Indian Institute of Information Technology, Allahabad

COD Project

Mini C-Java(CJ) Compiler

Submitted to :

Dr. A Agarwal
Dr. T Pant

Submitted by :

Sunil Kumar (RIT2009021)
Shivani Maheshwari (RIT2009037)
Pradeep Kumar Mishra (RIT2009025)
Kshitij Bansal (RIT2009078)
Saurabh Pandey (RIT2009021)

Content

Introduction

The name **compiler** is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g. assembly language or machine code). A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization. The term **compiler-compiler** is sometimes used to refer to a **parser generator**, a tool often used to help create the lexer and parser.

Tools Used

Tools used for development of the compiler are:

1. **Flex**

   Flex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Flex. The Flex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Flex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Flex, the corresponding fragment is executed. The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

2. **Yet Another Compiler Compiler(YACC)**

   Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the

user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions. Yacc is written in a portable dialect of C and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C. The parser generated by Yacc requires a lexical analyzer. Lexical analyzer generators, such as Lex or Flex are widely used for this purpose.

Language Identification

It is widely known that compiler relies on the file type to compile it successfully. Therefore, the first step is identification of the file whether it is a C-file or a Java-file. There are some points on basis of which it can determine the file-type.

1) On the basis of file-extension or suffix **.c** for C file and **.java** for java file (if provided).

2) On the basis of linkage of the C or Java library in the file.
   Example: #include <library-name(header file)>
   #include “header file” **for C**
   OR
   import package-name.class-name **for Java**

3) C is Procedure-Oriented language while Java is Object-Oriented language. Therefore there is no concept of classes defined for C while a java program format includes class as essential part.

Lexical analysis

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer, lexer, or scanner. A lexer often exists as a single function which is called by a parser or another function.

The specification of C and Java include a set of rules which defines the lexer. These rules usually consist of regular expressions, and they define the set of possible character sequences that are used to form individual tokens or lexemes. The ability to express lexical constructs as regular expressions facilitates the description of a lexical analyzer.

In Flex, a regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression **integer** matches the string "integer" wherever it appears and the expression **a57D** looks for the string "a57D". Similarly a regular expression in Flex to scan **comments** in C and Java would be

"**/*(^*|\r\n|(*+(^*/|\r\n)))**+/)|(//(.)***"

Syntax analysis

Syntax Analysis is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given grammar. A grammar is a set of rules (or productions) that specifies the syntax of the language (i.e. what is a valid sentence in the language).

Syntax of mini-Java language can be defined by the following grammar:

```
Goal              = MainClass, { ClassDeclaration }, EOF;
MainClass         = "class", Identifier, "{", "public", "static", "void",
                    "main", "(", "String", "[", "]", Identifier, ")", "{",
                    Statement, "}", "}";
ClassDeclaration  = "class", Identifier, [ "extends", Identifier ], "{", {
                    VarDeclaration }, { MethodDeclaration } "}";
VarDeclaration    = Type, Identifier, ";";
MethodDeclaration = "public", Type, Identifier, "(", [ Type, Identifier, { ",",
                    Type, Identifier }, ], ")", "{", { VarDeclaration }, {
                    Statement }, "return", Expression, ";", "}";
Type              = "int", "[", "]"
                    | "boolean"
                    | "int"
                    | Identifier
                    ;
Statement         = "{", { Statement }, "}"
                    | "if", "(", Expression, ")", Statement, "else",
                    Statement
                    | "while", "(", Expression, ")", Statement
                    | "System.out.println", "(" , Expression, ")", ";"
                    | Identifier, "=", Expression, ";"
                    | Identifier, "[", Expression, "]", "=", Expression, ";"
                    ;
Expression        = Expression , ( "&&" | "<" | "+" | "-" | "*" ) ,
                    Expression
                    | Expression, "[", Expression, "]"
                    | Expression, ".", "length"
```

```
| Expression, ".", Identifier, "(", [ Expression { ",",
Expression } ], ")"
| IntegerLiteral
| "true"
| "false"
| Identifier
| "this"
| "new", "int", "[", Expression, "]"
| "new", Identifier ,"(" ,")"
| "!", Expression
| "(", Expression, ")"
;
```

Identifier is one or more letters, digits, and underscores, starting with a letter

IntegerLiteral is one or more decimal digits

EOF is a distinguished token returned by the scanner at end-of-file

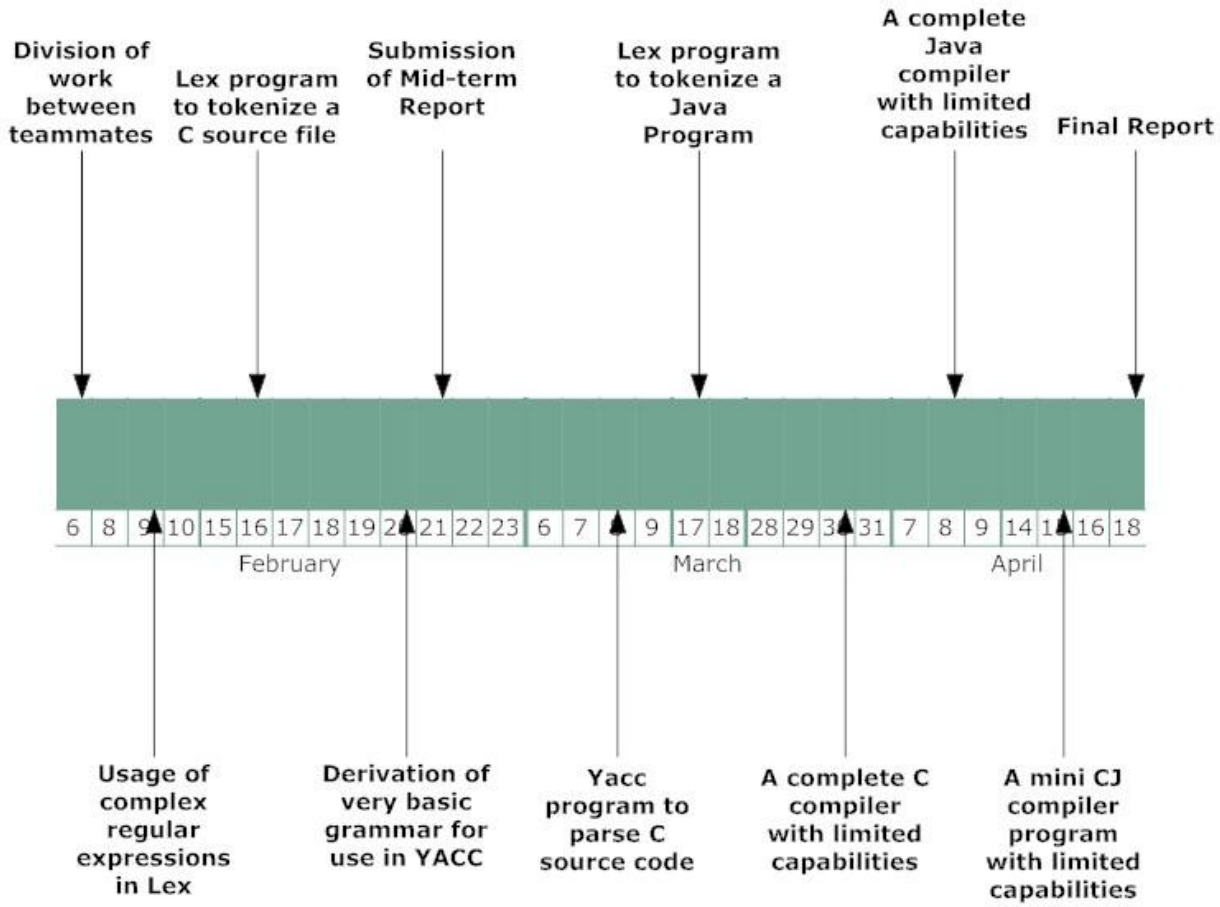
Similarly, syntax of mini-C language can be defined by the following grammar:

```
Function    = Type identifier ( ArgList ) CompoundStmt
ArgList     = Arg
            | ArgList , Arg
Arg         = Type identifier
Declaration = Type IdentList ;
Type        = int
            | float
IdentList   = identifier , IdentList
            |identifier
Stmt        = ForStmt
            | WhileStmt
            | Expr ;
            | IfStmt
            | CompoundStmt
            | Declaration
            | ;
ForStmt     = for ( Expr ; OptExpr ; OptExpr ) Stmt
OptExpr     = Expr
```

```
                    | ε
WhileStmt       =   while ( Expr ) Stmt
IfStmt          =   if ( Expr ) Stmt ElsePart
ElsePart        =   else Stmt
                    | ε
CompoundStmt    =   { StmtList }
StmtList        =   StmtList  Stmt
                    | ε
Expr            =   identifier = Expr
                    | Rvalue
Rvalue          =   Rvalue Compare Mag
                    | Mag
Compare         =   == | < | > | <= | >= | !=
Mag             =   Mag + Term
                    | Mag – Term
                    | Term
Term            =   Term * Factor
                    | Term / Factor
                    | Factor
Factor          =   ( Expr )
                    | - Factor
                    | + Factor
                    | identifier
                    | number
```

Timeline

Timeline for the project is as follows:



COD Project Timeline

References

[1] http://en.wikibooks.org/wiki/Compiler_Construction

[2] http://javacc.java.net/doc/lexertips.html

[3] http://cs.fit.edu/~ryan/cse4251/

[4] http://www.cse.iitd.ernet.in/~nvkrishna/courses/winter07/a2.html

[5] http://www.cs.ucla.edu/~palsberg/course/cs132/project.html

[6] http://cs.ucla.edu/classes/spring11/cs132/kannan/