Interpreter LISP - Michał Flak

Wstęp

Lispy są wyjątkowo prostymi językami do parsowania. W tym projekcie implementuję interpreter prostego lispa opartego na Scheme.
Użyłem bibloteki SLY oraz języka Python.
Interpreter oferuje pętle REPL jak i interpretowanie pliku.

W celu zainstalowania dependencji, należy uruchomić:

```
pip install poetry
poetry install
```

1. Lekser

```
class LispLexer(Lexer):
    # Set of token names.   This is always required
    tokens = { SYMBOL, STRING, NUMBER }

    SYMBOL = r'[a-zA-Z_+=\*\-\<\>][a-zA-Z0-9_+\*\-\<\>]*'

    @_(r'"(.*?)"')
    def STRING(self, t):
        t.value = t.value[1:-1] # Strip quotes
        return t

    @_(r'\d+')
    def NUMBER(self, t):
        t.value = int(t.value)   # Convert to a numeric value
        return t

    literals = { '(', ')' }

    # String containing ignored characters between tokens
    ignore = ' \t'

    ignore_comment = r'\#.*'
    ignore_newline = r'\n+'
```

2. Parser

Gramatyka (EBNF):

```
seq = expr , seq | ;
expr = SYMBOL
       | STRING
       | NUMBER
       | list ;
list = "(" , seq , ")" ;
```

Przykładowe wywołanie:

```
(venv) work@pop-os:~/repos/lisp-interpreter$ python3 lisp-interpreter.py -f
tests/fizzbuzz-lambda.scm
Parser debugging for LispParser written to parser.out
Source:
(define fizzbuzz (lambda (x y) (
  (display
    (cond (( = (modulo x 15) 0 ) "FizzBuzz")
          (( = (modulo x 3) 0 ) "Fizz")
          (( = (modulo x 5) 0 ) "Buzz")
          (else x)))
  (newline)

  (if (< x y) (fizzbuzz (+ x 1) y)))))

(fizzbuzz 1 100)
Parsed AST:  [[define, fizzbuzz, [lambda, [x, y], [[display, [cond, [[=, [modulo,
x, 15], 0], FizzBuzz], [[=, [modulo, x, 3], 0], Fizz], [[=, [modulo, x, 5], 0],
Buzz], [else, x]]], [newline], [if, [<, x, y], [fizzbuzz, [+, x, 1], y]]]]],
[fizzbuzz, 1, 100]]
```

Nadpisałem funkcję `error` żeby rzucała wyjątki, co jest pomocne w REPL.

Stworzyłem klasę Token oraz podklasy reprezentujące konkretne typy w celu rozróżniania ich w drzewie AST.

Kod źródłowy:

```python
import sys
from sly import Parser
from lexer import LispLexer

class Token():
    def __init__(self, x):
        self.x = x

    def __str__(self):
        return str(self.x)

    def __repr__(self):
        return str(self.x)

    def __eq__(self, other):
        if isinstance(other, Token):
            return self.x == other.x
        elif isinstance(other, str):
            return self.x == other

    def __hash__(self):
        return hash(self.x)

class StringToken(Token):
    pass

class NumberToken(Token):
    pass
```

```
class SymbolToken(Token):
    pass

class LispParser(Parser):
    # Get the token list from the lexer (required)
    tokens = LispLexer.tokens
    debugfile = 'parser.out'

    # Grammar rules and actions
    @_('expr seq')
    def seq(self, p):
        return [p.expr, *p.seq] if p.seq else [p.expr]

    @_('')
    def seq(self, p):
        pass

    @_('SYMBOL')
    def expr(self, p):
        return SymbolToken(p[0])

    @_('STRING')
    def expr(self, p):
        return StringToken(p[0])

    @_('NUMBER')
    def expr(self, p):
        return NumberToken(p[0])

    @_('list_')
    def expr(self, p):
        return p[0]

    @_('"(" seq ")"')
    def list_(self, p):
        return p.seq

    def error(self, token):
        if token:
            lineno = getattr(token, 'lineno', 0)
            if lineno:
                sys.stderr.write(f'sly: Syntax error at line {lineno}, token=
{token.type}\n')
                raise Exception
            else:
                sys.stderr.write(f'sly: Syntax error, token={token.type}')
                raise Exception
        else:
            raise EOFError
```

3. Ewaluacja drzewa

3.1. Środowisko

Środowisko (Environment) przechowuje wbudowane funkcje, operatory i stałe, jak również funkcje i zmienne zdefiniowane przez użytkownika.
Reprezentowane jest jako coś w rodzaju listy, gdzie kazde pod-środowisko ma dostęp do zawartości nad-środowiska.
Ta część jest w dużej mierze oparta na interpreterze Petera Norviga (http://norvig.com/lispy.html), dostosowanym jednak do SLY.

```
class Env(dict):
    def __init__(self, parms=(), args=(), outer=None):
        self.update(zip(parms, args))
        self.outer = outer
    def find(self, var):
        "Find the innermost Env where var appears."
        return self if (var in self) else self.outer.find(var) if self.outer is
not None else None


def standard_env() -> Env:
    env = Env()
    env.update(vars(math)) # sin, cos, sqrt, pi, ...
    env.update({
        '+':op.add, '-':op.sub, '*':op.mul, '/':op.truediv,
        '>':op.gt, '<':op.lt, '>=':op.ge, '<=':op.le, '=':op.eq,
        'abs':     abs,
        'append':  op.add,
        'apply':   lambda proc, args: proc(*args),
        'begin':   lambda *x: x[-1],
        'car':     lambda x: x[0],
        'cdr':     lambda x: x[1:],
        'cons':    lambda x,y: [x] + y,
        'display': lambda x: print(x, end=""),
        'eq?':     op.is_,
        'expt':    pow,
        'equal?':  op.eq,
        'length':  len,
        'list':    lambda *x: list(x),
        'list?':   lambda x: isinstance(x, list),
        'map':     map,
        'max':     max,
        'min':     min,
        'modulo':  op.mod,
        'newline': print,
        'not':     op.not_,
        'null?':   lambda x: x == None,
        'number?': lambda x: isinstance(x, int),
        'print':   print,
        'procedure?': callable,
    })
    return env
```

3.2. Ewaluacja

Wprowadzone zostaje pojęcie procedury do reprezentowania procedur zdefiniowanych przez użytkownika.

```
class Procedure(object):
    def __init__(self, parms, body, env):
        self.parms, self.body, self.env = parms, body, env
    def __call__(self, *args):
        return eval(self.body, Env(self.parms, args, self.env))


def eval(x, env):
    if os.environ.get('DEBUG'):
        print(f"Evaluating {x}:")

    if x is None:
        return None
    elif isinstance(x, SymbolToken):      # variable reference
        return env.find(x.x)[x.x]
    elif isinstance(x, NumberToken) or isinstance(x, StringToken):  # constant
literal
        return x.x # see Token class

    # Lists from now onwards
    elif len(x) == 0:              # ()
        return None
    elif x[0] == 'quote':          # (quote exp)
        (_, exp) = x
        return exp
    elif x[0] == 'if' and len(x) == 3:             # (if test conseq alt)
        (_, test, conseq) = x
        exp = (conseq if eval(test, env) else None)
        return eval(exp, env)
    elif x[0] == 'if' and len(x) == 4:             # (if test conseq alt)
        (_, test, conseq, alt) = x
        exp = (conseq if eval(test, env) else alt)
        return eval(exp, env)
    elif x[0] == 'cond' or x[0] == 'case':  # (cond (<test> <conseq>) ...)
        for (test, conseq) in x[1:]:
            if test == 'else' or eval(test, env):
                return eval(conseq, env)
    elif x[0] == 'define':         # (define var exp)
        (_, var, exp) = x
        env[var] = eval(exp, env)
    elif x[0] == 'set!':           # (set! var exp)
        (_, var, exp) = x
        env.find(var)[var] = eval(exp, env)
    elif x[0] == 'lambda':         # (lambda (var...) body)
        (_, parms, body) = x
        return Procedure(parms, body, env)
    else:                          # (proc arg...)
        proc = eval(x[0], env)
        args = [eval(exp, env) for exp in x[1:]]

        if os.environ.get('DEBUG'):
            print("X: ", x)
```

```
            print("Proc: ", proc)
            print("Args: ", args)

        if not callable(proc):
            return proc
        return proc(*args)
```

4. CLI / REPL

Program akceptuje albo wejście z pliku, w postaci:

```
python3 lisp-interpreter.py -f tests/fizzbuzz-lambda.scm
```

Albo bez argumentów, wtedy wchodzi w REPL:

```
python3 lisp-interpreter.py
```

REPL obsluguje:

- błędy składniowe / w ewaluacji (nie crashuje)
- wejście wieloliniowe
- anulowanie wejścia wieloliniowego

```
argparser = argparse.ArgumentParser(description='Lisp Interpreter')
argparser.add_argument('-f', '--file', help='File to interpret',
type=argparse.FileType('r'))

args = argparser.parse_args()
lexer = LispLexer()
parser = LispParser()

def repl():
    env = standard_env()
    prompt = "> "
    content = ""
    while True:
        try:
            content += input(prompt)
            tokens = lexer.tokenize(content)
            ast = parser.parse(tokens)
            val = eval(ast, env)
            if val is not None:
                print(prettyprint(val))
        except EOFError:
            # multiline input
            prompt = ''
            continue
        except Exception as ex:
            # eval error / syntax error
            print(ex)
        except KeyboardInterrupt:
            # cancel multiline input
            if prompt == "":
                print()
            else:
```

```
                raise

        content = ""
        prompt = "> "


def prettyprint(exp):
    if isinstance(exp, List):
        return '(' + ' '.join(map(prettyprint, exp)) + ')'
    else:
        return str(exp)

def eval_file(file):
    with args.file as file:
        content = file.read()
        print("Source: ")
        print(content)

        tokens = lexer.tokenize(content)
        ast = parser.parse(tokens)
        print("Parsed AST: ", ast)

        result = eval(ast, standard_env())

        if result:
            print(prettyprint(result))


if not args.file:
    # Run as REPL
    repl()
    exit()
else:
    # Run file
    eval_file(args.file)
```

5. Przykładowe wywołania

5.1. Z pliku

```
(venv) work@pop-os:~/repos/lisp-interpreter$ python3 lisp-interpreter.py -f
tests/fizzbuzz-lambda.scm
Parser debugging for LispParser written to parser.out
Source:
(define fizzbuzz (lambda (x y) (
  (display
    (cond (( = (modulo x 15) 0 ) "FizzBuzz")
          (( = (modulo x 3) 0 ) "Fizz")
          (( = (modulo x 5) 0 ) "Buzz")
          (else x)))
  (newline)

  (if (< x y) (fizzbuzz (+ x 1) y)))))

(fizzbuzz 1 100)
```

```
Parsed AST:  [[define, fizzbuzz, [lambda, [x, y], [[display, [cond, [[=, [modulo,
x, 15], 0], FizzBuzz], [[=, [modulo, x, 3], 0], Fizz], [[=, [modulo, x, 5], 0],
Buzz], [else, x]]], [newline], [if, [<, x, y], [fizzbuzz, [+, x, 1], y]]]]],
[fizzbuzz, 1, 100]]
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
```

5.2. Z REPL

```
(venv) work@pop-os:~/repos/lisp-interpreter$ python3 lisp-interpreter.py
Parser debugging for LispParser written to parser.out
> (* 21 37)
777
> (define fizzbuzz (lambda (x y) (
  (display
    (cond (( = (modulo x 15) 0 ) "FizzBuzz")
          (( = (modulo x 3) 0 ) "Fizz")
          (( = (modulo x 5) 0 ) "Buzz")
          (else x)))
  (newline)

  (if (< x y) (fizzbuzz (+ x 1) y)))))
> (fizzbuzz 5 20)
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
>
```

Źródła

lispy: http://norvig.com/lispy.html
gramatyka lisp: https://theory.stanford.edu/~amitp/yapps/yapps-doc/node2.html
dokumentacja SLY: https://sly.readthedocs.io/en/latest/sly.html
programy testowe: http://rosettacode.org/wiki/Category:Scheme