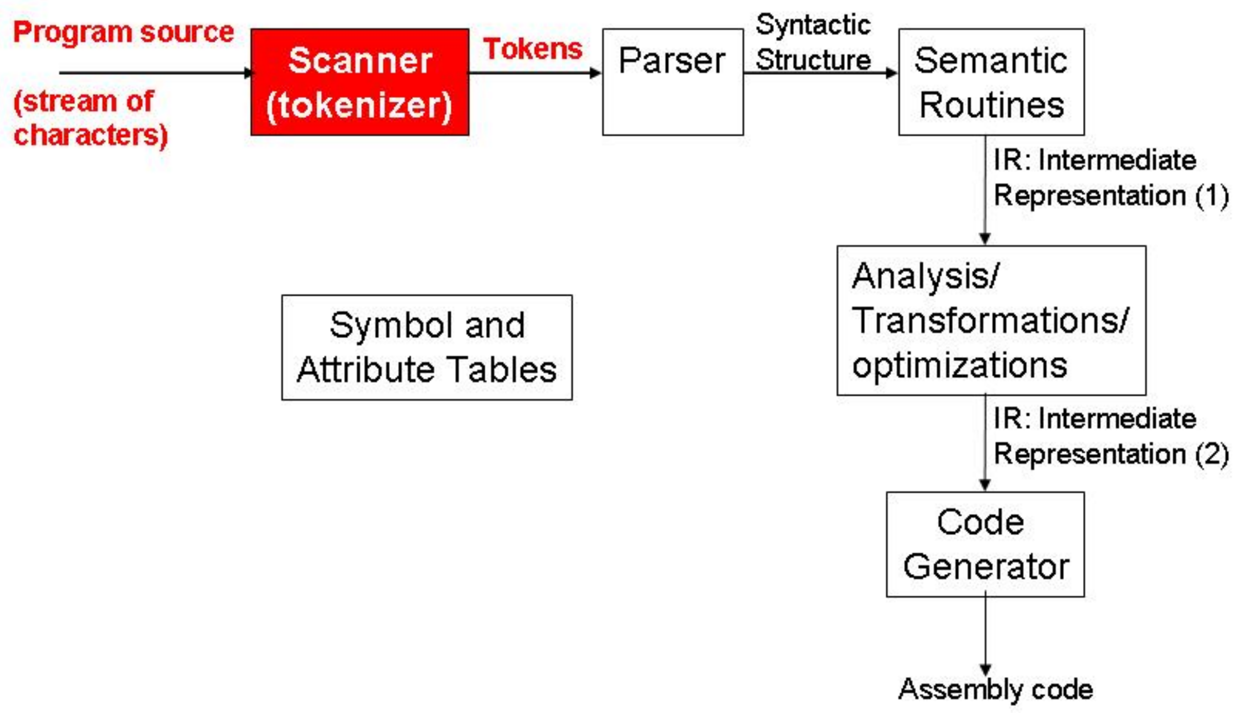
Step 1 - Scanner

You will develop a scanner (also known as a tokenizer) for the given grammar in this step. A scanner will take a sequence of characters (i.e. the source files of the LITTLE language) as input and produce a sequence of tokens which will be the input to the next step (Parser) as shown in the figure.


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

Scanner Generation Tools

The scanner's source code is normally generated by a scanner generation tool such as ANTLR. These tools normally work by taking the token definitions expressed by regular expressions and generates the source code for the scanner automatically (for this project, the tokens are specified in the language's grammer).

The programmer has to add the code to handle the scanner's output. For example, at this step you will print all the tokens in the standard output. In the step 2 of this project you will modify your scanner to feed the parser replacing the print routines by calls to the parser and passing the tokens as parameters. In order to learn how to merge the code of the scanner generator with the rest of your source code, please, read the user's manual of the tool you decide to use.

The compiler's tokenizer (scanner)

At the end of this step, the program developed (the very first part of our compiler) should be able to open a program's source file written in the LITTLE language and recognize its tokens. At this step, the output of your program should be the prints of each token's type and value (see below). In the next step the same output will be used to feed the parser.

Your scanner program should be able to open and read a LITTLE source file and print all the valid tokens within the source file and their respective type in the standard output. You might want to redirect the output to a file and compare your results with the output files provided for the testcases.

Testcases

Test inputs and the expected outputs are given in the Step1_files.zip archive given below. See grammar file for token definitions (Keywords, Operator, Literals).

Output Format

The program is expected to print each token in a predefined format. Here is an example

```
Token Type: INTLITERAL
Value: 20
```

Please read the output samples for more details.

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" command. Please make sure your outputs are identical to the sample outputs provided. Contact GTA if you have any questions.

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. (1992). Lex & yacc.

Q&A

- Q: What should I do if I encounter an illegal token?
- A: Stop the lexer at that point. You are not required to do error handling.