              {
                a=a+1;
              }
        }
        }
}
```

```
================================  Running TestCase 1  ================================
func begin main
t0 = 0
t1 = 0
i = t1
L0:
t2 = i < n
IF not t2 GoTo L1
t3 = i + 1
i = t3
t4 = i < n
IF not t4 GoTo L2
t5 = 1
t6 = a + t5
a = t6
GoTo L3
L2:
L3:
GoTo L0:
L1:
func end

16 syntax error }
Status: Parsing Failed - Invalid
```

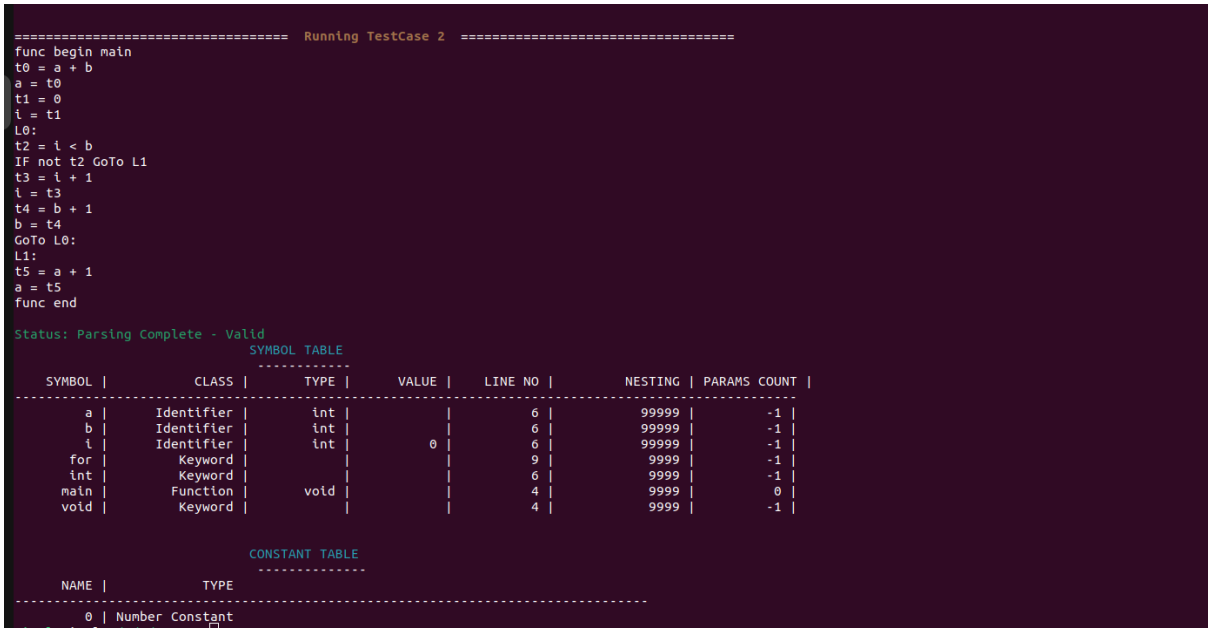
Test Case 2-

```c
#include <stdio.h>


void main()
{
        int a,b,i;

                a = a+b;
                for(i=0;i<b;i++)
                {
                        b++;
                }
                a++;

}
```



```
================================  Running TestCase 2  ================================
func begin main
t0 = a + b
a = t0
t1 = 0
i = t1
L0:
t2 = i < b
IF not t2 GoTo L1
t3 = i + 1
i = t3
t4 = b + 1
b = t4
GoTo L0:
L1:
t5 = a + 1
a = t5
func end

Status: Parsing Complete - Valid
                            SYMBOL TABLE
                             ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO |         NESTING | PARAMS COUNT |
-------------------------------------------------------------------------------------------------------
         a |      Identifier |        int |            |          6 |           99999 |         -1 |
         b |      Identifier |        int |            |          6 |           99999 |         -1 |
         i |      Identifier |        int |          0 |          6 |           99999 |         -1 |
       for |         Keyword |            |            |          9 |            9999 |         -1 |
       int |         Keyword |            |            |          6 |            9999 |         -1 |
      main |        Function |       void |            |          4 |            9999 |          0 |
      void |         Keyword |            |            |          4 |            9999 |         -1 |


                            CONSTANT TABLE
                             -------------
      NAME |            TYPE
-----------------------------------------------------------------------------
         0 | Number Constant
```

Github Link:-

https://github.com/vipulnitt/Compiler-project/