Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2005 Handout 7 — Scanner-Parser Project Wednesday, September 7

DUE: Wednesday, September 21

This project consists of two segments: lexical (scanning) and syntactic (parsing) analysis.

Scanner

Your scanner must be able to identify tokens of the Decaf language, a simple imperative language we will be compiling in 6.035. The language is described in Handout 5. Your scanner should note illegal characters, missing quotation marks, and other lexical errors with reasonable and specific error messages. The scanner should find as many lexical errors as possible, and should be able to continue scanning after errors are found. The scanner should also filter out comments and whitespace.

You will not be writing the scanner from scratch. Instead, you will generate the scanner using JLex, a Java scanner generator from Princeton University. This program reads in an input specification of regular expressions and creates a Java program to scan the specified language. More information on JLex (including the manual and examples) can be on the web at:

```
http://www.cs.princeton.edu/~appel/modern/java/JLex/
```

To get you started, we have provided a template on the course server.

Copy this file to your working directory and fill in the appropriate regular expressions and actions. This template contains some example directives and macros, but you'll probably want to read the JLex manual for more information on creating a complete specification. In order to run JLex on this file, set your `CLASSPATH` environment variable as described in Handout 4 and execute the following command:

```
java JLex.Main scanner.lex
```

This will generate a file called `scanner.lex.java`. Note that JLex merely generates a Java source file for a scanner class; it does not compile or even syntactically check its output. Thus, any typos or syntactic errors in the lex file will be propagated to the output.

For the parser portion of the project, we will use the CUP parser generator (described later). The parser will expect tokens to be returned as type `java_cup.runtime.Symbol`. The `Symbol` class provides a basic abstract representation for tokens. It contains four basic fields for information about the token: These values are passed to the constructor for `Symbol` in this order. If you'd like to look at the class definition for `Symbol`, there's a copy of the CUP source on the course server.

`sym`	the symbol identifier (of type int)
`left`	the left position in the original input file
`right`	the right position in the original input file
`value`	the lexical value (of type Object)

Every distinguishable terminal in your Decaf grammar must have a unique integer associated with it so that the parser can differentiate them. These values are stored in the `sym` field of `Symbol`. When creating new symbols, you should use the symbolic values generated automatically from your decaf CUP specification (more on this later).

Some tokens have values associated with them. Examples include integer and string literals. These are stored in the `value` field of `Symbol`. Note that `value` is of type `java.lang.Object`, and thus can store a reference to any object of any type. For this part of the project, we will simply use it to store the attribute strings of some tokens, although it will be used to store more complex information in the future.

Notice that the `Symbol` class does not contain any information about the token's line number. Instead, it stores the values of the token's absolute position in the input file. There are two possible approaches you can take to generate error messages that include line numbers. The approach outlined in the textbook by Appel is to define a global `ErrorMsg` object that stores all the positions of the line breaks in the input file. This can then be used to calculate the row and column location of a given token. Should you find this approach distasteful, another way is to subclass `Symbol` with a class that stores more information about the token (i.e. the line number). There will be no differences in grading or credit for either approach.

Parser

Your parser must be able to correctly parse programs that conform to the grammar of the Decaf language. Any program that does not conform to the language grammar must be flagged with at least one error message.

As mentioned, we will be using the CUP LALR parser generator. The documentation for CUP is available at:

```
http://www.cs.princeton.edu/~appel/modern/java/CUP/
```

CUP is invoked as follows:

```
java java_cup.Main < parser.cup
```

This will produce two Java source files from the specification in `parser.cup`. The parser action code is contained in `parser.java`, and the symbol constant code is in `sym.java`. These can be overridden with the `-parser` and `-symbols` flags. `sym.java` contains integer constants corresponding to each terminal and non-terminal defined in your parser specification. These should be used when returning tokens from the scanner. An example of using JLex and CUP together is on the course server.

You will need to transform the reference grammar (along with the precedence rules) in Handout 5 into a grammar expressed in the BNF-like input syntax of CUP. A suitable grammar is an LALR(1) grammar that CUP can process with no conflicts.

Your parser should include basic syntactic error recovery, by adding extra error productions that use the error symbol in CUP. Do not attempt any context-sensitive error recovery. Keep it simple. We do not require error recovery to go beyond the ability to recover from a missing right brace or a missing semicolon.

Your parser does not have to, and should not, recognize context-sensitive errors *e.g.,* using an integer identifier where an array identifier is expected. Such errors will be detected by the static semantic checker. *Do not* try to detect context-sensitive errors in your parser.

You might want to look at Section 3 in the "Tiger" book, or Sections 4.3 and 4.8 in the Dragon book for tips on getting rid of shift/reduce and reduce/reduce conflicts from your grammar. Also, we have modified CUP to provide a String, `PRODSTRING`, which represents the current reduction. You may find this useful during debugging.

What to Hand In

Follow the directions given in Handout 3 when writing up the hard copy for your project. Include a full description of how your parser is organized.

For the electronic portion of the hand-in, provide a gzipped tar file named `leNN-parser.tar.gz` in your group locker, where `NN` is your group number. This file should contain all relevant source code and a `Makefile`. Additionally, you should provide a Java archive, produced with the `jar` tool, named `leNN-parser.jar` in the same directory. Unpacking the tar file and running `make` should produce the same Java archive. (All future hand-ins will be roughly identical, except with different names for the files.)

A sample makefile has been provided on the course server.

This can be copied to your working directory and used to help you create the correct tar and jar files. Also, you'll need to create a driver to run your scanner/parser. The driver should be called Compiler.class in the root of your class hierarchy, so you can run it with:

```
java Compiler <infile>
```

An example driver is available on the course server.

Your driver must parse the command line and run your parser or standalone scanner depending on what is requested. You should handle both `scan` and `parse` as arguments to the `-target` flag (assume `parse` by default). We invite you to use the provided CLI tool instead of writing your own command line parser. Documentation for CLI is linked from the 6.035 homepage.

Scanner format: When `-target scan` is specified, the output of your compiler should be a table with one row for each token in the input. Each row has three columns: the line number (starting at 1) on which the token appears, the kind of the token, and the token's value.

The kind of a simple token (such as a keyword or an operator) is simply the name of the token itself, e.g. `return` or `>=`. Tokens with values can be one of the following: `CHARLITERAL`, `INTLITERAL`, `BOOLEANLITERAL`, `STRINGLITERAL`, `IDENTIFIER`.

For `INTLITERAL`, the value is the series of digits *as it appeared in the source program* (see appendix for explanation). For `BOOLLITERAL`, the value can be `true` or `false`. For `STRINGLITERAL` and `CHARLITERAL`, this is the value of the literal *after processing the escape sequences*. For simple tokens, the value column is left blank. Note that character and string literals may contain newline characters. When these are printed as described, it may disrupt the table formatting — that's fine.

Each error message should be printed on its own line, *before* the erroneous token, if any. Such messages should be formatted as follows: `<filename>:<linenumber>: <message>`

Here is an example table corresponding to `print("Hello, World!");`:

LINE	KIND	VALUE
2	`IDENTIFIER`	`print`
2	`(`	
2	`STRINGLITERAL`	`Hello, World!`
2	`)`	
2	`;`	

Parser format: When `-target parse` is specified, any syntactically incorrect program should be flagged with at least one error message, and the program should exit with a non-zero value (see `System.exit()`). Multiple error messages should be printed for programs with multiple syntax errors that are amenable to error recovery (e.g. a missing right brace or a missing semicolon). Given a syntactically valid program, your parser should produce no output, and exit with the value zero (success).

Test Cases

The provided test cases for scanning and parsing can be found on the course server.

We will test your scanner/parser on these and a set of hidden tests. You will receive points depending on how many of these tests are passed successfully. In order to receive full credit, we also expect you to complete the written portion of the project as described in Handout 3.

A Why we defer integer range checking until the next project

When considering the problem of checking the legality of the input program, there is no fundamental separation between the responsibilities of the scanner, the parser and the semantic checker. Often, the compiler designer has the choice of checking a certain constraint in a particular phase, or even of dividing the checking across multiple phases. However, for pragmatic reasons, we have had to divide the complete scan/parse/check unit into two parts simply to fit the course schedule better.

As a result, we will have to mandate certain details about your implementations that are not necessarily issues of correctness. For example, one cannot completely check whether integer literals are within range without constructing a parse tree.

Consider the input:

```
x+-2147483648
```

This corresponds to a parse tree of:

```
  +
 / \
x   -
    |
INT(2147483648)
```

We cannot confirm in the scanner that the integer literal -2147483648 is within range, since it is not a single token. Nor can we do this within the parser, since at this stage we are not constructing an abstract syntax tree. Only in the semantic checking phase, when we have an AST, are we able to perform this check, since it requires the unary minus operator to modify its argument if it is an integer literal, as follows:

```
  +                        +
 / \                      / \
x   -          ---->     x   INT(-2147483648)
    |
INT(2147483648)
```

Of course, if the integer token was clearly out of range (e.g. 99999999999) the scanner could have rejected it, but this check is not required since the semantic phase will need to perform it later anyway.

Therefore, rather than do some checking earlier and some later, we have decided that ALL integer range checking must be deferred until the semantic phase. So, your scanner/parser must not try to interpret the strings of decimal or hex digits in an integer token; the token must simply retain the string until the semantic phase.

When printing out the token table from your scanner, do not print the value of an `INTLITERAL` token in decimal. Print it exactly as it appears in the source program, whether decimal or hex.