Teaching Compilers with Python

Matthieu Amiguet

PyCon 2010 Atlanta

haute école arc ingénierie
neuchâtel berne jura
saint-imier le locle delémont

Teaching Compilers With Python?

Not a very common choice. . .

WHY?

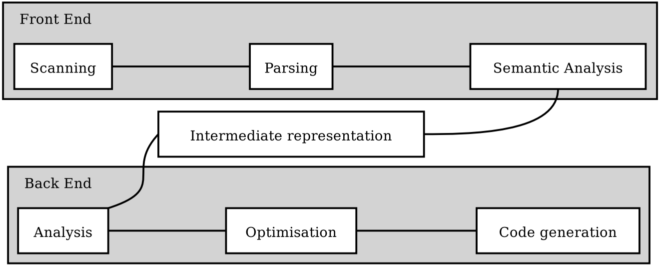
HOW?

RESULTS?

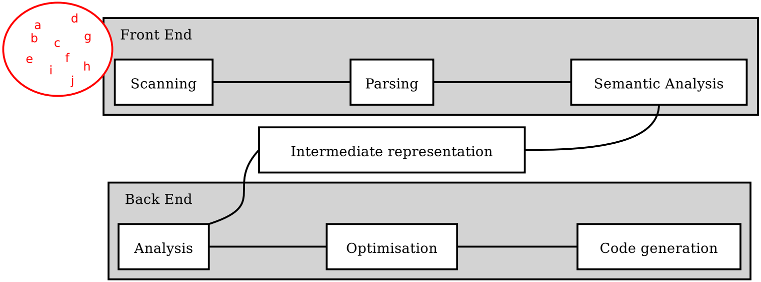
Teaching Compilers...

- 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

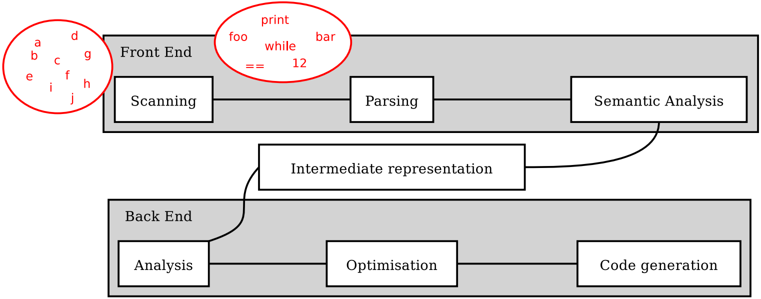


General Architecture of a Compiler



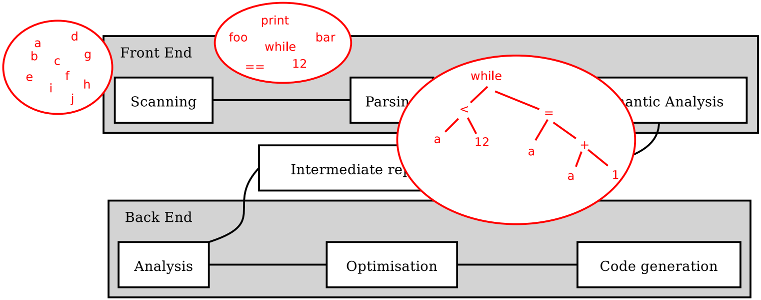
- flow of characters

General Architecture of a Compiler



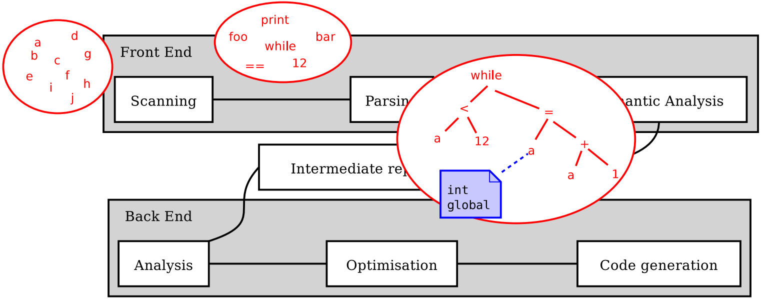
- flow of characters
- flow of *tokens*

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

What is PLY?

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

Let’s try to evaluate arithmetic expressions like

$$(1+2)*3-4$$

Using ply.lex

```
t_ADD_OP = r'[+-]'
t_MUL_OP = r'[*/]'
```

Using ply.lex

```
t_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

$$
\begin{array}{lcl}
\texttt{expression} & \longrightarrow & \texttt{NUMBER} \\
 & | & \texttt{expression ADD\_OP expression} \\
 & | & \texttt{expression MUL\_OP expression} \\
 & | & \texttt{'(' expression ')'} \\
 & | & \texttt{ADD\_OP expression}
\end{array}
$$

Using ply.yacc

```
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'''
    if p[2] == '+'  : p[0] = p[1] + p[3]
    elif p[2] == '-': p[0] = p[1] - p[3]
    elif p[2] == '*': p[0] = p[1] * p[3]
    elif p[2] == '/': p[0] = p[1] / p[3]
```

Teaching Compilers with Python

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

Graphical Representations

- 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

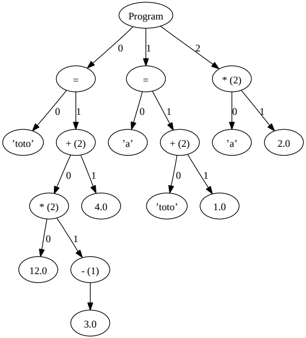
Using Pydot

```
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

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

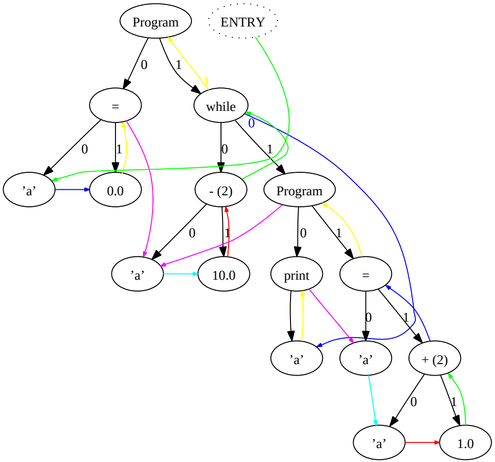
```
toto = 12*-3+4;
a = toto+1; a*2
```




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

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

- 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

```
@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]
    # [...]

@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

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

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

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

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

```
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
```

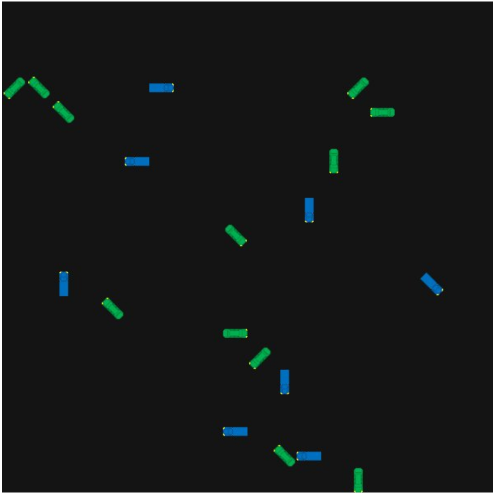
Roth & Voumard, 2008

- 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, ...

Example

```
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()
            ...
```

Running. . .



Teaching Compilers with Python

Conclusion

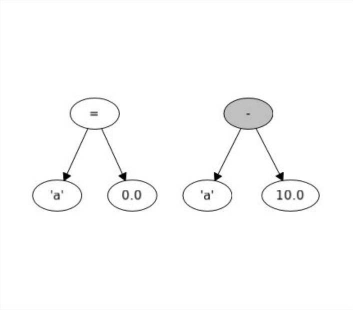
- Three years after introducing the Python/PLY approach, we're still very pleased with the results
- Students spend less time learning to use the tools...
- ... and more time understanding what they are doing!
- Also a great opportunity to introduce Python in the curriculum
    - Alternative to other major OO high-level languages

Perspectives

- 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

	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

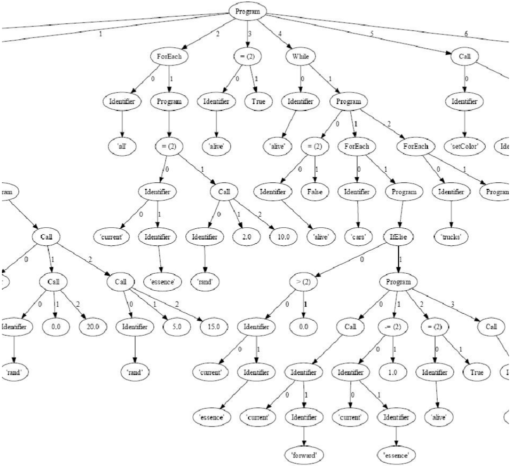


Further information

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

`http://www.matthieuamiguet.ch/`

Questions?


Program
ForEach
= (2)
While
Call
Identifier
Program
Identifier
True
Identifier
Program
Identifier
'all'
= (2)
'alive'
'alive'
= (2)
ForEach
ForEach
'setColor'
Identifier
Call
Identifier
False
Identifier
Program
Identifier
Program
Call
'current'
Identifier
Identifier
2.0
10.0
'alive'
'cars'
IfElse
'trucks'
Call
Call
'essence'
'rand'
> (2)
Program
Identifier
0.0
20.0
Identifier
5.0
15.0
Identifier
0.0
Call
-= (2)
= (2)
Call
'rand'
'rand'
'current'
Identifier
Identifier
Identifier
1.0
Identifier
True
'essence'
'current'
Identifier
'current'
Identifier
'alive'
'forward'
'essence'