demo

December 12, 2023

[ ]:

```
class Visitor(object):
    pass
```

[ ]:

```
from rply import LexerGenerator

lg = LexerGenerator()

lg.add('OPEN_SOLUCION', r'<solucion>')
lg.add('CLOSE_SOLUCION', r'</solucion>')


lg.add('NUMBER', r'\d+(.\d+)?')
lg.add('PLUS', r'\+')
lg.add('MINUS', r'-')
lg.add('MUL', r'\*')
lg.add('DIV', r'/')

lg.add('OPEN_PARENS', r'\(')
lg.add('CLOSE_PARENS', r'\)')
lg.add('OPEN_BRACKET', r'\{')
lg.add('CLOSE_BRACKET', r'\}')


lg.add('VAR', r'var')
lg.add('INT', r'entero')
lg.add('FLOAT', r'fraccionario')
lg.add('ELSE', r'sino')
lg.add('IF', r'si')
lg.add('WHILE', r'mientras')
lg.add('FUNCION', r'funcion')
lg.add('RETURN', r'devolver')

lg.add('ID', r'[a-zA-Z][a-zA-Z0-9]*')
lg.add('COMP', r'==|!=|<=|>=|<|>')
lg.add('EQUALS', r'=')
lg.add('SEMICOLOM', r'\;')
lg.ignore(r'\s+') # ignore spaces
lg.ignore(r'\n+') # ignore newlines


lexer = lg.build()
```

[ ]:

```
#ÁRVORE SINTÁTICA PREPARADA PARA RECEBER VISITORS

from rply.token import BaseBox

class Solucion(BaseBox):
    def __init__(self, decls,stmts):
        self.vardecls = decls
        self.stmts = stmts

    def accept(self, visitor):
        visitor.visit_solucion(self)

class VarDecls(BaseBox):
    def __init__(self, decl,decls):
        self.vardecl = decl
        self.vardecls = decls

    def accept(self, visitor):
        visitor.visit_vardecls(self)

class VarDecl(BaseBox):
    def __init__(self, id,tp):
        self.id = id
        self.tp = tp

    def accept(self, visitor):
        visitor.visit_vardecl(self)

class Statements(BaseBox):
    def __init__(self, stmt,stmts):
        self.stmt = stmt
        self.stmts = stmts

    def accept(self, visitor):
        visitor.visit_statements(self)

class Statement(BaseBox):
    def __init__(self,stmt):
        self.stmt = stmt

    def accept(self, visitor):
        visitor.visit_statement(self)

class Atrib(BaseBox):
    def __init__(self, id,expr):
        self.id = id
        self.expr = expr

    def accept(self, visitor):
        visitor.visit_atrib(self)

class IfElse(BaseBox):
    def __init__(self, expr1, comp, expr2, ie1,ie2):
        self.expr1=expr1
        self.comp = comp.getstr()
        self.expr2=expr2
        self.ie1=ie1
        self.ie2=ie2

    def accept(self, visitor):
        visitor.visit_ifelse(self)


class While(BaseBox):
    def __init__(self, expr1, comp, expr2, ie1):
        self.expr1=expr1
        self.comp = comp
        self.expr2=expr2
        self.ie1=ie1


    def accept(self, visitor):
        visitor.visit_while(self)

class Expr(BaseBox):
    def accept(self, visitor):
        method_name = 'visit_{}'.format(self.__class__.__name__.lower())
        visit = getattr(visitor, method_name)
        return visit(self)

class Id(Expr):
    def __init__(self, value):
        self.value = value

class Number(Expr):
    def __init__(self, value):
        self.value = value
```

```
class BinaryOp(Expr):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOp):
  pass


class Sub(BinaryOp):
  pass


class Mul(BinaryOp):
  pass


class Div(BinaryOp):
  pass

class Funcion(BaseBox):
    def __init__(self, type, id, param, expr):
        self.type = type
        self.id = id
        self.param = param
        self.expr = expr

    def accept(self, visitor):
        visitor.visit_funcion(self)

class CallFuncion(BaseBox):
    def __init__(self, id, expr):
        self.id = id
        self.expr = expr

    def accept(self, visitor):
        return visitor.visit_callfuncion(self)
```

[ ]:

```
from rply import ParserGenerator

pg = ParserGenerator(
# A list of all token names, accepted by the lexer.
[
    'OPEN_SOLUCION', 'CLOSE_SOLUCION',
    'NUMBER', 'PLUS', 'MINUS', 'MUL', 'DIV',
    'OPEN_PARENS', 'CLOSE_PARENS', 'OPEN_BRACKET', 'CLOSE_BRACKET',
    'VAR', 'INT', 'FLOAT', 'IF', 'ELSE', 'WHILE',
    'ID', 'COMP','EQUALS', 'SEMICOLOM', 'FUNCION', 'RETURN'
],
# A list of precedence rules with ascending precedence, to
# disambiguate ambiguous production rules.
precedence=[
    ('left', ['PLUS', 'MINUS']),
    ('left', ['MUL', 'DIV'])
]
)

@pg.production('solucion : OPEN_SOLUCION vardecls statements CLOSE_SOLUCION')
def prog(p):
    return Solucion(p[1],p[2])

##################################################
# DECLARAÇÕES DE VARIÁVEIS
##################################################

@pg.production('vardecls : vardecl')
def vardecls(p):
    return VarDecls(p[0],None)

@pg.production('vardecls : vardecl vardecls')
def vardecls(p):
    return VarDecls(p[0],p[1])

@pg.production('vardecl : VAR INT ID SEMICOLOM')
def vardecl_int(p):
    return VarDecl(p[2].getstr(), "int")

@pg.production('vardecl : VAR FLOAT ID SEMICOLOM')
def vardecl_float(p):
    return VarDecl(p[2].getstr(), "float")

@pg.production('vardecl : FUNCION INT ID OPEN_PARENS ID CLOSE_PARENS␣
 ↪OPEN_BRACKET RETURN expression SEMICOLOM CLOSE_BRACKET')
def vardecl_funcion(p):
    return Funcion("int", p[2].getstr(),p[4].getstr(),p[8])

@pg.production('vardecl : FUNCION FLOAT ID OPEN_PARENS ID CLOSE_PARENS␣
 ↪OPEN_BRACKET RETURN expression SEMICOLOM CLOSE_BRACKET')
def vardecl_funcion(p):
    return Funcion("float", p[2].getstr(),p[4].getstr(),p[8])

##################################################
```

```python
@pg.production('statements : statement')
def statement_statements(p):
    return Statements(p[0],None)

@pg.production('statements : statement statements')
def statement_statements(p):
    return Statements(p[0],p[1])

@pg.production('statement : ID EQUALS expression SEMICOLOM')
def statement_atrib(p):
    return Atrib(p[0].getstr(),p[2])

@pg.production('statement : IF OPEN_PARENS expression COMP expression CLOSE_PARENS OPEN_BRACKET statements CLOSE_BRACKET')
def expression_ifelse1(p):
    return IfElse (p[2],p[3],p[4],p[7],None)


@pg.production('statement : IF OPEN_PARENS expression COMP expression CLOSE_PARENS OPEN_BRACKET statements CLOSE_BRACKET ELSE OPEN_BRACKET statements CLOSE_BRACKET')
def expression_ifelse2(p):
    return IfElse (p[2],p[3],p[4],p[7],p[11])

@pg.production('statement : WHILE OPEN_PARENS expression COMP expression CLOSE_PARENS OPEN_BRACKET statements CLOSE_BRACKET')
def statement_while(p):
    return While (p[2],p[3],p[4],p[7])

@pg.production('expression : ID')
def expression_id(p):
    return Id(p[0].getstr())

@pg.production('expression : NUMBER')
def expression_number(p):
    value = p[0].getstr()

    if "." in value:
        value = float(value)
    else:
        value = int(value)

    return Number(value)

@pg.production('expression : ID OPEN_PARENS expression CLOSE_PARENS')
def expression_callfuncion(p):
    return CallFuncion(p[0].getstr(),p[2])

@pg.production('expression : OPEN_PARENS expression CLOSE_PARENS')
def expression_parens(p):
    return p[1]

@pg.production('expression : expression PLUS expression')
@pg.production('expression : expression MINUS expression')
@pg.production('expression : expression MUL expression')
@pg.production('expression : expression DIV expression')
def expression_binop(p):
    left = p[0]
    right = p[2]
    if p[1].gettokentype() == 'PLUS':
        return Add(left, right)
    elif p[1].gettokentype() == 'MINUS':
        return Sub(left, right)
    elif p[1].gettokentype() == 'MUL':
        return Mul(left, right)
    elif p[1].gettokentype() == 'DIV':
        return Div(left, right)
    else:
        raise AssertionError('Oops, this should not be possible!')

parser = pg.build()
```

[ ]:

```
class SymbolTable(Visitor):
    ST = {}
    func = {}
    variables = {}

    def __init__(self):
        SymbolTable.ST.clear()
        SymbolTable.func.clear()
        SymbolTable.variables.clear()

    def visit_solucion(self, solucion):
        solucion.vardecls.accept(self)

    def visit_vardecls(self, d):
        d.vardecl.accept(self)
        if d.vardecls!=None:
          d.vardecls.accept(self)

    def visit_vardecl(self, d):
        SymbolTable.ST[d.id]=d.tp
        SymbolTable.variables[d.id]=0
    def visit_funcion(self, d):
        SymbolTable.ST[d.id]='funcion'
        SymbolTable.func[d.id]=d
        SymbolTable.ST[d.param]=0
        SymbolTable.variables[d.param]=0
```

[ ]:

```
class Decorator(Visitor):

    def __init__(self, ST):
        self.ST = ST

    def visit_solucion(self, i):
        i.stmts.accept(self)

    def visit_statements(self, d):
        d.stmt.accept(self)
        if d.stmts!=None:
          d.stmts.accept(self)

    def visit_statement(self, d):
        d.stmt.accept(self)

    def visit_atrib(self, i):
        if i.id in self.ST:
          i.decor_type=self.ST[i.id]
        else:
          raise AssertionError('id not declared')
        i.expr.accept(self)

    def visit_ifelse(self, i):
        i.expr1.accept(self)
        i.expr2.accept(self)
        i.ie1.accept(self)
        if i.ie2!=None:
          i.ie2.accept(self)

    def visit_while(self, i):
        i.expr1.accept(self)
        i.expr2.accept(self)
        i.ie1.accept(self)


    def visit_id(self, i):

        if i.value in self.ST:
          i.decor_type=self.ST[i.value]
        else:
          raise AssertionError('id not declared')


    def visit_number(self, i):
        if i.value.__class__.__name__ == 'int':
          i.decor_type='int'
        else:
          i.decor_type='float'


    def visit_add(self, a):
        a.left.accept(self)
        a.right.accept(self)
        if a.left.decor_type=="float" or a.right.decor_type=="float":
          a.decor_type="float"
        else:
          a.decor_type="int"


    def visit_sub(self, a):
        a.left.accept(self)
        a.right.accept(self)
        if a.left.decor_type=="float" or a.right.decor_type=="float":
          a.decor_type="float"
        else:
          a.decor_type="int"

    def visit_mul(self, a):
        a.left.accept(self)
        a.right.accept(self)
        if a.left.decor_type =="float" or a.right.decor_type=="float":
          a.decor_type="float"
        else:
          a.decor_type="int"

    def visit_div(self, a):
        a.left.accept(self)
        a.right.accept(self)
        if a.left.decor_type=="float" or a.right.decor_type=="float":
          a.decor_type="float"
        else:
          a.decor_type="int"

    def visit_funcion(self, i):
       pass
    def visit_callfuncion(self, i):
        i.decor_type = SymbolTable.func[i.id].type
```

[ ]:

```
class TypeVerifier(Visitor):

    def visit_solucion(self, i):
        if i.stmts:
            i.stmts.accept(self)

    def visit_statements(self, d):
        d.stmt.accept(self)
        if d.stmts!=None:
          d.stmts.accept(self)

    def visit_statement(self, d):
        d.stmt.accept(self)

    def visit_atrib(self, i):
        pass

    def visit_ifelse(self, i):
        if i.expr1.decor_type != i.expr2.decor_type:
          raise AssertionError('incompatible types')

    def visit_while(self, i):
        if i.expr1.decor_type != i.expr2.decor_type:
          raise AssertionError('incompatible types')

    def visit_funcion(self, i):
        pass

    def visit_callfuncion(self, i):
        pass
```

[ ]:

```
class Generator(Visitor):

    def visit_solucion(self, solucion):
        solucion.vardecls.accept(self)
        solucion.stmts.accept(self)

    def visit_vardecls(self, decls):
        decls.vardecl.accept(self)
        if decls.vardecls:
            decls.vardecls.accept(self)
    def visit_vardecl(self, decl):
        pass

    def visit_statements(self, stmts):
        stmts.stmt.accept(self)
        if stmts.stmts:
            stmts.stmts.accept(self)

    def visit_statement(self, stmt):
        stmt.stmt.accept(self)

    def visit_atrib(self, atrib):
        SymbolTable.variables[atrib.id] = atrib.expr.accept(self)

    def visit_ifelse(self, d):
        if(eval(f"{d.expr1.accept(self)} {d.comp} {d.expr2.accept(self)}")):
            d.ie1.accept(self)
        elif d.ie2!=None:
            d.ie2.accept(self)

    def visit_while(self, d):
        while(eval(f"{d.expr1.accept(self)} {d.comp} {d.expr2.accept(self)}")):
            d.ie1.accept(self)

    def visit_number(self, number):
        return number.value

    def visit_id(self, id):
        return SymbolTable.variables[id.value]

    def visit_add(self, add):
        return add.left.accept(self) + add.right.accept(self)

    def visit_sub(self, sub):
        return sub.left.accept(self) - sub.right.accept(self)

    def visit_mul(self, mul):
        return mul.left.accept(self) * mul.right.accept(self)

    def visit_div(self, div):
        return div.left.accept(self) / div.right.accept(self)

    def visit_funcion(self, i):
        pass

    def visit_callfuncion(self, c):
        function = SymbolTable.func.get(c.id)
        if function:
            SymbolTable.variables[function.param] = c.expr.accept(self)
            value = function.expr.accept(self)
            del SymbolTable.variables[function.param]
            return value
        raise AssertionError(f'{c.id} not declared')
```

0.0.1 Exemplos

[ ]:

```
with open('ex1.txt', 'r') as f:
    program = f.read()

symbol = SymbolTable()
arvore = parser.parse(lexer.lex(program))
arvore.accept(SymbolTable())
arvore.accept(Decorator(symbol.ST))
arvore.accept(TypeVerifier())
arvore.accept(Generator())
print(SymbolTable.variables)
```

```
{'a': 2.5, 'b': 2}
```

[ ]:

```
with open('ex2.txt', 'r') as f:
    program = f.read()

symbol = SymbolTable()


arvore = parser.parse(lexer.lex(program))
arvore.accept(symbol)
arvore.accept(Decorator(symbol.ST))
arvore.accept(TypeVerifier())
arvore.accept(Generator())
print(SymbolTable.variables)
```

```
{'a': 2}
```