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

CS 164 **P. N. Hilfinger**
Spring 2010

Project #1: Lexer and Parser for Python Subset

Due: Tuesday, 2 March 2010 at 2400

Our first project is to write a lexer and parser for a (pretty large) subset of Python 2.5. This parser will take a source file and produce an abstract syntax tree (AST), which it will output as text to be read back by the next stage of the compiler.

This semester, you'll implement your solution in C++. The parsing tools FLEX and BISON are available, or you may write the whole thing "by hand," as a recursive-descent compiler.

Your job is to produce 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. 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)—as well as using it to hand in (tag) your team's submission.

1 Running your solution

The program we'll be looking for when we test your submission is called `apyc` (A PYthon Compiler). The command

`./apyc --phase=1` *FILE*$_1$`.py` *FILE*$_2$`.py ...`

will compile the given files, and produce output files named *FILE*$_1$*.ast*, *FILE*$_2$*.ast*, etc. Any other value for the `--phase` option will be an error for now (this option tells the compiler how far it should process its input). The command

`./apyc --phase=1 -o` *FILE*$_2$ *FILE*$_1$`.py`

Compiles just *FILE*$_1$`.py` into *FILE*$_2$.

2 Python Subset

You do not have to parse all of Python; we are making quite a few significant cuts from the full posted references, as detailed in this section.

Lexical structure.

L1. No Unicode strings (e.g., `u"Foo"`).

L2. No long integer literals. All integer literals must be in the range $[-2^{31}, 2^{31}]$, and a literal equal to 2^{31} actually means -2^{31} (Yeah, it's an ugly compromise. Sue me).

L3. No imaginary numbers.

Expressions.

E1. No list comprehensions (such as `[x for x in xrange(0,10)]`), or generator expressions (such as `(x for x in xrange(0,10))`), just expression lists.

E2. No string conversions (backquotes).

E3. No extended slices.

E4. No keyword arguments, `*` arguments, or `**` arguments in calls. Likewise in parameter lists for **defs** and `lambdas`. No default parameters.

E5. We don't use the obsolescent <> operator.

Statements.

S1. No **del** statement.

S2. No **yield** statement.

S3. Only the simple "`raise`" and "`raise` *E*" forms of raise statement.

S4. The only forms of **import** are "`import` *list-of-simple-identifiers*" and `from` *identifier* `import` *list-of-simple-identifiers*."

S5. No **future** statement.

S6. No **exec** statement.

S7. No **finally** or **else** clauses in **try** statements.

S8. No **with** statement.

S9. Only new-style class declarations with a single parent class, as in

```
class Foo(bar):
   ...
```

S10. No decorators (`@`A).

S11. Class declarations and import statements may not be nested inside any other construct.

S12. Function declarations may not appear within the statements of an **if**, **while**, **for**, or **try** statement.

S13. No **assert** statement.

S14. Function declarations may contain only simple identifiers as parameters, so that

```
def foo(x, (y, z)): ...
```

is illegal.

S15. A **global** statement for a variable V must appear before any assignment to V in a given function or class.

3 Output

Your `apyc` program should produce, in the `.ast` files, representations of the corresponding ASTs, using the abstract syntax and format given below. This is the same output that we will use as input to the next stage of the compiler. 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.

Figure 1 contains an example of a Python program and the resulting AST output as produced by

```
./apyc --phase=1 foo.py
```

The output is in Lisp-like notation. Parenthesized items represent tree nodes. Each node has the form

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

Figure 1: Example of a Python program and resulting AST.

Program foo.py:

```
1.    # This is a small test program (line numbers to left)
2.    import defns
3.    def f(n):
4.       i = 0
5.       while i <= n:
6.         if 1 < i % 7 <= 2:
7.            print i,
8.       else:
9.         s = i + 2; t += s ** 2
10.        print "s =", s, "t =", t
```

Resulting contents of foo.ast:

```
(module 0
  (import_module 2 (id 2 defns))
  (def 3 (id 3 f) (id_list 3 (id 3 n))
   (block 4
    (assign 4 (id 4 i) (int_literal 4 0))
    (while 5 (comparison 5 (id 5 i) (id 5 <=) (id 5 n))
     (if 6 (comparison 6 (int_literal 6 1)
                         (id 6 <)
                         (binop 6 (id 6 i) (id 6 %) (int_literal 6 7))
                         (id 6 <=)
                         (int_literal 6 2))
       (print 7 () (expr_list 7 (id 7 i)))
       ())
     (stmt_list 8
       (stmt_list 9
         (assign 9 (id 9 s) (binop 9 (id 9 i) (id 9 +) (int_literal 9 2)))
         (aug_assign 9 (id 9 t) (id 9 +)
                       (binop 9 (id 9 s) (id 9 **) (int_literal 9 2))))
       (println 10 () (expr_list 10
                       (string_literal 10 "s =")
                       (id 10 s)
                       (string_literal 10 "t =")
                       (id 10 t))))))))\}
```

The *line-number* identifies the initial line number of the source from which the node was translated. The *operands* are either tree nodes, quoted strings, integer literals, symbols, or the special symbol `()`, indicating an optional operand that is not present.

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—and then executing the resulting program and comparing results. You are allowed to translate your program into *any* AST that represents a program with equivalent results. In particular, the use of statement lists (`stmt_list`) is very flexible. If the 'else' clause of an **if** statement is a single statement, you are free to represent it as the AST for that statement, or as a statement list with a single statement in it. The translation of lines 9 and 10 of Figure 1 could have been rendered as one statement list containing three statements (rather than a statement list containing a two-element statement list and a statement).

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.

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 exit code 1 when processing is complete (it exits with code 0 normally). 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., '!').
- Any use of reserved words that are not used in our subset, but are not allowed as identifiers (see the list of keywords in the Python documentation).
- Inconsistent indentation.

4 Abstract Syntax Trees

The abstract syntax operators to be output by your parser are as given by the BNF in Table 1. The grammar uses the '`*`' and '`+`' notations from regular expressions to denote sequences of symbols, and unquoted parentheses for grouping. Besides the quoted tokens in the grammar, there are the following terminal symbols:

INT Denotes a non-negative decimal integer literal.

STRING Denotes a string literal in double quotes. These literals 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\040contains\t\"Hello, world!\"\n"
```

gets written out as

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

ID A symbol, appearing without quotation marks. For the purposes of the AST, symbols may contain letters, digits, underscores, and any of the Python operator symbols (but no, these are still not legal as identifiers in programs).

FLOAT A C/C++/Java-style floating-point literal (of type `double`).

4.1 Details of some ASTs

Most of the translations should be clear. Here, we describe a few possibly non-obvious cases. In the descriptions that follow, if X is a Python construct, X' denotes the AST tree that translates it.

pass There is no explicit 'pass' node. You can simply elide all **pass** statements or replace them with empty statement lists (in the AST, unlike Python, it is possible to have completely empty statement lists).

binop and unop These node types represent ordinary binary and unary operators in Python. In both cases, the Id operand is the operator symbol (e.g., `(id 3 +)`).

comparison Python comparisons have a special evaluation rule. An entire comparison yields a result of `True` or `False`, but an expression such as `x < y < z` is not equivalent to `(x < y) < z`. Instead, if the `x < y` part is true, its "value" is that of `y`, which is then compared to `z`. If the `x<y` part is false, the entire comparison is

Table 1: Abstract Syntax Trees

```
Compilation : '(' "module" N Stmt* ')'

N : INT

Expr : '(' "binop" N Expr Id Expr ')'
     | '(' "comparison" N Expr (Id Expr)+ ')'
     | '(' "unop" N Id Expr ')'
     | '(' "if_expr" N Expr Expr Expr ')'
     | '(' "and" N Expr Expr ')'
     | '(' "or" N Expr Expr ')'
     | '(' "lambda" N IdList Expr ')'
     | '(' "tuple" N Expr* ')'
     | '(' "list_display" N Expr* ')'
     | '(' "call" N Expr ExprList ')'
     | '(' "dict_display" N Pair* ')'
     | '(' "string_literal" N STRING ')'
     | '(' "int_literal" N INT ')'
     | '(' "float_literal" N FLOAT ')'
     | Target

Stmt : Expr
     | Assign
     | StmtList
     | '(' "aug_assign" N Target Id Expr ')'
     | '(' "print" N Expr0 ExprList  ')'
     | '(' "println" N Expr0 ExprList  ')'
     | '(' "return" N Expr0 ')'
     | '(' "raise" N Expr0 ')'
     | '(' "break" N ')'
     | '(' "continue" N ')'
     | '(' "import_module" N Id* ')'
     | '(' "import_from" N Id IdList ')'
     | '(' "global" N Id+ ')'
     | '(' "if" N Expr Stmt Stmt0 ')'
     | '(' "while" N Expr Stmt Stmt0 ')'
     | '(' "for" N TargetList Expr Stmt Stmt0 ')'
     | '(' "try" N Stmt (Expr0 Target0 Stmt)+ ')'
     | '(' "def" N Id IdList Block ')'
     | '(' "class" N Id Id Block ')'

Assign : '(' "assign" N TargetList RightSide ')'

Block : '(' "block" N Stmt* ')'

ExprList : '(' "expr_list" N Expr* ')'

Expr0 : Expr | "()"

Id : '(' "id" N ID ')'

IdList : '(' "id_list" N Id* ')'

Pair : '(' "pair" N Expr Expr ')'

RightSide : Expr | Assign

StmtList : '(' "stmt_list" N Stmt* ')'

Stmt0 : Stmt | "()"

Target:
       Id
     | '(' "attributeref" N Expr Id ')'
     | '(' "subscription" N Expr Expr ')'
     | '(' "slicing" N Expr Expr0 Expr0  ')'

TargetList:
       Target
     | '(' "target_list" N TargetList+   ')'

Target0: Target | "()"
```

false, and `z` is not even evaluated. Therefore we need a special operator for comparisons. Again, the Id operands are the operator symbols (`<`, `>`, `<=`, `>=`, `==`, `!=`, `in`, `notin`, `is`, `isnot`). For simplicity, we'll use `comparison` for single comparisons as well (such as `x<y`), even though those really act exactly like ordinary binary operators.

if_expr The expression 'E_0 `if` T `else` E_1' is represented as '`(ifexpr` N T' E_0' E_1'`)`'. As you can see, the operand order in the AST is not the same as in the source.

tuple Translates `(`$E_1, \ldots, E_k$`)`. It also translates cases where the parentheses are allowed to be omitted. For example, in the statements

```
x = 1, 2, 3
for y in 1, 2, 3: ...
return 1, 2, 3
```

the `1,2,3` should be translated as if it were `(1, 2, 3)`. When such a list is used as a bare statement, on the other hand, as in:

```
f(x), f(y), f(z)
```

you *may* translate this as a tuple or you *may* translate it as a list of three statements.

list_display Translates `[`$E_1, \ldots, E_k$`]`.

dict_display $\{K_1 : E_1, \ldots, K_n : E_n\}$ translates to `(dict_display (pair` $K_1' : E_1'$`)...(pair` $K_n' : E_n'$`))`.

assign There is a technical problem with parsing assignments such as

```
x, y, z = E
```

because `x, y, z` is an expression in its own right. As a result, obvious renderings of the grammar into Bison will cause conflicts (am I creating a TargetList or an ExprList? I don't know until I see the '`=`'). You can get around this easily by parsing the left side of an assignment as a plain expression and then checking the resulting AST with a specially written C++ function to make sure it is a proper target list.

aug_assign The statement X $\oplus$`=` E translates to `(aug_assign` N X' $\oplus$ E'`)`, where $\oplus$ is one of the Python binary operators.

try The statement

```
try:
    S0
except E1, V1:
    S1
except ...
    ...
except En, Vn:
    Sn
```

translates to `(try` N S'_0 E'_1 V'_1 ... E'_n V'_n S'_n`)`.

class The first `Id` is the class name, and the second is its parent's name.

attributeref Translates E`.`I.

subscription Translates E_1`[`E_2`]`. The Python syntax allows E_2 to be a list of expressions; however that is just a short hand. Translate X`[`A, B, C`]` as if it were X`[(`A, B, C`)]`.

print, println Translate the `print` command with and without a trailing comma, respectively. The optional first expression operand denotes the file written to (denoted in Python with >>*file*).

5 What to Turn In

The directory you turn in (see §6) should contain a file `Makefile` that is set up so that

```
gmake
```

(the default target) compiles your program,

```
gmake check
```

runs all your tests against your program, and finally,

```
gmake APYC=PROG check
```

runs all your tests against the program *PROG* (by default, in other words, *PROG* is your program, `./apyc`). We'll put a sample Makefile in the `~cs164/hw/proj1` directory and the staff project 1 repository:

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

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

We will test your program by first using it to translate a suite of correct Python programs (checking that your program exits cleanly with an exit code of 0), and we will check the translations by unparsing them (using a program `pyunparse`, which is a Python script we'll supply), running the resulting Python programs, and checking their output. Next, we will run your program against a suite of syntactically erroneous programs, and check that you produce an appropriate error message (its contents are not important as long as the form is as specified) and that your program exits with a standard error code (as produced by `exit(1)` in C++).

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.

6 How to Submit

Submit your project just as for homeworks, but in your team's tag directory rather than your personal directory. The tag names will be `proj1-`N, where N is an integer.

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. *Never, ever, ever* wait to submit or commit something pending some question you have for us (like "will it count as late if I. . . ")!!!! We can always undo a submission; that's what version control is good for. But the repository is not psychic; it does not know when you were ready to submit, only when you actually submitted.

7 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(s). 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` in Emacs, 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` debugger (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.