SRM INSTITUTE OF SCIENCE & TECHNOLOGY

Kattankulathur, Chennai, Tamil Nadu-603203

# **18CSC304J-COMPILER DESISN**

Project Report

Implementation of a Standard C++ Compiler using Lex, Yacc and C++

|  |  |  |
| --- | --- | --- |
| Saket Kumar Baranwal | Shashwat Raj | Tanmay Dutta |
| RA1911003010414 | RA1911003010419 | Ra1911003010418 |

**REPORT**

# Problem Statement and Introduction

To implement a Compiler for C++ using Lex, Yacc and C++ as the language to code the compiler in. The input could be any valid C++ Program with Functions, Conditions and Looping constructs. The output would be the expected output from “g++” when run on a standard Unix shell.

# Architecture of the Language

The syntax is the same as C++, for all the constructs that we are focusing on. We have made sure to implement the semantics of the language as close to ISO C++ as possible. We have handled all errors and border cases for every construct in our language and have added many error checking mechanisms too.

# Grammar

We used a regular Backus Normal Form Grammar to develop the entire language. Most of the productions inspired by ISO C++ and a few constructs that we created to provide some novelty to our project.

The grammar has been pasted below:

preprocessing-token:

#include header-name  
 #define literal literal

declaration-statement:  
 attribute declaration-specifier

function-definition:  
 keyword identifier (identifier-sequence) function-body

function-body:   
 compound-statement

token:  
 identifier  
 keyword   
 literal   
 operator-token   
 punctuator

header-name:  
 < string-path >  
 " string-path "

identifier:

identifier-nondigit  
 identifier digit

statement:   
 labeled-statement  
 expression-statement   
 compound-statement   
 selection-statement   
 iteration-statement   
 jump-statement

labeled-statement   
 identifier : statement

compound-statement:   
 { statement-seq }

statement-seq:   
 statement  
 statement-seq statement

selection-statement:  
 if ( condition ) statement  
 if ( condition ) statement else statement

condition:  
 expression  
 type-specifier-seq declarator = assignment-expression

iteration-statement:  
 while ( condition ) statement  
 for ( for-range-declaration : for-range-initializer ) statement

for-range-declaration:  
 attribute-specifier-seqopt type-specifier-seq declarator

for-range-initializer:  
 expression braced-init-list

jump-statement:  
 break ;  
 continue ;  
 return expressionopt ;  
 return braced-init-listopt ;

expression:   
 multiplicative-expression  
 additive-expression  
 relational-expression  
 equality-expression  
 logical-and-expression  
 logical-or-expression  
 conditional-expression  
 assignment-expression

multiplicative-expression:   
 pm-expression  
 multiplicative-expression \* pm-expression   
 multiplicative-expression / pm-expression   
 multiplicative-expression % pm-expression

additive-expression:   
 multiplicative-expression  
 additive-expression + multiplicative-expression   
 additive-expression - multiplicative-expression

relational-expression:   
 shift-expression  
 relational-expression < shift-expression   
 relational-expression > shift-expression   
 relational-expression <= shift-expression   
 relational-expression >= shift-expression

equality-expression:   
 relational-expression  
 equality-expression == relational-expression   
 equality-expression != relational-expression

logical-and-expression:   
 Inclusive-or-expression  
 logical-and-expression && inclusive-or-expression

logical-or-expression:   
 logical-and-expression  
 logical-or-expression || logical-and-expression

conditional-expression:   
 logical-or-expression  
 logical-or-expression ? expression : assignment-expression

assignment-expression:   
 Conditional-expression  
 logical-or-expression assignment-operator initializer-clause  
 throw-expression

assignment-operator:  
 =   
 \*=   
 /=   
 %=   
 +=   
 -=   
 >>=   
 <<=   
 &=   
 ^=   
 |=

identifier-nondigit:   
 nondigit

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

id-expression:   
 identifier

Identifier-digit:  
 0   
 1   
 2   
 3   
 4   
 5   
 6   
 7   
 8   
 9

keyword:   
 bool  
 break   
 char   
 continue   
 double   
 else   
 false   
 float  
 for  
 if  
 int   
 long   
 return   
 true   
 void

punctuator:   
 {  
 }   
 [   
 ]   
 #   
 (   
 )   
 ;   
 :   
 ?

literal:  
 integer-literal  
 character-literal   
 floating-literal   
 string-literal  
 boolean-literal

integer-literal:  
 decimal-literal integer-suffixopt

decimal-literal:   
 nonzero-digit  
 decimal-literal digit

nonzero-digit:  
 1  
 2   
 3   
 4   
 5   
 6   
 7   
 8   
 9

c-char:  
 any member of the source character set except the single quote ', backslash \, or new-line character   
 escape-sequence  
 universal-character-name

escape-sequence:   
 simple-escape-sequence

simple-escape-sequence:   
 \'  
 \"   
 \\   
 \n   
 \t

sign:  
 +  
 -

digit-sequence:   
 digit   
 digit-sequence digit

string-literal:   
 s-char-sequence

s-char-sequence:   
 s-char  
 s-char-sequence s-char

s-char:  
 any member of the source character set except the double-quote ", backslash \, or new-line character  
 escape-sequence  
 universal-character-name

boolean-literal:  
 false  
 true

# Design Strategies and Implementation Details

### Symbol Table Generation

We are using a linked list of structures to implement our Symbol Table. It is created on the Heap and is called to a print function just before the compilation ends. The print function outputs a formatted symbol table to STDOUT.

A node of the symbol table has the following structure.

1. Line
2. Name
3. Scope (-1 for Global, 0 for Main and >0 for extraneous scopes)
4. Value
5. ID
6. Data Type

Which is displayed in a tabular form during the end of the program to represent the symbol table that has been generated during the first phase.

We store the entire table in a dynamic list-esque data structure.

### Abstract Syntax Tree

The syntax tree is generated entirely using Yacc. We create an Abstract Syntax Tree data structure to store the Syntax Tree, which is essentially something that mimics the way Yacc parses the grammar.

The treeNode has:

1. nodeType
2. string
3. value
4. dataType
5. lineNo
6. Nchildren

Every production is evaluated to the tree. So, as soon as we encounter a non-terminal, we process the expression through the Yacc grammar to generate nodes in the AST. It contains all the meta data as enumerated above. We basically use Dollar variables ($) in Yacc that is used to store the various tokens and expressions that we parse along the way.

To print (export) the syntax tree as a text file, we essentially print all the above information to a single line for every node and use indentation to mimic a “tree” visualization. All of this is exported to a single file in the root directory.

The print function uses a structure that contains the follows:

1. Line Number
2. nodeType
3. string
4. value
5. dataType

Upon compilation, the tree is then printed out to a file, as a graphically accurate representation of a tree.

### Intermediate Code Generation

Intermediate Code Generation is implemented using various functions and data structures that are used to generate and store the Intermediate Codes. We have a list of structures of type Quadruple to store the Quadruples generated by the compiler. Several functions are used to accomplish the required processing. We then use a print function that neatly formats the IC and prints it onto STDOUT.

### Error Handling

[TYPE HERE]

### Target Code Generation

[TYPE HERE]

# Sample Input

A screenshot of a cell phone

Description automatically generated

# Abstract Syntax Tree Output

A screenshot of a cell phone

Description automatically generated

# Phase 4 + ICG Output

# A screenshot of a cell phone Description automatically generated

# Results

Our Compiler has been able to compile and generate code for the Sample Input files pretty accurately. It can detect a plethora of errors and satisfactorily compile and produce optimal code. We are confident that by the end of the semester, we will be able to build a reputable compiler for C++.

# Possible Shortcoming

The compiler we built is a mini-compiler and doesn’t entirely mimic or compile all C++ code. We haven’t implemented Object Oriented Programming or STLs that make up the majority of C++. And the functions we have generated have been optimized specifically for the current language and grammar that has been elaborated on in this document. The generated code may be a bit buffed up compared to a highly optimized version of the same, generated by an Official C++ Compiler.

# Conclusions

We can conclude that a satisfactorily accurate compiler can be build using Lex and Yacc for a number of different languages spreading across multiple genres. We can conclude that the various phases of a standard compiler can be built and implemented using these tools and by following all regulations, a standard compiler can be built for almost any language.

# Enhancements

We have included one construct of our own and plan to add a few more techniques we have learned during the duration of this course that may not be present in the standard C++ compiler.

——————————————————— Thank you———————————————————