CMPSC 470

Project 3

Due: Friday, December 1, 2017

Semantic Checker (200 points)

Semantic checker is a program that check semantic error in programming languages. It read a source program, and check if there is any error regarding type and scope of variables, and output error messages describing any semantic error found. In this assignment, you should build a class that maintain symbol tables to check scope of variable, and build the semantic checker using the class.

1. Chained Symbol Tables (50 points)

One way of checking the scope of variables is using the chained symbol tables. In this assignment, to build semantic checker, you should use the **`Env`** class that maintains the chained symbol tables. It has, at least, three operations as described in the class:

- **`Env(Env p)`** : a constructor;
- **`Put(String s, Object sym)`** : adding a new symbol entry into the current symbol table;
- **`Get(String s)`** : returning a symbol entry whose key is s, or **`null`** if the symbol entry does not exists.

The **`Env`** prototype class and the **`TestEnv`** class, containing 10 test cases, are provided in the startup program. You will get 5 points per each test case in **`TestEnv`** class.

2. Semantic Checker (150 points)

After completing the **`Env`** class, you should build the semantic checker implemented using java. Following describes the task of your program:

- Your program should use **`jflex`** to determine tokens from an input ***foo*** source program.
- Your program should use bottom-up parser, generated by **`BYacc/J`**. The updated ***foo*** grammar will be given at the end of this document, in the form of CFG.
- You should use the **`Env`** class to check scopes of variables, their types, and types of declared functions.
- If there is **<u>no</u>** semantic error, then your program should print "**`Success`**" on console;
  If there is **<u>any</u>** semantic error(s), then your program should print detailed error message on console, such as "**`Error at line 4: try to assign float value to int variable x.`**".

For example, let you have the `semantic-checker` java program, and the following sample input foo programs:

success03.foo	error03b.foo

```
main()
{
    int x;
    x = x + 1;
}
```
	```
	main()
	{
	    int x;
	    x = x + 1.0;
	}
	```

Running `semantic-checker` program with the above foo programs should print the following outputs on console:

```
> java semantic-checker success03.foo
Success
>
```

```
> java semantic-checker error03b.foo
Error at line 4: try to assign float value
to int variable x.
>
```

Note that this `semantic-checker` determines only the first semantic error, such as variable type mismatch or function type mismatch or the use of non-declared variables/functions, and then prints its appropriate error message with the line number of the error location in the input program.

Start-up program, test cases, and points

The start-up program, the **TestEnv** test program, and totally 75 test foo programs will be available at `/home/cmpsc470/f17/proj3-startup.zip` in sunlab and https://turing.cs.hbg.psu.edu/cmpsc470/proj3-startup.zip in course website.

Regarding the **TestEnv** test program, you will get 5 points from each 10 test cases, and you will get totally 50 points if your **Env** class passes all tests.

Regarding 75 test foo programs, each test foo program whose filename starts with "success" does not have semantic error, and whose file name starts with "error" has at least one semantic error, and whose file name ends with "-message.txt" shows its answer error message. You will get 2 points per each test foo program by the following rules, and the total 150 points will be determined from the 75 foo test programs.

- Regarding success files, you will get 2 points if your `semantic-checker` program correctly identify it.
- Regarding error files, you will get 1.5 points if your `semantic-checker` program prints correct error message, and another 0.5 points if your `semantic-checker` program correctly identifies the line number.

You **may lose some points** if your program does not terminate after printing "`Success`" or "`Error: ...`" on console.

What to submit:

- Submit one zip file containing following files via **canvas by 11:59:59 PM, Friday, December 1, 2017**.
- **Readme file** describing how to compile your semantic checker program using **`jflex`** and **`BYacc/J`** and how to run your program. Grader will recompile both your **`jflex`** `(*.flex)` and **`yacc`** `(*.y)` file(s).
- Your own **jflex** , **`yacc`**, and **java** source files that implements the semantic checker program.

The *foo* grammar:

The following describes the ***foo*** grammar.

```
1.  program        -> decl_list main_decl
2.  decl_list      -> decl_list decl | eps
3.  decl           -> fun_decl
4.  type_spec      -> "int" | "float" | "bool"
5.  main_decl      -> "main" "(" ")" compound_stmt
6.  fun_decl       -> type_spec ID "(" params ")" ";"
7.  params         -> param_list | eps
8.  param_list     -> param_list "," param | param
9.  param          -> type_spec
10. stmt_list      -> stmt_list stmt | eps
11. stmt           -> expr_stmt | print_stmt | compound_stmt | if_stmt | while_stmt | return_stmt
12. print_stmt     -> "print" expr ";"
13. expr_stmt      -> ID "=" expr ";" | ";"
14. while_stmt     -> "while" "(" expr ")" stmt
15. compound_stmt  -> "{" local_decls stmt_list "}"
16. local_decls    -> local_decls local_decl | eps
17. local_decl     -> type_spec ID ";"
18. if_stmt        -> "if" "(" expr ")" stmt "else" stmt
19. return_stmt    -> "return" ";"
20. arg_list       -> arg_list "," expr | expr
21. args           -> arg_list | eps
22. expr           -> expr "or" expr | expr "and" expr | "not" expr    | expr "==" expr
                    | expr "!=" expr | expr "<" expr   | expr ">" expr | expr ">=" expr
                    | expr "<=" expr | expr "+" expr   | expr "-" expr | expr "*" expr
                    | expr "/" expr  | expr "%" expr   | "(" expr ")" | ID | ID "(" args ")"
                    | NUM | REAL | "true" | "false"
```

In the grammar, italicized words with lower-case letters represent non-terminals (such as *program* and *type_spec*), *program* is the starting symbol, and *eps* represents the empty string. The words or symbols enclosed by quotation marks are keywords and symbols (such as `"int"` and `"{"`), respectively, and

- **NUM** represents positive integers,
- **REAL** represents positive real numbers, which is not integer. It can be `1E2`, `2.0`, `2.`, `1E+3`, `1E-4`, etc., but not `10`,
- **ID** represents C language identifiers, which start with lowercase letter or capital letter or '`_`'.

Comments (a string that starts with "`//`" and ends at the end of line), and whitespaces("`[ \t\r]+`") must be ignored in lexical analyzer implemented using **`jflex`**. New line ("`\n`") should count line number when reading input ***foo*** program to use it when printing semantic error, but should not return a token to parser implemented using **`BYacc/J`**.

In the foo grammar,

- The *fun_decl* in line 6 represents the function declaration that does not have function body, such as "**`double sin(double);`**",
- The *local_decl* in line 17 represents the local variable declaration such as "**`int x;`**",
- The *compound_stmt* in line 15 represent the block that has local variable declarations, such as "**`{ int x; x = x + 1; }`**" or "**`{ int x; bool z; z = (x+y)==10; }`**",
- The *while_stmt* and *if_stmt* in lines 14 and 18 receives only Boolean expression as *expr*.
- The *expr* in line 22 represents both algebraic expression and boolear expression, whose values can be **NUM**, **REAL**, `"true"`, or `"false"`. The *expr* has sub-rules.
  - Comparison operations (`"=="`, `"!="`, `"<"`, `">"`, `">="`, `"<="`) can be applied between algebraic expressions.
  - Boolean operations (`"or"`, `"and"`, `"not"`) can be applied between Boolean expressions.
  - algebraic operations (`"+"`, `"-"`, `"*"`, `"/"`, `"%"`) can be applied between integers and real numbers.
  - parenthesis (`"("` *expr* `")"`) can be applied to both boolean and algebraic expressions.
  - **ID** `"("` *args* `")"` represents the function call, which can return integer, real, or Boolean value. The function must be declared in *fun_decl*, and its argument types must match with parameter types.
  - All binary operations (`"or"`, `"and"`, `"=="`, `"!="`, `"<"`, `">"`, `">="`, `"<="`, `"+"`, `"-"`, `"*"`, `"/"`, `"%"`) are left associative, and the unary operation (`"not"`) is right associative.
  - Operators has the the following order of precedences:
    `"or"` (the lowest precedence)
    `"and"`
    `"not"`
    `"=="`, `"!="`, `"<"`, `">"`, `">="`, `"<="`
    `"+"`, `"-"`
    `"*"`, `"/"`, `"%"` (the highest precedence).

Note1: The ambiguity in *expr* in this grammar can be eliminated by defining rules (such as left/right associatives and precedences) in declaration part of Yacc file.

Note2: The startup program is composed of the following files and folders

```
•  src/Lexer.flex: flex source for limited number of tokens, which handle line counting
•  src/Parser.y             : yacc file. BYacc/J will translate this to Parser.java
•  src/Parser.grammar       : List of all grammars in text format
•  src/ParserBase.java      : superclass of Parser class, implementing functions to use in Parser
•  src/Env.java             : class for the chained symbol table
•  src/TestEnv.java         : Env tester class, which has 10 test cases
•  src/SemanticChecker.java : Semantic checker program
•  testcase/success         : list of foo program sources that do not have semantic error
•  testcase/success         : list of foo program sources that do have semantic error,
                              and their corresponding error messages in other text files
```

Here, `Parser.y` is designed to compile yacc using the following command:

```
yacc -Jthrows="Exception" -Jextends=ParserBase -Jnorun -J ./Parser.y
```