Simple Top-Down Parsing in Python

Fredrik Lundh | July 2008

In *Simple Iterator-based Parsing*, I described a way to write simple recursive-descent parsers in Python, by passing around the current token and a token generator function.

A recursive-descent parser consists of a series of functions, usually one for each grammar rule. Such parsers are easy to write, and are reasonably efficient, as long as the grammar is "prefix-heavy"; that is, that it's usually sufficient to look at a token at the beginning of a construct to figure out what parser function to call. For example, if you're parsing Python code, you can identify most statements simply by looking at the first token.

However, recursive-descent is less efficient for expression syntaxes, especially for languages with lots of operators at different precedence levels. With one function per rule, you can easily end up with lots of calls even for short, trivial expressions, just to get to the right level in the grammar.

For example, here's an excerpt from Python's expression grammar. The "test" rule is a basic expression element:

```
test: or_test ['if' or_test 'else' test] | lambdef or_test: and_test ('or' and_test)* and_test: not_test
```

With a naive recursive-descent implementation of this grammar, the parser would have to recurse all the way from "test" down to "trailer" in order to parse a simple function call (of the form "expression(arglist)").

In the early seventies, Vaughan Pratt published an elegant improvement to recursive-descent in his paper *Top-down Operator Precedence*. Pratt's algorithm associates semantics with tokens instead of grammar rules, and uses a simple "binding power" mechanism to handle precedence levels. Traditional recursive-descent parsing is then used to handle odd or irregular portions of the syntax.

In an article (and book chapter) with the same name, Douglas Crockford shows how to implement the algorithm in a subset of JavaScript, and uses it to develop a parser that can parse itself in the process.

In this article, I'll be a bit more modest: I'll briefly explain how the algorithm works, discuss different ways to implement interpreters and translators with it in Python, and finally use it to implement a parser for Python's expression syntax. And yes, there will be benchmarks too.

Introducing The Algorithm

Like most other parsers, a topdown parser operates on a stream of distinct syntax elements, or tokens. For example, the expression "1 + 2" could correspond to the following tokens:

```
literal with value 1 add operator literal with value 2 end of program
```

In the topdown algorithm, each token has two associated functions, called "nud" and "led", and an integer value called "lbp".

The “nud” function (for null denotation) is used when a token appears at the beginning of a language construct, and the “led” function (left denotation) when it appears inside the construct (to the left of the rest of the construct, that is).

The “lbp” value is a binding power, and it controls operator precedence; the higher the value, the tighter a token binds to the tokens that follow.

Given this brief introduction, we’re ready to look at the core of Pratt’s algorithm, the expression parser:

```
def expression(rbp=0):
    global token
    t = token
    token = next()
    left = t.nud()
    while rbp < token.lbp:
        t = token
        token = next()
        left = t.led(left)
    return left
```

(Pratt calls this function “parse”, but we’ll use the name from Crockford’s article instead.)

Here, “token” is a global variable that contains the current token, and “next” is a global helper that fetches the next token. The “nud” and “led” functions are represented as methods, and the “lbp” is an attribute. The “left” variable, finally, is used to pass some value that represents the “left” part of the expression through to the “led” method; this can be any object, such as an intermediate result (for an interpreter) or a portion of a parse tree (for a compiler).

If applied to the simple expression shown earlier, the parser will start by calling the “nud” method on the first token. In our example, that’s a literal token, which can be represented by something like the following class:

```
class literal_token:
    def __init__(self, value):
        self.value = int(value)
    def nud(self):
        return self.value
```

Next, the parser checks if the binding power of the next token is at least as large as the given binding power (the “rbp” argument, for “right binding power”). If it is, it calls the “led” method for that token. Here, the right binding power is zero, and the next token is an operator, the implementation of which could look like:

```
class operator_add_token:
    lbp = 10
    def led(self, left):
        right = expression(10)
        return left + right
```

The operator has a binding power of 10, and a “led” method that calls the expression parser again, passing in a right binding power that’s the same as the operator’s own power. This causes the expression parser to treat everything with a higher power as a subexpression, and return its result. The method then adds the left value (from the literal, in this case) to the return value from the expression parser, and returns the result.

The end of the program is indicated by a special marker token, with binding power zero (lower than any other token). This makes sure that the expression parser stops when it reaches the end of the program.

```
class end_token:
    lbp = 0
```

And that’s the entire parser. To use it, we need a tokenizer that can generate the right kind of token objects for a given source program. Here’s a simple regular expression-based version that handles the minimal language we’ve used this far:

```
import re

token_pat = re.compile("\s*(?:(\d+)|(.))")

def tokenize(program):
    for number, operator in token_pat.findall(program):
        if number:
            yield literal_token(number)
        elif operator == "+":
            yield operator_add_token()
        else:
            raise SyntaxError("unknown operator")
    yield end_token()
```

Now, let’s wire this up and try it out:

```
def parse(program):
    global token, next
    next = tokenize(program).next
    token = next()
    return expression()

>>> parse("1 + 2")
3
```

Not counting the calls to the tokenizer, the parser algorithm will make a total of four calls to parse this expression; one for each token, and one extra for the recursive call to the expression parser in the “led” method.

Extending the Parser

To see how this scales, let’s add support for a few more math operations. We need a few more token classes:

```
class operator_sub_token:
    lbp = 10
    def led(self, left):
        return left - expression(10)

class operator_mul_token:
    lbp = 20
    def led(self, left):
        return left * expression(20)

class operator_div_token:
    lbp = 20
    def led(self, left):
        return left / expression(20)
```

Note that “mul” and “div” uses a higher binding power than the other operators; this guarantees that when the “mul” operator is invoked in the expression “1 * 2 + 3”, it only gets the literal “2”, instead of treating “2 + 3” as a subexpression.

We also need to add the classes to the tokenizer:

```
def tokenize(program): for number, operator in token_pat.findall(program): if number: yield literal_toke
```

but that’s it. The parser now understands the four basic math operators, and handles their precedence correctly.

```
>>> parse("1+2")
3
>>> parse("1+2*3")
```

```
7
>>> parse("1+2-3*4/5")
1
```

Despite the fact that we've added more grammar rules, the parser still makes the same number of calls as before; the expression "1+2" is still handled by four calls inside the parser.

However, codewise, this isn't that different from a recursive-descent parser. We still need to write code for each token class, and while we've moved most of the dispatching from individual rules to the expression parser, most of that ended up in a big if/else statement in the tokenizer.

Before we look at ways to get rid of some of that code, let's add two more features to the parser: unary plus and minus operators, and a Python-style exponentiation operator (**).

To support the unary operators, all we need to do is to add "nud" implementations to the relevant tokens:

```
class operator_add_token:
    lbp = 10
    def nud(self):
        return expression(100)
    def led(self, left):
        return left + expression(10)

class operator_sub_token:
    lbp = 10
    def nud(self):
        return -expression(100)
    def led(self, left):
        return left - expression(10)
```

Note that the recursive call to expression uses a high binding power, to make sure that the unary operator binds to the token immediately to the right, instead of to the rest of the expression ("(-1)-2" and "-(1-2)" are two different things).

Adding exponentiation is a bit trickier; first, we need to tweak the tokenizer to identify the two-character operator:

```
token_pat = re.compile("\s*(?:(\d+)|(\*\*|.))") ... elif operator == "**": yield operator_pow_token() ..
```

A bigger problem is that the operator is right-associative (that it, it binds to the right). If you type "2**3**4" into a Python prompt, Python will evaluate the "3**4" part first:

```
>>> 2**3**4
2417851639229258349412352L
>>> (2**3)**4
4096
>>> 2**(3**4)
2417851639229258349412352L
>>>
```

Luckily, the binding power mechanism makes it easy to implement this; to get right associativity, just subtract one from the operator's binding power when doing the recursive call:

```
class operator_pow_token:
    lbp = 30
    def led(self, left):
        return left ** expression(30-1)
```

In this way, the parser will treat subsequent exponentiation operators (with binding power 30) as subexpressions to the current one, which is exactly what we want.

Building Parse Trees

A nice side-effect of the top-down approach is that it's easy to build parse trees, without much extra overhead; since the tokenizer is creating a new object for each token anyway, we can reuse these objects as nodes in the parse tree.

To do this, the "nud" and "led" methods have to add syntax tree information to the objects, and then return the objects themselves. In the following example, the literal leaf nodes has a "value" attribute, and the operator nodes have "first" and "second" attributes. The classes also have __repr__ methods to make it easier to look at the resulting tree:

```
class literal_token:
    def __init__(self, value):
        self.value = value
    def nud(self):
        return self
    def __repr__(self):
        return "(literal %s)" % self.value

class operator_add_token:
    lbp = 10
    def nud(self):
        self.first = expression(100)
        self.second = None
        return self
    def led(self, left):
        self.first = left
        self.second = expression(10)
        return self
    def __repr__(self):
        return "(add %s %s)" % (self.first, self.second)

class operator_mul_token:
    lbp = 20
    def led(self, left):
        self.first = left
        self.second = expression(20)
        return self
    def __repr__(self):
        return "(mul %s %s)" % (self.first, self.second)
```

(implementing "sub", "div", and "pow" is left as an exercise.)

With the new token implementations, the parser will return parse trees:

```
>>> parse("1")
(literal 1)
>>> parse("+1")
(add (literal 1) None)
>>> parse("1+2+3")
(add (add (literal 1) (literal 2)) (literal 3))
>>> parse("1+2*3")
(add (literal 1) (mul (literal 2) (literal 3)))
>>> parse("1*2+3")
(add (mul (literal 1) (literal 2)) (literal 3))
```

The unary plus inserts a "unary add" node in the tree (with the "second" attribute set to None). If you prefer, you can skip the extra node, simply by returning the inner expression from "nud":

```
class operator_add_token:
    lbp = 10
    def nud(self):
        return expression(100)
```

```
    ...

>>> parse("1")
(literal 1)
>>> parse("+1")
(literal 1)
```

Whether this is a good idea or not depends on your language definition (Python, for one, won't optimize them away in the general case, in case you're using unary plus on something that's not a number.)

Streamlining Token Class Generation

The simple parsers we've used this far all consist of a number of classes, one for each token type, and a tokenizer that knows about them all. Pratt uses associative arrays instead, and associates the operations with their tokens. In Python, it could look something like:

```
nud = {}; led = {}; lbp = {}

nud["+"] = lambda: +expression(100)
led["+"] = lambda left: left + expression(10)
lbp["+"] = 10
```

This is a bit unwieldy, and feels somewhat backwards from a Python perspective. Crockford's JavaScript implementation uses a different approach; he uses a single "token class registry" (which he calls "symbol table"), with a factory function that creates new classes on the fly. JavaScript's prototype model makes that ludicrously simple, but it's not that hard to generate classes on the fly in Python either.

First, let's introduce a base class for token types, to get a place to stuff all common behaviour. I've added default attributes for storing the token type name (the "id" attribute) and the token value (for literal and name tokens), as well as a few attributes for the syntax tree. This class is also a convenient place to provide default implementations of the "nud" and "led" methods.

```
class symbol_base(object):

    id = None # node/token type name
    value = None # used by literals
    first = second = third = None # used by tree nodes

    def nud(self):
        raise SyntaxError(
            "Syntax error (%r)." % self.id
        )

    def led(self, left):
        raise SyntaxError(
            "Unknown operator (%r)." % self.id
        )

    def __repr__(self):
        if self.id == "(name)" or self.id == "(literal)":
            return "(%s %s)" % (self.id[1:-1], self.value)
        out = [self.id, self.first, self.second, self.third]
        out = map(str, filter(None, out))
        return "(" + " ".join(out) + ")"
```

Next, we need a token type factory:

```
symbol_table = {}
```

```
def symbol(id, bp=0):
    try:
        s = symbol_table[id]
    except KeyError:
        class s(symbol_base):
            pass
        s.__name__ = "symbol-" + id # for debugging
        s.id = id
        s.lbp = bp
        symbol_table[id] = s
    else:
        s.lbp = max(bp, s.lbp)
    return s
```

This function takes a token identifier and an optional binding power, and creates a new class if necessary. The identifier and the binding power are inserted as class attributes, and will thus be available in all instances of that class. If the function is called for a symbol that's already registered, it just updates the binding power; this allows us to define different parts of the symbol's behaviour in different places, as we'll see later.

We can now populate the registry with the symbols we're going to use:

```
symbol("(literal)")
symbol("+", 10); symbol("-", 10)
symbol("*", 20); symbol("/", 20)
symbol("**", 30)
symbol("(end)")
```

To simplify dispatching, we're using the token strings as identifiers; the identifiers for the "(literal)" and "(end)" symbols (which replaces the literal_token and end_token classes used earlier) are strings that won't appear as ordinary tokens.

We also need to update the tokenizer, to make it use classes from the registry:

```
def tokenize(program):
    for number, operator in token_pat.findall(program):
        if number:
            symbol = symbol_table["(literal)"]
            s = symbol()
            s.value = number
            yield s
        else:
            symbol = symbol_table.get(operator)
            if not symbol:
                raise SyntaxError("Unknown operator")
            yield symbol()
    symbol = symbol_table["(end)"]
    yield symbol()
```

Like before, the literal class is used as a common class for all literal values. All other tokens have their own classes.

Now, all that's left is to define "nud" and "led" methods for the symbols that need additional behavior. To do that, we can define them as ordinary functions, and then simply plug them into the symbol classes, one by one. For example, here's the "led" method for addition:

```
def led(self, left):
    self.first = left
    self.second = expression(10)
    return self
symbol("+").led = led
```

That last line fetches the class from the symbol registry, and adds the function to it. Here are a few more “led” methods:

```
def led(self, left):
    self.first = left
    self.second = expression(10)
    return self
symbol("-").led = led

def led(self, left):
    self.first = left
    self.second = expression(20)
    return self
symbol("*").led = led

def led(self, left):
    self.first = left
    self.second = expression(20)
    return self
symbol("/").led = led
```

They do look pretty similar, don’t they? The only thing that differs is the binding power, so we can simplify things quite a bit by moving the repeated code into a helper function:

```
def infix(id, bp):
    def led(self, left):
        self.first = left
        self.second = expression(bp)
        return self
    symbol(id, bp).led = led
```

Given this helper, we can now replace the “led” functions above with four simple calls:

```
infix("+", 10); infix("-", 10)
infix("*", 20); infix("/", 20)
```

Likewise, we can provide helper functions for the “nud” methods, and for operators with right associativity:

```
def prefix(id, bp):
    def nud(self):
        self.first = expression(bp)
        self.second = None
        return self
    symbol(id).nud = nud

prefix("+", 100); prefix("-", 100)

def infix_r(id, bp):
    def led(self, left):
        self.first = left
        self.second = expression(bp-1)
        return self
    symbol(id, bp).led = led

infix_r("**", 30)
```

Finally, the literal symbol must be fitted with a “nud” method that returns the symbol itself. We can use a plain lambda for this:

```
symbol("(literal)").nud = lambda self: self
```

Note that most of the above is general-purpose plumbing; given the helper functions, the actual parser definition boils down to the following six lines:

```
infix("+", 10); infix("-", 10)
infix("*", 20); infix("/", 20)
infix_r("**", 30)
```

```
prefix("+", 100); prefix("-", 100)

symbol("(literal)").nud = lambda self: self
symbol("(end)")
```

Running this produces the same result as before:

```
>>> parse("1")
(literal 1)
>>> parse("+1")
(+ (literal 1))
>>> parse("1+2")
(+ (literal 1) (literal 2))
>>> parse("1+2+3")
(+ (+ (literal 1) (literal 2)) (literal 3))
>>> parse("1+2*3")
(+ (literal 1) (* (literal 2) (literal 3)))
>>> parse("1*2+3")
(+ (* (literal 1) (literal 2)) (literal 3))
```

Parsing Python Expressions

To get a somewhat larger example, let's tweak the parser so it can parse a subset of the Python expression syntax, similar to the syntax shown in the grammar snippet at the start of this article.

To do this, we first need a fancier tokenizer. The obvious choice is to build on Python's **tokenize** module:

```
def tokenize_python(program):
    import tokenize
    from cStringIO import StringIO
    type_map = {
        tokenize.NUMBER: "(literal)",
        tokenize.STRING: "(literal)",
        tokenize.OP: "(operator)",
        tokenize.NAME: "(name)",
        }
    for t in tokenize.generate_tokens(StringIO(program).next):
        try:
            yield type_map[t[0]], t[1]
        except KeyError:
            if t[0] == tokenize.ENDMARKER:
                break
            else:
                raise SyntaxError("Syntax error")
    yield "(end)", "(end)"

def tokenize(program):
    for id, value in tokenize_python(program):
        if id == "(literal)":
            symbol = symbol_table[id]
            s = symbol()
            s.value = value
        else:
            # name or operator
            symbol = symbol_table.get(value)
            if symbol:
                s = symbol()
            elif id == "(name)":
                symbol = symbol_table[id]
                s = symbol()
                s.value = value
            else:
```

```
                raise SyntaxError("Unknown operator (%r)" % id)
        yield s
```

This tokenizer is split into two parts; one language-specific parser that turns the source program into a stream of literals, names, and operators, and a second part that turns those into a token instances. The latter checks both operators and names against the symbol table (to handle keyword operators), and uses a psuedo-symbol (“(name)”) for all other names.

You could combine the two tasks into a single function, but the separation makes it a bit easier to test the parser, and also makes it possible to reuse the second part for other syntaxes.

We can test the new tokenizer with the old parser definition:

```
>>> parse("1+2") (+ (literal 1) (literal 2)) >>> parse(1+2+3") (+ (+ (literal 1) (literal 2)) (literal 3
```

The new tokenizer supports more literals, so our parser does that too, without any extra work. And we’re still using the 10-line **expression** implementation we introduced at the beginning of this article.

The Python Expression Grammar

So, let’s do something about the grammar. We could figure out the correct expression grammar from the grammar snippet shown earlier, but there’s a more practical description in the section “Evaluation order” in Python’s language reference. The table in that section lists all expression operators in precedence order, from lowest to highest. Here are the corresponding definitions (starting at binding power 20):

```
symbol("lambda", 20)
symbol("if", 20) # ternary form

infix_r("or", 30); infix_r("and", 40); prefix("not", 50)

infix("in", 60); infix("not", 60) # in, not in
infix("is", 60) # is, is not
infix("<", 60); infix("⩽", 60)
infix(">", 60); infix("⩾", 60)
infix("<>", 60); infix("≠", 60); infix("═", 60)

infix("|", 70); infix("^", 80); infix("&", 90)

infix("<<", 100); infix(">>", 100)

infix("+", 110); infix("-", 110)

infix("*", 120); infix("/", 120); infix("//", 120)
infix("%", 120)

prefix("-", 130); prefix("+", 130); prefix("~", 130)

infix_r("**", 140)

symbol(".", 150); symbol("[", 150); symbol("(", 150)
```

These 16 lines define the syntax for 35 operators, and also provide behaviour for most of them.

However, tokens defined by the **symbol** helper have no intrinsic behaviour; to make them work, additional code is needed. There are also some intricacies caused by limitations in Python’s tokenizer; more about those later.

But before we start working on those symbols, we need to add behaviour to the pseudo-tokens too:

```
symbol("(literal)").nud = lambda self: self
symbol("(name)").nud = lambda self: self
symbol("(end)")
```

We can now do a quick sanity check:

```
>>> parse("1+2")
(+ (literal 1) (literal 2))
>>> parse("2<<3")
(<< (literal 2) (literal 3))
```

Parenthesized Expressions

Let's turn our focus to the remaining symbols, and start with something simple: parenthesized expressions. They can be implemented by a "nud" method on the "(" token:

```
def nud(self):
    expr = expression()
    advance(")")
    return expr
symbol("(").nud = nud
```

The "advance" function used here is a helper function that checks that the current token has a given value, before fetching the next token.

```
def advance(id=None):
    global token
    if id and token.id ≠ id:
        raise SyntaxError("Expected %r" % id)
    token = next()
```

The ")" token must be registered; if not, the tokenizer will report it as an invalid token. To register it, just call the symbol function:

```
symbol(")")
```

Let's try it out:

```
>>> 1+2*3
(+ (literal 1) (* (literal 2) (literal 3)))
>>> (1+2)*3
(* (+ (literal 1) (literal 2)) (literal 3))
```

Note that the "nud" method returns the inner expression, so the "(" node won't appear in the resulting syntax tree.

Also note that we're cheating here, for a moment: the "(" prefix has two meanings in Python; it can either be used for grouping, as above, or to create tuples. We'll fix this below.

Ternary Operators

Most custom methods look more or less exactly like their recursive-descent counterparts, and the code for inline if-else is no different:

```
def led(self, left):
    self.first = left
    self.second = expression()
    advance("else")
    self.third = expression()
    return self
symbol("if").led = led
```

Again, we need to register the extra token before we can try it out:

```
symbol("else")

>>> parse("1 if 2 else 3")
(if (literal 1) (literal 2) (literal 3))
```

Attribute and Item Lookups

To handle attribute lookups, the “.” operator needs a “led” method. For convenience, this version verifies that the period is followed by a proper name token (this check could be made at a later stage as well):

```
def led(self, left):
    if token.id ≠ "(name)":
        SyntaxError("Expected an attribute name.")
    self.first = left
    self.second = token
    advance()
    return self
symbol(".").led = led

>>> parse("foo.bar")
(. (name foo) (name bar))
```

Item access is similar; just add a “led” method to the “[” operator. And since “]” is part of the syntax, we need to register that symbol as well.

```
symbol("]") def led(self, left): self.first = left self.second = expression() advance("]") return self s
```

Note that we’re ending up with lots of code of the form:

```
def led(self, left):
    ...
symbol(id).led = led
```

which is a bit inconvenient, if not else because it violates the “don’t repeat yourself” rule (the name of the method appears three times). A simple decorator solves this:

```
def method(s):
    assert issubclass(s, symbol_base)
    def bind(fn):
        setattr(s, fn.__name__, fn)
    return bind
```

This decorator picks up the function name, and attaches that to the given symbol. This puts the symbol name before the method definition, and only requires you to write the method name once.

```
@method(symbol(id))
def led(self, left):
    ...
```

We’ll use this in the following examples. The other approach isn’t much longer, so you can still use it if you need to target Python 2.3 or older. Just watch out for typos.

Function Calls

A function call consists of an expression followed by a comma-separated expression list, in parentheses. By treating the left parentesis as a binary operator, parsing this is straight-forward:

```
symbol(")"); symbol(",") @method(symbol("(")) def led(self, left): self.first = left self.second = [] if
```

This is a bit simplified; keyword arguments and the “*” and “**” forms are not supported by this version. To handle keyword arguments, look for an “=” after the first expression, and if that’s found, check that the subtree is a plain name, and then call expression again to get the default value. The other forms could be handled by “nud” methods on the corresponding operators, but it’s probably easier to handle these too in this method.

Lambdas

Lambdas are also quite simple. Since the “lambda” keyword is a prefix operator, we’ll implement it using a “nud” method:

```
symbol(":")

@method(symbol("lambda"))
def nud(self):
    self.first = []
    if token.id ≠ ":":
        argument_list(self.first)
    advance(":")
    self.second = expression()
    return self

def argument_list(list):
    while 1:
        if token.id ≠ "(name)":
            SyntaxError("Expected an argument name.")
        list.append(token)
        advance()
        if token.id ≠ ",":
            break
        advance(",")
```

```
>>> parse("lambda a, b, c: a+b+c")
(lambda [(name a), (name b), (name c)]
    (+ (+ (name a) (name b)) (name c)))
```

Again, the argument list parsing is a bit simplified; it doesn’t handle default values and the “*” and “**” forms. See above for implementation hints. Also note that there’s no scope handling at the parser level in this implementation. See Crockford’s article for more on that topic.

Constants

Constants can be handled as literals; the following “nud” method changes the token instance to a literal node, and inserts the token itself as the literal value:

```
def constant(id):
    @method(symbol(id))
    def nud(self):
        self.id = "(literal)"
        self.value = id
        return self

constant("None")
constant("True")
constant("False")
```

```
>>> parse("1 is None")
(is (literal 1) (literal None))
>>> parse("True or False")
(or (literal True) (literal False))
```

Multi-Token Operators

Python has two multi-token operators, “is not” and “not in”, but our parser doesn’t quite treat them correctly:

```
>>> parse("1 is not 2")
(is (literal 1) (not (literal 2)))
```

The problem is that the standard **tokenize** module doesn’t understand this syntax, so it happily returns these operators as two separate tokens:

```
>>> list(tokenize("1 is not 2"))
[(literal 1), (is), (not), (literal 2), ((end))]
```

In other words, “1 is not 2” is handled as “1 is (not 2)”, which isn’t the same thing:

```
>>> 1 is not 2
True
>>> 1 is (not 2)
False
```

One way to fix this is to tweak the tokenizer (e.g. by inserting a combining filter between the raw Python parser and the token instance factory), but it’s probably easier to fix this with custom “led” methods on the “is” and “not” operators:

```
@method(symbol("not"))
def led(self, left):
    if token.id != "in":
        raise SyntaxError("Invalid syntax")
    advance()
    self.id = "not in"
    self.first = left
    self.second = expression(60)
    return self

@method(symbol("is"))
def led(self, left):
    if token.id == "not":
        advance()
        self.id = "is not"
    self.first = left
    self.second = expression(60)
    return self
```

```
>>> parse("1 in 2")
(in (literal 1) (literal 2))
>>> parse("1 not in 2")
(not in (literal 1) (literal 2))
>>> parse("1 is 2")
(is (literal 1) (literal 2))
>>> parse("1 is not 2")
(is not (literal 1) (literal 2))
```

This means that the “not” operator handles both unary “not” and binary “not in”.

Tuples, Lists, and Dictionary Displays

As noted above, the “(” prefix serves two purposes in Python; it’s used for grouping, and to create tuples (it’s also used as a binary operator, for function calls). To handle tuples, we need to replace the “nud” method with a version that can distinguish between tuples and a plain parenthesized expression.

Python’s tuple-forming rules are simple; if a pair of parenteses are empty, or contain at least one comma, it’s a tuple. Otherwise, it’s an expression. Or in other

words:

- () is a tuple
- (1) is a parenthesized expression
- (1,) is a tuple
- (1, 2) is a tuple

Here's a "nud" replacement that implements these rules:

```
@method(symbol("(")) def nud(self): self.first = [] comma = False if token.id != ")": while 1: if token.
```

Lists and dictionaries are a bit simpler; they're just plain lists of expressions or expression pairs. Don't forget to register the extra tokens.

```
symbol("]") @method(symbol("[")) def nud(self): self.first = [] if token.id != "]": while 1: if token.id
```

Note that Python allows you to use optional trailing commas when creating lists, tuples, and dictionaries; an extra if-statement at the beginning of the collection loop takes care of that case.

Summary

At roughly 250 lines of code (including the entire parser machinery), there are still a few things left to add before we can claim to fully support the Python 2.5 expression syntax, but we've covered a remarkably large part of the syntax with very little work.

And as we've seen thoughout this article, parsers using this algorithm and implementation approach are readable, easy to extend, and, as we'll see in a moment, surprisingly fast. While this article has focussed on expressions, the algorithm can be easily extended for statement-oriented syntaxes. See Crockford's article for one way to do that.

All in all, Pratt's parsing algorithm is a great addition to the Python parsing toolbox, and the implementation strategy outlined in this article is a simple way to quickly implement such parsers.

Performance

As we've seen, the parser makes only a few Python calls per token, which means that it should be pretty efficient (or as Pratt put it, "efficient in practice if not in theory").

To test practical performance, I picked a 456 character long Python expression (about 300 tokens) from the Python FAQ, and parsed it with a number of different tools. Here are some typical results under Python 2.5:

- topdown parse (to abstract syntax tree): 4.0 ms
- built-in parse (to tuple tree): 0.60 ms
- built-in compile (to code object): 0.68 ms
- compiler parse (to abtract syntax tree): 4.8 ms
- compiler compile (to code object): 18 ms

If we tweak the parser to work on a precomputed list of tokens (obtained by running "list(tokenize_python(program))"), the parsing time drops to just under 0.9 ms. In other words, only about one fourth of the time for the full parse is spent on token instance creation, parsing, and tree building. The rest is almost entirely spent in

Python's **tokenize** module. With a faster tokenizer, this algorithm would get within 2x or so from Python's built-in tokenizer/parser.

The built-in parse test is in itself quite interesting; it uses Python's internal tokenizer and parser module (both of which are written in C), and uses the **parser** module (also written in C) to convert the internal syntax tree object to a tuple tree. This is fast, but results in a remarkably undecipherable low-level tree:

```
>>> parser.st2tuple(parser.expr("1+2"))
(258, (326, (303, (304, (305, (306, (307, (309,
(310, (311, (312, (313, (314, (315, (316, (317,
(2, '1'))))), (14, '+'), (314, (315, (316, (317,
(2, '2'))))))))))))))), (4, ''), (0, ''))
```

(In this example, 2 means number, 14 means plus, 4 is newline, and 0 is end of program. The 3-digit numbers represent intermediate rules in the Python grammar.)

The compiler parse test uses the parse function from the **compiler** package instead; this function uses Python's internal tokenizer and parser, and then turns the resulting low-level structure into a much nicer abstract tree:

```
>>> import compiler
>>> compiler.parse("1+2", "eval")
Expression(Add((Const(1), Const(2))))
```

This conversion (done in Python) turns out to be more work than parsing the expression with the topdown parser; with the code in this article, we get an abstract tree in about 85% of the time, despite using a really slow tokenizer.

Code Notes

The code in this article uses global variables to hold parser state (the "token" variable and the "next" helper). If you need a thread-safe parser, these should be moved to a context object. This will result in a slight performance hit, but there are some surprising ways to compensate for that, by trading a little memory for performance. More on that in a later article.

All code for the interpreters and translators shown in this article is included in the article itself. Assorted code samples are also available from:

http://svn.effbot.org/public/stuff/sandbox/topdown