pcc Documentation

Release 0

Erich Blume <blume.erich@gmail.com>

January 05, 2012

CONTENTS

1 License & Copying Notice **3**

2 Installation **5**
- 2.1 Requirements . . . 5
- 2.2 Installation on Linux / OS X . . . 5
- 2.3 Installation on Windows . . . 5
- 2.4 Testing . . . 5

3 Documentation **7**

4 Release History **9**

5 Contribution **11**

6 pcc Cookbook **13**

7 pcc Package **15**
- 7.1 `pcc` Package . . . 15
- 7.2 `lexer` Module . . . 15
- 7.3 `lexer_test` Module . . . 16
- 7.4 `parser` Module . . . 16

Python Module Index **19**

Index **21**

Python Compiler Compiler (pcc)

Context Free Grammar parsers for Python 3

CHAPTER

ONE

LICENSE & COPYING NOTICE

This file is part of pcc.

pcc is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

pcc is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with pcc, in a file called COPYING. If not, see <http://www.gnu.org/licenses/>.

CHAPTER

TWO

INSTALLATION

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

2.1 Requirements

- Python 3.0 or higher

2.2 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.

2.3 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.

2.4 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.

CHAPTER

THREE

DOCUMENTATION

Please see "pcc.pdf" at the root project directory for the complete documentation. The `docs` directory can be ignored unless you wish to edit or regenerate the documentation from the source.

To regenerate the documentation (generally only performed by me before a release), go to the `docs` directory and execute the following command:

```
$ make pcc
```

This will automatically (using Sphynx) scan the source code, build the API documentation in to ReStructuredText, transcribe the docs to LaTeX, and compile the LaTeX source in to "pcc.pdf" at the project's root directory.

CHAPTER

FOUR

RELEASE HISTORY

No release has been made yet. How cool are you?

CHAPTER

FIVE

CONTRIBUTION

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

SIX

PCC COOKBOOK

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

Coming soon!

CHAPTER

SEVEN

PCC PACKAGE

7.1 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`.

7.2 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 True, then in the case that no token matches the current input sequence, rather than raising an error, a token with the name `'LITERAL'` will be created with the next character of input (and only one character).

```
>>> p = Lexer()
>>> p.addtoken('NAME',r'[_a-zA-Z][_a-zA-Z0-9]+')
>>> p.addtoken('NUMBER',r'[0-9]+')
>>> p.addtoken('REAL_NUMBER',r'(-)?([1-9][0-9]*(\.[0-9]+)?|0\.[0-9]+)')
>>> p.addtoken('TWO_WORDS',r'[a-zA-Z]+ [a-zA-Z]+')
>>> input = '''42 is a number
... 3.14159
... _Long_Identifier_ banana
... ocelot!!
... '''
```

```
>>> for token in p.lex(input):
...     print("{} > {} | {},{}".format(token.name, token.match,
...                                    token.line, token.position))
NUMBER > 42 | 1,0
TWO_WORDS > is a | 1,3
NAME > number | 1,8
REAL_NUMBER > 3.14159 | 2,0
NAME > _Long_Identifier_ | 3,0
NAME > banana | 3,18
NAME > ocelot | 4,0
LITERAL > ! | 4,6
LITERAL > ! | 4,7
```

addtoken(*name*, *rule*, *silent=False*)

Add a token-generating rule.

name is a unique (to this Lexer) identifier which must match the regular expression `"[a-zA-Z][_a-zA-Z0-9]*"` - it may also not be named `'LITERAL'`, as this is reserved. Classically (and to help avoid conflicts with parser symbols), all tokens should be named in all capitals with underscores between words.

rule is a regular expression in a string (preferably a 'raw' string, but that's up to the user) that will be passed to `re.match` to find a token. Avoid using complex regular expressions to avoid breaking the system - try to just use literals, literal groups, and quantifiers, and definitely do not use position metacharacters like "^" and "$" or back- -references.

silent, if set to True, will supress the generation of this token as a result from the `lex` method. In this case, the token will still be lexed (and input consumed), but the `lex` method won't generate the token as output.

lex(*input*)

Generator that produces Token objects from the input string.

class `pcc.lexer.`**Token**(*name*, *match*, *line*, *position*)

Bases: `builtins.object`

A container for a token name, a bit of input that matched that token's lexing rule, and the position in the input stream at which it occured.

Every Token object has the fields `name`, `match`, `line`, and `position`.

7.3 `lexer_test` Module

7.4 `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 rules 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.

Use the helper method `parser()` to create a Parser object if you don't care about what algorithm it will use. Otherwise, you cannot instantiate a Parser object directly, but must instead use on of its subclasses, each of which uses a slightly different algorithm. (It is the author's plan to use this setup as a way to read through the 'Purple Dragon Book' - Aho, Lam, Sethi, and Ullman.

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 rules.

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 rule) then `parser.GrammarError` should be raised. If a parser encounters a syntax error during parsing, then `parser.SyntaxError` should be raised. Other errors should be reported in the most reasonably expressive manner possible.

Use `addrule` to add rules with semantic actions. Use `parse` to execute parsing. Parsing is immediate and returns no value, unlike other LALR 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 0-0 ....]
    ;
```

Note: the code below has been 'mangled' in formatting to avoid a minor test issue present in this early build. I apologze for its ugliness, it will be fixed shortly.

```
>> l = Lexer()
>> l.addtoken('NUMBER',r'[0-9]+')
>> p = parser(l)
>>
>> p.addrule('prog', "expr", lambda v: print(v[0]))
>>
>> p.addrule('expr', "expr '+' term", lambda v: v[0] + v[2])
>> p.addrule('expr', "expr '-' term", lambda v: v[0] - v[2])
>> p.addrule('expr', "term", lambda v: v[0])
>>
>> p.addrule('term', " '(' expr ')' ", lambda v: v[1])
>> p.addrule('term', " NUMBER ", lambda v: v[0])
>>
>> p.parse("5-(9+2)")
-6
```

`addrule`(*symbol*, *rule*, *action*)

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]+'`. 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, ValueError 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). It may also be a list of such strings (with all whitespace stripped). In other words, if the rule is `"foo '*' bar"` you could equivalently use rule.split() - both work just as well.

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.

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).

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 (at this writing, `r'[a-zA-Z][_a-zA-Z0-9]*'`).
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 (*addrule*).

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 combursome 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.

parse(*input*)

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

Note that some parser generators produce parsers that will return an Abstract Syntax Tree (AST) as a result, allowing the caller to walk the tree and produce the desired final product. However, this parser does not do that.

Instead, this function will return nothing, but will produce the desired final product by means of the *action* functions given in `Parser.addrule`.

exception `pcc.parser.`**SyntaxError**

Bases: `builtins.Exception`

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

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

Create a `parser.Parser` object using the default algorithm.

PYTHON MODULE INDEX

p

pcc.__init__, 15
pcc.lexer, 15
pcc.parser, 16

INDEX

A

addrule() (pcc.parser.Parser method), 17
addtoken() (pcc.lexer.Lexer method), 16

G

GrammarError, 16

L

lex() (pcc.lexer.Lexer method), 16
Lexer (class in pcc.lexer), 15

P

parse() (pcc.parser.Parser method), 18
Parser (class in pcc.parser), 17
parser() (in module pcc.parser), 18
pcc.__init__ (module), 15
pcc.lexer (module), 15
pcc.parser (module), 16

S

SyntaxError, 18

T

Token (class in pcc.lexer), 16