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

CS 164 **P. N. Hilfinger**
Spring 2008

Project #1: Lexer and Parser for Pyth

Due: Friday, 29 February 2008 at 2400

Our first project is to write a lexer and parser for Pyth. This parser will take a source file and produce an abstract syntax tree (AST) that it will output as text to be read back by the next stage of the compiler.

Your team has space in the class Subversion repository that the staff will maintain, and you will eventually hand in your project by creating a tag for it. Again, 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 parsing tools FLEX and BISON (for C++) or JFLEX and JBISON (for Java), or you may write the whole thing "by hand," as a recursive-descent compiler.

Your job is to hand in a program (the parser 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 Running your solution

The program we'll be looking for when we test your submission is called `ParseTest`. Our script will look to see whether the compilation process produces a file `ParseTest.class` (indicating that we need to use the Java interpreter to run it) or `ParseTest` (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

`./ParseTest` *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

```
ParseTest .: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.

2 Output

Your `ParseTest` program should produce, on the standard output, a representation of the corresponding AST, using the abstract syntax and format given below. Actually, this is the same output that we will use as input to the next stage of the compiler, so I expect `ParseTest` to be extremely simple. We are communicating information between phases in this fashion, by the way, rather than using something more efficient (like a shared data structure between compiler phases) in order to make it easy both to look at the output from the parser in isolation and to glue your parser together with any implementation of later compiler phases, regardless of the languages in which the two are written.

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

```
# This is a small test program
import defns
if i:
   print i, s, t
else:
   pass
```

and `lib/defns.py` contains

```
i = 3
s = i + 2; t = s ** 2
```

Then the command

```
ParseTest .:lib foo.py   or  java ParseTest .:lib foo.py
```

should produce the following output on the standard output:

```
(program 0 0
   [(assign 1 1 (id 1 1 "i") (int 1 1 "3"))
    [(assign 1 2 (id 1 2 "s")
        (call 1 2 (id 0 0 "__add__")
          [(id 1 2 "i") (int 1 2 "2")]))
     (assign 1 2 (id 1 2 "t")
        (call 1 2 (id 0 0 "__pow__")
          [(id 1 2 "s") (int 1 2 "2")]))]
    (if 0 3 (id 0 3 "i")
        [(print_newline 0 4 (id "None") [(id 0 4 "i") (id 0 4 "s") (id 0 4 "t")])]
        [])
   ]
   [(file 0 0 "foo.py") (file 0 0 "lib/defns.py")])
```

That is, this is all in Lisp-like notation. Parenthesized items represent tree nodes. Each node has the form

(*operator file-number line-number* $operand_1 \cdots operand_n$)

The *file-number* identifies which source file contained the text that translated into this node, and *line-number* identifies the line in that file. The *operands* are either tree nodes, quoted strings, or lists of tree nodes. Square-bracketed items represent lists of tree nodes. Quoted strings will use four-character octal escape sequences in place of all double quotes (`\042`), backslashes (`\134`), and all characters with ASCII codes less than 32 (`\000`–`\037`). They will not contain any other escape sequences. Thus, what appears in a program as

```
"Input file: C:\\FOO contains\t\"Hello, world!\"\n"
```

gets written out as

```
"Input file: C:\134FOO contains\011\042Hello, world!\042\012"
```

The complete output consists of a list of program statements followed by a list of file names, with file 0 first, then file 1, etc.

In fact, you have considerable latitude in laying this out. We will test the trees you output by running them through an "unparser" that we will supply, which will try to reconstruct an approximation of the original program. This is not a perfect test by any means, but we think it might make it easier for you to see errors. Whitespace (blanks and newlines) before and after parentheses, square brackets, and quoted items is optional, and any non-empty amount of whitespace may separate identifiers such as "`assign`" and source position numbers from each other.

In general, the line number to associate with a construct is the line number of the token that starts it. We are not going to be terribly fussy about this, but your line number should be reasonable. When identifiers are introduced by the parser that are not represented in the source text, it assigns them file 0 and line 0 as shown.

Your parser should detect and report syntax errors (on the standard error output) using the standard Unix format:

```
foo.py:5: syntax error
```

Also arrange that if the parser (or 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 some erroneous program text (which can be quite a bit in the case of unterminated strings) and trying to continue as helpfully as possible. However, the precise tree you produce in the presence of syntax or lexical errors is irrelevant.

In general, you will want the lexer part of your project to catch malformed tokens, while the parser catches malformed combinations of tokens. Lexical errors include:

- 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.

3 Abstract Syntax Trees

The abstract syntax operators to be output by your parser are as given in Table 1 (expressions), Table 2 (statements), and Table 3 (definitions and types). In each case, we show just the operator and operands, eliding the source-position numbers for brevity. For a language construct c, the notation $\hat{c}$ means "the AST that c translates to."

The program as a whole is to be represented as the tree

```
(program 0 0
  [ statements ]
  [ (file 0 0 "filename0") (file 0 0 "filename1") ... ])
```

where *statements* are the ASTs that the outer-level statements of the program translate to and the *filenames* are the names of file 0, file 1, etc., as used in the position information.

Note #1: Statement lists. For convenience in translation, you may freely group multiple statements into lists of single statements wherever convenient. For example, feel free to translate the "then" part of

```
if x > 0:
   Statement1; Statement 2; Statement 3;
   Statement4
```

into any of the lists

```
[ Tree1 Tree2 Tree3 Tree4 ],
[ [ Tree1 Tree2 Tree3 ] Tree4 ], or even
[ Tree1 [ Tree2 Tree3 ] [ Tree4 ] ]
```

where `Tree`i is the AST for `Statement`i.

Note #2: Pass statement. Likewise, feel free to eliminate extraneous **pass** nodes (which translate to empty lists):

```
pass; Statement1
pass
Statement2
```

Can translate to just

```
[ Tree1 Tree2 ]
```

This simplification is not obligatory (it's not even worth extra credit!).

Note #3: Complex assignment. Table 1 only shows the translation for simple assignments with one item on the left-hand side. Translate more complex left-hand sides as follows. The assignment

$L_0,\ L_1,\ \ldots, L_{n-1}$ `=` E `or` `(`$L_0,\ L_1,\ \ldots$`)` `=` E

translates to the tree

`(assigns [`$\hat{A}_0\ \hat{A}_1\ \cdots\ \hat{A}_{n-1}$`]` $\hat{E}$`)`

where $\hat{A}_i$ is the translation of A_i `=` RHS_i and RHS_i translates to `(getrhs (int` i`))`. E.g.,

```
x, a[0] = L ⟹  (assigns [ (assign (id "x") (getrhs (int "0")))
                           (call (id "__setitem__") [ (id "a") (int "0")
                                                      (getrhs (int "1")) ]) ]
                        (id "L"))
```

We suggest that you handle assignments by parsing them as if the left-hand sides could be any expressions, and then have a separate function specifically intended to transform them into the proper forms (and also to flag errors such as `x+3=4`).

Table 1: AST nodes for expressions, part 1.

Construct	AST	Notes
identifier	`(id "`*identifier*`")`	
integer literal	`(int "`n`")`	where n is the decimal representation of the integer literal (non-negative).
float literal	`(float "`x`")`	where x is the source text of the literal.
"string literal"	`(string "`*string literal*`")`	
$(e_1, \ldots, e_n)$	`(tuple [`$\hat{e}_1 \ \cdots \ \hat{e}_n$`])`	Includes $(e,)$ but not (e). Also includes tuple lists without parentheses, as in the right side of "`x = 1,2,3`".
$[e_1, \ldots, e_n]$	`(list [`$\hat{e}_1 \ \cdots \ \hat{e}_n$`])`	
$\{k_1 : v_1, \ \ldots, \ k_n : v_n\}$	`(dict [(pair `$\hat{k}_1 \ \hat{v}_1$`)...` `(pair `$\hat{k}_n \ \hat{v}_n$`)])`	
$F(e_1, \ldots, e_n)$	(call $\hat{F}$ [$\hat{e}_1 \ \cdots \ \hat{e}_n$])	
$e.i$	`(select (id "`i`") `$\hat{e}$`)`	
$e_1[e_2]$	`(call (id "__getitem__")` `[`$\hat{e}_1 \ \hat{e}_2$`])`	
$e_1[e_2 : e_3]$	`(call (id "__getslice__")` `[`$\hat{e}_1 \ \hat{e}_2 \ \hat{e}_3$`])`	
$e_1[e_2 :]$	`(call (id "__getslice__")` `[`$\hat{e}_1 \ \hat{e}_2 \ MAX$`])`	where MAX is the maximum integer.
`not` e	`(not `$\hat{e}$`)`	
e_1 `and` e_2	`(and `$\hat{e}_1 \ \hat{e}_2$`)`	
e_1 `or` e_2	`(or `$\hat{e}_1 \ \hat{e}_2$`)`	

Table 1: AST nodes for expressions, part 2.

Construct	**AST**	**Notes**
e_1 `in` e_2	`(call (id "__contains__")` `[`$\hat{e}_2$ $\hat{e}_1$`])`	
e_1 `not in` e_2	`(not` $\hat{e}$`)`	where $\hat{e}$ is the translation of e_1 `in` e_2
e_1 `is` e_2	`(is` $\hat{e}_1$ $\hat{e}_2$`)`	
e_1 `is not` e_2	`(not (is` $\hat{e}_1$ $\hat{e}_2$`))`	
$e_1 \prec e_2$	`(compare (id "`$\hat{\prec}$`") [`$\hat{e}_1$ $\hat{e}_2$`])`	$\prec$ is one of the comparison operators $<$, $>$, `<=`, `>=`, `==`, `!=`. $\hat{\prec}$ is the function-call equivalent according to Table 3 in the Pyth manual.
$e_1 \oplus e_2$	`(call (id "`$\hat{\oplus}$`") [`$\hat{e}_1$ $\hat{e}_2$`])`	$\oplus$ is a binary operator other than one handled above. $\hat{\oplus}$ is the function-call equivalent according to Table 3 in the Pyth manual.
$\oplus e$	`(call (id "`$\hat{\oplus}$`") [`$\hat{e}$`])`	$\oplus$ is a unary operator other than one handled above. $\hat{\oplus}$ is the function-call equivalent according to Table 3 in the Pyth manual.

Table 2: AST nodes for statements

Construct	AST	Notes
s_1`;`$\ldots s_n$	$[\hat{s}_1 \cdots \hat{s}_n]$	Statement lists include statements separated by explicit semicolons, as well as those separated by newlines. That is, $s_1; s_2$ has the same encoding as s_1 <`newline`> s_2. See Note #1.
`pass`	`[]`	See Note #2.
v_1 `=` e_2	`(assign` $\hat{v}_1$ $\hat{e}_2$`)`	where v_1 is a simple variable. e_2 may be an expression or other assignment. See Note #3.
e_1`.`i `=` e_2	`(setselect (id "`i`")` e_1 e_2`)`	
e_1`[`e_2`]` `=` e_3	`(call (id "__setitem__")` `[`$\hat{e}_1$ $\hat{e}_2$ $\hat{e}_3$`])`	
e_1`[`e_2`:`e_3`]` `=` e_4	`(call (id "__setslice__")` `[`$\hat{e}_1$ $\hat{e}_2$ $\hat{e}_3$ $\hat{e}_4$`])`	
e_1`[`e_2`:]` `=` e_3	`(call (id "__setslice__")` `[`$\hat{e}_1$ $\hat{e}_2$ MAX $\hat{e}_3$`])`	
v_1 $\oplus$`=` e_2	`(assign` $\hat{v}_1$ $\hat{e}_r$`)`	$\hat{e}_r$ is the translation of $v_1 \oplus e_2$. $\oplus$ is one of the arithmetic operators.
`return` e	`(return` $\hat{e}$`)`	
`return`	`(return (id "None"))`	
`break`	`(break)`	
`continue`	`(continue)`	
`print` $e_1, \ldots, e_n$	`(print_newline (id "None")` `[`$\hat{e}_1 \cdots \hat{e}_n$`])`	
`print` $e_1, \ldots, e_n$`,`	`(print (id "None")` `[`$\hat{e}_1 \cdots \hat{e}_n$`])`	
`print >>F,` $e_1, \ldots, e_n$	`(print_newline` $\hat{F}$ `[`$\hat{e}_1 \cdots \hat{e}_n$`])`	where F is an expression yielding a file object.
`print >>F,` $e_1, \ldots, e_n$`,`	`(print` $\hat{F}$ `[`$\hat{e}_1 \cdots \hat{e}_n$`])`	
`if` e`:` S_1 `else:` S_2	`(if` $\hat{e}$ $\hat{S}_1$ $\hat{S}_2$`)`	S_1 and S_2 are statement lists.
`if` e`:` S_1	`(if` $\hat{e}$ $\hat{S}_1$ `[])`	
`while` e`:` S_1 `else:` S_2	`(while` $\hat{e}$ $\hat{S}_1$ $\hat{S}_2$`)`	
`while` e`:` S_1	`(while` $\hat{e}$ $\hat{S}_1$ `[])`	
`for` v `in` e `:` S_1 `else:` S_2	(for $\hat{v}$ $\hat{e}$ $\hat{S}_1$ $\hat{S}_2$)	v is an identifier.
`for` v `in` e `:` S_1	`(for` $\hat{v}$ $\hat{e}$ $\hat{S}_1$ `[])`	

Table 3: AST nodes for declarations and types

Construct	AST	Notes
`def` i `=` e	`(defconst (id "`i`") `$\hat{e}$`)`	
`def` i_0 `(`$i_1, \ldots, i_n$`):` S	`(defun (id "`i_0`") [(id "`i_1`") ...(id `i_n`)]` $\hat{S}$`)`	
`class def` i_0 `(`$i_1, \ldots, i_n$`):` S	`(class_defun (id "`i_0`") [(id "`i_1`") ...(id `i_n`)]` $\hat{S}$`)`	
`def` i_0 `(`$i_1, \ldots, i_n$`):` `import "`$name$`"`	`(defun_native (id "`i_0`") [(id "`i_1`") ...(id `i_n`)] (string "`$name$`"))`	
`class def` i_0 `(`$i_1, \ldots, i_n$`):` `import "`$name$`"`	`(class_defun_native (id "`i_0`") [(id "`i_1`") ...(id `i_n`)] (string "`$name$`"))`	
`class` i`(`t`):` S	`(class (id "`i`") (id "`t`") `$\hat{S}$`)`	
`global` $i_1, \ldots, i_n$	`(global [(id "`i_1`")...(id "`i_n`")])`	
i `:` T	`(type_decl (id "`i`") `$\hat{T}$`)`	Where $\hat{T}$ is an encoded type, as described in the following entries.
i (as a type)	(type (id "i"))	
`(`$t_1, \ldots, t_n$`) -> `t_0	`(func_type [`$\hat{t}_1 \cdots \hat{t}_n$`] `$\hat{t}_0$`)`	

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 the `~cs164/hw/proj1` directory and the staff project 1 repository:

```
svn+ssh://cs164-tj@nova.cs.berkeley.edu/staff/proj1
```

Feel free to modify at will as long as these two `gmake` commands continue to work on the instructional machines.

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 `parser-tests`. Under that, have two subdirectories full of `.py` files: `parser-tests/correct` and `parser-tests/errors`. For each file `parser-tests/correct/`*foo*`.py`, have another file `parser-tests/correct/`*foo*`.py.out`, with the output that ParseTest is supposed to produce. The first set of tests will be: if we run `ParseTest` with arguments

```
parser-tests/correct:parser-tests parser-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 (modulo whitespace, as discussed above). The second set of tests will be: if we run `ParseTest` with arguments

```
parser-tests/errors:parser-tests parser-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 just the subdirectory `tags` under it. The usual practice is to create a `trunk` directory in which you keep the latest "wavefront" version of your project files, 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 `proj1-`N we will treat as a submission (the 'N' is a release 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-tj@host/myteam/trunk \
         svn+ssh://cs164-tj@host/myteam/tags/proj1.1
       -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/proj1-1
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.

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.*

Don't forget test cases. You can start writing them before you write a line of code.