Compiling Little Languages in Python

John Aycock
Department of Computer Science
University of Victoria
Victoria, B.C., Canada
aycock@csc.uvic.ca

Abstract

"Little languages" such as configuration files or HTML documents are commonplace in computing. This paper divides the work of implementing a little language into four parts, and presents a framework which can be used to easily conquer the implementation of each. The pieces of the framework have the unusual property that they may be extended through normal object-oriented means, allowing features to be added to a little language simply by subclassing parts of its compiler.

1 Introduction

Domain-specific languages, or "little languages," are frequently encountered when dealing with computers [3]. Configuration files, HTML documents, shell scripts — all are little structured languages, yet may lack the generality and features of full-blown programming languages. Whether writing an interpreter for a little language, or compiling a little language into another language, compiler techniques can be used.

In many cases, an extremely fast compiler is not needed, especially if the input programs tend to be small. Instead, issues can predominate such as compiler development time, maintainability of the compiler, and the ability to easily add new language features. This is Python's strong suit.

This paper describes some successful techniques I developed while working on two compilers for little languages: one for a subset of Java, the other an optimizing compiler for Guide, a CGI-programming language [12]. The net result is a framework which can be used to implement little languages easily.

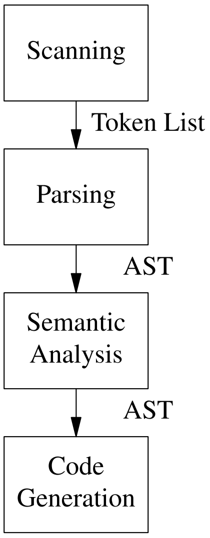

Figure 1: Compiler model.

2 Model of a Compiler

Like most nontrivial pieces of software, compilers are generally broken down into more manageable modules, or phases. The design issues involved and the details of each phase are too numerous to discuss here in depth; there are many excellent books on the subject, such as [1] and [2].

I began with a simple model of a compiler having only four phases, as shown in Figure 1:

1. Scanning, or lexical analysis. Breaks the input stream into a list of tokens. For example, the expression "2 + 3 * 5" can be broken up into five tokens: number plus number times number. The values 2, 3, and 5 are attributes associated with the corresponding number token.

2. Parsing, or syntax analysis. Ensures that a list of tokens has valid syntax according to a gram-

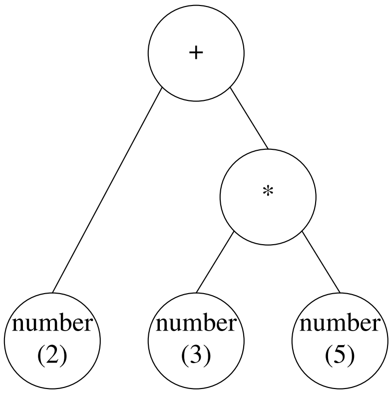

Figure 2: Abstract syntax tree (AST).

mar — a set of rules that describes the syntax of the language. For the above example, a typical expression grammar would be:

```
expr ::= expr + term
expr ::= term
term ::= term * factor
term ::= factor
factor ::= number
```

In English, this grammar's rules say that an expression can be an expression plus a term, an expression may be a term by itself, and so on. Intuitively, the symbol on the left-hand side of "::=" may be thought of as a variable for which the symbols on the right-hand side may be substituted. Symbols that don't appear on any left-hand side — like +, *, and number — correspond to the tokens from the scanner.

The result of my parsing is an abstract syntax tree (AST), which represents the input program. For "2 + 3 * 5," the AST would look like the one in Figure 2.

3. Semantic analysis. Traverses the AST one or more times, collecting information and checking that the input program had no semantic errors. In a typical programming language, this phase would detect things like type conflicts, redefined identifiers, mismatched function parameters, and numerous other errors. The information gathered may be stored in a global symbol table, or attached as attributes to the nodes of the AST itself.

4. Code generation. Again traversing the AST, this phase may directly interpret the program, or output code in C or assembly which would implement the input program. For expressions as simple as those in the example, they could be evaluated on the fly in this phase.

Each phase performs a well-defined task, and passes a data structure on to the next phase. Note that information only flows one way, and that each phase runs to completion before the next one starts [1]. This is in contrast to oft-used techniques which have a symbiosis between scanning and parsing, where not only may several phases be working concurrently, but a later phase may send some feedback to modify the operation of an earlier phase.

Certainly all little language compilers won't fit this model, but it is extremely clean and elegant for those that do. The main function of the compiler, for instance, distills into three lines which reflect the compiler's structure:

```
f = open(filename)
generate(semantic(parse(scan(f))))
f.close()
```

In the remainder of this paper, I will examine each of the above four phases, showing how my framework can be used to implement the little expression language above. Following this will be a discussion of some of the inner workings of the framework's classes.

3 The Framework

A common theme throughout this framework is that the user should have to do as little work as possible. For each phase, my framework supplies a class which performs most of the work. The user's job is simply to create subclasses which customize the framework.

3.1 Lexical Analysis

Lexical analyzers, or scanners, are typically implemented one of two ways. The first way is to write the scanner by hand; this may still be the method of choice for very small languages, or where use of a tool to generate scanners automatically is not possible. The second method is to use a scanner generator tool, like lex [11], which takes a high-level description of the permitted tokens, and produces a finite state machine which implements the scanner.

Finite state machines are equivalent to regular expressions; in fact, one uses regular expressions to

[1] There is some anecdotal evidence that parts of production compilers may be moving towards a similar model [18].

specify tokens to scanner generators! Since Python has regular expression support, it is natural to use them to specify tokens. (As a case in point, the Python module "tokenize" has regular expressions to tokenize Python programs.)

So GenericScanner, my generic scanner class, requires a user to create a subclass of it in which they specify the regular expressions that the scanner should look for. Furthermore, an "action" consisting of arbitrary Python code can be associated with each regular expression — this is typical of scanner generators, and allows work to be performed based on the type of token found.

Below is a simple scanner to tokenize expressions. The parameter to the action routines is a string containing the part of the input that was matched by the regular expression.

```
class SimpleScanner(GenericScanner):
    def __init__(self):
        GenericScanner.__init__(self)

    def tokenize(self, input):
        self.rv = []
        GenericScanner.tokenize(self, input)
        return self.rv

    def t_whitespace(self, s):
        r' \s+ '
        pass

    def t_op(self, s):
        r' \+ | \* '
        self.rv.append(Token(type=s))

    def t_number(self, s):
        r' \d+ '
        t = Token(type='number', attr=s)
        self.rv.append(t)
```

Each method whose name begins with "t_" is an action; the regular expression for the action is placed in the method's documentation string. (The reason for this unusual design is explained in Section 4.1.)

When the tokenize method is called, a list of Token instances is returned, one for each operator and number found. The code for the Token class is omitted; it is a simple container class with a type and an optional attribute. White space is skipped by SimpleScanner, since its action code does nothing. Any unrecognized characters in the input are matched by a default pattern, declared in the action GenericScanner.t_default. This default method can of course be overridden in a subclass. A trace of SimpleScanner on the input "2 + 3 * 5" is shown in Table 1.

Input	Method	Token Added
2	t_number	number (attribute 2)
space	t_whitespace	
+	t_op	+
space	t_whitespace	
3	t_number	number (attribute 3)
space	t_whitespace	
*	t_op	*
space	t_whitespace	
5	t_number	number (attribute 5)

Table 1: Trace of SimpleScanner.

Scanners made with GenericScanner are extensible, meaning that new tokens may be recognized simply by subclassing. To extend SimpleScanner to recognize floating-point number tokens is easy:

```
class FloatScanner(SimpleScanner):
    def __init__(self):
        SimpleScanner.__init__(self)

    def t_float(self, s):
        r' \d+ \. \d+ '
        t = Token(type='float', attr=s)
        self.rv.append(t)
```

How are these classes used? Typically, all that is needed is to read in the input program, and pass it to an instance of the scanner:

```
def scan(f):
    input = f.read()
    scanner = FloatScanner()
    return scanner.tokenize(input)
```

Once the scanner is done, its result is sent to the parser for syntax analysis.

3.2 Syntax Analysis

The outward appearance of GenericParser, my generic parser class, is similar to that of GenericScanner.

A user starts by creating a subclass of GenericParser, containing special methods which are named with the prefix "p_". These special methods encode grammar rules in their documentation strings; the code in the methods are actions which get executed when one of the associated grammar rules are recognized by GenericParser.

The expression parser subclass is shown below. Here, the actions are building the AST for the input program. AST is also a simple container class; each instance of AST corresponds to a node in the tree, with a node type and possibly child nodes.

```
class ExprParser(GenericParser):
    def __init__(self, start='expr'):
        GenericParser.__init__(self, start)

    def p_expr_1(self, args):
        ' expr ::= expr + term '
        return AST(type=args[1],
                   left=args[0],
                   right=args[2])

    def p_expr_2(self, args):
        ' expr ::= term '
        return args[0]

    def p_term_1(self, args):
        ' term ::= term * factor '
        return AST(type=args[1],
                   left=args[0],
                   right=args[2])

    def p_term_2(self, args):
        ' term ::= factor '
        return args[0]

    def p_factor_1(self, args):
        ' factor ::= number '
        return AST(type=args[0])

    def p_factor_2(self, args):
        ' factor ::= float '
        return AST(type=args[0])
```

The grammar's start symbol is passed to the constructor.

ExprParser builds the AST from the bottom up. Figure 3 shows the AST in Figure 2 being built, and the sequence in which ExprParser's methods are invoked.

The "args" passed in to the actions are based on a similar idea used by yacc [11], a prevalent parser generator tool. Each symbol on a rule's right-hand side has an attribute associated with it. For token symbols like +, this attribute is the token itself. All other symbols' attributes come from the return values of actions which, in the above code, means that they are subtrees of the AST. The index into args comes from the position of the symbol in the rule's right-hand side. In the running example, the call

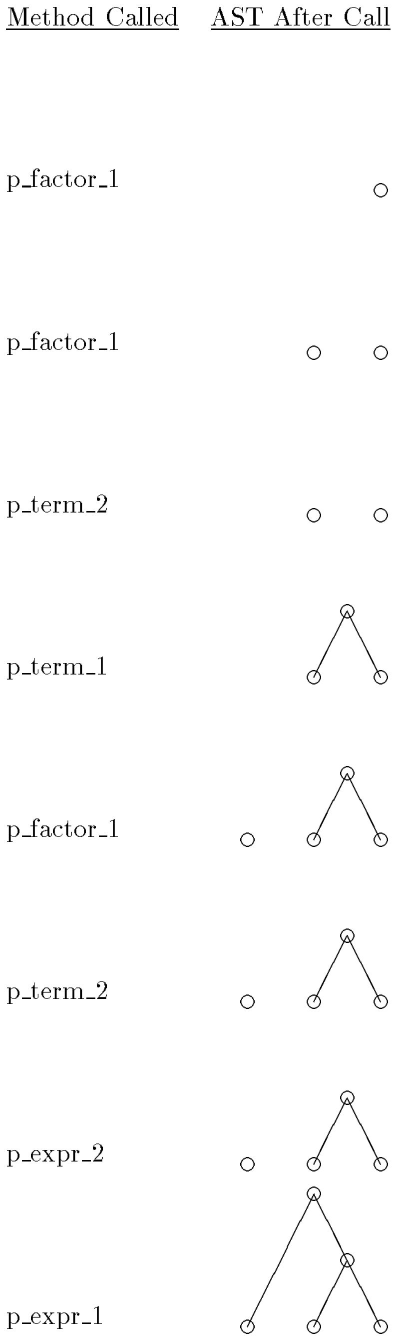

Figure 3: AST construction.

to p_expr_1 has `len(args) == 3`: args[0] is expr's attribute, the left subtree of + in the AST; args[1] is +'s attribute, the token +; args[2] is term's attribute, the right subtree of + in the AST.

The routine to use this subclass is straightforward:

```
def parse(tokens):
    parser = ExprParser()
    return parser.parse(tokens)
```

Although omitted for brevity, ExprParser can be subclassed to add grammar rules and actions, the same way the scanner was subclassed.

After syntax analysis, the parser has produced an AST, and verified that the input program adheres to the grammar rules. Next, the input's meaning must be checked by the semantic analyzer.

3.3 Semantic Analysis

Semantic analysis is performed by traversing the AST. Rather than spread code to traverse an AST all over the compiler, I have a single base class, ASTTraversal, which knows how to walk the tree. Subclasses of ASTTraversal supply methods which get called depending on what type of node is encountered.

To determine which method to invoke, ASTTraversal will first look for a method with the same name as the node type (plus the prefix "n_"), then will fall back on an optional default method if no more specific method is found.

Of course, ASTTraversal can supply many different traversal algorithms. I have found three useful: preorder, postorder, and a pre/postorder combination. (The latter allows methods to be called both on entry to, and exit from, a node.)

For example, say that we want to forbid the mixing of floating-point and integer numbers in our expressions:

```
class TypeCheck(ASTTraversal):
    def __init__(self, ast):
        ASTTraversal.__init__(self, ast)
        self.postorder()

    def n_number(self, node):
        node.exprType = 'number'
    def n_float(self, node):
        node.exprType = 'float'

    def default(self, node):
        # this handles + and * nodes
        leftType = node.left.exprType
        rightType = node.right.exprType
        if leftType != rightType:
            raise 'Type error.'
        node.exprType = leftType
```

I found the semantic checking code easier to write and understand by taking the (admittedly less efficient) approach of making multiple traversals of the AST — each pass performs a single task.

TypeCheck is invoked from a small glue routine:

```
def semantic(ast):
    TypeCheck(ast)
    #
    #  Any other ASTTraversal classes
    #  for semantic checking would be
    #  instantiated here...
    #
    return ast
```

After this phase, I have an AST for an input program that is lexically, syntactically, and semantically correct — but that does nothing. The final phase, code generation, remedies this.

3.4 Code Generation

The term "code generation" is somewhat of a misnomer. As already mentioned, this phase will traverse the AST and implement the input program, either directly through interpretation, or indirectly by emitting some code.

Our expressions, for instance, can be easily interpreted:

```
class Interpret(ASTTraversal):
    def __init__(self, ast):
        ASTTraversal.__init__(self, ast)
        self.postorder()
        print ast.value

    def n_number(self, node):
        node.value = int(node.attr)
    def n_float(self, node):
        node.value = float(node.attr)

    def default(self, node):
        left = node.left.value
        right = node.right.value

        if node.type == '+':
            node.value = left + right
        else:
            node.value = left * right
```

In contrast, my two compilers use an ASTTraversal to output an intermediate representation (IR) which is effectively a machine-independent assembly language. This IR then gets converted into MIPS assembly code in one compiler, C++ code in the other.

I am considering ways to incorporate more sophisticated code generation methods into this framework, such as tree pattern matching with dynamic programming [8].

4 Inner Workings

4.1 Reflection

Extensibility presents some interesting design challenges. The generic classes in the framework, without any modifications made to them, must be able to divine all the information and actions contained in their subclasses, subclasses that didn't exist when the generic classes were created.

Fortunately, an elegant mechanism exists in Python to do just this: reflection. Reflection refers to the ability of a Python program to query and modify itself at run time (this feature is also present in other languages, like Java and Smalltalk).

Consider, for example, my generic scanner class. GenericScanner searches itself and its subclasses at run time for methods that begin with the prefix "t_." These methods are the scanner's actions. The regular expression associated with the actions is specified using a well-known method attribute that can be queried at run time — the method's documentation string.

This wanton abuse of documentation strings can be rationalized. Documentation strings are a method of associating meta-information — comments — with a section of code. My framework is an extension of that idea. Instead of comments intended for humans, however, I have meta-information intended for use by my framework. As the number of reflective Python applications grows, it may be worthwhile to add more formal mechanisms to Python to support this task.

4.2 GenericScanner

Internally, GenericScanner works by constructing a single regular expression which is composed of all the smaller regular expressions it has found in the action methods' documentation strings. Each component regular expression is mapped to its action using Python's symbolic group facility.

Unfortunately, there is a small snag. Python follows the Perl semantics for regular expressions rather than the POSIX semantics, which means it follows the "first then longest" rule — the leftmost part of a regular expression that matches is always taken, rather than using the longest match. In the above example, if GenericScanner were to order the regular expression so that "`\d+`" appeared before "`\d+\.\d+`", then the input 123.45 would match as the number 123, rather than the floating-point number 123.45. To work around this, GenericScanner makes two guarantees:

1. A subclass' patterns will be matched before any in its parent classes.

2. The default pattern for a subclass, if any, will be matched only after all other patterns in the subclass have been tried.

One obvious change to GenericScanner is to automate the building of the list of tokens — each "t_" method could return a list of tokens which would be appended to the scanner's list of tokens. The reason this is not done is because it would limit potential applications of GenericScanner. For example, in one compiler I used a subclass of GenericScanner as a preprocessor which returned a string; another scanner class then broke that string into a list of tokens.

4.3 GenericParser

GenericParser is actually more powerful than was alluded to in Section 3.2. At the cost of greater coupling between methods, actions for similar rules may be combined together rather than having to duplicate code — my original version of ExprParser is shown below.

```
class ExprParser(GenericParser):
    def __init__(self, start='expr'):
        GenericParser.__init__(self, start)

    def p_expr_term(self, args):
        '''
            expr ::= expr + term
            term ::= term * factor
        '''
        return AST(type=args[1],
                   left=args[0],
                   right=args[2])

    def p_expr_term_2(self, args):
        '''
```

```
        expr ::= term
        term ::= factor
    '''
    return args[0]

def p_factor(self, args):
    '''
        factor ::= number
        factor ::= float
    '''
    return AST(type=args[0])
```

Taking this to extremes, if a user is *only* interested in parsing and doesn't require an AST, ExprParser could be written:

```
class ExprParser(GenericParser):
    def __init__(self, start='expr'):
        GenericParser.__init__(self, start)

    def p_rules(self, args):
        '''
            expr ::= expr + term
            expr ::= term
            term ::= term * factor
            term ::= factor
            factor ::= number
            factor ::= float
        '''
```

In theory, GenericParser could use any parsing algorithm for its engine. However, I chose the Earley parsing algorithm [6] which has several nice properties for this application [10]:

1. It is one of the most general algorithms known; it can parse all context-free grammars whereas the more popular LL and LR techniques cannot. This is important for easy extensibility; a user should ideally be able to subclass a parser without worrying about properties of the resulting grammar.

2. It generates all its information at run-time, rather than having to precompute sets and tables. Since the grammar rules aren't known until run-time, this is just as well!

Unlike most other parsing algorithms, Earley's method parses ambiguous grammars. Currently, ambiguity presents a problem since it is not clear which actions should be invoked. Future versions of GenericParser will have an ambiguity-resolution scheme to address this.

To accommodate a variety of possible parsing algorithms (including the one I used), GenericParser only makes one guarantee with respect to when the rules' actions are executed. A rule's action is executed only after all the attributes on the rule's right-hand side are fully computed. This condition is sufficient to allow the correct construction of ASTs.

4.4 ASTTraversal

ASTTraversal is the least unusual of the generic classes. It could be argued that its use of reflection is superfluous, and the same functionality could be achieved by having its subclasses provide a method for every type of AST node; these methods could call a default method themselves if necessary.

The problems with this non-reflective approach are threefold. First, it introduces a maintenance issue: any additional node types added to the AST require all ASTTraversal's subclasses to be changed. Second, it forces the user to do more work, as methods for all node types must be supplied; my experience, especially for semantic checking, is that only a small set of node types will be of interest for a given subclass. Third, some node types may not map nicely into Python method names — I prefer to use node types that reflect the little language's syntax, like +, and it isn't possible to have methods named "n_+"[2]. This latter point is where it is useful to have ASTTraversal reflectively probe a subclass and automatically invoke the default method.

4.5 Design Patterns

Although developed independently, the use of reflection in my framework is arguably a specialization of the Reflection pattern [4]. I speculate that there are many other design patterns where reflection can be exploited. To illustrate, ASTTraversal wound up somewhere between the Default Visitor [13] and Reflection patterns, although it was originally inspired by the Visitor pattern [9].

Two other design patterns can be applied to my framework too. First, the entire framework could be organized explicitly as a Pipes and Filters pattern [4]. Second, the generic classes could support interchangeable algorithms via the Strategy pattern [9]; parsing algorithms, in particular, vary widely in their characteristics, so allowing different algorithms could be a boon to an advanced user.

[2] Not directly, anyway...

5 Comparison to Other Work

The basis of this paper is the observation that little languages, and the need to implement them, are recurring problems. Not all authors even agree on this point — Shivers [16] presents an alternative to little languages and a Scheme-based implementation framework. Tcl was also developed to address the proliferation of little languages [14].

Other Python packages exist to automate parts of scanning and parsing. PyLR [5] uses a parsing engine written in C to accelerate parsing; kwParsing [17] automates the implementation of common programming language features at the cost of a more complex API. Both require precomputation of scanning and parsing information. YAPPS [15] uses the weakest parsing algorithm of the surveyed packages, and its author notes 'It is not fast, powerful, or particularly flexible.' There are occasional references to the PyBison package, which I was unable to locate.

For completeness, the mcf.pars package [7] is an interesting nontraditional system based on generalized pattern matching, but is sufficiently different from my framework to preclude any meaningful comparisons.

6 Not-so-Little Languages

This paper has presented a framework I have developed to build compilers in Python. It uses reflection and design patterns to produce compilers which can be easily extended using traditional object-oriented methods.

At present, this framework has proved its effectiveness in the implementation of two "little languages." I plan to further test this framework by using it to build a compiler for a larger language — Python.

Availability

The source code for the framework and the example used in this paper is available at `http://www.csc.uvic.ca/~aycock`.

Acknowledgments

I would like to thank Nigel Horspool and Shannon Jaeger for their comments on early drafts of this paper. Mike Zastre made several suggestions which improved Figure 3, and the anonymous referees supplied valuable feedback.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge, 1998.

[3] J. Bentley. Little Languages. In *More Programming Pearls*, pages 83–100. Addison-Wesley, 1988.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

[5] S. Cotton. PyLR. `http://starship.skyport.net/crew/scott/PyLR.html`.

[6] J. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[7] M. C. Fletcher. mcf.pars. `http://www.golden.net/~mcfletch/programming/`.

[8] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *ACM LOPLAS.* 1(3):213–226, 1992.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[10] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide.* Ellis Horwood, 1990.

[11] J. R. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly, 1992.

[12] M. R. Levy. Web Programming in Guide. *Software, Practice and Experience.* Accepted for publication July 1998.

[13] M. E. Nordberg III. Variations on the Visitor Pattern. *PLoP '96 Writer's Workshop.*

[14] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[15] A. J. Patel. YAPPS. `http://theory.stanford.edu/~amitp/Yapps/`.

[16] O. Shivers. A Universal Scripting Framework. In *Concurrency and Parallelism, Programming, Networking, and Security*, pages 254–265. Springer, 1996.

[17] A. Watters. kwParsing. `http://starship.skyport.net/crew/aaron_watters/-kwParsing/`.

[18] D. B. Wortman. Compiling at 1000 MHz and beyond. In *Systems Implementation 2000*, pages 194–206. Chapman & Hall, 1998.