Teaching Compilers with Python

Matthieu Amiguet

PyCon 2010 Atlanta
Teaching Compilers With Python?

Not a very common choice...

WHY?

HOW?

RESULTS?
IT students, last year of BSc
Relatively short period of time (8 weeks)
However, students are expected to realize a complete, working project using compiler techniques
General Architecture of a Compiler
General Architecture of a Compiler

- Front End: Scanning → Parsing → Semantic Analysis
  - Intermediate representation

- Back End: Analysis → Optimisation → Code generation

- Flow of characters
General Architecture of a Compiler

- flow of characters
- flow of *tokens*
General Architecture of a Compiler

- flow of characters
- flow of *tokens*
- *Abstract Syntax Tree (AST)*
General Architecture of a Compiler

- flow of characters
- flow of tokens
- Abstract Syntax Tree (AST)
- Decorated AST
Choices for the course

- Focus on practice
- Focus on front-end techniques
- Use code generators
Previous experience

- C/Lex/Yacc
  - The real thing, but...  
  - Too difficult

- Java/Jaccie
  - Many interesting ideas, but...  
  - Clumsy, buggy, unmaintained
Requirements For a Better Solution

- High-level programming language
- Good code separation between scanner, parser, ...
- Possibility to generate text and/or graphical representations of AST’s
- Mature, maintained, cross-platform
Teaching Compilers with Python

1. Python/PLY (+customization)
2. Results
3. Conclusion
Teaching Compilers with Python

1. Python/PLY (+customization)
   - PLY 101 by Example
     - Adding Graphical AST Representations
     - Getting good code separation

2. Results

3. Conclusion
What is PLY?

- PLY is a python re-implementation of Lex and Yacc
- Written by David Beazley
- Based on introspection \(\leadsto\) very “economic”

Let’s try to evaluate arithmetic expressions like

\[(1 + 2) \times 3 - 4\]
Using ply.lex

t__ADD__OP  =  r ' [+−] ' 

1

0

t__MUL__OP  =  r ' [∗/] ' 

2

1

3

2

1

0
Using ply.lex

```
ten_ADD_OP = r'\[+-\]'
t_MUL_OP = r'\[*/\]'

def t_NUMBER(t):
    r'\d+(\.\d+)?'
    t.value = float(t.value)
    return t
```
Grammar for the parser

expression → NUMBER
| expression ADD_OP expression
| expression MUL_OP expression
| '(' expression ')'
| ADD_OP expression
Using ply.yacc

```python
def p_expression_num(p):
    'expression : NUMBER'
    p[0] = p[1]
```
Using ply.yacc

def p_expression_num(p):
    'expression : NUMBER'
    p[0] = p[1]

def p_expression_op(p):
    'expression : expression ADD_OP expression |
                 expression MUL_OP expression'
Teaching Compilers with Python

1. Python/PLY (+customization)
   - PLY 101 by Example
   - Adding Graphical AST Representations
   - Getting good code separation

2. Results

3. Conclusion
PLY provides almost everything we need…

… except AST representation

- PLY is agnostic about what to do when parsing

We provide our students with a set of classes allowing to

- build an AST
- generate ASCII or graphical representations of it

Graphics generated by Graphviz via pydot
Using Pydot

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
Using the Node Class Hierarchy

```python
def p_expression_op(p):
    '''expression : expression ADD_OP expression
                  | expression MUL_OP expression'''
    p[0] = AST.OpNode(p[2], [p[1], p[3]])
```
toto = 12*3+4;
a = toto+1; a*2

```
Program
  =
  =
  * (2)
     =
     0
     1
     2
        =
        0
        1
        0
          =
          0
          1
          + (2)
            'toto'
              0
              1
              0
                * (2)
                  4.0
                  'toto'
                    0
                    1
                    0
                      12.0
                      - (1)
                        3.0
                        1
                          1
                          2
                            =
                            'a'
                              0
                              1
                              + (2)
                                'toto'
                                  1.0
                                    * (2)
                                      'a'
                                        0
                                        1
                                        2
                                          4.0
                                          1
                                          3.0
                                          1
                                          2.0
```

Program
  =
  =
  * (2)
     =
     0
     1
     2
        =
        0
        1
        0
          =
          0
          1
          + (2)
            'toto'
              0
              1
              0
                * (2)
                  4.0
                  'toto'
                    0
                    1
                    0
                      12.0
                      - (1)
                        3.0
                        1
                          1
                          2
                            =
                            'a'
                              0
                              1
                              + (2)
                                'toto'
                                  1.0
                                    * (2)
                                      'a'
                                        0
                                        1
                                        2.0
Representing *threaded* ASTs

```
a=0;
while (a-10) {
    print a;
    a = a+1
}
```
Teaching Compilers with Python

1. Python/PLY (+customization)
   - PLY 101 by Example
   - Adding Graphical AST Representations
   - Getting good code separation

2. Results

3. Conclusion
The Problem

- The approach based on the `Node` class hierarchy above works well for graphics...
- ... but it breaks the code separation we were looking for.

<table>
<thead>
<tr>
<th>Class</th>
<th>AST</th>
<th>Semantic analyzer</th>
<th>Interpreter</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockNode</td>
<td><code>__init__(), __draw__(), ...</code></td>
<td>thread()</td>
<td>execute()</td>
<td>compile()</td>
</tr>
<tr>
<td>StatementNode</td>
<td><code>__init__(), __draw__(), ...</code></td>
<td>thread()</td>
<td>execute()</td>
<td>compile()</td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
<td><code>...</code></td>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
</tbody>
</table>

Problem: we would like lines as classes and rows as modules...
The Answer: a (Very) Simple Decorator

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

```python
def execute(self):
    for c in self.children:
        c.execute()

@addToClass(AST.OpNode)
def execute(self):
    args = [c.execute() for c in self.children]
    # [...]

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

class Foo:
    pass

help(sys)

@addToClass(Foo)
def help(self):
    print "I'm Foo's help"

help(sys)
Teaching Compilers with Python

1. Python/PLY (+customization)
2. Results
3. Conclusion
Teaching Compilers with Python

1. Python/PLY (+customization)

2. Results
   - Comparison
   - Examples

3. Conclusion
Comparison

- The PLY-based solution is
  - Easier than C/Lex/Yacc
  - More stable and mature than Java/Jaccie

- Students get more time to
  - understand the concepts
  - develop interesting projects

- Graphical representations help to understand AST’s and threading

- Unexpected side effect: Python’s many libraries and high productivity allow for very interesting projects!
Teaching Compilers with Python

1. Python/PLY (+customization)

2. Results
   - Comparison
   - Examples

3. Conclusion
Mougin & Jacot, 2009

- Compiler
- Rather complex source language
  - Built-in types: int, float, string, array
  - Conditional, loops
  - Console & file input/output
  - Functions, recursion, imports, …
- The target is a kind of assembler language for a custom virtual machine (also written in Python)
- The compiler implements
  - Some error checking
  - Some AST and bytecode optimization
  - …
Example

```python
function main(args) {
    print(fact(500));
}

function fact(n) {
    if (n==1) ret = n;
    else ret = n*fact(n-1);
    return ret;
}
```

```assembly
GETPROGARGS
CALL main 1
main: PUSHI 500
CALL fact 1
WRITE
PUSHI 0
EXIT
fact: ALLOC 1
GETP 0
PUSHI 1
EQ
JZ ifsep0_0
GETP 0
SETL 0
JMP endif0
ifsep0_0:
    GETP 0
    GETP 0
    PUSHI 1
    SUB
    CALL fact 1
    MUL
    SETL 0
endif0:
    GETL 0
    RETURN 1
```
Roth & Voumard, 2008

- Interpreter for a simple multi-agent programming language
  - In the spirit of NetLogo
- With PyGame back-end
- Two types of objects (cars and trucks) move and interact in an environment
- Many built-ins functions to manipulate the objects
- Conditionals, loops, ...
while running {
    all [
        nb = current.pickNeighbours()
        nb = nb.count()
        if current.isCar() {
            min = 2
            max = 5
        } else {
            min = 0
            max = 0
        }
    ]
    if (nb < min || nb > max) {
        current.turn(rand(-1,1))
        fw = current.pickBackward()
        ...
    }
Teaching Compilers with Python

1. Python/PLY (+customization)
2. Results
3. Conclusion
Conclusion

- Three years after introducing the Python/PLY approach, we’re still very pleased with the results.
- Students spend less time learning to use the tools…
- … and more time understanding what they are doing!
- Also a great opportunity to introduce Python in the curriculum
  - Alternative to other major OO high-level languages
Migrate to Python 3
Find a solution to the namespace pollution problem of @addToClass
Develop tools to visualize the process of parsing and not only the result
   First prototype by David Jacot, 2010
Visualizing the Parsing Process

<table>
<thead>
<tr>
<th>Stack</th>
<th>Look-Ahead</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 ...ession ADD_OP</td>
<td>NUMBER</td>
<td>Shift</td>
</tr>
<tr>
<td>13 ...DD_OP NUMBER</td>
<td>PAR_CLOSE</td>
<td>Reduce</td>
</tr>
<tr>
<td>14 ..._OP expression</td>
<td>PAR_CLOSE</td>
<td>Reduce</td>
</tr>
<tr>
<td>15 ...PEN expression</td>
<td>PAR_CLOSE</td>
<td>Shift</td>
</tr>
<tr>
<td>16 ...sion PAR_CLOSE</td>
<td>BRACKET_OPEN</td>
<td>Reduce</td>
</tr>
<tr>
<td>17 ...HILE expression</td>
<td>BRACKET_OPEN</td>
<td>Shift</td>
</tr>
<tr>
<td>18 ...BRACKET_OPEN</td>
<td>PRINT</td>
<td>Shift</td>
</tr>
<tr>
<td>19 ...ET_OPEN PRINT</td>
<td>IDENTIFIER</td>
<td>Shift</td>
</tr>
<tr>
<td>20 ...INT IDENTIFIER</td>
<td>SEMICOLON</td>
<td>Reduce</td>
</tr>
<tr>
<td>21 ...RINT expression</td>
<td>SEMICOLON</td>
<td>Reduce</td>
</tr>
<tr>
<td>22 ...PEN statement</td>
<td>SEMICOLON</td>
<td>Shift</td>
</tr>
<tr>
<td>23 ...ent SEMICOLON</td>
<td>IDENTIFIER</td>
<td>Shift</td>
</tr>
</tbody>
</table>
Further information

- More details in the companion paper
- Code, student’s examples & tutorials (in french) on

http://www.matthieuamiguet.ch/
Questions?