Project Step 3 — Symbol Table

Your goal this is to process variable declarations and create Symbol Tables. A symbol table is a data structure that keeps information about non-keyword symbols that appear in source programs. Variables and function names are examples of such symbols. The symbols added to the symbol table will be used in many of the further phases of the compilation.

Previously, you didn't need token values since only token types are used by parser generator tools to guide parsing. But now your parser needs to get token values such as identifier names and string literals from your scanner. You also need to add *semantic actions* to create symbol table entries and add those to the symbol table.

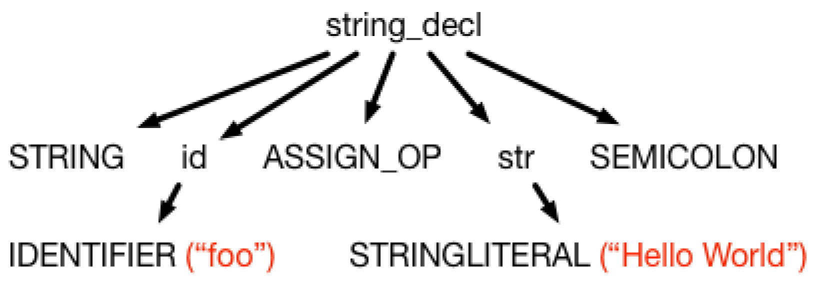
Semantic Actions

Semantic actions are steps that your compiler takes as the parser recognizes constructs in your program. Another way to think about this is that semantic actions are code that executes as your compiler matches various parts of the program (constructs like *variable declarations* or tokens like *identifiers*). By taking the right kind of action when the right kind of construct is recognized, you can make your compiler do useful work!

One way to think about when to take actions in your parser is to think about associating actions with nodes of your parse tree. Each element in the parse tree can take an action to generate a *semantic record* for that node in the parse tree; these actions can include looking at the semantic records generated for the children in the parse tree. For example, consider the parse tree for the following piece of code:

```
STRING foo := "Hello World";
```

Which produces the (partial) parse tree below:



We can create semantic records for each of the tokens `IDENTIFIER` and `STRINGLITERAL` that record their values ("foo" and "Hello World", respectively), and "pass them up" the tree so that those records are the records for `id` and `str`. We can then construct a semantic record for `string_decl` using the semantic records of its children to produce a structure that captures the necessary information for a string declaration entry in your symbol table (and even add the entry to the symbol table).

Parser-driven Semantic Actions

It may seem like a lot of work to keep track of pieces of data at each node of the parse tree and assemble them into more complicated records. But most parsers help do this *automatically*. As they build up the parse tree, they will call actions that execute to collect data from the parse tree and create semantic records. In essence, the parsers perform a *post-order* traversal of the parse tree *as they walk the tree*, and store information about the necessary semantic records in a way that you can easily retrieve them.

In ANTLR, you will *extend* the auto-generated *Listener* classes to implement sematic actions. It is worth remembering two things:

1. Tokens become leaves in the parse tree. The semantic record for a token is always the text associated with that token.
2. *Every* symbol that shows up in a grammar rule will be a node in your parse tree, and that if you recognize a grammar rule, there will be a node in your parse tree associated with the left-hand-side of the rule and that node will have a separate child for each of the symbols that appear on the right-hand side.

Symbol Tables

Your task is to construct symbol tables for each scope in your program. For each scope, construct a symbol table, then add entries to that symbol table as you see declarations. The declarations you have to handle are integer/float declarations, which should record the name and type of the variable, and string declarations, which should *additionally* record the value of the string. Note that typically function declarations/definitions would result in entries in the symbol table, too, but you are not required to record them.

Nested Symbol Tables

There are multiple *scopes* where variables can be declared:

- Variables can be declared before any functions. These are "global" variables and can be accessed from any function.
- Variables can be declared as part of a function's parameter list. These are "local" to the function and cannot be accessed by any other function.
- Variables can be declared at the beginning of a function body. These are "local" to the function as well.
- Variables can be declared at the beginning of a then block, an else block, or a repeat statement. These are "local" to the block itself. Other blocks, even in the same function, cannot access these variables.

Note that the scopes in the program are *nested* (function scopes are inside global scopes, and block scopes are nested inside function scopes, or each other). You will have to keep track of this nesting so that when a piece of code uses a variable named "x" you know which scope that variable is from.

What you need to do

You should define the necessary semantic actions and data structures to let you build the symbol table(s) for input programs.

At the end of the parsing phase, you should print out the symbols you found. For each symbol table in your program, use the following format:

```
Symbol table <scope_name>
name <var_name> type <type_name>
name <var_name> type <type_name> value <string_value>;
...
```

The global scope should be named "GLOBAL", function scopes should be given the same name as the function name, and block scopes should be called "BLOCK X" where X is a counter that increments every time you see a new block scope. *Function parameters should be included as part of the function scope.*

The order of declarations matters! We expect the entries in your symbol table to appear in the same order that they appear in the original program. Keep this in mind as you design the data structures to store your symbol tables.

See the sample outputs for more complete examples of what we are looking for.

CAVEAT You may be tempted to just print declarations as you see them, rather than building an actual symbol table. While that will suffice for passing some of the testcases, it *may not be sufficient* for passing all testcases. We *strongly suggest* that you build the necessary data structures.

Note: This step will be graded automatically, and the outputs generated by your code will be directly compared with the expected outputs using "diff -B -b -s" command. Please make sure your outputs are identical to the sample outputs provided.

Handling errors

Your compiler should output the string `DECLARATION ERROR <var_name>` if there are two declarations with the same name *in the same scope*.

How and where to implement actions?

We prohibit you to further edit the g4 file. It is okay to do minor modifications (especially if your code did not give 100% correct results for previous steps).

Specifically, what you cannot do is to add any application-specific code (e.g. java code) into the grammar file. In other words, embedding actions within grammar rules is prohibited. You should only use either *Listener* (or optionally *Visitor*) functionality for implementation. Otherwise, points may be deducted.

The reason for the above restriction is given below (excerpt is taken from the ANTLR book: https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference).

"To get reusable and retargetable grammars, we need to keep them completely clear of user-defined actions. This means putting all of the application-specific code into some kind of listener or visitor external to the grammar. Listeners and visitors operate on parse trees, and ANTLR automatically generates appropriate tree-walking interfaces and default implementations. Since the event method signatures are fixed and not application specific" this ensures that the same grammar can be used for many applications.

How to use ANTLR Listeners is given here:
https://github.com/antlr/antlr4/blob/master/doc/listeners.md

What you need to submit

- The ANTLR grammar dentition (.g4) you wrote for LITTLE. Make sure to name it **Little.g4**.
- Your source program that is the driver (i.e. one with the *main* method). Make sure to name it **Driver.java**. Section 2.5 and 3.4 of the textbook should be useful.
- Any other .java files that were not auto generated by ANTLR (meaning you wrote them)

- IMPORTANT: The grader will be using an executable called 'Micro' that takes care of running ANTLR, compiling source files, and running your code. On *osprey.unf.edu* server, the instructor will type './Micro.sh', followed by an input file name, on the terminal (as shown below), and have your code execute. Use chmod Unix command for setting the permission. **Submitting code that cannot be executed using the given Micro script may result in 50% penalty penalty.**

  ```
  $ ./Micro.sh sqrt.micro
  ```

What you will NOT submit

- The ANTLR jar file
- All files in 'step3_files' archive
- All files that were auto generated (i.e., you did not write them)