How Cython Works

A one hour guide to compiler design (sarcasm)

Josh Kantor

May 15, 2008

Goal for this talk

- Cython is an integral part of sage, and it is useful to be able to bend it to our will.
- However, understanding out how Cython works is difficult at first glance (and second, third, etc).
- It is quite sophisticated and figuring out what pieces of Cython do what is tricky at first.
- Thus the goal of this talk is to present a roadmap to the Cython codebase.
- Also, cython is very, very well engineered, and is an excellent example of a very sophisticated python program. I would say cython is from a software perspective the most interesting part of sage (this is of course an opinion).

< Show cython source structure in terminal >

Cython is a Compiler

The first thing to keep in mind when trying to understand Cython is that it really is a compiler, and it is designed very much like a textbook compiler. It just happens to compile to the python-C API instead of executable machine code. It was very helpful for me to skim over the standard compiler textbook (the dragon textbook) to figure out why Cython was designed the way it is.

(Disclaimer: Any apperance that I know much about compilers is purely showmanship.)

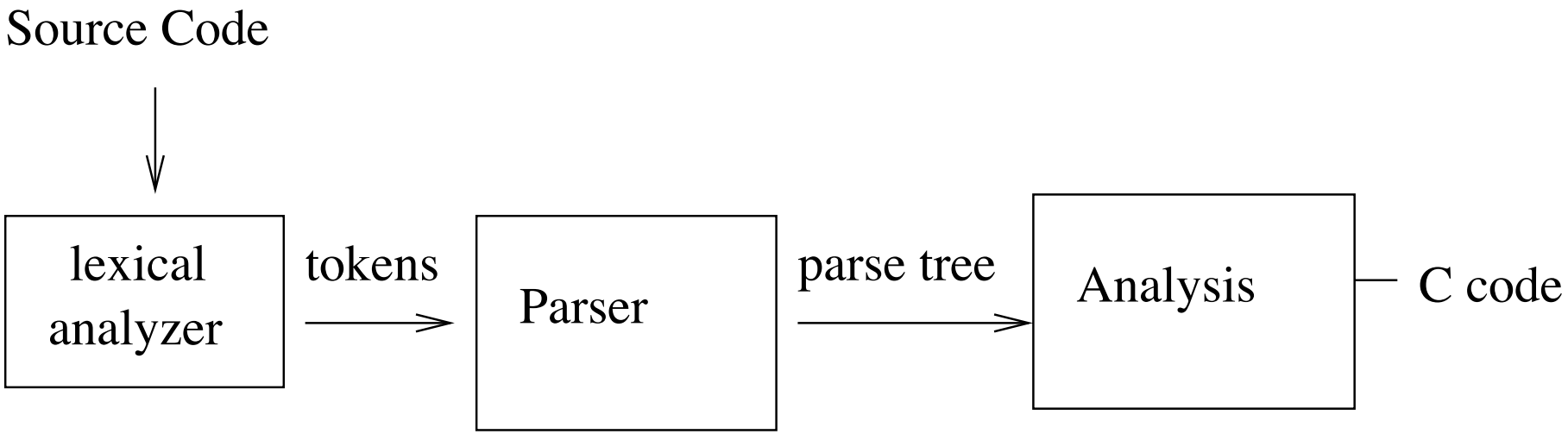
How do Compiler's Work

First how do we specify a programming language. Basically we specify the language through a grammar. A grammar tells us how to decompose complex expressions into simple ones. Just like english grammar tells us how to decompose sentences into nouns, verbs, phrases, etc. Lets illustrate through some pieces of the official python grammar

```
identifier ::= (letter|"_") (letter | digit | "_")*
assignment_stmt ::=(target_list "=")+
                   (expression_list | yield_expression)
target_list ::= target ("," target)* [","]
target ::=   identifier | "(" target_list ")" |
             "[" target_list "]"  | attributeref
             | subscription| slicing
```

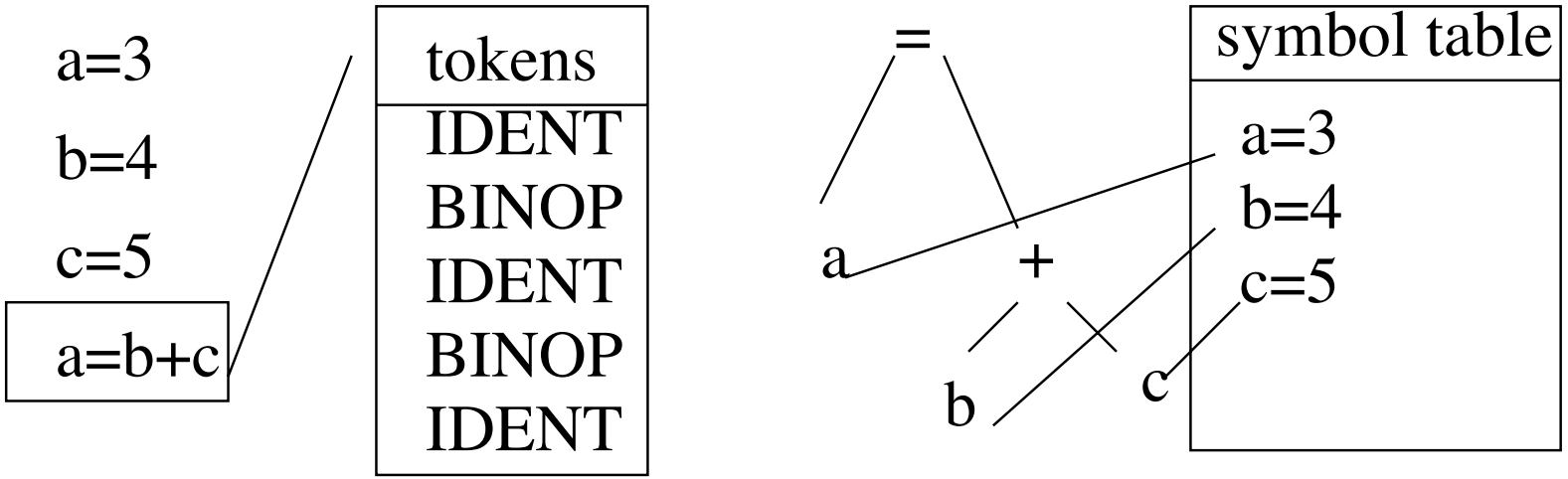
(Note + means one or more, * means 0 or more, | means or, [ ] means 0 or 1.)

Usually (and in Cython) there are two pieces cooperating to produce the parse tree, a Lexical analyzer and a Parser. The lexical analyzer produces a stream of tokens that the parser uses (together with the structure of the grammar) to produce a parse tree representation of the code.



- When we say a stream of tokens, we mean that the lexical analyzer classifies the source code text into categories of atomic expressions.
- So for example $+, -, *, <>=$,etc, might all produce the token BINOP for binary operator.
- Note we don't throw away the the underlying symbol we keep track of the token BINOP and also keep track of a reference to the exact piece of source that produced that token which will be analyzed when necessary.

Simple Illustration



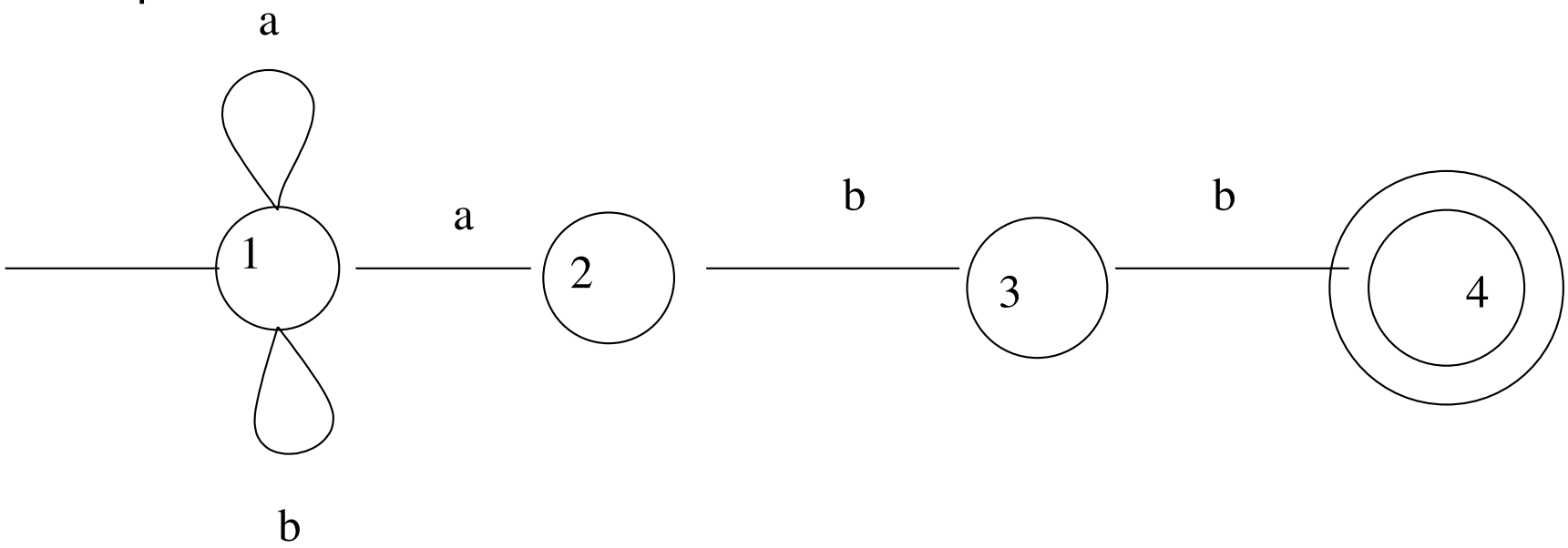
Plex

The cython code that does the lexical analysis is in the Plex module which is a general lexical analyzer. It takes a set of regular expressions and produces lexical analyzer from that. This is the scanner object. I found it interesting that the implementation of the lexical analyzer contains a very intricate use of finite automata (totally standard though).

For example the regular expression

```
(a|b)*abb
```

corresponds to the automata



You probably don't need to understand Plex other than to understand what it is doing. The general Plex lexical analyzer is specialized to the cython language in *Compiler/Scanning* where a PyrexScanner class is defined. The regular expressions controlling how the PyrexScanner tokenizes cython are contained in *Compiler/Lexicon.py*. (Look at Lexicon.py now).

Nodes.py,ExprNodes.py

- Befor we look at the parser let us discuss the building blocks of the parse tree. These are contained in *Compiler/Nodes.py* and *Compiler/ExprNodes.py* which are likely two of the most important files to understand if you are going to hack Cython.
- They contain a very large collection of node objects that the parser will link together to form the parse tree.
- The tree structure arises because complicated node object contain references to simpler more atomic node objects. For example there are binary operation nodes that contain references to nodes that represent what they are operating on, etc.
- The node objects have methods to generate the final C source code
- The Nodes in Nodes.py are higher level than those in ExprNodes.py which contains more atomic nodes.

An Example Node

```
class DefNode(FuncDefNode):
    # A Python function definition.
    #
    # name      string      the Python name of the function
    # args    [CArgDeclNode]   formal arguments
    # star_arg  PyArgDeclNode or None * argument
    # starstar_arg  PyArgDeclNode or None  ** argument
    # doc           string or None
    # body          StatListNode
    #
    #  The following subnode is constructed internally
    #  when the def statement is inside a Python class
                                definition.
    #
    #  assmt   AssignmentNode
                          Function  construction/assignment
```

Another Node

```
class FloatNode(ConstNode):
    type = PyrexTypes.c_double_type

    def compile_time_value(self, denv):
        return float(self.value)
```

Parsing.py

The second most important files for hacking cython is probably *Compiler/Parsing.py*. As its name would imply this is the parser. The entry point is p_statement_list, here *s* is a PyrexScanner lexical analyzer object.

```
def p_statement_list(s, level,
        cdef_flag = 0, visibility = 'private', api = 0):
    # Parse a series of statements separated by newlines.
    pos = s.position()
    stats = []
    while s.sy not in ('DEDENT', 'EOF'):
        stats.append(p_statement(s, level,
            cdef_flag = cdef_flag, visibility = visibility,
                                                api = api))
    if len(stats) == 1:
        return stats[0]
    else:
        return Nodes.StatListNode(pos, stats = stats)
```

- What happens is once the parser starts it repeatedly calls the PyrexScanner to get tokens and dispatches calls the the appropriate p_*function_name* functions which eventually call into *Nodes.py* to build up a tree of Node objects.
- Note that s is the scanner object doing the lexical analysis.
- s.sy is the current "symbol" or token name in the stream. s.position is the exact location of this symbol in the source file.
- s.systring is the actual string corresponding the the token.
- s.expect just checks that the next token matches what is expected and moves to the successive token. [[[Look at p_statement in emacs at this point]]]

For another example [[[Look at p_list_maker]]]

Symtab.py

Symtab is a very important file that I won't say much about (well I'll say some things)

- Symtab contains symbol table classes. It is used to organize information during the parsing and code generation process. These symbol tables are called scopes, there is a general scope object and a ModuleScope object. The scopes are passed into functions as the env variable.
- As a simple example the nodes in parse tree corresponding to most objects such as variables, functions, etc. each have a reference to the symbol table entry.
- The symbol table entry is created when the nodes is created or during the analysis phase.
- The symbol table keeps track of which variables correspond to which __pyx__⟨ ⟩_number names that occur in the generated C code

Code Generation

- Once the parse tree is constructed code generation proceeds by propagating calls down the tree to various methods that perform analysis and eventually code generation. The entry point into this part of the process is in *Compiler/ModuleNode.py* in the function process_implementation
- You can think of ModuleNodes as meta nodes which correspond to whole source code files and directories (modules).

```
def process_implementation(self, env, options, result):
  self.analyse_declarations(env)
  env.check_c_classes()
  self.body.analyse_expressions(env)
  env.return_type = PyrexTypes.c_void_type
  self.referenced_modules = []
  self.find_referenced_modules(env,self.referenced_modules,
                                                        {})
  if self.has_imported_c_functions():
      self.module_temp_cname=env.allocate_temp_pyobject()
      env.release_temp(self.module_temp_cname)
  self.generate_c_code(env, result)
  self.generate_h_code(env, options, result)
  self.generate_api_code(env, result)
```

I'm not going to say much about the code generation other than the fact that as indicated, function calls are propogated down the tree, prompting node objects to perform analysis, interact with the symbol table, and eventually emit C code. I don't really understand this in detail, and understanding it at all requires good understanding of the Python C api. Its best to look at some nodes and their associated functions to get a feel for what is happening.

Perhaps in contradiction to the previous slide. I'd like to now look a bit at the C code that results from cython and point out a few things.

example.pyx

- structure of cdef class variables
- vtable
- Example: List Comprehension Nodes

References

- Aho, Sethi, Ullman - “Compilers: Principles, Techniques, and Tools”
- The Cython Source