# Parsing S-expressions with Python 3

Justus Perlwitz

2015-11-08

## Contents

1 The Tokenizer 1

2 The Parser 1

This is a simple s-expression parser written in Python 3. It understands symbols and numbers and uses tuples to represent the data internally.

## 1 The Tokenizer

```
def tokenize(s):
    """
    >>> tokenize("(quote (+ 1 2) + (1 (1)))")
    ['(', 'quote', '(', '+', 1, 2, ')', '+', '(', 1, '(', 1, ')', ')', ')']
    """
    return list(map(lambda s: int(s) if s.isnumeric() else s, filter(
        bool, s.replace('(', '( ').replace(')', ' )').split(" ")))
```

First, the tokenizer adds padding to left and right parentheses. Then it splits the raw token stream by space characters. As a result empty items will appear, since `(` will turn into `(`, which will be split into `(`, ‘`. That is why the surroundingfilterdiscards empty items usingbool. Finally, the tokenizer turns all numeric tokens intoint`s. It returns the resulting token stream as a list.

## 2 The Parser

The token stream can now be parsed.

```
def parse(tokens):
    """
    >>> parse(['(', '(', '+', 1, 2, ')', '+', '(', 1, '(', 1, ')', ')', ')'])
    (('+', 1, 2), '+', (1, (1,)))
    """
    def _list():
        tokens.pop(0)
        while tokens[0] != ')':
            yield parse(tokens)
        tokens.pop(0)
    return tuple(_list()) if tokens[0] == '(' else tokens.pop(0)
```

A valid s-expression is either an atom (int or symbol) or a list of s-expressions. Since we operate on a token stream, the parser has to peek at the current token and then either parse a list or parse an atom.

A recursive descent parser lends itself to this type of recursive grammar.

All done!