UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Fall 2006

Project #1: Lexer for Pyth

Due: Friday, 22 September 2006 at 2359

This first project calls for writing a lexical analyzer ("lexer") for our Python dialect (Pyth). You will use this lexer in the next project (a parser). In order to test your work, you'll also produce a test harness that uses your lexer to list the tokens constituting a a given program, using a format that is sufficiently rigid that we can specify a single "right" sequence of lexemes for any program.

Each team will have space in a Subversion repository that the staff will maintain. Handing in your project will amount to creating a Subversion "tag" for the files you want to hand in. We'll expect you to use the repository during development, frequently storing versions so that we can see how you're doing (and, of course, so you can get all the usual advantages of version-control systems).

You may implement your solution in either C++ or Java. You may use the regular-expression-parsing tools Flex (for C++) or JFlex (for Java), or you may write the whole thing "by hand," which we don't recommend (although it's a great way to waste an awful lot of time). In either case, you'll be using the result in the next project, so take care to get it right!

Your job is to hand in a program (the lexer and its testing harness), including adequate internal documentation (comments), and a thorough set of test cases, which we will run both against your program and everybody else's.

1 Your ultimate client

The parser you will write for Project #2 will expect to be able to extract certain information from your lexer. Specifically, it will repeatedly call some method or function (which, as you'll see, is usually named `yylex`), and expect it to return something representing a kind of token, and also to provide (by access to some variables or calls to some methods)

a lexical value of some kind (a string, very likely) giving the text of the lexeme, a line number indicating the line on which the token ends, and the name of the file containing the token.

Unfortunately, the tools we'll be using next, Bison and JBison, like to supply their own definitions for syntactic categories. It would be nice to integrate definitions. So—and also to save time and prevent the nature of this client from being a complete surprise to you—we will supply two dummy parser files: one for Bison (C/C++) and one for JBison (Java). When processed, these will provide definitions for the symbolic syntactic categories we describe below.

2 Running your solution

The program we'll be looking for when we test your submission is called `LexTest`. Our script will look to see whether the compilation process produces a file `LexTest.class` (indicating that we need to use the Java interpreter to run it) or `LexTest` (indicating that we don't). In either case, the argument list will be the same. For a C++ program, for example, we will expect that the command

`LexTest` *SEARCH-PATH* *FILE*$_1$`.py` *FILE*$_2$`.py ...`

will compile a program consisting of the *concatenation* of files *FILE*$_i$`.py` in order, using *SEARCH-PATH* as the list of directories in which to search for imported files (see the "import" command in the Pyth documentation). Following a Unix convention, the directories in *SEARCH-PATH* are separated by colons (:), as in

```
LexTest .:includeDir:lib/myLibraryDir myprog.py
```

A statement in your Pyth program such as

```
import math
```

will look for a file `math.py` first in the current directory (.), then in `includeDir`, and then `lib/myLibraryDir`, in that order, taking the first that it finds.

In case you're curious, we will use the ability to have your lexer handle a concatenation of files like this to prepend a *standard prelude* to all programs that run through the final compiler. This will be a set of Pyth definitions that define the standard types and their operations—a much more elegant and convenient method to introduce these definitions than somehow "hardwiring" them into your compiler.

3 Output

Your `LexTest` program should produce, on the standard output, the sequence of all tokens that your lexer finds, including their lexical values ("lexemes") and their location in the

source (what file are they from and what line). The rest of your compiler (when written) will be able to use the location information to produce error messages.

Since `LexTest` is just a testing framework, it is very important that you keep it separate from the lexer proper (a distinct class, for example). In particular, the lexer should *not* print the output we describe below! That should be `LexTest`'s duty alone, using an interface to the lexer that you design to extract the necessary information.

Suppose that the file `foo.py` contains the following text:

```
# This is a small test program

import bar

while i > 0:
   i -= 1
```

and that there is file called `lib/bar.py` containing

```
def prt ():
    print "A string with \a funny characters\nin it"
```

If you start `LexTest` with

`LexTest .:lib foo.py` *or* `java LexTest .:lib foo.py`

then the output should be as follows:

```
File: lib/bar.py
1: DEF
1: ID "prt"
1: '('
1: ')'
1: ':'
1: NEWLINE
2: INDENT
2: PRINT
2: STRING_LITERAL "A string with \007 funny characters\012in it"
2: NEWLINE
3: DEDENT
File: foo.py
5: WHILE
5: ID "i"
5: '>'
5: INTEGER_LITERAL "0"
5: ':'
5: NEWLINE
```

```
6: INDENT
6: ID "i"
6: MINUSEQ
6: INTEGER_LITERAL "1"
6: NEWLINE
7: DEDENT
END
```

As illustrated in this example, print each token preceded with its line number in the source file (first line is line 1). Before the first token and before a token that comes from a file different from the token before, print the file name in the format shown. For the end-of-input token, print just `END`. Print the kind (syntactic category) of the token either as a single-quoted character, for one-character punctuation, or as an all-upper-case word from the following set for other tokens:

AMPSNDEQ	&=	GLOBAL	global	MINUSEQ	-=
AND	and	GTEQ	>=	NE	!=
ARROW	->	GTGT	>>	NEWLINE	
BAREQ	|=	GTGTEQ	>>=	NOT	not
BREAK	break	ID		OR	or
CARATEQ	^=	IF	if	PASS	pass
CLASS	class	IMPORT	import	PERCENTEQ	%=
CONTINUE	continue	IN	in	PLUSEQ	+=
DEDENT		INDENT		PRINT	print
DEF	def	INTEGER_LITERAL		RETURN	return
ELIF	elif	IS	is	SLASHEQ	/=
ELSE	else	LTEQ	<=	STAREQ	*=
EQEQ	==	LTLT	<<	STARSTAR	**
FLOAT_LITERAL		LTLTEQ	<<=	STARSTAREQ	**=
FOR	for			STRING_LITERAL	
				WHILE	while

As in the example, the lexer should handle outer-level **import** statements (the lexer, not the `LexTest` driver, because you'll need this capability in later projects).

Finally, print the lexical values of tokens where it is significant: in identifiers (ID) and literals. Translate strings into the illustrated canonical form: all characters greater than or equal to blank in the character set are printed as is, and the backslash character itself (ASCII code 92) and the control characters before blank (with codes less than 32), are printed as three-digit octal escape sequences as in C, C++, and Java (with no other escape sequences used). You'll be using this format for communicating string constants to later parts of the program, so it's a good idea to write the method that does it in such a way that later parts of the compiler can use it. Output integer literals in decimal (as longs, so they are all non-negative), and use `%.16e` format for floating-point literals.

Since we'll be doing literal comparisons to test your output, please adhere to this format exactly. Each DEDENT and INDENT takes the line number of the more- or less-indented

line (at the end of a file, this will be the imaginary line *after* the last line of the file). Comments, blank lines, and whitespace are removed entirely from the output.

Your lexer should detect and report lexical errors (the lexer, not just the `LexTest` driver, because you'll need this in future projects):

- Singly quoted strings that aren't complete by the end of the line;
- Triply quoted strings that aren't complete by the end of the file that contains them;
- Integer constants that are too large;
- Characters that cannot be interpreted as tokens (e.g., '!').
- **import** statements that refer to non-existent files.
- Any use of reserved words (such as **assert**) that are not used in Pyth, but are not allowed as identifiers (see the list of keywords in the Pyth document).
- Inconsistent indentation.

In each case, print an error message in standard form on the standard error output, e.g.,

```
foo.py:5: integer constant too large.
```

Also arrange that if the lexer detects any errors, the program as a whole exits with a non-zero exit code when processing is complete. Your program should always recover from errors by simply printing the message, throwing away erroneous text (which can be quite a bit in the case of unterminated strings) and trying to continue as helpfully as possible.

4 What to Turn In

The directory you turn in (see §5) should contain a file `Makefile` that is set up so that

```
gmake
```

(the default target) compiles your program and

```
gmake check
```

runs all your tests against your program. We'll put sample Makefiles (for C++ and Java) in `~cs164/hw/proj1` directory; feel free to modify at will as long as these two commands continue to work.

Because we want to run your tests against everyone else's program, we'd like you to adhere to a standard format. In the directory you submit, have a subdirectory called `lexer-tests`. Under that, have two subdirectories full of `.py` files: `lexer-tests/correct` and `lexer-tests/errors`. For each file `lexer-tests/correct/`*foo*`.py`, have another file `lexer-tests/correct/`*foo*`.py.out`, with the output that LexTest is supposed to produce. The first set of tests will be: if we run `LexTest` with arguments

`lexer-tests/correct:lexer-tests lexer-tests/correct/`*foo*`.py`

does it succeed (exit normally), print nothing on the standard error output, and produce the same output as in the corresponding `.out` file? The second set of tests will be: if we run `LexTest` with arguments

`lexer-tests/errors:lexer-tests lexer-tests/errors/`*foo*`.py`

will we get at least one error message on the standard output and will the program exit with exit code 1 (the usual way to indicate a compilation error)?

Not only must your program work, but it must also be well documented internally. At the very least, we want to see *useful and informative* comments on each method you introduce and each class.

5 How to Submit

We've set up a Subversion repository for your team, initially containing a directory `project` with the conventional subdirectories `trunk`, `branches`, and `tags` under it, as described in the on-line Subversion book. The general idea is that you keep the latest (HEAD) version of your project in the `trunk` directory, and make "cheap" copies of significant versions of the trunk in the `tags` directory. The staff can see everything you check in. Anything you put in the `tags` directory with a name of the form `release-1.`N we will treat as a submission (the '1' is the project number); and the one with the highest N we will treat as your latest "official" submission.

Let's assume that you have checked in a satisfactory working version of your project to the `trunk` subdirectory, and want to submit it. The command

```
svn copy svn+ssh://cs164-tb@host/myteam/project/trunk \
         svn+ssh://cs164-tb@host/myteam/project/tags/release-1.1
       -m "First submitted version of project 1"
```

does the job. Here, *myteam* is your team's name, and *host* is on of the instructional servers (e.g., nova.cs.berkeley.edu). Alternatively, from within the working directory that contains checked-out versions of the `trunk` and `tags` subdirectories, you can issue the two commands:

```
svn copy trunk tags/release-1.1
svn commit -m "First submitted version of project 1"
```

and get the same effect.

Submit early and often (at least up to the deadline). Don't worry about using up file space with lots of submissions. Subversion does not actually copy your files; it just makes notations that tell it that they're the same files as in version such-and-such of the trunk.

You can even delete a submission, with a command like

```
svn delete svn+ssh://cs164-tb@host/myteam/project/tags/release-1.1 \
       -m "Remove bogus submission"
```

Inevitably, the moment you hit return on this command, you'll realize that you meant to delete `release-2` instead. No worries: if `release-1` existed in revision 125, say, then you can recreate it with

```
svn copy --revision 125 \
      svn+ssh://cs164-tb@host/myteam/project/tags/release-1.1 \
      tags/release-1
svn commit -m "Recreated release 1"
```

6 Assorted Advice

First, get started as soon as possible. Second, don't *ever* waste time beating your head against a wall. If you come to an impasse, recognize it quickly and come see one of us or, if we are not immediately available, work on something else for a while (you can never have enough test cases, for example). Third, keep track of your partner. If possible, schedule time to do most of your work together. I've seen all too many instances of the Case of the Flaky Partner.

Learn your tools. You should be doing all of your compilations using `gmake`, Eclipse, or some other IDE. Get to know this tool and try to understand the "makefiles" we give you, even if you don't use them. These tools really do make life much easier for you. Learn to use the `gdb` and `gjdb` debuggers (also usable from within Emacs), or the equivalent in Eclipse or your favorite IDE. In most cases, if your C++ program blows up, you should be able to at least tell me *where*, it blew up (even if the error that caused it is elsewhere). I do not look kindly on those who do not at least make that effort before consulting me. Use your Subversion repository to coordinate with your partner and to save development versions *frequently.*