CMinus - Translator Design

Amzuloiu Andrei-Ciprian, Calcan Marius-Traian

January 2021

Contents

1 Project description

The project involves four main phases: scanning, parsing, semantic analysis.

For **Scanning**, the proposed language is constructed. The parser must detect the tokens of the language and report some lexical errors.

For **Parsing**, a parser for the proposed language has to be constructed. The parser has two main goals:

1. to check whether the input is a syntactically correct program
2. to generate an abstract syntax tree (AST) that records all important information about the program (the intermediate representation)

For **Semantic analysis**, the required tasks were the name analysis method and the type checking method.

The semantic analyzer will traverse the tree constructed in parsing phase to build a table of the symbols contained in the programs and to decorate AST with semantic information.

Name analysis perform several tasks, like: building symbol table, finding bad declarations, multiply declared names and uses of undeclared names, etc. They are used to generate useful error messages in this phase.

The job of the type checker is to determine the type of every expression in the abstract syntax tree and verify that the the type of each expression is appropriate for its context.

In order to implement the language, we used the java programming language with JFlex (for scanning) and Java CUP (for parsing).

2 Theoretical aspects

JFlex is a lexical analyzer generator for Java written in Java. It is also a rewrite of the very useful tool JLex which was developed by Elliot Berk at Princeton University.

The main design goals of JFlex are:

1. Full unicode support
2. Fast generated scanners
3. Fast scanner generation
4. Convenient specification syntax
5. Platform independence
6. JLex compatibility

One of the main design goals of JFlex was to make interfacing with the free Java parser generator CUP as easy as possibly. This has been done by giving the *%cup* directive a special meaning. Since CUP version 0.10j, this has been simplified greatly by the new CUP scanner interface j*ava_cup.runtime.Scanner*. JFlex lexers now implement this interface automatically when then *%cup switch is used*

CUP (Constructor of Useful Parser) is a tool for generating LALR parsers for Java. CUP uses the same role as the UNIX utility YACC. CUP takes an input file that includes specifications of the grammar for which a parser is needed. CUP needs receiving tokens from a scanner. The scanner can be a lexical analyzer generated by JFlex. The generated parser is written in Java.

CUP generates a parser from the input specifications file. The generated parser is composed by two classes: Sym.class, parser.class. Sym class is formed by the terminal simbols declared in the grammar. This class is used by the scanner to refer to symbols. Parser class contains the parser itself to be used.

CUP specification files have four sections which must be written in this order:

1. Package and import specifications
2. User code components.
3. Symbol lists (terminals and non terminals).
4. The grammar

2.1 Symbol lists

Each number represents a symbol:

1. **else**
2. **if**

3. **int**
4. **return**
5. **void**
6. **while**
7. *ID*
8. *NUM*
9. '+'
10. '-'
11. '*'
12. '/'
13. '¡'
14. '¡='
15. '¿'
16. '¿='
17. '=='
18. '!='
19. '='
20. ';'
21. ','
22. '('
23. ')'
24. '['
25. ']'
26. '{'
27. '}'

The items formatted in bold are ***keywords***.

The items formatted in italic are:

- ID - identifiers - consists of a letter, followed by zero or more letters, digits, or underscores ('_'). IDs are case-sensitive.

- NUM - an integer literal (a digit followed by 0 or more digits).

There are the following special symbols in language: ';', '(', ')', '{', '}', '=', '+', '*', '¡', '¿', '¡=', '¿=', '=', '==', '!=', '[', ']', '*/', '/*'

Between those symbols, the comments are surrounded by `/* */`. Comments cannot be placed within tokens. They may not be nested. Also, White space consists of blanks, newlines, and tabs. White space must separate IDs, NUMs, and keywords; otherwise, it is ignored.

2.2 The Grammar

The grammar for C- may be specified as follows:

1. program ::= declaration-list
2. declaration-list ::= declaration-list declaration $|declaration$
3. declaration ::= var-declaration $|fun-declaration$
4. var-declaration ::= type-specifier ID ; $|type-specifierID[NUM]$;
5. type-specifier ::= int $|void$
6. fun-declaration ::= type-specifier ID(params) compound-stmt
7. params ::= param-list $|void$
8. param-list ::= param-list , param $|param$
9. param ::= type-specifier ID $|type-specifierID[]$
10. compound-stmt ::= local-declarations statement-list
11. local-declarations ::= local-declarations var-declaration $|e$
12. statement-list ::= statement-list statement $|e$
13. statement ::= expression-stmt $|compound-stmt|selection-stmt|iteration-stmt|return-stmt$
14. expression-stmt ::= expression ; |;
15. selection-stmt::= if (expression) statement $|if(expression)statementelsestatement$
16. iteration-stmt ::= while (expression) statement
17. return-stmt ::= return ; $|returnexpression$;
18. expression ::= var = expression $|simple-expression$

19. var ::= ID $|ID[expression]$
20. simple-expression ::= additive-expression relop additive-expression $|additiveexpression$
21. relop ::= ¡= | < | > | >= | == |! =
22. additive-expression ::= additive-expression addop term $|term$
23. addop ::= + | −
24. term ::= term mulop factor $|factor$
25. mulop ::= * |/
26. factor ::= (expression) $|var|call|NUM$
27. call ::= ID (args)
28. args ::= arg-list $|e$
29. arg-list ::= arg-list , expression $|express/item$

3 Implementation details

The input file for JFlex is presented in Appendix A.

The input file for CUP is presented in Appendix B.

The project with the source code files can be found on github: Lexical-and-semantic-analyzer-for-C-

In order to implement the syntax tree, we used the *class AbstractSyntaxTreeNode* from the *astClasses* package. This class was extended for non-terminal symbol from the grammar, in order to implement the required verification for them. The abstract class will contain the following:

I *subNodes* - an ArrayList used to store each sub node of the current node from the syntax tree

II *semanticErrors* - an static ArrayList that stores the semantic errors found by parsing the tree.

III *public void addNode(AbstractSyntaxTreeNode node)* - Method used to add a node to the subNodes ArrayList.

IV *public void printTree(int level)* - method used to print the current node and the subnodes based on the current level

V *public abstract String getNodeType())* - method used to get the current node type.

VI *public abstract boolean checkSemantic(Symbol context)* - Method used to parse create the Symbol table and to make the required verification for the semantic analysis.

In order to implement the symbol table, we used the following:

I **ESymbolType** - An Enum responsyble for the symbol type: *Scope, VOIDfunc, VOID, INT, INTarray, INTfunc or Error*

II **Symbol** - class used to describe a symbol from symbol table. Also, this class can be used to store a symbol table. (because I defined a scope as a symbol as well). It contains the following:

i. *name* - The name of the symbol.

ii. *type* - the type of the symbol

iii. *ref* - the reference of the symbol (for example, the scope object, or the function object that contains the symbol).

iv. *symbolTable* - an ArrayList used to store the symbol table related to the current symbol - if the current symbol is a function or a scope object.

v. *argumentsNumber* - The arguments number if the current symbol is a function. The arguments will be stored in the first positions of the symbolTable ArrayList.

vi. *class constructors and getters/setters for the class members*

vii. *public Symbol searchSymbol(String SymbolName)* - method used to search a symbol available at the current scope. If the symbol is found, the symbol is returned. Otherwise, **null** is returned.

viii. *public ESymbolType getCurrentScopeFunctionType()* - method used to get the current return type if the current context is inside of a function.

ix. *public boolean addSymbol(Symbol symbol)* - method used to add a symbol to the current symbol table. If the symbol already exists, it returns false. Otherwise, the symbol is inserted and it returns true.

x. *public boolean checkOperation(String operation, String SymbolName)* - method used to check if the given symbol is compatible to the given operation.

4 Input and testing

In order to test the implementation, we tried several code files in C- with various errors.

Bellow I will present a input file with several errors.

For the following input file:

```
/* A C- program to compute gcd using Euclids Algorithm. */
/* A C- program to compute gcd using Euclids Algorithm. */
int test;

int gcd (int u, int v) {
  if ( v == 0 )
    return u ;
  else
    return gcd (v, u-u/v*v) ;
/* u-u/v*v == u mod v */
}

void gcd(void) {

}

void main(void) {
  int x;
  int y[4];

  {
    int x;
    int w;
  }

  x = main;
  x = 1 + kapa + 2 + 3 + 4;
  x = gcd();
  y = input();

  return main();
  /* output(gcd(x,y));*/
}
```

For this input file, the following errors will be printed:

Error: Multiple declarations for symbol gcd

Error: Multiple declarations for symbol x

Error: main expected to be used as an INT, but it is not an INT variable.

Error: kapa is used, but not declared.

Error: Invalid arguments for function gcd

Error: y expected to be used as an INT, but it is not an INT variable.

Error: Function not found - input

Error: Invalid RETURN value.

5 Bibliography

1. *Principles of Compiler Design by Alfred Aho and Jeffrey Ullman*
2. *Compiler Construction: Principles and Practice by Kenneth Louden*
3. JFlex documentation
4. Java CUP documentation

6 Appendix A – lexer.jflex

```
package jflex;
import java_cup.runtime.ComplexSymbolFactory;
import java_cup.runtime.ComplexSymbolFactory.Location;
import java_cup.runtime.Symbol;
import java.lang.*;
import java.io.InputStreamReader;

%%

%unicode
%class Lexer
%public
%cup
%implements sym

%line
%column

%{
private String lastToken = " ";


/*
Check if two IDs, NUMs, and keywords are delimited with other token than IDs, NU
Otherwise return error with the "merged" ID.
*/
private boolean checkSeparator() {
   if(Character.isDigit(lastToken.charAt(lastToken.length() - 1))
      ||
      Character.isLetter(lastToken.charAt(lastToken.length() - 1))) {
```

```
      System.out.println("ERROR");
      error(lastToken + yytext());
      return false;
   }
   else {
      return true;
   }
}

public Lexer(ComplexSymbolFactory sf, java.io.InputStream is){
        this(is);
        symbolFactory = sf;
}

public Lexer(ComplexSymbolFactory sf, java.io.Reader reader){
        this(reader);
        symbolFactory = sf;
}

private StringBuffer sb;
private ComplexSymbolFactory symbolFactory;
private int csline, cscolumn;

public Symbol symbol(String name, int code) {
    return symbolFactory.newSymbol(name, code,
                                   new Location(yyline + 1, yycolumn + 1, yychar),
                                   new Location(yyline + 1, yycolumn + yylength(),
                                   );
}

public Symbol symbol(String name, int code, Object lexem) {
    return symbolFactory.newSymbol(name, code,
                                   new Location(yyline + 1, yycolumn + 1, yychar),
                                   new Location(yyline + 1, yycolumn + yylength(),
                                   );
}

private void error(String message) {
   System.out.println("Error at line "+ (yyline+1) + ", column " + (yycolumn+ 1)-
}
%}

LineEnd = [\r\n]|\r\n
WhiteSpace = {LineEnd} | [ \t\f]

Comment = "/*" ~"*/"
```

```
Id = [a-zA-Z][a-zA-Z0-9_]*

Num = [0-9]+

%%
<YYINITIAL> {
    /* Arithmetic Operations */
    "-" { return symbol("SUB", SUB);}
    "+" { return symbol("ADD", ADD); }
    "*" { return symbol("MULT", MULT); }
    "/" { return symbol("DIV", DIV); }

    "<" { return symbol("LT", LT); }
    ">" { return symbol("GT", GT); }
    ">=" { return symbol("GTEQ", GTEQ); }
    "<=" { return symbol("LTEQ", LTEQ); }

    "==" { return symbol("EQ", EQ); }
    "!=" { return symbol("NOTEQ", NOTEQ); }

    "=" { return symbol("ASSIGN", ASSIGN); }
    ";" { return symbol("SEMICOL", SEMICOL); }
    "," { return symbol("COMMA", COMMA); }

    /* Keywords */
    "if" {
       if(checkSeparator()){

          return symbol("IF", IF);
       }
    }

    "int"      {
       if(checkSeparator()){

          return symbol("INT", INT);
       }
    }

    "void"     {
       if(checkSeparator()){

          return symbol("VOID", VOID);
       }
    }
```

```
"else"     {
    if(checkSeparator()){

        return symbol("ELSE", ELSE);
    }
}

"while"    {
    if(checkSeparator()){

        return symbol("WHILE", WHILE);
    }
}

"return"   {
    if(checkSeparator()){

        return symbol("RETURN", RETURN);
    }
}

"(" { return symbol("LPAREN", LPAREN); }
")" { return symbol("RPAREN", RPAREN); }
"[" { return symbol("LSQBKT", LSQBKT); }
"]" { return symbol("RSQBKT", RSQBKT); }
"{" { return symbol("LBRKT", LBRKT); }
"}" { return symbol("RBRKTS", RBRKT); }

<<EOF>>                 { return symbol("EOF", EOF); }

{Comment} {/* Ignore */}
{Id} {
       if(checkSeparator()){

              return symbol("ID", ID, yytext());
       }
    }

{Num} {
       if(checkSeparator()){

           return symbol("NUM", NUM, new Integer(Integer.parseInt(yytext())));
       }
    }
```

```
  {WhiteSpace} { lastToken = " ";/* Ignore */ }
}
.|\n { System.out.println("ERROR");error(yytext());}
```

7 Appendix B – parser.cup

```
/* Imported packages */
package jflex;
import astClasses.*;
import java.lang.*;
import java.util.*;
import java_cup.runtime.*;
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;

parser code {:
  protected Lexer lexer;
:}

/* define how to connect to the scanner! */
init with {:
  ComplexSymbolFactory f = new ComplexSymbolFactory();
  symbolFactory = f;
  File file = new File("test.c");
  FileInputStream fis = null;
  try {
    fis = new FileInputStream(file);
  } catch (IOException e) {
    e.printStackTrace();
  }
  lexer = new Lexer(f,fis);
:};
scan with {: return lexer.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal INT          ;
terminal IF           ;
terminal ELSE         ;
terminal VOID         ;
terminal RETURN       ;
terminal WHILE        ;
terminal EQ           ;
terminal NOTEQ        ;
terminal GT           ;
terminal LT           ;
terminal GTEQ         ;
terminal LTEQ         ;
terminal COMMA        ;
```

```
terminal ADD          ;
terminal MULT         ;
terminal DIV          ;
terminal SEMICOL      ;
terminal SUB          ;
terminal ASSIGN       ;
terminal LBRKT        ;
terminal RBRKT        ;
terminal RSQBKT       ;
terminal LSQBKT       ;
terminal RPAREN       ;
terminal LPAREN       ;
terminal String ID    ;
terminal Integer NUM  ;
terminal WHITESPACE   ;

/* Non terminals */
non terminal Program program;
non terminal DeclarationList declaration_list;
non terminal Declaration declaration;
non terminal VarDeclaration var_declaration;
non terminal TypeSpecifier type_specifier;
non terminal FunDeclaration fun_declaration;
non terminal Params params;
non terminal ParamList param_list;
non terminal Param param;
non terminal CompoundStmt compound_stmt;
non terminal LocalDeclarations local_declarations;
non terminal StatementList statement_list;
non terminal Statement statement;
non terminal ExpressionStmt expression_stmt;
non terminal SelectionStmt selection_stmt;
non terminal IterationStmt iteration_stmt;
non terminal ReturnStmt return_stmt;
non terminal Expression expression;
non terminal Var var;
non terminal SimpleExpression simple_expression;
non terminal Relop relop;
non terminal AdditiveExpression additive_expression;
non terminal Addop addop;
non terminal Term term;
non terminal Mulop mulop;
non terminal Factor factor;
non terminal Call call;
non terminal Args args;
non terminal ArgList arg_list;
```

```
/* Precedences */
precedence left LT, GT, LTEQ, GTEQ, EQ, NOTEQ, LPAREN, ID, RPAREN;
precedence left ASSIGN, ADD, SUB, MULT, DIV;
precedence left ELSE;

/* Rules */

start with program;

/* The grammar rules */
program ::= declaration_list:dl {: RESULT = new Program(dl); :}
;

declaration_list ::= declaration_list:dl declaration:d {:
                                    if(dl == null){
                                            RESULT = new DeclarationList(dl, d);
                                    }
                                    else {
                                            RESULT = dl;
                                            RESULT.addDeclaration(d);
                                    }
                            :}
                            | declaration:d {: RESULT = new DeclarationList(d); :}
;
declaration ::= var_declaration:vd {: RESULT = new Declaration(vd); :}
                                    | fun_declaration:fd {: RESULT = new Declaration
;
var_declaration ::= type_specifier:ts ID:i SEMICOL {: RESULT = new VarDeclaratio
:}
                                            | type_specifier:ts ID:i LSQBKT NUM:n RS
                                            | type_specifier:ts ID:i error {: RESULT
:}
                                            | type_specifier:ts ID:i LSQBKT NUM:n RS
                                            | error SEMICOL {: RESULT = new ErrorVa
;
type_specifier ::= INT {: RESULT = new TypeSpecifier("INT"); :}
                                            | VOID {: RESULT = new TypeSpecifier("VO
                                            | error {: RESULT = new ErrorTypeSpecifi
;
fun_declaration ::= type_specifier:ts ID:i LPAREN params:p RPAREN compound_stmt:
                                            | type_specifier:ts ID:i LPAREN error {:
;
params ::= param_list:pl {: RESULT = new Params(pl); :}
                            | VOID {: RESULT = new Params(); :}
;
```

```
param_list ::= param_list:pl COMMA param:p {:
                                        if(pl == null){
                                                RESULT = new ParamList(pl, p);
                                        }
                                        else {
                                                RESULT = pl;
                                                RESULT.addParam(p);
                                        }
                                :}
                                | param:p {: RESULT = new ParamList(p); :}
;
param ::= type_specifier:ts ID:i {: RESULT = new Param(ts, i, false); :}
                        | type_specifier:ts ID:i LSQBKT RSQBKT {: RESULT = new P
;
compound_stmt ::= LBRKT local_declarations:ld statement_list:sl RBRKT {: RESULT =
                                | error local_declarations:ld statement_list:sl
                                | LBRKT local_declarations:ld statement_list:sl
;
local_declarations ::= local_declarations:ld var_declaration:vd {:
                                        if(ld == null) {
                                                RESULT = new LocalDeclar
                                        }
                                        else {
                                                RESULT = ld;
                                                RESULT.addDeclaration(vd
                                        }
                                        :}
                                        | {: RESULT = new LocalDeclarati
;
statement_list ::= statement_list:sl statement:s {:
                                        if(sl == null){
                                                RESULT = new StatementLi
                                        }
                                        else {
                                                RESULT = sl;
                                                RESULT.addStatement(s);
                                        }
                                :}
                                | {: RESULT = new StatementList(); :}
;
statement ::= expression_stmt:es {: RESULT = new Statement(es); :}
                                | compound_stmt:cs {: RESULT = new Statement(cs)
                                | selection_stmt:ss {: RESULT = new Statement(ss
                                | iteration_stmt:is {: RESULT = new Statement(is
                                | return_stmt:rs {: RESULT = new Statement(rs);
;
```

```
expression_stmt ::= expression:e SEMICOL {: RESULT = new ExpressionStmt(e); :}
                                    | SEMICOL {: RESULT = new ExpressionStmt(); :}
                                    | error SEMICOL {: RESULT = new ErrorExpressionSt
;
selection_stmt ::= IF LPAREN expression:e RPAREN statement:s {: RESULT = new Sel
                                                | IF LPAREN expression:e RPAREN statemen
{: RESULT = new SelectionStmt(e,s1,s2);  :}
                                                | IF LPAREN expression:e RPAREN error {:
;
iteration_stmt ::= WHILE LPAREN expression:e RPAREN statement:s {:RESULT = new I
                                                | WHILE LPAREN expression:e RPAREN error
;
return_stmt ::= RETURN SEMICOL {: RESULT = new ReturnStmt(); :}
                                    | RETURN expression:e SEMICOL {: RESULT = new Re
                                    | RETURN error SEMICOL {: RESULT = new ErrorRetu
                                    | RETURN error {: RESULT = new ErrorReturnStmt("
                                    | RETURN expression:e error {: RESULT = new Retu
;
expression ::= var:v ASSIGN expression:e {: RESULT = new Expression(v,e);:}
                                    | simple_expression:se {:RESULT = new Expression
;
var ::= ID:i {: RESULT = new Var(i);:}
                | ID:i LSQBKT expression:e RSQBKT {:RESULT = new Var(e, i); :}
;
simple_expression ::= additive_expression:ae1 relop:r additive_expression:ae2 {:
                                                    | additive_expression:ae {:RESUL
;
relop ::= LTEQ {: RESULT = new Relop("LTEQ"); :}
                        | LT {: RESULT = new Relop("LT"); :}
                        | GT {: RESULT = new Relop("GT"); :}
                        | GTEQ {: RESULT = new Relop("GTEQ"); :}
                        | EQ {: RESULT = new Relop("EQ"); :}
                        | NOTEQ{: RESULT = new Relop("NOTEQ"); :}
;
additive_expression ::= additive_expression:ae addop:a term:t {:
                                    if(ae == null) {
                                            RESULT = new AdditiveExpression(
                                    }
                                    else {
                                            RESULT = ae;
                                            RESULT.addOperation(a, t);
                                    }
                                    :}
                                    | term:t {:RESULT = new AdditiveExpressi
;
addop ::= ADD {: RESULT = new Addop("ADD");:}
```

```
                        | SUB {: RESULT = new Addop("SUB");:}
;
term ::= term:t mulop:m factor:f {:
                                if(t == null) {
                                        RESULT = new Term(t,m,f);
                                }
                                else {
                                        RESULT = t;
                                        RESULT.addOperator(m, f);
                                }
                        :}
                        | factor:f {:RESULT = new Term(f); :}
;
mulop ::= MULT {: RESULT = new Mulop("MULT");:}
                        | DIV {: RESULT = new Mulop("DIV");:}
;
factor ::= LPAREN expression:e RPAREN {: RESULT = new Factor(e); :}
                        | var:v {: RESULT = new Factor(v); :}
                        | call:c {: RESULT = new Factor(c); :}
                        | NUM:n {: RESULT = new Factor(n); :}
;
call ::= ID:i LPAREN args:a RPAREN {: RESULT = new Call(a, i); :}
                        | ID:i LPAREN error {: RESULT = new ErrorCall("Invalid C
;
args ::= arg_list:al {: RESULT = new Args(al); :}
                | {: RESULT = new Args(); :}
;
arg_list ::= arg_list:al COMMA expression:e {:
                                if(al == null) {
                                        RESULT = new ArgList(al, e);
                                }
                                else {
                                        RESULT = al;
                                        RESULT.addArg(e);
                                }
                        :}
                        | expression:e {: RESULT = new ArgList(e); :}
;
```