Python Compiler Compiler (pcc)

Release 0.1

Erich Blume <blume.erich@gmail.com>

January 13, 2012

CONTENTS

CHAPTER

ONE

PYTHON COMPILER COMPILER (PCC)

Context Free Grammar parsers for Python 3

1.1 License & Copying Notice

1.2 About This Document

The documentation for `pcc` is generated automatically. The first section of the official documentation comes directly from the `README.rst` file at the root level of the `pcc` project - and you may in fact be reading from that file right now! Alternately, you could be reading from "pcc.pdf", which is the full documentation for `pcc`.

1.2.1 How to use "pcc.pdf"

The first section of "pcc.pdf" comes directly from `README.rst`, and you are reading it now. The second section is the Cookbook - **new developers will (hopefully) find the Cookbook as a helpful place to start.** The Cookbook contains color-coded and well-explained code examples on how to use `pcc`.

Finally, the third section (entitled "`pcc` Package") is the automatically generated API documentation that comes directly from the code itself. Notably, any code examples in this section are actually executed as part of the testing suite (see Testing, below). One thing to note about this documentation is that you can generally skip any module with "_test" at the end of its name - this denotes a module that contains only unit test cases, and thus isn't of much use to anyone other than a `pcc` developer.

1.2.2 How to regenerate “pcc.pdf”

If you suspect your documentation is out of date, you can re-generate it by going in to the `docs/` subdirectory of this project and executing:

```
$ make pcc
```

Assuming you have all the required packages (LaTeX and Sphinx are both required, for instance - although some python packages may hopefully install themselves) the “pcc.pdf” file will be updated automatically.

1.3 Installation

`pcc` uses `distribute`, a `setuptools`-like distribution wrapper, to automate installation.

1.3.1 Requirements

- Python 3.0 or higher

1.3.2 Installation on Linux / OS X

Execute the following command (or similar) to install the module for all users:

```
$ sudo python3 setup.py install
```

You can also see a more complete list of build options by entering:

```
$ python3 setup.py --help
```

or by reading the `distribute` documentation.

1.3.3 Installation on Windows

Coming soon! This should be fairly simple, as `distribute` should take care of the tricky stuff. It probably works already, I just haven’t tested it yet.

1.3.4 Testing

It is a good idea to run the unit test suite, and it is generally very simple to do so. Unit testing is performed using `nose`, and is automated by `distribute`, and augmented with testing coverage reports by `coverage`.

To execute the full testing suite, enter the following command:

```
$ python3 setup.py nosetests
```

No errors should be reported by this process. Also note that while setuptools may report that there is a `tests` command, this has not been configured - use `nosetests` instead.

1.4 Release History

0.1 - 13 January 2012 Working LL(1) parser with semantic actions.

1.5 Contributing

See AUTHORS for a (hopefully) complete list of all contributors to this project. Please add your name and - if you like - your email to the list if you contributed anything you felt was meaningul to the project.

For tips and procedures on how to contribute - or to report a bug or leave feedback - please visit the `pcc` project page on github. Please feel free to be bold in submitting patches, tickets, or push requests - I won't be offended and would greatly appreciate the effort!

CHAPTER

TWO

PCC COOKBOOK

Common procedures, recipes, operations, and usages of pcc - with clear documentation and example code.

Coming soon!

CHAPTER THREE

PCC PACKAGE

3.1 pcc Package

pcc - Create parsers for context free grammars (CFGs).

See pcc.lexer for a lexical analyzer/tokenizer class.

See pcc.parser for several implementations of CFG parser generators that utilize the Lexer class from pcc.lexer.

3.2 lexer Module

lexer.py - Tokenize lexemes from strings, similar to F/Lex, but Pythonic.

class pcc.lexer.**Lexer**(*ignore_whitespace=True*, *ignore_newlines=True*, *report_literals=True*)

Bases: builtins.object

Create a new Lexer object.

If *ignore_whitespace* is left True, a token called WHITESPACE will be created with the rule r"\s+" with the *silent* option on - the effect of which is that bare whitespace will be effectively stripped from the input. Note that this overrides *ignore_newlines*.

If *ignore_newlines* is left True but *ignore_whitespace* is False, then a new token called NEWLINE will be added with the rule r"[\n]+" and the *silent* option on.

For either of these whitespace-skipping rules, keep in mind that you can still have tokens with whitespace due to the greedy nature of pattern matching.

If *report_literals* is left True, then special behavior is added to the case that no defined token matches the next bit of input. When False, ValueError will be raised. When True, a special token named ‘LITERAL’ will be returned with the next **single** input character as its matching lexeme. This is helpful for grammar parsers (see pcc.parser) to define rules with custom literal symbols rather than creating an explicit token for each one. If you want to have a multi-character string literal, just define a token with the ‘rule’ as exactly the text you want to match.

```
>>> p = Lexer()
>>> p.addtoken(name='NAME',rule=r'[_a-zA-Z][_a-zA-Z0-9]+')
>>> p.addtoken(name='REAL_NUMBER',
...            rule=r'(-)?([1-9][0-9]*(\.[0-9]+)?|0\.[0-9]+)')
>>> p.addtoken(name='TWO_WORDS',rule=r'[a-zA-Z]+ [a-zA-Z]+')
>>> p.addtoken(name='MULTI_LINE',rule=r'foo\nbar')
>>> input = '''42 is a number
```

```
... 3.14159 foo
... bar sameline
... _Long_Identifier_ banana
... ocelot!!
... '''
>>> for lexeme in p.lex(input):
...     print("{} > {} | {},{}".format(lexeme.token.name, lexeme.match,
...                                     lexeme.line, lexeme.position))
REAL_NUMBER > 42 | 1,1
TWO_WORDS > is a | 1,4
NAME > number | 1,9
REAL_NUMBER > 3.14159 | 2,1
MULTI_LINE > foo
bar | 2,9
NAME > sameline | 3,5
NAME > _Long_Identifier_ | 4,1
NAME > banana | 4,19
NAME > ocelot | 5,1
LITERAL > ! | 5,7
LITERAL > ! | 5,8
```

addtoken(*token=None*, *name=None*, *rule=None*, *silent=False*)

Add the specified `pcc.symbols.Token` object to the lexer.

Either *token* or both *name* and *rule* need to be specified.

token, if used, must be a `pcc.symbols.Token` object which will be used in the manner described in that classes documentation. The *silent* flag from this function will be ignored, and instead the same field from the `Token` object will be honored.

If *token* is not specified, then *name* and *rule* must be this new token's name and lexing regular expression respectively. As per `pcc.symbols.Token`'s documentation, the *silent* flag denotes that this token will not be yielded by this `Lexer`'s `lex` generator, although the token may still consume input. This is useful for ignoring whitespace and comments, if that is desired.

The name of the token must be unique to this `Lexer`.

lex(*input*)

Generator that produces `Lexeme` objects from the input string.

3.3 `lexer_test` Module

This module provides unit tests for the `pcc.lexer` module.

As with all unit test modules, the tests it contains can be executed in many ways, but most easily by going to the project root dir and executing `python3 setup.py nosetests`.

class `pcc.lexer_test.`**LexerTester**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Test harness for `pcc.lexer.Lexer` class.

setUp()

Create the testing environment

tearDown()

Remove the testing environment

test_SimpleTokens()
lexer.py: Test a simple lexicon

test_greedy()
lexer.py: Test greedy matching

test_ignorenewlines()
lexer.py: Test ignoring newlines

test_literal()
lexer.py: Test string literals

3.4 `ll` Module

ll.py - Implementation of an LL(1) (Left to right Leftmost) parser generator

class `pcc.ll.`**LLParser**(*lexer*)
Bases: `pcc.parser.Parser`

addproduction(*symbol*, *rule*, *action*, *start_production=False*)

finalize()
This function actually performs the 'parser generation' that gives `pcc` its name (compiler compiling). Callable only once, it constructs the FIRST and FOLLOW sets, the parsing table, the action table, etc.

In general terms, it prepares the parser to be able to call 'parse'.

first(*symbols*)
Return the set of terminal ``Symbol``s which belong to this string's FIRST set. See Aho, Ullman et.al.'s 2nd edition "Compilers...", section 4.4.2.

symbols must be a `pcc.symbols.SymbolString` object.

The returned value will be a set of terminal``Symbol`` objects.

follow(*symbol*)
Compute the FOLLOW set of a nonterminal symbol.

Due to the constraint programming approach used to calculate the FOLLOW sets in this implementation, this function merely returns the pre-computed set - the actual set computation occurs in finalize()

parse(*input*)
Use the recursive descent method to parse the input.

3.5 `ll_test` Module

This module provides unit tests for the `pcc.ll` module.

As with all unit test modules, the tests it contains can be executed in many ways, but most easily by going to the project root dir and executing `python3 setup.py nosetests`.

class `pcc.ll_test.`**LLTester**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Test harness for `pcc.ll.LLParser` class.

setUp()
Create the testing environment

tearDown()
Remove the testing environment

test_first()
ll.py: Test FIRST set creation

test_follow()
ll.py: Test FOLLOW set creation

test_grammar()
ll.py: Test a basic LL(1) grammar

3.6 parser Module

parser.py - CFG parsing with python functions as semantic actions

Create Parser objects, which take tokenized input (via lexer.py) and use different parsing methods to synthesize BNF grammars in to productions with semantic python actions. All semantic actions are python functions that take a list as an argument (representing the values of sub-expressions) and return a value as the result for that expression.

exception pcc.parser.**GrammarError**
Bases: builtins.Exception

Exception raised by a parser.Parser object *before* parsing begins.

class pcc.parser.**Parser**
Bases: builtins.object

CFG parser of arbitrary BNF-like grammars.

Actual implementations should subclass this abstract base class, and may have slightly different levels of ‘computational power’. That is to say, an LL language is slightly less expressive than an SLR, which is less expressive than an LALR, which is less expressive than an LR language. All are essentially ‘context free grammars’. If a parser encounters an error before parsing has begun (ie. an error with a production) then parser.GrammarError should be raised. If a parser encounters a syntax error during parsing, then parser.ParsingError should be raised. Other errors should be reported in the most reasonably expressive manner possible.

Use addproduction to add productions with semantic actions. Use parse to execute parsing. Parsing is immediate and returns no value, unlike other parsers which might work stepwise - this is because this class takes advantage of python generator functions.

The following example emulates this simple grammar (in bison/YACC form):

```
expr : expr ’+’ term   { $$ = $1 + $3; }
     | expr ’-’ term   { $$ = $1 - $3; }
     | term            { $$ = $1; }
     ;

term : ’(’ expr ’)’    { $$ = $2; }
     | num             { $$ = $1; }
     ;

num : ’0’              {$$ = 0; }
    | ’1’              {$$ = 1; }
    [.... etc, for each number 0-9 ....]
    ;
```

Unfortunately, at this time the example doesn't work due to there not yet being a working LR parser. Therefore the following code example, which still gives you a good idea for the syntax, is greyed-out:

```
>> from pcc.lexer import Lexer
>> l = Lexer()
>> l.addtoken(name='NUMBER',rule=r'[0-9]+')
>> p = parser(l)
>>
>> p.addproduction('prog', "expr", lambda v: v[0], start_production=True)
>>
>> p.addproduction('expr', "expr '+' term", lambda v: v[0] + v[2])
>> p.addproduction('expr', "expr '-' term", lambda v: v[0] - v[2])
>> p.addproduction('expr', "term", lambda v: v[0])
>>
>> p.addproduction('term', " '(' expr ')' ", lambda v: v[1])
>> p.addproduction('term', " NUMBER ", lambda v: int(v[0]))
>>
>> p.parse("5-( 9+ 2 )")
```

-6

`addproduction`(*symbol*, *rule*, *action*, *start_production=False*)

Add a production (rule) to the grammar of the parser. Instructs the parser that *symbol* can be derived using *rule*, producing a result via *action*.

symbol is a string that must match the regular expression `r'[a-zA-Z][_a-zA-Z]*'`. Note that it is not allowed for a symbol to have the same name as a token that might be produced by the lexer. Therefor, if you try to add a rule with a *symbol* that is already the name of a token, `GrammarError` will be raised immediatly, and the rule will not be added.

rule is a string that has whitespace seperated terminal and nonterminal symbols (see below). To specify an empty production (X->epsilon), simply make *rule* be an empty string (NOT `None`).

action is a function (often a lambda expression, but there is no such requirement) that takes a list as input and may return some value. The list will be filled with returned values of the derivation actions of the symbols in the rule, in order, starting from 0. (See the example in the Parser class documentation for a more clear explanation.) Alternately, action may be any other value, in which case that value will be used for upper derivation actions direcly.

One and only one production must be marked as the *start_production*. All parsing will derive this production at the topmost level, and it is a syntax error if input remains after this production has concluded.

You may define rules that derive the same symbol any number of times, but keep in mind that what you are doing is defining multiple derivations. All symbols used in a rule must have a derivation before parsing, although you can use a symbol in a new rule before it has a derivation (so long as you eventually give it a derivation).

Terminal and Nonterminal Symbols

Recall that the *rule* parameter is a whitespace-seperated strings of 'symbols' (or a list of symbols). A symbol is defined as either:

1. All of the named tokens in the lexer, which have names conforming to the regex found in `lexer._token_ident`.
2. String literals, which are identified in the rule as any single character between two single quotes (including, possibly, a single quote itself - which would be three single quotes one after another.)
3. Any nonterminal symbol (see below), although keep in mind that all nonterminal symbols must have at least one derivation before parsing can commense.

1 and 2 describe "terminal symbols", in that they have no derivations and exist directly in the input stream. 3, the nontermimal symbols, are exactly the set of strings called "symbol" that get passed to this function.

Note that string literals are only supported when using `lexer.Lexer`'s *report_literals* option (which is on by default). It is also perfectly acceptable to use "LITERAL" as a named token, but recall that this will match *any* string literal, not just the specific one you could have given if you had used the single-quote shortcut.

It is currently very cumbursome to try and match multi-character string literals, so it is generally suggested to wrap such keywords in a token specificically for that keyword instead.

`ap`(**args*, ***kwargs*)

A shorthand for `addproduction`.

`parse`(*input*)

Parse the given *input* (a string) using the given rules and lexer.

Note that this parser does not return any value, unlike other parsers which might return a syntax tree. Instead, this function will produce the desired final product by means of the *action* functions given in `Parser.addproduction`.

exception `pcc.parser.`**`ParsingError`**

Bases: `builtins.Exception`

Exception raised by a `parser.Parser` object when a syntax error occurs.

`pcc.parser.`**`parser`**(*lexer*)

Create a Parser using the default algorithm.

3.7 `symbols` Module

symbols.py - Helper classes for grammar symbols, sentences, tokens, etc.

class `pcc.symbols.`**`Lexeme`**(*token*, *match*, *line*, *position*)

Bases: `builtins.object`

Input (string) that has been matched to some `Token`.

Every `Lexeme` object has the fields `token`, `match`, `line`, and `position`.

class `pcc.symbols.`**`Symbol`**(*name*)

Bases: `builtins.object`

Terminal or nonterminal member of a grammar.

By default, objects instantiated from Symbol will return `False` when queried by `Symbol.terminal` - subclasses may implement other behavior as needed.

name may be anything that matches `SYMBOL_MATCH_RULE`, a string that is used as a regular expression by this module.

`terminal`()

class `pcc.symbols.`**`SymbolString`**(*symbols*)

Bases: `builtins.object`

An ordered collection of `Symbol` objects.

Immutable, hashable, iterable, sliceable, etc. Works like strings.

class `pcc.symbols.`**`Token`**(*name*, *rule*, *silent=False*)

Bases: `pcc.symbols.Symbol`

Terminal Symbol

rule is a string that will be used as a regular expression to lex input by `pcc.lexer`. To avoid lexing issues, use only basic regular expression operations - avoid backreferences, unnecessary grouping, or the input boundary markers (^ and $). Lexing may still work with these, but your mileage will vary.

silent, if True, will tell lexers to supress generating this token. Input will still be consumed, but no token will be sent to the caller. This is primarily useful for ignore whitespace, comments, etc.

There are two special `Token``s that do not follow normal naming rules (their names start with an underscore) in this module: ``EOF` and `EPSILON`. `EPSILON` can be used for 'empty' productions, and `EOF` can be used to represent the end of the stream. Do not use either of these special tokens in lexing - they will produce undefined results.

`match`(*input*, *position*)

Returns a string matched from `input[position:]`, or None.

`terminal`()

3.8 `symbols_test` Module

This module provides unit tests for the `pcc.symbols` module.

As with all unit test modules, the tests it contains can be executed in many ways, but most easily by going to the project root dir and executing `python3 setup.py nosetests`.

class `pcc.symbols_test.`**`SymbolStringTester`**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Test harness for the `pcc.symbols.SymbolString` class.

`setUp`()

`tearDown`()

`test_sstring`()

symbols.py: Test SymbolString class.

`test_substring`()

symbols.py: Test SymbolString substrining.

class `pcc.symbols_test.`**`SymbolTester`**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Test harness for the `pcc.symbols.Symbol` class.

`setUp`()

`tearDown`()

`test_badname`()

symbols.py: Test invalid symbol names

`test_symbol`()

symbols.py: Test Symbol class

class `pcc.symbols_test.`**`TokenTester`**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Test harness for the `pcc.symbols.Token` class.

setUp()

tearDown()

test_token()

symbols.py: Test Token class

PYTHON MODULE INDEX

INDEX