Μεταφραστές 2016-2017
Προγραμματιστική άσκηση:
Η γλώσσα προγραμματισμού Ciscal

Ομάδα: Χρύσα Τεριζή
ΑΜ: 2553
username: cse32553
Ημερομηνία: 23/5/2017

Περιεχόμενα:

0.0 Εισαγωγή

0.1 Τι είναι ένας μεταγλωττιστής

Η βασική ιδέα ενός μεταγλωττιστή είναι ότι του δίνουμε ως είσοδο ένα πρόγραμμα το οποίο θέλουμε να μεταγλωττίσουμε και μας δίνει ως έξοδο ένα άλλο τελικό

πρόγραμμα(assembly). Κατά την διάρκεια της μεταγλώττισης μας εμφανίζονται μηνύματα λάθους και μερικές φορές μηνύματα προειδοποίησης.

0.1 Φάσεις μεταγλώττισης

Οι φάσεις της μεταγλώττισης είναι οι ακόλουθες:
1η φάση → Λεκτική ανάλυση
2η φάση → Συντακτική ανάλυση, στην φάση αυτή είναι απαραίτητη η γνώση της γραμματικής της γλώσσας που θέλουμε να μεταγλωττίσουμε.
3η φάση → Παραγωγή ενδιάμεσου κώδικα
4η φάση → Σημασιολογική ανάλυση
5η φάση → Πίνακας συμβόλων
6η φάση → Παραγωγή τελικού κώδικα

0.2 Κώδικας: global μεταβλητές λεκτικής και συντακτικής ανάλυσης

- **import sys** → χρειαζόμαστε την βιβλιοθήκη sys ώστε να κάνουμε exit από το πρόγραμμα μας.
- **arithmetic_operators** → λίστα η οποία περιέχει τα αριθμητικά σύμβολα(+, -, *, /).
- **relative_operators** → λίστα η οποία περιέχει τους τελεστές σύγκρισης(<, >, =).
- **grouping_symbols** → λίστα που περιέχει τα σύμβολα ομαδοποίησης((), [], {}).
- **reserved_words** → λίστα η οποία περιέχει τις δεσμευμένες λέξεις και οι οποίες λέξεις δεν μπορούν να είναι ονόματα μεταβλητών(and, exit, if, program, declare, procedure, in, or, do, function, inout, return, else, print, not, while, enddeclare, call, select, default).
- **other** → λίστα που περιέχει και κάποια ακόμα δεσμευμένα σύμβολα(!, \, ;, white space, newline, tab).

Υπάρχουν και κάποιες global μεταβλητές, οι οποίες είναι οι ακόλουθες,

- **numberOfLine = 1** → μεταβλητή για να ξέρω κάθε φορά σε ποια γραμμή του αρχείου είμαι. Το χρησιμοποιώ και όταν θέλω να εμφανίσω κάποια μηνύματα λάθους, να ξέρω ακριβώς την γραμμή όπου υπάρχει ο λάθος.
- **NumberOfColumns = 0** → μεταβλητή για να ξέρω σε ποια στήλη του εγγράφου βρίσκομαι.
- **TempLine = 1** → μεταβλητή που την χρησιμοποιώ για την παράλειψη των σχολίων.
- **tempColumn = 0** → μεταβλητή που την χρησιμοποιώ για την παράλειψη των σχολίων.
- **f = open(sys.argv[-1],"r")** → διαβάζει από το τερματικό το όνομα του αρχείου που θα μεταγλωττίσει.
- **line = f.readline()** → διαβάζει την γραμμή του εγγράφου.

- **letters = list(line)** → δημιουργείται μία λίστα με όλα τα γράμματα της γραμμής που διάβασα από το αρχείο.
- **previousWord** = "" → string που κρατάει την προηγούμενη λέξη.
- **reuseWord = 0** → αν θα χρησιμοποιήσω την λέξη ξανά σε περίπτωση που δεν καταναλωθεί.
- **isFunction = []** → λίστα που γεμίζει με True και False για το πότε έχω διαδικασία(procedure) ώστε να ελέγχω το return που πρέπει να βρίσκεται μέσα σε διαδικασίες μόνο.
- **isWhile = []** → λίστα που γεμίζει με True και False όταν βρίσκεται μέσα σε while ή όχι ώστε να ελέγχω το exit το οποίο πρέπει να βρίσκεται μόνο μέσα σε while.

1.0 1η φάση – Λεκτικός αναλυτής

Στον λεκτικό αναλυτή προσπαθούμε να βρούμε όλες τις λεκτικές μονάδες που υπάρχουν στο αρχείο το οποίο θέλουμε να μεταγλωττίσουμε. Είναι στην ουσία μία συνάρτηση η οποία καλείται από τον συντακτικό αναλυτή και κάθε φορά που καλείται επιστρέφει την επόμενη λεκτική μονάδα του αρχείου. (πχ. program hello {} θα πρέπει ο λεκτικός αναλυτής να μου επιστρέψει τις μονάδες program, hell, {, }, newline). Εσωτερικά λειτουργεί σαν ένα αυτόματο καταστάσεων το οποίο ξεκινά από μία αρχική κατάσταση και με την είσοδο κάθε χαρακτήρα αλλάζει κατάσταση έως ότου φτάσει σε μία τελική κατάσταση. Η δική μας γλώσσα Ciscal αναγνωρίζει ονόματα που αρχίζουν από γράμμα και συνεχίζουν με γράμμα ή αριθμό, φυσικούς αριθμούς, αριθμητικά σύμβολα, σχεσιακούς τελεστές και κάποια ακόμα σύμβολα όπου φαίνονται από τις global λίστες.

1.1 Κώδικας: λεκτικές μονάδες

Έχω υλοποιήσει κάποιες συναρτήσεις οι οποίες θα με βοηθήσουν στο να βρω τις λεκτικές μονάδες και οι οποίες είναι οι ακόλουθες,

- **def lexWord()** → αρχικά ελέγχω αν η λέξη που είχα πριν επιστρέψει χρειάζεται να ξανά χρησιμοποιηθεί, αν ναι τότε δεν επιστρέφω κάποια καινούρια αλλά την ίδια. Μέσα σε ένα ατέρμων βρόγχο καλώ την συνάρτηση ***lex()*** η οποία μου δίνει την επόμενη λεκτική μονάδα. Εάν η λέξη που θα μου επιστραφεί είναι διάφορη από το ***eof*** τότε συνεχίζω και ελέγχω το μήκος την λέξης που θα μου επιστραφεί. Αν είναι μεγαλύτερη από 30 χαρακτήρες την κόβω και επιστρέφω μόνο τους 30 πρώτους χαρακτήρες. Έπειτα ελέγχω αν η λέξη είναι αριθμός, αν ναι τότε ελέγχω αν βρίσκεται μέσα στα επιτρεπτά όρια [-32768, 32767] διαφορετικά εμφανίζεται μήνυμα λάθους και το πρόγραμμα δεν συνεχίζει παρακάτω. Μετά θέλω να ελέγξω αν έχω σχόλια ώστε να τα παραλείψω. Καλώ την συνάρτηση ***skipComments()*** και έπειτα συνεχίζω. Βάζω την λέξη στην μεταβλητή που έχω.

Σε περίπτωση που η λέξη που θα μου επιστραφεί είναι η ***eof*** τότε κάνω break από τον βρόγχο και κλείνω το αρχείο γιατί ξέρω ότι έχω τελειώσει την ανάγνωση από το αρχείο.

- **def lex()** → αρχικά πριν επιστρέψω κάποια λεκτική μονάδα καλώ την συνάρτηση ***skipSpaces()*** με την οποία παραλείπω όλα τα κενά πριν από κάποια λέξη. Μέσα σε ένα ατέρμων βρόγχο ελέγχω με την συνάρτηση ***eof()*** αν έχω φτάσει στο τέλος του αρχείου, εάν η λέξη που θα μου επιστραφεί είναι ***“”*** τότε ξέρω ότι έχω τελειώσει με το αρχείο μου διαφορετικά επιστρέφω την λεκτική μονάδα. Τώρα ξεκινάω να φτιάχνω τα αυτόματα. Δηλαδή, σε περίπτωση που μου έχει έρθει κάποιο γράμμα ελέγχω αν αυτή η λέξη ξεκινάει με αριθμό κάτι το οποίο δεν επιτρέπεται για όνομα μεταβλητής και επιστρέφω το αντίστοιχο λάθος και τερματίζει η λειτουργία. Διαφορετικά προσθέτω τα γράμματα στην λέξη και αυξάνω και τον αριθμό των στηλών και συνεχίζω. Εάν λάβω κάποιο ψηφίο τότε το προσθέτω στην λέξη μου και συνεχίζω. Αν το επόμενο γράμμα που θα διαβάσω ανήκει στην λίστα ***arithmetic_operators*** την διαβάζει και την προσθέτει στην λέξη του και συνεχίζει. Αν το γράμμα που θα μου έρθει ανήκει στην λίστα με τους ***relative_operators*** τότε διαβάζω και τον επόμενο χαρακτήρα σε περίπτωση που έχω κάποιο σύνθετο σύμβολο(<>, <=, >=). Πρώτα ελέγχω βέβαια αν υπάρχει κάποιος χαρακτήρας στην ίδια γραμμή για να διαβάσω. Έπειτα ελέγχω για το αν ο χαρακτήρας που θα διαβάζω ανήκει στις λίστες ***grouping_symbols*** ή είναι ***κόμμα***, αν ναι κάνω τα ίδια με πριν. Ακόμη, ελέγχω αν είναι το σύμβολο ***:*** και το οποίο μετά από αυτό θα πρέπει να βρίσκεται το σύμβολο = διαφορετικά έχω κάποιο άγνωστο λάθος. Εάν ο χαρακτήρας που θα διαβάσω είναι \ τότε σημαίνει ότι θέλω να ανοίξω σχόλια οπότε ελέγχω τον επόμενο χαρακτήρα, αν λείπει το ***** βγάζω κάποιο σχετικό μήνυμα λάθους διαφορετικά συνεχίζω. Ελέγχω για το ***;***, το ***white space***, το ***tab***. Τέλος, ελέγχω αν ο χαρακτήρας που μου έχει επιστραφεί είναι ***newline(για λειτουργικό windows)*** ή ***ord(letters[numberOfColumns]) == 13(για λειτουργικό linux)*** ώστε να επιστρέψω την λέξη που είχα διαβάσει και να πάω στην επόμενη γραμμή και μηδενίζω τις στήλες αφού ξεκινάω πάλι από την αρχή της να διαβάζω. Σε κάθε άλλη περίπτωση που δεν λάβω κάποιο από αυτά τα σύμβολα εμφανίζει μήνυμα λάθους και σταματάει η λειτουργία της λεκτικής ανάλυσης.
- ***def skipSpaces()*** → Ελέγχω αρχικά αν βρίσκομαι στο τέλος του αρχείου με την σχετική συνάρτηση. Εάν ο χαρακτήρας μου είναι το ***white space*** ή το ***tab*** συνεχίζω να διαβάζω χωρίς να προσθέτω χαρακτήρες στην λέξη μου. Αν συναντήσω ***newline*** ή το ***ord() == 13*** τότε καλώ την συνάρτηση ***skipNewlines()***.
- ***def skipNewlines()*** → η συνάρτηση αυτή περνάει κενές γραμμές που τυχών υπάρχουν μέσα στο αρχείο. Οπότε κάθε φορά που πηγαίνω σε καινούρια γραμμή διαβάζω και μία από αυτές με την εντολή ***line = f.readline()***.
- ***def eof()*** → συνάρτηση που ελέγχει αν έχω φτάσει στο τέλος του αρχείου.
- ***def skipComments()*** → Καλώ την συνάρτηση ***skipSpaces()*** ώστε να περάσω να κενά και να για διαπιστώσω αν τα σχόλια κλείνουν. Ελέγχω αν υπάρχει το

σύμβολο *** ώστε να ξέρω ότι έχουν κλείσει τα σχόλια ομαλά. Αν δεν βρω αυτό το σύμβολο τότε συνεχίζω να καλώ αναδρομικά την ***skipComments()***. Ελέγχω βέβαια και άμα έχω φτάσει στο τέλος του αρχείου χωρίς να κλείνουν κάπου και εμφανίζω σχετικό μήνυμα λάθους.

2.0 2η φάση – Συντακτικός αναλυτής

Με τον συντακτικό έλεγχο ουσιαστικά τεστάρουμε αν ο πηγαίος κώδικας ανήκει ή όχι στην γραμματική της γλώσσας. Η δική μας γλώσσα βασίζεται στην γραμματική ***LL(1)*** δηλαδή αναγνωρίζει από αριστερά προς τα 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. Για κάθε κανόνα της γραμματικής φτιάχνουμε και μία συνάρτηση.

2.1 Γραμματική γλώσσας Ciscal

Η γραμματική της δικής μου γλώσσας Ciscal είναι η ακόλουθη,

<PROGRAM> ::= **program** ID <BLOCK>

<BLOCK> ::= **{** <DECLARATIONS> <SUBPROGRAMS> <SEQUENCE> **}**

<DECLARATIONS> ::= ε | **declare** <VARLIST> **enddeclare**

<VARLIST> ::= ε | ID (, ID)*

<SUBPROGRAMS> ::= (<FUNC>) *

<FUNC> ::= **procedure** ID <FUNCBODY> |

function ID <FUNCBODY>

<FUNCBODY> ::= <FORMALPARS> <BLOCK>

<FORMALPARS> ::= **(** ε | <FORMALPARLIST> **)**

<FORMALPARLIST> ::= <FORMALPARITEM> (, <FORMALPARITEM>)*

<FORMALPARITEM> ::= **in** ID | **inout** ID

<SEQUENCE> ::= <STATEMENT> (; <STATEMENT>)*

<BRACKETS-SEQ> ::= { <SEQUENCE> }

<BRACK-OR-STAT> ::= <BRACKETS-SEQ> | <STATEMENT>;

<STATEMENT> ::= ε |

<ASSIGNMENT-STAT> |

<IF-STAT> |

<DO-WHILE-STAT> |

<WHILE-STAT> |

<SELECT-STAT> |

<EXIT-STAT> |

<RETURN-STAT> |

<PRINT-STAT> |

<CALL-STAT>

<ASSIGNMENT-STAT> ::= ID **:=** <EXPRESSION>

<IF-STAT> ::= **if (**<CONDITION>**)** <BRACK-OR-STAT> <ELSEPART>

<ELSEPART> ::= ε | **else** <BRACK-OR-STAT>

<WHILE-STAT> ::= **while (**<CONDITION>**)** <BRACK-OR-STAT>

<SELECT-STAT> ::= **select (**ID**)**

(CONST**:** <BRACK-OR-STAT>)*

DEFAULT**:** <BRACK-OR-STAT>

<DO-WHILE-STAT> ::= **do** <BRACK-OR-STAT> **while (**<CONDITION>**)**

<EXIT-STAT> ::= **exit**

<RETURN-STAT> ::= **return (**<EXPRESSION>**)**

<PRINT-STAT> ::= **print (**<EXPRESSION>**)**

<CALL-STAT> ::= **call** ID <ACTUALPARS>

<ACTUALPARS> ::= **(** ε | <ACTUALPARLIST> **)**

<ACTUALPARLIST> ::= <ACTUALPARITEM> (**,** <ACTUALPARITEM>)*

<ACTUALPARITEM> ::= **in** <EXPRESSION> | **inout** ID

<CONDITION> ::= <BOOLTERM> (**or** <BOOLTERM>)*

<BOOLTERM> ::= <BOOLFACTOR> (**and** <BOOLFACTOR>)*

<BOOLFACTOR> ::=**not [**<CONDITION>**]** |

[<CONDITION>**]** |

<EXPRESSION> <RELATIONAL-OPER> <EXPRESSION>

<EXPRESSION> ::= <OPTIONAL-SIGN> <TERM> (<ADD-OPER> <TERM>)*

<TERM> ::= <FACTOR> (<MUL-OPER> <FACTOR>)*

<FACTOR> ::= CONSTANT |

(<EXPRESSION>) |

ID <IDTAIL>

<IDTAIL> ::= ε | <ACTUALPARS>

<RELATIONAL-OPER> ::= = | < | <= | <> | >= | >

<ADD-OPER> ::= + | -

<MUL-OPER> ::= * | /

<OPTIONAL-SIGN> ::= ε | <ADD-OPER>

2.1 Κώδικας: συντακτική ανάλυση

Για κάθε κανόνα θα φτιάξουμε και μία συνάρτηση. Οπότε έχουμε τις ακόλουθες συναρτήσεις για την γραμματική της Ciscal,

- **def main()** → ο κώδικας μας πρέπει να ξεκινάει με την λέξη ***program*** οπότε καλώ την σχετική συνάρτηση.
- **def program()** → διαβάζω την επόμενη λεκτική μονάδα, αν είναι ***program*** τότε συνεχίζω να διαβάζω και την επόμενη. Ελέγχω αν μία λεκτική μονάδα είναι ***id*** με την συνάρτηση ***id(word)***, αν όντως ακολουθεί ***id*** τότε καλώ την συνάρτηση ***block()*** διαφορετικά εμφανίζει σχετικά μηνύματα λάθους.
- **def block()** → αν η λεκτική μου μονάδα είναι { τότε καλώ τις συναρτήσεις ***declarations(), subprograms() και sequence()***. Αν η επόμενη λεκτική μονάδα είναι διάφορη από το σύμβολο **}** τότε εμφανίζει λάθος διαφορετικά αφαιρεί από την λίστα ***isFunction*** το τελευταίο της στοιχείο. Αν λείπει το σύμβολο **{** εμφανίζει λάθος.
- **def declarations()** → εάν η λεκτική μονάδα είναι το declare τότε καλείται η συνάρτηση ***varlist()*.** Εάν δεν υπάρξει το ***enddeclare*** τότε θα εμφανίσει λάθος διαφορετικά θα ξανά χρησιμοποιήσει την λεκτική μονάδα.
- **def varlist()** → αν υπάρχουν περισσότερες από μία μεταβλητές θα πρέπει να χωρίζονται με κόμμα.
- **def subprograms()** → εάν η λεκτική μονάδα είναι function τότε γεμίζουμε την λίστα ***isFunction*** με ***True*** διαφορετικά με ***False***. Καλώ την συνάρτηση ***func()***. Σε περίπτωση που δεν είναι ούτε συνάρτηση και ούτε διαδικασία ξανά χρησιμοποιώ την λέξη.
- **def func()** → αν η λεκτική μονάδα είναι ***id*** τότε καλείται η συνάρτηση **funcbody()** διαφορετικά βγάλει λάθος.
- **def funcbody()** → καλούνται οι συναρτήσεις ***formalpars(), block()***.

- **def formalpars()** → αν η λεκτική μονάδα είναι **(** τότε πηγαίνουμε στον κανόνα **formalparlist()** διαφορετικά βγάζουμε μήνυμα ότι λείπει το σύμβολο **)**.
- **def formalparlist()** → καλούμε την συνάρτηση ***formalparitem(0)*** με όρισμα 0 που σημαίνει ότι την πρώτη φορά που θα καλεστεί δεν θέλουμε να βγάλει σφάλμα, ενώ με όρισμα 1 σημαίνει ότι υπάρχει σφάλμα. Θέλουμε ουσιαστικά να καλύψουμε και την περίπτωση του ***ε***.
- **def formalparitem(errorFlag)** →θέλουμε μπροστά από ένα ***id*** να υπάρχει η έκφραση ***in ή inout*** που δηλώνει αν η μεταβλητή περαστεί με αναφορά ή όχι.
- **def sequence()** →οι εντολές θέλουμε να χωρίζονται με **;**.
- **def bracketsSeq()** →τα statement ξεκινάνε και τελειώνουμε με ***{}***. Όταν δούμε το σύμβολο ***}*** τότε αφαιρούμε από την λίστα ***isWhile*** το τελευταίο στοιχείο γιατί έχει τελειώσει το συγκεκριμένο statement.
- **def brackOrStat()** →μπορεί να έχει ***bracketsSeq() ή statement()***. Όταν τελειώσει το statement τότε διαγράφω το τελευταίο στοιχείο από τη λίστα ***isWhile***.
- **def statement()** →αναλόγως με την λεκτική μονάδα που διάβασα διαλέγουμε ποια περίπτωση είναι και εκτελείται η αντίστοιχη συνάρτηση.
- **def assignmentStat()** →αν διάβασα μεταβλητή τότε θα πρέπει να ακολουθεί καταχώρηση με το σχετικό σύμβολο ανάθεσης διαφορετικά βγάζει λάθος.
- **def ifStat()** → ελέγχει την σωστή σύνταξη της εντολής if.
- **def elsePart()** →ελέγχει την σωστή σύνταξη της εντολής else.
- **def whileStat()** →ελέγχει την σωστή σύνταξη της εντολής while.
- **def selectStat()** → εάν η λεκτική μονάδα που ακολουθεί είναι αριθμός ελέγχω με μία μεταβλητή ***expectedConst*** αν οι αριθμοί είναι με την σειρά και αν ακολουθεί η :. Αν υπάρχει το ***default :*** τότε είναι και αυτό εντάξει όμως αν δεν ισχύει κάτι από αυτά τότε βγάζει λάθος. Επίσης, αφαιρείται και το τελευταίο στοιχείο από την λίστα ***isWhile***.
- **def doWhileStat()** →ελέγχει την σωστή σύνταξη της εντολής do while.
- **def exitStat()** → θέλουμε το exit να βρίσκεται μόνο μέσα σε while. Οπότε αν αν περιέχει False η λίστα ή είναι κενή σημαίνει ότι το exit δεν βρίσκεται μέσα σε while.
- **def returnStat()** →Το return θέλουμε να βρίσκεται μόνο μέσα σε συνάρτηση οπότε ελέγχω την λίστα ***isFunction***. Αν η τελευταία θέση είναι False τότε σημαίνει ότι δεν βρίσκομαι σε συνάρτηση και συναντάω return κάτι το οποίο είναι λάθος και εμφανίζω σχετικό μήνυμα λάθους.
- **def printStat()** → ελέγχω αν η έκφραση που θέλω να εκτυπώσω βρίσκεται μέσα σε παρενθέσεις αν όχι εμφανίζω λάθος.
- **def callStat()** →εάν έχω δει μεταβλητή τότε καλώ την συνάρτηση ***actualPars()*** διαφορετικά βγάζω σφάλμα.
- **def actualPars()** →αν έχω δει **(** τότε καλώ την συνάρτηση ***actualparlist()*** και έπειτα ελέγχω για ***)***. Σε διαφορετική περίπτωση που δε δω ***(ή)*** εμφανίζω σφάλμα.

- **def actualparlist()** → αρχικά καλώ την συνάρτηση ***actualparitem με όρισμα 0*** ώστε να μην μου βγάλει κάποιο σφάλμα. Διαβάζω έπειτα χωρισμένα με κόμμα τις μεταβλητές καλώντας τώρα την συνάρτηση ***actualparitem με όρισμα 1***.
- **def actualparitem(errorFlag)** → ελέγχουμε ότι μετά το ***In*** θα πρέπει να υπάρχει έκφραση και μετά το ***inout*** θα πρέπει να υπάρχει μεταβλητή.
- **def condition()** → καλώ την συνάρτηση ***boolterm()***. Δηλαδή μέσα στις συνθήκες να υπάρχουν περιπτώσεις με ***or*** και ***and***.
- **def boolterm()** → καλώ την ***boolfactor()*** και ελέγχω για να υπάρχει το ***and*** μεταξύ συνθηκών.
- **def boolfactor()** → ελέγχω αν η λεκτική μονάδα είναι ***not.*** Αν είναι τότε μία τοπική μεταβλητή ***isNot*** γίνεται True διαφορετικά γίνεται False και ξανά χρησιμοποιείται. Έπειτα ελέγχω τις περιπτώσεις της γραμματικής.
- **def expression()** → καλώ την ***optionalSign(), term()***. Αν ο ***addOper()*** επιστρέψει 1 σημαίνει ότι είναι + ή – και καλεί τότε ξανά το ***term()***. Διαφορετικά ξανά χρησιμοποιείται.
- **def term()** → καλώ την ***factor()***. Αν ο ***mulOper()*** επιστρέψει 1 σημαίνει ότι είναι * ή / και τότε ξανά καλώ την ***factor()*** ενώ αν δεν ήτανε * ή / θα το ξανά χρησιμοποιούσα το σύμβολο.
- **def factor()** → μπορώ να έχω είτε σταθερά, είτε έκφραση, είτε μεταβλητή.
- **def idtail()** → αν έχω **(** τότε καλώ ***actualparlist()*** και θέλω να κλείνει με **)**. Διαφορετικά ξανά χρησιμοποιώ την λεκτική μονάδα.
- **def relationalOper()** → αν ο χαρακτήρας που έχω σαν λεκτική μονάδα τελεστής σύγκρισης τότε επιστρέφω 1 διαφορετικά 0 και την χρησιμοποιώ πάλι.
- **def addOper()** → αν η λεκτική μου μονάδα είναι + ή – τότε επιστρέφω 1 διαφορετικά 0.
- **def mulOper()** → αν η λεκτική μου μονάδα είναι * ή / τότε επιστρέφω 1 διαφορετικά 0.
- **def optionalSign()** → ουσιαστικά ελέγχεται αν υπάρχει πρόσημο.
- **def id(word)** → ελέγχω αν η λεκτική μονάδα μου είναι id. Δηλαδή, αν ανήκει σε κάποιες από τις δεσμευμένες λέξεις ή σε κάποια από τις λίστες επιστρέφω 0 διαφορετικά είναι σωστό όνομα μεταβλητής και επιστρέφω 1.

Τέλος υπάρχει και μία ακόμα συνάρτηση, η **def error(message)** η οποία λαμβάνει σαν όρισμα το λάθος που θέλω να εμφανίσω και εκτυπώνει το λάθος σε ποια γραμμή και στήλη βρίσκεται. Τερματίζει την μεταγλώττιση αν βρει κάποιο λάθος.

3.0 3η φάση – Ενδιάμεσος κώδικας

Η παραγωγή του ενδιάμεσου κώδικα βασίζεται στο συντακτικό δέντρο. Ο ενδιάμεσος κώδικας είναι ένα σύνολο από τετράδες. Η κάθε τετράδα έχει μπροστά της έναν αριθμό

που την 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.

Οι τελεστές αριθμητικών πράξεων:

- Έχουμε τετράδες της μορφής op, x, y, z
- op → +, -, *, /
- x, y → μπορεί να είναι ονόματα μεταβλητών ή αριθμητικές σταθερές
- z → μπορεί να είναι όνομα μεταβλητής
- Ο τελεστής op εφαρμόζεται στα x, y και το αποτέλεσμα εκχωρείται στο z
- πχ. +, a, 2, c → c = a + 2

Ο τελεστής εκχώρησης:

- Έχουμε τετράδες της μορφής :=, x, _, z
- x → μπορεί να είναι όνομα μεταβλητής ή αριθμητική σταθερά
- z → μπορεί να είναι όνομα μεταβλητής
- πχ. :=, 3, _, z → z := 3

Ο τελεστής άλματος:

- Έχει τετράδες της μορφής jump, _, _, label
- Σημαίνει ότι θα μεταβώ στην ετικέτα με αριθμό label, το label αυτό θα πρέπει σίγουρα να υπάρχει

Τελεστής άλματος χωρίς συνθήκη:

- Έχει τετράδες της μορφής relop, x, y, z
- Το relop μπορεί να είναι <, >, <>, <=, >=
- Και σημαίνει ότι αν ισχύει x relop y τότε πήγαινε στο label z

Αρχή και τέλος ενότητας:

- Όταν ξεκινάει το αρχικό μας πρόγραμμα είτε ένα block(συνάρτηση ή διαδικασία) τότε η τετράδα που πρέπει να εισάγουμε είναι begin_block, name, _, _
- Όταν τελειώσει ένα block είτε της συνάρτησης είτε του κυρίως προγράμματος πρέπει να εισάγουμε την τετράδα end_block, name, _, _
- Όμως, όταν τελειώσει το κύριο πρόγραμμα δηλαδή το block που ξεκινάει πάνω πάνω μετά το όνομα του κυρίως προγράμματος πριν το end_block πρέπει να εισάγουμε την τετράδα halt, _, _, _

Συναρτήσεις – Διαδικασίες:

- Η τετράδα που έχουμε όταν καλούμε μία συνάρτηση ή διαδικασία είναι της μορφής call, name, _, _

- Όταν η συνάρτηση αυτή επιστρέφει κάτι τότε η τετράδα της είναι της μορφής ret, x, _, _
- Όταν έχουμε και παραμέτρους τότε η μορφή της τετράδας είναι par, x, m, _ όπου χ είναι η παράμετρος της συνάρτησης και το m είναι ο τύπος, ο οποίος μπορεί να είναι CV όταν περνιέται με τιμή, REF όταν περνιέται με αναφορά και RET όταν επιστρέφει τιμή συνάρτησης
- Για παράδειγμα όταν βρούμε μία κλήση συνάρτησης πρώτα στον ενδιάμεσο κώδικα πρέπει να γράψουμε τις τετράδες που αφορούν τις παραμέτρους και αν επιστρέφετε τιμή και μετά την τετράδα του call
- Η δική μας γλώσσα υποστηρίζει και τις περιπτώσεις που σαν όρισμα όταν καλούμε μία συνάρτηση μπορεί να είναι η κλήση μίας άλλης συνάρτησης

Για να μπορέσουμε να υλοποιήσουμε τον ενδιάμεσο κώδικα είναι απαραίτητη η ύπαρξη κάποιων βοηθητικών συναρτήσεων:

nextquad() → καλώντας την συνάρτηση αυτή θα μας επιστραφεί ένας αριθμός ο οποίος δείχνει στην επόμενη τετράδα που πρόκειται να παραχθεί
genquad(op, x, y, z) → καλώντας την συνάρτηση αυτή θα δημιουργηθεί η επόμενη τετράδα (op, x, y, z)
newtemp() → καλώντας την συνάρτηση αυτή δημιουργείται και επιστρέφεται μία νέα προσωρινή μεταβλητή της μορφής T1, T2, …
emptylist() → δημιουργεί μία κενή λίστα ετικετών τετράδων
makelist(x) → δημιουργεί μία λίστα ετικετών τετράδων που περιέχει μόνο το x
merge(list1, list2) → δημιουργεί μία λίστα ετικετών τετράδων από την συνένωση των λιστών list1 και list2
backpatch(list, z) → η list αποτελείται από δείκτες σε τετράδες των οποίων το τελευταίο τελούμενο δεν είναι συμπληρωμένο. Διατρέχουμε μία μία αυτές τις τετράδες και συμπληρώνουμε με την ετικέτα z

3.1 Κώδικας: παραγωγή ενδιάμεσου κώδικα

Έχω φτιάξει κάποιες βασικές συναρτήσεις που χρειάζονται για την παραγωγή του ενδιάμεσου κώδικα. Ο κώδικας για τον ενδιάμεσο κώδικα έχει προστεθεί σε διάφορα σημεία του κώδικα του συντακτικού αναλυτή. Οπότε πιο κάτω θα γράψω σε ποιες συναρτήσεις έχει προστεθεί κώδικας και για ποιον λόγο. Σε αυτό το σημείο, δηλαδή της παραγωγής του ενδιάμεσου κώδικα θα κάνουμε και σημασιολογική ανάλυση. Στην δική μας γλώσσα Ciscal πρέπει σε αυτό το σημείο να ελέγχουμε για:

α) κάθε συνάρτηση έχει μέσα της τουλάχιστον ένα return
β) δεν υπάρχει return έξω από συνάρτηση (σε διαδικασία ή το κυρίως πρόγραμμα)
γ) υπάρχει exit μόνο μέσα σε βρόχους do_while

δ) στο SELECT οι επιλογές ξεκινούν από το 1 και αυξάνονται κατά 1

Καθολικές μεταβλητές που έχουν χρησιμοποιηθεί για την παραγωγή του ενδιάμεσου κώδικα είναι οι λίστες “***sessionNames***”, “***exitList***”, “***callList***”, “***nextquad = 90***” τα labels των τετράδων ξεκινάνε από το 100 και αυξάνουν κατά 10, “***newtemp=-1***” οι προσωρινές μεταβλητές είδαμε ότι είναι της μορφής T1, T2, T3, ... οπότε η μεταβλητή αυτή βρίσκει τον αριθμό που συνοδεύει το T , “***quads = {}***” είναι ένα λεξικό όπου είναι της μορφής {nextquad:[τετράδα]}.

- **def genquad(op, x, y, z)** → αυξάνει την μεταβλητή nextquad κατά 10 και αποθηκεύει την τετράδα στο λεξικό quads. Επιστρέφει την τετράδα αυτή.
- **def nextQuad()** → Επιστρέφει την μεταβλητή nextquad.
- **def newTemp()** → Αυξάνει την μεταβλητή newtemp κατά 1 και επιστρέφει την μεταβλητή αυτή.
- **def emptyList()** → Επιστρέφει μια τετράδα η οποία είναι κενή.
- **def makeList(x)** → Επιστρέφει μία λίστα με περιεχόμενο το x.
- **def merge(list1, list2)** → Παίρνει σαν όρισμα δυο λίστες και επιστρέφει την μία λίστα και από πίσω προσθέτει και την άλλη.
- **def backpatch(list, z)** → Το όρισμα list περιέχει τους αριθμούς των τετράδων που πρέπει να συμπληρώσουμε το τελευταίο τους πεδίο με την τιμή z. Οπότε διατρέχω την λίστα list και ελέγχω αν ο αριθμός που έχει περιέχετε σαν κλειδί μέσα το λεξικό quads, αν υπάρχει τότε βάζω την τελευταία θέση της τετράδας την τιμή z διαφορετικά βγάζω μήνυμα λάθους ότι η αντίστοιχη ετικέτα που ψάχνει δεν υπάρχει κάτι το οποίο δεν θα έπρεπε να ισχύει.
- **def isTemp(variable)** → Δέχεται σαν όρισμα μία μεταβλητή και ελέγχει αν είναι προσωρινή, αν ναι τότε επιστρέφει True σε αντίθετη περίπτωση επιστρέφει False.

Μέσα από την συνάρτηση “**def writeQuadsToFile()**” γράφω στο αρχείο “***test.int***” τις τετράδες οι οποίες είναι μέσα στο λεξικό quads.

Τώρα θα διατρέξω τον κώδικα από πάνω μέχρι κάτω ώστε να δω σε ποια σημεία έχω προσθέσει τον αντίστοιχο κώδικα που χρειάζεται:

- **def program()** → Είναι η στιγμή που ξεκινάει το block του κυρίους προγράμματος. Οπότε βάζω μέσα στην λίστα “***sessionNames***” το όνομα του προγράμματος και προσθέτω την τετράδα “***"begin_block", token, "_", "_"***” εφόσον ξεκινάει το block.
- **def block(start)** → Ελέγχω αν το μέγεθος του “***sessionNames***” είναι ίσο με το 1, αν και σημαίνει ότι μέσα στην λίστα “***sessionNames***” έχει μείνει το όνομα του κύριου προγράμματος και έχω φτάσει στο τέλος οπότε πρώτα πρέπει να προσθέσω την τετράδα “***"halt", "_", "_", "_"***” και έπειτα την τετράδα “***"end_block", sessionNames[-1], "_", "_"***”.

- **def func(rtype, start)** → Σημαίνει ότι έχω διαβάσει μία νέα συνάρτηση ή διαδικασία οπότε θα προσθέσω το όνομα της στην λίστα “***sessionNames***” και θα προσθέσω και την τετράδα “***"begin_block", token, "_", "_"***”.
- **def assignmentStat()** → Είμαστε στην περίπτωση που έχουμε κάποια καταχώρηση να κάνουμε, οπότε για αρχή χρειάζομαι να βρω το όνομα της μεταβλητής που θα καταχωρήσω το αποτέλεσμα και αυτό το κάνω ξανά χρησιμοποιώντας το token που διάβασα πριν. Μετά αφού διαβάσω το σύμβολο της εκχώρησης “***:=***” καταχωρώ σε μία μεταβλητή με όνομα ***E_place*** αυτό που θα μου γυρίσει από το ***expression()*** που θα είναι ένα ***T_i***. Έπειτα βάζω την τετράδα “***":=", E_place, "_", myId***” μέσα στο λεξικό με τα υπόλοιπα quads.
- **def ifStat()** → Αρχικά ενώ έχω διαβάσει την παρένθεση “***(***” και μετά από αυτό θα ακολουθεί μία συνθήκη, οπότε επιστρέφω από την συνάρτηση ***condition()*** μία λίστα της μορφής ***B = [B_True, B_False]***(το τι περιέχει θα το δούμε όταν μιλήσουμε για την συνάρτηση condition()). Ξεχωρίζω το περιεχόμενο που μου έχει επιστραφεί σε 2 λίστες την ***B_True = B[0]*** και την ***B_False = B[1]***. Όταν διαβάσω τον χαρακτήρα “***)***” δηλαδή κλείνει το περιεχόμενο της συνθήκης καλώ την ***backpatch*** ώστε να πάω και να γεμίσω το περιεχόμενο με την λίστα ***B_True*** με το επόμενο αριθμό ετικέτας γιατί σημαίνει ότι η συνθήκη μου είναι σωστή και οι επόμενες εντολές που πρέπει να εκτελέσω βρίσκομαι στο αμέσως επόμενο quad από αυτό που έχω διαβάσει, άρα γράφω την τετράδα “***backpatch(B_True, nextQuad())***”. Μετά μέσα σε μία λίστα “***ifList***” βάζω την επόμενη ετικέτα η οποία αντιστοιχεί έξω από τον βρόγχο, βάζω την τετράδα “***"jump", "_", "_", "_"***” αλλά πρέπει να συμπληρώσω το τελευταίο της στοιχείο, οπότε κάνω ***backpatch(ifList, nextQuad())*** για να γεμίσω τις θέσεις αυτές με τη σωστή ετικέτα.
- **def whileStat()** → Όταν έχουμε while τώρα, όταν θα ανοίξει η αγκύλη και θα την έχω διαβάσει θα περιέχει μία συνθήκη οπότε ***Bquad = nextQuad()*** κρατάω τον αριθμό της τετράδας που θα έρθει γιατί θα πρέπει να επιστρέψω σε αυτήν όταν τελειώσει η while και θα χρειαστεί να ελέγχω την συνθήκη ξανά. Κρατάω πάλι όπως και πριν σε 2 λίστες την ***B_True***, ***B_False*** τις τετράδες που πρέπει να πάω να συμπληρώσω μετά. Όταν διαβάσω και τον τελευταίο χαρακτήρα που τελειώνει η συνθήκη κάνω ***backpatch(B_True, nextQuad())*** για τις τετράδες του B_True, όταν τελειώσει το block της while θα πρέπει να πάω ξανά πάλι πάνω στην while οπότε κάνω ***genquad("jump", "_", "_", Bquad)*** και μετά γεμίζω και τις τετράδες που είναι False.
- **def selectStat()** → Όταν βρίσκουμε select τώρα, όταν βρω ότι η επιλογή του select είναι 1 τότε κρατάω το token της ξανά, αν δεν είναι η πρώτη επιλογή μέσα στην μεταβλητή w παίρνω μία νέα προσωρινή μεταβλητή και βάζω την τετράδα “***":=", 0, "_", w***”, την λίστα “***exitSelectList***” την βάζω να είναι κενή, μέσα σε μία while διατρέχω και τις άλλες περιπτώσεις όταν βρω μία άλλη επιλογή κρατάω την τιμή της. Όταν διαβάσω τον χαρακτήρα “:” τότε φτιάχνω μία λίστα με όνομα

B_False και βάζω μέσα τον αριθμό της επόμενης τετράδας. Πρέπει να κάνω σύγκριση τώρα, θα εισάγω την τετράδα "<>***", myId, myConst, "_"*** αν ισχύει η ισότητα τότε πρέπει να πάω ***genquad(":=", 1, "_", w)***. Όταν τελειώσει το όλο block της στην λίστα "***exitSelectList***" βάζω τον αριθμό της επόμενης τετράδας και κάνω ***backpatch(B_False, nextQuad())***.

- **def doWhileStat()** → Κρατάω στο "***sizeExitList***" το μέγεθος της λίστας "***len(exitList)***", μετά κρατάω στο "***Dquad***" τον αριθμό της επόμενης τετράδας ώστε αν η συνθήκη είναι true να πρέπει να επιστρέψω εδώ ξανά. Στο τέλος θα φτάσω να συναντήσω το while και μία συνθήκη μέσα σε αυτό, όπως και τις προηγούμενες φορές όταν έχω συνθήκη κρατάω τις λίστες ***B_True*** και ***B_False.*** Την λίστα ***B_False*** την κάνω ***merge*** με την λίστα ***exitList***. Κάνω τα αντίστοιχα ***backpatch(B_True, Dquad)*** και ***backpatch(B_False, nextQuad())***.
- **def exitStat()** → Το exit πρέπει να υπάρχει μόνο μέσα στην do-while, οπότε αυτό σημαίνει ότι όταν το συναντήσω θα πρέπει να πάω με ένα jump εκτός του block οπότε βάζω μέσα στην λίστα "***exitList***" τον αριθμό της τετράδας που βρίσκομαι ώστε να πάω μετά να την συμπληρώσω με την σωστή ετικέτα για το jump. Γράφω την τετράδα "***"jump", "_", "_", "_"***".
- **def returnStat()** → Κάνω ***E_place = expression()***.
- def printStat() → Βάζω την τετράδα ***genquad("out", E_place, "_", "_")*** και στο "***E_place***" βρίσκεται μία προσωρινή μεταβλητή.
- **def actualPars()** → Σημαίνει ότι τώρα θα βρω παραμέτρους, άρα θα εισάγω την αντίστοιχη τετράδα για την τιμή της επιστροφής την ***genquad("par", newTemp(), "ret", "_")***.
- **def actualparitem(errorFlag)** → Πρέπει να ορίζουμε και τον τύπο της παραμέτρου in ή inout, μπορεί μετά από τον τύπο να ακολουθεί μία ακολουθία από πράξεις οπότε ελέγχω όταν έχω "***in***" αν από το "***expression()***" μου έχει επιστραφεί κάποια προσωρινή μεταβλητή, αν ναι τότε μέσα στην λίστα "***callList***" κρατάω όλη την τετράδα "***["par", E_place, "cv", "_"])***" αν δεν είναι προσωρινή μεταβλητή τότε την γράφω κατευθείαν με "***genquad("par", E_place, "cv", "_")***". Όταν έχω "***inout***" δεν μπορώ να έχω κάποια ακολουθία από πράξεις οπότε γράφω κατευθείαν την τετράδα "***genquad("par", token, "ref", "_")***".
- **def condition()** → Μέσα στην "***Q1***" κρατάω αυτό που μου επιστρέφει η συνάρτηση "***boolterm()***"(θα δούμε πιο μετά τι ακριβώς επιστρέφει), διαχωρίζω τις ***B_True*** και ***B_False***, έχουμε την περίπτωση που συναντάμε "***or***" οπότε όταν βρούμε "or" πρέπει να κάνουμε ***backpatch(B_False, nextQuad())***, μετά στην ***Q2*** κρατάμε το αποτέλεσμα της συνάρτησης "***boolterm()***" κάνουμε ***merge*** τις λίστες ***B_True*** και ***Q2[0]*** γιατί θα δείχνουν στο ίδιο αποτέλεσμα.
- **def boolterm()** → Καλούμε την "***boolfactor()***" και αυτό που μας επιστρέφει το διαχωρίζουμε σε True και False. Τώρα έχουμε την περίπτωση με το "a***nd***" η λογική είναι ίδια όπως πριν μόνο που τώρα θα κάνουμε merge τις λίστες False. Η συνάρτηση μας επιστρέφει "***[Q_True, Q_False]***".

- **def boolfactor()** → Είναι η περίπτωση του "***not***" που τώρα η λογική είναι ίδια όπως και πριν μόνο που αλλάζουν με ποια σειρά θα επιστρέψω τις λίστες, αν έχω "not" πρώτα θα επιστρέψω την False στην θέση της True και την True στην θέση της False. Μπορεί να συναντήσω κάποιον τελεστή σύγκρισης οπότε φτιάχνω τα jump και τις λίστες των True και False που χρειάζονται.
- **def expression()** → Σημαίνει ότι έχω ακολουθία από πράξεις, οπότε κρατάω σε προσωρινές μεταβλητές τα αποτελέσματα των πράξεων και γράφω τετράδα της μορφής "***genquad(addOperator, T1_place, T2_place, w)***", επιστρέφω την T1_place.
- **def term()** → Ακριβώς με την ίδια λογική όπως και στην ***expression*** τώρα κάνω το ίδιο για όταν έχω κάποιον ***mulOperator***.
- **def factor()** → Στην "***F_place***" κρατάω αυτό που μου επιστρέφει η "***expression()***".
- **def idtail()** → Διατρέχω την λίστα "***callList***" που είχα φτιάξει πιο πάνω και παίρνω τις τετράδες που είχε για όταν καλώ κάποια συνάρτηση ή διαδικασία και συμπληρώνω τις τετράδες με τις κατάλληλες τιμές ώστε να τις γράψω το λεξικό με τα υπόλοιπα quads.

4.0 Παραγωγή Κώδικα σε C

Στην φάση της παραγωγής κώδικα σε C, πρέπει να παράγω ένα αρχείο ****.c*** το οποίο θα λειτουργεί κανονικά και όταν το τρέχω με ***gcc -o *.exe *.c*** αν μου παραχθεί το αρχείο ****.exe*** το οποίο θα εκτελείται σωστά. Το αρχείο αυτό περιέχει μέσα μόνο εντολές ***goto***. Ένα παράδειγμα τέτοιου κώδικα είναι το ακόλουθο:

```
int main()
{
  int a,b,T_1,t,c;
  L_0:
  L_1: a=1; //(:=, 1,  , a)
  L_2: T_1=a+b; //(+, a, b, T_1)
  L_3: if (T_1<1) goto L_5; //(<, T_1, 1, 5)
  L_4: goto L_27; //(JUMP,  ,  , 27)
  L_5: if (b<5) goto L_7; //(<, b, 5, 7)
  L_6: goto L_27; //(JUMP,  ,  , 27)
  L_7: if (t==1) goto L_9; //(=, t, 1, 9)
  L_8: goto L_11; //(JUMP,  ,  , 11)
  L_9: c=2; //(:=, 2,  , c)
  L_10: goto L_16; //(JUMP,  ,  , 16)
  L_11: if (t==2) goto L_13; //(=, t, 2, 13)
  L_12: goto L_15; //(JUMP,  ,  , 15)
  L_13: c=4; //(:=, 4,  , c)
  L_14: goto L_16; //(JUMP,  ,  , 16)
  L_15: c=0; //(:=, 0,  , c)
  L_16: goto L_18; //(JUMP,  ,  , 18)
  L_17: goto L_26; //(JUMP,  ,  , 26)
  L_18: if (a==2) goto L_20; //(=, a, 2, 20)
  L_19: goto L_25; //(JUMP,  ,  , 25)
  L_20: if (b==1) goto L_22; //(=, b, 1, 22)
  L_21: goto L_24; //(JUMP,  ,  , 24)
  L_22: c=2; // (:=, 2,  , c)
  L_23: goto L_20; //(JUMP,  ,  , 20)
  L_24: goto L_25; //(JUMP,  ,  , 25)
  L_25: goto L_16; //(JUMP,  ,  , 16)
  L_26: goto L_2; //(JUMP,  ,  , 2)
  L_27: {}
}
```

Στην δική μας γλώσσα Ciscal ***όταν ο κώδικας περιέχει συναρτήσεις και διαδικασίες τότε ο ισοδύναμος κώδικας σε C δεν δουλεύει σωστά***. Δεν παράγεται ο κώδικας που περιέχετε μέσα και επίσης όταν συναντάω κλήση κάποιας συνάρτησης ή διαδικασίας επίσης αγνοείτε.

4.1 Κώδικας: παραγωγή κώδικα σε C

- **def makeCFile()** → Δημιουργώ ένα προσωρινό αρχείο σε C το "t***empTest.c***". Το αρχείο αυτό δεν περιέχει την γραμμή με τις δηλώσεις των μεταβλητών (***int a,b,T_1,t,c;***) γιατί ακόμα δεν ξέρουμε ποιες είναι οι μεταβλητές όλου του αρχείου μας και επίσης δεν μπορώ να πηγαίνω πάνω στο αρχείο και να προσθέτω καινούρια πράγματα. Οπότε έχω μία λίστα, την "l***istWithVariablesForInitializeAtTop***" στην οποία κρατάω όλες τις μεταβλητές που πρέπει να δηλώσω στην αρχή. Στην συνάρτηση ***def finalCFile()*** θα εξηγήσω περισσότερα για το πως παράγετε το τελικό ισοδύναμο αρχείο σε C. Αρχικά, καλώ την συνάρτηση ***writeMain()***. Έπειτα, διαβάζω ένα-ένα το αρχείο του ενδιάμεσου κώδικα που έχει παραχθεί. Για κάθε γραμμή που διαβάζω καλώ την συνάρτηση ***splitLine()*** από την οποία έχω μία λίστα με το σημαντικό περιεχόμενο του ενδιάμεσου κώδικα. Αρχίζουν όλοι οι έλεγχοι τώρα. Εάν, έχω ***halt*** τότε σημαίνει ότι έχω τελειώσει με το κύριο πρόγραμμα μου και θα πρέπει να καλέσω την συνάρτηση ***writeFinish()***. Ελέγχω αν το μέγεθος της λίστας που μου έχει επιστραφεί είναι ίση με ***6***, αν ναι τότε γράφω στο αρχείο την σωστή εντολή. Έχω μία μεταβλητή ***variableLCount*** η οποία είναι ***global*** και με την οποία ξέρω κάθε φορά τον αριθμό της ετικέτας L_ που πρέπει να προσθέτω. Ελέγχω αν υπάρχει κάποια μεταβλητή που πρέπει να προσθέσω μέσα στην λίστα για τις μεταβλητές αρχικοποίησης. Ελέγχω για τα σύμβολα **+, -, *, /,** και γράφω την αντίστοιχη εντολή. Στην περίπτωση που έχω <> τότε θα πρέπει η εντολή να περιέχει το != γιατί αυτό αναγνωρίζει η γλώσσα C. Στην περίπτωση που βρω <, >, <=, >= σημαίνει ότι η εντολή που θα γράψω θα περιέχει if. Όταν βρω jump τότε θα πρέπει να προσθέσω κάποιο goto L_ . Όταν βρω out τότε θα πρέπει να γράψω την εντόλη της C, ***printf("%d", variable);***. Όταν συναντάμε σκέτο = σημαίνει ότι το χρησιμοποιούμε σε κάποια temp μεταβλητή. Όταν βρω ***begin_block*** τότε έχω μία μεταβλητή ***intoBlockCounter*** που την αυξάνω +1 κάθε φορά που συναντάω ένα block. Αφαιρώ -1 όταν βγαίνω από αυτό. Τώρα υπάρχουν κάποιες τιμές οι οποίες αφορούν σε συναρτήσεις, παραμέτρους, κλήση διαδικασίας. Οπότε σε αυτές τις περιπτώσεις δεν γράφω κάτι στο αρχείο απλά αυξάνω την μεταβλητή για να βρίσκεται στο σωστό σημείο. Σε όλους τους ελέγχους πιο πάνω στην αρχή ελέγχω αν βρίσκομαι μέσα σε συνάρτηση ώστε να αγνοήσω το να γράψω στο αρχείο.

- **def writeMain(cFile)** → Γράφω κάποια υποχρεωτικά πεδία που πρέπει να υπάρχουν σε κάθε πρόγραμμα C. Δηλαδή, την δήλωση ***#include <stdio.h>***, και τον τρόπο που ξεκινάνε τα προγράμματα σε C. Δηλαδή, ***int main(){*** και ***L_0:***.

- **def writeFinish(cFile, variableLCount)** → Αυτή η συνάρτηση καλείται όταν συναντήσουμε στον ενδιάμεσο κώδικα “***halt***”. Οπότε γράφω στο αρχείο μου την τελευταία ετικέτα ***L_number:{}*** }.

- **def splitLine(line)** → Παίρνει γραμμές ενδιάμεσου κώδικα και παίρνω τα πεδία που περιέχει. Το κάνω split με βάση την ***:*** αλλά επειδή υπάρχει και σύμβολο εκχώρησης τιμής ***:=*** στην περίπτωση αυτή και μόνο επιστρέφει λίστα με 6 θέσεις αντί για 5 που επιστρέφει όλες τις άλλες φορές.

- **def finalCFile()** → Ανοίγω το αρχείο “***tempTest.c***”. Ξεκινάω και γράφω μία-μία τις γραμμές του αρχείο “***tempTest.c***” αλλά όταν διαβάσω την ***3η γραμμή*** θα πρέπει πριν την γράψω να προσθέσω τις μεταβλητές που πρέπει να έχουν ορισθεί. Αφού γράψω αυτήν την γραμμή συνεχίζω κανονικά και γράφω ότι άλλο έχει το αρχείο “***tempTest.c***” με την σειρά. Το τελικό αρχείο που τα γράφω είναι το “***test.c***”.

Στην συνέχεια δείχνω ένα παράδειγμα που δείχνει ένα αρχείο Ciscal, ποιος είναι ο παραγώμενος κώδικας σε C από έναν ενδιάμεσο κώδικα,

Αρχείο Ciscal

```
program Max{
	declare x, y enddeclare
	x := 2000;
	y := 1000;

	if(x >= y){
		print(x)
	}
	else{
		print(y)
	}
}
```

Ενδιάμεσος κώδικας

```
100:['begin_block', 'Max', '_', '_']
110:[':=', 2000, '_', 'x']
120:[':=', 1000, '_', 'y']
130:['>=', 'x', 'y', 150]
140:['jump', '_', '_', 170]
150:['out', 'x', '_', '_']
160:['jump', '_', '_', 180]
170:['out', 'y', '_', '_']
180:['halt', '_', '_', '_']
190:['end_block', 'Max', '_', '_']
```

Ισοδύναμος κώδικας σε C

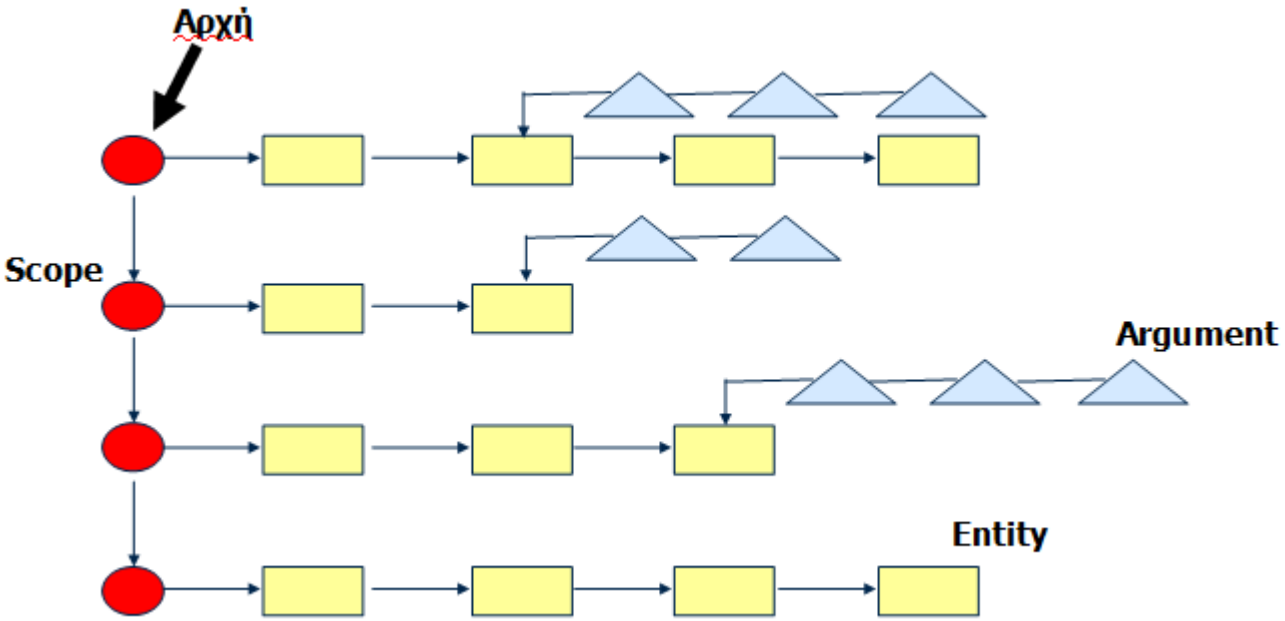
```
#include <stdio.h>
int main(){
	int y, x;
	L_0:
	L_110: x = 2000;
	L_120: y = 1000;
	L_130: if (x >= y) goto L_150;
	L_140: goto L_170;
	L_150: printf("%d", x);
	L_160: goto L_180;
	L_170: printf("%d", y);
	L_180: {}
}
```

```
C:\Users\Chryssa\Desktop\compilersTest\phase2>gcc -o example1.exe test.c

C:\Users\Chryssa\Desktop\compilersTest\phase2>example1.exe
2000
C:\Users\Chryssa\Desktop\compilersTest\phase2>
```

5.0 4η φάση – Πίνακας συμβόλων

Σε αυτήν την φάση ουσιαστικά θέλουμε να δούμε το βάθος κάθε μεταβλητής, συνάρτησης, διαδικασίας ώστε να ψάχνουμε όταν για παράδειγμα κάνουμε μία κλήση συνάρτησης ή χρησιμοποιούμε κάπου μία μεταβλητή αν όντως μπορούμε να την χρησιμοποιήσουμε. Μία γενική εικόνα για το τι θέλουμε να κάνουμε είναι η ακόλουθη:



Δημιουργία Scope

Ουσιαστικά κάθε φορά που συναντάμε μία καινούρια συνάρτηση/διαδικασία πρέπει να φτιάξουμε ένα scope. Όταν ξεκινάμε το κυρίως μας πρόγραμμα και για αυτό φτιάχνουμε ένα scope. Για κάθε scope πρέπει να κρατάμε κάποια πράγματα:

- **pointer Entity** (ουσιαστικά είναι μία από Entities, για παράδειγμα όταν έχω μία συνάρτηση σαν entity θα κρατήσω τις μεταβλητές που ορίζονται σε αυτήν και αυτές που χρησιμοποιεί)
- **int nestingLevel**, όπου είναι το βάθος φωλιάσματος δηλαδή το πόσο απέχει από την αρχή όλων των scopes που είναι από πάνω του
- **pointer Scope**

Δημιουργία Entity

Κάθε scope έχει μία λίστα από entities. Για παράδειγμα αν θεωρήσουμε ένα scope ότι είναι το κύριο πρόγραμμα τότε σαν Entities θα έχει τις μεταβλητές του. Λογικά θα περιέχει και κάποιες συναρτήσεις/διαδικασίες και αυτές θα θεωρηθούν σαν Entities αλλά μετά εφόσον τελειώσουμε με το scope του συγκεκριμένου scope θα πρέπει να εισάγουμε τις συναρτήσεις/διαδικασίες που θα συναντήσουμε σαν Entities και σαν scopes. Κάθε entity πρέπει να κρατάει και κάποιες μεταβλητές οι οποίες αναφέρονται παρακάτω:

- **char[] name**, το όνομα του entity
- **τύπος**, τον τύπο του entity δηλαδή αν είναι
  - **μεταβλητή**
    - **int type**, κρατάω τον τύπο της, βέβαια στην δική μας γλώσσα έχουμε μόνο integers οπότε δεν χρειάζεται να κρατάμε αυτό το πεδίο
    - **int offset**, κρατάμε και την απόσταση από την κορυφή της στοίβας
  - **συνάρτηση**
    - **int type**, τον τύπο της συνάρτησης
    - **int startQuad**, τον αριθμό που έχει η τετράδα που δείχνει ότι τώρα ξεκινάει η συνάρτηση
    - **list argument**, μία λίστα με τις παραμέτρους που έχει
    - **int framelength**, μήκος εγγραφήματος δραστηριοποίησης
  - **σταθερά**
    - **char[] value**, την τιμή που έχει η σταθερά
  - **παράμετρος**
    - **int parMode**, τον τρόπο που έχει περαστεί η μεταβλητή, με τιμή ή με αναφορά
    - **int offset**, απόσταση από την κορυφή της στοίβας
  - **προσωρινή μεταβλητή**

- **int offset**, απόσταση από την κορυφή της στοίβας

- **pointer entity next**, δείκτη για το επόμενο entity

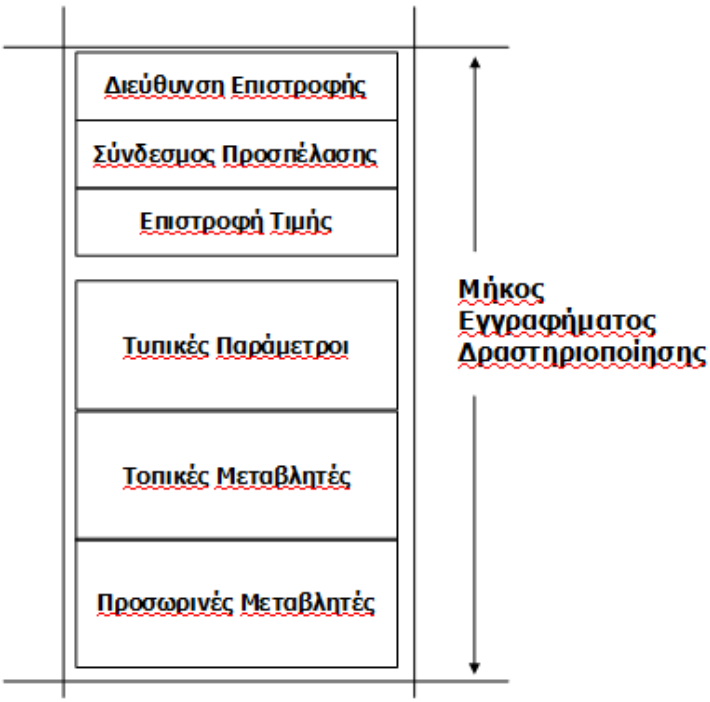
Δημιουργία argument

Δημιουργούμε ένα argument όταν για παράδειγμα έχουμε μία συνάρτηση σαν Entity και αυτή η συνάρτηση έχει κάποιες παραμέτρους, αυτές οι παράμετροι θα δημιουργηθούν σαν arguments. Για ένα argument είναι απαραίτητες κάποιες παράμετροι για να το εκφράσουν αυτές είναι οι ακόλουθες:

- **int parMode**, που δείχνει τον τρόπο περάσματος της μεταβλητής δηλαδή αν είναι με τιμή η αναφορά

- **int type**, τον τύπο της

- **pointer argument next**, έναν δείκτη που δείχνει στο επόμενο argument



Εγγράφημα δραστηριοποίησης
Δημιουργείται για κάθε συνάρτηση από αυτήν που την καλεί, όταν αρχίζει η εκτέλεση της συνάρτησης ο δείκτης στοίβας μεταφέρεται στην αρχή του εγγραφήματος δραστηριοποίησης, περιέχει πληροφορίες που χρησιμεύουν για την εκτέλεση και τον τερματισμό της συνάρτησης καθώς και πληροφορίες που σχετίζονται με τις μεταβλητές που χρησιμοποιεί, όταν τερματίζεται η συνάρτηση ο χώρος που καταλαμβάνει το εγγράφημα δραστηριοποίησης επιστρέφεται στο σύστημα.

Διεύθυνση επιστροφής: η διεύθυνση στην οποία θα μεταβεί η ροή του προγράμματος όταν ολοκληρωθεί η εκτέλεση της συνάρτησης
Σύνδεσμος Προσπέλασης: δείχνει στο εγγράφημα δραστηριοποίησης που πρέπει να αναζητηθούν μεταβλητές οι οποίες δεν είναι τοπικές αλλά η συνάρτηση έχει δικαίωμα να χρησιμοποιήσει
Επιστροφή τιμής: η διεύθυνση στην οποία θα γραφεί το αποτέλεσμα της συνάρτησης

όταν αυτό υπολογιστεί
Χώρος αποθήκευσης παραμέτρων συνάρτησης

- αποθηκεύεται η τιμή, αν πρόκειται για πέρασμα με τιμή
- αποθηκεύεται η διεύθυνση, αν πρόκειται για πέρασμα με αναφορά

Χώρος αποθήκευσης τοπικών μεταβλητών
Χώρος αποθήκευσης προσωρινών μεταβλητών

<u>Ενέργειες στον πίνακα συμβόλων</u>

Προσθήκη νέου Scope: όταν ξεκινάμε τη μετάφραση μιας νέας συνάρτησης
Διαφραφή 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
Προσθήκη νέου Entity
+ όταν συναντάμε δήλωση μεταβλητής
+ όταν δημιουργείται νέα προσωρινή μεταβλητή
+ όταν συναντάμε δήλωση νέας συνάρτησης
+ όταν συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης

Προσθήκη νέου Argument: όταν συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης
Αναζήτηση: μπορεί να αναζητηθεί κάποιο entity με βάση το όνομά του.
Η αναζήτηση ενός entity γίνεται ξεκινώντας από την αρχή του πίνακα και την πρώτη του γραμμή. Αν δε βρεθεί πηγαίνουμε στην επόμενη γραμμή έως ότου βρεθεί το entity ή τελειώσουν όλα τα entities οπότε επιστρέφουμε και μήνυμα λάθους. Αν με το ζητούμενο όνομα υπάρχει πάνω από ένα entity τότε επιστρέφουμε το πρώτο που θα συναντήσουμε

5.1: Κώδικας: Παραγωγή πίνακα συμβόλων

Έχω φτιάξει το Scope σαν κλάση, ουσιαστικά τα scopes θα είναι αντικείμενα τύπου scope. Έχω φτιάξει και για τον κάθε τύπο function, variable, parameter, temp μία κλάση. Πιο συγκεκριμένα,

- **class Scope(object)** → Αρχικοποιώ τα πεδία που έχω αναφέρει και πιο πάνω. Το offset το αρχικοποιώ στο 12. Έχω μία συνάρτηση "***def addEntity(self, entity)***" που προσθέτει κάθε φορά ένα καινούριο αντικείμενο Entity μέσα

στην λίστα με τα entities, φτιάχνω όλες τις get συναρτήσεις για να παίρνω και τα υπόλοιπα πεδία που έχω ορίσει στην ***__init__*** αν τα χρειαστώ.

- **class Variable(object)** → Είναι ένα entity που είναι μεταβλητή και κρατάω κάποια πεδία της.

- **class Function(object)** → Όταν έχω ένα entity που είναι συνάρτηση πρέπει να έχω και κάποια αντικείμενα Argument οπότε έχω και μία συνάρτηση "***def addArgument(self, argument)***" που προσθέτει αυτά τα argument.

- **class Temp(object)** → Είναι ένα entity που είναι προσωρινή μεταβλητή και κρατάω κάποια πεδία της.

- **class Parameter(object)** → Είναι ένα entity που είναι παράμετρος και κρατάω κάποια πεδία της.

Έχω σαν καθολικές μεταβλητές ***scopeList = []***, ***parModeList = []*** δύο λίστες που η πρώτη κρατάει αντικείμενα scopes και η δεύτερη το είδος της κάθε παραμέτρου. Αρχικά έχω φτιάξει κάποιες συναρτήσεις, θα πω πιο κάτω τις λειτουργίες τις και μετά θα αναφέρω σε ποια σημεία στον κώδικα που έχω γράψει στον συντακτικό αναλυτή σε ποια σημεία έχω προσθέσει κλήση των συναρτήσεων αυτών για την παραγωγή του πίνακα συμβόλων,

- **def addScope(name)** → Παίρνει το όνομα ενός νέου scope, φτιάχνει το σχετικό scope αντικείμενο και το προσθέτει στην scopeList.

- **def deleteScope()** → Διαγράφει το τελευταίο αντικείμενο scope που βρίσκεται μέσα στην scopeList.

- **def searchEntity(searchingEntity)** → Δέχεται σαν όρισμα ένα όνομα από entity, ψάχνω αν υπάρχει το όνομα του entity αυτού μέσα στα entities όλων των scopes, αν είναι function ο τύπος του entity τότε ψάχνω μέσα στην argument list αν υπάρχει και αν οι τύποι είναι σωστοί και με την σωστή σειρά. Επιστρέφω το entity σαν αντικείμενο και στο scope.

- **def searchEntityByName(entityName)** → Ψάχνω ένα entity με βάση το όνομα και όχι το αντικείμενο entity. Με την ίδια λογική όπως και στην από πάνω συνάρτηση και επιστρέφω το entity και το scope.

- **def searchFunctionByName(functionName)** → Ψάχνω ένα entity που είναι συνάρτηση με βάση το όνομα και όχι σαν αντικείμενο entity. Με την ίδια λογική όπως και στην από πάνω συνάρτηση και επιστρέφω το entity και το scope.

- **def searchDuplicate(searchingEntity)** → Η συνάρτηση αυτή ελέγχει αν ένα όνομα entity υπάρχει περισσότερες από 2 φορές μέσα στο πρόγραμμα και στο ίδιο βάθος φωλιάσματος.

Οι συναρτήσεις αυτές καλούνται σε διάφορα σημεία μέσα στον κώδικα,

- **def program()** → Όταν διαβάσω το όνομα του κυρίως προγράμματος πρέπει να φτιάξω ένα καινούριο Scope οπότε και αυτό κάνω με την εντολή "***addScope(token)***".

- **def block(start)** → Όταν τελειώνει ένα block, πρέπει να αφαιρέσω το τελευταίο scope για αυτό κάνω "***deleteScope()***".

- **def varlist()** → Όταν διαβάσω μία μεταβλητή ελέγχω αν πρώτα υπάρχει κάποια άλλη μεταβλητή με το ίδιο όνομα, αν όχι την προσθέτω στο τελευταίο scope σαν entity με την μορφή variable.

- **def func(rtype, start)** → Διαβάζω το όνομα μίας συνάρτησης, ελέγχω αν υπάρχει διπλότυπο "***searchDuplicate(Function(token, 0, "", 0))***", μετά βάζω το νέα entity με την μορφή συνάρτησης "***scopeList[-1].addEntity(Function(token, nextquad + 10, rtype, scopeList[-1].getOffset()))***" και πρέπει να την εισάγω και σαν scope "***addScope(token)***".

- **def formalparitem(errorFlag)** → Είμαστε στο σημείο που βρίσκουμε παραμέτρους, όταν δούμε την μορφή με την οποία περνάει η μεταβλητή την κρατάω στην μεταβλητή "***mode***", την προσθέτω σαν argument "***scopeList[-2].getEntityList()[-1].addArgument([token, mode])***" και στο αντίστοιχο entity προσθέτω ότι έχω συναντήσει ένα αντικείμενο με μορφή παραμέτρου "***scopeList[-1].addEntity(Parameter(token, mode, scopeList[-1].getOffset()))***".

- **def assignmentStat()** → είναι όταν μία εκχώρηση οπότε πρέπει να ελέγξω αν υπάρχει όντως η μεταβλητή στην οποία θα βάλω το αποτέλεσμα άρα κάνω "***searchEntity(Variable(myId, 0))***".

- **def selectStat()** → αντίστοιχα και εδώ όταν διαβάσω κάποια μεταβλητή θα πρέπει να ελέγξω αν υπάρχει "***searchEntity(Variable(myId, 0))***".

- **def callStat()** → Καλώ κάποια συνάρτηση, δημιουργώ μία προσωρινή "***tempFunction = Function(myId, 0, "", 0)***" και γεμίζω την argument λίστα της με τους τύπους των παραμέτρων που έχει όταν την καλεί "***tempFunction.setArgumentList(parModeList)***", ψάχνω αν υπάρχει αυτό το όνομα συνάρτησης "***searchEntity(tempFunction)***".

- **def actualparitem(errorFlag)** → ανάλογα με το είδος της κάθε παραμέτρου το βάζω στην λίστα “**parModeList**”, ψάχνω “***searchEntity(Variable(token, 0))***” αν υπάρχει κάποιο entity που είναι variable γιατί οι τιμές που βάζουμε όταν καλούμε μία συνάρτηση είναι συνήθως σταθεράς/μεταβλητές.

- **def idtail()** → Είναι η ίδια λογική όπως και στην “***callStat()***”.

- **def newTemp()** → Όταν δημιουργώ μία προσωρινή μεταβλητή, την προσθέτω και σαν temp entity στο τελευταίο scope “***scopeList[-1].addEntity(Temp("T_" + str(newtemp), scopeList[-1].getOffset()))***”.

6.0 6η φάση – Παραγωγή τελικού κώδικα

Αυτή είναι η τελευταία φάση του μεταγλωττιστή, η παραγωγή τελικού κώδικα σε αρχιτεκτονική MIPS. Τον τελικό κώδικα τον παράγουμε με βάση τον ενδιάμεσο κώδικα που έχουμε παράγει. Οι κύριες ενέργειες στην φάση αυτή είναι ότι οι μεταβλητές απεικονίζονται στην μνήμη(στοίβα) και το πέρασμα παραμέτρων και η κλήση συναρτήσεων.

6.1 Αρχιτεκτονική MIPS

Καταχωρητές που θα μας φανούν χρήσιμοι:
- **καταχωρητές προσωρινών τιμών**: $t0…$t7
- **καταχωρητές οι τιμές των οποίων διατηρούνται ανάμεσα σε κλήσεις συναρτήσεων**: $s0…$s7
- **καταχωτητές 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 t0=t1+t2

- **sub** $t0,$t1,$t2 t0=t1-t2

- **mul** $t0,$t1,$t2 t0=t1*t2

- **div** $t0,$t1,$t2 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**

6.2 Βοηθητικές συναρτήσεις

Είναι απαραίτητο να φτιάξω κάποιες βοηθητικές συναρτήσεις αυτές είναι οι παρακάτω,

- **gnvlcode** : μεταφέρει στον $t0 την διεύθυνση μιας μη τοπικής μεταβλητής από τον πίνακα συμβόλων βρίσκει πόσα επίπεδα επάνω βρίσκεται η μη τοπική μεταβλητή και μέσα από τον σύνδεσμο προσπέλασης την εντοπίζει,
  w $t0,-4($sp) **στοίβα του γονέα**
  όσες φορές χρειαστεί:
  lw $t0,-4($t0) **στοίβα του προγόνου που έχει τη μεταβλητή**
  add $t0,$t0,-offset **διεύθυνση της μη τοπικής μεταβλητής**

- **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,

1. **αν v είναι σταθερά**
   li $tr,v

2. **αν v είναι καθολική μεταβλητή – δηλαδή ανήκει στο κυρίως πρόγραμμα**
   lw $tr,-offset($s0)

3. **αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή**
   lw $tr,-offset($sp)

4. **αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον**
   lw $t0,-offset($sp)
   lw $tr,($t0)

5. **αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον**
   gnlvcode()
   lw $tr,($t0)

6. **αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον**
   gnlvcode()
   lw $t0,($t0)
   lw $tr,($t0)

- **storerv** : μεταφορά δεδομένων από τον καταχωρητή r στη μνήμη (μεταβλητή v), η σύνταξη της είναι storerv(r,v), διακρίνουμε περιπτώσεις,

1. **<u>αν v είναι καθολική μεταβλητή – δηλαδή ανήκει στο κυρίως πρόγραμμα</u>**
   sw $tr,-offset($s0)

2. **<u>αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή</u>**
   sw $tr,-offset($sp)

3. **<u>αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον</u>**
   lw $t0,-offset($sp)
   sw $tr,($t0)

4. **<u>αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον</u>**
   gnlvcode(v)
   sw $tr,($t0)

5. **<u>αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον</u>**
   gnlvcode(v)
   lw $t0,($t0)
   sw $tr,($t0)

Εντολές αλμάτων

- **jump, “_”, “_”, label**
  j label

- **relop(?),x,y,z**
  loadvr(x,1)
  loadvr(y,2)
  branch(?),$t1,$t2,z branch(?) : beq,bne,bgt,blt,bge,ble

Εκχώρηση

- **:=, x, “_”, z**
  loadvr(x,1)
  storerv(1,z)

Εντολές αριθμητικών πράξεων

- **op x,y,z**
  loadvr(x,1)

loadvr(y,2)
op $t1,$t1,$t2 op: add,sub,mul,div
storerv(1,z)

Εντολές εξόδου

- **out “_”, “_”, x**
  li $v0,1
  li $a0, x
  syscall

Επιστροφή τιμής συνάρτησης

- **retv “_”, “_”, x**
  loadvr(x,1)
  lw $t0,-8($sp)
  sw $t1,($t0)
  αποθηκεύεται ο x στη διεύθυνση που είναι αποθηκευμένη στην 3η θέση του εγγραφήματος δραστηριοποίησης

Παράμετροι συνάρτησης

- **πριν από την πρώτη παράμετρο, τοποθετούμε τον $fp να δείχνει στην στοίβα της συνάρτησης που θα δημιουργηθεί**
  add $fp,$sp,framelength

- **par,x,CV, _**
  loadvr(x,0)
  sw $t0, -(12+4i)($fp) όπου i ο αύξων αριθμός της παραμέτρου

- **par,x,REF, _**
  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
  add $t0,$sp,-offset
  sw $t0,-(12+4i)($fp)

- **par,x,REF, _**
  αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά
  lw $t0,-offset($sp)
  sw $t0,-(12+4i)($fp)

- **par,x,REF, _**
  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
  gnlvcode(x)
  sw $t0,-(12+4i)($fp)

- **par,x,REF, _**
  αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με τιμή
  gnlvcode(x)
  lw $t0,($t0)
  sw $t0,-(12+4i)($fp)

- **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
  add $t0,$sp,-offset
  sw $t0,-8($fp)

Κλήση συνάρτησης

- **call, _, _, f**
  αρχικά γεμίζουμε το 2ο πεδίο του εγγραφήματος δραστηριοποίησης της κληθείσας συνάρτησης 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

  - **αν καλούσα και κληθείσα έχουν το ίδιο βάθος φωλιάσματος, τότε έχουν τον ίδιο γονέα**
    lw $t0,-4($sp)
    sw $t0,-4($fp)

  - **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**
  add $sp,$sp,framelength

- **καλούμε τη συνάρτηση**
  jal f

- **και όταν επιστρέψουμε παίρνουμε πίσω τον δείκτη στοίβας στην καλούσα**
  add $sp,$sp,-framelength

- **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)

- **στην 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 Lmain

- **στη συνέχεια πρέπει να κατεβάσουμε τον $sp κατά framelength της main**
  add $sp,$sp,framelength

- **και να σημειώσουμε στον $s0 το εγγράφημα δραστηριοποίησης της main ώστε να έχουμε εύκολη πρόσβαση στις global μεταβλητές**
  move $s0,$sp

6.3 Κώδικας: παραγωγή τελικού κώδικα

Έχω φτιάξει τις βοηθητικές συναρτήσεις που έχω αναφέρει παραπάνω και κάποιες επιπλέον ώστε να μπορώ και να γράφω τον τελικό κώδικα στο αρχείο,

- **def gnvlcode(variable)** → Παίρνω σαν όρισμα μία μεταβλητή, κάνω αναζήτηση αυτού του entity με βάση το όνομα του variable, βρίσκω το ***currentNestingLevel*** και το ***scopeNestingLevel***. Γράφω στο τελικό αρχείο αυτό που πρέπει και μετά όσο το scope επίπεδο είναι μικρότερο του entity γράφω τις τελικές εντολές στο αρχείο.

- **def loadvr(v, r)** → Ελέγχω τις περιπτώσεις που αναφέρω παραπάνω και γράφω τις αντίστοιχες εντολές στο τελικό αρχείο.
- **def storerv(r, v)** → Κάνω ότι γράφω πιο πάνω και επειδή είναι απλές οι περιπτώσεις δεν θα τις αναφέρω αναλυτικά και εδώ.
- **def quadToFile(quad)** → Ελέγχω κάποιες περιπτώσεις ώστε να γράψω στο τελικό αρχείο τον αντίστοιχο κώδικα.
- **def writeCall(quad)** → Γράφω τις εντολές όταν συναντάω το call.
- **def writePar(quad)** → Γράφω τις εντολές για τον τελικό κώδικα όταν συναντάω παραμέτρους από την συνάρτηση.
- **def hasLocal(entity, local)** → Ελέγχει αν είναι τοπική μεταβλητή ένα entity που παίρνει σαν όρισμα και επιστρέφει true, false αντίστοιχα.
- **def hasArgument(entity, argumentName, parMode)** → Ελέγχει αν ένα entity έχει argument και είναι σωστά περασμένα.Επιστρέφει true και false αντίστοιχα.
- **def writeFinalFile(line)** → Παίρνει σαν όρισμα αυτά που θέλω να γράψω στο αρχείο για να μην καλώ συνέχεια το όνομα του αρχείου και το write.

7.0 Ολοκληρωμένο παράδειγμα

Παρακάτω έχω ένα ολοκληρωμένο παράδειγμα και όλα τα αποτελέσματα που παίρνω από τον compiler Που έχω φτιάξει. Το παράδειγμα θα περιέχει και διαδικασίες και συναρτήσεις, ο ενδιάμεσος κώδικας είναι ο σωστός που παράγεται, ο ισοδύναμος κώδικας σε C δεν είναι σωστός όπως μας έχει ζητηθεί άλλωστε ότι δεν θα παράξουμε κώδικα σε C όταν συναντάμε συναρτήσεις ή διαδικασίες και κλήσεις αυτών. Έστω το παρακάτω πρόγραμμα σε γλώσσα Cicsal,

```
program Example1{
	declare a1, a2, a3 enddeclare

	function speed(in number1, inout number2){
		declare y enddeclare

		if(number1 > number2){
			y := 10 * number1;
		}
		else{
			y := number2 * 10;
		};
		return(y)
	}

	procedure price(){
		if(a2 <> a3){
			a2 := 2 * a3
		}
	}

	a1 := (((30 - 5)*47)/7);
	a2 := (((23 + 67) - 50)/2);
```

```
        }

    }while(a1 <> 100);

    do{
        exit
    }while(2 <> 2)

}
```

```
    do{
        if(a1 > a2){
            a1 :=call speed(in 3, inout a1)
call price();
        }
        else{
            a1 := a1 + 30
```

Ο αντίστοιχος ενδιάμεσος κώδικας ο οποίος παράγεται είναι ο ακόλουθος,

```
100:['begin_block', 'Example1', '_', '_']
110:['begin_block', 'speed', '_', '_']
120:['>', 'number1', 'number2', 140]
130:['jump', '_', '_', 170]
140:['*', 10, 'number1', 'T_0']
150:[':=', 'T_0', '_', 'y']
160:['jump', '_', '_', 190]
170:['*', 'number2', 10, 'T_1']
180:[':=', 'T_1', '_', 'y']
190:['retv', 'y', '_', '_']
200:['end_block', 'speed', '_', '_']
210:['begin_block', 'price', '_', '_']
220:['<>', 'a2', 'a3', 240]
230:['jump', '_', '_', 270]
240:['*', 2, 'a3', 'T_2']
250:[':=', 'T_2', '_', 'a2']
260:['jump', '_', '_', 270]
270:['end_block', 'price', '_', '_']
280:['-', 30, 5, 'T_3']
290:['*', 'T_3', 47, 'T_4']
300:['/', 'T_4', 7, 'T_5']
310:[':=', 'T_5', '_', 'a1']
320:['+', 23, 67, 'T_6']
330:['-', 'T_6', 50, 'T_7']
340:['/', 'T_7', 2, 'T_8']
350:[':=', 'T_8', '_', 'a2']
360:['>', 'a1', 'a2', 380]
370:['jump', '_', '_', 450]
380:['par', 3, 'cv', '_']
390:['par', 'a1', 'ref', '_']
400:['par', 'T_9', 'ret', '_']
410:['call', 'speed', '_', '_']
420:[':=', 'T_9', '_', 'a1']
430:['call', 'price', '_', '_']
440:['jump', '_', '_', 470]
450:['+', 'a1', 30, 'T_10']
460:[':=', 'T_10', '_', 'a1']
470:['<>', 'a1', 100, 360]
480:['jump', '_', '_', 490]
490:['jump', '_', '_', 520]
500:['<>', 2, 2, 490]
510:['jump', '_', '_', 520]
520:['halt', '_', '_', '_']
530:['end_block', 'Example1', '_', '_']
```

Ο ισοδύναμος κώδικας που παράγεται σε C είναι ο ακόλουθος,

```
#include <stdio.h>
int main(){
    int T_10, T_9, a1, T_8, a2, T_3, T_6, T_7, T_4, T_5;
    L_0:
    L_280: T_3 = 30 - 5;
    L_290: T_4 = T_3 * 47;
    L_300: T_5 = T_4 / 7;
    L_310: a1 = T_5;
    L_320: T_6 = 23 + 67;
    L_330: T_7 = T_6 - 50;
    L_340: T_8 = T_7 / 2;
    L_350: a2 = T_8;
    L_360: if (a1 > a2) goto L_380;
    L_370: goto L_450;
    L_420: a1 = T_9;
    L_440: goto L_470;
    L_450: T_10 = a1 + 30;
    L_460: a1 = T_10;
    L_470: if (a1 != 100) goto L_360;
    L_480: goto L_490;
    L_490: goto L_520;
    L_500: if (2 != 2) goto L_490;
    L_510: goto L_520;
    L_520: {}
}
```

Το αποτέλεσμα όταν τρέξουμε τον κώδικα σε C είναι αυτό που ακολουθεί,

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>gcc -o test.exe test.c
test.c: In function 'main':
test.c:13:2: error: label 'L_380' used but not defined
```

Ο πίνακας συμβόλων πριν διαγράψω ένα scope είναι ο ακόλουθος,

['number1', 'param', 'cv', 12]
['number2', 'param', 'ref', 16]
['y', 'variable', 20]
['T_0', 'temp', 24]
['T_1', 'temp', 28]
None
['T_2', 'temp', 12]
None

Ο τελικός κώδικας που παράγεται για το συγκεκριμένο παράδειγμα είναι ο ακόλουθος,

j 90
110:120:lw $t1,-12($sp)
lw $t2,-16($sp)
bgt,$t1,$t2,140
130:j 170
140:li $t1,10
lw $t2,-12($sp)
mul$t1,$t1,$t2
sw $t1,-24($sp)
150:lw $t1,-24($sp)
sw $t1,-20($sp)
160:j 190
170:lw $t1,-16($sp)
li $t2,10
mul$t1,$t1,$t2
sw $t1,-28($sp)
180:lw $t1,-28($sp)
sw $t1,-20($sp)
190:lw $t1,-20($sp)
lw $t0,-8($sp)
sw $t1,($t0)
200:lw $ra,($sp)
jr $ra
210:220:lw $t1,-16($s0)
lw $t2,-20($s0)
bne,$t1,$t2,240
230:j 270
240:li $t1,2
lw $t2,-20($s0)
mul$t1,$t1,$t2
sw $t1,-12($sp)
250:lw $t1,-12($sp)
sw $t1,-16($s0)
260:j 270
270:lw $ra,($sp)
jr $ra
90: add $sp,$sp,52
move $s0,$sp
100:280:li $t1,30
li $t2,5
sub$t1,$t1,$t2
sw $t1,-24($s0)
290:lw $t1,-24($s0)
li $t2,47
mul$t1,$t1,$t2
sw $t1,-28($s0)
300:lw $t1,-28($s0)
li $t2,7
div$t1,$t1,$t2
sw $t1,-32($s0)
310:lw $t1,-32($s0)
sw $t1,-12($s0)
320:li $t1,23
li $t2,67
add$t1,$t1,$t2
sw $t1,-36($s0)
330:lw $t1,-36($s0)
li $t2,50
sub$t1,$t1,$t2
sw $t1,-40($s0)
340:lw $t1,-40($s0)
li $t2,2
div$t1,$t1,$t2
sw $t1,-44($s0)
350:lw $t1,-44($s0)
sw $t1,-16($s0)
360:lw $t1,-12($s0)
lw $t2,-16($s0)
bgt,$t1,$t2,380
370:j 450
380:390:400:410:add $fp,
$sp,24
li $t0,3
sw $t0,-(12+4)($fp)
add $t0,$sp,-12
sw $t0,-(12+4)($fp)
add $t0,$sp,-48
sw $t0,-8($fp)
lw $t0,-4($sp)
sw $t0,-4($fp)
add $sp,$sp,24
jal speed
add $sp,$sp,-24
sw $ra,($sp)
420:lw $t1,-48($s0)
sw $t1,-12($s0)
430:add $fp,$sp,24
lw $t0,-4($sp)
sw $t0,-4($fp)
add $sp,$sp,24
jal price
add $sp,$sp,-24
sw $ra,($sp)
440:j 470
450:lw $t1,-12($s0)
li $t2,30
add$t1,$t1,$t2
sw $t1,-52($s0)
460:lw $t1,-52($s0)
sw $t1,-12($s0)
470:lw $t1,-12($s0)
li $t2,100
bne,$t1,$t2,360
480:j 490
490:j 520
500:li $t1,2
li $t2,2
bne,$t1,$t2,490
510:j 520
520:530:

8.0 Tests ενδιάμεσου κώδικα

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py example.ci
Error: Line 36, Column 12, integer must be between [-32768, 32767]
```

α) κάθε συνάρτηση έχει μέσα της τουλάχιστον ένα return

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py example1.ci
Error: Line 13, Column 2, function should have at least one 'return'
```

β) δεν υπάρχει return έξω από συνάρτηση (σε διαδικασία ή το κυρίως πρόγραμμα)

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py ex.ci
Error: Line 3, Column 7, 'return' can only be used inside a function
```

γ) υπάρχει exit μόνο μέσα σε βρόχους do_while

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py ex.ci
Error: Line 3, Column 5, 'exit' can only be used inside 'do-while'
```

δ) στο SELECT οι επιλογές ξεκινούν από το 1 και αυξάνονται κατά 1

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py ex.ci
Error: Line 3, Column 11, constant or default expected
```

9.0 Tests πίνακα συμβόλων

α) κάθε μεταβλητή, συνάρτηση ή διαδικασία που έχει δηλωθεί να μην έχει δηλωθεί πάνω από μία φορά στο βάθος φωλιάσματος στο οποίο βρίσκεται

```
program ex{
        declare c enddeclare
        function a(){
                return(1)
        }
        function a(){
                return(2)
        }
}
```

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py ex.ci
Error: Line 6, Column 11, Duplicate of function "a"
```

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py example.ci
Error: Line 14, Column 15, Entity "s" not found
```

10.0 Tests τελικού κώδικα

α) κάθε μεταβλητή, συνάρτηση ή διαδικασία που χρησιμοποιείται έχει δηλωθεί και μάλιστα με τον τρόπο που χρησιμοποιείται

```
program ex{
        declare c enddeclare
        procedure a(in t, inout w){
                c := 10
        }
        c := call a(inout 6, in 2)
}
```

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py ex.ci
Error: Line 6, Column 10, wrong factor
```

β) οι παράμετροι με τις οποίες καλούνται οι συναρτήσεις και οι διαδικασίες είναι ακριβώς αυτές με τις οποίες έχουν δηλωθεί και με τη σωστή σειρά

```
program ex{
        declare c enddeclare
        function a(in t){
                return(1)
        }
        c := call a()
}
```

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py ex.ci
Error: Line 6, Column 10, wrong factor
```

```
program ex{
        declare c enddeclare
        function a(in t, inout w){
                return(1)
        }
        c := call a(inout 6, in 2)
}
```

```
C:\Users\Chryssa\Desktop\compilersTest\phase3>python met.py ex.ci
Error: Line 6, Column 10, wrong factor
```
