

# Προγραμματιστική Άσκηση στο μάθημα: **ΜΥΥ802 ΜΕΤΑΦΡΑΣΤΕΣ**

Πανεπιστήμιο Ιωαννίνων  
Πολυτεχνική Σχολή  
Τμήμα Μηχανικών Η/Υ και Πληροφορικής  
Διδάσκων: Γεώργιος Μανής  
Μάιος 2021



---

Ομάδα:

Tucaliuc Agnes Monalisa 3346

Χαρπαντίδου Ιωάννα 4199

# Περιεχόμενα

<b>1 Εισαγωγή.....</b>	<b>3</b>
1.1 Η Γλώσσα Προγραμματισμού Cimple .....	3
1.2 Το αλφάβητο της Cimple .....	3
<b>2 Στάδια υλοποίησης .....</b>	<b>5</b>
2.1 Λεκτική ανάλυση .....	5
2.2 Συντακτική ανάλυση .....	10
2.3 Ενδιάμεσος κώδικας .....	14
2.4 Πίνακας Συμβόλων .....	22
2.5 Τελικός κώδικας .....	24
<b>3 Παραδείγματα για ορθή λειτουργία του μεταφραστή .....</b>	<b>31</b>
3.1 Λειτουργία μεταφραστή με το αρχείο fact.ci . .....	31
3.2 Λειτουργία μεταφραστή με το αρχείο ex3.ci .....	36

# 1 Εισαγωγή

## 1.1 Η Γλώσσα Προγραμματισμού Cimple

Η Cimple είναι μια μικρή, εκπαιδευτική, γλώσσα προγραμματισμού. Θυμίζει τη γλώσσα C, από την οποία και αντλεί ιδέες και δομές, αλλά είναι αρκετά πιο μικρή, τόσο στις υποστηριζόμενες δομές, όσο φυσικά και σε προγραμματιστικές δυνατότητες. Το όνομα Cimple προέρχεται από τη γλώσσα C, τη λέξη implementation, ενώ ηχεί όπως η λέξη simple. ώστε να τονιστεί η απλότητά της.

Ο μεταφραστής είναι ένα πρόγραμμα υπεύθυνο για την μετάφραση αρχείων γραμμένων σε γλώσσα Cimple (.ci), σε αρχεία γλώσσας assembly (.asm). Αυτή η διαδικασία επιτυγχάνεται με την ολοκλήρωση 6 σταδίων: την λεκτική ανάλυση, την συντακτική ανάλυση, την παραγωγή του ενδιαμέσου κώδικα, την κατασκευή του πίνακα συμβόλων, την σημασιολογική ανάλυση και τέλος την παραγωγή του τελικού κώδικα. Παρακάτω θα δούμε αναλυτικά το κάθε στάδιο καθώς και τα βήματα που ακολουθούμε για να το υλοποιήσουμε σε Python.

Για την εκτέλεση του προγράμματος Cimple χρησιμοποιείται η εξής εντολή: **python cimple.py filename.ci** όπου cimple.py είναι το αρχείο του πηγαίου κώδικα του μεταγλωττιστή της Cimple και filename.ci είναι το αρχείο σε γλώσσα προγραμματισμού Cimple. Κατά την εκτέλεση της παραπάνω εντολής δημιουργούνται ορισμένα αρχεία, στα οποία θα αναφερθούμε σε βάθος στην συνέχεια. Αυτά τα αρχεία είναι τα εξής: test.int, test.c, tableOfSymbols\_File.txt, test.asm.

## 1.2 Το αλφάβητο της Cimple

Η Cimple έχει το δικό της αλφάβητο, δηλαδή έναν σταθερό αριθμό από “δεσμευμένες” λέξεις, αλλά και σύμβολα (τελεστές), προκειμένου να μπορεί να εκτελεστεί σωστά η σύνταξη και οι λογικές εκφράσεις.

Έτσι λοιπόν, το αλφάβητο της Cimple ορίζεται ως εξής:

- Τα γράμματα του λατινικού αλφαβήτου (κεφαλαία και πεζά) : “A-Z” , “a-z”, τα οποία χρησιμοποιούνται για ό,τι πρόκειται να γραφεί στο πρόγραμμα μας. Επομένως, η Cimple απαγορεύει την χρήση οποιουδήποτε άλλου αλφαβήτου πέραν του αγγλικού.
- Τα αριθμητικά ψηφία “0-9” με τον περιορισμό ότι ο μέγιστος αριθμός που θα μπορούν να συνθέσουν αυτά τα ψηφία να είναι μεταξύ του συνόλου όλων των ακεραίων αριθμών από  $-94967295(-(2^{32}-1))$  μέχρι  $+94967295(2^{32}-1)$ .
- Τα αριθμητικά σύμβολα “ + ”, “ - ”, “ \* ”, “ / ” που χρησιμεύουν για αριθμητικές πράξεις
- Τους λογικούς τελεστές “and”, “not” και “or” για την εκτέλεση λογικών πράξεων.

Αξίζει να αναφερθεί ότι η προτεραιότητα των παραπάνω τελεστών είναι η εξής:

1. Τελεστής “not”
  2. Πολλαπλασιασμοί (“\*”), Διαιρέσεις (“/”)
  3. Προσθέσεις (“+”), Αφαιρέσεις (“-”)
  4. Λογικό “and”
  5. Λογικό “or”
- Τους τελεστές συσχέτισης “<”, “>”, “=”, “<=”, “>=”, “<>” (διάφορο), για να μπορούν να γίνουν οι συγκρίσεις μεταξύ των αριθμών και μεταβλητών.
  - Το σύμβολο ανάθεσης “:=”, το οποίο χρησιμοποιείται για να δώσουμε τιμή σε μία μεταβλητή (π.χ. variable := 2) . Δεν έχει καμία σχέση με το σύμβολο “=”, το οποίο χρησιμοποιείται αποκλειστικά για συγκρίσεις.
  - Τους διαχωριστές “;”, “ , ”, “ : ” . Το κόμμα για τον διαχωρισμό δηλώσεων π.χ., μεταβλητών, το ερωτηματικό για τον διαχωρισμό των declarations, subprograms και statements, ενώ η άνω-κάτω τελεία χρησιμοποιείται για το σύμβολο της ανάθεσης μαζί με το “:=”.
  - Τα σύμβολα ομαδοποίησης “[“ , “]”, “(“ , “)” , “{“, “}”. Οι αγκύλες, χρησιμοποιούνται στις λογικές παραστάσεις π.χ. με τον λογικό τελεστή

“not” (conditions). Οι παρενθέσεις χρησιμοποιούνται στις αριθμητικές παραστάσεις όπως στα expressions και lists και τα άγκιστρα για τα blocks.

- Το σύμβολο της “.”, το οποίο χρησιμοποιείται για τον τερματισμό του προγράμματος.
- Τα σύμβολα διαχωρισμού σχολίων “#”, χρησιμοποιείται για την δημιουργία σχολίου σε μία γραμμή ως εξής π.χ. “#this is a comment#”.
- Οι δεσμευμένες λέξεις program, declare, if, else, while, switcase, forcase, incase, case, default, not, and, or, function, procedure, call, return, in, inout, input, print. Οι λέξεις αυτές αποτελούν τις “εσωτερικές” λέξεις του αλφαβήτου οι οποίες μία προς μία εκτελούν μία διαφορετική λειτουργία.
- Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία, αρχίζοντας όμως από γράμμα. Κάθε αναγνωριστικό αποτελείται από τριάντα το πολύ γράμματα. Αναγνωριστικά με περισσότερους από 30 χαρακτήρες θεωρούνται λανθασμένα.
- Οι λευκοί χαρακτήρες όπως (tab, space, return) αγνοούνται και μπορούν να χρησιμοποιηθούν με οποιονδήποτε τρόπο χωρίς να επηρεάζεται η λειτουργία του μεταγλωττιστή, αρκεί βέβαια, να μην βρίσκονται μέσα σε δεσμευμένες λέξεις, αναγνωριστικά, σταθερές.

## 2 Στάδια υλοποίησης

### 2.1 Λεκτική Ανάλυση

Η λεκτική ανάλυση αποτελεί την πρώτη φάση της μεταγλώττισης. Κατά τη φάση αυτή, διαβάζεται το αρχικό πρόγραμμα (το οποίο συνηθίζεται να ονομάζεται και πηγαίο πρόγραμμα) και παράγονται οι λεκτικές μονάδες. Χρησιμοποιούμε τον όρο λεκτική μονάδα για να αναπαραστήσουμε οτιδήποτε έχει νόημα να θεωρηθεί ως αυτόνομο σύνολο συνεχόμενων χαρακτήρων που

μπορεί να συναντηθεί σε ένα πρόγραμμα και βρίσκει σημασιολογία στην υπό υλοποίηση γλώσσα.

Ο σκοπός του λεκτικού αναλυτή είναι να ελέγχει αν οι χαρακτήρες του προγράμματος που εισάγεται προς μετάφραση είναι αποδεκτοί από το αλφάβητο της γλώσσας, να κατανοεί την μορφή τους προκειμένου να τους χρησιμοποιεί ο συντακτικός αναλυτής, στον οποίο θα αναφερθούμε στην συνέχεια, και να απορρίπτει τα προγράμματα με χαρακτήρες μη αποδεκτούς στην γλώσσα.

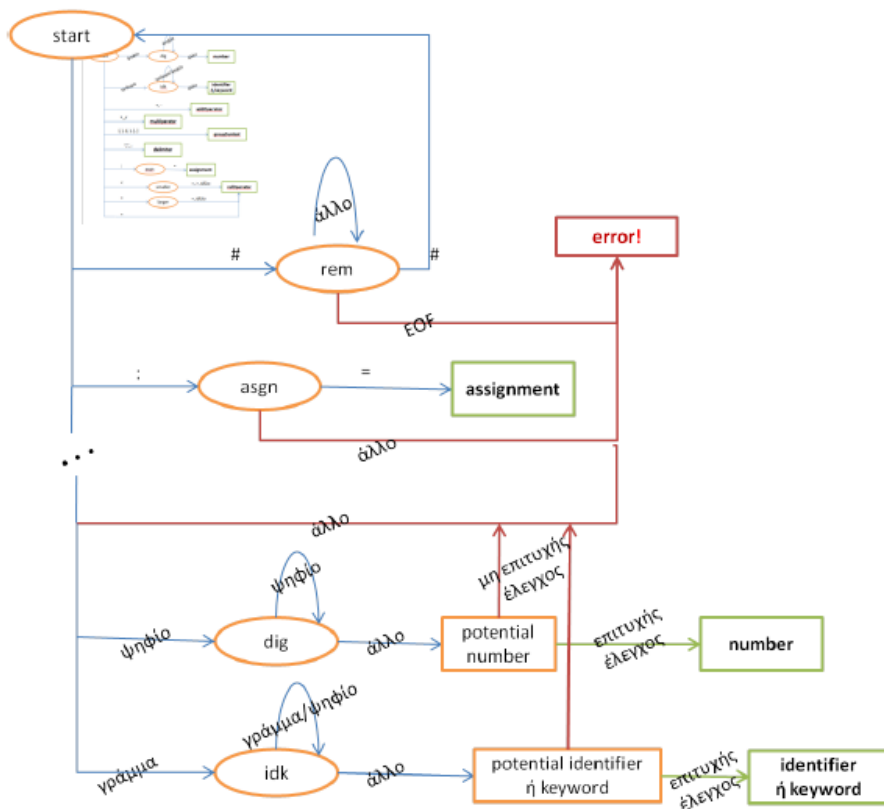
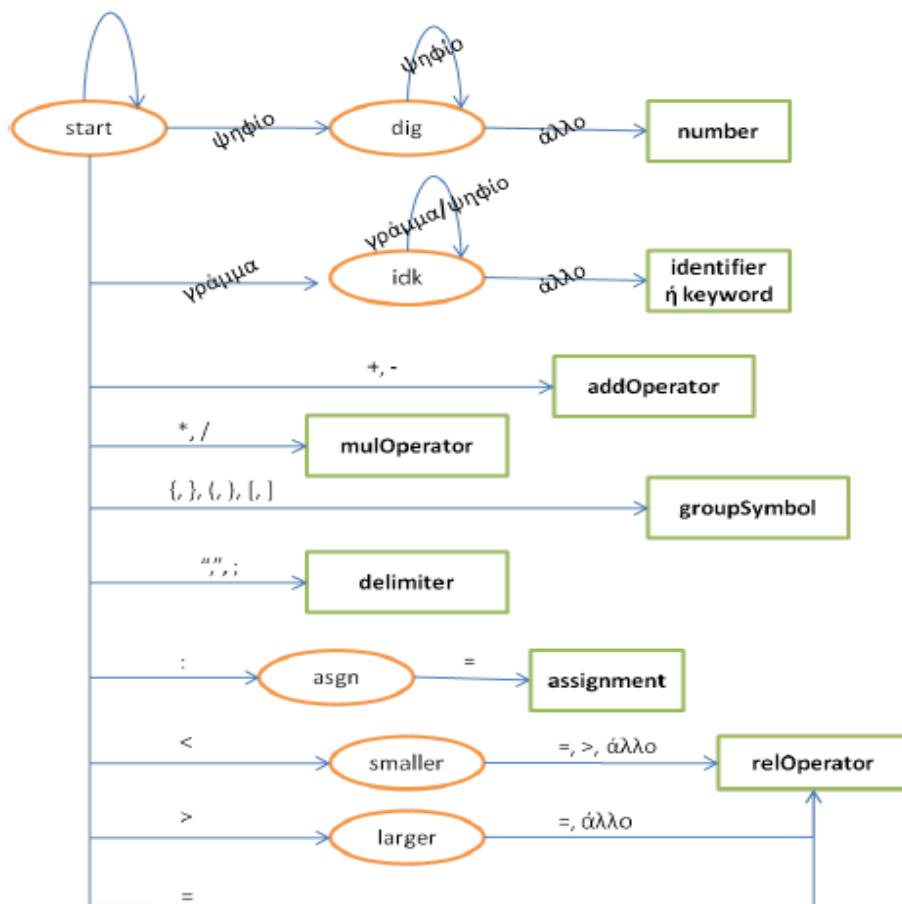
1. Διαβάζει μία-μία κάθε λέξη κλειδί και ανάλογα με το κλειδί αυτό πηγαίνει σε κάποια άλλη κατάσταση.
2. Καλείται ως συνάρτηση από το συντακτικό αναλυτή.
3. Κάθε φορά που καλείται επιστρέφει την επόμενη λεκτική μονάδα
4. Επιστρέφει στο συντακτικό αναλυτή
  - έναν ακέραιο που χαρακτηρίζει τη λεκτική μονάδα
  - τη λεκτική μονάδα

Ο λεκτικός αναλυτής εσωτερικά λειτουργεί σαν ένα αυτόματο καταστάσεων το οποίο ξεκινά από μία αρχική κατάσταση, με την είσοδο κάθε χαρακτήρα αλλάζει κατάσταση έως ότου συναντήσει μία τελική κατάσταση.

Το αυτόματο καταστάσεων, δηλαδή, αναγνωρίζει:

- δεσμευμένες λέξεις  
πχ. if, for, while
- σύμβολα της γλώσσας  
π.χ. «+», «;», «=»
- αναγνωριστικά και σταθερές  
π.χ. counter, a12, 32768
- λάθη  
π.χ. μη επιτρεπτός χαρακτήρας, κλείσιμο σχολίων χωρίς να έχουν ανοίξει προηγουμένως

Παρακάτω φαίνεται το αυτόματο μετάβασης καταστάσεων:



Ο Κώδικας μας για το αυτόματο είναι:

```
state_auto = [[0 for x in range(24)] for y in range(7)]

# STARTING FORM SCRATCH AND MOVING THE POINTER
# STATE TO STATE, DEPENDING ON WHAT WE FIND
state_auto[STATE0_START][WHITE_SPACE] = STATE0_START # FINDS EMPTY SPACE AND STAYS ON THE SAME STATE ,START STATE
state_auto[STATE0_START][LETTER] = STATE1_LETTER # FINDS LETTER AND GOES TO STATE1
state_auto[STATE0_START][DIGIT] = STATE2_DIGIT # FINDS DIGIT AND GOES TO STATE2
state_auto[STATE0_START][ADD] = plus_TK
state_auto[STATE0_START][SUB] = sub_TK
state_auto[STATE0_START][MUL] = multiply_TK
state_auto[STATE0_START][DIV] = divide_TK
state_auto[STATE0_START][LESSER_THAN] = STATE3 # WE CHANGE STATE CAUSE MIGHT FOLLOW = OR >
state_auto[STATE0_START][GREATER_THAN] = STATE4 # WE CHANGE STATE CAUSE MIGHT FOLLOW =
state_auto[STATE0_START][EQUAL] = equal_TK
state_auto[STATE0_START][COLON] = STATE5_ASSIGN # WE CHANGE STATE CAUSE COULD BE FOLLOWED =
state_auto[STATE0_START][SEMI_COLON] = semicolon_TK
state_auto[STATE0_START][COMMA] = comma_TK
state_auto[STATE0_START][RIGHT_BRACKET] = right_bracket_TK # [
state_auto[STATE0_START][LEFT_BRACKET] = left_bracket_TK # ]
state_auto[STATE0_START][LEFT_PARENTHESSES] = left_parentheses_TK # (
state_auto[STATE0_START][RIGHT_PARENTHESSES] = right_parentheses_TK # )
state_auto[STATE0_START][LEFT_BRACE] = left_brace_TK # {
state_auto[STATE0_START][RIGHT_BRACE] = right_brace_TK # }
state_auto[STATE0_START][KAGKELO] = STATE6_COMMENT # WE CHANGE STATE FOR THE COMMENT
state_auto[STATE0_START][DOT] = dot_TK
state_auto[STATE0_START][EOF] = error_TK

for i in range(24): # STRING
    state_auto[STATE1_LETTER][i] = id_TK # WORD THAT CONTAIN LETTERS OR DIGITS
state_auto[STATE1_LETTER][LETTER] = STATE1_LETTER # FOR WORD
state_auto[STATE1_LETTER][DIGIT] = STATE1_LETTER # IF WORD CONTAINS DIGIT

for i in range(24): # NUMBER
    state_auto[STATE2_DIGIT][i] = constant_TK # NUMBER
state_auto[STATE2_DIGIT][DIGIT] = STATE2_DIGIT # ONLY FOR DIGIT

for i in range(24): # OPERATORS
    state_auto[STATE3][i] = less_than_TK # LESS,LESS OR EQUAL, DIFFERENT
state_auto[STATE3][EQUAL] = less_equal_TK # LESS OR EQUAL <=
state_auto[STATE3][GREATER_THAN] = different_TK # DIFFERENT <>

for i in range(24): # OPERATORS
    state_auto[STATE4][i] = more_than_TK # >
state_auto[STATE4][EQUAL] = more_equal_TK # MORE OT EQUAL >=

for i in range(24): # ASSIGN
    state_auto[STATE5_ASSIGN][i] = colon_TK # COLON
state_auto[STATE5_ASSIGN][EQUAL] = assign_TK # := ASSIGH SYMBOL

for i in range(24): # COMMENT
    state_auto[STATE6_COMMENT][i] = STATE6_COMMENT # COMMENT IGNORE EVERYTHING INSIDE THE COMMENT
state_auto[STATE6_COMMENT][KAGKELO] = STATE0_START # END OF COMMENT #
state_auto[STATE6_COMMENT][NEWLINE] = ERROR_COMMENT_SYNTAX # THE COMMENT MUST END WITH # OTHERWISE ERROR
```



## Επεξήγηση κώδικα αυτομάτου:

- Κατάσταση 0 - STATE0\_START: Η κατάσταση αυτή είναι η αρχική κατάσταση. Μετα το τέλος κάθε άλλης κατάστασης, το αυτόματο μεταβαίνει πάλι στην 0. Από την 0 μπορεί να μεταβεί σε πολλές καταστάσεις, ανάλογα με το τί θα ακολουθήσει. Παραμένει στην ίδια κατάσταση μόνο αν ακολουθήσει λευκός χαρακτήρας (space, tab, newline).
- Κατάσταση 1 - STATE1\_LETTER: Αν βρεθεί γράμμα, μεταβαίνει στην κατάσταση 1. Παραμένει εκεί όσο βρίσκει γράμματα ή αριθμούς. Αν βρεθεί κάποια από τις δεσμευμένες λέξεις, ή κάποια άλλη λέξη ή κάποιο αλφαριθμητικό, σημαίνει ότι βρέθηκε μία λεκτική μονάδα και επιστρέφει στη κατάσταση 0.
- Κατάσταση 2 - STATE2\_DIGIT: Αν βρεθεί αριθμός, μεταβαίνει στην κατάσταση 2. Παραμένει εκεί όσο βρίσκει αριθμούς. Αν βρεθεί γράμμα μετά τον αριθμό εμφανίζεται μήνυμα λάθους διότι δεν επιτρέπεται να ακολουθεί γράμμα μετά από έναν αριθμό σε μία λεκτική μονάδα. Οπότε ως λεκτική μονάδα επιτρέπεται μόνο ένας αριθμός και ύστερα επιστρέφει στην κατάσταση 0.
- Κατάσταση 3 - STATE3: Αν βρεθεί ο χαρακτήρας "<" το αυτόματο μεταβαίνει στην κατάσταση 3.
  1. Αν δεν ακολουθήσει κάτι άλλο, σημαίνει ότι βρέθηκε ο χαρακτήρας "<" ως λεκτική μονάδα και το αυτόματο μεταβαίνει ομοίως στην κατάσταση 0.
  2. Αν στην κατάσταση αυτή ακολουθήσει ο χαρακτήρας "=" τότε σημαίνει ότι βρέθηκε ο χαρακτήρας "<=" ως λεκτική μονάδα και το αυτόματο μεταβαίνει στην κατάσταση 0.
  3. Αν ακολουθήσει ο χαρακτήρας ">" τότε σημαίνει ότι βρέθηκε ο χαρακτήρας "<>" ως λεκτική μονάδα και το αυτόματο μεταβαίνει στην κατάσταση 0.
- Κατάσταση 4 - STATE4: Αν βρεθεί ο χαρακτήρας ">" το αυτόματο μεταβαίνει στην κατάσταση 4.
  1. Αν στην κατάσταση αυτή ακολουθήσει ο χαρακτήρας "=" τότε σημαίνει ότι βρέθηκε ο χαρακτήρας ">=" ως λεκτική μονάδα και το αυτόματο μεταβαίνει στην κατάσταση 0.

2. Αλλιώς αν ακολουθήσει οτιδήποτε άλλο, σημαίνει ότι βρέθηκε ο χαρακτήρας ">" ως λεκτική μονάδα και το αυτόματο μεταβαίνει ομοίως στην κατάσταση 0.

- Κατάσταση 5 - STATE5\_ASSIGN: Αν βρεθεί ο χαρακτήρας ":" το αυτόματο μεταβαίνει στην κατάσταση 5. Αν στην κατάσταση αυτή ακολουθήσει ο χαρακτήρας "=" τότε σημαίνει ότι βρέθηκε ο χαρακτήρας "!=" ως λεκτική μονάδα και το αυτόματο μεταβαίνει στην κατάσταση 0.
- Κατάσταση 6 - STATE6\_COMMENT: Αν βρεθεί ο χαρακτήρας "#" το αυτόματο μεταβαίνει στην κατάσταση. Αγνοεί οτιδήποτε ακολουθεί μέχρι να βρεί newline.
  1. Αν πριν το newline δεν βρει άλλο "#" τότε επιστρέφει error.
  2. Αν βρει "#" τότε μεταβαίνει στην κατάσταση 0.

## 2.2 Συντακτική ανάλυση

Η συντακτική ανάλυση αποτελεί την δεύτερη φάση της μεταγλώττισης. Ο σκοπός του συντακτικού αναλυτή είναι να ελέγχει κάθε επόμενη λέξη αν είναι αποδεκτή βάσει της γραμματικής της γλώσσας. Ο συντακτικός αναλυτής αποτελείται από ένα σύνολο συναρτήσεων. Κάθε συνάρτηση αντιστοιχεί σε έναν από τους κανόνες της γραμματικής. Άρα προκειμένου να υλοποιήσουμε τον συντακτικό αναλυτή πρέπει πρώτα να φτιάξουμε την γραμματική της Cimple . Παρακάτω βλέπουμε το σύνολο των κανόνων της γραμματικής:

# "program" is the starting symbol  
program : **program** ID block .

# a block with declarations, subprograms and statements  
block : declarations subprograms statements

# declaration of variables, zero or more "declare" allowed  
declarations : ( **declare** varlist ; )\*

# a list of variables following the declaration keyword  
varlist : ID ( , ID)\* | ε

#zero or more subprograms allowed

subprograms : (subprograms)\*

# a subprogram is a function or a procedure followed by parameters and block

subprogram : **function** ID (formalparlist) block | **procedure** ID (formalparlist) block

# list of formal parameters

formalparlist : formalparitem ( . formalparitem)\* |  $\epsilon$

# a formal parameter ("in": by value, "inout" by reference)

formalparitem : **in** ID | **inout** ID

# one or more statements

statements : statement ; | { statement ( ; statement )\* }

# one statement

statement : assignStat | ifStat | whileStat | switchcaseStat | forcaseStat | incaseStat | callStat | returnStat | inputStat | printStat |  $\epsilon$

# assignment statement

assignStat : ID := expression

# if statement

ifStat : **if** ( condition ) statements elsepart

elsepart : **else** statements |  $\epsilon$

# while statement

while : **while** ( condition ) statements

# switch statement

switchcaseSat : **switchcase**  
                  ( **case** ( condition ) statements )\*  
                  **default** statements

# forcase statement

forcaseStat : **forcase**

(**case** ( condition ) statements )\*

**default** statements

# incase statement

incaseStat : **incase**

( **case** ( condition ) statements ) \*

# return statement

returnStat : **return** ( expression )

# call statement

callStat : **call** ID ( actualparlist )

# print statement

printStats : **print** ( expression )

# input statement

inputStat : **input** ( ID )

# list of actual parameters

actualparlist : actualparitem ( , actualparitem )\* |  $\epsilon$

# an actual parameter ("in" by value, "inout" by reference)

actualparitem : **in** expression | **inout** ID

# boolean expression

condition : boolean ( **or** boolterm )\*

# term in boolean expression

boolterm : boolfactor ( **and** boolfactor )\*

# factor in boolean expression

boolfactor : **not** [ condition ] | [ condition ] | expression | = | <= | >= | > | < | <>  
| ; expression

# arithmetic expression

expression : optionalSign term ( + | - term)\*

# term in arithmetic expression

term : factor ( \* | / factor ) \*

# factor in arithmetic expression

factor : [0-9]\* | ( expression ) | [a-zA-Z][a-zA-Z0-9]\* idtail

# follows a function or procedure ( parenthesis and parameters )

idtail : ( actualparlist ) | ε

# symbols “+” and “-” (are optional)

optionalSign : + | - | ε

Έτσι η λειτουργία του συντακτικού αναλυτή περιγράφεται από τα ακόλουθα βήματα:

- Όταν συναντά μη τερματικό σύμβολο καλεί την αντίστοιχη συνάρτηση.
- Όταν συναντάει τερματικό σύμβολο τότε:
  - εάν και ο λεκτικός αναλυτής επιστρέφει λεκτική μονάδα που αντιστοιχεί στο τερματικό αυτό σύμβολο έχουμε αναγνωρίσει επιτυχώς τη λεκτική μονάδα
  - αντίθετα εάν ο λεκτικός αναλυτής δεν επιστρέψει τη λεκτική μονάδα που περιμένει ο συντακτικός αναλυτής, έχουμε λάθος και καλείται ο διαχειριστής σφαλμάτων.
- Όταν τελικά βρίσκει την “.” στο τέλος του αρχείου τότε η συντακτική ανάλυση έχει τελειώσει με επιτυχία και δεν αναγνωρίζει τίποτα πέρα από αυτό, για αυτό πρέπει να προσέξουμε που θα βάλουμε την τελεία για να μην αγνοηθεί ο κώδικας πέρα από αυτήν.

## 2.3 Ενδιάμεσος κώδικας

Ο ενδιάμεσος κώδικας είναι ένα σύνολο από τετράδες (quads). Κάθε τετράδα αποτελείται από έναν τελεστή και τρία τελούμενα. Οι τετράδες είναι αριθμημένες. Κάθε τετράδα έχει μπροστά της έναν μοναδικό αριθμό που τη χαρακτηρίζει. Μόλις τελειώσει η εκτέλεση μιας τετράδας εκτελείται η τετράδα που έχει τον αμέσως μεγαλύτερο αριθμό, εκτός εάν η τετράδα που μόλις εκτελέστηκε υποδείξει κάτι διαφορετικό .

Οι τετράδες είναι της μορφής:

op, x, y, z

- όπου το op να είναι ένα εκ των: +, -, \*, /
- τα τελούμενα x, y μπορεί να είναι:
  - ονόματα μεταβλητών
  - αριθμητικές σταθερές
- το τελούμενο z μπορεί να είναι:
  - όνομα μεταβλητής
- εφαρμόζεται ο τελεστής op στα τελούμενα x και y και το αποτέλεσμα τοποθετείται στο τελούμενο z

Η αρχή και το τέλος του προγράμματος και του υποπρογράμματος ορίζονται ως εξής:

- begin\_block, name, \_\_, \_\_  
αρχή υποπρογράμματος ή προγράμματος με το όνομα name
- end\_block, name, \_\_, \_\_  
τέλος υποπρογράμματος ή προγράμματος με το όνομα name
- halt, \_\_, \_\_, \_\_  
τερματισμός προγράμματος

Οι συναρτήσεις και οι διαδικασίες εμπεριέχουν:

- par, x, m, \_\_  
όπου x παράμετρος συνάρτησης και m ο τρόπος μετάδοσης  
CV : μετάδοση με τιμή  
REF: μετάδοση με αναφορά  
RET: επιστροφή τιμής συνάρτησης
- call, name, \_\_, \_\_  
κλήση συνάρτησης name
- ret, x, \_\_, \_\_  
επιστροφή τιμής συνάρτησης

Οι global μεταβλητές και οι βοηθητικές συναρτήσεις για τον ενδιάμεσο κώδικα βάσει των οδηγιών καθηγητή είναι οι εξής:

global μεταβλητές :

quad\_list = [] , η λίστα με τις τετράδες

quad\_counter = 1 , μετρητή για τις τετράδες

keepers = [] , λίστα που αποθηκεύουμε τις προσωρινές μεταβλητές

T\_1,...,T\_x, x = 1,2,..x

Βοηθητικές συναρτήσεις:

- `nextquad()` : επιστρέφει τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί

```
def nextquad(): # returns the number of the next created quad
    global quad_counter
    return quad_counter
```

- `genquad(op, x, y, z)` : δημιουργεί την επόμενη τετράδα (op, x, y, z)

```
def genquad(op, x, y, z): # creates the next quad
    global quad_list
    global quad_counter

    gen_list = [nextquad()]
    gen_list.append(op)
    gen_list.append(x)
    gen_list.append(y)
    gen_list.append(z)

    quad_counter += 1
    quad_list += [gen_list]
    return quad_list

T_x = 1 # temporal variable
temp_list = [] # temporal variable list
declare_list = [] # temporal declare list
```

- `newtemp()` : 1.δημιουργεί και επιστρέφει μία νέα προσωρινή μεταβλητή  
2. οι προσωρινές μεταβλητές είναι της μορφής T\_1,T\_2,...

```
def newtemp(): # we create a new temporal variable, T_1, T_2, T_3,
    global T_x
    global temp_list
    temp_str = 'T_'
    temp_str += str(T_x)
    T_x += 1
    temp_list += [temp_str]

    # create entity for the table of symbols
    entity = Record_entity()
    entity.type = 'Temp'
    entity.name = temp_str
    entity.tempVar.offset = get_offset()
    new_entity(entity)

    return temp_str
```

- `emptylist()` : δημιουργεί μία κενή λίστα ετικετών τετράδων

```
def emptylist(): # creates and returns an empty quad list
    empty_list = []
    return empty_list
```

- `makelist(x)` : δημιουργεί μία λίστα ετικετών τετράδων που περιέχει μόνο το x

```
def makelist(x): # creates a list of labels for the quads that contains only x
    make_list = [x]
    return make_list
```

- `merge(list1, list2)` : δημιουργεί μία λίστα ετικετών τετράδων από τη συνένωση των λιστών list1, list2

```
def mergelist(list1, list2): # we merge the two list
    merge_list = []
    merge_list += list1
    merge_list += list2
    return merge_list
```

- `backpatch(list,z)` : 1. η λίστα list αποτελείται από δείκτες σε τετράδες των οποίων το τελευταίο τελούμενο δεν είναι συμπληρωμένο, 2. η `backpatch` επισκέπτεται μία μία τις τετράδες αυτές και τις συμπληρώνει με την ετικέτα z

```
def backpatch(listt, z): # has pointers to quads where last op doesnt exists,
    # backpatch visits one by one those quads and puts z
    global quad_list
    for x in range(0, len(listt)):
        for j in range(0, len(quad_list)):
            if listt[x] == quad_list[j][0] and str(quad_list[j][4]) == '_': # same label and in place 4 _ in quadlist
                quad_list[j][4] = z
                j = len(quad_list)
```

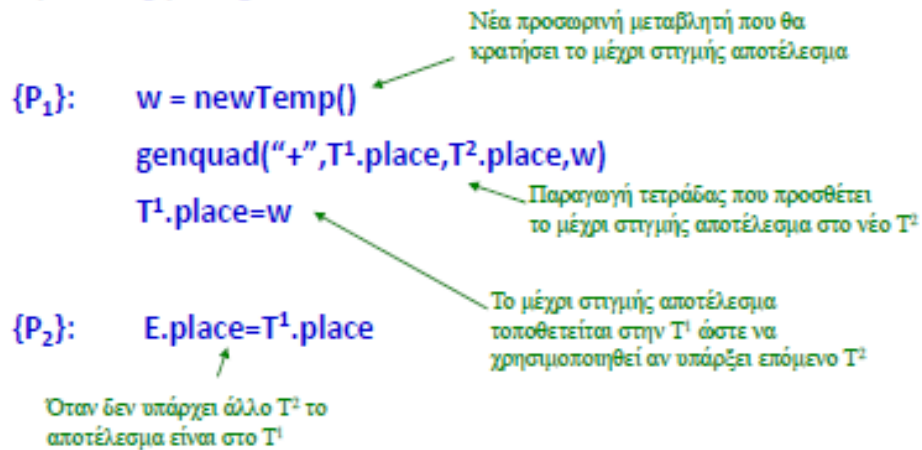
Οι παραπάνω συναρτήσεις τοποθετήθηκαν σε κατάλληλες θέσεις στον κώδικα του συντακτικού αναλυτή ώστε να αποθηκεύσουμε τις τετράδες για την δημιουργία του αρχείου ενδιάμεσου κώδικα , στη περίπτωση μας το `test.int` και στη συνέχεια για την δημιουργία του αρχείου C , στην περίπτωση μας του `test.c`.



Η μεθοδολογία για την παραγωγή του ενδιάμεσου κώδικα βάση των οδηγιών του καθηγητή είναι :

1. Αριθμητικές πράξεις οι οποίες τοποθετήθηκαν στις συναρτήσεις `expression()`, `term()`, `factor()`

$E \rightarrow T^1 (+ T^2 \{P_1\}) * \{P_2\}$



$T \rightarrow F^1 (\times F^2 \{P_1\}) * \{P_2\}$

**{P<sub>1</sub>}:    w = newTemp()**  
**genquad("×",F<sup>1</sup>.place,F<sup>2</sup>.place,w)**  
**F<sup>1</sup>.place=w**

**{P<sub>2</sub>}:    T.place=F<sup>1</sup>.place**

Ανάλογη λογική με τον κανόνα E

$F \rightarrow ( E ) \{P_1\}$

Απλή μεταφορά από το E.place στο F.place

**{P<sub>1</sub>}: F.place=E.place**

$F \rightarrow id \{P_1\}$

Απλή μεταφορά από το id.place στο F.place

**{P<sub>1</sub>}: F.place=id.place**

## 2. Λογικές παραστάσεις οι οποίες τοποθετήθηκαν στις συναρτήσεις condition(), boolfactor(), boolterm()

$R \rightarrow (B) \{P_1\}$

$\{P_1\}$ :  $R.true = B.true$   
 $R.false = B.false$

Μεταφορά των τετράδων από τη λίστα B στη λίστα R

### • NOT

$R \rightarrow \text{not} (B) \{P_1\}$

$\{P_1\}$ :  $R.true = B.false$   
 $R.false = B.true$

Αντιστροφή και μεταφορά τετράδων από τη λίστα B στη λίστα R

### • OR

$B \rightarrow Q^1 \{P_1\} ( \text{or} \{P_2\} Q^2 \{P_3\} )^*$

$\{P_1\}$ :  $B.true = Q^1.true$   
 $B.false = Q^1.false$

$\{P_2\}$ :  $\text{backpatch}(B.false, \text{nextquad}())$

$\{P_3\}$ :  $B.true = \text{merge}(B.true, Q^2.true)$   
 $B.false = Q^2.false$

Μεταφορά των τετράδων από τη λίστα  $Q^1$  στη λίστα B

Συμπλήρωση όσων τετράδων μπορούν να συμπληρωθούν μέσα στον κανόνα

Συσσώρευση στη λίστα true των τετράδων που δεν μπορούν να συμπληρωθούν και αντιστοιχούν σε αληθή αποτίμηση λογικής παράστασης

Η λίστα false περιέχει την τετράδα η οποία αντιστοιχεί σε στη μη αληθή αποτίμηση της λογικής παράστασης

### • AND

$Q \rightarrow R^1 \{P_1\} ( \text{and} \{P_2\} R^2 \{P_3\} )^*$

$\{P_1\}$ :  $Q.true = R^1.true$   
 $Q.false = R^1.false$

$\{P_2\}$ :  $\text{backpatch}(Q.true, \text{nextquad}())$

$\{P_3\}$ :  $Q.false = \text{merge}(Q.false, R^2.false)$   
 $Q.true = R^2.true$

Μεταφορά των τετράδων από τη λίστα  $R^1$  στη λίστα Q

Συμπλήρωση όσων τετράδων μπορούν να συμπληρωθούν μέσα στον κανόνα

Συσσώρευση στη λίστα false των τετράδων που δεν μπορούν να συμπληρωθούν και αντιστοιχούν σε μη αληθή αποτίμηση λογικής παράστασης

Η λίστα true περιέχει την τετράδα η οποία αντιστοιχεί σε στη αληθή αποτίμηση της λογικής παράστασης

$R \rightarrow E^1 \text{ relop } E^2 \{P_1\}$

```
{P1}:    R.true=makelist(nextquad())
          genQuad(relop, E1.place, E2.place, "_")

          R.false=makelist(nextquad())
          genQuad("jump", "_", "_", "_")
```

Δημιουργία μη συμπληρωμένης τετράδας και εισαγωγή στη λίστα μη συμπληρωμένων τετράδων για την αληθή αποτίμηση της relop

Δημιουργία μη συμπληρωμένης τετράδας και εισαγωγή στη λίστα μη συμπληρωμένων τετράδων για τη μη αληθή αποτίμηση της relop

### 3. Κλήση υποπρογραμμάτων

- def actualparitem(): # in expression | inout ID  
par, a, CV, \_  
par, b, REF, \_  
w = newTemp()  
par, w, RET, \_  
call, assign\_v, \_, \_
- εντολή return  
S -> return (E) {P1}  
{P1}: genquad("retv", E.place, "\_", "\_")
- εκχώρηση  
S -> id := E {P1};  
{P1} : genQuad(":=", E.place, "\_", id)
- Δομή while

$S \rightarrow \text{while } \{P_1\} B \text{ do } \{P_2\} S^1 \{P_3\}$

```
{P1}:    Bquad:=nextquad()
{P2}:    backpatch(B.true,nextquad())
{P3}:    genquad("jump", "_", "_", Bquad)
          backpatch(B.false,nextquad())
```

Συμπλήρωση των τετράδων που έχουν μείνει ασυμπληρωτές και γνωρίζουμε τώρα ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα, το true πάνω στην S και το false έξω από τη δομή

Μετάβαση στην αρχή της συνθήκης ώστε να ξαναγίνει έλεγχος

- Δομή if

```

S -> if B then {P1} S1 {P2} TAIL {P3}
{P1}:    backpatch(B.true,nextquad())
{P2}:    ifList=makelist(nextquad())
          genquad("jump","_","_","_")
          backpatch(B.false,nextquad())
{P3}:    backpatch(ifList,nextquad())
TAIL -> else S2 | TAIL -> ε

```

Συμπλήρωση των τετράδων που έχουν μείνει ασυμπληρωτές και γνωρίζουμε τώρα ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα, στο if και else αντίστοιχα

Εξασφαλίζουμε ότι εάν εκτελεστούν οι εντολές του if δε θα εκτελεστούν στη συνέχεια οι εντολές του else

- Δομή switch

```

S -> switch {P1}

( (cond): {P2} S1 break {P3} ) *
default: S2 {P4}

{P1}:    exitlist = emptylist()
{P2}:    backpatch(cond.true,nextquad())
{P3}:    e = makelist(nextquad())
          genquad('jump','_','_','_')
          mergelist(exitlist,e)
          backpatch(cond.false,nextquad())
{P4}:    backpatch(exitlist,nextquad())

```

- Εντολή input

```

S -> input (id) {P1}
{P1}:    genquad("inp",id.place,"_","_")

```

- Εντολή print

```

S -> print (E) {P2}
{P2}:    genquad("out",E.place,"_","_")

```

Με την ολοκλήρωση της εκτέλεσης των παραπάνω δημιουργήσαμε το αρχείο test.int όπου τοποθετήσαμε τις κατάλληλες τετράδες. Βάσει αυτού του αρχείου δημιουργήσαμε και το αντίστοιχο αρχείο test.c, το οποίο δημιουργείται ΜΟΝΟ όταν δεν υπάρχουν στο .ci αρχείο τα υποπρογράμματα function ή/και procedure.

## 2.4 Πίνακας Συμβόλων

Σκοπός των συναρτήσεων του πίνακα συμβόλων είναι να δημιουργούν μια δομή που κρατάει μοναδικές πληροφορίες για αντικείμενα που μπορούν να αντληθούν από τον κώδικα προς μετάφραση. Σε ένα πρόγραμμα κατα την μετάφραση, μας είναι απαραίτητο να έχουμε πληροφορία για την κάθε οντότητα που το αποτελεί. Οι οντότητες είναι το σύνολο όλων των μεταβλητών, συναρτήσεων και διαδικασιών. Πρέπει επομένως για κάθε μια από αυτές να γνωρίζουμε τι τύπος είναι δηλαδή αν είναι μια απλή μεταβλητή, μια προσωρινή μεταβλητή, αν είναι παράμετρος ή αν είναι υποπρόγραμμα(διαδικασία ή συνάρτηση).

Ορίσαμε τα αντικείμενα Record entity, Record scope, Record argument καθώς και τα πεδία τους. Η αρχικοποίηση τους γίνεται μέσα σε συναρτήσεις του συντακτικού αναλυτή και του ενδιάμεσου κώδικα όπως φαίνεται στον κώδικα μας στο αρχείο `cimple.py`. Επομένως, κάθε φορά που εντοπίζεται από τον λεκτικό αναλυτή το όνομα μιας μεταβλητή, μιας παραμέτρου ή ενός υποπρογράμματος, προστίθεται στον πίνακα η πληροφορία για κάθε μια από αυτές.

Οι εγγραφές στον πίνακα συμβόλων είναι οι εξής:

### Record Entity

- Μεταβλητή
  1. string name
  2. int type (Τον τύπο τους (πχ int, float, char). Η Cimple υποστηρίζει μόνο τύπο int οπότε δεν χρειάζεται να κρατάμε πληροφορία σε αυτή την περίπτωση)
  3. int offset (απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης)
- Συνάρτηση
  1. string name
  2. int type (Τον τύπο τους (διαδικασία-procedure ή συνάρτηση-function))
  3. int startQuad (ετικέτα της πρώτης τετράδας του κώδικα της συνάρτησης)
  4. list argument (λίστα παραμέτρων)
  5. int framelength (μήκος εγγραφήματος δραστηριοποίησης)
- Σταθερά

- 1. string name
- 2. string value (τιμή της σταθεράς)
- Παράμετρος
  - 1. string name
  - 2. int parMode (τρόπος περάσματος)
  - 3. int offset (απόσταση από την κορυφή της στοίβας)
- Προσωρινή μεταβλητή
  - 1. string name
  - 2. int offset (απόσταση από την κορυφή της στοίβας)

### Record Scope

- List Entity (λίστα από Entities)
- int nestingLevel (βάθος φωλιάσματος)

### Record Argument

- int parMode (τρόπος περάσματος)
- int type (τύπος μεταβλητής)

Ύστερα δημιουργήσαμε κάποιες βοηθητικές συναρτήσεις,

- μια συνάρτηση που προσθέτει νέο scope (def new\_scope(name)) όταν ξεκινάμε τη μετάφραση μιας νέας συνάρτησης,
- μια συνάρτηση που διαγράφει ένα scope όταν τελειώνουμε τη μετάφραση μιας συνάρτησης - με τη διαγραφή διαγράφουμε την εγγραφή (record) του Scope και όλες τις λίστες με τα Entity και τα Argument που εξαρτώνται από αυτήν (def delete\_scope()),
- μια για να προσθέσουμε νέο entity (def new\_entity(object)) όταν
  - συναντάμε δήλωση μεταβλητής
  - δημιουργείται νέα προσωρινή μεταβλητή
  - συναντάμε δήλωση νέας συνάρτησης
  - συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης
- μια για να υπολογίζουμε το offset στη στοίβα καθενός entity (def get\_offset()),
- μια για προσθήκη νέου argument (def add\_newargument(object)) όταν συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης
- μια για να παίρνουμε την επόμενη λίστα τετράδων (def get\_firstQuad())
- μια για να παίρνουμε το μήκος ενός υποπρογράμματος (def get\_frameLen()).

Τέλος ορίσαμε μια συνάρτηση που συμπληρώνει τον πίνακα συμβόλων τις παραμέτρους των υποπρογραμμάτων(`def add_parameters()`) και μία ακόμη που γράφει τον πίνακα συμβόλων σε ένα αρχείο κειμένου για ευκολότερη κατανόηση και πιο ευέλικτη χρήση(`def write_table_of_symbols()`).

## 2.5 Τελικός κώδικας

Η παραγωγή τελικού κώδικα αποτελεί το τελευταίο κομμάτι της δημιουργίας του μεταφραστή και είναι απαραίτητο για να μπορέσει να εκτελεστεί ο κώδικας. Κατά την διαδικασία αυτή, από κάθε μία “τετράδα” του ενδιαμέσου κώδικα παράγονται οι απαραίτητες εντολές μηχανής στη γλώσσα χαμηλού επιπέδου “assembly”, συγκεκριμένα θα δημιουργήσουμε κώδικα για τον επεξεργαστή MIPS. Κύριες ενέργειες στη φάση αυτή:

- οι μεταβλητές απεικονίζονται στην μνήμη (στοίβα)
- το πέρασμα παραμέτρων και η κλήση συναρτήσεων

Καταχωρητές που θα μας φανούν χρήσιμοι:

- καταχωρητές προσωρινών τιμών: `$t0...$t7`
- καταχωρητές οι τιμές των οποίων διατηρούνται ανάμεσα σε κλήσεις συναρτήσεων: `$s0...$s7`
- καταχωρητές ορισμάτων: `$a0...$a3`
- καταχωρητές τιμών: `$v0,$v1`
- stack pointer `$sp`
- frame pointer `$fp`
- return address `$ra`

Εντολές που θα μας φανούν χρήσιμες για αριθμητικές πράξεις:

- `add $t0,$t1,$t2` (πρόσθεση)  
`t0=t1+t2`
- `sub $t0,$t1,$t2` (αφαίρεση)  
`t0=t1-t2`
- `mul $t0,$t1,$t2` (πολλαπλασιασμός)  
`t0=t1*t2`
- `div $t0,$t1,$t2` (διαίρεση)  
`t0=t1/t2`

Εντολές που θα μας φανούν χρήσιμες για μετακίνηση δεδομένων:

- `move $t0,$t1`      `t0=t1`      μεταφορά ανάμεσα σε καταχωρητές
- `li $t0, value`      `t0=value`      σταθερά σε καταχωρητή
- `lw $t1,mem`      `t1=[mem]`      περιεχόμενο μνήμης σε καταχωρητή
- `sw $t1,mem`      `[mem]=t1`      περιεχόμενο καταχωρητή σε μνήμη
- `lw $t1,($t0)`      `t1=[t0]`      έμμεση αναφορά με καταχωρητή
- `sw $t1,-4($sp)`      `t1=[$sp-4]`      έμμεση αναφορά με βάση τον \$sp

Εντολές που θα μας φανούν χρήσιμες για άλματα:

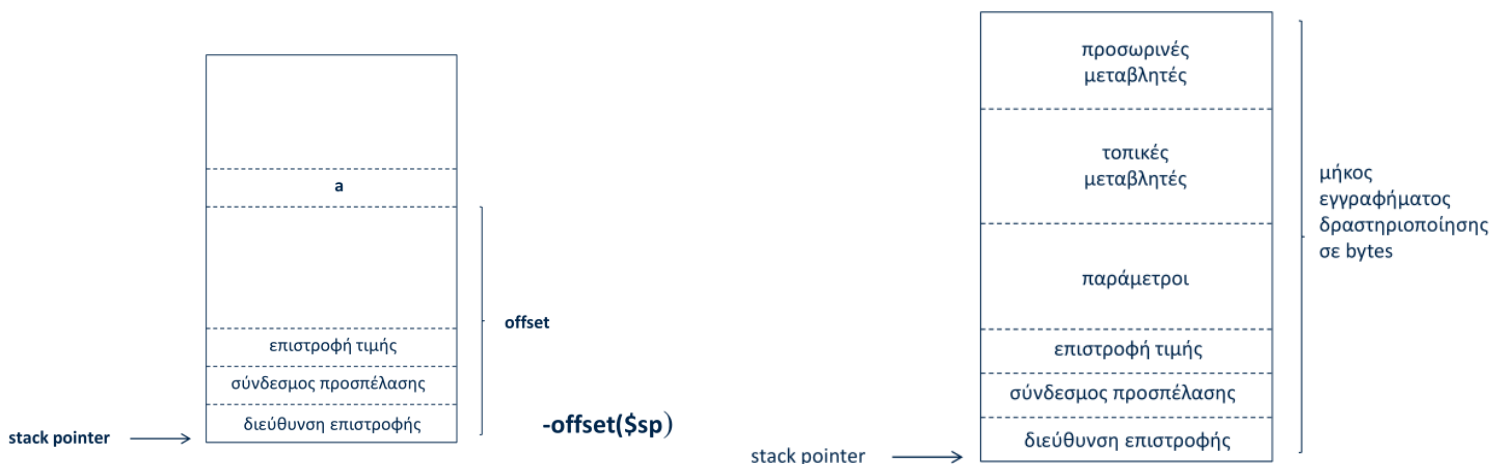
- `b label`      branch to label
- `beq $t1,$t2,label`      jump to label if `$t1=$t2`
- `blt $t1,$t2,label`      jump to label if `$t1<$t2`
- `bgt $t1,$t2,label`      jump to label if `$t1>$t2`
- `ble $t1,$t2,label`      jump to label if `$t1<=$t2`
- `bge $t1,$t2,label`      jump to label if `$t1>=$t2`
- `bne $t1,$t2,label`      jump to label if `$t1<>$t2`

Εντολές που θα μας φανούν χρήσιμες στην κλήση συναρτήσεων:

- `j label`      jump to label
- `jal label`      κλήση συνάρτησης
- `jr $ra`      άλμα στη διεύθυνση που έχει ο καταχωρητής, στο παράδειγμα είναι ο \$ra που έχει την διεύθυνση επιστροφής συνάρτησης

## Εγγραφήμα Δραστηριοποίησης

Το εγγραφήμα δραστηριοποίησης, είναι ένας θεωρητικός πίνακας, ο οποίος δημιουργείται στην αρχή του κυρίως προγράμματος καθώς και στην αρχή κάθε άλλου υποπρογράμματος.





Για να μπορέσουμε να παράξουμε τον τελικό κώδικα χρειαζόμαστε ορισμένες βοηθητικές συναρτήσεις. Συγκεκριμένα, τις εξής:

1. `gnvlcode`:

- μεταφέρει στον `$t0` την διεύθυνση μιας μη τοπικής μεταβλητής.
- απο τον πίνακα συμβόλων βρίσκει πόσα επίπεδα πάνω βρίσκεται η μη τοπική μεταβλητή και μέσα από τον σύνδεσμο προσπέλασης την εντοπίζει.

`lw $t0, -4($sp)`                    στοίβα του γονέα

όσες φορές χρειαστεί:

`lw $t0, -4($t0)`                    στοίβα του προγόνου που έχει τη μεταβλητή

`addi $t0, $t0, -offset`            διεύθυνση της μη τοπικής μεταβλητής

2. `loadvr`:

- μεταφορά δεδομένων στον καταχωρητή `r`
- η μεταφορά μπορεί να γίνει από τη μνήμη (στοίβα)
- ή να εκχωρηθεί στο `r` μία σταθερά
- η σύνταξη της είναι `loadvr(v,r)`
- διακρίνουμε περιπτώσεις:
  - a. αν `v` είναι σταθερά:  
`li $tr,v`
  - b. αν `v` είναι καθολική μεταβλητή – δηλαδή ανήκει στο κυρίως πρόγραμμα:  
`lw $tr,-offset($s0)`
  - c. αν η `v` έχει δηλωθεί στη συνάρτηση που αυτή τη στιγμή εκτελείται και είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή, ή προσωρινή μεταβλητή:  
`lw $tr,-offset($sp)`
  - d. αν η `v` έχει δηλωθεί στη συνάρτηση που αυτή τη στιγμή εκτελείται και είναι τυπική παράμετρος που περνάει με αναφορά:  
`lw $t0,-offset($sp)`  
`lw $tr,($t0)`
  - e. αν η `v` έχει δηλωθεί σε κάποιο πρόγονο και εκεί είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή:  
`gnvlcode()`  
`lw $tr,($t0)`

- f. αν η ν έχει δηλωθεί σε κάποιο πρόγονο και εκεί είναι τυπική παράμετρος που περνάει με αναφορά:  
`gnlvcde()`  
`lw $t0,($t0)`  
`lw $tr,($t0)`
- g. αν η ν έχει δηλωθεί σε κάποιο πρόγονο και εκεί είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή:  
`gnlvcde()`  
`lw $tr,($t0)`
- h. αν η ν έχει δηλωθεί σε κάποιο πρόγονο και εκεί είναι τυπική παράμετρος που περνάει με αναφορά:  
`gnlvcde()`  
`lw $t0,($t0)`  
`lw $tr,($t0)`

### 3. `storer`:

- μεταφορά δεδομένων από τον καταχωρητή r στη μνήμη (μεταβλητή ν)
- η σύνταξη της είναι `storer(r,v)`
- διακρίνουμε περιπτώσεις
  - a. αν ν είναι καθολική μεταβλητή – δηλαδή ανήκει στο κυρίως πρόγραμμα:  
`sw $tr,-offset($s0)`
  - b. αν ν είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή:  
`sw $tr,-offset($sp)`
  - c. αν ν είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον:  
`lw $t0,-offset($sp)`  
`sw $tr,($t0)`
  - d. αν ν είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον:  
`gnlvcde(v)`  
`sw $tr,($t0)`
  - e. αν ν είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον:  
`gnlvcde(v)`  
`lw $t0,($t0)`

sw \$tr,(\$t0)

#### Εντολές αλμάτων:

- jump, “\_”, “\_”, label  
b label
- relop(?),x,y,z  
loadvr(x,\$t1)  
loadvr(y, \$t2)  
branch(?),\$t1,\$t2,z                      branch(?) : beq,bne,bgt,blt,bge,ble

#### Εκχώρηση:

- :=, x, “\_”, z  
loadvr(x, \$t1)  
storerv(\$t1, z)

#### Εντολές Αριθμητικών Πράξεων:

- op x,y,z  
loadvr(x, \$t1)  
loadvr(y, \$t2)  
op \$t1,\$t1,\$t2                      op: add,sub,mul,div  
storerv(\$t1,z)

#### Εντολές Εισόδου-Εξόδου

- out “\_”, “\_”, x (για την print)  
li \$v0,1  
loadvr(x,\$a0)  
syscall
- inp “\_”, “\_”, x (για την input)  
li \$v0,5  
syscall  
storerv(\$v0,x)

#### Επιστροφή Τιμής Συνάρτησης

- retv “\_”, “\_”, x (για την return)  
loadvr(x, \$t1)  
lw \$t0,-8(\$sp)  
sw \$t1,(\$t0)

αποθηκεύεται ο x στη διεύθυνση που είναι αποθηκευμένη στην 3η θέση του εγγραφήματος δραστηριοποίησης

- εναλλακτικά μπορούμε να γράψουμε το αποτέλεσμα στον \$v0, και μετά πρέπει να φροντίσουμε να το πάρουμε από εκεί  
`loadvr(x, $t1)`  
`move $v0,$t1`

## Παράμετροι Συνάρτησης

- πριν κάνουμε τις ενέργειες ώστε να γίνει το πέραςμα της πρώτης παραμέτρου, τοποθετούμε τον \$fp να δείχνει στην στοίβα της συνάρτησης που θα δημιουργηθεί  
`addi $fp,$sp,framelength`
- στη συνέχεια, για κάθε παράμετρο και ανάλογα με το αν περνά με τιμή ή αναφορά κάνουμε τις εξής ενέργειες:
  1. par,x,CV, \_  
`loadvr(x, $t0)`  
`sw $t0, -(12+4i)($fp)`  
 όπου i ο αύξων αριθμός της παραμέτρου
  2. par,x,CV, \_  
`loadvr(x, $t0)`  
`sw $t0, -(12+4i)($fp)`  
 όπου i ο αύξων αριθμός της παραμέτρου
  3. par,x,REF, \_
    - a. αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή:  
`addi $t0,$sp,-offset`  
`sw $t0,-(12+4i)($fp)`
    - b. αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή:  
`gnlncode(x)`  
`sw $t0,-(12+4i)($fp)`
    - c. αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά  
`gnlncode(x)`

```
lw $t0,($t0)
sw $t0,-(12+4i)($fp)
```

4. par,x,RET, \_

γεμίζουμε το 3ο πεδίο του εγγραφήματος δραστηριοποίησης της κληθείσας συνάρτησης με τη διεύθυνση της προσωρινής μεταβλητής στην οποία θα επιστραφεί η τιμή

```
addi $t0,$sp,-offset
sw $t0,-8($fp)
```

## Κλήση Συνάρτησης

- call, \_, \_, f

αρχικά γεμίζουμε το 2ο πεδίο του εγγραφήματος δραστηριοποίησης της κληθείσας συνάρτησης, τον σύνδεσμο προσπέλασης, με την διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα της, ώστε η κληθείσα να γνωρίζει που να κοιτάξει αν χρειαστεί να προσπελάσει μία μεταβλητή την οποία έχει δικαίωμα να προσπελάσει, αλλά δεν της ανήκει.

Διακρίνουμε περιπτώσεις:

- a. αν καλούσα και κληθείσα έχουν το ίδιο βάθος φωλιάσματος, τότε έχουν τον ίδιο γονέα:

```
lw $t0,-4($sp)
sw $t0,-4($fp)
```
- b. αν καλούσα και κληθείσα έχουν διαφορετικό βάθος φωλιάσματος, τότε η καλούσα είναι ο γονέας της κληθείσας:

```
sw $sp,-4($fp)
```

Στη συνέχεια μεταφέρουμε τον δείκτη στοίβας στην κληθείσα

```
addi $sp,$sp,frameLength
```

καλούμε τη συνάρτηση

```
jal f
```

και όταν επιστρέψουμε παίρνουμε πίσω τον δείκτη στοίβας στην καλούσα

```
addi $sp,$sp,-frameLength
```

μέσα στην κληθείσα: στην αρχή κάθε συνάρτησης αποθηκεύουμε στην πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της, την οποία έχει τοποθετήσει στον \$ra η

```
jal
```

```
sw $ra,($sp)
```

ενώ στο τέλος κάθε συνάρτησης κάνουμε το αντίστροφο, παίρνουμε από την πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της συνάρτησης και την βάζουμε πάλι στον \$ra. Μέσω του \$ra επιστρέφουμε στην καλούσα

```
lw $ra,($sp)
jr $ra
```

## Αρχή Προγράμματος και Κυρίως Πρόγραμμα

Το κυρίως πρόγραμμα δεν είναι το πρώτο πράγμα που μεταφράζεται, οπότε στην αρχή του προγράμματος χρειάζεται ένα άλμα που να οδηγεί στην πρώτη ετικέτα του κυρίως προγράμματος “j L+(program name)”. Φυσικά η “j L+(program name)” πρέπει να δημιουργηθεί όταν ξεκινά η μετάφραση της main. Στη συνέχεια πρέπει να κατεβάσουμε τον \$sp κατά frameLength της main με την εντολή “addi \$sp,\$sp,frameLength” και να σημειώσουμε στον \$s0 το εγγράφημα δραστηριοποίησης της main ώστε να έχουμε εύκολη πρόσβαση στις global μεταβλητές με την εντολή “move \$s0,\$sp”.

Όλες οι περιπτώσεις για την δημιουργία των τετράδων του τελικού κώδικα υλοποιούνται στην συνάρτηση asm\_code() και τα αποθηκεύουμε στο αρχείο test.asm.

## 3 Παραδείγματα για ορθή λειτουργία του μεταφραστή

Παρακάτω θα παρουσιάσουμε δύο παραδείγματα αρχείων cimple για να δείξουμε πως εφαρμόζουμε τα βήματα που αναλύσαμε παραπάνω και πως εφαρμόζονται στην μετάφραση των αρχείων αυτών. Τα αρχεία περιέχουν διάφορες δηλώσεις μεταβλητών, κάποιους ελέγχους ροής ή επαναλήψεις, μερικές αναθέσεις τιμής, σχόλια και μια εντολή εκτύπωσης. Και τα δύο παραδείγματα περνάνε από τον λεκτικό και συντακτικό αναλυτή χωρίς λάθη καθώς ακολουθούν όλους τους κανόνες και έχουν ως σκοπό να δείξουν την ορθότητα παραγωγής του τελικού εκτελέσιμου κώδικα.

### 3.1 Λειτουργία μεταφραστή με το αρχείο fact.ci

Αρχίζουμε με το αρχείο fact.ci που υπολογίζει το παραγοντικό ενός αριθμού και ο κώδικας του φαίνεται παρακάτω. Να θυμίσουμε ότι η εντολή που τρέχουμε στο cmd/terminal είναι: `python cimple.py fact.ci`

```
e.py × fact.ci × switchCase.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);
}.
```

Συνεχίζουμε με τον ενδιάμεσο κώδικα που δημιουργεί το test.int αρχείο και το test.c που είναι το αρχείο για την C.

- 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_1
8: := T_1 _ fact
9: + i 1 T_2
10: := T_2 _ i
11: jump _ _ 5
12: out fact _ _
13: halt _ _ _
14: end_block factorial _ _
```

- test.c

```

int main()
{
    int T_1,T_2,x,i,fact;
    L_1:
    L_2: scanf("x = %d", &x); // 2:  inp  x  _  _
    L_3: fact = 1; // 3:  :=  1  _  fact
    L_4: i = 1; // 4:  :=  1  _  i
    L_5: if (i <= x) goto L_7; // 5:  <=  i  x  7
    L_6: goto L_12; // 6:  jump  _  _  12
    L_7: T_1 = fact * i; // 7:  *  fact  i  T_1
    L_8: fact = T_1; // 8:  :=  T_1  _  fact
    L_9: T_2 = i + 1; // 9:  +  i  1  T_2
    L_10: i = T_2; // 10:  :=  T_2  _  i
    L_11: goto L_5; // 11:  jump  _  _  5
    L_12: printf("fact = %d", fact); // 12:  out  fact  _  _
    L_13: {} // 13:  halt  _  _  _
}

```

Παρακάτω φαίνεται ο πίνακας συμβόλων:

```

| RECORD SCOPE-> Name: factorial, NestingLevel: 0
| ENTITIES:
| RECORD ENTITY-> Name: x | Type: Variable | VarType: int | Offset: 12|
| RECORD ENTITY-> Name: i | Type: Variable | VarType: int | Offset: 16|
| RECORD ENTITY-> Name: fact | Type: Variable | VarType: int | Offset: 20|
| RECORD ENTITY-> Name: T_1 | Type: Temp | TempType: int | Offset: 24|
| RECORD ENTITY-> Name: T_2 | Type: Temp | TempType: int | Offset: 28|

```



Και τέλος φαίνεται ο κώδικας του τελικού σταδίου assembly (Mars).

```
1  L0: b Lfactorial
2
3  L1:
4      add $sp,$sp,32
5      move $s0,$sp
6
7  L2:
8      li $v0, 5
9      syscall
10     sw $t0,-12($s0)
11
12     L3:
13         li $t1,1
14         sw $t1,-20($s0)
15
16     L4:
17         li $t1,1
18         sw $t1,-16($s0)
19
20     L5:
21         lw $t1,-16($s0)
22         lw $t2,-12($s0)
23         ble $t1,$t2,L7
24     L6:
25
26         j L12
27     L7:
28         lw $t1,-20($s0)
29         lw $t2,-16($s0)
30         mul $t1,$t1,$t2
31         sw $t1,-24($s0)
32
33     L8:
34         lw $t1,-24($s0)
35         sw $t1,-20($s0)
36
37     L9:
38         lw $t1,-16($s0)
39         li $t2,1
40         add $t1,$t1,$t2
41         sw $t1,-28($s0)
42
43     L10:
44         lw $t1,-28($s0)
45         sw $t1,-16($s0)
46
47     L11:
48
49         j L5
50     L12:
51         li $v0, 1
52         lw $t0,-20($s0)
53         move $a0,$t1
54         syscall
55
56     L13:
57
58     L14:
59         lw $ra,($sp)
60         jr $ra
61
```

Μερικά παραδείγματα σφαλμάτων και τα μηνύματα λάθους φαίνονται παρακάτω.

1. Αφαιρούμε το semicolon “;” στην γραμμή 8. Ο κώδικας και το αποτέλεσμα του φαίνονται στην παρακάτω εικόνα:



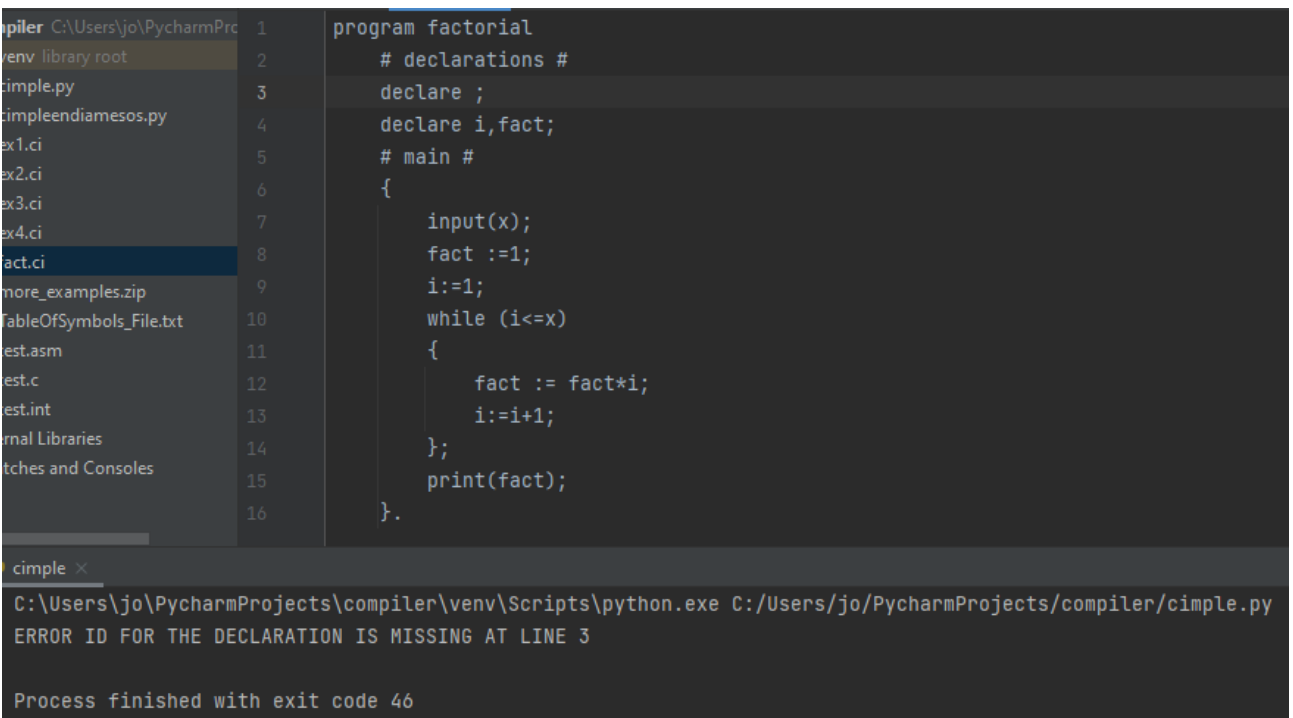
The screenshot shows the PyCharm IDE with a C program named 'factorial' in the editor. The program has 16 lines of code. Line 8 contains 'fact :=1' without a semicolon. The console output shows the command 'C:\Users\jo\PycharmProjects\compiler\venv\Scripts\python.exe C:/Users/jo/PycharmProjects/compiler/cimple.py' and the error message 'ERROR AT LINE 8, ; IS MISSING, NOTE: DO NOT ADD UNNECESSARY PARENTHESIS IN YOUR EXPRESSION'. The process finished with exit code 46.

```
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);
}.
```

ERROR AT LINE 8, ; IS MISSING, NOTE: DO NOT ADD UNNECESSARY PARENTHESIS IN YOUR EXPRESSION

Process finished with exit code 46

2. Αφαιρούμε την μεταβλητή x από την γραμμή 3, στο declare x;



The screenshot shows the PyCharm IDE with the same C program 'factorial'. Line 3 now contains 'declare ;' instead of 'declare x;'. The console output shows the command 'C:\Users\jo\PycharmProjects\compiler\venv\Scripts\python.exe C:/Users/jo/PycharmProjects/compiler/cimple.py' and the error message 'ERROR ID FOR THE DECLARATION IS MISSING AT LINE 3'. The process finished with exit code 46.

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

ERROR ID FOR THE DECLARATION IS MISSING AT LINE 3

Process finished with exit code 46

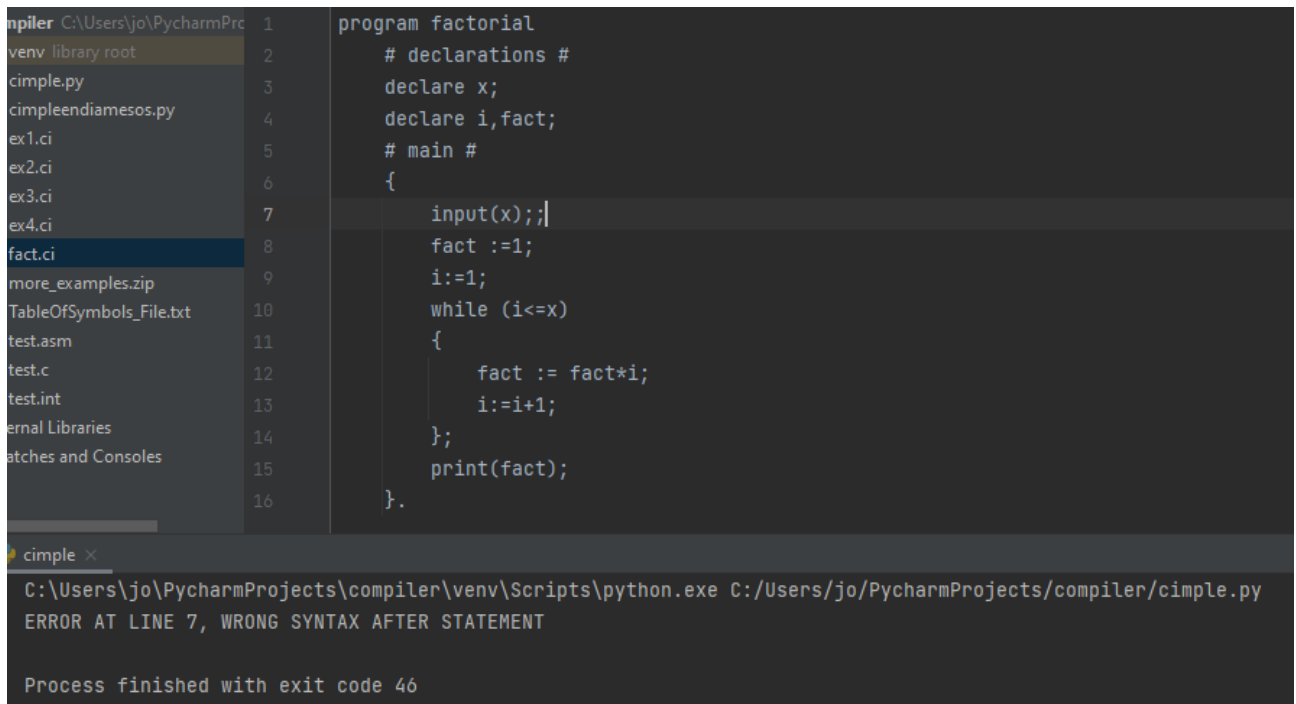
3. Αντικαθιστούμε στην γραμμή 12 το “:=” σε “=” που η γλώσσα cimple δεν το δέχεται σαν σύμβολο ανάθεσης.



```
1 program factorial
2 # declarations #
3 declare x;
4 declare i,fact;
5 # main #
6 {
7     input(x);
8     fact :=1;
9     i:=1;
10    while (i<=x)
11    {
12        fact = fact*i;
13        i:=i+1;
14    };
15    print(fact);
16 }.
```

Process finished with exit code 46

4. Στην γραμμή 7 προσθέσαμε ένα επιπλέον semicolon



```
1 program factorial
2 # declarations #
3 declare x;
4 declare i,fact;
5 # main #
6 {
7     input(x);;
8     fact :=1;
9     i:=1;
10    while (i<=x)
11    {
12        fact := fact*i;
13        i:=i+1;
14    };
15    print(fact);
16 }.
```

Process finished with exit code 46

## 3.2 Λειτουργία μεταφραστή με το αρχείο ex3.ci

Το άλλο παράδειγμα είναι το αρχείο ex3.ci, του οποίου ο κώδικας φαίνεται παρακάτω.

```
program example3
  declare x,y;
  declare z;
  if (x<z)
  {
    y :=x+1 ;
  }
  else
  {
    z :=z+2 ;
  };
  .
  #the end of phase 2#
  #we did it#
```

Συνεχίζουμε με τον ενδιάμεσο κώδικα που δημιουργεί το test.int αρχείο και το test.c που είναι το αρχείο για την C.

- test.int

```
1: begin_block  example3  _ _
2: < x z 4
3: jump _ _ 7
4: + x 1 T_1
5: := T_1 _ y
6: jump _ _ 9
7: + z 2 T_2
8: := T_2 _ z
9: halt _ _ _
10: end_block  example3  _ _
```

- test.c

```
int main()
{
  int T_1,T_2,x,y,z;
  L_1:
  L_2: if (x < z) goto L_4; // 2: < x z 4
  L_3: goto L_7; // 3: jump _ _ 7
  L_4: T_1 = x + 1; // 4: + x 1 T_1
  L_5: y = T_1; // 5: := T_1 _ y
  L_6: goto L_9; // 6: jump _ _ 9
  L_7: T_2 = z + 2; // 7: + z 2 T_2
  L_8: z = T_2; // 8: := T_2 _ z
```

Παρακάτω φαίνεται ο πίνακας συμβόλων:

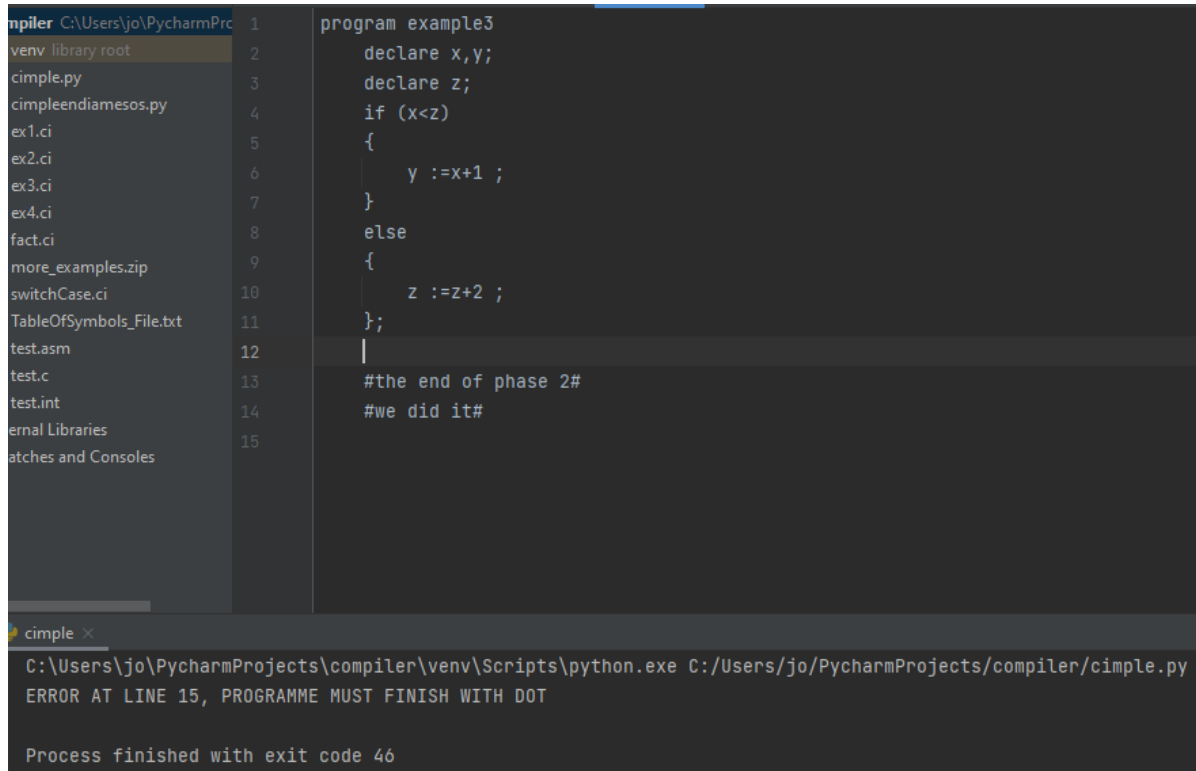
	RECORD SCOPE->	Name: example3,	NestingLevel: 0	
	ENTITIES:			
	RECORD ENTITY->	Name: x	Type: Variable	VarType: int   Offset: 12
	RECORD ENTITY->	Name: y	Type: Variable	VarType: int   Offset: 16
	RECORD ENTITY->	Name: z	Type: Variable	VarType: int   Offset: 20
	RECORD ENTITY->	Name: T_1	Type: Temp	TempType: int   Offset: 24
	RECORD ENTITY->	Name: T_2	Type: Temp	TempType: int   Offset: 28

Και τέλος φαίνεται ο κώδικας του τελικού σταδίου assembly (Mars).

```
1  L0: b Lexample3
2
3  L1:
4      add $sp,$sp,32
5      move $s0,$sp
6
7  L2:
8      lw $t1,-12($s0)
9      lw $t2,-20($s0)
10     blt $t1,$t2,L4
11  L3:
12
13     j L7
14  L4:
15     lw $t1,-12($s0)
16     li $t2,1
17     add $t1,$t1,$t2
18     sw $t1,-24($s0)
19
20  L5:
21     lw $t1,-24($s0)
22     sw $t1,-16($s0)
23
24  L6:
25
26     j L9
27  L7:
28     lw $t1,-20($s0)
29     li $t2,2
30     add $t1,$t1,$t2
31     sw $t1,-28($s0)
32
33  L8:
34     lw $t1,-28($s0)
35     sw $t1,-20($s0)
36
37  L9:
38
39  L10:
40     lw $ra,($sp)
41     jr $ra
42
```

Μερικά παραδείγματα σφαλμάτων και τα μηνύματα λάθους φαίνονται παρακάτω.

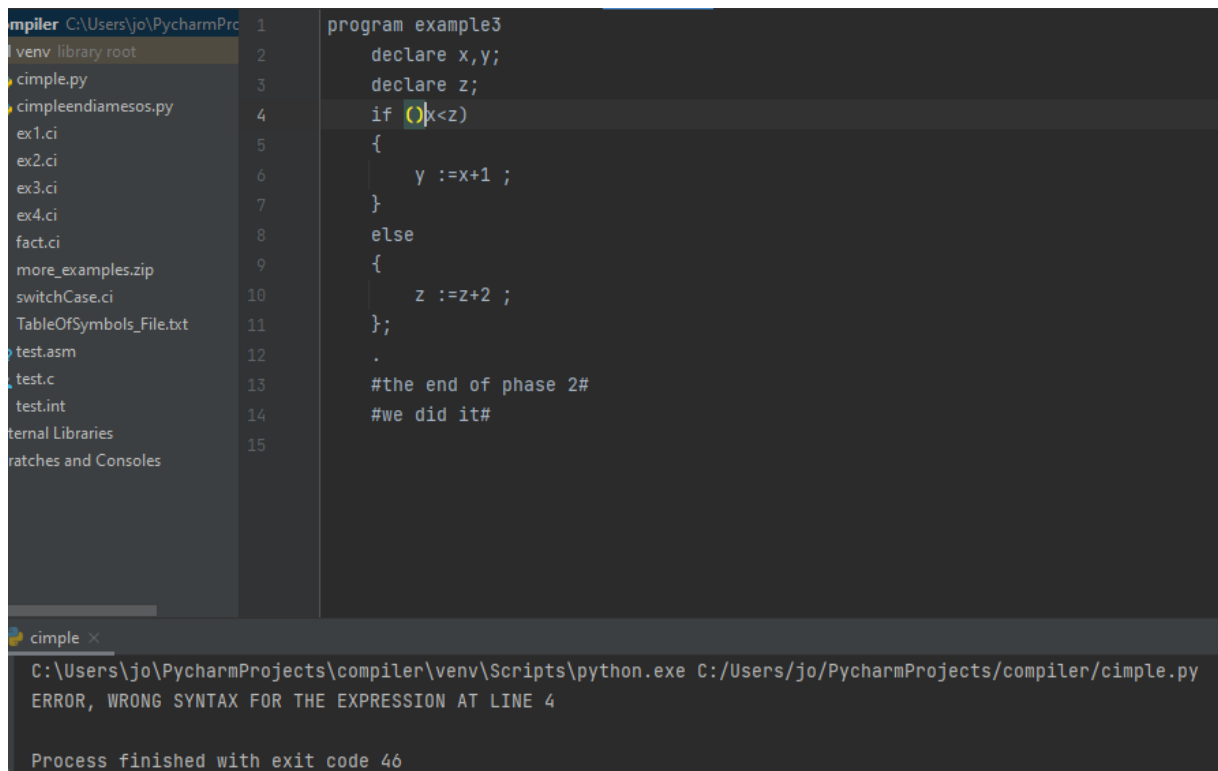
### 1. Αφαιρούμε το dot “.” από την γραμμή 12.



```
1 program example3
2 declare x,y;
3 declare z;
4 if (x<z)
5 {
6 y :=x+1 ;
7 }
8 else
9 {
10 z :=z+2 ;
11 };
12
13 #the end of phase 2#
14 #we did it#
15
```

C:\Users\jo\PycharmProjects\compiler\venv\Scripts\python.exe C:/Users/jo/PycharmProjects/compiler/cimple.py  
ERROR AT LINE 15, PROGRAMME MUST FINISH WITH DOT  
Process finished with exit code 46

### 2. Προσθέτουμε μια έξτρα παρένθεση στη γραμμή 4.



```
1 program example3
2 declare x,y;
3 declare z;
4 if (x<z)
5 {
6 y :=x+1 ;
7 }
8 else
9 {
10 z :=z+2 ;
11 };
12 .
13 #the end of phase 2#
14 #we did it#
15
```

C:\Users\jo\PycharmProjects\compiler\venv\Scripts\python.exe C:/Users/jo/PycharmProjects/compiler/cimple.py  
ERROR, WRONG SYNTAX FOR THE EXPRESSION AT LINE 4  
Process finished with exit code 46