Εργασία στην Σχεδίαση Γλωσσών Προγραμματισμού

Σωκράτης Αθανασιάδης

ΑΕΜ: 2547

Lexer and Parser σε python με χρήση της βιβλιοθήκης «ply»

Η ply είναι μια βιβλιοθήκη σε python η οποία περιέχει ένα Implementation lex και yacc . Χρησιμοποιήθηκε η python επειδή είναι γλώσσα πολύ υψηλού επιπέδου και είναι πολύ εύκολο το να αναπτύξεις έναν primitive compiler.

Ο lex που υπάρχει κάνει tokenize το string το οποίο δίνουμε σαν Input.

Για παράδειγμα αν δώσουμε ως είσοδο 1 + 2, ο lex θα το αναλύσει σε: INTEGER(1) PLUS INTEGER(2). Μετά από αυτό το στάδιο φτιάχνουμε το δέντρο της συντακτικής ανάλυσης και το τρέχουμε μέσω της συνάρτησης που υλοποιήσαμε.

Τα Tokens που αναγνωρίζονται από το πρόγραμμα είναι τα εξής:

```
tokens = [
    'INT',
    'FLOAT',
    'NAME',
    'PLUS',
    'MINUS',
    'DIVIDE',
    'MUL',
    'EQUALS'
    ]
```

Αυτή την λίστα με tokens θα χρησιμοποιεί η γραμματική μας για να γίνει ο έλεγχος αν οι κανόνες της γραμματικής είναι σωστές η όχι. Θα πρέπει λοιπόν να πούμε στον lex πως μοιάζει το σύμβολο MINUS πως είναι το PLUS σύμβολο κοκ. Αυτό το κάνουμε με τον εξής τρόπο:

```
t_PLUS = r'\+'
t_MINUS = r'\-'
t_MUL = r'\*'
t_DIVIDE = r'\/'
t_EQUALS = r'\='
```

Όμως με αυτόν τον τρόπο περιγράφουμε μόνο τα πολύ απλά tokens. Για τους INT FLOAT κλπ χρειαζόμαστε πιο σύνθετο τρόπο για να περιγράψουμε τα tokens. Για τους INT για παράδειγμα:

```
def t_INT(t):
    r'\d+' #any chars whose
           #1, 1201930102
    t.value = int(t.value)
    return t
```

Του ορίζουμε ως ένα regular expression το οποίο περιγράφεται ως d+ που είναι μια σειρά οποιονδήποτε χαρακτήρων που είναι περισσότεροι από ένας. Για παράδειγμα 1, η 129319293. Και στη συνέχεια μετατρέπουμε το value από string σε integer και το επιστρέφουμε. Με παρόμοιο τρόπο φτιάχνουμε τα FLOAT .

```
def t_FLOAT(t):
    r'\d+\.\d+' #means it is any
                #1.3123 1312.3415
    t.value = float(t.value)
    return t
```

Τα float είναι ένα regular expression το οποίο αποτελείτε από έναν αριθμό από integers d+ και έναν ακόμα αριθμό από integers οι οποίοι χωρίζονται από μία τελεία ‘.’ . Ενώ για τα NAMES κάνουμε το εξής

```
def t_NAME(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*' #firs
    t.type = 'NAME'
    return t
```

Τα ορίζουμε ως ένα regular expression το οποίο ξεκινάει από έναν κεφαλαίο η έναν πεζό αγγλικό χαρακτήρα, η μια κάτω παύλα ως πρώτο χαρακτήρα, και ακολουθείτε από έναν οποιοδήποτε αριθμό χαρακτήρων κάτω παύλες η νούμερα. Ο λόγος που το ορίσαμε έτσι είναι επειδή ένα NAME δεν μπορεί να ξεκινάει με αριθμό 0 εως 9, και πρέπει να βεβαιωθούμε ότι ο πρώτος χαρακτήρας δεν είναι αριθμός.

Στη συνέχεια φτιάχνουμε τον lexer μας με την ενοτλή lexer = lex.lex()

Στο σημείο αυτό ο LEXER μας είναι σε θέση να αναγνωρίσει τα tokens που έχουμε δώσει ως είσοδο. Για παράδειγμα για την είσοδο 1+2 ο lexer έχει το αποτέλεσμα

```
(base) C:\Users\Dani\Desktop\SGP>python calc.py
LexToken(INT,1,1,0)
LexToken(PLUS,'+',1,1)
LexToken(INT,2,1,2)
>>
```

Ενώ για είσοδο a=213.44 έχουμε το αποτέλεσμα

```
(base) C:\Users\Dani\Desktop\SGP>python calc.py
LexToken(NAME,'a',1,0)
LexToken(EQUALS,'=',1,1)
LexToken(FLOAT,213.44,1,2)
>>
```

Parsing

Σκοπός του parser είναι να κατασκευάσει το parsing tree, δηλαδή το δέντρο της συντακτικής ανάλυσης. Αρχικά κατασκευάζουμε τους κανόνες της γραμματικής ως εξής:

```
def p_calc(p):
    '''
    calc : expression
         | var_assign
         | empty
    '''
    print((p[1]))
```

Το p είναι μια δομή tuple της python επομένως το p(0) θα είναι το calc, ενώ το p(1) θα είναι το expression κοκ. Και ο κανόνας σημαίνει ότι το calc μπορεί να είναι ένα expression μια var_assign η να είναι κενό. Και κάνουμε print το p(1) που θα είναι ότι είναι το expression. Με παρόμοια λογική κατασκευάσουμε και τους υπόλοιπους κανόνες τις γραμματικής.

Για το empty:

```
def p_empty(p):
    '''
    empty :
    '''
    p[0] = None
```

Για το expression:

```
def p_expression(p):
    '''
    expression : expression MUL expression
               | expression DIVIDE expression
               | expression PLUS expression
               | expression MINUS expression
    '''
    p[0] = (p[2], p[1], p[3])
```

Ο λόγος που το κάναμε αυτό είναι για να δέχεται και strings του τύπου 1+2+3 και πρέπει να έχουμε αυτή την αναδρομικότητα που ορίζουν οι κανόνες. Η σειρά που δίνουμε στο tuple θα εξηγηθεί αργότερα όταν κατασκευάζουμε το δέντρο της συντακτικής ανάλυσης.

Ενώ για τα int και floats έχουμε τους κανόνες:

```
def p_expression_int_float(p):
    '''
    expression : INT
               | FLOAT
    '''
    p[0] = p[1]
```

Ενώ έχουμε ορίσει και την προτεραιότητα των τελεστών που χρησιμοποιήθηκαν ως εξής:

```
precedence = (
    #left associative tokens
    #devide and multiply have higher pre

    ('left', 'PLUS', 'MINUS'),
    ('left', 'MUL', 'DIVIDE')

)
```

Σύμφωνα με αυτούς τους κανόνες όλα τα tokens είναι left associative ενώ το Multiply και το Divide έχουν μεγαλύτερη προτεραιότητα από το PLUS και το MINUS. Με τον τρόπο αυτό αφαιρούμε την ασάφεια που θα υπήρχε στην γραμματική μας η οποία θα δημιουργούσε πρόβλημα.

Κατασκευάζουμε τον Parser μας με την εντολή parser = yacc.yacc() . Στο σημείο αυτό μπορούμε να κατασκευάσουμε το δέντρο συντακτικής ανάλυσης. Το οποίο θα έχει την μορφή που θα ορίζεται μέσα από τα tuples της python.

Πιο συγκεκριμένα το tuple : ('+', 1, ('*', 2, 10)) αντιστοιχεί στην είσοδο: 1+2*10 και το δέντρο έχει ρίζα το + αριστερό παιδί το 1 και δεξί παιδί το *. Ενώ το * έχει αριστερό παιδί το 2 και δεξί παιδί το 10. Φυσικά μπορούμε να βάλουμε και πιο σύνθετες εκφράσεις όπως την : 4-5*6+14/2 και θα λάβουμε το συντακτικό δέντρο: ('+', ('-', 4, ('*', 5, 6)), ('/', 14, 2)) που η ερμηνεία του έχει ως εξής. Το + είναι στην ρίζα, με αριστερό παιδί το – το οποίο με τη σειρά του έχει αριστερό παιδί το 4 και δεξί παιδί το *. Το * με τη σειρά του έχει αριστερό παιδί το 5 και δεξί το 6. Ενώ η ρίζα μας (το +) έχει αριστερό παιδί το σύμβολο / το οποίο έχει παιδιά τα 14 και 2. Αυτό το συντακτικό δέντρο, μπορεί να χρησιμοποιηθεί για να κάνουμε evaluate την τιμή της εισόδου.

Τα παραδείγματα που τρέξαμε σε command line είναι τα εξής:

```
(base) C:\Users\Dani\Desktop\SGP>python calc.py
>> 1+2*10
('+', 1, ('*', 2, 10))
>> 4-5*6+14/2
('+', ('-', 4, ('*', 5, 6)), ('/', 14, 2))
>>
```

Οι βιβλιοθήκες που χρησιμοποιήθηκαν

Η βιβλιοθήκη ply είναι ένα πολύ ισχυρό εργαλείο. Πιο συγκεκριμένα με τον κώδικα της python ορίσαμε την γραμματική που θα χρησιμοποιεί ο compiler μας. Αν ρίξουμε μια ματιά στα αρχεία που έχουν δημιουργηθεί από την βιβλιοθήκη που χρησιμοποιήθηκε θα δούμε το εξής:

Από το αρχείο (parser.out)

Η γραμματική μας είναι η εξής:

```
Created by PLY version 3.11 (http://www.dabeaz.com/ply)

Grammar

Rule 0     S' -> calc
Rule 1     calc -> expression
Rule 2     calc -> var_assign
Rule 3     calc -> empty
Rule 4     var_assign -> NAME EQUALS expression
Rule 5     expression -> expression MUL expression
Rule 6     expression -> expression DIVIDE expression
Rule 7     expression -> expression PLUS expression
Rule 8     expression -> expression MINUS expression
Rule 9     expression -> INT
Rule 10    expression -> FLOAT
Rule 11    expression -> NAME
Rule 12    empty -> <empty>
```

Ενώ ως parsing method χρησιμοποιήθηκε η LALR ανάλυση και έχουμε κατασκευάσει τα states ως εξής:

```
state 1

    (0) S' -> calc .
```

```
state 2

    (1) calc -> expression .
    (5) expression -> expression . MUL expression
    (6) expression -> expression . DIVIDE expression
    (7) expression -> expression . PLUS expression
    (8) expression -> expression . MINUS expression
```

```
state 3

    (2) calc -> var_assign .
```

```
state 4

    (3) calc -> empty .
```

```
state 5

    (9) expression -> INT .
```

```
state 6

    (10) expression -> FLOAT .
```

```
state 7

    (11) expression -> NAME .
    (4) var_assign -> NAME . EQUALS expression
```

```
state 8

    (5) expression -> expression MUL . expression
    (5) expression -> . expression MUL expression
    (6) expression -> . expression DIVIDE expression
    (7) expression -> . expression PLUS expression
    (8) expression -> . expression MINUS expression
    (9) expression -> . INT
    (10) expression -> . FLOAT
    (11) expression -> . NAME

state 9

    (6) expression -> expression DIVIDE . expression
    (5) expression -> . expression MUL expression
    (6) expression -> . expression DIVIDE expression
    (7) expression -> . expression PLUS expression
    (8) expression -> . expression MINUS expression
    (9) expression -> . INT
    (10) expression -> . FLOAT
    (11) expression -> . NAME

state 10

    (7) expression -> expression PLUS . expression
    (5) expression -> . expression MUL expression
    (6) expression -> . expression DIVIDE expression
    (7) expression -> . expression PLUS expression
    (8) expression -> . expression MINUS expression
    (9) expression -> . INT
    (10) expression -> . FLOAT
    (11) expression -> . NAME

state 11

    (8) expression -> expression MINUS . expression
    (5) expression -> . expression MUL expression
    (6) expression -> . expression DIVIDE expression
    (7) expression -> . expression PLUS expression
    (8) expression -> . expression MINUS expression
    (9) expression -> . INT
    (10) expression -> . FLOAT
    (11) expression -> . NAME
```

```
state 12

    (4) var_assign -> NAME EQUALS . expression
    (5) expression -> . expression MUL expression
    (6) expression -> . expression DIVIDE expression
    (7) expression -> . expression PLUS expression
    (8) expression -> . expression MINUS expression
    (9) expression -> . INT
    (10) expression -> . FLOAT
    (11) expression -> . NAME

state 13

    (5) expression -> expression MUL expression .
    (5) expression -> expression . MUL expression
    (6) expression -> expression . DIVIDE expression
    (7) expression -> expression . PLUS expression
    (8) expression -> expression . MINUS expression

state 14

    (11) expression -> NAME .

state 15

    (6) expression -> expression DIVIDE expression .
    (5) expression -> expression . MUL expression
    (6) expression -> expression . DIVIDE expression
    (7) expression -> expression . PLUS expression
    (8) expression -> expression . MINUS expression


state 16

    (7) expression -> expression PLUS expression .
    (5) expression -> expression . MUL expression
    (6) expression -> expression . DIVIDE expression
    (7) expression -> expression . PLUS expression
    (8) expression -> expression . MINUS expression
```

```
state 17

    (8) expression -> expression MINUS expression .
    (5) expression -> expression . MUL expression
    (6) expression -> expression . DIVIDE expression
    (7) expression -> expression . PLUS expression
    (8) expression -> expression . MINUS expression


state 18

    (4) var_assign -> NAME EQUALS expression .
    (5) expression -> expression . MUL expression
    (6) expression -> expression . DIVIDE expression
    (7) expression -> expression . PLUS expression
    (8) expression -> expression . MINUS expression
```

Φυσικά ο LALR parser που κατασκευάστηκε στο αρχείο parser.out κάτω από κάθε state έχει και τις κινήσεις shift η reduce που κάνουμε σε κάθε περίπτωση.

ΤΕΛΟΣ ΕΡΓΑΣΙΑΣ