CS335 (Compiler Design) : Project Milestone 2

Deepak Sangle
200860
Department of Computer Science
sangleds20@iitk.ac.in

Shreyasi Mandal
200956
Department of Computer Science
shreyansi20@iitk.ac.in

Vartika
201089
Department of Computer Science
vartikag20@iitk.ac.in

JAVA17 Symbol Table and IR

We extended the JAVA 17 (combined) lexer and parser called **ASTGenerator**, to implement support for the symbol table data structure, perform semantic analysis to do limited error checking, and then convert the input source program into 3-address code (3AC)

Symbol Table

We used the symbol table structure for nested scope -

- A global symbol table containing relevant information about all the field variables, methods, global variables, and local variables inside functions
- Local symbol tables for each scope inside the global symbol table, for eg.
  - Methods
  - Loops (for, while)
  - Control-flow statements, etc.

We create .csv files for the symbol table of each function (with method name as the file name) and for the class as a whole (with class name used as the file name).

The columns include the name (method name, identifier name etc.), data type, syntactic category and line number.

Taking a look into a simple example -

For the following java source code,

```
class test {
    int x = 0;
    public void main() {
        int y = 5;
        y = x * 6 + x * 2 + x * 3;
        double y1 = 5.5;
    }
}
```

We get the following .csv files (symbol tables) :

Preprint. Under review.

```
+-------------------------------------------+
|                  test.csv                 |
+------+------+--------------------+--------+
| Name | Type | Syntactic Category | Line no |
+------+------+--------------------+--------+
|   x  |  int |   local_variable   |    2    |
|      |      |     declaration    |         |
+------+------+--------------------+--------+
| main | void |       method       |    8    |
|      |      |     declaration    |         |
+------+------+--------------------+--------+
```

```
+-------------------------------------------+
|                  main.csv                 |
+------+--------+--------------------+--------+
| Name |  Type  | Syntactic Category | Line no |
+------+--------+--------------------+--------+
|   y  |   int  |   local_variable   |    10   |
|      |        |     declaration    |         |
+------+--------+--------------------+--------+
|  y1  | double |   local variable   |    12   |
|      |        |     declaration    |         |
+------+--------+--------------------+--------+
```

Type Checking

Semantic Analysis was performed on expressions and access modifiers to do limited error checking. The errors include -

- Type mismatch: cannot convert from type[1] to type[2]
- The operator 'op' is undefined for the argument types type[1] and type[2]
- Incompatible types: cannot be converted to type accepted for 'op' operator
- Bad operand types for arithmetic operator
- Duplicate modifier found ...
- Modifier '...' not allowed in method/class declaration, etc.

3-Address Code

We are maintaining the 3AC in an in-memory data structure, which is done by post-order traversal on the Abstract Syntax Tree (AST).

The 3AC code generated by our system for the above source code is as follows -

```
0:    x = add_to_symtab()

main:
1:    y = add_to_symtab()
2:    t0 = x * 6
3:    t1 = x * 2
4:    t2 = t0 + t1
5:    t0 = x * 3
6:    t1 = t2 + t0
7:    y = t1
8:    y1 = add_to_symtab()
```

Compilation and Execution Instructions

The compilation instructions are as follows -

```
make
./ASTGenerator --input=<input_file_name>.java --output=<
    output_file_name>.dot
dot -Tsvg <output_file_name>.dot -o <svg_file_name>.svg
```

Note: We are generating **.svg** file to show the visualization of the **Abstract Syntax Tree** (AST). This was because we had no application to support viewing of PostScript (**.ps**) files.

We support the following command-line options -

- input : Takes the input java file (<input_file_name.java>)
- output : Denotes the output dot file (<output_file_name.dot>)
- verbose : This command prints logs for the compilation as shown below -

```
...
Entering state 411
Reducing stack by rule 115 (line 240):
   $1 = nterm unary_expression (1.1-1.1: )
-> $$ = nterm multiplicative_expression (1.1-1.1: )
Stack now 0 25 63 30 32 73 87 107 130 174 257 458 217 288 229
    336 331 527 713 879 405 600
Entering state 410
Next token is token ')' (1.1-1.1: )
Reducing stack by rule 112 (line 235):
   $1 = nterm multiplicative_expression (1.1-1.1: )
-> $$ = nterm additive_expression (1.1-1.1: )
Stack now 0 25 63 30 32 73 87 107 130 174 257 458 217 288 229
    336 331 527 713 879 405 600
...
```

- help : This command generates a brief description of AST Generator.

Tools Used

The following tools are used for creating **AST Generator**.

- Flex - This is used for the lexer implementation of the java program
- Bison - This is used to generate the parser
- Graphviz - Graph visualisation tool to visualize the AST

Language Features Supported

- Operators
- Control Flow - If-else, For, While
- Primitive data types
- Basic Operators
- Recursion
- Classes and Methods

Note

If the makefile does not compile properly, use the following command,

```
    g++ lex.yy.c parser.tab.c -o ASTGenerator
```

instead of

```
    g++ -std=c++17 lex.yy.c parser.tab.c -o ASTGenerator
```

in the Makefile