COMP6421 Assignment 3

Current Version: 0.1

Date: Mar. 26, 2012

Author: Yuan Tao

Instructor: Dr. Joey Paquet

Concordia University

Table of Contents

1. Data Structures

The main data structure this program uses is ArrayList, which is a basic class provided by Java. The class has already defined a serial of easily used interfaces for us to access the List structure. Such as add(), size(), get() etc. As a result, based on the interfaces provided by Java, some of the functions defined in the requirement of assignment 3, like search, insert and delete, are not implemented by this program. Instead, this program provides a serial of semantic actions which provide similar functionalities as those functions in order to access the symbol tables.
Next, let us see the data structure of the symbol table and symbol.

1.1. Symbol Table

```
public class SymbolTable implements Cloneable {
    // pointer to its parent symbol which created this symbol table;
    // null means the first table
    Symbol parent = null;
    ArrayList<Symbol> symbols = new ArrayList<Symbol>();

    String addrPrefix;     // prefix of all the symbol addresses of this table
}
```

By using the parent member variable, the program can access the symbol table of the class or the function defined, as well as the global symbols.

1.2. Symbol

```
public class Symbol implements Cloneable {
    public static enum SYMBOLTYPE {
        UNKNOWN, CLASS, FUNCTION, VARIABLE, PARAMETER,
        UNKNOWN_EXITTABLE, ARRAYSIZE, CHKVAR, CHKTYPE, CHKMEMBER,};

    public SYMBOLTYPE symbolType;     // used by semantic actions
    public Token tk;                  // symbol name
    public Token dataType;            // data type of symbol
    public int size;                  // size of the symbol
    public String address;            // unique name

    public boolean ifAlreadyDefined = false;
    public boolean ifUnkownDataType = false;

    public SymbolTable self = null;   // pointer to the symbol table it is placed;

    // class or function
    public SymbolTable child = null;   // pointer to its symbol table;

    // array
    public boolean isArray = false;
    public int dimensions;             // the number of dimensions
    public ArrayList<Integer> sizeOfDimension = new ArrayList<Integer>();
}
```

Token Class is defined in the Lexical Analyzer, more information about it please refer to 'Readme.pdf' of assignment 1. Here is the definition:

```
public class Token implements Cloneable {
    public String file = null;     // file name
    public int line = 0;           // line number
    public int column = 0;         // the column of the first character
    public String token = null;    //
    public int type;               //

    public boolean error = false;  // true means the token is generated by error
recovery technique

}
```

By using the child member variable, the program can access all the symbols defined in a class or a function. For a function, the symbols include parameters. It means the parameters and variables of the function are in the same symbol table.

2. Semantic Actions

```
// create the first symbol table
public int newProg()

// add a new class
public int newClass(Symbol s)

// exit a class or function definition
public int exitCurSymbolTable()

// check if the data type of the symbol is defined
public boolean ifDataTypeDefined(Symbol s)

// add a new function
public int newFunction(Symbol s)

// check if the function is redefined
// s is the last parameter of this function or the function identification
public boolean ifFuncRedefined(Symbol s)

// add a new variable for the function
public int newVarible(Symbol s)

// check if the variable has been defined
// get the data type of s as well, if it is a function, wait until all the
// parameters of the function has been parsed.
public boolean ifVarDefined(Symbol s)

// check if the token is a class member
public boolean ifClassMember(Symbol s, Symbol m)

// check if the type of expression is a valid type for the index of array
public boolean ifValidIndexType(Symbol s)

// compare date types
public Symbol compDateType(Symbol a, Symbol b)

// only real or integer type is allowed
public Symbol compDateTypeNum(Symbol a, Symbol b)

// check all parameters for a function variable
// get the exact data type of the function variable
public boolean ifValidFuncParamType(Symbol var)
```

Besides, there are several actions are added directly into the grammar.

- `storeFunctionParam     // function definition`

- `addFunctionParam       // function variable`
- `getTokenInfo`
- `storeArrayInfo`

3. Grammar & Actions

The grammar of this compiler is defined as the following, more information about it please refer to 'Readme.pdf' of assignment 2.

Note that the semantic actions are inserted into the grammar in **red** color. The actions used to check the data type of the symbol are in **orange** color.

```
prog              -> newProg classDeclList progBody
classDeclList    -> classDecl classDeclList
                  | EPSILON
classDecl         -> class id newClass { varFuncDeclList } ; exitCurSymbolTable
varFuncDeclList   -> type id varFuncDeclListP
                  | EPSILON
varFuncDeclListP  ->  (newFunction  fParams  )  funcBody  ;  exitCurSymbolTable
funcDefList
                  | arraySizeList ; varFuncDeclList
funcDefList       -> funcDef funcDefList
                  | EPSILON
funcDef           -> funcHead funcBody ; exitCurSymbolTable
funcHead           -> type id (newFunction fParams ) ifFuncRedefined
funcBody           -> { varStateList }
varStateList     -> if ( expr ) then statBlock else statBlock ; statementList
                  | while ( expr ) do statBlock ; statementList
                  | read ( variable ) ; statementList
                  | return ( expr ) ; statementList
                  | write ( expr ) ; statementList
                  | integer id arraySizeList ; varStateList
                  | real id arraySizeList ; varStateList
                  | id varStateListP
                  | EPSILON
varStateListP     -> ifVarDefined indiceList variableP assignOp expr compDateType ;
statementList
                  | ifDataTypeDefined id arraySizeList ; varStateList
progBody           -> program funcBody ; funcDefList
statementList     -> statement statementList
                  | EPSILON
arraySizeList    -> arraySize arraySizeList newVarible
                  | EPSILON
statement          -> if ( expr ) then statBlock else statBlock ;
                  | while ( expr ) do statBlock ;
                  | read ( variable ) ;
                  | return ( expr ) ;
                  | write ( expr ) ;
                  | variable assignOp expr ; compDateType
statBlock         -> { statementList } | statement
                  | EPSILON
expr              -> arithExpr exprP
exprP             -> relOp arithExpr compDateTypeNum | EPSILON
```

```
arithExpr         -> term arithExprP
arithExprP        -> addOp term arithExprP compDateTypeNum
                  | EPSILON
sign              -> + | -
term              -> factor termP
termP             -> multOp factor termP compDateTypeNum
                  | EPSILON
factor            -> ( expr ) | id ifVarDefined factorPP | num getTokenInfo | not
factor
                  | sign factor
factorPP          -> ( aParams ) ifValidFuncParamType | indiceList factorP
factorP           -> . id ifClassMember factorPP | EPSILON
variable          -> id ifVarDefined indiceList variableP
variableP         -> . id ifClassMember indiceList variableP
                  | EPSILON
indiceList        -> indice indiceList | EPSILON
indice            -> [ arithExpr ] ifValidIndexType
arraySize         -> [ int storeArrayInfo  ]
type              -> id ifDataTypeDefined | integer | real
fParams           -> type id storeFunctionParam arraySizeList fParamsTailList
                  | EPSILON
fParamsTailList   -> fParamsTail fParamsTailList
                  | EPSILON
aParams           -> expr addFunctionParam aParamsTailList
                  | EPSILON
aParamsTailList   -> aParamsTail aParamsTailList
                  | EPSILON
fParamsTail       -> , type id storeFunctionParam arraySizeList
aParamsTail       -> , expr addFunctionParam
```

4. Semantic Errors & Warnings

The following are the semantic errors can be found by this program:

- Undeclared identifier
- Undeclared type
- Variable redefinition
- Class redefinition
- Function redefinition
- 'xxx' is not a member of 'yyy' // class yyy does not have member xxx
- left of '.xxx' must have class type
- Function parameters mismatch
- Cannot convert from 'xxx' to 'yyy'
- Type should be integer or real // for addOp, addOp and multOp
- Invalid array index type: xxx

The following are the semantic warnings defined by this program:

- Convert from 'integer' to 'real' // addOp, addOp and multOp
- Convert from 'real' to 'integer' // assignOp
- Convert parameter x from 'real' to 'integer' // x is the sequence number of the parameter
- Convert parameter x from 'integer' to 'real'

5. How to run

1. Install JRE (v5.0 or higher)
2. Extract COMP6421Ass3.zip to $DIR
3. cd $DIR\COMP6421Ass3
4. $ java -jar COMP6421Ass3.jar

The results of symbol tables and error messages are stored under $DIR\output.

Appendix A: Revision History

Version	Date	Author	Remark
V0.1	Mar. 26, 2012	Yuan Tao	Draft