

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

CS 164  
Fall 2006

P. N. Hilfinger

**Project #2: Parser for Pyth**

**Due:** Friday, 13 October 2006 at 2359

Our second project is to write a parser for Pyth. This parser will take a stream of lexemes provided by your first project and produce an abstract syntax tree (AST) that it will output as text to be read back by the next stage of the compiler. If you aren't happy with how your first project turned out, you are free to borrow (with attribution, of course), anyone else's lexer.

As before, your team will have space in a Subversion repository that the staff will maintain, and 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 BISON (for C++) or 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 FILE1.py FILE2.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:

```

[(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/defs.py")]

```

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

$$(operator\ file\text{-}number\ line\text{-}number\ operand_1 \dots operand_n)$$

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 the notation from Project #1, with one correction: the double quote character will also be represented in octal (as `\042`). 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; it need not look exactly like the example. 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. If you choose to write the output as

```
[(assign 1 1(id 1 1"i")(int 1 1"3"))(assign 1 2(id 1 2"s")(call ...
```

you may want to write a “pretty-printer” for your own use to help in reading the output.

In general, the line number to associate with a construct is the line number that the lexer gives to 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 should use file 0 and line 0 as shown.

Your parser should detect and report syntax errors (on the standard error output) using the same format as the lexer in Project #1:

```
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.

### 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.”

### 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/proj2` 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 `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.

**Table 1:** AST nodes for expressions, part 1.

Construct	AST	Notes
<i>identifier</i>	(id " <i>identifier</i> ")	
<i>integer literal</i>	(int " <i>n</i> ")	where <i>n</i> is the decimal representation of the integer literal (non-negative).
<i>float literal</i>	(float " <i>x</i> ")	where <i>x</i> is the decimal floating-point representation of the literal, as produced by the lexer.
" <i>string literal</i> "	(string " <i>string literal</i> ")	
$(e_1, \dots, e_n)$	(tuple [ $\hat{e}_1 \dots \hat{e}_n$ ])	Includes $(e,)$ but not $(e)$ . Also includes tuple lists without parentheses, as in " $x = 1,2,3$ ".
$[e_1, \dots, e_n]$ $\{k_1 : v_1, \dots, k_n : v_n\}$	(list [ $\hat{e}_1 \dots \hat{e}_n$ ]) (dict [(pair $\hat{k}_1 \hat{v}_1$ )... (pair $\hat{k}_n \hat{v}_n$ )])	
$F(e_1, \dots, e_n)$ $e.i$ $e_1[e_2]$ $e_1[e_2 : e_3]$ $e_1[e_2 :]$	(call $\hat{F}$ [ $\hat{e}_1 \dots \hat{e}_n$ ]) (select (id " <i>i</i> ") $\hat{e}$ ) (call (id "__getitem__") [ $\hat{e}_1 \hat{e}_2$ ]) (call (id "__getslice__") [ $\hat{e}_1 \hat{e}_2 \hat{e}_3$ ]) (call (id "__getslice__") [ $\hat{e}_1 \hat{e}_2 MAX$ ])	where <i>MAX</i> is the maximum integer.
not $e$ $e_1$ and $e_2$ $e_1$ or $e_2$	(not $\hat{e}$ ) (and $\hat{e}_1 \hat{e}_2$ ) (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_") [ $\hat{e}_1$ $\hat{e}_2$ ])	where $\hat{e}$ is the translation of $e_1$ in $e_2$
$e_1$ not in $e_2$	(not $\hat{e}$ )	
$e_1$ is $e_2$ $e_1$ is not $e_2$	(is $\hat{e}_1$ $\hat{e}_2$ ) (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 4 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 4 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 4 in the Pyth manual.

**Table 2:** AST nodes for statements

Construct	AST	Notes
$s_1; \dots s_n$	$[\hat{s}_1 \dots \hat{s}_n]$	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 $s_1 <\text{newline}> s_2$ .
pass		The <b>pass</b> statement never appears in the AST. For example, 'x=y; pass' translates to just the list consisting of one <b>assign</b> node, and a statement list consisting only of <b>pass</b> translates to the empty AST list.
$e_1 = e_2$	(assign $\hat{e}_1 \hat{e}_2$ )	$e_2$ may be an expression or other assignment. Treat $e_1$ as an ordinary expression; don't worry about enforcing the fact that it can't be, e.g., $x+3$ . We'll handle that in the next phase.
$e_1 \oplus = e_2$	(aug_assign (id " $\hat{\oplus}$ ") $\hat{e}_1 \hat{e}_2$ )	$e_2$ may be an expression, not an assignment. $\oplus$ is one of the arithmetic operators, and $\hat{\oplus}$ is the function name that it translates to (see Table 1 in the Pyth manual).
return $e$ return break continue	(return $\hat{e}$ ) (return (id "None")) (break) (continue)	
print $e_1, \dots, e_n$  print $e_1, \dots, e_n,$  print >>F, $e_1, \dots, e_n$  print >>F, $e_1, \dots, e_n,$	(print_newline (id "None") $[\hat{e}_1 \dots \hat{e}_n]$ ) (print (id "None") $[\hat{e}_1 \dots \hat{e}_n]$ ) (print_newline $\hat{F}$ $[\hat{e}_1 \dots \hat{e}_n]$ ) (print $\hat{F}$ $[\hat{e}_1 \dots \hat{e}_n]$ )	where $F$ is an expression yielding a file object.
if $e$ : $S_1$ else: $S_2$ if $e$ : $S_1$ while $e$ : $S_1$ else: $S_2$ while $e$ : $S_1$ for $v$ in $e$ : $S_1$ else: $S_2$ for $v$ in $e$ : $S_1$	(if $\hat{e} \hat{S}_1 \hat{S}_2$ ) (if $\hat{e} \hat{S}_1 []$ ) (while $\hat{e} \hat{S}_1 \hat{S}_2$ ) (while $\hat{e} \hat{S}_1 []$ ) (for $\hat{v} \hat{e} \hat{S}_1 \hat{S}_2$ )  (for $\hat{v} \hat{e} \hat{S}_1 []$ )	$S_1$ and $S_2$ are statement lists.  Treat $v$ as a general expression. The next phase will check it for validity.

**Table 3:** AST nodes for declarations and types

Construct	AST	Notes
<pre>def i = e def i<sub>0</sub> (i<sub>1</sub>, ..., i<sub>n</sub>): S</pre>	<pre>(defconst (id "i") ê) (defun (id "i<sub>0</sub>")   [(id "i<sub>1</sub>") ... (id i<sub>n</sub>)] <math>\hat{S}</math>)</pre>	
<pre>class def i<sub>0</sub> (i<sub>1</sub>, ..., i<sub>n</sub>):   S</pre>	<pre>(class_defun (id "i<sub>0</sub>")   [(id "i<sub>1</sub>") ... (id i<sub>n</sub>)] <math>\hat{S}</math>)</pre>	
<pre>def i<sub>0</sub> (i<sub>1</sub>, ..., i<sub>n</sub>):   import "name"</pre>	<pre>(defun_native   (id "i<sub>0</sub>")   [(id "i<sub>1</sub>") ... (id i<sub>n</sub>)]   (string "name"))</pre>	
<pre>class def i<sub>0</sub> (i<sub>1</sub>, ..., i<sub>n</sub>):   import "name"</pre>	<pre>(class_defun_native   (id "i<sub>0</sub>")   [(id "i<sub>1</sub>") ... (id i<sub>n</sub>)]   (string "name"))</pre>	
<pre>class i(t): S</pre>	<pre>(class (id "i") (id "t") <math>\hat{S}</math>)</pre>	
<pre>global i<sub>1</sub>, ..., i<sub>n</sub></pre>	<pre>(global   [(id "i<sub>1</sub>") ... (id "i<sub>n</sub>)])</pre>	
<pre>i : T</pre>	<pre>(type_decl (id "i") <math>\hat{T}</math>)</pre>	Where $\hat{T}$ is an encoded type, as described in the following entries.
<pre>i (as a type) (t<sub>1</sub>, ..., t<sub>n</sub>) -&gt; t<sub>0</sub></pre>	<pre>(type (id "i")) (func_type [<math>\hat{t}_1 \dots \hat{t}_n</math>] <math>\hat{t}_0</math>)</pre>	



## 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-2.N` we will treat as a submission (the ‘2’ 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-2.1  
-m "First submitted version of project 2"
```

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-2.1  
svn commit -m "First submitted version of project 2"
```

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*.