Σχεδίαση και Υλοποίηση μιας Συναρτησιακής Γλώσσας Προγραμματισμού

Κωνσταντίνος Γεωργιάδης

Πτυχιακή εργασία

Τ.Ε.Ι. Πελοποννήσου
Σχολή Τεχνολογικών Εφαρμογών
Τμήμα Μηχανικών Πληροφορικής Τ.Ε.

Σπάρτη
2014

Περιεχόμενα

1 Εισαγωγή **1**

1.1 Η γλώσσα προγραμματισμού tiger 1
1.2 Αρχιτεκτονική ενός μεταγλωττιστή 3
1.3 Σκοπός της εργασίας 5
1.4 Επισκόπηση της εργασίας 8

2 Η γλώσσα προγραμματισμού tiger **9**

2.1 Σχόλια 10
2.2 Βασικοί τύποι 10
2.3 Αριθμητικές πράξεις 10
2.4 Δηλώσεις μεταβλητών 11
2.5 Συναρτήσεις 12
2.5.1 Δήλωση Συναρτήσεων 12
2.5.2 Αναδρομικές Συναρτήσεις 12
2.5.3 Ένθετες Συναρτήσεις 13
2.6 Τύποι 13
2.6.1 Δηλώσεις τύπων 13
2.6.2 Αναδρομικοί τύποι 14
2.7 Εκφράσεις 15
2.8 Σκοπός 16
2.9 Εκτύπωση λεκτικών, συντακτικών και σημασιολογικών λαθών 17
2.10 Ορισμός της γλώσσας tiger 19
2.10.1 Λεκτικά χαρακτηριστικά 19
2.10.2 Δηλώσεις 19
2.10.3 Τύποι δεδομένων 19
2.10.4 Μεταβλητές 20
2.10.5 Συναρτήσεις 21
2.10.6 Κανόνες Σκοπού 21
2.10.7 Μεταβλητές και εκφράσεις 22
2.10.8 Εκφράσεις 22

3 Λεκτική Ανάλυση **27**
3.1 Η λεκτική ανάλυση ενός προγράμματος 27
3.2 Ο λεκτικός αναλυτής της tiger 28
3.2.1 Ο παραγωγός λεκτικού αναλυτή ml-lex 28
3.2.2 Η υλοποίηση του παραγωγού λεκτικού αναλυτή της tiger 29
3.3 Παράδειγμα 34

4 Συντακτική Ανάλυση **37**
4.1 Η συντακτική ανάλυση ενός προγράμματος 37
4.2 Συντακτικός αναλυτής της tiger 41
4.2.1 Ο παραγωγός συντακτικού αναλυτή ml-yacc 41
4.2.2 Η υλοποίηση του παραγωγού συντακτικού αναλυτή της tiger . . . 42
4.3 Παράδειγμα 43

5 Σημασιολογική ανάλυση **45**
5.1 Ο σημασιολογική ανάλυση ενός προγράμματος 45
5.2 Ο σημασιολογικός αναλυτής της tiger 45
5.2.1 Περιβάλλον και τεχνικές σημασιολογικής ανάλυσης 46
5.2.2 Τύποι δεδομένων 49
5.3 Στατικός έλεγχος vs Δυναμικό έλεγχο 52
5.4 Παραγωγή ενδιάμεσου κώδικα 54

6 Οργάνωση τρέχοντος προγράμματος και παραγωγή κώδικα **59**
6.1 Παραγωγή συμβολικού κώδικα ARM από ενδιάμεσο κώδικα. 62
6.1.1 Στοίβα κλήσεων 62
6.1.2 Maximal Munch 65
6.1.3 Βελτιστοποίηση ενδιάμεσου κώδικα. 68
6.2 Κατανομή καταχωρητών 69

Παραρτήματα **77**
Α Κώδικας λεκτικής ανάλυσης 77
Β Κώδικας συντακτικής ανάλυσης 79
Γ Κώδικας παραγωγής συμβολικού κώδικα ARM 85

Βιβλιογραφία **90**

1. Εισαγωγή

Οι γλώσσες προγραμματισμού είναι το μέσο που έχουν οι προγραμματιστές για να δίνουν εντολές σε μια μηχανή. Η ανάπτυξη μεταγλωττιστών για γλώσσες προγραμματισμού ήταν από τα πρώτα προγράμματα που φτιάχτηκαν. Ο πρώτος μεταγλωττιστής χρονολογείται από το 1954 (FORTRAN). Η ανάπτυξη ενός μεταγλωττιστή θεωρείται δύσκολη διαδικασία και ήταν πάντα πρόκληση για τους προγραμματιστές. Ακόμα και αν κάποιος προγραμματιστής δεν ασχοληθεί ποτέ επαγγελματικά με την ανάπτυξη μεταγλωττιστών θα έχει να αποκομίσει πολλή γνώση υλοποιώντας έστω και έναν απλό μεταγλωττιστή. Μην ξεχνάμε ότι στην καριέρα του ο προγραμματιστής θα χρησιμοποιήσει πολλές γλώσσες προγραμματισμού έτσι έχοντας μια ιδέα πως είναι υλοποιημένες οι γλώσσες θα είναι σε θέση να χρησιμοποιήσει τις γλώσσες πιο αποδοτικά.

Η παρούσα εργασία ασχολείται με την ανάπτυξη ενός μεταγλωττιστή για τη γλώσσα προγραμματισμού tiger. Πιο συγκεκριμένα, υλοποιούμε όλες τις "κλασσικές" φάσεις ενός μεταγλωττιστή, γράφοντας κώδικα σε ml-lex, ml-yacc και SML. Στο εισαγωγικό αυτό κεφάλαιο παρουσιάζεται μια γενική εικόνα της γλώσσας tiger και των μεταγλωττιστών γενικότερα, και στο τέλος δίνεται μία σύνοψη του αντικειμένου της εργασίας.

1.1 Η γλώσσα προγραμματισμού tiger

Η γλώσσα προγραμματισμού tiger δημιουργήθηκε από τον Andrew W. Appel. Η γλώσσα προγραμματισμού tiger είναι απλή αλλά δεν υπολείπεται κάποια βασική δομή ή έκφραση. Συνοπτικά υποστηρίζει τις παρακάτω λειτουργίες.

- Δηλώσεις μεταβλητών.
- Δηλώσεις καινούργιων τύπων.

- Εκφράσεις ροής ελέγχου επιλογής if – then – else και βρόχων while και for.
- Συναρτήσεις και ένθετες συναρτήσεις (χαρακτηριστικό συναρτησιακών γλωσσών).
- Δημιουργία πινάκων οποιουδήποτε τύπου.
- Δημιουργία εγγραφών οποιουδήποτε τύπου.

Η επιλογή της γλώσσας προγραμματισμού tiger. Η επιλογή της tiger έγινε με βάση το ότι είναι πιο ισχυρή από τις τυπικές διδακτικές γλώσσες προγραμματισμού και περιέχει πολλά συναρτησιακά χαρακτηριστικά. Για παράδειγμα, ένα πρόγραμμα γραμμένο στη γλώσσα tiger είναι μια έκφραση που περιέχει άλλες εκφράσεις. Οι συναρτησιακές γλώσσες δεν είναι και τόσο διάσημες στη βιομηχανία λογισμικού έτσι η tiger είναι μια καλή εισαγωγή στις βασικές αρχές των συναρτησιακών γλωσσών.

Συναρτησιακές γλώσσες προγραμματισμού. Οι συναρτησιακές γλώσσες προγραμματισμού είναι υποκατηγορία του δηλωτικού υποδείγματος. Στον δηλωτικό προγραμματισμό ένα πρόγραμμα φτιάχνεται χρησιμοποιώντας εκφράσεις. Πιο συγκεκριμένα, λύνουμε ένα πρόβλημα απλοποιώντας το και όταν καταλήξουμε στη πιο απλή του μορφή τότε το πρόβλημα έχει λυθεί. Αντίθετα, στο προστακτικό υπόδειγμα ο προγραμματιστής λύνει ένα πρόβλημα χρησιμοποιώντας εντολές, δηλαδή αλλάζει την κατάσταση του προγράμματος. Αυτή η αλλαγή ονομάζεται παρενέργεια (side-effect). Παρακάτω βλέπουμε τι σημαίνει ότι μια συνάρτηση έχει παρενέργεια μέσω μιας συνάρτησης η οποία προσθέτει δύο αριθμούς.

Πρόγραμμα 1.1: Η συνάρτηση add1

```
integer right = 8

integer add1(integer left)
            left + righ

integer result = add(5)
```

Πρόγραμμα 1.2: Η συνάρτηση add2

```
integer add2(integer left, integer right)
    left + right

integer result = add(5,8)
```

Στο πρώτο παράδειγμα η συνάρτηση add1 έχει παρενέργεια γιατί το αποτέλεσμά της συνάρτησης εξαρτάται από την μεταβλητή right. Δεν μπορούμε να πούμε ότι η add1 θα επιστρέφει πάντα το ίδιο αποτέλεσμα παρατηρώντας απλά την συνάρτηση γιατί το αποτέλεσμα της εξαρτάται από την μεταβλητή right. Αν η right αλλάξει, αλλάζει και το αποτέλεσμα. Σε ένα μεγάλο πρόγραμμα μπορεί να υπάρχουν πολλές τέτοιες μεταβλητές σε

διαφορετικά πηγαία αρχεία, οι οποίες καθιστούν το αποτέλεσμα μιας συνάρτησης απρόβλεπτο.

Σε αντίθεση με την add1, για τη συνάρτηση add2 μπορούμε με βεβαιότητα να πούμε ποιο θα είναι το αποτέλεσμα της, γιατί το αποτελέσμα προκύπτει από τον υπολογισμό του αθροίσματος της τιμής της μεταβλητής left και right. Αυτό είναι πλεονέκτημα σε ένα μεγάλο πρόγραμμα, γιατί όταν αποδειχτεί ότι μία συνάρτηση είναι ορθή τότε αυτό το κομμάτι κώδικα (αυτή η συνάρτηση) δεν μπορεί να προκαλέσει λογικό σφάλμα στο πρόγραμμα. Μπορούμε απλά και μόνο με την παρατήρηση να επιβεβαιώσουμε ότι είναι ορθή.

Οι συναρτησιακές γλώσσες μας δίνουν τα εργαλεία που μας βοηθάνε να φτιάχνουμε προγράμματα χωρίς παρενέργειες. Αν και το παράδειγμά μας είναι απλό, περιγράφει την γενική ιδέα του συναρτησιακού προγραμματισμού.

1.2 Αρχιτεκτονική ενός μεταγλωττιστή

Με απλά λόγια ένας μεταγλωττιστής είναι ένα πρόγραμμα το οποίο κάνει μετάφραση. Ξέρει τουλάχιστον δύο γλώσσες. Μεταφράζει από την γλώσσα Α, με την οποία γράφει και καταλαβαίνει ο προγραμματιστής, στην γλώσσα Γ που καταλαβαίνει ο υπολογιστής. Ένας μεταγλωττιστής μπορεί να γραφτεί σε οποιαδήποτε γλώσσα Β. Η γλώσσα Β συνήθως δεν έχει καμία σχέση με την καινούργια γλώσσα που θα γράψει ο προγραμματιστής (Α) ή την γλώσσα που καταλαβαίνει ο υπολογιστής (Γ).

Ένας μεταγλωττιστής ως πρώτη αφαίρεση χωρίζεται σε δύο μέρη: το εμπρόσθιο (front-end) μέρος και το οπίσθιο (back-end) μέρος. Το εμπρόσθιο μέρος είναι υπεύθυνο για την αναγνώριση και έλεγχο της γλώσσας Α, δηλαδή επιβάλλει τον ορισμό της γλώσσας Α. Περιλαμβάνει τις ακόλουθες φάσεις-αναλύσεις:

- λεκτική
- συντακτική
- σημασιολογική

Το οπίσθιο μέρος είναι υπεύθυνο για την μετάφραση της ενδιάμεσης γλώσσας σε γλώσσα Γ. Αναλυτικότερα περιλαμβάνει τις ακόλουθες φάσεις:

- ανάλυση και βελτιστοποίηση της ενδιάμεσης γλώσσας
- παραγωγή της γλώσσας Γ.

Η ενδιάμεση γλώσσα παράγεται από το εμπρόσθιο μέρος και αναλύεται από το οπίσθιο μέρος. Η ενδιάμεση γλώσσα είναι ο συνδετικός κρίκος μεταξύ του εμπρόσθιου και

οπίσθιου μέρους. Οι εμπορικοί μεταγλωττιστές μπορεί να περιλαμβάνουν πολλές ενδιάμεσες γλώσσες, δηλαδή μια ενδιάμεση γλώσσα μπορεί να παράγει άλλη ενδιάμεση γλώσσα. Αυτό γίνεται για να αντιμετωπιστεί η δυσκολία μετάφρασης υψηλού επιπέδου γλωσσών σε γρήγορο κώδικα μηχανής[8]. Σε διαφορετικές ενδιάμεσες γλώσσες μπορούν να γίνουν διαφορετικές βελτιστοποιήσεις.

Από τεχνικής και θεωρητικής άποψης το εμπρόσθιο και οπίσθιο μέρος είναι πολύ διαφορετικά στην υλοποίηση. Οι γνώσεις που απαιτούνται για την υλοποίηση του εμπρόσθιου μέρους είναι:

- οι αλγόριθμοι της λεκτικής και της συντακτικής ανάλυσης.
- θεωρία γλωσσών ώστε να αναπτυχθούν προχωρημένες προγραμματιστικές τεχνικές όπως παραμετρικός πολυμορφισμός, πρώτης κλάσης συναρτήσεις, χειρισμός εξαιρέσεων. Η υλοποίηση αυτών των τεχνικών γίνεται στην σημασιολογική ανάλυση.

Οι γνώσεις που απαιτούνται για την υλοποίηση του οπίσθιου μέρους χρειάζεται είναι:

- αποδοτικοί αλγόριθμοι και τεχνικές για την βελτιστοποίηση του ενδιάμεσου κώδικα ως προς την ταχύτητα εκτέλεσης αλλά και ως προς τον αποθηκευτικό χώρο στο τελικό πρόγραμμα.
- λεπτομέρειες για κάθε αρχιτεκτονική ΚΜΕ (Κεντρικής Μονάδας Επεξεργασίας). Ενδεικτικά μερικές αρχιτεκτονικές ΚΜΕ είναι οι intel x86, intel x86-64, MIPS, ARM, SPARC.

Όλα τα παραπάνω αφορούν μεταγλωττιστές που παράγουν κώδικα και φτιάχνουν εκτελέσιμο αρχείο. Υπάρχει άλλη μία κατηγορία μεταγλωττιστών που αντί να παράγει εκτελέσιμα αρχεία, εκτελεί τον πηγαίο κώδικα την ώρα που τρέχει ο μεταγλωττιστής. Οι μεταγλωττιστές που ανήκουν σε αυτή την κατηγορία ονομάζονται ερμηνευτές. Η υλοποίηση της γλώσσας tiger γίνεται με μεταγλώττιση.

Για να φτιαχτεί ένα εκτελέσιμο αρχείο χρειάζεται μετά την μετάφραση να μεταφραστούν τα μνημονικά (που έχει παράγει ο μεταγλωττιστής) στις αντίστοιχες αριθμητικές εκφράσεις (object file). Αυτό το πρόγραμμα λέγεται συμβολομεταφραστής (assembler) και καλείται έμμεσα από τους μεταγλωττιστές. Επιπρόσθετα, ένα object file χρειάζεται να συνδεθεί με τυχόν εξαρτημένες βιβλιοθήκες και να οριστεί μια αφετηρία για την έναρξη του εκτελέσιμου αρχείου. Το πρόγραμμα που πραγματοποιεί αυτή τη λειτουργία λέγεται συνδέτης (linker) και καλείται έμμεσα από τους μεταφραστές μετά το τέλος της συμβολομετάφρασης.

Συγκεκριμένα η tiger μετά την μετάφραση του πηγαίου κώδικα καλεί τον gnu assembler για να παραχθεί ένα object file και στη συνέχεια καλεί τον gnu linker ο οποίος συνδέει το object file μαζί με τις καθιερωμένες βιβλιοθήκες που χρησιμοποιεί η tiger.

1.3 Σκοπός της εργασίας

Η παρούσα εργασία υλοποιεί την γλώσσα προγραμματισμού tiger. Αν και η tiger δημιουργήθηκε για εκπαιδευτικό σκοπό η παρούσα υλοποίηση χρησιμοποιεί τεχνικές και αναλύσεις οι οποίες συναντώνται σε εμπορικούς μεταγλωττιστές. Υλοποιούμε σε κώδικα ml-lex, ml-yacc και SML όλες τις "κλασσικές" φάσεις ενός μεταγλωττιστή. Πιο συγκεκριμένα, υλοποιούμε σε κώδικα:

- ml-lex (παράγεται κώδικας SML όταν μεταγλωττιστεί) τα λεκτικά χαρακτηριστικά όπως αυτά δίνονται στον ορισμό της tiger.
- ml-yacc (παράγεται κώδικας SML όταν μεταγλωττιστεί) τα συντακτικά χαρακτηριστικά όπως αυτά δίνονται στον ορισμό της tiger.
- SML την εύρεση μεταβλητών που χρησιμοποιούνται σε ένθετες συναρτήσεις. Εδώ ψάχνουμε για μεταβλητές οι οποίες χρησιμοποιούνται σε συναρτήσεις άλλες από αυτές που δηλώθηκαν και μαρκάρονται ως διαφυγούσες (escaped). Όσες μεταβλητές χρησιμοποιήθηκαν σε άλλες συναρτήσεις έχουν διαφορετικό χειρισμό στη παραγωγή κώδικα.
- SML τα σημασιολογικά χαρακτηριστικά όπως αυτά ορίζονται από την tiger.
- SML την παραγωγή ενδιάμεσης γλώσσας από το αφηρημένο συντακτικό δένδρο. Αυτή είναι κοντά στις γλώσσες μηχανής.
- SML την κανονικοποίηση του ενδιαμέσου κώδικα. Στην κανονικοποίηση ξαναχτίζεται η ενδιάμεση γλώσσα στη πιο απλή της μορφή.
- SML την επιλογή συμβολικού κώδικα αρχιτεκτονικής ARM από την αντίστοιχη ενδιάμεση γλώσσα.
- SML τις αναλύσεις για βελτιστοποίησες κώδικα. Πιο αναλυτικά, γράφεται κώδικας για την υλοποίηση του γράφου ροής ελέγχου ο οποίος χρησιμοποιείται στην ανάλυση ροής δεδομένων και συγκεκριμένα στην ανάλυση ενεργών μεταβλητών. Η ανάλυση αυτή χρησιμοποιείται για την κατασκευή του γράφου παρεμβολών μεταξύ των μεταβλητών που πρέπει να τοποθετηθούν σε καταχωρητές.
- SML την καταχώριση μεταβλητών (register allocation) με χρωματισμό γραφήματος πάνω στο γράφο παρεμβολών.
- C την υλοποίηση υπηρεσιών για το πρόγραμμα όπως είναι για παράδειγμα η δέσμευση δυναμικής μνήμης. Επίσης υλοποιήθηκαν σε C οι βασικές συναρτήσεις της tiger.

Για την υλοποίηση των πιο πάνω γράφτηκαν πολλοί βοηθητικοί τύποι δεδομένων και υπηρεσίες σε κώδικα SML όπως τύπος για αφηρημένο συντακτικό δένδρο, τύποι για μεταβλητές, μεταφραστής αλφαριθμητικών σε σύμβολα, δένδρο ενδιάμεσου κώδικα, πίνακας συμβόλων, γράφοι και εικονικοί καταχωρητές.

Στο Σχήμα 1.1 βλέπουμε σχηματικά την βασική ροή των αρχείων της tiger. Αρχικά μεταγλωττίζονται τα αρχεία tiger.lex και tiger.grm με τα αντίστοιχα προγράμματα. Τα παραγόμενα αρχεία αποτελούν τον λεκτικό και τον συντακτικό αναλυτή και συνδυάζονται μέσα στο αρχείο parse.sml. Το parse.sml ουσιαστικά κάνει την συντακτική ανάλυση και επιστρέφει ένα αφηρημένο συντακτικό δένδρο. Το αφηρημένο συντακτικό δένδρο δίνεται ως είσοδος στο αρχείο findescape.sml το οποίο ψάχνει και μαρκάρει όλες τις ένθετες μεταβλητές. Στη φάση αυτή δεν επιστρέφεται κάποια τιμή. Στη συνέχεια, το δένδρο δίνεται ως είσοδος στο αρχείο semant.sml, το οποίο αποτελεί τον σημασιολογικό αναλυτή. Πραγματοποιείται σημασιολογική ανάλυση και παράγεται ένα δένδρο ενδιάμεσου κώδικα. Το νέο δένδρο που παράγεται δίνεται ως είσοδος στο αρχείο canon.sml το οποίο κανονικοποιεί το δένδρο παράγοντας μία απλούστερη μορφή. Το κανονικοποιημένο δένδρο δίνεται ως είσοδος στο αρχείο armgen.sml το οποίο αντικαθιστά τις εκφράσεις του δένδρου με συμβολικό κώδικα. Ο παραγόμενος συμβολικός κώδικας χρησιμοποιεί εικονικούς καταχωρητές. Οι εικονικοί καταχωρητές δίνονται ως είσοδος στο αρχείο regalloc.sml το οποίο καλεί με την σειρά του τα αρχεία color.sml, makegraph.sml, liveness.sml τα οποία κάνουν αναλύσεις για αποδοτική κατανομή καταχωρητών. Ο συμβολικός κώδικας μαζί με τους επιλεγμένους καταχωρητές περνιούνται στο αρχείο assem.sml το οποίο αντικαθιστά του κανονικούς καταχωρητές με τους εικονικούς και παράγει το μεταγλωττισμένο αρχείο out.s. Ο μεταγλωττιστής μετά καλεί τον συμβολομεταφραστή ο οποίος συμβολομεταφράζει το αρχείο σε γλώσσα μηχανής. Μετά ο μεταγλωττιστής καλεί το συνδετή ο οποίος συνδέει αρχείο out.o με το runtime.o το οποίο περιέχει συναρτήσεις της καθιερωμένης βιβλιοθήκης και συναρτήσεις για δέσμευση δυναμικής μνήμης.

Κατά την εκτέλεση των πιο πάνω αρχείων εκτελούνται και τα παρακάτω βοηθητικά αρχεία:

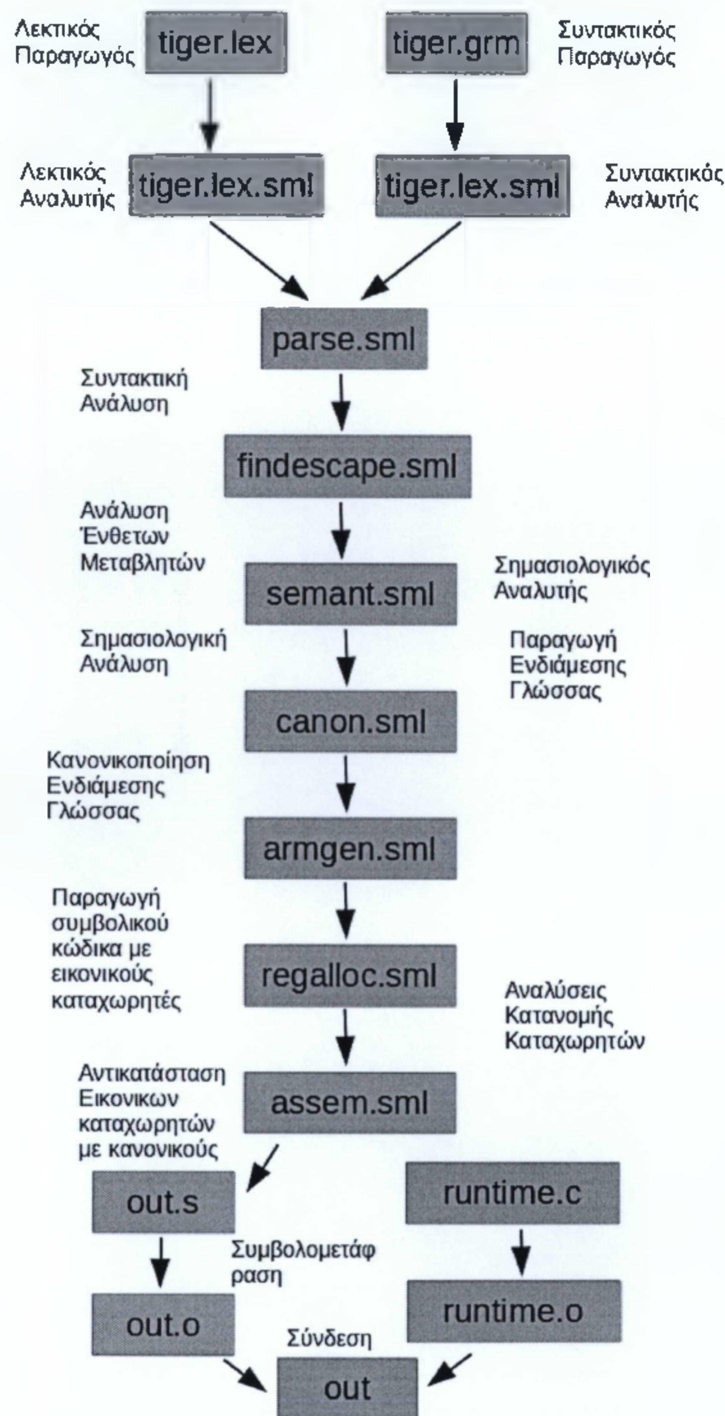
- main.sml, που αποτελεί το αρχείο στο οποίο βρίσκεται η έναρξη του μεταγλωττιστή και εκτελεί όλα τα πιο πάνω αρχεία.

- errormsg.sml, που χρησιμοποιείται για την αποθήκευση των σημείων και γραμμών των token και για την εκτύπωση λαθών.

- absyn.sml, που περιέχει την δομή δεδομένων του αφηρημένου συντακτικού δένδρου.

- frame.sml και armframe.sml, που περιέχουν το πρότυπο και την υλοποίηση λεπτομερειών που αφορούν την εκάστοτε αρχιτεκτονική.

- env.sig και env.sml, που περιέχουν την υλοποίηση του περιβάλλοντος μεταβλητών,



Σχήμα 1.1: Η βασική ροή των αρχείων της tiger.

συναρτήσεων και τύπων. Όλες οι καθιερωμένες συναρτήσεις και τύποι είναι προκαθορισμένοι εδώ.

- graph.sig και graph.sml που περιέχουν μία γενική υλοποίηση γράφου.
- prabsyn.sml και printtree.sml, που αποτελούν τα αρχεία τα οποία εκτυπώνουν τις δομές των δένδρων.
- symbol.sml, που χαρτογραφεί συμβολοσειρές με σύμβολα, καθώς τα σύμβολα είναι πολύ πιο γρήγορα στο χειρισμό τους.

Ο κώδικας όλων των προαναφερθέντων αρχείων δίνεται στο Παράρτημα Α.

1.4 Επισκόπηση της εργασίας

Στο Κεφάλαιο 2 δίνεται μια αναλυτική περιγραφή της γλώσσας tiger.

Στο Κεφάλαιο 3 παρουσιάζεται η υλοποίηση της λεκτικής ανάλυσης για τη γλώσσα tiger. Πιο συγκεκριμένα, παρουσιάζεται ο λεκτικός αναλυτής που χρησιμοποιήθηκε και παρουσιάζονται οι γενικές αρχές αρχές και οι αλγόριθμοι της λεκτική ανάλυσης γενικότερα.

Στο Κεφάλαιο 4 παρουσιάζεται ο συντακτικός αναλυτής που χρησιμοποιήθηκε και παρουσιάζονται οι γενικές αρχές και αλγόριθμοι της συντακτικής ανάλυσης γενικότερα.

Στο Κεφάλαιο 5 παρουσιάζεται η σημασιολογική ανάλυση. Η σημασιολογική ανάλυση έχει μεγάλο ενδιαφέρον γιατί εκεί γίνονται οι έλεγχοι τύπων, οι έλεγχοι για τις αδήλωτες μεταβλητές, για το ποιες μεταβλητές είναι σε σκοπό σε κάποιο σημείο του προγράμματος. Γενικά ελέγχεται οτιδήποτε είναι συντακτικά σωστό αλλά “δεν βγάζει νόημα” για την γλώσσα.

Στο Κεφάλαιο 6 παρουσιάζεται συνοπτικά πως ένα τρέχον πρόγραμμα οργανώνεται στη μνήμη του υπολογιστή. Επίσης εξηγείται πως μεταφράζεται ο ενδιάμεσος κώδικας σε γλώσσα μηχανής. Ακόμα δίνονται βασικές βελτιστοποιήσεις κώδικα τις οποίες υλοποιεί και η tiger.

2.

Η γλώσσα προγραμματισμού tiger

Το κεφάλαιο αυτό αποτελεί μια εισαγωγή στη γλώσσα tiger. Η γλώσσα tiger είναι μία απλή γλώσσα με συναρτησιακό προσανατολισμό. Η tiger υποστηρίζει όλα τα κλασσικά δομικά στοιχεία μιας γλώσσας προγραμματισμού για την ανάπτυξη προγραμμάτων. Δεν παρουσιάζουμε μια εξονυχιστική περιγραφή της γλώσσας αλλά μια γενική περιγραφή, η οποία περιλαμβάνει τις παρακάτω ενότητες:

- Σχόλια
- Βασικοί τύποι
- Αριθμητικές πράξεις
- Δηλώσεις μεταβλητών
- Δηλώσεις τύπων
- Δηλώσεις συναρτήσεων
- Εκφράσεις
- Σκοπός

Ο αναγνώστης που δεν έχει εμπειρία με τον προγραμματισμό διαβάζοντας τις πιο πάνω ενότητες θα μάθει την σύνταξη της tiger και με λίγη εξάσκηση και πειραματισμό θα είναι σε θέση να γράφει απλά προγράμματα. Η tiger είναι πολύ καλή γλώσσα για την εκμάθηση βασικών αρχών προγραμματισμού γιατί είναι απλή και παραλείπει προχωρημένα στοιχεία που συναντώνται σε πιο ισχυρές εμπορικές γλώσσες. Ένας έμπειρος προγραμματιστής μπορεί να παραλείψει τις εισαγωγικές αυτές ενότητες και να διαβάσει απευθείας τον αναλυτικό ορισμό της γλώσσας που βρίσκεται στο τέλος αυτού του κεφαλαίου.

2.1 Σχόλια

Ότι βρίσκεται ανάμεσα στα σύμβολα /* και */ θεωρείται ως σχόλιο και δεν λαμβάνεται ως μέρος του προγράμματος. Τα σχόλια είναι χρήσιμα για τον σχολιασμό του κώδικα ενός προγράμματος.

2.2 Βασικοί τύποι

Η tiger υποστηρίζει δύο αρχικούς τύπους. Τον ακέραιο int και τον αλφαριθμητικό string.

Ακέραιοι είναι όλες οι αριθμητικές τιμές. Οι αριθμητικοί τελεστές (πρόσθεση, αφαίρεση, πολλαπλασιασμός και διαίρεση) λειτουργούν πάνω στον ακέραιο τύπο.

Αλφαριθμητικά είναι οτιδήποτε περιλαμβάνεται σε διπλά εισαγωγικά. Οι αριθμητικοί τελεστές δεν λειτουργούν πάνω στα αλφαριθμητικά.

2.3 Αριθμητικές πράξεις

Ο βασικός τύπος int υποστηρίζει πράξεις με όλους τους αριθμητικούς τελεστές.

```
7 + 3
9 * 2 + 1 * 6
9 - 3
8 / 2
```

Οι αριθμητικές πράξεις έχουν την συνηθισμένη προτεραιότητα, δηλαδή, προτεραιότητα έχουν οι τελεστές πολλαπλασιασμού και διαίρεσης, ακολουθούμενοι από την πρόσθεση και την αφαίρεση. Δηλαδή η πιο κάτω έκφραση:

```
7 + 2 * 2
```

είναι ισοδύναμη με την:

```
7 + (2 * 2)
```

Ακόμα η προσεταιριστικότητα των πράξεων είναι από αριστερά στα δεξιά. Δηλαδή η πιο κάτω έκφραση:

```
7 - 4 + 2
```

είναι ισοδύναμη με την:

```
(7 - 4) + 2
```

2.4 Δηλώσεις μεταβλητών

Η παρακάτω έκφραση:

```
let
var a := 3
in
a
end
```

δηλώνει και ορίζει μια μεταβλητή a η οποία έχει τύπο ακεραίου. Όλες οι δηλώσεις μεταβλητών γίνονται μέσα στο περιβάλλον `let ... in`. Στην παρακάτω έκφραση δηλώνουμε τέσσερις μεταβλητές.

```
let
var a := 3
var b := a
var c := 9
var d := "string"
in
a
end
```

Γενικά:

- Μπορούμε να δηλώσουμε όσες μεταβλητές θέλουμε. Η μεταβλητή αμέσως μετά την δήλωση μπορεί να χρησιμοποιηθεί. Στο παραπάνω παράδειγμα, η μεταβλητή a χρησιμοποιείται ως η τιμή για τον ορισμό της μεταβλητής b.
- Ο τύπος της μεταβλητής δεν χρειάζεται να γράφεται ρητά. Η γλώσσα αντιλαμβάνεται τι τύπο παίρνει μια μεταβλητή από το αποτέλεσμα του δεξιού μέρους.

Αν κάποιος το επιθυμεί, μπορεί να χρησιμοποιήσει τη μακρά δήλωσή τύπων και να δηλώσει ρητά, τι τύπο θα έχει η μεταβλητή. Το αποτέλεσμα του δεξιού μέρους πρέπει να συμφωνεί με τον τύπο που δόθηκε.

```
let
var a : int := 3
var b : string := "df"
in
a
end
```

Βλέπουμε ότι η μεταβλητή a και b παίρνει τιμή ακεραίου και τιμή αλφαριθμητικού αντίστοιχα. Αν για παράδειγμα είχαμε μια δήλωση μεταβλητής

```
var v : string := 4
```

τότε θα είχαμε σφάλμα αφού εκφράζεται ότι η τιμή τις v θα είναι αλφαριθμητικό ενώ καταχωρείται τιμή ακεραίου.

2.5 Συναρτήσεις

2.5.1 Δήλωση Συναρτήσεων

Οι δηλώσεις συναρτήσεων γίνονται με το εξής πρότυπο:

function <όνομα – συνάρτησης> (<παράμετροι>) : <επιστρεφόμενος τύπος>

Στο πιο κάτω πρόγραμμα βλέπουμε την δήλωση και κλήση μιας συνάρτησης add.

```
let
    function add(a: int, b: int) : int
        a+b
    var result := add(4,6)
in
    print(itoa(result))
end
```

Η συνάρτηση add παίρνει σαν παράμετρο δύο μεταβλητές τύπου ακεραίου και επιστρέφει μια τιμή τύπου ακεραίου. Στην τέταρτη γραμμή βλέπουμε πως γίνεται η κλήση της συνάρτησης add με δύο ορίσματα ακέραιας τιμής (δηλαδή ίδιου τύπου με τους τύπους που περιμένει η συνάρτηση) και επιστρέφει το αποτέλεσμα της πράξης στην μεταβλητή result. Στην έκτη γραμμή βλέπουμε ότι η μεταβλητή τύπου ακέραιου result, περνιέται στην συνάρτηση iota. Η συνάρτηση itoa δέχεται μία μεταβλητή τύπου ακεραίου και επιστρέφει την αντίστοιχη αναπαράσταση του σε αλφαριθμητικό. Με την σειρά της η αλφαριθμητική αναπαράσταση περνιέται ως παράμετρος στην συνάρτηση print[1] όπου τυπώνει το αλφαριθμητικό στην κονσόλα. Οι συναρτήσεις print και itoa ανήκουν στην καθιερωμένη βιβλιοθήκη της tiger και είναι προσβάσιμες σε όλα τα σημεία ενός προγράμματος tiger. Οι υπογραφές καθώς και οι επεξήγηση των καθιερωμένων βιβλιοθηκών περιγράφονται στον ορισμό της γλώσσας που βρίσκεται στο τέλος του κεφαλαίου.

2.5.2 Αναδρομικές Συναρτήσεις

Αναδρομικές συναρτήσεις είναι οι συναρτήσεις οι οποίες καλούν τον εαυτό τους. Ένα πρόβλημα μπορεί να λυθεί αναδρομικά απλοποιώντας το πρόβλημα σε κάθε αναδρομική κλήση. Για παράδειγμα το παρακάτω πρόγραμμα τυπώνει το αποτέλεσμα του 5 παραγοντικό $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$.

```
let
    function factorial(n: int)
        if n = 0
        then 1
        else n * factorial(n - 1)
```

[1] Η print παίρνει παράμετρο τύπου ακεραίου.

```
in
    factorial(5)
end
```

2.5.3 Ένθετες Συναρτήσεις

Ένθετες συναρτήσεις είναι οι συναρτήσεις οι οποίες είναι δηλωμένες ανάμεσα σε άλλες συναρτήσεις.

```
let
    function example(n: int) : int
        let function add(a:int, b: int): int
                a + b
        in add(4, n) end
in example(9) end
```

Το παραπάνω πρόγραμμα παρουσιάζει την δομή μιας ένθετης συνάρτησης. Στην δεύτερη γραμμή βλέπουμε την δήλωση μιας συνάρτησης example η οποία παίρνει μία ακέραια τιμή ως παράμετρο. Μέσα στο σώμα της example στην τρίτη γραμμή βλέπουμε να δηλώνετε και μία άλλη συνάρτηση add. Η συνάρτηση add είναι ένθετη μέσα στη συνάρτηση example. Στην έκτη γραμμή καλείται η συνάρτηση example από το κυρίως πρόγραμμα και με την σειρά της η example καλεί την add.

2.6 Τύποι

2.6.1 Δηλώσεις τύπων

Οι εγγραφές είναι ο τρόπος με τον οποίο η tiger μας επιτρέπει να ομαδοποιούμε δεδομένα. Αυτή η ομαδοποίηση οδηγεί στη δημιουργία ενός καινούργιου τύπου. Σε άλλες γλώσσες προγραμματισμού οι εγγραφές ονομάζονται και δομές.

Για την δήλωση ενός τύπου χρησιμοποιούμε το παρακάτω πρότυπο:

type <όνομα τύπου> = <όνομα πεδίου> : <τύπος πεδίου>, ...

Για παράδειγμα, το παρακάτω πρόγραμμα αρχικά δηλώνει ένα καινούργιο τύπο student, ο οποίος έχει τρία πεδία, το όνομα, το επίθετο και τον αριθμό μητρώου.

```
let
    type student = { name : string,
                     surname : string,
                     am : int }

in
```

```
        student { name = "Giannis",
                  surnname = "Papadopoulos",
                  am = 2010087 }
    end
```

Στη συνέχεια ορίζεται μια τιμή τύπου student, η οποία δεν δεσμεύεται σε κάποια μεταβλητή. Τα πεδία μετά τον ορισμού μπορούν να αλλάξουν οι τιμές τους. Κάθε φορά που ορίζεται ένας τύπος τότε η tiger τον θεωρεί σαν ξεχωριστό τύπο, ακόμα και αν δύο τύποι έχουν ακριβός τα ίδια στοιχεία.

Για την πρόσβαση των στοιχείων χρησιμοποιούμε το όνομα της μεταβλητής ακολουθούμενη από τελεία και το όνομα του πεδίου:

<όνομα – μεταβλητής> . <όνομα πεδίου>

Για παράδειγμα για να τυπωθούν τα πεδία στην κονσόλα, γράφουμε το παρακάτω τροποποιημένο πρόγραμμα:

```
let
    type student = { name : string,
                     surname : string,
                     am : int }
    var john := student { name = "Giannis",
                          surnname = "papadopoulos",
                          am = 2010087 }
in
    (
       print(john.name);
       print"(\"n);
       print(john.papadopoulos);
       print"(\"n)
       print(itoa(john.am))
    )
end
```

2.6.2 Αναδρομικοί τύποι

Η γλώσσα υποστηρίζει αναδρομικούς τύπους, δηλαδή τύπους οι οποίοι κάνουν αναφορά στο εαυτό τους. Για παράδειγμα το παρακάτω πρόγραμμα υλοποιεί μια συνδεδεμένη λίστα.

```
let
    type list = { data : int, next : list }
    var l : list = nil
in
    ()
end
```

Το πρόγραμμα αυτό εισάγει μερικές καινούργιες έννοιες, όπως το nil το οποίο μπορεί να οριστεί μόνο σε εγγραφές. Η δήλωση της μεταβλητής πρέπει να χρησιμοποιεί την μεγάλη μορφή όταν πρόκειται για ορισμό του nil (αυτό γίνεται για να είναι σίγουρος ο μεταγλωττιστής ότι το nil χρησιμοποιήθηκε σε εγγραφή). Το nil είναι χρήσιμο για αμοιβαίους αναδρομικούς τύπους γιατί μπορεί να χρησιμοποιηθεί ως μέσο τερματισμού, όπως γίνεται στην περίπτωση μιας συνδεδεμένης λίστας.

2.7 Εκφράσεις

Τα πάντα στην tiger είναι εκφράσεις, ακόμα και ένα πρόγραμμα tiger είναι μια έκφραση. Όλα τα προγράμματα tiger ξεκινάνε με την έκφραση let . . in . . . end.

Έκφραση let <δηλώσεις> in <έκφραση> end. Η έκφραση let … in … end είναι η πιο βασική γιατί είναι και η μοναδική στην οποία μπορείς να δηλώσεις καινούργιες μεταβλητές, συναρτήσει και τύπους. Στο χώρο <δηλώσεις> γίνονται οι δηλώσεις και στο χώρο ανάμεσα στο in και end τοποθετείται μία έκφραση. Η έκφραση δεν μπορεί να είναι κενή.

Ακολουθία εκφράσεων (a; b; c; ...v;). Αν χρειάζεται να υπάρχουν περισσότερες από μία έκφραση σε ένα κομμάτι κώδικα τότε πρέπει η κάθε έκφραση να τερματίζεται με ελληνικό ερωτηματικό. Επίσης όλες οι εκφράσεις πρέπει να είναι ανάμεσα σε παρενθέσεις. Το αποτέλεσμα της τελευταίας έκφρασης είναι και η επιστρεφόμενη τιμή της ακολουθίας. Επίσης ο τύπος της επιστρεφόμενης τιμής ορίζει και τον τύπο της ακολουθίας.

Έκφραση καμίας τιμής (). Όταν υπάρχουν δύο παρενθέσεις χωρίς έκφραση ανάμεσα τους τότε δεν επιστρέφεται κάποια τιμή. Όπως θα δούμε παρακάτω, σε κάποιες εκφράσεις χρειάζεται να μην επιστρέφεται κάποια τιμή.

Έκφραση if<έκφραση1> then <έκφραση2> else <έκφραση3> και if <έκφραση1> then <έκφραση2>. Η tiger υποστηρίζει κανονικά τις εκφράσεις ροής ελέγχου επιλογής if – then - else ή if – then. Οι τύποι των επιστρεφόμενων τιμών στην if – then -else θα πρέπει να συμφωνούν. Η if – then έκφραση πρέπει να μην επιστρέφει κάποια τιμή.

```
let
    var a := 4
    var b := 3
in
    if a > b
    print(itoa(a))
    else print(itoa(b))
end
```

Έκφραση while <έκφραση1> do <έκφραση2>. Το while αποτιμά ως αληθείς όλες τις μη μηδενικές τιμές και ψευδές μόνο το 0. Όσο είναι αληθής η πρώτη έκφραση εκτελείται η δεύτερη έκφραση. Η while πρέπει να μην επιστρέφει κάποια τιμή. Για παράδειγμα το παρακάτω πρόγραμμα εκτυπώνει το μήνυμα "Hello World" δέκα φορές (η print δεν επιστρέφει τιμή).

```
let
    var a := 10
in
    while a do
        print"(Hello World\"n)
end
```

Η έκφραση for id := <έκφραση1> to <έκφραση2> do <έκφραση3>. Η τρίτη έκφραση εκτελείται όσες φορές είναι το εύρος ανάμεσα στην πρώτη και δεύτερη έκφραση. Το παρακάτω πρόγραμμα θα τυπώσει στην κονσόλα δέκα φορές το μήνυμα "Hello World". Το for είναι ειδική περίπτωση του while.

```
let
in
    for i := 0 to 10 do
        print"(Hello World\"n);
end
```

2.8 Σκοπός

Σκοπός ή εμβέλεια μιας μεταβλητής είναι σε ποιο σημείο ενός προγράμματος είναι ενεργή μία μεταβλητή. Η tiger χρησιμοποιεί λεκτικό σκοπό (λέγεται και στατικός σκοπός). Ο γενικός κανόνας του λεκτικού σκοπού είναι όταν μία μεταβλητή βρίσκεται πιο "ψηλά" από μία συνάρτηση τότε είναι προσβάσιμη στην συνάρτηση αυτή. Πιο συγκεκριμένα οι μεταβλητές που είναι σε σκοπό στη συνάρτηση είναι αυτές που ήταν δηλωμένες κατά την δήλωση της συνάρτησης και όχι κατά την κλήση της.

Στο πιο κάτω πρόγραμμα η μεταβλητή a είναι ενεργή μέσα στη συνάρτηση test λόγο του ότι είναι δηλωμένη πιο ψηλά.

```
let var a := 89
      function test(i: int) : int =
          i + a
in
    test(7)
end
```

Στο πρόγραμμα που ακολουθεί βλέπουμε ότι η μεταβλητή a που είναι δηλωμένη στο πιο ψηλό επίπεδο, είναι ενεργή σε όλα τα χαμηλότερα επίπεδα (test και test2). Η παράμετρος b της συνάρτησης test είναι ενεργή στο πιο χαμηλό επίπεδο test2 αλλά όχι στο πιο ψηλό που είναι η αρχική let . . in .. . end. Με την σειρά της η μεταβλητή c είναι ενεργή μόνο στη συνάρτηση test2.

```
let var a := 23
    function test(b: int) : int =
        let function test2(c: int) : int =
            c + b + a
        in test2(4) end
in test(5) end
```

2.9 Εκτύπωση λεκτικών, συντακτικών και σημασιολογικών λαθών

Ένας καλός συντακτικός αναλυτής πρέπει να παράγει χρήσιμα μηνύματα προς τον προγραμματιστή. Είναι ίσως το σημαντικότερο κομμάτι του εμπρόσθιου μέρους ενός μεταγλωττιστή αφού είναι ο μοναδικός τρόπος για να ενημερωθεί ο προγραμματιστής για τυχόν λάθη. Ο συντακτικός αναλυτής της tiger παράγει καλά μηνύματα. Μάλιστα έχει εισαχθεί και ξεχωριστό αρχείο κώδικα για την καλύτερη και περιεκτικότερη εκτύπωση μηνυμάτων. Συνοπτικά το αρχείο errormsg.sml παρέχει τις ακόλουθες υπηρεσίες:

- Έλεγχο για αν έχουν γίνει συντακτικά ή σημασιολογικά (αργότερα) λάθη.
- Εκτύπωση της θέσης και του μηνύματος στην κονσόλα.
- Επαναφορά των αρχικών τιμών.

Επίσης παρέχει τις ακόλουθες τιμές για εκχώρηση:

- Όνομα του αρχείου που αναλύεται, εκχωρείται από το αρχείο main.sml.
- Απαρίθμηση γραμμών, εκχωρείται από τον λεκτικό αναλυτή με κάθε εμφάνιση καινούργιας γραμμής.
- Λίστα θέσεων, εκχωρείται από τον λεκτικό αναλυτή. Κάθε φορά που γίνεται αλλαγή γραμμής προστίθεται στη λίστα η θέση πριν τον χαρακτήρα καινούργιας γραμμής(n). Αυτό είναι χρήσιμο για την εύρεση της θέσης ενός token από το αφηρημένο συντακτικό δένδρο στην σημασιολογική ανάλυση.

Στο παρακάτω παράδειγμα βλέπουμε μια εκτύπωση λεκτικού λάθους. Η tiger θα εκτυπώσει πως υπάρχει λεκτικό λάθος μόλις ο λεκτικός αναλυτής εντοπίσει έναν μη έγκυρο χαρακτήρα.

```
        let
            $ := 45
        in
        ()
        end
```

```
./tiger test.tig
test.tig:2.5:illegal character \$
```

Ο συντακτικός αναλυτής όταν εντοπίσει ένα συντακτικό λάθος προσπαθεί να το ξεπεράσει αντικαθιστώντας στο σημείο του λάθους διάφορα άλλα tokens. Αν τα καταφέρει συνεχίζει. Στην εκτύπωση του συντακτικού λάθους ο συντακτικός αναλυτής όχι μόνο εκτυπώνει το συντακτικό λάθος αλλά εκτυπώνει και με πιο token ο πηγαίος κώδικας μεταγλωττίζεται. Το διορθωμένο token δεν σημαίνει ότι είναι πάντα το σωστό αφού το πρόγραμμα μπορεί να τρέξει συμπτωματικά και με ένα άλλο token.

```
        let
            b := 0
        in
            for a := 0 to 9 do
                b := b + 1
        end
```

```
./tiger test.tig
test.tig:2.5:syntax error: inserting  VAR
```

Το inserting VAR μας υποδεικνύει πως στην δεύτερη γραμμή πρέπει να προστεθεί ένα token var.

Η εκτύπωση των σημασιολογικών λαθών γίνεται αν και μόνο η λεκτική και συντακτική ανάλυση έχει γίνει σωστά. Παρακάτω δίνουμε δυο ακόμα παραδείγματα.

```
        let
            function test(a:int,b:int) =
                a+b
        in
            test(3,"fg")
        end
```

```
./tiger test.tig
test.tig:5.5:expression type does not match of declaration
```

```
        let
            function test(a:int,b:int) : string =
                a+b
        in
            test(3,9)
        end
```

```
./tiger test.tig
test.tig:2.14:result type does not match expression type
```

2.10 Ορισμός της γλώσσας tiger

Η tiger είναι μια μικρή γλώσσα προγραμματισμού με ένθετες συναρτήσεις (functions), μεταβλητές εγγραφών(records) με άρρητους δείκτες, πίνακες, μεταβλητές ακεραίων και αλφαριθμητικών, και μερικές απλές δομημένες κατασκευές ελέγχου.

2.10.1 Λεκτικά χαρακτηριστικά

Αναγνωριστικά (Identifiers). Ένα αναγνωριστικό είναι μια σειρά από γράμματα, αριθμούς και το σύμβολο υπογράμμισης, η οποία ξεκινάει με γράμμα. Τα κεφαλαία γράμματα ξεχωρίζονται από τα πεζά. Σε αυτό το παράρτημα το σύμβολο id αντιπροσωπεύει ένα αναγνωριστικό (identifier).

Σχόλια (Comments). Ένα σχόλιο μπορεί να εμφανίζεται μεταξύ δύο tokens. Τα σχόλια ξεκινούν με /* και τερματίζουν με */ και μπορούν να είναι ένθετα.

2.10.2 Δηλώσεις

Μια σειρά δηλώσεων είναι μια σειρά τύπων(type, τιμής(value) και δηλώσεις συναρτήσεων. Κανένα σημείο στίξης δεν διαχωρίζει ή τερματίζει ξεχωριστές δηλώσεις.

```
decs  ->   {dec}

dec   ->   tydec
      ->   vardec
      ->   fundec
```

Στη συντακτική σημειογραφία που χρησιμοποιείται εδώ , το ε σημαίνει κενό αλφαριθμητικό και x σημαίνει για όποια πιθανή κενή περίπτωση των x. (ένα ή περισσότερα x)

2.10.3 Τύποι δεδομένων

Η σύνταξη των τύπων και δηλώσεις τύπων της Tiger είναι

```
tydec   ->   type type-id = ty

ty   ->   type-id
     ->   { tyfields } (Οι αγκύλες στο tyfields
```

```
                                έχουν νόημα στη γλώσσα)
        ->   array of type-id

    tyfields    ->   ε
                     ->   id : type-id {, id : type-id}
```

Ενσωματωμένοι τύποι (Built-in types): Οι τύποι int και string είναι προκαθορισμένοι. Πρόσθετοι τύποι μπορούν να οριστούν ή να ξανά-οριστούν (συμπεριλαμβανομένων και των τύπων που έχουν είδη οριστεί) με δηλώσεις τύπων. Εγγραφές (Records): Οι τύποι εγγραφών ορίζονται από μια λίστα πεδίων εσώκλειστοι μέσα σε αγκύλες, όπου το κάθε πεδίο περιγράφεται ως όνομα-πεδίου : όνομα-τύπου (fieldname : type-id), όπου το όνομα πεδίου είναι ένα αναγνωριστικό οριζόμενο από την δήλωση τύπου. Πίνακες (Arrays): Ένας πίνακας με οποιοδήποτε τύπο, μπορεί να δημιουργηθεί με array of type-id. Το μέγεθος του πίνακα δεν καθορίζεται ως κομμάτι του τύπου του. Κάθε πίνακας του κάθε τύπου μπορεί να έχει διαφορετικό μέγεθος, και το μέγεθος του μπορεί να αποφασιστεί στη δημιουργία του ή στο χρόνο εκτέλεσης (run time). Διάκριση εγγραφών (Record distinction): Κάθε δήλωση τύπου εγγραφής ή πίνακα δημιουργεί ένα καινούργιο τύπο, μη συμβατό από όλους τους άλλους τύπους εγγραφών ή πινάκων (ακόμα και αν έχουν όλα τα πεδία ίδια). Αμοιβαίοι αναδρομικοί τύποι (Matually recursive types): Μια συλλογή από τύπους μπορούν να είναι αναδρομικοί ή αμοιβαίοι αναδρομικοί. Οι αμοιβαίοι αναδρομικοί τύποι δηλώνονται με μια ακολουθία συνεχόμενων δηλώσεων τύπων χωρίς να παρεμβαίνουν οι δηλώσεις των μεταβλητών ή των συναρτήσεων. Κάθε φορά, η αναδρομή πρέπει να περάσει μέσο του τύπου της εγγραφής ή του πίνακα. Έτσι, ο τύπος μιας λίστας ακεραίων:

```
    type intlist = {hd: int, tl: intlist}

    type tree = {key: int, children: treelist}
    type treelist = {hd: tree, tl: treelist}
```

Αλλά η ακόλουθη σειρά δήλωσης είναι άκυρη:

```
type b = c
type c = b
```

Επαναχρησιμοποίηση ονόματος πεδίου (Field name reusability): Διαφορετικοί τύποι εγγραφών μπορούν να χρησιμοποιούν τα ίδια ονόματα στα πεδία τους. (Τέτοιο ώστε το πεδίο hd με intlist και treelist στο πιο πάνω παράδειγμα).

2.10.4 Μεταβλητές

```
    var    ->   var id := exp
           ->   var id : type-id := exp
```

Στην σύντομη μορφή της δήλωσης μεταβλητής, δίνεται το όνομα της μεταβλητής, ακολουθούμενη από μια έκφραση που αναπαριστά την αρχική τιμή της μεταβλητής. Σε

αυτή την περίπτωση, ο τύπος της μεταβλητής καθορίζεται από τύπο της έκφρασης. Στη μεγάλη μορφή, ο τύπος της μεταβλητής δίνεται και αυτός. Η έκφραση πρέπει να έχει τον ίδιο τύπο.

Αν η αρχικοποιημένη έκφραση είναι nil, τότε η μεγάλη μορφή πρέπει να χρησιμοποιηθεί. Κάθε δήλωση μεταβλητής δημιουργεί μια νέα μεταβλητή, η οποία διαρκεί όσο ο σκοπός (scope) της δήλωσης.

2.10.5 Συναρτήσεις

```
fundec ->   function id ( tyfields ) = exp
           ->   function id ( tyfields ) : type-id = exp
```

Το πρώτο από αυτά είναι η δήλωση ρουτίνας, το δεύτερο είναι η δήλωση συνάρτησης. Οι ρουτίνες δεν επιστρέφουν τιμές ενώ οι συναρτήσεις επιστρέφουν και ο τύπος τους καθορίζεται μετά την άνω κάτω τελεία. Το exp είναι το σώμα της ρουτίνας ή της συνάρτησης και το tyfields καθορίζει τα ονόματα και τους τύπους των παραμέτρων. Όλες οι παράμετροι περνιούνται κατά κλήση. Οι συναρτήσεις μπορούν να είναι αναδρομικές. Αμοιβαίες αναδρομικές συναρτήσεις και ρουτίνες δηλώνονται από μια σειρά ακολουθούμενων δηλώσεων συναρτήσεων (χωρίς παρέμβαση των δηλώσεων τύπων ή μεταβλητών):

```
function treeLeaves(t : tree) : int =
                      if t=nil then 1
                     else treelistLeaves(t.children)
function treelistLeaves(L : treelist) : int =
                     if L=nil then 0
                     else treeLeaves(L.hd) + treelistLeaves(L.tl)
```

2.10.6 Κανόνες Σκοπού

Τοπικές μεταβλητές (Local variables): Στην έκραση let ... vardec ... in exp end, ο σκοπός μιας δηλωμένης μεταβλητής ξεκινάει αμέσως μετά την vardec της και διαρκεί μέχρι το end. Παράμετροι (Parameters): Μέσα στο funtion id (.... id1 : id2 ...) = exp ο σκοπός της παραμέτρου id1 διαρκεί σε όλο το σώμα της exp. Ένθετοι σκοποί (Nested scopes): Ο σκοπός μιας μεταβλητής ή παραμέτρου περιλαμβάνει τα σώματα οποιονδήποτε ορισμών συναρτήσεων σε αυτό το σκοπό. Δηλαδή, η πρόσβαση στις μεταβλητές εξωτερικών σκοπών επιτρέπεται, όπως είναι είναι στην Pascal και τη Algol. Τύποι (Types): Στην έκφραση let ... tydecs ... in exps end ο σκοπός αναγνωριστικού τύπου ξεκινάει από την αρχή της ακολουθούμενης σειράς των δηλώσεων τύπων ορίζοντας το και διαρκεί μέχρι το end. Αυτό περιλαμβάνει τις κεφαλίδες και τα σώματα οποιονδήποτε συναρτήσεων μέσα στο σκοπό. Συναρτήσεις (Funtions): Στην έκφραση let ... fundecs ... in exps end ο σκοπός ενός αναγνωριστικού μιας συνάρτησης ξεκινάει από την αρχή μιας ακολουθούμενης σειράς των δηλώσεων των συναρτήσεων που τες ορίζουν και διαρκεί μέχρι το end. Αυτό περιλαμβάνει τις κεφαλίδες και τα σώματα οποιουδήποτε συναρτήσεων μέσα στο σκοπό. Χώρος ονομάτων (Name spaces): Υπάρχουν δύο διαφορετικοί χώροι ονομάτων: Ένας για τύπους,

και ένας για συναρτήσεις και μεταβλητές. Ο τύπος a μπορεί να είναι "μέσα στο σκοπό" ταυτόχρονα και σαν μεταβλητή a η μια συνάρτηση α, αλλά μεταβλητές και συναρτήσεις ιδίου ονόματος δεν μπορούν να υπάρχουν, και τα δύο μέσα στο σκοπό ταυτόχρονα (το ένα θα επικαλύπτει το άλλο). Τοπικές ξανά-δηλώσεις (Local redeclarations): Μια δήλωση μεταβλητής ή συνάρτησης μπορεί να επικαλύπτει από μια νέα δήλωση του ίδιου ονόματος (σαν μεταβλητή ή συνάρτηση) σε πιο μικρό σκοπό. Για παράδειγμα, αυτή η συνάρτηση τυπώνει "6 7 6 8 6" όταν εφαρμόζεται στο 5:

```
function f(v: int) =
let var v := 6
 in print(v);
      let var v := 7 in print (v) end;
      print (v);
      let var v := 8 in print (v) end;
      print (v)
  end
```

Συναρτήσεις μπορούν να επικαλύπτουν μεταβλητές με το ίδιο όνομα, και αντίθετα. Ομοίως, μία δήλωση τύπου μπορεί να έχει επικαλυφτεί από μία ξανά-δήλωση ιδίου ονόματος(σαν τύπος) σε πιο συγκεκριμένο σκοπό. Ωστόσο, δύο συναρτήσεις σε ακολουθία αμοιβαίων αναδρομικών συναρτήσεων δεν μπορούν να έχουν το ίδιο όνομα και δύο τύποι σε ακολουθία αμοιβαίων αναδρομικών τύπων επίσης δεν μπορούν να έχουν το ίδιο όνομα.

2.10.7 Μεταβλητές και εκφράσεις

L-VALUES Μία l-value είναι μια τοποθεσία όπου μια μεταβλητή μπορεί να διαβαστεί ή να ανατεθεί. Μεταβλητές, παράμετροι διαδικασιών, πεδία εγγραφών και στοιχεία πινάκων είναι όλα l-values.

```
Lvalue ->  id
       ->  lvalue   .   id
       ->  lvalue  [  exp  ]
```

Μεταβλητή (Variable): Η μορφή id αναφέρετε σε μια μεταβλητή ή παράμετρο προσβάσιμη από τους κανόνες του σκοπού. Πεδίο εγγραφής (Record field): Το σύμβολο τελεία επιτρέπει την επιλογή μιας τιμής της εγγραφής με το αντίστοιχο όνομα του πεδίου. Δείκτης πίνακα (Array subcript): Το σύμβολο αγκύλη επιτρέπει την αντίστοιχη επιλογή ενός αριθμημένου κελιού σε ένα πίνακα. Οι πίνακες κατατάσσονται απο μιά σειρά συνοχόνενων ακαιρέων ξεκινώντας απο το μηδέν (μέχρι το μεγεθος του πίνακα μείον ένα).

2.10.8 Εκφράσεις

l-value: Μια l-value, όταν χρησιμοποιείται σαν μια έκφραση, αξιολογείτε στα περιεχόμενα της αντίστοιχης τοποθεσίας. Εκφράσεις χωρίς τιμές (Valueless expressions):

Ορισμένες εκφράσεις δεν παράγουν τιμή: κλήση διαδικασίας, εκχωρήσεις, if-then, while, break, και μερικές φορές το if-then-else. Επομένως η έκφραση (a := b) + c είναι συντακτικά σωστή αλλά αποτυγχάνει στον έλεγχο τύπου (type-check).

nil: Η έκφραση nil (δεσμευμένη λέξη) υποδηλώνει μια τιμή nil που ανοίκει σε κάθε τύπο εγγραφής. Αν μια μεταβλητή εγγραφής v περιέχει την τιμή nil, είναι σφάλμα ελέγχου την ώρα που τρέχει το πρόγραμμα, να επηλέξεις ένα πεδίο απο το v. Το nil πρέπει να χρησιμοποιείται σε συμφραζόμενα όπου ο τύπος μπορεί να καθοριστεί, δηλαδή:

```
var a : my_record := nil    OK
a := nil        OK
if a <> nil then ...        OK
if nil <> a then ...        OK
if a = nil then ...         OK
function f(p : my_record) = ...  f(nil)  OK
var a := nil A
if nil = nil then ...       Άκυρο
```

Αλληλουχία (Sequencing): Μια ακολουθία δύο ή περισσότερων εκφράσεων, περιλαμβανόμενές από παρενθέσεις και διαχωριζόμενες από ερωτηματικά (exp; exp; ... exp) αξιολογεί όλες τις εκφράσεις με σειρά. Το αποτέλεσμα μιας ακολουθίας είναι το αποτέλεσμα(αν υπάρχει) που παράχθηκε από την τελευταία των εκφράσεων.

Καμία τιμή (No value): Μία ανοιχτή παρένθεση ακολουθούμενη από μια κλειστή παρένθεση (δύο διαφορετικά tokens) είναι μια έκφραση που παράγει καμία τιμή. Ομοίως, μια let έκφραση με τίποτα ανάμεσα στο in και end παράγει καμία τιμή.

Ακέραιος (Integer literal): Μια ακολουθία από ακέραιους αριθμούς είναι μια ακέραια σταθερά η οποία υποδηλώνει την αντίστοιχη ακέραια τιμή.

Αλφαριθμιτικό (String literal): Μια αλφαριθμητική σταθερά είναι μια ακολουθία, ανάμεσα σε (αγγλικά) εισαγωγικά (»), από μηδέν ή περισσότερους εκτυπώσιμους χαρακτήρες, κενά, ή ακολουθίες διαφυγής. Κάθε ακολουθία διαφυγής εισάγεται με ένα χαρακτήρα διαφυγής και στέκει για κάθε ακολουθία χαρακτήρα. Οι επιτρεπόμενες ακολουθίες διαφυγής είναι οι ακόλουθες (όλες οι άλλες χρήσεις του \είναι άκυρες):

- \n Χαρακτήρας ο οποίος αναγνωρίζεται από το σύστημα ως τέλος γραμμής.
- \t tab
- \^c Χαρακτήρας ελέγχου c, για όποιο οποιοδήποτε κατάλληλο c.
- \ddd Ένας χαρακτήρας ο περιγράφετε με 3 ακεραίους στον πίνακα ASCII.
- \" Χαρακτήρας εισαγωγικά

- \Χαρακτήρας backslash
- \f _ _ _ \f Επιτρέπει την επέκταση ενός αλφαριθμητικού σε νέα γραμμή, βάζοντας στο τέλος το και ξεκινώντας στην επόμενη με .

Άρνηση (Negation): Μια έκφραση ακέραιας τιμής μπορεί να πάρει αρνητικό πρόθεμα.

Κλήση συνάρτησης (function call): Η εκτέλεση της συνάρτησης id() ή id(exp, exp) υποδεικνύει την εφαρμογή της συνάρτησης id σε μια λίστα τιμών των πραγματικών παραμέτρων που αποκτήθηκαν με την εκτέλεση των εκφράσεων από αριστερά στα δεξιά. Η πραγματικές παράμετροι είναι δεμένες στις αντίστοιχες επίσημες παραμέτρους του ορισμού της συνάρτησης και στο σώμα της συνάρτησης δένετε το αποτέλεσμα του χρησιμοποιώντας συμβατικό(κλασσικό) στατικό έλεγχο. Αν το id στέκει για ρουτίνα (συνάρτηση που δεν επιστρέφει αποτέλεσμα), τότε το σώμα της συνάρτησης πρέπει να παράγει καμία τιμή, και η εκτέλεση της συνάρτησης επίσης πρέπει να παράγει καμία τιμή.

Αριθμητική (Arithmetic): Εκφράσεις της μορφής exp op exp, όπου op είναι +,-,*,/, χρειάζονται ακέραιες τιμές και παράγουν ένα ακέραιο αποτέλεσμα.

Σύγκριση (Comparison): Εκφράσεις της μορφής exp op exp, όπου op είναι =,<>,>,<,>=,<= συγκρίνουν τους τελεστές για ισότητα ή ανισότητα και παράγουν τον ακέραιο 1 για σωστό, 0 για λάθος. Αυτά τα σύμβολα μπορούν να εφαρμοστούν σε ακέραιους τελεστές. Τα σύμβολα ισότητας και ανισότητας μπορούν επίσης να εφαρμοστούν σε δύο εγγραφές ή δύο πίνακες και συγκρίνονται για «αναφορική» ή «δεικτική» ισότητα. (εξετάζεται κατά πόσο δύο εγγραφές έχουν την ίδια περίπτωση(instance), όχι κατά πόσο έχουν το ίδιο περιεχόμενο).

Σύγκριση αλφαριθμιτικών (String comparison): Τα τελεστές σύγκρισης μπορούν επίσης να εφαρμόζονται σε αλφαριθμητικά. Δύο αλφαριθμητικά είναι ίσα αν τα περιεχόμενα τους είναι ίσα. Η σύγκριση γίνεται σύμφωνα με λεξικογραφική σειρά.

Τελεστές μπουλ (Boolean operators): Εκφράσεις της μορφής exp op exp, όπου op είναι & ή |, είναι τελεστές οι οποίοι δεν εκτελούν το δεξί τελεστή αν το αποτέλεσμα έχει καθοριστεί από τον αριστερό τελεστή. Οποιαδήποτε μη μηδενική ακέραια τιμή θεωρείται σωστή, και μια ακέραια τιμή μηδέν λανθασμένη.

Προτεραιότητα τελεστών (Precedence of operators): Μοναδιαία αφαίρεση (άρνηση) έχει την μεγαλύτερη προτεραιότητα. Μετά οι τελεστές *,/ έχουν την επόμενη μεγαλύτερη προτεραιότητα, ακολουθόμενοι απο +,-, και μετά =,<>,>,<,>=,<= και μετά το &, και μετά το |. Προσεταιριστικότητα τελεστών (Associativity of operators): Οι τελεστές *,/,+,- έχουν όλοι αριστερή προσεταιριστικότητα. Οι τελεστές σύγκρισης δεν εχουν προσεταιριστικότητα, έτσι a=b=c δεν είναι ορθή έκφραση, παρόλο που a=(b=c) είναι ορθό.

Δημιουργία εγγραφών (Record creation): Η έκφραση type-id id=exp, id=exp ή (για όποιο κενό τύπο εγγραφής) type-id δημιουργεί μια νέα περίπτωση εγγραφής τύπου type-id. Το όνομα και ο τύπος των πεδίων της έκφρασης της εγγραφής πρέπει να συμφωνούν με αυτά που έχουν δοθεί στον ορισμό.

Δημιουργία πίνακα (Array creation): Η έκφραση type-id [exp1] of exp2 εκτελεί την exp1 και exp2 (με αυτή την σειρά) για να βρεί το n, τον αριθμό των στοιχείων, και v την αρχική τιμή. Ο τύπος type-id πρέπει να έχει δηλωθεί σαν τύπος πίνακα. Το αποτέλεσμα της έκφρασης είναι ένας νέος πίνακας με τύπο type-id, ξεκινώντας με θέση (βάση) το 0 μέχρι n – 1, με κάθε θέση να αρχικοποιείται με την τιμή v.

Εκχώρηση πίνακα και εγγραφής (Array and record assignment): Όταν μιά μεταβλητή a ενός πίνακα ή εγγραφής εκχωρείται μια τιμή b, τότε η a αναφέρεται στον ίδιο πίνακα ή εγγραφή όπως η b. Μελλοντικές ενημερώσεις στα στοιχεία του a θα επηρεάσουν και το b, και αντίθετα, μέχρι το a να ξανά-ενημερωθεί. Το πέρασμα των παραμέτρων πινάκων ή εγγραφών γίνονται με κατά αναφορά και όχι με αντιγραφή.

Έκταση (Extent): Οι πίνακες και οι εγγραφές έχουν απεριόριστη έκταση. Κάθε τιμή εγγραφής ή πίνακα διαρκεί για πάντα, ακόμα και μετά από τον σκοπό τον οποίο δημιουργήθηκε.

Εκχώρηση (Assignment): Η εκχώρηση lvalue := exp υπολογίζει την lvalue, και μετά υπολογίζει την exp, μετά θέτει τα περιεχόμενα της lvalue από το αποτέλεσμα της έκφρασης. Συντακτικά, το := ειναι ασθενέστερο απο τους μπουλ τελεστές & και |. Η έκφραση της εκχώρησης παράγει καμία τιμή.

if-then-else: Η έκφραση if, if exp1 then exp2 υπολογίζει την ακέραια έκφραση exp1. Αν το αποτέλεσμα δεν είναι μηδέν επιστρέφει το αποτέλεσμα του υπολογισμού της exp2, ιδιαλιός επιστρέφει το αποτέλεσμα της exp3. Οι εκφράσεις exp2 και exp3 πρέπει να έχουν τον ίδιο τύπο, ο οποίος επίσης είναι ο τύπος ολόκληρης της εκφρασης if (ή οι δύο εκφράσεις πρέπει να παράγουν καμία τιμή).

if-then: Η έκφραση if exp1 then exp2 υπολογίζει την ακέραια έκφραση exp1. Αν το αποτέλεσμα είναι μη μηδέν, τότε το exp2 (το οποίο παράγει καμία τιμή) υπολογίζεται. Ολόκληρη η έκφραση if παράγει καμία τιμή.

While: Η έκφραση while exp1 do exp2 υπολογίζει την ακέραια έκφραση exp1. Αν το αποτέλεσμα είναι μη μηδέν, τότε το exp2 (το οποίο παράγει καμία τιμή) εκτελείται, και μετά ολόκληρη η έκφραση while επαναεκτελείται.

for: Η έκφραση for id := exp1 to exp2 do exp3 επαναλαμβάνει το exp3 με την κάθε ακέραια τιμή id, που βρίσκεται ανάμεσα στο exp1 και exp2. Η μεταβλητή id είναι μια καινούργια μεταβλητή η οποία δηλώνεται σιωπηρά από την for, όπου ο σκοπός της καλύπτει μόνο το exp3, και δεν μπορεί να εκχωρηθεί. Το σώμα του exp3 πρέπει

να παράγει καμία μεταβλητή. Το άνω και κάτω φράγμα υπολογίζονται μόνο μια φορά, πριν μπει στο σώμα του βρόγχου. Αν το άνω φράγμα είναι πιο μικρό απο το κάτω, τότε το σώμα δεν εκτελείτε.

break: Η έκφραση break τερματίζει τον υπολογισμό της πιο κοντινής while έκφρασης ή for έκφρασης. Μια break σε ρουτίνα p δεν μπορεί να τερματίσει ένα βρόγχο σε μια ρουτίνα q, ακόμα και άν η p είναι ένθετη μέσα στη q. Μια break που δεν είναι μέσα σε while ή for είναι άκυρη.

let: Η έκφραση let decs in expseq end υπολογίζει τις δηλώσεις decs, δεσμεύοντας τύπους, μεταβλητές, και ρουτίνες όπου ο σκοπός τους μετά επεκτείνετε στην expseq. Η expseq είναι μια σειρά απο μηδέν ή περισσότερες εκφράσεις, διαχωριζόμενες από αγγλικά ερωτηματικά. Το αποτέλεσμα (αν υπάρχει) της τελευταίας exp στη σειρά είναι και το αποτέλεσμα ολόκληρης της έκφρασης let.

Παρενθέσεις(Parentheses): Οι παρενθέσεις γύρο από κάθε έκφραση επιβάλουν συντακτική ομαδοποίηση, όπως και στις πιο πολλές γλώσσες προγραμματισμού.

3.

Λεκτική Ανάλυση

Στο κεφάλαιο αυτό περιγράφουμε αρχικά τα βασικά σημεία της πρώτης φάσης της μεταγλώττισης ενός προγράμματος και στη συνέχεια παρουσιάζουμε την υλοποίηση της στην περίπτωση της tiger. Για περισσότερες λεπτομέρειες σχετικά με τη φάση της λεκτικής ανάλυσης ο αναγνώστης παραπέμπεται στο [1].

Η υλοποίηση της λεκτικής ανάλυσης της tiger γίνεται μέσω της γεννήτριας λεκτικών αναλυτών ml-lex[3]. Ο κώδικας που ορίζει τις προδιαγραφές του λεκτικού αναλυτή που προκύπτει βρίσκεται στο αρχείο tiger.lex. Όταν τρέξει ο ml-lex πάνω στο αρχείο tiger.lex παράγει το αρχείο πηγαίου κώδικα tiger.lex.sml. Αυτό είναι το αρχείο που χρησιμοποιείται κατά την μεταγλώττιση της tiger.

3.1 Η λεκτική ανάλυση ενός προγράμματος

Η λεκτική ανάλυση αποτελεί την πρώτη φάση της μεταγλώττισης (μετάφρασης) ενός προγράμματος. Ο λεκτικός αναλυτής δέχεται ως είσοδο ένα ρεύμα (stream) από χαρακτήρες και παράγει ένα άλλο ρεύμα (stream) από λεκτικά (lexemes). Αυτά τα λεκτικά τα κατηγοριοποιούνται ανάλογα με τον ορισμό της γλώσσας. Για παράδειγμα, μπορεί να κατηγοριοποιούνται ως ονόματα, ως δεσμευμένες λέξεις ή ως σημεία στίξης.

Διαχωρισμός λεκτικών μονάδων. Τα λεκτικά μαζί με την κατηγοριοποίηση τους ονομάζονται λεκτικές μονάδες (tokens). Η διαδικασία της εξαγωγής των tokens από το ρεύμα εισόδου χαρακτήρων ονομάζεται διαχωρισμός λεκτικών μονάδων (tokenization).

Πιο συγκεκριμένα, μπορούμε να αναπαραστήσουμε ένα token ως ένα ζευγάρι της μορφής <κλάση, αλφαριθμητικό> όπου η κλάση αντιστοιχεί σε κάποια κατηγορία των λεκτικών μονάδων και το αλφαριθμητικό αντιστοιχεί στο ίδιο το λεκτικό. Για παράδειγμα, τα παρακάτω ζευγάρια αναπαριστούν tokens: <id, “hey”> , <int, “45”> , <string, “me”>,

όπου το πρώτο αντιστοιχεί στο λεκτικό "hey" που είναι όνομα (id), το δεύτερο αντιστοιχεί στο λεκτικό 45 που είναι ακέραιος (int), και το τρίτο στο αλφαριθμητικό (string) "me".

Ο κύριος λόγος που χρησιμοποιείται ο λεκτικός αναλυτής ως ξεχωριστό στάδιο στην μεταγλώττιση είναι για να απλοποιήσει την διαδικασία της συντακτικής ανάλυσης.

Κανονικές εκφράσεις & αυτόματα Οι κανονικές εκφράσεις αποτελούν ένα μέσο περιγραφής μιας γλώσσα, ουσιαστικά ορίζουν τα έγκυρα λεκτικά τη γλώσσας. Στην λεκτική ανάλυση μπορούν να χρησιμοποιηθούν λεκτικοί αναλυτές οι οποίοι μεταφράζουν τις κανονικές εκφράσεις που περιγράφουν την αρχική γλώσσα σε κώδικα μιας γλώσσας προγραμματισμού. Μία άλλη επιλογή είναι ο κώδικας της λεκτικής ανάλυσης να γραφτεί στο "χέρι" (hard coded) σε μια γλώσσα προγραμματισμού.

Για παράδειγμα, τα αναγνωριστικά (identifiers) ή ονόματα (μεταβλητών ή άλλων οντοτήτων που προγράμματος) μιας γλώσσας ορίζονται από την παρακάτω κανονική έκφραση

```
ID = [A-Za-z][_A-Za-z0-9]*
```

η οποία δηλώνει πως τα αναγνωριστικά πρέπει να ξεκινάνε με ένα γράμμα, μικρό ή κεφαλαίο και μπορούν να ακολουθούνται από κανένα ή περισσότερα μικρά ή κεφαλαία γράμματα, υπογράμμιση ή αριθμούς.

Αφού πραγματοποιηθεί η περιγραφή των λεκτικών μονάδων του προγράμματος με κανονικές εκφράσεις, στη συνέχεια οι λεκτικοί αναλυτές μετατρέπουν τις κανονικές εκφράσεις σε Μη Ντετερμινιστικά Αυτόματα (ΜΝΑ). Τα ΜΝΑ από μια κατάσταση μπορούν να μεταβαίνουν σε πολλές. Αυτό ώμος δεν είναι βολικό για τους υπολογιστές γιατί δεν μπορούν να μαντέψουν πιο είναι το σωστό μονοπάτι. Έτσι οι λεκτικοί αναλυτές μετατρέπουν το ΜΝΑ σε Ντετερμινιστικά Αυτόματα (ΝΑ). Τα ΝΑ έχουν ένα και μόνο μονοπάτι που οδηγεί στην αποδεκτή κλάση token. Έτσι τα ΝΑ είναι ιδανικά για τους υπολογιστές.

3.2 Ο λεκτικός αναλυτής της tiger

Ο παραγωγός λεκτικών αναλυτών ml-lex που χρησιμοποιείται για την υλοποίηση της tiger, παράγει ένα λεκτικό αναλυτή ο οποίος είναι η υλοποίηση ενός ντετερμινιστικού πεπερασμένου αυτόματου (ΝΠΑ) με συνολικά 115 καταστάσεις. Το αυτόματο αυτό χρησιμοποιείται για την αναγνώριση των λεκτικών χαρακτηριστικών της tiger.

3.2.1 Ο παραγωγός λεκτικού αναλυτή ml-lex

Ο παραγωγός λεκτικού αναλυτή ml-lex χρησιμοποιείται γιατί παράγει τον λεκτικό αναλυτή σε γλώσσα SML. Το συντακτικό του μοιάζει με άλλους παρόμοιους λεκτικούς αναλυτές. Υπάρχουν τρία βασικά μέρη τα οποία χωρίζονται με %%. Η μορφή του είναι ως εξής:

Δηλώσεις χρήστη

%% Ορισμοί

%% Κανόνες

Στις δηλώσεις χρήστη τοποθετείται κώδικας σε γλώσσα SML ο οποίος είναι χρήσιμος για τον προγραμματιστή. Στους ορισμούς του ml-lex τοποθετούνται οδηγίες για το πρόγραμμα καθώς επίσης και οι ορισμοί των κανονικών εκφράσεων. Στους κανόνες τοποθετούνται οι κανόνες οι οποίοι περιγράφουν τα λεκτικά χαρακτηριστικά της γλώσσας.

3.2.2 Η υλοποίηση του παραγωγού λεκτικού αναλυτή της tiger

Δηλώσεις Χρήστη. Στις δηλώσεις χρήστη του λεκτικού αναλυτή βρίσκεται ο κώδικας ο οποίος ενσωματώνετε στο παραγόμενο αρχείο. Ουσιαστικά είναι ένας ευέλικτος τρόπος για την προσθήκη καινούριων υπηρεσιών στον λεκτικό αναλυτή. Συνοπτικά στην συγκεκριμένη υλοποίηση γίνονται οι ακόλουθες καταγραφές και έλεγχοι:

- Καταγραφή σε πια θέση και γραμμή βρίσκετε το token. (μεταβλητή linePos και lineNum αντίστοιχα)
- Καταγραφή αν ένα σχόλιο έχει κλείσει. (μεταβλητή commentsClosed)
- Καταγραφή αλφαριθμητικών για τις επόμενες αναλύσεις. (μεταβλητή str)
- Καταγραφή αν ένα αλφαριθμητικό έχει κλείσει.
- Στο τέλος της λεκτικής ανάλυσης καλείται η συνάρτηση eof() η οποία ελέγχει αν υπάρχουν σχόλια και αλφαριθμητικά ανοιχτά. Αν υπάρχουν τότε ο μεταγλωττιστής τερματίζεται και τυπώνει το αντίστοιχο σφάλμα.

```
val lineNum = ErrorMsg.lineNum
val linePos = ErrorMsg.linePos


val commentsClosed = ref true
val stringClosed = ref 0
val str = ref ""

fun eof() =
if not (!commentsClosed)
then (ErrorMsg.error (!lineNum) ("unmatch comment");
      Tokens.EOF(!lineNum,!lineNum))
else if !stringClosed <> 0
then (ErrorMsg.error (!lineNum) ("unmatch string");
      Tokens.EOF(!lineNum,!lineNum))
else Tokens.EOF(!lineNum,!lineNum)
```

Ορισμοί. Στους ορισμούς ορίζονται οι κανονικές εκφράσεις οι οποίες θα χρησιμοποιηθούν στο επόμενο μέρος. Για παράδειγμα ορίζεται μία μεταβλητή ID (identifier) στην οποία δεσμεύετε η κανονική έκφραση [A-Za-z][A-Za-z0-9]*, η οποία κανονική έκφραση περιγράφει τα αλφαριθμητικά της tiger. Παρομοίως ορίζονται και οι επόμενες κανονικές εκφράσεις. Επίσης στο μέρος του ορισμού δηλώνονται οι καταστάσεις οι οποίες αναλύονται στους κανόνες.

```
%s COMMENT STRING ESC FORMAT;
ID = [A-Za-z][_A-Za-z0-9]*;
```

Κανόνες. Η γενική μορφή ενός κανόνα είναι:

```
<start state list> regular expression => ( code );
```

όπου το `<start state>` είναι μία κατάσταση, της οποίας τη σημασία εξηγούμε αναλυτικά παρακάτω, το `regular expression` αποτελεί μια κανονική έκφραση που περιγράφει κάποιο token της γλώσσας, ενώ το code αντιστοιχεί στην κατηγοριοποίηση του token.

Το παρακάτω απόσπασμα κώδικα είναι από τον λεκτικό αναλυτή της tiger.

```
<INITIAL>"to"  => (Tokens.TO(yypos,yypos+2));

<INITIAL>"for" => (Tokens.FOR(yypos,yypos+3))
          .
          .
          .
```

Το πρώτο κομμάτι έχει την δομή <κατάσταση>. Αν ο λεκτικός αναλυτής βρίσκεται στην κατάσταση που αναγράφετε τότε θα εκτελεστεί το δεύτερο κομμάτι του κανόνα. Το δεύτερο κομμάτι στο απόσπασμα κώδικα είναι η κανονική έκφραση η οποία περιγράφει τα αλφαριθμητικά της γλώσσας που μας ενδιαφέρει. Το τρίτο κομμάτι (μετά το συνεπάγεται) είναι η κατηγοριοποίηση του αλφαριθμητικού με βάση τα λεκτικά χαρακτηριστικά της γλώσσας. Η κατηγοριοποίηση μαζί με το αλφαριθμητικό αποτελούν το token.

Τα tokens της tiger Τα tokens που χρησιμοποιεί η tiger είναι τα ακόλουθα:

```
type var function break of end in nil

let do to for while else then if array  /* */

"  \  /   := | & >= > <= < <> = / * - + . } {

] [ ) ( ; : ,
```

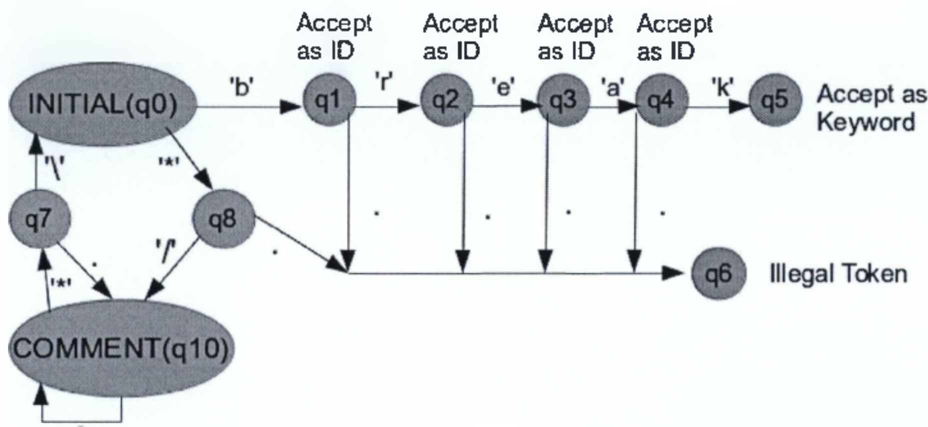
Οι αντίστοιχες κανονικές εκφράσεις. Παρακάτω παρουσιάζεται ο αντίστοιχος κώδικας στο ml-lex που περιγράφει τα tokens. Βρίσκεται στο μέρος με τους κανόνες στο τρίτο μέρος του ml-lex. Παρατηρούμε όλα τα έγκυρα tokens είναι δηλωμένα ανάμεσα σε διπλά εισαγωγικά.

```
<INITIAL>"type"      => (Tokens.TYPE(yypos,yypos+4));
<INITIAL>"var"       => (Tokens.VAR(yypos,yypos+3));
<INITIAL>"function"  => (Tokens.FUNCTION(yypos,yypos+8));
<INITIAL>"break"     => (Tokens.BREAK(yypos,yypos+5));
<INITIAL>"of"        => (Tokens.OF(yypos,yypos+2));
<INITIAL>"end"       => (Tokens.END(yypos,yypos+3));
<INITIAL>"in"        => (Tokens.IN(yypos,yypos+2));
<INITIAL>"nil"       => (Tokens.NIL(yypos,yypos+3));
<INITIAL>"let"       => (Tokens.LET(yypos,yypos+3));
<INITIAL>"do"        => (Tokens.DO(yypos,yypos+2));
<INITIAL>"to"        => (Tokens.TO(yypos,yypos+2));
<INITIAL>"for"       => (Tokens.FOR(yypos,yypos+3));
<INITIAL>"while"     => (Tokens.WHILE(yypos,yypos+5));
<INITIAL>"else"      => (Tokens.ELSE(yypos,yypos+4));
<INITIAL>"then"      => (Tokens.THEN(yypos,yypos+4));
<INITIAL>"if"        => (Tokens.IF(yypos,yypos+2));
<INITIAL>"array"     => (Tokens.ARRAY(yypos,yypos+5));
<INITIAL>"/*"        => (YYBEGIN COMMENT;
                         commentsClosed := false; continue());
<COMMENT>"*/"        => (YYBEGIN INITIAL;
                         commentsClosed := true; continue());
<COMMENT>.           => (continue());


<STRING>"\""         => (YYBEGIN INITIAL;
                        stringClosed := !stringClosed - 1;
                        let val temp = !str
                            val _ = str := ""
                        in
                            Tokens.STRING(temp,yypos,yypos)
                        end);

<STRING>"\\"         => (YYBEGIN ESC; continue());
<STRING>{newline}    => (ErrorMsg.error yypos
                        ("strings can not be splited to
                          separate lines " ^ yytext ^
                         " [use backslash for splitting
                          string]"); continue());
<STRING>.            => (str := !str ^ yytext; continue());
```

```
<ESC>{esqchar}        => (YYBEGIN STRING; str := !str ^ "\\"
                          ^ yytext; continue());
<ESC>{formatchar}     => (YYBEGIN FORMAT; continue());
<ESC>{oct}            => (YYBEGIN STRING; str := !str ^
                          String.str(Char.chr
                         (valOf(Int.fromString yytext)));
                          continue());
<ESC>.                => (ErrorMsg.error yypos
                         ("illegal escape character " ^ yytext);
                          continue());

<FORMAT>{formatchar} => (continue());
<FORMAT>"\\"          => (YYBEGIN STRING; continue());
<FORMAT>.             => (ErrorMsg.error yypos
                         ("illegal character " ^ yytext ^
                                  " [use backslash for splitting
                                  string]"); continue());

<INITIAL>"\""         => (YYBEGIN STRING;
                         stringClosed := !stringClosed + 1;
                         continue());
<INITIAL>":="         => (Tokens.ASSIGN(yypos,yypos+2));
<INITIAL>"|"          => (Tokens.OR(yypos,yypos+1));
<INITIAL>"&"          => (Tokens.AND(yypos,yypos+1));
<INITIAL>">="         => (Tokens.GE(yypos,yypos+2));
<INITIAL>">"          => (Tokens.GT(yypos,yypos+1));
<INITIAL>"<="         => (Tokens.LE(yypos,yypos+2));
<INITIAL>"<"          => (Tokens.LT(yypos,yypos+1));
<INITIAL>"<>"         => (Tokens.NEQ(yypos,yypos+2));
<INITIAL>"="          => (Tokens.EQ(yypos,yypos+1));
<INITIAL>"/"          => (Tokens.DIVIDE(yypos,yypos+1));
<INITIAL>"*"          => (Tokens.TIMES(yypos,yypos+1));
<INITIAL>"-"          => (Tokens.MINUS(yypos,yypos+1));
<INITIAL>"+"          => (Tokens.PLUS(yypos,yypos+1));
<INITIAL>"."          => (Tokens.DOT(yypos,yypos+1));
<INITIAL>"}"          => (Tokens.RBRACE(yypos,yypos+1));
<INITIAL>"{"          => (Tokens.LBRACE(yypos,yypos+1));
<INITIAL>"]"          => (Tokens.RBRACK(yypos,yypos+1));
<INITIAL>"["          => (Tokens.LBRACK(yypos,yypos+1));
<INITIAL>")"          => (Tokens.RPAREN(yypos,yypos+1));
<INITIAL>"("          => (Tokens.LPAREN(yypos,yypos+1));
<INITIAL>";"          => (Tokens.SEMICOLON(yypos,yypos+1));
<INITIAL>":"          => (Tokens.COLON(yypos,yypos+1));
```

```
<INITIAL>","         => (Tokens.COMMA(yypos,yypos+1));

<INITIAL>{ID}        => (Tokens.ID(yytext,yypos,
                         yypos+size(yytext)));
<INITIAL>{digit}+    => (Tokens.INT
                        (valOf(Int.fromString(yytext)),
                         yypos,yypos+size(yytext)));
```

Η έννοια της κατάστασης. Το μέρος `<start state>` ενός κανόνα αντιστοιχεί σε μία κατάσταση, ενώ μπορεί να υπάρχουν πολλές τέτοιες καταστάσεις. Η έννοια της κατάστασης στον ml-lex δεν έχει σχέση με τις καταστάσεις ενός ΝΠΑ. Η αξία των καταστάσεων ενός λεκτικού αναλυτή φαίνεται όταν ο αναλυτής χρειάζεται να αναγνωρίσει ίδια tokens αλλά χρειάζεται να έχουν διαφορετική ερμηνεία σε ένα σημείο ενός προγράμματος. Για παράδειγμα όπως όλες οι γλώσσες προγραμματισμού η tiger υποστηρίζει σχόλια. Τα σχόλια της tiger ξεκινάνε με /* και τελειώνουν με */. Τώρα αν μέσα στα σχόλια υπάρχει μια (ή περισσότερες) λέξεις κλειδιά θα ήταν δύσκολο να αναγνωριστεί στην ίδια κατάσταση το token σαν σχόλιο και όχι σαν λέξη κλειδί. Ο λεκτικός αναλυτής της tiger χρησιμοποιεί 5 καταστάσεις. Στην εικόνα φαίνονται διαισθητικά δύο καταστάσεις ενός λεκτικού αναλυτή:



Σχήμα 3.1: Διαισθητική παρουσίαση των καταστάσεων του ml-lex

Οι καταστάσεις με τα επιγράμματα INITIAL και COMMENT είναι οι καταστάσεις όπως τις αντιλαμβάνεται ο προγραμματιστής στο κομμάτι <state list> του ml-lex. Ανεξαρτήτως των επιγραμμάτων το κανονικό όνομα των καταστάσεων ως ΝΜΑ είναι q0,q1,q2 ... qn. Βλέπουμε την αντιστοιχία του επιγράμματος INITIAL με το q0. Αν για παράδειγμα εισαχθεί το αλφαριθμητικό break τότε από την κατάσταση q0 πάμε στην q1 μέχρι την

q5 και ταξινομείτε ως λέξη κλειδί. Αν εισαχθεί το αλφαριθμητικό br τότε από την κατάσταση q0 πάμε στην q1 και από την q1 στην q2. Αν ο επόμενος χαρακτήρας είναι 'e' τότε θα πάμε στην κατάσταση q3 αλλά επειδή είναι ο τελευταίος χαρακτήρας τερματίζει στην κατάσταση q2, ταξινομείται ως αναγνωριστικό. Αν εισαχθεί το αλφαριθμητικό br-dollar τότε θα πάμε στη κατάσταση q2 αλλά ο επόμενος χαρακτήρας δεν θα είναι e αλλά ούτε και ο τελευταίος χαρακτήρας για να τερματίσει. Έτσι θα εκτελεστεί η μετάβαση της τελείας (η τελεία υπονοεί όλους τους χαρακτήρες) και θα τερματίσει στην κατάσταση q6 αναγνωρίζοντας το ως άκυρο token. Αν εισαχθεί το αλφαριθμητικό /* break */ τότε από την κατάσταση INITIAL (q0) πάμε στη κατάσταση COMMENT εκτελώντας την διαδρομή του /*. Τώρα λόγο του ότι βρισκόμαστε σε διαφορετική κατάσταση το break για κάθε χαρακτήρα εκτός του * παραμένει στην κατάσταση COMMENT(q0). Πηγαίνει πίσω στην INITIAL μόνο όταν διαβάσει το αλφαριθμητικό */ .

3.3 Παράδειγμα

Στην ενότητα αυτή δίνουμε ένα παράδειγμα προγράμματος στην tiger και παρουσιάζουμε το αποτέλεσμα που εκτυπώνει ο λεκτικός αναλυτής για τα tokens του προγράμματος.

Το πρόγραμμα. Το παρακάτω πρόγραμμα υπολογίζει αναδρομικά το άθροισμα.

```
let
        var N := 5
        function summation(sum : int) : int =
                if sum=0
                then 0
                else sum + summation(sum-1)
in
        print(chr(summation(N)))
end
```

Το αποτέλεσμα της λεκτικής ανάλυσης. Ο λεκτικός αναλυτής δεν μπορεί να εκτελεστεί ξεχωριστά στην tiger. Στο παρακάτω παράδειγμα έχει εκτελεστεί ο ml-lex ως ξεχωριστό πρόγραμμα και όχι σαν μέρος της tiger. Αν εκτελεστεί η λεκτική ανάλυση στο παραπάνω πρόγραμμα ο λεκτικός αναλυτής θα εκτυπώσει:

```
LET   2
VAR   7
ID(N)     11
ASSIGN   13
INT(5)   16
FUNCTION   19
```

...

Βλέπουμε ότι στο αριστερό κομμάτι είναι εκτυπωμένο το όνομα (ταξινόμηση) του token που αναγνωρίστηκε και αν συνοδεύεται με κάποια τιμή τότε αυτή αναγράφετε μέσα στις παρενθέσεις. Στο δεξί κομμάτι είναι η θέση που βρίσκεται η λέξη. Η θέση είναι η διαφορά σε χαρακτήρες από την αρχή του αρχείου.

Στην υλοποίηση της tiger χρησιμοποιήθηκε ο λεκτικός αναλυτής ml-lex και ο κώδικας του λεκτικού αναλυτή της βρίσκεται στο αρχείο tiger.lex. Όταν τρέξει ο ml-lex πάνω στο αρχείο tiger.lex παράγει το αρχείο πηγαίου κώδικα tiger.lex.sml. Αυτό είναι το αρχείο που χρησιμοποιείτε κατά την μεταγλώττιση της tiger. Αν γίνει μια αλλαγή στο αρχείο με τις κανονικές εκφράσεις τότε πρέπει πρώτα να παραχθεί το αρχείο πηγαίου κώδικα με τις αλλαγές και μετά να μεταγλωττιστεί ξανά η tiger.

4.

Συντακτική Ανάλυση

Στο κεφάλαιο αυτό περιγράφουμε αρχικά τα βασικά σημεία της δεύτερης φάσης της μεταγλώττισης ενός προγράμματος και στη συνέχεια παρουσιάζουμε την υλοποίηση της στην περίπτωση της tiger. Για την υλοποίηση της tiger χρησιμοποιήθηκε ο παραγωγός συντακτικού αναλυτή ml-yacc και ο κώδικας του συντακτικού αναλυτή βρίσκεται στο αρχείο tiger.grm. Όταν τρέξει ο ml-yacc πάνω στο αρχείο tiger.grm παράγει το αρχείο πηγαίου κώδικα tiger.grm.sml και tiger.grm.sig. Αυτά είναι τα αρχεία που χρησιμοποιούνται κατά την μεταγλώττιση της tiger.

4.1 Η συντακτική ανάλυση ενός προγράμματος

Η συντακτική ανάλυση είναι η δεύτερη φάση της μεταγλώττισης. Ο συντακτικός αναλυτής είναι υπεύθυνος κατά πόσο ένα πρόγραμμα είναι συντακτικά σωστό. Δηλαδή παίρνει σαν είσοδο ένα ή περισσότερα tokens και αποφασίζει αν είναι συντακτικά σωστά.

Για παράδειγμα, τα tokens

```
while   (    )    test
```

είναι έγκυρα στην tiger. Πέραν των έγκυρων tokens χρειάζεται και να τοποθετηθούν και στη σωστή σειρά ώστε να έχουμε μια συντακτικά ορθή έκφραση. Έτσι η παρακάτω έκφραση

```
while test ( )
```

δεν είναι συντακτικά ορθή και αποτελεί μη έγκυρη έκφραση της γλώσσας, ενώ η παρακάτω είναι έγκυρη

```
while ( test )
```

Γραμματικές χωρίς συμφραζόμενα Για την συντακτική ανάλυση χρησιμοποιείται η έννοια των γραμματικών χωρίς συμφραζόμενα (context-free grammars). Οι γραμματικές χωρίς συμφραζόμενα χρησιμοποιούνται αντί για τα Μη Ντετερμινιστικά Αυτόματα για το λόγο ότι τα ΜΝΑ δεν μπορούν να μετρούν. Για παράδειγμα ένα ΜΝΑ δεν μπορεί να αντιληφθεί αν μια ανοιχτή παρένθεση έχει κλείσει, γιατί τα ΜΝΑ απλά αλλάζουν καταστάσεις χωρίς να υπάρχει κάποιος τρόπος να ταιριάζουν ανοιχτές με κλειστές παρενθέσεις. Παρακάτω βλέπουμε εκφράσεις γραμματικών χωρίς συμφραζόμενα. Είναι απόσπασμα από τον συντακτικό αναλυτή της tiger.

```
ty_field: ID COLON ID

ty_fields:  ty_field ty_fields_more
       | (* empty *)

ty_fields_more: COMMA ty_field ty_fields_more
       | (* empty *)
```

Οι εκφράσεις ty_fields χρησιμοποιούνται για την δήλωση των πεδίων μιας εγγραφής και των παραμέτρων μιας συνάρτησης. Ξεκινώντας από το ty_fields η δεύτερη περίπτωση είναι ένα κενό. Αυτό συνεπάγεται ότι μια εγγραφή μπορεί να δηλωθεί κενή . Η πρώτη περίπτωση βλέπουμε ότι χρειάζεται ένα ty_field δηλαδή ένα αναγνωριστικό ακολουθούμενο από μια άνω κάτω τελεία ακολουθούμενη από ένα ακόμη αναγνωριστικό. Μετά ακολουθεί το ty_fields_more, η δεύτερη περίπτωση του είναι κενό, αυτό έχει σαν αποτέλεσμα μια εγγραφή με ένα δηλωμένο πεδίο, για παράδειγμα data : int . Η πρώτη περίπτωση του είναι ένα κόμμα ακολουθούμενα από ένα ty_field και μετά ένα ty_fields_more. Συνεπάγεται ότι κάθε φορά που επιλέγετε η πρώτη περίπτωση αναγνωρίζει ένα δηλωμένο πεδίο. Όταν επιλεχτεί η πρώτη περίπτωση τότε δεν υπάρχουν άλλα πεδία προς αναγνώριση. Αν για παράδειγμα στην πρώτη περίπτωση του ty_fields_more ο λεκτικός αναλυτής επιστρέψει ένα token DOT αντί για COMMA τότε ο έλεγχος token == COMMA θα αποτύχει και ο συντακτικός αναλυτής θα εκτυπώσει συντακτικό σφάλμα.

Συντακτικοί παραγωγοί Συντακτικοί παραγωγοί είναι προγράμματα τα οποία βοηθάνε τους προγραμματιστές να γράφουν σε μορφή γραμματικών χωρίς συμφραζόμενα χωρίς να εκτίθενται στις λεπτομέρειες των αλγορίθμων και δομών δεδομένων για την υλοποίηση της συντακτικής ανάλυσης. Συνήθως οι συντακτικοί παραγωγοί υλοποιούν ανάλυση από-κάτω-προς-τα-πάνω (bottom-up parsing) αλλά υπάρχουν και από-πάνω-προς-τα-κάτω (top-down parsing).

Η διαδικασία της υλοποίησής της απο-κάτω-προς-τα-πάνω ανάλυσης είναι πολύ χρονοβόρα και γιαυτό συνήθως χρησιμοποιείτε έτοιμος συντακτικός αναλυτής. Αντιθέτως οι απο-πάνω-προς-τα-κάτω συνήθως γράφονται στο χέρι.

Διαδικασία μεταγλώττισης Στην υλοποίηση της tiger χρησιμοποιήθηκε ο συντακτικός αναλυτής ml-yacc και ο κώδικας του συντακτικού αναλυτή της βρίσκεται στο αρχείο

tiger.grm.

Λόγω του ότι ο συντακτικός αναλυτής χρειάζεται συνεχώς καινούργια tokens είναι στενά συνδεδεμένος με τον λεκτικό αναλυτή. Κάθε φορά που χρειάζεται ένα η περισσότερα tokens ο συντακτικός αναλυτής καλεί τον λεκτικό να του δώσει. Έτσι η διαδικασία του δεσίματος λεκτικού αναλυτή – συντακτικού αναλυτή γίνετε ως εξής:

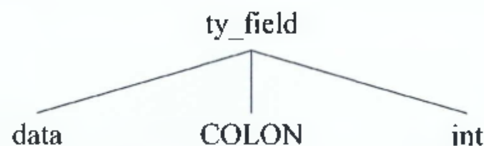
- Τρέχει το ml-yacc στο tiger.grm και παράγει τα αρχεία tiger.grm.sig και tiger.grm.sml. Το αρχείο tiger.grm.sig περιέχει τα πρότυπα όλων των token καθώς και πρότυπα δομών και συναρτήσεων. Το αρχείο tiger.grm.sml περιέχει την υλοποίηση του συντακτικού αναλυτή.

- Μετά τρέχει το ml-lex και παράγει την υλοποίηση του λεκτικού αναλυτή που βρίσκεται στο αρχείο tiger.lex.sml.

- Τα τρία παραγόμενα αρχεία τα δένει το αρχείο parse.sml. Επίσης περιέχει μια συνάρτηση που παίρνει σαν παράμετρο το όνομα του αρχείου κώδικά της tiger και όταν κληθεί ξεκινά την συντακτική ανάλυση.

Συντακτικά δένδρα Το αποτέλεσμα της συντακτικής ανάλυσης είναι μια δομή δένδρου.

Στο παράδειγμα της συντακτικής ανάλυσης είχαμε την γραμματική ty_field: ID COLON ID . Όταν αναγνωριστεί μια τέτοια έκφραση τότε σχηματικά έχουμε το παρακάτω σχήμα:



Σχήμα 4.1: Συντακτικό δέντρο της ty_field.

Παρατηρούμε ότι τα ID έχουνε αντικατασταθεί με πραγματική τιμή.

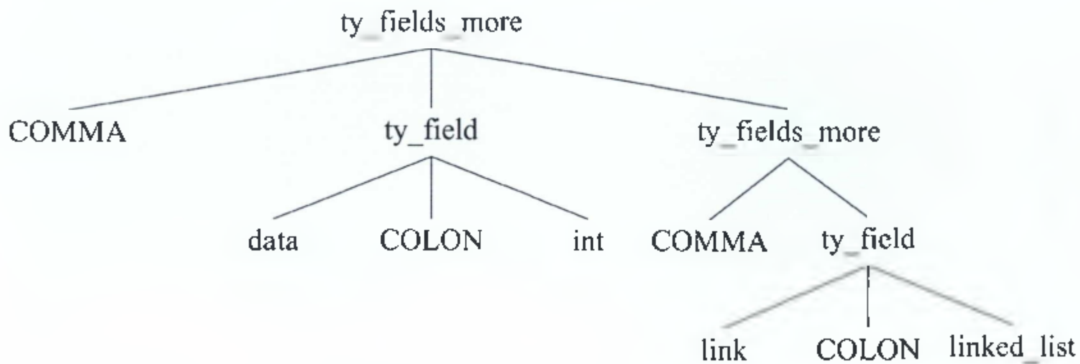
Το ty_field θα μπορούσε ακόμα να είναι μέρος ενός πιο μεγάλου δένδρου. Όπως για παράδειγμα της γραμματικής,

```
ty_fields_more: COMMA ty_field ty_fields_more
      | (* empty *)
```

Βλέπουμε ότι η ty_fields_more περιέχει την ty_field και έτσι μπορεί να φτιαχτεί ένα πιο σύνθετο δένδρο όπως βλέπουμε παραπάνω.

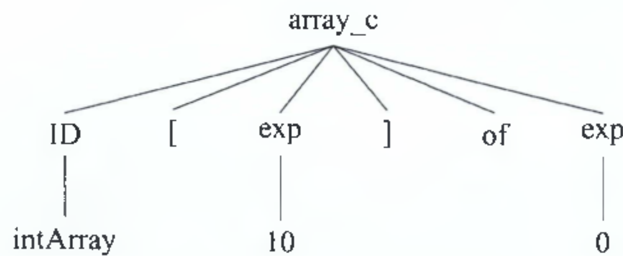
Υπάρχουν δύο τύποι συντακτικών δένδρων στην συντακτική ανάλυση, τα συμπαγή και αφηρημένα.

Οι συντακτικοί αναλυτές φτιάχνουν συμπαγή συντακτικά δένδρα καθώς αναλύουν την σύνταξη του προγράμματος. Το δένδρο αυτό φτιάχνετε εικονικά ως αποτέλεσμα της συντακτικής ανάλυσης αλλά δεν αποθηκεύεται κάπου. Τα πιο πάνω παραδείγματα δένδρων ήταν συμπαγές συντακτικά δένδρα.



Σχήμα 4.2: Συντακτικό δέντρο της ty_fields_more.

Βλέπουμε στο παρακάτω σχήμα το συμπαγές δένδρο για την δημιουργία ενός πινάκα 10 θέσεων και αρχικοποιημένες με 0.

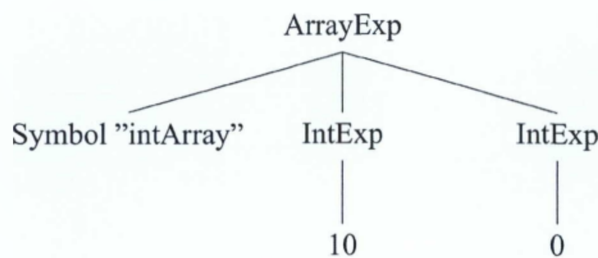


Σχήμα 4.3: Συμπαγές συντακτικό δέντρο.

Παράλληλα με το συντακτικό δένδρο ο αναλυτής φτιάχνει και ένα αφηρημένο συντακτικό δένδρο. Σε αντίθεση με το συμπαγές το αφηρημένο δεν περιέχει πληροφορίες όπως σημεία στίξης ή οτιδήποτε άλλο token το οποίο απλά προσφέρει συντακτική ευκολία στην γλώσσα. Για παράδειγμα, το ερωτηματικό χρησιμοποιείται σε πολλές γλώσσες για να δηλώσει το τέλος μιας έκφρασης αλλά δεν έχει κάποια σημασιολογική αξία. Ο λόγος που φτιάχνετε το δένδρο αυτό είναι για να περαστεί στην επόμενη φάση της μεταγλώττισης, την σημασιολογικής ανάλυσης Στο παρακάτω σχήμα δίνεται ένα παράδειγμα αφηρημένου συντακτικού δένδρου:

Βλέπουμε ότι το αφηρημένο δένδρο έχει απλοποιηθεί περιέχοντας μόνο σημασιολογικά, χρήσιμες πληροφορίες, για την δημιουργία ενός πίνακα. Δηλαδή το όνομα του τύπου, το μέγεθος του και την αρχική τιμή των θέσεων.

Οι τύποι δεδομένων του αφηρημένου δένδρου βρίσκονται στο αρχείο absyn.sml. Κάποιος μελετώντας τους τύπους δεδομένων μπορεί να δει πως αναπαριστούν την δομή δε-



Σχήμα 4.4: Αφηρημένο συντακτικό δέντρο.

δομένων ενός δένδρου. Υπάρχουν τρις κυρίως τύποι.

Ο τύπος var περιγραφεί τις μεταβλητές η οποίες μπορούν να είναι απλή μεταβλητή, μεταβλητή ενός πεδίου εγγραφής ή μεταβλητή μιας θέσης σε ένα πινάκα. Ο τύπος dec περιγράφει τις τρις πιθανές δηλώσεις σε μια έκφραση let. Την δήλωση συναρτήσεων, τύπων και μεταβλητών. Ο τύπος exp περιγράφει όλες τις πιθανές εκφράσεις. Οι var και dec είναι δεμένες με τον τύπο exp μέσο του κατασκευαστή (constructor) VarExp και LetExp αντίστοιχα.

4.2 Συντακτικός αναλυτής της tiger

Ο συντακτικός αναλυτής που χρησιμοποιείται για την υλοποίηση της tiger, ml-yacc[3], μεταφράζει κώδικα που είναι γραμμένος σε BNF γραμματική σε συντακτικό αναλυτή LALR στη γλώσσα SML. Η γραμματική BNF χρησιμοποιείτε για να περιγράφει εύκολα γραμματικές χωρίς συμφραζόμενα. Ο συντακτικός αναλυτής LALR είναι το αποτέλεσμα του ml-yacc και είναι ο συντακτικός αναλυτής της tiger. Το LALR κάνει συντακτική από αριστερά στα δεξιά και παράγει ένα συντακτικό δένδρο από-κάτω-προς-τα-πάνω από δεξιά στα αριστερά.

4.2.1 Ο παραγωγός συντακτικού αναλυτή ml-yacc

Ο παραγωγός συντακτικού αναλυτή ml-yacc χρησιμοποιείται γιατί παράγει κώδικα σε γλώσσα SML. Το συντακτικό του μοιάζει με άλλους παρόμοιους συντακτικούς αναλυτές (yacc, gnu bison, κτλ.). Υπάρχουν τρία βασικά μέρη τα οποία χωρίζονται με %%. Η μορφή του είναι ως εξής:

Δηλώσεις χρήστη

%% Ορισμοί

%% Κανόνες

Στις δηλώσεις χρήστη τοποθετείτε κώδικας σε γλώσσα SML ο οποίος είναι χρήσιμος για τον προγραμματιστή. Στους ορισμούς του ml-yacc τοποθετούνται οδηγίες για το πρό-

γραμμα. Επίσης και οι ορισμοί των tokens και η προτεραιότητα τους (αν υπάρχει). Στους κανόνες τοποθετούνται οι κανόνες οι οποίοι περιγράφουν τα συντακτικά χαρακτηριστικά της γλώσσας.

4.2.2 Η υλοποίηση του παραγωγού συντακτικού αναλυτή της tiger

Δηλώσεις χρήστη Στις δηλώσεις χρήστη του συντακτικού αναλυτή βρίσκεται ο κώδικας ο οποίος ενσωματώνετε στο παραγόμενο αρχείο. Ουσιαστικά είναι ένας ευέλικτος τρόπος για την προσθήκη καινούριων υπηρεσιών στον συντακτικό αναλυτή.

Στην συγκεκριμένη υλοποίηση δηλώνετε μία συνάρτηση η οποία βοηθά στην παραγωγή αφηρημένου δένδρου για σύνθετες προσβάσεις πεδίων εγγραφών ή πινάκων (πχ x.y.p[5]).

```
fun createLvalue(var, x::xs, pos) =
(case x of
Field s => createLvalue(A.FieldVar(var, s, pos), xs, pos)
        | Subscript ss => createLvalue(A.SubscriptVar(var,
                                ss, pos), xs, pos))
        | createLvalue(var, nil,_) = var
```

Οριμοί Στους ορισμούς συνοπτικά ορίζονται τα παρακάτω:

- Τα τερματικά της σύνταξης(tokens) και οι τύποι τους (εαν υπάρχουν)

```
%term
EOF
| ID of string
| INT of int
| STRING of string
| COMMA | COLON | SEMICOLON | LPAREN | RPAREN
| LBRACK | RBRACK
      .
      .
      .
```

- Τα μη τερματικά της σύνταξης και οι παραγόμενοι τύποι.

```
%nonterm program of A.exp
| exp of A.exp
| decs of A.dec list
| dec of A.dec
| tydec of {name : symbol, ty : A.ty, pos : int}
| tydec_more of {name : symbol, ty : A.ty, pos : int} list
      .
```

.

.

- Η προτεραιότητα[1] και η προσεταιριστικότητα τελεστών.

```
%left NEQ GT GE LT LE
%left PLUS MINUS
%left TIMES DIVIDE
```

.

.

.

Κανόνες Στους κανόνες ορίζεται με σύνταξη BNF το συντακτικό της tiger (στα αριστερά). Κάθε φορά που ταιριάζεται μία συντακτική πρόταση BFN εκτελείται το δεξί κομμάτι (ανάμεσα στις παρενθέσεις) που είναι κώδικας SML. Στον κώδικα της SML φτιάχνουμε το αφηρημένο συντακτικό δένδρο το οποίο θα επεξεργαστούμε στις επόμενες αναλύσεις.

Για παράδειγμα στο πιο κάτω απόσπασμα κώδικα βλέπουμε στη πρώτη γραμμή ότι αν ταιριάξει η πρόταση if <έκφραση> THEN <έκφραση> τότε θα εκτελεστεί ο κώδικας που βρίσκετε ανάμεσα στις παρενθέσεις. Ο εκτελέσιμος κώδικας δημιουργεί ένα τύπο IfExp ο οποίος κρατάει πληροφορίες για την έκφραση μετά το if, την έκφραση μετά το then,την έκφραση μετά το else και την τοποθεσία της έκφρασης. Η πληροφορία της τοποθεσίας της έκφρασης χρειάζεται γιατί μετά το τέλος της συντακτικής ανάλυσης δεν υπάρχει τρόπος να ξέρουμε την τοποθεσία των εκφράσεων όταν θα εκτελεστεί η σημασιολογική ανάλυση.

```
| IF exp THEN exp ELSE exp (A.IfExp{test=exp1, then'=exp2,
                             else'=SOME(exp3), pos=exp1left})
| IF exp THEN exp (A.IfExp{test=exp1, then'=exp2,
                   else'=NONE, pos=exp1left})
| WHILE exp DO exp (A.WhileExp{test=exp1, body=exp2,
                    pos=exp1left})
| FOR ID ASSIGN exp TO exp DO exp (A.ForExp{var=symbol ID,
                                   escape=ref false, lo=exp1,
                                   hi=exp2, body=exp3,
                                   pos=IDleft})
```

4.3 Παράδειγμα

Στην ενότητα αυτή δίνουμε ένα παράδειγμα προγράμματος στην tiger και παρουσιάζουμε το αποτέλεσμα που εκτυπώνει ο συντακτικός αναλυτής.

[1] όσο πιο ψηλά δηλώνοντε τα tokens τόσο πιο μεγάλη προτεραιότητα έχουν

Το πρόγραμμα. Το παρακάτω πρόγραμμα απλά δηλώνει μία μεταβλητή και επιστρέφει την τιμή a.

```
let
        var a := 8
in
        a
end
```

Το αποτέλεσμα της συντακτικής ανάλυσης. Ο συντακτικός αναλυτής δεν μπορεί να εκτελεστεί ξεχωριστά στην tiger. Στο παρακάτω παράδειγμα έχει εκτελεστεί ο ml-yacc ως ξεχωριστό πρόγραμμα και όχι σαν μέρος της tiger. Αν εκτελεστεί η συντακτική ανάλυση στο παραπάνω πρόγραμμα ο συντακτικός αναλυτής θα εκτυπώσει:

1. vardec
2. dec
3. lvalue
4. let

Βλέπουμε ότι εκτυπώνετε η έκφραση που ταίριαξε. Παρατηρώντας την σειρά που εκτυπώθηκαν οι εκφράσεις μπορούμε να συμπεράνουμε ότι είναι λογική ακολουθώντας το εξής μονοπάτι: Ο συντακτικός αναλυτής αρχικά ταιριάζει την δήλωση μεταβλητής(vardec), μετά ταιριάζει το μέρος let … in (dec) αφού δεν υπάρχουν άλλες δηλώσεις, έπειτα ταιριάζει την έκφραση που βρίσκεται ανάμεσα στο in … end και τέλος ταιριάζει την αρχική έκφραση let.

5.

Σημασιολογική ανάλυση

Η σημασιολογία γλωσσών είναι αυτή που διαφοροποιεί τις γλώσσες προγραμματισμού και επομένως είναι και η πιο ενδιαφέρουσα φάση. Αυτό αποδεικνύεται και στο ότι όλες οι περιγραφές (ορισμοί) γλωσσών περιγράφουν κυρίως την σημασιολογία της γλώσσας παρά την σύνταξη της. Μετά την λεκτική, συντακτική και σημασιολογική ανάλυση ολοκληρώνετε και ο ορισμός μιας γλώσσας.

5.1 Ο σημασιολογική ανάλυση ενός προγράμματος

Στην σημασιολογική ανάλυση ελέγχονται οι τύποι σύμφωνα με την γλώσσα, αδήλωτες μεταβλητές, ποιες μεταβλητές είναι σε σκοπό στην συγκεκριμένη συνάρτηση, δήλωσεις μεταβλητών και συναρτήσεων. Βλέπουμε ότι η σημασιολογική ανάλυση φέρει μεγάλη ευθύνη για την εγκυρότητα και την συνέπεια μιας γλώσσας. Φανταστείτε στην παραγωγή κώδικα αντί να παραχθεί κώδικας για μια τοπική μεταβλητή a, να παραχθεί για μια a που βρίσκετε σε άλλο σκοπό.

5.2 Ο σημασιολογικός αναλυτής της tiger

Ο σημασιολογικός αναλυτής είναι γραμμένος στο "χέρι" (όπως και οι περισσότεροι). Παίρνει σαν είσοδο ένα αφηρημένο συντακτικό δένδρο και επιστρέφει μια λίστα. Η λίστα περιέχει ένα δένδρο ενδιάμεσου κώδικα για κάθε συνάρτηση μαζί με το επίπεδο της συνάρτησης. Η σημασιολογική ανάλυση γίνετε αναδρομικά πάνω στο αφηρημένο δένδρο. Κατά την εκτέλεση του συλλέγονται (η διασκορπίζονται) πληροφορίες οι οποίες χρησιμοποιούνται κατά την διάρκεια της ανάλυσης του δένδρου. Η υλοποίηση του σημασιολογικού αναλυτή συλλέγει τέσσερις πληροφορίες κατά την εκτέλεση του. Το περιβάλλον

μεταβλητών και συναρτήσεων, το περιβάλλον τύπων, το επίπεδο συναρτήσεων και το βάθος των ένθετων βρόχων.

5.2.1 Περιβάλλον και τεχνικές σημασιολογικής ανάλυσης

Το περιβάλλον[1] κάνει την αντιστοιχία ενός αναγνωριστικού με τον τύπο και την τοποθεσία του στη μνήμη (μιλώντας αυστηρά η ακριβές τοποθεσία του στη μνήμη ορίζεται πολύ αργότερα, η τοποθεσία που κρατάει ο πίνακας συμβόλων είναι προσωρινή). Οι πίνακες συμβόλων λόγου του ότι θα επεξεργασθούν πολλά αναγνωριστικά στη διάρκεια της μεταγλώττισης πρέπει η πρόσβαση και η προσθήκη τους να είναι γρήγορη. Οι εμπορικοί μεταγλωττιστές υλοποιούν τους πίνακες συμβόλων με τις παρακάτω δομές δεδομένων που είναι πολύ γρήγορες στην αναζήτηση.

1. Πίνακα κατακερματισμού

2. Δυαδικά δένδρα αναζήτησης

Περιβάλλον για μεταβλητές και συναρτήσεις Το περιβάλλον περιέχει ποιες μεταβλητές είναι ενεργές σε ένα σημείο στο πρόγραμμα. Οι πληροφορίες που καταγράφονται για κάθε μεταβλητή είναι οι ακόλουθες όπως βλέπουμε στο πιο κάτω κώδικα.

```
signature ENV =
sig
  type ty = Types.ty
  datatype enventry =
      VarEntry of {access : Translate.access,
                   ty : ty}
    | FunEntry of {level : Translate.level,
                      label : Temp.label,
                                formals : ty list,
                                 result : ty}
  (* predefined types *)
  val base_tenv : ty Symbol.table
  (* predefined functions *)
  val base_venv : enventry Symbol.table
end
```

Ο τύπος δεδομένων enventry μπορεί να δημιουργηθεί με δύο κατασκευαστές. Ο ένας είναι τύπος VarEntry για μεταβλητές, ο οποίος κρατάει πληροφορίες για την τοποθεσία της μεταβλητής (συμβόλου) και τον τύπο της. Ο άλλος είναι ο τύπος FunEntry για συναρτήσεις, ο οποίος κρατάει πληροφορίες για το επίπεδο της συνάρτησης, το όνομα της συνάρτησης, τους τύπους των παραμέτρων και τέλος τον τύπο της επιστρεφόμενης τιμής.

[1] λέγεται και πίνακας συμβόλων

Το όνομα της συνάρτησης χρησιμοποιείται σαν σύμβολο για την αναζήτηση του τύπου FunEntry στον πίνακα συμβόλων αλλά και στον ίδιο τον τύπο FunEntry γιατί το όνομα χρειάζεται σαν σημείο εισόδου στην γλώσσα παραγόμενη γλώσσα μηχανής. Οι πιο πάνω πληροφορίες καταγράφονται στις δηλώσεις μίας έκφρασης let … in … end και προσκομίζονται όταν χρειάζεται μία μεταβλητή ή συνάρτηση. Για παράδειγμα στο πιο κάτω πρόγραμμα στην δήλωση της a ο πίνακας συμβόλων κρατάει την πληροφορία σε πια τοποθεσία βρίσκεται και τον τύπο της. Όταν χρειαστεί η μεταβλητή a τότε αναζητείται στον πίνακα συμβόλων η μεταβλητή a (και αν υπάρχει!) τότε ενσωματώνεται η διεύθυνση της στο δένδρο της ενδιάμεσης γλώσσας.

```
let
    var a := 5
in
    a + 4
end
```

Περιβάλλον για τύπους Ο πίνακας συμβόλων τύπων είναι πολύ πιο απλός απο αυτό των μεταβλητών και συναρτήσεων και χρησιμοποιείται παρόμοια, αλλά για τον έλεγχο τύπων. Εδώ κρατούνται πληροφορίες μόνο για τον τύπο που ορίζεται σε ένα σύμβολο. Ο λόγος που υπάρχει ξεχωριστός πίνακας συμβόλων για τους τύπους είναι για λόγους οργάνωσης αλλά και στο ότι ο ορισμός της γλώσσας επιτρέπει την δήλωση μεταβλητής και τύπου με το ίδιο όνομα. Για παράδειγμα το πιο κάτω πρόγραμμα είναι πλήρες έγγυρο.

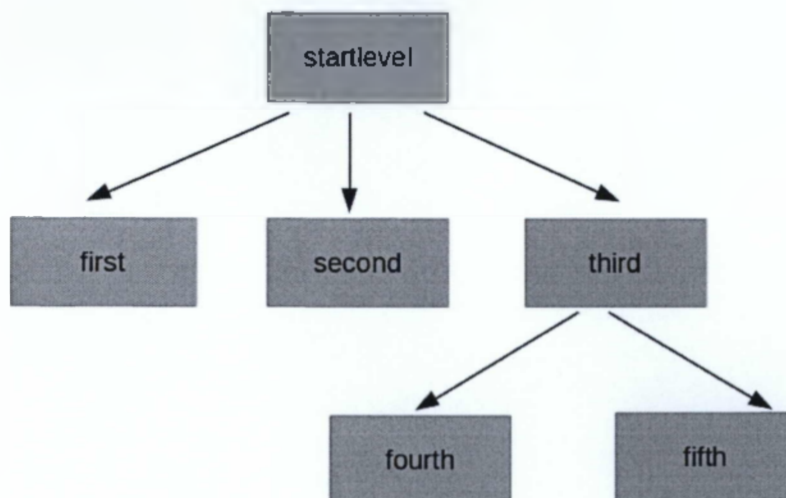
```
let
    type test := {a: int, b: int}
    var test := 5
in
    test
end
```

Επίπεδο μεταβλητών και συναρτήσεων Η πληροφορία αυτή αφορά κυρίως γλώσσες οι οποίες υποστηρίζουν ένθετες συναρτήσεις και μεταβλητές. Κάθε φορά που δηλώνετε μία συνάρτηση καταγράφετε το επίπεδο της. Το επίπεδο αυτό αυξάνετε κάθε φορά που δηλώνετε μία συνάρτηση μέσα σε μία άλλη συνάρτηση. Το επίπεδο δεν μπορεί να επεξηγηθεί καλύτερα από ένα παράδειγμα. Σχηματικά το επίπεδο αυξάνεται απο πάνω προς τα κάτω και δημιουργείται ένα δένδρο.

```
let
    var i := 8
    function first(a: int) : int =
        a + 8
    function second(a: int) : int =
        let var k := 8
```

```
            var f := 7
        in k + f + a end
    function third(a: int) : int =
        let var j := 4
            var o := 3
            function fourth(a: int) : int =
                a + 8 + i + j
            function fifth(a: int) : int =
                a + 4 + i
         in
             fourth(5) + j + o + a
         end
in
    ()
end
```

Τα first, second και third επίπεδα προέρχονται από κοινό πατέρα και έτσι τα τρία αυτά επίπεδα έχουν πρόσβαση σε όλα τα δεδομένα του αρχικού επιπέδου. Τα δεδομένα όμως που δημιουργήθηκαν στα πιο πάνω επίπεδα δεν είναι προσβάσιμα από άλλα επίπεδα, ένα επίπεδο για να έχει πρόσβαση σε δεδομένα άλλου πρέπει να είναι απόγονος του. Για παράδειγμα τα επίπεδα forth και fifth έχουν πρόσβαση στα επίπεδα third και startlevel. Γενικά ένα επίπεδο για να έχει πρόσβαση σε πιο χαμηλό επίπεδο πρέπει να είναι απόγονος του.



Σχήμα 5.1: Σχηματική αναπαράσταση των επιπέδων

Ένθετοι βρόχοι και break Κάθε φορά που εκτελείται ένα βρόχος καταγράφετε το βάθος του. Δηλαδή αν έχουμε δύο ένθετους βρόχους το βάθος θα είναι δύο. Η πληροφορία αυτή χρειάζεται για την break. Η break προκαλεί έξοδο από την πιο εσωτερική έκφραση ενός βρόχου οπότε πρέπει να ξέρουμε σε πιο βάθος βρόχου βρισκόμαστε. Η λογική των ένθετων βρόχων και break γίνεται αναδρομικά ως εξής.

1. Η σημασιολογική ανάλυση του μέρους ελέγχου ενός βρόχου γίνεται με τις παλιές πληροφορίες.

2. Πριν τη σημασιολογική ανάλυση του κορμιού ενός βρόχου αυξάνεται το βάθος κατά ένα και δημιουργείται μία επιγραφή(label) η οποία θα τοποθετηθεί στο τέλος του βρόχου ούτως ώστε να χρησιμοποιεί ως έξοδος διαφυγής όταν συναντηθεί μία break.

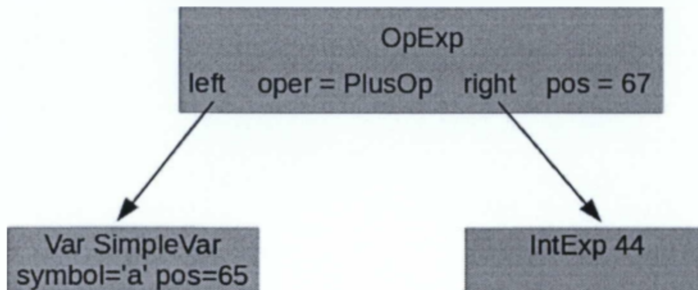
3. Κατά την σημασιολογική ανάλυση του κορμιού αν συναντηθεί μία break γίνεται άλμα στην βαθύτερη επιγραφή η οποία βαθύτερη είναι αυτή η οποία είχε περαστεί στην συάρτηση αναδρομής τελευταία φορά. Έτσι επιτυγχάνεται η έξοδος από την πιο εσωτερική έκφραση ενός βρόχου.

5.2.2 Τύποι δεδομένων

Τύπος είναι το νόημα που δίνουμε στα δεδομένα που μοιράζονται κοινά χαρακτηριστικά και τρόπο αποθήκευσης στη μνήμη, αλλά επίσης και όλες οι διαδικασίες που επιτρέπονται στους τύπους αυτούς. Ο λόγος που φτιάχνουμε τύπους είναι για την μείωση των σημασιολογικών λαθών. Η γραφή κώδικα χωρίς τύπους (όπως γίνεται στη γλώσσα μηχανής) χρειάζεται τεράστια τεκμηρίωση κώδικα για το νόημα των δεδομένων καθιστώντας τη γραφή και αποσφαλμάτωση κώδικα εξαιρετικά δύσκολη και χρονοβόρα.

Έλεγχος τύπων Η tiger έχει ισχυρό έλεγχο τύπων. Με αυτό τον όρο εννοούμε πως ο κάθε τύπος πρέπει να χρησιμοποιείται με ίδιους τύπους. Οι καινούργιοι τύποι που δημιουργούνται στην tiger είναι πάντα ξεχωριστοί τύποι ακόμα και αν έχουν τα ίδια πεδία. Ο έλεγχος τύπων των εκφράσεων γίνεται αναδρομικά πάνω στο συντακτικό αφηρημένο δένδρο από την συνάρτηση transExp. Για παράδειγμα στο σχήμα 1 έχουμε ένα απόσπασμα από ένα αφηρημένο συντακτικό δένδρο το οποίο δηλώνει πως πρέπει να προστεθούν η μεταβλητή a με το 44.

Στο παρακάτω απόσπασμα κώδικα η διαδικασία ελέγχου γίνεται ως εξής. Στην πρώτη γραμμή αναλύεται η έκφραση ως πράξη τελεστή. Στη δεύτερη γραμμή περνιέται αναδρομικά η αριστερή έκφραση και στην τρίτη γραμμή η δεξιά έκφραση. Το αποτέλεσμα τους θα είναι ένα κομμάτι δένδρου ενδιάμεσης γλώσσας το οποίο αποθηκεύεται στην μεταβλητή exp και ο τύπος της έκφρασης ο οποίος αποθηκεύεται στην μεταβλητή ty. Ας υποθέσουμε ότι η μεταβλητές ty είναι τύποι int. Στην πέμπτη γραμμή ελέγχεται τι ίδιους πράξη είναι(πρόσθεση,αφαίρεση ...), για όλους τους τελεστές επιτρέπονται μόνο ακέραιοι



Σχήμα 5.2: Αφηρημένο συντακτικό δένδρο μιας έκφρασης τελεστή πρόσθεσης

εκτός της ισότητας η οποία μπορεί να χρησιμοποιηθεί για όλους τους τύπους. Στο παράδειγμα μας έχουμε τελεστή πρόσθεσης οπότε όταν εκτελεστεί η συνάρτηση checkInts και θα γίνει έλεγχος αν οι δύο τύποι είναι ακέραιοι. Αν περάσει τον έλεγχο τότε θα κτιστεί ένα μεγαλύτερο κομμάτι δένδρου ενδιαμέσου κώδικα το οποίο θα περιέχει τις δύο exp. Αν δεν περάσει τον έλεγχο τότε θα τυπωθεί το μήνυμα "Operators must be integer" και θα τερματιστεί με σφάλμα η σημασιολογική ανάλυση.

```
| trexp (A.OpExp{left, oper, right, pos}) =
let val {exp = expl, ty = tyleft} = trexp(left)
    val {exp = expr, ty = tyright} = trexp(right)
in
  if oper=A.PlusOp orelse
     oper=A.MinusOp orelse
           . . . . .
  then if checkInts(tyleft,tyright)
           then {exp = Tr.opExp(oper,expl,expr),
                 ty = Types.INT}
           else (error pos "Operators must be integers";
                   {exp=Tr.dummy, ty = Types.INT})
  else if checkEqual(tyleft,tyright)
           then {exp = Tr.opExp(oper,expl,expr),
                 ty = Types.INT}
           else (error pos "the two operants must
                 have the same type, or must be
                 record-nil comparison";
             {exp=Tr.dummy, ty = Types.INT})
end
```

Στην περίπτωση της ισότητας ελέγχονται αν οι τιμές δείχνουν στο ίδιο αντικείμενο. Στα αλφαριθμητικά για να γίνει έλεγχος ισότητας με βάση το περιεχόμενο χρησιμοποιείται η συνάρτηση strEqual από την καθιερωμένη βιβλιοθήκη.

Ο έλεγχος έγκυρων μεταβλητών γίνεται αναδρομικά πάνω στο συντακτικό αφηρημένο δένδρο από την συνάρτηση trvar. Στο παράδειγμα μας όταν έρθει η περίπτωση της μεταβλητής a τότε καλείται η συνάρτηση trvar η οποία ψάχνει στο περιβάλλον για το αναγνωριστικό a. Αν το βρει επιστρέφει τον τύπο του και το κομμάτι του δένδρου ενδιάμεσου κώδικα. Στην περίπτωση εγγραφών ελέγχετε πρώτα ότι το πεδίο που ζητήθηκε υπάρχει.

Ο έλεγχος τύπων των δηλώσεων γίνεται αναδρομικά πάνω στο συντακτικό αφηρημένο δένδρο από την συνάρτηση transDec. Για παράδειγμα στο παρακάτω πρόγραμμα θα γίνουν οι εξείς έλεγχοι.

```
let var a : int := 3
      function test (b : int ) : int =
          b + 9
in
    test(7)
end
```

Θα εκτελεστεί η συνάρτηση transDec για την περίπτωση της δήλωσης μεταβλητής μακράς μορφής. Αρχικά θα ελεγχθεί ο τύπος του δεξιού μέρους, όπως φαίνεται στο πιο κάτω απόσπασμα (στην τρίτη γραμμή). Μετά θα ελεγχθεί αν υπάρχει ο τύπος ο οποίος δηλώσαμε για την μεταβλητή. Αν υπάρχει τότε θα ελεγχθεί ο τύπος του δεξιού μέρους αν είναι ο ίδιος με αυτό που δηλώσαμε και θα αποθηκεύετεί στο περιβάλλον ο τύπος μαζί με την τοποθεσία του. Αν δεν υπάρχει ο τύπος που δηλώσαμε τότε θα τυπωθεί το μήνυμα σφάλματος "unbound type: <type>". Αν δεν συμφωνούν οι δύο τύποι τότε θα τυπωθεί το μήνυμα σφάλματος "type specifier does not much expression type".

```
| transDec (venv, tenv, A.VarDec{name, escape,
            typ=SOME(id,posId), init, pos},
                   lev, expslist, breaklabel) =

let
  val {exp = e, ty = ty} = transExp(venv, tenv,
         init, lev, breaklabel, expslist)
  (* the ty's field name, of the expression,
     conflicts with the VarEntry's field name *)
  val ty_exp = ty
in
  (case Symbol.look(tenv, id) of
    SOME ty =>
          let val aty = ty
          in
            if checkEqual(aty, ty_exp)
```

```
            then let
              val acc = Tr.allocLocal lev (!escape)
                  val venv' = Symbol.enter(venv, name,
                            Env.VarEntry{access = acc, ty = aty
                               })
                  val var = Tr.assign(Tr.simpleVar(acc,lev),e)
              in
                ({tenv = tenv, venv = venv'},lev,expslist@[var])
              end
        else (error posId "type specifier does
                    not much expression type";
                 ({tenv = tenv, venv = venv},lev,expslist))
         end
  | NONE => (error posId ("unbound type: '" ^
                          Symbol.name id ^ "'");
            ({tenv = tenv, venv = venv},lev,expslist)))
end
```

Ο έλεγχος των συναρτήσεων είναι λίγο πιο περίπλοκος. Ο έλεγχος τύπων γίνεται με την εξής σειρά.

1. Ελέγχονται αν όλοι οι τύποι των παραμέτρων υπάρχουν.
2. Ελέγχεται αν ο επιστρεφόμενος τύπος υπάρχει.
3. Ενημερώνεται το παρόν περιβάλλον με το όνομα της συνάρτησης το οποίο θα περιέχει, τον επιστρεφόμενο τύπο και όλα τα ζευγάρια <όνομα,τύπος> των παραμέτρων.
4. Εκτελείται το σώμα της συνάρτησης και ελέγχεται ο παραγώμενος τύπος της ταιριάζει με τον επιστρεφόμενο τύπο.

Όλα τα παραπάνω υλοποιούνται στη περίπτωση συναρτήσεων της συνάρτησης transDec.

```
| transDec (venv, tenv, A.FunctionDec funs,
            lev, expslist, breaklabel) =
            .
            .
            .
```

5.3 Στατικός έλεγχος vs Δυναμικό έλεγχο

Οι πιο πάνω σημασιολογικοί έλεγχοι γίνονταν στατικά (στατικός έλεγχος). Δηλαδή τα λάθη εντοπίζονταν την ώρα που τρέχει ο μεταγλωττιστής. Υπάρχουν κάποια λάθη τα

οποία δεν μπορούν να εντοπιστούν στατικά (η δεν είναι βολικό) αλλά εντοπίζονται με ελέγχους όταν τρέχει ένα πρόγραμμα. Δηλαδή έχουμε δυναμικό έλεγχο.

Δυναμικός έλεγχος γίνεται συνήθως στην εξαγωγή/εισαγωγή στοιχείου ενός πίνακα. Το πιο κάτω πρόγραμμα μας δείχνει γιατί είναι αδύνατον να εντοπιστεί ένας δείκτης εκτός ορίου (out of bounds).

```
let type intArray = array of int
    var ar := intArray[5] of 0
    var i := ord(getchar())
in
    ar[i] := 45
end
```

Στην δεύτερη γραμμή δημιουργούμε ένα πινάκα με πέντε ακεραίους που έχουν αρχικοποιηθεί με μηδέν. Η μεταβλητή i θα περιεχέι ένα ακέραιο τον οποίο πήραμε από μια είσοδο (getchar()) σαν αλφαριθμητικό και τον μετατρέψαμε σε ακέραιο (ord). Στην πέμπτη γραμμή βλέπουμε ότι εισάγουμε την τιμή 45 στη θέση i. Ο σημασιολογικός αναλυτής μας εγγυάστε ότι το i είναι ακέραια τιμή (λόγο του ότι το ord επιστρέφει ακέραια τιμή) αλλά αν η τιμή του i είναι μέσα στα όρια του πίνακα δεν μπορούμε να το ξέρουμε! Πρέπει ελεγχθεί την ώρα που τρέχει το πρόγραμμα, ότι ο χρήστης εισήγαγε μια τιμή από 0 έως 4. Αυτό σημαίνει πως στην παραγωγή κώδικα θα παραχθεί κώδικας ο οποίος ελέγχει. i < 5 και i >= 0. Στην έκφραση a[i] γίνεται στατικός αλλά και δυναμικός έλεγχος.

Πιο κάτω βλέπουμε τον κώδικα του ar[i] όπως παράγεται σε ενδιάμεση γλώσσα.

```
CJUMP(LE,
 CONST 5,
 TEMP t117,
 L3,L4)
LABEL L4
CJUMP(LT,
 TEMP t117,
 CONST 0,
 L3,L5)
LABEL L5
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t116,
   BINOP(MUL,
    TEMP t117,
    CONST 4))),
 CONST 45)
MOVE(
 TEMP t100,
 CONST 0)
```

```
...
LABEL L3
EXP(
 MEM(
  CONST 0))
```

Βλέπουμε ότι ο εικονικός καταχωρητής 117 ελέγχετε αν είναι μικρότερος η ίσος με το 5 και αν δεν είναι τότε ελέγχετε αν είναι μικρότερος του μηδέν, αν δεν είναι τότε προσκομίζεται η τιμή του ar[i]. Ο λόγος που οι έλεγχοι είναι ανάποδα (το σωστό λάθος και αντίθετα) είναι ότι οι ΚΜΕ δεν κάνουν άλμα (jump) όταν η διακλάδωση είναι λάθος. Οπότε στις δύο περιπτώσεις ελέγχου δεν θα κάνει άλμα (επειδή το σωστό είναι λάθος). Αυτό γίνεται γιατί τα άλματα στη γλώσσα μηχανής κοστίζουν πολύ[7] και συνήθως ο δείκτης θα είναι μέσα στα όρια αφού το να μην είναι αποτελεί την εξαίρεση.

Η προσκόμιση τιμής εκτός ορίου αντιμετωπίζεται ως εξής. Αν μια από τους δυο ελέγχους είναι σωστή τότε κάνει άλμα στο label l3 και ανακαλεί τα δεδομένα στη θέση μνήμης 0x00000000 το οποίο εγείρει πάντα την εξαίρεση “μνήμης που δεν μας ανήκει” (segmentation fault).

5.4 Παραγωγή ενδιάμεσου κώδικα

Η παραγωγή ενδιάμεσου κώδικα είναι η φάση η οποία μεταφράζει το αφηρημένο συντακτικό δένδρ σε δένδρο αφηρημένης γλώσσας μηχανής.

Ο σκοπός της είναι κυρίως για δύο λόγους: Μετασχηματισμός της γλώσσας Α σε μια βολική (παρόμοια) γλώσσα με την Γ για την καλύτερη επεξεργασία της από το πισινό μέρος. Λόγο στη περίπτωση μας του ότι η γλώσσα Γ θα είναι γλώσσα μηχανής η ενδιάμεση γλώσσα θα είναι της μορφής ψευδό-γλώσσα μηχανής.

Χρησιμοποιείται σαν φορητή γλώσσα για την γραφή ενός και μόνου πισινού μέρους. Παράδειγμα: Θέλω να μεταφράσω τρεις γλώσσες σε άλλες δύο γλώσσες (αρχιτεκτονικές υπολογιστών). Εχω τις γλώσσες Α, Β και Γ και θέλω να τις μεταφράσω στις γλώσσες Δ και Ε. Χωρίς την ενδιάμεση γλώσσα θα χρειαζόταν να γράψω τα ακόλουθα εμπρόσθια και πισινά μέρη. Το Α σε Δ και Α σε Ε, το Β σε Δ και Β σε Ε, το Γ σε Δ και Γ σε Ε. Άρα παράγονται τρία εμπρόσθια μέρη στα οποία δημιουργούνται για το καθένα δύο ξεχωριστά πισινά μέρη. Βλέπουμε ότι η αναλογία $n * m$ όπου n είναι το εμπρόσθιο μέρος και m το πισινό μέρος . Η αξία της ενδιάμεσης γλώσσας φαίνεται παρακάτω. Για κάθε γλώσσα χρειάζεται να μεταφραστεί σε μία και μόνο γλώσσα, την ενδιάμεση. Άρα παράγονται τρία εμπρόσθια και δύο πισινά μέρη και επικοινωνούν με την ενδιάμεση γλώσσα η αναλογία αυτομάτως γίνεται $n + m$!

Στην περίπτωση μας ο ενδιάμεσος κώδικας επιβάλλεται να μοιάζει με γλώσσα μηχανής. Πρέπει να έχει την λογική των γλωσσών μηχανής αλλά επίσης και να είναι βολική

για την δημιουργία της από το αφηρημένο συντακτικό δένδρο.

Πιο συγκεκριμένα πρέπει να έχει τις παρακάτω ιδιαιτερότητες:

1. Πρέπει να είναι βολική για να παραχθεί από το αφηρημένο δένδρο στη φάση της σημασιολογική ανάλυσης

2. Πρέπει να είναι βολική για την μετάφραση της σε πραγματική γλώσσα μηχανής, για όλες τις επιθυμητές μηχανές.

3. Κάθε κατασκευή πρέπει να έχει ένα καθαρό και απλό νόημα, έτσι οι βελτιστοποιημένοι μετασχηματισμοί της ενδιάμεσης γλώσσας που θα ξαναγραφτούν να μπορούν εύκολα να καθοριστούν και να υλοποιηθούν.

Στην tiger η παραγωγή ενδιάμεσου κώδικα γίνεται με μια γκάμα συναρτήσεων όπου οι υπογραφές τους βρίσκονται στο αρχείο translate.sig και η υλοποίηση τους στο translate.sml. Οι συναρτήσεις παίρνουν ως είσοδο κομμάτια αφηρημένου συντακτικού δένδρου και το μετασχηματίζουν. Το δένδρο ενδιάμεσου κώδικα δημιουργείται ακριβώς με τον ίδιο τρόπο όπως το αφηρημένο δένδρο, φτιάχνεται ένα κομμάτι κάθε φορά το οποίο δεσμεύεται σε ένα άλλο. Ο μετασχηματισμός γίνεται αναδρομικα πάνω στο αφηρημένο συντακτικό δένδρο από αριστερά προς στα δεξιά. Στα φύλλα του δένδρου υπάρχουν οι απλούστερες εκφράσεις και μεταφράζονται σε ενδιάμεσο κώδικα και επιστρέφονται ούτως έτσι να κτιστούν οι πιο περίπλοκες εκφράσεις.

Παρακάτω περιγράφεται ένα παράδειγμα πως ο σημασιολογικός αναλυτής μετατρέπει το αφηρημένο συντακτικό δένδρο σε δένδρο ενδιάμεσου κώδικα. Έστω ότι έχουμε το παρακάτω απόσπασμα κώδικα tiger.

```
if 5 > 2
then 7
else 5
```

Πιο κάτω βλέπουμε την υπογραφή και υλοποίηση της συνάρτησης cond2Exp η οποία μετασχηματίζει μία ifExp σε ενδιάμεσο κώδικα και θα χρησιμοποιηθεί από τον σημασιολογικό αναλυτή για τον μετασχηματισμό της. Για κάθε σημασιολογική έκφραση υπάρχει υλοποιημένη η αντίστοιχη συνάρτηση που κάνει τον μετασχηματισμό.

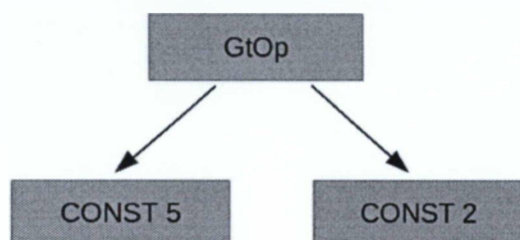
```
(* exp1: the condition,
   exp2: then expression,
   exp3: else expression *)
val cond2Exp : exp * exp * exp -> exp

fun cond2Exp (e1,e2,e3) =
    let val r = Temp.newtemp()
        val t = Temp.newlabel() and
            f = Temp.newlabel() and
            fin = Temp.newlabel()
```

```
    in
        Ex (
        T.ESEQ(
        T.SEQ(unCx e1(t,f),
        T.SEQ(T.LABEL t,
        T.SEQ(T.MOVE(T.TEMP r, unEx e2),
        T.SEQ(T.JUMP(T.NAME fin,[fin]),
        T.SEQ(T.LABEL f,
        T.SEQ(T.MOVE(T.TEMP r, unEx e3),
        T.SEQ(T.JUMP(T.NAME fin,[fin]),
              T.LABEL fin))))))), T.TEMP r))
    end
```

Αρχικά, στη σημασιολογική ανάλυση βρισκόμαστε στη περίπτωση μίας έκφρασης if-then-else.

```
| trexp (A.IfExp{test, then',else' = SOME else', pos}) =
  let val {exp=e1, ty = test'} = trexp(test)
  in if checkInt(test')
   then let val {exp = e2, ty = exp1} = trexp(then')
            val {exp = e3, ty = exp2} = trexp(else')
        in
            if checkEqualIf(exp1, exp2)
            then {exp = Tr.cond2Exp(e1,e2,e3), ty = exp1}
            else (error pos "then and else should produce
              the same type"; {exp=Tr.dummy, ty = Types.INT})
        end
   else {exp=Tr.dummy, ty = Types.UNIT}
  end
```

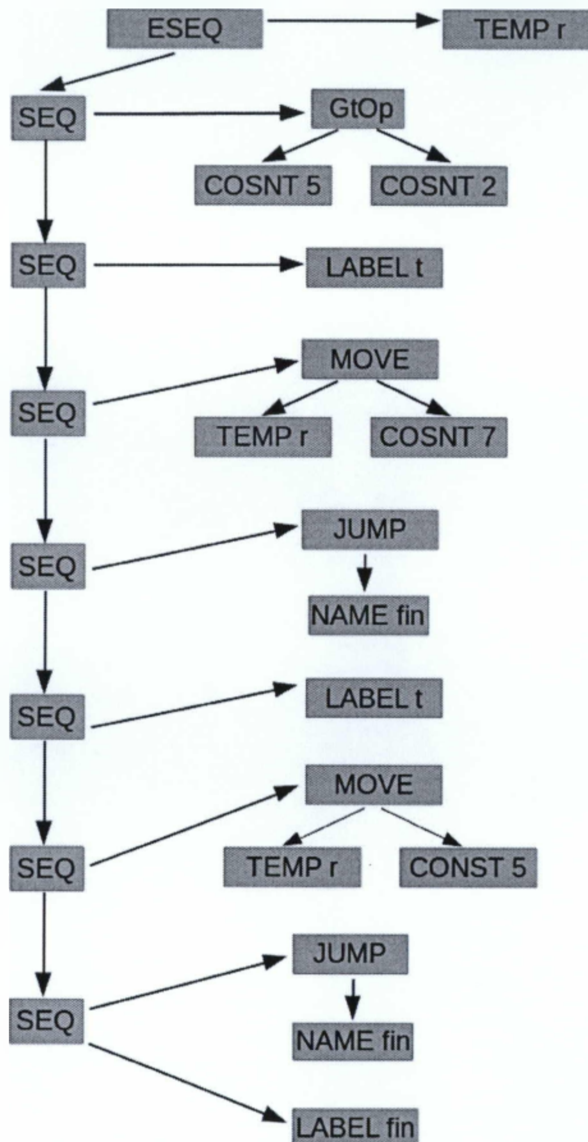
Στη δεύτερη γραμμή γίνεται έλεγχος τύπων στο κομμάτι του ελέγχου της if και επιστρέφεται το κομμάτι του ενδιάμεσου κώδικα (e1) μαζί με τον τύπο της έκφρασης (test'). Μετά στην τρίτη γραμμή ελέγχεται ο τύπος της έκφρασης αν είναι ακέραιος, αφού στην έκφραση ελέγχου της if επιτρέπονται μόνο ακέραιες τιμές σύμφωνα με τον ορισμό της tiger. Στην τέταρτη και πέμπτη γραμμή γίνεται έλεγχος τύπων στην έκφραση του then και του else και επιστρέφονται τα κομμάτια του ενδιάμεσου κώδικα μαζί με τον τύπο τους. Οι τύποι αυτοί ελέγχονται αν είναι οι ίδιοι και μετά τα κομμάτια που επέστρεψαν (e1, e2, e3), ενώνονται κατάλληλα και παράγουν ένα μεγαλύτερο δένδρο ενδιάμεσου κώδικα από την συνάρτηση μετασχηματισμού cond2Exp. Τα κομμάτια ενδιάμεσου κώδικα του ελέγχου (e1), της then (e2) και else (e3) παρουσιάζονται σχηματικά στα σχήματα 3, 4 και 5 αντίστοιχα. Στο σχήμα 6 φαίνεται σχηματικά το δένδρο ενδιάμεσου κώδικα της έκφρασης if-then-else μετά την ένωση τους από την συνάρτηση cond2Exp.



Σχήμα 5.3: Έκφραση τελεστή μεγαλύτερου

Σχήμα 5.4: Έκφραση σταθεράς 5

Σχήμα 5.5: Έκφραση σταθεράς 7



Σχήμα 5.6: Ολοκληρωμένο δένδρο ενδιάμεσου κώδικα της έκφραση if-the-else

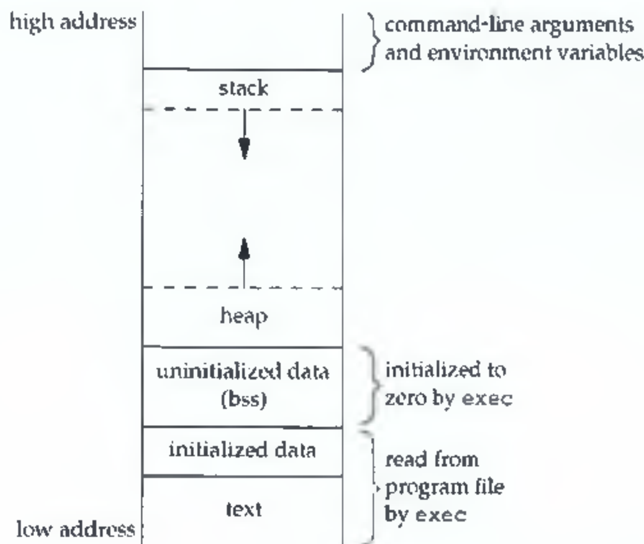
6.

Οργάνωση τρέχοντος προγράμματος και παραγωγή κώδικα

Ένα πρόγραμμα για να εκτελεστεί από τον υπολογιστή πρέπει να φορτωθεί στην κυρίως μνήμη του. Όταν δώσουμε εντολή να εκτελεστεί ένα πρόγραμμα το λειτουργικό σύστημα αναλαμβάνει να φορτώσει το πρόγραμμα στη μνήμη. Ένα πρόγραμμα χωρίζεται στη μνήμη σε τέσσερις τομείς.
Τον κώδικα του προγράμματος (text) που περιέχει τον εκτελέσιμο κώδικα.
Τα στατικά δεδομένα (initialized και unnitialized data) όπου αποθηκεύονται οι στατικές μεταβλητές[1] που όρισε ο προγραμματιστής.
Την στοίβα (stack) όπου ενεργοποιούνται οι συναρτήσεις και αποθηκεύονται τα δεδομένα των ενεργών συναρτήσεων.
Τον σωρό (heap) όπου αποθηκεύονται δεδομένα που δεν ξέρουμε εξαρχής το μέγεθος τους.
Οι παραπάνω πληροφορίες θα είχαν οργανωθεί όπως φαίνεται στο παρακάτω σχήμα.

Ένα πρόγραμμα tiger κάνει χρήση των τομέων text, stack, heap και initialized data. Στο κείμενο (text) είναι αποθηκευμένος ο κώδικας ο οποίος μεταφράστηκε από τον μεταγλωττιστή. Στη στοίβα (stack) ενεργοποιούνται οι συναρτήσεις και όλες οι τοπικές μεταβλητές των συναρτήσεων οι οποίες "ξεχνιούνται" όταν βγουν εκτός σκοπού. Η στοίβα μεγαλώνει από ψηλότερες σε χαμηλότερες διευθύνσεις. Αυτό γίνεται για να είναι οι συναρτήσεις της tiger συμβατές με άλλες συναρτήσεις άλλων γλωσσών αφού στις περισσότερες γλώσσες χρησιμοποιείται αυτή η σύμβαση για ιστορικούς λόγους. Η tiger καλεί συναρτήσεις που είναι γραμμένες σε C για την δέσμευση μνήμης των εγγραφών, πινάκων καθώς και όλων των συναρτήσεων της καθιερωμένης βιβλιοθήκης. Στο σωρό (heap) στη tiger αποθηκεύ-

[1] Στατικές μεταβλητές είναι οι μεταβλητές που η ζωή τους διαρκεί καθ όλη την διάρκεια του προγράμματος



Σχήμα 6.1: Τυπικός χάρτης μνήμης

ονται οι εγγραφές, πίνακες και χρησιμοποιείται επίσης από την καθιερωμένη βιβλιοθήκη της tiger. Η tiger διαχειρίζεται τον σωρό από την συνάρτηση malloc της C αφού η SML δεν διαθέτει δυνατότητες δέσμευσης δυναμικής μνήμης. Στα δεδομένα (data) αποθηκεύονται τυχόν αλφαριθμητικά μαζί με το μέγεθος τους.

Στο κείμενο βρίσκονται οι μεταφρασμένες (απο τον μεταγλωττιστή) συναρτήσεις στον συμβολικό κώδικα μηχανής ARM. Πιο κάτω βλέπουμε πως το κυρίως let ... in ... end ενός προγράμματος της tiger είναι ουσιαστικά και αυτό μια συνάρτηση (είναι η αντίστοιχη main για την γλώσσα C).

```
let
in
    print("Hello World\n")
end
```

Το πιο πάνω πρόγραμμα μεταταφράζεται απο τον μεταγλωττιστή της tiger ως εξής.

```
tigermain:              @ Έναρξη συνάρτησης
stmfd sp!, {fp, lr}     @ Αποθηκεύονται ο δείκτης πλαισίου(fp) και η διεύθυνση
    επιστροφής(lr)
add fp, sp, #4          @ Μετακινείται ο δείκτης πλαισίου κατά 4 bytes πιο ψηλά στη μνήμη
sub sp, sp, #80         @ Αφαιρείται ο δείκτης στοίβας κατά 80 bytes πιο χαμηλά στη μνήμη
str fp, [sp]            @ Αποθηκεύεται ο δείκτης πλαισίου στη στοίβα
L3:                     @ Επιγραφή L3
adr r0, L1              @ Φορτώνετε η διεύθυνση του L1 στον καταχωρητή r0
mov r0, r0              @ Αντιγράφεται το περιεχόμενο του r0 στο r0 (περιττή εντολή)
bl printT               @ Καλείται η συνάρτηση printT
```

```
mov r0, r0               @ Αντιγράφεται το περιεχόμενο του r0 στο r0 (περιττή εντολή)
mov r0, r0               @ Αντιγράφεται το περιεχόμενο του r0 στο r0 (περιττή εντολή)
b L2                     @ Άλμα στην επιγραφή L2
L2:                      @ Επιγραφή L2
sub sp, fp, #4           @ Μετακινείται ο δείκτης πλαισίου κατά 4 bytes πιο χαμηλά στη μνήμη
ldmfd sp!, {fp, pc}      @ Φορτώνονται πίσω στο δείκτη πλαισίου και στο μετρητή
                         @ προγράμματος το περιεχόμενο του δείκτη πλαισίου και η
                         @ διεύθυνση επιστροφής αντίστοιχα
L1:                      @ Επιγραφή L1
.word 13                 @ Μία λέξη η οποία δηλώνει το μέγεθος του αλφαριθμητικού
                         @ που ακολουθεί
.ascii "Hello World\n"   @ Το αλφαριθμητικό
```

Στα στατικά δεδομένα βρίσκονται μεταβλητές όπου έχουν διάρκεια ζωής από την αρχή μέχρι το τέλος του προγράμματος. Τα δεδομένα υπό-διερούνται σε δύο μέροι. Αν είναι αρχικοποιημένα τότε τοποθετούνται στην περιοχή των αρχικοποιημένων δεδομένων και τα μη στην περιοχή των μη αρχικοποιημένων (bss). Στο παρακάτω πρόγραμμα σε γλώσσα C, η μεταβλητή global θα είχε τοποθετηθεί στη περιοχή στατικών δεδομένων.

```
int global = 80;

int main()
{
    int a = 5;
    int c = a + global;
    return 0;
}
```

Στην tiger στον τομέα των στατικών δεδομένων αποθηκεύονται τα αλφαριθμητικά μαζί με τον μέγεθος τους. Η tiger αποθηκεύει το μέγεθος του αλφαριθμητικού στην αρχή και έτσι ξέρει που τερματίζει. Αυτό είναι μία εναλλακτική λύση αποθήκευσης αλφαριθμητικών από το τερματικό μηδέν που χρησιμοποιούν άλλες γλώσσες προγραμματισμού. Η υπεύθυνη συνάρτηση που δημιουργεί τον κώδικα των αλφαριθμητικών είναι η string και βρίσκεται στο αρχείο armframe.sml.

```
fun string (lab,s) =
  let val len = Int.toString(String.size s)
  in
    Symbol.name(lab)^":"^"\n"^
    ".word "^len^"\n"^
    ".ascii "^"\""^s^"\""^"\n"^
    ".align 4"^"\n"
  end
```

Όπως βλέπουμε πιο πάνω η συνάρτηση περιέχει ένα πρότυπο και ενσωματώνει το αλφαριθμητικό και το μέγεθος του μέσα στο πρότυπο.

Η στοίβα είναι η περιοχή όπου ζουν οι τοπικές μεταβλητές των συναρτήσεων. Οι τοπικές μεταβλητές μπορεί να παίρνουν διαφορετικές τιμές με κάθε κλήση της συνάρτησης ή μπορεί ακόμα να ενεργοποιούνται διαφορετικές αναλόγως με την κλήση της συνάρτησης. Έτσι δεν θα ήταν καθόλου βολικό όλες οι μεταβλητές να βρίσκονται στο χώρο των στατικών δεδομένων και να παίρνουν άσκοπα χώρο στη μνήμη. Η στοίβα μεγαλώνει κάθε φορά που καλείται μια συνάρτηση. Το πρότυπο της συνάρτησης βρίσκεται στον τομέα του κειμένου και κάθε φορά που καλείται η συνάρτηση ενεργοποιείται στη στοίβα (activation of record). Το πρότυπο ενεργοποίησης περιέχει πληροφορίες για το πώς να εκτελεστεί σωστά η συνάρτηση. Δεν υπάρχει κάποιο τέλειος σχεδιασμός για το σχεδιασμό ενός προτύπου. Είναι στην κρίση του προγραμματιστή να σχεδιάσει το πρότυπο ενεργοποίησης σύμφωνα με τις ανάγκες της γλώσσας προγραμματισμού. Πολλά πρότυπα όμως μοιράζονται κοινούς σχεδιασμούς.

Ο σωρός είναι ο χώρος στη μνήμη όπου μπορούμε να απαιτούμε από το λειτουργικό σύστημα να μας κατανέμει περισσότερη μνήμη για το πρόγραμμα καθώς τρέχει. Αυτό είναι βολικό γιατί κάποιες φορές δεν μπορούμε να ξέρουμε εκ το προτέρων το μέγεθος των δεδομένων. Για παράδειγμα ένα πρόγραμμα το οποίο παίρνει είσοδο δεν μπορούμε να προβλέψουμε πόσο μεγάλη θα είναι. Έτσι οι γλώσσες προγραμματισμού έχουν την δυνατότητα να δεσμεύσουν όση μνήμη χρειάζονται κατά την εκτέλεση του προγράμματος. Στην C ο χειρισμός της δυναμική δέσμευση και αποδέσμευσης γίνεται με τις συναρτήσεις malloc και free αντίστοιχα.

6.1 Παραγωγή συμβολικού κώδικα ARM από ενδιάμεσο κώδικα.

Η ARM είναι η αρχιτεκτονική η οποία συντριπτικά επικρατεί σε μικροσυσκευές, όπως έξυπνα τηλέφωνα, φορητούς υπολογιστές, ταμπλέτες κτλ. Η ARM βασίζεται στην αρχιτεκτονική RISC. Η RISC σε αντίθεση με την CISC είναι η στρατηγική σχεδιασμού της ΚΜΕ να εκτελεί μικρές και πολλές εντολές σε αντίθεση με σύνθετες και λίγες. Συγκεκριμένα η tiger παράγει κώδικα για τον επεξεργαστή ARM Cortex-A7MPCore. Ο ενδιάμεσος κώδικας περιγράφει μια γενική εικόνα όλων των αρχιτεκτονικών, όμως σε κάποια φάση ο ενδιάμεσος κώδικας θα πρέπει να μεταφραστεί σε αληθινή γλώσσα μηχανής. Η μετατροπή του ενδιάμεσου κώδικα σε κάποια γλώσσα μηχανής γίνεται σχετικά εύκολα με το αξίωμα του Maximal Munch.

6.1.1 Στοίβα κλήσεων

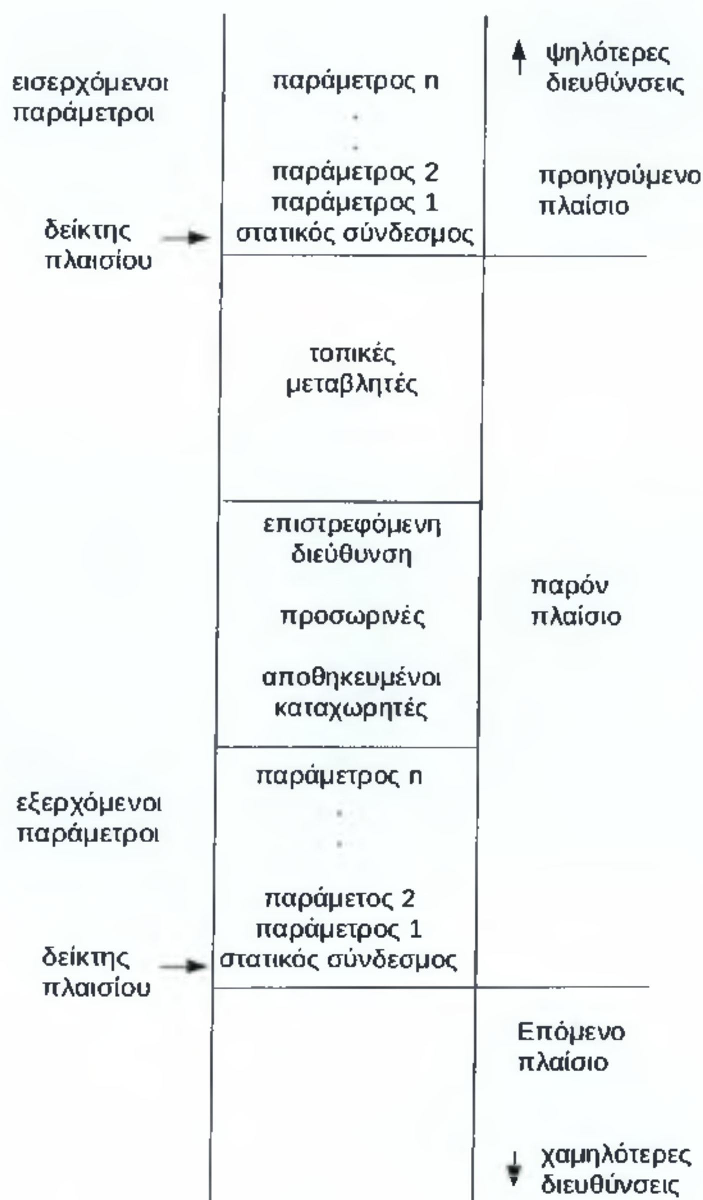
Η στοίβα κλήσεων είναι η δομή δεδομένων η οποία κρατάει πληροφορίες για τις ενεργές συναρτήσεις. Η στοίβα κλήσεων βρίσκεται στο χώρο μνήμης της στοίβας. Ένα πρόγραμμα tiger αποτελείται από πολλές συναρτήσεις οπότε κάθε φορά που καλείται μία

συνάρτηση αποθηκεύεται η κατάσταση της και δημιουργείται χώρος για την καινούργια συνάρτηση. Μία στοίβα κλήσεων αποτελείται απο πλαίσια στοίβας. Τα πλαίσια στοίβας είναι η δομή του χώρου που αποθηκεύονται οι πληροφορίες για την κάθε συνάρτηση. Κάθε φορά που δημιουργείται χώρος για μία καινούργια συνάρτηση, οι πληροφορίες θα αποθηκευτούν με βάση αυτό το πλαίσιο. Πολλές καινούργιες γλώσσες προγραμματισμού επιλέγουν να έχουν κοινή δομή πλαισίου με κάποια διάσημη γλώσσα προγραμματισμού για να επωφεληθούν από τις έτοιμες συναρτήσεις της. Άν το πλαίσιο στοίβας είναι το ίδιο τότε δεν υπάρχει κανένα πρόβλημα να καλείς συναρτήσεις απο άλλες γλώσσες. Η tiger χρησιμοποιεί συναρτήσεις από την γλώσσα C αλλά δεν έχει κοινό πλαίσιο. Όμως το πλαίσιο που δημιουργείται απο το πλαίσιο της C δεν επηρεάζει αυτό της tiger. Απλά διατηρείται μία σύμβαση όλες οι συναρτήσεις που γράφονται σε C να τερματίζουν με ένα κεφαλαίο T. Η υπεύθυνη συνάρτηση για την κλήση εξωτερικών συναρτήσεων είναι η externalCall που βρίσκεται στο αρχείο armframe.sml

```
fun externalCall (s,args) =
  let val _ = if K > List.length(args)
              then ()
              else print("Error: externalCall
                     in armframe.sml: "^s^"\n")
  in Tree.CALL(Tree.NAME(Temp.namedlabel(s^"T")), args) end
```

Το πλαίσιο των συναρτήσεων της tiger φαίνεται στο παρακάτω σχήμα.

Κάθε φορά που καλείται μια συνάρτηση γίνονται οι παρακάτω διαδικασίες. Η "συνάρτηση που καλεί"(caller function) μια άλλη συνάρτηση, περνάει τις πρώτες τέσσερις παραμέτρους στους πρώτους τέσσερις καταχωρητές. Από σύμβαση οι πρώτοι τέσσερις καταχωρητές χρησιμοποιούνται για αποθήκευση των παραμέτρων. Αν μια συνάρτηση έχει να μεταβιβάσει περισσότερες από τέσσερις παραμέτρους οι πρώτες τέσσερις περνιούνται στους καταχωρητές και οι υπόλοιπες στο χώρο μνήμης των εξερχομένων παραμέτρων. Παρατηρήστε ότι ο χώρος μνήμης που αποθηκεύονται οι παράμετροι στην tiger είναι κοντά στο καινούργιο πρότυπο που θα δημιουργηθεί. Αυτό γίνεται ώστε η "κληθέντα συνάρτηση"(calee function) να έχει εύκολη πρόσβαση στις παραμέτρους. Μετά την δημιουργία του καινούργιου προτύπου η κληθέντα συνάρτηση αποθηκεύει τις τιμές των καταχωρητών που βρίσκονται στις θέσεις r4-r10,r12 στο πρότυπο της, και αντιγράφει όλες τις παραμέτρους στους καταχωρητές r4-r10,r12. Όταν τελειώσει η κληθέντα συνάρτηση αποθηκεύει το αποτέλεσμα στον καταχωρητή r0 και επαναφέρει της τιμές των r4-r11 της "συνάρτησης που καλεί". Ουσιαστικά η "κληθέντα συνάρτηση" διαφυλάσσει τα δεδομένα της "συνάρτησης που καλεί", έτσι όταν επανέρθει ξανά ο έλεγχος στην "συνάρτηση που καλεί" να έχει όλους τους υπολογισμούς της άθικτους στους καταχωρητές r4-r11 συν το αποτέλεσμα της κληθέντας συνάρτησης στον καταχωρητή r0. Οι υπεύθυνες συναρτήσεις για την ενεργοποίηση των συνάρτήσεων είναι οι procEntryExit1 και procEntryExit3 που βρίσκονται στο αρχείο armframe.sml. Η procEntryExit1 απλά παράγει τον ενδιάμεσο κώδικα αποθήκευσης του αποτελέσματος μιας συνάρτησης στον κατάλληλο καταχωρητή. Η procEntryExit3 είναι υπεύθυνη για να παραγάγει τον κώδικα εισόδου και εξόδου μίας



Σχήμα 6.2: Πλαίσιο για την γλώσσα tiger

συνάρτησης. Στο πιο κάτω απόσπασμα κώδικα της συνάρτησης procEntryExit3 βλέπουμε πως στην είσοδο και έξοδο μιας συνάρτησης τοποθετείται ο συμβολικός κώδικας της μεταβλητής funProlog και funEpilog αντίστοιχα. Ο συμβολικός κώδικας αυτός δεσμεύει τον χώρο στη στοίβα για μια καινούργια συνάρτηση. Επίσης αποθηκεύει τις τιμές των καταχωρητών r4-r10, r12 και τον lr (διεύθυνση επιστροφής) στο καινούργιο πλαίσιο ούτως ώστε οι καταχωρητές αυτοί να μπορούν να χρησιμοποιηθούν από την καινούργια συνάρτηση. Όταν η συνάρτηση τελειώσει (βγει εκτός σκοπού) τότε αποθηκεύει τις παλιές τιμές των καταχωρητών από το πλαίσιο και αποδεσμεύει τον χώρο στη μνήμη. Η δέσμευση και αποδέσμευση μνήμης στη στοίβα γίνεται απλά με την μετακίνηση ενός δείκτη, του δείκτη στοίβας. Ο δείκτης στοίβας είναι συνέχεια αποθηκευμένος στον ειδικό καταχωρητή sp (stack pointer). Δεσμεύεται μνήμη εκχωρώντας σε αυτόν μία πιο χαμηλή διεύθυνση η αποδεσμεύεται μνήμη αποθηκεύοντας σε αυτόν μία πιο ψηλή διεύθυνση.

```
val funProlog =
  n'^":\n"^
  "sub fp, fp, #"^Int.toString(numlocals)^"\n"^
  "stmdb fp, {r4-r10,r12,lr}\n"^
  "mov fp, sp\n"^
  "sub sp, sp, #80\n"^
  "str fp, [sp]\n"

val funEpilog =
  "mov sp, fp\n"^
  "ldr fp, [sp]\n"^
  "sub r3, fp, #"^Int.toString(numlocals)^"\n"^
  "ldmdb r3, {r4-r10,r12,pc}\n"
```

6.1.2 Maximal Munch

Ο Maximal Munch[2] ψάχνει σε μια είσοδο μια σειρά στοιχείων που να είναι τα ίδια με κάποιο πρότυπο μέχρι το πρότυπο να εξαντληθεί. Στη περίπτωση μας ψάχνουμε στο δένδρο ενδιάμεσου κώδικα πρότυπα τα οποία μεταφράζονται σε έγκυρες εντολές ARM. Ψάχνουμε πρώτα το μεγαλύτερο δυνατό ταίριασμα, καταλήγοντας στο μικρότερο δυνατό. Προσπαθώντας να βρούμε πρώτα το μεγαλύτερο ταίριασμα είμαστε σίγουροι πως δεν θα μας ξεφύγει κάποιο μεγάλο ταίριασμα μίας εντολής, με κάποιο μικρότερο που τυχένει να ταιριάζει μέσα στο μεγάλο.

Κάθε φορά που ταιριάζεται ενα πρότυπο ενδίαμεσου κώδικα που αντιστοιχεί σε μία εντολή ARM κρατούντε διάφορες πληροφορίες σε μια δομή (τύπο) instr που ορίζεται στο αρχείο assem.sml. Όλες οι δομές αυτές κρατόνται σε μία λίστα.

```
datatype instr =
    OPER of {assem: string,
                        dst: temp list,
```

```
                              src: temp list,
                              jump: label list option}
  | LABEL of {assem: string, lab: Temp.label}
  | MOVE of {assem: string,
                              dst: temp,
                              src: temp}
```

Όπως βλέπουμε πιο πάνω, ο τύπος instr μπορεί να δημιουργηθεί με τρεις κατασκευαστές. OPER(artion) για εντολές λειτουργίας, LABEL για εντολές επιγραφών και move για εντολές που κάνουν εκχωρήσεις. Ποιος κατασκευαστής θα χρησιμοποιηθεί σε κάθε ταίριασμα εξαρτάται από την φύση της εντολής που θα φτιαχτεί. Ο τύπος OPER κρατάει τέσσερις πληροφορίες, στο πεδίο assem αποθηκεύονται οι εντολές που επιλέχτηκαν από το αξίωμα maximal munch. Παράδειγμα ο ενδιάμεσος κώδικας πρόσθεσης δύο ακεραίων T.BINOP(T.PLUS,e1,e2) μεταφράζεται σε συμβολικό κώδικα ως add <καταχωρητής>, <καταχωρητής>, <καταχωρητής> . Λόγου του ότι ακόμα δεν ξέρουμε ακριβώς ποιοι καταχωρητές θα χρησιμοποιηθούν αποθηκεύονται στα πεδία src και dst εικονικοί καταχωρητές. Στο πεδίο src αποθηκεύονται καταχωρητές που χρησιμοποιούνται σε μια πράξη ενώ στο πεδίο dst καταχωρητές που θα γίνουν εκχωρήσεις σε αυτούς. Δηλαδή στην εντολή add έχουμε δύο καταχωρητές που χρησιμοποιούνται για την πρόσθεση δύο καταχωρητών και ένα για να αποθηκευτεί το αποτέλεσμα.

```
| munchExp (T.BINOP(T.PLUS,e1,e2)) =
  result (fn r =>
           emit(A.OPER{assem="add `d0, `s0, `s1\n",
                       src=[munchExp e1,munchExp e2],
                       dst=[r], jump=NONE}))
```

Στο πεδίο jump αποθηκεύεται αν μία εντολή πού κάνει άλμα. Άλματα υπάρχουν σε εντολές που αλλάζουν την ροή ενός προγράμματος. Ο τύπος LABEL κρατάει δύο πληροφορίες. Στο πεδίο assem αποθηκεύεται το όνομα της επιγραφής και το lab αποθηκεύεται η αριθμητική αναπαράσταση της επιγραφής. Ο τύπος MOVE κρατάει τρις πληροφορίες. Στο πεδίο assem αποθηκεύεται το όνομα της εντολής και στα src και dst αποθηκεύονται πληροφορίες ίδιες με αυτές του τύπου OPER. Παρατηρούμε ότι δεν υπάρχει πεδίο jump αφού σε μια εντολή μετακίνισης δεν υπάρχει περίπτωση να γίνεται άλμα. Ο λόγος που υπάρχει ο κατασκεύαστής MOVE είναι γιατί αυτές οι εντολές έχουν ιδική μεταχήρηση απο τον κατανεμητή καταχωρητών. Για παράδειγμα μπορούν μία εντολή να αφαιρεθεί αν αντιγράφει τον ίδιο καταχωρητή, mov r0, r0 .

Πιο κάτω βλέπουμε ένα απόσπασμα κώδικα από την υλοποίηση του maximal munch που βρίσκεται στο αρχείο παραγωγής κώδικα armgen.sml.

```
(* store a variable in a record field
   e.g record.field := 7 *)
| munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,T.TEMP t,
                  T.BINOP(T.MUL,T.CONST i1,T
                  .CONST i2))), e)) =
```

```
  emit(A.OPER{assem="str `d0, [`s0, #"^i2s(i1*i2)^"]"^"\n",
                        dst=[munchExp e], src=[t], jump=NONE})
                              .
                              .
                              .

(* storing a variable in a record field, array index *)
| munchStm(T.MOVE(T.MEM(
          T.BINOP(T.PLUS,T.TEMP t,T.CONST i)), e)) =
  emit(A.OPER{assem="str `s0, [`s1, #"^i2s(i)^"]"^"\n",
                        src=[munchExp e, t],
                        dst=[], jump=NONE})
```

Η πρώτη περίπτωση χρησιμοποιείται για την εκχώρηση μιας μεταβλητής σε ένα πεδίο μιας εγγραφής. Η δεύτερη περίπτωση χρησιμοποιείται για την εκχώρηση των αρχικών τιμών στα πεδία εγγραφών και εκχώρηση μιας μεταβλητής σε πίνακες. Η πρώτη περίπτωση είναι πιο ειδική απο την δεύτερη λόγο του ότι η πρώτη προσπαθεί να ταιριάξει εννιά κόμβους ενό η δεύτερη έξι. (T.MOVE(T.MEM(T.BINOP(T.PLUS,T.TEMP t, ...)). Στην υλοποίηση υλοποιούνται όλοι οι δυνατοί συνδυασμοί μετατροπής ενδιάμεσου κώδικα σε συμβολικό κώδικα μηχανής ARM.

Εικονικοί καταχωρητές είναι συμβολικοί/εικονικοί καταχωρητές που παράγει η φάση της μετάφρασης ενδιάμεσου κώδικα σε συμβολικό κώδικα. Για παράδειγμα στη πιο κάτω η εντολή πρόσθεσης δύο αριθμών έχουν παραχθεί στη θέση των καταχωρητών τιμές όπως 'd0 και 's0 αντί r0, r1 κτλ.

```
add 'd0, 's0, 's1
```

Οι εικονικοί καταχωρητές d(estination) <αριθμός> δηλώνουν καταχωρητές προορισμού-/εκχώρησης τιμών. Οι εικονικοί καταχωρητές s(ource) <αριθμός> δηλώνουν καταχωρητές πηγής/χρήσης τιμών. Ο λόγος που παράγονται εικονικοί καταχωρητές αντί κανονικών είναι επειδή δεν ξέρουμε ακόμα πόσοι καταχωρητές θα χρειαστούν. Το πόσοι και ποιοι καταχωρτητές θα τοποθετηθούν σε κάθε εντολή αποφασίζεται από τις αναλύσεις της φάσης κατανομής καταχωρητών. Όταν γίνουν οι ανάλογες αναλύσεις τότε αντικαθιστώνται οι τιμές 'd0 και 's0 με κανονικούς καταχωρητές. Το πρότυπο και η υλοποίηση των εικονικών καταχωρητών βρίσκεται στο αρχείο temp.sig και temp.sml αντίστοιχα. Οι εικονικοί καταχωρητές είναι ακέραιες τιμές.

```
type temp = int
val temps = ref 100
fun newtemp() = let val t = !temps
                in
                  temps := t+1; t
                end
```

6.1.3 Βελτιστοποίηση ενδιάμεσου κώδικα.

Η πρώτη βελτιστοποίηση που κάνει η tiger βρίσκεται πάνω στον ενδιάμεσο κώδικα, τέτοιες βελτιστοποιήσεις συναντώνται και σε εμπορικούς μεταγλωττιστές Η βελτιστοποίηση ουσιαστικά εκμεταλλεύεται τις ειδικότερες περιπτώσεις του ενδιάμεσου κώδικα παράγοντας όσο το δυνατό πιο συγκεκριμένο κώδικα.

Για παράδειγμα το πιο κάτω πρόσθεση

```
5 + 6
```

μεταφράζεται στην ενδιάμεση γλώσσα ως,

```
BINOP(PLUS,
      CONST 4,
      CONST 5))
```

Η περίπτωση η οποία χειρίζεται το πιο πάνω κώδικα είναι η ακόλουθη.

```
| munchExp (T.BINOP(T.PLUS,e1,e2)) =
  result (fn r => emit(A.OPER{assem="add `d0,
          `s0, `s1\n",src=[munchExp e1,
          munchExp e2], dst=[r], jump=NONE}))
```

Η περίπτωση αυτή παίρνει την αριστερή και δεξιά έκφραση της πρόσθεσης και ξανάκαλεί την συνάρτηση munchExp στη τρίτη γραμμή με αποτέλεσμα να αναλυθούν στην απλούστερη τους μορφή. Ως αποτέλεσμα παράγεται ο ακόλουθος συμβολικός κώδικας.

```
mov r0, 4          @ Φορτώνει τον αριθμό 4 στον καταχωρητή r0
mov r1, r0         @ Αντιγράφει το περιεχόμενο του καταχωρητή r0
                   @ στον καταχωρητή r1
mov r0, 5          @ Φορτώνει τον αριθμό 5 στον καταχωρητή r0
add r0, r0, r1     @ Προσθέτει τα περιεχόμενα των καταχωρητών r0 και r1 και
                   @ αποθηκεύει το αποτέλεσμα στον καταχωρητή r0
```

Παρατηρούμε πως χρειάστηκαν τέσσερις εντολές για την εκτέλεση της πράξης. Η πιο πάνω πράξη μπορεί να εκτελεστεί με μία και μόνο εντολή! Αυτό γίνεται αφού μπορούμε εκ των προτέρων να υπολογίσουμε την πρόσθεση. Στο παρακάτω απόσπασμα κώδικα βλέπουμε ότι αν στην πρόσθεση βρίσκονται αριθμοί, προσθέτονται και απλά γράφουμε το αποτέλεσμα σε έναν καταχωρητή.

```
| munchExp (T.BINOP(T.PLUS,T.CONST i1,T.CONST i2)) =
  result (fn r => emit(A.OPER{assem="ldr `d0, ="
          ^ i2s(i1+i2) ^ "\n",
          src=[], dst=[r], jump=NONE}))
```

Το αποτέλεσμα είναι μία εντολή εκχώρησης.

```
ldr r0, =9
```

6.2 Κατανομή καταχωρητών

Μετά το τέλος των πιο πάνω φάσεων έχει παραχθεί κανονικά κώδικας σε γλώσσα μηχανής. Όμως υπάρχει υπάρχει ένα πρόβλημα. Οι πιο πάνω φάσεις υποθέτουν ότι η ΚΜΕ έχει άπειρους καταχωρητές. Κατά τη φάση της παραγωγής του ενδιάμεσου κώδικα για κάθε μεταβλητή θα δημιουργηθεί ένας εικονικός καταχωρητής. Δηλαδή αν σε ένα μπλοκ κώδικα χρειάζονταν 7 καταχωρητές για την λειτουργία του και η ΚΜΕ έχει 6 τότε οι καταχωρητές δεν αρκούν και τα δεδομένα θα πρέπει να αποθηκεύονται στην μνήμη ώστε να αποδεσμεύονται καταχωρητές για την επεξεργασία άλλων δεδομένων.

Οι παλιοί μεταγλωττιστές αποθήκευαν όλα τα δεδομένα στη στοίβα και τα προσκόμιζαν όταν χρειάζονταν έστω και αν υπήρχαν ελεύθεροι καταχωρητές. Αυτή η οργάνωση διαχείρισης δεδομένων ονομάζεται "μηχανές στοίβας" (stack machines)[4]. Το πλεονέκτημα είναι η απλή κατασκευή διαχείρισης δεδομένων. Στη πράξη όμως η τεχνική αυτή αποδείχτηκε αργή λόγο του ότι η μεταφορά εντολών από την κυρίως μνήμη είναι αργή σε σχέση με τους καταχωρητές της ΚΜΕ. Έτσι αναπτύχθηκαν ΚΜΕ με πολλούς καταχωρητές. Οι ΚΜΕ που είναι σχεδιασμένες με την RISC αρχιτεκτονική συνήθως δεν έχουν πρόβλημα έλλειψης καταχωρητών (σε συνδιασμό με ένα καλό αλγόριθμο κατανεμητή καταχωρητών) γιατί έχουν πολλούς καταχωρητές. RISC είναι η αρχιτεκτονική επεξεργαστών οι οποίοι εκτελούν μικρές εντολές σε λίγο χρόνο και έχουν πολλούς καταχωρητές, ο ARM Cortex-A7MPCore στον οποίο παράγει κώδικα η tiger έχει δεκαέξι καταχωρητές που μπορεί να χρησιμοποιήσει ο προγραμματιστής. Η υλοποίηση της κατανομής καταχωρητών χρειάζεται αρκετές αναλύσεις για να υλοποιηθεί. Η βελτιστοποίηση αυτή δεν γίνετε πάνω στον ενδιάμεσο κώδικα αλλά στους εικονικούς καταχωρητές που έχουν δημιουργηθεί από την σημασιολογική ανάλυση σε ενδιάμεση γλώσσα. Ο μεταγλωττιστής της tiger για κάθε μπλοκ κώδικα (προσπαθεί να) βρει τους λιγότερο δυνατούς καταχωρητές που χρειάζονται για να εκτελεστεί. Στις επόμενες παραγράφους θα περιγραφούν οι αναλύσεις για να αντιμετωπιστεί το πιο πάνω πρόβλημα κατανομής καταχωρητών.

Γράφος ροής ελέγχου (Control flow graph) Είναι η ανάλυση στην οποία φτιάχνουμε ένα γράφο ο οποίος αναπαριστά την ροή ενός προγράμματος. Για παράδειγμα σε μια if else το πρόγραμμα μπορεί να πάρει δύο διαφορετικά μονοπάτια αναλόγως με τα δεδομένα του βρόγχου ελέγχου. Ο γράφος ροής ελέγχου υλοποιείται στο αρχείο makegraph.sml.

Η συνάρτηση instrs2graph από το αρχείο makegraph.sml παίρνει σαν είσοδο μία λίστα με τιμές τύπου instr οι οποίες δημιουργήθηκαν από την παραγωγή συμβολικού κώδικα ARM από το αρχείο armgen.sml. Αρχικά η instrs2graph δημιουργεί ένα γράφο. Το πρότυπο του γράφου βρίσκεται στο αρχείο graph.sig και η υλοποίηση του στο αρχείο graph.sml. Μετα δημιουργείται ο αρχικός κόμβος και προσπελάζεται η λίστα από την ένθετη συνάρτηση iter. Για κάθε στοιχείο στη λίστα αναλύεται απο ποιον κατασκευαστή, κατασκευάστηκε. Γενικά στην προσπέλαση αποθηκεύονται οι πληροφορίες των πεδίων src, dts, αν είναι εντολή μετακίνησης (move), οι επιγραφές και οι τοποθεσίες των αλμάτων για κάθε κόμβο του γράφου. Στο πιο κάτω απόσπασμα κώδικα φαίνεται η επιλογή

του κατασκευαστή για την κάθε μεταβλητή instr τύπου instr.

```
case instr of
 Assem.OPER {assem=a,src=slst,dst=dlst,jump=j} =>
 (case j of
      SOME jumplist =>
           ...
    | NONE =>
           ...
    | Assem.MOVE {assem=a,src=s,dst=d} =>
           ...
    | Assem.LABEL{assem=a,lab=l} =>
           ...
```

Για τον κατασκευαστή OPER που είναι αληθή το πεδίο των αλμάτων αποθηκεύονται οι πληροφορίες των πεδίων src, dst, move ως ψευδής και οι τοποθεσίες αλμάτων (jump). Μετά δημιουργείται ένας καινούργιος κόμβος στον γράφο και συνεχίζεται η προσπέλαση της λίστας. Αν είναι ψευδή το πεδίο αλμάτων τότε αποθηκεύονται όλα τα παραπάνω αλλά δεν δημιουργείται καινούργιος κόμβος.

Για τον κατασκευαστή MOVE αποθηκεύονται οι πληροφορίες των πεδίων src και dst και η move ως αληθής.

Για τον κατασκευαστή LABEL αποθηκεύονται οι ετικέτες.

Μετά καλείται η ένθετη συνάρτηση makeEdges η οποία συνδέει του κόμβους που δημιουργήθηκαν χρησιμοποιώντας την πληροφορία των αλμάτων. Η συνάρτηση makeEdges προσπελάζει την λίστα των κόμβων και για κάθε άλμα του κόμβου, τον συνδέει με τους αντίστοιχους κόμβους.

Ανάλυση ροής δεδομένων Είναι η ανάλυση η οποία μας δίνει πληροφορίες σε διάφορα σημεία ενός προγράμματος. Η ανάλυση ροής δεδομένων χρησιμοποιεί τον γράφο ροής ελέγχου για να αντλήσει τις πληροφορίες . Η ανάλυση ροής δεδομένων είναι γενικός όρος και υπάρχουν διάφορες αναλύσεις ροής δεδομένων. Συγκεκριμένα η ανάλυση ροής δεδομένων που χρησιμοποιείται είναι η ανάλυση των ενεργών μεταβλητών (liveness analysis). Η ανάλυση ενεργών μεταβλητών μας δίνει πληροφορίες για το ποιες μεταβλητές είναι ενεργές σε ένα σημείο στο πρόγραμμα. Για Παράδειγμα, στην πράξη a = b + c, στην εκχώρηση της μεταβλητής a οι μεταβλητές b και c είναι ενεργές. Δηλαδή με αυτή την πληροφορία συμπεραίνουμε πως χρειάζονται δύο καταχωρητές για την εκτέλεση της εκχώρησης. Η ανάλυση ενεργών μεταβλητών υλοποιείται στο αρχείο liveness.sml.

Ο αλγόριθμος της ανάλυσης ενεργών μεταβλητών είναι ο ακόλουθος.

- Το n εναι ο κόμβος.
- Το use και def είναι τα αντίστοιχα src και dst της ανάλυσης ροής δεδομένων.

- Το use[n] και def[n] κάνουν αντιστοιχία n κόμβου με τους εικονικούς καταχωρητές (μεταβλητές) του κόμβου.
- Το pred είναι οι εικονικοί καταχωρητές (μεταβλητές) των προκάτοχων κόμβων. Δηλαδή το pred[n] επιστρέφει όλους τους εικονικούς καταχωρητές των προκατόχων κόμβων του κόμβου n.
- live-in είναι ο εικονικός καταχωρητής (μεταβλητή) που είναι ενεργός κατά την είσοδο του κόμβου.
- live-out είναι ο εικονικός καταχωρητής (μεταβλητή) που είναι ενεργός κατά την έξοδο του κόμβου.

1. Αν μια μεταβλητή είναι μέσα στο use[n], τότε είναι live-in στο κόμβο n. Δηλαδή, αν μία δήλωση χρησιμοποιεί μία μεταβλητή, η μεταβλητή είναι ενεργή κατα την είσοδο της στην δήλωση.
2. Αν μία μεταβλητή είναι live-in σε ένα κόμβο n, τότε είναι live-out σε όλους τους κόμβους στην αντιστοιχία pred[n].
3. Αν μία μεταβλητή είνα live-out σε ένα κόμβο n, και όχι στο def[n], τότε η μεταβλητή είναι επίσης live-in στον n. Δηλαδή, αν κάποιος χρειάζεται την τιμή του a στο τέλος της δήλωσης

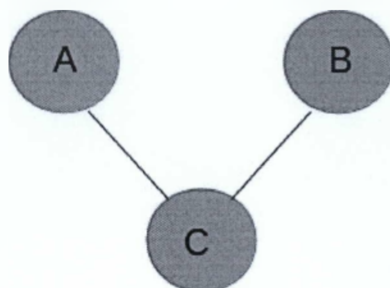
Το αρχείο liveness.sml περιέχει την συνάρτηση liveness η οποία παίρνει ένα γράφο ροής δεδομένων και επιστρέφει μία συνάρτηση η οποία παίρνει είσοδο ένα κόμβο και επιστρέφει μία λίστα live-out εικονικών καταχωρητών. Η συνάρτηση liveness καλεί την συνάρτηση livenessalgo η οποία υλοποιεί τον πιο πάνω αλγόριθμο.

Το πρώτο και τρίτο βήμα τα οποία παράγουν το αποτέλεσμα του συνόλου live-in για τον κόμβο n, υλοποιούνται ως η ένωση του συνόλου use και συνόλου differ όπου το σύνολο differ περιέχει το συμπλήρωμα του συνόλου def στο σύνολο out.

```
val differ = Set.difference(out',def)
val in' = IGraph.Table.enter(intablesets,node,Set.union(use,
   differ))
```

Το δεύτερο βήμα το οποίο παράγει το αποτέλεσμα του συνόλου live-out για τον κόμβο n υλοποιούνται ως η εκχώρηση τιμών του συνόλου in όλων των διάδοχων κόμβων του n.

```
val succnodeslist = Flow.Graph.succ node
fun succins (succnode, set) =
  (case IGraph.Table.look(intablesets,succnode) of
           SOME inset => Set.union(set,inset)
         | NONE => raise Impossible)
val succinset = foldl succins Set.empty succnodeslist
val out' = IGraph.Table.enter(outtablesets,node,succinset)
```



Σχήμα 6.3: Εικονική αναπαράσταση γράφου παρεμβολής

Ο πιο πάνω κώδικας στη πρώτη γραμμή προσκομίζει όλους τους διάδοχους κόμβους m στη μεταβλητή succnodeslist. Οι διάδοχοι κόμβοι m ενός κόμβου n είχαν παραχθεί από την ανάλυση ροή δεδομένων. Στο σύνολο succinset αποθηκεύονται όλες οι τιμές όλων των διαδόχων κόμβων m. Τέλος στην μεταβλητή out' αποθηκεύεται η αντιστοιχία κόμβου n με το σύνολο succinset.

Γράφος παρεμβολών καταχωρητών Ο γράφος παρεμβολών φτιάχνεται μετά την ανάλυση ενεργών μεταβλητών. Σκοπός του γράφου αυτού είναι για να ξέρουμε πόσοι καταχωρητές χρειάζεται να είναι ενεργοί σε ένα σημείο του προγράμματος. Κάθε κόμβος του γράφου αναπαριστάνει ένα καταχωρητή και κάθε σύνδεση του με άλλο κόμβο δηλώνει με ποιον άλλο καταχωρητή είναι ενεργός. Για παράδειγμα στο πιο κάτω σχήμα η μεταβλητή c είναι ενεργή στις μεταβλητές a και b.

Ο γενικός αλγόριθμος κατασκευής του γράφου είναι ο εξής.

1. Σε οποιαδήποτε εντολή η οποία ορίζει μία μεταβλητή a, όπου οι live-out μεταβλητές είναι οι b_1, ..., b_j, πρόσθεσε ακμές παρεμβολής (a, b_1), ..., (a, b_j).

2. Σε μια εντολή μετακίνησης(move) $a \leftarrow c$, όπου οι μεταβλητές b_1, ..., b_j είναι οι live-out, πρόσθεσε ακμές παρεμβολής (a, b_1), ..., (a, b_j) για όποια b_i η οποία δεν είναι ίδια με την c.

Ο γράφος υλοποιείται στο αρχείο color.sml απο την συνάρτηση build. Η build προσπελάζει όλους του κόμβους κώδικα και εκτελεί τον πιο πάνω αλγόριθμο. Η build για να ενώσει τις ακμές παρεμβολής καλεί την συνάρτηση addEdge. Η build επιστρέφει μια

αντιστοιχία adjList εικονικού καταχωρητή με ένα σύνολο εικονικών καταχωρητών, δηλαδή τους καταχωρητές που γειτνιάζει. Ένα σύνολο γειτνίασης adjSet το οποίο απαντάει αν ένας καταχωρητής γειτνιάζει με άλλο. Η adjList αντιστοιχία απαντάει γρήγορα ποιοι καταχωρητές γειτνιάζουν ενώ το σύνολο adjSet απαντάει γρήγορα αν γειτνιάζει με κάποιο συγκεκριμένο καταχωρητή.

```
fun defsI (d,(adjList,degree,adjSet)) =
  let val _ = initial := d::(!initial)
      fun lives (l,(adjList,degree,adjSet)) =
        (initial := l::(!initial);
         addEdge(d,l,adjList,degree,adjSet))
  in
    Set.foldl lives (adjList,degree,adjSet) live''
  end
val (adjList',degree',adjSet') =
                                      Set.foldl defsI
                                      (adjList,degree,adjSet)
                                       defset
```

Στο πιο πάνω απόσπασμα κώδικα της συνάρτησης build εκτελείται το πρώτο μέρος του αλγορίθμου. Στην ένατη γραμμή προσπελάζεται το σύνολο defset το οποίο περιέχει τις μεταβλητές a κάθε εκάστοτε κόμβου κώδικα. Στην συνάρτηση defsI αποθηκεύεται η κάθε μεταβλητή d(ef) σε μία λίστα initial. Μετά καλείται η συνάρτηση lives και προσπελάζεται τις τιμές του συνόλου live''. Το σύνολο live'' περιέχει όλες τις τιμές live-out του εκάστοτε κόμβου κώδικα. Στην συνάρτηση lives αποθηκεύονται οι τιμές του live'' στην λίστα initial και καλείται η συνάρτηση addEdge η οποία προσθέτει τις παρεμβολές (a, b_1), ... (a, b_j).

```
val (moveList',worklistMoves',live') =
  (case ismove of
    true =>
      let val live' = Set.difference(live,useset)
          val defunionuse = Set.union(useset,defset)
          fun addl (item,moveList) =
            let val defunionuse' = case Temp.Table.look(moveList,item)
                of
                                        SOME i => Set.union(i,
                                           defunionuse)
                                      | NONE => defunionuse
                val defunionuse'' = Set.difference(defunionuse',
                                           Set.singleton(item))
            in Temp.Table.enter(moveList,item,defunionuse'') end
          val moveList' = Set.foldl addl moveList defunionuse
          val worklistMoves' = Set.union(worklistMoves,defunionuse)
          in (moveList',worklistMoves',live') end
  | false => (moveList,worklistMoves,live))
```

Στο πιο πάνω απόσπασμα κώδικα της εκτελείται το δεύτερο κομμάτι του αλγορίθμου. Αν είναι εντολή μετακίνησης τότε γίνοντε οι εξής υπολογισμοί. Στην μεταβλητή

defunionset αποθηκεύονται όλες οι τιμές του εκάστοτε κόμβου κώδικα. Στην μεταβλητή moveList' αποθηκεύεται το αποτέλεσμα της προσπέλασης όλων των στοιχείων του συνόλου defunionset όπου το κάθε στοιχείο περνιέται στην συνάρτηση addI. Στη συνάρτηση addI και συγκεκριμένα στη μεταβλητή defunionuse" προσθέτουντε όλες οι b_i(defunionuse') οι οποίες δεν είναι ίδιες με την c(item). Τέλος η συνάρτηση αποθηκεύει στον πίνακα συσχέτισης την αντιστοιχία c(item) με τις τιμές b_i.

Κατανομή καταχωρητών με τον αλγόριθμο απλοποίησης Ο αλγόριθμος απλοποίησης χρησιμοποιεί τον γράφο παρεμβολών για να βρει πόσοι καταχωρητές χρειάζονται για ένα μπλοκ κώδικα. Η κατανομή καταχωρητών γενικά είναι η διαδικασία με την οποία εισάγουμε ένα μεγάλο αριθμό μεταβλητών σε ένα συγκεκριμένο αριθμό καταχωρητών της ΚΜΕ. Πολλοί προγραμματιστές έχουν την διαίσθηση ότι μπορούν αυθαίρετα να δημιουργούν όσες μεταβλητές θέλουν. Στην πραγματικότητα όμως στην διάρκεια της μεταγλώττισης τρέχουν αλγόριθμοι κατανομής καταχωρητών για την κατανομή (η μη) των διαθέσιμων καταχωρητών της ΚΜΕ. Σε αρχιτεκτονικές με λίγους καταχωρητές η ανάλυση της κατανομής καταχωρητών γίνεται κρίσιμη γιατί αν δεν μπορεί να υπολογιστεί μια εντολή με τους υπάρχοντες καταχωρητές τότε πρέπει τα δεδομένα και τα αποτελέσματα να αποθηκεύονται στην μνήμη. Συγκεκριμένα η κατανομή καταχωρητών που υλοποιείται στην tiger θα βρει τους λιγότερο δυνατούς καταχωρητές που χρειάζονται για κάθε εντολή.

Ο αλγόριθμος δίνεται ως εξής.

Έχουμε n καταχωρητές και ένα γράφο με $p_1, \ldots, p_m$ κόμβους.

Αν οι ακμές ενός κόμβου pi είναι $n - 1$ τότε μπορούμε να τον αφαιρέσουμε απο τον γράφο, να τον βάλουμε σε μία λίστα και να ενημερώνουμε τον γράφο. Αν δεν είναι τότε συνεχίζουμε με τον επόμενο κόμβο μέχρι τον τελευταίο. Συνεχίζουμε την πιο πάνω διαδικασία μέχρι να εξαλειφτούν όλοι οι κόμβοι. Αν δεν υπάρχει κόμβος που να έχει ακμές $n - 1$ τότε ο αλγόριθμος της απλοποίησης αποτυγχάνει και η μεταγλώττιση τερματίζει. Μετά παίρνουμε την λίστα και χρωματίζουμε τους κόμβους με ξεχωριστό χρώμα. Ο αριθμός των χρωμάτων θα είναι ίσος με αυτός των καταχωρητών. Με τον πιο πάνω ευρετικό αλγόριθμο μπορούμε να ζωγραφίσουμε τον γράφο με ξεχωριστά χρώματα. Κάθε χρώμα αναπαριστά και ένα καταχωρητη της ΚΜΕ. Ο αλγόριθμος υλοποιείται από τις συναρτήσεις simplify, decrementDegree και assignColors στο αρχείο color.sml.

Η συνάρτηση simplify και decrementDegree εκτελεί το πρώτο μέρος του αλγορίθμου. Η simplify προσπελάζει τους κόμβους και ψάχνει για $n - 1$ και η decrementDegree αφαιρεί τον κόμβο από τον γράφο και ενημερώνει τον γράφο. Η συνάρτηση assignColors εκτελεί το δεύτερο μέρος του αλγορίθμου. Χρωματίζει την παραγόμενη λίστα απωθώντας (popping) ένα ένα τους κόμβους.

```
fun chooseNode nodes =
  case nodes of
      n::ns => if mapDegree(degree,n) >= K
               then chooseNode(ns) (* Try other node *)
               else n
```

```
| nil => raise RegisterOverflow
```

Στην πιο πάνω ένθετη συνάρτηση της συνάρτησης simplifiy προσπελάζει αναδρομικά όλους τους κόμβους. Επιλέγει τον πρώτο κόμβο όπου οι ακμές του είναι ίσοι οι περισσότεροι απο τους καταχωρητές Κ. Μετά την επιλογή του καλείται η συνάρτηση decrementDegree για να ενημερωθούν οι ακμές όλων των γειτονικών κόμβων. Δηλαδή η συνάρτηση decrementDegree καλείται για κάθε κόμβο του γράφου και η παράμετρος m είναι ο κόμβος.

```
fun decrementDegree (m,(simplifyWorklist,freezeWorklist,
                        degree,spillWorklist,activeMoves,worklistMoves,
                        moveList,coalescedNodes,selectStack)) =
  let val d = mapDegree(degree,m)
      val degree' = Temp.Table.enter(degree,m,d-1)
  in (simplifyWorklist,freezeWorklist,degree',
      spillWorklist,activeMoves,worklistMoves,
      moveList,coalescedNodes,selectStack)
  end
```

Αρχικά αποθηκεύεται πόσες ακμές έχει ο κόμβος στην μεταβλητή d και αφαιρείται κατά ένα και αποθηκεύεται το αποτέλεσμα στον στον πίνακα αντιστοιχίας.

```
let val adjListn = mapAdjList(adjList,n)
    fun diffcolors(w,okColor) =
      let val g = getAlias(w,alias,coalescedNodes)
          val cp = Set.union(coloredNodes,precolored)
      in if Set.exists (fn x => if g = x then true else false) cp
         then let val mapc = mapColor(color,g)
              in Set.difference(okColor,Set.singleton mapc) end
         else okColor
    end
   val okColors = Set.fromList(Frame.registers)
   val okColors' = Set.foldl diffcolors okColors adjListn
   in if Set.isEmpty okColors'
      then iter(selectStack,Set.add(spilledNodes,n),coloredNodes,color)
      else let val SOME(c,okColors'') = List.getItem(Set.listItems
         okColors')
              val color' = Temp.Table.enter(color,n,c)
              val coloredNodes' = Set.add(coloredNodes,n)
              val okColors''' = Set.fromList okColors''
         in iter(selectStack,spilledNodes,coloredNodes',color') end
end
```

Το πιο πάνω απόσπασμα κώδικα είναι απο την συνάρτηση assignColors. Για να εκτελεστεί η συνάρτηση assignColors σημαίνει ότι έγινε με επιτυχία η φάση της απλοποίησης και όλοι οι κόμβοι είναι σε μία λίστα selectStack. Απο την λίστα αυτή προσπελάζεται ο κάθε κόμβος n. Αρχικά αποθηκεύεται στην μεταβλητή (λίστα) adjListn όλοι οι κόμβοι που γειτνιάζουν με τον κόμβο n. Μετά αποθηκεύεται στη μεταβλητή okColors ο αριθμός των καταχωρητών που υπάρχουν για την συγκεκριμένη αρχιτεκτονική και στη συνέχεια εκτελείται για κάθε γειτονικό κόμβο η συνάρτηση diffcolors. Η diffcolors βασικά φιλτράρει

όλα τα χρώματα όπου έχουν χρησιμοποιηθεί για τον κόμβο αυτό και το αποτέλεσμα αποθηκεύεται στο σύνολο χρωμάτων okColors'. Στη συνέχεια αν δεν είναι άδειο το σύνολο okColors' τότε επιλέγεται ένα χρώμα απο τον σύνολο okColors' και με αυτό χρωματίζεται ο κόμβος. Επίσης προσθέτεται στο σύνολο των χρωματισμένων κόμβων το οποίο χρησιμοποιείται από την συνάρτηση diffcolors.

Παραρτήματα

A Κώδικας λεκτικής ανάλυσης

Πρόγραμμα 1: **tiger.lex**

```
type pos = int
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult  = (svalue,pos) token

val lineNum = ErrorMsg.lineNum
val linePos = ErrorMsg.linePos
fun err(p1,p2) = ErrorMsg.error p1

val commentsClosed = ref true
val stringClosed = ref 0
val str = ref ""

fun eof() =
    if not (!commentsClosed)
    then (ErrorMsg.error (!lineNum) ("unmatch comment"); Tokens.EOF(!
       lineNum,!lineNum))
    else if !stringClosed <> 0
    then (ErrorMsg.error (!lineNum) ("unmatch string"); Tokens.EOF(!
       lineNum,!lineNum))
    else Tokens.EOF(!lineNum,!lineNum)

%%

%header (functor TigerLexFun(structure Tokens : Tiger_TOKENS));
%s COMMENT STRING ESC FORMAT;
ID = [A-Za-z][_A-Za-z0-9]*;
digit = [0-9];
oct = {digit}{3};
newline = [\n\r];
ws = [ \t];
esqchar = ([nt\\\"] | "^c");
formatchar = ({newline} | {ws});
```

```
%%

<INITIAL, COMMENT>{newline}  => (lineNum := !lineNum+1; linePos :=
    yypos :: !linePos; continue());
<INITIAL>{ws}         => (continue());
<INITIAL>"type"       => (Tokens.TYPE(yypos,yypos+4));
<INITIAL>"var"        => (Tokens.VAR(yypos,yypos+3));
<INITIAL>"function"   => (Tokens.FUNCTION(yypos,yypos+8));
<INITIAL>"break"      => (Tokens.BREAK(yypos,yypos+5));
<INITIAL>"of"         => (Tokens.OF(yypos,yypos+2));
<INITIAL>"end"        => (Tokens.END(yypos,yypos+3));
<INITIAL>"in"         => (Tokens.IN(yypos,yypos+2));
<INITIAL>"nil"        => (Tokens.NIL(yypos,yypos+3));
<INITIAL>"let"        => (Tokens.LET(yypos,yypos+3));
<INITIAL>"do"         => (Tokens.DO(yypos,yypos+2));
<INITIAL>"to"         => (Tokens.TO(yypos,yypos+2));
<INITIAL>"for"        => (Tokens.FOR(yypos,yypos+3));
<INITIAL>"while"      => (Tokens.WHILE(yypos,yypos+5));
<INITIAL>"else"       => (Tokens.ELSE(yypos,yypos+4));
<INITIAL>"then"       => (Tokens.THEN(yypos,yypos+4));
<INITIAL>"if"         => (Tokens.IF(yypos,yypos+2));
<INITIAL>"array"      => (Tokens.ARRAY(yypos,yypos+5));
<INITIAL>"/*"         => (YYBEGIN COMMENT; commentsClosed := false;
    continue());
<COMMENT>"*/"         => (YYBEGIN INITIAL; commentsClosed := true;
    continue());
<COMMENT>.            => (continue());


<STRING>"\""          => (YYBEGIN INITIAL;
                           stringClosed := !stringClosed - 1;
                           let val temp = !str
                               val _ = str := ""
                           in
                               Tokens.STRING(temp,yypos,yypos)
                           end);

<STRING>"\\"          => (YYBEGIN ESC; continue());
<STRING>{newline}     => (ErrorMsg.error yypos ("strings can not be
    splited to separate lines " ^
                           yytext ^ " [use backslash for splitting string
                               ]"); continue());
<STRING>.             => (str := !str ^ yytext; continue());

<ESC>{esqchar}        => (YYBEGIN STRING; str := !str ^ "\\" ^ yytext;
    continue());
<ESC>{formatchar}     => (YYBEGIN FORMAT; continue());
<ESC>{oct}            => (YYBEGIN STRING; str := !str ^ String.str(Char.
```

```
    chr(valOf(Int.fromString yytext))); continue());
<ESC>.                  => (ErrorMsg.error yypos ("illegal escape
    character " ^ yytext); continue());

<FORMAT>{formatchar} => (continue());
<FORMAT>"\\"            => (YYBEGIN STRING; continue());
<FORMAT>.               => (ErrorMsg.error yypos ("illegal character " ^
    yytext ^
                           " [use backslash for splitting string]");
                           continue());

<INITIAL>"\""           => (YYBEGIN STRING;
                           stringClosed := !stringClosed + 1;
                           continue());
<INITIAL>":="           => (Tokens.ASSIGN(yypos,yypos+2));
<INITIAL>"|"            => (Tokens.OR(yypos,yypos+1));
<INITIAL>"&"            => (Tokens.AND(yypos,yypos+1));
<INITIAL>">="           => (Tokens.GE(yypos,yypos+2));
<INITIAL>">"            => (Tokens.GT(yypos,yypos+1));
<INITIAL>"<="           => (Tokens.LE(yypos,yypos+2));
<INITIAL>"<"            => (Tokens.LT(yypos,yypos+1));
<INITIAL>"<>"           => (Tokens.NEQ(yypos,yypos+2));
<INITIAL>"="            => (Tokens.EQ(yypos,yypos+1));
<INITIAL>"/"            => (Tokens.DIVIDE(yypos,yypos+1));
<INITIAL>"*"            => (Tokens.TIMES(yypos,yypos+1));
<INITIAL>"-"            => (Tokens.MINUS(yypos,yypos+1));
<INITIAL>"+"            => (Tokens.PLUS(yypos,yypos+1));
<INITIAL>"."            => (Tokens.DOT(yypos,yypos+1));
<INITIAL>"}"            => (Tokens.RBRACE(yypos,yypos+1));
<INITIAL>"{"            => (Tokens.LBRACE(yypos,yypos+1));
<INITIAL>"]"            => (Tokens.RBRACK(yypos,yypos+1));
<INITIAL>"["            => (Tokens.LBRACK(yypos,yypos+1));
<INITIAL>")"            => (Tokens.RPAREN(yypos,yypos+1));
<INITIAL>"("            => (Tokens.LPAREN(yypos,yypos+1));
<INITIAL>";"            => (Tokens.SEMICOLON(yypos,yypos+1));
<INITIAL>":"            => (Tokens.COLON(yypos,yypos+1));
<INITIAL>","            => (Tokens.COMMA(yypos,yypos+1));

<INITIAL>{ID}           => (Tokens.ID(yytext,yypos,yypos+size(yytext)));
<INITIAL>{digit}+       => (Tokens.INT(valOf(Int.fromString(yytext)),yypos
    ,yypos+size(yytext)));

<INITIAL>.              => (ErrorMsg.error yypos ("illegal character " ^
    yytext); continue());
```

B Κώδικας συντακτικής ανάλυσης

Πρόγραμμα 2: **tiger.grm**

```
structure A = Absyn

open Symbol

datatype lval_more = Field of symbol
                   | Subscript of A.exp

fun createLvalue(var, x::xs, pos) =
      (case x of
         Field s => createLvalue(A.FieldVar(var, s, pos), xs, pos)
       | Subscript ss => createLvalue(A.SubscriptVar(var, ss, pos), xs
          , pos))
  | createLvalue(var, nil,_)  = var

%%
%term
    EOF
  | ID of string
  | INT of int
  | STRING of string
  | COMMA | COLON | SEMICOLON | LPAREN | RPAREN | LBRACK | RBRACK
  | LBRACE | RBRACE | DOT
  | PLUS | MINUS | TIMES | DIVIDE | EQ | NEQ | LT | LE | GT | GE
  | AND | OR | ASSIGN
  | ARRAY | IF | THEN | ELSE | WHILE | FOR | TO | DO | LET | IN | END |
     OF
  | BREAK | NIL
  | FUNCTION | VAR | TYPE | NEG

%nonterm  program of A.exp
        | exp of A.exp
        | decs of A.dec list
        | dec of A.dec
        | tydec of {name : symbol, ty : A.ty, pos : int}
        | tydec_more of {name : symbol, ty : A.ty, pos : int} list
        | ty of A.ty
        | ty_field of A.field list
        | ty_fields of A.field list
        | ty_fields_more of A.field list
        | vardec of A.dec
        | fundec_more of A.fundec list
        | fundec of A.fundec
        | lvalue of A.var
        | funcall of A.exp
        | binops of A.exp
        | record of (symbol * A.exp * A.pos)
        | record_c of A.exp
        | record_cf of (symbol * A.exp * A.pos) list
```

```
        | array_c of A.exp
        | expseq of (A.exp * int) list
        | funcall_more of A.exp list
        | lvalue_more of lval_more list
        | fun_parameters of A.exp list

          (* record_c = record create, record_cf = record create fields

             array_c = array create *)


%pos int
%verbose
%start program
%eop EOF
%noshift EOF

%name Tiger

%keyword WHILE FOR TO BREAK LET IN END FUNCTION VAR TYPE ARRAY IF THEN
    ELSE
         DO OF NIL

%prefer THEN ELSE LPAREN

%value ID ("bogus")
%value INT (1)
%value STRING ("")

%nonassoc ASSIGN DO THEN OF
%nonassoc ELSE

%left OR
%left AND
%nonassoc EQ
%left NEQ GT GE LT LE
%left PLUS MINUS
%left TIMES DIVIDE
%left NEG

%%
program : exp (exp)


exp: lvalue              (A.VarExp(lvalue))
   | lvalue ASSIGN exp
         (A.AssignExp{var=lvalue, exp=exp, pos=lvalueleft})
   | NIL        (A.NilExp)
   | BREAK      (A.BreakExp(BREAKleft))
```

```
    | INT       (A.IntExp(INT))
    | STRING    (A.StringExp(STRING, STRINGleft))
    | funcall   (funcall)
    | binops    (binops)
    | MINUS exp %prec NEG
        (A.OpExp{left=A.IntExp 0, oper=A.MinusOp, right=exp, pos=
            MINUSleft})
    | record_c              (record_c)
    | array_c               (array_c)
    | IF exp THEN exp ELSE exp
        (A.IfExp{test=exp1, then'=exp2, else'=SOME(exp3), pos=exp1left
            })
    | IF exp THEN exp
         (A.IfExp{test=exp1, then'=exp2, else'=NONE, pos=exp1left})
    | WHILE exp DO exp
        (A.WhileExp{test=exp1, body=exp2, pos=exp1left})
    | FOR ID ASSIGN exp TO exp DO exp
      (A.ForExp{var=symbol ID, escape=ref false,
          lo=exp1, hi=exp2, body=exp3,pos=IDleft})
    | LET decs IN exp END
        (A.LetExp{decs=decs, body=exp, pos=LETleft}) (* check list *)
    | LPAREN expseq RPAREN  (A.SeqExp(expseq))


decs: decs dec          (decs @ [dec])
    | (* empty *)       (nil)

dec: tydec_more         (A.TypeDec(tydec_more))
   | vardec             (vardec)
   | fundec_more        (A.FunctionDec(fundec_more)

(*shift-reduce conflict, shift is desired so ok *)
tydec_more: tydec_more tydec  (tydec_more @ [tydec])
          | tydec                   ([tydec])

tydec: TYPE ID EQ ty  ({name=symbol ID, ty=ty, pos=IDleft})

ty: ID                        (A.NameTy(symbol ID, IDleft))
  | LBRACE ty_fields RBRACE (A.RecordTy(ty_fields))
  | ARRAY OF ID               (A.ArrayTy(symbol ID, ARRAYleft))

ty_field: ID COLON ID  ([{name=symbol ID1, escape=ref false,
                          typ=symbol ID2, pos=IDleft}])

ty_fields:  ty_field ty_fields_more (ty_field @ ty_fields_more)
        | (* empty *)                (nil)

ty_fields_more: COMMA ty_field ty_fields_more
```

```
                              (ty_field @ ty_fields_more)
              | (* empty *)                  (nil)

vardec: VAR ID ASSIGN exp
              (A.VarDec{name=symbol ID, escape=ref false,
                          typ=NONE, init=exp, pos=IDleft})
      | VAR ID COLON ID ASSIGN exp
        (A.VarDec{name=symbol ID1, escape=ref false,
                      typ=SOME (symbol ID2, ID2left),
               init=exp, pos=ID1left})

(* shift-reduce conflict, shift is desired so ok *)
fundec_more: fundec_more fundec  (fundec_more @ [fundec])
           | fundec                   ([fundec])

fundec: FUNCTION ID LPAREN ty_fields RPAREN EQ exp
      ({name=symbol ID,
               params=ty_fields,
               result=NONE,
               body=exp,
               pos=IDleft})

      | FUNCTION ID LPAREN ty_fields RPAREN COLON ID EQ exp
      ({name=symbol ID1,
        params=ty_fields,
        result=SOME(symbol ID2,
        ID2left),
        body=exp,
        pos=ID1left})

fun_parameters: exp                         ([exp])
              | fun_parameters COMMA exp  (fun_parameters @ [exp])

funcall: ID LPAREN funcall_more RPAREN
        (A.CallExp{func=symbol ID,
                   args=funcall_more,
                   pos=IDleft})

funcall_more: fun_parameters (fun_parameters)
            | (* empty *)    (nil)

expseq: exp                    ([(exp, expleft)])
      | expseq SEMICOLON exp (expseq @ [(exp, expleft)])
      | (* empty *)            (nil)

binops: exp PLUS exp  (A.OpExp{left=exp1, oper=A.PlusOp,
                               right=exp2, pos=PLUSleft})
      | exp MINUS exp (A.OpExp{left=exp1, oper=A.MinusOp,
                               right=exp2, pos=MINUSleft})
```

```
        | exp TIMES exp (A.OpExp{left=exp1, oper=A.TimesOp,
                                 right=exp2, pos=TIMESleft})
        | exp DIVIDE exp(A.OpExp{left=exp1, oper=A.DivideOp,
                                 right=exp2, pos=DIVIDEleft})
        | exp EQ exp    (A.OpExp{left=exp1, oper=A.EqOp,
                                 right=exp2, pos=EQleft})
        | exp NEQ exp   (A.OpExp{left=exp1, oper=A.NeqOp,
                                 right=exp2, pos=NEQleft})
        | exp LT exp    (A.OpExp{left=exp1, oper=A.LtOp,
                                 right=exp2, pos=LTleft})
        | exp LE exp    (A.OpExp{left=exp1, oper=A.LeOp,
                                 right=exp2, pos=LEleft})
        | exp GT exp    (A.OpExp{left=exp1, oper=A.GtOp,
                                 right=exp2, pos=GTleft})
        | exp GE exp    (A.OpExp{left=exp1, oper=A.GeOp,
                                 right=exp2, pos=GEleft})
        | exp AND exp   (A.IfExp{test=exp1, then'=exp2,
                                          else'=SOME(A.IntExp 0),
                                          pos=exp1left})
        | exp OR exp    (A.IfExp{test=exp1, then'=A.IntExp 1,
                                          else'=SOME(exp2),
                                          pos=exp1left})


lvalue: ID lvalue_more
(createLvalue(A.SimpleVar(symbol ID, IDleft),
                          lvalue_more, IDleft))

lvalue_more: DOT ID lvalue_more
                   (lvalue_more @ [Field(symbol ID)])
           | LBRACK exp RBRACK lvalue_more
               (lvalue_more @ [Subscript(exp)])
           | (* empty *)
             (nil)

record: ID EQ exp
               ((symbol ID, exp, IDleft))

record_c: ID LBRACE record record_cf RBRACE
               (A.RecordExp{fields=record :: record_cf,
                            typ=symbol ID, pos=IDleft})

record_cf: COMMA record record_cf
               ([record] @ record_cf)
         | (* empty *)  (nil)

array_c: ID LBRACK exp RBRACK OF exp
               (A.ArrayExp{typ=symbol ID, size=exp1,
                           init=exp2, pos=IDleft})
```

Γ Κώδικας παραγωγής συμβολικού κώδικα ARM

Πρόγραμμα 3: **armgen.sml**

```
structure Arm : CODEGEN =
struct
    structure A = Assem
    structure T = Tree
    structure Frame = ArmFrame
    exception TestStm of string
    exception TestExp of string

    fun i2s i =
        if i < 0 then "-" ^ Int.toString (~i) else Int.toString i

    fun codegen frame (stm: Tree.stm) : Assem.instr list =
        let val ilist = ref (nil: A.instr list)
            val esc = ref nil
            fun emit x =
                ilist := x :: (!ilist)
        fun result gen =
                let val t = Temp.newtemp()
                    in gen t;
                       t
                    end

        fun cmp branch =
            case branch of
               T.EQ => "beq"
             | T.NE => "bne"
             | T.LT => "blt"
             | T.GT => "bgt"
             | T.LE => "ble"
             | T.GE => "bge"
             | _ => (print("armcodegen:␣impossible\n");"")

            fun munchStm (T.SEQ(a,b)) =
                (munchStm a; munchStm b)
          | munchStm (T.EXP exp) =
            (munchExp exp; ())
               | munchStm (T.LABEL lab) =
                emit(A.LABEL{assem=Symbol.name(lab) ^ ":\n", lab=lab})
          | munchStm (T.JUMP(exp,labelList)) =
            emit(A.OPER{assem="b␣`j0\n",src=[],dst=[],jump=SOME
                labelList})


                            (* store a variable in a record field  e.g
                               record.field := 7 *)
```

```
| munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,T.TEMP t,T.BINOP(T.MUL
    ,T.CONST i1,
            T.CONST i2))), e)) =
                (print("FIRST\n");
  emit(A.OPER{assem="str␣'d0,␣['s0,␣#"^i2s(i1*i2)^"]"^"\n",
                dst=[munchExp e], src=[t], jump=NONE}))

  (* loading a record field, array index on a register *)
| munchStm(T.MOVE(T.TEMP t1, T.MEM(T.BINOP(T.PLUS,T.TEMP t2,T
    .BINOP(T.MUL,
            T.CONST i1, T.CONST i2))))) =
  emit(A.MOVE{assem="ldr␣'d0,␣['s0,␣#"^i2s(i1*i2)^"]"^"\n",
      src=t2,
                dst=t1})

  (* storing a variable in a record field, array index *)
| munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,T.TEMP t,T.CONST i)),
    e)) =
                (print("SECOND\n");
  emit(A.OPER{assem="str␣'s0,␣['s1,␣#"^i2s(i)^"]"^"\n", src=[
      munchExp e, t],
                dst=[], jump=NONE}))

  (* calling a function and storing the result on return
      value register *)
| munchStm(T.MOVE(T.TEMP rv, T.CALL(T.NAME name,args))) =
  let
  in
      (* force the usage of calling arguments with Frame.
          callargs *)
       (emit(A.OPER{assem="str␣r0,␣[sp,␣#4]"^"\n", src=Frame.
           callargs, dst=[], jump=NONE});
        emit(A.OPER{assem="str␣r1,␣[sp,␣#8]"^"\n", src=[Frame.
           a2], dst=[], jump=NONE});
        emit(A.OPER{assem="str␣r2,␣[sp,␣#12]"^"\n", src=[Frame
           .a3], dst=[], jump=NONE});
        emit(A.OPER{assem="str␣r3,␣[sp,␣#16]"^"\n", src=[Frame
           .a4], dst=[], jump=NONE});
        emit(A.OPER{assem="bl␣"^Symbol.name(name)^"\n", src=
           munchArgs(0,args), dst=[], jump=NONE});
        emit(A.MOVE{assem="mov␣'d0,␣'s0"^"\n", src=Frame.RV,
           dst=rv});
        emit(A.OPER{assem="ldr␣r0,␣[sp,␣#4]"^"\n", src=[], dst
           =Frame.callargs, jump=NONE});
        emit(A.OPER{assem="ldr␣r1,␣[sp,␣#8]"^"\n", src=[], dst
           =[Frame.a2], jump=NONE});
        emit(A.OPER{assem="ldr␣r2,␣[sp,␣#12]"^"\n", src=[],
           dst=[Frame.a3], jump=NONE});
        emit(A.OPER{assem="ldr␣r3,␣[sp,␣#16]"^"\n", src=[],
```

```
          dst=[Frame.a4], jump=NONE}))
    end
  | munchStm(T.MOVE(T.TEMP t1, T.TEMP t2)) =
    emit(A.MOVE{assem="mov␣`d0,␣`s0\n",src=t2, dst=t1})

  | munchStm(T.MOVE(T.TEMP t, e)) =
    emit(A.MOVE{assem="mov␣`d0,␣`s0\n", src=munchExp e, dst=t})

    (* storing expression to memory *)
  | munchStm(T.MOVE(T.MEM e1,e2)) =
    emit(A.OPER{assem="str␣`s0,␣[`s1]"^"\n", src=[munchExp e2,
        munchExp e1],
              dst=[], jump=NONE})

  | munchStm (T.MOVE(e1,e2)) =
    emit(A.MOVE{assem="ldr␣`d0,␣`s0\n", dst=munchExp e1, src=
        munchExp e2})

  | munchStm(T.CJUMP(branch,e1,e2,lt,lf)) =
    emit(A.OPER{assem="cmp␣`s0,␣`s1" ^ "\n" ^ (cmp branch) ^ "␣
        `j0\n",
    src=[munchExp e1,munchExp e2], dst=[], jump=SOME [lt,lf]})

      | munchStm _ = raise TestStm "Out␣of␣statements\n"


    and
    munchExp (T.BINOP(T.MUL,T.CONST i1,T.CONST i2)) =
    result (fn r => emit(A.OPER{assem="ldr␣`d0,␣=" ^ i2s(i1*i2)
        ^ "\n",
                        src=[], dst=[r], jump=NONE}))
  | munchExp (T.BINOP(T.PLUS,T.CONST i1,T.CONST i2)) =
    result (fn r => emit(A.OPER{assem="ldr␣`d0,␣=" ^ i2s(i1+i2)
        ^ "\n",
                        src=[], dst=[r], jump=NONE}))
  | munchExp (T.BINOP(T.MINUS,T.CONST i1,T.CONST i2)) =
    result (fn r => emit(A.OPER{assem="ldr␣`d0,␣=" ^ i2s(i1-i2)
        ^ "\n",
                        src=[], dst=[r], jump=NONE}))
  | munchExp (T.BINOP(T.DIV,T.CONST i1,T.CONST i2)) =
    result (fn r => emit(A.OPER{assem="ldr␣`d0,␣=" ^ i2s(i1 div
        i2) ^ "\n",
                        src=[], dst=[r], jump=NONE}))
  | munchExp (T.BINOP(T.PLUS,e1,e2)) =
    result (fn r => emit(A.OPER{assem="add␣`d0,␣`s0,␣`s1\n",
                        src=[munchExp e1,munchExp e2], dst=[r],
                            jump=NONE}))
  | munchExp (T.BINOP(T.MINUS,e1,e2)) =
    result (fn r => emit(A.OPER{assem="sub␣`d0,␣`s0,␣`s1\n",
```

```
                        src=[munchExp e1,munchExp e2], dst=[r],
                            jump=NONE}))
| munchExp (T.BINOP(T.MUL,e1,e2)) =
  result (fn r => emit(A.OPER{assem="mul␣`d0,␣`s0,␣`s1\n",
                        src=[munchExp e1,munchExp e2], dst=[r],
                            jump=NONE}))
| munchExp (T.BINOP(T.DIV,e1,e2)) =
  result (fn r => emit(A.OPER{assem="sdiv␣`d0,␣`s0,␣`s1\n",
                        src=[munchExp e1,munchExp e2], dst=[r],
                            jump=NONE}))
| munchExp (T.CONST i) =
  result (fn r => emit(A.OPER{assem="ldr␣`d0,␣=" ^ i2s(i) ^ "
     \n",
                        src=[], dst=[r], jump=NONE}))
| munchExp(T.CALL(T.NAME name,args)) =
  let
      val t = Temp.newtemp()
  in
      (* force the usage of calling arguments with Frame.
         callargs *)
   result (fn r =>
       (emit(A.OPER{assem="str␣r0,␣[sp,␣#4]"^"\n", src=Frame.
           callargs, dst=[], jump=NONE});
       emit(A.OPER{assem="str␣r1,␣[sp,␣#8]"^"\n", src=[Frame.
           a2], dst=[], jump=NONE});
       emit(A.OPER{assem="str␣r2,␣[sp,␣#12]"^"\n", src=[Frame
           .a3], dst=[], jump=NONE});
       emit(A.OPER{assem="str␣r3,␣[sp,␣#16]"^"\n", src=[Frame
           .a4], dst=[], jump=NONE});
       emit(A.OPER{assem="bl␣"^Symbol.name(name)^"\n", src=
           munchArgs(0,args), dst=[], jump=NONE});
       emit(A.MOVE{assem="mov␣`d0,␣`s0"^"\n", src=Frame.RV,
           dst=r});
       emit(A.OPER{assem="ldr␣r0,␣[sp,␣#4]"^"\n", src=[], dst
           =Frame.callargs, jump=NONE});
       emit(A.OPER{assem="ldr␣r1,␣[sp,␣#8]"^"\n", src=[], dst
           =[Frame.a2], jump=NONE});
       emit(A.OPER{assem="ldr␣r2,␣[sp,␣#12]"^"\n", src=[],
           dst=[Frame.a3], jump=NONE});
       emit(A.OPER{assem="ldr␣r3,␣[sp,␣#16]"^"\n", src=[],
           dst=[Frame.a4], jump=NONE}))
       )
  end
| munchExp(T.NAME n) =
  result (fn r => emit(A.OPER{assem="adr␣`d0,␣"^Symbol.name(n
     )^"\n", src=[], dst=[r], jump=NONE}))

   (* loading a record field in a register *)
| munchExp(T.MEM(T.BINOP(T.PLUS,T.TEMP t,T.BINOP(T.MUL,T.
```

```
        CONST i1,T.CONST i2)))) =
       result (fn r => emit(A.MOVE{assem="ldr␣'d0,␣['s0,␣#"^i2s(i1
          *i2)^"]"^"\n",
                                  src=t, dst=r}))
    | munchExp(T.MEM(T.BINOP(T.PLUS,e,T.CONST i))) =
       result (fn r => emit(A.MOVE{assem="ldr␣'d0,␣['s0,␣#"^i2s(i)
          ^"]"^"\n",
                                  src=munchExp e, dst=r}))
    | munchExp(T.MEM e) =
       result (fn r => emit(A.MOVE{assem="ldr␣'d0,␣['s0]"^"\n",
          src=munchExp e, dst=r}))
         | munchExp (T.TEMP t) = t

         | munchExp _ = raise TestExp "Out␣os␣expressions\n"

and munchArgs (i,args) =
    let val n = Frame.name frame
        val (escs,_) = Frame.getEsc n
        val esc = ref escs
        fun iter (i,args) =
        let val r = List.nth(Frame.registerTemps,i)
            (*val _ = print("TEST register: "^Temp.makestring(r)
               ^"\n")*)
        in
           case args of
               arg::args =>
                           let val res = r::iter(i+1,args)
                                 val _ = emit(A.MOVE{assem="mov␣'
                                    d0,␣'s0"^"\n",
                                         src=munchExp arg, dst=r})
                                 val t1 = i >= Frame.K
                                 val t2 = List.exists (fn n => if
                                     n = i then true else false)
                                     (!esc)
                            in
                                 if t1 andalso not t2
                                 then (esc := i::(!esc);res)
                                 else res
                            end
             | nil => nil
        end
        fun regs (esclst,i) =
           case esclst of
               e::nil =>
                emit(A.OPER{assem="str␣r"^i2s(e)^
                        ",[sp,␣#"^Int.toString(i)^"]"
                        ^"\n", src=[],
                              dst=[],
                             jump=NONE})
```

```
                  | e::es =>
                  (emit(A.OPER{assem="str␣r"^i2s(e)^
                          ",␣[sp,␣#"^Int.toString(i)^"]"
                          ^"\n", src=[],
                                 dst=[],
                                  jump=NONE});
                               regs(es,i+4))
                  | nil => ()

              val nl = iter(0,args)
              val regstr = regs((!esc),4)
           in nl end
        in
           munchStm stm;
           rev(!ilist)
        end
end
```

Βιβλιογραφία

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman, *Compilers Principles, Techiques & Tools.*

[2] Andrew W. Appel, *Modern compiler implementation in ML.*

[3] Andrew W. Appel, James S. Mattson και David R, *Tarditi2User's, Guide to ML-Lex and ML-Yacc.*

[4] Keith D. Cooper & Linda Torczon, *Engineering a Compiler.*

[5] Lawrence C. Paulson, *ML for the Working Programmer.*

[6] Robert Sedgewick, *Algorithms.*

[7] William Holm, *ARM Assembly Language Fundamentals and Techniques.*

[8] Mlton compiler, *http://mlton.org/CompilerOverview.*