ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ.

Εργασία

ΜΕΤΑΦΡΑΣΤΕΣ

Μπουρλή Ευτυχία AM: 4441

2022

Σκοπός της άσκησης:

Μέσα από αυτήν την εργασία να μπορέσουμε να αντιληφθούμε βασικές έννοιες της θεωρίας των μεταφραστών και να υλοποιήσουμε ένα μεταφραστή της γλώσσας προγραμματισμού cimple, ώστε αργότερα να είμαστε σε θέση να μπορούμε να υλοποιήσουμε τον μεταφραστή για οποιαδήποτε άλλη γλώσσα.

Περίληψη:

Το Report αποτελείται αρχικά από την ανάλυση των μερών του κώδικα που υλοποιήθηκαν σε Python και στη συνέχεια ακολουθούν παραδείγματα που εκτελέσαμε μαζί με τα αποτελέσματα που τύπωσε το πρόγραμμά μας.

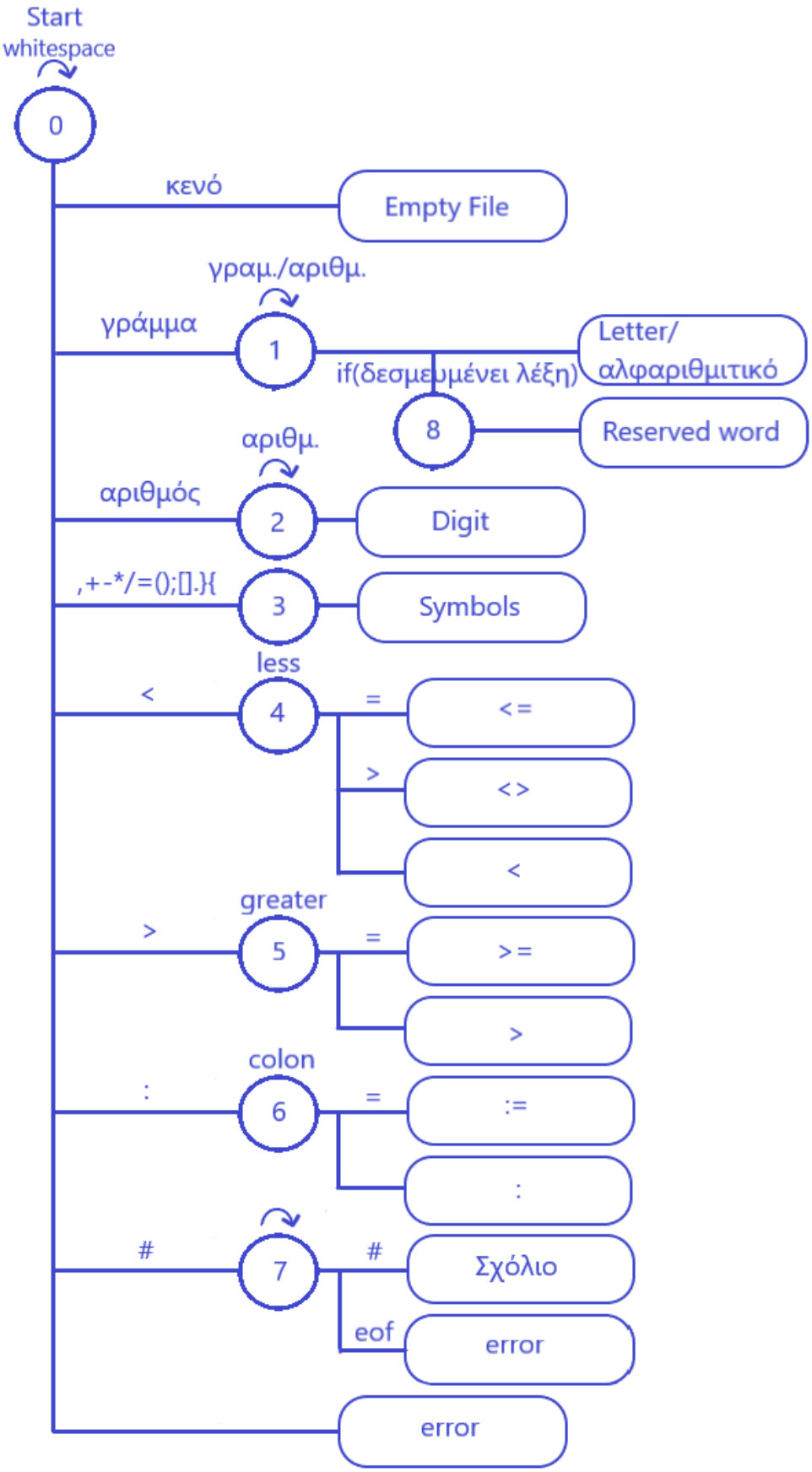
Ο κώδικας αποτελείται από 5 μέρη. Τον Λεκτικό αναλυτή, τον Συντακτικό αναλυτή, τον ενδιάμεσο κώδικα, τον σημασιολογικό αναλυτή (Πίνακα συμβόλων) και τον Τελικό κώδικα.

Λεκτικός Αναλυτής:

Σκοπός του λεκτικού αναλυτή είναι να αναγνωρίζει όλα τα σύμβολα της γλώσσας που μεταγλωττίζουμε. Δουλειά του είναι να διαβάζει το αρχείο με τον κώδικα της γλώσσας να το σπάει σε πακετάκια/λεκτικές μονάδες και να μπορεί να αναγνωρίσει αν το πακετάκι αυτό αποτελεί για παράδειγμα κάποια μεταβλητή, αριθμό, δεσμευμένη λέξη ή σύμβολο και να επιστρέφει το πακετάκι αυτό μαζί με ένα αναγνωριστικό, με το οποίο καταλαβαίνουμε την ιδιότητα του πακέτου. Σε περίπτωση που το πακετάκι δεν ανήκει σε καμία κατηγορία χτυπάει επιστρέφουμε μήνυμα λάθους.

Παρακάτω ακολουθεί το αυτόματο σύμφωνα με το οποίο κατηγοριοποιούμε τις λεκτικές μονάδες της γλώσσας cimple μαζί με λίγες πληροφορίες για την υλοποίησή του.

Αυτόματο



Μέσα στον Λεκτικό Αναλυτή δίνουμε τιμές στις global μεταβλητές:

Word_unit, η οποία κρατά την λεκτική μονάδα

Identifier, η οποία είναι το αναγνωριστικό της λεκτικής μονάδας

Επιπλέον για κάθε αλλαγή γραμμής αλλάζουμε την μεταβλητή Line, έτσι ώστε να ξέρουμε σε ποια γραμμή του προγράμματος βρίσκεται η λεκτική μονάδα.

Σε περίπτωση που δεν χρησιμοποιήσουμε τον επόμενο χαρακτήρα για την λεκτική μονάδα μετακινούμε τον δείκτη μία θέση πίσω ώστε να τον ξαναδιαβάσουμε για την επόμενη.

Σε περίπτωση σφάλματος μεταβαίνουμε στην κατάσταση -1

Συντακτικός Αναλυτής:

Κατασκευάσαμε έναν συντακτικό αναλυτή αναδρομικής κατάβασης με βάση την γραμματική τύπου LL(1), ο οποίος ελέγχει για συντακτικά σφάλματα.

Καλεί τον λεκτικό αναλυτή ώστε να πάρουν τιμές οι global μεταβλητές και μόλις καταναλώσει την λεκτική μονάδα τον ξανά καλεί ώστε να πάρει την επόμενη.

Σε περίπτωση λάθους μας τυπώνει το αντίστοιχο μήνυμα λάθους και σε ποια γραμμή βρίσκεται μέσα στον κώδικα cimple καθώς και από ποια μέθοδο του προγράμματός μας προήλθε το σφάλμα, ώστε να είναι εύκολος ο εντοπισμός τυχόν λαθών.

Σε περίπτωση ορθού κώδικα το πρόγραμμά μας τερματίζει χωρίς να τυπώσει τίποτα.

Θεωρία:

Γραμματική LL(1):

- L : left to right
- L : leftmost derivation
- (1) : one look-ahead symbol

  - Η γραμματική LL(1) αναγνωρίζει από **αριστερά στα δεξιά**, **την αριστερότερη δυνατή παραγωγή** και όταν βρίσκεται σε δίλλημα ποιον κανόνα να ακολουθήσει της αρκεί να κοιτάξει το **αμέσως επόμενο σύμβολο** στην συμβολοσειρά εισόδου

1. Δοθέντος συγκεκριμένου μη τερματικού συμβόλου εφαρμόζεται ο πρώτος κανόνας της γραμματικής.
2. Στην προτασιακή μορφή, που προκύπτει επιλέγεται το πρώτο από αριστερά μη τερματικό σύμβολο και εφαρμόζεται ο πρώτος κανόνας, που αναφέρεται σε αυτό.
3. Γίνεται επαναληπτική εφαρμογή του βήματος 2, για κάθε ένα από τα μη τερματικά σύμβολα, που ακολουθούν, μέχρις ότου παραχθεί:
   - *μια σειρά τερματικών συμβόλων (πρόταση γλώσσας) ή*
   - *τμήμα τερματικών συμβόλων της προτασιακής μορφής, που διαφέρει από το αντίστοιχο τμήμα της συμβολοσειράς εισόδου*

Ενδιάμεσος Κώδικας:

Θεωρία:

- Ο ενδιάμεσος κώδικας είναι ένα σύνολο από τετράδες
  - ένας τελεστής
  - τρία τελούμενα

  π.χ. + , a,b,t_1

  *, t_1,2,t_2

  :=,t_2,_,c

- Οι τετράδες είναι αριθμημένες. Κάθε τετράδα έχει μπροστά της έναν μοναδικό αριθμό που τη χαρακτηρίζει. Μόλις τελειώσει η εκτέλεση μίας τετράδας εκτελείται η τετράδα που έχει τον αμέσως μεγαλύτερο αριθμό, εκτός εάν η τετράδα που μόλις εκτελέστηκε υποδείξει κάτι διαφορετικό .

  π.χ.: 100: +,a,b,c

  110: +,d,e,f

Υλοποιήσαμε τις βοηθητικές υπορουτίνες (genquad() : για την δημιουργία μιας νέας τετράδας, newtemp() : για την δημιουργία νέας προσωρινής μεταβλητής, emptylist() : για την δημιουργία κενής λίστας , makelist(x) : δημιουργία λίστας που περιλαμβάνεις μόνο το όρισμα που δόθηκε , merge(list1,list2) : για την ένωση δύο λιστών, backpatch() : για την συμπλήρωση του τελευταίου πεδίου της τετράδας) και τις χρησιμοποιήσαμε μέσα στις δομές που είχαμε ήδη κατασκευάσει για τον συντακτικό αναλυτή. Για δική μας αλλά και δική σας διευκόλυνση, για κάθε νέα γραμμή που προσθέσαμε ή αλλάξαμε στον αρχικό κώδικα του συντακτικού αναλυτή για την υλοποίηση του ενδιάμεσου κώδικα, έχουμε προσθέσει από δίπλα το σχόλιο « # intermediate code ».

Το κείμενο από τετράδες που παράγουμε το τυπώνουμε σε ένα αρχείο κειμένου με όνομα «test.int»

Στην συνέχεια στην c_code() μεταφράσαμε το πρόγραμμα σε c και το τρέξαμε ώστε να ελέγξουμε ότι μέχρι τώρα λειτουργούν σωστά οι δομές που φτιάξαμε.(test.c)

Πίνακας Συμβόλων:

Ο πίνακας συμβόλων αποτελεί μέρος της σημασιολογικής ανάλυσης. Κρατάει πληροφορίες για όλες τις οντότητες (μεταβλητές, συναρτήσεις και διεργασίες) του κώδικα σχετικά με το βάθος στο οποίο βρίσκονται. Μέσα από τον πίνακα συμβόλων μπορούμε να αντλήσουμε πληροφορίες σχετικά με την εμβέλεια της κάθε οντότητας (πχ αν είναι τοπική μία μεταβλητή, global ή αν προέρχεται από όρισμα και ανήκει αλλού).

Για την υλοποίηση του πίνακα συμβόλων κατασκευάσαμε κλάσεις/αντικείμενα και τους ορίσαμε πεδία.

class Scope : Ορίζει ένα νέο επίπεδο. Για πεδία έχουμε το βάθος στο οποίο βρίσκεται (level), τις εγγραφές που περιέχονται (entities) και το offset του.

class Entity : Ορίζει μία εγγραφή.

Τα παρακάτω αντικείμενα αποτελούν τα διάφορα είδη εγγραφών:

class Variable : Ορίζει μία εγγραφή τύπου variable (type), για την οποία κρατάμε ακόμα το όνομά της (name), τον τύπο δεδομένων της (datatype) και το offset της.

class FormalParameter : Ορίζει μία εγγραφή τύπου Formal Parameter (type), για την οποία κρατάμε ακόμα το όνομά της (name), τον τύπο δεδομένων της (datatype) και το mode της.

class Procedure : Ορίζει μία εγγραφή τύπου Procedure (type), για την οποία κρατάμε ακόμα το όνομά της (name), την αρχική τετράδα της (starting quad), το μέγεθος μνήμης που χρειάζεται για να αποθηκεύσει το αποτέλεσμά της (frame length) καθώς και τις παραμέτρους της (formal parameters).

class TemporaryVariable : Ορίζει μία εγγραφή τύπου temporary variable (type), για την οποία κρατάμε ακόμα το όνομά της (name), τον τύπο δεδομένων της (datatype) και το offset της.

class Parameter : Ορίζει μία εγγραφή τύπου parameter (type), για την οποία κρατάμε ακόμα το όνομά της (name), τον τύπο δεδομένων της (datatype) ,το offset της και το mode της.

class Function : Ορίζει μία εγγραφή τύπου Function (type), για την οποία κρατάμε ακόμα το όνομά της (name), την αρχική τετράδα της (starting quad), το μέγεθος μνήμης που χρειάζεται για να αποθηκεύσει το αποτέλεσμά της (frame length) καθώς και τις παραμέτρους της (formal parameters).

Στην συνέχεια υλοποιήσαμε τις βοηθητικές συναρτήσεις (addScope() : για την προσθήκη νέου επιπέδου, deleteScope() : για την διαγραφή επιπέδου, addEntity() : για την προσθήκη εγγραφής μία οντότητας, startingQuad() : για την συμπλήρωσή του πεδίου που κρατά την πρώτη τετράδα σε μία συνάρτησης ή διαδικασία, frameLength() : για την συμπλήρωση του πεδίου μίας συνάρτησης ή διαδικασίας που κρατά την μνήμη που χρειάζεται για το return,

addFormalParameter() : για την προσθήκη παραμέτρων σε μία συνάρτηση ή διαδικασία, searchEntity() : για την αναζήτηση εγγραφής μέσα στον πίνακα συμβόλων) και μια global στοίβα την «scopes», όπου εκεί κρατάμε τα επίπεδα, όπου κάθε επίπεδο περιλαμβάνει έναν πίνακα με τις οντότητες που περιέχονται σε αυτό. Στην symbolTable() παράγουμε το αρχείο «test.symb» όπου αποθηκεύει τον πίνακα συμβόλων δυναμικά. Την μέθοδο αυτήν την καλούμε στο τέλος της δημιουργίας του κάθε block , διαβάζουμε το περιεχόμενο της στοίβας scopes εκείνη την χρονική στιγμή και συμπληρώνουμε το αρχείο του πίνακα συμβόλων με τα δεδομένα της. Δίπλα από κάθε νέα γραμμή που προσθέσαμε, έχουμε συμπληρώσει το σχόλιο «# symbol table»

Τέλος, με την μέθοδο searchEntity() κάνουμε σημασιολογικό έλεγχο. Στην ουσία ελέγχουμε αν η μεταβλητή που δίνουμε έχει δηλωθεί κάπου πιο πριν. (# semantic analyst)

Τελικός Κώδικας:

Στη φάση του τελικού κώδικα γίνεται η διαδικασία μετατροπής του κώδικα της cimple σε γλώσσα assembly, χρησιμοποιώντας τις δομές που υλοποιήσαμε για τον ενδιάμεσο κώδικα και τον πίνακα συμβόλων.

Για τον τελικό κώδικα υλοποιήσαμε τις βοηθητικές συναρτήσεις που ζητήθηκαν (gnlvcode(v), : η οποία δίνει πρόσβαση για μία μεταβλητή ή διεύθυνση η οποία δεν ανήκει τοπικά, loadvr(v, reg) : η οποία βρίσκει μια μεταβλητή από την μνήμη και την αποθηκεύει σε έναν καταχωρητή, storerv(reg, v) : διαβάζει την μεταβλητή από έναν καταχωρητή και την αποθηκεύει στην μνήμη, produce() : δημιουργεί μία νέα γραμμή στον τελικό κώδικα) + την findVariable() η οποία αναζητεί μία εγγραφή με το όνομά της στον πίνακα συμβόλων (scopes) και επιστρέφει έναν πίνακα όπου στην πρώτη θέση περιέχει την εγγραφή αυτή και στην δεύτερη το επίπεδο στο οποίο βρίσκεται.

Τέλος στην final_code() συμπληρώνουμε πάλι δυναμικά τον τελικό κώδικα γράφοντάς τον στο αρχείο «test.asm». Η μέθοδος αυτή καλείται στο τέλος κάθε block.

Παρακάτω διαθέτουμε μερικά παραδείγματα που έχουμε δοκιμάσει καθώς και τις λύσεις που τύπωσε το πρόγραμμά μας. (Το test.c δημιουργείται μόνο για προγράμματα που δεν περιέχουν functions ή procedures. Ο έλεγχος αυτός γίνεται με μία global μεταβλητή που λειτουργεί ως flag και ενεργοποιείται μέσα στην subprograms().)

Στο τέλος δίνουμε και τους κώδικες σε περίπτωση που θέλετε να τους τρέξετε και ο ίδιος.

(Στην δεύτερη φάση είχαμε παραδώσει και ένα txt αρχείο που περιείχε και εκείνο κώδικες που τρέξαμε και δοκιμάσαμε για να τεστάρουμε τον κώδικά μας)

Test 1

test.ci

```
program factorial
{
# declarations #
declare x;
declare i,fact;
# main #
input(x);
fact:=1;
i:=1;
while (i<=x)
{
fact:=fact*i;
i:=i+1;
};
print(fact);
}.
```

test.int

```
1: begin_block, factorial, _, _
2: inp, x, _, _
3: :=, 1, _, fact
4: :=, 1, _, i
5: <=, i, x, 7
6: jump, _, _, 12
7: *, fact, i, T_0
8: :=, T_0, _, fact
9: +, i, 1, T_1
10: :=, T_1, _, i
11: jump, _, _, 5
12: out, fact, _, _
13: halt, _, _, _
14: endblock, factorial, _, _
```

text.symb

```
Scope: 0
Entities:
[Variable, name: x, datatype: Integer, offset: 12]
[Variable, name: i, datatype: Integer, offset: 16]
[Variable, name: fact, datatype: Integer, offset: 20]
[TemporaryVariable, name: T_0, datatype: Integer, offset: 24]
[TemporaryVariable, name: T_1, datatype: Integer, offset: 28]
```

text.asm

```
    .data
str_ln: .asciz '\n'
    .text

L0:
    j Lmain
L1:
Lmain:
    addi sp, sp, 32
    move gp, sp
L2:
    li a7, 5
    ecall
    mv t1, a0
    sw t1, -12(sp)
L3:
    li t1, 1
    sw t1, -20(sp)
L4:
    li t1, 1
    sw t1, -16(sp)
L5:
    lw t1, -16(sp)
    lw t2, -12(sp)
    ble, t1, t2, 7
L6:
    j L12
L7:
    lw t1, -20(sp)
    lw t2, -16(sp)
    mul t1, t1, t2
    sw t1, -24(sp)
L8:
    lw t1, -24(sp)
    sw t1, -20(sp)
L9:
    lw t1, -16(sp)
    li t2, 1
    add t1, t1, t2
    sw t1, -28(sp)
L10:
    lw t1, -28(sp)
    sw t1, -16(sp)
L11:
    j L5
L12:
    lw t1, -20(sp)
    mv a0, t1
    li a7, 1
    ecall
    la a0, str_ln
    li a7, 4
    ecall
L13:
L14:
    li a0, 0
    li a7, 93
    ecall
```

test.c

```c
#include <stdio.h>

int main()
{
    int x, fact, i, T_0, T_1;
    L_1:
    L_2: scanf("%d", &x);
    L_3: fact = 1;
    L_4: i = 1;
    L_5: if(i <= x) goto L_7;
    L_6: goto L_12;
    L_7: T_0 = fact * i;
    L_8: fact = T_0;
    L_9: T_1 = i + 1;
    L_10: i = T_1;
    L_11: goto L_5;
    L_12: printf("fact =  %d", fact);
    L_13: {}
}
```

```
eftychia@EftychosLaptop:/mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh$ gcc -o factorial test.c
eftychia@EftychosLaptop:/mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh$ ./factorial
4
fact =  24eftychia@EftychosLaptop:/mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh$ ./factorial
2
fact =  2eftychia@EftychosLaptop:/mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh$
```

Εδώ βλέπουμε ότι το πρόγραμμα factorial τρέχει και υπολογίζει σωστά το x! του αριθμού x που δίνουμε ως input.

Test 2

test.ci

```
program MAX
{
declare a,b,c,d,e,z,w;

function max(in x, in y)
{
    if (x>y)
    {
        return(x);
    }
    else
    {
        return(y);
    }
}
    e:=max(in max(in a, in b), in max(in c, in d));
}.
```

test.symb

```
Scope: 1
Entities:
[Parameter, name: x, datatype: Integer, mode: in, offset: 12]
[Parameter, name: y, datatype: Integer, mode: in, offset: 16]

Scope: 0
Entities:
[Variable, name: a, datatype: Integer, offset: 12]
[Variable, name: b, datatype: Integer, offset: 16]
[Variable, name: c, datatype: Integer, offset: 20]
[Variable, name: d, datatype: Integer, offset: 24]
[Variable, name: e, datatype: Integer, offset: 28]
[Variable, name: z, datatype: Integer, offset: 32]
[Variable, name: w, datatype: Integer, offset: 36]
[Function, name: max, startingQuad: 2, framelength: 20]
Formal parameters modes : in in
[TemporaryVariable, name: T_0, datatype: Integer, offset: 40]
[TemporaryVariable, name: T_1, datatype: Integer, offset: 44]
[TemporaryVariable, name: T_2, datatype: Integer, offset: 48]
```

test.int

```
1: begin_block, max, _, _
2: >, x, y, 4
3: jump, _, _, 6
4: retv, x, _, _
5: jump, _, _, 7
6: retv, y, _, _
7: endblock, max, _, _
8: begin_block, MAX, _, _
9: par, a, CV, _
10: par, b, CV, _
11: par, T_0, RET, _
12: call, max, _, _
13: par, c, CV, _
14: par, d, CV, _
15: par, T_1, RET, _
16: call, max, _, _
17: par, T_0, CV, _
18: par, T_1, CV, _
19: par, T_2, RET, _
20: call, max, _, _
21: :=, T_2, _, e
22: halt, _, _, _
23: endblock, MAX, _, _
```

text.asm

```
L0:
    j Lmain
L1:
    sw ra, -0(sp)
L2:
    lw t1, -12(sp)
    lw t2, -16(sp)
    bgt, t1, t2, 4
L3:
    j L6
L4:
    lw t1, -12(sp)
    lw t0, -8(sp)
    sw t1, (t0)
L5:
    j L7
L6:
    lw t1, -16(sp)
    lw t0, -8(sp)
    sw t1, (t0)
L7:
    lw ra, -0(sp)
    jr ra
L8:
Lmain:
    addi sp, sp, 52
    move gp, sp
L9:
    addi fp, sp, 20
    lw t1, -12(sp)
    sw t1, -12(fp)
L10:
    lw t1, -16(sp)
    sw t1, -16(fp)
L11:
    addi t0, sp, -40
    sw t0, -8(fp)
L12:
    sw sp, -4(fp)
    addi sp, sp, 20
    jal L1
    addi sp, sp, -20
L13:
    addi fp, sp, 20
    lw t1, -20(sp)
    sw t1, -20(fp)
L14:
    lw t1, -24(sp)
    sw t1, -24(fp)
L15:
    addi t0, sp, -44
    sw t0, -8(fp)
L16:
    sw sp, -4(fp)
    addi sp, sp, 20
    jal L1
    addi sp, sp, -20
L17:
    addi fp, sp, 20
    lw t1, -40(sp)
    sw t1, -40(fp)
L18:
    lw t1, -44(sp)
    sw t1, -44(fp)
L19:
    addi t0, sp, -48
    sw t0, -8(fp)
L20:
    sw sp, -4(fp)
    addi sp, sp, 20
    jal L1
    addi sp, sp, -20
L21:
    lw t1, -48(sp)
    sw t1, -28(sp)
L22:
L23:
    li a0, 0
    li a7, 93
    ecall
```

Test 3

test.ci

```
program fibonacci
{
    declare x;
    function fibonacci (in x)
    {
        return (fibonacci(in x-1)+fibonacci(in x-2));
    }

# main #
input(x);
print(fibonacci(in x));
}.
```

test.symb

```
Scope: 1
Entities:
[Parameter, name: x, datatype: Integer, mode: in, offset: 12]
[TemporaryVariable, name: T_0, datatype: Integer, offset: 16]
[TemporaryVariable, name: T_1, datatype: Integer, offset: 20]
[TemporaryVariable, name: T_2, datatype: Integer, offset: 24]
[TemporaryVariable, name: T_3, datatype: Integer, offset: 28]
[TemporaryVariable, name: T_4, datatype: Integer, offset: 32]

Scope: 0
Entities:
[Variable, name: x, datatype: Integer, offset: 12]
[Function, name: fibonacci, startingQuad: 0, framelength: 36]
Formal parameters modes : in
[TemporaryVariable, name: T_5, datatype: Integer, offset: 16]
```

test.int

```
1: begin_block, fibonacci, _, _
2: -, x, 1, T_0
3: par, T_0, CV, _
4: par, T_1, RET, _
5: call, fibonacci, _, _
6: -, x, 2, T_2
7: par, T_2, CV, _
8: par, T_3, RET, _
9: call, fibonacci, _, _
10: +, T_1, T_3, T_4
11: retv, T_4, _, _
12: halt, _, _, _
13: endblock, fibonacci, _, _
14: begin_block, fibonacci, _, _
15: inp, x, _, _
16: par, x, CV, _
17: par, T_5, RET, _
18: call, fibonacci, _, _
19: out, T_5, _, _
20: halt, _, _, _
21: endblock, fibonacci, _, _
```

text.asm

```
    .data
str_ln: .asciz '\n'
    .text

L0:
    j Lmain
L1:
Lmain:
    addi sp, sp, 16
    move gp, sp
L2:
    lw t1, -12(sp)
    li t2, 1
    sub t1, t1, t2
    sw t1, -16(sp)
L3:
    addi fp, sp, 36
    lw t1, -16(sp)
    sw t1, -16(fp)
L4:
    addi t0, sp, -20
    sw t0, -8(fp)
L5:
    sw sp, -4(fp)
    addi sp, sp, 36
    jal L-1
    addi sp, sp, -36
L6:
    lw t1, -12(sp)
    li t2, 2
    sub t1, t1, t2
    sw t1, -24(sp)
L7:
    addi fp, sp, 36
    lw t1, -24(sp)
    sw t1, -24(fp)
```

text.asm

```
    lw t1, -24(sp)
    sw t1, -24(fp)
L8:
    addi t0, sp, -28
    sw t0, -8(fp)
L9:
    sw sp, -4(fp)
    addi sp, sp, 36
    jal L-1
    addi sp, sp, -36
L10:
    lw t1, -20(sp)
    lw t2, -28(sp)
    add t1, t1, t2
    sw t1, -32(sp)
L11:
    lw t1, -32(sp)
    lw t0, -8(sp)
    sw t1, (t0)
L12:
L13:
    li a0, 0
    li a7, 93
    ecall
L14:
Lmain:
    addi sp, sp, 20
    move gp, sp
L15:
    li a7, 5
    ecall
    mv t1, a0
    sw t1, -12(sp)
L16:
    addi fp, sp, 36
    lw t1, -12(sp)
    sw t1, -12(fp)
```

```
    sw t1, -12(fp)
L17:
    addi t0, sp, -16
    sw t0, -8(fp)
L18:
    sw sp, -4(fp)
    addi sp, sp, 36
    jal L-1
    addi sp, sp, -36
L19:
    lw t1, -16(sp)
    mv a0, t1
    li a7, 1
    ecall
    la a0, str_ln
    li a7, 4
    ecall
L20:
L21:
    li a0, 0
    li a7, 93
    ecall
```

Test 4

test.ci

```
program countDigits
{
declare x, count;
# main #
input(x);
count := 0;
while (x>0)
{
x := x/10;
count := count+1;
};
print(count);
}.
```

test.int

```
1: begin_block, countDigits, _, _
2: inp, x, _, _
3: :=, 0, _, count
4: >, x, 0, 6
5: jump, _, _, 11
6: /, x, 10, T_0
7: :=, T_0, _, x
8: +, count, 1, T_1
9: :=, T_1, _, count
10: jump, _, _, 4
11: out, count, _, _
12: halt, _, _, _
13: endblock, countDigits, _, _
```

test.symb

```
Scope: 0
Entities:
[Variable, name: x, datatype: Integer, offset: 12]
[Variable, name: count, datatype: Integer, offset: 16]
[TemporaryVariable, name: T_0, datatype: Integer, offset: 20]
[TemporaryVariable, name: T_1, datatype: Integer, offset: 24]
```

test.c

```
#include <stdio.h>

int main()
{
    int x, count, T_0, T_1;
    L_1:
    L_2: scanf("%d", &x);
    L_3: count = 0;
    L_4: if(x > 0) goto L_6;
    L_5: goto L_11;
    L_6: T_0 = x / 10;
    L_7: x = T_0;
    L_8: T_1 = count + 1;
    L_9: count = T_1;
    L_10: goto L_4;
    L_11: printf("count =  %d", count);
    L_12: {}
}
```

text.asm

```
    .data
str_ln: .asciz '\n'
    .text

L0:
    j Lmain
L1:
Lmain:
    addi sp, sp, 28
    move gp, sp
L2:
    li a7, 5
    ecall
    mv t1, a0
    sw t1, -12(sp)
L3:
    li t1, 0
    sw t1, -16(sp)
L4:
    lw t1, -12(sp)
    li t2, 0
    bgt, t1, t2, 6
L5:
    j L11
L6:
    lw t1, -12(sp)
    li t2, 10
    div t1, t1, t2
    sw t1, -20(sp)
L7:
    lw t1, -20(sp)
    sw t1, -12(sp)
L8:
    lw t1, -16(sp)
    li t2, 1
    add t1, t1, t2
    sw t1, -24(sp)
L9:
    lw t1, -24(sp)
    sw t1, -16(sp)
L10:
    j L4
L11:
    lw t1, -16(sp)
    mv a0, t1
    li a7, 1
    ecall
    la a0, str_ln
    li a7, 4
    ecall
L12:
L13:
    li a0, 0
    li a7, 93
    ecall
```

```
eftychia@EftychosLaptop: /mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh
eftychia@EftychosLaptop:/mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh$ gcc -o test test.c
eftychia@EftychosLaptop:/mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh$ ./test
34
count =  2eftychia@EftychosLaptop:/mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh$ ./test
123456
count =  6eftychia@EftychosLaptop:/mnt/c/Users/bourl/OneDrive/Desktop/MetafAskhsh$
```

Εδώ βλέπουμε ότι το πρόγραμμα count digit τρέχει και υπολογίζει σωστά το πλήθος των αριθμών που δίνουμε ως input.

Test 5

test.ci

```
program random1
{
    declare a,b,c;
    function f(in a, inout b)
    {
        b := a+1;
        c := 4;
        return(b);
    }
    a := 1;
    c := f(in a, inout b);
    print(c);
    print(b);
}.
```

test.symb

```
Scope: 1
Entities:
[Parameter, name: a, datatype: Integer, mode: in, offset: 12]
[Parameter, name: b, datatype: Integer, mode: inout, offset: 16]
[TemporaryVariable, name: T_0, datatype: Integer, offset: 20]

Scope: 0
Entities:
[Variable, name: a, datatype: Integer, offset: 12]
[Variable, name: b, datatype: Integer, offset: 16]
[Variable, name: c, datatype: Integer, offset: 20]
[Function, name: f, startingQuad: 2, framelength: 24]
Formal parameters modes : in in
[TemporaryVariable, name: T_1, datatype: Integer, offset: 24]
```

text.asm

```
    .data
str_ln: .asciz '\n'
    .text

L0:
    j Lmain
L1:
    sw ra, -0(sp)
L2:
    lw t1, -12(sp)
    li t2, 1
    add t1, t1, t2
    sw t1, -20(sp)
L3:
    lw t1, -20(sp)
    lw t0, -16(sp)
    sw t1, (t0)
L4:
    li t1, 4
    sw t1, -20(gp)
L5:
    lw t0, -16(sp)
    lw t1, (t0)
    lw t0, -8(sp)
    sw t1, (t0)
L6:
    lw ra, -0(sp)
    jr ra
L7:
Lmain:
    addi sp, sp, 28
    move gp, sp
L8:
    li t1, 1
    sw t1, -12(sp)
L9:
    addi fp, sp, 24
    lw t1, -12(sp)
    sw t1, -12(fp)
L10:
    addi t0, sp, -16
    sw t0, -16(fp)
L11:
    addi t0, sp, -24
    sw t0, -8(fp)
L12:
    sw sp, -4(fp)
    addi sp, sp, 24
    jal L1
    addi sp, sp, -24
L13:
    lw t1, -24(sp)
    sw t1, -20(sp)
L14:
    lw t1, -20(sp)
    mv a0, t1
    li a7, 1
    ecall
    la a0, str_ln
    li a7, 4
    ecall
L15:
    lw t1, -16(sp)
    mv a0, t1
    li a7, 1
    ecall
    la a0, str_ln
    li a7, 4
    ecall
L16:
L17:
    li a0, 0
    li a7, 93
    ecall
```

test.int

```
1: begin_block, f, _, _
2: +, a, 1, T_0
3: :=, T_0, _, b
4: :=, 4, _, c
5: retv, b, _, _
6: endblock, f, _, _
7: begin_block, random1, _, _
8: :=, 1, _, a
9: par, a, CV, _
10: par, b, REF, _
11: par, T_1, RET, _
12: call, f, _, _
13: :=, T_1, _, c
14: out, c, _, _
15: out, b, _, _
16: halt, _, _, _
17: endblock, random1, _, _
```

Κώδικες:

Test 1:

```
program factorial
{
# declarations #
declare x;
declare i,fact;
# main #
input(x);
fact:=1;
i:=1;
while (i<=x)
{
fact:=fact*i;
i:=i+1;
};
print(fact);
}.
```

Test 2:

```
program MAX
{
declare a,b,c,d,e,z,w;

function max(in x, in y)
{
   if (x>y)
   {
      return(x);
   }
   else
   {
      return(y);
   }
}
   e:=max(in max(in a, in b), in max(in c, in d));
}.
```

Test 3:

```
program fibonacci
{
   declare x;
   function fibonacci (in x)
   {
      return (fibonacci(in x-1)+fibonacci(in x-2));
   }

# main #
input(x);
print(fibonacci(in x));
}.
```

Test 4:

```
program countDigits
{
declare x, count;
# main #
input(x);
count := 0;
while (x>0)
{
x := x/10;
count := count+1;
};
print(count);
}.
```

Test 5:

```
program random1
{
	declare a,b,c;
	function f(in a, inout b)
	{
		b := a+1;
		c := 4;
		return(b);
	}
	a := 1;
	c := f(in a, inout b);
	print(c);
	print(b);
}.
```

Αποτελέσματα και Συμπεράσματα άσκησης:

Είχαμε την ευκαιρία να ασχοληθούμε με ένα πολύ ενδιαφέρον project. Οι διαφάνειες που μα δόθηκαν ήταν περιεκτικές και πολύ βοηθητικές. Ξεκινήσαμε χωρίς να έχουμε ιδέα πως μελετάμε μία γλώσσα και τι χρειάζεται για την μεταγλώττισή της, παρόλα αυτά μάθαμε τα βήματα που χρειάζονται για την μεταγλώττιση, καθώς και το πως να κωδικοποιούμε μία γλώσσα και να την μελετάμε σε τμήματα. Είμαστε περήφανοι που καταφέραμε να την υλοποιήσουμε και η ικανοποίηση που παίρνεις όταν τρέχεις κάτι που προγραμμάτισες και τρέχει σωστά είναι απερίγραπτη. Το σίγουρο είναι ότι το ευχαριστηθήκαμε και αν ποτέ στο μέλλον χρειαστεί να υλοποιήσουμε κάτι παρόμοιο θα ανατρέξουμε αμέσως στις διαφάνειές σας και τον κώδικά μας.