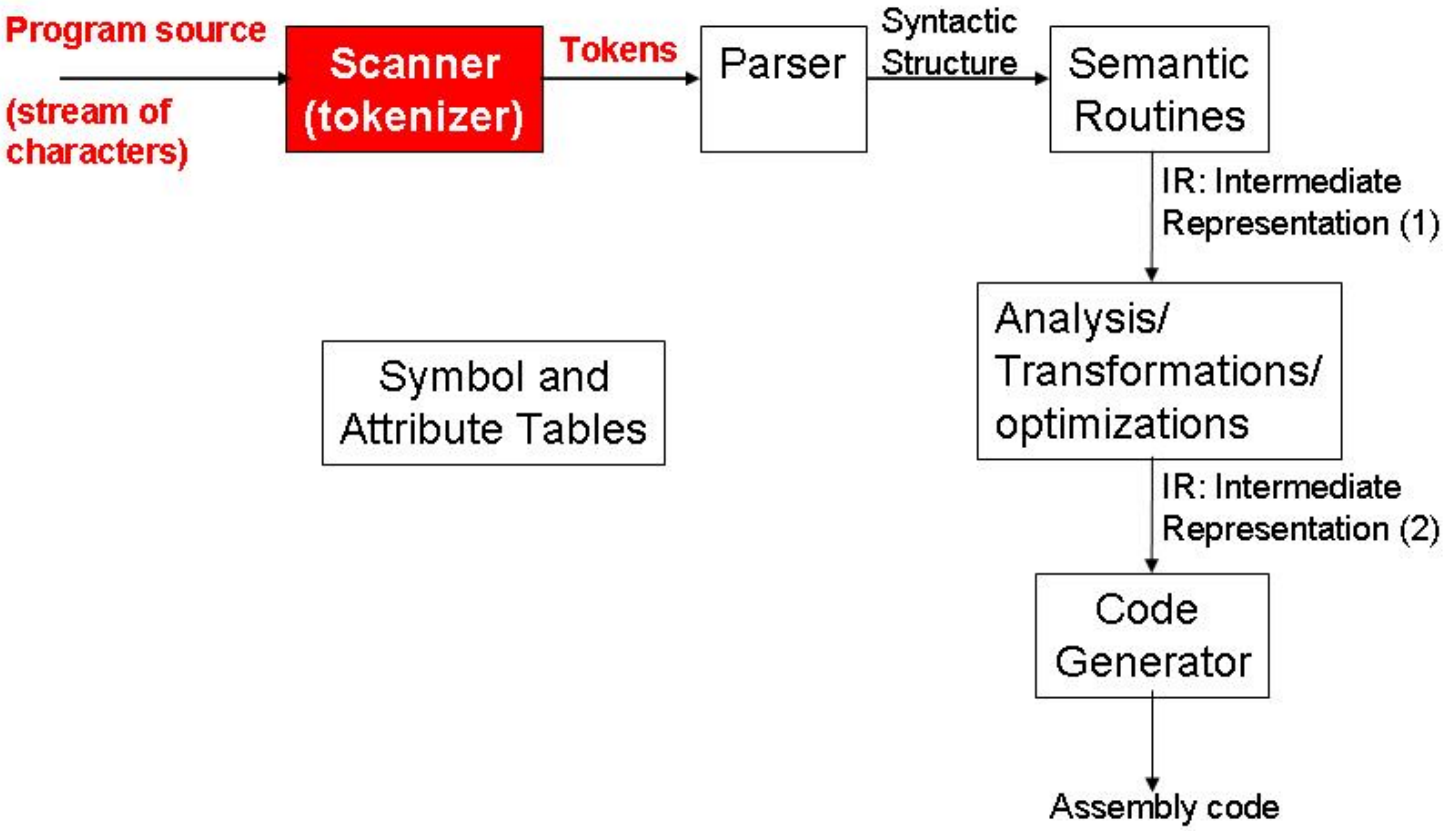
# Step 1: Scanner (tokenizer) -- Due: Friday, September 10th, 11:59pm

*Please read this handout carefully before submitting the project. Contact the TAs if you have any questions about the project. Also please read the newsgroup constantly for any updates in the project handout.*

## Introduction

Your task in this step is to generate a scanner (also referred as tokenizer) for the project's grammar provided to you. Scanner is a program that takes a sequence of characters (the source file of the program) and produces a sequence of tokens that will be used to feed the compiler's parser.

The figure below shows the general structure of a simple compiler similarly to the one presented in the Fischer & LeBlanc's book. The picture highlights the part of the compiler that will be developed at this step and conceptually relate it to other parts that will be developed in the future.




## Scanner Generation Tools

The scanner's source code is normally generated by a scanner generation tool. You can use one of the tools available to generate your scanner in this step. 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 especified in the language's grammar). 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 parserand 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.

Some popular tools are:

- Antlr: The most popular option if you're developing the compiler in Java. (Recommended)
- Lex (or Flex) : Generates the scanner's source in C language.

- JFlex(Scanner generator): Lexical analyzer written in Java.

# Step 1: 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 LITTLE source file (see the testcases:fibonacci.micro and sqrt.micro) and print the 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 (fibonacci.scanner, sqrt.scanner).

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.

## Running your scanner

This is how we will test your scanner:

```
java Micro fibonacci.micro > fibonacci.scanner
```

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`" command. Please make sure the your outputs are identical to the sample outputs provided: fibonacci.scanner, sqrt.scanner.

## Project template

To help you start the project we are providing templates for step1. We have template for ANTLR available.

# 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:

**Alfred v. Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 1986.**
*A simple one-pass compiler* - Chapter 2
*Lexical analysis* - Chapter 3

**Andrew W. Appel, "Modern Compiler Implementation in Java", Cambridge Univ. Press, 1998**.
*Lexical Analysis* - Chapter 2

**Charles Fischer, Richard LeBlanc, "Crafting a Compiler with C", The Benjamin/Commings Publishing Company, Inc, 1991.**
*A simple compiler* - Chapter 2
*Scanning - Theory and Practice* - Chapter 3

# 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.