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

CS 164 **P. N. Hilfinger**
Spring 2018

Project #1: Lexer and Parser for Python Dialect (revision 2)

Due: Mon., 5 March at 2400

Our first project is to write a lexer and parser for a (pretty large) dialect of Python 2.5. This parser will take a source file and produce an abstract syntax tree (AST), which it will output as text for testing purposes.

This semester, we'll provide some support for implementing your solution in C++ 11. The parsing tools FLEX and BISON are available, or you may write the whole thing "by hand," as a recursive-descent compiler, or find and use another parser-generator system (such as Antlr for Java). You may also use another language if you wish.

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`

will compile the given file, and produce an output file named $FILE_1$*.ast*. 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 Dialect

We have deleted a great deal of Python 2.5 for the purposes of this project, and added some significant features. The resulting language is statically typed, unlike Python.

2.1 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 $[0, 2^{30}]$, and a literal equal to 2^{30} actually means -2^{30} (Yeah, it's an ugly compromise. Sue me).

L3. No imaginary numbers.

L4. No floating-point literals.

L5. The following words may not appear, either as keywords or identifiers. The statements or expressions that use them in full Python are not in our dialect:

```
as assert del except exec from finally future global import lambda
raise try with yield
```

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 (such as `s[1:n:2]`).

E4. No keyword, `*`, or `**` arguments. No default parameters.

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

E6. No **lambda** expressions.

E7. No '`|`', '`&`', '`^`', '`<<`', '`>>`', or '`~`' operators.

Statements.

S1. No **import** statements.

S2. Only class declarations with a default parent class, as in

```
class Foo:
   ...
```

S3. No decorators (`@`A).

S4. Class declarations may not be nested inside any other construct.

S5. Function declarations may not appear within the statements of an **if**, **while**, or **for**.

S6. Function declarations may contain only simple identifiers as parameters, so that

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

is illegal.

S7. No augmented assignment (e.g., "`+=`").

S8. Target lists (in assignments and **for** statements) are not nested. One may not write

```
(a, (b, c)) = L        or      a, (b,c) = L
```

Instead, to get the same effect, one must use

```
(a, T) = L             or      a, T = L
(b, c) = T                     b,c = T
```

S9. No **global** statement.

S10. No >> clauses on **print** statements (all output goes to the standard output.)

S11. The identifiers `None`, `True`, and `False` are keywords, and may not be used as variables (in Id nodes). Thus, they may not be assigned to.

2.2 Additional features

Overloading. Function names may be overloaded, as in Java or C++. One may use the same name for two functions. A call is disambiguated by the number and types of its parameters. (This addition makes no difference for project #1.)

Typing declarations. Standard Python is purely dynamically typed. We've added a bit of static typing for own purposes.

A target variable in any assignment (which includes the control variables of **for** nodes), and a formal parameter in a **def** may have the form "I`::`T", where '`::`' is a new token, I is an identifier, and T denotes a type. A type denotation may be one of the following:

1. an identifier (e.g., `int`);
2. a type variable, written as an identifier prepended with '`$`' (e.g., `$AType`);
3. a type constructor of the form

   I **of** `[`$T_1, T_2, \ldots, T_n$`]`

   where I is an identifier, $n \geq 1$, and the T_i are type denotations. If $n = 1$, the brackets are optional. For example, `list of [int]`, `list of int`, or `dict of [str, list of $T]`;
4. a function-type denotation, whose form is `(`$T_1, \ldots, T_n$`)->`T_0. Here, the T_i are (recursively) type denotations. The types in parentheses are types of the arguments and the type after the arrow is the return type.

   You may also follow the parameter list of a function with a return type, as in

   ```
       def incr(n::int)::int:
           return n+1
   ```

For future reference (it's irrelevant for project #1), the '`::`' notation will indicate that the variable denoted by I or the return value here has type T. A typing on an identifier I applies to *all* instances of I that refer to the same variable (so that in

```
a = "Hello"

def f(x):
    a::int = x
    print a + 3
```

the '`a`' defined in '`f`' has integer type, but the one defined outside '`f`' has unknown type.

Type arguments on classes. To define a class that takes type arguments, use the syntax

```
class I of [V1,...,Vn]:
   ...
```

where the V_i are type variables. For example:

```
class PriorityList of [$ItemType]:
   ...
```

Native functions. The body of a function may be replaced in its entirety by the single statement "**native** *string-literal*," either in the form

```
def foo(...):  native "NAME"
```

or

```
def foo(...):
    native "NAME"
```

which will mean that the function is implemented by a "native" function (in C or C++ in our case), as part of the run-time system.

3 Output

Your `apyc` program should produce, in the `.ast` files, representations of the corresponding ASTs, using the abstract syntax and format given below.

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*$_1$ · · · *operand*$_n$)

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 (you can put it all on one line if you want, although this might make output difficult to read and debug). 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).

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.
3.    def f(n):
4.       i::int = 0
5.       while i <= n:
6.         if 1 < i % 7 <= 2:
7.            print i,
8.       else:
9.         s = i + 2; t = t + s ** 2
10.        print "s =", s, "t =", t
```

Resulting contents of foo.ast:

```
(module 0
  (def 3 (id 3 f) (formals_list 3 (id 3 n))
   (block 4
    (assign 4 (typed_id 4 (id 4 i) (type 4 (id 4 int) (type_list 4)))
              (int_literal 4 0))
    (while 5 (compare 5 (id 5 __le__) (id 5 i) (id 5 n))
     (if 6 (compare 6 (id 6 __le__)
                      (left_compare 6 (id 6 __lt__)
                         (int_literal 6 1)
                         (binop 6 (id 6 __mod__)
                                  (id 6 i) (int_literal 6 7)))
                      (int_literal 6 2))
       (print 7 (id 7 i)))
     (stmt_list 8
       (stmt_list 9
         (assign 9 (id 9 s) (binop 9 (id 9 __add__) (id 9 i) (int_literal 9 2)))
         (assign 9 (id 9 t)
                   (binop 9 (id 9 __add__) (id 9 t)
                            (binop 9 (id 9 __pow__) (id 9 s) (int_literal 9 2))))
       (println 10 (string_literal 10 "s =")
                   (id 10 s)
                   (string_literal 10 "t =")
                   (id 10 t))))))))
```

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 on pages 9 and 10. 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`) or greater than 126 (`0177`–`0377`). 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).

4.1 Details of some ASTs

In Table 1, as in regular expressions, a trailing asterisk means "0 or more of," a plus means "1 or more of," and a question mark means "0 or 1 of." The table also uses parentheses for grouping.

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 the concrete Python syntax, it is possible to have completely empty statement lists).

binop, unop, and compare These node types represent ordinary binary, unary, and comparison operators in Python. In the AST, however, we give them the form of calls. The Id operand will always be one of the special identifiers used in real Python as the names of user-definable operators (see Table 3.)

The idea is that these can be treated as function calls on the named special functions, so that an expression `a + b` that appeared on line 3 would translate to the AST

```
    (binop 3 (id 3 __add__) (id 3 a) (id 3 b))
```

The translation to functions is not exactly what true Python does (for example, the starred identifiers the table exist only in our dialect), but will serve for our purposes.

left_compare Python comparisons have a special evaluation rule. The comparison functions (like `__lt__`) yield `True` or `False`. Likewise, 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 false, and `z` is not even evaluated. Therefore we need a special operator for comparisons that yield their right operands when true. If a comparison (unparenthesized) is the left operand of another comparison (e.g., `x<y` in the expression `x<y<z` or `x<y<z` in the expression

Table 1: Abstract Syntax Trees, Part I

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

N : INT

Expr : ’(’ "int_literal" N INT ’)’
  | StringLiteral
  | VarRef
  | SpecialId
  | ’(’ "next_value" N ’)’
  | ’(’ "call" N Expr Expr* ’)’
  | ’(’ "call" N TypeId Expr* ’)’
  | ’(’ "binop" N Id Expr Expr ’)’
  | ’(’ "left_compare" N Id Expr Expr ’)’
  | ’(’ "compare" N Id Expr Expr ’)’
  | ’(’ "unop" N Id Expr ’)’
  | ’(’ "subscript" N Id Expr Expr ’)’
  | ’(’ "slice" N Id Expr Expr Expr ’)’
  | ’(’ "if_expr" N Expr Expr Expr ’)’
  | ’(’ "and" N Expr Expr ’)’
  | ’(’ "or" N Expr Expr ’)’
  | ’(’ "tuple" N Expr* ’)’
  | ’(’ "list_display" N Expr* ’)’
  | ’(’ "dict_display" N Pair* ’)’

StringLiteral:
    ’(’ "string_literal" N STRING ’)’

Id : ’(’ "id" N ID ’)’

SpecialId : ’(’ "None" N ’)’
  | ’(’ "True" N ’)’
  | ’(’ "False" N ’)’

VarRef:
    Id
  | ’(’ "attributeref" N Expr Id ’)’

Pair : ’(’ "pair" N Expr Expr ’)’
```

```
Stmt : Expr
  | Assign
  | StmtList
  | ’(’ "print" N Expr*  ’)’
  | ’(’ "println" N Expr*  ’)’
  | ’(’ "return" N Expr ’)’
  | ’(’ "break" N ’)’
  | ’(’ "continue" N ’)’
  | ’(’ "if" N Expr Stmt Stmt ’)’
  | ’(’ "while" N Expr Stmt Stmt ’)’
  | ’(’ "for" N TargetList Expr
                Stmt Stmt ’)’
  | ’(’ "def" N Id Formals Type Stmt* ’)’
  | ’(’ "class" N Id TypeFormals Stmt* ’)’
  | ’(’ "native" N StringLiteral ’)’

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

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

RightSide : Expr | Assign

Target:
    VarRef
  | ’(’ "subscript_assign" N Id Expr Expr Expr ’)’
  | ’(’ "slice_assign" N Id Expr Expr Expr Expr ’)’
  | TypedId

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

Formals : ’(’ "formals_list" N (Id | TypedId)* ’)’
```

Table 2: Abstract Syntax Trees, Part II

```
TypedId : '(' "typed_id" N Id Type ')'

Type : TypeId
  | TypeVar
  | '(' "function_type" N Type Type* ')'
TypeId : '(' "type" N Id Type* ')'
TypeVar : '(' "type_var" N ID ')'

TypeFormals: '(' "type_formals_list" N TypeVar* ')'
```

Table 3: Table of function names for Python operators

Op	Identifier	Op	Identifier	Unary Op	Identifier	Op	Identifier
+	`__add__`	>	`__gt__`	+	`__pos__`	in	`__in__` *
-	`__sub__`	<	`__lt__`	-	`__neg__`	not in	`__notin__` *
*	`__mul__`	<=	`__le__`	not	`__not__` *	is	`__is__` *
/	`__floordiv__`	>=	`__ge__`			is not	`__isnot__` *
%	`__mod__`	==	`__eq__`				
**	`__pow__`	!=	`__ne__`				

* Identifiers not used in actual Python.

`x<y<z==p`), then we use the `left_compare` AST operator for that operand rather than `compare`. In all other cases, we use the `compare` operator.

subscription, slicing When used as values, subscription (`x[k]`) and slicing (`x[1:5]`) are also treated as function calls, translated respectively to

```
(subscript N (id N __getitem__) (id N x) (id N k))
(slice N (id N __getslice__) (id N x) (int_literal N 1) (int_literal N 5))
```

Use `int_literal` nodes denoting 0 and $2^{30} - 1$ for missing arguments of slice (as in `x[3:]`).

When they are used as targets of assignments (or **for**), we translate subscription and slicing them a bit differently. The assignments `x[k] = 3` and `x[a:b] = L` (say on lines 1 and 2) become

```
(assign 1 (subscript_assign 1 (id 1 __setitem__)
               (id 1 x) (id 1 k) (next_value 1))
```

```
                (int_literal 1 3))
    (assign 2 (slice_assign 2 (id 2 __setslice__)
                  (id 2 x) (id 2 a) (id 2 b) (next_value 2))
                (id 2 L))
```

The identifiers `__setitem__` and (before Python 3) `__setslice__` are used in Python for user-defined subscript and slice assignment operations. The special operator `next_value` means "whatever value is supposed to be assigned here." It's utility becomes clearer in the case of multiple assignments like this:

```
x[1], x[2] = f(L)
```

which becomes

```
(assign 1
   (target_list 1
     (subscript_assign 1
        (id 1 __setitem__) (id 1 x) (int_literal 1 1) (next_value 1))
     (subscript_assign 1
        (id 1 __setitem__) (id 1 x) (int_literal 1 2) (next_value 1)))
   (call 1 (id 1 f) (id 1 L)))
```

The assign operator takes care of deconstructing the return value of `f` and causing its elements to get yielded as the values of `next_value`, as for an iterator.

if_expr The expression 'E_0 `if` T `else` E_1' is represented as '`(if_expr` 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 target list or an expression list? 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. Alternatively, you can create a GLR parser, which puts off the decision of whether to interpret the left-hand side as a target list or an expression list until it sees (or does not see) the '`=`'.

return A Python return without an expression is equivalent to **return** `None`. In the AST, represent the `None` explicitly with the corresponding node.

attributeref Translates E`.`I.

subscript 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.

def Translates **def** statements. The optional `Type` is the optional return type.

5 What to Turn In

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

```
make
```

(the default target) compiles your program,

```
make check
```

runs all your tests against your program, and finally,

```
make APYC=PROG check
```

runs all your tests against the program *PROG* (by default, in other words, *PROG* is your program, `./apyc`). We've put a sample Makefile in the `~cs164/hw/proj1` directory and the shared staff repository. If you have set up your local copy of your GIT team repository as in our setup directions, one member of your team can start your project from this point with

```
$ cd team-repo
$ git fetch shared
$ git checkout -b proj1 shared/proj1
$ git push -u origin proj1
```

and other team members can then get on board with their local copies:

```
$ cd team-repo
$ git fetch origin
$ git checkout proj1
```

You may modify any of our skeleton files (or add or delete files) at will as long as the three `make` commands continue to work on the instructional machines.

Members of the team can bring their own local copies of the project up to date with what others have pushed to their shared `cs164-ta` repository with

```
$ git pull --rebase
```

and can push their own commits with

```
$ git push
```

Should we make modifications to our skeleton, you can incorporate them into your files (if desired) with the following steps, assuming you are in your `proj1` branch:

```
$ git commit -a    # If needed.
$ git fetch shared
$ git merge shared/proj1
```

If changes we've made overlap yours, you'll be told there are conflicts that must be resolved. The `git status` command will tell you what files are still conflicted. GIT will have marked the conflicting portions with matching <<<<<<< and >>>>>>> markers in the conflicted files. Simply edit until correct, and use `git add` on the file to tell GIT the problem is resolved. Use `git commit` to finish the resolution, or `git merge --abort` to try to roll back the whole merge to try again. Again, one team member should do all this, and others should use `git pull --rebase` to get the result.

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 homework, but in your team's GIT repository 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. GIT 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 partners. 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 `make` 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 GIT repository to coordinate with your partners and to save development versions *frequently.* Also, please push commits to your `cs164-ta` repository frequently, both so your partners have access to your work and so that we can keep tabs on your progress.

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