International Institute of Information Technology, Hyderabad

Compilers Project

Compiler for Decaf

Mohit Sharma

201505508

Objective:

Our objective for phase 1 & 2 is to implement the syntax analyzer and parser for the Decaf programming Language. Analyzer should be able to parse all valid programs and give an error for invalid program. Check Decaf manual to get an idea about the language.

Tools used:

1. **Flex:** Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers").
   In simple words we can consider it as a tool which provides an easy way to specify the regular expressions corresponding to lexemes expected in source file. It also provides us a way to specify the action to be taken when we encounter a particular lexeme. In action part we can do operations like printing the lexeme (text matched) or returning corresponding token to the parser program. Check the following example. Here we have defined the regex for "if" and returning corresponding token in action part.

```
%{
      /* Global declarations and control information*/
 %}

 IF    "if"
 %%
      {IF}  { foutlex<<"IF"<<"\n";yylval.sval = strdup(yytext); return If; }
 %%
```

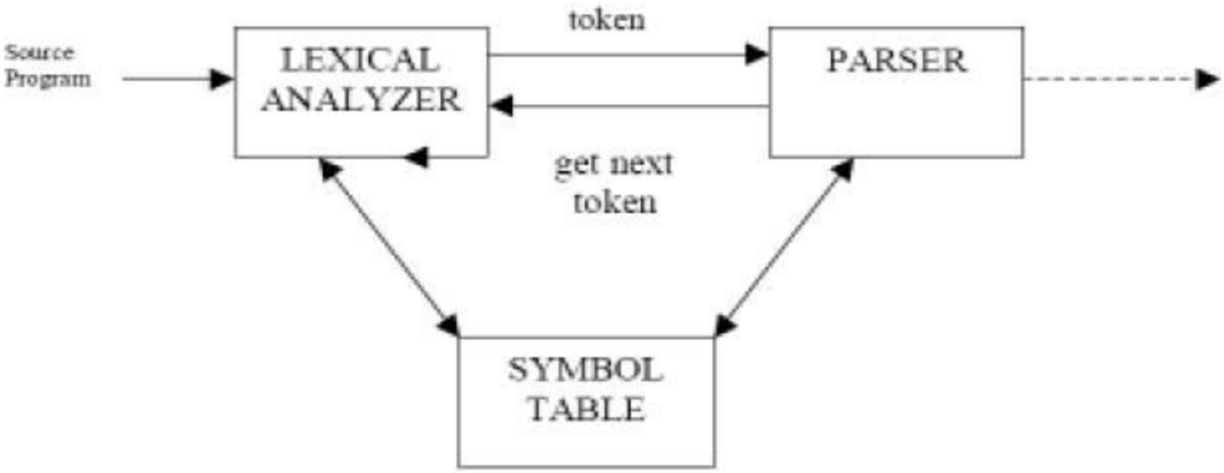
   The token we are returning here are specified in the parser file (bison file) explained below. We pass this .l file to flex to generate corresponding C code for lexical analyzer which can scan any text for the lexemes specified in .l file. The output file from flex is named "lex.yy.c".

2. **Bison:** It is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser (either in C, C++, or Java) which reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. Bison by default generates LALR parsers but can also create GLR parsers.
   We specify the grammar and tokens that are needed for that grammar in a .y file. We provide this .y file to bison as input and it generates two files ".tab.h" (to be included in flex code, it contains all the terminal token declarations that flex will return when it encounters the lexeme corresponding to that token) and ".tab.c" (contains the parser code). To understand how to specify the grammar specifications, consider following dummy grammar which contains just one string "x y".

```
%{
      /* Global declarations and control information */
%}

// Union for data types of tokens that can be returned by flex
%union {
      int ival;
      char *sval;
 }

 // Different tokens that can be returned by flex
 %token <sval> X Y

 %%
      // Grammar that'll be parsed by bison
      Program
            :     X Y   {fout <<"PROGRAM ENCOUNTERED\n"; };
%%
```

In this grammar we are expecting just two lexemes or terminals: “x” and “y”. Both are specified as tokens. The same are used in the only production rule. The parser generated will work in bottom up manner i.e. it’ll first try to match X then Y, if it is able to complete the rule “X Y”, reduction will happen and matched rule will be replaced by corresponding LHS i.e. “Program” in this case. The moment it is able to reduce to start symbol, parser stops successfully. Whenever it reduces, the corresponding statements specified in the action part will be executed. If any unexpected symbol is encountered it calls the error handler “yyerror()”.

Check complete codes for flex and bison at the end.



Basic diagram to understand the interaction between lexical analyzer and parser.

Issues faced:

1. **Shift/reduce warnings:** These warnings can occur whenever grammar contains productions which can be reduced to a non-terminal and also one more symbol can be shifted instead of reducing it. If we don’t specify the precedence and associativity of operators then there will be many shift/reduce warnings. The token specified earlier has lower precedence than that of specified later. For specifying associativity , we can use %left, %right and %noassoc. E.g. following three tokens are left associative.

   ```
   %left Multiplication Division Remainder
   ```

2. **Same token to be used in two different contexts (binary/unary minus):** Both the minus are represented by ‘-’ but we can’t return two different tokens from flex corresponding to same string ‘-’. We need a way to use same token as per context. It can be done by specifying a dummy token for Unary minus just to specify that it has higher precedence than binary minus. Same is specified in the production using %prec. Check following example for better understanding:

   ```
   …
   %left '+' '-'
   %left '*'
   %left UMINUS
   ```

Now the precedence of UMINUS can be used in specific rules:

```
exp:
        …
        | exp '-' exp
        …
        | '-' exp %prec UMINUS
```

Phase 3:

Our objective for phase 3 is to build the Abstract Syntax Tree (AST) for the parsed program. Then to traverse the tree to produce nested XML output which represents the structure of the program.

Input:

```
        class Program
        {
             boolean field;
             int x[10];
             int foo(int y)
             {
             }
        }
```

Output:

```
<program>
  <field_declarations count=“2”>
       Normal
       <declaration name=”field” type=”boolean”/>
       Array
       <declaration name=”x” count=“10” type=”integer”/>
  </field_declarations>
  <methods count=”1”>
       <method name=”foo” return_type=”int”>
            <params>
                 <param name=”y” type=”int”>
            </params>
            <locals>
            </locals>
            <body>
            </body>
       </method>
  </methods>
</program>
```

Steps:

1. Create a base class Node.
2. Create a class corresponding to every non-terminal symbol in grammar. All these classes will inherit Node class. Every class contains a Node* for each of its children.
3. In bison file create a node corresponding to every production rule and assign it to $$ so that it can be passed to its parent rule. E.g. :

```
Identifier1
        :    Identifier
             { $$ = new Identifier1Node(new IdentifierNode($1), NULL); }
        |    Identifier OpenSquare Int_literal CloseSquare
             { $$ = new Identifier1Node(new IdentifierNode($1), $3); };
```

   Check source code at the end for detailed actions.

4. Now follow visitor pattern to display output corresponding to every node. It’ll help to avoid type castings to be done manually from Node* to any of its child classes. Type casting is required so that we can map correct display() method to each node. The task is can be done easily using visitor patterns. It utilizes run time binding using virtual functions and function overloading technique.
5. Implement the functionality to display appropriate output in corresponding visit methods.

References:

1. https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator)
2. https://en.wikipedia.org/wiki/GNU_bison
3. http://www.gnu.org/software/bison/manual/html_node/Contextual-Precedence.html
4. http://ashimg.tripod.com/Parser.html
5. http://aquamentus.com/flex_bison.html
6. https://en.wikipedia.org/wiki/Visitor_pattern

Codes(Phase 1 & 2)

```
%{
/* Lex code to implement a parser for a decaf grammar */

#define YY_DECL extern "C" int yylex()
#include<fstream>
#include <iostream>
using namespace std;

#include "project.tab.h"    // to get the token types that we return
fstream foutlex("flex_output.txt",ios::out);
%}


COMMENT                    "//".*
CLASS                      "class"
PROGRAM_STRING             "Program"
IDENTIFIER                 [_a-zA-Z][_a-zA-Z0-9]*
OPEN_BRACKET               "{"
CLOSE_BRACKET              "}"
OPEN_SQUARE                "["
CLOSE_SQUARE               "]"
SEMICOLON                  ";"
OPEN_PAREN                 "("
CLOSE_PAREN                ")"
COMMA                      ","
VOID                       "void"
BREAK                      "break"
CALLOUT                    "callout"
CONTINUE                   "continue"
ELSE                       "else"
RETURN                     "return"
EQUAL                      "="
PLUS_EQUAL                 "+="
MINUS_EQUAL                "-="
FOR                        "for"
IF                         "if"
INT                        "int"
BOOLEAN                    "boolean"
DECIMAL_LITERAL            [-]?[0-9]+
HEX_LITERAL                0[xX][0-9a-fA-F]+
CHAR_LITERAL     "\'"([^\"\'\\]|"\\\""|"\\\'"|"\\\\"|"\\t"|"\\n")"\'"
STRING_LITERAL             \".*\"
FALSE                      "false"
TRUE                       "true"
CONDITIONAL_OR             "||"
CONDITIONAL_AND            "&&"
EQUAL_TO                   "=="
NOT_EQUAL_TO               "!="
LESS_THAN                  "<"
LESS_THAN_EQUAL            "<="
GREATER_THAN_EQUAL         ">="
GREATER_THAN               ">"
ADDITION                   "+"
MINUS                      "-"
MULTIPLICATION             "*"
DIVISION                   "/"
REMAINDER                  "%"
```

```
NEGATION                    "!"

%%
[ \t\n]                     ;
{COMMENT}                   ;

{CLASS}                     { foutlex<<"CLASS"<<"\n";
      yylval.sval = strdup(yytext); return Class; }
{PROGRAM_STRING}            { foutlex<<"PROGRAM"<<"\n";
      yylval.sval = strdup(yytext); return ProgramString; }
{VOID}                      { foutlex<<"VOID"<<"\n";
      yylval.sval = strdup(yytext); return Void; }
{BREAK}                     { foutlex<<"BREAK"<<"\n";
      yylval.sval = strdup(yytext); return Break; }
{CALLOUT}                   { foutlex<<"CALLOUT"<<"\n";
      yylval.sval = strdup(yytext); return Callout; }
{CONTINUE}                  { foutlex<<"CONTINUE"<<"\n";
      yylval.sval = strdup(yytext); return Continue; }
{ELSE}                      { foutlex<<"ELSE"<<"\n";
      yylval.sval = strdup(yytext); return Else; }
{RETURN}                    { foutlex<<"RETURN"<<"\n";
      yylval.sval = strdup(yytext); return Return; }
{FOR}                       { foutlex<<"FOR"<<"\n";
      yylval.sval = strdup(yytext); return For; }
{IF}                        { foutlex<<"IF"<<"\n";
           yylval.sval = strdup(yytext); return If; }
{INT}                       { foutlex<<"INT_DECLARATION"<<"\n";
      yylval.sval = strdup(yytext); return Int; }
{BOOLEAN}                   { foutlex<<"BOOLEAN_DECLARATION"<<"\n";
      yylval.sval = strdup(yytext); return Boolean; }
{FALSE}                     { foutlex<<"BOOLEAN:false"<<"\n";
      yylval.sval = strdup(yytext); return False; }
{TRUE}                      { foutlex<<"BOOLEAN:true"<<"\n";
      yylval.sval = strdup(yytext); return True; }
{MINUS}                     { foutlex<<"-"<<"\n";
           yylval.sval = strdup(yytext); return Minus; }
{IDENTIFIER}                { foutlex<<"ID:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return Identifier; }
{OPEN_BRACKET}              { yylval.sval = strdup(yytext); return
OpenBracket; }
{CLOSE_BRACKET}             { yylval.sval = strdup(yytext); return
CloseBracket; }
{OPEN_SQUARE}               { yylval.sval = strdup(yytext); return
OpenSquare; }
{CLOSE_SQUARE}              { yylval.sval = strdup(yytext); return
CloseSquare; }
{SEMICOLON}                 { yylval.sval = strdup(yytext); return
SemiColon; }
{OPEN_PAREN}                { yylval.sval = strdup(yytext); return
OpenParen; }
{CLOSE_PAREN}               { yylval.sval = strdup(yytext); return
CloseParen; }
{COMMA}                     { yylval.sval = strdup(yytext); return
Comma; }
{EQUAL}                     { foutlex<<"="<<"\n";
           yylval.sval = strdup(yytext); return Equal; }
{PLUS_EQUAL}                { foutlex<<"+="<<"\n";
           yylval.sval = strdup(yytext); return PlusEqual; }
```

```
{MINUS_EQUAL}               { foutlex<<"-="<<"\n";
           yylval.sval = strdup(yytext); return MinusEqual; }
{DECIMAL_LITERAL}           { foutlex<<"INT:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return Decimal_literal; }
{HEX_LITERAL}               { foutlex<<"INT:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return Hex_literal; }
{CHAR_LITERAL}              { foutlex<<"CHARACTER:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return Char_literal; }
{STRING_LITERAL}            { foutlex<<"STRING:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return String_literal; }
{CONDITIONAL_OR}            { yylval.sval = strdup(yytext); return
ConditionalOr; }
{CONDITIONAL_AND}           { yylval.sval = strdup(yytext); return
ConditionalAnd; }
{EQUAL_TO}                  { yylval.sval = strdup(yytext); return
EqualTo; }
{NOT_EQUAL_TO}              { yylval.sval = strdup(yytext); return
NotEqualTo; }
{LESS_THAN}                 { yylval.sval = strdup(yytext); return
LessThan; }
{LESS_THAN_EQUAL}           { yylval.sval = strdup(yytext); return
LessThanEqual; }
{GREATER_THAN_EQUAL}  { yylval.sval = strdup(yytext); return
GreaterThanEqual; }
{GREATER_THAN}              { yylval.sval = strdup(yytext); return
GreaterThan; }
{ADDITION}                  { yylval.sval = strdup(yytext); return
Addition; }
{MULTIPLICATION}            { yylval.sval = strdup(yytext); return
Multiplication; }
{DIVISION}                  { yylval.sval = strdup(yytext); return
Division; }
{REMAINDER}                 { yylval.sval = strdup(yytext); return
Remainder; }
{NEGATION}                  { yylval.sval = strdup(yytext); return
Negation; }
%%
```

```
%{
// Bison code to parse decaf
#include <cstdio>
#include<string.h>
#include<fstream>
#include <iostream>
using namespace std;

// stuff from flex that bison needs to know about:
extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;
void yyerror(const char *s);
extern "C" fstream foutlex;
fstream fout("bison_output.txt",ios::out);
%}
// Union for data types of tokens that can be returned by flex
%union {
      int ival;
      char *sval;
}

// Different tokens that can be returned by flex
%token <sval> Class ProgramString Identifier OpenBracket CloseBracket
OpenSquare CloseSquare SemiColon OpenParen CloseParen Comma Void
Break Callout Continue Else Return Equal PlusEqual MinusEqual For If
Int Boolean Decimal_literal Hex_literal Char_literal String_literal
True False
%left ConditionalOr
%left ConditionalAnd
%left EqualTo NotEqualTo
%left LessThan LessThanEqual GreaterThanEqual GreaterThan
%left Addition Minus
%left Multiplication Division Remainder
%left Negation
%left UMinus

%%
// Grammar that'll be parsed by bison
Program
      :     Class ProgramString OpenBracket field_decls method_decls
CloseBracket     {fout <<"PROGRAM ENCOUNTERED\n"; };

field_decls
      :     field_decls field_decl

      |;


field_decl
      :     Type Identifiers SemiColon;

Identifiers
      :     Identifier1
      |     Identifiers Comma Identifier1;

Identifier1
```

```
      :    Identifier
                                            { fout<<"ID="<<$1<<"\n";
}
      |    Identifier OpenSquare Decimal_literal CloseSquare
                      { fout<<"ID="<<$1<<"\n"<<"SIZE="<<$3<<"\n"; }
      |    Identifier OpenSquare Hex_literal CloseSquare
                            {
fout<<"ID="<<$1<<"\n"<<"SIZE="<<$3<<"\n"; };
method_decls
      :    method_decl method_decls

      |;

method_decl
      :    Type Identifier OpenParen Params CloseParen Block
                      { fout<<"METHOD="<<$2<<"\n"; }
      |    Type Identifier OpenParen CloseParen Block
                            { fout<<"METHOD="<<$2<<"\n"; }
      |    Void Identifier OpenParen Params CloseParen Block
                      { fout<<"METHOD="<<$2<<"\n"; }
      |    Void Identifier OpenParen CloseParen Block
                            { fout<<"METHOD="<<$2<<"\n"; };

Params
      :    Type Identifier

      |    Type Identifier Comma Params;

Block
      :    OpenBracket var_decls Statements CloseBracket;

var_decls
      :    var_decl var_decls

      |;

var_decl
      :    Type decls SemiColon;

decls
      :    Identifier
                                       { fout<<"ID="<<$1<<"\n"; }
      |    Identifier Comma decls
                                       { fout<<"ID="<<$1<<"\n"; };

Statements
      :    Statement Statements

      |;


Statement
      :    Location Assign_op Expr SemiColon
                                 { fout<<"ASSIGNMENT OPERATION
ENCOUNTERED\n"; }
      |    Method_call SemiColon
      |    If OpenParen Expr CloseParen Block
                                 { fout<<"IF ENCOUNTERED\n"; }
      |    If OpenParen Expr CloseParen Block Else Block
                            { fout<<"IF ENCOUNTERED\n"; }
      |    For Identifier Equal Expr Comma Expr Block
                            { fout<<"FOR ENCOUNTERED\n"; }
      |    Return SemiColon
                                 { fout<<"RETURN ENCOUNTERED\n"; }
      |    Return Expr SemiColon
                                 { fout<<"RETURN ENCOUNTERED\n"; }

      |    Break SemiColon
                                       { fout<<"BREAK ENCOUNTERED\n";
}
      |    Continue SemiColon
                                       { fout<<"CONTINUE
ENCOUNTERED\n"; }
      |    Block;

Method_call
      :    Method_name OpenParen PassParams CloseParen

      |    Method_name OpenParen CloseParen
      |    Callout    OpenParen String_literal CloseParen
                                 { fout<<"CALLOUT TO "<<$3<<"
ENCOUNTERED\n"; }
      |    Callout    OpenParen String_literal Comma CalloutArgs
CloseParen                  { fout<<"CALLOUT TO "<<$3<<"
ENCOUNTERED\n"; };

PassParams
      :    Expr
      |    Expr Comma PassParams;

Method_name
      :    Identifier
                                       { fout<<"METHOD CALL="<<$1; };

Location
      :    Identifier
                                       { fout<<"LOCATION
ENCOUNTERED="<<$1<<"\n"; }
      |    Identifier OpenSquare Expr CloseSquare;

CalloutArgs
      :    Expr
      |    String_literal
      |    Expr Comma CalloutArgs
      |    String_literal Comma CalloutArgs;

Expr
      :    Location
      |    Method_call
      |    Literal
      |    Expr ConditionalOr Expr
                                       { fout<<"CONDITIONAL OR
ENCOUNTERED\n"; }
      |    Expr ConditionalAnd Expr
                                 { fout<<"CONDITIONAL AND
ENCOUNTERED\n"; }
```

```
      |     Expr EqualTo Expr
                                       { fout<<"EQUAL TO
ENCOUNTERED\n"; }
      |     Expr NotEqualTo Expr
                                 { fout<<"NOT EQUAL TO
ENCOUNTERED\n"; }
      |     Expr LessThan Expr
                                       { fout<<"LESS THAN
ENCOUNTERED\n"; }
      |     Expr LessThanEqual Expr
                                       { fout<<"LESS THAN EQUAL
ENCOUNTERED\n"; }
      |     Expr GreaterThanEqual Expr
                                 { fout<<"GREATER THAN EQUAL
ENCOUNTERED\n"; }
      |     Expr GreaterThan Expr
                                 { fout<<"GREATER THAN
ENCOUNTERED\n"; }
      |     Expr Addition Expr
                                       { fout<<"ADDITION
ENCOUNTERED\n"; }
      |     Expr Minus Expr
                                       { fout<<"SUBTRACTION
ENCOUNTERED\n"; }
      |     Expr Multiplication Expr
                                 { fout<<"MULTIPLICATION
ENCOUNTERED\n"; }
      |     Expr Division Expr
                                       { fout<<"DIVIDION
ENCOUNTERED\n"; }
      |     Expr Remainder Expr
                                       { fout<<"MOD ENCOUNTERED\n"; }
      |     Minus Expr %prec UMinus
                                       { fout<<"UNARY MINUS
ENCOUNTERED\n"; }
      |     Negation Expr
                                       { fout<<"NEGATION
ENCOUNTERED\n"; }
      |     OpenParen Expr CloseParen;

Literal
      :     Int_literal
      |     Char_literal
                                       { fout<<"CHAR
ENCOUNTERED="<<$1<<"\n"; }
      |     Bool_literal;

Bool_literal
      :     True
                                       { fout<<"BOOLEAN
ENCOUNTERED="<<$1<<"\n"; }
      |     False
                                       { fout<<"BOOLEAN
ENCOUNTERED="<<$1<<"\n"; };

Int_literal
```

```
      :    Decimal_literal
                                        { fout<<"INT
ENCOUNTERED="<<$1<<"\n"; }
      |    Hex_literal
                                             { fout<<"INT
ENCOUNTERED="<<$1<<"\n"; };

Type
      :    Int
                                             { fout<<"INT DECLARATION
ENCOUNTERED\n"; }
      |    Boolean
                                             { fout<<"BOOLEAN
DECLARATION ENCOUNTERED\n"; };

Assign_op
      :    Equal
                                             { fout<<"ASSIGNMENT
ENCOUNTERED\n"; }
      |    PlusEqual
                                        { fout<<"ADDITION ASSIGNMENT
ENCOUNTERED\n"; }
      |    MinusEqual
                                        { fout<<"SUBTRACTION
ASSIGNMENT ENCOUNTERED\n"; };
%%

int main(int argc, char** argv)
{
      FILE *myfile = fopen(argv[1], "r");
      if (!myfile)
      {
            cout << "I can't open input file!" << endl;
            return -1;
      }
      yyin = myfile;
      do
      {
            yyparse();
      } while (!feof(yyin));
      cout<<"Success\n";
      fout.close();
      foutlex.close();
      return 0;
}

void yyerror(const char *s) {
      cout<<"Syntax error\n";
      fout.close();
      exit(-1);
}
```

```
# Use this file to run above codes
# Input: “test_input” containing decaf code
# Output: flex_output.txt, bison_output.txt

#!/bin/bash

bison -dv Project.y
flex Project.l
g++ Project.tab.c lex.yy.c -lfl -o proj
./proj test_input
```

Codes(Phase 3)

```
%{
/* Lex code to implement a parser for a decaf grammar */

#define YY_DECL extern "C" int yylex()
#include<fstream>
#include <iostream>
using namespace std;
#include "decaf.h"
#include "decaf.tab.h"  // to get the token types that we return
fstream foutlex("flex_output.txt",ios::out);
%}


COMMENT                          "//".*
CLASS                      "class"
PROGRAM_STRING             "Program"
IDENTIFIER                 [_a-zA-Z][_a-zA-Z0-9]*
OPEN_BRACKET               "{"
CLOSE_BRACKET              "}"
OPEN_SQUARE                "["
CLOSE_SQUARE               "]"
SEMICOLON                  ";"
OPEN_PAREN                       "("
CLOSE_PAREN                ")"
COMMA                            ","
VOID                       "void"
BREAK                      "break"
CALLOUT                          "callout"
CONTINUE                   "continue"
ELSE                       "else"
RETURN                           "return"
EQUAL                            "="
PLUS_EQUAL                       "+="
MINUS_EQUAL                "-="
FOR                              "for"
IF                               "if"
INT                              "int"
BOOLEAN                          "boolean"
DECIMAL_LITERAL       [-]?[0-9]+
HEX_LITERAL                0[xX][0-9a-fA-F]+
CHAR_LITERAL
      "\'"([^\"\'\\]|"\\\""|"\\\'"|"\\\\"|"\\t"|"\\n")"\'"
STRING_LITERAL             \"[^\"]*\"
FALSE                      "false"
TRUE                       "true"
CONDITIONAL_OR             "||"
CONDITIONAL_AND            "&&"
EQUAL_TO                   "=="
NOT_EQUAL_TO               "!="
LESS_THAN                  "<"
LESS_THAN_EQUAL       "<="
GREATER_THAN_EQUAL         ">="
GREATER_THAN               ">"
ADDITION                   "+"
MINUS                      "-"
MULTIPLICATION             "*"
DIVISION                   "/"
```

```
REMAINDER                  "%"
NEGATION                   "!"

%%
[ \t\n]                    ;
{COMMENT}                  ;

{CLASS}                         { foutlex<<"CLASS"<<"\n";
           yylval.sval = strdup(yytext); return Class; }
{PROGRAM_STRING}           { foutlex<<"PROGRAM"<<"\n";
      yylval.sval = strdup(yytext); return ProgramString; }
{VOID}                          { foutlex<<"VOID"<<"\n";
           yylval.sval = strdup(yytext); return Void; }
{BREAK}                         { foutlex<<"BREAK"<<"\n";
           yylval.sval = strdup(yytext); return Break; }
{CALLOUT}                  { foutlex<<"CALLOUT"<<"\n";
      yylval.sval = strdup(yytext); return Callout; }
{CONTINUE}                 { foutlex<<"CONTINUE"<<"\n";
      yylval.sval = strdup(yytext); return Continue; }
{ELSE}                          { foutlex<<"ELSE"<<"\n";
           yylval.sval = strdup(yytext); return Else; }
{RETURN}                   { foutlex<<"RETURN"<<"\n";
      yylval.sval = strdup(yytext); return Return; }
{FOR}                      { foutlex<<"FOR"<<"\n";
      yylval.sval = strdup(yytext); return For; }
{IF}                       { foutlex<<"IF"<<"\n";
           yylval.sval = strdup(yytext); return If; }
{INT}                      { foutlex<<"INT_DECLARATION"<<"\n";
      yylval.sval = strdup(yytext); return Int; }
{BOOLEAN}                  { foutlex<<"BOOLEAN_DECLARATION"<<"\n";
      yylval.sval = strdup(yytext); return Boolean; }
{FALSE}                         { foutlex<<"BOOLEAN:false"<<"\n";
           yylval.sval = strdup(yytext); return False; }
{TRUE}                          { foutlex<<"BOOLEAN:true"<<"\n";
           yylval.sval = strdup(yytext); return True; }
{MINUS}                         { foutlex<<"-"<<"\n";
                yylval.sval = strdup(yytext); return Minus; }
{IDENTIFIER}               { foutlex<<"ID:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return Identifier; }
{OPEN_BRACKET}             { yylval.sval = strdup(yytext); return
OpenBracket; }
{CLOSE_BRACKET}       { yylval.sval = strdup(yytext); return
CloseBracket; }
{OPEN_SQUARE}              { yylval.sval = strdup(yytext); return
OpenSquare; }
{CLOSE_SQUARE}             { yylval.sval = strdup(yytext); return
CloseSquare; }
{SEMICOLON}                { yylval.sval = strdup(yytext); return
SemiColon; }
{OPEN_PAREN}               { yylval.sval = strdup(yytext); return
OpenParen; }
{CLOSE_PAREN}              { yylval.sval = strdup(yytext); return
CloseParen; }
{COMMA}                    { yylval.sval = strdup(yytext); return
Comma; }
{EQUAL}                    { foutlex<<"="<<"\n";
           yylval.sval = strdup(yytext); return Equal; }
```

```
{PLUS_EQUAL}                { foutlex<<"+="<<"\n";
           yylval.sval = strdup(yytext); return PlusEqual; }
{MINUS_EQUAL}               { foutlex<<"-="<<"\n";
           yylval.sval = strdup(yytext); return MinusEqual; }
{DECIMAL_LITERAL}           { foutlex<<"INT:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return Decimal_literal; }
{HEX_LITERAL}               { foutlex<<"INT:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return Hex_literal; }
{CHAR_LITERAL}              { foutlex<<"CHARACTER:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return Char_literal; }
{STRING_LITERAL}      { foutlex<<"STRING:"<<yytext<<"\n";
      yylval.sval = strdup(yytext); return String_literal; }
{CONDITIONAL_OR}      { yylval.sval = strdup(yytext); return
ConditionalOr; }
{CONDITIONAL_AND}           { yylval.sval = strdup(yytext); return
ConditionalAnd; }
{EQUAL_TO}                        { yylval.sval = strdup(yytext);
return EqualTo; }
{NOT_EQUAL_TO}              { yylval.sval = strdup(yytext); return
NotEqualTo; }
{LESS_THAN}                 { yylval.sval = strdup(yytext); return
LessThan; }
{LESS_THAN_EQUAL}           { yylval.sval = strdup(yytext); return
LessThanEqual; }
{GREATER_THAN_EQUAL}  { yylval.sval = strdup(yytext); return
GreaterThanEqual; }
{GREATER_THAN}              { yylval.sval = strdup(yytext); return
GreaterThan; }
{ADDITION}                        { yylval.sval = strdup(yytext);
return Addition; }
{MULTIPLICATION}            { yylval.sval = strdup(yytext); return
Multiplication; }
{DIVISION}                        { yylval.sval = strdup(yytext);
return Division; }
{REMAINDER}                       { yylval.sval = strdup(yytext);
return Remainder; }
{NEGATION}                  { yylval.sval = strdup(yytext); return
Negation; }
%%
```

```
%{
// Bison code to parse decaf
#include "decaf.h"
#include <cstdio>
#include<string.h>
#include<fstream>
#include <iostream>
using namespace std;

// stuff from flex that bison needs to know about:
extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;
string output = "";
string localType = "";
int declarationCount = 0;
int methodCount = 0;
void yyerror(const char *s);
fstream fout("XML_visitor.txt",ios::out);
%}
// Union for data types of tokens that can be returned by flex
%union {
     Node *astNode;
     int ival;
     char *sval;
}

// Different tokens that can be returned by flex
%token <sval> Class ProgramString Identifier OpenBracket CloseBracket
OpenSquare CloseSquare SemiColon OpenParen CloseParen Comma Void
Break Callout Continue Else Return Equal PlusEqual MinusEqual For If
Int Boolean Decimal_literal Hex_literal Char_literal String_literal
True False
%left ConditionalOr
%left ConditionalAnd
%left EqualTo NotEqualTo
%left LessThan LessThanEqual GreaterThanEqual GreaterThan
%left Addition Minus
%left Multiplication Division Remainder
%left Negation
%left UMinus
%type <sval> ConditionalOr ConditionalAnd EqualTo NotEqualTo LessThan
LessThanEqual GreaterThanEqual GreaterThan Addition Minus
Multiplication Division Remainder Negation
%type <astNode> Program field_decls field_decl Identifiers
Identifier1 method_decls method_decl Params Block var_decls var_decl
decls Statements Statement Method_call PassParams Method_name
Location CalloutArgs Expr Literal Bool_literal Int_literal Type
Assign_op

%%
// Grammar that'll be parsed by bison
Program
     :    Class ProgramString OpenBracket field_decls method_decls
CloseBracket
     { $$ = new ProgramNode($4, $5); Visitor visitor; $$-
>display(visitor, ""); };
```

```
field_decls
      :    field_decls field_decl
                                     { $$ = new
FieldDeclsNode($1,$2); }
      |
                                            { $$ = new
FieldDeclsNode(NULL, NULL); };

field_decl
      :    Type Identifiers SemiColon
                                 { $$ = new FieldDeclsNode($1, $2);
};

Identifiers
      :    Identifier1
                                            { $$ = new
IdentifiersNode($1, NULL); }
      |    Identifiers Comma Identifier1
                                 { $$ = new IdentifiersNode($3, $1);
};

Identifier1
      :    Identifier

      { $$ = new Identifier1Node(new IdentifierNode($1), NULL); }
      |    Identifier OpenSquare Int_literal CloseSquare

      { $$ = new Identifier1Node(new IdentifierNode($1), $3); };

method_decls
      :    method_decl method_decls
                                 { $$ = new MethodDeclsNode($1, $2);
}
      |
                                            { $$ = new
MethodDeclsNode(NULL, NULL); };

method_decl
      :    Type Identifier OpenParen Params CloseParen Block

      { $$ = new MethodDeclNode($1, new IdentifierNode($2), $4, $6);
}
      |    Type Identifier OpenParen CloseParen Block

      { $$ = new MethodDeclNode($1, new IdentifierNode($2), NULL,
$5); }
      |    Void Identifier OpenParen Params CloseParen Block

      { $$ = new MethodDeclNode(NULL, new IdentifierNode($2), $4,
$6); }
      |    Void Identifier OpenParen CloseParen Block

      { $$ = new MethodDeclNode(NULL, new IdentifierNode($2), NULL,
$5); };

Params
      :    Type Identifier
      { $$ = new ParamsNode($1, new IdentifierNode($2), NULL); }
      |     Type Identifier Comma Params

      { $$ = new ParamsNode($1, new IdentifierNode($2), $4); };

Block
      :     OpenBracket var_decls Statements CloseBracket
                            { $$ = new BlockNode($2, $3); };

var_decls
      :     var_decl var_decls
                                       { $$ = new VarDeclsNode($1,
$2); }
      |
                                            { $$ = new
VarDeclsNode(NULL, NULL); };

var_decl
      :     Type decls SemiColon
                                 { $$ = new VarDeclNode($1, $2); };

decls
      :     Identifier

      { $$ = new DeclsNode(new IdentifierNode($1), NULL); }
      |     Identifier Comma decls

      { $$ = new DeclsNode(new IdentifierNode($1), $3); };

Statements
      :     Statement Statements
                                 { $$ = new StatementsNode($1, $2); }
      |
                                            { $$ = new
StatementsNode(NULL, NULL); };

Statement
      :     Location Assign_op Expr SemiColon

      { $$ = new StatementNode($1, $2, $3, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL); }
      |     Method_call SemiColon

      { $$ = new StatementNode(NULL, NULL, NULL, $1, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL); }
      |     If OpenParen Expr CloseParen Block

      { $$ = new StatementNode(NULL, NULL, NULL, NULL, $3, $5, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL); }
      |     If OpenParen Expr CloseParen Block Else Block

      { $$ = new StatementNode(NULL, NULL, NULL, NULL, $3, $5, $7,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL); }
      |     For Identifier Equal Expr Comma Expr Block

      { $$ = new StatementNode(NULL, NULL, NULL, NULL, NULL, NULL,
NULL, new IdentifierNode($2), $4, $6, $7, NULL, NULL, NULL, NULL,
NULL); }
```

```
        |     Return SemiColon

        { $$ = new StatementNode(NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, new ReturnStrNode($1), NULL, NULL,
NULL, NULL); }
        |     Return Expr SemiColon

        { $$ = new StatementNode(NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, new ReturnStrNode($1), $2, NULL, NULL,
NULL); }
        |     Break SemiColon

        { $$ = new StatementNode(NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, new BreakStrNode($1), NULL,
NULL); }
        |     Continue SemiColon

        { $$ = new StatementNode(NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, new
ContinueStrNode($1), NULL); }
        |     Block

        { $$ = new StatementNode(NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, $1); };

Method_call
        :     Method_name OpenParen PassParams CloseParen
                                { $$ = new MethodCallNode($1, $3, NULL,
NULL); }
        |     Method_name OpenParen CloseParen
                                { $$ = new MethodCallNode($1, NULL, NULL,
NULL); }
        |     Callout    OpenParen String_literal CloseParen

        { $$ = new MethodCallNode(NULL, NULL, new
StringLiteralNode($3), NULL); }
        |     Callout    OpenParen String_literal Comma CalloutArgs
CloseParen
        { $$ = new MethodCallNode(NULL, NULL, new
StringLiteralNode($3), $5); };

PassParams
        :     Expr
                                           { $$ = new PassParamsNode($1,
NULL); }
        |     Expr Comma PassParams
                                      { $$ = new PassParamsNode($1, $3);
};

Method_name
        :     Identifier
                                           { $$ = new MethodNameNode(new
IdentifierNode($1)); };

Location
        :     Identifier

        { $$ = new LocationNode(new IdentifierNode($1), NULL); }
```

```
      |     Identifier OpenSquare Expr CloseSquare

      { $$ = new LocationNode(new IdentifierNode($1), $3); };

CalloutArgs
      :     Expr
                  { $$ = new CalloutArgsNode($1, NULL, NULL, NULL,
NULL, NULL); }
      |     String_literal

      { $$ = new CalloutArgsNode(NULL, new StringLiteralNode($1),
NULL, NULL, NULL, NULL); }
      |     Expr Comma CalloutArgs
                  { $$ = new CalloutArgsNode(NULL, NULL, $1, $3, NULL,
NULL); }
      |     String_literal Comma CalloutArgs

      { $$ = new CalloutArgsNode(NULL, NULL, NULL, NULL, new
StringLiteralNode($1), $3); };

Expr
      :     Location
                  { $$ = new ExprNode($1, NULL, NULL, NULL, NULL,
NULL, NULL); }
      |     Method_call
                        { $$ = new ExprNode(NULL, $1, NULL, NULL, NULL,
NULL, NULL); }
      |     Literal
                        { $$ = new ExprNode(NULL, NULL, $1, NULL, NULL,
NULL, NULL); }
      |     Expr ConditionalOr Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr ConditionalAnd Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr EqualTo Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr NotEqualTo Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr LessThan Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr LessThanEqual Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr GreaterThanEqual Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
```

```
      |     Expr GreaterThan Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr Addition Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr Minus Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr Multiplication Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr Division Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Expr Remainder Expr

      { $$ = new ExprNode(NULL, NULL, NULL, $1, new OperatorNode($2),
$3, NULL); }
      |     Minus Expr %prec UMinus

      { $$ = new ExprNode(NULL, NULL, NULL, NULL, new
OperatorNode($1), $2, NULL); }
      |     Negation Expr

      { $$ = new ExprNode(NULL, NULL, NULL, NULL, new
OperatorNode($1), $2, NULL); }
      |     OpenParen Expr CloseParen

      { $$ = new ExprNode(NULL, NULL, NULL, NULL, NULL, NULL, $2); };

Literal
      :     Int_literal

      { $$ = new LiteralNode($1, NULL, NULL); }
      |     Char_literal

      { $$ = new LiteralNode(NULL, new CharLiteralNode($1), NULL); }
      |     Bool_literal

      { $$ = new LiteralNode(NULL, NULL, $1); };

Bool_literal
      :     True
                                      { $$ = new
BoolLiteralNode($1); }
      |     False
                                      { $$ = new
BoolLiteralNode($1); };

Int_literal
```

```
      :    Decimal_literal
                                       { $$ = new IntLiteralNode($1);
}
      |    Hex_literal
                                            { $$ = new
IntLiteralNode($1); };

Type
      :    Int
                                            { $$ = new TypeNode($1);
}
      |    Boolean
                                            { $$ = new TypeNode($1);
};

Assign_op
      :    Equal
                                            { $$ = new
AssignOpNode($1); }
      |    PlusEqual
                                       { $$ = new AssignOpNode($1); }
      |    MinusEqual
                                       { $$ = new AssignOpNode($1);
};
%%

int main(int argc, char** argv)
{
      FILE *myfile = fopen(argv[1], "r");
      if (!myfile)
      {
           cout << "I can't open input file!" << endl;
           return -1;
      }
      yyin = myfile;
      do
      {
           yyparse();
      } while (!feof(yyin));
      fout.close();
      cout<<"Success\n";
      return 0;
}

void yyerror(const char *s) {
      cout<<"Syntax error\n";
      exit(-1);
}
```

```
// Classes corresponding to all the non terminal nodes and
corresponding visitors

#include <string>
#include <iostream>
#include <sstream>
#include<fstream>

using namespace std;

class AbstractVisitor;
extern string output;
extern string localType;
extern int declarationCount;
extern int methodCount;
extern fstream fout;

class Node
{
     public:
          virtual void display(AbstractVisitor &visitor, string
tabs) = 0;
};

class ProgramNode;
class FieldDeclsNode;
class FieldDeclNode;
class IdentifiersNode;
class Identifier1Node;
class MethodDeclsNode;
class MethodDeclNode;
class ParamsNode;
class BlockNode;
class VarDeclsNode;
class VarDeclNode;
class DeclsNode;
class StatementsNode;
class StatementNode;
class MethodCallNode;
class PassParamsNode;
class MethodNameNode;
class LocationNode;
class CalloutArgsNode;
class ExprNode;
class LiteralNode;
class BoolLiteralNode;
class IntLiteralNode;
class TypeNode;
class AssignOpNode;

class IdentifierNode;
class ReturnStrNode;
class BreakStrNode;
class ContinueStrNode;
class StringLiteralNode;
class CharLiteralNode;
class OperatorNode;
class AbstractVisitor
{
     public:
           virtual void visit(ProgramNode &node, string tabs) = 0;
           virtual void visit(FieldDeclsNode &node, string tabs) = 0;
           virtual void visit(FieldDeclNode &node, string tabs) = 0;
           virtual void visit(IdentifiersNode &node, string tabs) =
0;
           virtual void visit(Identifier1Node &node, string tabs) =
0;
           virtual void visit(MethodDeclsNode &node, string tabs) =
0;
           virtual void visit(MethodDeclNode &node, string tabs) = 0;
           virtual void visit(ParamsNode &node, string tabs) = 0;
           virtual void visit(BlockNode &node, string tabs) = 0;
           virtual void visit(VarDeclsNode &node, string tabs) = 0;
           virtual void visit(VarDeclNode &node, string tabs) = 0;
           virtual void visit(DeclsNode &node, string tabs) = 0;
           virtual void visit(StatementsNode &node, string tabs) = 0;
           virtual void visit(StatementNode &node, string tabs) = 0;
           virtual void visit(MethodCallNode &node, string tabs) = 0;
           virtual void visit(PassParamsNode &node, string tabs) = 0;
           virtual void visit(MethodNameNode &node, string tabs) = 0;
           virtual void visit(LocationNode &node, string tabs) = 0;
           virtual void visit(CalloutArgsNode &node, string tabs) =
0;
           virtual void visit(ExprNode &node, string tabs) = 0;
           virtual void visit(LiteralNode &node, string tabs) = 0;
           virtual void visit(BoolLiteralNode &node, string tabs) =
0;
           virtual void visit(IntLiteralNode &node, string tabs) = 0;
           virtual void visit(TypeNode &node, string tabs) = 0;
           virtual void visit(AssignOpNode &node, string tabs) = 0;

           virtual void visit(IdentifierNode &node, string tabs) = 0;
           virtual void visit(ReturnStrNode &node, string tabs) = 0;
           virtual void visit(BreakStrNode &node, string tabs) = 0;
           virtual void visit(ContinueStrNode &node, string tabs) =
0;
           virtual void visit(StringLiteralNode &node, string tabs) =
0;
           virtual void visit(CharLiteralNode &node, string tabs) =
0;
           virtual void visit(OperatorNode &node, string tabs) = 0;
};

class ProgramNode : public Node
{
     public:
     Node *fieldDecls;
     Node *methodDecls;

     ProgramNode(Node *l, Node *m)
     {
           fieldDecls = l;
           methodDecls = m;
     }
     void display(AbstractVisitor &visitor, string tabs)
```

```
	{
		visitor.visit(*this, tabs);
	}
};

class FieldDeclsNode : public Node
{
	public:
	Node *fieldDecls;
	Node *fieldDecl;

	FieldDeclsNode(Node *l, Node *m)
	{
		fieldDecls = l;
		fieldDecl = m;
	}
	void display(AbstractVisitor &visitor, string tabs)
	{
		visitor.visit(*this, tabs);
	}
};

class FieldDeclNode : public Node
{
	public:
	Node *type;
	Node *identifiers;

	FieldDeclNode(Node *l, Node *m)
	{
		type = l;
		identifiers = m;
	}
	void display(AbstractVisitor &visitor, string tabs)
	{
		visitor.visit(*this, tabs);
	}
};

class IdentifiersNode : public Node
{
	public:
	Node *identifiers1;
	Node *identifiers;

	IdentifiersNode(Node *l, Node *m)
	{
		identifiers1 = l;
		identifiers = m;
	}
	void display(AbstractVisitor &visitor, string tabs)
	{
		visitor.visit(*this, tabs);
	}
};

class IdentifierNode : public Node
{
    public:
    string identifier;

    IdentifierNode(char *l)
    {
        identifier = string(l);
    }
    void display(AbstractVisitor &visitor, string tabs)
    {
        visitor.visit(*this, tabs);
    }
};

class ReturnStrNode : public Node
{
    public:
    string returnStr;

    ReturnStrNode(char *l)
    {
        returnStr = string(l);
    }
    void display(AbstractVisitor &visitor, string tabs)
    {
        visitor.visit(*this, tabs);
    }
};

class BreakStrNode : public Node
{
    public:
    string breakStr;

    BreakStrNode(char *l)
    {
        breakStr = string(l);
    }
    void display(AbstractVisitor &visitor, string tabs)
    {
        visitor.visit(*this, tabs);
    }
};

class ContinueStrNode : public Node
{
    public:
    string continueStr;

    ContinueStrNode(char *l)
    {
        continueStr = string(l);
    }
    void display(AbstractVisitor &visitor, string tabs)
    {
        visitor.visit(*this, tabs);
    }
};
```

```
class StringLiteralNode : public Node
{
      public:
      string stringLiteral;

      StringLiteralNode(char *l)
      {
            stringLiteral = string(l);
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};

class CharLiteralNode : public Node
{
      public:
      string charLiteral;

      CharLiteralNode(char *l)
      {
            charLiteral = string(l);
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};

class OperatorNode : public Node
{
      public:
      string operator1;

      OperatorNode(char *l)
      {
            operator1 = string(l);
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};

class Identifier1Node : public Node
{
      public:
      Node* identifier;
      Node *intLiteral;

      Identifier1Node(Node *l, Node *m)
      {
            identifier = l;
            intLiteral = m;
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
        }
};

class MethodDeclsNode : public Node
{
        public:
        Node *methodDecl;
        Node *methodDecls;

        MethodDeclsNode(Node *l, Node *m)
        {
            methodDecl = l;
            methodDecls = m;
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
            visitor.visit(*this, tabs);
        }
};

class MethodDeclNode : public Node
{
        public:
        Node *type;
        Node *identifier;
        Node *params;
        Node *block;

        MethodDeclNode(Node *l, Node *m, Node *n, Node *o)
        {
            type = l;
            identifier = m;
            params = n;
            block = o;
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
            visitor.visit(*this, tabs);
        }
};

class ParamsNode : public Node
{
        public:
        Node *type;
        Node *identifier;
        Node *params;

        ParamsNode(Node *l, Node *m, Node *n)
        {
            type = l;
            if(m)
                identifier = m;
            params = n;
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
                visitor.visit(*this, tabs);
        }
};

class BlockNode : public Node
{
        public:
        Node *varDecls;
        Node *statements;

        BlockNode(Node *l, Node *m)
        {
                varDecls = l;
                statements = m;
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
                visitor.visit(*this, tabs);
        }
};

class VarDeclsNode : public Node
{
        public:
        Node *varDecl;
        Node *varDecls;

        VarDeclsNode(Node *l, Node *m)
        {
                varDecl = l;
                varDecls = m;
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
                visitor.visit(*this, tabs);
        }
};

class VarDeclNode : public Node
{
        public:
        Node *type;
        Node *decls;

        VarDeclNode(Node *l, Node *m)
        {
                type = l;
                decls = m;
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
                visitor.visit(*this, tabs);
        }
};

class DeclsNode : public Node
{
        public:
```

```
      Node *identifier;
      Node *decls;

      DeclsNode(Node *l, Node *m)
      {
            identifier = l;
            decls = m;
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};

class StatementsNode : public Node
{
      public:
      Node *statement;
      Node *statements;

      StatementsNode(Node *l, Node *m)
      {
            statement = l;
            statements = m;
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};

class StatementNode : public Node
{
      public:
      Node *location;
      Node *assignOp;
      Node *Expr;
      Node *methodCall;
      Node *exprInsideParen;
      Node *blockAfterIf;
      Node *blockAfterElse;
      Node *identifierInsideFor;
      Node *exprInsideFor;
      Node *exprInsideForAfterComma;
      Node *blockInsideFor;
      Node *returnStr;
      Node *exprAfterReturn;
      Node *breakStr;
      Node *continueStr;
      Node *block;

      StatementNode(Node *l, Node *m, Node *n, Node *o, Node *p, Node
*q, Node *r, Node *s, Node *t, Node *u, Node *v, Node *w, Node *x,
Node *y, Node *z, Node *a)
      {
            location = l;
            assignOp = m;
            Expr = n;
```

```
			methodCall = o;
			exprInsideParen = p;
			blockAfterIf = q;
			blockAfterElse = r;
			identifierInsideFor = s;
			exprInsideFor = t;
			exprInsideForAfterComma = u;
			blockInsideFor = v;
			returnStr = w;
			exprAfterReturn = x;
			breakStr = y;
			continueStr = z;
			block = a;
		}
		void display(AbstractVisitor &visitor, string tabs)
		{
			visitor.visit(*this, tabs);
		}
	};


	class MethodCallNode : public Node
	{
		public:
		Node *methodName;
		Node *passParams;
		Node *stringLiteralInsideParen;
		Node *callOutArgs;

		MethodCallNode(Node *l, Node *m, Node *n, Node *o)
		{
			methodName = l;
			passParams = m;
			stringLiteralInsideParen = n;
			callOutArgs = o;
		}
		void display(AbstractVisitor &visitor, string tabs)
		{
			visitor.visit(*this, tabs);
		}
	};

	class PassParamsNode : public Node
	{
		public:
		Node *expr;
		Node *passParams;

		PassParamsNode(Node *l, Node *m)
		{
			expr = l;
			passParams = m;
		}
		void display(AbstractVisitor &visitor, string tabs)
		{
			visitor.visit(*this, tabs);
		}
	};
```

```
class MethodNameNode : public Node
{
      public:
      Node *identifier;

      MethodNameNode(Node *l)
      {
            identifier = l;
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};

class LocationNode : public Node
{
      public:
      Node *identifier;
      Node *expr;

      LocationNode(Node *l, Node *m)
      {
            identifier = l;
            expr = m;
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};

class CalloutArgsNode : public Node
{
      public:
      Node *expr;
      Node *stringLiteral;
      Node *exprBeforeComma;
      Node *calloutArgsWithExpr;
      Node *stringLiteralBeforeComma;
      Node *calloutArgsWithStringLiteral;

      CalloutArgsNode(Node *l, Node *m, Node *n, Node *o, Node *p,
Node *q)
      {
            expr = l;
            stringLiteral = m;
            exprBeforeComma = n;
            calloutArgsWithExpr = o;
            stringLiteralBeforeComma = p;
            calloutArgsWithStringLiteral = q;
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};
```

```
class ExprNode : public Node
{
      public:
      Node *location;
      Node *methodCall;
      Node *literal;
      Node *exprBeforeOperator;
      Node *operator1;
      Node *exprAfterOperator;
      Node *exprInsideParen;

      ExprNode(Node *l, Node *m, Node *n, Node *o, Node *p, Node *q,
Node *r)
      {
            location = l;
            methodCall = m;
            literal = n;
            exprBeforeOperator = o;
            operator1 = p;
            exprAfterOperator = q;
            exprInsideParen = r;
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};

class LiteralNode : public Node
{
      public:
      Node *intLiteral;
      Node *charLiteral;
      Node *boolLiteral;

      LiteralNode(Node *l, Node *m, Node *n)
      {
            intLiteral = l;
            charLiteral = m;
            boolLiteral = n;
      }
      void display(AbstractVisitor &visitor, string tabs)
      {
            visitor.visit(*this, tabs);
      }
};


class BoolLiteralNode : public Node
{
      public:
      string boolLiteral;

      BoolLiteralNode(char *l)
      {
            boolLiteral = string(l);
```

```
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
              visitor.visit(*this, tabs);
        }
};

class IntLiteralNode : public Node
{
        public:
        string intLiteral;

        IntLiteralNode(char *l)
        {
              intLiteral = string(l);
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
              visitor.visit(*this, tabs);
        }
};

class TypeNode : public Node
{
        public:
        string type;

        TypeNode(char *l)
        {
              type = string(l);
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
              visitor.visit(*this, tabs);
        }
};
class AssignOpNode : public Node
{
        public:
        string assignOp;
        AssignOpNode(char *l)
        {
              assignOp = string(l);
        }
        void display(AbstractVisitor &visitor, string tabs)
        {
              visitor.visit(*this, tabs);
        }
};

// Visitor class to visit all the nodes
class Visitor : public AbstractVisitor
{
        public:
              void visit(ProgramNode &node, string tabs)
              {
                    stringstream countOutD;
                    stringstream countOutS;
```

```
                tabs = tabs + "\t";
                if(node.fieldDecls)
                     node.fieldDecls->display(*this, tabs+"\t");
                output = output + "\t</field_declarations>\n";
                string temp = output;
                output = "";
                if(node.methodDecls)
                     node.methodDecls->display(*this, tabs+"\t");
                output = output + "\t</methods>\n";

                countOutD << declarationCount;
                countOutS << methodCount;

                output = "\t<methods count=\"" + countOutS.str() +
"\">\n" + output;
                output = temp + output;
                output = "\t<field_declarations count=\"" +
countOutD.str() + "\">\n" + output;
                output = "<program>\n" + output;
                output = output + "</program>\n";
                fout<<output;
           }

           void visit(FieldDeclsNode &node, string tabs)
           {
                if(node.fieldDecls)
                     node.fieldDecls->display(*this, tabs);
                if(node.fieldDecl)
                     node.fieldDecl->display(*this, tabs);
           }

           void visit(FieldDeclNode &node, string tabs)
           {
                if(node.type)
                     node.type->display(*this, tabs);
                if(node.identifiers)
                     node.identifiers->display(*this, tabs);
           }

           void visit(IdentifiersNode &node, string tabs)
           {
                if(node.identifiers1)
                     node.identifiers1->display(*this, tabs);
                if(node.identifiers)
                     node.identifiers->display(*this, tabs);
           }

           void visit(Identifier1Node &node, string tabs)
           {
                declarationCount++;
                if(node.intLiteral)
                {
                     output = output + tabs+"Array\n";
                     output = output + tabs+"<declaration name=\"";
                     if(node.identifier)
                          node.identifier->display(*this, tabs);
                     output = output + "\"";
                     output = output + " count=\"";
                node.intLiteral->display(*this, tabs);

                output = output + "\"";
                output = output + " type=\""+localType+"\"
/>\n";
            }
            else
            {
                output = output + tabs+"Normal\n";
                output = output + tabs+"<declaration name=\"";
                if(node.identifier)
                    node.identifier->display(*this, tabs);
                output = output + "\"";
                output = output + " type=\""+localType+"\"
/>\n";
            }
        }

        void visit(MethodDeclsNode &node, string tabs)
        {
            if(node.methodDecl)
                node.methodDecl->display(*this, tabs);
            if(node.methodDecls)
                node.methodDecls->display(*this, tabs);
        }

        void visit(MethodDeclNode &node, string tabs)
        {
            methodCount++;
            output = output + tabs+"<method name=\"";
            if(node.identifier)
                node.identifier->display(*this, tabs);
            output = output + "\" return_type=\"";
            if(node.type)
            {
                node.type->display(*this, tabs);
                output = output + localType+"\"";
            }
            else
                output = output + "void";
            output = output + ">\n";
            if(node.params)
            {
                output = output + tabs + "\t" +"<params>\n";
                node.params->display(*this, tabs + "\t\t");
                output = output + tabs+"\t"+"</params>\n";
            }
            if(node.block)
            {
                output = output + tabs + "\t" +"<body>\n";
                node.block->display(*this, tabs + "\t\t");
                output = output + tabs+"\t"+"</body>\n";
            }
            output = output + tabs+"</method>\n";
        }

        void visit(ParamsNode &node, string tabs)
        {
            if(node.type)
            {
                output = output + tabs+"<param type=\"";
                node.type->display(*this, tabs);
                output = output + localType+"\" name=\"";
                if(node.identifier)
                    node.identifier->display(*this, tabs);
                output = output + "\">\n";
            }
            if(node.params)
                node.params->display(*this, tabs);
        }

        void visit(BlockNode &node, string tabs)
        {
            if(node.varDecls)
            {
                output = output + tabs+"<locals>\n";
                node.varDecls->display(*this, tabs+"\t");
                output = output + tabs+"</locals>\n";
            }
            if(node.statements)
            {
                output = output + tabs+"<statements>\n";
                node.statements->display(*this, tabs + "\t");
                output = output + tabs+"</statements>\n";
            }
        }

        void visit(VarDeclsNode &node, string tabs)
        {
            if(node.varDecl)
                node.varDecl->display(*this, tabs);
            if(node.varDecls)
                node.varDecls->display(*this, tabs);
        }

        void visit(VarDeclNode &node, string tabs)
        {
            if(node.type)
                node.type->display(*this, tabs);
            if(node.decls)
                node.decls->display(*this, tabs);
        }

        void visit(DeclsNode &node, string tabs)
        {
            if(node.identifier)
            {
                output = output + tabs+"<local name=\"";
                if(node.identifier)
                    node.identifier->display(*this, tabs);
                output = output + "\"";
                output = output + " type=\""+localType+"\"
/>\n";
            }
            if(node.decls)
                node.decls->display(*this, tabs);
            }

            void visit(StatementsNode &node, string tabs)
            {
                  if(node.statement)
                        node.statement->display(*this, tabs);
                  if(node.statements)
                        node.statements->display(*this, tabs);
            }

            void visit(StatementNode &node, string tabs)
            {
                  if(node.assignOp)
                  {
                        output = output + tabs+"<assignment>\n";
                        node.assignOp->display(*this, tabs + "\t");
                        if(node.location)
                              node.location->display(*this, tabs +
"\t");
                        if(node.Expr)
                              node.Expr->display(*this, tabs + "\t");
                        output = output + tabs+"</assignment>\n";
                  }
                  if(node.methodCall)
                        node.methodCall->display(*this, tabs);
                  if(node.exprInsideParen)
                        node.exprInsideParen->display(*this, tabs);
                  if(node.blockAfterIf)
                        node.blockAfterIf->display(*this, tabs);
                  if(node.blockAfterElse)
                        node.blockAfterElse->display(*this, tabs);
                  if(node.identifierInsideFor)
                  {
                        output = output + tabs + "<for>\n";
                        output = output + tabs+"\t"+"identifier
value=\"";
                        node.identifierInsideFor->display(*this, tabs);
                        output = output + ">\n";
                        if(node.exprInsideFor)
                              node.exprInsideFor->display(*this,
tabs+"\t");
                        if(node.exprInsideForAfterComma)
                              node.exprInsideForAfterComma-
>display(*this, tabs+"\t");
                        if(node.blockInsideFor)
                              node.blockInsideFor->display(*this,
tabs+"\t");
                        output = output + tabs + "</for>\n";
                  }
                  if(node.returnStr)
                        node.returnStr->display(*this, tabs);
                  if(node.exprAfterReturn)
                        node.exprAfterReturn->display(*this, tabs);
                  if(node.breakStr)
                        node.breakStr->display(*this, tabs);
                  if(node.continueStr)
                        node.continueStr->display(*this, tabs);
                  if(node.block)
```

```
                      node.block->display(*this, tabs);
           }

           void visit(MethodCallNode &node, string tabs)
           {
                if(node.methodName)
                {
                      output = output + tabs+"<calling method
name=\"";
                      node.methodName->display(*this, tabs);
                      output = output + "\">\n";
                      if(node.passParams)
                           node.passParams->display(*this,
tabs+"\t");
                      output = output + tabs+"</calling>\n";
                }
                if(node.stringLiteralInsideParen)
                {
                      output = output + tabs+"<callout function=";
                      node.stringLiteralInsideParen->display(*this,
tabs);
                      output = output + ">\n";
                      if(node.callOutArgs)
                           node.callOutArgs->display(*this, tabs +
"\t");
                      output = output + tabs+"</callout>\n";
                }
           }

           void visit(PassParamsNode &node, string tabs)
           {
                if(node.expr)
                      node.expr->display(*this, tabs);
                if(node.passParams)
                      node.passParams->display(*this, tabs);
           }

           void visit(MethodNameNode &node, string tabs)
           {
                if(node.identifier)
                      node.identifier->display(*this, tabs);
           }

           void visit(LocationNode &node, string tabs)
           {
                if(node.expr)
                {
                      output = output + tabs+"Array\n";
                      if(node.identifier)
                      {
                           output = output +  tabs +"<location
id=\"";
                           node.identifier->display(*this, tabs);
                           output = output + "\"/>\n";
                      }
                      output = output + tabs + "\t"+"<position>\n";
                      node.expr->display(*this, tabs + "\t\t");
                      output = output + tabs + "\t"+"</position>\n";
                output = output +  tabs +"</location>\n";
            }
            else
            {
                output = output + tabs+"Normal\n";
                if(node.identifier)
                {
                    output = output +  tabs +"<location
id=\"";
                    node.identifier->display(*this, tabs);
                    output = output + "\"/>\n";
                }
            }
        }

        void visit(CalloutArgsNode &node, string tabs)
        {
            if(node.expr)
                node.expr->display(*this, tabs);
            if(node.stringLiteral)
            {
                output = output + tabs + "<string value =";
                node.stringLiteral->display(*this, tabs);
                output = output + "/>\n";
            }
            if(node.exprBeforeComma)
                node.exprBeforeComma->display(*this, tabs);
            if(node.calloutArgsWithExpr)
                node.calloutArgsWithExpr->display(*this, tabs);
            if(node.stringLiteralBeforeComma)
            {
                output = output + tabs + "<string value =";
                node.stringLiteralBeforeComma->display(*this,
tabs);
                output = output + "/>\n";
            }
            if(node.calloutArgsWithStringLiteral)
            {
                output = output + tabs + "<string value =";
                node.calloutArgsWithStringLiteral-
>display(*this, tabs);
                output = output + "/>\n";
            }
        }

        void visit(ExprNode &node, string tabs)
        {
            if(node.location)
                node.location->display(*this, tabs);
            if(node.methodCall)
                node.methodCall->display(*this, tabs);
            if(node.literal)
                node.literal->display(*this, tabs);
            if(node.exprBeforeOperator)
            {
                output = output +  tabs + "<binary_expression
type=\"";
                if(node.operator1)
```

```
                    node.operator1->display(*this, tabs);
                output = output + "\">\n";
                node.exprBeforeOperator->display(*this, tabs +
"\t");
                if(node.exprAfterOperator)
                    node.exprAfterOperator->display(*this,
tabs + "\t");
                output = output +  tabs +
"</binary_expression>\n";
            }
            else if(node.operator1)
            {
                output = output +  tabs + "<unary_expression
type=\"";
                node.operator1->display(*this, tabs);
                output = output + "\">\n";
                if(node.exprAfterOperator)
                    node.exprAfterOperator->display(*this,
tabs + "\t");
                output = output +  tabs +
"</unary_expression>\n";
            }
            if(node.exprInsideParen)
                node.exprInsideParen->display(*this, tabs);
        }

        void visit(LiteralNode &node, string tabs)
        {
            if(node.intLiteral)
            {
                output = output + tabs + "<integer value =\"";
                node.intLiteral->display(*this, tabs);
            }
            if(node.charLiteral)
            {
                output = output + tabs + "<boolean value =\"";
                node.charLiteral->display(*this, tabs);
            }
            if(node.boolLiteral)
            {
                output = output + tabs + "<character value
=\"";

                node.boolLiteral->display(*this, tabs);
            }
            output = output + "\"/>\n";
        }

        void visit(BoolLiteralNode &node, string tabs)
        {
            output = output + node.boolLiteral;
        }

        void visit(IntLiteralNode &node, string tabs)
        {
            output = output + node.intLiteral;
        }

        void visit(CharLiteralNode &node, string tabs)
```

```
{
      output = output + node.charLiteral;
}

void visit(StringLiteralNode &node, string tabs)
{
      output = output + node.stringLiteral;
}

void visit(TypeNode &node, string tabs)
{
      localType = node.type;
}

void visit(AssignOpNode &node, string tabs)
{
}

void visit(IdentifierNode &node, string tabs)
{
      output = output + node.identifier;
}
void visit(ReturnStrNode &node, string tabs)
{
}
void visit(BreakStrNode &node, string tabs)
{
}
void visit(ContinueStrNode &node, string tabs)
{
}

void visit(OperatorNode &node, string tabs)
{
      if(node.operator1.compare("+") == 0)
            output = output + "addition";
      if(node.operator1.compare("-") == 0)
            output = output + "minus";
      if(node.operator1.compare("*") == 0)
            output = output + "multiplication";
      if(node.operator1.compare("/") == 0)
            output = output + "division";
      if(node.operator1.compare("%") == 0)
            output = output + "remainder";
      if(node.operator1.compare("<") == 0)
            output = output + "less_than";
      if(node.operator1.compare(">") == 0)
            output = output + "greater_than";
      if(node.operator1.compare("<=") == 0)
            output = output + "less_equal";
      if(node.operator1.compare(">=") == 0)
            output = output + "greater_equal";
      if(node.operator1.compare("==") == 0)
            output = output + "is_equal";
      if(node.operator1.compare("!=") == 0)
            output = output + "is_not_equal";
      if(node.operator1.compare("&&") == 0)
            output = output + "and";
```

```
                if(node.operator1.compare("||") == 0)
                     output = output + "or";
                if(node.operator1.compare("!") == 0)
                     output = output + "not";
           }
};
```

```
# Use this file to run above codes
# Input: “test_input” containing decaf code
# Output: XML_visitor.txt

#!/bin/bash

g++ decaf.h
bison -dv decaf.y
flex decaf.l
g++ decaf.tab.c lex.yy.c -lfl -o decaf
./decaf  test_input
```