Teaching compilers with python

Matthieu Amiguet[*]

January 30, 2010

In the University of Applied Sciences ARC, compilers are taught in a relatively short amount of time. Focus is put on the main conceptual ideas, leaving aside many technical details. Still, the students are expected to write a full compiler within just a few weeks.

After trying the traditional C/Lex/Yacc based approach, and a more education-oriented Java/Jaccie solution, we settled on Python and PLY plus a few enhancements (syntax tree graphical representation, decorator to achieve better code separation). As a result, the students get a better understanding of the compiler concepts and produce more interesting and creative projects.

Contents

1 Introduction **2**

2 Short reminder on compilers **3**
- 2.1 General architecture 3
- 2.2 Front end 3
- 2.3 Back end 4

3 Previous experience **4**
- 3.1 First try: C/Lex/Yacc 5
- 3.2 Second try: Java/Jaccie 5

4 Requirements for a better solution **6**

5 The solution: Python+PLY+some customization **6**
- 5.1 PLY 101 6
  - 5.1.1 Scanner 7
  - 5.1.2 Parser 7

[*]Institut des systèmes d'information et de communications (ISIC), Haute École Arc, Switzerland – `http://www.he-arc.ch/hearc/fr/isic/`

5.2 Adding graphical representations 9
5.2.1 AST representations 9
5.2.2 Threading 11
5.2.3 Discussion 12
5.3 Getting good code separation 13
5.3.1 The problem 13
5.3.2 A decorator-based solution 14
5.3.3 Discussion 15

6 Results, from an educational point of view **15**

7 Conclusion **16**

1 Introduction

In the IT curriculum of the University of Applied Sciences ARC, compilers are taught as a part of a wider teaching unit “Advanced Programming”. As a result, the amount of time spent on this subject is relatively short: about 8 weeks, with 6×45 minutes sessions per week – students are supposed to put in about as much work at home. About half of the time in class is spent on theory, the other half being practical (tutorials and a project). By the end of that part of the course, students are expected to have realized a complete project using compiler techniques.

To achieve this rather ambitious goal, the teaching focuses on general compiler concepts, leaving aside many fancy details about numerous parsing techniques, optimization and such. Also, a strong focus is put on so-called “front-end” techniques (scanning, parsing, semantic analysis, . . .) rather than “back-end” ones (interpretation, code generation, optimization, . . .).

After successively trying solutions based on C/Lex/Yacc and Java/Jaccie, we finally settled on a Python/PLY combination with a few customizations. Our experience is that with this last solution, students get a better understanding of compiler concepts and produce more interesting and creative projects.

Note however that this last statement should be taken with some care. Similar claims have been made before at PyCon [PE09], based on a relatively large number of students. Our school being a very small school, we cannot provide any significant numerical data; the small number of students we have only allows us to make some qualitative claims about the gain we observe.

This paper is organized as follows: section 2 is a very short reminder of the main compiler concepts. Section 3 summaries the other solutions we tried before settling on the one we present here; section 4 presents the requirements we had for a better solution. The core of the paper is section 5, where we present the solution we are currently using. Section 6 discusses the pedagogical results of this approach before some concluding remarks in section 7.

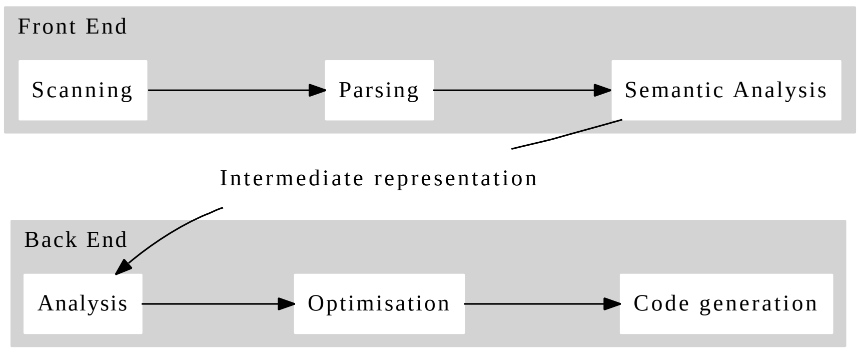


Figure 1: Simplified structure of a compiler

2 Short reminder on compilers

This section briefly presents the main compiler concepts used in this paper. For more details, refer to one of the many available compiler textbooks [ASU06, GBJL00, Med08] or the more language-oriented references [Sco05, Ste07].

2.1 General architecture

Compilers are generally structured in two parts:

The front end analyzes the *source code* to produce a complete, verified internal representation of the program. This so-called *intermediate representation* typically takes the form of a tree (usually called an *Abstract Syntax Tree* – *AST* for short) or some kind of idealized machine language.

The back end analyzes this intermediate representation, performs a number of optimizations and finally generates code into the target language. In the case of an interpreter, this very last step is replaced by direct execution.

Figure 1 presents a somewhat simplified structure of a general compiler. Note that the same front end techniques can be used for a compiler, an interpreter, a document generator, etc. However, the backend techniques will vary much more depending on the application.

2.2 Front end

The front end is traditionally composed of (at least) three parts:

The scanner (or lexer, or lexical analyzer) breaks the source code text into small, atomic pieces called *tokens*. Tokens are roughly the equivalent of *words* in natural languages (or *lexemes* in linguistics).

The parser (or syntax analyzer) identifies the (syntactic) structure of the sequence of lexemes. This phase typically builds a tree, called an *Abstract Syntax Tree*, or *AST*. In natural languages, this would correspond to identifying the grammatical structure of a text.

The semantic analyzer is more concerned with the "meaning" of the program. This may involve, e.g.

- Type checking: in some languages, a statement like `int i = 0.1` might be syntactically correct, but meaningless (semantically incorrect).
- Object binding: to execute a statement like `foo.bar()`, the compiler must link the name `foo` to an object in memory and possibly determine which of the many `bar()` methods defined in the program this invocation should call.
- Checking that variables are always assigned before they are used.
- ...

2.3 Back end

The backend takes a complete, verified representation of a program as input. It can then:

- execute the program's instructions immediately. Such a "compiler" is rather called an *interpreter*.
- generate another – more "executable" – representation of the program. This last representation can be machine language, assembler, "bytecode" for a virtual machine, etc.

3 Previous experience

A few years ago, various institutional changes lead us to rethink our compiler course from scratch. A few choices we've made at that time are:

- Strong focus on practice. Students must be able to produce a working compiler-like program, even if they cannot tell if a grammar is LL(1) or not off the top of their head.
- Focus on front-end techniques. The course will be much more about scanners and parsers than, say, about assembler, esoteric addressing modes and keeping the CPU's pipeline full.
- Use code generators. It's important to understand what they are doing under the hood, but in many real situations, you don't write a compiler's front end from scratch, you generate it from a higher-level description using a so-called compiler-compiler.

As told in the introduction, the course is about 50% practice and 50% theory. However, this paper concentrates only on the choices made and tools developed for the practical part. Note that the interested reader can find both the practical and theoretical supporting material on the author's website [Ami10].

3.1 First try: C/Lex/Yacc

The first solution we experienced was using the traditional C/Lex/Yacc tool combination. This seemed a good idea, because these are robust, proven, real-world tools, thus providing a realistic situation for the students to learn.

However, in our case, we quickly found that this solution had major shortcomings:

- Although Lex and Yacc allow to code the lexer and the scanner separately, these two steps are quite difficult to run separately. This makes it both more difficult to understand clearly what each step does and more difficult to debug.
- C is a rather low-level language. This means students spent a lot of time chasing memory leaks, trying to implement associative arrays (or even sometimes linked lists. . .). More time spent on generic programming problems means less time spent on understanding compilers concepts – and less interesting projects.

3.2 Second try: Java/Jaccie

After this first experience, we looked for tools in a higher-level language – and providing better debug facilities. Java looked like a good choice, as our students had some experience in that language. Among the Java-based solutions was the educational tool Jaccie, and we decided to give it a try.

Jaccie [Sch04] is an interactive environment aimed at illustrating the main compiler phases by implementing them in Java. Jaccie has many good points:

- Code organized into well-defined phases: scanner, parser, evaluator.
- It's easy to see what data goes in and out of these phases.
- Some useful tools are provided, like automatic AST building from a grammar specification, with an ASCII representation (this last feature proved much more useful to the students than we first thought).
- It's Java, so you get memory management, HashTables, . . .

We used Jaccie for two years, but we were more and more aware of some drawbacks of this solution:

- The swing-based GUI is slow to use (too much time spent on figuring what to drag-and-drop where in what order to make it work); the user is stuck with a sub-optimal built-in code editor.

- The compiler structure is fixed: scanner, parser, evaluator. The semantic analysis must be performed using attributed grammars, which was not the technique we wanted to focus on.
- Use from the IDE is tedious, but it is no easier to produce a stand-alone application. This did not encourage students to make many tests with the compilers they developed. Also, students had obviously much more the feeling that they were doing an exercise, detached from reality, than with the previous solution.
- The whole IDE is buggy. Not too much so, but enough to be rather annoying.
- Development seems to have stopped in 2004.

4 Requirements for a better solution

So we decided to try yet another solution. The first two attempts allowed us to be a little more precise in our requirements, in addition to what has been mentioned in the beginning of section 3:

- Good code separation between main parts (scanner, parser, ...)
- Possibility to (easily) see what data is flowing between these parts
- Possibility to generate textual and/or graphical representations of AST's
- High-level programming language
- Ability to (easily) produce standalone applications
- Mature, maintained, cross-platform

5 The solution: Python+PLY+some customization

The requirements above led us to consider PLY [Bea09], a python-based reimplementation of Lex and Yacc. The remainder of this paper discusses the solution we developed based on this tool and the results obtained.

5.1 PLY 101

PLY (Python Lex-Yacc) is a pure-python implementation of Lex and Yacc. Unlike the original tools however, it doesn't require a separate explicit code-generation step: a heavy (and rather clever) use of reflection allows on-the fly code generation. This allows for very quick write-and-test cycles.

5.1.1 Scanner

Here is an example of a complete scanner implementation in PLY for lexing parenthesized arithmetic expressions with floating points numbers (e.g. `12.3*(3-4)`):

```
 1  import ply.lex as lex

 3  tokens = (
        'NUMBER',
 5      'ADD_OP',
        'MUL_OP'
 7  )

 9  literals = '()'

11  t_ADD_OP = r'[+-]'
    t_MUL_OP = r'[*/]'
13
    def t_NUMBER(t):
15      r'\d+(\.\d+)?'
        t.value = float(t.value)
17      return t

19  def t_newline(t):
        r'\n+'
21      t.lexer.lineno += len(t.value)

23  t_ignore = ' \t'

25  def t_error(t):
        print "Illegal character '%s'" % t.value[0]
27      t.lexer.skip(1)

29  lex.lex()

31  if __name__ == '__main__':
        lex.runmain()
```

As expected, tokens are specified using regular expressions; naming conventions allow to use plain variable affectations to define simple tokens (see lines 11-12). A somewhat unusual use of docstrings makes it easy to manipulate token values (or even types) when they are recognized (lines 14-17). Several shortcuts are available, e.g. to specify one-character tokens in a compact manner (line 9) or ignore certain characters (line 23).

5.1.2 Parser

This is an example of a complete parser implementation, based on the scanner above, that evaluates an arithmetic expression, complete with parentheses, operator precedence and unary plus and minus:

```
   import ply.yacc as yacc
 2
   from scanner import tokens
 4
   operations = {
 6     '+' : lambda x,y: x+y,
       '-' : lambda x,y: x-y,
 8     '*' : lambda x,y: x*y,
       '/' : lambda x,y: x/y,
10 }

12 def p_expression_op(p):
       '''expression : expression ADD_OP expression
14               | expression MUL_OP expression'''
       p[0] = operations[p[2]](p[1],p[3])
16
   def p_expression_num(p):
18     'expression : NUMBER'
       p[0] = p[1]
20
   def p_expression_paren(p):
22     '''expression : '(' expression ')' '''
       p[0] = p[2]
24
   def p_minus(p):
26     ''' expression : ADD_OP expression %prec UMINUS'''
       p[0] = operations[p[1]](0,p[2])
28
   def p_error(p):
30     print "Syntax error in line %d" % p.lineno
       yacc.errok()
32
   precedence = (
34     ('left', 'ADD_OP'),
       ('left', 'MUL_OP'),
36     ('right', 'UMINUS'),
   )
38
   yacc.yacc()
40
   if __name__ == "__main__":
42     expr = raw_input('>')
       result = yacc.parse(expr)
44     print result
```

Again, docstring are used, somewhat unusually, to bind grammar rules to functions (see e.g. lines 12-15). The effects of reductions can be described very conveniently in the function body, using the special `p` list-like parameter: `p[0]` corresponds to the value of the first symbol in the rule, `p[1]` to the next one, etc.

Simple shift-reduce conflicts can be solved with precedence rules (lines 33-37). To deal with more complex cases, a very detailed parsing log can be generated (which should be read "with an appropriately high level of caffeination", as the doc states!).

Of course these examples only scratch the surface of what PLY can do. However, they should be enough to get an idea of the power of this module.

5.2 Adding graphical representations

5.2.1 AST representations

PLY provides almost everything stated in section 4, with the notable exception of the generation of graphical AST representation. This is no surprise, as PLY is completely agnostic about what to do when parsing: it doesn't even provide any AST construction tool.

That's why we decided to provide our students with a set of classes that allows them to easily construct an AST and generate graphical representations of it.

The core of the implementation is the `Node` class. A `Node` is simply a container for a list of children with a unique ID (the `next` attribute will be useful for *threading* in section 5.2.2):

```
class Node:
    count = 0
    type = 'Node (unspecified)'
    shape = 'ellipse'
    def __init__(self,children=None):
        self.ID = str(Node.count)
        Node.count+=1
        if not children: self.children = []
        elif hasattr(children,'__len__'):
            self.children = children
        else:
            self.children = [children]
        self.next = []
```

Defining new nodes types is simply a matter of redefining the type attribute, and possibly overloading the construction and representation methods:

```
class ProgramNode(Node):
    type = 'Program'

class TokenNode(Node):
    type = 'token'
    def __init__(self, tok):
        Node.__init__(self)
        self.tok = tok

    def __repr__(self):
        return repr(self.tok)
```

```
toto = 12*-3+4;
a = toto+1; a*2
```

```
Program
| =
| | 'toto'
| | + (2)
| | | * (2)
| | | | 12.0
| | | | - (1)
| | | | | 3.0
| | | 4.0
| =
| | 'a'
| | + (2)
| | | 'toto'
| | | 1.0
| * (2)
| | 'a'
| | 2.0
```

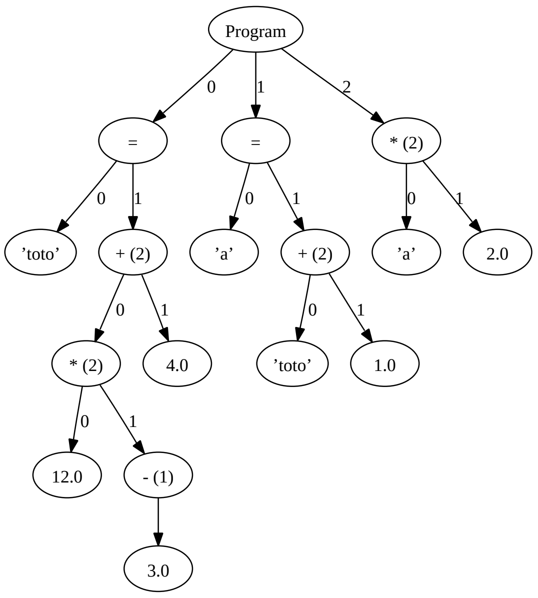


Figure 2: A source code and its AST, in ASCII and graphical representation

It is then straightforward to generate ASCII representations of the trees:

```
class Node:
    # [...]
    def asciitree(self, prefix=''):
        result = "%s%s\n" % (prefix, repr(self))
        prefix += '|   '
        for c in self.children:
            if not isinstance(c,Node):
                result += "%s*** Error: Child of type %r: %r\n" \
                                            % (prefix,type(c),c)
                continue
            result += c.asciitree(prefix)
        return result
```

The result is usable, as shown in figure 2. However, when the trees get very large, this representation is barely readable.

To generate better representations, we make use of the excellent software Graphviz, that we access via its python binding pydot. With these tools, it's surprisingly easy to generate a full graphical representation of a tree using a recursive method:

```
class Node:
    # [...]
    def makegraphicaltree(self, dot=None, edgeLabels=True):
        if not dot: dot = pydot.Dot()
        dot.add_node(pydot.Node(self.ID,label=repr(self), \
                                   shape=self.shape))
        label = edgeLabels and len(self.children)-1
        for i, c in enumerate(self.children):
            c.makegraphicaltree(dot, edgeLabels)
            edge = pydot.Edge(self.ID,c.ID)
            if label:
                edge.set_label(str(i))
            dot.add_edge(edge)
        return dot
```

Now, by replacing direct evaluation like in

```
def p_expression_op(p):
        '''expression : expression ADD_OP expression
                      | expression MUL_OP expression'''
        p[0] = operations[p[2]](p[1],p[3])
```

by the construction of a tree

```
def p_expression_op(p):
        '''expression : expression ADD_OP expression
                      | expression MUL_OP expression'''
        p[0] = AST.OpNode(p[2], [p[1], p[3]])
```

the result of the `yacc.parse` method is a `Node` object representing the root of the AST. Simple code such as

```
result = yacc.parse(source_code)
graph = result.makegraphicaltree()
graph.write_pdf(AST_filename)
```

produces a nice graphical output as in figure 2.

5.2.2 Threading

One of the fundamental - and relatively tricky - techniques of semantic analysis is called *threading*.

Threading an AST amounts to superimposing a second structure on top of the tree structure. This second structure represents the *control flow* of the program. Roughly speaking, this is the order in which the nodes of the tree will be executed.

As it seems to be difficult for some students to work with two different structures on the same set of nodes, we extended our class `Node` to include the possibility of threading the tree, and generating a representation of the two structures simultaneously.

This is the code for graphically threading an existing AST:

```
class Node:
    # [...]
    def threadTree(self, graph, seen = None, col=0):
        colors = ('red', 'green', 'blue', 'yellow', 'magenta', 'cyan')
        if not seen: seen = []
        if self in seen: return
        seen.append(self)
        new = not graph.get_node(self.ID)
        if new:
            graphnode = pydot.Node(self.ID,label=repr(self), \
                                    shape=self.shape)
            graphnode.set_style('dotted')
            graph.add_node(graphnode)
        label = len(self.next)-1
        for i,c in enumerate(self.next):
            if not c: return
            col = (col + 1) % len(colors)
            color = colors[col]
            c.threadTree(graph, seen, col)
            edge = pydot.Edge(self.ID,c.ID)
            edge.set_color(color)
            edge.set_arrowsize('.5')
            # Edges corresponding to the threading of the AST
            # are ignored in the graph layout. The AST will have a
            # "standard" appearance, but the threading edges might
            # take weird routes. If we comment this line out,
            # the general layout will be much better, but the AST
            # will be much less recognisable...
            edge.set_constraint('false')
            if label:
                edge.set_taillabel(str(i))
                edge.set_labelfontcolor(color)
            graph.add_edge(edge)
```

Figure 3 shows the kind of results obtained with this code.

5.2.3 Discussion

The AST graphical representation works very well. It allows inspecting relatively complex trees without problems. The variety of output formats provided by Graphviz is often appreciated by the students. For exploring very large trees, a more interactive solution could be needed, but in the context of our course, we can often limit ourselves to small to medium trees.

There are some problems with the threaded version though. In order to keep the tree recognizable, we freeze its layout before we add the threading edges. This sometimes results in weird layouts for these edges and for the nodes added at this stage. We added colors to make it easier to read, but there is definitely some place for improvement here. However, we found that even with a sub-optimal layout, these graphs help students to visualize what they are doing.

```
a=0;
while (a-10) {
    print a;
    a = a+1
}
```

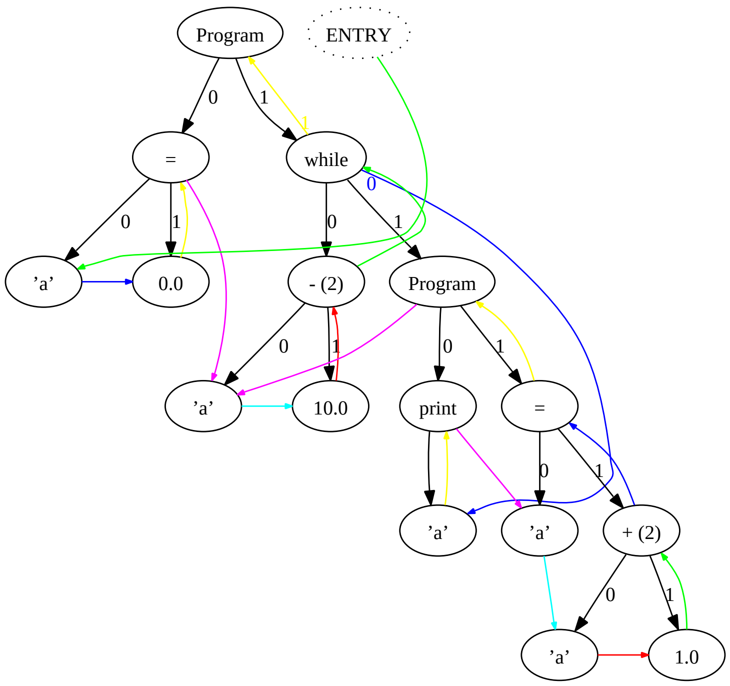


Figure 3: A source code and the corresponding threaded AST

5.3 Getting good code separation

5.3.1 The problem

The approach based on the `Node` class above works very well for generating graphics, but it breaks the code separation that was a requirement in section 4.

Of course, the scanner and the parser can still reside each in their own file, but everything that is made after parsing will very likely end up being implemented as a method of a subclass of the `Node` class. With this approach, the semantic analyzer and the whole back end of the compiler will be scattered over the `Node` subclasses hierarchy.

This will not only make it more difficult for the students to make a clear conceptual distinction between those treatments, but also this approach discourages them from experimenting with various implementations of the same treatment.

For instance, suppose we want to implement a recursive and an iterative version of an interpreter. We would first implement a series of `execute()` methods for the different node types. For the second version, we would have either to use a different method name (possibly necessitating modification in some client code), subclass every node type, or maintain a separate version of the code for the alternate implementation, with much of the base code duplicated.

Class	AST	Semantic analyzer	Interpreter	Compiler
BlockNode	`__init__()`, `__draw__()`, ...	`thread()`	`execute()`	`compile()`
StatementNode	`__init__()`, `__draw__()`, ...	`thread()`	`execute()`	`compile()`
...	...	...	...	...

Figure 4: Two-dimensional separation of concern in a compiler

5.3.2 A decorator-based solution

The problem stated above is a particular example of a general problem about the so-called Separation of Concerns: The decomposition of a problem into concerns usually results in concerns crosscutting one another. Choosing any main decomposition along some concerns (e.g. a class hierarchy) usually scatters other concerns over this decomposition. In our case, we clearly have a two-dimensional decomposition with classes on one axis and the compiler's various phases on the other, as suggested in figure 4.

The solution we chose for our compilers course is very simple, yet quite powerful. We write a simple decorator as follows:

```
def addToClass(cls):
    def decorator(func):
        setattr(cls,func.__name__,func)
        return func
    return decorator
```

This decorator allows us to easily add a method to a class from outside the class definition, even from a another module if desired.

With this decorator, a recursive interpreter implementation could look like this:

```
import AST
from AST import addToClass

@addToClass(AST.ProgramNode)
def execute(self):
    for c in self.children:
        c.execute()

@addToClass(AST.OpNode)
def execute(self):
    args = [c.execute() for c in self.children]
    if len(args) == 1:
        args.insert(0,0)
    return reduce(operations[self.op], args)

@addToClass(AST.WhileNode)
def execute(self):
    while self.children[0].execute():
        self.children[1].execute()
```

```
# [...]

if __name__ == "__main__":
    from parser import parse
    import sys
    prog = file(sys.argv[1]).read()
    ast = parse(prog)

    ast.execute()
```

Now if we want to try and implement an iterative interpreter, we would only need to make a new file with alternate definitions of the `execute` method. No name changing, no subclassing everywhere, no code duplication; it's simply a matter of importing the right file in the client code.

5.3.3 Discussion

The approach presented above, which could be seen as a very basic form of Aspect Oriented Programming, works very well in many practical situations.

A drawback is that a method decorated with the `addToClass` decorator still exists in the namespace where it was defined. This is definitely not very elegant, and may cause some unexpected behaviour when the method name shadows a name in the current namespace. It may also confuse some code checking tools like pychecker.

We are not aware of any way to avoid this side effect. However, if we avoid using conflicting names for methods, this is a minor drawback, and we find this decorated-based solution to be very useful.

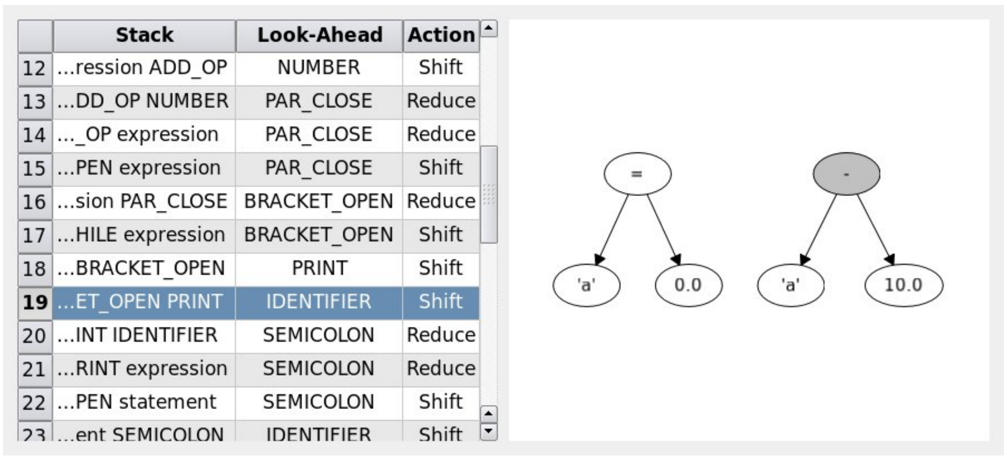
6 Results, from an educational point of view

The solution presented in section 5 is both easier than the C/Lex/Yacc we first tried and more stable and mature than the Java/Jaccie one.

This means that the students get more time to understand the concepts and develop interesting projects. Also, concepts as AST and threading benefit very much from the possibility of getting graphical representations. At the end-of-year exam, we found the students to be much more at ease with these concepts after we introduced this new approach.

We tend to allow a lot freedom regarding the kind of project developed by the students in this course, as long as it's making a significant use of the compiler techniques we studied. A completely unexpected side-effect of using python in this course is that the richness of the libraries, combined with the high productivity of the language, allow for a very wide range of projects, some of which are quite mature when considering the short time available. Among others, we've seen

- compilers producing code for existing virtual machines (parrot), for custom-made virtual machines, or for a micro-controller;



	Stack	Look-Ahead	Action
12	...ression ADD_OP	NUMBER	Shift
13	...DD_OP NUMBER	PAR_CLOSE	Reduce
14	..._OP expression	PAR_CLOSE	Reduce
15	...PEN expression	PAR_CLOSE	Shift
16	...sion PAR_CLOSE	BRACKET_OPEN	Reduce
17	...HILE expression	BRACKET_OPEN	Shift
18	...BRACKET_OPEN	PRINT	Shift
19	...ET_OPEN PRINT	IDENTIFIER	Shift
20	...INT IDENTIFIER	SEMICOLON	Reduce
21	...RINT expression	SEMICOLON	Reduce
22	...PEN statement	SEMICOLON	Shift
23	...ent SEMICOLON	IDENTIFIER	Shift

Figure 5: A prototype of a parser log visualizer

- some compilers targeting high-level languages, but adding features like simple parallelism;
- a variety of programs producing SVG pictures of PDF documents from document description languages;
- an interpreter for a language specialized in producing interactive stories for children (with a PyGame backend);
- an interpreter allowing experimentation with collective behaviors by controlling cars and trucks in a grid (PyGame backend) in the spirit of NetLogo;
- ...

A few examples are available for download on the author's website [Ami10].

7 Conclusion

Three years after launching the Python/PLY approach in our compilers course, we are still very pleased with the results. Students don't spend too much time trying to understand how to use the tools we give them, and have more time to understand the concepts and produce nice and creative results.

This was also a great opportunity to introduce the python language in the curriculum, which allows the students to discover an alternative to other major object-oriented high-level languages they have studied earlier.

The tools we have put together meet the requirements we have for this course. Our plans for the future are definitely based on the existing solution, with the following enhancements:

- Migration to Python 3.x

- Find a solution to the namespace pollution problem of the `addToClass` decorator discussed in section 5.3.3.
- The current implementation of the `Node` hierarchy often relies on child order instead of explicit semantics (e.g. an 'If' Node must have three children that correspond to the condition, the 'then' part and the 'else' part respectively). A more explicit approach would be preferable.

Also, in the present state, the visualization tools only allow to see the *result* of the parsing and threading algorithms. It could be very useful to have a tool allowing to investigate the progress of those algorithms. A master student, David Jacot, has developed a prototype of a software providing a visualization of a parser log file. A screenshot is shown in figure 5. This is still in an early stage of development, but a complete version could probably save a considerable amount of caffeine when debugging a grammar!

References

[Ami10] Matthieu Amiguet. Matthieu Amiguet's website [french]. `http://www.matthieuamiguet.ch`, 2010.

[ASU06] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2006.

[Bea09] David M. Beazley. PLY (Python Lex-Yacc). `http://www.dabeaz.com/ply`, 2009.

[GBJL00] D. Grune, H. Bal, C. Jacobs, and K. Langendoen. *Modern Compiler Design.* Wiley, 2000.

[Med08] Alexander Meduna. *Elements of Compiler Design.* Auerbach Publications, 2008.

[PE09] Bill Punch and Rich Enbody. Python for CS1 Not Harmful to CS Majors (and good for everyone). `http://us.pycon.org/media/2009/talkdata/PyCon2009/008/pycon09-Punch_.pdf`, 2009.

[Sch04] Lothar Schmitz. Visual syntax tools. `http://www2.cs.unibw.de/Tools/Syntax/english/index.html`, 2004.

[Sco05] Michael L. Scott. *Programming Language Pragmatics.* Morgan Kaufmann, 2005.

[Ste07] D. E. Stevenson. *Programming Language Fundamentals by Example.* Auerbach Publications, 2007.