C-*imple*
Programming language Compiler

Αγαμέμνων Κυριαζής
Χρήστος Περγαμηνέλης
Πανεπιστήμιο Ιωαννίνων
Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Μάιος 2022

Αγαμέμνων Κυριαζής 4400
Χρήστος Περγαμηνέλης 4474

1 Εισαγωγή

Η Cimple είναι μια εκπαιδευτική γλώσσα προγραμματισμού περιορισμένων δυνατοτήτων και λειτουργειών. Αποτελεί μια απλοποίηση της γνωστής γλώσσας C από την οποία αντλεί ιδέες και δομές.

1.1 Λεκτικές μονάδες

Το αλφάβητο της Cimple αποτελείται από τα εξής:

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

Σημειώνεται ότι τα σύμβολα [,] χρησιμοποιούνται στις λογικές παραστάσεις όπως τα σύμβολα (,) στις αριθμητικές παραστάσεις.
Οι δεσμευμένες λέξεις της γλώσσας Cimple είναι οι:

- program (απαραίτητο στην αρχή κάθε προγράμματος)
- declare (δήλωση μεταβλητής)
- procedure | function (δήλωση υπό-προγραμμάτων)
- if | else | while | switchcase | forcase | incase | case | default (εντολές ελέγχου ροής)
- not | and | or (λογικοί τελεστές)
- input | print (είσοδος και έξοδος δεδομένων)
- call (κλήση διαδικασίας)
- return (επιστροφή τιμής από συνάρτηση)
- in | inout (καθορισμός τρόπου περάσματος παραμέτρου σε συνάρτηση ή διαδικασία)

Οι λέξεις αυτές δεν μπορούν να χρησιμοποιηθούν ως μεταβλητές.

Σημειώνεται επιπλέον ότι οι αριθμητικές σταθερές της γλώσσας είναι ακέραιες σταθερές που αποτελούνται από ένα προαιρετικό πρόσημο (+ ή -) και από μια ακολουθία αριθμητικών ψηφίων. Το αποδεκτό πεδίο τιμών για μία αριθμητική σταθερά είναι από $-(2^{32}-1)$ έως $2^{32}-1$. Οποιαδήποτε αριθμητική σταθερά δεν πληροί αυτόν τον κανόνα θα θεωρείται λανθασμένη και η μετάφραση του προγράμματος θα τερματίζεται ανεπιτυχώς.

Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία, αρχίζοντας όμως από γράμμα. Ο μεταγλωττιστής λαμβάνει υπόψη του μόνο τα τριάντα πρώτα γράμματα του αναγνωριστικού. Αναγνωριστικά με μέγεθος περισσότερους από 30 χαρακτήρες θα θεωρούνται λανθασμένα και η μετάφραση του προγράμματος θα τερματίζει ανεπιτυχώς.

1.2 Μορφή προγράμματος

Κάθε πρόγραμμα Cimple ξεκινάει με την λέξη κλειδί program. Στη συνέχεια ακολουθεί ένα αναγνωριστικό (identifier) για το πρόγραμμα αυτό. Μέσα σε άγκιστρα τοποθετούνται τα τρία βασικά block του προγράμματος:

- Δηλώσεις μεταβλητών (declarations).
- Συναρτήσεις και διαδικασίες (subprograms {function | procedure}).
- Εντολές του κυρίως προγράμματος (statements).

Η δομή ενός προγράμματος Cimple φαίνεται παρακάτω:

```
program id {
    declarations
    subprograms
    statements
}.
```

1.3 Τύποι και δηλώσεις μεταβλητών

Ο μοναδικός τύπος δεδομένων που υποστηρίζεται από την γλώσσα προγραμματισμού Cimple είναι οι ακέραιοι αριθμοί μήκους 32 bit. Η δήλωση τους γίνεται με την εντολή declare, την οποία ακολουθεί αναγνωριστικό που αποτελεί το όνομα της μεταβλητής. Επιτρέπεται η δήλωση παραπάνω από μιας μεταβλητής στην ίδια declare χωρίζοντας τες μεταξύ τους με κόμματα “,”. Επίσης επιτρέπεται να έχουμε περισσότερες από μία συνεχόμενες χρήσης της εντολής declare. Τέλος επιτρέπεται να έχουμε περισσότερες των μία συνεχόμενες χρήσεις της declare και το τέλος της κάθε δήλωσης αναγνωρίζεται με το αγγλικό semicolon (ελληνικό ερωτηματικό).

```
declare x, y, z;
declare k;
```

1.4 Τελεστές και εκφράσεις

Η προτεραιότητα των τελεστών από την μεγαλύτερη προς τη μικρότερη είναι:

- Εκφράσεις μέσα σε τελεστές ομαδοποίησης (…) , […]
- Πολλαπλασιαστικοί *, /
- Προσθετικοί +, -
- Σχεσιακοί =, <, >, <>, <=, >=
- Λογικό not
- Λογικό and
- Λογικό or

1.5 Δομές της γλώσσας

1.5.1 Εκχώρηση

Με τον τελεστή εκχώρησης γίνεται ανάθεση της τιμής μιας μεταβλητής ή μιας αριθμητικής σταθερά ή μιας έκφρασης σε μια μεταβλητή. Ο τελεστής εκχώρησης της Cimple είναι όμοιος με τον τελεστή εκχώρησης της Pascal και συντάσσεται ως:

```
ID := expression
```

1.5.2 Απόφαση if

Η εντολή απόφασης if εκτιμά εάν ισχύει η συνθήκη condition, τότε εκτελούνται οι εντολές statements$_1$ που το ακολουθούν. Το else δεν αποτελεί υποχρεωτικό τμήμα της εντολής. Οι εντολές statements$_2$ που ακολουθούν το else εκτελούνται εάν η συνθήκη condition δεν ισχύει. Τα statements πρέπει να περικλείονται υποχρεωτικά γύρω από άγκιστρα εκτός και αν περιέχουν μόνο ένα statement, όπου τότε δεν είναι απαραίτητα. Ο τελεστής απόφασης if συντάσσεται ως:

```
if (condition) {
    statements1
}
else {
    statements2
}
```

1.5.3 Επανάληψη while

Η εντολή επανάληψης while επαναλαμβάνει τις εντολές statements, όσο η συνθήκη condition ισχύει. Σημειώνεται ότι αν την πρώτη φορά που θα αποτιμηθεί η condition, το αποτέλεσμα της αποτίμησης είναι ψευδές, τότε τα statements δεν εκτελούνται ποτέ (εν αντιθέσει με αν είχαμε μια do – while). Η σύνταξη της εντολής while γίνεται ως:

```
while (condition) {
      statements
}
```

1.5.4 Επιλογή switchcase

Η δομής switchcase ελέγχει τις condition που ακολουθούν τα case. Μόλις μία από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες statements$_1$ μετά το condition. Στη συνέχεια ο έλεγχος του προγράμματος μεταβαίνει έξω από την switchcase. Αν κατά τον έλεγχο κανένα από τα cases δεν βρεθεί αληθές, τότε ο έλεγχος μεταβαίνει στην default και εκτελούνται τα statements$_2$ που την ακολουθούν. Τέλος ο έλεγχος μεταβαίνει έξω από την switchcase. Η σύνταξη της δομής γίνεται ως:

```
switchcase
        case (condition)
            statements1
            ...
        default
            statements2
```

1.5.5 Επανάληψη forcase

Η δομή επανάληψης forcase ελέγχει τις condition που ακολουθούν τα case. Μόλις μια από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες $statements_1$ μετά το condition. Μετά ο έλεγχος μεταβαίνει στην αρχή της δομής. Αν καμία από τις case δεν βρεθεί αληθής, τότε ο έλεγχος μεταβαίνει στην default και εκτελούνται τα $statements_2$ που την ακολουθούν. Τέλος ο έλεγχος μεταβαίνει έξω από την forcase. Η δομή συντάσσεται ως:

```
forcase
        case (condition)
            statements1
            ...
        default
            statements2
```

1.5.6 Επανάληψη incase

Η δομή επανάληψης incase ελέγχει τις condition που βρίσκονται μέσα στα case εξετάζοντας κάθε μία κατά σειρά που εμφανίζονται στη δομή. Για κάθε condition όπου βρίσκεται αληθής εκτελούνται τα αντίστοιχα statements που τις ακολουθούν. Αφού εξεταστούν όλες οι condition, αν δεν έχει βρεθεί αληθής καμία από αυτές τότε ο έλεγχος μεταβαίνει εξωτερικά της δομής, ενώ αν έστω και μία από τις case ισχύει η επανάληψη ξεκινάει από την αρχή μεταβιβάζοντας τον έλεγχο του προγράμματος στην αρχή της incase. Η σύνταξη της δομής φαίνεται ακολούθως:

```
incase
        case (condition)
            statements
            ...
```

Σημειώνεται ότι η incase σε αντίθεση με την switchcase και την forcase δεν διαθέτει default statement!

1.5.7 Επιστροφή τιμής συνάρτησης

Χρησιμοποιείται μέσα σε συναρτήσεις για να επιστραφεί το αποτέλεσμα της συνάρτησης. Είναι το γνωστό return statement που περιέχει σχεδόν κάθε δημοφιλής γλώσσα προγραμματισμού.

```
return (expression);
```

1.5.8 Έξοδος δεδομένων

Εμφανίζει στην οθόνη το αποτέλεσμα που περιέχεται μέσα στις παρενθέσεις (expression).

```
print (expression);
```

1.5.9 Είσοδος δεδομένων

Ζητάει από τον χρήστη να δώσει μια τιμή μέσα από το πληκτρολόγιο στην μεταβλητή ID.

```
input (ID);
```

1.5.10 Κλήση διαδικασίας

Καλεί μια διαδικασία. Προσοχή, όχι συνάρτηση…θα δούμε αργότερα πως χειριζόμαστε τις συναρτήσεις :).

```
call procedureName(actualParameters);
```

1.5.11 Συναρτήσεις και διαδικασίες

Η γλώσσα προγραμματισμού Cimple υποστηρίζει συναρτήσεις και διαδικασίες. Για τις συναρτήσεις η σύνταξη είναι:

```
function ID(formalParameters); {
    declarations
    subprograms
    statements
    return (expression);
}.
```

Ενώ για τις διαδικασίες ισχύει:

```
procedure ID(formalParameters); {
    declarations
    subprograms
    statements
}.
```

Η formalParameters είναι μια λίστα τυπικών παραμέτρων. Η παρουσία τους δεν είναι υποχρεωτική στον ορισμό μιας συνάρτησης, συνεπώς η παρένθεση που ακολουθεί το αναγνωριστικό της υπό-ρουτίνας μπορεί να είναι κενή.

Ισχύει ότι τόσο οι συναρτήσεις όσο και οι διαδικασίες μπορούν να φωλιάσουν η μία μέσα. Οι κανόνες εμβέλειας είναι παρόμοιοι με αυτούς της γνωστής γλώσσας προγραμματισμού PASCAL, οι πιο πρόσφατα ορισμένες μεταβλητές επικαλύπτουν αυτές με παρόμοιο όνομα που ορίστηκαν παλαιότερα, ακριβώς όπως λειτουργεί μια στοίβα.

Η κλήση μιας συνάρτησης γίνεται από μία αριθμητική παράσταση ως τελούμενο της όπως φαίνεται στο παράδειγμα που ακολουθεί:

```
D := a + f(in x);
```

Σε περίπτωση που αναρωτηθήκατε, θα δούμε αμέσως τι είναι το in που προηγείται της μεταβλητής x :).

1.6 Μετάδοση παραμέτρων σε υπό-ρουτίνες

Η Cimple υποστηρίζει δύο τρόπους μετάδοσης παραμέτρων:

- Με τιμή, όπου δηλώνεται με την λέξη κλειδί in και οποιεσδήποτε αλλαγές στην τιμή της δεν επιστρέφονται στο πρόγραμμα που κάλεσε την συνάρτηση.
- Με αναφορά, όπου δηλώνεται με την λέξη κλειδί inout και κάθε αλλαγή στην τιμή της μεταβλητής μεταφέρεται αμέσως στο πρόγραμμα που κάλεσε την συνάρτηση.

Στην κλήση μίας συνάρτησης οι πραγματικοί παράμετροι (actualParameters) συντάσσονται μετά από τις λέξεις κλειδιά in και inout ανάλογα με το αν μεταδίδονται με τιμή η αναφορά αντίστοιχα.

1.7 Κανόνες εμβέλειας

Στην Cimple υπάρχουν καθολικές και τοπικές μεταβλητές. Καθολικές μεταβλητές ονομάζονται αυτές που δηλώνονται στο κυρίως πρόγραμμα και είναι προσβάσιμες από το κυρίως πρόγραμμα και κάθε υπό-πρόγραμμα. Τοπικές μεταβλητές είναι αυτές που δηλώνονται σε μία συνάρτηση ή διαδικασία και είναι προσβάσιμες μόνο από τη συγκεκριμένη συνάρτηση ή διαδικασία καθώς και τους απογόνους της, δηλαδή τις υπό-ρουτίνες που βρίσκονται σε μεγαλύτερο επίπεδο φωλιάσματος από αυτή (τη συνάρτηση ή διαδικασία).

Ισχύει ο κανόνας ότι αν δύο (ή περισσότερες) μεταβλητές ή παράμετροι έχουν το ίδιο όνομα και έχουν δηλωθεί σε διαφορετικό επίπεδο φωλιάσματος, τότε οι τοπικές μεταβλητές της υπό-ρουτίνας υπερκαλύπτουν τις μεταβλητές των προγόνων.

Τέλος ισχύει ότι μια συνάρτηση ή διαδικασία έχει δικαίωμα να καλέσει τον εαυτό της (αναδρομικές κλήσεις) και οποιαδήποτε άλλη συνάρτηση ή διαδικασία βρίσκεται στο ίδιο ή μικρότερο επίπεδο φωλιάσματος με αυτήν αρκεί η δήλωση της να προηγείται στον κώδικα.

2 Λεκτική ανάλυση

Η λεκτική ανάλυση ενός προγράμματος γραμμένο σε γλώσσα προγραμματισμού Cimple αποτελεί την πρώτη φάση της μεταγλώττισης. Κατά την λεκτική ανάλυση, διαβάζεται το πηγαίο πρόγραμμα και παράγονται λεκτικές μονάδες, tokens. Σε αυτή τη φάση της εργασίας υλοποιήσαμε σε python3 τον λεκτικό αναλυτή ο οποίος διαβάζει ένα πηγαίο πρόγραμμα Cimple το οποίο δίνεται ως όρισμα πριν την εκτέλεση και παράγει λεκτικές μονάδες, tokens.

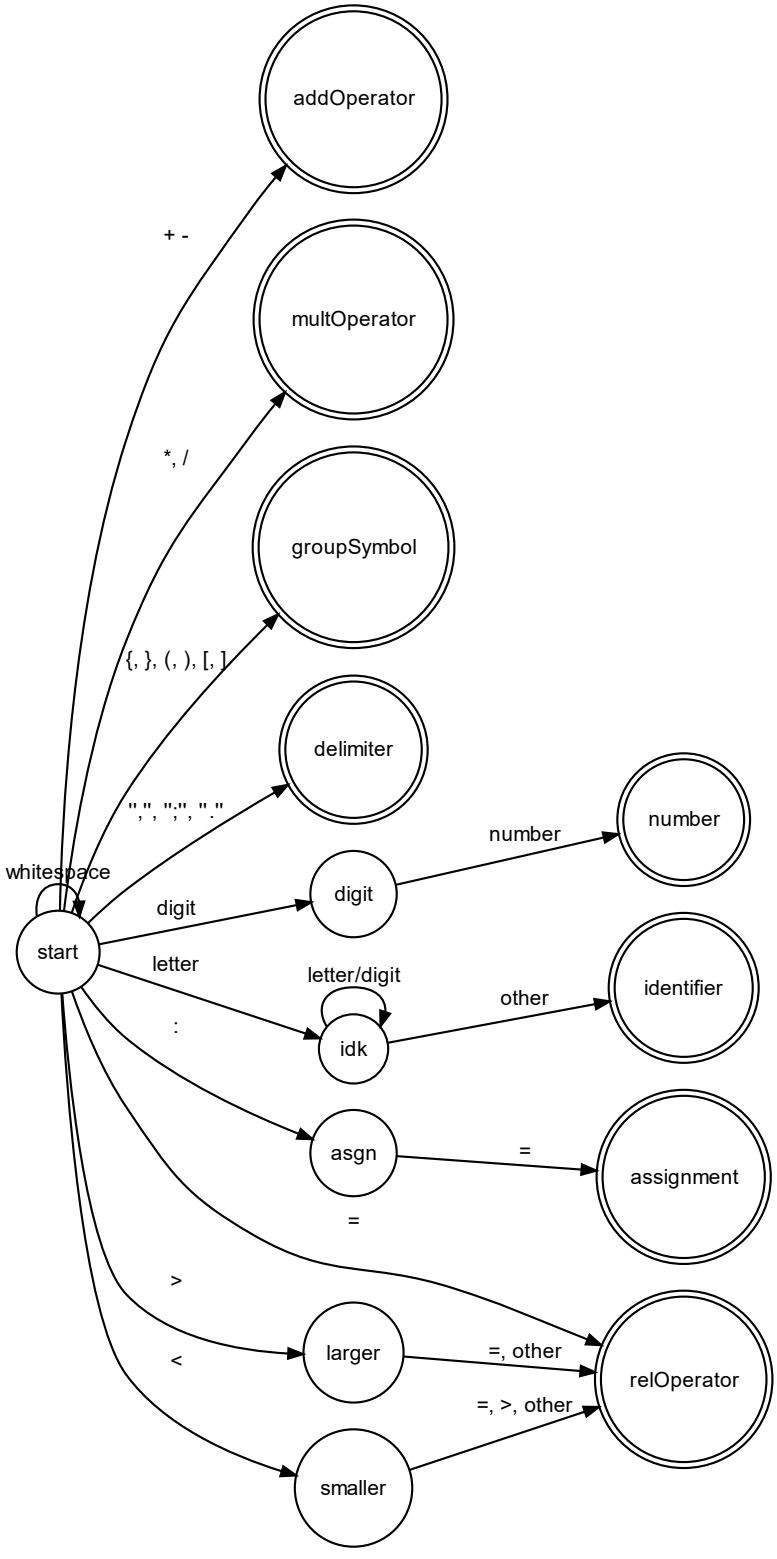
2.1 Λεκτικές μονάδες

Για να αναπαραστήσουμε τις λεκτικές μονάδες κατασκευάσαμε την κλάση Token η οποία αποθηκεύει την οικογένεια που ανήκει η λεκτική μονάδα (family), το αλφαριθμητικό της λεκτικής μονάδας (recognized string) και την γραμμή στην οποία το αναγνωρίσαμε (line number).

```
class Token:

    def __init__(self, family, recognized_string, line_number):
        self.family = family
        self.recognized_string = recognized_string
        self.line_number = str(line_number)
```

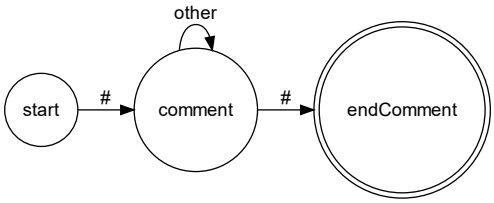
2.2 Εσωτερική λειτουργία του λεκτικού αναλυτή

Η λειτουργία του λεκτικού αναλυτή βασίζεται σε ένα πεπερασμένο αιτιοκρατικό αυτόματο. Για την Cimple το αυτόματο που αναγνωρίζει τις λεκτικές καταστάσεις φαίνεται στο σχήμα 1 που ακολουθεί. Στο αυτόματο του σχήματος η αρχική κατάσταση είναι η κατάσταση start. Οι μη τελικές καταστάσεις αναπαρίστανται με κύκλο ενώ οι τελικές με διπλό κύκλο.



Εικόνα 1

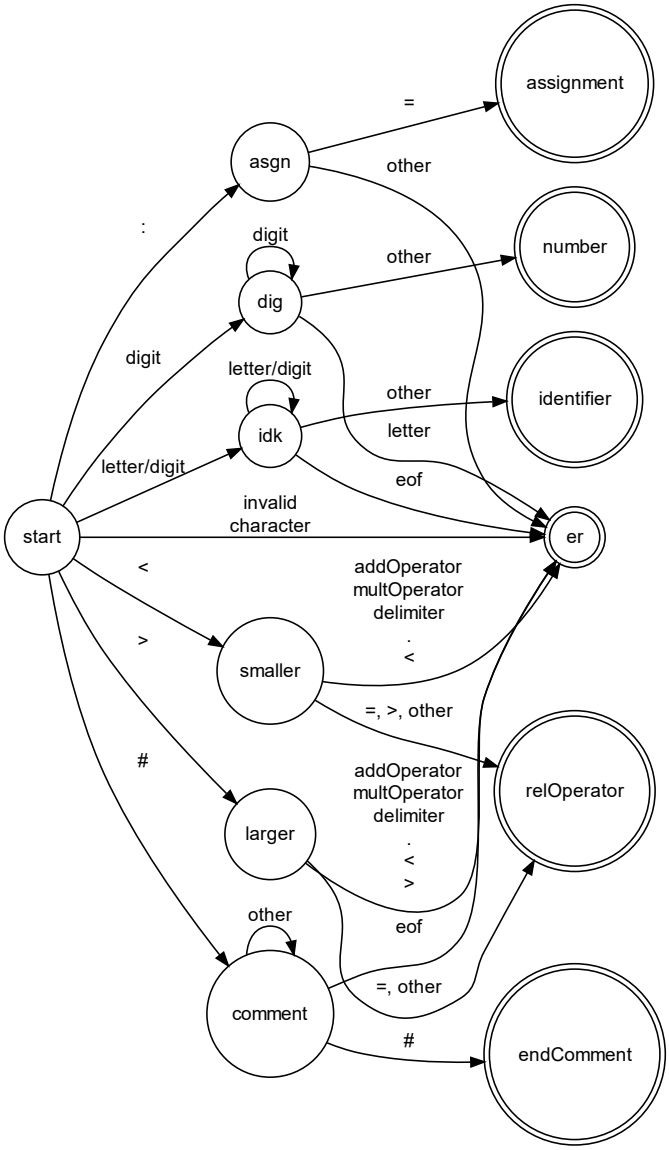
Για την αναγνώριση των σχολίων εντός του προγράμματος ισχύει το παρακάτω αυτόματο, το οποίο αποτελεί συνέχεια του F.S.M. που δείξαμε στην εικόνα 1:



Εικόνα 2

2.2 Διαχείριση σφαλμάτων

Η διαχείριση σφαλμάτων του λεκτικού αναλυτή περιλαμβάνει τα σφάλματα τα οποία σχετίζονται με λεκτικές μονάδες και ανακαλύπτονται κατά τη διάσχιση του αυτόματου. Σε γενικές γραμμές αυτά τα σφάλματα είναι απλό να εντοπιστούν με μια μικρή επέκταση των καταστάσεων του ήδη υπάρχοντος αυτόματου που δείξαμε στην εικόνα 1.



Εικόνα 3

Επιπλέον σφάλματα που αναγνωρίζονται από τον λεκτικό αναλυτή αποτελούν αυτά που αναλύσαμε στην ενότητα 1.1 για τις λεκτικές μονάδες της γλώσσας Cimple καθώς και το μέγιστο (και ελάχιστο) μέγεθος που μπορεί να λάβει μία αριθμητική σταθερά. Το πρώτο σφάλμα περιγράφεται ως V.L.I. σφάλμα (Very Large Identifier) και αφορά αναγνωριστικά με μέγεθος μεγαλύτερο των 30 χαρακτήρων. Το δεύτερο σφάλμα περιγράφεται ως V.L.N. σφάλμα (Very Large Number) και αφορά τις αριθμητικές σταθερές με τιμή μεγαλύτερη του 2^{32}-1 ή μικρότερη του -2^{32}-1. Τα σφάλματα αυτά δεν εμφανίζονται στο αυτόματο αλλά αντ' αυτού έχουν υλοποιηθεί με δομές ελέγχου if της python και εξετάζονται μετά την αναγνώριση της λεκτικής μονάδας από τον λεκτικό αναλυτή.

2.3 Γενικός πίνακας καταστάσεων του αυτόματου του λεκτικού αναλυτή

	Input:													
State:	blank	digit	letter	addOperator	multOperator	groupSymbol	delimiter	colonSign	lessSign	greaterSign	equalsSign	commentSign	eof	other
start	start	dig	idk	addOperator	multOperator	groupSymbol	delimiter	asgn	smaller	larger	relOperator	comment	eof	er
dig	number	dig	er	number	number	number	number	number	number	number	number	number	er	number
idk	identifier	idk	idk	identifier	identifier	identifier	identifier	identifier	identifier	identifier	identifier	identifier	er	identifier
asgn	er	er	er	er	er	er	er	er	er	er	assignment	er	er	er
smaller	relOperator	relOperator	relOperator	er	er	relOperator	er	er	er	relOperator	relOperator	er	er	relOperator
larger	relOperator	relOperator	relOperator	er	er	relOperator	er	er	er	er	relOperator	er	er	relOperator
comment	comment	comment	comment	comment	comment	comment	comment	comment	comment	comment	comment	endComment	er	comment

2.4 Λεκτική ανάλυση προγράμματος και βοηθητικές συναρτήσεις

Κάθε λεκτική μονάδα η οποία αναγνωρίζεται τοποθετείτε στην ουρά που βρίσκεται στην κλάση LexicalAnalyzer. Στην συνέχεια αφού ολοκληρωθεί η λεκτική ανάλυση του προγράμματος, σειρά έχει η απόδοση νοήματος στις λεκτικές μονάδες μέσω του συντακτικού αναλυτή. Για αυτό υλοποιήθηκαν δυο μέθοδοι που θα χρησιμοποιήσουμε αργότερα :) :

- Η get_next_token( ), μας επιστρέφει την πιο πρόσφατη λεκτική μονάδα που αναγνώρισε ο λεκτικός αναλυτής.
- Η pop_next_token( ), αφαιρεί την πιο πρόσφατη λεκτική μονάδα από την ουρά και μας την επιστρέφει.

2.5 Παράδειγμα εκτέλεσης

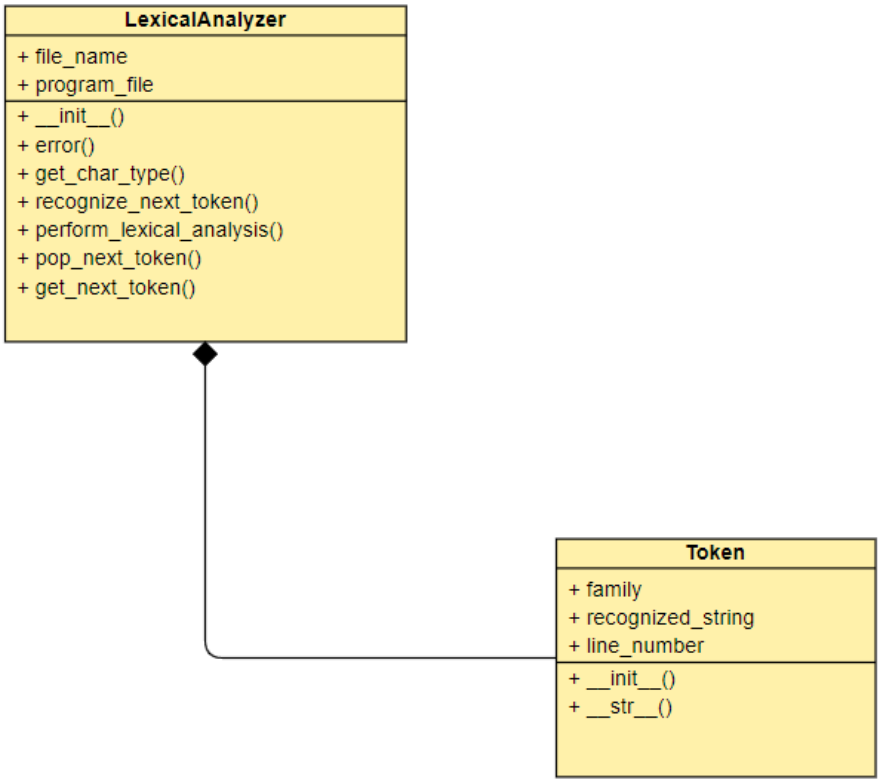
Ας δούμε ένα παράδειγμα εκτέλεσης του λεκτικού αναλυτή για ένα απλό πρόγραμμα Cimple. Το program summation ζητάει από τον χρήστη να του δώσει ως είσοδο από το πληκτρολόγιο έναν αριθμό x, εκτελεί την πράξη $sum := \sum_{i=1}^{x} i$ και εκτυπώνει το sum.

```
program summation
{
    declare x,sum;

    input(x);
    sum := 0;
    forcase
    case(x>0)
    {
        sum := sum+x;
        x := x-1
    }
    default
        print(sum);
}.
```

eof	eof	16
.	delimiter	15
}	groupSymbol	15
;	delimiter	14
)	groupSymbol	14
sum	identifier	14
(	groupSymbol	14
print	keyword	14
default	keyword	13
}	groupSymbol	12
1	number	11
-	addOperator	11
x	identifier	11
:=	assignment	11
x	identifier	11
;	delimiter	10
x	identifier	10
+	addOperator	10
sum	identifier	10
:=	assignment	10
sum	identifier	10
{	groupSymbol	9
)	groupSymbol	8
0	number	8

>	relOperator	8
x	identifier	8
(	groupSymbol	8
case	keyword	8
forcase	keyword	7
;	delimiter	6
0	number	6
:=	assignment	6
sum	identifier	6
;	delimiter	5
)	groupSymbol	5
x	identifier	5
(	groupSymbol	5
input	keyword	5
;	delimiter	3
sum	identifier	3
,	delimiter	3
x	identifier	3
declare	keyword	3
{	groupSymbol	2
summation	identifier	1
program	keyword	1

2.6 Διάγραμμα κλάσεων λεκτικού αναλυτή



Εικόνα 4

3 Συντακτική ανάλυση

Τη φάση της λεκτικής ανάλυσης ακολουθεί η φάση της συντακτικής ανάλυσης. Κατά τη συντακτική ανάλυση ελέγχεται εάν η ακολουθία των λεκτικών μονάδων (tokens) , που σχηματίστηκαν από τον λεκτικό αναλυτή, αποτελεί μια νόμιμη ακολουθία με βάση τη γραμματική της γλώσσας. Έτσι ο συντακτικός αναλυτής δέχεται ως είσοδο ένα αντικείμενο λεκτικού αναλυτή LexicalAnalyzer, αφού γίνει η διαδικασία της λεκτικής ανάλυσης. Το αντικείμενο αυτό επιτρέπει στον συντακτικό αναλυτή να έχει πρόσβαση και να διαχειρίζεται τις λεκτικές μονάδες που δημιουργήθηκαν στην προηγούμενη φάση.

3.1 Γραμματική και μεθοδολογία

Η γραμματική της Cimple ανήκει στις γραμματικές άνευ συμφραζομένων. Θα μπορούσαμε να υλοποιήσουμε μια γραμματική με συμφραζόμενα, αλλά κάτι τέτοιο παρουσιάζει αυξημένη πολυπλοκότητα στην υλοποίηση και θα απαιτούσε υπερβολικά περισσότερο χρόνο για να εκτελεστεί.

Επιπλέον, η γραμματική της Cimple είναι κατάλληλα σχεδιασμένη, για υλοποίηση των κανόνων της με την τεχνική της προβλέπουσας αναδρομικής κατάβασης. Η προβλέπουσα αναδρομική κατάβαση στηρίζεται στην πρόγνωση του κατάλληλου κάθε φορά κανόνα με βάση την λεκτική μονάδα που ακολουθεί κάθε φορά.

Για την υλοποίηση του συντακτικού αναλυτή λοιπόν αναπτύχθηκαν οι παρακάτω κανόνες γραμματικής:

```
perform_syntax_analysis :=
    program ID
    block
    .
    eof

block :=
    '{'
    declarations
    subprograms
    block_statements
    '}'

subprograms :=
    (subprogram)*

subprogram :=
    (procedure | function)
    ID
    formalParList
    block

formalParList :=
    formalParItem (, formalParItem)* | e

formalParItem :=
    in ID | inout ID

block_statements :=
    statement (; statement)*

statement :=
    ID |
    if |
    while |
    switchcase |
    forcase |
    incase |
    return |
    print |
    input |
    call |
    e
```

```
incase_statement :=
    incase
    (case '(' condition ')' statements)*

forcase_statement :=
    forcase
    (case '(' condition ')' statements)*
    default statements

switchcase_statement :=
    switchcase
    (case '(' condition ')' statement)*
    default statements

while_statement :=
    while '(' condition ')' statements

if_statement :=
    if '(' condition ')' statements
    else_statement

else_statement :=
    else statements |
    e

statements :=
    statement ; |
    '{' statement
    (; statement)*
    '}'

condition :=
    boolterm (or boolterm)*

boolterm :=
    boolfactor (and boolfactor)*

boolfactor :=
    not '[' condition ']' |
    '[' condition ']' |
    expression [>|<|>=|<=|=|<>] expression
```

```
assign_statement :=
    ID ':=' expression

actualParList :=
    actualParItem (, actualParItem)* |
    e

actualParItem :=
    in expression | inout ID

declarations :=
    (declare varlist ;)*

varlist :=
    ID (, ID)* |
    e

expression :=
    [+|-|e] term
    ([+|-] term)*

term :=
    factor [*|/] factor

factor :=
    number |
    '(' expression ')' |
    ID idtail


call_statement :=
    call ID '(' actualParList ')'

input_statement :=
    input '(' ID ')'

print_statement :=
    print '(' expression ')'

return_statement :=
    return '(' expression ')'

idtail :=
    '(' actualParList ')' |
    e
```

3.2 Υλοποίηση

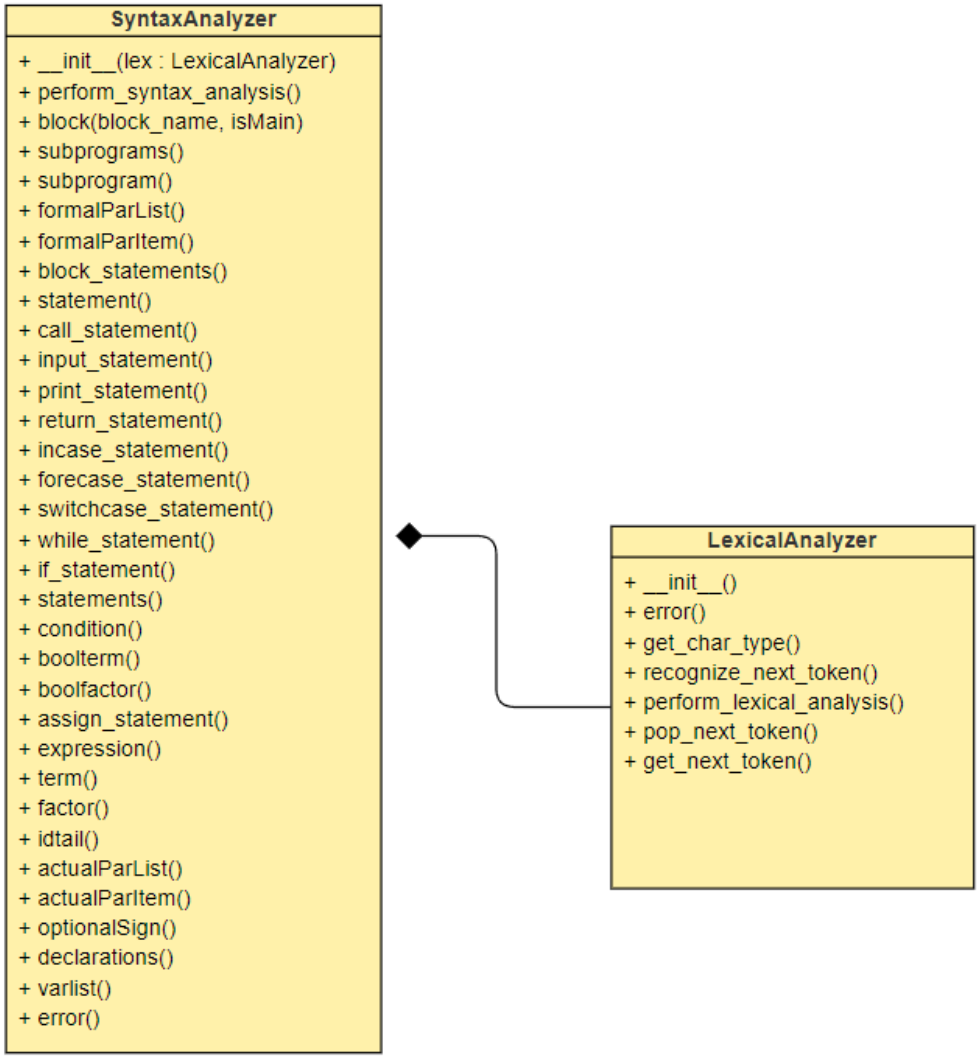
Έχοντας λοιπόν δομήσει την γραμματική της Cimple προχωρήσαμε στην υλοποίηση του κάθε statement ως μια συνάρτηση python η οποία αξιοποιεί αναδρομικές κλήσεις άλλων κανόνων, συναρτήσεων, ανάλογα με το σύμβολο το οποίο συναντάται κατά την εκτέλεση της διαδικασίας της συντακτικής ανάλυσης.

Παραδείγματος χάρη, η μέθοδος expression θα καλέσει την μέθοδο term. Η μέθοδος term την factor. Η factor είτε θα επιστρέψει έναν ακέραιο αριθμό, είτε αν συναντήσει μια αρχή παρένθεσης θα καλέσει την expression, είτε αν συναντήσει ένα αναγνωριστικό θα ενεργοποιήσει τον κανόνα idtail.

3.3 Διάγραμμα κλάσεων συντακτικού αναλυτή

Σημειώνεται ότι το αντικείμενο lex που υπάρχει μέσα στον συντακτικό αναλυτή δεν δημιουργείται εσωτερικά του συντακτικού αναλυτή. Δημιουργείται στην main κλάση του προγράμματος (Main) την οποία θα δούμε αρκετά αργότερα και αφού ολοκληρωθεί η διαδικασία της λεκτικής ανάλυσης περνιέται ως όρισμα στον constructor της SyntaxAnalyzer.

Παρακάτω παρατίθεται το διάγραμμα κλάσεων για τον συντακτικό αναλυτή:



Εικόνα 5

4 Παραγωγή ενδιάμεσου κώδικα

Ταυτόχρονα της συντακτικής ανάλυσης εκτελείται και η διαδικασία που ονομάζουμε παραγωγή ενδιάμεσου κώδικα Η μετατροπή του κώδικα σε Cimple από την αρχική γλώσσα σε γλώσσα assembly RISC-V δεν γίνεται απευθείας. Μεσολαβεί η μετατροπή του σε μία ενδιάμεση γλώσσα, την οποία λέμε και ενδιάμεση αναπαράσταση. Αν και αυτή η γλώσσα εξακολουθεί να θεωρείται γλώσσα υψηλού επιπέδου, οι δομές που την αποτελούν είναι αρκετά απλές.

Η παραγωγή του ενδιάμεσου κώδικα, γίνεται ταυτόχρονα με την συντακτική ανάλυση του προγράμματος που δίνεται στην αρχή ως είσοδος. Μόλις ολοκληρωθεί η ανάπτυξη του συντακτικού δένδρου ενός κανόνα της Cimple, χρησιμοποιούμε την δομή του για να παράξουμε τις ψευδό-εντολές του ενδιάμεσου κώδικα.

4.1 Ενδιάμεση αναπαράσταση κώδικα

Ένα πρόγραμμα Cimple συμβολισμένο στην ενδιάμεση γλώσσα αποτελείται από μία σειρά από τετράδες (Quads). Οι τετράδες αυτές είναι αριθμημένες με έναν μοναδικό αύξων κωδικό (id) βάσει της σειράς με την οποία παράχθηκαν. Αυτή η λειτουργία προστέθηκε προκειμένου να μπορούμε να αναφερθούμε σε κάθε τετράδα χρησιμοποιώντας τον αριθμό της ως ετικέτα.

Κάθε τετράδα αποτελείτε από έναν τελεστή (operator) και τρία τελούμενα (operands[1,2,3]). Ο τελεστής καθορίζει την ενέργεια που πρόκειται να γίνει και τα τελούμενα είναι εκείνα που θα συμμετάσχουν στην πραγματοποίηση της ενέργειας. Οι τετράδες του ενδιάμεσου κώδικα αποθηκεύονται σε μια κατάλληλη δομή στην μνήμη όσο εκτυλίσσεται η συντακτική ανάλυση. Στην γλώσσα Python ο καταλληλότερος τρόπος για να αναπαραστήσουμε τις τετράδες είναι η κλάση Quad.

```
class Quad:

    def __init__(self, operator, operand1, operand2, operand3):
        self.operator = operator
        self.operand1 = operand1
        self.operand2 = operand2
        self.operand3 = operand3

    def __str__(self):
        return str(self.operator) + ", " + str(self.operand1) + ", " +
str(self.operand2) + ", " + str(self.operand3)
```

4.2 Εντολές ενδιάμεσης αναπαράστασης

Παρακάτω θα περιγράψουμε τις εντολές της ενδιάμεσης γλώσσας που θα χρησιμοποιήσουμε για τη μετατροπή του αρχικού κώδικα Cimple σε ενδιάμεση αναπαράσταση. Η ενδιάμεση αναπαράσταση δεν είναι, όχι απαραίτητα, ορατή στον χρήστη αλλά παράγεται και καταναλώνεται εσωτερικά του μεταγλωττιστή μας.

- Εντολές ομαδοποίησης block κώδικα, χρησιμοποιείται στις συναρτήσεις, στις διαδικασίες καθώς και στο κυρίως πρόγραμμα προκειμένου να ομαδοποιηθούν οι εντολές που περιέχουν:

```
begin_block, name, _, _
end_block, name, _, _
```

- Εντολή εκχώρησης, εκχωρεί την τιμή του source στην μεταβλητή target:

```
:=, source, _, target
```

- Αριθμητική πράξη:

```
op, operand1, operand2, target
```

- Εντολή άλματος άνευ συνθήκης, μεταφέρει τον έλεγχο στην εντολή με ετικέτα label:

```
jump, _, _, label
```

- Εντολή άλματος υπό συνθήκη, πραγματοποιεί τον λογικό έλεγχο που περιγράφει τον conditional_jump μεταξύ του operand1 και του operand2. Αν ο έλεγχος αποβεί αληθής τότε ο έλεγχος μεταβαίνει στην εντολή με ετικέτα label. Διαφορετικά ο έλεγχος μεταβαίνει στην αμέσως επόμενη εντολή:

```
conditional_jump , operand1, operand2, label
```

- Δήλωση κλήσης συνάρτησης ή διαδικασίας:

```
call, name, _, _
```

- Πέρασμα πραγματικής παραμέτρου, την συγκεκριμένη εντολή την συναντάμε, αν υπάρχει, αμέσως πριν την κλήση μιας συνάρτησης ή διαδικασίας. Κάθε παράμετρος που βρίσκεται πάνω από μια εντολή call ανήκει στην συνάρτηση που καλείται:

```
par, name, mode, _
```

Το πεδίο name είναι το όνομα της παραμέτρου ενώ το mode είναι ο τρόπος περάσματος. Πιθανές τιμές για το πεδίο mode είναι { cv αν γίνεται πέρασμα με τιμή, ref αν γίνεται πέρασμα με αναφορά και τέλος ret αν πρόκειται για την μεταβλητή που θα αποθηκεύσει το αποτέλεσμα της συνάρτησης }.

- Είσοδος/ έξοδος:

```
in, v, _, _
out, v, _, _
```

- Εντολή τερματισμού προγράμματος, είναι η γνωστή “.” που συναντάμε στο τέλος κάθε προγράμματος:

```
halt, _, _, _
```

4.3 Κλάση ενδιάμεσου κώδικα και βοηθητικές συναρτήσεις

Για το λογισμικό που παράγει τον ενδιάμεσο κώδικα σχεδιάσαμε την κλάση IntermediateCode. Στην κλάση αυτή αποθηκεύεται η λίστα με τις τετράδες που συνθέτουν τον ενδιάμεσο κώδικα καθώς και υλοποιούνται ορισμένες βοηθητικές συναρτήσεις.

Οι βοηθητικές συναρτήσεις που υλοποιήσαμε είναι οι ακόλουθες:

- empty_list():
  Δημιουργεί και επιστρέφει μία κενή λίστα στην οποία στη συνέχεια θα τοποθετηθούν ετικέτες τετράδων.

- make_list(label)
  Δημιουργεί και επιστρέφει μία κενή λίστα η οποία έχει σαν μοναδικό στοιχείο της την ετικέτα τετράδας label.

- merge(list1, list2)
  Δημιουργεί μία λίστα και συνενώνει τις list1 και list2 σε αυτήν.

- gen_quad(op, x, y, z)
  Δημιουργεί μια τετράδα, το πρώτο πεδίο της οποίας είναι το operator και τα τρία επόμενα x, y, z.

- backpatch(list_obj, label)
  Διαβάζει μία προς μία τις τετράδες που σημειώνονται στην λίστα list_obj και για την τετράδα που αντιστοιχεί στην ετικέτα αυτή, συμπληρώνουμε το τελευταίο πεδίο της με το label.

- next_quad()
  Επιστρέφει την ετικέτα της επόμενης τετράδας που θα δημιουργηθεί όταν κληθεί η gen_quad(…).

- new_temp()
  Επιστρέφει το όνομα της επόμενης προσωρινής μεταβλητής.

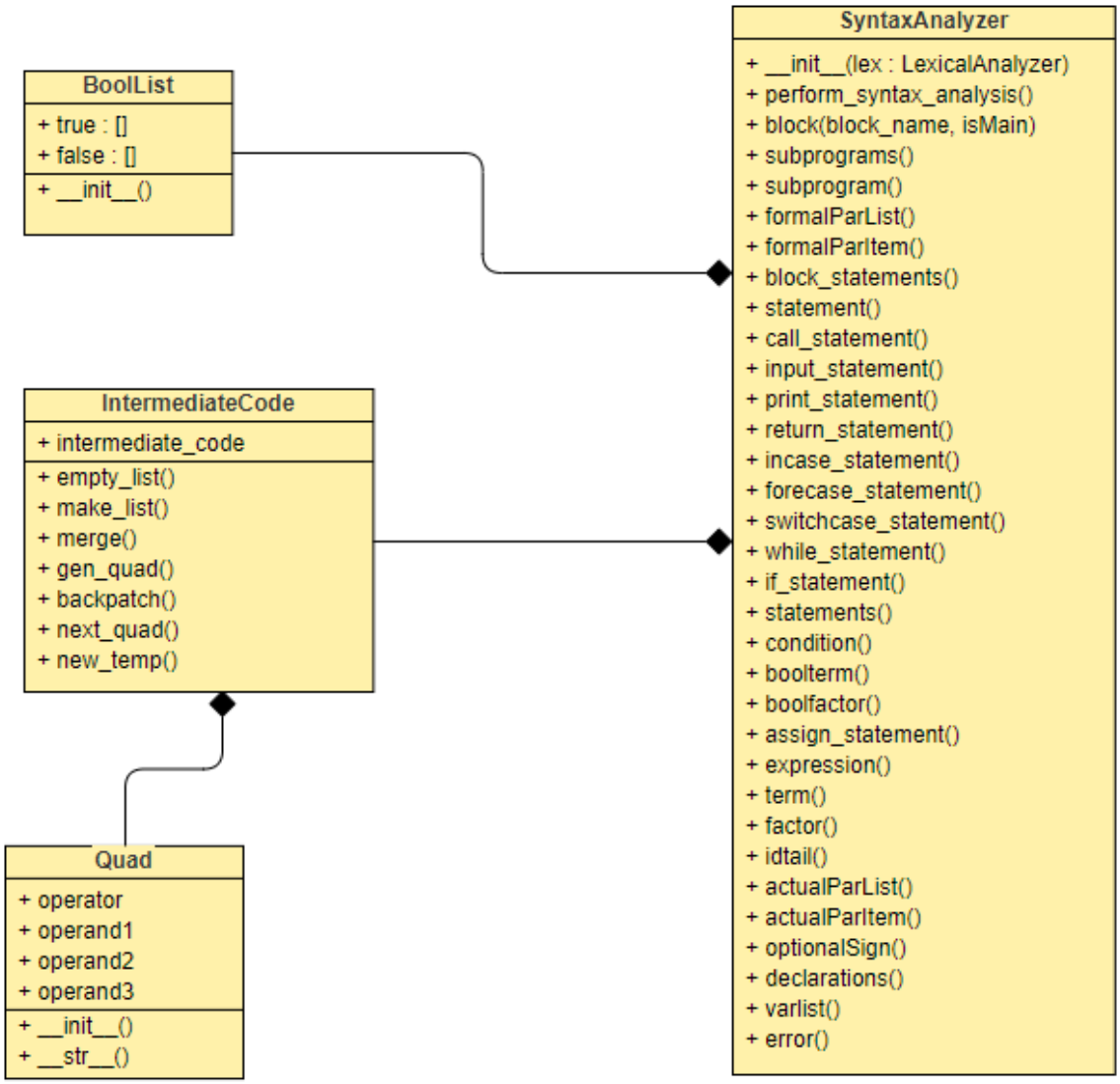
4.5 Η βοηθητική κλάση BoolList

Η κλάση BoolList αποθηκεύει μια λίστα true και μια λίστα false.

- Η λίστα true αποτελείται από όλες τις τετράδες αλμάτων που έχουν μείνει ασυμπλήρωτες διότι ο κανόνας αδυνατεί να τις συμπληρώσει. Οι τετράδες πρέπει να συμπληρωθούν με την ετικέτα της τετράδας εκείνης στην οποία θα μεταβεί ο έλεγχος αν οι λογικές συνθήκες που υπάρχουν στη λίστα είναι αληθής.

- Η λίστα false ομοίως με την λίστα true αποτελείται από όλες τις τετράδες αλμάτων που έχουν μείνει ασυμπλήρωτες διότι ο κανόνας αδυνατεί να τις συμπληρώσει. Οι τετράδες πρέπει να συμπληρωθούν με την ετικέτα της τετράδας εκείνης στην οποία θα μεταβεί ο έλεγχος αν οι λογικές συνθήκες που υπάρχουν στη λίστα είναι ψευδής.

4.6 Διάγραμμα κλάσεων ενδιάμεσου κώδικα



Εικόνα 6

4.5 Παράδειγμα ενδιάμεσου κώδικα

Για το παρακάτω πρόγραμμα:

```
program fibonacci
{
    declare y;
    function fibonacci(in x)
    {
        if(x <= 1)
            return (x);;
        return (fibonacci(in x-1)+fibonacci(in x-2));
    }
    # main #
    input(y);
    print(fibonacci(in y));
}.
```

Παράγονται οι εξής τετράδες:

```
L1,begin_block, fibonacci, _, _
L2,<=, x, 1, L4
L3,jump, _, _, L6
L4,ret, _, _, x
L5,jump, _, _, L6
L6,-, x, 1, T_1
L7,par, T_1, cv, _
L8,par, T_2, ret, _
L9,call, fibonacci, _, _
L10,-, x, 2, T_4
L11,par, T_4, cv, _
L12,par, T_5, ret, _
L13,call, fibonacci, _, _
L14,+, T_2, T_5, T_3
L15,ret, _, _, T_3
L16,end_block, fibonacci, _, _
L17,begin_block, fibonacci, _, _
L18,in, y, _, _
L19,par, y, cv, _
L20,par, T_6, ret, _
L21,call, fibonacci, _, _
L22,out, T_6, _, _
L23,halt, _, _, _
L24,end_block, fibonacci, _, _
```

5 Πίνακας συμβόλων

Ο πίνακας συμβόλων είναι δυναμική δομή στην οποία αποθηκεύεται πληροφορία σχετιζόμενη με τα συμβολικά ονόματα που χρησιμοποιούνται στο υπό μεταγλώττιση πρόγραμμα.

Σε ένα πίνακα συμβόλων διατηρούμε πληροφορία για τα συμβολικά ονόματα που εμφανίζονται στο πρόγραμμα. Έτσι, σε έναν πίνακα συμβόλων αποθηκεύεται πληροφορία που σχετίζεται με τις μεταβλητές του προγράμματος, τις διαδικασίες και τις συναρτήσεις, τις παραμέτρους και με τα ονόματα των σταθερών. Για κάθε ένα από αυτά υπάρχει διαφορετική εγγραφή στον πίνακα και στην εγγραφή αυτή αποθηκεύεται διαφορετική πληροφορία, ανάλογα με το είδος του συμβολικού.

5.1 Βοηθητικές κλάσεις πίνακα συμβόλων

- Variable
  - name, το όνομα της μεταβλητής.
  - dataType, ο τύπος δεδομένου της μεταβλητής (πάντα ακέραιος).
  - offset, η απόσταση της μεταβλητής στην μνήμη από την αρχή του εγγραφήματος δραστηριοποίησης της τρέχουσας συνάρτησης.
- TemporaryVariable
  - name, το όνομα της προσωρινής μεταβλητής.
  - dataType, (πάντα ακέραιος).
  - offset, ομοίως με το Variable.
- FormalParameter
  - name, το όνομα της τυπικής παραμέτρου.
  - dataType, (πάντα ακέραιος).
  - mode, ο τρόπος περάσματος της τυπικής παραμέτρου [cv | ref].
- Parameter
  - name, το όνομα της πραγματικής παραμέτρου.
  - dataType, (πάντα ακέραιος).
  - mode, ο τρόπος περάσματος της πραγματικής παραμέτρου [cv | ref | ret].
  - offset, η απόσταση της παραμέτρου στην μνήμη από την αρχή του εγγραφήματος δραστηριοποίησης της τρέχουσας συνάρτησης.
- Procedure
  - name, το όνομα της διαδικασίας.
  - startinQuad, η ετικέτα της πρώτης τετράδας της διαδικασίας.
  - frameLength, το μήκος του εγγραφήματος δραστηριοποίησης της διαδικασίας.
  - formalParameters, η λίστα των τυπικών παραμέτρων της διαδικασίας.
- Function
  - name, το όνομα της συνάρτησης.
  - startinQuad, η ετικέτα της πρώτης τετράδας της συνάρτησης.
  - frameLength, το μήκος του εγγραφήματος δραστηριοποίησης της συνάρτησης.
  - dataType, ο τύπος δεδομένου που επιστρέφει η συνάρτηση (πάντα integer).
  - formalParameters, η λίστα των τυπικών παραμέτρων της συνάρτησης.

Για την προσθήκη αντικειμένων στις λίστες τυπικών παραμέτρων υλοποιήθηκε η βοηθητική μέθοδος addFormalParameter(), η οποία δέχεται τα πεδία απαραίτητα για την κατασκευή ενός αντικειμένου FormalParameter, κατασκευάζει το αντικείμενο και το προσθέτει στη λίστα,

Συνεχίζοντας επειδή η ετικέτα της αρχικής τετράδας ενός υπό-προγράμματος δεν είναι άμεσα γνωστή κατά την δημιουργία του, υλοποιήθηκε η μέθοδος updateStartingQuad() η οποία δέχεται ως όρισμα μια ετικέτα τετράδας.

Τέλος η updateFrameLength() μετά το πέρας της ανάπτυξης του συντακτικού δένδρου του υπό-προγράμματος θέτει το μήκος του εγγραφήματος δραστηριοποίησης του. Η μέθοδος αυτή μας χρειάστηκε επειδή το μήκος του εγγραφήματος δραστηριοποίησης είναι γνωστό μόνο αφού παραχθεί και η τελευταία τετράδα που αντιστοιχεί στην μέθοδο.

5.2 Σύνθεση του πίνακα συμβόλων

Ο πίνακας συμβόλων αποτελείται από επίπεδα τα οποία ας ονομάσουμε scope. Ο όρος σημαίνει εμβέλεια και κάθε τέτοιο επίπεδο αντιστοιχεί στη μετάφραση μίας συνάρτησης. Όταν ξεκινάει η μετάφραση μίας συνάρτησης, τότε δημιουργείται ένα νέο επίπεδο (scope). Όταν τερματίζεται η μετάφραση μίας συνάρτησης, τότε αφαιρείται το επίπεδο της από τον πίνακα συμβόλων. Σημειώνεται ότι σε αυτήν ακριβώς την στιγμή κρατάμε στην μνήμη και ένα

στιγμιότυπο του πίνακα συμβόλων ακριβώς πριν αφαιρέσουμε το επίπεδο από την δομή. Θα δούμε αργότερα για ποιο λόγο το κάνουμε αυτό :).

5.3 Η κλάση Scope

Η κλάση Scope υλοποιήθηκε με σκοπό να εξυπηρετήσει τις λειτουργίες τις οποίες περιγράψαμε στο σημείο 5.2. Θεωρήθηκε σκόπιμο ότι μια δομή με αυτήν την αρκετά αυξημένη λειτουργικότητα θα ήταν χρήσιμο να μοντελοποιηθεί σε μία ανεξάρτητη κλάση. Η κλάση αυτή παρέχει την εξής λειτουργικότητα:

- addProcedure(), προσθήκη μιας νέας διαδικασίας στο επίπεδο που αναπτύσσεται αυτή τη στιγμή.
- addFunction(), προσθήκη μιας νέας συνάρτησης στο επίπεδο που αναπτύσσεται αυτή τη στιγμή.
- addVariable(), προσθήκη μιας νέας μεταβλητής στο επίπεδο που αναπτύσσεται αυτή τη στιγμή.
- addTempVariable(), προσθήκη μιας νέας προσωρινής μεταβλητής στο επίπεδο που αναπτύσσεται αυτή τη στιγμή.
- addFormalParameter(), προσθήκη μιας νέας τυπικής παραμέτρου στην λίστα τυπικών παραμέτρων στο υπό-πρόγραμμα που μόλις ξεκίνησε να αναπτύσσεται.
- addParameter(), προσθήκη μιας νέας πραγματικής παραμέτρου στο εγγράφημα δραστηριοποίησης του υπό-ανάπτυξη υπό-προγράμματος.
- updateStartingQuad(), θέτει την ετικέτα της αρχικής τετράδας που αντιστοιχεί στο τελευταίο υπό-πρόγραμμα που αναπτύχθηκε.
- updateFrameLength(), θέτει το μήκος του εγγραφήματος δραστηριοποίησης του τελευταίου υπό-προγράμματος που αναπτύχθηκε.
- searchByName(), πραγματοποιεί αναζήτηση στις οντότητες που βρίσκονται αποθηκευμένες μέσα στην κλάση Scope με βάση το όνομα τους. Αν δεν βρει την οντότητα που ζητήσαμε να αναζητήσει η συνάρτηση επιστρέφει None.

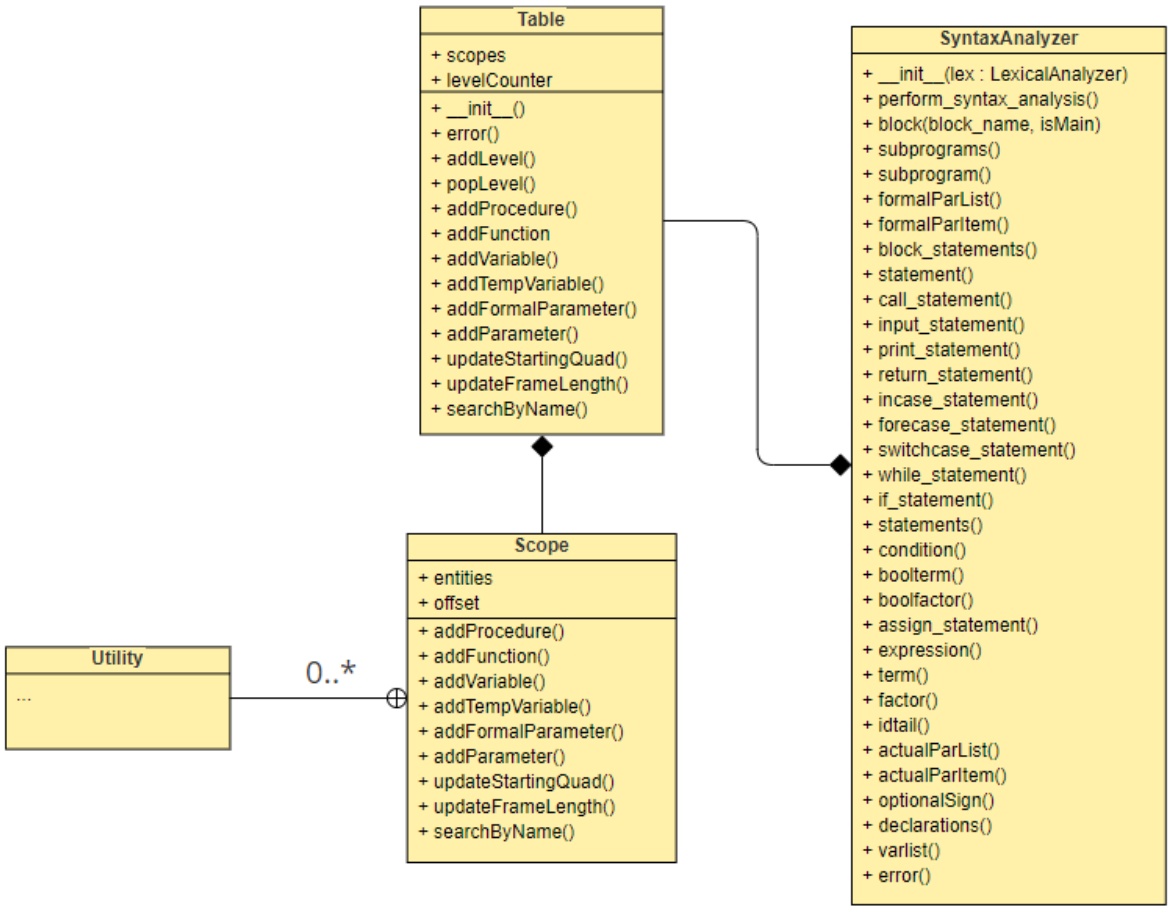
5.4 Η κλάση Table

Η κλάση Table αποτελεί το τελευταίο επίπεδο μοντελοποίησης του πίνακα συμβόλων. Στην πραγματικότητα είναι ο πίνακας συμβόλων. Αποθηκεύει εσωτερικά του μια στοίβα από scopes και έναν μετρητή ο οποίος μας δείχνει σε ποιο scope βρισκόμαστε αυτή τη στιγμή. Επιπλέον δίνεται η δυνατότητα να αλληλοεπιδράσει ο κώδικας του συντακτικού αναλυτή, ο οποίος παράγει τις οντότητες των scopes, με το scope που αναπτύσσεται αυτή τη στιγμή. Η λειτουργικότητα αυτή επιτυγχάνεται χάρη στις εξής συναρτήσεις:

- addLevel(), προσθήκη ενός νέου επιπέδου στην στοίβα.
- popLevel(), αφαίρεση του πιο πάνω επιπέδου από την στοίβα.
- addProcedure(), καλεί την addProcedure() του τελευταίου scope που προστέθηκε.
- addFunction(), καλεί την addFunction() του τελευταίου scope που προστέθηκε.
- addVariable(), καλεί την addVariable() του τελευταίου scope που προστέθηκε.
- addParameter(), καλεί την addParameter() του τελευταίου scope που προστέθηκε.
- addFormalParameter(), καλεί την addFormalParameter() του τελευταίου scope που προστέθηκε.
- addTempVariable(), καλεί την addTempVariable() του τελευταίου scope που προστέθηκε.
- updateStartingQuad(), καλεί την updateStartingQuad() του τελευταίου scope που προστέθηκε με σκοπό και εκείνο με την σειρά του να καλέσει την updateStartingQuad() του υπό-προγράμματος που μόλις αναπτύχθηκε.

- updateFrameLength(), καλεί την updateFrameLength () του τελευταίου scope που προστέθηκε με σκοπό και εκείνο με την σειρά του να καλέσει την updateFrameLength () του υπό-προγράμματος που μόλις αναπτύχθηκε.
- searchByName(), πραγματοποιεί αναζήτηση από το ανώτερο επίπεδο της στοίβας προς τα κάτω και καλεί για κάθε επίπεδο την searchByName() του αντίστοιχου επιπέδου. Αν η οντότητα που ζητήθηκε κατά την κλήση της μεθόδου δεν βρεθεί, προχωράει στο επόμενο επίπεδο. Αν η οντότητα δεν υπάρχει σε κανένα επίπεδο τότε καλείται η συνάρτηση error. Η συνάρτηση error θα μας ενημερώσει ότι πιθανώς δεν έχουμε ορίσει κάτι πριν το χρησιμοποιήσουμε και θα τερματίσει την μεταγλώττιση του προγράμματος με ένα μήνυμα σφάλματος.

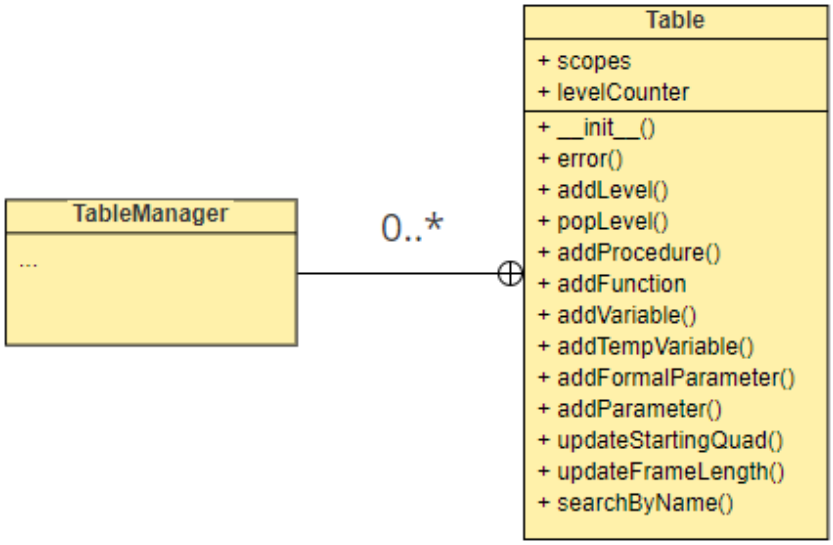
5.5 Διάγραμμα κλάσεων πίνακα συμβόλων



Εικόνα 7

5.6 Διαχειριστής στιγμιότυπων πίνακα συμβόλων

Η κλάση TableManager διαχειρίζεται τα διάφορα στιγμιότυπα του πίνακα συμβόλων αποθηκεύοντας τα σε μία λίστα. Θα δούμε αργότερα στην φάση του τελικού κώδικα που μας χρησιμεύει αυτή η λειτουργία :).



Εικόνα 8

5.7 Παράδειγμα σύνθεσης πίνακα συμβόλων

Δοθέντος του παρακάτω προγράμματος σε Cimple:

```
program fibonacci
{
    declare y;
    function fibonacci(in x)
    {
        if(x <= 1)
            return (x);;
        return (fibonacci(in x-1)+fibonacci(in x-2));
    }
    # main #
    input(y);
    print(fibonacci(in y));
}.
```

Κατασκευάζουμε τον παρακάτω πίνακα συμβόλων:

```
1 : (None|None|None)x/12 cv T_1/16 T_2/20 T_3/24 T_4/28 T_5/32
0 : (None|None|None)y/12 fibonacci/36 ['x cv'](1)
################################################################################
0 : (None|None|None)y/12 fibonacci/36 ['x cv'](1) T_6/16
################################################################################
```

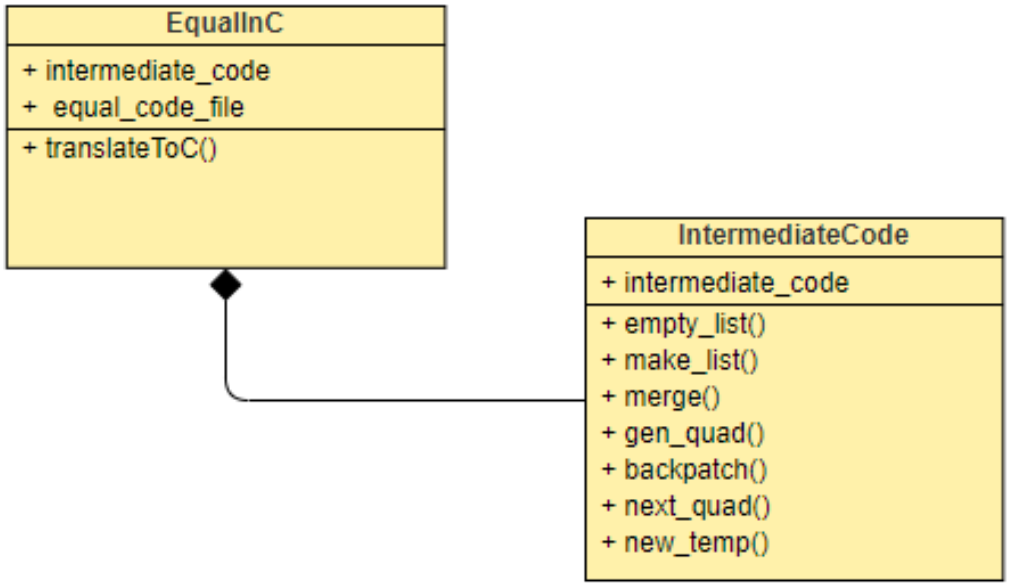
6 Μετατροπή ενδιάμεσου κώδικα σε γλώσσα C

Για την μετατροπή του ενδιάμεσου κώδικα σε γλώσσα προγραμματισμού C υλοποιήσαμε την κλάση EqualInC. Η κλάση αυτή διατηρεί μία λίστα με τις τετράδες που προέκυψαν από την ανάλυση του αρχικού προγράμματος σε ενδιάμεσο κώδικα. Στην συνέχεια μπορούμε να καλέσουμε την ρουτίνα translateToC(), για να μετατρέψουμε τις τετράδες σε γλώσσα C. Αρχικά δημιουργούμε το αρχείο a.c στο οποίο κάνουμε include την βιβλιοθήκη <stdio.h>. Το συγκεκριμένο header file είναι απαραίτητο για να δέχεται το πρόγραμμα είσοδο από το πληκτρολόγιο και να μπορεί να εκτυπώσει δεδομένα στην κονσόλα.

Η γενική δομή της μεθόδου translateToC() είναι μια ακολουθία από if-else if προκειμένου να αντιστοιχηθούν οι operators του ενδιάμεσου κώδικα με τις αντίστοιχες εντολές τους σε C. Π.χ. η εντολή halt μετατρέπεται σε return 0; κα η εντολή out μετατρέπεται στην printf(). Ιδιαίτερη προσοχή χρειάστηκε την χρήση των μεταβλητών αφού οι μεταβλητές δεν ορίζονται κάπου στον ενδιάμεσο κώδικα αλλά βλέπουμε κατευθείαν την χρήση τους. Για να λύσουμε αυτό το πρόβλημα προσθέσαμε μία λίστα η οποία διαχειρίζεται τις μεταβλητές που έχουν οριστεί. Αν μία μεταβλητή δεν έχει οριστεί τότε γράφεται η εντολή “int var_name;” πριν την χρήση της μεταβλητής και την προσθέτουμε στην λίστα των μεταβλητών που έχουμε ορίσει.

Σημειώνεται ότι αυτή η λειτουργία του μεταφραστή δεν λειτουργεί κανονικά όταν το υπό μεταγλώττιση πρόγραμμα σε Cimple διαθέτει υπό-προγράμματα, δηλαδή συναρτήσεις και διαδικασίες.

6.1 Διάγραμμα κλάσεων της κλάσης EqualInC



Εικόνα 9

6.2 Παράδειγμα εκτέλεσης μετατροπής προγράμματος σε ισοδύναμο σε C

Δοθέντος του παρακάτω προγράμματος σε Cimple:

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

Το αποτέλεσμα του ισοδύναμου αρχείου σε C είναι:

```
#include <stdio.h>
int main()
{
    L1: ; /* countDigits */
    L2: ;
 int x;
    scanf("%d", &x);
    L3: ;
 int count = 0;
    L4: if(x > 0)    { goto L6; }
    L5: goto L11;
    L6: ;
    int T_1 = x / 10;
    L7: x = T_1;
    L8: ;
    int T_2 = count + 1;
    L9: count = T_2;
    L10: goto L4;
    L11: printf("%d\n", count);
    L12: return 0;
    L13: ; /* countDigits */
}
```

7 Τελικός κώδικας σε assembly RISC-V

Η τελευταία φάση της παραγωγής κώδικα είναι η παραγωγή του τελικού κώδικα σε RISC-V assembly. Ο τελικός κώδικας προκύπτει με την βοήθεια του πίνακα συμβόλων. Συγκεκριμένα, από κάθε εντολή ενδιάμεσης αναπαράστασης παράγεται μία σειρά εντολών τελικού κώδικα, αυτό συμβαίνει γιατί κάποιες εντολές είναι πιο σύνθετες στην αναπαράσταση τους σε assembly από ότι στην ενδιάμεση αναπαράσταση. Επιπλέον για να παράξουμε την σωστή ακολουθία των εντολών αυτών χρειάζεται να ανακτήσουμε πληροφορία από τον πίνακα συμβόλων.

7.1 Βοηθητικές συναρτήσεις

Προκειμένου να διαχειριστούμε συχνές συμπεριφορές απαραίτητες για την ορθή λειτουργία του μεταγλωττιστή υλοποιήσαμε τέσσερεις βοηθητικές συναρτήσεις.

- gnlvcode(), δημιουργεί μια ακολουθία εντολών assembly για την προσπέλαση πληροφορίας που βρίσκεται αποθηκευμένη στο εγγράφημα δραστηριοποίησης κάποιου προγόνου της συνάρτησης ή της διαδικασίας που αυτή τη στιγμή μεταφράζεται.

- produce(), δέχεται μια αλφαριθμητική συμβολοσειρά ως όρισμα και ένα index (μη υποχρεωτικό). Η συμβολοσειρά αντιστοιχεί σε μία εντολή assembly, π.χ. “li t5 4” ενώ το index είναι ένας ακέραιος. Η χρήση του index προέκυψε επειδή σε ορισμένες περιπτώσεις έπρεπε να συμπληρώσουμε εντολές assembly σε προηγούμενες γραμμές από αυτήν που δημιουργούσαμε εκείνη την στιγμή. Ένα τέτοιο παράδειγμα είναι η αύξηση του καταχωρητή frame pointer πριν γίνει το πέρασμα των παραμέτρων του υπό-προγράμματος που πρόκειται να γίνει κλήση. Όταν βρίσκαμε για ποιο υπό-πρόγραμμα πρόκειται να γίνει η κλήση τοποθετούσαμε στην αρχή του πρώτου περάσματος παραμέτρου την εντολή “addi fp, sp, frameLength”, με frameLength να είναι το μήκος του εγγραφήματος δραστηριοποίησης του υπό-προγράμματος.

- loadvr(), δημιουργεί εντολές assembly με σκοπό την ανάγνωση της τιμής μιας μεταβλητής από την μνήμη και την μεταφορά αυτής της τιμής σε έναν καταχωρητή για να μπορέσομε να την χρησιμοποιήσουμε.

- storerv(), δημιουργεί εντολές assembly με σκοπό την αποθήκευση της τιμής μιας μεταβλητής στην μνήμη. Η τιμή αυτή θα βρίσκεται σε κάποιον καταχωρητή τον οποίο θα δώσουμε ως όρισμα, όπως και στην loadvr().

7.2 Η μέθοδος translateToAssembly

Η συγκεκριμένη μέθοδος μετατρέπει μια προς μία τις τετράδες του ενδιάμεσου κώδικα με την βοήθεια των στιγμιότυπων του πίνακα συμβόλων σε εντολές assembly. Δεν είναι κάτι πιο σύνθετο παρά μια επαναληπτική διαδικασία while για κάθε τετράδα, με μια ακολουθία if – else if εσωτερικά της.

Κατά την ολοκλήρωση της μετάφρασης ενός υπό-προγράμματος κάνουμε pop το τελευταίο στιγμιότυπο του πίνακα συμβόλων από την δομή που τα αποθηκεύει προκειμένου να προχωρήσουμε στο επόμενο. Σημειώνεται ότι η ολοκλήρωση της μετάφρασης ενός υπό-προγράμματος πραγματοποιείται όταν ο αλγόριθμος συναντάει την τετράδα “end_block, name, _, _”.

Για την μετατροπή των πράξεων (λογικών και αριθμητικών) σε εντολές assembly χρησιμοποιούμε δυο διαφορετικά λεξικά για να γίνει η αντιστοίχιση τους.

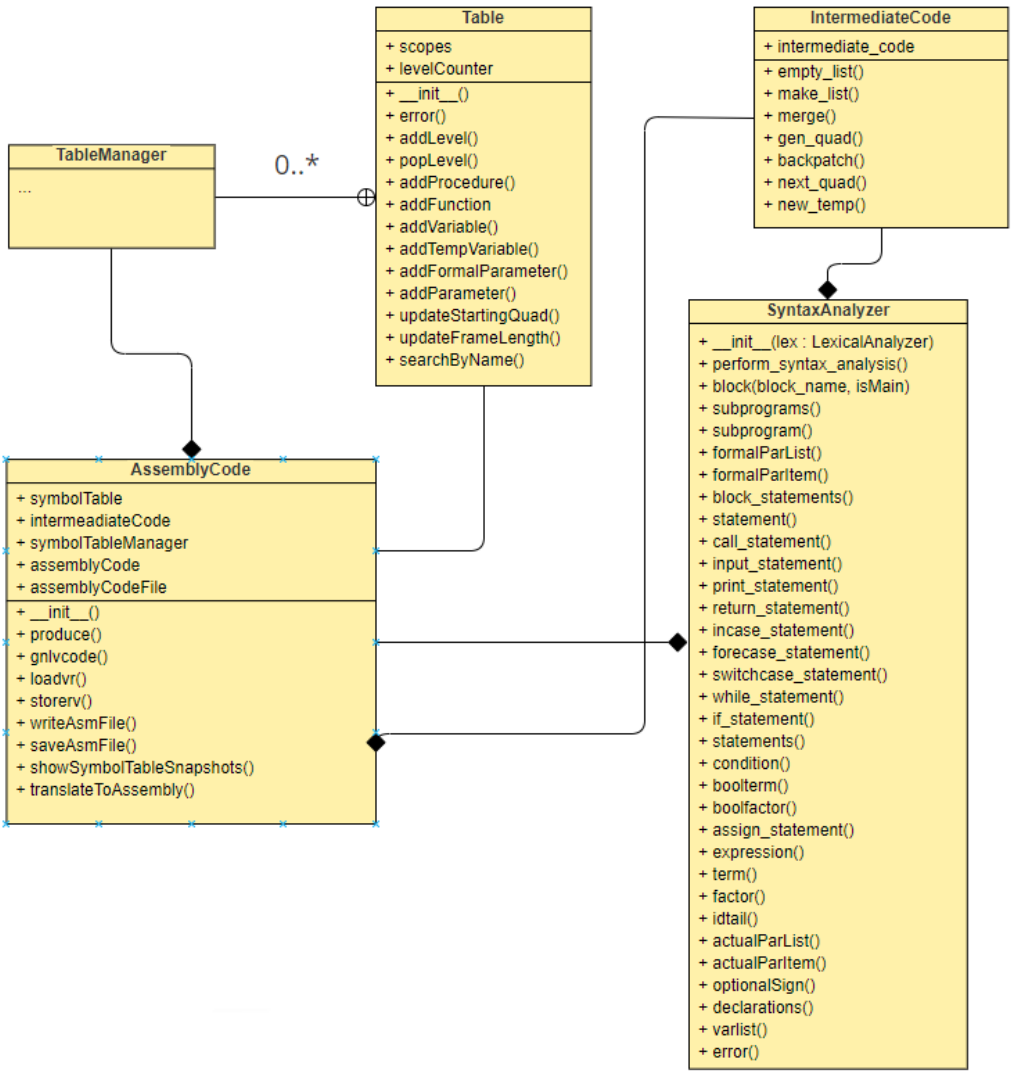
```
operatorToOperationDecoder = {'+': 'add', '-': 'sub', '*': 'mul', '/': 'div'}
relOperatorToOperationDecoder = {'=': 'beq', '<>': 'bne', '<': 'blt', '>': 'bgt', '<=': 'ble', '>=': 'bge'}
```

Αξίζει να σημειωθούν τα εξής, υπάρχουν περιπτώσεις όπου ένα υπό-πρόγραμμα χρειάζεται να ανακτήσει πληροφορία από το εγγράφημα δραστηριοποίησης κάποιου προγόνου του. Σε αυτή την περίπτωση κάνουμε χρήση του συνδέσμου προσπέλασης του τρέχοντος εγγραφήματος δραστηριοποίησης. Κατά την κλήση ενός υπό-προγράμματος ο σύνδεσμος προσπέλασης αντιστοιχίζεται στον σύνδεσμο προσπέλασης του καλώντος υπό-προγράμματος αν τα δύο υπό-προγράμματα βρίσκονται στο ίδιο βάθος φωλιάσματος. Διαφορετικό αν δεν έχουν το ίδιο βάθος φωλιάσματος τότε απλά τοποθετούμε στον σύνδεσμο προσπέλασης τον δείκτη μνήμης στοίβας του καλώντος υπό-προγράμματος (stack pointer, sp).

Συνεχίζοντας, ένας ακόμα καταχωρητής που οφείλουμε να συντηρήσουμε είναι ο ra (return address). Στον καταχωρητή αυτόν αποθηκεύεται η διεύθυνση στην οποία πρέπει να επιστρέψει ο έλεγχος του προγράμματος όταν ολοκληρωθεί η εκτέλεση μιας συνάρτησης ή διαδικασίας. Η επιστροφή γίνεται με χρήση της εντολής “jr ra”. Για να συντηρήσουμε αυτόν τον καταχωρητή, στην αρχή κάθε συνάρτησης αποθηκεύουμε στην πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της την οποία έχει τοποθετήσει στον ra η εντολή jal (jump and link). Στο τέλος κάθε υπό-προγράμματος κάνουμε το αντίστροφο. Παίρνουμε από την πρώτη θέση του εγγραφήματος δραστηριοποίησης του υπό-προγράμματος την διεύθυνση επιστροφής και την τοποθετούμε πάλι πίσω στον ra.

Τέλος το κυρίως πρόγραμμα δεν είναι το πρώτο block κώδικα που μεταφράζεται. Έτσι απαιτείται στην αρχή της λίστας εντολών assembly να τοποθετήσουμε ένα jump στην ετικέτα που αντιστοιχεί στην αρχική τετράδα του κυρίως προγράμματος. Έτσι βλέπουμε μία ακόμα χρήση του index που αναλύσαμε προηγουμένως στην βοηθητική μέθοδο produce().

7.3 Διάγραμμα κλάσεων παραγωγής τελικού κώδικα



Εικόνα 10

7.4 Παράδειγμα λειτουργείας ανάπτυξης τελικού κώδικα

Δοσμένου του παρακάτω προγράμματος σε γλώσσα Cimple:

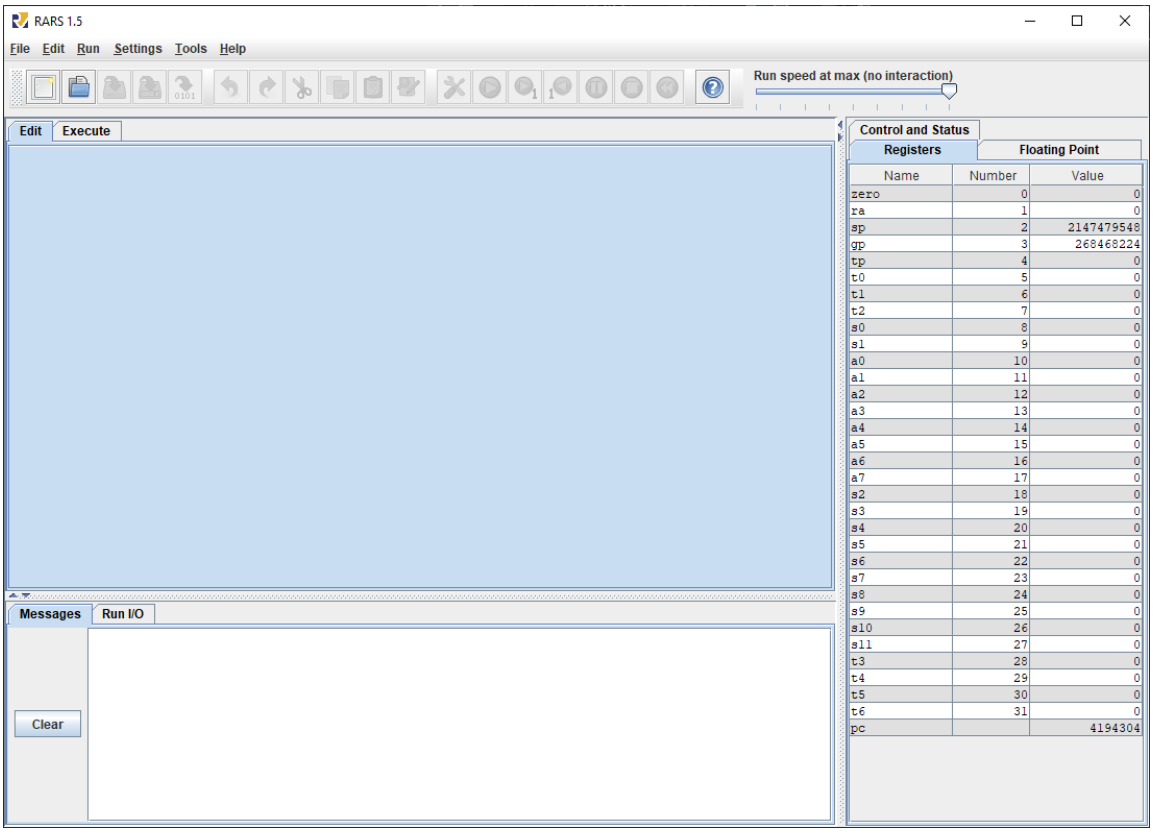
```
program forLoop
{
    declare i;

    i := 0;
    forcase
        case (i < 10)
        {
            i := i + 1;
            print(i)
        }
        default
            print(-1);

}.
```

Προκύπτει ο παρακάτω κώδικας σε assembly αρχιτεκτονικής RISC-V:

```
    .data
str_nl: .asciz "\n"
    .text
L0:
    j L1 # main
L1:
    # begin_block forLoop
    addi sp, sp, 24
    mv gp, sp
L2:
    li t1, 0
    sw t1, -12(gp)
L3:
    lw t1, -12(gp)
    li t2, 10
    blt t1, t2, L5
L4:
    j L9
L5:
    lw t1, -12(gp)
    li t2, 1
    add t1, t1, t2
    sw t1, -16(gp)
L6:
    lw t1, -16(gp)
    sw t1, -12(gp)
L7:
    lw a0, -12(gp)
    li a7, 1
    ecall
    la a0, str_nl
    li a7, 4
    ecall
```

```
L8:
    j L3
L9:
    li t1, 0
    li t2, 1
    sub t1, t1, t2
    sw t1, -20(gp)
L10:
    lw a0, -20(gp)
    li a7, 1
    ecall
    la a0, str_nl
    li a7, 4
    ecall
L11:
    li a0, 0
    li a7, 93
    ecall
L12:
    # end block forLoop
```

Το πρόγραμμα που χρησιμοποιήσαμε για να εκτελέσουμε την assembly που παράχθηκε είναι ο προσομοιωτής RISC-V RARS.



Εικόνα 11