Μεταφραστές
Αναφορά Άσκησης Εξαμήνου:

Υλοποίηση μεταγλωττιστή για γλώσσα προγραμματισμού υψηλού επιπέδου

Μέλη ομάδας:

Αντωνίου Χριστόδουλος AM: 2641
Τσιούρη Αγγελική AM: 3354

ΜΕΡΟΣ 1ο: Εισαγωγή - Η γλώσσα Cimple

Το πρώτο βήμα στην υλοποίηση ενός μεταγλωττιστή είναι να προσδιορίσουμε την γλώσσα που θέλουμε να μεταγλωττίζει. Στην περίπτωση μας πρόκειται για μία απλοποιημένη και ελαφρώς τροποποιημένη εκδοχή της γλώσσας C, την γλώσσα **Cimple**. Το όνομα της προέρχεται από τη γλώσσα C, ενώ ηχεί όπως η λέξη simple, για να τονιστεί η απλότητα της. Ως μία εκπαιδευτική γλώσσα προγραμματισμού, οι δυνατότητες της είναι περιορισμένες σε σύγκριση με άλλες. Ωστόσο, η υλοποίηση του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται από δημοφιλείς γλώσσες. Τα αρχεία της Cimple έχουν κατάληξη ***.ci***. Ακολουθεί η λεπτομερής περιγραφή της γλώσσας.

Α. Λεκτικές μονάδες

Το **αλφάβητο της Cimple** αποτελείται από:

- Τα μικρά και κεφαλαία γράμματα της λατινικής αλφαβήτου: **`A, …, Z`** και **`a, …, z`**
- Τα αριθμητικά ψηφία: **`0, …, 9`**
- Τα σύμβολα των αριθμητικών πράξεων: **`+, -, *, /`**
- Τους τελεστές συσχέτισης: **`<, >, =, <=, >=, <>`**
- Το σύμβολο ανάθεσης: **`:=`**
- Τους διαχωριστές: **`;, “,”, :`**
- Τα σύμβολα ομαδοποίησης: **`[, ], (, ), {, }`**
- Το σύμβολο τερματισμού προγράμματος: **`.`**
- Και το σύμβολο διαχωρισμού σχολίων: **`#`**

Τα σύμβολα **`[, ]`** χρησιμοποιούνται στις λογικές παραστάσεις όπως τα σύμβολα **`(, )`** στις αριθμητικές παραστάσεις.

Οι **δεσμευμένες λέξεις** είναι:

```
program     declare
if          else        while
switchcase  forcase     incase    case     default
not         and         or
function    procedure   call      return   in       inout
input       print
```

Η Cimple υποστηρίζει μόνο ακέραιους αριθμούς ως τύπο δεδομένων. Οι ακέραιοι αποτελούνται από προαιρετικό πρόσημο και από μία ακολουθία αριθμητικών ψηφίων, ενώ οι

τιμές τους κυμαίνονται από $-(2^{32}-1)$ έως $2^{32}-1$. Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία (προαιρετικά), αρχίζοντας όμως από γράμμα. Οι δεσμευμένες λέξεις δεν μπορούν να χρησιμοποιηθούν ως αναγνωριστικά ενώ αναγνωριστικά με περισσότερους από 30 χαρακτήρες θεωρούνται λανθασμένα.

Οι λευκοί χαρακτήρες (tab, space, return) αγνοούνται και μπορούν να χρησιμοποιηθούν με οποιοδήποτε τρόπο χωρίς να επηρεάζεται η λειτουργία του μεταγλωττιστή εφόσον δεν βρίσκονται ανάμεσα σε δεσμευμένες λέξεις, αναγνωριστικά ή ακέραιους. Το ίδιο ισχύει και για τα σχόλια τα οποία βρίσκονται ανάμεσα στα σύμβολα #.

Β. Μορφή προγράμματος

Κάθε πρόγραμμα ξεκινάει με τη λέξη κλειδί `program`. Στη συνέχεια ακολουθεί ένα αναγνωριστικό (όνομα) για το πρόγραμμα αυτό και τα **τρία βασικά block** του προγράμματος:

- **declarations**: οι δηλώσεις μεταβλητών.
- **subprograms**: οι συναρτήσεις και διαδικασίες οι οποίες μπορούν να είναι και φωλιασμένες μεταξύ τους.
- **statements**: και οι εντολές του κυρίως προγράμματος.

Γ. Δηλώσεις μεταβλητών

Η δήλωση μεταβλητών γίνεται με την εντολή `declare`. Ακολουθούν τα ονόματα των αναγνωριστικών χωρίς καμία άλλη δήλωση, αφού γνωρίζουμε ότι πρόκειται για ακέραιες μεταβλητές και χωρίς να είναι αναγκαίο να βρίσκονται στην ίδια γραμμή. Οι μεταβλητές χωρίζονται μεταξύ τους με κόμματα. Το τέλος της δήλωσης αναγνωρίζεται με το ελληνικό ερωτηματικό. Επιτρέπεται να έχουμε πολλαπλές συνεχόμενες χρήσεις της `declare`.

Δ. Τελεστές και εκφράσεις

Η **προτεραιότητα των τελεστών** από τη μεγαλύτερη στη μικρότερη είναι:

- Πολλαπλασιαστικοί: `*, /`
- Προσθετικοί: `+, -`
- Σχεσιακοί: `=, , <>, <=, >=`
- Λογικοί: `not`
- Λογική σύζευξη: `and`
- Λογική διάζευξη: `or`

Ε. Δομές της γλώσσας

Η Cimple υποστηρίζει τις ακόλουθες δομές:

- `ID := expression`
  Εκχώρηση: Χρησιμοποιείται για την ανάθεση της τιμής μιας μεταβλητής ή μιας σταθεράς, ή μιας έκφρασης σε μία μεταβλητή.

- `if (condition)`
  `statements`$_1$
  `[ else`
  `statements`$_2$ `]`
  Απόφαση if: Η εντολή απόφασης `if` εκτιμά εάν ισχύει η συνθήκη *condition* και εάν πράγματι ισχύει, τότε εκτελούνται οι εντολές *statements*$_1$ που το ακολουθούν. Το `else` δεν αποτελεί υποχρεωτικό τμήμα της εντολής και γι' αυτό βρίσκεται σε αγκύλη. Οι εντολές *statements*$_2$ που ακολουθούν το `else` εκτελούνται εάν η συνθήκη *condition* δεν ισχύει.

- `while (condition)`
  `statements`
  Επανάληψη while: Η εντολή επανάληψης `while` επαναλαμβάνει τις εντολές *statements*, όσο η συνθήκη *condition* ισχύει. Αν την πρώτη φορά που θα αποτιμηθεί η *condition* το αποτέλεσμα της αποτίμησης είναι ψευδές ,τότε οι *statements* δεν εκτελούνται ποτέ.

- `switchcase`
  `(case (condition) statements`$_1$`)*`
  `default statements`$_2$
  Επιλογή switchcase: Η δομή `switchcase` ελέγχει τις *condition* που βρίσκονται μετά τα `case`. Μόλις μία από αυτές βρεθεί αληθής , τότε εκτελούνται οι αντίστοιχες *statements*$_1$(που ακολουθούν το *condition*). Μετά ο έλεγχος μεταβαίνει έξω από την `switchcase`. Αν, κατά το πέρασμα καμία από τις case δεν ισχύσει, τότε ο έλεγχος μεταβαίνει στην `default` και εκτελούνται οι *statements*$_2$. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την `switchcase`.

- `forcase`
  `(case (condition) statements`$_1$`)*`
  `default statements`$_2$
  Επανάληψη forcase: Η δομή `forcase` ελέγχει τις *condition* που βρίσκονται μετά τα `case`. Μόλις μία από αυτές βρεθεί αληθής , τότε εκτελούνται οι αντίστοιχες *statements*$_1$(που ακολουθούν το *condition*). Μετά ο έλεγχος μεταβαίνει στην αρχή της `forcase`. Αν καμία από τις case δεν ισχύσει, τότε ο έλεγχος μεταβαίνει στην `default` και εκτελούνται οι *statements*$_2$. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την `forcase`.

- `incase`
  `(case (condition) statements`$_1$`)*`
  Επανάληψη incase: Η δομή `incase` ελέγχει τις *condition* που βρίσκονται μετά τα `case`, εξετάζοντας τες κατά σειρά. Για κάθε μία για τις οποίες η αντιστοίχιση *condition* ισχύει εκτελούνται οι *statements* που ακολουθούν το *condition*. Θα εξεταστούν με τη σειρά όλες οι *condition* και θα εκτελεστούν όλες οι *statements* των οποίων οι *condition* ισχύουν. Αφότου εξεταστούν όλες οι `case` , ο έλεγχος μεταβαίνει έξω από τη δομή `incase`, εάν καμία από τις *statements* δεν έχει εκτελεστεί, ή μεταβαίνει στην αρχή της `incase`, εάν έστω και μία από τις *statements* έχει εκτελεστεί.

- `return (expression)`
  Επιστροφή τιμής συνάρτησης: Χρησιμοποιείται μέσα σε συναρτήσεις για να επιστραφεί το αποτέλεσμα της συνάρτησης, το οποίο είναι το αποτέλεσμα της αποτίμησης του *expression*.

- `print (expression)`
  Έξοδος δεδομένων: Εμφανίζει στην οθόνη το αποτέλεσμα της αποτίμησης του *expression*.

- `input (ID)`
  Είσοδος δεδομένων: Ζητάει από τον χρήστη να δώσει μία τιμή μέσα από το πληκτρολόγιο. Η τιμή που θα δώσει θα μεταφερθεί στην μεταβλητή *ID*.

- `call functionName (actualParameters)`
  Κλήση διαδικασίας: Καλεί μία διαδικασία.

- `function ID (formalPars)`
  ```
  {
      declarations
      subprograms
      statements
  }
  ```
  Συναρτήσεις: κλήση γίνεται μέσα από τις αριθμητικές παραστάσεις σαν τελούμενο.

- `procedure ID (formalPars)`
  ```
  {
      declarations
      subprograms
      statements
  }
  ```
  Διαδικασίες: κλήση γίνεται με την `call`.

Η formalPars είναι η λίστα των τυπικών παραμέτρων. Οι **συναρτήσεις** και οι **διαδικασίες** μπορούν να φωλιάσουν η μία μέσα στην άλλη. Οι κανόνες εμβέλειας ακολουθούν τους κανόνες της PASCAL. Η επιστροφή της τιμής μιας συνάρτησης γίνεται με την `return`.

ΣΤ. Μετάδοση παραμέτρων

Η Cimple υποστηρίζει δύο τρόπους μετάδοσης παραμέτρων:

- ***με τιμή***: Δηλώνεται με τη λεκτική μονάδα `in`. Αλλαγές στην τιμή της δεν επιστρέφονται στο πρόγραμμα που κάλεσε τη συνάρτηση.
- ***με αναφορά***: Δηλώνεται με τη λεκτική μονάδα `inout`. Κάθε αλλαγή στη τιμή της μεταφέρεται αμέσως στο πρόγραμμα που κάλεσε τη συνάρτηση.

Στην κλήση μίας συνάρτησης οι πραγματικοί παράμετροι συντάσσονται μετά από τις λέξεις κλειδιά `in` και `inout`, ανάλογα με το αν περνούν με τιμή ή αναφορά.

Ζ. Κανόνες εμβέλειας

Καθολικές ονομάζονται οι μεταβλητές που δηλώνονται στο κυρίως πρόγραμμα και είναι προσβάσιμες σε όλους. ***Τοπικές*** είναι οι μεταβλητές που δηλώνονται σε μία συνάρτηση ή διαδικασία και είναι προσβάσιμες μόνο μέσα από τη συγκεκριμένη συνάρτηση ή διαδικασία. Κάθε συνάρτηση ή διαδικασία, εκτός των τοπικών μεταβλητών, των παραμέτρων της και των καθολικών μεταβλητών, έχει επίσης πρόσβαση και στις μεταβλητές που έχουν δηλωθεί σε συναρτήσεις ή διαδικασίες προγόνους ή και σαν παραμέτρους αυτών.

Ισχύει ο δημοφιλής κανόνας ότι, αν δύο (ή περισσότερες) μεταβλητές ή παράμετροι έχουν το ίδιο όνομα και έχουν δηλωθεί σε διαφορετικό επίπεδο φωλιάσματος, τότε οι τοπικές μεταβλητές και παράμετροι υπερκαλύπτουν τις μεταβλητές και παραμέτρους των προγόνων, οι οποίες με τη σειρά τους υπερκαλύπτουν τις καθολικές μεταβλητές.

Μία συνάρτηση ή διαδικασία έχει δικαίωμα να καλέσει τον εαυτό της και όποια συνάρτηση βρίσκεται στο ίδιο επίπεδο φωλιάσματος με αυτήν, της οποίας η δήλωση προηγείται στον κώδικα.

Η. Η γραμματική της Cimple

Προς αποφυγή επαναλήψεων στο σημείο αυτό παρατίθενται ονομαστικά οι κανόνες γραμματικής της Cimple οι οποίοι δίνουν και την ακριβή περιγραφή της γλώσσας. Η αναλυτική επεξήγηση των κανόνων βρίσκεται στο μέρος της συντακτικής ανάλυσης παρακάτω.

```
program: program ID block.
block: declarations subprograms statements
declarations : ( declare varlist ; )*
varlist : ID ( , ID )*
        |ε
subprograms : ( subprogram )*
subprogram  : function ID ( formalparlist ) block
            | procedure ID ( formalparlist ) block

formalparlist : formalparitem ( , formalparitem )*
              |ε
formalparitem : in ID
              | inout ID
statements : statement ;
           | { statement ( ; statement )* }
```

```
Statement : assignStat
    | ifStat
    | whileStat
    | switchcaseStat
    | forcaseStat
    | incaseStat
    | callStat
    | returnStat
    | inputStat
    | printStat
    |ε
```

```
assignStat : ID := expression
```

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

```
elsepart : else statements
         |ε
```

```
whileStat : while (condition)statements
```

```
switchcaseStat : switchcase
   ( case ( condition ) statements )*
   default statements
```

```
forcaseStat : forcase
   ( case ( condition ) statements )*
   default statements
```

```
incaseStat : incase
   ( case ( condition ) statements )*
```

```
returnStat : return ( expression )
```

```
callStat : call ID ( actualparlist )
```

```
printStat : print ( expression )
```

```
inputStat : input ( ID )
```

```
actualparlist : actualparitem ( , actualparitem )*
              |ε
```

```
actualparitem : in expression
              | inout ID
```

```
condition : boolterm ( or boolterm )*
```

```
boolterm : boolfactor ( and boolfactor )*
```

```
boolfactor : not [ condition ]
           | [ condition ]
           | expression REL_OP expression
```

```
expression : optionalSign term ( ADD_OP term )*
```

```
term: factor ( MUL_OP factor )*
```

```
factor : INTEGER
       | ( expression )
       | ID idtail
```

```
idtail : ( actualparlist )
       | ε
```

```
optionalSign : ADD_OP
             | ε
```

```
REL_OP : =|<=|>=|>|<|<>;
```

ADD_OP : +|-
MUL_OP : *|/
INTEGER : [0-9]+
ID : [a-zA-Z][a-zA-Z0-9]*

Θ. Παραδείγματα

Ακολουθούν δύο μικρά παραδείγματα προγραμμάτων της γλώσσας Cimple.

Υπολογισμός παραγοντικού:

```
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);
    }.
```

Η ακολουθία Fibonacci:

```
program fibonacci

    declare x;

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

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

ΜΕΡΟΣ 2ο: Λεκτική ανάλυση

Έχοντας προσδιορίσει και κατανοήσει την γλώσσα προς μεταγλώττιση, μπορούμε να προχωρήσουμε βήμα-βήμα στην υλοποίηση του μεταγλωττιστή. Η υλοποίηση έχει γίνει μόνο με τη γλώσσα προγραμματισμού **python 3**, χωρίς τη χρήση εξωτερικών εργαλείων. Το πρώτο βήμα είναι η λεκτική ανάλυση, δηλαδή η εξαγωγή **λεκτικών μονάδων** από το αρχικό πρόγραμμα Cimple. Για λόγους συνέπειας με τον κώδικα του μεταγλωττιστή που έχουμε υλοποιήσει θα αναφερόμαστε στην έννοια της λεκτικής μονάδας με τον αγγλικό όρο, **token**. Ακολουθεί η αναλυτική περιγραφή του λεκτικού αναλυτική, αρχικά σε επίπεδο **διαπροσωπείας** και έπειτα σε επίπεδο **εσωτερικής λειτουργίας**.

Α. Διαπροσωπεία λεκτικού αναλυτή

Η λεκτική ανάλυση υλοποιείται από την συνάρτηση **lex()**. Διαβάζει γράμμα-γράμμα το αρχικό πρόγραμμα μέχρι να βρει το επόμενο token και το επιστρέφει. Ένα token περιγράφεται από την αντίστοιχη **κλάση Token** η οποία έχει τα εξής πεδία:

- ***tokenType***: Ακέραια σταθερά που συμβολίζει τον τύπο του token. (Κάθε δεσμευμένη λέξη και κάθε σύμβολο του αλφαβήτου της Cimple έχει τον δικό του κωδικό.)
- ***tokenString***: Το token σε μορφή string.
- ***lineNo***: Ο αριθμός της γραμμής στο αρχικό πρόγραμμα όπου βρέθηκε το token.

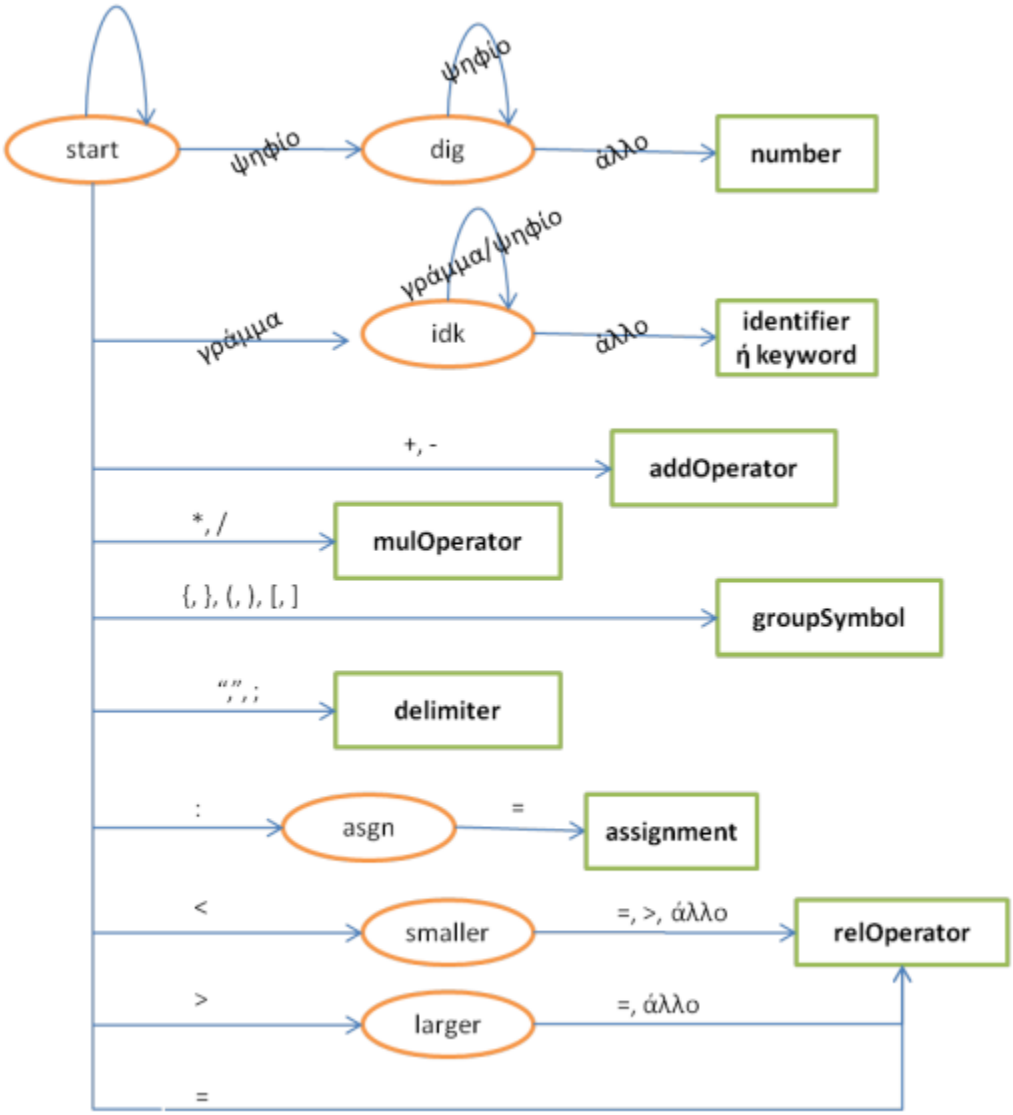
Η lex() καλείται πολλαπλές φορές από τον συντακτικό αναλυτή ο οποίος επεξηγείται παρακάτω και κάθε φορά συνεχίζει να διαβάζει από το σημείο που σταμάτησε στην προηγούμενη κλήση. Φυσικά, η πρώτη κλήση ξεκινά από την αρχή του προγράμματος. Σε περίπτωση σφάλματος τυπώνεται το αντίστοιχο μήνυμα και η μεταγλώττιση σταματά.

Β. Εσωτερική λειτουργία λεκτικού αναλυτή

Ο λεκτικός αναλυτής ουσιαστικά υλοποιεί ένα **πεπερασμένο αυτόματο** καταστάσεων. Το αυτόματο ξεκινά από μία αρχική κατάσταση και με την είσοδο κάθε χαρακτήρα αλλάζει κατάσταση μέχρι να φτάσει σε κάποια τελική κατάσταση όπου και επιστρέφεται είτε το token που βρέθηκε είτε μήνυμα σφάλματος. Τα tokens που αναγνωρίζει μπορεί να είναι:

- ***δεσμευμένες λέξεις:*** π.χ. program, print, return
- ***σύμβολα της γλώσσας:*** π.χ. +, :=, >=
- ***ακέραιοι αριθμοί:*** π.χ. 5, 512
- ***αναγνωριστικά:*** π.χ. var, x, lamda12

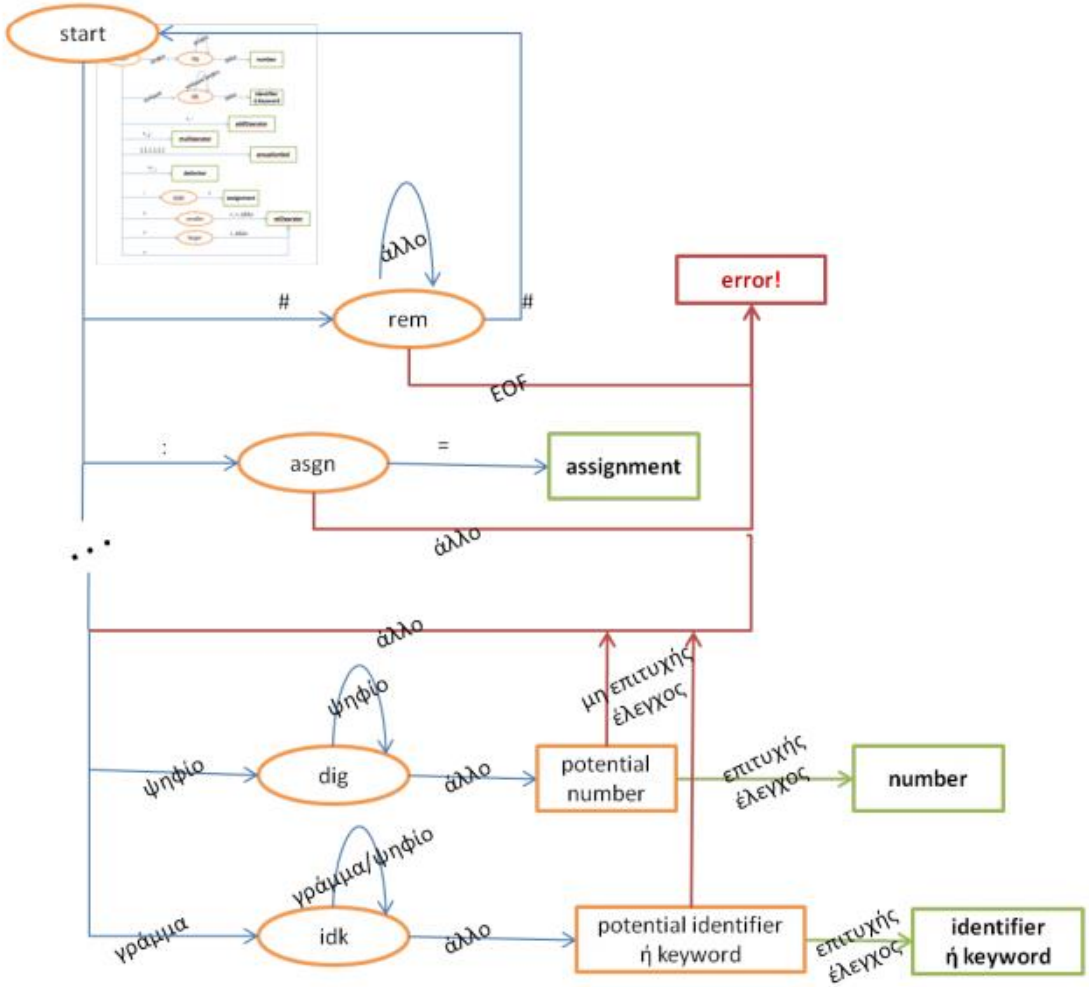
Ακολουθεί το αυτόματο λειτουργίας του λεκτικού αναλυτή, όπου η αρχική κατάσταση είναι η *start,* οι ενδιάμεσες καταστάσεις συμβολίζονται με έλλειψη, ενώ οι τελικές με παραλληλόγραμμο.



Εκτός από τα tokens που βλέπουμε πως αναγνωρίζονται στο παραπάνω σχήμα, ο λεκτικός αναλυτής είναι υπεύθυνος και για τα σχόλια του προγράμματος καθώς και για της εύρεση κάποιων σφαλμάτων. Τα σχόλια είναι αναπόσπαστο κομμάτι ενός προγράμματος και ο λεκτικός αναλυτής τα διαχειρίζεται αποκλειστικά. Τα αναγνωρίζει και τα αφαιρεί από το πρόγραμμα έτσι ώστε ο συντακτικός αναλυτής να μην χρειάζεται να πληροφορηθεί για την ύπαρξη τους. Όσον αφορά τα σφάλματα, ο λεκτικός αναλυτής μπορεί να αναγνωρίσει τα ακόλουθα:

- ***Μη έγκυρος χαρακτήρας:*** Ο χαρακτήρας που διαβάστηκε δεν ανήκει στο αλφάβητο της Cimple.
- ***Σφάλμα σε ακέραιο αριθμό:*** Το σφάλμα αυτό προκύπτει όταν διαβαστεί χαρακτήρας που δεν είναι ψηφίο σε έναν ενδεχόμενο ακέραιο αριθμό.
- ***Σφάλμα ανάθεσης:*** Όταν το σύμβολο ":" δεν ακολουθείται από το σύμβολο "=" και άρα δεν προκύπτει το σύμβολο ανάθεσης.
- ***Τέλος αρχείου σε σχόλιο:*** Προκύπτει όταν συναντήσουμε το τέλος του αρχείου μέσα σε σχόλιο. Τα σχόλια πάντα πρέπει να αρχίζουν και να τελειώνουν με το σύμβολο "#".
- ***Αναγνωριστικό εκτός ορίων:*** Όταν ο λεκτικός αναλυτής βρει αναγνωριστικό το οποίο αποτελείται από περισσότερους από 30 χαρακτήρες.
- ***Ακέραιος εκτός ορίων:*** Όταν ο λεκτικός αναλυτής βρει ακέραιο αριθμό εκτός των ορίων της γλώσσας.

Προσθέτοντας τα παραπάνω στο αρχικό σχήμα έχουμε το εξής αυτόματο:



Γ. Ο κώδικας του λεκτικού αναλυτή

Σε επίπεδο κώδικα, το παραπάνω αυτόματο έχει υλοποιηθεί με ένα **διδυάστατο πίνακα**. Πιο συγκεκριμένα, έχουμε ορίσει **σταθερές** για να περιγράψουμε τις καταστάσεις και τις μεταβάσεις του αυτομάτου. Κάθε κατάσταση που συμβολίζεται με έλλειψη στο αυτόματο (δηλ. η αρχική και οι ενδιάμεσες) έχει την δική της σταθερά και αντιστοιχεί σε μία γραμμή του πίνακα. Με τον ίδιο τρόπο κάθε χαρακτήρας που ενδέχεται να διαβάσει ο λεκτικός αναλυτής έχει δική του σταθερά και αντιστοιχεί σε μία στήλη του πίνακα. Επίσης, σταθερές έχουν ανατεθεί και για τις τελικές καταστάσεις, όπου κάθε τελική αντιστοιχεί σε ένα token. Φυσικά, οι τιμές τους πρέπει να είναι διαφορετικές από αυτές των μη τελικών καταστάσεων για να μπορούμε να τις διακρίνουμε. Σημειώνεται πως υπάρχει μία καθολική σταθερά για τα αναγνωριστικά tokens ενώ για κάθε δεσμευμένη λέξη έχουμε ξεχωριστή σταθερά. Εάν το αναγνωριστικό token είναι κάποια δεσμευμένη λέξη έχουμε την ανάλογη αλλαγή σταθεράς. Σταθερές έχουν ανατεθεί και σε κάποια από τα σφάλματα που βρίσκει ο συντακτικός αναλυτής

Στο σημείο αυτό, δεν μένει παρά να δημιουργήσουμε και τον ίδιο τον πίνακα. Ακολουθώντας το αυτόματο, ξεκινάμε από την αρχική κατάσταση και εξετάζουμε τις γραμμές (μη τελικές καταστάσεις) μία-μία. Συμπεριλαμβάνουμε όλα τα ενδεχόμενα, συμπληρώνοντας την αντίστοιχη κατάσταση (σταθερά token, ενδιάμεσης κατάστασης ή σφάλματος) σε κάθε στήλη του πίνακα. Επιπλέον πίνακες που έχουν δημιουργηθεί αφορούν την ομαδοποίηση των ψηφίων των γραμμάτων και των δεσμευμένων λέξεων με σκοπό την διευκόλυνση του κώδικα.

Εσωτερικά, η συνάρτηση lex() χρησιμοποιεί την global μεταβλητή line, η οποία αρχικοποιείται στο 1 στην αρχή της μεταγλώττισης, για να γνωρίζει σε ποια γραμμή του αρχικού προγράμματος βρίσκεται. Όπως είναι αναμενόμενο, κάθε φορά ξεκινάει από την σταθερά που αντιστοιχεί στην αρχική κατάσταση και μπαίνει σε loop για να «ακολουθήσει» το αυτόματο. Σε κάθε επανάληψη κάνει τα παρακάτω:

- Διαβάζει τον επόμενο χαρακτήρα από το αρχικό πρόγραμμα Cimple.
- Καλεί την φωλιασμένη συνάρτηση **findCharacterToken()** η οποία αντιστοιχίζει τον χαρακτήρα που διαβάστηκε με την σταθερά του.
- Μεταβαίνει στην επόμενη κατάσταση χρησιμοποιώντας τον πίνακα με γραμμή την σταθερά της τωρινής κατάστασης και στήλη την σταθερά του χαρακτήρα που διαβάστηκε.
- Εάν δεν βρισκόμαστε στην αρχική κατάσταση και δεν πρόκειται για σχόλιο προσθέτει τον χαρακτήρα που διαβάστηκε σε ένα buffer με σκοπό να σχηματιστεί το string του token.

Η επανάληψη τελειώνει όταν φτάσουμε σε σταθερά που αντιστοιχεί σε τελική κατάσταση. Πράγμα το οποίο σημαίνει πως ο λεκτικός αναλυτής έχει σίγουρα βρει token ή έχει εντοπίσει κάποιο σφάλμα. Στο σημείο αυτό ελέγχει αν κατά την αναγνώριση του token έχει διαβάσει επιπλέον χαρακτήρα ο οποίος ανήκει στο επόμενο token. Αυτό γίνεται σε token που αφορούν

αναγνωριστικά, αριθμούς και τα δεσμευμένα σύμβολα "<" και ">". Στην περίπτωση αυτή αφαιρεί τον χαρακτήρα από το string του token και πηγαίνει μία θέση πίσω στο αρχείο του προγράμματος έτσι ώστε ο χαρακτήρας αυτός να διαβαστεί από το επόμενο token. Εάν ο επιπλέον χαρακτήρας είναι ο χαρακτήρας της νέας γραμμής "\n" υπάρχει και η αντίστοιχη ενημέρωση της μεταβλητής line.

Στη συνέχεια, σε περίπτωση που το token είναι κάποιο αναγνωριστικό, καλεί την φωλιασμένη συνάρτηση **identifyToken()** η οποία ελέγχει εάν το αναγνωριστικό αντιστοιχεί σε κάποια δεσμευμένη λέξη. Έπειτα καλείται η φωλιασμένη συνάρτηση **errorCheck()** η οποία ελέγχει εάν βρέθηκε σφάλμα και στην περίπτωση αυτή τυπώνει ανάλογο μήνυμα και τερματίζει την μεταγλώττιση καλώντας την **handleError()**. Τέλος, δημιουργεί το αντικείμενο του token που βρέθηκε καλώντας τον constructor της κλάσης Token και το επιστρέφει. Τα ορίσματα του constructor είναι η σταθερά που αντιστοιχεί στο token, το string του token και ο αριθμός γραμμής που βρέθηκε. Εάν έχει δοθεί το προαιρετικό όρισμα **-lex** από την γραμμή εντολών, η πληροφορία του token τυπώνεται στην οθόνη πριν την επιστροφή της συνάρτησης.

ΜΕΡΟΣ 3ο: Συντακτική ανάλυση

Έχοντας φτιάξει και ελέγξει την ορθή λειτουργία του λεκτικού αναλυτική, το επόμενο βήμα στην υλοποίηση του μεταγλωττιστή είναι η συντακτική ανάλυση. Δηλαδή, ο έλεγχος για το εάν το αρχικό πρόγραμμα ανήκει στην γλώσσα Cimple ή όχι. Αυτό γίνεται υλοποιώντας τους κανόνες της γραμματικής της Cimple που έχουν επεξηγηθεί παραπάνω στον κώδικα του μεταγλωττιστή. Ακολουθεί η μία γενικότερη περιγραφή της λειτουργίας του λεκτικού αναλυτή και στη συνέχεια η αναλυτική επεξήγηση του κάθε κανόνα γραμματικής.

Α. Λειτουργία συντακτικού αναλυτή

Η συντακτική ανάλυση υλοποιείται από την συνάρτηση **syn()** η οποία περιέχει μία φωλιασμένη συνάρτηση για κάθε κανόνα γραμματικής της Cimple. Τέλος καλεί την συνάρτηση **program()** που είναι και ο πρώτος κανόνας γραμματικής που θέλουμε να ελέγξουμε. Οι υπόλοιπες κλήσεις συναρτήσεων είναι εσωτερικά των φωλιασμένων συναρτήσεων της γραμματικής και εξαρτώνται κάθε φορά από το token που επιστρέφει ο λεκτικός αναλυτής. Η Cimple υποστηρίζει και αυτή φωλιασμένη δήλωση υποπρογραμμάτων, άρα μπορεί να υπάρξει μία έμεση αναδρομή στις κλήσεις συναρτήσεων της γραμματικής.

Αξίζει να σημειωθεί πως η συνάρτηση του συντακτικού αναλυτή δημιουργεί το περιβάλλον στο οποίο μετέπειτα θα ολοκληρωθεί και η μεταγλώττιση του κώδικα με την προσθήκη περαιτέρω ρουτίνων. Παρόμοια με τον λεκτικό αναλυτή, οι ρουτίνες των επόμενων φάσεων μεταγλώττισης καλούνται και αυτές εσωτερικά της συνάρτησης syn(). Ουσιαστικά, πρόκειται για την κύρια συνάρτηση του μεταγλωττιστή η οποία καλείται μία φορά στην αρχή του προγράμματος, ελέγχει τους κανόνες γραμματικής ενώ παράλληλα εκτελεί και τις υπόλοιπες ρουτίνες. Με την ολοκλήρωση της συνάρτησης του συντακτικού αναλυτή, ολοκληρώνεται και η μεταγλώττιση του προγράμματος Cimple.

Β. Οι κανόνες γραμματικής στον κώδικα

Ακολουθεί η αναλυτική περιγραφή των συναρτήσεων που υλοποιούν τους κανόνες γραμματικής σε επίπεδο κώδικα.

- `program : program ID block .`
  Η δεσμευμένη λέξη `program` είναι πάντα το πρώτο token που περιμένουμε να διαβαστεί σε ένα πρόγραμμα Cimple. Καλούμε τον λεκτικό αναλυτή και ελέγχουμε αν έχει επιστρέψει το token αυτό. Στη συνέχεια καλούμε πάλι τον λεκτικό αναλυτή και ελέγχουμε εάν το επόμενο token είναι αναγνωριστικό. Εάν είναι, καλούμε τον λεκτικό αναλυτή ακόμη μία φορά και έπειτα καλούμε την συνάρτηση του κανόνα `block`. Τέλος, ελέγχουμε εάν το τελευταίο token είναι το σύμβολο τερματισμού και εάν όλα πάνε καλά

στο σημείο αυτό ολοκληρώνεται η μεταγλώττιση. Σε κάθε έλεγχο υπάρχει και η αντίστοιχη εναλλακτική περίπτωση όπου δεν έχει βρεθεί το αναμενόμενο token. Στις περιπτώσεις αυτές τυπώνεται μήνυμα σφάλματος και η μεταγλώττιση σταματάει. Το ίδιο γίνεται και στους υπόλοιπους κανόνες γραμματικής και για αυτό θα ήταν περιττό να το εξηγούσαμε σε κάθε κανόνα ξεχωριστά. Αντίθετα, εξηγούμε μόνο τα σημεία που δεν υπάρχει τέτοια εναλλακτική.

- `block : declarations subprograms statements`
  Ένα `block` μπορεί να αναφέρεται στο `block` του κυρίως προγράμματος ή στο `block` ενός υποπρογράμματος. Καλεί τις συναρτήσεις των κανόνων `declarations` `subprograms` και `statements` με αυτή τη σειρά.

- `declarations : ( declare varlist ; )*`
  Ο κανόνας `declarations` διαχειρίζεται τις δηλώσεις μεταβλητών στο κυρίως πρόγραμμα ή σε κάποιο υποπρόγραμμα. Επιτρέπονται πολλαπλές δηλώσεις ή καμία δήλωση. Για το λόγο αυτό, ελέγχουμε επαναληπτικά σε μία while εάν έχουμε βρει token με την δεσμευμένη λέξη `declare`. Αν δεν βρεθεί το token αυτό δεν έχουμε περίπτωση σφάλματος αφού η δήλωση μεταβλητών είναι προεραιτική. Εσωτερικά της while, αν βρεθεί το token, καλούμε τον λεκτικό αναλυτή και έπειτα τον κανόνα `varlist.` Στη συνέχεια ελέγχουμε για το σύμβολο `;` και καλούμε τον λεκτικό αναλυτή μία ακόμη φορά.

- ```
  varlist : ID ( , ID )*
          | ε
  ```
  Ο κανόνας `varlist` είναι υπεύθυνος για την λίστα από αναγνωριστικά μεταβλητών έπειτα από την δεσμευμένη λέξη `declare`. Αρχικά ελέγχει αν το επόμενο token είναι αναγνωριστικό. Δεν έχουμε σενάριο σφάλματος αφού μπορούμε να έχουμε και κενή λίστα μεταβλητών. Εάν βρεθεί αναγνωριστικό token καλείται ο λεκτικός αναλυτής. Στη συνέχεια ελέγχει επαναληπτικά αν υπάρχει το token `,` ακολουθόμενο από επιπλέον αναγνωριστικό.

- `subprograms : ( subprogram )*`
  Ο κανόνας `subprograms` ελέγχει για υποπρογράμματα τα οποία μπορεί να ξεκινάνε με μία από τις δεσμευμένες λέξεις `function` ή `procedure`, ανάλογα με το είδος του υποπρογράμματος. Καλέι τον κανόνα `subprogram` για κάθε υποπρόγραμμα.
  ```

- ```
  subprogram : function ID ( formalparlist ) block
             | procedure ID ( formalparlist ) block
  ```
  Ο κανόνας `subprogram` ελέγχει ξεχωριστά για τις δεσμευμένες λέξεις `function` ή `procedure`. Καλεί τον λεκτικό αναλυτή και ελέγχει για το αναγνωριστικό του υποπρογράμματος. Ελέγχει για το σύμβολο της αριστερής παρένθεσης, καλεί τον κανόνα `formalparlist` που διαχειρίζεται τις παραμέτρους και έπειτα ελέγχει για το σύμβολο της δεξιάς παρένθεσης. Τέλος, καλεί τον κανόνα `block`.

- ```
  formalparlist : formalparitem ( , formalparitem )*
                | ε
  ```
  Ο κανόνας `formalparlist` διαχειρίζεται την λίστα από παραμέτρους ενός υποπρογράμματος κατά τη δήλωση του. Η λίστα παραμέτρων μπορεί να είναι και κενή και άρα δεν έχουμε περίπτωση σφάλματος. Αρχικά καλεί την `formalparitem` μία φορά και στη συνέχεια επαναληπτικά ελέγχει για το σύμβολο `,` καλεί τον λεκτικό αναλυτή και πάλι την `formalparitem`.

- ```
  formalparitem : in ID
                | inout ID
  ```
  Ο κανόνας `formalparitem` είναι υπεύθυνος για την διαχείριση μίας παραμέτρου υποπρογράμματος κατά τη δήλωση του. Ελέγχει ξεχωριστά εάν υπάρχει token με την δεσμευμένη λέξη `in` ή `inout`. Για τον έλεγχο αυτό δεν έχουμε μήνυμα λάθους αφού η λίστα παραμέτρων μπορεί να είναι και κενή. Στη συνέχεια καλεί τον λεκτικό αναλυτή, ελέγχει εάν υπάρχει token αναγνωριστικού και εάν υπάρχει καλεί πάλι τον λεκτικό αναλυτή.

- ```
  statements : statement ;
             | { statement ( ; statement )* }
  ```
  Ο κανόνας `statements` είναι υπεύθυνος για τις εκφράσεις που περιέχει είτε το κυρίως πρόγραμμα είτε ένα υποπρόγραμμα. Η Cimple μας δίνει την δυνατότητα να παραλείψουμε τα curly brackets της ομαδοποίησης εάν υπάρχει μόνο μία έκφραση αρκεί να ακολουθείται από το σύμβολο `;`. Για την περίπτωση αυτή ο κανόνας απλά καλεί την `statement` και έπειτα ελέγχει για token που αντιστοιχεί στο σύμβολο `;` και καλεί τον λεκτικό αναλυτή. Στην δεύτερη περίπτωση ο κανόνας ελέγχει για token που αντιστοιχεί στο σύμβολο `{` και αν το βρει καλεί τον λεκτικό αναλυτή και έπειτα την `statement`. Έπειτα σε μία while ελέγχει για το σύμβολο `;` και καλεί τον λεκτικό αναλυτή και πάλι την `statement`. Στο τέλος αφού τελειώσει η επανάληψη ελέγχει και για το σύμβολο `}` που σηματοδοτεί το τέλος των εκφράσεων και καλεί τον λεκτικό αναλυτή.

- ```
  statement : assignStat
            | ifStat
            | whileStat
            | switchcaseStat
            | forcaseStat
            | incaseStat
            | callStat
            | returnStat
            | inputStat
            | printStat
            | ε
  ```
  Ο κανόνας `statement` είναι υπεύθυνος για μία έκφραση. Ελέγχει για δεσμευμένη λέξη που αντιστοιχεί σε έκφραση μίας δομής και καλεί τον αντίστοιχο κανόνα. Η Cimple υποστηρίζει και την κενή έκφραση επομένως δεν έχουμε ενδεχόμενο σφάλματος εάν δεν βρεθεί κάποιο από τα αναμενόμενα tokens.

- `assignStat : ID := expression`
  Ο κανόνας `assignStat` διαχειρίζεται την δομή της εκχώρησης. Αρχικά, ελέγχει για token αναγνωριστικού, καλεί τον λεκτικό αναλυτή, ελέγχει για token συμβόλου εκχώρησης και καλεί τον κανόνα `expression`.

- `ifStat : if ( condition ) statements elsepart`
  Ο κανόνας `ifStat` διαχειρίζεται την δομή απόφασης if. Ο έλεγχος για το token `if` έχει προηγηθεί στον κανόνα `statement` επομένως καλεί τον λεκτικό αναλυτή και ελέγχει για το σύμβολο `(`. Εάν το σύμβολο βρεθεί καλεί ξανά τον λεκτικό αναλυτή και έπειτα τον κανόνα `condition.` Στη συνέχεια ελέγχει για το σύμβολο `)`, καλεί τον λεκτικό αναλυτή και έπειτα τους κανόνες `statements` και `elsepart`.

- ```
  elsepart : else statements
           | ε
  ```
  Ο κανόνας `elsepart` διαχειρίζεται το κομμάτι του `else` από την δομή απόφασης if. Το κομμάτι αυτό δεν είναι υποχρεωτικό άρα δεν υπάρχει περίπτωση σφάλματος. Ο κανόνας ελέγχει για την δεσμευμένη λέξη `else` καλεί τον λεκτικό αναλυτή και τον κανόνα `statements.`

- `whileStat : while ( condition ) statements`
  Ο κανόνας `whileStat` είναι υπεύθυνος για την δομή επανάληψης while. Ο έλεγχος για το token `while` έχει προηγηθεί στον κανόνα `statement` επομένως καλεί τον λεκτικό αναλυτή και ελέγχει για το σύμβολο `(`. Εάν το σύμβολο βρεθεί καλεί ξανά τον λεκτικό

αναλυτή και έπειτα τον κανόνα condition. Στη συνέχεια ελέγχει για το σύμβολο) και εάν βρεθεί και αυτό καλεί τον κανόνα statements.

- ```
  switchcaseStat : switchcase
     ( case ( condition ) statements )*
     default statements
  ```
  Ο κανόνας switchcaseStat είναι υπεύθυνος για την δομή επιλογής switchcase. Ο έλεγχος για το token switchcase έχει προηγηθεί στον κανόνα statement επομένως καλεί τον λεκτικό αναλυτή και ελέγχει επαναληπτικά για το token case. Στη συνέχεια καλεί ξανά τον λεκτικό αναλυτή και ελέγχει για το σύμβολο (. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και τον κανόνα condition και έπειτα ελέγχει για το σύμβολο). Εάν βρεθεί καλεί πάλι τον λεκτικό αναλυτή και έπειτα τον κανόνα statements. Η επανάληψη τελειώνει όταν δεν βρέθει άλλο token case και τοτε γίνεται έλεγχος για την δεσμευμένη λέξη default. Εάν η λέξη βρεθεί καλείται ο λεκτικός αναλυτής και ο κανόνας statements.

- ```
  forcaseStat : forcase
     ( case ( condition ) statements )*
     default statements
  ```
  Ο κανόνας forcaseStat είναι υπεύθυνος για την δομή επανάληψης forcase. Ο έλεγχος για το token forcase έχει προηγηθεί στον κανόνα statement επομένως καλεί τον λεκτικό αναλυτή και ελέγχει επαναληπτικά για το token case. Στη συνέχεια καλεί ξανά τον λεκτικό αναλυτή και ελέγχει για το σύμβολο (. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και τον κανόνα condition και έπειτα ελέγχει για το σύμβολο). Εάν βρεθεί καλεί πάλι τον λεκτικό αναλυτή και έπειτα τον κανόνα statements. Η επανάληψη τελειώνει όταν δεν βρέθει άλλο token case και τοτε γίνεται έλεγχος για την δεσμευμένη λέξη default. Εάν η λέξη βρεθεί καλείται ο λεκτικός αναλυτής και ο κανόνας statements.

- ```
  incaseStat : incase
     ( case ( condition ) statements )*
  ```
  Ο κανόνας incaseStat είναι υπεύθυνος για την δομή επανάληψης incase. Ο έλεγχος για το token incase έχει προηγηθεί στον κανόνα statement επομένως καλεί τον λεκτικό αναλυτή και ελέγχει επαναληπτικά για το token case. Στη συνέχεια καλεί ξανά τον λεκτικό αναλυτή και ελέγχει για το σύμβολο (. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και τον κανόνα condition και έπειτα ελέγχει για το σύμβολο). Εάν βρεθεί καλεί πάλι τον λεκτικό αναλυτή και έπειτα τον κανόνα statements. Η επανάληψη τελειώνει όταν δεν βρέθει άλλο token case.
  ```

- `returnStat : return ( expression )`
  Ο κανόνας `returnStat` είναι υπεύθυνος για την δομή επιστροφής τιμής συνάρτησης. Ο έλεγχος για το token `return` έχει προηγηθεί στον κανόνα `statement` επομένως καλεί τον λεκτικό αναλυτή και ελέγχει για το token `return`. Εάν βρεθεί καλεί ξανά τον λεκτικό αναλυτή και ελέγχει για το σύμβολο `(`. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και τον κανόνα `expression`. Τέλος, ελέγχει για το σύμβολο `)` και καλεί τον λεκτικό αναλυτή.

- `callStat : call ID ( actualparlist )`
  Ο κανόνας `callStat` είναι υπεύθυνος για την δομή κλήσης διαδικασίας. Ο έλεγχος για το token `call` έχει προηγηθεί στον κανόνα `statement` επομένως καλεί τον λεκτικό αναλυτή και ελέγχει για token αναγνωριστικού. Εάν βρεθεί καλεί ξανά τον λεκτικό αναλυτή και ελέγχει για το σύμβολο `(`. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και τον κανόνα `actualparlist`. Τέλος, ελέγχει για το σύμβολο `)` και καλεί τον λεκτικό αναλυτή.

- `printStat : print ( expression )`
  Ο κανόνας `printStat` είναι υπεύθυνος για την δομή εξόδου δεδομένων. Ο έλεγχος για το token `print` έχει προηγηθεί στον κανόνα `statement` επομένως καλεί τον λεκτικό αναλυτή και ελέγχει για το σύμβολο `(`. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και τον κανόνα `expression`. Τέλος, ελέγχει για το σύμβολο `)` και καλεί τον λεκτικό αναλυτή.

- `inputStat : input ( ID )`
  Ο κανόνας `inputStat` είναι υπεύθυνος για την δομή εισόδου δεδομένων. Ο έλεγχος για το token `input` έχει προηγηθεί στον κανόνα `statement` επομένως καλεί τον λεκτικό αναλυτή και ελέγχει για το σύμβολο `(`. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και ελέγχει για token αναγνωριστικού. Εάν βρεθεί, ελέγχει για το σύμβολο `)` και καλεί τον λεκτικό αναλυτή.

- ```
  actualparlist : actualparitem ( , actualparitem )*
                | ε
  ```
  Ο κανόνας `actualparlist` διαχειρίζεται την λίστα από παραμέτρους ενός υποπρογράμματος κατά τη κλήση του. Η λίστα παραμέτρων μπορεί να είναι και κενή και άρα δεν έχουμε περίπτωση σφάλματος. Αρχικά καλεί την `actualparitem` μία φορά και στη συνέχεια επαναληπτικά ελέγχει για το σύμβολο `,` καλεί τον λεκτικό αναλυτή και πάλι την `actualparitem`.
  ```

- `actualparitem : in expression`
  `| inout ID`

  Ο κανόνας `actualparitem` είναι υπεύθυνος για την διαχείριση μίας παραμέτρου υποπρογράμματος κατά τη δήλωση του. Ελέγχει ξεχωριστά εάν υπάρχει token με την δεσμευμένη λέξη `in` ή `inout`. Για τον έλεγχο αυτό δεν έχουμε μήνυμα λάθους αφού η λίστα παραμέτρων μπορεί να είναι και κενή. Στη συνέχεια καλεί τον λεκτικό αναλυτή, ελέγχει εάν υπάρχει token αναγνωριστικού και εάν υπάρχει καλεί πάλι τον λεκτικό αναλυτή.

- `condition : boolterm ( or boolterm )*`

  Ο κανόνας `condition` είναι υπεύθυνος για μία boolean έκφραση. Συγκεκριμένα, σύμφωνα με την προτεραιότητα των λογικών τελεστών διαχειρίζεται την λογική πράξη or ενώ οι υπόλοιπες λογικές πράξεις διαχειρίζονται από τους επόμενους κανόνες. Αρχικά, καλεί τον κανόνα `boolterm`. Στη συνέχεια ελέγχει επαναληπτικά για την δεσμευμένη λέξη `or` και καλεί πάλι τον κανόνα `boolterm`. Η επανάληψη τελειώνει όταν δεν βρεθεί άλλη δεσμευμένη λέξη `or`.

- `boolterm : boolfactor ( and boolfactor )*`

  Ο κανόνας `boolterm` είναι υπεύθυνος για έναν όρο σε μία boolean έκφραση. Σύμφωνα με την προτεραιότητα των λογικών τελεστών διαχειρίζεται την λογική πράξη and. Αρχικά, καλεί τον κανόνα `boolfactor`. Στη συνέχεια ελέγχει επαναληπτικά για την δεσμευμένη λέξη `and` και καλεί πάλι τον κανόνα `boolfactor`. Η επανάληψη τελειώνει όταν δεν βρεθεί άλλη δεσμευμένη λέξη `and`.

- `boolfactor : not [ condition ]`
  `| [ condition ]`
  `| expression REL_OP expression`

  Ο κανόνας `boolfactor` είναι υπεύθυνος για έναν παράγοντα σε μία boolean έκφραση και υλοποιείται σε τρία σενάρια. Στο πρώτο σενάριο, διαχειρίζεται την τελευταία λογική πράξη not. Ελέγχει για την δεσμευμένη λέξη `not`. Εάν βρεθεί καλεί τον λεκτικό αναλυτή και ελέγχει για token `[`. Εάν βρεθεί καλεί τον κανόνα `condition` για να διαχειριστεί τυχόν εσωτερικές λογικές εκφράσεις και ελέγχει για token `]`. Εάν δεν βρεθεί η δεσμευμένη λέξη `not` ακολουθεί το δεύτερο σενάριο το οποίο είναι ίδιο με το πρώτο δίχως όμως τον αρχικό έλεγχο για την δεσμευμένη λέξη `not`. Στον τρίτο σενάριο καλεί τον κανόνα `expression` έπειτα τον κανόνα `REL_OP` και τέλος πάλι τον κανόνα `expression`.

- `expression : optionalSign term ( ADD_OP term )*`

  Ο κανόνας `expression` διαχειρίζεται μία αριθμητική έκφραση. Καλεί τους κανόνες `optionalSign` και `term`. Έπειτα καλεί επαναληπτικά τους κανόνες `ADD_OP` και `term` όσο βρίσκει token που αντιστοιχεί στο σύμβολο πρόσθεσης ή αφαίρεσης.

- `term: factor ( MUL_OP factor )*`
  Ο κανόνας `term` είναι υπεύθυνος για έναν όρο σε μία αριθμητική έκφραση. Αρχικά, καλεί τον κανόνα `factor` και έπειτα καλεί επαναληπτικά τους κανόνες `MUL_OP` και `factor` όσο βρίσκει token που αντιστοιχεί στο σύμβολο πολλαπλασιασμού ή διαίρεσης.

- ```
  factor : INTEGER
         | ( expression )
         | ID idtail
  ```
  Ο κανόνας `factor` είναι υπεύθυνος για έναν παράγοντα σε μία αριθμητική έκφραση και υλοποιείται σε τρία σενάρια. Στο πρώτο σενάριο, ελέγχει για token που αντιστοιχεί σε αριθμό και έαν βρεθεί καλέι τον λεκτικό αναλυτή. Στο δεύτερο σενάριο ελέγχει για το σύμβολο `(`. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και τον κανόνα `expression` για να διαχειριστεί τυχών εσωτερικές αριθμητικές εκφράσεις. Τέλος, ελέγχει για το σύμβολο `)` και καλεί τον λεκτικό αναλυτή. Στο τρίτο σενάριο ελέγχει για token αναγνωριστικού και εάν βρεθεί καλεί τον λεκτικό αναλυτή και έπειτα τον κανόνα `idtail.`

- ```
  idtail : ( actualparlist )
         | ε
  ```
  Ο κανόνας `idtail` διαχειρίζεται την παρένθεση και τις παραμέτρους σε μία κλήση υποπρογράμματος. Δεν υπάρχει περίπτωση σφάλματος καθώς ένα υποπρόγραμμα μπορεί να μην δέχεται καμία παράμετρο. Ο κανόνας ελέγχει για το σύμβολο `(`. Εάν το σύμβολο βρεθεί καλεί τον λεκτικό αναλυτή και τον κανόνα `actualparlist` για να διαχειριστεί τις παραμέτρους. Τέλος, ελέγχει για το σύμβολο `)` και καλεί τον λεκτικό αναλυτή.

- ```
  optionalSign : ADD_OP
               | ε
  ```
  Ο κανόνας `optionalSign` διαχειρίζεται τα σύμβολα της πρόσθεσης και αφαίρεσης. Τα σύμβολα μερικές φορές είναι προεραιτικά άρα δεν έχουμε μήνυμα σφάλματος. Έλεγχει εάν έχει βρεθέι token που αντιστοιχεί στο σύμβολο πρόσθεσης ή αφαίρεσης και καλεί και επιστρέφει τον κανόνα `ADD_OP`.

- `REL_OP : =|<=|>=|>|<|<>;`
  Ο κανόνας `REL_OP` είναι υπεύθυνος για τους σχεσιακούς τελεστές της γλώσσας. Έλεγχει εάν έχει βρεθεί token που αντιστοιχεί σε κάποιον σχεσιακό τελεστή και έαν έχει βρεθεί καλεί τον λεκτικό αναλυτή.
  ```

- `ADD_OP : +|-`
  Ο κανόνας `ADD_OP` είναι υπεύθυνος για τους προσθετικούς τελεστές της γλώσσας. Έλεγχει εάν έχει βρεθεί token που αντιστοιχεί σε κάποιον προσθετικό τελεστή και έαν έχει βρεθεί καλεί τον λεκτικό αναλυτή. Σε περίπτωση που δεν βρεθεί τέτοιο token δεν έχουμε μήνυμα σφάλματος αφού οι προσθετικοί τελεστές είναι προεραιτικοί.

- `MUL_OP : *|/`
  Ο κανόνας `MUL_OP` είναι υπεύθυνος για τους πολλαπλασιαστικούς τελεστές της γλώσσας. Έλεγχει εάν έχει βρεθεί token που αντιστοιχεί σε κάποιον προσθετικό τελεστή και έαν έχει βρεθεί καλεί τον λεκτικό αναλυτή.

ΜΕΡΟΣ 4ο: Παραγωγή ενδιάμεσου κώδικα

Στο σημείο αυτό, μπορούμε να προσθέσουμε κώδικα στις συναρτήσεις του συντακτικού αναλυτή που υλοποιούν την γραμματική της γλώσσας Cimple. Το επόμενο βήμα στην διαδικασία της μεταγλώττισης είναι η παραγωγή του ενδιάμεσου κώδικα. Ο ενδιάμεσος κώδικας είναι ένα σύνολο από τετράδες και αξιοποιείται για την παραγωγή του τελικού κώδικα Assembly. Μας βοηθάει στην μεταγλώττιση των δομών της γλώσσας. Π.χ. while, forcase, if κλπ. Ουσιαστικά, μετατρέπουμε το αρχικό πρόγραμμα σε ψευδοεντολές οι οποίες «θυμίζουν» κώδικα σε γλώσσα μηχανής.

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

Στον κώδικα μας, μία τετράδα περιγράφεται από την κλάση **Quad**, η οποία έχει τα εξής πεδία:

- ***counter***: Μοναδικός αριθμός που χαρακτηρίζει την τετράδα.
- ***operation***: Η λειτουργία της τετράδας.
- ***x, y, z***: Τα τρία τελούμενα.

Για λόγους συμβατότητας με τον κώδικα θα αναφερόμαστε σε μία τετράδα και με τον αγγλικό όρο, quad.

Α. Βοηθητικές συναρτήσεις και μεταβλητές

Για την παραγωγή του ενδίαμεσου κώδικα χρησιμοποιούνται οι ακόλουθες global μεταβλητές:

- ***quadCount***: Ακέραιος μετρητής τετράδων.
- ***tempCount***: Ακέραιος μετρητής μεταβλητών temp.
- ***quadsTable***: Λίστα όλων των τετράδων.
- ***blockQuads:*** Λίστα των τετράδων ενός block. Αλλάζει δυναμικά κατά την εκτέλεση της μεταγλώττισης.
- ***tempTable:*** Λίστα των μεταβλητών temp.
- ***varTable:*** Λίστα με αναγνωριστικά μεταβλητών που έχουν δηλωθεί στο αρχικό πρόγραμμα.

Επιπλέον των μεταβλητών έχουν δημιουργηθεί βοηθητικές συναρτήσεις οι οποίες έχουν ομαδοποιηθεί μέσα στην κλάση **interCode** για να διαχωρίζονται από τις υπόλοιπες κλάσεις και συναρτήσεις του μεταγλωττιστή. Οι συναρτήσεις αυτές έχουν δηλωθεί ως *@staticmethod,*

δηλαδή καλούνται με μία στατική αναφορά στο όνομα της κλάσης που ανήκουν και δεν απαιτείται δημιουργία αντικειμένου της κλάσης. Οι βοηθητικές συναρτήσεις είναι οι εξής:

- ***nextQuad()***: Επιστρέφει την τιμή του ακέραιου μετρητή τετράδων.
- ***genQuad(op, x, y, z)***: Δημιουργεί το επόμενο Quad, το προσθέτει στους πίνακες quadsTable και blockQuad και αυξάνει τον μετρητή quadCount κατά ένα.
- ***newTemp()***: Δημιουργεί και επιστρέφει την επόμενη temp μεταβλητή, την προσθέτει στη λίστα tempTable και αυξάνει τον μετρητή tempCount κατά ένα.
- ***emptyList():*** Επιστρέφει μία κενή λίστα.
- ***makeList(x):*** Δημιουργεί και επιστρέφει μία λίστα τετράδων που περιέχει μόνο το x.
- ***merge(list1, list2):*** Ενώνει τις δύο λίστες σε μία την οποία και επιστρέφει.
- ***backpatch(list, z):*** Επισκέπτεται τις τετράδες της λίστας list και συμπληρώνει το τελευταίο πεδίο με z.
- ***outputFile():*** Είναι υπεύθυνη για το output του ενδιάμεσου κώδικα σε αντίστοιχο αρχείο. Καλείται μία φορά μετά τη μεταγλώττιση και γράφει τα περιεχόμενα της λίστας quadsTable στο αρχείο.
- ***outputFileC():*** Είναι υπεύθυνη για την μετατροπή και το output του ενδιάμεσου κώδικα σε εντολές C χαμηλού επιπέδου. Η επεξήγηση της συνάρτησης βρίσκεται στο 8ο μέρος της αναφοράς.

Β. Δομές ενδιάμεσου κώδικα

Έχοντας υλοποιήσει τις βοηθητικές μεταβλητές και συναρτήσεις δεν μένει παρά να προσθέσουμε τις δομές του ενδιάμεσου κώδικα μέσα στις συναρτήσεις της γραμματικής του συντακτικού αναλυτή. Συγκεκριμένα, έχουν υλοποιηθεί οι ακόλουθες δομές:

- ***Αρχή και τέλος Block στον κανόνα block():***
  Καλώντας την genQuad() δημιουργούμε τετράδες που σηματοδοτούν την αρχή και το τέλος ενός block. Για το σκοπό αυτό περνάμε το αναγνωριστικό και μία boolean μεταβλητή ως παραμέτρους στην κανόνα block. Η boolean μεταβλητή μας παρέχει πληροφορία για το εάν το block αναφέρεται στο program ή σε υποπρόγραμμα καθώς στην περίπτωση του κυρίως προγράμματος δημιουργούμε μία επιπλέον τετράδα, την τετράδα halt που σηματοδοτεί το τέλος του προγράμματος.

- ***Εκχώρηση στον κανόνα assignStat():***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:

  S -> id := E {P1};

  Όπου **{P1}:** `interCode.genQuad(':=', E, '_', ID)`

- ***Έξοδος στον κανόνα printStat():***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:
  S -> print (E) {P2}
  Όπου **{P2}:** `interCode.genQuad('out', E, '_', '_')`

- ***Είσοδος στον κανόνα inputStat():***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:
  S -> input (id) {P1}
  Όπου **{P1}:** `interCode.genQuad('inp', ID, '_', '_')`

- ***Δομή if στον κανόνα ifStat():***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:
  S -> if B then {P1} S^1 {P2} TAIL {P3}
  TAIL -> else S^2 | TAIL -> ε
  Όπου **{P1},** συμπλήρωση των τετράδων στο if που έχουν μείνει ασυμπλήρωτες και γνωρίζουμε τώρα ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα:

```
interCode.backpatch(B[0], interCode.nextQuad())
```

  Όπου **{P2},** συμπλήρωση των τετράδων στο else που έχουν μείνει ασυμπλήρωτες και γνωρίζουμε τώρα ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα και δημιουργία τετράδας jump για να εξασφαλίσουμε ότι ο κώδικας του else δεν θα εκτελεστεί:

```
ifList = interCode.makeList(interCode.nextQuad())
interCode.genQuad('jump', '_', '_', '_')
interCode.backpatch(B[1], interCode.nextQuad())
```

  Όπου **{P3},** εξασφαλίσουμε ότι ο κώδικας του else δεν θα εκτελεστεί:

```
interCode.backpatch(ifList, interCode.nextQuad())
```

- ***Λογικές παραστάσεις – OR στον κανόνα condition():***
  Για την διαχείριση της λογικής παράστασης δημιουργούνται ξεχωριστές τετράδες για την αληθή και την ψευδή αποτίμιση της, τις οποίες και επιστρέφει η συνάρτηση του κανόνα. Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:

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

  Όπου **{P1},** μεταφορά των τετράδων από την λίστα Q1 στις αντίστοιχες μεταβλητές**:**

```
conditionTrue = Q1[0]
conditionFalse = Q1[1]
```

  Όπου **{P2},** συμπλήρωση όσων τετράδων μπορούν να συμπληρωθούν εντός του κανόνα**:**

```
interCode.backpatch(conditionFalse, interCode.nextQuad())
```

  Όπου **{P3},** συσσώρευση των τετράδων που δεν μπορούν να συμπληρωθούν και αντιστοιχούν σε αληθή αποτίμιση της λογικής παράστασης:

```
conditionTrue = interCode.merge(conditionTrue, Q2[0])
conditionFalse = Q2[1]
```

- ***Λογικές παραστάσεις – AND στον κανόνα boolterm():***
  Για την διαχείριση της λογικής παράστασης δημιουργούνται ξεχωριστές τετράδες για την αληθή και την ψευδή αποτίμιση της, τις οποίες και επιστρέφει η συνάρτηση του κανόνα. Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:

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

  Όπου **{P1},** μεταφορά των τετράδων από την λίστα R1 στις αντίστοιχες μεταβλητές**:**

```
booltermTrue = R1[0]
booltermFalse = R1[1]
```

  Όπου **{P2},** συμπλήρωση όσων τετράδων μπορούν να συμπληρωθούν εντός του κανόνα**:**

```
interCode.backpatch(booltermTrue, interCode.nextQuad())
```

  Όπου **{P3},** συσσώρευση των τετράδων που δεν μπορούν να συμπληρωθούν και αντιστοιχούν σε ψευδή αποτίμιση της λογικής παράστασης:

```
booltermFalse = R2[0]
booltermTrue = interCode.merge(booltermFalse, R2[1])
```

- ***Λογικές παραστάσεις στον κανόνα boolfactor():***
  Για την διαχείριση της λογικής παράστασης δημιουργούνται ξεχωριστές τετράδες για την αληθή και την ψευδή αποτίμιση της, τις οποίες και επιστρέφει η συνάρτηση του κανόνα. Στον κανόνα αυτό προστίθενται τρείς διαφορετικές δομές, μία για κάθε σενάριο του. Η πρώτη δομή αφορά την λογική πράξη not:

  R -> not (B) {P_1}

  Όπου **{P1},** αντιστροφή και μεταφορά των αληθών τετράδων ως ψευδή και το αντίστροφο**:**

  ```
  boolfactorTrue = B1[1]
  boolfactorFalse = B1[0]
  ```

  Η δεύτερη δομή αφορά το σενάριο όπου δεν υπάρχει η λογική πράξη not:

  R -> (B) {P_1}

  Όπου **{P1},** απλή μεταφορά των τετράδων**:**

  ```
  boolfactorTrue = B1[0]
  boolfactorFalse = B1[1]
  ```

  Η τρίτη δομή αφορά τις σχεσιακές συγκρίσεις:

  R -> E^1 relop E^2 {P_1}

  Όπου **{P1},** δημιουργία μη συμπληρωμένων τετράδων για την αληθή και ψευδή αποτίμηση της relop αντίστοιχα:

  ```
  boolfactorTrue = interCode.makeList(interCode.nextQuad())
  interCode.genQuad(relop, E1, E2, '_')
  boolfactorFalse = interCode.makeList(interCode.nextQuad())
  interCode.genQuad('jump', '_', '_', '_')
  ```

- ***Δομή while στον κανόνα whileStat():***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:

  S -> while {P1} B do {P2} S^1 {P3}

  Όπου **{P1}:** `Bquad = interCode.nextQuad()`

  Όπου **{P2},** συμπλήρωση των τετράδων που έχουν μείνει ασυμπλήρωτες και γνωρίζουμε τώρα ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα:

  ```
  interCode.backpatch(B[0], interCode.nextQuad())
  ```

  Όπου **{P3},** μετάβαση στην αρχή της συνθήκης ώστε να ξαναγίνει έλεγχος και συμπλήρωση τετράδων:

  ```
  interCode.genQuad('jump', '_', '_', Bquad)
  interCode.backpatch(B[1], interCode.nextQuad())
  ```

- ***Δομή switch στον κανόνα switchcaseStat():***
  // TODO

- ***Δομή forcase στον κανόνα forcaseStat():***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:
  S -> forcase {P1} (case (condition) {P2} statements {P3})*
  Όπου **{P1}:**
  ```
  p1Quad = interCode.nextQuad()
  ```
  Όπου **{P2}:**
  ```
  interCode.backpatch(C[0], interCode.nextQuad())
  ```
  Όπου **{P3}:**
  ```
  interCode.genQuad('jump', '_', '_', p1Quad)
  interCode.backpatch(C[1], interCode.nextQuad())
  ```

- ***Δομή incase στον κανόνα incaseStat():***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:
  S -> incase {P1} (case (condition) {P2} statements {P3})* {P4}
  Όπου **{P1}:**
  ```
  w = interCode.newTemp()
  p1Quad = interCode.nextQuad()
  interCode.genQuad(':=', '1', '_', w)
  ```
  Όπου **{P2}:**
  ```
  interCode.backpatch(C[0], interCode.nextQuad())
  interCode.genQuad(':=', '0', '_', w)
  ```
  Όπου **{P3}:**
  ```
  interCode.backpatch(C[1], interCode.nextQuad())
  ```
  Όπου **{P4}:**
  ```
  interCode.genQuad('=', w, '0', p1Quad)
  ```

- ***Εντολή return στον κανόνα returnStat():***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:
  S -> return (E) {P1}
  Όπου **{P1}:**
  ```
  interCode.genQuad('retv', E, '_', '_')
  ```

- ***Παράμετροι κλήσεων υποπρογραμμάτων στον κανόνα actualparitem():***
  Για κάθε παράμετρο που εντοπίζεται να περνά με τιμή δημιουργούμε μία τετράδα ως εξής:
  ```
  interCode.genQuad('par', E, 'CV', '_')
  ```
  Και για κάθε παράμετρο που εντοπίζεται να περνά με αναφορά δημιουργούμε μία τετράδα ως εξής:
  ```
  interCode.genQuad('par', parameterIdentifier, 'REF', '_')
  ```

- ***Κλήση διαδικασίας στον κανόνα callStat():***
  Σε κάθε κλήση διαδικασίας δημιουργούμε μία τετράδα ως εξής:
  ```
  interCode.genQuad('call', procedureIdentifier, '_', '_')
  ```

- ***Κλήση συνάρτησης στον κανόνα idtail():***
  Σε κάθε κλήση συνάρτησης δημιουργούμε μία επιπλέον προσωρινή μεταβλητή, μία τετράδα για την τιμή επιστροφή της συνάρτησης και ακόμη μία τετράδα για την κλήση της συνάρτησης:
  ```
  w = interCode.newTemp()
  interCode.genQuad('par', w, 'RET', '_')
  interCode.genQuad('call', functionIdentifier, '_', '_')
  ```

- ***Προσθετικές αριθμητικές παραστάσεις στον κανόνα expression():***
  Για την διαχείριση των αριθμητικών παραστάσεων χρησιμοποιούμε προσωρινές μεταβλητές έτσι ώστε το αποτέλεσμα να υπολογίζεται βηματικά, όπως και στο επίπεδο του τελικού κώδικα. Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:
  E -> T1 (+ | - T2 {P1})* {P2}
  Όπου **{P1},** δημιουργούμε μία νέα προσωρινή μεταβλητή και μία τετράδα για το μέχρι στιγμής αποτέλεσμα και το μέχρι στιγμής αποτέλεσμα αποθηκεύεται στο T1:
  ```
  w = interCode.newTemp()
  interCode.genQuad(addOperator, T1, T2, w)
  T1 = w
  ```

  Όπου **{P2},** όταν δεν υπάρχει άλλο T2 το τελικό αποτέλεσμα είναι το T1:
  ```
  E = T1
  ```

  Σημειώνεται πως στον κανόνα αυτό, έχει προστεθεί και ένας επιπλέον έλεγχος για το πρόσημο σε περίπτωση αρνητικού αριθμού. Ελέγχουμε εάν ο κανόνας optionalSign έχει επιστρέψει το token της αφαίρεσης και εάν το έχει επιστρέψει το βάζουμε στην αρχή του T1.

- ***Πολλαπλασιαστικές αριθμητικές παραστάσεις στον κανόνα term():***
  Για την διαχείριση των αριθμητικών παραστάσεων χρησιμοποιούμε προσωρινές μεταβλητές έτσι ώστε το αποτέλεσμα να υπολογίζεται βηματικά, όπως και στο επίπεδο του τελικού κώδικα. Στον υπάρχοντα κώδικα του κανόνα προστίθενται η δομή:
  T -> F1 (* | / F2 {P1})* {P2}
  Όπου **{P1},** δημιουργούμε μία νέα προσωρινή μεταβλητή και μία τετράδα για το μέχρι στιγμής αποτέλεσμα και το μέχρι στιγμής αποτέλεσμα αποθηκεύεται στο F1:

```
w = interCode.newTemp()
interCode.genQuad(mulOperator, F1, F2, w)
F1 = w
```

  Όπου **{P2},** όταν δεν υπάρχει άλλο F2 το τελικό αποτέλεσμα είναι το F1:

```
E = F1
```

- ***Αριθμητικές παραστάσεις στον κανόνα factor:***
  Στον υπάρχοντα κώδικα του κανόνα προστίθενται οι δομές:
  F -> (E) {P_1}
  και
  F -> ID {P2}
  Όπου **{P1}:**

```
F = E
```

  Όπου **{P2}:**

```
F = token.tokenString
token = lex()
F = idtail(F)
```

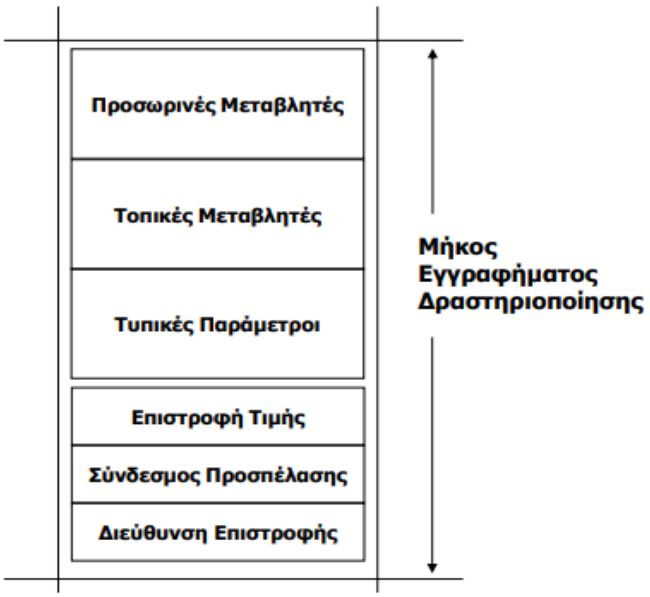
Η υλοποίηση των παραπάνω δομών του ενδιάμεσου κώδικα, έχει ως αποτέλεσμα οι τετράδες να δημιουργούνται και να αποθηκεύονται στη λίστα quadsTable. Με την ολοκλήρωση της συντακτικής ανάλυσης η λίστα αυτή περιέχει όλες τις τετράδες του αρχικού προγράμματος επομένως δεν μένει παρά να καλέσουμε την συνάρτηση interCode.outputFile() για να γράψουμε τις τετράδες σε ένα αρχείο. Εναλλακτικά, δίνοντας το όρισμα **-ic** από την γραμμή εντολών, οι τετράδες τυπώνονται και στην οθόνη. Η λίστα blockQuads περιέχει τις τετράδες μόνο του συγκεκριμένου block που μεταγλωττίζεται μία χρονική στιγμή. Χρησιμοποιείται στην παραγωγή του τελικού κώδικα, και άρα επεξηγείται παρακάτω.

ΜΕΡΟΣ 5ο: Ο πίνακας συμβόλων

Έχοντας υλοποιήσει τον ενδιάμεσο κώδικα, το επόμενο βήμα στην μεταγλώττιση είναι η διαχείριση των μεταβλητών του προγράμματος καθώς και των υποπρογραμμάτων με τις παραμέτρους τους. Για την διαχείριση αυτή, απαιτείται η δημιουργία του πίνακα συμβόλων. Ο πίνακας συμβόλων δημιουργείται και τροποποιείται δυναμικά για κάθε block κατά την διάρκεια της μεταγλώττισης και μας παρέχει πληροφορία για τις μεταβλητές και τα υποπρογράμματα με τις παραμέτρους τους που περιέχει το αρχικό πρόγραμμα. Συγκεκριμένα μέσα από τον πίνακα συμβόλων μπορούμε δούμε την «ορατότητα» που έχει το κάθε block και άρα να ελέγξουμε έαν μια χρήση μεταβλητής ή παραμέτρου είναι έγκυρη.

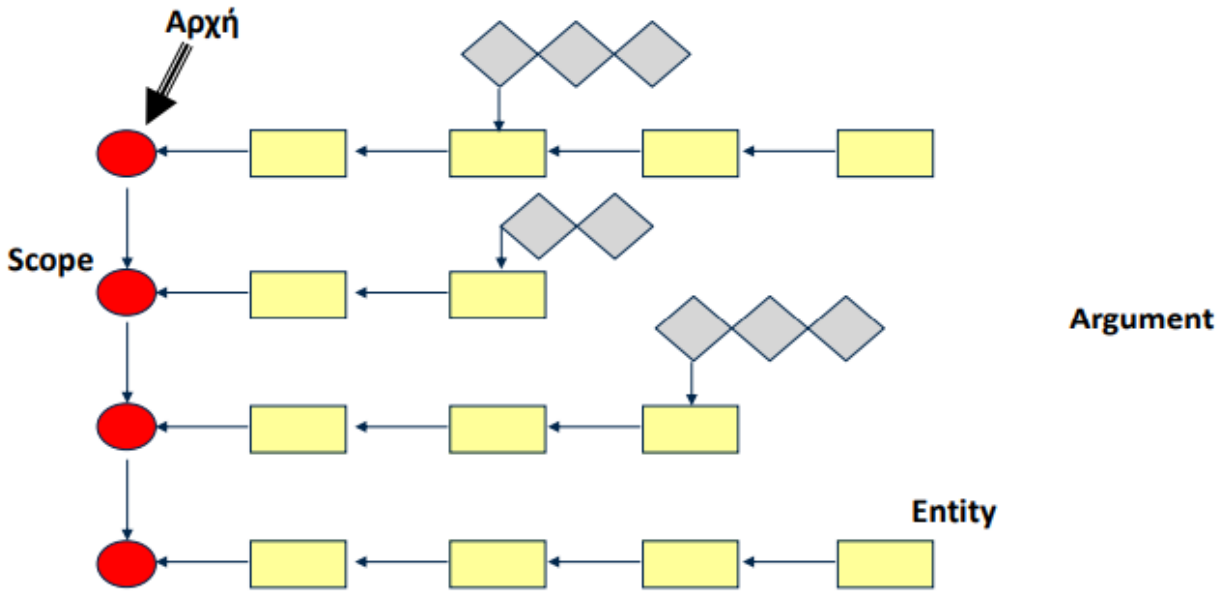
Α. Εγγράφημα δραστηριοποίησης

Η δημιουργία του πίνακα συμβόλων προϋποθέτει γνώση της διαχείρισης μνήμης σε επίπεδο γλώσσας μηχανής. Πιο συγκεκριμένα, η γλώσσα μηχανής διαχειρίζεται τις κλήσεις υποπρογραμμάτων και τις μεταβλητές του προγράμματος με μία στοίβα. Το εγγράφημα δραστηριοποίησης, αποτελεί μία περιγραφή της στοίβας αυτής. Δημιουργείται για κάθε υποπρόγραμμα από αυτό που το καλεί με τον δείκτη στοίβας να βρίσκεται στην αρχή. Πάντα προστίθενται η διεύθυνση επιστροφής, ο σύνδεσμος προσπέλασης και η επιστροφή τιμής με την σειρά αυτή. Ξεκινώντας από το 0, κάθε προσθήκη καταλαμβάνει 4 bytes, επομένως από την θέση 12 και μετά στο εγγράφημα δραστηριοποίησης προστίθενται οι τυπικές παράμετροι, οι τοπικές μεταβλητές και οι προσωρινές μεταβλητές του υποπρογράμματος. Ο λόγος που κρατάμε τις πληροφορίες αυτές είναι για να μπορέσει να συνεχίσει η εκτέλεση του συνολικού προγράμματος έπειτα από την επιστροφή του υποπρογράμματος. Το εγγράφημα δραστηριοποίησης έχει την ακόλουθη μορφή:



Β. Εγγραφές στον πίνακα συμβόλων

Ο πίνακας συμβόλων έχει την ακόλουθη μορφή:



Αρχικά έχουμε μία λίστα από **Scopes** όπου κάθε Scope αναφέρεται σε ένα υποπρόγραμμα ενώ το τελευταίο Scope αναφέρεται στο κυρίως πρόγραμμα. Κάθε Scope περιέχει μία λίστα από **Entities** τα οποία μπορεί να δει όπου κάθε Entity μπορεί να αναφέρεται σε μεταβλητή, υποπρόγραμμα, παράμετρο ή προσωρινή μεταβλητή. Στα Entities που αναφέρονται σε υποπρογράμματα έχουμε μία ακόμη λίστα για τυχών παραμέτρους οι οποίες περιγράφονται ως **Arguments**.

Σε επίπεδο κώδικα οι παραπάνω εγγραφές περιγράφονται από τις εξής τρεις κλάσεις:

- ***Κλάση Scope με τα ακόλουθα πεδία:***
  - **identifier:** Το αναγνωριστικό του Scope.
  - **nestingLevel:** Το βάθος φωλιάσματος.
  - **entities:** Λίστα από Entities.

- ***Κλάση Argument με τα ακόλουθα πεδία:***
  - **identifier:** Το αναγνωριστικό του Argument.
  - **type:** Ο τύπος του Argument. Στην περίπτωση αυτή μπορεί να είναι μόνο ‘arg’.
  - **parMode:** Τρόπος περάσματος παραμέτρου.

- ***Κλάση Entity με τις ακόλουθες 4 υποκλάσεις:***
  - ***Variable με τα ακόλουθα πεδία:***
    - **identifier:** Το αναγνωριστικό του Variable.
    - **type:** Ο τύπος του Entity, π.χ. μεταβλητή, παράμετρος κλπ.
    - **offset:** Απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης.
  - ***Subprogram με τα ακόλουθα πεδία:***
    - **identifier:** Το αναγνωριστικό του Variable.
    - **type:** Ο τύπος του Entity, π.χ. μεταβλητή, παράμετρος κλπ.
    - **startQuad:** Το begin_block quad του υποπρογράμματος.
    - **arguments:** Λίστα από Arguments.
    - **framelength:** Μήκος εγγραφήματος δραστηριοποίησης
  - ***Parameter με τα ακόλουθα πεδία:***
    - **identifier:** Το αναγνωριστικό του Variable.
    - **type:** Ο τύπος του Entity. Στην περίπτωση αυτή μπορεί να είναι μόνο 'par'.
    - **parMode:** Τρόπος περάσματος παραμέτρου.
    - **offset:** Απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης.
  - ***TempVariable με τα ακόλουθα πεδία:***
    - **identifier:** Το αναγνωριστικό του Variable.
    - **type:** Ο τύπος του Entity. Στην περίπτωση αυτή μπορεί να είναι μόνο 'tmp'.
    - **offset:** Απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης.

Γ. Βοηθητικές συναρτήσεις

Για την παραγωγή του πίνακα συμβόλων χρησιμοποιείται η global μεταβλητή **scopes** η οποία είναι μία λίστα με τα scopes που υπάρχουν κάθε χρονική στιγμή της μεταγλώττισης. Επιπλέον έχουν δημιουργηθεί βοηθητικές συναρτήσεις οι οποίες έχουν ομαδοποιηθεί μέσα στην κλάση **symbolTable** για να διαχωρίζονται από τις υπόλοιπες κλάσεις και συναρτήσεις του μεταγλωττιστή. Όπως και οι βοηθητικές συναρτήσεις του ενδιάμεσου κώδικα, έχουν δηλωθεί ως *@staticmethod,* δηλαδή καλούνται με μία στατική αναφορά στο όνομα της κλάσης που ανήκουν και δεν απαιτείται δημιουργία αντικειμένου της κλάσης. Οι βοηθητικές συναρτήσεις είναι οι εξής:

- ***addEntity(entity)***: Προσθέτει το entity στο τελευταίο scope της λίστας scopes.
- ***addScope(identifier)***: Δημιουργεί ένα Scope και το προσθέτει στη λίστα scopes.
- ***removeScope()***: Διαγράφει το τελευταίο Scope στην λίστα scopes.

- ***addArgument(argument):*** Προσθέτει ένα Argument στη λίστα από arguments του τελευταίου Entity του τελευταίου Scope της λίστας scopes.
- ***addParameters():*** Μετατρέπει τα Arguments του προηγούμενου Scope σε Parameter Entities για το επόμενο Scope.
- ***getOffset():*** Υπολογίζει την απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης.
- ***getFramelength():*** Υπολογίζει το μήκος του εγγραφήματος δραστηριοποίησης για ένα υποπρόγραμμα.
- ***getStartQuad():*** Αποθηκεύει το begin_block για ένα υποπρόγραμμα. Είναι σημαντικό να καλεστεί μόλις έχει δημιουργηθεί η τετράδα αυτή.
- ***search(identifier):*** Αναζήτηση ενός Entity με βάση το όνομα του.
- ***outputFile():*** Είναι υπεύθυνη για το output του πίνακα συμβόλων σε αντίστοιχο αρχείο. Καλείται στο τέλος κάθε block και γράφει τα περιεχόμενα της λίστας scopes στο αρχείο. Με το προαιρετικό όρισμα **-st** από τη γραμμή εντολών, ο πίνακας συμβόλων τυπώνεται και στην οθόνη.

Δ. Ενέργειες στον πίνακα συμβόλων

Για την δημιουργία και τις τροποποιήσεις του πίνακα συμβόλων απαιτούνται οι ακόλουθες ενέργειες:

- ***Προσθήκη νέου Scope:*** Όταν ξεκινάει η μεταγλώττιση ενός υποπρογράμματος ή του κυρίως προγράμματος. Δηλαδή, στην αρχή του κανόνα block καλούμε την συνάρτηση addScope.
- ***Διαγραφή Scope:*** Όταν τελειώνει η μεταγλώττιση ενός υποπρογράμματος ή του κυρίως προγράμματος. Δηλαδή, στο τέλος του κανόνα block καλούμε την συνάρτηση removeScope. Με την διαγραφή ενός Scope, διαγράφονται και οι λίστες με Entities και Arguments που περιέχει.
- ***Προσθήκη νέου Entity:***
  - Όταν συναντάμε δήλωση μεταβλητής, δηλαδή στον κανόνα varlist, δημιουργούμε ένα αντικείμενο της κλάσης Variable και καλούμε την συνάρτηση addEntity.
  - Όταν δημιουργείται νέα προσωρινή μεταβλητή, δηλαδή στην βοηθητική συνάρτηση newTemp του ενδιάμεσου κώδικα, δημιουργούμε ένα αντικείμενο της κλάσης TempVariable και καλούμε την συνάρτηση addEntity.
  - Όταν συναντάμε δήλωση νέας συνάρτησης, δηλαδή στον κανόνα subprogram, δημιουργούμε ένα αντικείμενο της κλάσης Subprogram και καλούμε την συνάρτηση addEntity.

  - Όταν συναντάμε δήλωση τυπικής παραμέτρου υποπρογράμματος, δηλαδή στην βοηθητική συνάρτηση addParameters του πίνακα συμβόλων, για κάθε Argument που εντοπίζεται, δημιουργούμε ένα αντικείμενο της κλάσης Parameter και καλούμε την συνάρτηση addEntity.
- ***Προσθήκη νέου Argument:*** Όταν συναντάμε δήλωση τυπικής παραμέτρου υποπρογράμματος, δηλαδή στον κανόνα formalparitem δημιουργούμε ένα αντικείμενο της κλάσης Argument και καλούμε την συνάρτηση addArgument.
- ***Μετατροπή των Arguments σε Parameter Entities:*** Όταν ξεκινάει η μεταγλώττιση ενός υποπρογράμματος (όχι του κυρίως προγράμματος). Δηλαδή, στην αρχή του κανόνα block καλούμε την συνάρτηση addParameters.
- ***Αποθήκευση της τετράδας έναρξης:*** Αμέσως πριν τη δημιουργία του begin_block ενός υποπρογράμματος κρατάμε τον αύξοντα αριθμό του στο Entity που αντιστοιχεί στο υποπρόγραμμα. Δηλαδή, στον κανόνα block, πριν την κλήση της genQuad για το begin_block καλούμε την συνάρτηση getStartQuad.
- ***Αποθήκευση του μήκους εγγραφήματος δραστηριοποίησης:*** Αμέσως πριν τη δημιουργία του end_block ενός υποπρογράμματος κρατάμε το μήκος του εγγραφήματος δραστηριοποίησης. Δηλαδή, στον κανόνα block, πριν την κλήση της genQuad για το end_block καλούμε την συνάρτηση getFramelength.
- ***Έξοδος πίνακα συμβόλου σε αρχείο:*** Στο τέλος του κάθε υποπρογράμματος ή του κυρίως προγράμματος γράφουμε τα μέχρι στιγμής περιεχόμενα του πίνακα συμβόλων στο αντίστοιχο αρχείο. Δηλαδή, στον κανόνα block, πριν την διαγραφή ενός Scope καλούμε την συνάρτηση outputFile.

ΜΕΡΟΣ 6ο: Παραγωγή τελικού κώδικα για την αρχιτεκτονική MIPS

Στο σημείο αυτό δεν μένει παρά να αξιοποιήσουμε τις τετράδες του ενδιάμεσου κώδικα και την πληροφορία που μας παρέχει ο πίνακας συμβόλων για να παράξουμε τον τελικό κώδικα σε γλώσσα μηχανής. Η γλώσσα μηχανής που παράγει ο μεταγλωττιστής είναι για τον επεξεργαστή MIPS. Συγκεκριμένα, για κάθε τετράδα του ενδιάμεσου κώδικα παράγουμε τις αντίστοιχες εντολές του τελικού κώδικα. Στη φάση αυτή οι μεταβλητές απεικονίζονται στην μνήμη, δηλαδή στην στοίβα ενώ ιδιαίτερη προσοχή χρειάζεται στις κλήσεις συναρτήσεων και στο πέρασμα παραμέτρων.

Α. Βοηθητικές συναρτήσεις

// TODO

Β. Παραγωγή τελικού κώδικα

// TODO

ΜΕΡΟΣ 7ο: Χρήση μεταγλωττιστή και έλεγχος

Ο μεταγλωττιστής μπορεί να χρησιμοποιηθεί από το τερματικό ως εξής:

```
python3 cimple_2641_3354.py <cimple file> <optional arg>
```

Όπου `<cimple file>`: Το όνομα/path του αρχείου που περιέχει το αρχικό πρόγραμμα σε γλώσσα Cimple. Η κατάληξη του αρχείου πρέπει να είναι '.ci'.

Όπου `<optional arg>`: Προαιρετικά ορίσματα. Μπορεί να είναι **ένα** από τα ακόλουθα:

- ***-lex :*** Τα tokens της λεκτικής ανάλυσης τυπώνονται στην οθόνη.
- ***-ic :*** Οι τετράδες του ενδιάμεσου κώδικα τυπώνονται στην οθόνη.
- ***-st :*** Ο πίνακας συμβόλων τυπώνεται στην οθόνη.
- ***-asm :*** Ο τελικός κώδικας τυπώνεται στην οθόνη.

Π.χ. `python3 cimple_2641_3354.py fibonacci.ci -st`

Κατά την εκτέλεση ο μεταγλωττιστής δημιουργεί τρία αρχεία τα οποία έχουν τις ακόλουθες καταλήξεις:

- ***.int:*** Περιέχει τις τετράδες του ενδιάμεσου κώδικα.
- ***.sym :*** Περιέχει τον πίνακα συμβόλων.
- ***.asm :*** Περιέχει τον τελικό κώδικα.

Ως ένας τρόπος ελέγχου για την παραγωγή του ενδιάμεσου κώδικα, ο μεταγλωττιστής δημιουργεί ένα ακόμα αρχείο με κατάληξη '**.c**'. Πρόκειται για ένα αρχείο κώδικα σε γλώσσα C στο οποίο οι τετράδες του ενδιάμεσου κώδικα έχουν μετατραπεί σε εντολές C χαμηλού επιπεδού. Η διαδικασία αυτή γίνεται από την συνάρτηση outputFileC της κλάσης interCode. Το αρχείο αυτό μπορεί να μεταγλωττιστεί και να εκτελεστεί όπως όλα τα προγράμματα σε γλώσσα C. Ωστόσο, το αρχείο δημιουργείται μόνο σε περίπτωση που το αρχικό πρόγραμμα Cimple δεν περιέχει υποπρογράμματα, καθώς η προσπάθεια που θα απαιτούνταν αν είχαν συμπεριληφθεί υποπρογράμματα θα ήταν σχεδόν αντίστοιχη με την προσπάθεια που χρειάζεται η παραγωγή του τελικού κώδικα.

Περαιτέρω έλεγχος του τελικού κώδικα μπορεί να γίνει με κάποιον προσομοιωτή της αρχιτεκτονικής MIPS όπως είναι ο MARS. Το παραγόμενο αρχείο που περιέχει τον τελικό κώδικα μπορεί να φορτωθεί στον προσομοιωτή και εκεί να εκτελεστούν οι εντολές που περιέχει.