Assignment 2: Syntax analysis and Semantic analysis

The goals of this assignment include:

1. Use Bison to write a syntax specification for the given MiniJava grammar and generate the abstract syntax tree (AST).
2. Traverse the AST to construct the symbol table (ST).
3. Traverse the AST and use the ST to perform the semantic analysis.
4. Use the set of valid java programs to verify the correctness of your approach (http://www.cambridge.org/us/features/052182060X/ (Links to an external site)). Extend the set of programs with your own programs verify various aspects of the semantic analysis. Show that your compiler can produce correct error messages when something is wrong.

MiniJava Grammar

We use the same MiniJava grammar that we have seen in the first Assignment. For your convenience I have attached it below:

```
Goal              ::= MainClass ( ClassDeclaration )* <EOF>
MainClass         ::= "class" Identifier "{" "public" "static" "void" "main"
                      "(" "String" "[" "]" Identifier ")" "{" Statement "}" "}"
ClassDeclaration  ::= "class" Identifier ( "extends" Identifier )? "{"
                      ( VarDeclaration )* ( MethodDeclaration )* "}"
VarDeclaration    ::= Type Identifier ";"
MethodDeclaration ::= "public" Type Identifier "(" ( Type Identifier (
                      "," Type Identifier )* )? ")" "{" ( VarDeclaration )*
                      ( Statement )* "return" Expression ";" "}"
Type              ::= "int" "[" "]"
                  |   "boolean"
                  |   "int"
                  |   Identifier
Statement         ::= "{" ( Statement )* "}"
                  |   "if" "(" Expression ")" Statement "else" Statement
                  |   "while" "(" Expression ")" Statement
                  |   "System.out.println" "(" Expression ")" ";"
                  |   Identifier "=" Expression ";"
                  |   Identifier "[" Expression "]" "=" Expression ";"
Expression        ::= Expression ( "&&" | "<" | "+" | "-" | "*" ) Expression
                  |   Expression "[" Expression "]"
                  |   Expression "." "length"
                  |   Expression "." Identifier "(" ( Expression (
                      "," Expression )* )? ")"
                  |   <INTEGER_LITERAL>
                  |   "true"
                  |   "false"
                  |   Identifier
```

```
                   |   "this"
                   |   "new" "int" "[" Expression "]"
                   |   "new" Identifier "(" ")"
                   |   "!" Expression
                   |   "(" Expression ")"
Identifier         ::= <IDENTIFIER>
```

Step 1: Writing the grammar

Use the Bison tool to write the MiniJava grammar. Note that bison can automatically generate a parser for given a grammar. A short introduction to bison is attached in the “introduction_to_bison.pdf” file. An extended reference for the bison specification is provided in chapter 6 of the Flex and Bison book.

You need to consider the following restrictions to the MiniJava grammar:

1. The built-in function “`System.out.println(…)`” can only print integers.
2. The member “`.length`” can only be applied to integer arrays (`int [ ]`)
3. Overloading is not allowed: Example: `int a(int x, int z){…};` and `int a(int x){…}` is considered overloading. Showing an error that method a is already defined should be accepted.

The recommended approach:

1. Get familiarized with the getting started example provided as part of this assignment.
2. Replace the lexer file with the one that you did write in the first assignment. Note that the definition part of the lexer file now is slightly different. You might want to preserve these changes, because they enable Flex and Bison to work together where we use the lexer as a co-routine to return the sequence of tokens to be handled by bison (see page 4-8 of Flex and Bison book).
3. Add all the tokens that you have printed from the first assignment in the Bison file (parser.yy). An example of defining tokens is provided in the getting started example (parser.yy).
   a. We can define tokens as `%token <token_type> TOKENNAME`
   b. We can also define types for the production rules as `%type <type> productionRuleName`
4. Change the actions in the lexer, such that instead of printing the tokens you would return the corresponding tokens. E.g.,

```
0|[-]?[1-9[0-9]*    {return yy::parser::make_INT(yytext);}
```

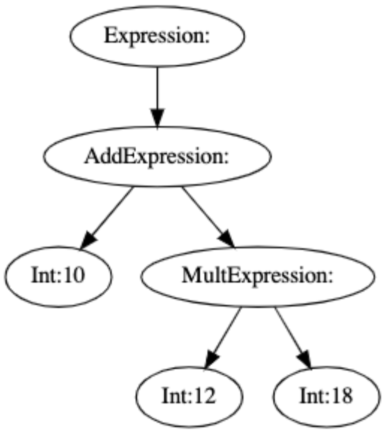
5. Take the MiniJava EBNF grammar and convert it into a left-recursive form
6. Put the grammar in Bison. It is recommended that you use a step-by-step approach, where you select a small feature of MiniJava and add the grammar for that. Test it, and if it works, then you may start adding another feature.

Step 2: Constructing the abstract syntax tree

As part of the getting started example, you may find a class “Node.h”, which is used to store the AST. This is a very simple class, it has a variable named *id* (which is mainly used for pretty printing the tree), *type* and *value* (which are used to store the information such as type of the node and the value of that node, for instance type Int: value 3; or type ID: value varName).

Note that the implementation of the node class is a single type of node, which basically stores the type of the node in the type string. The getting started example uses a single type of node to construct the AST. An OOP developer may prefer to have a class hierarchy of node types, such that the Node would be the base class and then have specific types of nodes for each language construct by simply deriving the base class. For example, we could have a specific node type for arithmetic expression, assignment statements, if and while statements, … Then in Bison (in the rule action part) you would call the corresponding constructor.

The node class has a list of other Nodes, that is used to store the children of this node. Note that the entry point in our grammar creates the root node, and then we iteratively generate other Nodes that represent subtrees each corresponding to a specific language construct. For example, the tree for a simple expression 10 + 12 * 18 should look like the figure below. In this case the root node is Expression, which has a child named AddExpression. The AddExpression has two children Int and MultExpression, and so on. This example depicts how we can construct a large AST by combining subtrees that correspond to subexpressions in our language.



The node class has two pre-implemented methods named “*print_tree()*” and “*generate_tree()*”, both corresponding to printing the AST. In this case, *print_tree()* prints the tree to the console, and *generate_tree()* generates a dot file that could be compiled using *graphviz* tool to generate a visual representation of the AST (use “`make tree`” command to generate the tree). If you do not have *graphviz* tool, then you can install it using this command: `apt-get install graphviz`.

For each feature of the MiniJava check that the tree is properly constructed. In specific cases, you might need to rewrite the grammar if the tree is not generated correctly. For instance, the operator precedence should be encoded in the grammar to generate the correct AST.

Step 3: Constructing the symbol table

Traverse the AST to construct the symbol table (ST). The ST is a data structure for storing information about the identifiers, such as the type and scope. Note that we have three types of identifiers: class identifiers, method identifiers, and variable identifiers. Each of those types represent the scope of such identifiers. It is common for an identifier to have different type or value on a different scope in the program.

So, the implementation of the symbol table needs to keep track of the current scope when analyzing our program. It needs to provide functionality to enter and exit scopes, add new identifiers to the symbol table, as well as functionality to lookup a given identifier.

Every time we start a new traversal of the AST, we need to reset the ST, hence we need to provide such functionality. For more implementation details on how we can construct a symbol table I suggest listening to the Semantic analysis lecture.

Recommended approach:

1. For each type of the identifiers (e.g., classes, methods, and variables) use a different type of records in the symbol table. A record represents an identifier in the symbol table. We therefore suggest a hierarchy of record types.
2. The symbol table should be able to deal with the scopes of the identifiers. For instance, the variables defined inside a class could be accessed from anywhere in the class, whereas identifiers defined inside a method could only be accessed from that method. Note that in MiniJava, we can only define variables at the beginning of a method, hence we do not consider the cases where we define new variables inside the body of the *if-else* and *while* statements (in such cases we would need a nested level of scopes).
3. Note that the symbol table can be constructed using a single left-to-right traversal of the AST.
4. The construction of the ST is concerned only with the declarations of identifiers (including variables, methods, and classes) in our program. The way those identifiers will be used is considered in the semantic analysis phase.
5. For debugging purposes, you should implement a method that can print the symbol table (the name and the type of each record in the symbol table).

Step 4: Semantic analysis

The semantic analysis verifies the semantic correctness of the program. The main task of the semantic analysis is the type checking, which basically checks if: (1) the identifiers are declared before they are used; (2) the type of a left-hand-side of an assignment statement is the same as the one on the right-hand-side; (3) the return type of a method is the same the type of the method declaration; (4) the number of method parameters and their types are the same according to the method declaration; and others.

The semantic analysis uses the information in the symbol table to perform the above listed tasks. The tasks are listed in such a way that the easiest ones are listed first, so it is recommended to start with the first one and continue in a step-by-step fashion.

It is expected that the semantic analysis to be implemented as a separate phase of the compiler, which means that you need to perform another traversal of the AST.

While the MiniJava grammar allows inheritance and polymorphism, in order to keep the assignment manageable within the given time frame the semantic analysis disregards inheritance and polymorphism. This means that the "TreeVisitor.java" test file in the set of valid test programs is expected to produce some semantic errors.

Recommended approach:

1. Each class should have a list of its methods
    a. Calling a function A.foo() from class B, may require to lookup for the method declarations in A, instead of looking up in the method declarations of the current class (in this case B).
2. Each method should have a list of its parameters
    a. The number of parameters and the type of parameters used inside a method call should correspond to the number and type of parameters in the method declaration
3. "This" reference requires to keep track for each scope to which class it belongs.
    a. Calling this.foo() from class A, means that foo should be a method inside A.
4. Implement the functionality that checks for undeclared identifiers. Start with the simplest cases, where we check if a variable of a method is declared inside the same class. Then extend the functionality to check for function calls outside the current class (as seen in the example in point 1)
5. Perform type checking analysis
    a. Expressions: the type of all terminals inside an expression should be the same. For example, a+b, both a and b should be int type. 10+false should report a semantic error.

b. Statements: check that the left-hand-side and right-hand-side of assignment statements are of the same type; check that the condition inside if-statements and while-statements is Boolean type; check that the type of the expression inside the print-statement is of type integer;

c. Method declaration: check that the return type of the method is in accordance with the declared return type.

d. Method calls: verify that the number of parameters and the type of parameters inside a method call matches the number and type of parameters of the method declaration.

e. Array access: check that the expression inside the int [ ] is of type integer; verify that the left hand side of the expression that has the .length member is an array of integer.

f. and more…

Testing

A set of valid test programs is provided by the Cambridge webpage. Use this set as a starting point. Extend this set of classes with your own test cases. A list of things you can use to test your compiler with is provided below:

Valid tests:

- What happens if we return a class type?
- What happens if we use a method call inside the condition expression of an if statement?
- What happens if we return an array of integers?
- What happens if variable foo is in both class A and B?
- Operator priorities? …

Invalid tests:

- What is we use a method call as a statement?
- What if we have a void method?
- Method/Variable/Class duplications?
- Method/Variable/Class undefined?
- Undefined types?
- int a; a.length; ?
- boolean a; int b; b = b*a; if(a && b) …?
- Int a[]; boolean b; a[b] ? …

Preparation for demonstration:

- Your compiler should be easily compiled using the Makefile.
- You should be able to explain briefly how you have dealt with specific issues. For example, “How did you handle operator precedence”? “Show me the code where you build the symbol table?” …
- Have a set of valid test classes and invalid test classes ready. During the demonstration, you should use those test classes to demonstrate that the compiler accepts the valid ones and rejects or shows error messages for invalid ones.
    - You are expected to show the AST visually (use the make tree command) for each of the test classes used.

Grading scheme:

Factors that influence the grade include:

- The functionality of the code.
- The correctness of the AST.
- The ability of your compiler to catch syntax and semantic errors.
- The ability to answer the questions during the demonstration.
- To get a pass grade (E),
    - you need to have a parser that can recognize all valid test classes and generate the corresponding AST; and
    - construct the symbol table.
- A semantic analysis that handles all of the listed semantic related issues (in step 4) will result with the maximum grade (A).
    - Partially implementing the features of the semantic analysis will result in a grade between D and B.