<u>AST Documentation</u>

To build the AST, we started by adding actions to our grammar in the cup file. To a large extent, we used the provided starter code, modifying it slightly as went. We started sequentially from the top of the grammar, and built our way down. An example of the smaller changes we made, was adding constructors to all of the classes. Most of the classes already had some required member variables declared. We assigned to those by passing in values through the constructors.

One of the bigger changes made, was including into each node a line number. We got this from the cup file by appeding "left" to a symbol. For example:

```
        program ::= scope:s                         // the main program
    {: RESULT = new Program((Scope)s, sleft); :};
```

here, sleft is the line number.

In order to type track, we added a "type" variable to the Expn class which is the parent class to all Expressions. We then have getters and setters to access and modify the variable. It defaults to null. The type can resolve to either an *IntegerType* or a *BooleanType*. We use *instanceof* and *getClass()* when we are trying to do checks using the type field.

In order to maintain the Symbol Table, we made links to our AST classes within the symbol table. This will be discussed in greater detail in the Symbol Table section of the documentation.

In order to deal with Major and Minor Scopes, We also added a *isMajor* boolean to our Scope class which gets set to false from the cup file when we parse a scope as a statement. For scopes caused by conditionals, we don't use the scope class explicitly, but handle the addScope as a minor scope in our Semantics class. This will also be touched on in a later section of the documentation.

Symbol Table Documentation

We chose Hash Table as our data structure for the symbol table. Using a hash table we could have fast lookup, inserts and deletes. Since we lookup a symbol table a lot having a fast lookup time was essential and played a big role in choosing our symbol table data structure. The Key for the symbol table hash map is a String which represents the name of the entry and the value is an Entry object defined below.

Each symbol table entry is an Entry class object. This object holds 5 pieces of information:
1) Kind: The kind of object (Array, Variable, Function, Procedure, etc)
2) Name: The name of the object.
3) Type: The class type of the object (Integer, Boolean).
4) AST: The AST class object itself.
5) Order Number: A number which uniquely identifies a variable in a scope.

Kind was implemented using an enum in the Entry class. This allowed us to conveniently use this information when needed.

Each entry contains an AST class which connects the symbol table entry to the class. This allows the symbol table to reference the AST class and find it when doing lookups. This is necessary because we need to have links from the symbol table to the AST classes so we can get information about the symbol table entries.

The order number is incremented by the symbol table. Each put operation sets the Entry objects order number to the symbol table order number and then increments the symbol table order number by 1. This ensures each Entry object will have a unique order number.

Each scope major or minor has a corresponding symbol table. As variables and functions are declared we add them using the put method and when we want to remove something such as a forward declaration we use the remove operation. We never perform an iterative search over the symbol table. We always use the put or remove operations therefore get optimal search time O(1) for lookups.

Semantic Analysis Documentation

The Semantics class is the class which holds all the information needed to perform semantic analysis. This class holds 3 important objects which are used throughout our code.

1) Scope Type Stack: This is a stack which holds the current type of scope. When we enter a new scope, we push in the type of that scope into the stack and when we exit the scope we pop it.

Each scope has a scope type. In the Semantics class we have an enum which has 6 different types of scopes:

1) Function
2) If
3) Procedure
4) Loop
5) Stmt
6) Program

When a scope is opened, it's class is determined by the AST class which called it. This helps us keep track of which scope we are in or which major scope we are in. This comes especially in handy when trying to figure out where each of result, return and exit statements occur.

2) Symbol Table Linked List: This is a linked list of symbol tables. Each element in the linked list holds a symbol table for a scope. The last element in the linked list holds the symbol table for the current scope.

Every time a new scope is opened a symbol table is created by the class opening the scope which is then added to this linked list. There exists a symbol table for each scope major or minor. We use this list to find the symbol table for the current major scope. Also we use it to do lookups in all the scopes incase a variable was defined in a parent scope.

3) Error List: This is another linked list, which holds the list of errors. Everytime we encounter an error we add it to this list. At the end of semantic analysis we iterate through the list and display all the errors.

We have our own SemanticError object. This object holds holds the string error message and the line number where this error occurred. If an error occurs during semantic analysis this SemanticError object is created with the corresponding message and line number. Which is then added to this list. At the end of semantic analysis in the Main.java class we iterate through this list and print each error message with line number to standard error.

The AST class from which all our nodes extend, was modified to take a line number into its constructor. Therefore, on creation, each AST node gets passed in a line number as described in the AST documentation.

The one and only semantic object gets created in the Main.java file under src/compiler. Calling the constructor for this class we create an object which creates the 3 objects above. Each class in the AST Node has a semanticCheck function. This function takes a semantic object and using the information in the object performs the necessary semantic analysis. In the Main.java class we have the root node (programAST). We call the semanticCheck function of this node to start off the semantic analysis process. This node will first call it's childrens semanticCheck function first. There is only one semantic object passed around through the AST objects therefore the scope definitions, error lists, and symbol table lists remain consistent throughout the program.

Several methods defined in the Semantics class to perform semantic analysis:

curScopeLookup - Looks through the current major scope and returns the symbol table entry corresponding to the name. If none found returns null.

allScopeLookup - Looks through all scopes starting from the current and moving up through parent scopes and returning the first symbol table entry that matches the name.

getCurrScopeType - Returns the current scope type.

getCurrMajorScope - Returns the current major scope.

getCurrScopeType - Adds the symbol table entry object to the current scope. Using String name as the key.

remove - Removes the first symbol table entry found starting from the current scope and moving up to parent scopes.

openScope - Pushes the ScopeType object on to the stack and adds the symbol table to the symbol table list.

closeScope - Pops the first element of the scope type stack and removes the last symbol table from the symbol table list.