



Πανεπιστήμιο Ιωαννίνων
Τμήμα Μηχανικών Η/Υ & Πληροφορικής
Ακαδημαϊκό Έτος 2022-2023

ΜΥΥ802 Μεταφραστές

Αναφορά

Υλοποίηση μεταφραστή

ΟΜΑΔΑ

Αρχοντής Νέστωρας - 4747

Σπυρίδων Χαλιδιάς - 4830

Περιεχόμενα

Τρόπος Σκέψεις ----- 3 -

- ❖ Εισαγωγή
- ❖ Κώδικας Παραδείγματος
- ❖ Τρόπος Εκτέλεσης

Λεκτικός Αναλυτής ----- 6 -

- ❖ Εισαγωγή
- ❖ Υλοποίηση
- ❖ Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος
- ❖ Συμπέρασμα-Παρατηρήσεις

Συντακτικός Αναλυτής ----- 9 -

- ❖ Εισαγωγή
- ❖ Υλοποίηση
- ❖ Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος
- ❖ Error handler

Ενδιάμεσος Κώδικας ----- 13 -

- ❖ Εισαγωγή
- ❖ Υλοποίηση
- ❖ Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος
- ❖ Αποτέλεσμα Εκτέλεσης Άλλου Παραδείγματος

Πίνακας Συμβόλων ----- 16 -

- ❖ Εισαγωγή
- ❖ Υλοποίηση
- ❖ Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος
- ❖ Αποτέλεσμα Εκτέλεσης Άλλου Παραδείγματος

Τελικός Κώδικας ----- 21 -

- ❖ Εισαγωγή
- ❖ Υλοποίηση
- ❖ Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος
- ❖ Αποτέλεσμα Εκτέλεσης Με Python
- ❖ Συμπέρασμα-Παρατηρήσεις

Τρόπος Σκέψεις

Εισαγωγή

Σκοπός της συγκεκριμένης αναφοράς είναι να δείξουμε πως ο κώδικας που γράψαμε για το μεταφραστή λειτουργεί σωστά σύμφωνα με τις προδιαγραφές που δόθηκαν και όχι η αναλυτική εξήγηση του κώδικα και τρόπος λειτουργίας του. Ο τρόπος λοιπόν που επιλέξαμε να δομήσουμε την αναφορά είναι ο εξής. Θα δίνονται κάποια test για κάθε φάση του κώδικα τα θεωρητικά αποτελέσματα που περιμένουμε να βγουν καθώς και τα αποτελέσματα που παράγει ο κώδικας και εμφανίζει είτε στο τερματικό(Λεκτικός Αναλυτής, Συντακτικός Αναλυτής[σε περίπτωση λάθους], Ενδιάμεσος Κώδικας και Πίνακας Συμβόλων[σε κάθε end_block]) είτε σε ένα ξεχωριστό αρχείο(Τελικός Κώδικας). Έτσι από τη σύγκριση των δύο αυτών αποτελεσμάτων θέλουμε να δείξουμε την ορθότητα του κώδικα.

Κώδικας Παραδείγματος

Το κύριο test/παράδειγμα που θα χρησιμοποιήσουμε σε όλες τις επιμέρους ενότητες, είναι το παράδειγμα που μας δόθηκε στο τέλος του documentation της cutePy, και παρατίθεται παρακάτω.

```
1. def main_factorial():
2.     #{
3.         #$ declarations #$
4.         #declare x
5.         #declare i,fact
6.
7.         #$ body of main_factorial #$
8.         x = int(input());
9.         fact = 1;
10.        i = 1;
11.        while (i<=x):
12.            #{
13.                fact = fact * i;
14.                i = i + 1;
15.            #}
16.        print(fact);
17.    #}
18.
19. def main_fibonacci():
20.     #{
21.         #declare x
22.
23.         def fibonacci(x):
24.             #{
25.                 if (x<=1):
```

```

26.         return(x);
27.     else:
28.         return (fibonacci(x-1)+fibonacci(x-2));
29.     #}
30.
31.     x = int(input());
32.     print(fibonacci(x));
33. #}
34.
35. def main_countdigits():
36. # {
37.     #declare x, count
38.
39.     x = int(input());
40.     count = 0;
41.     while (x>0):
42.     # {
43.         x = x // 10;
44.         count = count + 1;
45.     # }
46.     print(count);
47. # }
48.
49. def main_primes():
50. # {
51.     #declare i
52.
53.     def isPrime(x):
54.     # {
55.         #declare i
56.
57.         def divides(x,y):
58.         # {
59.             if (y == (y//x) * x):
60.                 return (1);
61.             else:
62.                 return (0);
63.         # }
64.
65.         i = 2;
66.         while (i<x):
67.         # {
68.             if (divides(i,x)==1):
69.                 return (0);
70.             i = i + 1;

```

```

71.         #}
72.         return (1);
73.     #}
74.
75.     #$ body of main_primes #$
76.     i = 2;
77.     while (i<=30):
78.     #{
79.         if (isPrime(i)==1):
80.             print(i);
81.             i = i + 1;
82.     #}
83. #}
84.
85. if __name__ == "__main__":
86.     #$ call of main functions #$
87.     main_factorial();
88.     main_fibonacci();
89.     main_countdigits();
90.     main_primes();

```

Επίσης, σε ορισμένες ενότητες παρακάτω, θα χρησιμοποιήσουμε και άλλα παραδείγματα που έχουν παραδοθεί στις διάφορες φάσεις, αλλά ξαναπαρουσιάζονται εδώ πιο αναλυτικά.

Τρόπος Εκτέλεσης

Για να εκτελέσουμε το project μας και πρακτικά να κάνουμε την μεταγλώττιση σε ένα αρχείο τύπου .cpy που περιέχει κώδικα σε γλώσσα CutePy θα εκτελέσουμε την παρακάτω εντολή στο τερματικό μας.

```
python cutePy_4747_4830.py test1.cpy
```

Δηλαδή, θα καλέσουμε πρώτα το αρχείο .py που υλοποιήσαμε και έπειτα θα γράψουμε όποιο .cpy αρχείο θέλουμε να μεταγλωττίσουμε.

Λεκτικός Αναλυτής

Εισαγωγή

Σκοπός αυτής της αρχικής φάσης, είναι να παίρνουμε τον πηγαίο μας κώδικα, που είναι σε γλώσσα CutePy, να τον διαχωρίζουμε και να κατηγοριοποιούμε κάθε στοιχείο του με βάση το documentation που μας έχει δοθεί για αυτήν την συγκεκριμένη καινούργια γλώσσα που υλοποιούμε. Για τον έλεγχο της ορθότητας του Λεκτικού Αναλυτή, θα βλέπουμε κάθε Token που έχει εκτυπωθεί στο τερματικό και θα το κρίνουμε αν είναι σωστό με βάση το πεπερασμένο αυτόματο.

Υλοποίηση

Ο Λεκτικός Αναλυτής διαβάζει από ένα αρχείο τύπου .cpry και το παίρνει γραμμή γραμμή. Έπειτα, το κάθε σύμβολο περνάει από το πεπερασμένο αυτόματο που έχουμε φτιάξει για να δημιουργηθεί ένα token. Τα αποτελέσματα αυτά, θα χρησιμοποιηθεί αργότερα στους ελέγχους του Συντακτικού Αναλυτή.

Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος

Με το παράδειγμα που έχουμε επιλέξει να κάνουμε τους ελέγχους, καταφέρνουμε να τεστάρουμε σε έναν μεγάλο βαθμό το Λεκτικό Αναλυτή, παίρνοντας τις περισσότερες δυνατές περιπτώσεις που μπορούν να δημιουργηθούν.

Στη συνέχεια φαίνεται το αποτέλεσμα της εκτέλεσης που εμφανίστηκε στο τερματικό αφού τρέξαμε το πρόγραμμα.

```
...TOKENS...
def, Family: Keyword, Line: 1
main_factorial, Family: Id, Line: 1
(, Family: GroupSymbol, Line: 1
), Family: GroupSymbol, Line: 1
:, Family: Delimiter, Line: 1
{, Family: GroupSymbol, Line: 2
START OF COMMENTS
END OF COMMENTS
#declare, Family: Declare, Line: 4
x, Family: Id, Line: 4
#declare, Family: Declare, Line: 5
i, Family: Id, Line: 5
, Family: Delimiter, Line: 5
fact, Family: Id, Line: 5
START OF COMMENTS
END OF COMMENTS
x, Family: Id, Line: 8
=, Family: Assignment, Line: 8
int, Family: Keyword, Line: 8
(, Family: GroupSymbol, Line: 8
input, Family: Keyword, Line: 8
(, Family: GroupSymbol, Line: 8
), Family: GroupSymbol, Line: 8
), Family: GroupSymbol, Line: 8
;, Family: Delimiter, Line: 8
fact, Family: Id, Line: 9
=, Family: Assignment, Line: 9
1, Family: Number, Line: 9
;, Family: Delimiter, Line: 9
i, Family: Id, Line: 10
=, Family: Assignment, Line: 10
1, Family: Number, Line: 10
;, Family: Delimiter, Line: 10
while, Family: Keyword, Line: 11
(, Family: GroupSymbol, Line: 11
i, Family: Id, Line: 11
<=, Family: REL_OP, Line: 11
x, Family: Id, Line: 11
), Family: GroupSymbol, Line: 11
:, Family: Delimiter, Line: 11
#{, Family: GroupSymbol, Line: 12
fact, Family: Id, Line: 13
=, Family: Assignment, Line: 13
fact, Family: Id, Line: 13
*, Family: MUL_OP, Line: 13
i, Family: Id, Line: 13
;, Family: Delimiter, Line: 13
i, Family: Id, Line: 14
=, Family: Assignment, Line: 14
i, Family: Id, Line: 14
+, Family: ADD_OP, Line: 14
1, Family: Number, Line: 14
;, Family: Delimiter, Line: 14
#}, Family: GroupSymbol, Line: 15
print, Family: Keyword, Line: 16
(, Family: GroupSymbol, Line: 16
fact, Family: Id, Line: 16
), Family: GroupSymbol, Line: 16
;, Family: Delimiter, Line: 16
#}, Family: GroupSymbol, Line: 17
def, Family: Keyword, Line: 19
main_fibonacci, Family: Id, Line: 19
(, Family: GroupSymbol, Line: 19
), Family: GroupSymbol, Line: 19
:, Family: Delimiter, Line: 19
#{, Family: GroupSymbol, Line: 20
#declare, Family: Declare, Line: 21
x, Family: Id, Line: 21
def, Family: Keyword, Line: 23
fibonacci, Family: Id, Line: 23
(, Family: GroupSymbol, Line: 23
x, Family: Id, Line: 23
), Family: GroupSymbol, Line: 23
:, Family: Delimiter, Line: 23
#{, Family: GroupSymbol, Line: 24
if, Family: Keyword, Line: 25
(, Family: GroupSymbol, Line: 25
x, Family: Id, Line: 25
<=, Family: REL_OP, Line: 25
1, Family: Number, Line: 25
), Family: GroupSymbol, Line: 25
), Family: GroupSymbol, Line: 26
x, Family: Id, Line: 26
), Family: GroupSymbol, Line: 26
;, Family: Delimiter, Line: 26
else, Family: Keyword, Line: 27
:, Family: Delimiter, Line: 27
return, Family: Keyword, Line: 28
(, Family: GroupSymbol, Line: 28
fibonacci, Family: Id, Line: 28
(, Family: GroupSymbol, Line: 28
x, Family: Id, Line: 28
-, Family: ADD_OP, Line: 28
1, Family: Number, Line: 28
), Family: GroupSymbol, Line: 28
+, Family: ADD_OP, Line: 28
fibonacci, Family: Id, Line: 28
(, Family: GroupSymbol, Line: 28
x, Family: Id, Line: 28
-, Family: ADD_OP, Line: 28
2, Family: Number, Line: 28
), Family: GroupSymbol, Line: 28
), Family: GroupSymbol, Line: 28
;, Family: Delimiter, Line: 28
#}, Family: GroupSymbol, Line: 29
x, Family: Id, Line: 31
=, Family: Assignment, Line: 31
int, Family: Keyword, Line: 31
(, Family: GroupSymbol, Line: 31
input, Family: Keyword, Line: 31
(, Family: GroupSymbol, Line: 31
), Family: GroupSymbol, Line: 31
), Family: GroupSymbol, Line: 31
;, Family: Delimiter, Line: 31
```

```

print, Family: Keyword, Line: 32
(, Family: GroupSymbol, Line: 32
fibonacci, Family: Id, Line: 32
(, Family: GroupSymbol, Line: 32
x, Family: Id, Line: 32
), Family: GroupSymbol, Line: 32
), Family: GroupSymbol, Line: 32
;, Family: Delimiter, Line: 32
#{, Family: GroupSymbol, Line: 33
def, Family: Keyword, Line: 35
main_countdigits, Family: Id, Line: 35
(, Family: GroupSymbol, Line: 35
), Family: GroupSymbol, Line: 35
;, Family: Delimiter, Line: 35
#{, Family: GroupSymbol, Line: 36
#declare, Family: Declare, Line: 37
(, Family: GroupSymbol, Line: 37
), Family: Delimiter, Line: 37
count, Family: Id, Line: 37
x, Family: Id, Line: 39
=, Family: Assignment, Line: 39
int, Family: Keyword, Line: 39
(, Family: GroupSymbol, Line: 39
input, Family: Keyword, Line: 39
(, Family: GroupSymbol, Line: 39
), Family: GroupSymbol, Line: 39
), Family: GroupSymbol, Line: 39
;, Family: Delimiter, Line: 39
count, Family: Id, Line: 40
=, Family: Assignment, Line: 40
0, Family: Number, Line: 40
;, Family: Delimiter, Line: 40
while, Family: Keyword, Line: 41
(, Family: GroupSymbol, Line: 41
x, Family: Id, Line: 41
>, Family: REL_OP, Line: 41
0, Family: Number, Line: 41
), Family: GroupSymbol, Line: 41
;, Family: Delimiter, Line: 41
#{, Family: GroupSymbol, Line: 42
x, Family: Id, Line: 43
=, Family: Assignment, Line: 43
x, Family: Id, Line: 43
//, Family: MUL_OP, Line: 43
10, Family: Number, Line: 43
;, Family: Delimiter, Line: 43
count, Family: Id, Line: 44
=, Family: Assignment, Line: 44
count, Family: Id, Line: 44
+, Family: ADD_OP, Line: 44
1, Family: Number, Line: 44
;, Family: Delimiter, Line: 44
#{, Family: GroupSymbol, Line: 45
print, Family: Keyword, Line: 46
(, Family: GroupSymbol, Line: 46
count, Family: Id, Line: 46
), Family: GroupSymbol, Line: 46
;, Family: Delimiter, Line: 46
#{, Family: GroupSymbol, Line: 47
def, Family: Keyword, Line: 49
main_primes, Family: Id, Line: 49
(, Family: GroupSymbol, Line: 49
), Family: GroupSymbol, Line: 49
;, Family: Delimiter, Line: 49
#{, Family: GroupSymbol, Line: 50
#declare, Family: Declare, Line: 51
i, Family: Id, Line: 51
def, Family: Keyword, Line: 53
isPrime, Family: Id, Line: 53
(, Family: GroupSymbol, Line: 53
x, Family: Id, Line: 53
), Family: GroupSymbol, Line: 53
;, Family: Delimiter, Line: 53
#{, Family: GroupSymbol, Line: 54
#declare, Family: Declare, Line: 55
i, Family: Id, Line: 55
), Family: GroupSymbol, Line: 55
def, Family: Keyword, Line: 57
divides, Family: Id, Line: 57
(, Family: GroupSymbol, Line: 57
x, Family: Id, Line: 57
), Family: Delimiter, Line: 57
y, Family: Id, Line: 57
), Family: GroupSymbol, Line: 57
;, Family: Delimiter, Line: 57
#{, Family: GroupSymbol, Line: 58
if, Family: Keyword, Line: 59
(, Family: GroupSymbol, Line: 59
y, Family: Id, Line: 59
==, Family: REL_OP, Line: 59
(, Family: GroupSymbol, Line: 59
y, Family: Id, Line: 59
//, Family: MUL_OP, Line: 59
x, Family: Id, Line: 59
), Family: GroupSymbol, Line: 59
*, Family: MUL_OP, Line: 59
x, Family: Id, Line: 59
), Family: GroupSymbol, Line: 59
;, Family: Delimiter, Line: 59
return, Family: Keyword, Line: 60
(, Family: GroupSymbol, Line: 60
1, Family: Number, Line: 60
), Family: GroupSymbol, Line: 60
;, Family: Delimiter, Line: 60
else, Family: Keyword, Line: 61
;, Family: Delimiter, Line: 61
return, Family: Keyword, Line: 62
(, Family: GroupSymbol, Line: 62
0, Family: Number, Line: 62
), Family: GroupSymbol, Line: 62
;, Family: Delimiter, Line: 62
#{, Family: GroupSymbol, Line: 63
i, Family: Id, Line: 65
=, Family: Assignment, Line: 65
2, Family: Number, Line: 65
;, Family: Delimiter, Line: 65

```

```

while, Family: Keyword, Line: 66
(, Family: GroupSymbol, Line: 66
i, Family: Id, Line: 66
<, Family: REL_OP, Line: 66
x, Family: Id, Line: 66
), Family: GroupSymbol, Line: 66
;, Family: Delimiter, Line: 66
#{, Family: GroupSymbol, Line: 67
if, Family: Keyword, Line: 68
(, Family: GroupSymbol, Line: 68
divides, Family: Id, Line: 68
(, Family: GroupSymbol, Line: 68
i, Family: Id, Line: 68
), Family: Delimiter, Line: 68
x, Family: Id, Line: 68
), Family: GroupSymbol, Line: 68
==, Family: REL_OP, Line: 68
1, Family: Number, Line: 68
), Family: GroupSymbol, Line: 68
;, Family: Delimiter, Line: 68
return, Family: Keyword, Line: 69
(, Family: GroupSymbol, Line: 69
0, Family: Number, Line: 69
), Family: GroupSymbol, Line: 69
;, Family: Delimiter, Line: 69
i, Family: Id, Line: 70
=, Family: Assignment, Line: 70
i, Family: Id, Line: 70
+, Family: ADD_OP, Line: 70
1, Family: Number, Line: 70
;, Family: Delimiter, Line: 70
#{, Family: GroupSymbol, Line: 71
return, Family: Keyword, Line: 72
(, Family: GroupSymbol, Line: 72
1, Family: Number, Line: 72
), Family: GroupSymbol, Line: 72
;, Family: Delimiter, Line: 72
#{, Family: GroupSymbol, Line: 73

```

```

START OF COMMENTS
END OF COMMENTS
i, Family: Id, Line: 76
=, Family: Assignment, Line: 76
2, Family: Number, Line: 76
;, Family: Delimiter, Line: 76
while, Family: Keyword, Line: 77
(, Family: GroupSymbol, Line: 77
i, Family: Id, Line: 77
<=, Family: REL_OP, Line: 77
30, Family: Number, Line: 77
), Family: GroupSymbol, Line: 77
;, Family: Delimiter, Line: 77
#{, Family: GroupSymbol, Line: 78
if, Family: Keyword, Line: 79
(, Family: GroupSymbol, Line: 79
isPrime, Family: Id, Line: 79
(, Family: GroupSymbol, Line: 79
i, Family: Id, Line: 79
), Family: GroupSymbol, Line: 79
==, Family: REL_OP, Line: 79
1, Family: Number, Line: 79
), Family: GroupSymbol, Line: 79
;, Family: Delimiter, Line: 79
print, Family: Keyword, Line: 80
(, Family: GroupSymbol, Line: 80
i, Family: Id, Line: 80
), Family: GroupSymbol, Line: 80
;, Family: Delimiter, Line: 80
i, Family: Id, Line: 81
=, Family: Assignment, Line: 81
i, Family: Id, Line: 81
+, Family: ADD_OP, Line: 81
1, Family: Number, Line: 81
;, Family: Delimiter, Line: 81
#{, Family: GroupSymbol, Line: 82
#, Family: GroupSymbol, Line: 83
if, Family: Keyword, Line: 85
__name__, Family: Id, Line: 85
==, Family: REL_OP, Line: 85

```

```

"__main__", Family: QuotationMarksMain, Line: 85
;, Family: Delimiter, Line: 85
START OF COMMENTS
END OF COMMENTS
main_factorial, Family: Id, Line: 87
(, Family: GroupSymbol, Line: 87
), Family: GroupSymbol, Line: 87
;, Family: Delimiter, Line: 87
main_fibonacci, Family: Id, Line: 88
(, Family: GroupSymbol, Line: 88
), Family: GroupSymbol, Line: 88
;, Family: Delimiter, Line: 88
main_countdigits, Family: Id, Line: 89
(, Family: GroupSymbol, Line: 89
), Family: GroupSymbol, Line: 89
;, Family: Delimiter, Line: 89
main_primes, Family: Id, Line: 90
(, Family: GroupSymbol, Line: 90
), Family: GroupSymbol, Line: 90
;, Family: Delimiter, Line: 90
eof, Family: EndOfFile, Line: 90
Compilation successfully completed

```

Συμπέρασμα-Παρατηρήσεις

Στις παραπάνω φωτογραφίες φαίνονται τα αποτελέσματα αφού τρέξουμε τον λεκτικό αναλυτή. Βλέποντας κάθε γραμμή και συγκρίνοντας με τον κώδικα που τρέξαμε κάθε λέξη αναπαριστάται σωστά σαν token έχοντας τη σωστή οικογένεια και βρίσκεται στη γραμμή που θα περιμέναμε να βρίσκεται. Τρέχοντας έτσι ένα ολοκληρωμένο παράδειγμα με όλες τις περιπτώσεις καταλαβαίνουμε πως το πρόγραμμα τρέχει σωστά. Σε αυτό το σημείο να επισημάνουμε ότι έχουμε τρέξει και λάθος προγράμματα για να δούμε τα λάθη που εμφανίζονται. Στην αναφορά αποφασίσαμε να μην βάλουμε τέτοια παραδείγματα καθώς μία τέτοια προσθήκη κυρίως θα έκανε άσκοπα μεγάλη την αναφορά χωρίς να συμβάλλει στον έλεγχο της ορθότητας του κώδικα. Εξάλλου, ένα μέρος του `errorhandler` που θα δούμε στην επόμενη ενότητα, στον Συντακτικό Αναλυτή, είναι υπεύθυνο στην διαχείριση τέτοιου τύπου λαθών.

Συντακτικός Αναλυτής

Εισαγωγή

Ένας, και από τους σημαντικότερους ρόλους του Συντακτικού Αναλυτή, είναι ο έλεγχος της συντακτικής ορθότητας του πηγαίου κώδικα που θέλουμε να εκτελέσουμε. Επομένως, για να γίνει ο έλεγχος για την ορθή λειτουργία του, αρκεί να τρέξουμε ένα συντακτικά σωστό πρόγραμμα, για να δούμε να μας εκτυπώνεται στο τερματικό το αντίστοιχο σωστό μήνυμα, και ένα συντακτικά λάθος πρόγραμμα, για να δούμε να μας εκτυπώνεται στο τερματικό το αντίστοιχο λανθασμένο μήνυμα, που κάθε φορά ανάλογα με τον τύπο του λάθους θα εμφανίζει και διαφορετικό μήνυμα.

Υλοποίηση

Ο Συντακτικός Αναλυτής αποτελείται στην ουσία από μία γραμματική και τον `error handler`. Η γραμματική παίρνει ένα-ένα τα `token` που δημιουργήσε ο Λεκτικός Αναλυτής και εξετάζει εάν εφαρμόζονται οι κανόνες της γλώσσας όπως μας έχουν δοθεί από το `documentation` της γλώσσα `CutePy`. Αυτό το κάνει κοιτώντας την σειρά των `token` που έρχονται. Στην περίπτωση που βρεθεί κάποια ασυνέπεια με βάση τους κανόνες, καλείται ο διαχειριστής σφαλμάτων(`error handler`) για να εμφανίσει το κατάλληλο λάθος. Θα μπορούσαμε να παραλείψουμε στο συντακτικό αναλυτή να βάλουμε `error handler` και τα λάθη να τα διαχειρίζονταν απευθείας ο κώδικας της γραμματικής. Για λόγους όπως, η καθαρότητα του κώδικα και για την χρήση του στον Πίνακα Συμβόλων αργότερα, αποφασίσαμε να τον υλοποιήσουμε.

Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος

Αρχικά θα τρέξουμε το γνωστό μας παράδειγμα που εξ ορισμού είναι σωστό. Αυτό που περιμένουμε να γίνει είναι ότι πρέπει να τρέξει χωρίς λάθη. Πράγματι, το μήνυμα που βγάζει κάτω από τις εκτυπώσεις του Λεκτικού Αναλυτή, είναι το παρακάτω.

```
Compilation successfully completed
```

Τώρα μπορούμε να κάνουμε σκόπιμα κάποια τυχαία λάθη πάνω σε αυτό το παράδειγμα, που αφορούν όμως μόνο το συντακτικό κομμάτι, για να δούμε πώς θα το χειριστεί. Γενικά, το πρόγραμμα είναι τσεκαρισμένο σε πολλές περιπτώσεις. Ενδεικτικά εδώ όμως θα δώσουμε ένα τυχαίο λάθος, καθώς είναι ατελέσφορο να δώσουμε παράδειγμα για κάθε πιθανό σφάλμα.

```
1. def main_factorial():
2.     #{
3.         #$ declarations #$
4.         #declare x
5.         #declare i,fact
6.
7.         #$ body of main_factorial #$
```

```

8.     x = int(input());
9.     fact = 1
10.    i = 1;
11.    while (i<=x):
12.        #{
13.            fact = fact * i;
14.            i = i + 1;
15.        #}
16.    print(fact);
17. #}

```

Ο λανθασμένος κώδικας φαίνεται παραπάνω. Το λάθος το δημιουργήσαμε στην γραμμή 9 του αρχικού μας παραδείγματος, αφαιρώντας το ";" από το τέλος εκείνης της γραμμής. Το αποτέλεσμα που βγαίνει στο τερματικό δίνεται παρακάτω και φαίνεται ότι ήταν αυτό που περιμέναμε.

```

...TOKENS...
def , Family: Keyword , Line: 1
main_factorial , Family: Id , Line: 1
( , Family: GroupSymbol , Line: 1
) , Family: GroupSymbol , Line: 1
: , Family: Delimiter , Line: 1
#{ , Family: GroupSymbol , Line: 2
START OF COMMENTS
END OF COMMENTS
#declare , Family: Declare , Line: 4
x , Family: Id , Line: 4
#declare , Family: Declare , Line: 5
i , Family: Id , Line: 5
, , Family: Delimiter , Line: 5
fact , Family: Id , Line: 5
START OF COMMENTS
END OF COMMENTS
x , Family: Id , Line: 8
= , Family: Assignment , Line: 8
int , Family: Keyword , Line: 8
( , Family: GroupSymbol , Line: 8
input , Family: Keyword , Line: 8
( , Family: GroupSymbol , Line: 8
) , Family: GroupSymbol , Line: 8
) , Family: GroupSymbol , Line: 8
; , Family: Delimiter , Line: 8
fact , Family: Id , Line: 9
= , Family: Assignment , Line: 9
1 , Family: Number , Line: 9
i , Family: Id , Line: 10
SYNTAX ERROR at Line(9): expected to end with ;

```

Error handler

Ένα από τα δύο σημαντικά κομμάτια του Συντακτικού Αναλυτή είναι ο Διαχειριστής Σφαλμάτων. Σε περίπτωση λάθους, το πρόγραμμα θα εμφάνιζε ένα από τα παρακάτω λάθη.

```

def error(self, type_error):
    if type_error == "def_main_func_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct definition for main_functions"
        sys.exit(print_str)
    if type_error == "def_main_func_:_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): that is not a correct definition for main_functions"
        sys.exit(print_str)
    if type_error == "def_func_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct definition for functions"
        sys.exit(print_str)
    if type_error == "def_func_:_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): that is not a correct definition for functions"
        sys.exit(print_str)
    elif type_error == "#{_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected to open the function with '#{'"
        sys.exit(print_str)
    elif type_error == "#}_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected to close the function with '#}'"
        sys.exit(print_str)
    elif type_error == "var_exp_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): name variable expected"
        sys.exit(print_str)
    elif type_error == "def_exp_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected to start with def"
        sys.exit(print_str)
    elif type_error == "if_exp_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected to start with if or maybe def"
        sys.exit(print_str)
    elif type_error == "st_exp_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected some statements here"
        sys.exit(print_str)
    elif type_error == "var_name_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): illegal variable name"
        sys.exit(print_str)
    elif type_error == "call_main_part_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct call for main part"
        sys.exit(print_str)
    elif type_error == "call_main_part_:_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): that is not a correct call for main part"
        sys.exit(print_str)
    elif type_error == "call_main_func_err":
        print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct call for main functions"
        sys.exit(print_str)

```

```

elif type_error == "call_main_func_:_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): that is not a correct call for main functions"
    sys.exit(print_str)
elif type_error == "assignment_stat_:_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): expected to end with ;"
    sys.exit(print_str)
elif type_error == "assignment_stat_)_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected )"
    sys.exit(print_str)
elif type_error == "assignment_stat(_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected ("
    sys.exit(print_str)
elif type_error == ":_exp_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): expected :"
    sys.exit(print_str)
elif type_error=="assignment_stat_input_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected the word input"
    sys.exit(print_str)
elif type_error=="assignment_stat_start=_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct way to start after ="
    sys.exit(print_str)
elif type_error=="assignment_stat=_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected ="
    sys.exit(print_str)
elif type_error == "call_print_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct call for print"
    sys.exit(print_str)
elif type_error == "call_print_:_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): that is not a correct call for print"
    sys.exit(print_str)
elif type_error == "call_return_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct call for return"
    sys.exit(print_str)
elif type_error == "call_return_:_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): that is not a correct call for return"
    sys.exit(print_str)

```

```

elif type_error == "(_exp_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected to open the parentheses with '('"
    sys.exit(print_str)
elif type_error == ")_exp_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected to close the parentheses with ')'"
    sys.exit(print_str)
elif type_error == "expr_exp_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected some expression here"
    sys.exit(print_str)
elif type_error == "define_err":
    print_str="SYNTAX ERROR at Line("+str(token.getLineNumber())+"): '"+str(token.getRecognizedString())+"' is not defined"
    sys.exit(print_str)
elif type_error == "[_exp_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected to open the brackets with '['"
    sys.exit(print_str)
elif type_error == "]_exp_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected to close the brackets with ']'"
    sys.exit(print_str)
elif type_error == "relOp_exp_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected some RelOperations here"
    sys.exit(print_str)
elif type_error == "cond_exp_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): expected some condition here"
    sys.exit(print_str)
elif type_error == "if_body_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct start for if statement"
    sys.exit(print_str)
elif type_error == "while_body_err":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber())+"): that is not a correct start for while statement"
    sys.exit(print_str)
elif type_error == "no_return_main":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): no return statement in main functions"
    sys.exit(print_str)
elif type_error == "must_return_func":
    print_str="SYNTAX ERROR at Line("+str(self.token.getLineNumber()-1)+"): functions must have a return statement"
    sys.exit(print_str)

```

Ενδιάμεσος Κώδικας

Εισαγωγή

Ο Ενδιάμεσος Κώδικας, πολύ περιγραφικά, εκμεταλλεύεται τον Συντακτικό Αναλυτή για να δημιουργηθεί και χρησιμοποιείται στην συνέχεια για την παραγωγή του Τελικού Κώδικα. Είναι δηλαδή ένας μεσάζοντας, ανάμεσα στον πηγαίο και στον τελικό κώδικα. Για να ελεγχθεί η ορθότητα του, που αυτό είναι το κύριο μέλημά μας, πρέπει να δοθεί ιδιαίτερη έμφαση στην σειρά με την οποία εμφανίζονται οι τετράδες και στο περιεχόμενο κάθε μίας εξ αυτών.

Υλοποίηση

Ο ενδιάμεσος κώδικας είναι το σημείο στο οποίο φτιάχνουμε μία λίστα από τετράδες. Κάτω από το συντακτικό αναλυτή βρίσκεται ο κώδικας για τις τετράδες και για τις μεθόδους τους. Έπειτα προσθέσαμε στο κώδικα του συντακτικού στα κατάλληλα σημεία τη δημιουργία των τετράδων που θα χρησιμοποιήσουμε αργότερα στο τελικό κώδικα.

Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος

Τρέχοντας παρακάτω το κλασικό μας παράδειγμα, θα διαπιστώσουμε ότι συγκρίνοντας τον πηγαίο κώδικα με τις τετράδες του Ενδιάμεσου Κώδικα που δημιουργούνται, καταλαβαίνουμε ότι δίνονται σωστά οι προτεραιότητες, είτε πράξεων είτε ορισμών συναρτήσεων, και το περιεχόμενο κάθε τετράδας αντιστοιχίζεται ορθά με τις αντίστοιχες γραμμές του πηγαίου κώδικα του παραδείγματός μας.

```
...ENDIAMESOS KVDIKAS...
1: begin_block , main_factorial , _ , _
2: in , x , _ , _
3: := , 1 , _ , fact
4: := , 1 , _ , i
5: <= , i , x , 7
6: jump , _ , _ , 12
7: * , fact , i , T_1
8: := , T_1 , _ , fact
9: + , i , 1 , T_2
10: := , T_2 , _ , i
11: jump , _ , _ , 5
12: out , fact , _ , _
13: end_block , main_factorial , _ , _
14: begin_block , fibonacci , _ , _
15: <= , x , 1 , 17
16: jump , _ , _ , 19
17: ret , x , _ , _
18: jump , _ , _ , 29
19: - , x , 1 , T_3
20: par , T_3 , cv , fibonacci
21: par , T_4 , ret , fibonacci
22: call , fibonacci , _ , _
23: - , x , 2 , T_5
24: par , T_5 , cv , fibonacci
25: par , T_6 , ret , fibonacci
26: call , fibonacci , _ , _
27: + , T_4 , T_6 , T_7
28: ret , T_7 , _ , _
29: end_block , fibonacci , _ , _
30: begin_block , main_fibonacci , _ , _
31: in , x , _ , _
32: par , x , cv , fibonacci
33: par , T_8 , ret , fibonacci
34: call , fibonacci , _ , _
35: out , T_8 , _ , _
36: end_block , main_fibonacci , _ , _
37: begin_block , main_countdigits , _ , _
38: in , x , _ , _
39: := , 0 , _ , count
40: > , x , 0 , 42
41: jump , _ , _ , 47
42: // , x , 10 , T_9
43: := , T_9 , _ , x
44: + , count , 1 , T_10
45: := , T_10 , _ , count
46: jump , _ , _ , 40
47: out , count , _ , _
48: end_block , main_countdigits , _ , _
49: begin_block , divides , _ , _
50: // , y , x , T_11
51: * , T_11 , x , T_12
52: == , y , T_12 , 54
53: jump , _ , _ , 56
54: ret , 1 , _ , _
55: jump , _ , _ , 57
56: ret , 0 , _ , _
57: end_block , divides , _ , _
58: begin_block , isPrime , _ , _
59: := , 2 , _ , i
60: < , i , x , 62
61: jump , _ , _ , 73
62: par , i , cv , divides
63: par , x , cv , divides
64: par , T_13 , ret , divides
65: call , divides , _ , _
66: == , T_13 , 1 , 68
67: jump , _ , _ , 70
68: ret , 0 , _ , _
69: jump , _ , _ , 70
70: + , i , 1 , T_14
```

```

71 : := , T_14 , _ , i
72 : jump , _ , _ , 60
73 : ret , 1 , _ , _
74 : end_block , isPrime , _ , _
75 : begin_block , main_primes , _ , _
76 : := , 2 , _ , i
77 : <= , i , 30 , 79
78 : jump , _ , _ , 89
79 : par , i , cv , isPrime
80 : par , T_15 , ret , isPrime
81 : call , isPrime , _ , _
82 : == , T_15 , 1 , 84
83 : jump , _ , _ , 86
84 : out , i , _ , _
85 : jump , _ , _ , 86
86 : + , i , 1 , T_16
87 : := , T_16 , _ , i
88 : jump , _ , _ , 77
89 : end_block , main_primes , _ , _
90 : begin_block , main , _ , _
91 : call , main_factorial , _ , _
92 : call , main_fibonacci , _ , _
93 : call , main_countdigits , _ , _
94 : call , main_primes , _ , _
95 : halt , _ , _ , _
96 : end_block , main , _ , _

```

Αποτέλεσμα Εκτέλεσης Άλλου Παραδείγματος

Για μεγαλύτερη απόδειξη της ορθής λειτουργίας του Ενδιάμεσου Πίνακα που υλοποιήσαμε, είναι η χρήση ενός άλλου παραδείγματος που μας έχει δοθεί στις σημειώσεις του αντίστοιχου κεφαλαίου. Εκεί παρατίθεται και μία λίστα από τετράδες του Ενδιάμεσου Κώδικα που θα την συγκρίνουμε με την αντίστοιχη δικιά μας λίστα από τετράδες που εκτυπώνεται στο τερματικό.

Αξίζει να σημειωθεί ότι έγινε μία προσθήκη στον κώδικα που μας δόθηκε ώστε να λειτουργεί στην δική μας γλώσσα, την CutePy. Εδώ παρακάτω φαίνεται αυτός ο νέος κώδικας. *Η προσθήκη που έγινε είναι μόνο να βάλουμε εκεί στο τέλος το τύπου main κομμάτι ώστε να καλέσουμε την κύρια συνάρτηση[γραμμές 22-24].*

```

1. def main_ifWhile():
2.     #{
3.         #declare c,a,b,t
4.
5.         a=1;
6.         while (a+b<1 and b<5):
7.             #{
8.                 if (t==1):
9.                     c=2;
10.                else:
11.                    if (t==2):
12.                        c=4;
13.                    else:
14.                        c=0;
15.                while (a<1):
16.                    if (a==2):
17.                        while(b==1):
18.                            c=2;

```

```

19.    #}
20.#}
21.
22.if __name__ == "__main__":
23.    #$ call of main functions #$
24.    main_ifWhile();

```

Η αριστερή λίστα από τετράδες του Ενδιάμεσου Κώδικα είναι αυτό που παράγουμε εμείς στο τερματικό ενώ η δεξιά είναι αυτή που παίρνουμε από το pdf των σημειώσεων:

...ENDIAMESOS KVDIKAS...	1 :	begin_block, main_ifWhile, _, _
1 : begin_block , main_ifwhile , _ , _	2 :	:=, 1, _, a
2 : := , 1 , _ , a	3 :	+, a, b, T_1
3 : + , a , b , T_1	4 :	<, T_1, 1, 6
4 : < , T_1 , 1 , 6	5 :	jump, _, _, 28
5 : jump , _ , _ , 28	6 :	<, b, 5, 8
6 : < , b , 5 , 8	7 :	jump, _, _, 28
7 : jump , _ , _ , 28	8 :	=, t, 1, 10
8 : == , t , 1 , 10	9 :	jump, _, _, 12
9 : == , t , 1 , 10	10 :	:=, 2, _, c
10 : := , 2 , _ , c	11 :	jump, _, _, 17
11 : jump , _ , _ , 17	12 :	=, t, 2, 14
12 : == , t , 2 , 14	13 :	jump, _, _, 16
13 : jump , _ , _ , 16	14 :	:=, 4, _, c
14 : := , 4 , _ , c	15 :	jump, _, _, 17
15 : jump , _ , _ , 17	16 :	:=, 0, _, c
16 : := , 0 , _ , c	17 :	<, a, 1, 19
17 : < , a , 1 , 19	18 :	jump, _, _, 27
18 : jump , _ , _ , 27	19 :	=, a, 2, 21
19 : == , a , 2 , 21	20 :	jump, _, _, 26
20 : == , a , 2 , 21	21 :	=, b, 1, 23
21 : == , b , 1 , 23	22 :	jump, _, _, 25
22 : jump , _ , _ , 25	23 :	:=, 2, _, c
23 : := , 2 , _ , c	24 :	jump, _, _, 21
24 : := , 2 , _ , c	25 :	jump, _, _, 26
25 : jump , _ , _ , 26	26 :	jump, _, _, 17
26 : jump , _ , _ , 17	27 :	jump, _, _, 3
27 : jump , _ , _ , 17	28 :	halt, _, _, _
28 : jump , _ , _ , 3	29 :	end_block, main_ifWhile, _, _
29 : end_block , main_ifwhile , _ , _		
30 : call , main_ifwhile , _ , _		
31 : halt , _ , _ , _		
32 : end_block , main , _ , _		

Παρατηρούμε, εκτός εννοείτε από τις τελευταίες γραμμές που ήταν αναμενόμενο να αλλάξουν ελαφρώς χωρίς να βγάλουν όμως λανθασμένο λογικό αποτέλεσμα, ότι βγάζουμε τα ίδια αποτελέσματα.

Πίνακας Συμβόλων

Εισαγωγή

Σε γενικές γραμμές, ο Πίνακας Συμβόλων έχει την δυνατότητα να διακρίνει λάθη τύπου μη δήλωσης είτε μεταβλητών είτε συναρτήσεων κλπ, άρα επικοινωνεί και με τον `error handler`. Αλλά επίσης, προσφέρει και πολλές άλλες χρήσιμες πληροφορίες που θα αναλύσουμε στην συνέχεια.

Εξετάζοντας όμως τώρα τον Πίνακα Συμβόλων στο πεδίο που μας ενδιαφέρει περισσότερο, βλέπουμε ότι είναι μία δομή που αλλάζει σε ορισμένα σημεία το περιεχόμενό του, πράγμα που καθιστά δύσκολο τον έλεγχο της ορθότητάς του. Ο τρόπος λοιπόν να ελέγξουμε τον πίνακα συμβόλων είναι να δούμε στιγμιότυπα κατά τη διάρκεια που τρέχει ο κώδικας. Τα καίρια σημεία που κρίναμε σκόπιμο να εκτυπώνεται το περιεχόμενο του πίνακα συμβόλων, είναι σε κάθε `end_block` κάθε συνάρτησης, κύριας και μη, διότι σε αυτά τα σημεία έχουμε την μέγιστη και την πιο ενημερωμένη πληροφορία που χρειαζόμαστε κάθε φορά. Δηλαδή, θα μπορούμε να δούμε αν στο συγκεκριμένο σημείο έχουν μπει τα κατάλληλα στοιχεία με τις κατάλληλες τιμές στα `offset` και `framelength`.

Υλοποίηση

Όλη η πληροφορία που βάζουμε στον Πίνακα Συμβόλων είναι εξίσου χρήσιμη και σημαντική, αλλά θα εστιάσουμε και θα αναλύσουμε μόνο την εισαγωγή Μεταβλητών, Παραμέτρων και Συναρτήσεων, για να σχολιάσουμε μία ιδιαιτερότητα της γλώσσας CUTEPy. Επομένως, αξίζει να σημειωθεί το πώς και το γιατί φτιάχτηκε το πρώτο επίπεδο στον Πίνακα που έχει όλες τις κύριες συναρτήσεις. Επειδή στην γλώσσα μας, έχουμε κάτω-κάτω στον κώδικα κάθε προγράμματος μία τύπου `main`, που καλεί όλες τις κύριες συναρτήσεις του προγράμματός μας, αποφασίσαμε να δεσμεύσουμε το πρώτο επίπεδο του Πίνακα Συμβόλων, το επίπεδο μηδέν, για να βάζουμε εκεί όλες αυτές τις συναρτήσεις.

Πέρα από αυτήν την ιδιαιτερότητα, μπορούμε σε αυτό το σημείο να εξηγήσουμε πότε μπαίνουν αυτοί οι 3 τύποι που αναφέρθηκαν στην αρχή. Σε γενικές γραμμές, εισάγονται στον Πίνακα Συμβόλων όταν δηλώνονται (πχ `#declare` για μεταβλητές και `def func_name` για συναρτήσεις). Αυτό που ακολουθεί μετά από την δήλωσή τους συνήθως, είναι η χρήση ή η κλίση τους αντίστοιχα. Τότε, γίνεται μία διαδικασία που ψάχνει αν υπάρχει αυτή η συγκεκριμένη μεταβλητή ή συνάρτηση, ανάλογα την περίπτωση. Αν δεν την βρει σε οποιοδήποτε επίπεδο του Πίνακα Συμβόλων, τότε θα καλέσει τον `error handler`. Ένα παράδειγμα διαχείρισης λάθους, παρουσιάζεται και αργότερα παρακάτω.

Προφανώς ο Πίνακας Συμβόλων έχει και άλλες δυνατότητες και λειτουργίες που έχουν υλοποιηθεί στο παραδοτέο κώδικα, αλλά κρίθηκε σκόπιμο να μην παρουσιαστούν στα πλαίσια της αναφοράς.

Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος

Παρακάτω, φαίνονται κάποια τυχαία στιγμιότυπα του Πίνακα Συμβόλων. Πιο συγκεκριμένα φαίνονται αυτά των γραμμών 29 και 33. Παρατηρούμε ότι η εκτύπωση του Πίνακα Συμβόλων γίνεται ενδιαμέσως της εκτύπωσης του Λεκτικού Αναλυτή, όπως ήταν αναμενόμενο αφού επιλέξαμε αυτή την στρατηγική δυναμικής παρουσίασης. Καθαρά οι Πίνακες Συμβόλων βρίσκονται μέσα στα κόκκινα πλαίσια.

```
; , Family: Delimiter , Line: 28
#{ , Family: GroupSymbol , Line: 29
...PINAKAS SYBOLON...
2 <- x/int/12/cv <- T_3/int/16 <- T_4/int/20 <- T_5/int/24 <- T_6/int/28 <- T_7/int/32
1 <- x/int/12 <- fibonacci/14/36[x/int/cv]
0 <- main_factorial/1/32[] <- main_fibonacci/0/0[]
x , Family: Id , Line: 31
= , Family: Assignment , Line: 31
int , Family: Keyword , Line: 31
( , Family: GroupSymbol , Line: 31
input , Family: Keyword , Line: 31
( , Family: GroupSymbol , Line: 31
) , Family: GroupSymbol , Line: 31
) , Family: GroupSymbol , Line: 31
; , Family: Delimiter , Line: 31
print , Family: Keyword , Line: 32
( , Family: GroupSymbol , Line: 32
fibonacci , Family: Id , Line: 32
( , Family: GroupSymbol , Line: 32
x , Family: Id , Line: 32
) , Family: GroupSymbol , Line: 32
) , Family: GroupSymbol , Line: 32
; , Family: Delimiter , Line: 32
#{ , Family: GroupSymbol , Line: 33
...PINAKAS SYBOLON...
1 <- x/int/12 <- fibonacci/14/36[x/int/cv] <- T_8/int/16
0 <- main_factorial/1/32[] <- main_fibonacci/30/20[]
def , Family: Keyword , Line: 35
main_countdigits , Family: Id , Line: 35
```

Επίσης, πέρα από την εκτύπωση του περιεχομένου του Πίνακα Συμβόλων, αξίζει εδώ να τρέξουμε και ένα λανθασμένο παράδειγμα όπως κάναμε και στον Συντακτικό Αναλυτή. Αλλά τώρα, για παράδειγμα, το είδος του λάθους μας θα είναι η χρήση μιας μη δηλωμένης μεταβλητής.

```
1. def main_factorial():
2.   #{
3.     #$ declarations #$
4.
5.     #declare i,fact
6.
7.     #$ body of main_factorial #$
8.     x = int(input());
9.     fact = 1;
10.    i = 1;
11.    while (i<=x):
12.      #{
13.        fact = fact * i;
14.        i = i + 1;
```

```

15.     #}
16.     print(fact);
17. #}

```

Ο λανθασμένος κώδικας φαίνεται παραπάνω. Το λάθος το δημιουργήσαμε στην γραμμή 4 του αρχικού μας παραδείγματος, αφαιρώντας ολόκληρη την συγκεκριμένη γραμμή, η οποία ήταν υπεύθυνη για την δήλωση της μεταβλητής "x". Το αποτέλεσμα που βγαίνει στο τερματικό δίνεται παρακάτω και φαίνεται ότι ήταν αυτό που περιμέναμε.

```

...TOKENS...
def , Family: Keyword , Line: 1
main_factorial , Family: Id , Line: 1
( , Family: GroupSymbol , Line: 1
) , Family: GroupSymbol , Line: 1
: , Family: Delimiter , Line: 1
#{ , Family: GroupSymbol , Line: 2
START OF COMMENTS
END OF COMMENTS
#declare , Family: Declare , Line: 5
i , Family: Id , Line: 5
, , Family: Delimiter , Line: 5
fact , Family: Id , Line: 5
START OF COMMENTS
END OF COMMENTS
x , Family: Id , Line: 8
...TOKENS...
SYNTAX ERROR at Line(8): 'x' is not defined

```

Η ίδια διαδικασία μπορεί να γίνει, όχι μόνο με μεταβλητές, αλλά και με συναρτήσεις. Δηλαδή, αν καλέσουμε μία συνάρτηση η οποία δεν έχει οριστεί πιο πάνω, εδώ θα είναι το σωστό σημείο να καλεστεί ο error handler και να εμφανίσει πάλι το κατάλληλο μήνυμα λάθους.

Αποτέλεσμα Εκτέλεσης Άλλου Παραδείγματος

Για μεγαλύτερη απόδειξη της ορθής λειτουργίας του Πίνακα Συμβόλων που υλοποιήσαμε, είναι η χρήση ενός άλλου παραδείγματος που μας έχει δοθεί στις σημειώσεις του αντίστοιχου κεφαλαίου. Εκεί παρατίθεται και ένα στιγμιότυπο του πίνακα που θα το συγκρίνουμε με το αντίστοιχο δικό μας στιγμιότυπο που εκτυπώνεται στο τερματικό.

Αξίζει να σημειωθεί ότι έγιναν κάποιες αλλαγές στον κώδικα που μας δόθηκε ώστε να λειτουργεί στην δική μας γλώσσα, την CutePy. Εδώ παρακάτω φαίνεται αυτός ο ελαφρός αλλαγμένος κώδικας. *Οι αλλαγές που έγιναν είναι μόνο να βάλουμε σε σχόλια την αρχικοποίηση της global μεταβλητής "A"[γραμμή 3] και να προσθέσουμε στο declare άλλες 2 μεταβλητές, την "f1" και την "f2"[γραμμή 4], για να μπορέσει να αποθηκευτεί κάπου το αποτέλεσμα της κλίσης των συναρτήσεων P1 και P2 αντίστοιχα[γραμμές 41-42].*

```

1. def main_symbol():
2.     #{
3.         #$ const A=1; #$

```

```

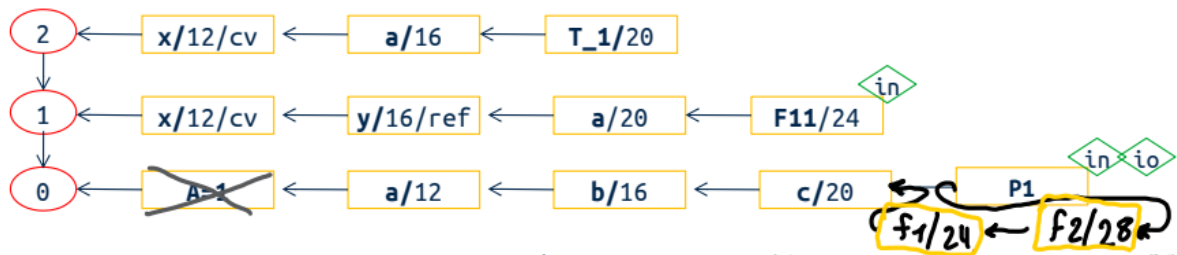
4.      #declare a,b,c,f1,f2
5.
6.      def P1(x,y):
7.          #{
8.              #declare a
9.              def F11(x):
10.                 #{
11.                     #declare a
12.                     #$ body of F11 #$
13.                     b = a;
14.                     a = x;
15.                     c = F11(x);
16.                     return (c);
17.                 #}
18.
19.             def F12(x):
20.                 #{
21.                     #$ body of F12 #$
22.                     c = F11(x);
23.                     return (c);
24.                 #}
25.
26.             #$ body of P1 #$
27.             y = x;
28.             return (y);
29.         #}
30.
31.     def P2(x):
32.         #{
33.             #declare y
34.             #$ body of P2 #$
35.             y = 1;
36.             y = P1(x,y);
37.             return (c);
38.         #}
39.
40.     #$ main program #$
41.     f1 = P1(a,b);
42.     f2 = P2(c);
43. #}
44.
45. if __name__ == "__main__":
46.     #$ call of main functions #$
47.     main_symbol();

```

Τα 2 στιγμιότυπα των Πινάκων Συμβόλων που παρουσιάζουμε παρακάτω, είναι στο end_block της συνάρτησης F11, δηλαδή μετά την ολοκλήρωση της μετάφρασής της. Αυτό το σημείο βρίσκεται στην γραμμή 17 του παραπάνω κώδικα. Το πρώτο στιγμιότυπο είναι αυτό που παράγουμε εμείς στο τερματικό:

```
; , Family: Delimiter , Line: 16
#} , Family: GroupSymbol , Line: 17
...PINAKAS SYBOLON...
3 <- x/int/12/cv <- a/int/16 <- T_1/int/20
2 <- x/int/12/cv <- y/int/16/cv <- a/int/20 <- F11/1/24[x/int/cv]
1 <- a/int/12 <- b/int/16 <- c/int/20 <- f1/int/24 <- f2/int/28 <- P1/0/0[x/int/cv | y/int/cv]
0 <- main symbol/0/0[]
def , Family: Keyword , Line: 19
```

Και το δεύτερο είναι αυτό που παίρνουμε από το pdf των σημειώσεων:



Παρατηρούμε, κάνοντας προφανώς τις απαραίτητες αλλαγές στην δεύτερη εικόνα για τους λόγους που εξηγήσαμε παραπάνω, ότι βγάζουμε τα ίδια αποτελέσματα. Επίσης, έχουμε εξηγήσει για ποιόν λόγο έχουμε αυτό το πρώτο επίπεδο(επίπεδο_0), στο κεφάλαιο Υλοποίηση της ίδιας ενότητας.

Τελικός Κώδικας

Εισαγωγή

Φτάσαμε στην τελευταία φάση, όπου είναι υπεύθυνη για την παραγωγή του τελικού κώδικα. Δηλαδή εδώ παράγεται ένα .asm αρχείο που περιέχει τον μετατραμμένο πηγαίο κώδικα σε γλώσσα μηχανής(assembly). Ο κώδικας αυτός, είναι κατασκευασμένος κατάλληλα ώστε να τρέχει σε Risc-V επεξεργαστές. Επομένως, εκτελώντας το συγκεκριμένο αρχείο σε ένα τέτοιου τύπου επεξεργαστή, θα δούμε τα αποτελέσματα της εκτέλεσης του αρχικού μας προγράμματος, που αυτός είναι και ο απώτερός μας στόχος.

Υλοποίηση

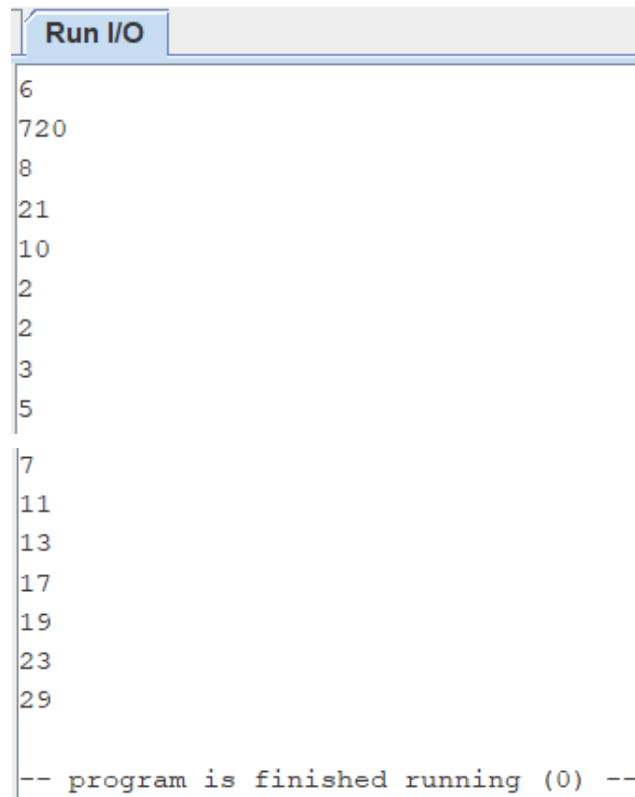
Προφανώς για να υλοποιηθεί ο τελικός κώδικας χρειαζόμαστε όλες τις φάσεις που προαναφέραμε, αλλά ιδιαίτερα εκμεταλλευόμαστε τον ενδιάμεσο κώδικα και τον πίνακα συμβόλων. Πιο συγκεκριμένα, κάθε γραμμή του ενδιάμεσου κώδικα αντιστοιχίζεται σε ένα label που έχει τις κατάλληλες εντολές assembly, αναλόγως την κάθε περίπτωση, καθώς και κάθε στιγμιότυπο του πίνακα συμβόλων που δημιουργείται σε κάθε end_block, για να πάρουμε τις απαραίτητες πληροφορίες, όπως εμβέλεις, ορισμούς, offset, framelength κλπ.

Αποτέλεσμα Εκτέλεσης Του Γνωστού Παραδείγματος

Παρακάτω φαίνεται αρχικά ένα μικρό κομμάτι από τον παραγόμενο τελικό κώδικα(assembly), η αρχή του και το τέλος του μόνο, αφού είναι αρκετά μεγάλος ώστε να τον παρουσιάσουμε όλον εδώ.

1	.data	326	L89:
2	str_nl: .asciz "\n"	327	lw ra, (sp)
3	.text	328	jrr ra
4		329	Lmain:
5	L0:	330	L91:
6	j Lmain	331	addi sp, sp, 32
7	L1:	332	jal L1
8	sw ra, (sp)	333	addi sp, sp, -32
9	L2:	334	L92:
10	li a7, 5	335	addi sp, sp, 20
11	ecall	336	jal L30
12	sw a0, -12(sp)	337	addi sp, sp, -20
13	L3:	338	L93:
14	li t1, 1	339	addi sp, sp, 28
15	sw t1, -20(sp)	340	jal L37
16	L4:	341	addi sp, sp, -28
17	li t1, 1	342	L94:
18	sw t1, -16(sp)	343	addi sp, sp, 24
19	L5:	344	jal L75
20	lw t1, -16(sp)	345	addi sp, sp, -24
21	lw t2, -12(sp)	346	L95:
22	ble t1, t2, L7	347	li a0, 0
23	L6:	348	li a7, 93
24	j L12	349	ecall
25	L7:		

Αλλά έπειτα παρακάτω, φαίνεται το πιο σημαντικό, το τελικό αποτέλεσμα, το οποίο σηματοδοτεί και την ολική λήξη του προγράμματος. Είναι μία φωτογραφία που έχει παρθεί από το γραφικό περιβάλλον της εφαρμογής του επεξεργαστή Risc-V.



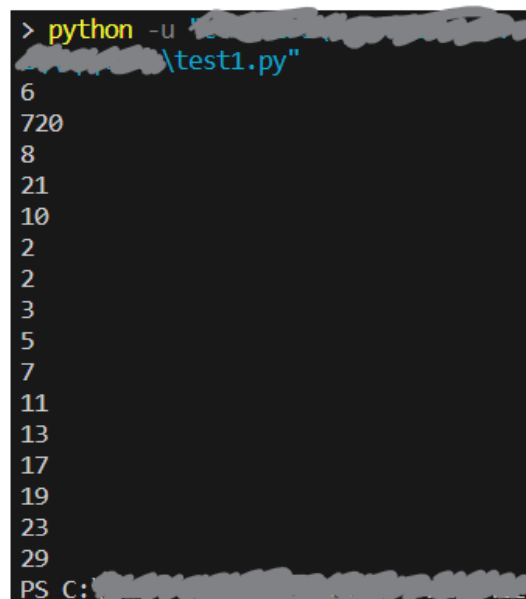
```
Run I/O
6
720
8
21
10
2
2
3
5

7
11
13
17
19
23
29

-- program is finished running (0) --
```

Αποτέλεσμα Εκτέλεσης Με Python

Η CutePy είναι μία γλώσσα που έχει φτιαχτεί έτσι ώστε ο κώδικάς της, να είναι ταυτόχρονα και κώδικας που μπορεί να τρέξει σε Python. Επομένως μπορούμε να τρέξουμε το ίδιο παράδειγμα και σαν ένα αρχείο .py, για να συγκρίνουμε τα αποτελέσματα που θα βγουν.



```
> python -u test1.py
6
720
8
21
10
2
2
3
5

7
11
13
17
19
23
29
PS C:\>
```

Συμπέρασμα-Παρατηρήσεις

Πράγματι, συμπεραίνουμε ότι με το τυχαίο παράδειγμα με τις εισόδους “6”, “8” και “10” που βάλαμε, και οι 2 εκτελέσεις, και της CutePy και της Python, βγάζουν πανομοιότυπα αποτελέσματα.

Όμως εντοπίσαμε μία πολύ ενδιαφέρουσα περίπτωση. Μία περίπτωση όπου το αποτέλεσμα της εκτέλεσης ενός συγκεκριμένου κώδικα διαφέρει ανάλογα το ποιος compiler καλείτε, δηλαδή της CutePy ή της Python. Αυτό είναι λογικό και αναμενόμενο, αφού αυτές οι δύο γλώσσες έχουν διαφορετικό τρόπο λειτουργίας ως προς τον τρόπο διαχείρισης και αποθήκευσης της μνήμης. Παρακάτω φαίνεται ένας τέτοιου τύπου κώδικας. Πιο συγκεκριμένα, εστιάζουμε στην μεταβλητή “A”, που μέσα σε μία φωλιασμένη συνάρτηση της εκχωρούμε την τιμή της μεταβλητής “B” [γραμμή 14]. Όταν μετά την κλίση της συγκεκριμένης φωλιασμένης συνάρτησης παρακάτω, εκτυπώνουμε την μεταβλητή “A” [γραμμές 25 και 41], παρατηρούμε την διαφορά των αποτελεσμάτων ανάμεσα σε αυτές τις δύο γλώσσες. Στην CutePy το “A” έχει αλλάξει την τιμή του από “1” σε “2”, ενώ στην Python η τιμή του έχει μείνει στην αρχική του, την “1”.

```
1. def main_finalCodeExample():
2.     #{
3.         #declare A,B
4.         #declare C
5.
6.         def proc(a, b):
7.             #{
8.                 #declare c
9.
10.                def func():
11.                    #{
12.                        #declare d
13.                        d=4;
14.                        A=B;
15.                        #$B=A; den xreiazetai#$
16.                        print(A);
17.                        return(c+d);
18.                    #}
19.
20.                #$ body of proc #$
21.                c=3;
22.                print(A); #1
23.                print(B); #2
24.                C=func(); #2 from line 16
25.                print(A); #?
26.                print(B); #2
27.                print(C); #7
```

```

28.
29.     #$ #$
30.     return(0);
31.     #$ #$
32.     #}
33.
34.     #$ body of finalCodeExample #$
35.     A=1;
36.     B=2;
37.
38.     #$ call proc(A, B); #$
39.     C = proc(A, B);
40.
41.     print(A); #?
42.     print(B); #2
43. #}
44.
45. if __name__ == "__main__":
46.     #$ call of main functions #$
47.     main_finalCodeExample();
48.

```

Τα αποτελέσματα φαίνονται παρακάτω, με το αριστερό να αντιστοιχίζεται στο αποτέλεσμα της κλήσης της CutePy και το δεξί στο αποτέλεσμα της κλήσης της Python.

Run I/O	python
1	\test7.py"
2	1
2	2
2	2
2	1
2	2
7	7
2	1
2	2