Python Abstract Syntax

Code Spelunking in Python

Cleveland Python Users Group - March 2009

Who are you?

@mcrute

mike.crute.org

AGINTERACTIVE

Abstract Syntax Trees?

- High-level view of code without any actual code
  - Represents the concepts of the code
- Different from Concrete Syntax Trees
  - Represent the actual parsed grammar
- Side effect of compilation
- Part of the compiler package (for now)

AST represents code, it's not actually the code itself.
Prior to 2.5 python did not generate an intermediate
Concrete syntax trees, aka parse trees.

History Lesson

- Introduced in 2.2
  - Created by Greg Stein and Bill Tutt
  - Part of Python-to-C compiler project
- Pure python implementation until 2.6
- Major changes in 2.5, 2.6 and 3.0

Changes in 2.5

- Introduced _ast module
  - Provides only node classes
- Provided access to compiler AST
- Not very useful yet, building the future

Passing _PyCF_ONLY_AST as a flag to compile will give access to the compiler's This is different than the one generated by compiler.ast

Changes in 2.6

- Introduction of ast module
- Total reliance on the compiler's AST
- Introduced NodeTransformer
- Deprecated compiler package
- AST trees can now be compiled to bytecode

Changes in 3.0

- Removal of compiler package
- Last remnants of original AST now gone

Compiler Tourism

Take a quick tour through the Python compiler. This process has changed a couple of times in recent years.

The Dark Ages (pre-2.5)

1. Generate parse tree
2. Emit bytecode

Quick and dirty way to do bytecode generation. Wasn't really the easiest way but fast to implement. (See PEP 339)

```python
def test():
    return 2*2
```

```
(257,
 (266,
  (291,
   (261,
    (1, 'def'),
    (1, 'test'),
    (262, (7, '('), (8, ')')),
    (11, ':'),
    (299,
     (4, ''),
     (5, ''),
     (266,
      (267,
       (268,
        (274,
         (277,
          (1, 'return'),
          (326,
           (303,
            (304,
             (305,
              (306,
               (307,
                (309,
                 (310,
                  (311,
                   (312,
                    (313,
                     (314,
                      (315, (316, (317, (2, '2')))),
                      (16, '*'),
                      (315, (316, (317, (2, '2')
                          ))))))))))))))))))),
       (4, ''))),
     (6, ''))))),
 (4, ''),
 (0, ''))
```

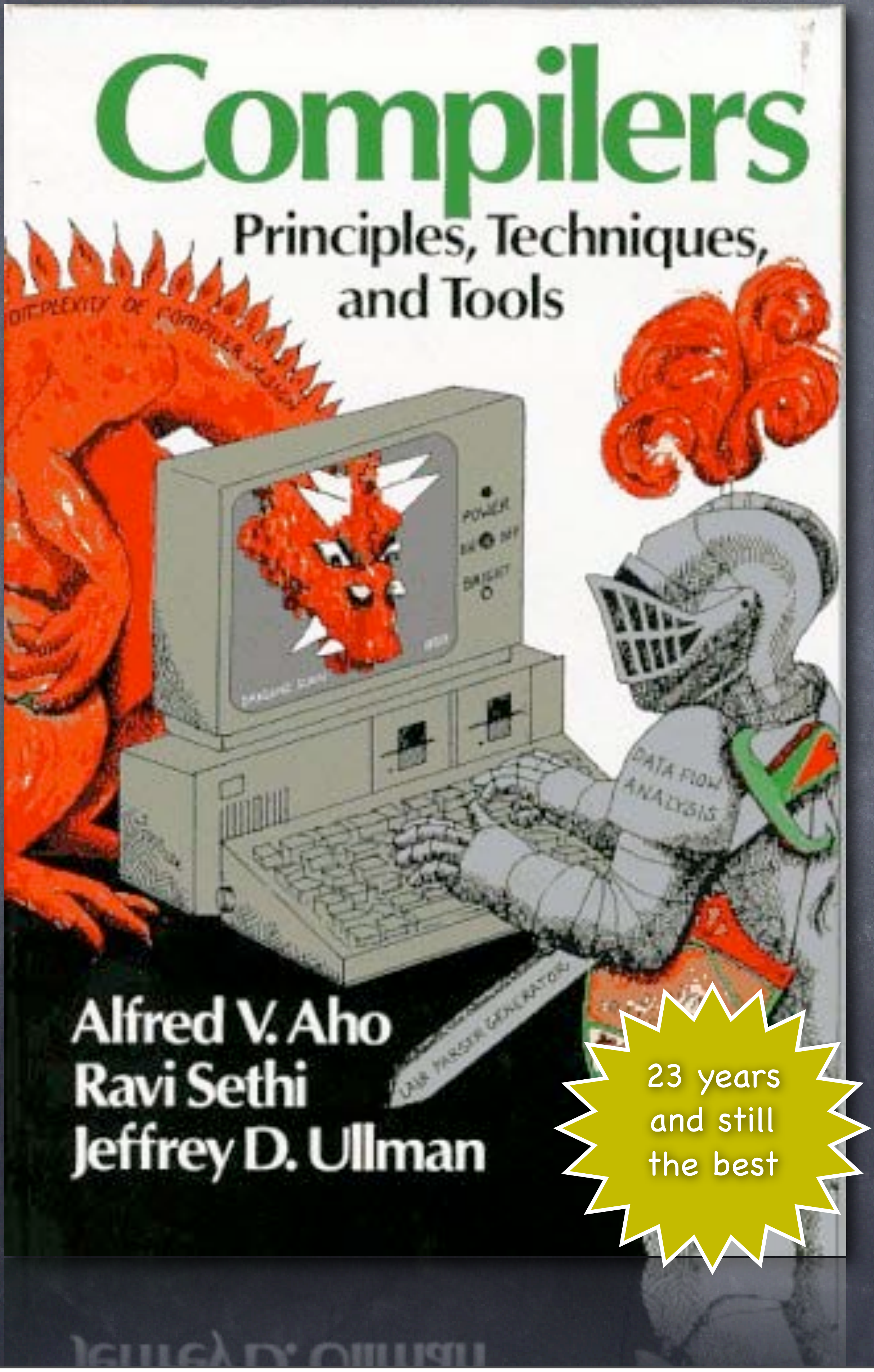
```
0 LOAD_CONST        2 (4)
3 RETURN_VALUE
```

Keep it simple, CST’s get crazy quickly. parser module gives you access to parse trees. dis module (disassemble) gives you access to bytecode.

Simple, right?

Not so much.

Not really all that easy to maintain, forced to go from parse tree to bytecode in one shot isn't easy, error prone.



23 years and still kicking! Still the authoritative reference for compiler builders. Newer python compilers implemented using the algorithms described in the book.

Anybody read it? You'll feel much more at home now.

The World Since 2.5

1. Generate parse tree
2. Parse tree => AST
3. AST => Control Flow Graph (CFG)
4. CFG => Bytecode

```
def test():
    return 2*2
```

```
(257,
 (266,
  (291,
   (261,
    (1, 'def'),
    (1, 'test'),
    (262, (7, '('), (8, ')')),
    (11, ':'),
    (299,
     (4, ''),
     (5, ''),
     (266,
      (267,
       (268,
        (274,
         (277,
          (1, 'return'),
          (326,
           (303,
            (304,
             (305,
              (306,
               (307,
                (309,
                 (310,
                  (311,
                   (312,
                    (313,
                     (314,
                      (315, (316, (317, (2, '2')))),
                      (16, '*'),
                      (315, (316, (317, (2, '2')
                          ))))))))))))))))))),
        (4, ''))),
      (6, ''))))),
 (4, ''),
 (0, ''))
```

```
Module(None,
    Stmt([
        Function(None, 'test', (), (), 0, None,
            Stmt([
                Return(
                    Mul((
                        Const(2),
                        Const(2))))
                    ]))
            ]))
```

```
0 LOAD_CONST        2 (4)
3 RETURN_VALUE
```

‘round and ‘round we go.
Same basic process as 2.3 except that AST is generated in the middle.
Purposely ignoring CFG because it’s not accessible to us.

That's better!

We can work with that.

Possible Uses

- Code spelunking
- Template engines
- Language conversion
- Static analysis
- Runtime code manipulation
- ... the possibilities are endless...

Code Spelunking

- Code is just text content
- Legacy code not safe for intropsection
- Determine relationships in the code
- Find unused code in a large codebase

Template Engines

- Still have to parse the code
- No more string concatenation
- Safer and easier to build

Language Conversion

- Originally for Python-C compiler
- Can transform Python to other languages
- XSLT of Code (sorry)

Static Analysis

- Enforce coding standards (PEP8)
- Detect potentially complex code
- Determine where re-factoring may be helpful

Runtime Code Manipulation

- Not a very nice thing to do
- May be useful in places where using string concatenation to build up functions
- Not a very nice thing to do
- On second thought... don't do this

Provided Tools

NodeVisitor

- Walks the node tree
- Allows for read-only processing of nodes
- Subclass of NodeVisitor
  - Implement your visit_Node funtions

The Visitor

```
class ImportVisitor(ASTVisitor, object):

    def visitImport(self, node):
        for name in node.names:
            self.put_symbol(name[0])

    def visitFrom(self, node):
        symbols = []
        for symbol, _ in node.names:
            symbols.append(symbol)
        self.put_symbol(node.modname, symbols)
```

Usage

```
my_file = open('/Users/cruteme/Desktop/test.py').read()

tree = compiler.parseFile('/Users/cruteme/Desktop/test.py')
# or preferably
tree = compile(my_file, '<string>', 'exec', ast.PyCF_ONLY_AST)

my_visitor = ImportVisitor()
my_visitor.virtual_symbols = get_virtual_symbols(tree)
visitor.walk(tree, my_visitor)
```

Results

```
set(['InvitationsCategoryResultsPage',
     'DownloadsSearchResultsPage',
     'DesktopDownloadsCategoryResultsPage',
     ...
    ])
```

NodeTransformer

- New in 2.6
- Specialized NodeVisitor
- Allows for transformation of nodes in-place
- Can remove nodes in a tree

Code?

```
class GenshiSemantics(NodeTransformer):

    def context(self, method):
        return Expr(Call(
            Attribute(Name('data', Load()), method, Load()),
            [], [], None, None
        ))

    def visit_For(self, node):
        self.generic_visit(node)
        return [self.context('push'), node, self.context('pop')]

    def visit_Name(self, node):
        self.generic_visit(node)
        return Subscript(
            Name('data', Load()),
            Index(Str(node.id)),
            node.ctx
        )
```

Usage

```
my_file = open('/Users/cruteme/Desktop/test.py').read()

tree = compile(my_file, '<string>', 'exec', ast.PyCF_ONLY_AST)

out_tree = GenshiSemantics().visit(tree)
```

Results

```
seq = [1, 2, 3, 4, 5]
item = 42
for item in seq:
    print 'inside', item
print 'outside', item
```

$\longrightarrow$

```
data['seq'] = [1, 2, 3, 4, 5]
data['item'] = 42
data.push()
for data['item'] in data['seq']:
    print 'inside', data['item']
data.pop()
print 'outside', data['item']
```

So, Who Uses It?

- Jinja (Template Engine)
- PyCC (Code Analysis Tool)
- Pylint (Code Analysis)
- CodeConnector (Vaporware)
- ... and more...

PyCC

Cyclomatic Complexity

Detecting Code Complexity

CodeConnector

Inter-Code-Base Relationships

Finding Unused Code

Questions?