**TP compilateurs 2 :**

**Perfectionnement de PLY**

*Fichier : lex4.py*

import ply.lex as lex

tokens = (

'NUMBER',

'ADD\_OP',

'MUL\_OP',

**'IDENTIFIER'**,

)

literals = '()**;=**'

def t\_ADD\_OP(t):

r'[+-]'

return t

def t\_MUL\_OP(t):

r'[\*/]'

return t

def t\_NUMBER(t):

r'\d+(\.\d+)?'

try:

t.value = float(t.value)

except ValueError:

print ("Line %d: Problem while parsing %s!" % (t.lineno,t.value))

t.value = 0

return t

**def t\_IDENTIFIER(t):**

**r'[A-Za-z\_]\w\*'**

**return t**

def t\_newline(t):

r'\n+'

t.lexer.lineno += len(t.value)

t\_ignore = ' \t'

def t\_error(t):

print ("Illegal character '%s'" % repr(t.value[0]))

t.lexer.skip(1)

lex.lex()

if \_\_name\_\_ == "\_\_main\_\_":

import sys

prog = open(sys.argv[1]).read()

lex.input(prog)

while 1:

tok = lex.token()

if not tok: break

print ("line %d: %s(%s)" % (tok.lineno, tok.type, tok.value))

*Fichier : parser3.py*

import ply.yacc as yacc

from lex4 import tokens

operations = {

'+' : lambda x,y: x+y,

'-' : lambda x,y: x-y,

'\*' : lambda x,y: x\*y,

'/' : lambda x,y: x/y,

}

vars = {}

def p\_programme\_expr(p):

''' programme : statement

| statement ';' programme '''

try:

p[0]=p[3]

except:

p[0]=p[1]

def p\_statement(p):

''' statement : assignation

| expression '''

p[0] = p[1]

def p\_expression\_op(p):

'''expression : expression ADD\_OP expression

| expression MUL\_OP expression'''

p[0] = operations[p[2]](p[1],p[3])

def p\_expression\_num(p):

'expression : NUMBER'

p[0] = p[1]

def p\_expression\_paren(p):

'''expression : '(' expression ')' '''

p[0] = p[2]

def p\_minus(p):

''' expression : ADD\_OP expression %prec UMINUS'''

p[0] = operations[p[1]](0,p[2])

def p\_assign(p):

''' assignation : IDENTIFIER '=' expression '''

vars[p[1]] = p[3]

p[0] = p[3]

def p\_expression\_var(p):

''' expression : IDENTIFIER '''

p[0] = vars[p[1]]

def p\_error(p):

print ("Syntax error in line %d" % p.lineno)

yacc.errok()

precedence = (

('left', 'ADD\_OP'),

('left', 'MUL\_OP'),

('right', 'UMINUS'),

)

yacc.yacc(outputdir='generated')

if \_\_name\_\_ == "\_\_main\_\_":

import sys

prog = open(sys.argv[1]).read()

result = yacc.parse(prog)

print (result)

Grammaire

**program → statement**

**program → statement; program**

**statement → assignation**

**statement → expression**

**assignation → IDENTIFIER = expression**

**expression → IDENTIFIER**

expression **→** expression addOP expression

expression **→** expression mulOP expression

expression **→** nombre

expression **→** ( expression )

expression **→** addOP expression

**Explications**

**program → statement**

**program → statement ; program**

def p\_programme\_expr(p):

**''' programme** (p[0]) **: statement**(p[1])

**| statement**(p[1]) **';'**(p[2]) **programme**(p[3])**'''**

try:

p[0]=p[3] Retourne la valeur de programme (p[3]) s’il existe ( prog ->state ;**prog)**

except:

p[0]=p[1]...sinon retourne la valeur de statement (p[1]) (prog -> **statement**)

**statement → assignation**

**statement → expression**

def p\_statement(p):

''' **statement : assignation**

**| expression '''**

p[0] = p[1]

**assignation → IDENTIFIER = expression (lvalue)**

Initialisation du dictionnaire

**vars** = {}

def p\_assign(p):

''' **assignation : IDENTIFIER '=' expression** '''

# p[0] p[1] p[2] p[3]

**vars[**p[1]**]** = p[3]

p[0] = p[3]

**expression → IDENTIFIER**

def p\_expression\_var(p):

''' expression : IDENTIFIER '''

p[0] = **vars**[p[1]]

**Explications**

toto = 12\*3+4;

a=toto+1; a\*2

**>python parser3.py prog2.txt**

Generating LALR tables

82.0

toto=12\*-3+4;

a=toto+1 ;

a\*2.0

|  |  |
| --- | --- |
| Program  **| =**  **| |** ’toto’  **| |** + (2)  **| | |** \* (2)  **| | | |** 12.0  **| | | |** - (1)  **| | | | |** 3.0  **| | |** 4.0  **| =**  **| |** ’a’  **| |** + (2)  **| | |** ’toto’  **| | |** 1.0  **| \* (2)**  **| |** ’a’  **| |** 2.0 |  |

**AST (ABSTRCT SYNTAXIC TREE)**

<http://www.graphviz.org/Download_windows.php> -> [graphviz-2.38.msi](http://www.graphviz.org/pub/graphviz/stable/windows/graphviz-2.38.msi)

<https://pypi.python.org/pypi/pydot2/1.0.33>

>python setup.py install

**AST (ABSTRCT SYNTAXIC TREE)**

import ply.yacc as yacc

from lex4 import tokens

import AST

operations = {

'+' : lambda x,y: x+y,

'-' : lambda x,y: x-y,

'\*' : lambda x,y: x\*y,

'/' : lambda x,y: x/y,

}

vars = {}

**def p\_programme\_statement(p):**

''' programme : statement '''

**p[0] = AST.ProgramNode(p[1])**

**def p\_programme\_recursive(p):**

''' programme : statement ';' programme '''

**p[0] = AST.ProgramNode([p[1]]+p[3].children)**

**def p\_statement(p):**

''' statement : assignation

| expression '''

**p[0] = p[1]**

**def p\_expression\_op(p):**

'''expression : expression ADD\_OP expression

| expression MUL\_OP expression'''

**p[0] = AST.OpNode(p[2], [p[1], p[3]])**

**def p\_expression\_num\_or\_var(p):**

'''expression : NUMBER

| IDENTIFIER '''

**p[0] = AST.TokenNode(p[1])**

**def p\_expression\_paren(p):**

'''expression : '(' expression ')' '''

**p[0] = p[2]**

**def p\_minus(p):**

''' expression : ADD\_OP expression %prec UMINUS'''

**p[0] = AST.OpNode(p[1], [p[2]])**

**def p\_assign(p):**

''' assignation : IDENTIFIER '=' expression '''

**p[0] = AST.AssignNode([AST.TokenNode(p[1]),p[3]])**

**Attention : Ici, p[1] renvoie un identificateur sous la forme d’une chaine de caractère, il faut donc créer un objet de type TokenNode comme fils de AssignNode**

def p\_error(p):

print ("Syntax error in line %d" % p.lineno)

yacc.errok()

precedence = (

('left', 'ADD\_OP'),

('left', 'MUL\_OP'),

('right', 'UMINUS'),

)

yacc.yacc(outputdir='generated')

**if \_\_name\_\_ == "\_\_main\_\_":**

import sys

prog = open(sys.argv[1]).read()

**ast** = yacc.parse(prog)

print (**ast**)

import os

**graph** = **ast.**makegraphicaltree()

name = os.path.splitext(sys.argv[1])[0]+'-ast.pdf'

**graph**.write\_pdf(name)

print ("wrote ast to", name)