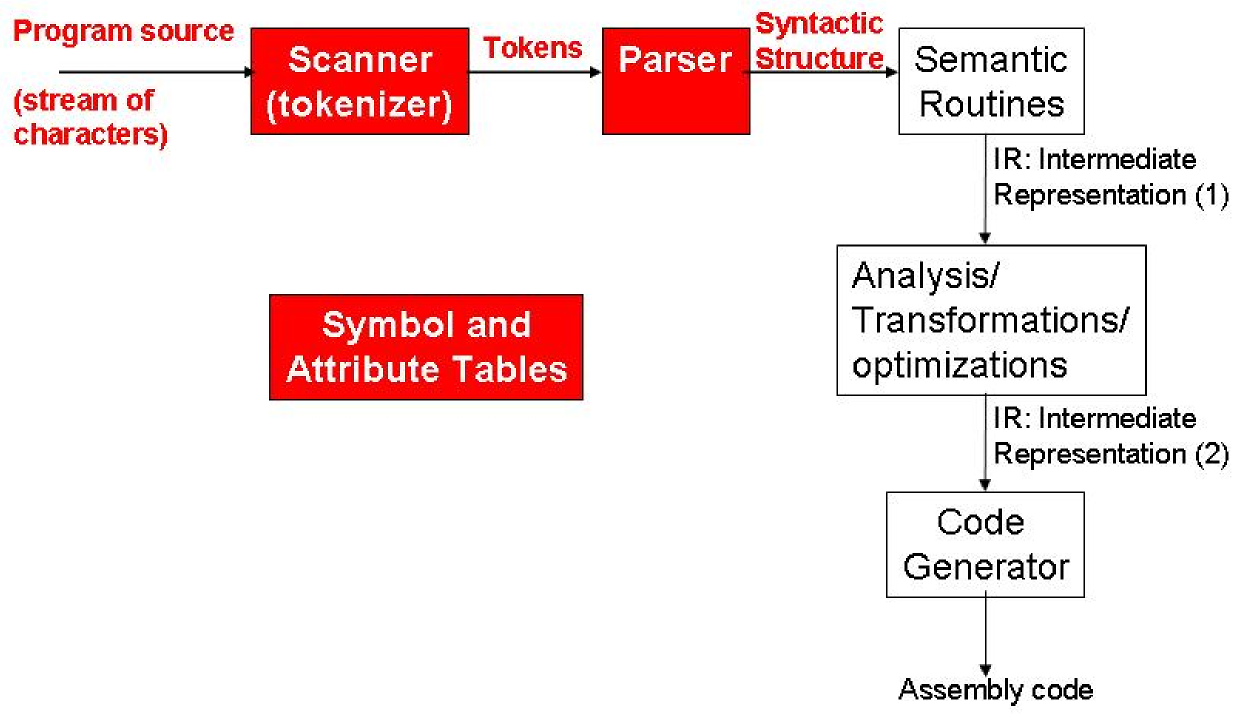
Step 3: Symbol Table Generation

Your goal in this step 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 **String Variables** are examples of such symbols. Other example of symbols kept by the symbol table are the names of functions or procedures. The symbols added to the symbol table will be used in many of the further phases of the compilation. The diagram below shows the progress in our compiler construction at the end of this step:


Program source
(stream of characters)
Scanner (tokenizer)
Tokens
Parser
Syntactic Structure
Semantic Routines
IR: Intermediate Representation (1)
Analysis/ Transformations/ optimizations
IR: Intermediate Representation (2)
Code Generator
Assembly code
Symbol and Attribute Tables

The Task: The Compiler's Symbol Table

In Step 2 you didn't need token values since only token types are used by parser generator tools to guide parsing. But in this step 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.

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.

Your task in this step of the project 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 do not have to record them for this step.

Nested symbol tables

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.

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*.

Output format

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! It is expected that the entries in your symbol table 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.

Testcases

Available in “Step2_files.zip” archive.

Readings

Several books introduce the theory necessary for a good understanding of lexical analysis and how a compiler tokenizer works, as well a good description of how a simple compiler is structured. You may find it helpful to check some of the following references:

1. Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers, Principles, Techniques*. Addison wesley.
2. Andrew, W. A., & Jens, P. (2002). *Modern compiler implementation in Java*.
3. Fischer, C. N., LeBlanc. RJ Jr. (1991). *Crafting a Compiler with C*
4. Levine, John R., Tony Mason, and Doug Brown. *Lex & yacc*. " O'Reilly Media, Inc.", 1992.