ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

MPAS

A Mini-Pascal Compiler

Γινάργυρος Νίκος 2038 – Μπούρης Δημήτρης 1894

5/27/2014

Μάθημα : Μεταφραστές ΠΛΥ 602

Διδάσκων : Γ. Μανής

ΙΩΑΝΝΙΝΑ

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΡΟΛΟΓΟΣ

Σε αυτό το report θα δωθούν και θα περιγραφούν όλες οι βασικές έννοιες και πληροφορίες είναι αναγκάιο να γνωρίζουμε για να κατανοήσουμε σε βάθος τις επιμέρους λειτουργίες ενός μεταγλωτιστή. Να καταλάβουμε τον λόγο για τον οποίο χρειάζονται και τι κάνουν αυτές οι επιμέρους φάσεις μιας μεταγλώτισης. Επιπλέον θα γίνει μια καλύτερη επεξήγηση του πηγαίο κώδικα που γράφουμε στα πλαίσια αυτού του project.

Στόχος αυτού του project είναι να δημιουργήσουμε έναν μετταγλωτιστή για μια δική μας γλώσσα προγραμμάτισμού την Mini Pascal. Ο ορισμός του αλφαβήτου και των κανόνων της θα περιγραφούν αναλυτικά σε παρακάτω κεφάλαιο.

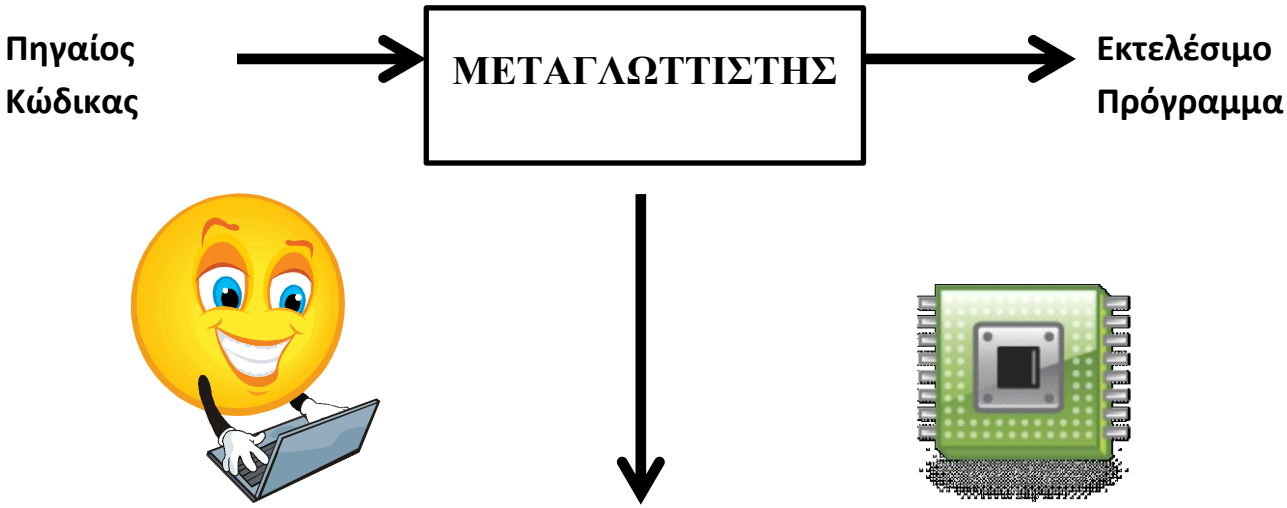
Επίσης εδώ θα ήταν καλό να σημειώσουμε πως ο κώδικας μας έχει γίνει documentated με την χρήση του εργαλείου Doxygen. Για την ευκολότερη ανάγνωση και κατανόηση κώδικα συνήθως τα σχόλια μας βοηθούν σε μεγάλο βαθμό. Έτσι επιλέξαμε να χρησιμοποιήσουμε αυτό το εργαλείο για την επεξήγηση του κώδικα μας.

Θα αναφερθούμε εκτενώς για το Doxygen σε παρακάτω κεφάλαιο.

ΚΕΦΑΛΑΙΟ 1 : ΕΙΣΑΓΩΓΗ

1.1Ορισμός Μεταγλωτιστή

Μεταγλωττιστής ή **μεταφραστής** (στα αγγλικά **compiler**) ονομάζεται ένα πρόγραμμα που μετατρέπει/μεταφράζει κείμενο γραμμένο σε μια γλώσσα προγραμματισμού (πηγαία γλώσσα) σε μια άλλη γλώσσα προγραμματισμού (τη *γλώσσα στόχο*). Το κείμενο της εισόδου ονομάζεται *πηγαίος κώδικας* (source code) και η έξοδος του προγράμματος *αντικειμενικός κώδικας* (object code).



Ο όρος «μεταγλωττιστής» χρησιμοποιείται κυρίως για προγράμματα που μεταφράζουν μια γλώσσα προγραμματισμού υψηλού επιπέδου σε μια γλώσσα χαμηλότερου επιπέδου (όπως η συμβολική γλώσσα ή η γλώσσα μηχανής). Αν το μεταγλωττισμένο πρόγραμμα πρόκειται να εκτελεστεί σε έναν υπολογιστή που έχει διαφορετικό επεξεργαστή ή λειτουργικό σύστημα σε σχέση με την πλατφόρμα που εκτελείται ο μεταγλωττιστής, ο τελευταίος τότε ονομάζεται cross-compiler. Ένα πρόγραμμα που μεταφράζει από μια γλώσσα χαμηλού επιπέδου σε μια υψηλότερου επιπέδου ονομάζεται *decompiler*. Ένα πρόγραμμα που μεταφράζει από μια γλώσσα

υψηλού επιπέδου σε μια άλλη, επίσης υψηλού επιπέδου, ονομάζεται συνήθως *γλωσσικός μεταφραστής, μεταφραστής από πηγαίο κώδικα σε πηγαίο κώδικα* (*source to source translator*) ή *μετατροπέας γλωσσών*. Ένα πρόγραμμα που μεταφράζει τη μορφή εκφράσεων σε άλλη μορφή, διατηρώντας την ίδια γλώσσα, ονομάζεται *language rewriter*.

Ένας μεταγλωττιστής μπορεί να περιλαμβάνει οποιαδήποτε από τις εξής λειτουργίες: λεκτική ανάλυση, προεπεξεργασία, συντακτική ανάλυση, σημασιολογική ανάλυση (μετάφραση καθοδηγούμενη από τη σύνταξη), παραγωγή κώδικα και βελτιστοποίηση κώδικα.

Τα σφάλματα προγραμμάτων που προκύπτουν από λανθασμένη μεταγλώττιση είναι πολύ δύσκολο να εντοπιστούν και να αντιμετωπιστούν. Για αυτόν τον λόγο οι κατασκευαστές μεταγλωττιστών κάνουν σημαντικές προσπάθειες για να βεβαιώσουν την ορθότητα λειτουργίας του λογισμικού τους.

Ο όρος μεταγλωττιστής μεταγλωττιστών (compiler-compiler) χρησιμοποιείται συχνά για τις γεννήτριες συντακτικών αναλυτών, που είναι εργαλεία που βοηθούν στην κατασκευή λεκτικών και συντακτικών αναλυτών.

1.2 Ιστορία των Μεταγλωτιστών

Το λογισμικό των πρώτων υπολογιστών ήταν γραμμένο κυρίως σε συμβολική γλώσσα. Οι γλώσσες προγραμματισμού υψηλού επιπέδου εφευρέθηκαν αργότερα, όταν τα οφέλη της εκτέλεσης του ίδιου λογισμικού σε διαφορετικούς επεξεργαστές έγιναν πιο σημαντικά από το κόστος συγγραφής ενός μεταγλωττιστή. Το περιορισμένο μέγεθος της μνήμης των πρώτων υπολογιστών υπήρξε σοβαρός περιοριστικός παράγοντας στη σχεδίαση των πρώτων μεταγλωττιστών.

Κατά τα τέλη της δεκαετίας του 1950, προτάθηκαν οι πρώτες γλώσσες προγραμματισμού που ήταν ανεξάρτητες από τη μηχανή και ακολούθησε η ανάπτυξη αρκετών πειραματικών μεταγλωττιστών. Ο πρώτος μεταγλωττιστής γράφτηκε από τη Γκρέις Χόπερ, το 1952, για τη γλώσσα προγραμματισμού A-0. Η ομάδας της FORTRAN με αρηχγό τον Τζον Μπάκους στην IBM θεωρείται γενικά ότι

παρουσίασε τον πρώτο πλήρη μεταγλωττιστή το 1957. Η COBOL ήταν μια από τις πρώτες γλώσσες που μεταγλωττίστηκαν σε πολλαπλές αρχιτεκτονικές, το 1960.

Η ιδέα της χρήσης μιας γλώσσας υψηλού επιπέδου υπήρξε γρήγορα δημοφιλής σε πολλά πεδία εφαρμογών. Οι νεότερες γλώσσες έχουν όλο και περισσότερες απαιτήσεις και οι νεότερες αρχιτεκτονικές υπολογιστών γίνονται συνέχεια πιο πολύπλοκες, με αποτέλεσμα οι μεταγλωττιστές να έχουν γίνει και αυτοί πιο πολύπλοκοι.

Οι πρώτοι μεταγλωττιστές γράφτηκαν σε συμβολική γλώσσα. Ο πρώτος μεταγλωττιστής που μπορούσε να μεταγλωττίσει τον εαυτό του σε μια γλώσσα υψηλού επιπέδου (*self-hosting*) κατασκευάστηκε το 1962 για τη Lisp από τους Τιμ Χαρτ και Μάικ Λέβιν στο MIT.[2] Από τη δεκαετία του 1970 είναι συνηθισμένο να υλοποιείται ένας μεταγλωττιστής στη γλώσσα που μεταγλωττίζει, αν και τόσο η Pascal όσο και η C έχουν υπάρξει δημοφιλείς επιλογές ως γλώσσες υλοποίησης. Η κατασκευή ενός μεταγλωττιστή που να μπορεί να μεταγλωττίσει τον εαυτό του απαιτεί μια μέθοδο bootstrapping: ο πρώτος μεταγλωττιστής πρέπει να μεταγλωττιστεί είτε με το χέρι, είτε από έναν μεταγλωττιστή γραμμένο σε άλλη γλώσσα, είτε (όπως στον μεταγλωττιστή Lisp των Χαρτ και Λέβιν) να μεταγλωττιστεί εκτελώντας τον μεταγλωττιστή σαν διερμηνέα.

Από τη Βικιπαίδεια, την ελεύθερη εγκυκλοπαίδεια

1.3 Η Δομή των Μεταγλωτιστών

Ένας μεταγλωττιστής αποτελεί τη γέφυρα μεταξύ προγραμμάτων πηγαίου κώδικα σε κάποια γλώσσα υψηλού επιπέδου και του υλικού. Πρέπει 1) να επαληθεύσει ότι τα προγράμματα έχουν σωστή σύνταξη, 2) να παράγει σωστό και γρήγορο αντικειμενικό κώδικα, 3) να οργανώσει το πώς εκτελείται το πρόγραμμα, και 4) να δώσει μορφή στην έξοδο που να είναι κατάλληλη για τον συμβολομεταφραστή ή τον συνδέτη. Ένας μεταγλωττιστής αποτελείται από τρία κύρια μέρη: το εμπρόσθιο, το ενδιάμεσο και το οπίσθιο τμήμα.

- ένα προεπεξεργαστή που αναλαμβάνει να επεξεργαστεί κάποιες ειδικές εντολές ή άλλα χαρακτηριστικά του πηγαίου κώδικα, ώστε να είναι σε κατάλληλη μορφή για τη μεταγλώττιση,

- ένα λεκτικό αναλυτή (lexer) που τεμαχίζει τον πηγαίο κώδικα σε λεκτικές μονάδες (tokens), ξεχωρίζοντας για παράδειγμα τις λέξεις-κλειδιά, τις εντολές της γλώσσας και τις τιμές του προγράμματος,
- ένα συντακτικό αναλυτή (parser) που συνθέτει τις λεκτικές μονάδες με βάση τη σύνταξη της γλώσσας, ώστε να προκύψει μια αφηρημένη μορφή του προγράμματος (συντακτικό δέντρο), κατάλληλη για περαιτέρω επεξεργασία.

Τα τμήματα του λεκτικού αναλυτή και του συντακτικού αναλυτή είναι καθιερωμένο να υλοποιούνται με ειδικά εργαλεία για αυτό το σκοπό, τις γεννήτριες λεκτικών και συντακτικών αναλυτών. Στις γεννήτριες της πρώτης κατηγορίας, όπως το Lex, ο προγραμματιστής του μεταγλωττιστή ορίζει τις λεκτικές μονάδες που μπορεί να συναντηθούν στον πηγαίο κώδικα (όπως οι δεσμευμένες λέξεις και τα αλφαριθμητικά) και η γεννήτρια αναλαμβάνει να παράγει το αντίστοιχο τμήμα του μεταγλωττιστή. Αντίστοιχα, στις γεννήτριες της δεύτερης κατηγορίας, όπως το Yacc, ο προγραμματιστής ορίζει τη γραμματική της πηγαίας γλώσσας σε μια κατάλληλη μορφή (όπως η μορφή Μπάκους-Νάουρ) και στη συνέχεια παράγεται ο συντακτικός αναλυτής που διαβάζει αυτή τη γραμματική.

Στο ενδιάμεσο τμήμα (**middle end**) γίνονται οι βελτιστοποιήσεις. Συνηθισμένοι μετασχηματισμοί βελτιστοποίησης είναι η αφαίρεση άχρηστου ή απρόσιτου κώδικα, ο εντοπισμός και η διάδοση των σταθερών (constant propagation), η μεταφορά υπολογισμών εκτός συχνά χρησιμοποιούμενων τμημάτων (για παράδειγμα, μετακίνηση έξω από μια δομή επανάληψης), ή η εξειδίκευση ενός υπολογισμού ανάλογα με τον κώδικα που τον περιβάλλει. Το ενδιάμεσο τμήμα παράγει στη συνέχεια μια άλλη ενδιάμεση αναπαράσταση, για το οπίσθιο τμήμα. Οι περισσότερες βελτιστοποιήσεις έχουν ήδη γίνει στο ενδιάμεσο τμήμα.

Το οπίσθιο τμήμα (**back end**) είναι υπεύθυνο για τη μετάφραση της ενδιάμεσης αναπαράστασης του ενδιάμεσου τμήματος σε γλώσσα μηχανής, συμβολική γλώσσα, γλώσσα προγραμματισμού (όπως η C) ή κώδικα για κάποια αφηρημένη μηχανή (abstract machine) όπως ο κώδικας byte (bytecode). Κάθε εντολή της ενδιάμεσης αναπαράστασης αντιστοιχεί σε κάποιες συμβολικές εντολές. Η κατανομή καταχωρητών αντιστοιχεί καταχωρητές στις μεταβλητές του προγράμματος. Το οπίσθιο τμήμα χρησιμοποιεί το υλικό με τέτοιο τρόπο ώστε να χρησιμοποιούνται όλες οι λειτουργικές μονάδες του υλικού με αποδοτικό τρόπο. Αν και οι περισσότεροι

αλγόριθμοι βελτιστοποίησης για αυτά τα προβλήματα είναι πολυπλοκότητας NP, έχουν αναπτυχθεί και αρκετά προχωρημένες ευριστικές τεχνικές.

1.4 Έξοδος του Μεταγλωτιστή

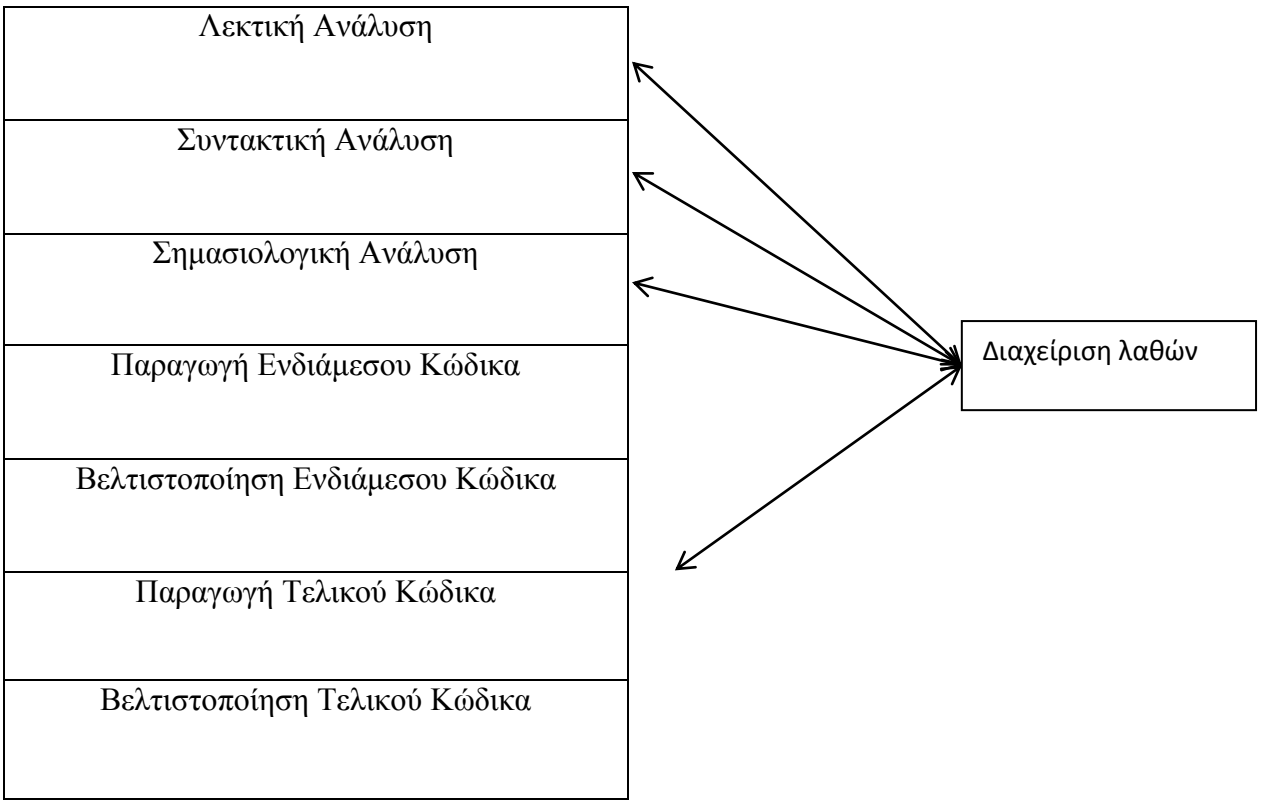
Οι μεταγλωττιστές μπορούν να χωριστούν σε κατηγορίες ανάλογα με την πλατφόρμα στην οποία πρόκειται να εκτελεστεί ο κώδικας που παράγεται (*target platform*).

Ένας μεταγλωττιστής ονομάζεται *native* ή *hosted* αν παράγει κώδικα που πρόκειται να εκτελεστεί στον ίδιο τύπο υπολογιστή και λειτουργικού συστήματος με αυτά στα οποία εκτελείται ο ίδιος ο μεταγλωττιστής. Αν ο παραγόμενος κώδικας πρόκειται να εκτελεστεί σε διαφορετική πλατφόρμα, τότε ονομάζεται *cross compiler* – αυτοί οι μεταγλωττιστές χρησιμοποιούνται συχνά στην ανάπτυξη λογισμικού για ενσωματωμένα συστήματα που δε μπορούν τα ίδια να υποστηρίξουν κάποιο περιβάλλον ανάπτυξης λογισμικού.

Η έξοδος ενός μεταγλωττιστή που παράγει κώδικα για μια εικονική μηχανή (virtual machine, VM) μπορεί να εκτελεστεί στην πλατφόρμα που εκτελείται ο μεταγλωττιστής ή σε άλλη πλατφόρμα. Για αυτόν τον λόγο, οι μεταγλωττιστές αυτού του τύπου δεν θεωρείται ότι ανήκουν σε μια από τις προηγούμενες κατηγορίες.

Η γλώσσα χαμηλού επιπέδου στην οποία παράγει κώδικα ο μεταγλωττιστής μπορεί η ίδια να είναι μια γλώσσα υψηλού επιπέδου. Η C, η οποία συχνά θεωρείται ένα είδος φορητής συμβολικής γλώσσας, μπορεί επίσης να είναι η γλώσσα του παραγόμενου κώδικα του μεταγλωττιστή. Για παράδειγμα, το Cfront, ο αρχικός μεταγλωττιστής της C++, χρησιμοποιούσε τη C σαν γλώσσα παραγόμενου κώδικα. Ο κώδικας C που παράγεται από μεταγλωττιστές αυτού του τύπου συνήθως δεν προορίζεται για ανάγνωση και χρήση από ανθρώπους. Αυτό σημαίνει ότι μπορεί να μην ακολουθεί κανόνες στοίχισης ή άλλου τύπου για ευανάγνωστο κώδικα. Κάποια χαρακτηριστικά της C την κάνουν καλή γλώσσα για αυτόν τον σκοπό, για παράδειγμα ο κώδικας C που χρησιμοποιεί τη λέξη #line μπορεί να βοηθήσει στην αποσφαλμάτωση του αρχικού πηγαίου κώδικα.

1.5 Οι φάσεις της Μεταγλώττισης



ΚΕΦΑΛΑΙΟ 2: Mini Pascal

2.1 Η γλώσσα Mini Pascal

Η Περιγραφή της Mini Pascal όπως μας δόθηκε από τον διδάσκον Γεώργιο Μανή.

Η Mini Pascal είναι μια μικρή γλώσσα προγραμματισμού. Είναι μια απλή υλοποίηση της Pascal τροποποιημένη έτσι ώστε να είναι απλουστευμένη και κατάλληλη για την υλοποίησή της για εκπαιδευτικούς σκοπούς.

Παρόλο που οι προγραμματιστικές ικανότητες της Mini Pascal είναι μικρές, η εκπαιδευτική αυτή γλώσσα περιέχει πλούσια στοιχεία και η κατασκευή του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον, αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται από άλλες γλώσσες. Η Mini-Pascal υποστηρίζει συναρτήσεις και διαδικασίες, μετάδοση παραμέτρων με αναφορά και τιμή,αναδρομικές κλήσεις, και τις περισσότερες γνωστές δομές ελέγχου. Επίσης, επιτρέπει φώλιασμα στη δήλωση συναρτήσεων κάτι που λίγες γλώσσες υποστηρίζουν.

Από την άλλη όμως πλευρά, η Mini Pascal δεν υποστηρίζει πραγματικούς αριθμούς, χαρακτήρες, συμβολοσειρές και πίνακες ή άλλες δομές δεδομένων τα οποία συνήθως οι προγραμματιστές περιμένουν από μία γλώσσα προγραμματισμού υψηλού επιπέδου. Υποστηρίζει όμως πολλές δομές ελέγχου ροής προγράμματος. Οι παραλήψεις αυτές έχουν γίνει ώστε να απλουστευτεί η διαδικασία κατασκευής του μεταγλωττιστή, μία απλούστευση όμως που έχει να κάνει κυρίως με τη μείωση των γραμμών του κώδικα που πρέπει να γραφεί και όχι με τη δυσκολία κατασκευής του, αφού οι δομές που υποστηρίζει είναι αυτές που παρουσιάζουν περισσότερο ενδιαφέρον όσον αφορά την υλοποίηση.

2.2 Λεκτικές μονάδες

Το αλφάβητο της Mini-Pascal αποτελείται από:

- τα μικρά και κεφαλαία γράμματα της λατινικής αλφαβήτου («A»,...,«Z» και «a»,...,«z»)
- τα αριθμητικά ψηφία («0»,...,«9»)
- τα σύμβολα των αριθμητικών πράξεων («+», «-», «*», «/»)
- τους τελεστές συσχέτισης («<», «>», «=», «<=», «<=», «<>»
- το σύμβολο ανάθεσης «:=»
- τους διαχωριστές («;», «,»)
- τα σύμβολα ομαδοποίησης («(», «)»,«[», «]»)
- και τα σύμβολα διαχωρισμού σχολίων («/*»,«*/»)

τις δεσμευμένες λεκτικές μονάδες :

- program
- function
- procedure
- return
- call
- begin
- end
- if
- then
- else
- do
- while
- for
- to
- step
- var
- const

- input
- print
- and
- or
- not
- select
- endselect
- is
- equal

Οι λέξεις αυτές δε μπορούν να χρησιμοποιηθούν ως μεταβλητές ή γενικότερα ως αναγνωριστικά.

Οι μεταβλητές, τα ονόματα των συναρτήσεων και των διαδικασιών, το όνομα του προγράμματος και τα ονόματα των σταθερών λέγονται αναγνωριστικά. Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία, αρχίζοντας όμως από γράμμα. Ο μεταγλωττιστής λαμβάνει υπόψη του μόνο τα τριάντα πρώτα
γράμματα. Τα υπόλοιπα απλά αγνοούνται χωρίς αυτό να αποτελεί συντακτικό λάθος.

Οι αριθμητικές σταθερές της γλώσσας είναι ακέραιες σταθερές που αποτελούνται από προαιρετικό πρόσημο και από μία ακολουθία αριθμητικών ψηφίων. Επιτρεπτές τιμές είναι οι
τιμές που είναι μεγαλύτερες ίσες από το $-(2^{31}-1)$ και μικρότερες ίσες από το $2^{31}-1$.

Οι λευκοί χαρακτήρες (tab, space, return) αγνοούνται και μπορούν να χρησιμοποιηθούν με
οποιονδήποτε τρόπο χωρίς να επηρεάζεται η λειτουργία του μεταγλωττιστή, αρκεί βέβαια να
μη βρίσκονται μέσα σε δεσμευμένες λέξεις (αναγνωριστικά, στ
αθερές κλπ.).

Το ίδιο ισχύει και για τα σχόλια, τα οποία επίσης αγνοούνται και πρέπει να βρίσκονται ανάμεσα στα σύμβολα: «/*» και «*/».

2.3 Δηλώσεις Μεταβλητών και σταθερών

2.3.1 Δηλώσεις σταθερών

Οι σταθερές είναι μνημονικά ονόματα τα οποία δίνουμε σε αριθμητικές σταθερές, έτσι ώστε
να κάνουμε το πρόγραμμα περισσότερο ευανάγνωστο. Για να δηλώσουμε σταθερά χρησιμοποιούμε τη λεκτική μονάδα const ακολουθούμενη από το μνημονικό όνομα το
σύμβολο της εκχώρησης, την αριθμητική τιμή της σταθεράς και τέλος ένα ερωτηματικό. Για παράδειγμα:

```
const N=100;
const M=1000;
```

Οι αριθμητικές σταθερές πρέπει να έχουν τιμές από $-(2^{31}-1)$ έως $2^{31}-1$.

2.3.2 Δηλώσεις μεταβλητών

Ο μοναδικός τύπος δεδομένων που υποστηρίζει η Mini-Pascal είναι οι ακέραιοι αριθμοί. Η δήλωση γίνεται με την εντολή var. Ακολουθούν τα ονόματα των αναγνωριστικών χωρίς καμία άλλη δήλωση, αφού γνωρίζουμε ότι πρόκειται για ακέραιες μεταβλητές. Οι μεταβλητές χωρίζονται μεταξύ τους με κόμματα. Το τέλος της δήλωσης των μεταβλητών γίνεται με το διαχωριστικό «;». Παράδειγμα:

```
var a,b,c;
var d;
```

2.4 Τελεστές και εκφράσεις

2.4.1 Τελεστές

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

- Πολλαπλασιαστικοί«*», «/»
- Μοναδικοί προσθετικοί «+», «-»
- Δυαδικοί προσθετικοί «+», «-»

- Επίσης έχουμε τους σχεσιακούς τελεστές: «=», «<», «>», «<>», «<=», «>=»

Και τους λογικούς

- not
- and
- or

2.4.2 Εκχώρηση

Id := expression

Χρησιμοποιείται για την ανάθεση της τιμής μίας μεταβλητής ή μίας σταθεράς ή μίας έκφρασης σε μία μεταβλητή. Παράδειγμα:

a:=3;

a:=b;

a:=(b+3)*a

2.4.3 Δομή Απόφασης if

if (condition)

begin ... end

[else

begin ... end]

Η δομή απόφασης if εκτιμάει εάν ισχύει η συνθήκη condition και εάν πράγματι ισχύει εκτελούνται οι εντολές που ακολουθούν. Το else δεν αποτελεί υποχρεωτικό τμήμα της εντολής και γι' αυτό βρίσκεται σε αγκύλη. Οι εντολές που το ακολουθούν εκτελούνται εάν η συνθήκη condition δεν ισχύει. Τα
begin ... end δεν είναι υποχρεωτικά όταν έχουμε μόνο μία εντολή μέσα σε αυτά.

2.4.4 Δομή επανάληψης με συνθήκη while

while (condition)
    begin ... end

Η δομή επανάληψης while επαναλαμβάνει συνεχώς τις εντολές που βρίσκονται ανάμεσα στα
begin ... end έως ότου η συνθήκη condition δεν ισχύει.

2.4.5 Δομή επανάληψης με μεταβλητή

for variable:=init_value to final_value step s
    begin
    ...
    end
Η δομή επανάληψης for επαναλαμβάνει συνεχώς τις εντολές που βρίσκονται ανάμεσα στα begin ... end όσες φορές χρειαστεί, έτσι ώστε έως ότου γίνουν τόσες επαναλήψεις ώστε στην πρώτη επανάληψη η μεταβλητή να ξεκινήσει από την τιμή init_value σε κάθε επόμενη να αυξάνεται κατά step_value και στην τελευταία να έχει final_value (ή την αμέσως μικρότερη που δεν ξεπερνά την final_value).

2.4.6 Δομή επιλογής

select if (expression)
    is equal to expression1: begin ... end
    is equal to expression2: begin ... end
    ...
    is equal to expressionN: begin ... end
endselect

Στη δομή select αποτιμάται το expression και έπειτα γίνεται σύγκριση της τιμής του expression με έναν ένα κατά σειρά των expression 1, expression 2, ..., expressionN. Για το πρώτο από αυτά που ισούται με το expression εκτελούνται οι εντολές που το

ακολουθούν και βρίσκονται μέσα στα begin...end. Αφού εκτελεστούν οι εντολές μέσα στα begin...end ο έλεγχος βγαίνει έξω από τη δομή. Αν κανένα εκ των expression1, expression2, ..., expressionN δεν ισούται με την τιμή του expression, τότε ο έλεγχος περνάει έξω από τη δομή select χωρίς να εκτελεστεί καθόλου κώδικας μέσα στα begin...end.

2.4.7 Εντολή εισόδου

input(variable)

Εισαγωγή τιμής από το πληκτρολόγιο.

2.4.8 Εντολή εξόδου

print(expression)

Εμφανίζει την τιμή μίας expression στην οθόνη.

2.5 Υποπρογράμματα

Η Mini-Pascal υποστηρίζει συναρτήσεις:

```
function id (formal_parameters)
        declarations
        subprograms
        begin
                sequence of statements
        end
```

ή διαδικασίες :

```
procedure id (formal_parameters)
        declarations
        subprograms
        begin
                sequence of statements
        end
```

Η « formal_parameters» είναι η λίστα των τυπικών παραμέτρων. Οι διαδικασίες και οι συναρτήσεις μπορούν να φωλιάσουν η μία μέσα στην άλλη και οι κανόνες

εμβέλειας είναι όπως της PASCAL. Η επιστροφή της τιμής μιας συνάρτησης γίνεται με τη return. Η κλήση μιας διαδικασίας γίνεται με την CALL.

2.6 Μετάδοση και επιστροφή παραμέτρων

2.6.1 Μετάδοση παραμέτρων

Η Mini-Pascal υποστηρίζει δύο τρόπους μετάδοσης παραμέτρων:

- με σταθερή τιμή. Αλλαγές στην τιμή της δεν επιστρέφονται στον καλόν πρόγραμμα

- με αναφορά. Δηλώνεται με τη λεκτική μονάδα var.Κάθε αλλαγή στη τιμή της μεταφέρεται και στο πρόγραμμα που κάλεσε τη συνάρτηση.

2.6.2 Κλήση διαδικασίας

call id (actual_parameters)

Καλεί τη διαδικασία id με τις παραμέτρους που δίνονται μέσα στην παρένθεση.

2.6.3 Επιστροφή τιμής συνάρτησης

return(expression)

Χρησιμοποιείται μέσα σε συναρτήσεις για να επιστραφεί το αποτέλεσμα της συνάρτησης και παίρνει σαν όρισμα ένα expression.

2.7 Μορφή προγράμματος

Κάθε πρόγραμμα Mini-Pascal ξεκινάει με τη λέξη program, μετά ακολουθεί ένα μνημονικό όνομα για το πρόγραμμα και στη συνέχεια δηλώσεις σταθερών, δηλώσεις μεταβλητών, συναρτήσεις και διαδικασίες και τέλος οι εντολές του κυρίως προγράμματος.

program id

declarations of constants

declaration of variables

subprograms

begin

sequence of statements

end

Η ακριβής σύνταξη της γλώσσας σε μορφή γραμματικής θα περιγραφεί παρακάτω.

2.8 Η γραμματική της Mini Pascal

2.8.1 Η Κανονική Μορφή του Μπάκους-Νάουρ (BNF)

Στη θεωρητική πληροφορική, η **BNF** (**Κανονική μορφή του Μπάκους**, αγγλ. **Backus Normal Form** ή **Μορφή Μπάκους-Νάουρ**, αγγλ. **Backus–Naur Form**) είναι μια τεχνική συμβολισμού (μετασύνταξη) για γραμματικές χωρίς συμφραζόμενα (context-free grammars),που συχνά χρησιμοποιείται για να περιγράψει τη σύνταξη μιας γλώσσας της πληροφορικής, όπως οι γλώσσες προγραμματισμού υπολογιστών, οι τύποι εγγράφων (document formats), τα σύνολα εντολών (instruction sets) και τα πρωτόκολλα επικοινωνιών. Εφαρμόζεται όπου χρειάζονται ακριβείς περιγραφές γλωσσών, για παράδειγμα σε επίσημους ορισμούς γλωσσών, σε εγχειρίδια, ή σε βιβλία για θεωρία γλωσσών προγραμματισμού.

2.8.2 Περιγραφή της Γραμματικής

Παρακάτω θα περιγράψουμε τους συντακτικούς κανόνες της Mini Pascal σε κανονική μορφή του Μπάκους-Νάουρ γνωστή ως BNF.

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

<PROGRAMBLOCK> ::= <DECLARATIONS>

<SUBPROGRAMS>

<BLOCK>

<BLOCK> ::= **begin**

<SEQUENCE>

end

<DECLARATIONS> ::= (<CONSTDECL>)* (<VARDECL>)*

<CONSTDECL> ::= **const** <ASSIGNLIST> **;** |ε

<ASSIGNLIST>::= ASSIGNCONST (**,** ASSIGNCONST)*

<ASSIGNCONST> ::= ID := CONSTANT

<VARDECL>::= **var** <VARLIST> **;** | ε

<VARLIST> ::= ID (, ID)*

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

<PROCORFUNC> ::= **procedure** ID <PROCORFUNCBODY> |

function ID <PROCORFUNCBODY>

<PROCORFUNCBODY>::= <FORMALPARS>

<PROGRAMBLOCK>

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

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

<FORMALPARITEM> ::= ID | **var** ID

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

<BLOCK-OR-STAT>::= <BLOCK> | <STATEMENT>

<STATEMENT> ::= <ASSIGNMENT-STAT> |

<IF-STAT> |

<WHILE-STAT> |

<FOR-STAT> |

<SELECT-STAT>|

<CALL-STAT> |

<PRINT-STAT> |

<INPUT-STAT>

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

<IF-STAT>::= **if** <CONDITION>

then <BLOCK-OR-STAT> <ELSEPART>

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

<WHILE-STAT>::= **while** <CONDITION>

do <BLOCK-OR-STAT>

<FOR-STAT>::= **for** <ASSIGNMENT-STAT>

to <EXPRESSION> <STEP-PART>

<BLOCK-OR-STAT>

<STEP-PART> ::= **step** <EXPRESSION> | ε

<SELECT-STAT> ::= **select if (**<EXPRESSION>**)**

<SELECT-CON-STAT> (SELECT-CON-STAT)*

endselect

<SELECT-CON-STAT>::= **is equal to:** <EXPRESSION>

<BLOCK>

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

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

<INPUT-STAT>::= **input (**ID**)**

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

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

<ACTUALPARITEM>::= <EXPRESSION> | **var** ID

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

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

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

[<CONDITION> **]** |

<EXPRESSION> <RELATIONAL-OPER> <EXPRESSION>

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

< |

> |

>=|

<=|

<>

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

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

<FACTOR>::= CONSTANT |

(<EXPRESSION>**)** |

ID <OPTACTUALPARS>

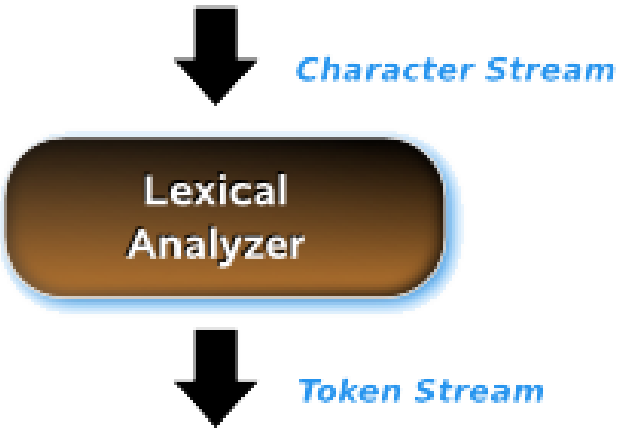
<OPTACTUALPARS>::= <ACTUALPARS> | ε

<ADD-OPER> ::= **+** | **-**

<MUL-OPER>::= ***** | **/**

<OPTIONAL-SIGN> ::= ε | <ADD - OPER>

ΚΕΦΑΛΑΙΟ 3 : ΛΕΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ



3.1 Γενικά

Η πρώτη φάση ενός μεταγλωττιστή ονομάζεται λεκτική ανάλυση ή σάρωση. Ο λεκτικός αναλυτής διαβάζει τη ροή των χαρακτήρων που αποτελούν το πηγαίο πρόγραμμα και ομαδοποιεί τους χαρακτήρες σε ακολουθίες με κάποιο νόημα, που αποκαλούνται λεξήματα. Για κάθε λέξημα, ο λεκτικός αναλυτής παράγει ως έξοδο ένα λεκτικό σύμβολο της μορφής :

<όνομα-λεκτικού συμβόλου, τιμή-ιδιότητας>

το οποίο περνά στην επόμενη φάση την συντακτική ανάλυση. Στο λεκτικό σύμβολο, το πρώτο συστατικό όνομα-λεκτικού-συμβόλου είναι ένα αφηρημένο σύμβολο που χρησιμοποιείται κατά τη συντακτική ανάλυση και το δεύτερο συστατικό τιμή-ιδιότητας δεικτοδοτεί μια καταχώρηση στον πίνακα συμβόλων γι'αυτό το λεκτικό σύμβολο. (Στην υλοποίηση μας που θα εξηγήσουμε παρακάτω η εισαγωγή πληροφοριών στον πίνακα συμβόλων γίνεται με διαφορετικό τρόπο.) Η πληροφορία της καταχώρησης του πίνακα συμβόλων χρειάζεται για τη σημασιολογική ανάλυση και την παραγωγή κώδικα.

Για παράδειγμα, υποθέστε ότι το πηγαίο πρόγραμμα περιέχει τη δήλωση ανάθεσης : *position := initial + rate * 60*

Οι χαρακτήρες σε αυτή την ανάθεση θα μπορούν να ομαδοποιηθούν στα ακόλουθα λεξήματα και να αντιστοιχηθούν στα ακόλουθα λεκτικά σύμβολα που θα περάσουν στον συντακτικό αναλυτή :

1. Το *position* είναι ένα λέξημα που θα αντιστοιχιζόταν στο λεκτικό σύμβολο <id,1> όπου το id είναι ένα αφηρημένο σύμβολο που αντιπροσωπεύει το προσδιοριστικό (identifier) και το 1 δεικτοδοτεί την καταχώρηση του πίνακα συμβόλων για το position. Η καταχώρηση του πίνακα συμβόλων για ένα προσδιοριστικό κρατά πληροφορία σχετική με αυτό, όπως το όνομα του και τον τύπο του.(Στην υλοποίηση μας και το offset του)
2. Το σύμβολο ανάθεσης := είναι ένα λέξημα που αντιστοιχίζεται στο λεκτικό σύμβολο <:=>. Χρησιμοποιούμε εδώ ένα αφηρημένο σύμβολο assign για όνομα υποκατάστατου.
3. Το λέξημα initial που αντιστοιχίζεται στο λεκτικό σύμβολο <id,2> όπου το 2 δεικτοδοτεί την καταχώρηση του πίνακα συμβόλου για το initial.
4. Το λέξημα + που αντιστοιχίζεται στο υποκατάστατο <+>.
5. Το λέξημα rate που αντιστοιχίζεται στο λεκτικό σύμβολο <id,3> όπου το 3 δεικτοδοτεί την καταχώρηση του πίνακα συμβόλου για το rate.
6. Το * είναι ένα λέξημα που αντιστοιχίζεται στο λεκτικό σύμβολο <*>.
7. Το 60 είναι ένα λέξημα που αντιστοιχίζεται στο λεκτικό σύμβολο <60>.

Τα κενά που χωρίζουν τα λεξήματα θα απορρίπτονταν από τον λεκτικό αναλυτή.

Σε αυτή την αρχική φάση της μεταγλώτισης λοιπόν θα πρέπει ο μεταγλωττιστής να διαβάσει την είσοδο που δέχεται (ένα πήγαιο πρόγραμμα σε κάποια γλώσσα προγραμματισμού) και σύμφωνα με το αλφάβητο της γλώσσας θα πρέπει να επιστρέψει στον συντακτικό αναλυτή την λεκτική μονάδα που αναγνώρισε μαζί με τον αριθμό που την προσδιορίζει.

Μια λεκτική μονάδα προσδιορίζεται μοναδικά από ένα αριθμητικό αναγνωριστικό. Εκτός από τις αριθμητικές σταθερές και τα αναγνωριστικά (ονόματα μεταβλητών, σταθερών, συναρτήσεων, ...) που έχουν αναγνωριστικό ανά ομάδα.

Στον μεταγλωτιστή που υλοποιούμε εμείς χρησιμοποιούμε ένα αυτόματο αναγνώρισης για να ξεχωρίσουμε τις λεκτικές μονάδες. Το αυτόματο αυτό, δέχεται ως είσοδο μια τρέχουσα κατάσταση και τον χαρακτήρα που μόλις διαβάστηκε από το αρχείο (το πηγαίο πρόγραμμα) και μεταβαίνει σε μια διαφορετική κατάσταση εως ότου φτάσει σε μια τελική όπου και μπορεί να σταματήσει διότι είτε έχει αναγνωρίσει μια λεκτική μονάδα είτε έχει εντοπίσει κάποιο σφάλμα.

Επίσης ο λεκτικός αναλυτής μας είναι υπεύθυνος να αναγνωρίσει τα παρακάτω σφάλματα :

- Χαρακτήρες που δεν ανήκουν στο αλφάβητο της γλώσσας.
- Κλείσιμο σχόλιων χωρίς προηγούμενο άνοιγμα σχόλιων.
- Αν μια αριθμητική σταθερά περιέχει γράμμα. *(1)*
- Αν μια αριθμητική σταθερά έχει τιμή μικρότερη από $-(2^{31}-1)$ και μεγαλύτερη από $(2^{31}-1)$. *(2)*

(1),(2) : Είναι δυο περιορισμοί που θέτει η Mini Pascal.

3.1.1 Λεκτική Ανάλυση έναντι Συντακτικής Ανάλυσης

Υπάρχει μια σειρά από λόγους για τους οποίους το τμήμα της ανάλυσης ενός μεταγλωτιστή κανονικά διαχωρίζεται στις φάσεις της λεκτικής και της συντακτικής ανάλυσης.

1. Η απλότητα του σχεδιασμού είναι ο πιο σημαντικός παράγοντας. Ο διαχωρισμός της λεκτικής και της συντακτικής ανάλυσης συχνά μας επιτρέπει να απλοποιήσουμε τουλάχιστον μια από αυτές τις εργασίες. Για παράδειγμα ένας συντακτικός αναλυτής που θα είχε να ασχοληθεί με σχόλια και χαρακτήρες λευκού κενού ως συντακτικές μονάδες, θα ήταν αρκετά πιο πολύπλοκος σε σχέση με έναν που μπορεί να υποθέσει ότι τα σχόλια και οι χαρακτήρες λευκού κενού έχουν ήδη αφαιρεθεί από το λεκτικό αναλυτή. Αν σχεδιάζουμε μια νέα γλώσσα, ο διαχωρισμός λεκτικών και συντακτικών θεμάτων μπορεί να οδηγήσει σε ένα συνολικά καθαρότερο σχεδιασμό της γλώσσας.
2. Η αποδοτικότητα του μεταγλωττιστή βελτιώνεται. Ένας ξεχωριστός λεκτικός αναλυτής μας επιτρέπει να εφαρμόζουμε εξειδικευμένες τεχνικές που εξυπηρετούν μόνο το λεκτικό έργο, όχι την εργασία της συντακτιής ανάλυσης. Επιπλέον, εξειδικευμένες τεχνικές προσωρινής αποθήκευσης (buffering) για το διάβασμα χαρακτήρων εισόδου μπορούν να επιταχύνουν σημαντικά το μεταγλωττιστή.

3. Η φορητότητα του μεταγλωττιστή βελτιώνεται. Συγκεκριμένες ιδιαιτερότητες συκευών εισόδου μπορούν να περιοριστούν στο λεκτικό αναλυτή.

3.1.2 Λεκτικά Σφάλματα

Είναι δύσκολο για ένα λεκτικό αναλυτή να πει, χωρίς τη βοήθεια άλλων αρθρωμάτων, ότι υπάρχει ένα σφάλμα στον πηγαίο κώδικα. Για παράδειγμα, αν συναντήσει τη συμβολοσειρά fi για πρώτη φορά σε ένα πρόγραμμα C στα πλαίσια του παρακάτω κώδικα:

fi (a == f(x)) ….

ο λεκτικός αναλυτής δεν μπορεί να αποφανθεί αν το fi είναι λανθασμένος συλλαβισμός της λέξης κλειδί if ή ένας αδήλωτος προσδιοριστής συνάρτησης. Αφού το fi είναι ένα έγκυρο λέξημα για την λεκτική μονάδα id, ο λεκτικός αναλυτής πρέπει να επιστρέψει τη λεκτική μονάδα id στο συντακτικό αναλυτή και να αφήσει κάποια άλλη φάση του μεταγλωττιστή – πιθανότερα το συντακτικό αναλυτή σ’αυτήν την περίπτωση να χειριστεί του σφάλμα που οφείλεται στην μετάθεση των γραμμάτων.

Παρ’ όλα αυτά, ας υποθέσουμε ότι ανακύπτει μια κατάσταση στην οποία ο λεκτικός αναλυτής δεν είναι ικανός να προχωρήσει, διότι κανένα από τα πρότυπα για λεκτικές μονάδες δεν ταυτίζεται με κανένα πρόθεμα από την εναπομένουσα είσοδο. Η απλούστερη στρατηγική ανάνηψης είναι η ανάνηψη <<κατάστασης πανικού>>. Σβήνουμε διαδοχικούς χαρακτήρες από την εναπομένουσα είσοδο, μέχρι ο λεκτικός αναλυτής να μπορεί να βρει μια σωστά σχηματιζόμενη λεκτική μονάδα στην αρχή απ’ ότι απέμεινε από την είσοδο. Αυτή η τεχνική ανάκτησης μπορεί να μπερδέψει τον συντακτικό αναλυτή, αλλά μπορεί να είναι αρκετά κατάλληλη σε ένα διαδραστικό υπολογιστικό περιβάλλον.

Άλλες πιθανές δράσεις ανάνηψης σφάλματος είναι :

1. Σβήσιμο ενός χαρακτήρα από την εναπομένουσα είσοδο.
2. Εισαγωγή ενός ελλιπούς χαρακτήρα στην εναπομένουσα είσοδο.
3. Αντικατάσταση ενός χαρακτήρα από έναν άλλο χαρακτήρα.
4. Μετάθεση δυο γειτονικών χαρακτήρων.

Μετασχηματισμοί σαν κι αυτούς μπορεί να δοκιμαστούν σε μια προσπάθεια να επιδιορθωθεί η είσοδος. Η απλούστερη τέτοια στρατηγική είναι να δούμε αν ένα πρόθεμα της εναπομένουσας εισόδου μπορεί να μετασχηματιστεί σε ένα έγκυρο λέξημα με έναν απλό μετασχηματισμό. Αυτή η στρατηγική έχει νόημα, καθώς στην πράξη με τα περισσότερα λεκτικά σφάλματα σχετίζονται με ένα μοναδικό χαρακτήρα. Μια πιο γενική στρατηγική διόρθωσης είναι να βρούμε τον μικρότερο αριθμό μετασχηματισμών που χρειάζονται για να μετατρέψουμε το πηγαίο πρόγραμμα σε ένα πρόγραμμα που αποτελείται μόνο από έγκυρα λεξήματα, αλλά αυτή η προσσέγγιση θεωρείται πολύ χρονοβόρα στην πράξη ώστε να αξίζει την προσπάθεια.

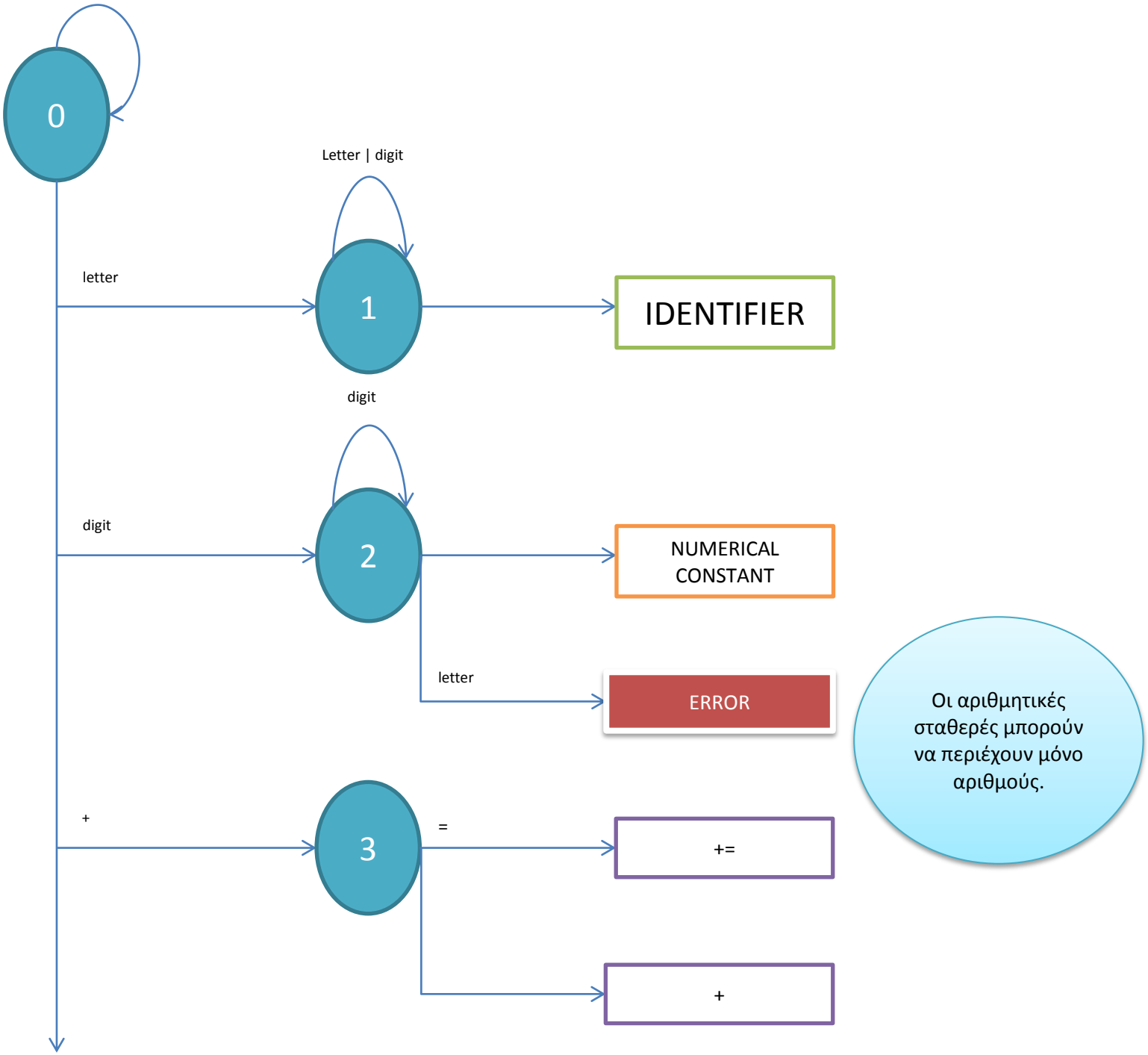
3.2 Η λειτουργία του λεκτικού Αναλυτή

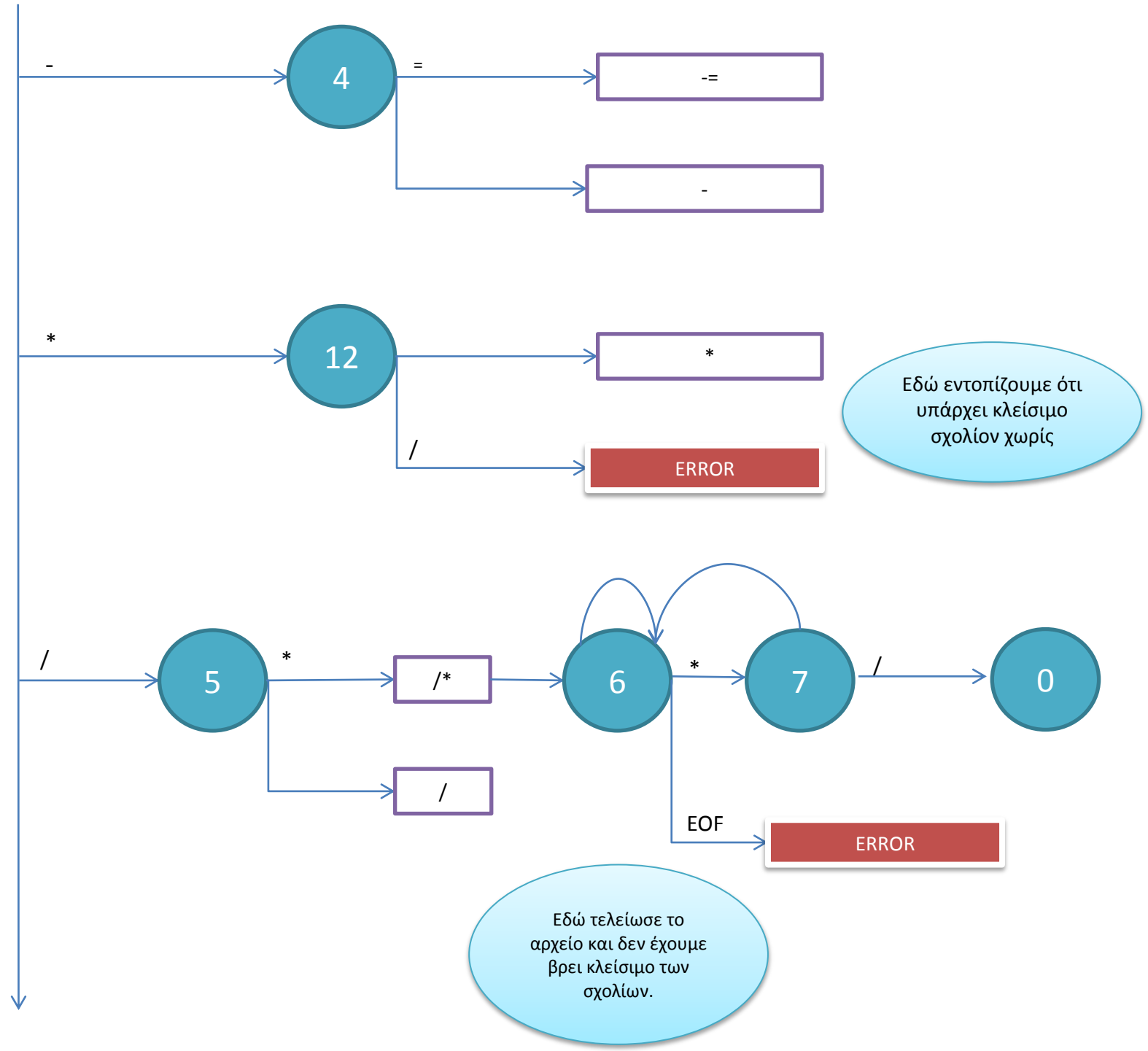
Όπως αναφέρουμε και παραπάνω η δουλειά του λεκτικού αναλυτή είναι να αναγνωρίσει λεκτικές μονάδες (tokens) που ανήκουν στην πηγαία γλώσσα. Έτσι λοιπόν, διαβάζει την είσοδο γράμμα γράμμα και χρησιμοποιώντας ένα αυτόματο αναγνώρισης, αποφασίζει για την αναγνώριση μιας λεκτικής μονάδας και της δίνει και το αντίστοιχο αριθμητικό αναγνωριστικό.

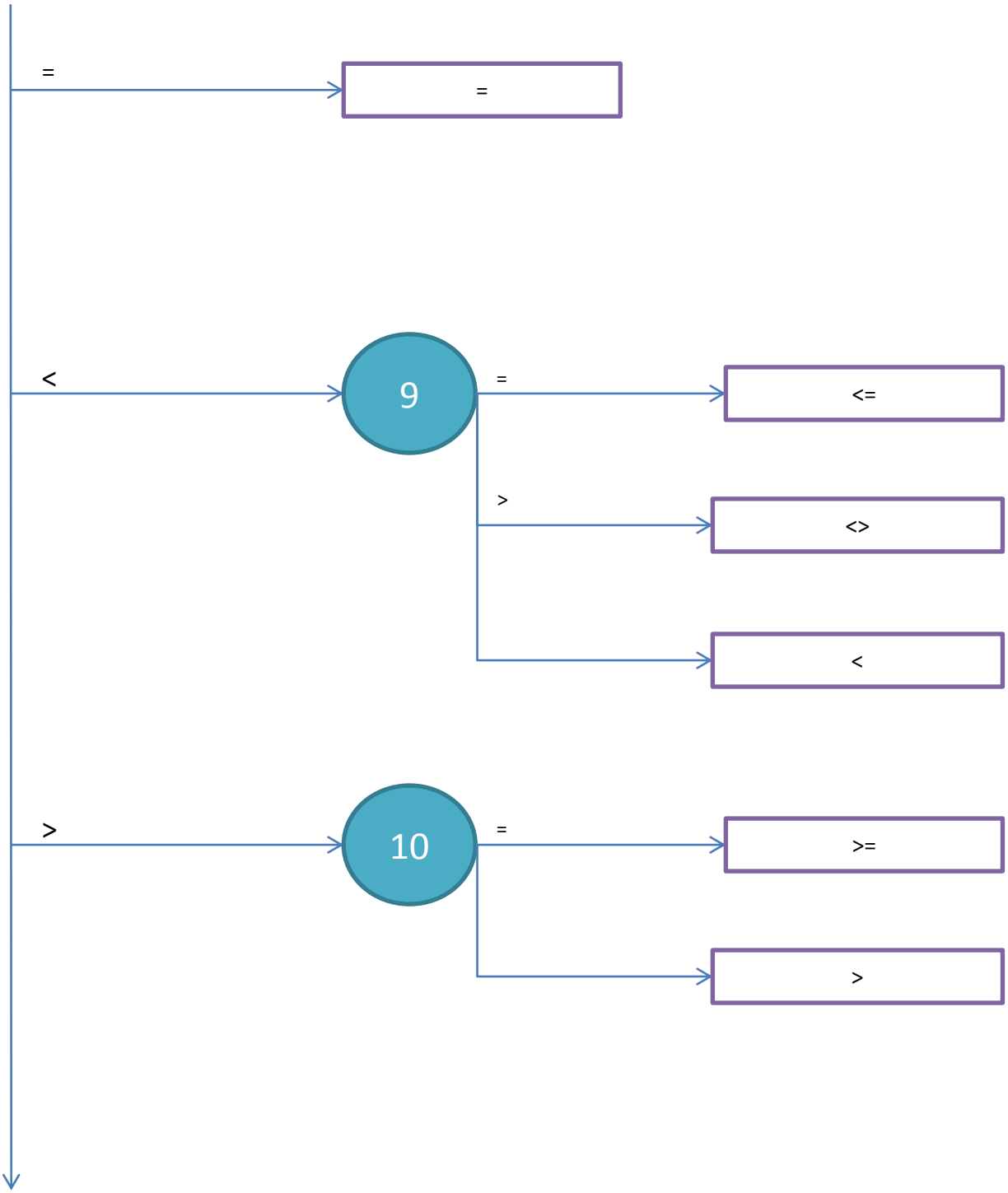
Για παράδειγμα αν ο λεκτικός αναλυτής διαβάσει από την είσοδο την αλφαριθμητική ακολουθία myTempVariable12 θα πρέπει να καταλάβει ότι είναι ένα αναγνωριστικό και να επιστρέψει τον κατάλληλο αριθμητικό κωδικό (identifierTk).

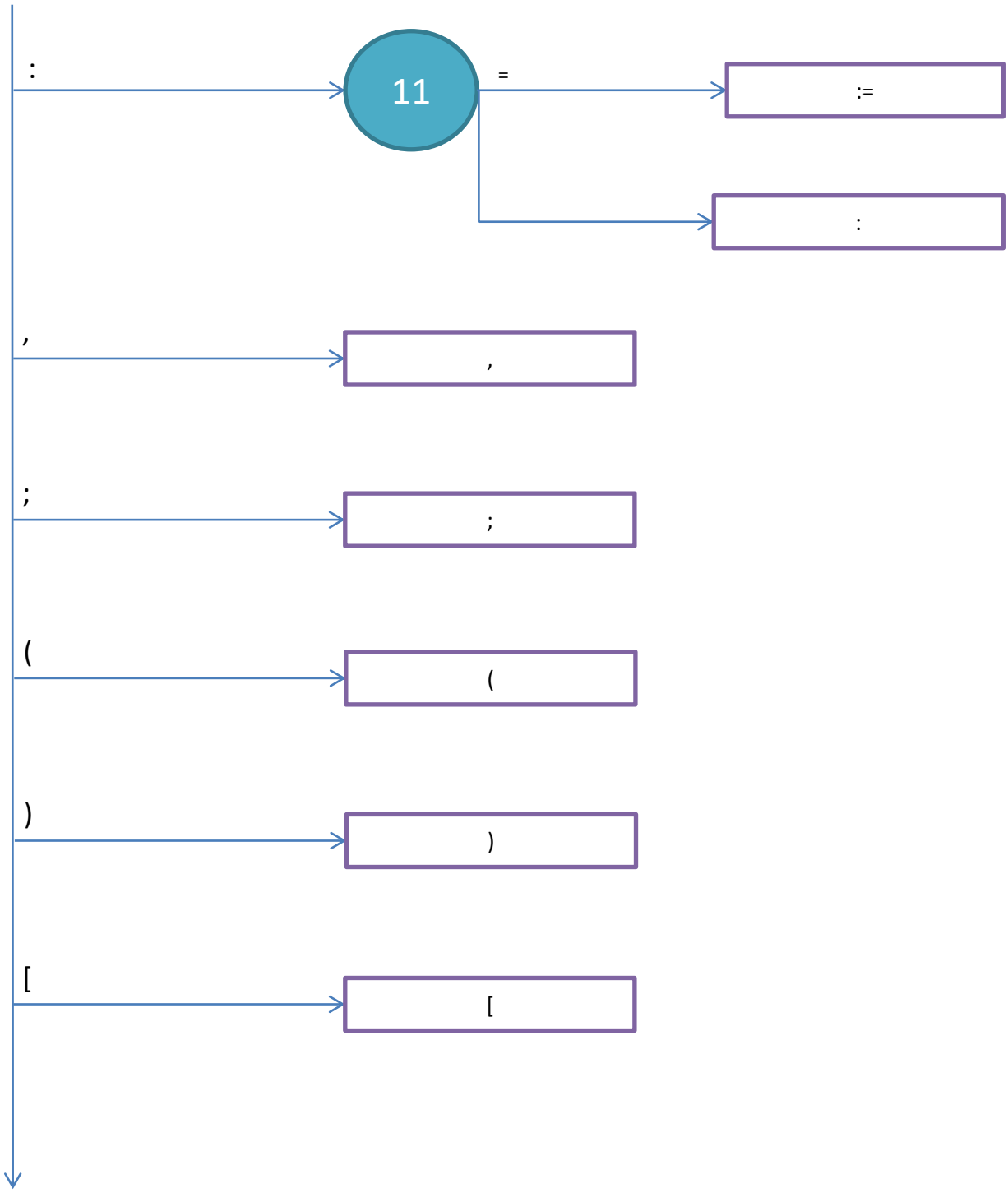
Αν μετά έρθει στην είσοδο η αριθμητική ακολουθία 123456 θα πρέπει να επιστρέψει ότι είναι αριθμός και να επιστρέψει τον αριθμητικό κωδικό (numConst) .

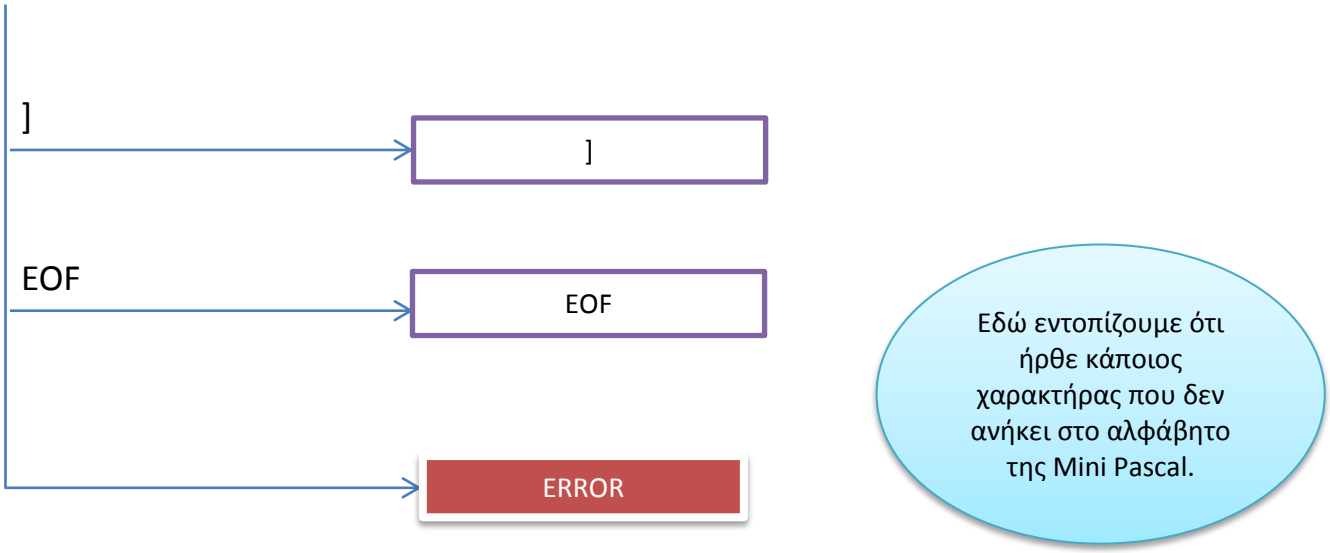
3.3 Το Αυτόματο Αναγνώρισης




-
4
=
-=
-
*
12
*
/
ERROR
Εδώ εντοπίζουμε ότι υπάρχει κλείσιμο σχολίον χωρίς
/
5
*
/*
6
*
7
/
0
/
EOF
ERROR
Εδώ τελείωσε το αρχείο και δεν έχουμε βρει κλείσιμο των σχολίων.


=
=
<
9
=
<=
>
<>
<
>
10
=
>=
>


:
11
=
:=
:
,
,
;
;
(
(
)
)
[
[


]
]
EOF
EOF
ERROR
Εδώ εντοπίζουμε ότι ήρθε κάποιος χαρακτήρας που δεν ανήκει στο αλφάβητο της Mini Pascal.

Το παραπάνω αυτόματο που περιγράψαμε έχει υλοποιηθεί στα πλαίσια της άσκησης μας με την μορφή ενός πίνακα. Παραθέτουμε τον πίνακα που περιγράφει το αυτόματο αναγνώρισης :

	(a – z) (A-Z)	Digit (0-9)	Plus +	Minus -	Star *	Slash /	Equal =	Less eq <	More eq >	Colon :	Comma ,	Semicolon ;	leftPar (	RightPar)	Leftbrk [	Rightbrk]	whitespace	EOF	Other
State 0	state1	state2	state3	state4	State 12	state5	OK	state9	State 10	state11	OK	OK	OK	OK	OK	OK	state0	OK	ERROR
State 1	state1	state1	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR
State 2	ERROR	state2	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR
State 3	OK	OK	OK	OK	OK	OK	OK **(+=)**	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR
State 4	OK	OK	OK	OK	OK	OK	OK **(-=)**	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR
State 5	OK	OK	OK	OK	state6	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR
State 6	state6	state6	state6	state6	state7	state6	state6	state6	state6	state6	state6	state6	state6	state6	state6	state6	state6	ERROR	ERROR
State 7	state6	state6	state6	state6	state6	state0	state6	state6	state6	state6	state6	state6	state6	state6	state6	state6	state6	ERROR	ERROR
State 8	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
State 9	OK	OK	OK	OK	OK	OK	OK **(<=)**	OK	OK **(<>)**	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR
State 10	OK	OK	OK	OK	OK	OK	OK **(>=)**	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR
State 11	OK	OK	OK	OK	OK	OK	OK **(:=)**	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR
State 12	OK	OK	OK	OK	OK	ERROR	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	ERROR

(*) Η κατάσταση 8 πλέον είναι αχρησιμοποίητη λόγω των τελευταίων αλλαγών για τον λογικό τελεστή ισότητας.

Έτσι λοιπόν για την αναγνώριση κάθε λεκτικής μονάδας το αυτόματο αναγνώρισης αρχίζει από την κατάσταση 0. Αναλόγως με το κάθε γράμμα που έρχεται από την είσοδο μεταβαίνει σε άλλη κατάσταση ή αναγνωρίζει κάποια λεκτική μονάδα. Από την άλλη αν δεν μπορέσει να αναγνωρίσει κάποια λεκτική μονάδα ή εντοπίσει κάποιον χαρακτήρα που δεν ανήκει στο αλφάβητο της γλώσσας ενημερώνει τον χρήστη με κατάλληλο μήνυμα.

Στην παρακάτω εικόνα μπορείτε να δείτε και την ακριβής υλοποίηση του λεκτικού αναλυτή:

```
int lex()
{

        char currentChar,buffer[BUFFER_SIZE] = {0};
        int i=0,state=0,typeOfInput=0;
        int prev_state=0,commentCounter = 0,comment_exists;

        while( state != okState && state != error && state != okNoUnget){
                currentChar=(char)fgetc(sourceFile);
                prev_state = state;
                typeOfInput = recognizeTypeOfInput(currentChar);

                state = trans_diag[state][typeOfInput];
                /*if the automato pass from state7 to state0 then comments
                exists.*/
                if (state == state0 && prev_state == state7)
                        commentCounter ++;

                if(state == okState && currentChar != '\n' )
                        ungetc(currentChar,sourceFile);
                else if (state == okNoUnget)
                        buffer[i] = currentChar;
                else if(state != state6 && state != state7 &&
                        typeOfInput != whiteSpace){
                        if( i < BUFFER_SIZE) /*prevent buffer overflow*/
                                buffer[i] = currentChar;
                        i++;
                }
        }
        if(state == error)
                errorHandler(currentChar,typeOfInput,prev_state);
        if(strlen(buffer) > MAX_TOKEN_CHARACTERS )
                errorHandler(currentChar,typeOfInput,warning);
        if( commentCounter == 0)
                strncpy(token,buffer,MAX_TOKEN_CHARACTERS);
        else
                strncpy(token,buffer+(2*commentCounter),MAX_TOKEN_CHARACTERS);

        return (keyWordRecognizer(currentChar,typeOfInput));
}
```

(*) trans_diag είναι ο πίνακας που περιγράφει το αυτόματο αναγνώρισης.

Στον κώδικα εμφανίζονται οι παρακάτω βοηθητικές συναρτήσεις :

1. recognizeTypeOfInput()

   Αυτή η συνάρτηση είναι υπεύθυνη να αναγνωρίσει αν ο χαρακτήρας της ροής εισόδου είναι αποδεκτός χαρακτήρας για το αλφάβητο της γλώσσας. Αν το αναγνωρίσει επιστρέφει ένα αριθμητικό αναγνωριστικό που προσδιορίζει αλλιώς ένα γενικό αναγνωριστικό (other) που δηλώνει άγνοια γι' αυτόν τον χαρακτήρα.

2. errorHandler()

   Αυτή η συνάρτηση είναι υπεύθυνη να ενημέρωσει τον χρήστη για οποιοδήποτε σφάλμα είναι υπεύθυνος ο λεκτικός αναλυτής να αναγνωρίσει.

3. keyWordRecognizer()

   Αυτή η συνάρτηση είναι υπεύθυνη αφού πλέον έχει αναγνωριστεί η λεκτική μονάδα να επιστρέψει ένα αριθμητικό αναγνωριστικό που την προσδιορίζει.

3.4 Το εργαλείο Lex

Στην επιστήμη υπολογιστών το **lex** είναι ένα πρόγραμμα που παράγει λεκτικούς αναλυτές ("scanners" ή "lexers").[1][2] Το lex συχνά χρησιμοποιείται μαζί με τη γεννήτρια συντακτικών αναλυτών yacc. Αρχικά το lex γράφτηκε από τον Mike Lesk και τον Eric Schmidt, και αποτελεί την κλασική γεννήτρια παραγωγής λεκτικών αναλυτών σε πολλά συστήματα Unix, και ένα εργαλείο που έχει τη συμπεριφορά του περιλαμβάνεται στο πρότυπο POSIX.

Το lex διαβάζει ένα ρεύμα εισόδου (input stream) που ορίζει το λεκτικό αναλυτή και παράγει πηγαίο κώδικα που υλοποιεί το λεκτικό αναλυτή στη γλώσσα προγραμματισμού C.

Αν και παραδοσιακά αποτέλεσε κλειστό λογισμικό, είναι διαθέσιμες εκδόσεις ανοιχτού κώδικα του lex που βασίζονται στον αρχικό κώδικα της AT&T, σαν μέρος συστημάτων όπως το OpenSolaris και το Plan 9 from Bell Labs. Μια άλλη δημοφιλής έκδοση ανοιχτού κώδικα του lex είναι το flex, ο "γρήγορος λεκτικός αναλυτής" ("fast lexical analyzer").

3.4.1 Δομή ενός αρχείου lex

Η δομή ενός αρχείου lex έχει σχεδιαστεί ώστε να μοιάζει με αυτήν ενός αρχείου yacc - τα αρχεία χωρίζονται σε τρεις ενότητες που διακρίνονται μεταξύ τους από γραμμές που περιέχουν μόνο δύο σύμβολα "τοις εκατό", όπως το εξής:

Ενότητα των ορισμών (*Definition section*)
%%
Ενότητα των κανόνων (*Rules section*)
%%
Ενότητα κώδικα C (*C code section*)

- Στην **ενότητα των ορισμών** ορίζονται οι μακροεντολές και εισάγονται τα αρχεία επικεφαλίδας που είναι γραμμένα σε C. Μπορεί επίσης να γραφεί οποιοσδήποτε κώδικας C σε αυτό το σημείο, ο οποίος θα αντιγραφεί ως έχει στο παραγόμενο αρχείο πηγαίου κώδικα.
- Η **ενότητα των κανόνων** είναι η σημαντικότερη: αντιστοιχίζει πρότυπα (patterns) με εντολές της C . Τα πρότυπα είναι απλά κανονικές εκφράσεις. Όταν ο λεκτικός αναλυτής εντοπίσει στην είσοδό του κάποιο κείμενο που ταιριάζει με κάποιο από τα δοθέντα πρότυπα, εκτελεί τον αντίστοιχο κώδικα C. Αυτός είναι ο βασικός τρόπος με τον οποίο λειτουργεί το lex.
- Η **ενότητα κώδικα C** περιέχει εντολές C και συναρτήσεις που αντιγράφονται ως έχει στο παραγόμενο αρχείο πηγαίου κώδικα. Αυτές οι εντολές υποτίθεται ότι περιλαμβάνουν κώδικα που καλείται από τους κανόνες στην ενότητα των κανόνων. Σε μεγαλύτερα προγράμματα είναι πιο βολικό να τοποθετείται αυτός ο κώδικας σε ένα ξεχωριστό αρχείο και να γίνεται σύνδεση με αυτό στο χρόνο μεταγλώττισης.

3.4.2 Χρήση του lex με άλλα εργαλελια προγραμματισμού

Χρήση του lex με γεννήτριες συντακτικών αναλυτών:

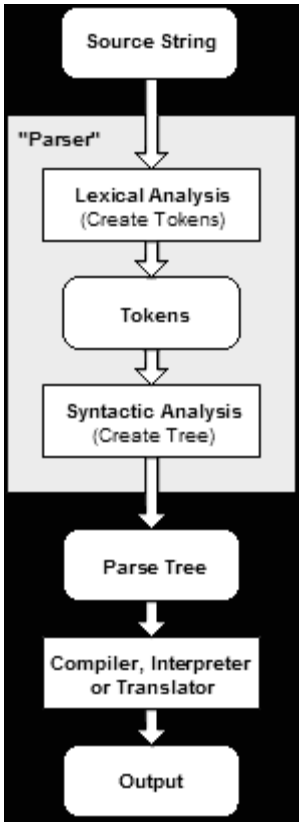
Το lex και οι γεννήτριες συντακτικών αναλυτών, όπως το Yacc ή το Bison, συχνά χρησιμοποιούνται μαζί. Οι γεννήτριες συντακτικών αναλυτών χρησιμοποιούν μια τυπική γραμματική για να αναλύσουν συντακτικά ένα ρεύμα εισόδου, κάτι που το lex δε μπορεί να κάνει χρησιμοποιώντας απλές κανονικές εκφράσεις (το lex

περιορίζεται σε απλά αυτόματα πεπερασμένων καταστάσεων). Όμως οι γεννήτριες συντακτικών αναλυτών δε μπορούν να διαβάσουν από ένα απλό ρεύμα εισόδου – χρειάζονται μια σειρά από λεκτικές μονάδες (tokens). Το lex συχνά χρησιμοποιείται για να παρέχει αυτές τις λεκτικές μονάδες στη γεννήτρια συντακτικών αναλυτών.

Lex και make

Το make είναι ένα εργαλείο που μπορεί να χρησιμοποιηθεί για να συντηρεί προγράμματα που χρησιμοποιούν το lex. Το make θεωρεί ότι ένα αρχείο με την κατάληξη .l είναι αρχείο πηγαίου κώδικα lex. Η εσωτερική μακροεντολή του make LFLAGS μπορεί να χρησιμοποιηθεί για να ορίσει επιλογές του lex που θα κληθούν αυτόματα από το make.

ΚΕΦΑΛΑΙΟ 4 : ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ



4.1 Γενικά

Στο προηγούμενο κεφάλαιο μιλήσαμε για την λεκτική ανάλυση και αναλύσαμε την υλοποιησή της.Επόμενο βήμα για την μεταγλώτιση ενός προγράμματος ειναι η Συντακτική ανάλυση(Parsing or syntactic analysis).Με τον όρο συντακτική ανάλυση αναφαιρόμαστε στην διαδικασία αναλύσης μιας γραματοσειράς απο σύμβολα σύμφωνα με τους κανόνες μιας τυπικής γραμματικής(formal grammar).

Ο συντακτικός αναλυτής δέχεται ως είσοδο ενα πρόγραμμα με τη μορφή ακολουθίας λεκτικών μοναδών και ελέγχει αν το πρόγραμμα συμφωνεί με την γραμματική της γλώσσας που υλοποιεί(αν ανήκει στην συγκεκριμέμη γλώσσα).Σε περίπτωση συντακτικού λάθους ενημερώνει τον χρήστη με κατάλληλο μήνυμα και τέλος παράγει το συντακτικό δεντρο που αντιστοιχεί στην ακολουθία εισόδου.Οι παραπάνω ενέργειες πρέπει να υλοποιηθούν με ιδιαίτερη προσοχή. Ένας λόγος ειναι

πως σε διάφορα σημεία της φασής αυτής υπάρχει διεπαφή με τον χρήστη(μηνύματα λάθους).Τα μηνύματα αυτα πρέπει να ειναι κατατοπιστικά και ακριβή.

4.2 Τρόποι υλοποίησης.

Διαχορίζουμε δύο ειδη συντακτικών αναλυτών ανάλογα με τον τρόπο που σχηματίζουν το δέντρο:

4.2.1 Απο πανω προς τα κατω(top down Syntactic analyzer)

Ο αναλυτής αυτός ξεκινά από τη ρίζα του δέντρου και αντικαθιστά μη τερματικά σύμβολα από το αριστερό τμήμα παραγωγών με τα αντίστοιχα δεξιά τους τμήματα μέχρι να αναγνωρίσει όλη την είσοδο.Το μη τερματικό συμβολο που επιλέγεται για αντικατάσταση είναι κάθε φορα το αριστερότερ(leftmolst)(L) μη τερματικό σύμβολο.Ακόμα, ο αναλυτής για να μπορέσει να υλοποιηθεί πρέπει να αρκεί να διαβάζει κάθε φορά ενα(1) μόνο σύμβολο από την είσοδο από αριστερά προς τα δεξιά(L).Έτσι ο αναλυτής και η αντίστοιχη γραμματική ονομάζονται **LL(1)**

Ο αναλυτής αυτός μπορεί να υλοποιηθεί είτε ως ένας συντακτικός αναλυτής αναδρομικής κατάβασης(recursive descent analyzer) ή ως ένα ειδικό αυτόματο στοίβας.Ο συντακτικός αναλυτής αναδρομικής κατάβασης αποτελείται από συναρτήσεις ,μια για κάθε μη τερματικό σύμβολο,που καλούν η μια την άλλη.Το ρόλο της στοίβας παίζει η στοίβα κλήσεων των συναρτήσεων που υλοποιεί ο επεξεργαστής.Για τον λόγο αυτό οι συντακτικοί αυτοί αναλυτές είνα συχνά γρηγορότεροι από αυτούς που υλοποιούνται με αυτόματα..

4.2.2 Απο κάτω προς τα πάνω(bottom up Syntactic analyzer)

Ο αναλυτής αυτός χρησιμοποιεί κάθε φορα τη δεξιότερη(rightmost) παραγωγή.Ανάλογα με το αν χρειάζεται να εξετάσει 0,1 η Κ σύμβολα εισόδου για να υλοποιήσει μια παραγωγή ονομάζεται **LR(0), LR(1)** ή **LR(K)**.Η υλοποίηση υου αναλυτή γίνεται απο ειδικό αυτοματο στοίβας.Ένα αυτόματο στοίβας Μ δέχεται τη συμβολοσειρα w αν και μόνο αν,ξεκινώντας από την αρχική κατάσταση με άδεια

στοιβα, επεξεργαστεί το w και καταλήξει σε κάποια τελική κατασταση του F(σύνολο τελικών καταστάσεων) με αδεία στοίβα παλι. ,

4.3 Αναλυτής αναδρομικής κατάβασης(Recursive descent analyzer)

Συην ανάλυση αναδρομικής κατάβασης ο κάθε κανόνας ,που αναφέρεται σε κάποιο μη τερματικό σύμβολο,εκφράζεται από τον ορισμό μιας διαδικασίας, που θα το αναγνωρίζει.

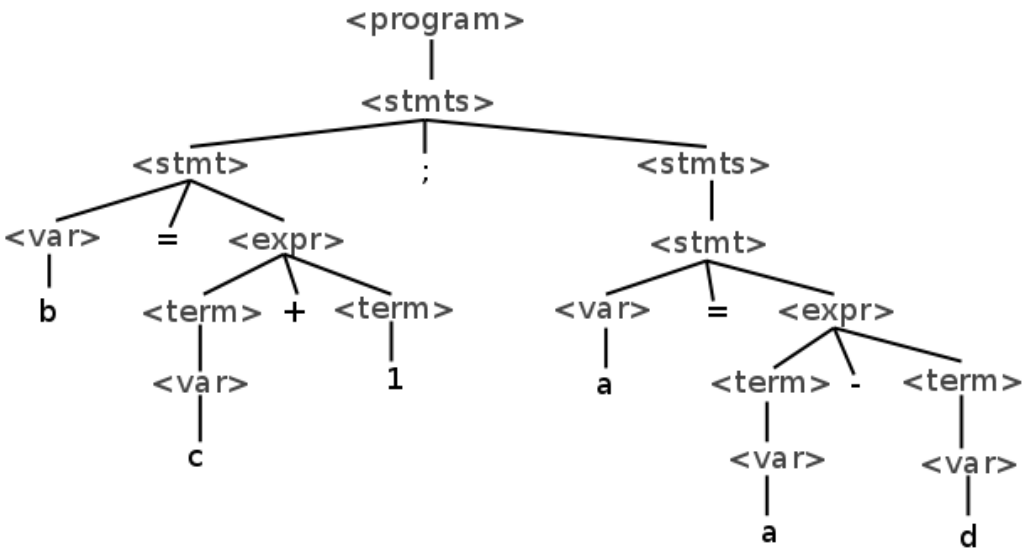
Ένας αναλυτής αναδρομικής κατάβασης αποτελείται:

- Από μια καθόλικη μεταβλητή,που περιέχει την τιμή της τρέχουσας λεκτικής μονάδας.
- Από μια βοηθητική διαδικασία αναγνώρισης,που ελέγψει αν η τρέχουσα λεκτική μονάδα είναι η αναμενόμενη που καλεί τη διαδικασία λεξικής μονάδας και την ενημέρωση της καθολικής μεταβλητής.
- Απο μια διαδικασία εκκιησης, που αφού διαβάσει την πρώτη λεξική μονάδα καλεί τη διαδικασία, που αντιστοιχεί στο μη τερματικό σύμβολο της αρχής.

Επίσης ο συντακτικός αναλυτής κατάβασης υλοποιεί ια συνάρτηση για κάθε κανόνα της γραμματικής.

Η καθε συναρτηση:

- Για τα μη τερματικά συμβολο του κανόνα καλεί την αντίστοιχη συνάρτηση.
- Διαβάζει τα τερματικά σύμβολα του κανόνα από το λεκτικό αναλυτή.
- Αν διαβάσει ένα τερματικό σύμβολο που δεν αντιστοιχεί στον κανόνα το σπρώχνει πίσω στην είσοδο(ugetch()) και επιστρέφει.
- Αν εχει να επιλέξει ανάμεσα σε διαφορετικές παραγωγές για ένας κανόνα,διαβάζει ένα σύμβολο εισόδου και αποφασίζει ανάλογα με αυτό.



Ενα συντακτικο δεντρο μετά τη φάση της συντακτικής ανάλυσης.

4.4 Mini Pascal Υλοποιηση.

Στα πλαίσια της εργάσιας και όσον αφορα στο κομμάτι της συντακτικής ανάλυσης υλοποιήσαμε ενα συντακικό ανάλυτη αναδρομικής κατάβασης(Recursive descent analyzer).Ακολουθήσαμε τις αρχές που περιγράφονται παραπάνω τις οποίες θα περιγράχουμε αναλυτικά σην συνέχεια.

Στο προηγούμενο κεφάλαιο αναλυσαμε τον λεκτικό αναλυτή, παρόλα αυτά καποιές έννοιες τις οποίες εξηγήσαμε εκεί ειναι καλό να τις θυμηθούμε ξανά γιατί παιζουν πολυ σημαντικό ρόλο και σε αυτήν τη φάση.Στην υλοποίηση χρησιμοποιούμε δυο καθολικές μεταβλητές(global variables) οι οποιες χρησιμοποιούνται για την αναγνώριση και τον έλεγχο των λεκτικών μονάδων.Η πρώτη μεταβλητή ειναι μια συμβολοσειρά **char * token** η οποία σε κάθε στιγμή της συντακτικής ανάλυσης περιέχει την λεκτική μονάδα που διάβασε ο λεκτικός ανάλυτης.Η δεύτερη μεταβλητή **int token_id** κρατάει έναν μονάδικό ακέραιο κωδικό για τον τύπο της λεκτικής μονάδας(περισσότερες πληροφοριες στο προηγούμενο κεφάλαιο).

Η φάση της συντακτικής ανάλυσης ξεκινάει με την κλήση της συνάρτησης **program()** στο αρχείο **syntaxAnalyzer.c** Πιο πριν είναι απαραίτητο οι καθόλικες

μεταβλητές token,token_id να έχουν αρχικοποιηθεί.Αυτό γινεται με την κλήση του λετικού αναλυτή **token_id = lex()**; Όπως βλέπουμε παρακάτω έτσι και γίνεται.

4.4.1 Ο κανόνας program

Η συντακτική ανάλυση ξεκινάει με την κλήση της συνάρτησης syntaxAnalyzer().Εκεί καλείται ο κανόνας program() καθώς όπως γνωρίζουμε η πρώτη λέξη ενός προγράμματος Mini Pascal είναι "program".

```
int syntaxAnalyzer()
{       token_id = lex();
        program();
        return 1;
}
```

Ο λεκτικός αναλυτης γεμίζει τις καθολικές μεταβλητές.Τώρα καλείται η συνάρτηση του συντακτικού αναλυτή program().Αυτή η συνάρτηση ελέγχει εαν η πρώτη λεκτική μόναδα του η οποία βρίσκεται στην μεταβλητή token και ο κωδικός της στην μεταβλητή token_id είναι η λέξη **"program"**.Αν ναι τότε συνεχίζει την συντακτική ανάλυση με τον επόμενο κατά σειρά κανόνα της γραμματικής αλλιώς ενημερώνει τον χρήστη με ενα διαγνωστικό μηνυμα για το σφάλμα που συνέβει.Στη συνέχεια καλούνται οι κανόνες σύμφωνα με την γραμματική που έχουμε περιγράψει σε προηγούμενο κεφάλαιο.

Η συνάρτηση του κανόνα program():

```
void program()
{
	char blockId[TOKEN_SIZE];
	struct labelList * startList = NULL;
	struct scope* mainScope = NULL;
	struct entity* newEntity;

	allocateScope(&mainScope);

	genQuad("jump","_","_","_");

	compileCounter = programList->head;
	startList = makeList(quadNumber);
	if (token_id == programTk)
	{
		token_id = lex();
		if (token_id == identifierTk)
		{
			strncpy(blockId,token,MAX_TOKEN_CHARACTERS);
			strncpy(finalCodeFileName,token,MAX_TOKEN_CHARACTERS);
			openFinalCodeFile();
			fprintf(finalCode,"L1:\tjmp L0\n");

			newEntity = createNewEntity(blockId,256,FUNCTION);
			addEntityToScope(newEntity,mainScope);
			token_id = lex();
			programBlock(blockId,PROGRAMBLOCK,startList,mainScope);

			produceFinalCodeForSubProgram(mainScope,blockId);

			freeScope(&mainScope);
		}
		else
		{
			printf("%s :Syntax Error:%d:Program name expected."
			"Found: \" %s \".\n",file_name,currentLineInFile,token);
			exit(EXIT_FAILURE);
		}
	}
	else
	{
		printf("%s :Syntax Error:%d:The keyword 'program' was expected."
		"Found: \" %s \"\n",file_name,currentLineInFile,token);
		exit(EXIT_FAILURE);
	}
}
```

4.4.2 Ο κανόνας while

Με την ίδια λογική όταν φτάσουμε στον κανόνα while έχουμε ήδη αναγνωρίσει την λεκτική μονάδα while και πρέπει να ελέγξουμε την συνθήκη(κανόνας condition) που ακολουθεί.Στη συνέχεια αν δεν υπάρχει κάποιο συντακτικό σφάλμα πρέπει να αναγνωριστεί η λεκτική μόναδα

“do” .Εάν αναγνωριστεί τότε πρέπει να αναγνωρίσουμε τις εντολές πρέπει να εκτελεστούν σε αυτό το loop.Γιαυτό καλείται ο κανόνας blockOrStat που αναγνωρίζει το μπλοκ εντολών.

Η συναρτηση του κανόνα while :

```
void whileStat(struct scope *currentScope)
{
        int Bquad=0;
        char repQuad[TOKEN_SIZE];
        struct labelList *condTrueList = NULL, *condFalseList = NULL;
        int condQuad = quadNumber + 1 ;

        condition(&condTrueList,&condFalseList,currentScope);

        if(token_id == doTk)
        {
                Bquad = quadNumber + 1 ;
                sprintf(repQuad,"%d",Bquad);
                backPatch(&condTrueList,Bquad);

                token_id = lex();
                blockOrStat(currentScope);

                sprintf(repQuad,"%d",condQuad);
                genQuad("jump","_","_",repQuad);
                backPatch(&condFalseList,quadNumber + 1);

        }
        else
        {
                printf("%s :Syntax Error:%d:The keyword 'do' was expected.\n"
                ,file_name,currentLineInFile);
                printSpaces();
                printf("Proper syntax for \"while\"is: while <condition> do "
                "<block or statement>.\n");
                customExit();
        }
}
```

4.5 Το εργαλείο Yacc.

Το Yacc είναι ένα προγραμμα για το Unix.Το όνομα **Yacc** είναι ακρόνυμο του **Yet Another Copiler Compiler(**Ενας ακόμα μεταφραστής μετάφραστών).Το πρόγραμμα αυτό οπως δηλώνει και το όνομα του είναι ένας αυτοματοποιημένος τρόπος παραγωγής ενός συντακτικού αναλυτή.Ανήκει στην κατηγορία LALR ή Look-Ahead LR parser generator.Πιο συγκεκριμένα ενας LALR parser βασίζεται στην αναλυτική γραμματική γραμμένη σε σημειογραφεία παρόμοια με BNF την οποία πέρνει σαν είσοδο και με βάση αυτη παράγει τον συντακτικό αναλυ΄τη.Αναπτύχθηκε το 1970 από τον Stephnen C. Jonshon και είναι γραμμενο στην γλώσσα προγραμματισμού B.

Το Yacc και πολλά παρόμοια προγράμματα είναι πολύ δημοφιλή.Το ίδιο το Yacc ήταν ο προεπιλεγμένος παραγωγέας συντακτικης ανάλυσης στα περισσότερα συστήματα Unix.Σήμερα έχει αντικατασταθεί απο κάποια πριο πρόσφτατα εργαλεία ,μεγάλης συμβατότητας όπως το Berkley Yacc,GNU bison ,MKS Yacc and Abraxas PCYACC. Μια ανανεομένη έκδοση του πρώτου(original) Yacc συμεπριλαμβάνεται σαν κομμάτι του Sun's OpenSolaris project.Σε αυτή την έκδοση έχουν γίνει μικρές βελτιώσεις και έχουν προστεθεί επιπλέον χαρακτηριστικά και δυνατότητες.Παρόλα αυτά η λογική του προγράμματος και ο τρόπος με τον οποίο τελικά παραγεί το τελικο συντακτικό αναλυτή παραμένει ο ίδιος.

Το συγκεκριμένο εργαλείο παράγει μονο εναν parser(αναλυτή φράσεων).Για πλήρη συντακτική ανάλυση είναι απαραίτητη η χρηση ενός εξωτερικού λεκτικόυ αναλυτή ο οποίος θα πραγματοποιήσει το πρώτο στάδιο διαβάσματος τον λεκτικών μονάδων(tokenization stage),και μετά ακολουθεί η παράγωγή του συντακτικού δέντρου.Κάποια τέτοια προγράμματα (Λεκτικοί αναλυτές) είναι το Lex και το Flex.Μία αναφόρα για τον συντακτικό αναλυτή Lex γίνεται στο προγηγούμενο κεφάλαιο.

Στον παρακάτω σχήμα φαίνεται ενα διάγραμμα στο οποιιό ο λεκτικός αναλυτηης παράγεται από το εργαλει Lex και ο συντακτικός από το εργαλείο Yacc.

ΚΕΦΑΛΑΙΟ 5 : Σημασιολογική Ανάλυση

Η φάση της σημασιολογικής ανάλυσης ενός μεταγλωττιστή ασχολείται κυρίως με το στατικό έλεγχο (static checking) του προγράμματος που μεταγλωτίζεται.Ακολουθεί, η φάση της παραγωγής ενδιάμεσου κώδικα και έτσι ολοκληρώνεται το front-end τμήμα του μεταγλωττιστή.Η σημασιολογική ανάλυση μπορεί να γίνεται ταυτόχρονα με την συντακτική ανάλυση(μεταγλωττιστές ενός περάσματος) ή να την ακολουθεί(μεταγλωτιστές πολλών περασμάτων.

5.1 Στατικός έλεγχος

Ο στατικός έλεγχος βασίζεται στη σημασιολογία την γλώσσας που μεταγλωττίζουμε και γίνεται κατά την διάρκεια της μεταγλώττισης.Περιλαμβάνει:

- Συντακτικούς ελέγχους που δεν καλύπτονται από την γραμματική.Για παράδειγμα κάθε αναγνωριστικό πρέπει δηλώνεται μόνο μια φορά σε μία εμβέλεια.
- Ελέγχους τύπων(type checking).Ελέγχουμε τους κανόνες τύπων της γλώσσας , δηλαδή αν ενας τελεστής εφαρμόζεται στο σωστό αριθμό και τύπο ορισμάτων.
- Ύπαρξη μιας εντολής break εκτός κατάλληλου βρόγχου.

5.2 Δυναμικός έλεγχος

Ο δυναμικός έλεγχος είναι υπεύθυνος για την απόδοση ερμηνείας σε προγράμματα που περιγράφει την συμπεριφορά τους.Γενικά μπορεί να περιγραφεί με τους παρακάτω τρόπους:

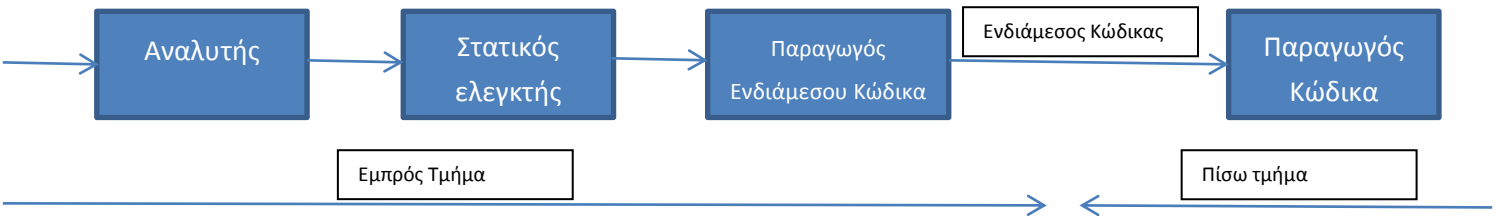
- Λειτουργική Σημασιολογία (operational semantics): περιγράφει συμπεριφορά σαν υπολογιστικά βήματα.
- Δηλωτική Σημασιολογία (denotational semantics): Περιγραφή με κάποιο μαθηματικό φορμαλισμό.
- Αξιωματική Σημασιολογία (axiomatic semantics): περιγραφή ως λογικές προτάσεις πάνω στις ιδιότητες του προγράμματος.

ΚΕΦΑΛΑΙΟ 6 : Παραγωγή Ενδιάμεσου Κώδικα

6.1 Γενικά

Στο μοντέλο ανάλυσης – σύνθεσης ενός μεταγλωττιστή, το εμπρός τμήμα αναλύει ένα πηγαίο πρόγραμμα (source program) και δημιουργεί μια ενδιάμεση αναπαράσταση, από την οποία το πίσω τμήμα παράγει τον τελικό κώδικα (target code) . Ιδανικά, οι λεπτομέρειες της πηγαίας γλώσσας (source language) περιορίζονται στο εμπρός τμήμα, και οι λεπτομέρειες της μηχανής στόχου στο πίσω τμήμα. Με μια κατάλληλα ορισμένη ενδιάμεση αναπαράσταση, ένας μεταγλωττιστής για την γλώσσα i και την μηχανή j μπορεί να κατασκευαστεί με τον συνδιασμό του εμπρός τμήματος για την γλώσσα i με το πίσω τμήμα για την μηχανή j. Αυτή η προσέγγιση για την δημιουργία συνόλων μεταγλωττιστών μπορεί να μας γλυτώσει από σημαντικά μεγάλο φόρτο εργασίας : m x n μεταγλωττιστές μπορούν να κατασκευαστούν με το γράψιμο ακριβώς m εμπρός τμημάτων και n πίσω τμημάτων.

Αυτό το κεφάλαιο εξετάζει τις ενδιάμεσες αναπαραστάσεις, το στατικό έλεγχο τύπων και την παραγωγή ενδιάμεσου κώδικα. Για απλότητα, υποθέτουμε ότι το εμπρός τμήμα ενός μεταγλωττιστή οργανώνεται ως εξής :



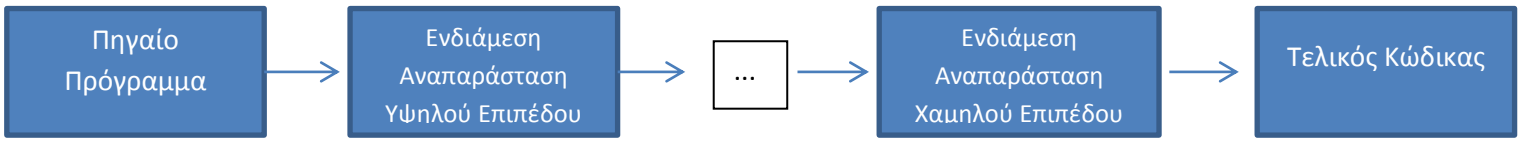
όπου η ανάλυση, ο στατικός έλεγχος και η παραγωγή ενδιάμεσου κώδικα γίνονται διαδοχικά. Μερικές φορές μπορούν να συνδιαστούν και να ενσωματωθούν στην συντακτική ανάλυση. Όλα τα σχήματα μπορούν να υλοποιηθούν με την δημιουργία ενός συντακτικού δένδρου και στην συνέχεια με την διάσχιση του δένδρου.

Ο στατικός έλεγχος περιλαμβάνει τον έλεγχο τύπων, ο οποίος εξασφαλίζει ότι οι τελεστές εφαρμόζονται σε συμβατούς τελεστές. Περιλαμβάνει επίσης οποιουσδήποτε συντακτικούς ελέγχους που παραμένουν μετά την ανάλυση. Για παράδειγμα, ο στατικός έλεγχος βεβαιώνει ότι μια εντολή break στην C βρίσκεται μέσα σε μια

εντολή while ή for ή switch, αν δεν υπάρχει μια περικλείουσα εντολή αναφέρεται σφάλμα.

Μια συνυθισμένη μορφή αναπαράστασης ενδιάμεσου κώδικα είναι ο κώδικας τριών διευθύνσεων. Ο όρος ‘κώδικας τριών διευθύνσεων’ προέρχεται από τις εντολές με την γενική μορφή x = y op z με τρείς διευθύνσεις : δύο για τους τελεστές y και z και μια για το αποτέλεσμα x.

Στην διαδικασία μετάφρασης ενός προγράμματος μιας δεδομένης πηγαίας γλώσσας σε κώδικα για μια δεδομένη μηχανή στόχο, ένας μεταγλωττιστής μπορεί να κατασκευάσει μια ακολουθία ενδιάμεσων αναπαραστάσεων όπως :



Η υψηλού επιπέδου αναπαραστάσης είναι κοντά στην πηγαία γλώσσα και οι χαμηλού επιπέδου αναπαραστάσεις είναι κοντά στην μηχανή στόχο. Τα συντακτικά δένδρα είναι υψηλού επιπέδου, απεικονίζουν την φυσική ιεραρχική δομή του πηγαίου προγράμματος και ταιριάζουν καλά σε εργασίες όπως ο στατικός έλεγχος τύπων.

Μια αναπαράσταση χαμηλού επιπέδου είναι κατάλληλη για εργασίες που εξαρτώνται από την μηχανή όπως η κατανομή καταχωρητών και η επιλογή εντολών. Ο κώδικας τριών διευθύνσεων μπορεί να κυμαίνεται από υψηλό σε χαμηλό επίπεδο ανάλογα με την επιλογή των τελεστών. Για τις εκφράσεις, οι διαφορές μεταξύ των συντακτικών δένδρων και του κώδικα τριών διευθύνσεων είναι επιφανειακές όπως θα δούμε σε παρακάτω ενότητα. Για τις δηλώσεις βρόχων, για παράδειγμα, ένα συντακτικό δένδρο αναπαριστά τα συστατικά μια εντολής, ενώ ο κώδικας τριών διευθύνσεων περιέχει ετικέτες και εντολές μετάβασης (jump instructions) για να αναπαραστήσει την ροή ελέγχου όπως στην γλώσσα μηχανής.

Η επιλογή ή η σχεδίαση μια ενδιάμεσης αναπαράστασης ποικίλλει από μεταγλωττιστή σε μεταγλωττιστή. Με ενδιάμεση αναπαράσταση μπορεί, είτε να είναι μια πραγματική γλώσσα, είτε μπορεί να αποτελείται από τις εσωτερικές δομές δεδομένων που μοιράζονται από τις φάσεις του μεταγλωττιστή. Η C είναι μια γλώσσα προγραμματισμού, όμως χρησιμοποιείται συχνά ως ενδιάμεση μορφή επειδή είναι ευέλικτη, μεταγλωττίζεται σε αποδοτικό κώδικα μηχανής και οι μεταγλωττιστές της είναι ευρέως διαθέσιμοι. Ο αρχικός μεταγλωττιστής C++ διαθέτει ένα εμπρός τμήμα που παρήγαγε C, και ένα μεταγλωττιστή της C ως πίσω τμήμα.

6.2 Διευθύνσεις και Εντολές

Ο κώδικας τριών διευθύνσεων κατασκευάζεται από δυο έννοιες : τις διευθύνσεις και τις εντολές. Με αντικειμενοστραφής όρους, αυτές οι έννοιες αντιστοιχούν στις κλάσεις, και τα διάφορα είδη διευθύνσεων και εντολών αντιστοιχούν στις κατάλληλες υποκλάσεις. Εναλλακτικά, ο κώδικας τριων διευθύνσεων μπορεί να υλοποιηθεί χρησιμοποιώντας εγγραφές με πεδία για τις διευθύνσεις, οι εγγραφές αποκαλούνται τετράδες (quadruples) και τριάδες (triples).

Μια διεύθυνση μπορεί να είναι ένα από τα ακόλουθα:

- Ένα όνομα. Για ευκολία, επιτρέπουμε στα ονόματα των μεταβλητών των πηγαίων προγραμμάτων να εμφανιστούν ως διευθύνσεις στον κώδικα τριών διευθύνσεων. Σε μια υλοποίηση, ένα πηγαίο όνομα αντικαθίσταται από έναν δείκτη στην εγγραφή του στον πίνακα συμβόλων, όπου κρατούνται όλες οι πληροφορίες για το όνομα.
- Μια σταθερά. Στην πράξη, ένας μεταγλωττιστής πρέπει να χρειάζεται πολλούς διαφορετικούς τύπους σταθερών και μεταβλητών.
- Ένα προσωρινό όνομα παραγόμενο από τον μεταγλωττιστή. Είναι χρήσιμο, ειδικά στους μεταγλωττιστές που πραγματοποιούν βελτιστοποίηση να δημιουργούν ξεχωριστά ονόμα κάθε φορά που απαιτείται ένα προσωρινό όνομα. Αυτά τα προσωρινά ονόματα μπορούν να συνδιαστούν, αν είναι δυνατό, όταν οι καταχωρητές εκχωρούνται σε μεταβλητές.

Θα επεξηγηθούν αρχικά οι συμβολικές ετικέτες (symbolic labels) που θα χρησιμοποιηθούν από τις εντολές που αλλάζουν την ροή ελέγχου. Μια συμβολική ετικέτα αναπαριστά τον αριθμό δείκτη μια εντολής τριών διευθύνσεων μέσα στην

ακολουθία εντολών. Οι πραγματικοί δείκτες μπορούν να αντικατασταθούν με ετικέτες, είτε κάνοντας ένα ξεχωριστό πέρασμα είτε με «οπισθωμπάλωμα» (backpatching). Εδώ είναι μια λίστα των κοινών εντολών τριών διευθύνσεων:

1. Εντολές ανάθεσης της μορφής x = y op z, όπου op είναι μια δυαδική αριθμητική ή λογική λειτουργία και x,y,z είναι οι διευθύνσεις.
2. Αναθέσεις της μορφής x = op y όπου op μια μοναδιαία λειτουργία (unary operation). Οι βασικές μοναδιαίες λειτουργίες περιλαμβάνουν τα μοναδιαία πλην (unary minus) , την λογική άρνηση, τους τελεστές ολίσθησης και τους τελεστές μετατροπής οι οποίοι για παράδειγμα μετατρέπουν έναν ακέραιο αριθμό σε αριθμό κινητής υποδιαστολής.
3. Εντολές αντιγραφής της μορφής x = y όπου στο x εκχωρείτε η τιμή του y.
4. Μια μετάβαση χωρίς συνθήκη goto L. Η εντολή τριών διευθύνσεων με ετικέτα L είναι η επόμενη που εκτελείτε.
5. Μεταβάσεις υπό συνθήκη της μορφής: if x goto L και if false x goto L. Αυτές εντολές εκτελούν την εντολή με ετικέτα L, εαν το x είναι αληθές και ψευδές αντίστοιχα. Διαφορετικά, ως συνήθως, εκτελείτε η επόμενη εντολή τριών διευθύνσεων μέσα στην ακολουθία.
6. Μεταβάσεις υπό συνθήκη όπως if x relop y goto L, που εφαρμόζουν έναν σχεσιακό τελεστή <,==,>=,κτλπ.. στα x και y και εκτελούν την εντολή με ετικέτα L εαν η έκφραση x relop y είναι αληθής. Αν όχι εκτελείτε η εντολή τριών διευθύνσεων που ακολουθεί την if x relop y goto L στην ακολουθία των εντολών.
7. Η κλήσεις διαδικασιών και οι επιστροφές υλοποιούνται χρησιμοποιώντας τις ακόλουθες εντολές: par x για τις παραμέτρους, call p, n και y = call p για κλήσεις διαδικασιών και συναρτήσεων αντίστοιχα και retv y όπου y αναπαριστά μια τιμή επιστροφής η οποία είναι προαιρετική. Η τυπική χρήση τους είναι ως ακολουθία εντολών τριών διευθύνσεων.

par x1

par x2

……

par x_n

call p

που παράγεται ως τμήμα μια κλήσης της διαδικασιας p(x1,x2,…,x_n) . Ο ακέραιος n, δείχνει τον αριθμό πραγματικών παραμέτρων στην «call p,n» και δεν είναι περιττός επειδή οι κλήσεις μπορούν να είναι εμφωλευμένες. Δηλαδή μερικές από τις πρώτες δηλώσεις par θα μπορούσαν να είναι παράμετροι μια κλήσεις που πραγματοποιείτε αφού το p έχει επιστρέψει την τιμή του, αυτή η τιμή γίνεται μια άλλη παράμετρος της επόμενης κλήσης.

8. Εντολές αντιγραφής με αριθμοδείκτες της μορφή x = y[i] και x[i] = y εκχωρεί στο x την τιμή της θέσεις μνήμης που βρίσκεται i μονάδες μνήμης μετά το y. Η εντολή x[i] = y εκχωρεί στην θέση που βρίσκεται i μονάδες μνήμης μετά το x την τιμή του y.
9. Αναθέσεις διευθύνσεων και δεικτών της μορφής x = & y, x = *y και *x = x. Η εντολή x = & y θέτει την τιμή-r (r value) του x να είναι η θέση (τιμή l[l value]) του y. Πιθανώς το y είναι ένα όνομα ίσως προσωρινό που υποδηλώνει μια έκφραση με τιμή l όπως η A[i][j], και το x ένα όνομα δείκτη ή ένα προσωρινό όνομα. Στην εντολή x = *y , πιθανώς το y είναι ένας δείκτης ή κάτι προσωρινό του οποίου η τιμή-r είναι μια θέση. Η τιμή-r του x γίνεται ίση με τα περιεχόμενα εκείνης της θέσης. Τέλος, η *x = y θέτει την τιμή-r του αντικειμένου που δείχνεται από το x στην τιμή-r του y.

6.3 Τετράδες

Η περιγραφή των εντολών τριών-διευθύνσεων καθορίζει τα συστατικά κάθε τύπου εντολής, αλλά δεν καθορίζει την αναπαράσταση αυτών των εντολών σε μια δομή δεδομένων. Σε ένα μεταγλωττιστή, αυτές οι εντολές μπορούν να υλοποιηθούν ως αντικείμενα ή ως εγγραφές με πεδία για τον τελεστή και τους τελεστέους. Τρεις τέτοιες αναπαραστάσεις καλούνται «τετράδες», «τριάδες» και «έμμεσες τριάδες».

Μια τετράδα έχει τέσσερα πεδία, τα οποία καλούμε op,arg_1,arg_2 και result. Το πεδίο op περιέχει έναν εσωτερικό κωδικό για τον τελεστή. Για παράδειγμα, η εντολή τριων διευθύνσεων x = y + z αναπαρίσταται με την τοποθέτηση του + στο op , του y στο arg_1, του z στο arg_2 και του x στο result. Τα παρακάτω είναι μερικές εξαιρέσεις σε αυτόν τον κανόνα :

1. Εντολές με μοναδιαίους τελεστές όπως x = minus y ή x = y δεν χρησιμοποιούν το arg_2. Για μια δήλωση αντιγραφής όπως x=y , το op είναι =, ενώ για τις περισσότερες άλλες λειτουργίες, ο τελεστής ανάθεσης υπονοείται.
2. Τελεστές όπως το param δεν χρησιμοποιούν ούτε το arg_2 ούτε το result.
3. Οι μεταβάσεις υπό συνθήκη και οι μεταβάσεις χωρίς συνθήκη τοποθετούν την ετικέτα στόχο στο result.

6.4 Δομές Ενδιάμεσου Κώδικα

Για την αναπαράσταση του ενδιάμεσου κώδικα χρειαζόμαστε μια δομή που θα αποθηκεύουμε τις τετράδες. Στην υλοίηση μας έχουμε χρησιμοποιήσει συνδεδεμένες λίστες. Για τις ανάγκες της υλοποίησης έχουμε δυο διαφορετικές δομές, η πρώτη υπάρχει για να αποθηκεύουμε όλες τις τετράδες που παράγονται κατά την μεταγλώττιση και η δεύτερη για να αποθηκεύουμε δείκτες προς τις παραπάνω τετράδες ώστε να κάνουμε κάποια τροποποίηση τους αργότερα.

6.4.1 Λίστα Τετράδων (programList)

Η βασική λίστα τετράδων που αποθηκεύουμε όλες τις τετράδες που παράγονται από τον μεταγλωττιστή μας υλοποιείτε με την δομή struct list που περιέχει τα παρακάτω πεδία :

- head Ένας δείκτης που δείχνει στην αρχή της λίστας. (δείκτης σε ένα struct listNode)
- tail Ένας δείκτης που δείχνει στο τέλος της λίστας.
- size Το Μέγεθος της λίστας.

Η δομή struct listNode περιέχει τα παρακάτω πεδία :

- quadNumber Είναι ο αριθμός που προσδιορίζει μοναδικά μια τετράδα.
- op Σε αυτό το πεδίο αποθηκεύουμε την ‘εντολή’ που περιγράφει η τετράδα.
- x Το δεύτερο πεδίο της τετράδας (arg_1)
- y Το τρίτο πεδίο της τετράδας (arg_2)
- z Το τέταρτο πεδίο της τετράδας (result)
- next Δείκτης στο επόμενο κόμβο της λίστας

```
struct listNode
{
        char quadNumber[MAX_QUADS];
        char op[TOKEN_SIZE];
        char x[TOKEN_SIZE];
        char y[TOKEN_SIZE];
        char z[TOKEN_SIZE];
        struct listNode * next;

};

struct list
{
        struct listNode * head;
        struct listNode * tail;
        int size;

};
```

6.4.2 Λίστα Ετικετών Τετράδων

Εδώ θα περιγράψουμε την δομή που μας βοηθάει να ‘σημειώνουμε’ κάποιες τετράδες από την προηγούμε δομή.

Αυτή η λίστα υλοποιείτε με την δομή struct labelList που περιέχει τα παρακάτω πεδία :

- head Ένας δείκτης που δείχνει στην αρχή της λίστας. (δείκτης σε ένα struct LabelListNode)
- tail Ένας δείκτης που δείχνει στο τέλος της λίστας.
- size Το Μέγεθος της λίστας.

Η δομή struct listNode περιέχει τα παρακάτω πεδία :

- data Ένας δείκτης σε μια τετράδα.
- next Δείκτης στο επόμενο κόμβο της λίστας

```
struct labelListNode
{
        struct listNode* data;
        struct LabelListNode *next;
};

struct labelList
{
        struct labelListNode * head;
        struct labelListNode * tail;
        int size;
};
```

6.4.3 Συναρτήσεις Υποστήριξης Των Παραπάνω Δομών

- **allocateList()**

δεσμεύει τον απαραίτητο χώρο που χρειάζεται μια δομή struct list.

- **allocateLabelList()**

δεσμεύει τον απαραίτητο χώρο που χρειάζεται μια δομή struct labelList.

- **addToList()**

εισάγει έναν νέο κόμβο σε μια λίστα τετράδων.

- **addNodeToLabelList()**

εισάγει έναν νέο κόμβο σε μια λίστα ετικετών τετράδων.

- **isEmpty()**

ελέγχει αν η λίστα που παίρνει ως όρισμα είναι κενή.

- **mergeLists()**

συνενώνει τις δυο λίστες που παίρνει ως όρισμα.

- **freeList()**

αποδεσμεύει τον χώρο που έχουμε δεσμεύσει για την list.

- **freeLabelList()**

αποδεσμεύει τον χώρο που έχουμε δεσμεύσει για την labelList.

6.5 Βασικές Συναρτήσεις

Για να επεξηγήσουμε καλύτερα τον τρόπο που παράγουμε τον ενδιάμεσο κώδικα καλό είναι να περιγράψουμε την λειτουργία κάποιων βασικών συναρτήσεων αρχικά.

- **genQuad()**

Αυτή η συνάρτηση θα μας βοηθησει να δημιουργήσουμε μια νέα τετράδα και να την εισάγουμε στην λίστα ετικετών τετράδων.

```
void genQuad(char* op,char* x,char* y, char* z)
{
        addToList(programList,nextQuad(),op,x,y,z);
}
```

- **nextQuad()**

Αυτή η συνάρτηση αυξάνει τον τρέχων αριθμό τετράδων που έχουμε παράγει και μας τον επιστρέφει.

```
int nextQuad()
{
        quadNumber++;
        return quadNumber;
}
```

- **newTemp()**

Αυτή η συνάρτηση δημιουργεί μια νέα προσωρινή μεταβλητή που θα χρησιμοποιηθεί από τον μεταγλωττιστή για την αποθήκευση κάποιας προσωρινής τιμής.

```
void newTemp(char* tempName,struct scope * scope)
{
        struct entity *anEntity;
        sprintf(tempName,"T_%d",countTempVariables);
        countTempVariables++;
        anEntity = createNewEntity(tempName,1,TEMPVARIABLE);
        anEntity->data.tmpVar.offset = scope -> currentOffset;
        scope -> currentOffset += ENTITY_MEM_BYTES;
        addEntityToScope(anEntity,scope);
}
```

- **emptyList()**

Αυτή η συνάρτηση δημιουργεί μια νέα λίστα ετικετών τετράδων και μας επιστρέφει έναν δείκτη προς αυτήν.

```
struct labelList * emptyList()
{
        struct labelList * newLabelList;
        allocateLabelList(&newLabelList);
        return newLabelList;
}
```

- **makeList()**

Αυτή η συνάρτηση μας επιστρέφει μια λίστα ετικετών τετράδων που περιέχει μια την τετράδα με το αναγνωριστικό (quad Number) που της περάσαμε ως είσοδο.

```
struct labelList* makeList(int x)
{
        struct labelList * newLabelList;
        allocateLabelList(&newLabelList);
        addNodeToLabelList(programList,newLabelList,x);
        return newLabelList;
}
```

- **merge()**

Αυτή η συνάρτηση πέρνει ως είσοδο δυο λίστες ετικετών τετράδων και τις ενώνει στην πρώτη. (Στην υλοποιήση μας εδώ δεν φαίνεται ακριβώς πως γίνετε διότι παραπέμπουμε την λειτουργία αυτή σε συνάρτηση που έχουμε υλοποιήσει για να υποστηρίξουμε τις δομές λίστας που χρησιμοποιούμε στο πρόγραμμα μας)

```
void merge(struct labelList *list1,struct labelList *list2)
{
        mergeLists(list1,list2);
}
```

- **backPatch()**

Αυτή η συνάρτηση παίρνει ως είσοδο μια λίστα ετικετών τετράδων και έναν αριθμό(quadNumber) . Πάει και συμπληρώνει σε όλες αυτές τις τετράδες το τελευταίο πεδίο. Τέλος ελευθερώνει την λίστα για την αποδοτικότερη διαχείρηση μνήμης του προγράμαματος.

```
void backPatch(struct labelList **aLabelList,int z)
{
        struct labelListNode *node;

        char convertZtoChar[TOKEN_SIZE];
        sprintf(convertZtoChar,"%d",z);
        if( (*aLabelList)== NULL)
        {
                printf("The list doesnt exist.\n");
                return  ;
        }
        if( (*aLabelList)->head==NULL)
        {
                printf("List is empty.\n");
                return;
        }

        node = (*aLabelList)-> head;
        while(node != NULL)
        {
                strncpy(node->data->z,convertZtoChar,MAX_TOKEN_CHARACTERS);
                node = (struct labelListNode*) node -> next;
        }

        freeLabelList(aLabelList);
}
```

6.6 Παράδειγμα Παραγωγής Ενδιάμεσου Κώδικα

6.6.1 Αποδόμηση της δομής While

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

{P1}: Bquad := nextquad()

{P2}: backpatch(BTrue.nextQuad())

{P3}: genquad(“jump”, “_”, “_”, Bquad)

backpatch(BFalse,nextQuad())

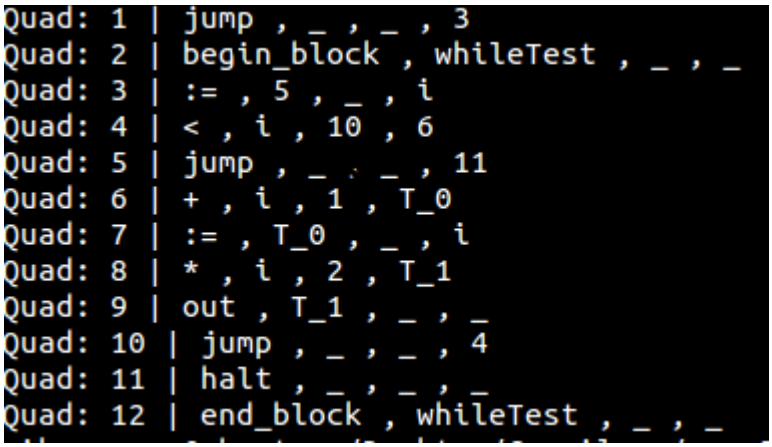
Η Υλοποίηση μας για την εντολή while:

```
void whileStat(struct scope *currentScope)
{
        int Bquad=0;
        char repQuad[TOKEN_SIZE];
        struct labelList *condTrueList = NULL, *condFalseList = NULL;
        int condQuad = quadNumber + 1 ;
        condition(&condTrueList,&condFalseList,currentScope);
        if(token_id == doTk)
        {
                Bquad = quadNumber + 1 ;
                sprintf(repQuad,"%d",Bquad);
                backPatch(&condTrueList,Bquad);

                token_id = lex();
                blockOrStat(currentScope);
                sprintf(repQuad,"%d",condQuad);
                genQuad("jump","_","_",repQuad);
                backPatch(&condFalseList,quadNumber + 1);

        }
        else
        {
                printf("%s :Syntax Error:%d:The keyword 'do' was expected.\n"
                ,file_name,currentLineInFile);
                printSpaces();
                printf("Proper syntax for \"while\"is: while <condition> do "
                "<block or statement>.\n");
                customExit();
        }
}
```

Για το παρκάτω while loop:

```
program  whileTest
var i;

begin
        i:= 5;
        while i < 10 do
        begin
                i := i+1;
                print(i*2)
        end
end
```

Η λίστα με τις τετράδες του ενδιάμεσου κώδικα:



6.6.2 Αποδόμηση της δομής For

for id := exp_0 **{P1}** to exp_1 **{P2}** step exp_2 **{P3}**

begin **{P4}** *Sequence* **{P5}** end

{P1}: *genQuad(":=", exp_0 ,_, id)*

Με αυτήν την εντολή εκχωρούμε στην εντολή διάσχησης του for την αρχική τιμή που δίνει ο χρήστης.

{P2}: *conQuad = nextQuad();*

asc = makeList(nextQuad());

genQuad("<", exp_0 ,exp_1 , "_");

desc = makeList(nextQuad());

genQuad(">", exp_0 ,exp_1 , "_");

eq = makeList(nextQuad());

genQuad("=", exp_0 ,exp_1 , "_");

Σε αυτό το κομμάτι θέλουμε ουσιαστικά να καταλάβουμε αν η εντολή for που αποδομούμε θα έχει loop που θα αυξάνεται ή που θα μειώνεται ώστε να διαμορφώσουμε κατάλληλα την συνθήκη που θα πρέπει να ελέγχεται σε κάθε επανάληψη. Γι' αυτό δημιουργούμε τις παραπάνω τρεις λίστες asc (ascend), desc (descend) και eq(equal) για να κάνει jump στην σωστή τετράδα ελέγχου.

batchPatch(asc,nextQuad());

condTrue = makeList(nextQuad());

genQuad("<=", id , exp₁, "_");

condFalse = makeList(nextQuad());

genQuad("jump","_","_","_")

backPatch(desc,nextQuad())

t = makeList(nextQuad())

merge(condTrue,t)

genQuad(">=", id , exp₁, "_")

t = makeList(nextQuad())

merge(condFalse,t)

genQuad("jump","_","_","_";

backpatch(eq, nextQuad())

t = makeList(nextQyad())

merge(condTrue,t)

genQuad("=", id , exp₁, "_");

t = makeList(nextQuad())

merge (condFalse,t)

genQuad("jump","_","_","_")

stepQuad = nextQuad()

Σε αυτό το κομμάτι παράγεται κώδικας για την κάθε περίπτωση του loop όπως εξηγήθηκε παραπάνω.

{P3}:

genQuad(“t”,”id”,”exp_1”,”id”)

genQuad(“jump”,”_”,”_”,”condQuad”)

{P4}:

backPatch(condTrue, nextQuad)

{P5}:

genQuad(“ jump” , “_”, “_”, stepQuad)

backPatch(condFalse, nextQuad()

Η Υλοποίηση μας για την εντολή for

```
void forStat(struct scope *currentScope){
        char condQuadS[TOKEN_SIZE],stepQuadSt[TOKEN_SIZE] ;
        char Eplace[TOKEN_SIZE],ASSIGNMENTplace[TOKEN_SIZE];
        char valuePlace[TOKEN_SIZE],STEPplace[TOKEN_SIZE];
        struct labelList * condTrue = NULL,* condFalse = NULL;
        struct labelList * asc = NULL,* desc = NULL;
        struct labelList * eq = NULL, * t = NULL;
        int condQuad = 0,stepQuad = 0;

        assignmentStat(ASSIGNMENTplace,valuePlace,currentScope);
        if(token_id == toTk){
                token_id = lex();
                expression(Eplace,currentScope);

                condQuad = quadNumber + 1;
                genQuad("<", valuePlace ,Eplace, "_");
                asc = makeList(quadNumber);
                genQuad(">", valuePlace ,Eplace, "_");
                desc = makeList(quadNumber);
                genQuad("=",valuePlace ,Eplace, "_");
                eq = makeList(quadNumber);
                backPatch(&asc,quadNumber+1);
                genQuad("<=",ASSIGNMENTplace,Eplace,"_");
                condTrue = makeList(quadNumber);
                genQuad("jump","_","_","_");
                condFalse = makeList(quadNumber);
                backPatch(&desc,quadNumber+1);
                genQuad(">=",ASSIGNMENTplace,Eplace,"_");
                t = makeList(quadNumber);
                merge(condTrue,t);
                genQuad("jump","_","_","_");
                t = makeList(quadNumber);
                merge(condFalse,t);
                backPatch(&eq,quadNumber+1);
                genQuad("=", ASSIGNMENTplace ,Eplace, "_");
                t = makeList(quadNumber);
                merge(condTrue,t);
                genQuad("jump","_","_","_");
                t = makeList(quadNumber);
                merge(condFalse,t);
                stepQuad = quadNumber + 1;

                stepPart(STEPplace,currentScope);

                sprintf(condQuadS,"%d",condQuad);
                genQuad("+",ASSIGNMENTplace,STEPplace,ASSIGNMENTplace);
                genQuad("jump","_","_",condQuadS);
                backPatch(&condTrue,quadNumber+1);

                blockOrStat(currentScope);

                sprintf(stepQuadSt,"%d",stepQuad);
                genQuad("jump","_","_",stepQuadSt);
                backPatch(&condFalse,quadNumber+1);
        }
        else{
                printf("%s :Syntax Error:%d:The keyword 'to' was expected.\n"
                ,file_name,currentLineInFile);
                printSpaces();
```

Για το παρακάνω for loop :

```
program forTest
var i;

begin
        for i := 10 to 0 step -2
                print(i)
end
```

Η λίστα με τις τετράδες :

```
Quad: 1 | jump , _ , _ , 3
Quad: 2 | begin_block , forTest , _ , _
Quad: 3 | := , 10 , _ , i
Quad: 4 | < , 10 , 0 , 7
Quad: 5 | > , 10 , 0 , 9
Quad: 6 | = , 10 , 0 , 11
Quad: 7 | <= , i , 0 , 15
Quad: 8 | jump , _ , _ , 17
Quad: 9 | >= , i , 0 , 15
Quad: 10 | jump , _ , _ , 17
Quad: 11 | = , i , 0 , 15
Quad: 12 | jump , _ , _ , 17
Quad: 13 | + , i , -2 , i
Quad: 14 | jump , _ , _ , 4
Quad: 15 | out , i , _ , _
Quad: 16 | jump , _ , _ , 13
Quad: 17 | halt , _ , _ , _
Quad: 18 | end_block , forTest , _ , _
```

ΚΕΦΑΛΑΙΟ 7 : ΠΙΝΑΚΑΣ ΣΥΜΒΟΛΩΝ

7.1 Γενικά

Κατά την διάρκεια της μεταλώτισσης υπάρχει ανάγκη για την συγκέντρωση πληροφοριών σχετικών με τα ονόματα (identifiers) τα οποία εμφανίζονται μέσα στον κώδικα του προγράμματος εισόδου.Η πληροφορία αυτή κατά τη φάση της μετάφρασης κράτιεται στην πίνακα συμβόλων.

Ο **πίνακας συμβολων** είναι μία δομή δεδομένων που χρησιμοποιείται απο έναν μεταφραστή(ή διερμινέα) .Στην δομή αυτή για κάθε αναγνωριστικό μέσα στον κώδικα του προγράμματος το οποίο πρόκειται να μεταφραστεί υπάρχουν πληροφορίες οι οποίες σχετίζονται με την δήλωση την εμφανιση ,την εμβέλεια , το μέγεθος που καταλαμβάνει στην μνήμη.Οι κατηγορίες μεταβλητών για τις οποίες κραταμε πληροφορία στον πίνακα συμβόλων είναι οι εξείς:

- Μεταβλητές (int x , string y κλπ.)
- Υποπρογράμματα(διαδικασίες και συναρτήσεις)
- Παραγματικες παράμετροι υποπρογραμμάτων
- Τυπικες παράμετρροι υποπρογραμμάτων
- Στάθερές
- Τύποι δεδομένων

Η πληροφορία πόυ αποθηκεύεται στον πίνακα συμβόλων χρησιμοποιείται κατά την σημασιολογική ανάλυση(έλεγχος τύπων) και κατα την παραγωγή του τελικού κώδια (πχ.πόσο χώρο στην μνήμη απαιτεί ενα αναγνωριστικό).Για παράδειγμα ας πούμε ότι δηλώνεται μαι μεταβλητή ως int y. Στον πίνακα συμβόλων απόθηκεύεται ότι η y είναι τύπου ακέραιου.Κάθε φόρα που απαιτείται το x είναι ακέραιος πρέπει να γίνεται ανάκληση απο τον πίνακα συμβόλων(πχ κατά την ανάθεση τιμής στην μεταβλητή x).

Ο πίνακας συμβόλων κατασκευάζεται κατά τη φάση της ανάλυσης, δηλαδή κατά τη λεκτική ή συντακτική και σημασιολογική ανάλυση.Κατά τη λεκτική ανάλυση είναι δυνατό να δημιουργηθεί μία νέα θέση στον πίνακα που να περιέχει αρχικές πληροφορίες για το σύμβολο.Το πιο σύνηθες όμως είναι η δημιουργία τον εγγραφών του πίνακα συμβόλων κατά την συντακτική ανάλυση.Αυτό συμβαίνει γιατί σε αυτή τη φάση της μετάφρασης υπάρχει σχεδόν όλη η πληροφορία που χρειάζεται για το κάθε σύμβολο.

7.2 Εμβέλεια , Ορατότητα και Πίνακας Συμβόλων

Σε αυτή την φάση είναι σωστό να ανεφαίρουμε καποιές εννοιές τις οποίες ο πίνακας συμβολών ορίζει για τα αναγνωριστικά και επειρεάζουν καθώς και διαμορφώνουν το χαρακτήρα της γλώσσας που κατασκευάζουμε.Τέτοιες έννοιες ειναι η εεβέλεια ,η ορατότητα, η διάρκεια ζωής ενός αναγνωριστικού(μεταβλητή, υποπρόγραμμα κλπ).

Ως **εμβέλεια** (ή περιοχή εμβλέλειας) ορίζουμε μια δομική μονάδα προγράμματος η οποία περέχει δηλώσεις μιας ή περισσότερων μεταβλητών.Με τον όρο δομική μονάδα αναφερόμαστε σε ένα υποπρόγραμμα μία συνάρτηση κλπ.Στους μεταφράστες υπάρχουν δυο είδη εμβέλειας .

- **Δυναμική** εμβέλεια(dynamic scope): όταν η απόφαση για το σε ποια μεταβλητή αναφέρεται ένα όνομα βρίσκεται σε πληροφορία που είναι διαθέσιμη μόνο κατά την εκτέλεση.
- **Στατική** εμβέλεια(static scope): όταν η απόφαση για το σε ποια μεταβλητή αναφέρεται ενα όνομα

Στις περισσότερες γλώσσες προγραμματισμού επιτρέπεται η ύπαρξη φωλιασμένων δομικώβ μονάδων(block structure).Αυτό σημαίνει πως μπρούν να υπάρχουν υποπρογράμματα το ένα δηλώμενο μέσα στο άλλο παραδείγματος χαρη μία συνάρτηση που δηλώνεται μεσα σε κάποια άλλη.Τότε λέμε πως έχουμε την συνάρτηση παιδί και την συνάρτηση γονιό αντίστοιχα.Με αυτό τον τρόπο, η φωλιασμένες εμβέλειες ορίζουν ένα δέντρο εμβελειών το οποίο καθορίζει την ορατοτητα της κάθε μεταβλητής και δομικής μονάδας.

Η **Ορατότητα** μιας μεταβλητής είναι η περιοχή στην οποία το ονομά της μεταβλητής αναφέρεται στη συγκεκριμένη μεταβλητή.Μία μεταβλητή είναι ορατή στην εβλέλεια

s που δηλώνεται από το σημείο δήλωσης και μετά καθώς και σε κάθε φωλιασμένη εμβέλεια s_n αν:

- η s_n βρίσκεται μετά τη δήλωση της μεταβλητής στην s
- Το όνομα της μεταβλητής δεν δηλώνεται ξανά στην s_n ή σε κάποια φωλιασμένη εβλέλεια s_p της s που περιέχει την s_n.

Με βάση τις δύο θεμελιώδεις έννοιες που είδαμε παραπάνω μπορούμε να κατηγοριοποιήσουμε τις μεταβλητές ενός προγράμματος ως εξής :

- Καθολικές μεταβλητές(global).Μεταβλήτες που έχουν την μέγιστη δυνατή εμβέλεια και κατά βαση είναι ορατές από ολα τα σημεία του προγράμματος σε καθε δομική μονάδα.
- Μεταβλητές Στοίβας(stack):Η εμβέλεια και διάρκεια ζωής είναι ίδια με τη συγκεκριμένη δομική μονάδα που τις περιέχει.
- Στατικές Μεταβλητές.(static) : διατηρούν την τιμή τους μεταξύ διαδοχικών κλήσεων της δομικής μονάδας στην οποία ανήκουν.
- Νεότερες γλώσσες: public, protected, private κλπ.

Τέλος αναφαίρουμε ότι σαν Διάρκεια ζωής ενός χαρακτηριστικού που βρίσκεται στον πίνακα συμβόλων είναι χρονικό διάστημα από τη στιγμή που δεσμεύεται μνήμη μέχρι αυτή να ελευθερωθεί.

7.3 Πληροφορία στον Πίνακα Συμβόλων

Για κάθε διαφορετικό τύπο χαρακτηριστικού(identifier) που υπάρχουν στον πίνακα συμβόλων κρατιέται και ένα συγκεκριμένο σύνολο πληροφοριών.

Για κάθε μεταβλητή στον πρόγραμμα δημιουργείται μία δομή πληροφορίας που περιέχει το όνομα της μεταβλητής , τον τύπο της καθώς και την θέση της στην μνήμη(offset).Κάθε παράμετρος έχει ενα τρόπο περάσματος(passing mode) πχ μέσω τιμής, μέσω αναφοράς με τιμή αποτέλεσμα κλπ.Για τα ονόματα των διαδικάσιων/συναρτήσεων υπάρχει το όνομα, ο αριθμός των παραμέτρων, ο τύπος των παραμέτρων, τυπος της συνάρτησης(τι τυπο θα επιστρέψει) καθώς και το μέγεθος

του εγγραφήματος δραστηριοποίηση(frame length)Επίσης για τα υποπρογράμματα κρατάμε και την αρχική ετικέτα δηλάδη την πρώτη εκτελέσιμη εντολή σε ενδιάμεσο κώδικα.Πχ στην υλοποιησή μας την αρχική τετράδα του καθε υποπρογράμματος.

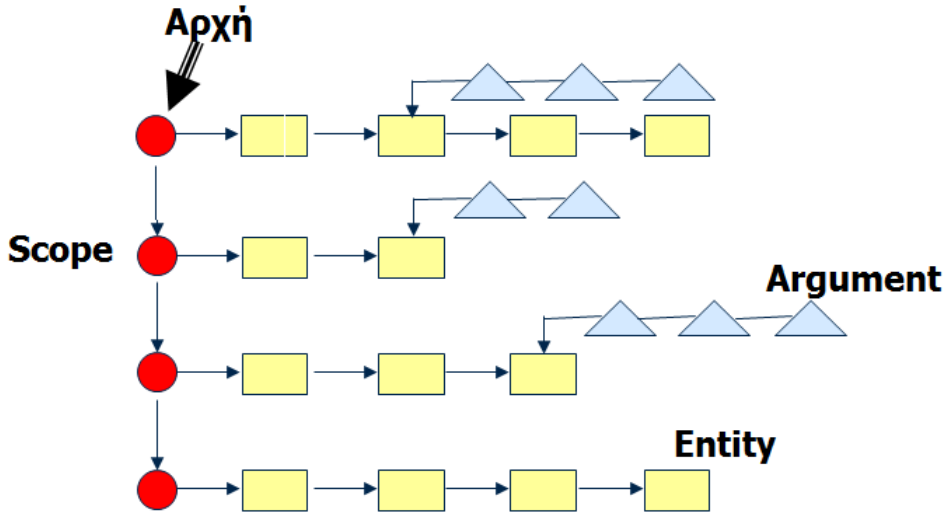
7.4 Τρόποι υλοποίησης Πίνακα Συμβόλων

Κατα τη διάρκεια της μετάφρασης υπάρχει πάρα πολυ μεγάλη προσπέλαση του πίνακα συμβόλων.Η τρόπος με τον οποίο επιλέγουμε την υλοποίηση του είναι πολύ σημαντικός και παίζει δραστικό ρόλο στην απόδοση του μεταγλωτιστή .Ο πίνακας συμβόλων πρέπει να παρέχει αποδοτικά λειτουργίες που έχουν να κάνουν με την γρήγορη εύρεση,αποθήκευση πληροφοριών για τα αναγνωριστικά.Επίσης όταν οι πληροφορίες δεν είναι πια χρήσιμες πχ η μεταγλώτισση μιας συνάρτησης ευτασε στο τέλος της τότε πρεπει να σβήνεται και το κομμάτι του πίνακα που αφορά αυτή τη συνάρτηση.

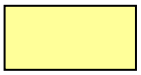
Καποιοί συνήθεις τρόποι υλοποίησης παραθέτονται παρακάτω:

- Σειριακές λίστες : Εύκολη και απλή υλοποίηση.Μεγάλο κόστος διαγραφής και εισαγωγής.
- Δυαδικά δέντρα: Σε κάθε κόμβο όλοι οι κόμβοι που λεξικογραφικά προηγούνται βρίσκονται αριστερά,ενώ οι κόμβοι που έπονται βρισκονταί δεξια του κόμβου.Γρηγορότερη αναζήτηση.
- Πίνακες κατακερματισμού: Βασίζεται στην ύπαρξη δύο πινάκων κατακερματισμού.Πολυ αποδοτική μέθοδος.

7.5 Ο Πίνακας Συμβόλων της Mini Pascal



Ο πίνακας συμβόλων που υλοποιήσαμε στα πλαίσια της εργάσιας υπάρχει στα αρχεία symbolTableStructures.c kai symbolTableStructures.h . Για την υλοποίηση του χρησιμοποιήσαμε γραμμικές λίστες.Πριν περάσουμε στην επέξηγηση των βασικών σημείων στον κώδικα θα εξηγήσουμε τις βασικές εγγραφές του πίνακα συμβόλων.

7.6 Η εγγραφή Entity 

Η συγκεκριμένη εγγραφή είναι η πιο σημαντική εγγραφή του πίνακα συμβόλων.Σε αυτη την εγγραφή η οποία υλοποιείται ως ενα union αποθηκεύονται οι πληροφορίες για κάθε χαρακτηριστικό(identifier).Καθε αόλα τα entities και κάποια μοναδικά πεδια για τον συγκεκριμένο τύπο.**Κάθε entity** δηλαδή κάθε αναγνωριστικό έχει **Όνομα ,Τύπο, Τύπο ενότητας** εναν **Pointer** στο επομενο entity. Επίσης τα entities έχουν και ένα **union entityData** το οποίο περιγράφεται από τις παρακάτω δομές(structs).

Τα επιμέρους πεδία για κάθε αναγνωριστικό ειναι :

Μεταβλητες:

- Τύπο
- Offset (απόσταση από την κορυφή της στοίβας)

```
struct variable
{
        int offset;
};
```

Συνάρτηση:

- Τύπο
- Offset
- Start Quad (ετικέτα της πρώτης τετράδας κώδικα της συνάρτησης.
- Λιστα παραμετρων(βοηθητικη λίστα που κρατάει για κάθε παράμετρο τον τύπο της)
- Framelength (το μήκος του εγγραφήματος δραστηριοποίησης.

```
struct function
{
        char startQuad[TOKEN_SIZE];
        struct parametersList* arguments;
        int frameLength;
};
```

Σταθερά:

- Value (η τιμή της σταθεράς)

```
struct constant
{
        char value[TOKEN_SIZE];
};
```

Παράμετρος: 

- Mode (πεδίο που κρατάει τον τύπο περάσματος Ref,byValue κλπ)
- Offset

```
struct parameter
{
        int mode;
        int offset;
};
```

Προσωρινές μεταβλητές:

- Offset

```
struct tempVariable
{
        int offset;
};
```

Παρακάτω παραθέτεται ο κώδικας για τα entities του πίνακα συμβόλων:

```
struct entity
{
        char name[TOKEN_SIZE];
        int type;
        int typeOfEntity;
        union entityData
        {
                struct variable var;
                struct function fun;
                struct constant cons;
                struct parameter par;
                struct tempVariable tmpVar;

        }data;
        struct entity* next;
};
```

7.7 Η εγγραφή Scope

Η εγγραφή Scope του πίνακα συμβόλων εμπεριέχει όλες τις ενότητες(entities) που περιγράφουν μία δεδομένη συνάρτηση.Είναι ουσιαστικά ο συνδετικός κρίκος μεταξύ των συναρτήσεων και παίζει πολύ σημαντικό ρόλο γιατί καθορίζει τους κανόνες εμβέλειας και την ορατότητα.

Η εγγραφη Scope περιέχει αναλυτικά τα παρακάτω πεδία:

- Βάθος φωλιάσματος(nestingLevel).Πεδίο που περιέχει την πληροφορία του πόσα scope απέχουν από την αρχική συνάρτηση.
- Current offset .Το μέγεθος του scope.
- Δείκτη στο επόμενο Scope.
- Δείκτη στο προηγούμενο Scope.
- Δείκτη στην πρώτη entity της τρέχουσας συνάρτησης.

Παρακάτω φαίνεται ο κώδικας της δομής Scope:

```
struct scope
{
        int nestingLevel;
        int currentOffset;
        struct scope *nextScope;
        struct scope *prevScope;
        struct entity *head;
};
```

7.8 Συναρτήσεις Προσπέλασης Πίνακα Συμβόλων

Έχοντας εξηγήσει τη βασική δομή του πίνακα και τα επιμέρους στοιχεία του τώρα θα δούμε τις βασικές μεθόδους που υλοποιήσαμε για την προσπέλαση του πίνακα.Οι μέθοδοι αυτοί χωρίζονται σε τρεις κατηγορίες.Αυτές που δημιουργούν τις εγγραφές του πίνακα, αυτές που βρίσκουν ενα αναγνωριστικό(identifier) μεσα στον πίνακα, και αυτές που αποδεσμεύουν κάποιες εγγραφές του.Παρακάτω θα παραθέσουμε ενα μικρό σύνολο από αυτες τις μεθόδους.Ο κώδικας τους βρίσκεται στο αρχειο symbolTableStructures.c .

- Συναρτήσεις δημιουργίας εγγραφών:
    - createNewEntity()
    - allocateScope()
    - addEntityToScope()
    - κα.

```
struct entity* createNewEntity(char* name,int type,int typeOfEntity)
{

        struct entity* newEnt;
        newEnt = malloc(sizeof(struct entity));
        strncpy(newEnt->name,name,MAX_TOKEN_CHARACTERS);
        newEnt->type = type;
        newEnt->typeOfEntity = typeOfEntity;
        newEnt->next = NULL;
        return newEnt;
}
```

- Συναρτήσεις εύρεσης αναγνωριστικών
  - findEntity()
  - findVariableInScope()
  - findGlobalVariable()
  - findGlobalConstasnt()
  - findLocalOrTemporatyParameter()
  - findByReferenceParameter()
  - κ.α.

```
int findLocalOrTemporaryOrParameter(char * variableName,struct scope *aScope){

        if( aScope != NULL)
        {
                struct entity * node;
                if( aScope -> head != NULL)
                {
                        node = aScope->head;
                        while(node != NULL)
                        {
                                if(strcmp(node -> name ,variableName) == 0 ){

                                        if(node->typeOfEntity == VARIABLE)
                                                return node->data.var.offset;
                                        else if(node->typeOfEntity ==
                                                                        PARAMETER ){
                                                if(node->data.par.mode ==
                                                                                BYVALUE)
                                                        return node->data.
                                                                        par.offset;
                                        }
                                        else if(node -> typeOfEntity == TEMPVARIABLE)
                                                return node->data.tmpVar.offset;
                                }
                                node = node->next;
                        }
                }
        }
        return 0;
}
```

- Συναρτήσεις αποδέσμευσης εγγραφών.
  - freeScope()

7.9 Ο Πινακας συμβόλων στη μεταγλώτισση

Ολες οι παράπανω δομές του πίνακα καθώς και οι συναρτήσεις χρησιμοποιούνται κατα τη διάρκεια της μεταγλώττισης .Ο πίνακας γεμίζει τις εγγραφές του παράλληλα με την συντακτική ανάλυση ενώ οι συναρτήσεις για την

εύρεση των επιμέρους αναγνωριστικών καλούνται για την παραγωγή του τελικού κώδικα.Επίσης ο πίνακας συμβόλων βοηθάει στην σημασιολόγικη ανάλυση κάθως κρατάει πληροφορίες για απάραιτητες για την ανάγνωριση της μονάδικοτητας ενός αναγνωριστικού και την διατήρηση των σημασιολογικών κανόνων.

Γενίκα στην αρχή της συντακτικής ανάλυσης δημιουργέιται το πρώτο 'βασικό' Scope(mainScope) που θα περιέχει πληροφορίες για την 'main' συνάρτηση του προγράμματος μας.Επιπλέον, μας βοηθάει στον τρόπο που συνδέουμε τα νέα scope που δημιουργούνται για τις υπόλοιπες συναρτήσεις διατηρώντας τους κανόνες εμβέλειας που ορίστηκαν παραπάνω.Επίσης κάθε κανόνας(συνάρτηση) του συντακτικού αναλυτή μόλις αναγνωρίσει μία πληροφόρια που χρειάζεται να προστεθέι στον πίνακα συμβόλων δημιουργεί την εγγραφή και προσθέτει την πληροφορία στον scope της συνάρτησης που μεταγλωττίζεται εκείνη την στιγμή.

Στο τέλος της μεταλώττισης μιας συνάρτησης ο πίνακας συμβόλων έχει όλες τις απαραίτητες πληροφορίες που χρείαζονται για την παραγωγή του τελικού κώδικα.Ο τελικός κώδικας χρησιμοποιεί αυτές τις πληροφορίες , μόλις τελειώσει πρέπει να διαγραφεί το κομμάτι του πίνακα συμβόλων που αφορά σε αυτή τη συνάρτηση.Αυτό συμβαίνει έτσι ώστε να γίνεται αποδοτική διαχείρηση της μνήμης και να μην υπάρξει σύγχυση μεταξύ ονόματων τοπικών μεταβλητών.

ΚΕΦΑΛΑΙΟ 8 : Παραγωγή τελικού κώδικα

Η τελική φάση στο μοντέλο του μεταγλωττιστή μας είναι ο παραγωγός κώδικα. Δέχεται ως είσοδο την ενδιάμεση αναπαράσταση όπου παράγεται από το μετωπιαίο τμήμα(front-end) του μεταγλωττιστή, μαζί με τις σχετικές πληροφορίες του πίνακα συμβόλων, και παράγει ως έξοδο ένα σημασιολογικά ισοδύναμο τελικό πρόγραμμα.

Οι απαιτήσεις που επιβάλλονται σε έναν παραγωγό κώδικα είναι αυστηρές. Το τελικό πρόγραμμα πρέπει να διατηρήσει το σημασιολογικό νόημα του πηγαίου προγράμματος και να είναι υψηλής ποιότητας. Δηλαδή, πρέπει να κάνει αποτελεσματική χρήση των διαθέσιμων πόρων της μηχανής στόχου. Επιπλέον ο ίδιος ο παραγωγός κώδικας πρέπει να εκτελείται αποδοτικά.

Η πρόκληση είναι ότι, απο μαθηματική άποψη το πρόβλημα της παραγωγής ενός βέλτιστου τελικού προγράμματος για ένα δεδομένο πηγαίο πρόγραμμα δεν ορίζεται. Πολλά από τα υποπροβλήματα που αντιμετωπίζονται στην παραγωγή κώδικα όπως η κατανομή των καταχωρητών είναι υπολογιστικά δύσκολο να αντιμετωπιστούν αποδοτικά. Στην πράξη, πρέπει να αρκεστούμε στις ευριστικές (heuristic) τεχνικές που παράγουν καλό αλλά όχι απαραιτήτος βέλτιστο κώδικα. Ευτυχώς, οι ευριστικές τεχνικές έχουν ωριμάσει αρκετά με αποτέλεσμα ένας προσεκτικά σχεδιασμένος παραγωγός κώδικα να μπορεί να παράγει κώδικα που είναι πολλές φορές γρηγορότερος από τον κώδικα που παράγεται από έναν απλοικό παραγωγό κώδικα.

Οι μεταγλωττιστές που πρέπει να παράγουν αποδοτικά τελικά προγράμματα, περιλαμβάνουν μια φάση βελτιστοποίησης πριν από την παραγωγή κώδικα. Ο βελτιστοποιητής αντιστοιχίζει την ενδιάμεση αναπαράσταση σε ενδιάμεση αναπαράσταση από την οποία μπορεί να παραχθεί περισσότερο αποδοτικός κώδικας. Γενικά, η βελτιστοποίηση κώδικα και η φάση παραγωγής κώδικα ενός μεταγλωττιστή συχνά αναφέρονται ως το νοτιαίο τμήμα (backend), μπορούν να κάνουν πολλαπλά περάσματα στην ενδιάμεση αναπαράσταση πρωτού παραχθεί το τελικό πρόγραμμα.

Ένας παραγωγός κώδικα πραγματοποιεί τρεις βασικές εργασίες: την επιλογή εντολών, την κατανομή και ανάθεση καταχωρητών και την διάταξη των εντολών. Η επιλογή εντολών περιλαμβάνει την επιλογή κατάλληλων εντολών της τελικής μηχανής για να υλοποιήσει τις εντολές της ενδιάμεσης αναπαράστασης. Η κατανομή και ανάθεση καταχωρητών περιλαμβάνει την απόφαση του ποιες τιμές πρέπει να

κρατηθούν και σε ποιους καταχωρητές. Η διάταξη εντολών περιλαμβάνει την απόφαση της σειράς με την οποία θα εκτελεστούν οι εντολές.

8.1 Θέματα της σχεδίασης ενός παραγωγού κώδικα

Ενώ οι λεπτομέρειες εξαρτώνται από τα χαρακτηριστικά της ενδιάμεσης αναπαράστασης, την τελική γλώσσα και το σύστημα εκτέλεσης, εργασίες όπως η επιλογή εντολών, η κατανομή και η ανάθεση καταχωρητών και η διάταξη εντολών αντιμετωπίζονται κατά τον σχεδιασμό σχεδόν όλων των παραγωγών κώδικα.

Το σημαντικότερο κριτήριο για έναν παραγωγό κώδικα είναι να παράγει τον ορθό κώδικα. Η ορθότητα είναι ιδιαίτερης σημασίας λόγω του αριθμού των ιδιαίτερων περιπτώσεων που ενδέχεται να αντιμετωπίσει ένας παραγωγός κώδικα. Δεδομένης της σημασίας της ορθότητας, η σχεδίαση ενός παραγωγού κώδικα που να μπορεί εύκολα να υλοποιηθέι, να ελεγχθέι και να συντηρηθεί είναι ένας σημαντικός σχεδιαστικός στόχος.

8.2 Είσοδοι του παραγωγού Κώδικα

Η είσοδοι του παραγωγού κώδικα είναι η ενδιάμεση αναπαράσταση του πηγαίο προγράμματος που παράγεται από το εμπρός τμήμα, μαζί με τις πληροφορίες στον πίνακα συμβόλων που χρησιμοποιούνται για να καθορίσουν τις διευθύνσεις κατά την διάρκεια της εκτελέσης των αντικειμένων δεδομένων που υποδηλώνονται από τα ονόματα στην Ενδιάμεση Αναπαράσταση.

Οι πολλές επιλογές για την ενδιάμεση αναπαράσταση περιλαμβάνουν τις αναπαραστάσεις τριών διευθύνσεων όπως τριάδες, τετράδες, έμμεσες τριάδες, τις αναπαραστάσεις εικονικών μηχανών όπως byteCodes και κώδικα σωρού μηχανής, τις γραμμικές αναπαραστάσεις όπως η επιθεματική σημιογραφεία και τις γραφικές αναπαραστάσεις όπως τα συντακτικά δένδρα.

8.3 Το τελικό πρόγραμμα

Η αρχιτεκτονική του συνόλου εντολών της τελικής μηχανής έχει σημαντική επιρροή στην δυσκολία κατασκευής ενός καλού παραγωγού κώδικα ο οποίος παράγει κώδικα μηχανής υψηλής ποιότητας. Οι πιο κοινές αρχιτεκτονικές τελικής μηχανής είναι ο

RISC (Reduced Instruction Set Computer), CISC (Complex Instruction Set Computer) και αυτές που είναι βασισμένες σε στοίβα.

Μια μηχανή RISC τυπικά έχει πολλούς καταχωρητές εντολών τριών διευθύνσεων, απλούς τρόπους διευθυνσιοδότησης και μια σχετικά απλή αρχιτεκτονική συνόλου εντολών. Αντίθετα μια μηχανή CISC τυπικά έχει λίγους καταχωρητές , εντολές δυο διευθύνσεων, μια ποικιλία τρόπων διευθυνσιοδότησης, αρκετές κατηγορίες καταχωρητών, εντολές μεταβλητού μήκους και εντολές με παράπλευρες ενέργειες.

Σε μια μηχανή βασισμένη σε στοίβα, οι λειτουργίες γίνονται τοποθετώντας τους τελεστές στην στοίβα και στην συνέχεια εκτελώντας τις λειτουργίες με τους τελεσταίους στην κορυφή της στοίβας. Για να επιτύχουμε υψηλή απόδοση η κορυφή της στοίβας τυπικά αποθηκεύεται σε καταχωρητές. Οι μηχανές που βασίζονται σε στοίβα, σχεδόν εξαφανίστηκαν επειδή έγινε αισθητό ότι η οργάνωση στοίβας ήταν αρκετά περιοριστική και απαιτούσε πάρα πολλές λειτουργίες εναλλαγής και αντιγραφής.

Ωστόσο, οι βασισμένες σε στοίβα αρχιτεκτονικές αναβίωσαν με την εμφάνιση της εικονικής μηχανής της Java JVM – Java Virtual Machine. Η JVM είναι ένας διερμηνευτής σε λογισμικό για τα byteCodes της Java, μια ενδιάμεση γλώσσα παράγεται από τους μεταγλωττιστές της java. Ο διερμηνευτής παρέχει συμβατότητα λογισμικού για πολλές πλατφόρμες, γεγονός που αποτέλεσε σημαντικό παράγοντα επιτυχίας της Java.

Για να υπερνικηθεί το υψηλό κόστος που δέχεται η απόδοση λόγω της διερμηνείας, η οποία μπορεί να είναι πολλαπλάσια μια τάξης του 10 δημιουργήθηκαν οι μεταγλωττιστές JIT (Just In Time) της Java. Αυτοί οι μεταγλωττιστές μεταφράζουν κατά την διάρκεια της εκτέλεσης τα BytesCodes στο τοπικό σύνολο εντολών που παρέχει το υλικό της τελικής μηχανής. Μια άλλη προσσέγγιση για να βελτιώσουμε την απόδοση της Java είναι η κατασκευή ενός μεταγλωττιστή που μεταφράζει άμεσα σε εντολές μηχανής της τελικής μηχανής, παρακάμπτωντας τα byteCodes της Java.

Η παραγωγή ως εξόδου ενός προγράμματος σε απόλυτη γλώσσα μηχανής έχει το πλεονέκτημα ότι μπορεί να τοποθετηθεί σε μια συγκεκριμένη θέση στην μνήμη και να εκτελεστεί αμέσως. Τα τελικά προγράμματα μπορούν να μεταγλωττιστούν και να εκτελεστούν γρήγορα.

Η παραγωγή ως έξοδος ενός επανατοποθετήσημου προγράμματος μηχανής επιτρέπει στα υποπρογράμματα να μεταγλωττιστούν χωριστά. Ένα σύνολο επανατοποθετήσημων αντικειμενικών μονάδων μπορεί να συνδεθεί μαζί και να φορτωθεί για εκτέλεση από ένα φορτώτή σύνδεσης (linking loader). Αν και πρέπει να πληρώνουμε επιπλέον κόστος για την σύνδεση και την φόρτωση, εαν παράγουμε επανατοποθετήσιμες μονάδες αντικειμένου, κερδίζουμε σε ευελιξία καθώς μπορούμε να μεταγλωττίσουμε χωριστά υπορουτίνες και να καλέσουμε από μια αντικειμενική μονάδα τμήματα προγράμματος που έχουν μεταγλωττιστεί νωρίτερα. Εαν η τελική μηχανή δεν χειρίζεται αυτόματα την επανατοποθέτηση, ο μεταγλωττιστής πρέπει να παρέχει στον φορτωτή ρητές πληροφορίες επανατοποθέτησης για να συνδέσει τις μονάδες προγράμματος που έχουν μεταγλωττιστεί χωριστά.

Η παραγωγή ως εξόδου ενός προγράμματος σε συμβολική γλώσσα (assembly language) κάνει την διαδικασία παραγωγής κώδικα κάπως ευκολότερη. Μπορούμε να παράγουμε συμβολικές εντολές και να χρησιμοποιήσουμε τις ευκολίες μακροεντολών του συμβολομεταφραστή για να βοηθήσουμε την παραγωγή κώδικα. Το τίμημα που πληρώνουμε είναι το βήμα της συμβολομετάφρασης μετά την παραγωγή κώδικα.

8.4 Υλοποίηση παραγωγού κώδικα στην Mini Pascal

8.4.1 Η γλώσσα metasim

Στην υλοποίηση μας για τη miniPascal η γλωσσα μηχανής είναι η γλώσσα μηχανής που αναγνωρίζει ο metasim.Παρακάτω εξηγούμε τις βασικές εντολές και λειτουργίες του.Ο metasim εχει ένα σύνολο καταχωρητών R[0],R[1],R[2], ... ,R[255].Ο καταχωρητής R[0] χρησιμοποιείται σαν δείκτης στοίβας και ο program counter συμβολίζεται με ‘$’.Σε αυτούς τους καταχωρητές έχουμε δυνατότητα να αποθηκεύσουμε δεδομένα καθώς και να κάνουμε πράξεις με τα περιεχόμενα ή τις διευθύνσεις τους.

Έχουμε την δυνατότητα να προσπελάσουμε την μνήμη με τους τρεις παρακάτω τρόπους

- M[address]
- M[R[…]]
- M[R[0] + offset]

8.4.2 Βασικές εντολές

Οι βασικές εντολές που υποστηρίζει ο metasim είναι οι παρακάτω:

- ini tr target register
  Χρησιμοποιώντας αυτή την εντολή αποθηκεύεται στον target register η τιμή που έδωσε ο χρήστης από το πληκτρολόγιο.
- outi so1 source register
  Αυτή η εντολή τυπώνει στο τερματικό τα περιεχόμενα του καταχωρητή source.
- addi tr,so1,so2 tr = so1 + so2
- sub1 tr,so1,so2 tr = so1 – so2
- muli tr,so1,so2 tr = so1 * so2
- divi tr,so1,so2 tr = so1/so2

Αυτές οι εντολές υλοποιούν τις πράξεις της πρόσθεσης , αφαίρεσης ,πολλαπλασιασμού και διαίρεσης αντίστοιχα.

- movi tr,so1 tr := so1

Η εντολή αυτή μεταφέρει στον tr register το so1.Το so1 μπορεί να είναι ένας register,μνήμη,αριθμος,ο stack pointer, ο program counter.

- jmp addr addr label, M[..],R[..],δ/ση(αριθμός)
  η εντολή αυτή υλόποιει ένα άλμα χωρίς συνθήκη σε μία διευθυνση μνήμης.
- cmpi so1,so2 so1,so2 registers
  - jb addr
  - jbe addr
  - ja addr
  - jae addr
  - je addr
  - jne addr

Ο καταχωρητής SR είναι ένας πίνακας 8 θέσεων και τον επηρεάζουν η cmpi και ο έλεγχος για overflow.Η cmpi επηρεάζει το SR[0] και το SR[1].Αν οι δύο αριθμοί είναι ίσοι το SR[0] γίνεται 1 και αν ο πρώτος αριθμός είναι έλεγχος υπερχείλισης από

τον δεύτερο τότε το SR[1] γίνεται 1 και αν ο πρώτος αριθμός είναι μεγαλύτερος από τον δεύτερο τότε το SR[1] γίνεται 1.Ο έλεγχος υπερχείλισης επηρεάζει το SR[7] και αν υπάρχει υπερχείλιση τότε γίνεται 1.Όταν η εντολή ειναι jb,jbe,ja,jae,je,jne,jo ελέγχει τα SR[0] και SR[1] και αν ισχύουν οι κατάλληλες συνθήκες πηγαίνει τον program counter στο κατάλληλο σημείο.

8.4.3 Βασικές συναρτήσεις

Η συνάρτηση **gnlvcode** είναι υπεύθυνη να τοποθετήσει την διεύθυνση μίας μη τοπικής μεταβλητής στον καταχωρητή R[255].Παίρνει σαν ορίσμα το όνομα της μεταβλητής .Η υλοποίηση μας για την gnlvcode φαίνεται παρακάτω:

```
void gnlvCode(char * variableName,struct scope * currentScope){
        struct scope * aScope ;
        int varOffset;
        if(currentScope -> prevScope != NULL)
                aScope = currentScope -> prevScope;
        else{
                printf("ERROR GNLVCODE\n");
                customExit();
        }

        fprintf(finalCode,"\tmovi R[254],4\n");
        fprintf(finalCode,"\taddi R[255],R[0],R[254]\n");
        fprintf(finalCode,"\tmovi R[255],M[R[255]]\n");

        varOffset = findDataInScope(variableName,aScope);

        while(varOffset == 0){
                fprintf(finalCode,"\tmovi R[254],4\n");
                fprintf(finalCode,"\taddi R[255],R[255],R[254]\n");
                fprintf(finalCode,"\tmovi R[255],M[R[255]]\n");

                if(aScope -> prevScope != NULL)
                        aScope = aScope -> prevScope;
                else{
                        printf("%s :Syntax Error:%d: Variable: \"%s\" "
                                        "                       undeclared!\n",
                        file_name,currentLineInFile,variableName);
                        customExit();
                }
                varOffset = findDataInScope(variableName,aScope);
        }
        fprintf(finalCode,"\tmovi R[254],%d\n",varOffset);
        fprintf(finalCode,"\taddi R[255],R[255],R[254]\n");

}
```

Η συνάρτηση **loadvr**(v,r) είναι υπεύθυνη να φορτώσει την τιμή της παραμέτρου ‘v’ στον καταχωρητή R[r].Στην υλοποίηση μας η παράμετρος v μπορεί να είναι:

- Αριθμός
- Σταθερά
- Καθολική μεταβλητή (global)
- Τοπική μεταβλητή
- Προσωρινή μεταβλητή
- Παράμετρος συναρτησης

Ο κώδικας της loadbvr φαίνεται παρακάτω:

```
void loadvr(char * variableName,int regNum,struct scope * currentScope){
        struct scope * aScope;
        struct entity* entity;
        int offset=0;

        if(isNumber(variableName,regNum))
                return;
        if(isConst(variableName,regNum,currentScope))
                return;
        offset = findLocalOrTemporaryOrParameter(variableName,currentScope);
        if(offset != 0){
                fprintf(finalCode,"\tmovi R[255],%d\n",offset);
                fprintf(finalCode,"\taddi R[255],R[255],R[0]\n");
                fprintf(finalCode,"\tmovi R[%d],M[R[255]]\n",regNum);
                return;
        }

        offset = findByReferenceParameter(variableName,currentScope);
        if( offset != 0 ){
                fprintf(finalCode,"\tmovi R[255],%d\n",offset);
                fprintf(finalCode,"\taddi R[255],R[255],R[0]\n");
                fprintf(finalCode,"\tmovi R[255],M[R[255]]\n");
                fprintf(finalCode,"\tmovi R[%d],M[R[255]]\n",regNum);
                return;
                }

        offset = findLocalOrTemporaryOrParameter(variableName,currentScope->prevScope);
        if(offset != 0){
                gnlvCode(variableName,currentScope);
                fprintf(finalCode,"\tmovi R[%d],M[R[255]]\n",regNum);
                return;
        }

        offset = findByReferenceParameter(variableName,currentScope->prevScope);
        if(offset != 0){
                gnlvCode(variableName,currentScope);
                fprintf(finalCode,"\tmovi R[255],M[R[255]]\n");
                fprintf(finalCode,"\tmovi R[%d],M[R[255]]\n",regNum);
                return;
        }
        offset = findGlobalVariable(variableName,currentScope);
        if (offset != 0){
                fprintf(finalCode,"\tmovi R[254],600\n");
                fprintf(finalCode,"\tmovi R[255],%d\n",offset);
                fprintf(finalCode,"\taddi R[255],R[255],R[254]\n");
                fprintf(finalCode,"\tmovi R[%d],M[R[255]]\n",regNum);
                return;
        }
        else{
                printf("%s :Syntax Error:%d: Variable: \"%s\" undeclared!\n",
                file_name,currentLineInFile,variableName);
                customExit();
        }
}
```

Η συνάρτηση **storevr(r,v)** αποθηκεύει την τιμή της παραμέτρου 'v' στον καταχωρητη R[r].Ο κώδικας της συνάρτησης φαίνεται εδώ:

```
void storerv(int regNum,char* variableName,struct scope* currentScope){
        int offset = 0;
        offset = findLocalOrTemporaryOrParameter(variableName,currentScope);
        if(offset != 0){
                fprintf(finalCode,"\tmovi R[254],%d\n",offset);
                fprintf(finalCode,"\taddi R[255],R[254],R[0]\n");
                fprintf(finalCode,"\tmovi M[R[255]],R[%d]\n",regNum);
                return;
        }

        offset = findByReferenceParameter(variableName,currentScope);
        if(offset != 0 ){
                fprintf(finalCode,"\tmovi R[255],%d\n",offset);
                fprintf(finalCode,"\taddi R[255],R[255],R[0]\n");
                fprintf(finalCode,"\tmovi R[255],M[R[255]]\n");
                fprintf(finalCode,"\tmovi M[R[255]],R[%d]\n",regNum);
                return;
        }

        offset = findLocalOrTemporaryOrParameter(variableName,
                                                 currentScope->prevScope);
        if(offset != 0){
                gnlvCode(variableName,currentScope);
                fprintf(finalCode,"\tmovi M[R[255]],R[%d]\n",regNum);
                return;
        }

        offset = findByReferenceParameter(variableName,currentScope->prevScope);
        if (offset !=0 ){
                gnlvCode(variableName,currentScope);
                fprintf(finalCode,"\tmovi R[255],M[R[255]]\n");
                fprintf(finalCode,"\tmovi M[R[255]],R[%d]\n",regNum);
                return;
        }

        offset = findGlobalVariable(variableName,currentScope);
        if(offset != 0){
                fprintf(finalCode,"\tmovi R[254],600\n");
                fprintf(finalCode,"\tmovi R[255],%d\n",offset);
                fprintf(finalCode,"\taddi R[255],R[255],R[254]\n");
                fprintf(finalCode,"\tmovi M[R[255]],R[%d]\n",regNum);
                return;
        }
        else{
                printf("%s :Syntax Error:%d: Variable: \"%s\" undeclared!\n",
                file_name,currentLineInFile,variableName);
                customExit();
        }
}
```

Για την υλοποίηση αυτών των συναρτήσεων είναι απαραίτητες οι πληροφόριες του πίνακα συμβόλων.Για τον λόγω αυτό χρησιμοποιουνται οι συναρτήσεις που παρέχονται απο τον πίνακα συμβόλων για την εύρεση αναγνωριστικών κλπ(ενότητα 7.9)

8.4.4 Αντιστοίχιση Ενδιάμεσης αναπαράστασης με Τελικο Κωδικα

Για την παραγωγή ποιοτικού και αποδοτικού τελικού κώδικα είναι αναγκαία μια σωστή αντιστοίχηση μεταξύ της ενδιάμεσης αναπαράστασης και του τελικού κώδικα που πρέπει να παραχθεί.Έτσι λοιπόν σε αυτή την ενότητα θα παρουσιάσουμε τις αντιστοιχίες μεταξύ των εντολών της ενδιάμεσης αναπαράστασης και των τελικών εντολών.

jump '_','_','w' → jmp LW

relop x,y,w → loadvr(x,1)

loadvr(y,2)

cmpi R[1],R[2]

condjmp Lw condjmp={je,jne,jb,jbe,ja,jae}

:= ,x , _ ,z → loadvr(x,1)

storerv(1,z)

op ,x ,y ,z → loadvr(x,1)

loadvr(y,2)

op R[3],R[1],R[2]

storerv(3,z)

out x,_,_ → loadvr(x,1)

out R[1]

in,x,_,_ → ini R[1]

storerv(1,x)

retv , _ , x → loadvr(x,1)

movi R[255],M[8+R[0]]

movi M[R[255]],R[1]

par,x,CV,_ → loadvr(x,255)

movi M[d + R[0]],R[255]

d = framelength + 12 + 4 *i

i = αύξοντας αριθμός παραμέτρων

par x,REF,_

- Αν η καλούσα και η μεβλητη x έχουν το ίδιο βάθος φωλιάσματος:
  - Αν η παράμετρος χ είνα στην καλόυσα τοπική μεταβλητή ή παραμετρος που έχει περάστει με τιμή:

    movi R[255],R[0]

    movi R[254],offset

    addi R[255],R[254],R[255]

    movi M[d+R[0]],R[255]

  - Αν η παράμετρος χ έχει περαστεί στην καλούσα συνάρτηση με αναφορά:

    movi R[255],R[0]

    movi R[254],offset

    addi R[255],R[254],R[255]

    movi R[1],M[R[255]]

    movi M[d+R[0]],R[1]

- Αν η καλούσα και η μεταλητή x έχουν διαφορετικό βάθος φωλιάσματος

- Αν η παράμετρος x είναι στην καλούσα συναρτηση τοπική μεταβλητή η παράμετρος που έχει περαστεί με τιμή:

  gnlvcode()

  movi M[d + R[0]], R[255]

  - Αν η παράμετρος χ εχει περαστεί στην καλούσα συνάρτηση με αναφορά:

    gnlvcode()

    movi R[1],M[R[255]]

    movi M[d+R[0]],R[1]

par x,RET,_ →

movi R[255],R[0]

movi R[254], offset της προσωρινής μεταβλητής όπου θα επιστραφεί η τιμή

addi R[255], R[254], R[255]

movi M[framelength + 8 + R[0]] , R[255]

call ,_,_z →

Με την κλήση της συνάρτησης z πρέπει να ενημερώσουμε το δεύτερο πεδίο του εγγραφήματος δραστηριοποίησης που είναι ο συνδεσμός προσπέλασης της νέας συνάρτησης.

- Αν η καλούσα και η κληθείσα έχουν το ίδιο βάθος φωλιάσματος

  movi R[255],M[4 + R[0]]

  movi M[framelength + 4 + R[0]] ,R[255]

- Αν καλείται συνάρτηση με μεγαλύτερο βάθος φωλιάσματος από την καλούσα :

  movi M[framelength + 4 + R[0]],R[0]

Στη συνέχεια πρέπει να ενημερώσουμε το πρώτο πεδίο του εγγραφήματος δραστηριοποίησης ώστε να η νέα συνάρτηση να γνωρίζει που πρέπει να επιστρέψει όταν τελείωσει την εκτέλεση της:

movi R[255] , framelength

addi R[0], R[255], R[0]

movi R[255], $

movi R[254], 15

addi R[255], R[255], R[254]

movi M[R[0]], R[25]]

jmp Lstart_quad

Με την ολοκλήρωσει της συνάρτησης που κλήθηκε ο έλεγχος θα επιστρέψει στην αμέσβς επόμενη εντολή όμως ο δείκτης στοίβας συνεχίζει να δείχνει στην συνάρτηση που κλήθηκε.Γιαυτο εκτελούμε τις παρακάτω εντολές

movi R[255],framelength

subi R[0],R[0],R[255]

end_block,x,_,_ → jmp M[R[0]]

halt ,_,_,_ → halt

ΚΕΦΑΛΑΙΟ 9 : Βελτιστοποίηση

Οι δομές γλωσσών προγραμματισμού υψηλού επιπέδου μπορεί να προκαλέσουν επιβάρυνση στον χρόνο εκτέλεσης αν μεταφράζουμε με αφέλη τρόπο κάθε δομή σε γλώσσα μηχανής.Η απολοιφή των μη απαραίτητων εντολών στον αντικειμενικό κώδικα ή η αντικατάσταση μιας αλληλουχίας εντολων από μια πιο γρήγορα εκτέλεσιμη αλληλουχία εντολών που έχει το ίδιο αποτέλεσμα, ονομάζεται συνηθως βελτιστοποίηση κώδικα.

9.1 Οι κύριες πηγές βελτιστοποίησης

Η βελτιστοποίηση ενος μεταγλωττιστή πρέπει να διατηρεί τη σημασιολογία του αρχικού προγράμματος.Εκτός απο πολύ ειδικές περιστάσεις όταν δηλαδή ο προγραμματιστής επιλέγει και εφαρμόζει εναν συγκεκριμένο αλγόριθμο, ο μεταγλωττιστής δεν μπορεί να αντιληφθέι αρκετά στοιχεία για αυτό το πρόγραμμα ώστε να το αντικαθαστήσει με ένα ουσιαστικά διαφορετικό και περισσοτερο αποδοτικό αλγόριθμο.Ο μεταγλωττιστής γνωρίζει μόνο πως να εφαρμόζει χαμηλού επιπέδου σημασιολογικούς μετασχηματισμούς, χρησιμοποιώντας γενικά απδεκτές προτάσεις όπως αλγεβρικές ταυτότητες πχ. i+0 = I ή σημασιολογίες προγράμματος όπως το γεγονός πως η εκτέλεση της ίδιας λειτουργίας στις ίδιες τιμές αποδίδει τα ίδια αποτελέσματα.

9.2 Αιτίες πλεονασμού

Υπάρχουν πολλές πλεονάζουσες λειτουργίες σε ένα τυπικό πρόγραμμα.Πολλές φορές ο πλεονασμός ειναι διαθέσιμος σε επίπεδο πηγής.Για παράδειγμα, ένας προγραμματιστής μπορεί να βρεί περισσοότερο άμμεσο και βολικό το να ξαναυπολογίσει καποιο αποτέλεσμα, αφήνοτας το μεταγλωττιστή να αναγνωρίσει πως μόνο ένας τέτοιος υπολογισμός είναι απαραίτητος.Αλλά συχνα, ο πλεονασμός είναι μια παρενέργεια του γεγονότος ότι το πρόγραμμα έχει γραφτεί σε μια γλώσσα υψηλού επιπέδου.Στις περισσότερες γλώσσες (εκτός των C και C++, όπου επιτρέπονται οι αριθμιτικοί δείκτες) οι προγραμματιστές δεν έχουν άλλη επιλογή από το να αναφαίρονται στα στοιχεία ενός πίνακα ή σε περιοχές σε μία δομή μέσω των προσπελάσεων όπως a[i][j].

Καθώς το πρόγραμμα μεταγλωττίζεται κάθε μία από αυτές τις υψηλού επιπέδου προσπελάσεις δομών δεδομένων επεκτείνεται σε έναν αριθμό χαμηλού επιπέδου αριθμιτικών λειτουργιών, όπως ο υπολογισμός της θέσης του (i ,j)στου ενός πίνακα a. Οι προσπελάσεις της ίδιας δομής δεδομένων συχνά μοιράζονται πολλές κοινές χαμηλού επιπέδου λειτουργιές.Οι προγραμματιστές δεν έχουν υπόψη τους αυτές τις χαμηλού επιπέδου λειτουργίες και δεν μπορούν να απαλείψουν τους πλεονασμούς αυτούς καθ'αυτούς.Προτιμάται λοιπόν στην πράξη, από άποψη μηχανικής λογισμικού , οι προγραμματιστές να προσπελάζουν στοιχεία δεδομένων μόνο χρησιμοποιώντας τα υψηλού επιπέδου ονοματά τους.Τότε, τα προγράμματα είναι πιο εύκολο να κατασκευαστούν και κυρίως πιο εύκολο να γίνουν κατανοητά αλλά και να εξελιχθούν.Αναθέτοντας σε ένα μεταγλωττιστή να απαλείφει τους πλεονασμούς επιτυγχάνουμε το καλύτερο δυνατό αποτέλεσμα : τα προγράμματα είναι αποτελεσματικά καθώς και εύκολο να διατηρηθούν.

9.3 Κίνηση κώδικα

Οι βρόχοι αποτελούν πολύ σηματικές περιοχές για βελτιστοποίησεις, ειδικότερα εσωτερικοί βρόχοι όπου τα προγράμματα τείνουν να ξοδεύουν τον περισσότερο από τον χρόνο τους.Ο χρόνος εκτέλεσης ενός προγράμματος μπορεί να βελτιωθεί εάν μειώσουμε τον αριθμό των εντολών εντός ενός εσωτερικού βρόχου,έστω και εάν για το λόγο αυτό αυξήσουμε τις εντολές εκτός του βρόχου αυτού.

Μια σημαντική τροποποίηση η οποία μειώνει την ποσότητα του κώδικα σε έναν βρόχο είναι η κίνιση κώδικα.Αυτός ο μετασχηματισμός παίρνει μια παράσταση η οποία παράγει το ίδιο αποτέλεσμα ανεξάρτητα από τον αριθμό των εκτελέσεων ενός βρόχου και εκτιμά την παράσταση αυτή πριν από το βρόχο.

ΚΕΦΑΛΑΙΟ 10 : Εξοικείωση με την Mini Pascal

Σε αυτό το κεφάλαιο θα παρουσιάσουμε κάποια παραδείγματα ολοκληρωμένα σε Mini Pascal ώστε να δούμε συνολικά πια όλα αυτά που εξηγήσαμε στα προηγούμενα κεφάλαια.

10.1 Ο αλγόριθμος του ευκλείδη

Ένα παράδειγμα miniPascal είναι το παρακάτω που υλοποποιείται ο αλγόριθμος του ευκλείδη , που είναι μια αποτελεσματική μέθοδος για τον υπολογισμό του μέγιστου κοινού διαιρέτη (ΜΚΔ) δύο ακέραιων αριθμών, είναι επίσης γνωστός ως ο μεγαλύτερος κοινός παράγοντας ή υψηλότερος κοινός παρονομαστής. Το όνομά του προέρχεται από τον Έλληνα μαθηματικό Ευκλείδη, ο οποίος τον περιγράφει στα βιβλία VII και X του βιβλίου του *Στοιχεία*.[1]

Στην απλούστερη μορφή του, ο αλγόριθμος του Ευκλείδη ξεκινά με ένα ζεύγος θετικών ακεραίων και σχηματίζει ένα νέο ζευγάρι που αποτελείται από το μικρότερο αριθμό και τη διαφορά μεταξύ των μεγαλύτερων και των μικρότερων αριθμών. Η διαδικασία επαναλαμβάνεται έως ότου οι αριθμοί είναι ίσοι. Αυτός ο αριθμός είναι τότε ο μέγιστος κοινός διαιρέτης του αρχικού ζεύγους.

Η παλαιότερη περιγραφή που υπάρχει για τον Ευκλείδειο αλγόριθμο είναι στα *Στοιχεία* του Ευκλείδη (περ. 300 π.Χ.), καθιστώντας τον ένα από τους παλαιότερους αριθμητικούς αλγορίθμους ακόμα σε κοινή χρήση. Ο αρχικός αλγόριθμος αφορούσε μόνο φυσικούς αριθμούς και γεωμετρικά μήκη (πραγματικοί αριθμοί), αλλά ο αλγόριθμος γενικεύτηκε τον 19ο αιώνα και σε άλλους τύπους αριθμών, όπως Gaussian ακέραιοι και πολυώνυμα μιας μεταβλητής. Αυτό οδήγησε στις σύγχρονες αφηρημένες αλγεβρικές έννοιες, όπως οι Ευκλείδειες δομές. Ο Ευκλείδειος αλγόριθμος έχει γενικευτεί περαιτέρω και σε άλλες μαθηματικές δομές, όπως κόμβους κύκλων nots και ολυώνυμα πολλών μεταβλητών .

Ο αλγόριθμος έχει πολλές θεωρητικές και πρακτικές εφαρμογές. Μπορεί να χρησιμοποιηθεί για να δημιουργήσει σχεδόν όλους τους πιο σημαντικούς παραδοσιακούς μουσικούς ρυθμούς που χρησιμοποιούνται σε διαφορετικούς πολιτισμούς σε όλο τον κόσμο.[2].Πρόκειται για ένα βασικό στοιχείο του RSA αλγορίθμου , μια μέθοδο κρυπτογράφησης δημόσιου κλειδιού που χρησιμοποιείται ευρέως στο ηλεκτρονικό εμπόριο. Χρησιμοποιείται για την επίλυση Διοφαντικών εξισώσεων, όπως η εξεύρεση αριθμών που ικανοποιούν πολλαπλά σχήματα με ίδιο ύψος και μέγεθος (Κινέζικο θεώρημα υπολοίπων) ή πολλαπλασιαστικά αντίστροφα ενός πεπερασμένου πεδίου. Μπορεί επίσης να χρησιμοποιηθεί για την κατασκευή συνεχών κλασμάτων , στην αλυσίδα Sturm,μέθοδο για την εύρεση πραγματικών ριζών ενός πολυωνύμου, και σε πολλές σύγχρονες ακέραιες παραγοντοποιήσεις αλγορίθμων. Τέλος, είναι ένα βασικό εργαλείο για την απόδειξη θεωρημάτων στη σύγχρονη θεωρία αριθμών, όπως το θεώρημα των τεσσάρων τετραγώνων του Lagrange και το θεμελιώδες θεώρημα της αριθμητικής (μοναδική παραγοντοποίηση).

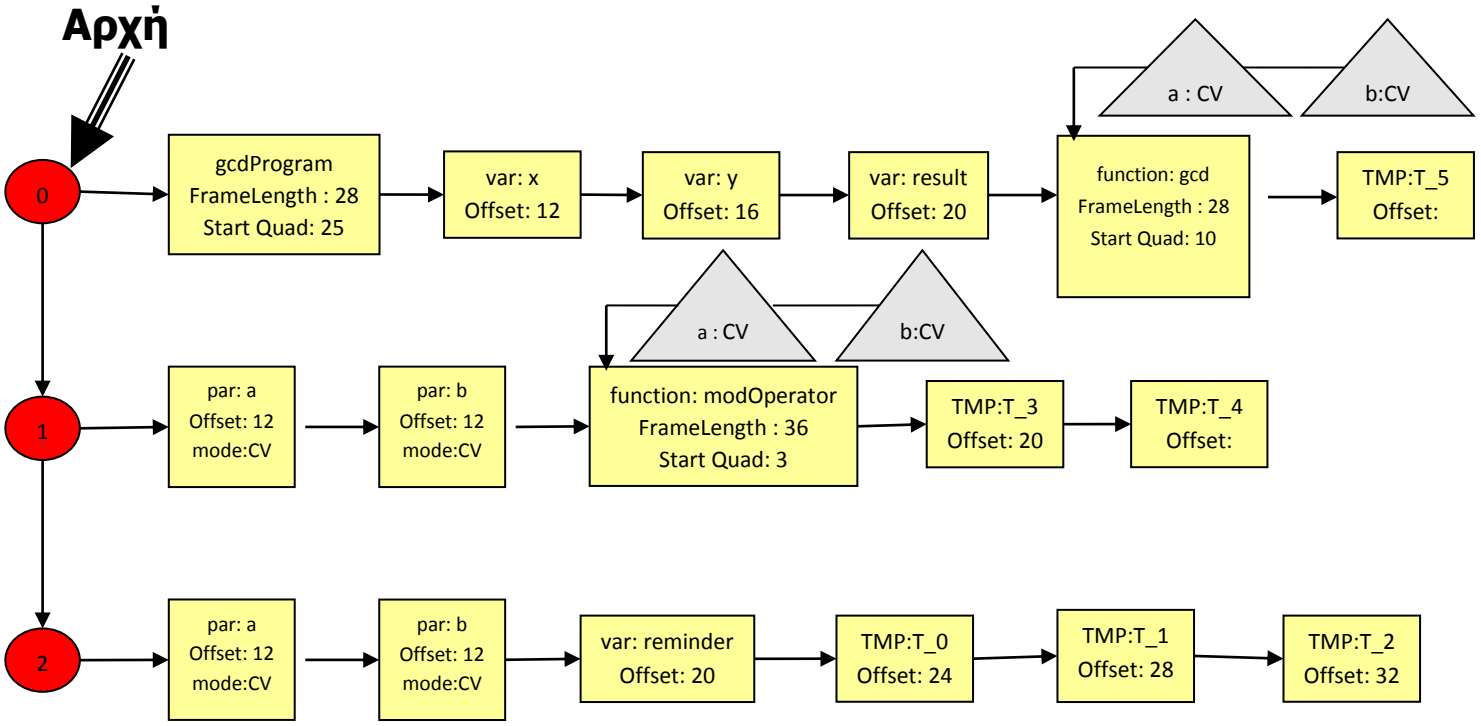
Η υλοποίηση σε Mini Pascal είναι η παρακάτω:

```
program gcdProgram
var x,y,result ;

function gcd(a,b)
        function modOperator(a,b)
                var reminder;
                begin
                        reminder := a - (a/b)*b;
                        return (reminder)

                end
        begin
                if b = 0  then
                        return (a)
                else
                        return (gcd(b,modOperator(a,b)))
        end

begin
        input(x);
        input(y);
        result := gcd(x,y);
        print(result)
end
```

Ο ενδιάμεσος Κώδικας που παράγεται φαίνεται παρακάτω:

```
Quad: 1 | jump , _ , _ , 25
Quad: 2 | begin_block , modOperator , _ , _
Quad: 3 | / , a , b , T_0
Quad: 4 | * , T_0 , b , T_1
Quad: 5 | - , a , T_1 , T_2
Quad: 6 | := , T_2 , _ , reminder
Quad: 7 | retv , reminder , _ , _
Quad: 8 | end_block , modOperator , _ , _
Quad: 9 | begin_block , gcd , _ , _
Quad: 10 | = , b , 0 , 12
Quad: 11 | jump , _ , _ , 14
Quad: 12 | retv , a , _ , _
Quad: 13 | jump , _ , _ , 23
Quad: 14 | par , a , CV , _
Quad: 15 | par , b , CV , _
Quad: 16 | par , T_3 , RET , _
Quad: 17 | call , modOperator , _ , _
Quad: 18 | par , b , CV , _
Quad: 19 | par , T_3 , CV , _
Quad: 20 | par , T_4 , RET , _
Quad: 21 | call , gcd , _ , _
Quad: 22 | retv , T_4 , _ , _
Quad: 23 | end_block , gcd , _ , _
Quad: 24 | begin_block , gcdProgram , _ , _
Quad: 25 | inp , x , _ , _
Quad: 26 | inp , y , _ , _
Quad: 27 | par , x , CV , _
Quad: 28 | par , y , CV , _
Quad: 29 | par , T_5 , RET , _
Quad: 30 | call , gcd , _ , _
Quad: 31 | := , T_5 , _ , result
Quad: 32 | out , result , _ , _
Quad: 33 | halt , _ , _ , _
Quad: 34 | end_block , gcdProgram , _ , _
```

Ο πίνακας συμβόλων στην πλήρη του μορφή φαίνεται παρακάτω :



Ο τελικός κώδικας που παράγεται από τον μεταφραστή μας είναι :

```
L1:     jmp L0

L3:     movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[255],16
        addi R[255],R[255],R[0]
        movi R[2],M[R[255]]
        divi R[3],R[1],R[2]
        movi R[254],24
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L4:     movi R[255],24
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[255],16
        addi R[255],R[255],R[0]
        movi R[2],M[R[255]]
        muli R[3],R[1],R[2]
        movi R[254],28
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L5:     movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[255],28
        addi R[255],R[255],R[0]
        movi R[2],M[R[255]]
        subi R[3],R[1],R[2]
        movi R[254],32
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L6:     movi R[255],32
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[254],20
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L7:     movi R[255],20
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[254],8
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi M[R[255]],R[1]

L8:     jmp M[R[0]]

L10:    movi R[255],16
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[2],0
        cmpi R[1],R[2]
        je L12

L11:    jmp L14

L12:    movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[254],8
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi M[R[255]],R[1]

L13:    jmp L23

L14:    movi R[255],12
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],40
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L15:    movi R[255],16
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],44
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L16:    movi R[255],R[0]
        movi R[254],20
        addi R[255],R[254],R[255]
        movi R[254],36
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L17:    movi M[32 + R[0]],R[0]
        movi R[255],28
        addi R[0],R[255],R[0]
        movi R[255],$
        movi R[254],15
        addi R[255],R[255],R[254]
        movi M[R[0]],R[255]
        jmp L3
        movi R[255],28
        subi R[0],R[0],R[255]

L18:    movi R[255],16
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],40
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]
```

```
L19:    movi R[255],20
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],44
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L20:    movi R[255],R[0]
        movi R[254],24
        addi R[255],R[254],R[255]
        movi R[254],36
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L21:    movi R[254],4
        addi R[254],R[254],R[0]
        movi R[255],M[R[254]]
        movi R[254],32
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]
        movi R[255],28
        addi R[0],R[255],R[0]
        movi R[255],$
        movi R[254],15
        addi R[255],R[255],R[254]
        movi M[R[0]],R[255]
        jmp L10
        movi R[255],28
        subi R[0],R[0],R[255]

L22:    movi R[255],24
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[254],8
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi M[R[255]],R[1]

L23:    jmp M[R[0]]

L0:     movi R[0],600
        jmp L25

L25:    ini R[1]
        movi R[254],12
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L26:    ini R[1]
        movi R[254],16
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L27:    movi R[255],12
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],40
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L28:    movi R[255],16
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],44
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L29:    movi R[255],R[0]
        movi R[254],24
        addi R[255],R[254],R[255]
        movi R[254],36
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L30:    movi M[32 + R[0]],R[0]
        movi R[255],28
        addi R[0],R[255],R[0]
        movi R[255],$
        movi R[254],15
        addi R[255],R[255],R[254]
        movi M[R[0]],R[255]
        jmp L10
        movi R[255],28
        subi R[0],R[0],R[255]

L31:    movi R[255],24
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[254],20
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L32:    movi R[255],20
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        outi R[1]

L33:    halt
```

10.2 Ο αλγόριθμος αντιμετάθεσης

Στους αλγορίθμους ταξινόμησης συχνά επεισέρχεται η ανάγκη να αντιμεταθέσουμε δυο στοιχεία, έτσι εδώ παρουσιάζουμε έναν απλοικό αλγόριθμο αντιμετάθεσης που μας βοηθά να γλυτώνουμε διπλότυπα κώδικα.

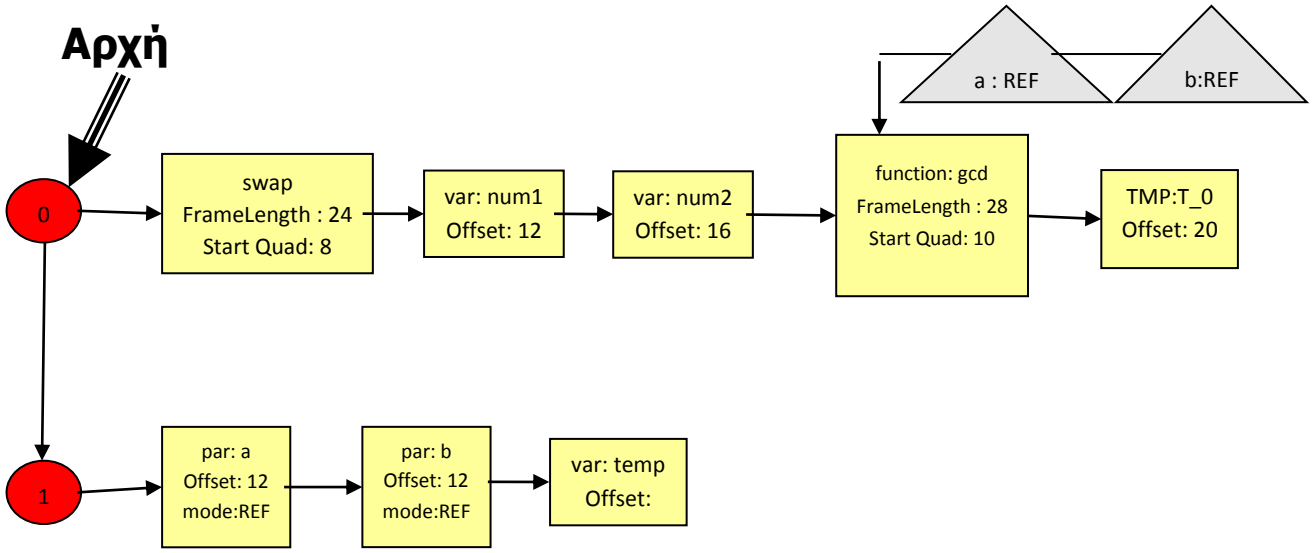
Ο κώδικας σε Mini Pascal :

```
program swap
var num1,num2;

procedure switch(var a,var b)
        var temp ;
        begin
                temp := a;
                a := b;
                b := temp
        end
begin
        input(num1);
        input(num2);
        call switch(var num1,var num2);
        print(num1);
        print(num2)
end
```

Ο ενδιάμεσος κώδικας :

```
Quad: 1 | jump , _ , _ , 8
Quad: 2 | begin_block , switch , _ , _
Quad: 3 | := , a , _ , temp
Quad: 4 | := , b , _ , a
Quad: 5 | := , temp , _ , b
Quad: 6 | end_block , switch , _ , _
Quad: 7 | begin_block , swap , _ , _
Quad: 8 | inp , num1 , _ , _
Quad: 9 | inp , num2 , _ , _
Quad: 10 | par , num1 , REF , _
Quad: 11 | par , num2 , REF , _
Quad: 12 | par , T_0 , RET , _
Quad: 13 | call , switch , _ , _
Quad: 14 | out , num1 , _ , _
Quad: 15 | out , num2 , _ , _
Quad: 16 | halt , _ , _ , _
Quad: 17 | end_block , swap , _ , _
```

Ο πίνακας συμβόλων στην πλήρη του μορφή φαίνεται παρακάτω:



Και ο τελικός κώδικας :

```
L1:     jmp L0

L3:     movi R[255],12
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[1],M[R[255]]
        movi R[254],20
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L4:     movi R[255],16
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[1],M[R[255]]
        movi R[255],12
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi M[R[255]],R[1]

L5:     movi R[255],20
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[255],16
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi M[R[255]],R[1]

L6:     jmp M[R[0]]

L0:     movi R[0],600
        jmp L8

L8:     ini R[1]
        movi R[254],12
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L9:     ini R[1]
        movi R[254],16
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L10:    movi R[255],R[0]
        movi R[254],12
        addi R[255],R[254],R[255]
        movi R[254],36
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L11:    movi R[255],R[0]
        movi R[254],16
        addi R[255],R[254],R[255]
        movi R[254],40
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L12:    movi R[255],R[0]
        movi R[254],20
        addi R[255],R[254],R[255]
        movi R[254],32
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L13:    movi M[28 + R[0]],R[0]
        movi R[255],24
        addi R[0],R[255],R[0]
        movi R[255],$
        movi R[254],15
        addi R[255],R[255],R[254]
        movi M[R[0]],R[255]
        jmp L3
        movi R[255],24
        subi R[0],R[0],R[255]

L14:    movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        outi R[1]

L15:    movi R[255],16
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        outi R[1]

L16:    halt
```

10.3 Παρουσίαση της Δομής Select If

Στο παρακάτω πρόγραμμα εκτελείται μια δομή select if η την οπία υποστηρίζει η Mini Pascal.

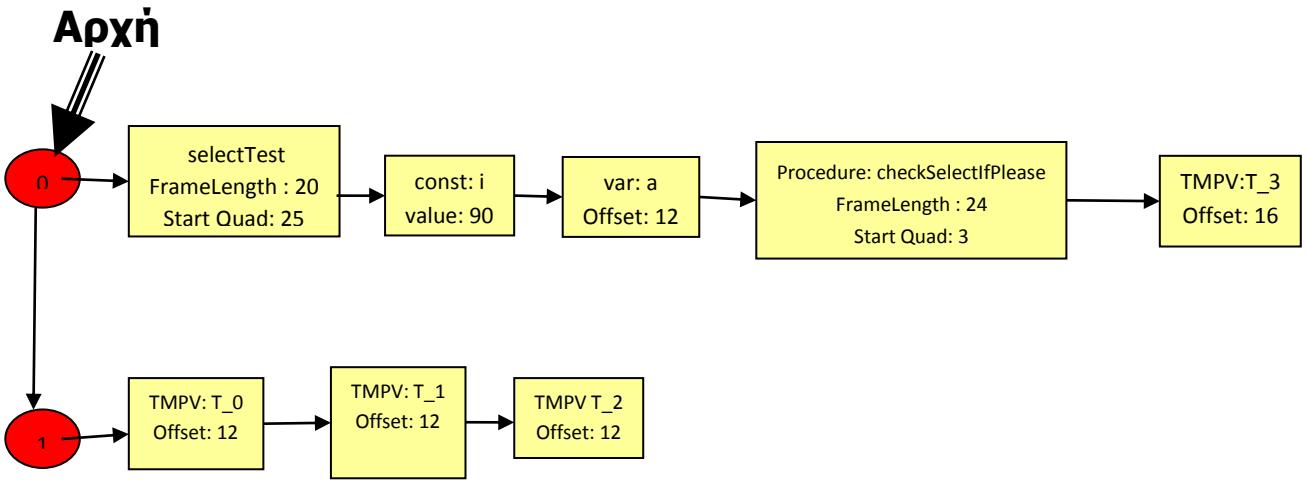
Ο κώδικας:

```
program selectTest
const i := 90;
var a;

procedure checkSelectIfPlease()
begin
	input(a);
	select if (a)
		is equal to: 1
		begin
			print(i-1)
		end
		is equal to : 2
		begin
			print(i*i)
		end
		is equal to: 3
		begin
			print(i+2)
		end
		is equal to: 4
		begin
			print(0)
		end
	endselect
end

begin

	call checkSelectIfPlease()
end
```

Ο ενδιάμεσος κώδικας :

```
Quad: 1 | jump , _ , _ , 25
Quad: 2 | begin_block , checkSelectIfPlease , _ , _
Quad: 3 | inp , a , _ , _
Quad: 4 | = , a , 1 , 6
Quad: 5 | jump , _ , _ , 9
Quad: 6 | - , i , 1 , T_0
Quad: 7 | out , T_0 , _ , _
Quad: 8 | jump , _ , _ , 23
Quad: 9 | = , a , 2 , 11
Quad: 10 | jump , _ , _ , 14
Quad: 11 | * , i , i , T_1
Quad: 12 | out , T_1 , _ , _
Quad: 13 | jump , _ , _ , 23
Quad: 14 | = , a , 3 , 16
Quad: 15 | jump , _ , _ , 19
Quad: 16 | + , i , 2 , T_2
Quad: 17 | out , T_2 , _ , _
Quad: 18 | jump , _ , _ , 23
Quad: 19 | = , a , 4 , 21
Quad: 20 | jump , _ , _ , 23
Quad: 21 | out , 0 , _ , _
Quad: 22 | jump , _ , _ , 23
Quad: 23 | end_block , checkSelectIfPlease , _ , _
Quad: 24 | begin_block , selectTest , _ , _
Quad: 25 | par , T_3 , RET , _
Quad: 26 | call , checkSelectIfPlease , _ , _
Quad: 27 | halt , _ , _ , _
Quad: 28 | end_block , selectTest , _ , _
```

Ο πίνακας συμβόλων :



Και ο τελικός κώδικας

```
L1:     jmp L0

L3:     ini R[1]
        movi R[254],4
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi R[254],12
        addi R[255],R[255],R[254]
        movi M[R[255]],R[1]

L4:     movi R[254],4
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi R[254],12
        addi R[255],R[255],R[254]
        movi R[1],M[R[255]]
        movi R[2],1
        cmpi R[1],R[2]
        je L6

L5:     jmp L9

L6:     movi R[1],90
        movi R[2],1
        subi R[3],R[1],R[2]
        movi R[254],12
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L7:     movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        outi R[1]

L8:     jmp L23

L9:     movi R[254],4
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi R[254],12
        addi R[255],R[255],R[254]
        movi R[1],M[R[255]]
        movi R[2],2
        cmpi R[1],R[2]
        je L11

L10:    jmp L14

L11:    movi R[1],90
        movi R[2],90
        muli R[3],R[1],R[2]
        movi R[254],16
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L12:    movi R[255],16
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        outi R[1]

L13:    jmp L23

L14:    movi R[254],4
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi R[254],12
        addi R[255],R[255],R[254]
        movi R[1],M[R[255]]
        movi R[2],3
        cmpi R[1],R[2]
        je L16

L15:    jmp L19

L16:    movi R[1],90
        movi R[2],2
        addi R[3],R[1],R[2]
        movi R[254],20
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L17:    movi R[255],20
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        outi R[1]

L18:    jmp L23

L19:    movi R[254],4
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi R[254],12
        addi R[255],R[255],R[254]
        movi R[1],M[R[255]]
        movi R[2],4
        cmpi R[1],R[2]
        je L21

L20:    jmp L23
```

```
L25:    movi R[255],R[0]
        movi R[254],16
        addi R[255],R[254],R[255]
        movi R[254],28
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L26:    movi M[24 + R[0]],R[0]
        movi R[255],20
        addi R[0],R[255],R[0]
        movi R[255],$
        movi R[254],15
        addi R[255],R[255],R[254]
        movi M[R[0]],R[255]
        jmp L3
        movi R[255],20
        subi R[0],R[0],R[255]

L27:    halt
```

10.4 Ο αλγόριθμος Fibonacci

Η **Ακολουθία Φιμπονάτσι** ονομάστηκε έτσι από τον Λεονάρντο της Πίζας, γνωστό και ως Φιμπονάτσι. Το βιβλίο του Φιμπονάτσι, το 1202, με τίτλο Liber Abaci, εισήγαγε την ακολουθία στα Μαθηματικά της Δυτικής Ευρώπης αν και η ακολουθία είχε περιγραφεί πιο πριν από τους Ινδούς. (Κατά μία πιο σύγχρονη σύμβαση, η ακολουθία ξεκινάει με F_0=0. Στο Liber Abaci, όμως, η ακολουθία ξεκινάει με F_1=1, παραλείποντας το αρχικό 0, κάτι που ακολουθείται από κάποιους ακόμη και σήμερα).

Οι Αριθμοί Φιμπονάτσι σχετίζονται με τους Αριθμούς Λούκας δεδομένου ότι είναι συμπληρωματικό ζεύγος της Ακολουθίας Λούκας, ενώ είναι άρρηκτα συνδεδεμένοι και με τη χρυσή αναλογία. Έχει αρκετές εφαρμογές σε υπολογιστικούς αλγόριθμους, όπως για παράδειγμα η τεχνική αναζήτησης Φιμπονάτσι και η δομή δεδομένων σωρός Φιμπονάτσι. Επιπλέον υπάρχουν γραφικές παραστάσεις οι οποίες ονομάζονται κύβοι Φιμπονάτσι και χρησιμοποιούνται στις παράλληλες διασυνδέσεις και στα κατανεμημένα συστήματα. Τέλος, οι Αριθμοί Φιμπονάτσι, εμφανίζονται και στη Βιολογία, όπως για παράδειγμα η διακλάδωση στα δέντρα, η διάταξη των φύλλων σε ένα στέλεχος, τα στόμια του καρπού ενός ανανά, η ανάπτυξη της αγκινάρας και πολλά άλλα.

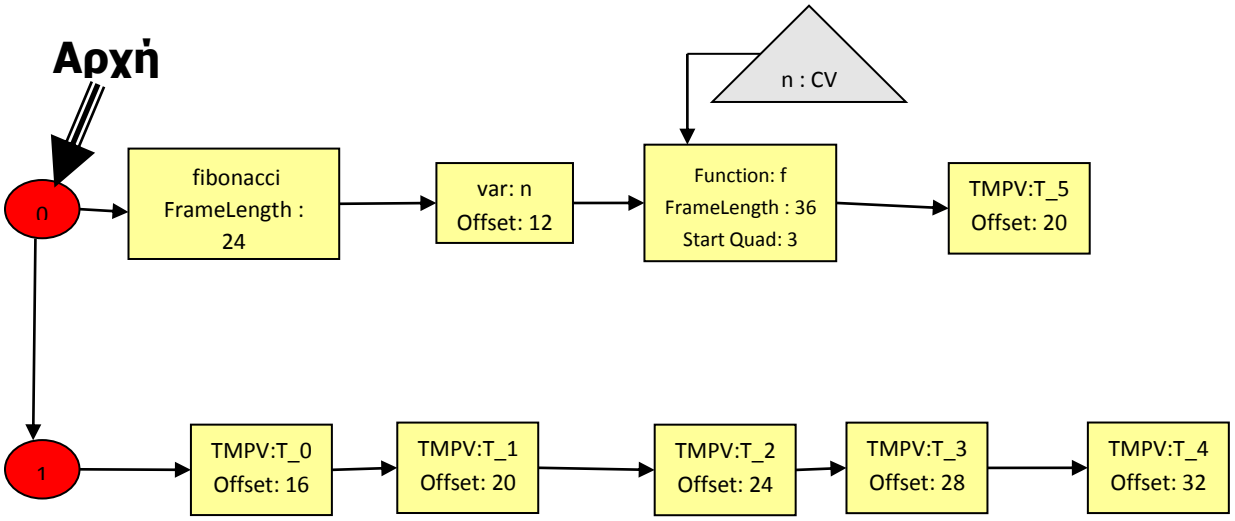
Η υλοποίηση μας σε miniPascal :

```
program fibonacci
var n;
var apotelesma;


function f(n)
begin
        if n = 0 or n = 1 then
                return(n)
        else
                return(f(n-1)+f(n-2))
end

begin
n := 0;
while  n >= 0 do
        begin

                input(n);

                if n >= 0 then
                begin
                        apotelesma := f(n);
                        print(apotelesma)
                end

        end
end
```

Ο ενδιάμεσος κώδικας :

```
Quad: 1 | jump , _ , _ , 21
Quad: 2 | begin_block , f , _ , _
Quad: 3 | = , n , 0 , 7
Quad: 4 | jump , _ , _ , 5
Quad: 5 | = , n , 1 , 7
Quad: 6 | jump , _ , _ , 9
Quad: 7 | retv , n , _ , _
Quad: 8 | jump , _ , _ , 19
Quad: 9 | - , n , 1 , T_0
Quad: 10 | par , T_0 , CV , _
Quad: 11 | par , T_1 , RET , _
Quad: 12 | call , f , _ , _
Quad: 13 | - , n , 2 , T_2
Quad: 14 | par , T_2 , CV , _
Quad: 15 | par , T_3 , RET , _
Quad: 16 | call , f , _ , _
Quad: 17 | + , T_1 , T_3 , T_4
Quad: 18 | retv , T_4 , _ , _
Quad: 19 | end_block , f , _ , _
Quad: 20 | begin_block , fibonacci , _ , _
Quad: 21 | := , 0 , _ , n
Quad: 22 | >= , n , 0 , 24
Quad: 23 | jump , _ , _ , 34
Quad: 24 | inp , n , _ , _
Quad: 25 | >= , n , 0 , 27
Quad: 26 | jump , _ , _ , 33
Quad: 27 | par , n , CV , _
Quad: 28 | par , T_5 , RET , _
Quad: 29 | call , f , _ , _
Quad: 30 | := , T_5 , _ , apotelesma
Quad: 31 | out , apotelesma , _ , _
Quad: 32 | jump , _ , _ , 33
Quad: 33 | jump , _ , _ , 22
Quad: 34 | halt , _ , _ , _
Quad: 35 | end_block , fibonacci , _ , _
```

Ο πίνακας συμβόλων :



Και ο τελικός κώδικας :

```
L1:     jmp L0

L3:     movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[2],0
        cmpi R[1],R[2]
        je L7

L4:     jmp L5

L5:     movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[2],1
        cmpi R[1],R[2]
        je L7

L6:     jmp L9

L7:     movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[254],8
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi M[R[255]],R[1]

L8:     jmp L19

L9:     movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[2],1
        subi R[3],R[1],R[2]
        movi R[254],16
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L10:    movi R[255],16
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],48
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L11:    movi R[255],R[0]
        movi R[254],20
        addi R[255],R[254],R[255]
        movi R[254],44
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L12:    movi R[254],4
        addi R[254],R[254],R[0]
        movi R[255],M[R[254]]
        movi R[254],40
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]
        movi R[255],36
        addi R[0],R[255],R[0]
        movi R[255],$
        movi R[254],15
        addi R[255],R[255],R[254]
        movi M[R[0]],R[255]
        jmp L3
        movi R[255],36
        subi R[0],R[0],R[255]

L13:    movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[2],2
        subi R[3],R[1],R[2]
        movi R[254],24
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L14:    movi R[255],24
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],48
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L15:    movi R[255],R[0]
        movi R[254],28
        addi R[255],R[254],R[255]
        movi R[254],44
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]
```

```
L16:    movi R[254],4
        addi R[254],R[254],R[0]
        movi R[255],M[R[254]]
        movi R[254],40
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]
        movi R[255],36
        addi R[0],R[255],R[0]
        movi R[255],$
        movi R[254],15
        addi R[255],R[255],R[254]
        movi M[R[0]],R[255]
        jmp L3
        movi R[255],36
        subi R[0],R[0],R[255]

L17:    movi R[255],20
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[255],28
        addi R[255],R[255],R[0]
        movi R[2],M[R[255]]
        addi R[3],R[1],R[2]
        movi R[254],32
        addi R[255],R[254],R[0]
        movi M[R[255]],R[3]

L18:    movi R[255],32
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[254],8
        addi R[255],R[0],R[254]
        movi R[255],M[R[255]]
        movi M[R[255]],R[1]

L19:    jmp M[R[0]]

L0:     movi R[0],600
        jmp L21

L21:    movi R[1],0
        movi R[254],12
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L22:    movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[2],0
        cmpi R[1],R[2]
        jbe L24

L23:    jmp L34
L24:    ini R[1]
        movi R[254],12
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L25:    movi R[255],12
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[2],0
        cmpi R[1],R[2]
        jbe L27

L26:    jmp L33

L27:    movi R[255],12
        addi R[255],R[255],R[0]
        movi R[255],M[R[255]]
        movi R[254],36
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L28:    movi R[255],R[0]
        movi R[254],20
        addi R[255],R[254],R[255]
        movi R[254],32
        addi R[254],R[254],R[0]
        movi M[R[254]],R[255]

L29:    movi M[28 + R[0]],R[0]
        movi R[255],24
        addi R[0],R[255],R[0]
        movi R[255],$
        movi R[254],15
        addi R[255],R[255],R[254]
        movi M[R[0]],R[255]
        jmp L3
        movi R[255],24
        subi R[0],R[0],R[255]

L30:    movi R[255],20
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        movi R[254],16
        addi R[255],R[254],R[0]
        movi M[R[255]],R[1]

L31:    movi R[255],16
        addi R[255],R[255],R[0]
        movi R[1],M[R[255]]
        outi R[1]
L32:    jmp L33

L33:    jmp L22

L34:    halt
```

ΚΕΦΑΛΑΙΟ 11 : Βοηθητικά εργαλεία για την ανάπτυξη του Μεταγλωττιστή

Για να κατασκευάσουμε τον Compiler μας χρησιμοποιήσαμε κάποια βοηθητικά εργαλεία και θα τα παρουσιάσουμε σε αυτό το κεφάλαιο.

11.1 Doxygen Documentation

Το Doxygen είναι ένα εργαλείο τεκμηρίωσης κώδικα. Η τεκμηρίωση γράφεται στα αρχεία του πηγαίου κώδικα με την μορφή σχολίων και είναι σχετικά εύκολο να ενημερώνεται. Το doxygen υποστηρίζει πολλές γλώσσες προγραμματισμού όπως : C++, C, C#, Java , Python, IDL, VHDL, Fortran, PHP και άλλες. Είναι ένα ελεύθερο λογισμικό το οποίο δημοσιεύθηκε κάτω από το πρότυπο δημοσίευσης GNU General Public Licence. Σαν το javadoc το doxygen παίρνει την τεκμηρίωση από τον κώδικα. Επιπλέον, το doxygen υποστηρίζει παραγωγή τεκμηρίωσης σε html,rtf,pdf,latex,PostScript ή Man Pages.

11.2 Valgrind Memory Debugging

Το valgrind είναι ένα εργαλέιο για διόρθωση σφαλμάτων που αφορούν την μνήμη, για διάγνωση διαρροής μνήμης. Πήρε το όνομα του από την κεντρική είσοδο για την Valhalla στην Σκανδιναβική Μυθολογία.

Το valgrind είχει σχεδιαστεί επίσημα να είναι ένα ελεύθερο εργαλείο αποσφαλμάτωσης μνήμης για το Linux (x86) αλλά πλεον έχει εξελιχθεί σε ένα γενικό εργαλείο για δημιουργία εργαλείων δυναμικής ανάλυσης. Χρησιμοποιείται από πολλά προγράμματα που αναπτύσονται στο περιβάλλον του Linux.

11.3 Αποσφαλματωτής GNU

Ο αποσφαλματωτής GNU (GNU Debugger), που συνήθως αποκαλείται GDB, με το εκτελέσιμό του να έχει το όνομα *gdb*, είναι ο αποσφαλματωτής του συστήματος λογισμικού GNU. Είναι μεταφέρσιμος: εκτελείται σε πολλά συστήματα Unix και λειτουργεί με διάφορες γλώσσες προγραμματισμού, όπως η Ada, η C, η C++, η Objective-C, η FreeBASIC, η Free Pascal και η Fortran.

Ο GDB προσφέρει σημαντικά χαρακτηριστικά για την παρακολούθηση (tracing) και την τροποποίηση της εκτέλεσης των προγραμμάτων. Ο χρήστης μπορεί να εξετάζει και να τροποποιεί τις τιμές των εσωτερικών μεταβλητών των προγραμμάτων, και μπορεί ακόμα και να καλέσει υπορουτίνες που είναι ανεξάρτητες από την κανονική συμπεριφορά του προγράμματος.

Επεξεργαστές στους οποίους εκτελείται ο GDB είναι μεταξύ άλλων: οι Alpha, οι ARM, οι AVR, οι H8/300, τα συστήματα System/370 και System 390, η πλατφόρμα x86 και η επέκτασή της στα 64-bit x86-64, οι IA-64 "Itanium", ο Motorola 68000, οι MIPS, οι PA-RISC, οι PowerPC, οι SuperH, οι SPARC και οι VAX. Λιγότερο γνωστοί επεξεργαστές που υποστηρίζονται είναι ο A29K, ο ARC, ο ETRAX CRIS, ο D10V, ο D30V, ο FR-30, ο FR-V, ο Intel i960, ο M32R, ο 68HC11, ο Motorola 88000, ο MCORE, ο MN10200, ο MN10300, ο NS32K, ο Stormy16, ο V850 και ο Z8000. (Είναι πιθανό κάποιες από τις νεότερες εκδόσεις να μην υποστηρίζουν όλους τους παραπάνω.) Ο GDB περιέχει μεταγλωττισμένους εξομοιωτές συνόλου εντολών (Instruction Set Simulators) για πιο σπάνιους επεξεργαστές όπως ο M32R ή ο V850.

Ο GDB εξακολουθεί να αναπτύσσεται. Η έκδοση 7.0 εισήγαγε υποστήριξη για σενάρια Python και «αντιστρεπτή αποσφαλμάτωση» ("reversible debugging"), τη δυνατότητα μια συνεδρία αποσφαλμάτωσης να πηγαίνει προς τα πίσω, ώστε να φαίνεται τι συνέβη σε ένα πρόγραμμα που οδηγήθηκε σε τερματισμό λόγω σφάλματος.

ΒΙΒΛΙΟΓΡΑΦΕΙΑ

- Wikipedia η ελεύθερη εγκυκλοπαίδεια
- “Compilers: Principles, Techniques, and Tools”, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman 2014
- Διαφάνειες διαλέξεων του Διδάσκοντα Γ. Μανή
- Διαφάνεις διαλέξεων του Λέκτορα Πανεπιστημίου Μακεδονίας Ηλία Σακελλαρίου