Αλεξανδρειο Τ.Ε.Ι. Θεσσαλονικης
Σχολη Τεχνολογικων Εφαρμογων
Τμημα Πληροφορικης

Πτυχιακή εργασία

Κατασκευή Μεταγλωττιστή (compiler) για τη Γλώσσα PCL

Του φοιτητή
Ευάγγελου Ιτσκου
*Αρ.Μητρώου:*04/2602

Επιβλέπων καθηγητής:
Θεοδόσιος Χειμωνιδης

Θεσσαλονίκη 20 Μαρτίου, 2010

Περίληψη

Ο στόχος της παρούσας πτυχιακής εργασίας ήταν η κατασκευή ενός Μεταγλωττιστή για την Γλώσσα PCL, που παράγει κώδικα Assembly x86. Η Γλώσσα PCL είναι βασισμένη σε ένα υποσύνολο της Pascal, που ορίζεται στο βιβλίο των Παπασπύρου / Σκορδαλάκη, "Μεταγλωττιστές' των εκδόσεων Συμμετρία. Σε αυτό το πλαίσιο διερευνήθηκαν η Θεωρία Γλωσσών και η Αρχιτεκτονική Υπολογιστών. Ειδικότερα, αφού μελετήθηκε το θεωρητικό υπόβαθρο της τυπικής περιγραφής μίας Γλώσσας ,δόθηκε βάση στην περιγραφή αυτών μέσω των Κανονικών Εκφράσεων και των Γραμματικών Ανεξάρτητων Συμφραζομένων, στην συνέχεια διερευνήθηκαν πρακτικοί τρόποι Σημασιολογικών ελέγχων, όπως οι έλεγχοι τύπων. Τέλος μελετήθηκε η αρχιτεκτονική της οικογένειας επεξεργαστών, που ακολουθούν το πρότυπο x86 αφού αυτή ήταν η μηχανή στόχος του Μεταγλωττιστή μας. Χρησιμοποιήθηκαν οι γεννήτριες flex και bison για την αυτόματη παραγωγή του κώδικα, που εκτελεί την Λεξική, την Συντακτική και μέρος της Σημασιολογικής Ανάλυσης. Η ολοκληρωμένη Σημασιολογική Ανάλυση καθώς και η Παραγωγή του Ενδιάμεσου και του Τελικού κώδικα υλοποιήθηκαν με χρήση κατάλληλων δομών δεδομένων και τη Γλώσσα Προγραμματισμού ANSI C. Για τη μεταγλώττιση του κώδικα που αποτέλεσε τον Μεταγλωττιστή της PCL χρησιμοποιήθηκε ο gcc από την Συλλογή Μεταγλωττιστών GNU. Για την μεταγλώττιση του παραχθέντος assembly - κώδικα χρησιμοποιήθηκαν η MASM και μία έτοιμη βιβλιοθήκη χρόνου εκτέλεσης (runtime) της Γλώσσας.

Abstract

The objective of the current thesis was the construction of a Compiler for the PCL language that produces code in the form of Assembly x86. PCL is based upon a subset of Pascal, as it is defined in the book "Compilers"[1] by Papaspyrou / Skordalakis of the publisher Symetria. In this context, research was done on Language Theory and Computer Architecture. In particular, after studying the theoretical background of formal language definition, emphasising the use of Regular Expressions and Context-Free Grammars (CFGs), practical ways for conducting Semantic Checks,such as type checking, were further explored. In addition, the architecture of the x86 CPU family was studied, since that was the target machine. The code that implements the Lexical, Syntactic and part of the Semantic Analysis was generated using flex and bison. Complete Semantic Analysis and the Production of Intermediate and Final Code were implemented using appropriate data structures and coded in the ANSI C programming language. The code that implemented the PCL compiler was compiled itself by gcc, which is part of the GNU Compiler Collection. The generated assembly - code was copiled by MASM and a pre-existing runtime library of PCL.

[1] "Μεταγλωττιστές"

ΠΡΟΛΟΓΟΣ

Το παρόν πόνημα αποτελεί την πτυχιακή εργασία του συγγραφέα για την ολοκλήρωση των σπουδών του στο Τμήμα Πληροφορικής[2] της Σχολής Τεχνολογικών Εφαρμογών του Αλεξάνδρειου Τεχνολογικού Εκπαιδευτικού Ιδρύματος Θεσσαλονίκης.[3]

Πραγματεύεται τις αρχές λειτουργίας ενός Μεταγλωττιστή, μίας διάταξης που παραλαμβάνει ακολουθία βασικών προτάσεων (statements), μιας αυστηρά ορισμένης Γλώσσας την οποία μετατρέπει και εξάγει ως μία νέα ακολουθία προτάσεων μίας άλλης Γλώσσας, κατά κανόνα χαμηλότερου επιπέδου, που τις πιο πολλές φορές είναι Γλώσσα μίας φυσικής ή εικονικής μηχανής. Οι διατάξεις αυτές βρίσκουν επιτυχή εφαρμογή και στον τομέα της μετάφρασης φυσικών (ανθρωπίνων) Γλωσσών, οι οποίες όμως απαιτούν άλλα συνθετότερα εργαλεία ανάλυσης διότι είναι Γλώσσες εξαρτώμενες από το περιβάλλον (sensitive) με πολύ μεγαλύτερες πολυπλοκότητες και πολυσημαντότητες. Έτσι οι τεχνολογίες των μεταφραστών Φυσικών Γλωσσών, βρίσκονται σήμερα, από πλευράς αποτελεσμάτων, ποιο πίσω από τις τεχνολογίες που μελετούμε.

Το θεωρητικό υπόβαθρο των Μεταγλωττιστών αντλείται σε ένα μεγάλο βαθμό από την Επιστήμη της Γλωσσολογίας, πράγμα αναμενόμενο αφού, ασχολούμαστε με Γλώσσες. Δεδομένου όμως του ότι, οι Γλώσσες δεδομένων εξαγωγής είναι συνήθως Γλώσσες κάποιας μηχανής, είναι αυτονόητο ότι απαιτείται πολύ καλή γνώση της εκάστοτε μηχανής και εδώ είναι που υπεισέρχεται ο τομέας της Αρχιτεκτονικής των Υπολογιστών. Αν θέλαμε να κάνουμε μία αναλογία θα μπορούσαμε να πούμε ότι, όπως ένας άνθρωπος διερμηνέας οφείλει να γνωρίζει αρκετά καλά τον πολιτισμό μέσα στον οποίο υπάρχει η αρχική και η τελική Γλώσσα, για να μπορεί να μεταφέρει ορθά και με σαφήνεια νοήματα από την μία στην άλλη, έτσι και για την κατασκευή ενός Μεταγλωττιστή, εκτός της πολύ καλής γνώσης της πηγαίας Γλώσσας, πρέπει να γνωρίζουμε την "κουλτούρα" της μηχανής στην οποία θα μεταγλωττίσουμε.

Πέρα, όμως, από το θεωρητικό υπόβαθρο υπάρχει και το ζήτημα της υλοποίησης των αλγορίθμων με τους οποίους καταλήγουμε να περιγράφουμε έναν Μεταγλωττιστή. Σε αυτή μας την προσπάθεια έχουμε δύο δρόμους, ο ένας είναι η χειροποίητη υλοποίηση του συνόλου των αλγορίθμων, ο δεύτερος έχει να κάνει με την χρήση γεννητριών, τις οποίες τροφοδοτούμε με κατάλληλες περιγραφές της πηγαίας Γλώσσας και αυτές παράγουν τον ζητούμενο κώδικα ή μέρη αυτού. Από εκείνο το σημείο και μετά, ουσιαστικά, συνεχίζουμε, όπως θα κάναμε και στον πρώτο τρόπο.

Παράλληλα με την εκπόνηση του παρόντος πονήματος υλοποιήθηκε και ένας πειραματικός Μεταγλωττιστής για την Γλώσσα PCL, η οποία αποτελεί εμπλουτισμένο υποσύνολο της Pascal. Σε κάθε κεφάλαιο αναφέρονται αντί-

[2] `www.it.teithe.gr`

[3] `www.teithe.gr`

στοιχα τμήματα αυτού και στο τέλος γίνεται μία συνολική παρουσίασή του. Είναι αλήθεια πως η κατασκευή του, ήταν ένα αρκετά επίπονο έργο, αφού κατά την υλοποίηση των αλγορίθμων, που παρέδωσε η θεωρητική μελέτη του προβλήματος, έπρεπε να αντιμετωπιστούν τόσο οι αδυναμίες των γεννητριών μεταγλωττιστών, που χρησιμοποιήθηκαν, όσο και τα λογικά σφάλματα που εισήχθησαν κατά την διάρκεια της συγγραφής του κώδικα.

Ειδικότερα, για τη υλοποίηση του πειραματικού Μεταγλωττιστή χρησιμοποιήθηκαν οι γεννήτριες flex (Lex) και bisson (yacc), ενώ ως Γλώσσα υλοποίησης επιλέχθηκε η ANSI C.

Περιεχόμενα

Σχήματα

Πίνακες

Κεφάλαιο 1

ΕΙΣΑΓΩΓΗ

1.1 Προγραμματισμός Η/Υ

Προγραμματισμός Η/Υ ονομάζεται η περιγραφή μίας εργασίας σε βήματα, με τέτοιο τρόπο, ώστε να είναι δυνατή η εκτέλεση της από έναν Η/Υ (μία μηχανή). Ο σκοπός ενός προγραμματιστή είναι να ρυθμίσει (προγραμματίσει) μία μηχανή (Η/Υ), έτσι ώστε αυτή, είτε να εκτελεί μία εργασία χωρίς επίβλεψη, είτε να λειτουργεί ως βοηθός ενός ανθρώπου (του χειριστή) στη εκτέλεση μίας εργασίας.

Ο Προγραμματισμός των μηχανών έχει καταλυτική επίδραση σε ολόκληρο το φάσμα των κοινωνικών μας αλληλεπιδράσεων. Δίνει στης μηχανές την δυνατότητα να εκτελούν απλές ή ακόμη και πολύπλοκες εργασίες, με μαθηματική ακρίβεια και το σημαντικότερο, με μεγάλη ταχύτητα. Έτσι, αρχικά απαλλάσσεται ο άνθρωπος από εργασίες που χρειάζονταν μεγάλα χρονικά διαστήματα για να ολοκληρωθούν, στην συνέχεια ανατίθενται στις μηχανές εργασίες, οι οποίες είτε είναι ασύμφορο, είτε είναι πολλές φορές αδύνατο να εκτελεστούν από ανθρώπους, με αποδεκτό πλήθος πόρων και αρκετά μικρή διάρκεια ώστε να έχει σημασία η ολοκλήρωσή τους.

Τα παραπάνω έγιναν αντιληπτά από την επιστημονική αλλά και την επιχειρηματική κοινότητα. Έτσι σταδιακά φτάσαμε στο σημείο να μην υπάρχουν παρά ελάχιστες δραστηριότητες, που να μην υποβοηθούνται από υπολογιστικές μηχανές, τουλάχιστον στον "Ανεπτυγμένο κόσμο". Τα προϊόντα του προγραμματισμού βρίσκονται παντού, στις επιχειρήσεις, στους οργανισμούς, στα ακαδημαϊκά ιδρύματα και στις δημόσιες υπηρεσίες, παίζοντας ουσιαστικό ρολό στην διεκπεραίωση των καθηκόντων και την επίτευξη των στόχων τους.

Στην συνέχεια γίνεται μία παρουσίαση της πορείας του Προγραμματισμού

Η/Υ από την γέννησή του έως σήμερα.

1.1.1 Εξελικτική πορεία

Αναφορές σε μηχανές που λειτουργούν βάση προκαθορισμένων εντολών υπάρχουν από την αρχαιότητα.[1] Τα πρώτα καταγεγραμμένα προγραμματιζόμενα αυτόματα είναι ο Βηματικός Υπολογιστής (Step Reckoner) του Leibniz (1672 - υλοποιήθηκε το 1694) και η Αναλυτική Μηχανή[2] του Charles Babbage ($19^{ος}$ αιώνας). Με την έναρξη της Βιομηχανικής Επανάστασης η εξέλιξη του προγραμματισμού ήταν ραγδαία.

Τα πράγματα άρχισαν να παίρνουν την σημερινή τους μορφή με την εφεύρεση της αρχιτεκτονικής Von Neumann η οποία επέτρεπε την αποθήκευση των δεδομένων στην μνήμη του Η/Υ. Αρχικά, βέβαια, οι προγραμματιστές έπρεπε να δημιουργούν τα προγράμματα σε γλώσσα μηχανής, η οποία φυσικά ήταν διαφορετική για κάθε μηχανή. Στην συνέχεια, δημιουργήθηκαν οι γλώσσες assembly οι οποίες αποτελούνταν από λέξεις-κλειδιά που απλώς αντιστοιχούσαν μία προς μία σε εντολές μηχανής. Εδώ, εκτός από την χειροποίητη μετατροπή των προγραμμάτων σε γλώσσα μηχανής, δημιουργήθηκαν και οι πρώτοι συμβολομεταφραστές (assemblers) που σκοπό είχαν την μετατροπή των συμβολικών εντολών σε κώδικα μηχανής.

Το 1950-52 έχουμε την εφεύρεση της πρώτης γλώσσας υψηλού επιπέδου Fortran. Η Fortan επέτρεπε στους προγραμματιστές να γράψουν τα προγράμματά τους σε μία μορφή που πλησίαζε αρκετά τον Μαθηματικό Συμβολισμό.

[1] Χαρακτηριστικό παράδειγμα είναι ο μηχανισμός των Αντικυθήρων ο οποίος δοθέντος του χρόνου υπολόγιζε και παρουσίαζε μία πληθώρα αστρονομικών δεδομένων.

[2] Ένας γενικού σκοπού μηχανικός υπολογιστής ο οποίος μπορούσε να δεχτεί δεδομένα και εντολές μέσω διάτρητων καρτών.

Τα προγράμματα αυτά μεταγλωττίζονταν σε γλώσσα μηχανής με τη χρήση ενός άλλου προγράμματος, του μεταγλωττιστή. Για πρώτη φορά η συγγραφή των προγραμμάτων μπορούσε να γίνει γρήγορα, με λιγότερα λάθη και ευκολότερο εντοπισμό και διόρθωση τους. Ενδεικτικά μέχρι τότε για τα προβλήματα που λύνονταν από H/Y τα $^2/_3$ του κόστους και το 90% του χρόνου αφιερωνόντουσαν στον σχεδιασμό, την υλοποίηση και την αποσφαλμάτωση του προγράμματος. Η νέα αυτή τεχνική προγραμματισμού υποσχόταν την μείωση του απαιτούμενου χρόνου στο $^1/_5$. Από τότε μέχρι σήμερα επινοήθηκαν πολλές τεχνικές προγραμματισμού όπως ο Προστακτικός (imperative), ο συναρτησιακός (functional) και ο λογικός (logical). Πάνω σε αυτές δημιουργήθηκαν πολυάριθμες γλώσσες, άλλες με μεγαλύτερη και άλλες με μικρότερη αποδοχή. Κάθε τεχνολογία προγραμματισμού, αλλά και κάθε νέα γλώσσα έχει σαν στόχο να μεταφέρει την διαδικασία δημιουργίας ενός προγράμματος πιο κοντά στην ανθρώπινη σκέψη, έχοντας ως απώτερους στόχους τους ίδιους με την Fortan, ελαχιστοποίηση των χρόνων ανάπτυξης και αποσφαλμάτωσης, με παράλληλη μείωση του αριθμού των σφαλμάτων κατά την υλοποίηση. Οι προσπάθειες για την δημιουργία νέων καλύτερων, στα πλαίσια που ορίστηκαν προηγουμένως, γλωσσών δεν έχουν σταματήσει σε καμία περίπτωση, επιπλέον με το πέρασμα του χρόνου η ευκολία με την οποία μπορεί κάποιος να υλοποιήσει τις ιδέες του για μια νέα γλώσσα, τουλάχιστον σε επίπεδο πρωτοτύπου, αυξάνεται.

1.2 Το σημαντικότερο εργαλείο - Οι Μεταγλωττιστές

Όλα τα παραπάνω έγιναν εφικτά χάρη στην εξέλιξη που επιτεύχθηκε στον τομέα των Μεταγλωττιστών. Ταυτόχρονα με την δημιουργία της Fortran τέθηκαν τα θεμέλια και ξεκίνησε η ανάπτυξη της Θεωρίας αλλά και της τεχνολογίας τους. Η διαδικασία δημιουργίας ενός Μεταγλωττιστή, από την πρώτη προσπάθεια ακόμη, είναι τυποποιημένη και υποστηριζόμενη από το ανάλογο θεωρητικό υπόβαθρο. Οι τεχνικές αυτές από το 1950 και μετά βελτιώνονται συστηματικά και έχει σημειωθεί αξιόλογη πρόοδος. Επιπλέον, πολλά μέρη τους, εξαιτίας της τυποποίησης της διαδικασίας κατασκευής τους, μπορούν να δημιουργηθούν από άλλα προγράμματα, τις **γεννήτριες Μεταγλωττιστών**, στα οποία δίνουμε κατάλληλες περιγραφές μερών μίας Γλώσσας και παίρνουμε το ανάλογο τμήμα του Μεταγλωττιστή.

1.2.1 Τί είναι οι Μεταγλωττιστές

Οι μεταγλωττιστές είναι προγράμματα Η/Υ που παίρνουν μία ακολουθία προτάσεων (statements), οι οποίες περιγράφουν την εκτέλεση μίας εργασίας, αποτυπωμένων με τη χρήση κάποιας Γλώσσας και τις μεταφέρουν σε μία ισοδύναμη άλλη. Συνήθως η τελική Γλώσσα είναι χαμηλότερου επιπέδου από την αρχική, χωρίς αυτό να είναι περιοριστικό, η τελική Γλώσσα θα μπορούσε κάλλιστα να είναι του ίδιου ή και υψηλοτέρου επιπέδου (αν και στις δύο αυτές περιπτώσεις μιλούμε πια για Μεταφραστές και όχι Μεταγλωττιστές). Αξίζει να σημειωθεί πως οι προτάσεις αποτυπωμένες στην χαμηλοτέρου επιπέ-

δου Γλώσσα είναι αναλυτικότερες και αποτελούνται από απλούστερα βήματα, πράγμα το οποίο απορρέει από την φύση της Γλώσσας αυτής.

Το συνηθέστερο είναι η τελική Γλώσσα να είναι η Γλώσσα κάποιας μηχανής (φυσικής ή εικονικής). Οι ενέργειες που περιγράφονται σε αυτή την Γλώσσα μπορούν να εκτελεστούν από την μηχανή, με αποτέλεσμα να εκτελείται η εργασία που περιγράφηκε από τον άνθρωπο, ως ακολουθία προτάσεων, από την ίδια την μηχανή.

Ένα ακόμα χαρακτηριστικό ενός Μεταγλωττιστή είναι η Γλώσσα στην οποία έχει υλοποιηθεί. Συνήθως είναι κάποια Γλώσσα υψηλού επιπέδου. Πολλές φορές χρησιμοποιείται ως Γλώσσα υλοποίησης η Πηγαία Γλώσσα. Αυτός ο τρόπος κατασκευής ονομάζεται μέθοδος εκκίνησης (bootstrapping). Αρχικά δημιουργείται μία πρώτη έκδοση του Μεταγλωττιστή χρησιμοποιώντας κάποια άλλη Γλώσσα υλοποίησης, στην συνέχεια ξαναγράφεται ο κώδικας στην Πηγαία Γλώσσα, και μεταγλωττίζεται από αυτόν. Από δω και πέρα ο μεταγλωττιστής βελτιώνεται στην Πηγαία Γλώσσα, η οποία έχει γίνει πλέον και γλώσσα υλοποίησης, και ο κώδικας μεταγλωττίζεται χρησιμοποιώντας κάθε φορά την τελευταία έκδοση του Μεταγλωττιστή.

1.2.2 Τί μας προσφέρουν

Οι Μεταγλωττιστές επιτρέπουν στον προγραμματιστή να σχεδιάσει και να εκφράσει την λύση ενός προβλήματος ή γενικότερα την εκτέλεση μίας διαδικασίας σε υψηλότερο επίπεδο από αυτό που απαιτεί το σύνολο εντολών της μηχανής και πιο κοντά στον τρόπο με τον οποίο σκέφτεται. Αυτό έχει σαν αποτέλεσμα την ευκολότερη και ταχύτερη δημιουργία του προγράμματος. Ου-

σιαστικά πρόκειται για μία περισσότερο μακροσκοπική θεώρηση της λύσης του προβλήματος. Συνέπεια αυτού είναι το πρόγραμμα να μπορεί να κατανοηθεί καλύτερα είτε από τον αρχικό συγγραφέα, σε αργότερο χρόνο, είτε από άλλους προγραμματιστές οι οποίοι θα αναλάβουν μελλοντικά την συντήρηση και την επέκταση του. Ένα ακόμα σημαντικό πλεονέκτημα της καλύτερης κατανόησης του νοήματος που κρύβεται πίσω από τις εντολές του προγράμματος είναι ότι ο προγραμματιστής είναι σε θέση να εντοπίσει γρήγορα και με ακρίβεια τα σημεία στα οποία έχουν γίνει λογικά λάθη, έχοντας στην διάθεσή του μόνο τα λανθασμένα αποτελέσματα.

Επιπλέον, οι γραμματικοί και συντακτικοί κανόνες των ανωτέρων Γλωσσών Προγραμματισμού, εφαρμοζόμενοι από τους Μεταγλωττιστές, έχουν ως αποτέλεσμα των εντοπισμό ενός μεγάλου αριθμού σφαλμάτων του προγράμματος κατά την μεταγλώττισή του.

Στα σημαντικότερα πλεονεκτήματα που μας δίνουν συγκαταλέγεται ότι, υλοποιώντας έναν μεταγλωττιστή για μια διαφορετική μηχανή, μπορούμε να μεταφέρουμε προγράμματα που είναι γραμμένα σε μία ανώτερη Γλώσσα Προγραμματισμού στη νέα αυτή μηχανή τροποποιώντάς τα ελάχιστα έως καθόλου. Αντίθετα, αν το πρόγραμμα ήταν γραμμένο σε γλώσσα μηχανής, λόγω ασυμβατότητας των γλωσσών των δύο μηχανών, θα έπρεπε να ξαναγραφτεί από το μηδέν.

Είναι αξιοσημείωτο ότι ο κώδικας σε Γλώσσα μηχανής που παράγει ένας Μεταγλωττιστής, ιδίως σήμερα, είναι εξαιρετικά καλής ποιότητας, ενώ πάνω του γίνονται μία σειρά βελτιστοποιήσεις με τις οποίες, ανάλογα με τις επιλογές του προγραμματιστή, κάνουν τον εκτελέσιμο κώδικα είτε ταχύτερο, είτε μικρότερο ή ακόμη και συνδυασμό των δύο σε διαφορετικούς βαθμούς.

Επικρατεί στις μέρες μας ότι ο κώδικας που παράγει μία γεννήτρια προγραμμάτων είναι υποδιέστερος από αυτόν που γράφει ένας προγραμματιστής. Στην πραγματικότητα αυτό ισχύει μόνο όταν ο προγραμματιστής έχει ιδιαίτερες ικανότητες και ταλέντο, πολύ πάνω από τον μέσο όρο. Γενικά, μία γεννήτρια παράγει καλύτερο κώδικα από έναν μέσο προγραμματιστή.

1.2.3 Η δομή τους

Όπως έχει ήδη αναφερθεί (1.2.1) ένας Μεταγλωττιστής παίρνει ένα **αρχικό (πηγαίο) πρόγραμμα** γραμμένο σε μία πηγαία Γλώσσα και εξάγει το ίδιο πρόγραμμα, το οποίο τώρα ονομάζεται **τελικό (στόχος) πρόγραμμα**, σε μία άλλη Γλώσσα - στόχο. Η πλέον συνηθισμένη περίπτωση είναι η είσοδος και η έξοδος του Μεταγλωττιστή να αποτελούν η καθεμία ένα φάκελο (file) Η/Υ.

Για να φτάσει από το αρχικό πρόγραμμα στο τελικό ο Μεταγλωττιστής το περνά από κάποια στάδια. Αρχικά το πρόγραμμα, έχοντας την μορφή μιας ακολουθίας συμβόλων, περνά από την φάση της **Λεξικής Ανάλυσης** (2) από την οποία βγαίνει ως μία σειρά **λεξικών μονάδων** (λεξήματα). Στην συνέχεια η σειρά αυτή τροφοδοτείται στην φάση της **Συντακτικής Ανάλυσης** (3) από την οποία παράγεται το **συντακτικό δέντρο** (Abstract Syntax Tree - AST). Το συντακτικό δέντρο περνάει από την φάση της **Σημασιολογικής Ανάλυσης** (2) όπου προστίθενται σε αυτό σημασιολογικές πληροφορίες, οι οποίες θα βοηθήσουν σε επόμενες φάσεις. Παράλληλα σε αυτή τη φάση γίνεται και ο έλεγχος των τύπων. Η επόμενη φάση είναι η **παραγωγή του ενδιάμεσου κώδικα** (6). Εδώ, δοθέντος του συντακτικού δέντρου παράγεται ο ενδιάμεσος κώδικας αποτυπωμένος σε μία **ενδιάμεση Γλώσσα**. Τρεις είναι οι πιο

συνηθισμένες ενδιάμεσες Γλώσσες:

- Οι **Τετράδες** (quadruples), όπου κάθε τετράδα αποτελείται από τον τελεστή, τα δύο τελούμενα στα οποία εφαρμόζεται ο τελεστής και το τέταρτο στοιχείο στο οποίο αποθηκεύεται το αποτέλεσμα.

- Ο **επιθεματικός κώδικας** (postfix code), ο οποίος είναι η γλώσσα μίας εικονικής μηχανής στοίβας. Σαρώνεται σειριακά και αν το στοιχείο που βρίσκεται στην θέση του δείκτη είναι τελούμενο τότε τοποθετείται στην στοίβα, αν πρόκειται για τελεστή αφαιρούνται από την στοίβα τα τελούμενά του και, αφού ο τελεστής εφαρμοστεί σε αυτά, το αποτέλεσμα επανατοποθετείται στην στοίβα.

- Τα **αφηρημένα συντακτικά δέντρα** (AST), τα οποία προέρχονται από την απλοποίηση των συντακτικών δέντρων της προηγούμενης φάσης και προσθέτοντας στους κόμβους τους επιπλέον πληροφορίες.

Τέλος, περνάμε τον ενδιάμεσο κώδικα από την φάση της **παραγωγής του τελικού κώδικα** ο οποίος αποτελεί το αρχικό μας πρόγραμμα αποτυπωμένο στην τελική Γλώσσα.

Αν αντικαθιστούσαμε την τελευταία φάση με μία φάση **εκτέλεσης του ενδιάμεσου κώδικα** τότε θα είχαμε έναν **Διερμηνευτή**[3]. Είναι προφανές ότι ένας Μεταγλωττιστής διαφέρει ελάχιστα από έναν Διερμηνευτή, χωρίς να λάβουμε φυσικά υπόψιν τις βελτιστοποιήσεις που επιτελούνται από τον Μεταγλωττιστή. Αυτός είναι ένας από τους λόγους που ωθεί αρκετούς δημιουργούς

[3]Οι Διερμηνευτές είναι προγράμματα H/Y τα οποία παίρνουν ένα πρόγραμμα γραμμένο σε κάποια Γλώσσα προγραμματισμού και το εκτελούν, χωρίς να παράγουν εκτελέσιμο κώδικα.

Γλωσσών να δημιουργήσουν αρχικά έναν Διερμηνευτή της Γλώσσας και αργότερα να προχωρήσουν στην δημιουργία του Μεταγλωττιστή.

Είναι σύνηθες για Μεταγλωττιστές παραγωγής, μετά από κάθε μια από τις δύο τελευταίες φάσεις, να προσθέτουμε δύο ακόμα φάσεις βελτιστοποίησης του ενδιάμεσου και του τελικού κώδικα. Ο λόγος για τον οποίο δεν αναφέρθηκαν παραπάνω, είναι πως αρχικά ο στόχος είναι η δημιουργία ενός Μεταγλωττιστή που θα είναι σε θέση να παράγει ορθό κώδικα και στην συνέχεια προσθέτουμε τις διαδικασίες βελτιστοποίησης. Οι βελτιστοποιήσεις που εφαρμόζονται στον κώδικα αφορούν είτε την ταχύτερη εκτέλεσή του, είτε μικρότερες απαιτήσεις μνήμης ή ακόμη και συνδυασμό των δύο. Το προς τα που θα κινηθεί ο Μεταγλωττιστής εξαρτάται από τις οδηγίες που του έχουν δοθεί από τον προγραμματιστή. Η σύγχρονη τάξη πραγμάτων στις τεχνολογίες μεταγλώττισης είναι η παγίωση των πρώτων φάσεων Λεξικής και Συντακτικής Ανάλυσης καθώς και της Σημαντικής (Σημασιολογικής) Ανάλυσης και συνέχιση της έρευνας και ανάπτυξης των διαφόρων ειδών βελτιστοποιήσεων σε χώρο (μέγεθος) τελικού κώδικα και χρόνου εκτέλεσης του παραγομένου εκτελέσιμου.

Ταυτόχρονα από την αρχή ο Μεταγλωττιστής παράγει και χρησιμοποιεί τον **Πίνακα Συμβόλων**, όπου διατηρούνται πληροφορίες σχετικές με τον αρχικό κώδικα, που θα χρειαστούν σε όλες τις μετέπειτα φάσεις, καθώς και για τον καθορισμό της ακριβούς ή περίπου θέσης των πιθανά εντοπισμένων λαθών.

1.2.4 Οι αρχιτεκτονικές των Μεταγλωττιστών

Όταν πρόκειται να υλοποιηθούν οι φάσεις ενός Μεταγλωττιστή κάθε φάση υλοποιείται ως μία **λειτουργική μονάδα** (module). Το ίδιο ισχύει και για

την αναφορά σφαλμάτων.

Η πρώτη χωρίζει τις φάσεις σε δύο ομάδες, το **Εμπρόσθιο** και το **Οπίσθιο τμήμα**. Στο εμπρόσθιο τμήμα βρίσκονται όλες οι λειτουργίες που έχουν άμεση σχέση με την αρχική Γλώσσα, ξεκινώντας από την Λεξική Ανάλυση και καταλήγοντας στην παραγωγή του πίνακα συμβόλων και του ενδιάμεσου κώδικα. Στην συνέχεια ο ενδιάμεσος κώδικας τροφοδοτεί στο Οπίσθιο τμήμα, το οποίο περιλαμβάνει τις λειτουργικές μονάδες που σχετίζονται με την τελική Γλώσσα, και τελειώνει με την παραγωγή του τελικού, συνήθως εκτελέσιμου, κώδικα. Οι λειτουργικές μονάδες που αποτελούν τις φάσεις εκτελούνται σειριακά. Δηλαδή, το πρόγραμμα εισέρχεται στην πρώτη λειτουργική μονάδα και αφού επεξεργασθεί το σύνολό του εξέρχεται από αυτή. Στην συνέχεια το αποτέλεσμα εισάγεται εξ' ολοκλήρου στην επόμενη μονάδα και επεξεργάζεται συνολικά. Η διαδικασία αυτή συνεχίζεται μέχρι το τελευταίο στάδιο.

Η τεχνική αυτή είναι πολύ ελκυστική γιατί είναι εύκολη τόσο στην κατανόηση όσο και στην υλοποίηση, με αποτέλεσμα να προτιμάται σε Ακαδημαϊκές και ερευνητικές υλοποιήσεις. Έχει όμως το μειονέκτημα των υψηλών απαιτήσεων σε μνήμη, αφού μετά από κάθε φάση δημιουργείται συνολικά η νέα αναπαράσταση του προγράμματος. Το γεγονός αυτό, δεδομένων των προδιαγραφών των Η/Υ, που ανήκαν σε προηγούμενες δεκαετίες, καθιστούσε τις υλοποιήσεις αυτού του τύπου απαγορευτικές ακόμη και για εκπαιδευτικούς σκοπούς. Τα τελευταία χρόνια, όμως, τα μεγέθη της διαθέσιμης μνήμης έχουν αλλάξει. Αυτό έχει σαν αποτέλεσμα όλο και περισσότεροι κατασκευαστές Μεταγλωττιστών να προτιμούν την συγκεκριμένη αρχιτεκτονική. Επιπλέον, ένας ακόμα λόγος να επιλέγεται αυτή η οργάνωση, είναι πως για τις περισσότερες νέες Γλώσσες δημιουργείται αρχικά ένας Διερμηνευτής, και αργότερα απλά

αλλάζει το Οπίσθιο τμήμα μετατρέποντάς τον σε Μεταγλωττιστή.

Στη πράξη όμως, ειδικά όταν πρόκειται για εμπορικούς Μεταγλωττιστές, εφαρμόζεται η Αρχιτεκτονική **Οργάνωση σε Περάσματα** (N-pass ή N-scan Compiler). Σκοπός αυτής της αρχιτεκτονικής είναι να μειώσει τις απαιτήσεις του Μεταγλωττιστή από το περιβάλλον εκτέλεσής του. Πάλι, οι φάσεις υλοποιούνται ως λειτουργικές μονάδες, μόνο που τώρα δεν περιμένει η επόμενη μονάδα την προηγούμενη να ολοκληρώσει την επεξεργασία του προγράμματος στο σύνολό του. Επιπλέον, η δομή των λειτουργικών μονάδων αλλάζει, αφού ουσιαστικά υλοποιούνται ως ρεύματα δεδομένων. Κατά την φάση του σχεδιασμού του Μεταγλωττιστή επιλέγεται μία από τις λειτουργικές μονάδες που επωμίζεται τον ρόλο του ελέγχου της όλης διαδικασίας. Ρόλος της, είναι να ζητά δεδομένα από τις προηγούμενες και αφού τα επεξεργαστεί, να τα προωθεί στις επόμενες. Επομένως, οι υπόλοιπες λειτουργικές μονάδες χωρίζονται σε δύο κατηγορίες, τις πριν και τις μετά μονάδα ελέγχου, και διαφέρουν στην υλοποίησή τους. Όσες προηγούνται, κάθε φορά που τους ζητείται να παράξουν μία λογική μονάδα πληροφορίας, ζητούν τα απαραίτητα δεδομένα από την προηγούμενή τους μονάδα και αφού τα επεξεργαστούν τα επιστρέφουν στην επόμενη. Σε περίπτωση που δεν έχει κάτι να τους δώσει η προηγούμενη μονάδα ενημερώνουν για το τέλος του προγράμματος την επόμενη. Αυτές που ακολουθούν όταν δεχτούν μία λογική μονάδα πληροφορίας την επεξεργάζονται και προωθούν το αποτέλεσμα στην επόμενη. Προφανώς η πρώτη και η τελευταία μονάδα έχουν ειδικές λειτουργίες. Η πρώτη λειτουργική μονάδα αντί να ζητήσει δεδομένα από κάποια προηγούμενη, τα ανακτά από κάποιον φάκελο (file) στην μνήμη του Η/Υ, ενώ η τελευταία αποθηκεύει το αποτέλεσμα σε κάποιον άλλο φάκελο. Φυσικά η μονάδα ελέγχου, όταν λάβει μήνυμα για το

τέλος του προγράμματος, έχοντας ενημερώσει την επόμενη μονάδα, τερματίζει την εκτέλεση.

Εξαιτίας της συγκεκριμένης οργάνωσης, ο Μεταγλωττιστής μπορεί να έχει χαμηλές απαιτήσεις σε πόρους του συστήματος, αφού κάθε φορά καταπιάνεται μόνο με το μέρος του προγράμματος που έχει εκείνη την στιγμή μπροστά του, αρκεί να διατηρεί τις πληροφορίες του προγράμματος που αφορούν το σύνολό του, όπως παραδείγματος χάριν τύπους δεδομένων προσβάσιμους από το σύνολο του προγράμματος ή υπογραφές συναρτήσεων/μεθόδων. Όμως δεν εγγυώνται όλες οι Αρχικές Γλώσσες ότι αυτές οι πληροφορίες θα εμφανιστούν στο πρόγραμμα πριν χρειαστούν. Για να λυθεί το συγκεκριμένο πρόβλημα εφαρμόζεται η τεχνική των πολλαπλών περασμάτων. Ουσιαστικά γίνεται παραπάνω από μία φορές η επεξεργασία του προγράμματος (δύο, τρεις ή και συνηθέστερα περισσότερες) σε διάφορες λειτουργικές μονάδες με σκοπό την συλλογή των απαραίτητων πληροφοριών για την τελική επεξεργασία. Ενδεικτικά Γλώσσες που απαιτούν πέραν του ενός περασμάτων είναι η **ALGOL 60** (τουλάχιστον 2 περάσματα) και η **ALGOL 68** (τουλάχιστον 3). Όμως οι νεότερες Γλώσσες τείνουν να σχεδιάζονται, έχοντας κατά νου τα παραπάνω, έτσι ώστε να μπορούν να μεταγλωττιστούν σε ένα πέρασμα, προσθέτοντας σε αυτές χαρακτηριστικά, όπως η δήλωση της υπογραφής μίας συνάρτησης ή ο ορισμός μίας κλάσης πριν από τη χρήση τους. Παραδείγματα τέτοιων Γλωσσών είναι η **Pascal**,**C**, **C++**, **Java** και άλλες συγγενικές αυτών.

1.2.5 Επιθυμητά Χαρακτηριστικά ενός Μεταγλωττιστή

Για να θεωρηθεί ένας Μεταγλωττιστής χρήσιμος είναι απαραίτητο να διαθέτει κάποια κοινώς αποδεκτά χαρακτηριστικά.

Το σημαντικότερο από αυτά είναι να παράγει **ορθό κώδικα**, με την έννοια του κώδικα που εκτελείται χωρίς λάθη στην τελική μηχανή αλλά πολύ περισσότερο είναι ισοδύναμος[4] με τον πηγαίο.

Αναγκαίο είναι, ακόμη, να συμμορφώνεται πλήρως με τον **Ορισμό της Γλώσσας** την οποία Μεταγλωττίζει. Σε αντίθετη περίπτωση, παρόλο που θα είναι χρήσιμος και θα παράγει σωστά προγράμματα, θα πάσχει όσον αφορά την μεταφερσιμότητα του πηγαίου κώδικα[5]. Ειδικότερα, δεν θα είναι σε θέση να μεταγλωττίσει προγράμματα που μεταγλωττίζονται από κάποιον άλλον Μεταγλωττιστή, ενώ ταυτόχρονα όσα ο Μεταγλωττιστής αυτός μεταγλωττίζει δεν θα μπορούν να μεταγλωττιστούν από άλλους. Το συνηθέστερο είναι οι κατασκευαστές του Μεταγλωττιστή να εμπλουτίζουν την Γλώσσα με χαρακτηριστικά τα οποία θεωρούν ότι θα επιτύχουν μεγιστοποίηση της παραγωγικότητας του προγραμματιστή. Σε κάποιες άλλες περιπτώσεις οι κατασκευαστές Μεταγλωττιστών για ιστορικούς λόγους ή για λόγους συμβατότητας δεν συμπεριλαμβάνουν όλα τα χαρακτηριστικά μιας Γλώσσας. Αυτό έχει σαν αποτέλεσμα να υλοποιούν υποσύνολα της Γλώσσας, όπως παραδείγματος χάρην η C++ και οι διάφοροι Μεταγλωττιστές της.

Επιπλέον, δεν πρέπει να υπάρχουν **περιορισμοί στο μέγεθος των προγραμμάτων** που μπορεί να Μεταγλωττίσει [GBJL02]. Τα προγράμματα πρέπει

[4]Ισοδύναμος με την έννοια του ότι παράγει τα ίδια αποτελέσματα με τα ίδια δεδομένα εισαγωγής.

[5]Πηγαίος κώδικας ονομάζεται ένα οποιοδήποτε πρόγραμμα όταν αυτό είναι αποτυπωμένο σε κάποια **Αρχική Γλώσσα**

να μπορούν να είναι οσοδήποτε μεγάλου μεγέθους, χωρίς να δημιουργούν πρόβλημα στον Μεταγλωττιστή, φυσικά μέσα στα όρια του συστήματος στο οποίο εκτελείται. Τέτοιοι περιορισμοί στο μέγεθος μπορούν να είναι, πέρα από το συνολικό μήκος του προγράμματος, ο αριθμός των παραμέτρων μίας συνάρτησης/μεθόδου ή το μήκος των αναγνωριστικών. Οι περιορισμοί αυτοί ίσως εισάγονται με στόχο την ευκολία της υλοποίησης, έχοντας κατά νου τις επιλογές που θα μπορούσαν να γίνουν από προγραμματιστές, όμως πρέπει να λαμβάνεται υπόψιν ότι οι Μεταγλωττιστές πρέπει να είναι σε θέση να εξυπηρετήσουν γεννήτριες προγραμμάτων[6] όπως είναι οι Μεταγλωττιστές Μεταγλωττιστών[7] στους οποίους ανήκουν ο lex [8] και ο yacc[9].

Σημαντικό είναι να εκτελεί την διαδικασία της Μεταγλώττισης με ικανοποιητική **ταχύτητα**. Σήμερα, οι αποδεκτοί χρόνοι είναι μερικά δευτερόλεπτα για μικρού μεγέθους προγράμματα και μερικά λεπτά για αρκετά μεγάλα προγράμματα με πολλές χιλιάδες γραμμές κώδικα. Ουσιαστικά, η διάρκεια της διαδικασίας θα πρέπει να είναι αρκετά σύντομη ώστε να μην προκαλεί δυσκολίες στην ανάπτυξη του λογισμικού. Το ιδεατό είναι η αύξηση του χρόνου σε σχέση με το μέγεθος του προγράμματος να είναι γραμμική. Αυτό το πρόβλημα συνήθως αντιμετωπίζεται με σχετική ευκολία από τους Μεταγλωττιστές με Οργάνωση σε Περάσματα (1.2.4) εξαιτίας της φύσης τους.

Πέρα από την ταχύτητα παραγωγής του κώδικα πρέπει να προσεχθεί και η

[6]Πρόκειται για προγράμματα τα οποία με βάση πληροφορίες που περιγράφουν κάποιο πρόγραμμα παράγουν το πρόγραμμα αυτό σε κάποια εκ των προτέρων καθορισμένη Γλώσσα.

[7]Είναι εφαρμογή η οποία δοθείσης της περιγραφής και των λεξημάτων μιας Γλώσσας, παράγει κώδικα σε κάποια επιλεγμένη Γλώσσα, ο οποίος υλοποιεί μία λειτουργική μονάδα του Μεταγλωττιστή (1.2.3).

[8]Εφαρμογή η οποία δοθείσης της σύνταξης μίας Γλώσσας παράγει τον Λεξικό Αναλυτή (2) του Μεταγλωττιστή. Η GNU έκδοσή του ονομάζεται flex.

[9]Εφαρμογή η οποία δοθείσης της σύνταξης μίας Γλώσσας παράγει τον Συντακτικό Αναλυτή (3) του Μεταγλωττιστή. Η GNU έκδοσή του ονομάζεται bison.

ταχύτητα των βελτιστοποιήσεων. Αν και κάτι τέτοιο δεν είναι εύκολο να επιτευχθεί. Η δυσκολία εντοπίζεται συνήθως στο ότι για τις φάσεις βελτιστοποίησης είναι αναγκαία η ανάλυση του κώδικα, πέραν των άλλων, και ως προς τις συσχετίσεις των επιμέρους μερών του προγράμματος μεταξύ τους, πράγμα που απαιτεί μεγάλα ποσά μνήμης και ανάλογη επεξεργαστική ισχύ.

Ένα δυνατό χαρακτηριστικό ενός Μεταγλωττιστή, που είχε ιδιαίτερη σημασία παλαιότερα όχι όμως τα τελευταία χρόνια, είναι το όσο δυνατό μικρότερο **μέγεθος του Μεταγλωττιστή**. Είναι απαραίτητο ο Μεταγλωττιστής όχι μόνο να χωρά στην διαθέσιμη μνήμη του υπολογιστή αλλά να μένει και αρκετή μνήμη για την εκτέλεση της Μεταγλώττισης. Χαρακτηριστικό παράδειγμα της σημασίας του μεγέθους του Μεταγλωττιστή είναι πως οι πρώτες εκδόσεις του Λειτουργικού Συστήματος Linux δεν μπορούσαν να Μεταγλωττιστούν στο ίδιο το Λειτουργικό αφού η τότε διαθέσιμη μνήμη τυχαίας προσπέλασης (RAM) των Η/Υ δεν ήταν αρκετή για να φιλοξενήσει τους Μεταγλωττιστές της C ενώ το Linux δεν υποστήριζε ακόμη την τεχνολογία της Εικονικής Μνήμης. Βέβαια σήμερα ο περιορισμός αυτός δεν υφίσταται δεδομένου ότι η διαθέσιμη μνήμη είναι υπερπολλαπλάσια από τα τυπικά μεγέθη των Μεταγλωττιστών.

Τέλος, είναι αναμενόμενο ο Μεταγλωττιστής να είναι φιλικός προς τον χρήστη, πράγμα το οποίο κατά βάση μεταφράζεται ως **αναφορά σφαλμάτων**. Είναι δεδομένο ότι κατασκευάζοντας ένα πρόγραμμα θα γίνουν αρκετά λάθη. Επομένως ο Μεταγλωττιστής είναι απαραίτητο να εμφανίζει ανάλογα μηνύματα που επεξηγούν το λάθος, εμφανίζουν τον φάκελο αλλά και την γραμμή στην οποία εμφανίστηκε το σφάλμα και όταν αυτό είναι δυνατό, να κάνουν προτάσεις πάνω στον τρόπο με τον οποίο θα μπορούσαν να γίνουν οι διορθώσεις. Βασική προϋπόθεση είναι να μην τερματίζει η διαδικασία της Μεταγλώττισης

στο πρώτο σφάλμα που θα βρει, αντίθετα πρέπει να συνεχίζει ως το τέλος με σκοπό να εμφανίζει το σύνολο των σφαλμάτων που μπορεί να εντοπίσει στο πρόγραμμα. Για να επιτευχθεί η ανάνηψη μετά από κάποιο σφάλμα ο Μεταγλωττιστής είτε δοκιμάζει να συμπληρώσει κάποια πληροφορία που λείπει, είτε να το αγνοήσει και να συνεχίσει. Δεν μπορούν όμως όλα τα σφάλματα να εντοπιστούν από τον Μεταγλωττιστή. Τα σφάλματα που μπορούν να υπάρξουν μέσα σε ένα πρόγραμμα κατηγοριοποιούνται ως εξής:

1. Τα **Λεξικά σφάλματα**, όπως είναι ένα μη έγκυρο αναγνωριστικό (π.χ. ένα όνομα μεταβλητής που ξεκινά με αριθμό στις περισσότερες Γλώσσες, είναι μία κατηγορία που εντοπίζει στο σύνολό της ο Μεταγλωττιστής κατά την φάση της Λεκτικής Ανάλυσης (2).

2. Τα **Συντακτικά σφάλματα**, όπως είναι ο αριθμός των παραμέτρων μίας κλήσης συνάρτησης/μεθόδου, μπορούν και αυτά να εντοπιστούν με ευκολία κατά την φάση της Συντακτικής Ανάλυσης (3). Είναι γεγονός πως το μεγαλύτερο μέρος των σφαλμάτων που εντοπίζει ένας Μεταγλωττιστής βρίσκεται στην Συντακτική Ανάλυση, αφού είναι αρκετά εύκολη η αναγνώρισή τους βάση του ορισμού της αρχικής Γλώσσας.

3. Τα **Σημασιολογικά σφάλματα**, όπως είναι το πέρασμα παραμέτρων σε μία συνάρτηση/μέθοδο άλλων από αυτή που ορίζει η υπογραφή της, εντοπίζονται στην φάση της Σημασιολογικής Ανάλυσης (3), αν και ο εντοπισμός τους δεν είναι τόσο εύκολος.

4. Τα **Σφάλματα εκτέλεσης**, όπως είναι η εισαγωγή του προγράμματος σε έναν ατέρμονα βρόγχο χωρίς την δυνατότητα ενεργοποίησης κάποιας

συνθήκης εξόδου, τα οποία ο Μεταγλωττιστής αδυνατεί να εντοπίσει. Όπως υποδηλώνει και το όνομά τους γίνονται αντιληπτά κατά την εκτέλεση, όταν το πρόγραμμα σταματά να αποκρίνεται ή τερματίζεται αναπάντεχα η εκτέλεσή του.

5. Τα **Λογικά σφάλματα**, όπως είναι ο έλεγχος ανισότητας[10] όταν το πρόβλημα απαιτεί ανισοϊσότητα[11], τα οποία γίνονται αντιληπτά συγκρίνοντας τα αποτελέσματα της εκτέλεσης του προγράμματος με τα αναμενόμενα.

Πέρα από τα μηνύματα για σφάλματα του προγράμματος, ο Μεταγλωττιστής μπορεί να εμφανίσει και μηνύματα για εσωτερικά σφάλματα που αφορούν την εκτέλεσή του ή προειδοποιήσεις. Οι τελευταίες σκοπό έχουν είτε να επιστήσουν την προσοχή του προγραμματιστή σε δομές του κώδικα, που συνηθίζεται να κρύβουν σφάλματα ή ακόμη να ενημερώσουν για πιθανή αλλαγή στο τρόπο ερμηνείας κάποιων εντολών από τον Μεταγλωττιστή σε μελλοντικές εκδόσεις.

1.3 Θεωρία Γλωσσών

Ενώ οι Φυσικές Γλώσσες που χρησιμοποιούν οι άνθρωποι ανά τον κόσμο για την καθημερινή τους επικοινωνία, την καλλιέργεια του πολιτισμού και την διατήρηση των ιστορικών γεγονότων, που αποτελούν την ταυτότητά τους, είναι γεμάτες από αοριστολογίες και αμφισημίες, ανήκουν στις εξαρτώμενες, από το περιβάλλον (context sensitive) γλώσσες, οι γλώσσες προγραμματισμού είναι ανεξάρτητες περιβάλλοντος (context free). Θα επικεντρώσουμε το ενδιαφέρον

[10] $<$ ή $>$
[11] $\leq$ ή $\geq$

μας σε αυτές τις τελευταίες, αφού πρώτα κάνουμε μία γενική επισκόπηση των τυπικών, λεγομένων, Γλωσσών.

1.3.1 Σύμβολο

Η έννοια του **συμβόλου** (symbol) στην Πληροφορική δεν ορίζεται με χρήση άλλων γνωστών εννοιών, που έχουν οριστεί προηγουμένως. Πρόκειται για μία βασική έννοια, που είναι αναγκαίο να κατακτήσει καθένας που θέλει να καταπιαστεί με πεδία των Μεταγλωττιστών και συναφών τεχνολογιών. Για το σκοπό αυτό, η έννοια πρέπει να εμπεδωθεί εμπειρικά, μέσα, δηλαδή, από παραδείγματα συμβόλων και αντιπαραδείγματα "μη συμβόλων".

Σύμβολα χρησιμοποιούμε καθημερινά στη ζωή μας. Τέτοια είναι τα Γράμματα της Ελληνικής Αλφαβήτου $\{A, B, \Gamma, ..., X, \Psi, \Omega, \alpha, \beta, \gamma, ..., \chi, \psi, \omega\}$, τα Γράμματα της Λατινικής Αλφαβήτου $\{A, B, C, ..., X, Y, Z, a, b, c, ..., z, y, z\}$, τα ψηφία της αραβικής αρίθμησης $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, τα σήματα οδικής κυκλοφορίας κ.α. Τα βουνά, τα ποτάμια, τα μέλη της οικογενείας μας, ο εθνικός μας ύμνος κ.α. δεν είναι σύμβολα, Ο διαχωρισμός δεν είναι πάντα εύκολος και σαφείς. Μία ελληνική σημαία δεν είναι σύμβολο, μία εικόνα της, όμως, μπορεί να θεωρηθεί σύμβολο...

Επιπλέον τα σύμβολα τα απαντούμε με διαφορετικές ονομασίες, όπως γράμμα (letter), μάρκα, χαρακτήρας (character), ταυτοποιητής (identifier), όνομα κ.ο.κ.

1.3.2 Αλφάβητο

Κάθε **μη κενό** και **πεπερασμένο** σύνολο συμβόλων ορίζεται ως Αλφάβητο. Τα Αλφάβητα χαρακτηρίζονται από τον αριθμό των συμβόλων που περιέχουν

και όχι από τα σύμβολα αυτά καθεαυτά. Έτσι έχουμε το Αλφάβητο $\Sigma_1 = \{I\}$ του ενός συμβόλου, το Αλφάβητο των 2 συμβόλων π.χ. $\Sigma = \{0, 1\}$ ή $\Sigma = \{\alpha, \beta\}$, το Αλφάβητο των 3 συμβόλων π.χ. $\Sigma = \{1, 2, 3\}$ ή $\Sigma = \{x, y, z\}$... Επιπλέον, από τη στιγμή που τα Αλφάβητα είναι σύνολα, ισχύουν για αυτά οι **πράξεις συνόλων**, όπως η τομή $\cap$, η ένωση $\cup$ και η διαφορά $\setminus$ και η συμπληρωματικότητα ως προς ένα σύμπαν U, που συνήθως είναι ένα υπεραλφάβητο.

1.3.3 Συμβολοσειρά

Η παράθεση πεπερασμένου πλήθους συμβόλων ενός αλφαβήτου, όχι απαραίτητα διαφορετικών, ονομάζεται συμβολοσειρά (string). Επομένως, δεδομένου ενός αλφαβήτου $\Sigma_3 = \{\alpha, \beta, \gamma\}$, μπορούμε να έχουμε, μεταξύ άλλων, τις εξής συμβολοσειρές:

- Η κενή συμβολοσειρά, που συμβολίζεται με το ελληνικό γράμμα ε και έχει μηδενικό πλήθος συμβόλων.
- Οι συμβολοσειρές "α", "β", "γ" μήκους ενός συμβόλου.
- Οι συμβολοσειρές "$\alpha\alpha$", "$\alpha\beta$", "$\alpha\gamma$", "$\beta\alpha$", "$\beta\beta$", "$\beta\gamma$", "$\gamma\alpha$", "$\gamma\beta$", "$\gamma\gamma$" μήκους δύο συμβόλων.
- Οι συμβολοσειρές "$\alpha\alpha\alpha$", "$\alpha\alpha\beta$", "$\alpha\alpha\gamma$", "$\alpha\beta\alpha$", "$\alpha\beta\beta$", "$\alpha\beta\gamma$", "$\alpha\gamma\alpha$", "$\alpha\gamma\beta$", "$\alpha\gamma\gamma$", "$\beta\alpha\alpha$", "$\beta\alpha\beta$", "$\beta\alpha\gamma$", "$\beta\beta\alpha$", "$\beta\beta\beta$", "$\beta\beta\gamma$", "$\beta\gamma\alpha$", "$\beta\gamma\beta$", "$\beta\gamma\gamma$", "$\gamma\alpha\alpha$", "$\gamma\alpha\beta$", "$\gamma\alpha\gamma$", "$\gamma\beta\alpha$", "$\gamma\beta\beta$", "$\gamma\beta\gamma$", "$\gamma\gamma\alpha$", "$\gamma\gamma\beta$", "$\gamma\gamma\gamma$" μήκους τριών συμβόλων.

- Οι συμβολοσειρές "αααα", "αααβ", "αααγ", "ααβα", "ααββ", "ααβγ", "ααγα", "ααγβ", "ααγγ", "αβαα", "αβαβ", "αβαγ", "αββα", "αβββ", "αββγ", "αβγα", "αβγβ", "αβγγ", "αγαα", "αγαβ", "αγαγ", "αγβα", "αγββ", "αγβγ", "αγγα", "αγγβ", "αγγγ", "βααα", "βααβ"... μήκους τεσσάρων συμβόλων.

- κ.ο.κ.

1.3.4 Ορισμός της Τυπικής Γλώσσας

Κάθε σύνολο, περασμένου ή απείρου πλήθους, συμβολοσειρών, του κενού συνόλου μη εξαιρουμένου, ονομάζεται τυπική Γλώσσα. Το κενό σύνολο ονομάζεται κενή Γλώσσα και τα σύνολα απείρου πλήθους συμβολοσειρών, απειρογλώσσες. Το σύνολο μιας συμβολοσειράς που είναι κενή (ε) είναι προφανώς διάφορο του κενού συνόλου (1.1) και άρα δεν είναι κενή Γλώσσα.

$$\{\varepsilon\} \neq \emptyset \quad \text{ή} \quad \{\varepsilon\} \neq \{\} \tag{1.1}$$

Όμως, στην πράξη τα σύμβολα, αυτά καθεαυτά, δεν παίζουν σημαντικό ρόλο (1.3.2) στην δομή του συνόλου των συμβολοσειρών μίας κανονικής γλώσσας. Μπορούμε, για παράδειγμα, να αλλάξουμε τα 26 λατινικά σύμβολα του αλφαβήτου της γλώσσας C με τα 24 ελληνικά και τα σύμβολα □,⊠, χωρίς, το σύνολο των συμβολοσειρών της να αλλοιωθεί στο ελάχιστο.

Κεφάλαιο 2

ΛΕΞΙΚΟΙ ΑΝΑΛΥΤΕΣ

2.1 Τί είναι οι Λεξικοί Αναλυτές

2.1.1 Μία σύντομη περιγραφή

Οι **Λεξικοί Αναλυτές** (σαρωτές - scanners) αποτελούν την πρώτη συνιστώσα ενός μεταγλωττιστή, που έρχεται σε επαφή με τον πηγαίο κώδικα. Ο ρόλος τους είναι να παραλάβουν το ρεύμα συμβόλων, που αποτελεί τον πηγαίο κώδικα τους προς μετάφραση προγράμματος και να το μετατρέψουν σε ένα ρεύμα γλωσσικών συμβόλων (σύνθετων ως επί το πλείστον)[1] το οποίο θα χρησιμοποιηθεί από τις επόμενες συνιστώσες του μεταγλωττιστή.

2.1.2 Οι αρχές λειτουργίας τους

Ένας λεξικός αναλυτής αναφέρεται πάντα σε συγκεκριμένη γλώσσα προγραμματισμού της οποίας εκτελεί την λεξική ανάλυση. Ο κατασκευαστής του λεξικού αναλυτή ξεκινά με την δημιουργία μίας τυπικής περιγραφής των συμβόλων της γλώσσας. Εργαλεία του στο έργο αυτό είναι οι **κανονικές εκφράσεις** (2.1.3). Με αυτές μπορεί να περιγράψει τα σύμβολα μίας γλώσσας με ακρίβεια και χωρίς αμφισημίες. Στη συνέχεια βασιζόμενος στην τυπική αυτή περιγραφή προχωρά στην δημιουργία **πεπερασμένων αυτομάτων** (2.1.5) τα οποία όταν καταλήξουν στην μορφή του **Ντετερμινιστικού Πεπερασμένου Αυτομάτου** (2.1.6) είναι έτοιμα να μετατραπούν σε κώδικα. Το εκτελέσιμο που παράγεται από τον κώδικα αυτόν είναι ο λεξικός αναλυτής. Αξίζει να σημειωθεί πως μετά τον ορισμό των συμβόλων της γλώσσας τα υπόλοιπα βήματα είναι αλγοριθμικά και επομένως μπορούν να αναπαραχθούν από ένα πρόγραμμα ηλεκτρονικού

[1] Σύνθετα σύμβολα ονομάζονται συμβολοσειρές απλών συμβόλων στις οποίες, συνήθως, έχουμε προσδώσει κάποια σημασία (εξ' ορισμού). π.χ. τα σύμβολα while, for, case είναι σύνθετα γλωσσικά σύμβολα της γλώσσας C

υπολογιστή. Τέτοιο πρόγραμμα είναι η γεννήτρια προγραμμάτων lex και η GNU έκδοσή της, flex, που χρησιμοποιείται και στην συγκεκριμένη πτυχιακή εργασία. Αποτελεί ευχάριστη έκπληξη, τουλάχιστον για τους λιγότερους ειδικούς, ότι η ποιότητα του παραγόμενου κώδικα από τέτοια προγράμματα είναι σχεδόν πάντα καλύτερη από τα αντίστοιχα χειροποίητα. Γι αυτό προτιμάται η αυτόματη παραγωγή των λεξικών αναλυτών.

Συνοπτικά τα βήματα που ακολουθούνται για την δημιουργία ενός χειροποίητου λεξικού αναλυτή είναι τα εξής:

- **Ορισμός των συμβόλων της γλώσσας** που πρόκειται να αναγνωριστεί με χρήση κανονικών εκφράσεων
- Από τον ορισμό αυτό για κάθε κανονική έκφραση **παράγουμε ένα αυτόματο** που την αναγνωρίζει.
- **Τα αυτόματα αυτά ενώνονται** για να παραχθεί ένα αυτόματο το οποίο αναγνωρίζει το σύνολο της γλώσσας.
- Το ενιαίο αυτόματο μετατρέπεται σε **ντετερμινιστικό πεπερασμένο και πλήρες αυτόματο**.
- Η τελευταία μορφή του ενιαίου αυτομάτου **υλοποιείται** από ένα πρόγραμμα Η/Υ.

Τα παραπάνω βήματα ακολουθούν, εξ' άλλου, και οι γεννήτριες.

Όπως αναφέραμε παραπάνω, ένα από τα πλέον σημαντικά και καθοριστικά στοιχεία μίας γλώσσας είναι οι κανόνες με τους οποίους σχηματίζονται τα σύμβολα της Γλώσσας. Οι κανόνες αυτοί μπορούν να αποτυπωθούν τυπικά και με σαφήνεια χρησιμοποιώντας συγκεκριμένα εργαλεία. Στην λεκτική ανάλυση

γλωσσών προγραμματισμού χρησιμοποιούνται συνήθως οι **κανονικές εκφράσεις**. Το εργαλείο αυτό έχει ένα σημαντικό χαρακτηριστικό, απαραίτητο για τον ορισμό των συμβόλων μίας Γλώσσας Προγραμματισμού, μας επιτρέπει να παράγουμε τους ορισμούς των συμβόλων των γλωσσών, χωρίς αοριστίες και αμφισημίες. Στη συνέχεια, οι Κανονικές Εκφράσεις θα περιγραφούν εκτενέστερα.

2.1.3 Κανονικές Εκφράσεις

Μία κανονική έκφραση μας επιτρέπει να ορίσουμε ένα σύνολο συμβολοσειρών χωρίς να χρειαστεί να αναγράψουμε μία προς μία τις συμβολοσειρές αυτές. Για τη σύνταξη των κανονικών εκφράσεων σε ένα αλφάβητο A χρησιμοποιούμε ένα διευρυμένο αλφάβητο $A \cup \{\cdot, |, *, (,)\}$. Ακολουθούν οι αρχικές (primitive) κανονικές εκφράσεις.

- $\emptyset$ ή $\{\}$ είναι κανονική έκφραση που περιγράφει τη Γλώσσα (σύνολο συμβολοσειρών) L_0 ή L^0 ή $\{\}$ ή $\emptyset$.
- c είναι η κανονική έκφραση που περιγράφει τη Γλώσσα $\{\text{“}c\text{”}\}$.

Πίνακας 2.1: Τελεστές κανονικών εκφράσεων

Τελεστής	**Κανονική Έκφραση**	**Γλώσσα**
$\cdot$	$r_1 \cdot r_2$ ή συχνά $r_1 r_2$	$L_{r_1} \cdot L_{r_2}$ ή συχνά $L_{r_1} L_{r_2}$
$\mid$	$r_1 \mid r_2$	$L_{r_1} \cup L_{r_2}$
$*$	r^* ή $r*$	$L^0 \cup L^1 \cup L^2 \cup L^3 \cup ... \cup L^v$
$+$	r^+ ή $r+$	$L^1 \cup L^2 \cup L^3 \cup ... \cup L^v$

όπου r_1, r_2, r είναι κανονικές εκφράσεις από τις από τις αρχικές (primitive) ή και συνθέσεις τους και

- $L^0 = \{\varepsilon\}$
- $L^1 = L$, όπου L γλώσσα των συμβολοσειρών του ενός συμβόλου για κάθε σύμβολο του αλφαβήτου
- $L^x = L^{x-1}L$

2.1.4 Θεωρία Αυτομάτων

Η **Θεωρία Αυτομάτων** καταπιάνεται με την μελέτη αφηρημένων μηχανών (αυτομάτων) και τα προβλήματα που αυτές επιλύουν. Είναι συνυφασμένη με την **Θεωρία Γλωσσών** αφού η κατηγοριοποίηση των αυτομάτων γίνεται βάση των **κλάσεων Γλωσσών** που αυτά αναγνωρίζουν. Επιπλέον, τα αυτόματα είναι Μαθηματικά μοντέλα **Μηχανών Πεπερασμένων Καταστάσεων** (Finite State Machine), τα οποία δοθέντος ενός ρεύματος συμβόλων (1.3.1), "καταναλώνουν" ένα – ένα τα σύμβολα και μεταβαίνουν, βάση μίας **συνάρτησης μετάβασης**, η οποία λαμβάνει υπόψιν το τρέχον σύμβολο και την τρέχουσα κατάσταση, σε μία επόμενη κατάσταση. Η συνάρτηση μετάβασης συνηθίζεται να κωδικοποιείται ως **πίνακας μετάβασης**. Ένα αυτόματο έχει μία η περισσότερες **αρχικές καταστάσεις**, σε μία από τις οποίες βρίσκεται όταν καταναλώνει το πρώτο σύμβολο. Ακόμη, κάποιες από τις καταστάσεις ονομάζονται **τερματικές καταστάσεις** και όταν το αυτόματο βρίσκεται σε μία από αυτές, μετά την κατανάλωση του τελευταίου συμβόλου, τότε το αυτόματο έχει αναγνωρίσει το ρεύμα συμβόλων (ή αλλιώς την **συμβολοσειρά**) ενώ σε αντίθετη περίπτωση λέμε ότι δεν το αναγνώρισε.

Τα αυτόματα χρησιμοποιούνται για να αναγνωρίζουν ένα σύνολο συμβολοσειρών εκ του αυτού αλφαβήτου, δηλαδή μία **Γλώσσα**.

2.1.5 Πεπερασμένα Αυτόματα

Από την Θεωρία Αυτομάτων χρησιμοποιούμε τα **Πεπερασμένα ντετερμινιστικά αυτόματα** για την κατασκευή των Λεξικών Αναλυτών. Είναι το είδος των αυτομάτων που αναγνωρίζει τις **Κανονικές Γλώσσες** στις οποίες ανήκουν τα **σύμβολα των Γλώσσες Προγραμματισμού**.

Συνηθίζεται να περιγράφονται ως μία διατεταγμένη πεντάδα η οποία συμβολίζεται ως $(Q, \Sigma, \delta, q_0, F)$ όπου:

- Q είναι το πεπερασμένο σύνολο **καταστάσεων**
- Σ είναι το σύνολο των συμβόλων που μπορεί να δεχτεί στην είσοδο το Αυτόματο, δηλαδή το **Αλφάβητό** του
- δ είναι η **συνάρτηση μετάβασης** η οποία, δοθήσεις της τρέχουσας κατάστασης ($q_x \in Q$) και του επόμενου συμβόλου εισαγωγής ($\sigma_x \in \Sigma$) επιστρέφει την επόμενη κατάσταση ($q_{x+1} \in Q$)
- q_0 είναι μία κατάσταση ($q_0 \in Q$) στην οποία βρίσκεται το Αυτόματο όταν είναι έτοιμο να αρχίσει να δέχεται τα σύμβολα εισαγωγής και ονομάζεται **αρχική κατάσταση**
- F είναι το σύνολο των **τελικών καταστάσεων**

Συνοπτικά, το αυτόματο ξεκινώντας βρίσκεται στην αρχική κατάσταση q_0, αρχίζει να "καταναλώνει" σύμβολα από την συμβολοσειρά εισαγωγής ένα προς ένα και δίνοντας στην συνάρτηση μετάβασης την κατάσταση που βρίσκεται, και το σύμβολο που μόλις έχει πάρει, μεταβαίνει στη νέα κατάσταση που η συνάρτηση επιστρέφει. Όταν το αυτόματο τερματίσει πλέον, έχουν δηλαδή

"καταναλωθεί" όλα τα σύμβολα της συμβολοσειράς εισαγωγής, τότε αν η κατάσταση στην οποία βρίσκεται το αυτόματο ανήκει στις τελικές καταστάσεις ($q_\tau \in F$) θεωρούμε ότι έχει αναγνωρίσει την συμβολοσειρά εισαγωγής, διαφορετικά λέμε ότι δεν αναγνωρίζει τη συμβολοσειρά αυτή. Επιπλέον, αν κατά την διάρκεια της λειτουργίας του αυτομάτου η συνάρτηση μετάβασης δεν επιστρέψει κάποια επόμενη κατάσταση τότε πάλι θεωρούμε ότι δεν αναγνωρίζεται η συμβολοσειρά και το αυτόματο σταματά (μη πλήρη αυτόματα).

Τα Πεπερασμένα Αυτόματα κατηγοριοποιούνται βάση του τρόπου που εκτελούνται οι μεταβάσεις από την μία κατάσταση στη επόμενη στις εξής κατηγορίες:

- Ντετερμινιστικά Πεπερασμένα Αυτόματα – DFA (2.1.6)
- Μη-Ντετερμινιστικά Πεπερασμένα Αυτόματα – NFA (2.1.7)
- Μη-Ντετερμινιστικά Πεπερασμένα Αυτόματα με ε-μεταβάσεις – NFA-ε (2.1.8)

Κάθε μία από τις πιο πάνω κατηγορίες αυτομάτων, μετατρέπεται σε ισοδύναμο πεπερασμένο ντετερμινιστικό πλήρες αυτόματο, με ελάχιστο πλήθος καταστάσεων. Ίρα μπορούμε στη μελέτη μας να υποθέσουμε ότι εργαζόμαστε με την πιο πάνω μορφή.

2.1.6 Ντετερμινιστικά Πεπερασμένα Αυτόματα – DFA

Τα Ντετερμινιστικά Πεπερασμένα Αυτόματα είναι ειδικές περιπτώσεις των Πεπερασμένων Αυτομάτων. Με ιδιαίτερα χαρακτηριστικά, ότι η συνάρτηση μετάβασης δ επιστρέφει αυστηρά το πολύ μία επόμενη κατάσταση, έχουν μία μόνο

αρχική κατάσταση και δεν περιέχουν ε-μεταβάσεις, πράγμα το οποίο επιτρέπει την εύκολη αλγοριθμική δημιουργία ενός προγράμματος Η/Υ που εξομοιώνει το αυτόματο.

2.1.7 Μη-Ντετερμινιστικά Πεπερασμένα Αυτόματα – NFA

Τα Μη-Ντετερμινιστικά Πεπερασμένα Αυτόματα είναι πανομοιότυπα με τα Ντετερμινιστικά Πεπερασμένα Αυτόματα, διαφέρουν όμως στο ότι η συνάρτηση μετάβασης επιστρέφει ένα σύνολο επόμενων καταστάσεων, ότι μπορεί να έχουν περισσότερες από μία αρχικές καταστάσεις και τέλος μπορεί να περιέχουν μία ή περισσότερες ε-μεταβάσεις. Αντί λοιπόν να έχουμε μία σειριακή διαδρομή αλληλοσυνδεόμενων καταστάσεων, όπως στα Ντετερμινιστικά Πεπερασμένα Αυτόματα, σε διάφορα σημεία της διαδρομής μπορούν να εμφανιστούν διακλαδώσεις. Φυσικά οι επιπλέον εναλλακτικές διαδρομές που δημιουργούνται, φτάνουν σε αδιέξοδο και απορρίπτονται πριν τον τερματισμό του αυτόματου, ώστε τελικά να καταλήγει κανείς σε μία σειριακή διαδρομή, γι αυτό και τα NFA έχουν πάντα ισοδύναμά τους DFA.

2.1.8 Μη-Ντετερμινιστικά Πεπερασμένα Αυτόματα με ε-μεταβάσεις – NFA-ε

Τα Μη-Ντετερμινιστικά Πεπερασμένα Αυτόματα με ε-μεταβάσεις είναι επεκτάσεις των Μη-Ντετερμινιστικών Πεπερασμένων Αυτομάτων. Η ειδοποιός διαφορά τους έγκειται στις ε–μεταβάσεις. Αυτού του είδους τα αυτόματα έχουν την δυνατότητα να μεταβούν σε κάποιες επόμενες καταστάσεις χωρίς να λάβουν υπόψιν το τρέχον σύμβολο της συμβολοσειράς εισαγωγής και φυ-

σικά χωρίς να το "καταναλώσουν". Το "ε" στην ονομασία "ε–μεταβάσεις" συμβολίζει την κενή συμβολοσειρά "".

2.2 Χειροποίητη Κατασκευή Λεξικού Αναλυτή

2.2.1 Ορισμός των συμβόλων της Γλώσσας

Η κατασκευή ενός Λεξικού Αναλυτή ξεκινά πάντα από τον τυπικό ορισμό των συμβόλων της Αρχικής Γλώσσας του Μεταγλωττιστή. Για τον σκοπό αυτό χρησιμοποιούνται οι Κανονικές Εκφράσεις. Το σύνολο των Κανονικών Εκφράσεων ομαδοποιείται σε κλάσεις, που περιγράφουν τα σύμβολα της Γλώσσας. Τέτοια σύμβολα είναι τα αναγνωριστικά, οι σταθερές (ακεραίων, συμβολοσειρών, κινητής υποδιαστολής), οι δεσμευμένες λέξεις και ονομάζονται λεξήματα. Κάθε κλάση λεξημάτων, που μπορεί συχνά να περιέχει και ένα μόνο λέξημα, ονομάζεται κέρμα (token). Σκοπός του τελευταίου είναι να επιστρέφεται από τον Λεξικό Αναλυτή ο ίδιος κωδικός για λέξημα του ίδιου token, έτσι ώστε να διευκολύνονται τα επόμενα στάδια της Μεταγλώττισης.

2.2.2 Δημιουργία του ανάλογου Αυτομάτου

Κάθε Κανονική Έκφραση μετατρέπεται σε ένα NFA-ε, ακολουθώντας τα βήματα ενός αλγορίθμου (αλγόριθμος Thompson). Το σύνολο αυτό των Πεπερασμένων Αυτομάτων συμπτύσσεται σε ένα μοναδικό NFA-ε. Στην συνέχεια γίνεται η επεξεργασία του NFA-ε και αφαιρώντας του τις ε–μεταβάσεις μετατρέπεται σε ένα NFA και στη συνέχεια σε DFA (2.1). Θα μπορούσε κανείς να θεωρήσει ότι τώρα πια είμαστε έτοιμοι να μετατρέψουμε τα αυτόματά μας σε

κώδικα μίας Γλώσσας Προγραμματισμού και να πάρουμε τον Λεξικό Αναλυτή μεταγλωττίζοντάς τον. Κάτι τέτοιο θα ήταν εφικτό εάν η εισαγωγή του Λεξικού Αναλυτή ήταν σαφώς διαχωρισμένες συμβολοσειρές τις οποίες είτε θα αναγνώριζε είτε θα απέρριπτε. Στην πραγματικότητα η εισαγωγή του Λεξικού Αναλυτή είναι ολόκληρο το Πρόγραμμα ως μία ενιαία συμβολοσειρά. Ο διαχωρισμός της σε επιμέρους τμήματα (λεξήματα) και η κατάταξή τους σε κέρματα (tokens), είναι δουλειά του ίδιου του Λεξικού Αναλυτή. Το γεγονός αυτό εισάγει πολυπλοκότητα στο έργο της αναγνώρισης, την οποία θα πρέπει να αντιμετωπίσουμε τροποποιώντας τα αυτόματά μας.

$$NFA - e \rightarrow NFA \rightarrow DFA \rightarrow DFA - minimal \tag{2.1}$$

Το προφανές πρόβλημα που δημιουργείται είναι, πως δεν γνωρίζουμε πότε θα πρέπει να σταματήσει το αυτόματο να "καταναλώνει" σύμβολα από την είσοδο του και να αποφασίσει αν αναγνωρίζει την υποσυμβολοσειρά, που έχει "καταναλώσει" μέχρι στιγμής. Χαρακτηριστικό παράδειγμα μιας τέτοιας περίπτωσης από την C++ είναι ο τελεστής μοναδιαίας αύξησης "++" και ο τελεστής της πρόσθεσης "+". Τί θα πρέπει να κάνει το αυτόματο όταν, ξεκινώντας την αναγνώριση της επόμενης υποσυμβολοσειράς, καταναλώσει το σύμβολο '+'; Μία επιλογή είναι να σταματήσει και να αποφασίσει ότι αναγνωρίζει τον τελεστή πρόσθεσης, όμως αν ακολουθεί ένα ακόμα '+' κατά πάσα πιθανότητα δεν πρόκειται για τον τελεστή πρόσθεσης αλλά για τον τελεστή μοναδιαίας αύξησης, παρόλα αυτά, το αυτόματο θα αναγνώριζε δύο διαδοχικούς τελεστές πρόσθεσης. Από την άλλη, αν το αυτόματο προχωρήσει στο επόμενο

σύμβολο, περιμένοντας πιθανόν να συναντήσει ένα ακόμα '+' αλλά βρει ένα διαφορετικό σύμβολο τότε δεν θα αναγνωρίσει την υποσυμβολοσειρά και θα χαθεί ο τελεστής πρόσθεσης, που θα έπρεπε να έχει αναγνωρίσει. Η λύση δίνεται τροποποιώντας το αυτόματο έτσι ώστε να έχει την δυνατότητα να δει μερικά από τα επόμενα σύμβολα χωρίς να τα καταναλώσει και να αποφασίσει με ασφάλεια αν αναγνωρίζει μία υποσυμβολοσειρά ή όχι. Αυτό από την πρακτική πλευρά της υλοποίησης σημαίνει πως θα πρέπει να έχει την δυνατότητα να διαβάζει περισσότερα σύμβολα από όσα έχει η υποσυμβολοσειρά και αν χρειαστεί να οπισθοδρομεί. Φυσικά εγείρεται το ερώτημα πώς θα αποφασίσει το αυτόματο, αν θα αναγνωρίσει έναν τελεστή μοναδιαίας προσαύξησης ή δύο τελεστές πρόσθεσης; Η απάντηση βρίσκεται στην σύμβαση ότι το αυτόματο θα αναγνωρίζει πάντα την μακρύτερη δυνατή συμβολοσειρά.

Το δεύτερο σημαντικότερο πρόβλημα είναι, πως δεν μπορούμε να γνωρίζουμε, ποια είναι η υποσυμβολοσειρά που αναγνωρίστηκε από το αυτόματο. Απάντηση έρχεται να δώσει μία ακόμη τροποποίηση του αυτομάτου, όπου ως έξοδο, πέρα από το αν αναγνώρισε ή όχι μία υποσυμβολοσειρά δίνει και την ίδια την υποσυμβολοσειρά που έλεγξε (symbol talbe - 4).

2.2.3 Παραγωγή του Λεξικού Αναλυτή

Έχοντας λοιπόν δημιουργήσει το DFA εμπλουτισμένο με τα προαναφερθέντα αναγκαία χαρακτηριστικά, προχωρούμε στην αλγοριθμική μετατροπή του σε ένα πρόγραμμα Η/Υ, χρησιμοποιώντας μια Αρχική Γλώσσα υψηλού επιπέδου της αρεσκείας μας. Κλασσική επιλογή για το έργο αυτό είναι η Γλώσσα C και σπανιότερα η C++, όταν πρόκειται για Ακαδημαϊκές υλοποιήσεις μπορεί

να χρησιμοποιηθεί οποιαδήποτε Γλώσσα όπως η Java, η Pascal και η Scheme. Τέλος, μετατρέποντας το παραπάνω πρόγραμμα, με την χρήση ενός Μεταγλωττιστή, σε εκτελέσιμο κώδικα αποκτούμε έναν πλήρη Λεξικό Αναλυτή που αναγνωρίζει την Αρχική μας Γλώσσα.

2.3 Κατασκευή Ενός Λεξικού Αναλυτή με χρήση του flex

Όπως έχει προαναφερθεί (2.1.2) από τη στιγμή που έχουμε ορίσει τα σύμβολα της Γλώσσας, με την βοήθεια Κανονικών Εκφράσεων, μπορούμε, αντί της χειροποίητης διαδικασίας, να εισάγουμε τον ορισμό αυτό στην **Γεννήτρια Λεξικών Αναλυτών** flex και να πάρουμε άμεσα τον Λεξικό Αναλυτή που αναγνωρίζει την Αρχική μας Γλώσσα ως ένα πρόγραμμα σε Γλώσσα C. Φυσικά ο ορισμός των συμβόλων της Γλώσσας είναι απαραίτητο να δοθεί στον flex σε τυποποιημένη μορφή, flex, πολύ συγγενική εκείνης των καθαρών κανονικών εκφράσεων.

2.3.1 Η μορφή ενός φακέλου flex

Η περιγραφή των συμβόλων μίας Γλώσσας δίνεται στον flex μέσα σε έναν φάκελο ο οποίος χωρίζεται σε τρία τμήματα και περιέχει επιπλέον πληροφορίες πέρα από την περιγραφή αυτή καθεαυτή.

Ένας τυπικός φάκελος flex έχει την εξής δομή:

Παράθεση 2.1: "Δομή τυπικού φακέλου flex"

```
ορισμοί
```

```
%%
κανόνες
%%
κώδικας χρήστη
```

Στη δομή αυτή μόνο το πρώτο σύμβολο "%%" είναι απαραίτητο, τα υπόλοιπα είναι προαιρετικά.

Το δεσπόζον τμήμα του flex είναι το τμήμα των ορισμών. Το τμήμα των ορισμών αποτελείται από καμία, μία, έως και πεπερασμένου πλήθους προτάσεις, που έχουν τη μορφή περιγραφέας - κενό - δράση. Οι προτάσεις αυτές μπορεί να εκτείνονται σε μία ή περισσότερες φυσικές γραμμές. Κάθε περιγραφέας είναι μία κανονική έκφραση που περιγράφει ένα σύνολο λεξημάτων. Κάθε δράση διατυπώνεται σε μορφή κάποιας ανωτέρου επιπέδου γλώσσας προγραμματισμού, συνήθως C (αυτή χρησιμοποιούμε και στο παρόν πόνημα). Η σημαντική κάθε πρότασης του flex, όπως περιγράφηκε πιο πάνω, είναι:

> Όταν ο παραγόμενος αναλυτής στη σειριακή επεξεργασία του πηγαίου προγράμματος, που του δίδεται στην είσοδό του, συναντήσει ένα λέξημα του περιγραφέα της πρότασης, τότε εκτελεί τις ενέργειες που περιγράφονται στο τμήμα της δράσης, για το τρέχων λέξημα. Η διαδικασία εκτελείται σειριακά για κάθε λέξημα που περιγράφεται από τον περιγραφέα κάθε πρότασης του τμήματος ορισμού του flex.

Για να γίνουν καλύτερα κατανοητά τα προαναφερθέντα παρατίθεται ένας βασικός φάκελος flex ως παράδειγμα.

Παράθεση 2.2: "Βασικός Φάκελος flex"

```
integer     [0-9]+
real        {integer}+"."{integer}


%{
#include <stdio.h>
%}


%%


{integer}     { fprintf("Integer: %s", yytext);}
{real}        { fprintf("Real: %s", yytext);}
.*            { fprintf("Error: Nothing recognised.");}
%%


int main ()
{
    yylex();
    return 0;
}
```

Η περιγραφή αυτή, σε καμία περίπτωση, δεν καλύπτει το εύρος των δυνατοτήτων του εργαλείου lex και κατ' επέκταση της GNU έκδοσής του flex. Σκοπός είναι να περιγραφούν τα απαραίτητα χαρακτηριστικά της Γεννήτριας Λεξικών Αναλυτών τα οποία θα επιτρέψουν την κατανόηση της κατασκευής του πειραματικού Μεταγλωττιστή που συνοδεύει το παρόν πόνημα.

2.4 Ο Λεξικός Αναλυτής του πειραματικού Μεταγλωττιστή

Για την κατασκευή του Λεξικού Αναλυτή χρησιμοποιείται η δεύτερη μέθοδος που βασίζεται στον flex. Η Γλώσσα της οποίας τα σύμβολα πρόκειται να αναγνωρίζει είναι η PCL, μία απλουστευμένη εκδοχή της Pascal, ένας αναλυτικός ορισμός της οποίας βρίσκεται στο Παράρτημα Α. Ο ορισμός αυτός προέρχεται από το βιβλίο "Μεταγλωττιστές των Παπασπύρου/Σκορδαλάκη που υπήρξε το textbook στο μάθημα "Γλώσσες και Μεταγλωττιστές".

2.4.1 Ορισμός της Γλώσσας PCL

Η διαδικασία υλοποίησης ξεκινάει μετατρέποντας τον ορισμό των συμβόλων της Γλώσσας PCL σε μορφή κατάλληλη για να εισαχθεί σε έναν φάκελο flex. Η διαδικασία ξεκινά χωρίζοντας τις συμβολοσειρές σε **Ομάδες** ή **Κλάσεις** (tokens). Εύκολα συμπεραίνει κανείς πως ο λόγος, για αυτού του είδους την ομαδοποίηση, είναι η διευκόλυνση της υλοποίησης των επόμενων τμημάτων του Μεταγλωττιστή. Φυσικά, μερικές κλάσεις περιέχουν ένα μόνο λέξημα, όπως π.χ. οι τελεστές, οι δεσμευμένες λέξεις και οι τελεστές.

Στη συνέχεια δημιουργείται μία Κανονική Έκφραση για κάθε Κλάση. Για την PCL δημιουργήθηκαν οι Κλάσεις και οι Κανονικές Εκφράσεις που τις περιγράφουν (Πίνακας 2.2).

Πίνακας 2.2: Κλάσεις συμβολοσειρών της Γλώσσας PCL

Κλάση	Κανονική Έκφραση	Περιγραφή

and	and	Δεσμευμένη λέξη “and”
array	array	Δεσμευμένη λέξη “array”
begin	begin	Δεσμευμένη λέξη “begin”
boolean	boolean	Δεσμευμένη λέξη “boolean”
char	char	Δεσμευμένη λέξη “char”
dispose	dispose	Δεσμευμένη λέξη “dispose”
div	div	Δεσμευμένη λέξη “div”
do	do	Δεσμευμένη λέξη “do”
else	else	Δεσμευμένη λέξη “else”
end	end	Δεσμευμένη λέξη “end”
false	false	Δεσμευμένη λέξη “false”
forward	forward	Δεσμευμένη λέξη “forward”
function	function	Δεσμευμένη λέξη “function”
goto	goto	Δεσμευμένη λέξη “goto”
if	if	Δεσμευμένη λέξη “if”
integer	integer	Δεσμευμένη λέξη “integer”
label	label	Δεσμευμένη λέξη “label”
mod	mod	Δεσμευμένη λέξη “mod”
new	new	Δεσμευμένη λέξη “new”
nil	nil	Δεσμευμένη λέξη “nil”
not	not	Δεσμευμένη λέξη “not”
of	of	Δεσμευμένη λέξη “of”
or	or	Δεσμευμένη λέξη “or”
procedure	procedure	Δεσμευμένη λέξη “procedure”

program	program	Δεσμευμένη λέξη “program”
real	real	Δεσμευμένη λέξη “real”
result	result	Δεσμευμένη λέξη “result”
return	return	Δεσμευμένη λέξη “return”
then	then	Δεσμευμένη λέξη “then”
true	true	Δεσμευμένη λέξη “true”
var	var	Δεσμευμένη λέξη “var”
while	while	Δεσμευμένη λέξη “while”
name	$[a-zA-Z][a-zA-Z0-9\_]*$	Ονόματα της Γλώσσας
integer	$[0-9]+$	Ακέραιοι αριθμοί
real	$(([0-9]+)+\text{‘.’}...$[2]	Πραγματικοί αριθμοί
const char	$\text{‘}c|(\backslash e)\text{’}$[3]	Σταθερές ενός συμβόλου
const string	“$(d|(\backslash e))*$”[4]	Σταθερές συμβολοσειρές
=	=	Σύμβολο “=”
>	>	Σύμβολο “>”
<	<	Σύμβολο “<”
<>	<>	Σύμβολο “<>”
>=	>=	Σύμβολο “>=”
<=	<=	Σύμβολο “<=”
+	+	Σύμβολο “+”
−	−	Σύμβολο “−”
*	*	Σύμβολο “*”

[2] $(([0-9]+)+\text{‘.’}([0-9]+))|((([0-9]+)+\text{‘.’}([0-9]+))(e|E)(\text{‘+’}|-)([0-9]+))$

[3] όπου c όλα τα σύμβολα που ανήκουν στο Αλφάβητο ASCII εκτός από το σύμβολο “αλλαγή γραμμής” και τα μονά εισαγωγικά, $e \in \{n, t, r, \backslash, \text{‘}, \text{”}\}$.

[4] όπου d όλα τα σύμβολα που ανήκουν στο Αλφάβητο ASCII εκτός από τα σύμβαλα “αλλαγή γραμμής” και τα διπλά εισαγωγικά.

/	/	Σύμβολο “/”
^	^	Σύμβολο “^”
@	@	Σύμβολο “@”
:=	:=	Σύμβολο “:=”
;	;	Σύμβολο “;”
.	.	Σύμβολο “.”
(	(	Σύμβολο “(”
)	)	Σύμβολο “)”
:	:	Σύμβολο “:”
,	,	Σύμβολο “,”
[	[	Σύμβολο “[”
]	]	Σύμβολο “]”
blankchars	[κενό, αλλαγή γραμμής...]	Λευκοί Χαρακτήρες
comments	\(*(.*)*\)	Σχόλια

Οι δύο τελευταίας ψεύτο–Κλάσεις περιγράφουν στον Λεξικό Αναλυτή ότι τα σχετικά λεξήματα δεν μεταφέρονται στα επόμενα στάδια αφού δεν έχουν κάποιο ρόλο αλλά χρησιμοποιούνται για την διευκόλυνση του συντάκτη του Προγράμματος. Δεν ανήκουν λοιπόν σε κάποιο token.

2.4.2 Ο φάκελος flex για την Γλώσσα PCL

Από τον παραπάνω ορισμό δημιουργούμε τον φάκελο flex με βάση τις κανονικές εκφράσεις. Ο Λεξικός Αναλυτής που παράγεται επιστρέφει κάθε φορά έναν ακέραιο (token) που χαρακτηρίζει την κλάση, καθώς και το λέξημα που αναγνώρισε. Παράλληλα, έχει την δυνατότητα να αναφέρει σφάλματα που εντο-

πίζει κατά την ανάλυση. Επιπλέον, υπάρχει μία ψεύτο–Κλάση για τα σχόλια καθώς και μία για τους λευκούς χαρακτήρες (whitespaces). Ότι αναγνωρίζεται με βάση τις δύο αυτές Κλάσεις απλά δεν μεταφέρεται στα επόμενα βήματα. Παρατηρώντας κανείς τον τελικό φάκελο flex (Παράρτημα Β) θα δει ότι χρησιμοποιείται ένα ακόμα χαρακτηριστικό του flex, τις σωρούς καταστάσεων (condition stacks) ως συμπλήρωμα για κάποιες Κλάσεις όπου δεν επαρκούν οι Κανονικές Εκφράσεις, όπως οι σταθερές συμβολοσειρών και τα σχόλια.

Κεφάλαιο 3

ΣΥΝΤΑΚΤΙΚΟΙ ΑΝΑΛΥΤΕΣ

3.1 Τί είναι οι Συντακτικοί Αναλυτές

3.1.1 Μία σύντομη περιγραφή

Ο Συντακτικός Αναλυτής (Parser) είναι το τμήμα του Μεταγλωττιστή που ακολουθεί λογικά μετά τον Λεξικό Αναλυτή (2). Παίρνει τα αποτελέσματα της Λεξικής Ανάλυσης, δηλαδή το Πρόγραμμα μετατρεμένο από ένα ρεύμα χαρακτήρων, σε ένα ρεύμα λεξημάτων (γλωσσικών συμβόλων), και παράγει το Συντακτικό Δέντρο, το οποίο αποτελεί περιγραφή του προγράμματος που δόθηκε στην είσοδο του Μεταγλωττιστή. Αυτό γίνεται με τη βοήθεια του Συντακτικού της Αρχικής Γλώσσας, ενώ παράλληλα με τη δημιουργία του Συντακτικού Δέντρου γίνεται και έλεγχος ως προς την συντακτική ορθότητα του Προγράμματος.

3.1.2 Αρχές λειτουργίας

Όπως αναφέρθηκε ήδη, ο Συντακτικός Αναλυτής τροφοδοτείται με το πρόγραμμα που έχει δοθεί στον Μεταγλωττιστή ως μία ακολουθία λεξημάτων. Θα πρέπει να αναγνωριστούν οι συντακτικές δομές της Αρχικής Γλώσσας, που αποτελούν το υπό εξέταση πρόγραμμα και φυσικά αν κάποιο σημείο του δεν αντιστοιχεί σε μία από τις συντακτικές δομές, να αναφερθεί ως σφάλμα. Περισσότερα για τον χειρισμό σφαλμάτων θα αναφερθούν παρακάτω (3.2.3).

Φυσικά απαραίτητη προϋπόθεση είναι το Συντακτικό της Γλώσσας, να έχει προηγουμένως οριστεί, με τυπικό πάντα τρόπο. Για το σκοπό αυτό χρησιμοποιούμε ένα εργαλείο από τον χώρο των Τυπικών Γλωσσών, τις ανεξάρτητες συμφραζομένων Γραμματικές (CFG - Context-Free Gremmar). Συγκεκρι-

μένα, χρησιμοποιούμε τις ανεξάρτητες συμφραζομένων Γραμματικές, αφού τα χαρακτηριστικά αυτών αφενός, ταιριάζουν με την φύση των γλωσσών προγραμματισμού, αφετέρου είναι εύκολη η κατασκευή προγραμμάτων που τις υλοποιούν είτε χειροποίητα είτε αυτόματα.

Χρησιμοποιώντας τον ορισμό του Συντακτικού της Αρχικής Γλώσσας προχωρούμε στην μετατροπή της περιγραφής του προγράμματος, από ένα ρεύμα Λεξημάτων σε ένα Συντακτικό Δέντρο. Εργαλεία μας για την μετατροπή αυτή είναι μία σειρά από μεθοδολογίες επεξεργασίας του ρεύματος λεξημάτων και παραγωγή του Συντακτικού Δέντρου, η κάθε μία με τις αδυναμίες της και τα δυνατά της σημεία, οι οποίες, βάση του τρόπου αντιμετώπισης του προβλήματος, χωρίζονται σε δύο μεγάλες κατηγορίες, στην **κατιούσα (top-down** και την **ανιούσα (bottom-up)**. Είναι αξιοσημείωτο πως Συντακτικοί αναλυτές βασισμένοι στην **κατιούσα** μεθοδολογία μπορούν είτε να κατασκευαστούν χειροποίητα είτε να παραχθούν από μία γεννήτρια Συντακτικών Αναλυτών, ενώ αυτοί που βασίζονται στην **ανιούσα** μεθοδολογία μπορούν να παραχθούν μόνο από γεννήτριες. Για κάθε μία από τις δύο κατηγορίες υπάρχει μία μεθοδολογία που χρησιμοποιείται ευρέως και επομένως έχει μελετηθεί ανάλογα. Η μία από αυτές είναι η **ανεξάρτητη συμφραζομένων από τα αριστερά προς τα δεξιά και κατιούσα (LL)** μέθοδος και δεύτερη η **ανεξάρτητη συμφραζομένων από τα αριστερά προς τα δεξιά και ανιούσα (LR)** μέθοδος. Οι όροι "ανιολυσα" και "κατιούσα" αναφέρονται στον τρόπο παραγωγής του Συντακτικού Δέντρου, ενώ ο όρος "από τα αριστερά προς τα δεξιά" αναφέρεται στην σειρά με την οποία "καταναλώνονται" τα λεξήματα από το ρεύμα εισαγωγής του Συντακτικού Αναλυτή. Τέλος ο όρος "ανεξάρτητα συμφραζομένων", με απλά λόγια, αναφέρεται στο ότι δεν χρειάζεται αναζήτηση στο ρεύμα ει-

σαγωγής, όπως γίνεται στον Λεξικό Αναλυτή (2.2.2), κατά την επεξεργασία το υ. Ουσιαστικά, "καταναλώνοντας" ένα λέξημα ο Συντακτικός Αναλυτής μπορεί να αναγνωρίσει την σημασία αυτοτελώς, χωρίς να επηρεάζεται από τα λεξήματα που το περιβάλλουν.

Βάζοντας σε λειτουργία κάποια από τις παραπάνω μεθοδολογίες το τελικό προϊόν του Συντακτικού Αναλυτή είναι ένα Συντακτικό Δέντρο, όπου ο κόμβος που αποτελεί την ρίζα του αλλά και όλοι οι ενδιάμεσοι κόμβοι, αποτελούν μη-τερματικά σύμβολα της Γραμματικής που περιγράφει τη Σύνταξη της Γλώσσας, ενώ όλοι οι κόμβοι-φύλλα αποτελούν τερματικά σύμβολα της Γραμματικής.

Όσον αφορά την συντακτική ορθότητα του Προγράμματος που αναλύει ο Συντακτικός Αναλυτής, δεδομένου ότι οι μεθοδολογίες αφορούν Γραμματικές ανεξάρτητες συμφραζομένων, εύκολα συμπεραίνει κανείς, πως για να είναι δυνατή η παραγωγή του Συντακτικού Δέντρου αναγκαία συνθήκη είναι το Πρόγραμμα να μην περιέχει συντακτικά σφάλματα. Προφανώς, δεν είναι δυνατή η παραγωγή Συντακτικού Δέντρου προγράμματος, που περιέχει συντακτικά σφάλματα.

3.1.3 Γραμματικές Ανεξάρτητες Συμφραζομένων

Σε αντίθεση με τις Κανονικές Εκφράσεις, οι **ανεξάρτητες συμφραζομένων Γραμματικές** μας επιτρέπουν να ορίσουμε μία Γλώσσα ανεξάρτητη συμφραζομένων με μεγαλύτερη ευελιξία, χωρίς σε καμία περίπτωση να χάνεται η σαφήνεια του ορισμού. Επιπλέον, όπως και οι Κανονικές Εκφράσεις, μας δίνουν ορισμούς που είναι αρκετά απλοί, έτσι ώστε να είμαστε σε θέση να δη-

μιουργήσουμε έναν αλγόριθμο αναγνώρισης συμβολοσειρών που ανήκουν στην περιγραφόμενη γλώσσα.

Μία Γραμματική ανεξάρτητη συμφραζομένων G προσδιορίζεται ως μία τετράδα $\langle V, \Sigma, R, S\rangle$ όπου

- **V** το πεπερασμένο σύνολο μη τερματικών συμβόων ή μεταβλητών.
- **Σ** το πεπερασμένο σύνολο των τερματικών συμβόλων (ή αλλιώς το Αλφάβητο της γλώσσας)
- **R** το σύνολο των κανόνων παραγωγής της γραμματικής. Η σύνταξή τους περιλαμβάνει τα σύμβολα του συνόλου $(V \cup \Sigma)$ και είναι της μορφής $K \rightarrow BaTa$ όπου το $K \in V$ και τα $B, a, T \in (V \cup \Sigma)$.
- **S** ένα μη τερματικό σύμβολο $S \in V$ το οποίο είναι το σύμβολο από το οποίο ξεκινά η παραγωγή της Γραμματικής. Ονομάζεται σύμβολο εκκίνησης (start symbol).

Παράδειγμα:

Έστω ότι έχουμε την Γραμματική ανεξάρτητη συμφραζομένων G. Αναλυτικότερα μπορούμε να γράψουμε $G = \langle V, \Sigma, R, S\rangle$ όπου:

- $V = \{A, T, X\}$
- $\Sigma = \{\alpha, \beta, \gamma\}$
- $R = \{$

  $A \rightarrow \alpha T \beta X$

  $T \rightarrow \beta\gamma$

$X \rightarrow \alpha T \beta$

}

- $S \equiv A$

Η παραπάνω Γραμματική μέσω της διαδικασίας της **παραγωγής** μας δίνει την συμβολοσειρά "αβγβαβγβ". Υπάρχουν δύο τρόποι να παραχθεί η συμβολοσειρά αυτή από τους κανόνες της Γραμματικής και έχουν να κάνουν με το αν η Γραμματική είναι **δεξιότατη (right-most)** ή **αριστερότατη (left-most)**.

Αν πρόκειται για μια δεξιότατη Γραμματική τότε η διαδικασία που ακολουθείται είναι η εξής:

$A \rightarrow \alpha T \beta X \rightarrow \alpha T \beta \alpha T \beta \rightarrow \alpha \beta \gamma \beta \alpha T \beta \rightarrow \alpha \beta \gamma \beta \alpha \beta \gamma \beta$

Όπως φαίνεται παραπάνω ονομάζεται "δεξιότατη" γιατί σε κάθε βήμα αντικαθιστούμε το δεξιότερο μη τερματικό σύμβολο με την συμβολοσειρά που προβλέπει ο κανόνας/οι κανόνες.

Αν, όμως, πρόκειται για μια αριστερότατη Γραμματική τότε η διαδικασία που ακολουθείται είναι η εξής:

$A \rightarrow \alpha T \beta X \rightarrow \alpha \beta \gamma \beta X \rightarrow \alpha \beta \gamma \beta \alpha T \beta \rightarrow \alpha \beta \gamma \beta \alpha \beta \gamma \beta$

Εδώ, σε κάθε βήμα αντικαθιστούμε το αριστερότερο μη τερματικό σύμβολο με την συμβολοσειρά που προβλέπει ο κανόνας/οι κανόνες.

Η τελική συμβολοσειρά που παράγεται από την δεδομένη Γραμματική G είναι η ίδια, είτε χρησιμοποιήσουμε την δεξιότατη είτε την αριστερότατη παραγωγή. Το γεγονός αυτό οφείλεται στο ότι ο τρόπος παραγωγής δεν συμμετέχει στον καθορισμό της Γλώσσας που περιγράφει η Γραμματική.

3.1.4 Συντακτικά Δέντρα

Τα Συντακτικά Δέντρα αποτελούνται από κόμβους, οι οποίοι αντιστοιχούν σε σύμβολα της Γραμματικής που περιγράφει τη Γλώσσα. Η ρίζα του δέντρου αντιστοιχεί πάντα στο σύμβολο εκκίνησης της Γραμματικής, τα φύλλα του δέντρου αντιστοιχούν στα τερματικά σύμβολα της Γραμματικής, ενώ όλοι οι υπόλοιποι ενδιάμεσοι κόμβοι αντιστοιχούν στα μη τερματικά σύμβολα της Γραμματικής. Ταυτόχρονα, η σειρά με την οποία είναι διατεταγμένα τα φύλλα του δέντρου αντιστοιχεί στην σειρά με την οποία είναι διατεταγμένα τα τερματικά σύμβολα στην συμβολοσειρά εισαγωγής. Επιπλέον, για κάθε γονικό κόμβο οι κόμβοι - παιδιά του είναι διατεταγμένοι όπως ακριβώς και στο δεξιό τμήμα του αντίστοιχου κανόνα παραγωγής της Γραμματικής, όπου φυσικά ο γονικός κόμβος είναι το μη τερματικό σύμβολο που βρίσκεται στο αριστερό τμήμα του κανόνα παραγωγής.

Αν θεωρήσουμε τη Γραμματική $G = \langle V, \Sigma, R, S \rangle$ όπου:

- $V = \{A, \Pi, D\}$
- $\Sigma = \{1, 2, (,), +\}$
- $R = \{$

  $A \rightarrow \Pi$

  $\Pi \rightarrow D$

  $\Pi \rightarrow (\Pi + \Pi)$

  $D \rightarrow 1$

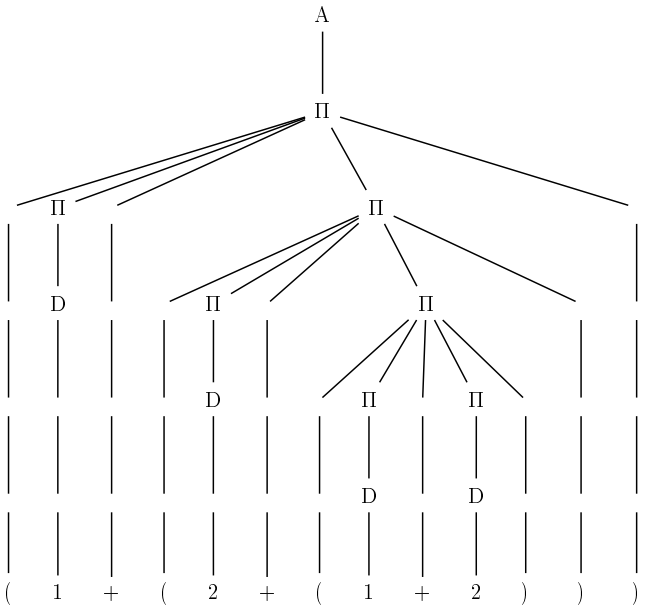
$D \rightarrow 2$

}

- $S \equiv A$

τότε για την συμβολοσειρά "$(1 + (2 + (1 + 2)))$" παράγεται το Συντακτικό Δέντρο του διαγράμματος 3.1.

Διάγραμμα 3.1: Συντακτικό Δέντρο για τη συμβολοσειρά "(1+(2+(1+2)))".



Διάσχιση Δέντρων

Ο τρόπος με τον οποίο διασχίζουμε (traverse) ένα Δέντρο έχει μεγάλη σημασία αφού καθένας από αυτούς έχει ιδιότητες που μας είναι χρήσιμες στην δημιουργία και επεξεργασία των Δέντρων. Για τις ανάγκες της Συντακτικής Ανάλυσης χρησιμοποιούνται κυρίως δύο μέθοδοι, η Προδιατεταγμένη Διάσχιση

(pre-order traversal) και η Μεταδιατεταγμένη Διάσχιση (post-order traversal).

Και στις δύο περιπτώσεις η Διάσχιση ενός Δέντρου ξεκινά από την ρίζα του και συνεχίζει προς τον αριστερότερο κόμβο-παιδί. [1] Η ειδοποιός διαφορά τους είναι πως στην Προδιατεταγμένη Διάσχιση έχουμε πρώτα την επίσκεψη[2] της ρίζας και στην συνέχεια την διάσχιση των υποδέντρων του (πάντα από τα αριστερά προς τα δεξιά), ενώ στην Μεταδιατεταγμένη Διάσχιση πρώτα έχουμε την διάσχιση των υποδέντρων και μετά την επίσκεψη της ρίζας του.[3]

Για να γίνει κατανοητή η διαφορά των δύο μεθόδων διάσχισης δέντρων παρατίθενται δύο σχεδιαγράμματα όπου το ένα (3.2) έχει αριθμημένους τους κόμβους του με Προδιατεταγμένη Διάσχιση ενώ το άλλο (3.3) με Μεταδιατεταγμένη Διάσχιση. Όπως φαίνεται και στα δύο σχεδιαγράμματα για την Προδιατεταγμένη Διάσχιση ακολουθείται η σειρά **κόμβος-ρίζα — αριστερός κλάδος — δεξιός κλάδος** ενώ για την Μεταδιατεταγμένη Διάσχιση ακολουθείται η σειρά **αριστερός κλάδος — δεξιός κλάδος — κόμβος-ρίζα**.

3.1.5 Κατιούσα Ανάλυση

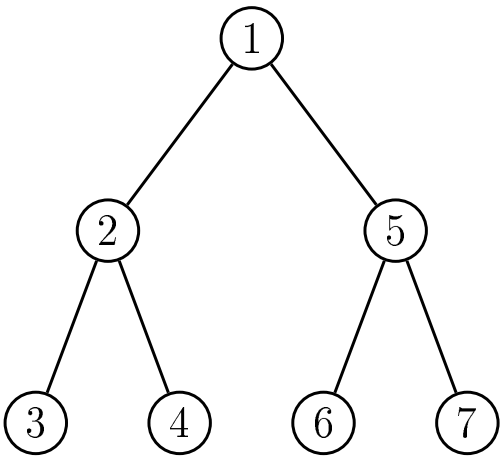
Ένας Συντακτικός Αναλυτής που εφαρμόζει την Κατιούσα Ανάλυση ξεκινά δημιουργώντας τον κόμβο-ρίζα του Συντακτικού Δέντρου και τοποθετεί σε αυτόν το μη-τερματικό σύμβολο εκκίνησης της Γραμματικής της Αρχικής

[1] Κάθε κόμβος-παιδί της ρίζας μπορεί να θεωρηθεί ρίζα του Δέντρου που αποτελείται από τους κόμβους-παιδιά που βρίσκονται κάτω από αυτόν.

[2] Ο όρος "Επίσκεψη" ενός κόμβου σημαίνει την εκτέλεση κάποιας ενέργειας στον κόμβο αυτό.

[3] Ένας κόμβος-φύλλο είναι ειδική περίπτωση Δέντρου, του οποίου αποτελεί την ρίζα, χωρίς κόμβους-παιδιά.

Διάγραμμα 3.2: Κόμβοι δέντρου αριθμημένοι βάση Προδιατεταγμένης Διάσχισης.

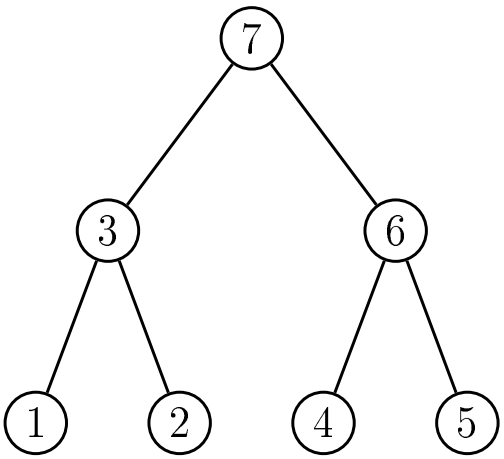


Γλώσσας. Στην συνέχεια αναπτύσσει αναδρομικά τους κόμβους-παιδιά της ρίζας τοποθετώντας σε αυτά τα ανάλογα τερματικά ή μη-τερματικά σύμβολα της Γραμματικής ακολουθώντας την Προδιατεταγμένη Διάσχιση και εφαρμόζοντας τους Κανόνες τις Γραμματικής.

Για να είναι σε θέση να επιλέξει τους Κανόνες της Γραμματικής που θα χρησιμοποιήσει, λαμβάνει υπ' όψιν και το ρεύμα εισαγωγής. Ο αριθμός των λεξημάτων που απαιτείται να ελεγχθούν για να γίνει η επιλογή του επόμενου συμβόλου που θα τοποθετηθεί στο Συντακτικό Δέντρο, εξαρτάται από τα χαρακτηριστικά των κανόνων της Γραμματικής και επηρεάζει καταλυτικά τις δυνατότητες κατασκευής ενός αποδοτικού αλγορίθμου. Για το λόγο αυτό οι Γραμματικές κατηγοριοποιούνται και ως προς αυτόν τον αριθμό.

Επιπλέον, από τη στιγμή που το ρεύμα εισαγωγής καταναλώνεται από τα Αριστερά προς τα Δεξιά και η ακολουθία συμβόλων της Γραμματικής αναπτύσσεται από το αριστερότερο μη-τερματικό σύμβολο κάθε φορά η οικογένεια

Διάγραμμα 3.3: Κόμβοι δέντρου αριθμημένοι βάση Μεταδιατεταγμένης Διάσχισης.



Γραμματικών ονομάζεται LL.[4]

Στη συνέχεια αναλύονται οι όροι **FOLLOW** και **FIRST** οι οποίοι θα χρησιμοποιηθούν στην περιγραφή της οικογένειας των LL Γραμματικών.

FOLLOW

Ο όρος FOLLOW αναφέρεται σε ένα σύνολο για τον ορισμό του οποίου θα χρησιμοποιηθεί η Γραμματική $G_f = \langle V, \Sigma, R, S \rangle$. Τυπικά, η έννοια FOLLOW ορίζεται ως εξής:

$\forall A \in \Sigma$ **ορίζεται** $FOLLOW(A) \subseteq V \cup \{EOF\}$

όπου EOF συμβολίζει το τέλος του ρεύματος εισαγωγής. Φυσικά από το σύνολο V δεν επιλέγονται τυχαία τα σύμβολα που συμπεριλαμβάνονται στο σύνολο $FOLLOW(A)$.

Για να προστεθεί ένα σύμβολο που ανήκει στο V και στο σύνολο $FOLLOW(A)$

[4] Το πρώτο L αντιπροσωπεύει το γεγονός ότι η κατανάλωση του ρεύματος εισαγωγής γίνεται από τα αριστερά προς τα δεξιά (Left-to-Right) ενώ το δεύτερο ότι στην ακολουθία συμβόλων της Γραμματικής αναπτύσσεται πάντα το αριστερότερο μη-τερματικό σύμβολο (leftmost derivation).

θα πρέπει:

1. να υπάρχει ένας τουλάχιστον κανόνας $K \in R$ της μορφής $X \rightarrow^* \beta A \gamma \delta$,[5] όπου $X \in V$, τα $\beta, \delta \in (V \cup \Sigma)^*$ (οποιοδήποτε τερματικό ή μη-τερματικό σύμβολο) και $\gamma \in V$ (ένα μη-τερματικό σύμβολο).

2. να υπάρχει ένας τουλάχιστον κανόνας $K \in R$ της μορφής $X \rightarrow^* \beta \delta A$, όπου $X \in V$, τα $\beta, \delta \in (V \cup \Sigma)^*$ (συμβολοσειρές που περιέχουν οποιοδήποτε τερματικό ή μη-τερματικό σύμβολο).

Για παράδειγμα έστω η Γραμματική G που παρουσιάζεται στην ενότητα (3.1.3), τότε $FOLLOW(X) = \{EOF\}$.

FIRST

Όμοια, ο όρος FIRST αναφέρεται και αυτός σε ένα σύνολο. Για τον ορισμό του θα χρησιμοποιήσουμε την Γραμματική $G_f = \langle V, \Sigma, R, S \rangle$. Τυπικά, η έννοια FIRST ορίζεται ως εξής:

$\forall A \in (V \cup \Sigma)^*$ **ορίζεται** $FIRST(A) \subseteq V \cup \{\varepsilon\}$

όπου ε η κενή συμβολοσειρά.

Για να προστεθεί ένα σύμβολο που ανήκει στο $(V \cup \Sigma)$ και στο σύνολο $FIRST(A)$ θα πρέπει:

1. να υπάρχει ένας τουλάχιστον κανόνας $K \in R$ της μορφής $A \rightarrow^* \beta \gamma \delta$, όπου τα $\gamma, \delta \in (V \cup \Sigma)^*$ (συμβολοσειρές που περιέχουν οποιοδήποτε τερματικό ή μη-τερματικό σύμβολο) και $\beta \in V$ (ένα μη-τερματικό σύμβολο).

[5] το άστρο '*' στον κανόνα παραγωγής σημαίνει ότι μεταξύ του αριστερού και του δεξιού μέρους του κανόνα μπορούν να παρεμβάλλονται οσοιδήποτε, πεπερασμένου πλήθους, κανόνες παραγωγής

2. να υπάρχει ένας τουλάχιστον κανόνας $K \in R$ της μορφής $A \rightarrow^* \varepsilon$.

Για παράδειγμα έστω η Γραμματική G που παρουσιάζεται στην ενότητα (3.1.3), τότε $FIRST(X) = \{\alpha\}$.

Γραμματικές και Συντακτικοί Αναλυτές LL(1)

Οι Συντακτικοί Αναλυτές LL(k) παίρνουν τον προσδιορισμό τους από τις Γραμματικές που είναι σε θέση να αναγνωρίσουν, όπου k είναι ένας φυσικός αριθμός που δίνει το πλήθος των λεξημάτων που θα πρέπει να ελεγχθούν για να αποφασιστεί το περιεχόμενο ενός κόμβου του Συντακτικού Δέντρου. Από την σκοπιά του Συντακτικού Αναλυτή αυτό σημαίνει πόσες θέσεις (λεξήματα) θα πρέπει να έχει την δυνατότητα να οπισθοχωρήσει στο ρεύμα εισαγωγής ο Συντακτικός Αναλυτής για να μπορεί να αναγνωρίσει την αντίστοιχη Γραμματική LL(k).

Στην πράξη, για k μεγαλύτερα του 1 οι διαθέσιμοι αλγόριθμοι δεν είναι σε θέση να εκτελεστούν χωρίς να χρησιμοποιούν υπερβολικά πολλούς πόρους, είτε πρόκειται για μνήμη είτε για χρόνο (υπολογιστική ισχύ), και να έχουν τουλάχιστον γραμμική σχέση απαιτήσεων σε πόρους και μήκος προγράμματος εισαγωγής. Για το λόγο αυτό περιοριζόμαστε στην κατασκευή Συντακτικών Αναλυτών LL(1).

Βέβαια, παρόλο που η πλειονότητα των σχεδιαστών Γλωσσών Προγραμματισμού έχουν κατά νου αυτόν τον περιορισμό και προσπαθούν να πλησιάσουν την Γραμματική της Γλώσσας στην κατηγορία LL(1), τις περισσότερες φορές, ο ορισμός της Αρχικής Γλώσσας, όπως αυτός δίνεται στο εγχειρίδιό της, δεν περιγράφεται μία Γραμματική LL(1). Η λύση σε αυτό το πρόβλημα δίνεται με

αλγόριθμους μετασχηματισμού μίας Γραμματικής LL(K) σε LL(1). Εδώ θα πρέπει να τονιστεί πως αυτοί οι αλγόριθμοι δεν είναι δυνατό να εφαρμοστούν σε όλες τις Γραμματικές LL(k) αλλά μόνο σε ένα υποσύνολό τους.

Πρώτα, όμως, θα παρουσιάσουμε τις προϋποθέσεις που θα πρέπει να πληρούνται για να μπορεί να χαρακτηριστεί μία Γραμματική LL(1).

- Αν υπάρχουν εναλλακτικοί κανόνες στην Γραμματική[6] τότε τα σύνολα FIRST για τους κανόνες αυτούς θα πρέπει να είναι ξένα μεταξύ τους.
  Τυπικά γράφουμε αν

  $$K \rightarrow r, K \rightarrow s$$

  τότε πρέπει

  $$FIRST(r) \cap FIRST(s) = \emptyset$$

- Αν υπάρχει ένας κανόνας που παράγει την κενή συμβολοσειρά θα πρέπει τα σύνολα FIRST και FOLLOW για το αριστερό μέρος του κανόνα αυτού να είναι ξένα μεταξύ τους.
  Τυπικά γράφουμε αν

  $$A \rightarrow \varepsilon$$

  τότε πρέπει

  $$FIRST(A) \cap FOLLOW(A) = \emptyset$$

[6] Δύο κανόνες ονομάζονται εναλλακτικοί όταν έχουν ίδιο αριστερό μέρος.

Επιγραμματικά, οι μετασχηματισμοί που μπορούμε να εφαρμόσουμε για να φέρουμε μία Γραμματική σε μορφή LL(1) είναι

- η Αντικατάσταση, όπου αντικαθιστούμε ένα μη τερματικό σύμβολο για το οποίο έχουμε εναλλακτικές παραγωγές στα δεξιά μέρη των κανόνων που εμφανίζεται με όλες τις δυνατές παραγωγές του.
  Αν έχουμε τους κανόνες:

  $$A \rightarrow \alpha_1|\alpha_2|...|\alpha_{\nu-1}|\alpha_\nu$$

  $$B \rightarrow \beta_1 A \beta_2$$

  τότε ο B μπορεί να μετασχηματιστεί ως εξής:

  $$B \rightarrow \beta_1\alpha_1\beta_2|\beta_1\alpha_2\beta_2|...|\beta_1\alpha_{\nu-1}\beta_2|\beta_1\alpha_\nu\beta_2$$

- η Αριστερή Παραγοντοποίηση, όπου για κανόνες της μορφής:

  $$A \rightarrow \alpha\beta_1|\alpha\beta_2|...|\alpha\beta_{\nu-1}|\alpha\beta_\nu$$

  το A παραγοντοποιείται ως εξής:

  $$A \rightarrow \alpha B$$

  $$B \rightarrow \beta_1|\beta_2|...|\beta_{\nu-1}|\beta_\nu$$

- η Απαλοιφή Αριστερής Αναδρομής, όπου πραγματευόμαστε την απαλοιφή κανόνων της μορφής $A \rightarrow A\beta$ (άμεσα αριστερή αναδρομή) και $A \rightarrow \beta X$,

$X \rightarrow oA\tau$ (έμμεσα αριστερή αναδρομή). Αν υπάρχει έμμεσα αριστερή αναδρομή, τότε αυτή μετατρέπεται σε άμεσα αριστερή αναδρομή. Οι άμεσα αριστερές αναδρομές αντιμετωπίζονται μετατρέποντάς τες σε δεξιές αναδρομές.

3.1.6 Ανιούσα Ανάλυση

Ένας Συντακτικός Αναλυτής που εφαρμόζει την Ανιούσα Ανάλυση ξεκινά δημιουργώντας τους κόμβους - φύλλα του Συντακτικού Δέντρου, έναν για κάθε λέξημα που "καταναλώνεται", κινούμενος από τα αριστερά προς τα δεξιά στο ρεύμα εισαγωγής. Στην συνέχεια, προσπελαύνει τους κόμβους - φύλλα του Δέντρου, πάλι από τα αριστερά προς τα δεξιά, και εντοπίζει ακολουθίες κόμβων, που αποτελούν ακριβή ακολουθία των παιδιών ενός κόμβου υψηλότερου επιπέδου, πάντα σύμφωνα με τους κανόνες παραγωγής της Γραμματικής. Όταν, και μόνο όταν, βρεθεί το σύνολο των κόμβων - παιδιών ενός γονικού κόμβου τότε αυτός κατασκευάζεται και σημειώνεται με το αντίστοιχο μη - τερματικό σύμβολο.

Είναι προφανές, πως στο πρώτο πέρασμα όλες οι ακολουθίες κόμβων - παιδιών αποτελούνται από φύλλα του δέντρου και περιέχουν τερματικά σύμβολα, όμως από το επόμενο πέρασμα συμπεριλαμβάνουν και νεοδημιουργημένους κόμβους οι οποίοι περιέχουν μη - τερματικά σύμβολα. Η διαδικασία αυτή συνεχίζεται έως ότου δημιουργηθεί ο κόμβος - ρίζα του Συντακτικού δέντρου. Όπως εύκολα θα παρατηρήσει κανείς η διαδικασία κατασκευής του Δέντρου ακολουθεί τα βήματα της Μεταδιατεταγμένης Διάσχισης. Με απλά λόγια, η ανιούσα διαδικασία κατασκευής ενός Συντακτικού Δέντρου αποτελείται από

την αναζήτηση του αριστερότερου κόμβου, του οποίου όλα τα παιδιά έχουν ήδη δημιουργηθεί και στην συνέχεια την δημιουργία του.

Από την ομάδα Συντακτικών Αναλυτών που εφαρμόζουν την ανιούσα μεθοδολογία συχνότερα υλοποιούνται Αναλυτές που ανήκουν στην υποομάδα Συντακτικών Αναλυτών Ολίσθησης-Σύμπτυξης (shift-reduce parsers). Βασικό του χαρακτηριστικό είναι πως βασίζονται στην χρήση μίας στοίβας, ενώ κατά την λειτουργία τους εκτελούν δύο ειδών πράξεις:

- την **ολίσθηση** (shift) όπου καταναλώνουν ένα λέξημα από το ρεύμα εισαγωγής και τοποθετούν το ανάλογο τερματικό σύμβολο στην κορυφή της στοίβας.

- την **σύμπτυξη** (reduce) που εφαρμόζεται όταν η κορυφή της στοίβας ταυτίζεται με το δεξιό τμήμα ενός κανόνα παραγωγής, την οποία και αντικαθιστά με το μη-τερματικό σύμβολο που του αριστερού τμήματος του ίδιου κανόνα.

Θεωρούμε ότι η εκτέλεση του Συντακτικού Αναλυτή έχει ολοκληρωθεί χωρίς σφάλματα, έχει δηλαδή αναγνωριστεί η συμβολοσειρά εισαγωγής, όταν έχουν καταναλωθεί στο σύνολό τους τα λεξήματα του ρεύματος εισαγωγής και στην στοίβα υπάρχει μόνο ένα σύμβολο, το μη-τερματικό σύμβολο εκκίνησης της Γραμματικής. Σε αντίθετη περίπτωση εντοπίστηκαν ένα ή περισσότερα σφάλματα στην συμβολοσειρά εισαγωγής.

Φυσικά, και σε αυτού του είδους τους Συντακτικούς Αναλυτές, είναι σημαντικό να μην χρειάζεται να ανακτήσει, κανείς, επόμενα σύμβολα από το ρεύμα εισαγωγής για να ερμηνεύσει το τρέχων σύμβολο, αφού σε αντίθετη περίπτωση αδυνατούμε να κατασκευάσουμε αποδοτικούς αλγορίθμους. Ένας

Συντακτικός Αναλυτής που πληρεί αυτό το κριτήριο παίρνει τον προσδιορισμό "ντετερμινιστικός" (deterministic).

Γραμματικές και Συντακτικοί Αναλυτές LR

Ειδική κατηγορία των Συντακτικών Αναλυτών Ολίσθησης-Σύμπτυξης είναι οι Συντακτικοί Αναλυτές LR(k),[7] όπου k είναι ένας φυσικός αριθμός που δίνει το πλήθος των λεξημάτων που θα πρέπει να ελεγχθούν για να αποφασιστεί το περιεχόμενο ενός κόμβου του Συντακτικού Δέντρου.

Όπως ήδη αναφέρθηκε δεν είναι πρακτικό να κατασκευάσουμε Συντακτικούς Αναλυτές που απαιτούν οπισθοδρόμηση, επομένως στην πλειονότητα των περιπτώσεων το k είναι 1, άρα δουλεύουμε με Γραμματικές και Συντακτικούς Αναλυτές LR(1) ή, συνηθέστερα, υποσύνολά τους.

Για να υλοποιηθεί ένας LR(1) Συντακτικός Αναλυτής χρησιμοποιείται ένα αυτόματο στοίβας, του οποίου τα δομικά στοιχεία είναι τα εξής:

- Το ρεύμα εισαγωγής από το οποίο ανακτά ένα-ένα τα λεξήματα.
- Ένα σύνολο δυνατών καταστάσεων στις οποίες συμπεριλαμβάνεται και μία ειδική κατάσταση s_0 η οποία αντιστοιχεί στην κατάσταση εκκίνησης.
- Η στοίβα στην οποία τοποθετείται η αρχική κατάσταση και στην συνέχεια ζευγάρια γραμματικών συμβόλων (τερματικού ή μη τερματικών συμβόλων) και καταστάσεων.
- Ένας πίνακας ACTION του οποίου οι γραμμές αντιστοιχούν σε καταστάσεις και οι στήλες στο σύνολο των τερματικών συμβόλων της Γραμ-

[7] Το L αντιπροσωπεύει το γεγονός ότι η κατανάλωση του ρεύματος εισόδου γίνεται από τα αριστερά προς τα δεξιά (Left-to-Right) ενώ το R ότι στην ακολουθία συμβόλων της Γραμματικής αναπτύσσεται πάντα το δεξιότερο μη-τερματικό σύμβολο (rightmost derivation).

ματικής, συμπεριλαμβανομένου του συμβόλου "τέλος συμβολοσειράς εισαγωγής" (EOF), ενώ τα κελιά του περιέχουν μία από τις τρεις δυνατές πράξεις:

  - ολίσθηση
  - σύμπτυξη με κάποιον από τους κανόνες παραγωγής της Γραμματικής
  - αποδοχή της συμβολοσειράς εισαγωγής

- Ένας πίνακας NEXT του οποίου οι γραμμές αντιστοιχούν σε καταστάσεις του αυτομάτου και οι στήλες στα σύμβολα της Γραμματικής, τερματικά και μη, ή στο σύμβολο "τέλος συμβολοσειράς εισαγωγής" (EOF), ενώ τα κελιά του, περιέχουν είτε την επόμενη κατάσταση, αν υπάρχει τέτοια, είτε είναι κενά, αν δεν υπάρχει μετάβαση για τον συνδυασμό κατάστασης - συμβόλου.

Η εκτέλεση του αυτομάτου στοίβας οδηγείται από την τρέχουσα κατάσταση, την κατάσταση που βρίσκεται στην κορυφή τη στοίβας, και του επόμενου λεξήματος που "καταναλώνεται" από το ρεύμα εισαγωγής. Ειδικότερα ανακτάται το περιεχόμενο του κελιού του πίνακα ACTION, το οποίο βρίσκεται στην αντίστοιχη γραμμή της κατάστασης στην κορυφή της στοίβας και την στήλη που αντιστοιχεί στο λέξημα που "καταναλώνεται". Αν το περιεχόμενο του του κελιού αυτού υποδεικνύει:

- την πράξη της αποδοχής, τότε η λειτουργία του αυτομάτου σταματά και θεωρούμε ότι η συμβολοσειρά εισαγωγής έχει αναγνωριστεί επιτυχώς.

- την πράξη της ολίσθησης, τότε το σύμβολο που "καταναλώθηκε" από το ρεύμα εισαγωγής τοποθετείται στην κορυφή της στοίβας, ακολουθούμενο από την κατάσταση που περιέχεται στο κελί του πίνακα NEXT, το οποίο προσδιορίζεται από την γραμμή της τρέχουσας κατάστασης και την στήλη του λεξήματος που μόλις προστέθηκε στην στοίβα.

- την πράξη της σύμπτυξης με έναν κανόνα της Γραμματικής, όπου τότε αφαιρούνται από την κορυφή της στοίβας ισάριθμα ζευγάρια καταστάσεων - συμβόλων, με το πλήθος των συμβόλων στο δεξιό μέρος του κανόνα. Στην συνέχεια ανακτάται από τον πίνακα NEXT η κατάσταση που καθορίζεται από την κατάσταση που βρίσκεται τώρα στην κορυφή της στοίβας και την στήλη του συμβόλου στο αριστερό τμήμα του κανόνα. Τέλος τοποθετούνται στην στοίβα πρώτα το μη-τερματικό σύμβολο του αριστερού μέρους του κανόνα, ακολουθούμενο από την κατάσταση που επιστράφηκε από τον πίνακα NEXT.

Υπάρχουν δύο περιπτώσεις τερματισμού του αυτομάτου. Μία είναι να ανακτηθεί η πράξη της αποδοχής, οπότε και αναγνωρίζεται επιτυχώς η συμβολοσειρά εισαγωγής. Η δεύτερη είναι να προσπελάσει στους πίνακες NEXT ή ACTION, ένα κενό κελί με αποτέλεσμα να τερματίσει το αυτόματο με σφάλμα και να μην αναγνωριστεί η συμβολοσειρά εισαγωγής.

Είναι αξιοσημείωτο ότι ο ορισμός των πινάκων ελέγχου για NEXT και ACTION, για Γραμματικές LR(1) δεν είναι χωρίς αμφισημίες (ντετερμινιστικός - deterministic), είναι όμως για τις Γραμματικές LR(0).[8] Αλλά οι Γραμματικές

[8] Όπως εύκολα συμπεραίνει κανείς, από τη στιγμή που η Γραμματικές αυτές έχουν $k = 0$ αυτό σημαίνει πως δεν χρειάζεται να γνωρίζουν ούτε το επόμενο λέξημα στο ρεύμα εισαγωγής για εκτελέσουν την αναγνώριση. Αρκούνται σε αυτά που έχουν "καταναλώσει" ήδη.

αυτές είναι πρακτικά άχρηστες λόγω των περιορισμών που επιβάλουν. Στην προσπάθεια να ξεπεράσουμε το εμπόδιο αυτό, κατασκευάσαμε κάποιες ομάδες Γραμματικών οι οποίες κυμαίνονται μεταξύ των LR(0) και LR(1). Έχει επικρατήσει να χρησιμοποιείται η οικογένεια Γραμματικών LALR(1) η οποία βρίσκεται μεταξύ των LR(0) και LR(1) ο αλγόριθμος της LALR(1) βρίσκεται από πλευράς δυνατοτήτων, αλλά και απαιτήσεων σε πόρους, εγγύτερα σε αυτόν της LR(1).

3.2 Αυτόματη Κατασκευή Συντακτικών Αναλυτών

Όπως συμβαίνει και για τον Λεξικό Αναλυτή, έτσι και ο Συντακτικός Αναλυτής μπορεί να κατασκευαστεί αυτόματα με χρήση μίας γεννήτριας Συντακτικών αναλυτών. Τέτοιες είναι ο yacc[9] και ο bison[10]. Ο bison μπορεί είτε να συνεργαστεί με έναν Λεκτικό Αναλυτή που έχει κατασκευαστεί χειρονακτηκά είτε να συνεργαστεί με τον flex, το ίδιο ισχύει και για το ζεύγος yacc - lex.

Όμως τα εργαλεία αυτά δεν έχουν την δυνατότητα να δεχτούν την Γραμματική της Γλώσσας, διατυπωμένη στην μαθηματική μορφή που παρουσιάστηκε παραπάνω. Για το λόγο αυτό χρησιμοποιείται μία μορφή ανάλογη αλλά φιλικότερη ως προς την εισαγωγή σε Η/Υ. Πρόκειται για την BNF (Backus Naur Form)[11] την οποία θα παρουσιάσουμε παρακάτω.

[9] Κατασκευασμένη από την AT&T ως συνοδευτικό του Λειτουργικού Συστήματος UNIX

[10] Πρόκειται για λογισμικό GNU, που διανέμεται κάτω από την άδεια GPL, σε μεγάλο βαθμό συμβατό με τον yacc.

[11] Αρχικά προερχόταν από τις λέξεις Backus Normal Form, όμως μετά από πρόταση του Donald Knuth το Normal αντικαταστάθηκε από το όνομα του Naur για την προσφορά του στον τομέα αυτό.(Βλ. ACM, Volume 1, Issue 12, Dec. 1964)

3.2.1 Backus Naur Form

Η μορφή BNF βασίζεται κυρίως στην περιγραφή των κανόνων παραγωγής της Γραμματικής, ενώ πληροφορίες, όπως το σύνολο των τερματικών συμβόλων, το σύνολο των μη-τερματικών συμβόλων και το σύμβολο εκκίνησης δηλώνονται έμμεσα από την αποτύπωση των κανόνων αυτών.

Τα μη-τερματικά σύμβολα της Γραμμικής είναι αυτά που περικλείονται από τους μεταχαρακτήρες '<' και '>'. Τα τερματικά σύμβολα είναι όσα δεν περικλείονται στους προαναφερθέντες χαρακτήρες. Εξαίρεση αποτελούν τα μετασύμβολα, τα οποία είναι το σύμβολο "ορίζεται ως" ("::=") και η διάζευξη '|'.

Ακόμη, το σύμβολο εκκίνησης ορίζεται ως το **αριστερό τμήμα του πρώτου κανόνα**, ενώ οι εναλλακτικοί κανόνες (ίδιο το αριστερό τμήμα) ισοδυναμούν με την διάζευξη. Τέλος, υπάρχει δυνατότητα να χρησιμοποιηθεί αναδρομή, τοποθετώντας το μη-τερματικό σύμβολο του αριστερού τμήματος του κανόνα και στο αριστερό τμήμα του κανόνα.

Παράθεση 3.1: "Περιγραφή Γραμματικής σε μορφή BNF"

```
<έκφραση>          ::= '(' <έκφραση> ')' | <πρόσθεση>
                     | <πολλαπλασιασμός>
<πρόσθεση>         ::= <αριθμός> '+' <αριθμός>
<πολλαπλασιασμός> ::= <αριθμός> '*' <αριθμός>
<αριθμός>          ::= '0' | '1' | '2' | '3' | '4' | '5'
                     | '6' | '7' | '8' | '9'
```

3.2.2 bison

Ο bison είναι μία γεννήτρια Συντακτικών Αναλυτών που χρησιμοποιεί τον αλγόριθμο LALR(1). Η εισαγωγή του είναι η περιγραφή της Γραμματικής σε απλή παραλλαγή της μορφής BNF και η εξαγωγή ένα πρόγραμμα σε Γλώσσα C που υλοποιεί τον ανάλογο Συντακτικό Αναλυτή.

Ο Συντακτικός Αναλυτής που παράγεται από τον bison είναι το δεσπόζον τμήμα του Μεταγλωττιστή, ενώ μαζί με την περιγραφή των συμβόλων της Γλώσσας, εισάγουμε σε αυτόν την **Σημαντική Ανάλυση**, την κατασκευή του **Πίνακα συμβόλων**, αλλά και την **Παραγωγή Κώδικα**, τα οποία θα αναλυθούν σε επόμενα κεφάλαια.

Η δομή του φακέλου εισαγωγής του bison είναι ανάλογη με αυτή του flex και παρουσιάζεται παρακάτω. Όπως και στον flex τα τρία τμήματα μπορούν να είναι κενά, με μόνο απαραίτητο το πρώτο "%%" σύμβολο.

Παράθεση 3.2: "Δομή τυπικού φακέλου bison"

```
ορισμοί
%%
κανόνες
%%
κώδικας χρήστη
```

Το δεσπόζον τμήμα του bison είναι το τμήμα των ορισμών. Το τμήμα των ορισμών αποτελείται από καμία, μία, έως και πεπερασμένου πλήθους προτάσεις, που έχουν τη μορφή περιγραφέας - κενό - δράση. Οι προτάσεις αυτές μπορεί να εκτείνονται σε μία ή περισσότερες φυσικές γραμμές. Κάθε περιγραφέας είναι μία κανονική έκφραση που περιγράφει ένα σύνολο λεξημάτων.

Κάθε δράση διατυπώνεται σε μορφή κάποιας ανωτέρου επιπέδου γλώσσας προγραμματισμού, συνήθως C (αυτή χρησιμοποιούμε και στο παρόν πόνημα). Η σημαντική κάθε πρότασης του bison, όπως περιγράφηκε πιο πάνω, είναι:

> Όταν ο παραγόμενος αναλυτής στη σειριακή επεξεργασία του πηγαίου προγράμματος, που του δίδεται στην είσοδό του, συναντήσει ένα λέξημα του περιγραφέα της πρότασης, τότε εκτελεί τις ενέργειες που περιγράφονται στο τμήμα της δράσης, για το τρέχων λέξημα. Η διαδικασία εκτελείται σειριακά για κάθε λέξημα που περιγράφεται από τον περιγραφέα κάθε πρότασης του τμήματος ορισμού του bison.

3.2.3 Χειρισμός Σφαλμάτων

Είναι φυσικό, όπως και κατά την Λεκτική Ανάλυση, έτσι κι εδώ, να βρεθούν σφάλματα, που έχουν εισαχθεί στα προγράμματα κατά την συγγραφή τους. Στην φάση της Συντακτικής Ανάλυσης εντοπίζουμε σφάλματα Συντακτικής φύσης, όπως είναι παραδείγματος χάριν η παράλειψη εισαγωγής του ελληνικού ερωτηματικού ';' στο τέλος μίας πρότασης σε ένα πρόγραμμα γραμμένο στη Γλώσσα PCL.

Ο ρόλος του Συντακτικού Αναλυτή σε αυτόν τον τομέα είναι αφενός να αναγνωρίσει τα σφάλματα και να μην τα αφήσει να περάσουν σε επόμενα τμήματα του Μεταγλωττιστή, αφετέρου δε να ενημερώσει για την ύπαρξή τους και αν είναι δυνατό για την θέση και την φύση τους, τον προγραμματιστή.

Αναλυτικότερα, το τμήμα του Συντακτικού Αναλυτή που είναι επιφορτισμένο με την διαχείριση σφαλμάτων πρέπει να εμφανίζει αρκετές πληροφορίες

για το σφάλμα, ώστε να μπορεί ο προγραμματιστής να εντοπίσει και να διορθώσει το σφάλμα στο πρόγραμμα, που είναι διατυπωμένο στην Αρχική Γλώσσα. Απαραίτητα, θα πρέπει να αναφερθεί η ακριβής θέση του σφάλματος στο αρχικό πρόγραμμα. Συνήθως αναφέρεται η γραμμή και αρκετές φορές η στήλη στην οποία εντοπίστηκε στο σφάλμα. Επιπλέον, είναι επιθυμητό και ιδιαίτερα σημαντικό να εξακριβωθεί από τον Διαχειριστή Σφαλμάτων, η φύση του σφάλματος. Τις περισσότερες φορές οι Συντακτικοί Αναλυτές αναφέρουν τί περίμεναν να βρουν αντιπαραθέτοντας αυτό που εν τέλει βρήκαν.

Εκτός, όμως, από τον εντοπισμό σφαλμάτων, οι Συντακτικοί Αναλυτές θα πρέπει να έχουν την δυνατότητα να ανανήψουν από αυτά. Γίνεται προσπάθεια, είτε μέσω αντικατάστασης, είτε μέσω εισαγωγής λεξημάτων να βρεθεί ο Συντακτικός Αναλυτής σε μία σταθερή κατάσταση,[12] που θα του επιτρέψει να συνεχίσει την Ανάλυση, με σκοπό, όχι κυρίως την ολοκλήρωση της Μεταγλώττισης, αλλά τον εντοπισμό και την αναφορά όσο περισσότερων σφαλμάτων είναι δυνατό. Σκοπός αυτού είναι κυρίως η φιλικότητα - ευκολία προς τον χειριστή του Μεταγλωττιστή. Όμως, στην προσπάθεια που γίνεται να βρεθεί σε σταθερή κατάσταση, υπάρχει περίπτωση να προκαλέσει την εμφάνιση πολλαπλάσιων σφαλμάτων στην συνέχεια του προγράμματος. Ένας τρόπος να μετριασθεί αυτό το φαινόμενο, είναι να αγνοούνται τα σφάλματα, που εμφανίζονται "κοντά" στο σημείο, όπου εντοπίστηκε το τελευταίο σφάλμα. Στην πράξη η έκρηξη σφαλμάτων δεν αποφεύγεται, ενώ οι έμπειροι προγραμματιστές είναι σε θέση να τα αγνοήσουν και να εντοπίσουν ανάμεσά τους τα πραγματικά σφάλματα. Ένα ακόμα πρόβλημα που προκύπτει, είναι πως η διαδικασία

[12]Θεωρείται ότι ένας Συντακτικός Αναλυτής βρίσκεται σε "σταθερή κατάσταση" όταν τα λεξήματα που έχει στην διάθεσή του του επιτρέπουν να συνεχίσει να κατασκευάζει ένα έγκυρο Συντακτικό Δέντρο.

εισαγωγής ενός λεξήματος, μπορεί να είναι ανεπιτυχής, με τελικό αποτέλεσμα, να χρειαστεί η εισαγωγή ενός ακόμα κ.ο.κ. με αποτέλεσμα να εισέλθει ο χειριστής σφαλμάτων, σε έναν ατέρμονα βρόγχο. Για τον λόγο αυτό αλλά και για την αποφυγή αναφοράς υπερβολικά μεγάλου αριθμού σφαλμάτων, τα οποία προέρχονται από μία αντικατάσταση σε ένα μεγάλο πρόγραμμα, πολλές φορές υπάρχει ένας ανώτατος αριθμός σφαλμάτων, μετά από τον οποίο η Μεταγλώττιση σταματά, εμφανίζοντας το ανάλογο μήνυμα.

Βέβαια, για να έχουν νόημα όλα τα παραπάνω θα πρέπει να μην προκαλούν ιδιαίτερη αύξηση του απαιτούμενου χρόνου Μεταγλώττισης. Για το λόγο αυτό αποφεύγονται οι Συντακτικοί Αναλυτές που αντιμετωπίζουν τα σφάλματα προσπαθώντας να τα διορθώσουν, αφού για να γίνει κάτι τέτοιο απαιτούνται υπέρμετροι πόροι, τόσο σε μνήμη όσο και σε υπολογιστική ισχύ.

Στρατηγικές Ανάνηψης από Σφάλματα

Όσα προαναφέρθηκαν έχουν μελετηθεί εκτενώς, με αποτέλεσμα να έχουμε τυποποιήσει, όσο αυτό ήταν δυνατό, της διαθέσιμες στρατηγικές Ανάνηψης από Σφάλματα κατά την Συντακτική Ανάλυση.

Ειδικότερα έχουμε διακρίνει τις εξής στρατηγικές:

- η μέθοδος του πανικού (panic mode) προβλέπει ότι σε περίπτωση σφάλματος, ο Συντακτικός Αναλυτής, αγνοεί τα λεξήματα που ακολουθούν, μέχρις ότου εντοπίσει ένα λέξημα, που θεωρείται ότι μπορεί να χρησιμοποιηθεί ως σημείο συγχρονισμού. Τέτοιες λεκτικές μονάδες είναι, λόγου χάρη, ο τερματισμός μίας έκφρασης που δηλώνεται με το σύμβολο ';' ή ο τερματισμός ενός μπλοκ κώδικα που δηλώνεται με το σύμβολο '}' για τις Γλώσσες της οικογένειας της C.

- η ανάνηψη σε επίπεδο φράσης (phrase level) όπου εδώ εφαρμόζεται η μέθοδος της αντικατάστασης του υπολοίπου της φράσης με μία ακολουθία λεξημάτων που συμφωνούν με την Γραμματική της Γλώσσας. Φυσικά αυτή συνοδεύεται από τα μειονεκτήματα που προαναφέρθηκαν.

- η χρήση κανόνων παραγωγής σφαλμάτων, όπου επαυξάνεται η Γραμματική της Γλώσσας με κανόνες παραγωγής, που αντιστοιχούν σε κοινά σφάλματα. Όταν, οι κανόνες αυτοί ταυτίζονται με κάποιο τμήμα του ρεύματος εισαγωγής, θεωρούμε ότι εντοπίστηκε σφάλμα και εμφανίζεται το ανάλογο διαγνωστικό μήνυμα. Το βασικό μειονέκτημα της μεθόδου αυτής, είναι πως είναι σχεδόν σίγουρη η αναγωγή της Γραμματικής σε μία υψηλότερη κατηγορία, την οποία αδυνατούμε να υλοποιήσουμε με έναν Συντακτικό Αναλυτή. Όπως αναφέραμε προγενέστερα δεν είναι πάντα δυνατή η μετατροπή μίας Γραμματικής υψηλότερης βαθμίδας σε μία αντίστοιχη χαμηλότερης. (3.1.5 και 3.1.6)

- η καθολική διόρθωση προσπαθεί να διορθώσει το πρόγραμμα, δημιουργώντας εναλλακτικά Συντακτικά Δέντρα, βάση διορθώσεων, που χρειάστηκε να γίνουν και στην συνέχεια επιλέγοντας το Δέντρο, για το οποίο απαιτήθηκαν οι λιγότερες τροποποιήσεις. Πέρα από το γεγονός, ότι δεν αποδεικνύεται, πως το συντακτικό δέντρο με τις λιγότερες διορθώσεις, είναι αυτό το οποίο ήθελε να δημιουργήσει ο προγραμματιστής, μάλιστα η εμπειρία δείχνει το αντίθετο, οι απαιτήσεις της συγκεκριμένης στρατηγικής σε πόρους είναι πολύ μεγάλες. Όπως εύκολα μπορεί να συμπεράνει κανείς πρόκειται για μία μέθοδο η οποία μελετάτε καθαρά θεωρητικά και δεν χρησιμοποιείται σε συστήματα παραγωγής.

Κεφάλαιο 4

ΠΙΝΑΚΑΣ ΣΥΜΒΟΛΩΝ

4.1 Ο ρόλος του Πίνακα Συμβόλων (ST, Symbol Table)

Ο ρόλος του Πίνακα Συμβόλων (Symbol Table) σε έναν Μεταγλωττιστή, είναι να διατηρεί και να επιστρέφει πληροφορίες, για κάθε αναγνωριστικό (ID) του προγράμματος που Μεταγλωττίζεται, απαραίτητες για την ολοκλήρωση της Μεταγλώττισης.

4.1.1 Τι είναι αναγνωριστικό

Τα αναγνωριστικά είναι τα ονόματα που εμφανίζονται σε ένα πρόγραμμα. Επομένως, παραδείγματα αναγνωριστικών, είναι τα ονόματα μεταβλητών, τα ονόματα συναρτήσεων/μεθόδων, τα ονόματα κλάσεων, τα ονόματα τύπων κ.ο.κ. Κάθε αναγνωριστικό που ορίζεται στην αρχική γλώσσα πρέπει να συμπεριλαμβάνεται στον Πίνακα Συμβόλων.

4.1.2 Η χρήση

Ο Πίνακας Συμβόλων χρησιμοποιείται από τα περισσότερα τμήματα του Μεταγλωττιστή, που ακολουθούν το τμήμα της Συντακτικής Ανάλυσης. Έχει κυρίαρχο ρόλο στην Σημαντική Ανάλυση, την οποία τροφοδοτεί με απαραίτητες πληροφορίες, όπως οι υπογραφές συναρτήσεων και οι τύποι των μεταβλητών, για τον έλεγχο ορθότητας των εκφράσεων που απαντώνται στο πρόγραμμα. Στην συνέχεια, κατά την παραγωγή κώδικα χρησιμοποιείται, λόγου χάρη, για τον υπολογισμό του μήκους μνήμης κάθε μεταβλητής και τη δημιουργία των τμημάτων εισαγωγής/εξαγωγής δεδομένων των υποπρογραμμάτων βάση της

υπογραφής τους και του τύπου επιστροφής.

4.1.3 Πληροφορίες που συντηρούνται

Το κυριότερο χαρακτηριστικό κάθε αναγνωριστικού είναι το όνομά του. Είναι αυτό που είτε καθορίζει μονοσήμαντα το αναγνωριστικό, είτε συμμετέχει στον μονοσήμαντο ορισμό του.

Επιπλέον, αν πρόκειται για μεταβλητές, ενδιαφέρει η θέση τους στην κύρια μνήμη ή σε κάποιον καταχωρητή, αλλά και ο τύπος τους, ο οποίος με τη σειρά του υποδεικνύει τόσο τις δυνατές ενέργειες, που μπορούν να εκτελεστούν πάνω στο αναγνωριστικό, όσο και το μήκος της μνήμης που πρέπει να δεσμευτεί στο τελικό πρόγραμμα. Ακόμη, αρκετές Γλώσσες διαχωρίζουν τις μεταβλητές σε δύο ομάδες, βάση της θέσης τους στην μνήμη, που έχει αποδοθεί στο πρόγραμμα. Η μία ομάδα είναι οι μεταβλητές σωρού (heap) και η άλλη οι μεταβλητές στοίβας (stack), όπου η πρώτη ξεκινά από την αρχή της διαθέσιμης μνήμης και επεκτείνεται προς το τέλος, ενώ η δεύτερη αντιστρόφως.

Στην περίπτωση των υπογραφών συναρτήσεων/μεθόδων, διατηρούνται το πλήθος και ο τύπος των παραμέτρων, αν αυτές περνιούνται κατά τιμή ή κατά αναφορά καθώς και ο τύπος της επιστρεφόμενης τιμής, αν υπάρχει.

Δύο πολύ βασικά χαρακτηριστικά των αναγνωριστικών, είναι η εμβέλεια και η θέα. Η εμβέλεια έχει να κάνει με τη χρονική διάρκεια της συντήρησης της μνήμης, που έχει αποδοθεί σε ένα αναγνωριστικό, ενώ η θέα αφορά την Σημαντική Ανάλυση και ειδικότερα τα σημεία του προγράμματος στα οποία το αναγνωριστικό μπορεί να χρησιμοποιηθεί. Ένα παράδειγμα εμβέλειας που απαντάται σε πολλές σύγχρονες Γλώσσες Προγραμματισμού, είναι η διατή-

ρηση μίας τοπικής μεταβλητής από το σημείο δήλωσής της, μέχρι το τέλος του μπλοκ κώδικα στο οποίο δηλώθηκε. Θέα είναι η σκίαση μίας μεταβλητής εξωτερικού μπλοκ, από μία μεταβλητή, του εσωτερικού μπλοκ με το ίδιο όνομα, όχι απαραίτητα του ίδιου τύπου.

4.2 Υλοποίηση ενός Πίνακα Συμβόλων

Ο Πίνακας Συμβόλων για να εκτελεί το ρόλο του, θα πρέπει να υποστηρίζει μία σειρά από λειτουργίες, που μπορούν να εκτελεστούν επάνω του. Οι λειτουργίες αυτές είναι:

- **Εισαγωγή**, αναγνωριστικά και συνοδευτικές τους πληροφορείες εισάγονται στον ST
- **Αναζήτηση**, αναζητάται ID και εφόσον εντοπιστέι, επιστρέφονται οι συνδεδεμένες με αυτό πληροφορίες
- **Διαγραφή**, διαγράφονται από το ST μεμονομένες ή ομάδες αναγνωριστικών

Τόσο η πρόσθεση και η διαγραφή, αλλά πολύ περισσότερο η αναζήτηση, θα πρέπει να μπορούν να γίνουν το δυνατόν ταχύτερα, αφού ενδιαφέρουν αρκετά τμήματα του Μεταγλωττιστή. Επιπλέον, θα πρέπει να υποστηρίζεται η έννοια της εμβέλειας και της θέας, αναπόσπαστες έννοιες των σύγχρονων γλωσσών προγραμματισμού.

Οι βασικές δομές που προσφέρονται για την υλοποίηση του Πίνακα Συμβόλων είναι οι **Συνδεδεμένες Λίστες** (linked lists) (4.2.1), τα **Δέντρα Δυαδικής Αναζήτησης** (binary trees) (4.2.2) και οι **Πίνακες Κατακερματισμού**

(hash tables) (4.2.3). Στην πράξη συνηθίζεται να χρησιμοποιείται ένας συνδυασμός αυτών, για να μπορούν να υποστηριχθούν οι πολλαπλές εμβέλειες, που είναι πιθανό να εμφανιστούν σε ένα πρόγραμμα (4.2.4).

4.2.1 Συνδεδεμένες Λίστες (Linked Lists)

Πρόκειται για την κλασική δομή της απλής συνδεδεμένης λίστας, που αποτελείται από έναν κόμβο για κάθε αναγνωριστικό. Τα δεδομένα κάθε κόμβου αποτελούνται από δύο τμήματα. Το πρώτο περιέχει όλες τις σχετικές πληροφορίες και το δεύτερο είναι δείκτης στον κόμβο που ακολουθεί.

Το πλεονέκτημά τους, είναι το μικρό μήκος μνήμης που απαιτεί η υλοποίησή τους. Μειονέκτημά τους, είναι το μεγάλο κόστος σειριακής αναζήτησης, που πρέπει να εκτελείται όχι μόνο κατά την ανάκτηση αλλά και για κάθε εισαγωγή.

4.2.2 Δέντρα Δυαδικής Αναζήτησης (Binary Trees)

Τα Δέντρα Δυαδικής Αναζήτησης, είναι σε θέση να προσφέρουν καλύτερες επιδόσεις αναζήτησης και κατ' επέκτασιν, εισαγωγής, έναντι των απλών συνδεδεμένων λιστών. Οι κόμβοι τους έχουν όμοια μορφή με αυτούς της προηγούμενης δομής, με την διαφορά ότι το πλήθος των δεικτών, προς επόμενους κόμβους είναι δύο, ένας για τον αριστερό κόμβο και ένας για τον δεξί κόμβο.

Το βασικό τους πλεονέκτημα είναι, πως για εισαγωγή τυχαίων κόμβων στο δέντρο, κάθε κόμβος έχει κατά μέσο όρο, ίσο πλήθος υποκόμβων στον αριστερό και στον δεξί του κλώνο. Αυτό σημαίνει ότι κατά μέσο όρο η αναζήτηση είναι ανάλογη του λογαρίθμου του πλήθους των κόμβων με βάση το

2 ($O(log_2(\pi))$ όπου π το πλήθος των κόμβων). Πρόκειται για την βέλτιστη (best) περίπτωση και λέμε ότι το δέντρο είναι ισοσταθμισμένο (balanced).

Όμως στον αντίποδα βρίσκεται η χειρότερη (worst) περίπτωση, όπου τα αναγνωριστικά εισάγονται ήδη ταξινομημένα, οπότε το αποτέλεσμα είναι να καταλήξουμε σε ένα δέντρο που μοιάζει με μια συνδεδεμένη λίστα, με την διαφορά ότι απαιτεί περισσότερη μνήμη, λόγω του δεύτερου, αχρησιμοποίητου δείκτη, του κάθε κόμβου.

4.2.3 Πίνακες Κατακερματισμού (Hash Tables)

Πρόκειται για πίνακες σταθερού μεγέθους k, όπου η θέση στην οποία θα πρέπει να τοποθετηθεί κάθε αναγνωριστικό, υπολογίζεται από την **συνάρτηση κατακερματισμού** (hash function). Αυτή υπολογίζει τον αύξοντα αριθμό της θέσης, συνήθως βάση της συμβολοσειράς του αναγνωριστικού. Μία πολύ απλή συνάρτηση κατακερματισμού, θα μπορούσε να παίρνει το άθροισμα των ASCII κωδικών των χαρακτήρων που απαρτίζουν την συμβολοσειρά και να το διαιρεί με το μήκος του πίνακα κατακερματισμού k. Φυσικά σε περίπτωση που για δύο διαφορετικά αναγνωριστικά η συνάρτηση κατακερματισμού επιστρέψει την ίδια θέση τότε λέμε ότι έχουμε σύγκρουση(collision). Την διαχείριση των συγκρούσεων μπορεί να την κάνει κανείς, με αρκετούς τρόπους, πιο συνηθισμένοι από τους οποίους είναι, με χρήση μιας ή περισσοτέρων συνδεδεμένων λιστών.

Το πλεονέκτημά τους είναι ότι προσφέρει σταθερό χρόνο πρόσβασης σε οποιοδήποτε αναγνωριστικό ίσο με το χρόνο εκτέλεσης της συνάρτησης κατακερματισμού.

Βασικό μειονέκτημά του είναι το σταθερό μήκος του. Είτε θα έχουμε σπατάλη μνήμης, όταν υπάρχουν λιγότερα αναγνωριστικά, είτε θα αναγκαστούμε να δημιουργήσουμε έναν νέο, μεγαλύτερο πίνακα, και να ξανα-υπολογίσουμε όλες τις θέσεις, αν το πλήθος των αναγνωριστικών, ξεπερνά το μήκος του.

Αξίζει να σημειωθεί, πως προσθέτοντας έναν πίνακα αποθήκευσης στον οποίο εντέλει δείχνει ο πίνακας κατακερματισμού, μπορούμε να υποστηρίξουμε την έννοια της θέας, έχοντας τον πίνακα κατακερματισμού να δείχνει στον κόμβο του άμεσα ορατού αναγνωριστικού, το οποίο με την σειρά του, έχει ένα δείκτη προς το ομώνυμο αναγνωριστικό του αμέσως υψηλότερου επιπέδου.

4.2.4 Υποστήριξη Πολλαπλών Εμβελειών

Στην πράξη υπάρχει η ανάγκη να υποστηριχθεί η έννοια των πολλαπλών εμβελειών, αφού αυτό προστάζουν οι ορισμοί των περισσότερων Γλωσσών Προγραμματισμού. Μία πιθανή υλοποίηση, που επιτρέπει κάτι τέτοιο, αποτελείται από μία **στοίβα αποθήκευσης**, στην οποία δείχνει ένας **πίνακας κατακερματισμού** και μία **στοίβα εμβελειών**.

Η εισαγωγή και διαχείριση των αναγνωριστικών, όσον αφορά τον πίνακα κατακερματισμού και την στοίβα/πίνακα αποθήκευσης, γίνεται όπως ακριβώς και στην υλοποίηση με βάση τον Πίνακα Κατακερματισμού (4.2.3). Η διαφορά έγκειται στο γεγονός, ότι κάθε φορά που συναντάται μία νέα εμβέλεια στο πρόγραμμα, τότε αυτή εισάγεται στην στοίβα εμβελειών, με έναν δείκτη στην εκάστοτε κορυφή της στοίβας αποθήκευσης. Όταν πάψει να υφίσταται μία εμβέλεια, αφαιρείται από την στοίβα εμβελειών, ενώ παράλληλα αφαιρούνται και όσα αναγνωριστικά βρίσκονται πάνω από το σημείο του πίνακα αποθήκευσης,

στο οποίο έδειχνε η εμβέλεια.

Κεφάλαιο 5

ΣΗΜΑΝΤΙΚΗ ΑΝΑΛΥΣΗ

5.1 Τι είναι η Σημαντική Ανάλυση

Η Σημαντική Ανάλυση βασίζεται στον Σημασιολογικό ορισμό της Γλώσσας, όπως η Συντακτική Ανάλυση βασίζεται στον Συντακτικό ορισμό της Γλώσσας μέσω μίας Γραμματικής (CFG). Η Συντακτική Ανάλυση χρησιμοποιείται για τον έλεγχο της συντακτικής ορθότητας των προγραμμάτων, με αυτήν εντοπίζουμε λάθη, όπως η πρόσθεση ενός ακέραιου αριθμού με την δεσμευμένη λέξη if, ενώ δεν πρόκειται να διαμαρτυρηθεί για μία έκφραση της μορφής "αναγνωριστικό + αναγνωριστικό" ακόμη και αν το ένα από τα δύο αναγνωριστικά είναι το όνομα μίας συνάρτησης. Αντίθετα κατά την διάρκεια της Σημαντικής Ανάλυσης ελέγχουμε την ορθότητα των εκφράσεων, βάση της σημασίας τους, λαμβάνοντας δηλαδή υπόψιν την φύση τους ή αλλιώς το τύπο τους. Στο παραπάνω παράδειγμα, λόγου χάρη, αν δεν αντιστοιχούν και τα δύο αναγνωριστικά σε μεταβλητές του ίδιου αριθμητικού τύπου, για τις περισσότερες Γλώσσες Προγραμματισμού, θα θεωρούταν σημαντικό σφάλμα και θα εμφανιζόταν το ανάλογο διαγνωστικό μήνυμα.

Μία ακόμη διαφορά μεταξύ του Συντακτικού και της Σημαντικής μιας Γλώσσας, είναι, πως η δεύτερη, σε αντίθεση με τον πρώτο, συνήθως δεν ορίζονται τυπικά αλλά μέσω φυσικής Γλώσσας και με χρήση παραδειγμάτων. Εν μέρη αυτό οφείλεται στο ότι είναι δύσκολο να διατυπωθεί τυπικά η πλήρης σημασιολογία ενός προγράμματος, όπως θα δούμε παρακάτω.

Φυσικά για να μπορεί να γίνει η Σημαντική Ανάλυση, θα πρέπει προηγουμένως να έχει διαπιστωθεί τόσο η Λεξική όσο και η Συντακτική αρτιότητα του προγράμματος.

Στην πράξη, μέχρι στιγμής, δεν έχει καταστεί δυνατό να γίνεται πλήρης

Σημαντική Ανάλυση των προγραμμάτων. Οι περισσότεροι Μεταγλωττιστές υλοποιούν μόνο την Στατική Σημαντική, ενώ κάποιοι περιέχουν ψήγματα Δυναμικής Σημαντικής την οποία χρησιμοποιούν μόνο για την εμφάνιση προειδοποιητικών μηνυμάτων, χωρίς να παρακωλύουν την ολοκλήρωση της Μεταγλώττισης.

5.1.1 Στατική Σημαντική

Η Στατική Σημαντική είναι υπεύθυνη για την ερμηνεία των λεξημάτων και τον έλεγχο της ορθής χρήσης τους. Μέσα από την ακολουθία λεξημάτων, ο Μεταγλωττιστής, αρχικά αντιλαμβάνεται την φύση τους. Για παράδειγμα, από μία υπογραφή συνάρτησης, ανακτά πληροφορίες, όπως ο αριθμός και ο τύπος των παραμέτρων, καθώς και ο τύπος της τιμής επιστροφής, ενώ το αναγνωριστικό πλέον θεωρείται συνάρτηση. Από τη δήλωση μιας μεταβλητής ή μιας σταθεράς αναγνωρίζει τον τύπο τους. Για ένα μπλοκ κώδικα την αρχή και το τέλος του.

Επιπλέον την Στατική Σημαντική μίας Γλώσσας μπορούμε να την περιγράψουμε και τυπικά με την βοήθεια των περιβαλλόντων τύπων (type environments). Έστω η Γλώσσα C. Για την δήλωση:

Παράθεση 5.1: "Δήλωση συνάρτησης add στην Γλώσσα C"

```
int add(int& a, int b) {
        int sum;
        sum = a + b;

        return sum;
}
```

κατασκευάζεται το περιβάλλον Γ_1 όπου:

$$\Gamma_1 = \left\{ \begin{array}{l} sum \mapsto int, a \mapsto int, b \mapsto int, \\ add \mapsto int\ function(\ byref\ integer, byval\ integer \mapsto integer \end{array} \right\}$$

Έχοντας δηλώσει το περιβάλλον Γ_1 μπορούμε να πούμε ότι μία έκφραση E έχει τύπο τ σε αυτό, πράγμα το οποίο συμβολίζεται και ως:

$$\Gamma \vdash E : \tau$$

Έτσι μπορούμε να ορίσουμε κανόνες για τον τύπο μιας έκφρασης, σύμφωνα με τους τύπους των υποεκφράσεων, που την αποτελούν, χρησιμοποιώντας μία οριζόντια γραμμή, πάνω από την οποία βάζουμε τις υποεκφράσεις, ή αλλιώς τις προϋποθέσεις, ενώ κάτω από την γραμμή τον τύπο της έκφρασης, ή αλλιώς το συμπέρασμα.

Ως παράδειγμα παρουσιάζονται οι κανόνες που αφορούν τον πολλαπλασιασμό ακέραιου με ακέραιο και ακέραιου με πραγματικό:

$$\frac{\Gamma_1 \vdash E_1 : int \qquad \Gamma_1 \vdash E_2 : int}{\Gamma_1 \vdash E_1 + E_2 : int} \qquad \frac{\Gamma_1 \vdash E_1 : int \qquad \Gamma_1 \vdash E_2 : double}{\Gamma_1 \vdash E_1 + E_2 : double}$$

Όμοια μπορεί να οριστεί και ο τύπος που απαιτεί μία δήλωση βρόγχου όπως είναι η `while` για την Γλώσσα C:

$$\frac{\Gamma_1 \vdash E_1 : bool \qquad \Gamma_1 \vdash E_2 : statement}{\Gamma_1 \vdash while(E_1)\ E_2 : statement}$$

Τέλος ένας κανόνας μπορεί να είναι φωλιασμένος, παίρνοντας τη θέση μίας προϋπόθεσης.

5.1.2 Δυναμική Σημαντική

Η Δυναμική Σημαντική (dynamic semantics) καταπιάνεται με την περιγραφή της συμπεριφοράς του προγράμματος, κατά τον χρόνο εκτέλεσης. Πρόκειται για μία περιγραφή των ενεργειών του προγράμματος. Απαραίτητη προϋπόθεση για να μπορεί να γίνει η Δυναμική Σημαντική Ανάλυση, είναι το πρόγραμμα να έχει περάσει με επιτυχία όλους τους προηγούμενους ελέγχους, συμπεριλαμβανομένης της Στατικής Σημαντικής Ανάλυσης.

Επιγραμματικά, οι μέθοδοι τυπικών περιγραφών της Δυναμικής Σημαντικής χωρίζονται στις εξής κατηγορίες:

- η Λειτουργική Σημαντική (operational semantics) που θεωρεί το πρόγραμμα ως ακολουθία υπολογιστικών βημάτων, που αυτό εκτελεί.
- η Δηλωτική Σημαντική (denotational semantics) για την οποία το πρόγραμμα είναι ένα μαθηματικό αντικείμενο, συνηθέστερα μία συνάρτηση με πεδίο ορισμού και σύνολο τιμών.
- η Αξιωματική Σημαντική (axiomatic semantics) η οποία δίνει έμμεση περιγραφή του προγράμματος, ως ένα σύνολο λογικών προτάσεων που εκφράζουν τις ιδιότητές του.

Η μεγάλη δυσκολία τυπικής περιγραφής της Δυναμικής Σημαντικής, περιορίζει τους δημιουργούς Γλωσσών Προγραμματισμού στην περιγραφή της σε απλή γλώσσα (μεταγλώσσα), ενώ οι κατασκευαστές Μεταγλωττιστών δεν

συνηθίζουν να την συμπεριλαμβάνουν στις υλοποιήσεις τους, με μερικές εξαιρέσεις, συνήθως, στην φάση βελτιστοποίησης του παραγόμενου κώδικα.

5.2 Σημαντικοί Έλεγχοι

Οι Σημαντικοί έλεγχοι που γίνονται από τον Μεταγλωττιστή, είναι προσανατολισμένοι, στο να εντοπίζουν σφάλματα Μεταγλώττισης και πολύ λιγότερο σφάλματα χρόνου εκτέλεσης. Αυτό έχει ως επακόλουθο, να χρησιμοποιείται κυρίως η Στατική Σημαντική και ελάχιστα η Δυναμική, συνήθως για την διευκόλυνση των βημάτων παραγωγής κώδικα και βελτιστοποιήσεων.

Οι συνηθέστεροι έλεγχοι εντάσσονται σε μία από τις ακόλουθες κατηγορίες:

- έλεγχοι τύπων, παραδείγματα των οποίων χρησιμοποιήθηκαν πολύ συχνά μέχρι στιγμής.
- έλεγχοι ροής, που εντοπίζουν σφάλματα, όπως η παράληψη επιστροφής τιμής από συνάρτηση σε όλες της περιπτώσεις.
- έλεγχοι ύπαρξης ονομάτων, αφού δεν είναι δυνατόν να χρησιμοποιηθεί ένα αναγνωριστικό σε κάποια έκφραση, χωρίς να είναι γνωστός το τύπος του.
- έλεγχοι μοναδικότητας, που εντοπίζουν σφάλματα, όπως η διπλή δήλωση ενός αναγνωριστικού εντός της ίδιας εμβέλειας.
- έλεγχοι συνέπειας, που αφορούν σφάλματα, όπως η χρήση του ίδιου αναγνωριστικού, τόσο στην δήλωση αρχής, όσο και στην δήλωση τερ-

ματισμού μίας δομής κώδικα, όπως είναι οι μονάδες προγράμματος στην Γλώσσα Ada.

Οι έλεγχοι αυτοί γίνονται με την βοήθεια του πίνακα συμβόλων, από τον οποίο αντλούνται οι σημαντικές πληροφορίες, που συνοδεύουν έναν αναγνωριστικό, όπως είναι ο τύπος μίας μεταβλητής. Κάποιοι από τους ελέγχους μπορούν φυσικά να ενσωματωθούν και στην Γραμματική της Γλώσσας, με σκοπό την εκτέλεσή τους κατά την Συντακτική Ανάλυση, όμως γενικά αποφεύγεται, αφού προσθέτει πολυπλοκότητα στην Γραμματική με αποτέλεσμα να δυσχεραίνει την υλοποίησή της.

5.3 Συστήματα Τύπων

Οι τύποι παίζουν πολύ σημαντικό ρόλο στις Γλώσσες Προγραμματισμού, αφενός, γιατί επιτρέπουν τον εντοπισμό λαθών κατά συγγραφή των προγραμμάτων, αφετέρου, γιατί επιτρέπουν στον Μεταγλωττιστή να οργανώσει αποτελεσματικότερα και με μεγαλύτερη ασφάλεια, την μνήμη στο εκτελέσιμο πρόγραμμα.

5.4 Βασικοί τύποι

Σχεδόν όλες οι Γλώσσες προγραμματισμού έχουν μία σειρά βασικών τύπων οι οποίοι συνήθως είναι κάποιοι ή όλοι από τους:

- ακεραίους αριθμούς (integers)
- πραγματικούς αριθμούς (real numbers), σε κάποια μορφή κινητής υποδιαστολής (floating point)

- λογικές τιμές (boolean)
- χαρακτήρες (characters)

Αξίζει να σημειωθεί, πως στις αντικειμενοστρεφείς γλώσσες ο χρήστης (προγραμματιστής) μπορεί να ορίσει νέους τύπους, που βασίζονται στους βασικούς τύπους και ίσως σε άλλους χρηστικούς τύπους, που έχουν οριστεί ή τουλάχιστον διακηρυχθεί προηγουμένως (user defined types). Κάποιες δηλώσεις επιβάλουν την ύπαρξη ενός κύριου τύπου (κλάσης) από τον οποίο κρέμονται όλοι οι άλλοι χρηστικοί τύποι (Java), καμιά φορά ακόμη και οι βασικοί τύποι (Ruby).

5.5 Κατασκευαστές Τύπων

Πέρα, όμως, από τους βασικούς τύπους, οι δημιουργοί Γλωσσών Προγραμματισμού συνηθίζουν να προσφέρουν την δυνατότητα της κατασκευής σύνθετων τύπων, χρησιμοποιώντας ως δομικά στοιχεία του βασικούς.

Κλασικά παραδείγματα σύνθετων τύπων είναι:

- οι **πίνακες** (arrays) οι οποίοι αποτελούν μία ακολουθία αντικειμένων/οντοτήτων του ίδιου τύπου, στα στοιχεία της οποίας αναφερόμαστε χρησιμοποιώντας κάποιον δείκτη (index) σε ένα σταθερό όνομα, το όνομα του πίνακα, συνήθως ακέραιο, ενώ το πλήθος τους ορίζεται με την δημιουργία του πίνακα.
- τα **καρτεσιανά γινόμενα** (products, tuples) τα οποία αποτελούνται από μία διατεταγμένη σειρά στοιχείων, όχι απαραίτητα του ίδιου τύπου, που προσπελαύνονται με κάποιο δείκτη.

- οι **εγγραφές** (structures, records), όπου πρόκειται για καρτεσιανά γινόμενα τον οποίων τα στοιχεία είναι ονοματισμένα.

- οι **διευθυσιοδέικτες** (pointers) οι οποίοι περιέχουν την διεύθυνση μνήμης άλλων στοιχείων συγκεκριμένου τύπου.

- οι **συναρτήσεις** (functions) στις οποίες εισάγεται ένας αριθμός παραμέτρων διαφόρων τύπων[1] και επιστρέφουν ένα στοιχείο ορισμένου τύπου.[2].

Βέβαια δεν δίνουν όλες οι Γλώσσες την ίδια ελευθερία με τους κατασκευαστές τύπων. Σε κάποιες, λόγου χάρη, μπορεί να επιτρέπεται η εφαρμογή ενός κατασκευαστή τύπου σε έναν σύνθετο τύπο, όπως είναι η δημιουργία ενός πίνακα πινάκων ακεραίων,[3] ενώ σε κάποιες άλλες, η εφαρμογή των κατασκευαστών, να περιορίζεται, είτε μόνο στους βασικούς, είτε σε υποσύνολο των σύνθετων τύπων.

5.6 Αντικειμενοστρεφείς τύποι

Η οικογένεια των Αντικειμενοστραφών Γλωσσών Προγραμματισμού επιτρέπει και σε κάποιες περιπτώσεις επιβάλει στους προγραμματιστές, να δημιουργούν νέους τύπους, επεκτείνοντας άλλους είτε βασικούς είτε σύνθετους. Αυτού του είδους οι τύποι, ονομάζονται κλάσεις ενώ τα στοιχεία αυτών των τύπων στιγμιότυπα ή αντικείμενα της κλάσης.

[1] Ο αριθμός των παραμέτρων θα μπορούσε να είναι μηδενικός, στην περίπτωση που η συνάρτηση δεν δέχεται παραμέτρους

[2] Στην υποπερίπτωση που δεν επιστρέφουν κάποιο στοιχείο ονομάζονται συνήθως **διαδικασίες** (procedures)

[3] Πρόκειται ουσιαστικά για έναν δισδιάστατο πίνακα ακεραίων.

Πρόκειται για τον ακρογωνιαίο λίθο του Αντικειμενοστρεφούς Προγραμματισμού, ενώ η σύνταξη αυτών των τύπων, συνδυάζει χαρακτηριστικά τόσο των συνηθισμένων τύπων, όσο και των συναρτήσεων (μεθόδων).

Κεφάλαιο 6

ΠΑΡΑΓΩΓΗ ΚΩΔΙΚΑ

6.1 Εισαγωγή

Το αρχικό πρόγραμμα, έχοντας περάσει από όλα τα προηγούμενα στάδια, έχει πλέον αναλυθεί επαρκώς και έχουν εξαχθεί από αυτό όλες οι απαραίτητες πληροφορίες για την απόδοσή του σε μία άλλη Γλώσσα.

Για να διευκολυνθεί το έργο του κατασκευαστή του Μεταγλωττιστή, συνηθίζεται να αποδίδεται το πρόγραμμα πρώτα σε μία Ενδιάμεση Γλώσσα (6.2) και μετά από αυτήν στην Τελική Γλώσσα (6.3). Η Ενδιάμεση Γλώσσα έχει το ρόλο του σκαλοπατιού και επομένως το επίπεδό της, από την άποψη της πολυπλοκότητας, και της εκφραστικής δύναμης, βρίσκεται μεταξύ της Αρχικής και της Τελικής Γλώσσας. Δεδομένου ότι δεν είναι εύκολη η εύρεση της μέσης κατάστασης, μεταξύ αυτών των δύο, τις περισσότερες φορές η Ενδιάμεση Γλώσσα πλησιάζει σε ένα από τα δύο άκρα, ανάλογα με τις ανάγκες του κατασκευαστή του Μεταγλωττιστή.

6.2 Ενδιάμεσος Κώδικας

Το τμήμα του Μεταγλωττιστή που ασχολείται με την παραγωγή του ενδιάμεσου κώδικα είναι το τέλος του εμπρόσθιου τμήματος (front), επομένως επιτρέπει τον διαχωρισμός σε εμπρόσθιο και οπίσθιο τμήμα (back). Όπως προαναφέρθηκε ένας από τους λόγους που χρησιμοποιούμε την παραγωγή ενδιάμεσου κώδικα, είναι η δημιουργία ενός σκαλοπατιού, μεταξύ Αρχικού και Τελικού Κώδικα, με αποτέλεσμα την απλούστευση της μετάβασης. Στο ίδιο πνεύμα, είναι θεμιτό να παρεμβληθούν περισσότερα του ενός ενδιάμεσα βήματα, με φυσικό αποτέλεσμα την ακόμα ευκολότερη μετάβαση προς τον Τελικό Κώδικα.

Επιπλέον, ο Ενδιάμεσος Κώδικας είναι ένα ακόμα σημείο στο οποίο μπορούν να εφαρμοστούν βελτιστοποιήσεις.

Για την παραγωγή Ενδιάμεσου κώδικα χρησιμοποιείται συχνότατα η τεχνική μετάφρασης οδηγούμενης από την Σύνταξη (syntax-directed translation). Εναλλακτικά μπορεί να χρησιμοποιηθεί η τεχνική μετάφρασης οδηγούμενης από τη Σημαντική, με ότι αυτό συνεπάγεται (βλέπε 5), γι αυτό υιοθετείται σπάνια.

6.2.1 Μετάφραση Οδηγούμενη από τη Σύνταξη

Για να εφαρμοστεί η Μετάφραση Οδηγούμενη από τη Σύνταξη, ξεκινά η προετοιμασία συνδέοντας τις δομές της Αρχικής Γλώσσας με τις αντίστοιχες δομές της Ενδιάμεσης Γλώσσας. Στη συνέχεια, δουλεύουμε πάνω στη Γραμματική της Γλώσσας, που περιγράφει τη Σύνταξη, την οποία διευρύνουμε τοποθετώντας ανάμεσα στα σύμβολα της Γραμματικής του δεξιού μέρους των κανόνων παραγωγής, **σύμβολα μετάφρασης**, περικλειόμενα από άγκιστρα ('{' και '}'). Η διαδικασία ολοκληρώνεται υλοποιώντας τα παραπάνω και ενσωματώνοντάς τα στον Συντακτικό Αναλυτή.

Κατά τον σχεδιασμό της Μετάφρασης η διαδικασία παριστάνεται με την βοήθεια ενός διαγράμματος, ενώ η Γραμματική επαυξάνεται με σύμβολα μετάφρασης, που αναλύονται με την βοήθεια σημασιολογικών ρουτινών, οι οποίες περιγράφονται με την βοήθεια ψευτοκώδικα.[1]

Επιπλέον, για κάθε σύμβολο της Γραμματικής, τερματικό ή μη, ορίζουμε μία σειρά από μεταβλητές/ιδιότητες (attributes) στις οποίες διατηρούνται πλη-

[1]Για τις ανάγκες του πειραματικού Μεταγλωττιστή της PCL χρησιμοποιείται ψευτοκώδικας στη Γλώσσα C, αφού αυτή είναι η Γλώσσα υλοποίησης.

ροφορίες που βοηθούν την Μετάφραση. Ο συμβολισμός που ακολουθείται είναι της μορφής

$$\chi.\alpha$$

όπου χ είναι το σύμβολο της Γραμματικής ενώ α είναι η ιδιότητα. Για τις περιπτώσεις, όπου εμφανίζεται περισσότερες από μία φορές, ένα σύμβολο, στον ίδιο κανόνα της Γραμματικής, οι εμφανίσεις απαριθμούνται με φυσικούς αριθμούς.

6.2.2 Ενδιάμεση Γλώσσα

Η επιλογή της ενδιάμεσης Γλώσσας γίνεται βάση δύο κυρίως κριτηρίων, της διευκόλυνσης της μετάβασης από την Αρχική στην Τελική Γλώσσα, αλλά και την δυνατότητα εφαρμογής βελτιστοποιήσεων στο πρόγραμμα, όταν είναι εκφρασμένο σε αυτή τη Γλώσσα.

Συνηθέστερα, η παράσταση της ενδιάμεσης Γλώσσας, γίνεται σε δυαδική, μορφή από τον Μεταγλωττιστή, αφού έτσι αφενός εξοικονομείται χώρος, αφετέρου είναι ευκολότερη και σαφώς ταχύτερη η επεξεργασία των προγραμμάτων, όταν βρίσκονται σε αυτή τη μορφή. Παράλληλα, όμως, η κάθε ενδιάμεση Γλώσσα συνοδεύεται και από μία αντίστοιχη παράσταση αναγνώσιμη από ανθρώπους, με σκοπό την διευκόλυνση του σχεδιασμού, αλλά και του ελέγχου του συγκεκριμένου τμήματος του Μεταγλωττιστή.

Υπάρχουν αρκετές γλώσσες που συνηθίζεται να χρησιμοποιούνται ως ενδιάμεσες. Οι γλώσσες αυτές είναι συνήθως γλώσσες θεωρητικών (virtual, εικονικών) μηχανών, όπως παραδείγματος χάρην one-register, two-register,

three-register,unlimited register machines, pushdown και Turing machines. Στην συνέχεια αναφέρονται οι σημαντικότερες.

Τετράδες

Οι Τετράδες (quadruples) ορίζονται ως εντολές της μορφής:

$$n : op, x, y, z$$

όπου n είναι ένας φυσικός αριθμός που έχει την θέση ετικέτας για την εντολή αυτή, *op* είναι ο τελεστής (operator), π.χ. $+, -, *, /, := ...$, και x, y, z είναι τα τελούμενα (operants). Η ερμηνεία των τελουμένων γίνεται κάθε φορά βάση του τελεστή της τετράδας. Αν, λόγου χάρη, έχουμε τον τελεστή του πολλαπλασιασμού τότε η δήλωση `z := x * y;` της Γλώσσας PCL μεταφράζεται στην Γλώσσα των Τετράδων ως `1: *, x, y, z`. Ακόμη στην περίπτωση που δεν χρησιμοποιούνται τα τελούμενα στο σύνολό τους, στη θέση όσων περισσεύουν τοποθετείται ο χαρακτήρας '$-$'.

Είναι συνηθισμένο πως πολλές από της δηλώσεις της Αρχικής Γλώσσας να αντιστοιχούν σε περισσότερες από μία δήλωσεις της Ενδιάμεσης Γλώσσας. Σε αυτές τις περιπτώσεις, οπουδήποτε απαιτούνται ενδιάμεσες μεταβλητές, παίρνουν ονόματα της μορφής $\$n$, όπου n ένας φυσικός αριθμός. Για παράδειγμα, η έκφραση της PCL `z := 6*x*x + 8*y + 7` αντιστοιχεί στις ακόλουθες προτάσεις Τετράδων:

```
1: *,  x,  x, $1
2: *,  6, $1, $2
3: *,  8,  y, $3
```

```
4: +, $2, $3, $4
5: +, $4,  7,  z
```

Τριάδες

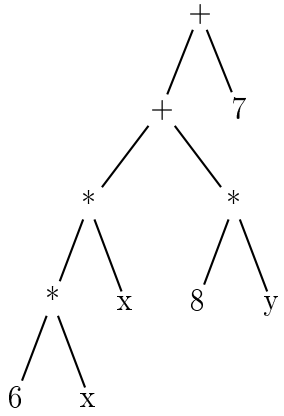
Οι Τριάδες (triples) είναι μία παραλλαγή των Τετράδων, με την διαφορά ότι έχουν δύο τελούμενα, ενώ η ετικέτα μίας εντολής μπορεί να χρησιμοποιηθεί στη θέση του αποτελέσματός της. Ως παράδειγμα, η έκφραση της PCL `z := 6*x*x + 8*y + 7` αντιστοιχεί στις ακόλουθες προτάσεις Τριάδων:

```
1: * ,  x ,  x
2: * ,  6 , (1)
3: * ,  8 ,  y
4: + , (2), (3)
5: + , (4),  7
6: :=, (5),  z
```

Αφηρημένα Συντακτικά Δέντρα

Τα Αφηρημένα Συντακτικά Δέντρα (abstract syntax trees) βασίζονται στα αντίστοιχα Συντακτικά Δέντρα, που παράγει η φάση της Συντακτικής Ανάλυσης. Όμως, η δομή τους διαφοροποιείται σε μεγάλο βαθμό. Ειδικότερα, τα τελούμενα, αποτελούν φύλλα του δέντρου, οι τελεστές ενδιάμεσους κόμβους, ενώ τα ενδιάμεσα αποτελέσματα, είναι τα υποδέντρα, που σχηματίζονται. Τα παραπάνω γίνονται εύκολα αντιληπτά, αν ανατρέξει κανείς στο διάγραμμα 6.1.

Διάγραμμα 6.1: Η έκφραση "$6 * x * x + 8 * y + 7$" ως Αφηρημένο Συντακτικό Δέντρο.



Επιπλέον, ένα Αφηρημένο Συντακτικό Δέντρο μπορεί να παρασταθεί γραμμικά διασχίζοντας το δέντρο προδιατεταγμένα, προσθέτοντας παρενθέσεις και κόμμα, για την διευκόλυνση της ανάγνωσης. Επομένως η έκφραση `6*x*x + 8*y + 7` γράφεται `+(+(*(*(6,x),x),*(8,y)),7)`.

Προθεματικός και Επιθεματικός Κώδικας

Ο **Προθεματικός κώδικας** (prefix) συντάσσεται γράφοντας τον τελεστή, ακολουθούμενο από τα τελούμενα, τα οποία μπορούν να είναι ακόμη και εκφράσεις γραμμένες σε προθεματικό κώδικα. Είναι ίδιος με την γραμμική μορφή του Αφηρημένου Συντακτικού Δέντρου χωρίς όμως, τις παρενθέσεις και τα κόμματα. Έτσι η έκφραση `6*x*x + 8*y + 7` γράφεται `+ + * * 6 x x * 8 y 7`.

Στην ίδια λογική κινείται και ο **Επιθεματικός Κώδικας** (postfix), με την διαφορά, ότι ο τελεστής ακολουθεί τα τελούμενα. Σε αντιδιαστολή με την παραγωγή από ένα Αφηρημένο Συντακτικό Δέντρο πρόκειται για μία "πρώτα σε βάθος" μεταδιατεταγμένη διάσχισή του. Η έκφραση `6*x*x + 8*y + 7` γράφεται `6x*x*8y*+7+`.

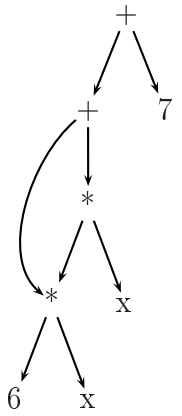
Το κυριότερο μειονέκτημα των δύο αυτών Γλωσσών είναι ότι δεν επιδέχονται βελτιστοποιήσεις σε ικανοποιητικό βαθμό.

Κατευθυνόμενοι Ακυκλικοί Γράφοι

Οι Κατευθυνόμενοι Ακυκλικοί Γράφοι βασίζονται και αυτοί στα Αφηρημένα Συντακτικά Δέντρα, τα οποία βελτιώνουν, αποφεύγοντας την εκτέλεση υπολογισμού ταυτόσημων αποτελεσμάτων, περισσότερες από μία φορά, συνδέοντας τα ανάλογα υποδέντρα, με όσους κόμβους γονείς είναι απαραίτητο.

Είναι προφανές ότι εξ' ορισμού οι Κατευθυνόμενοι Ακυκλικοί Γράφοι περιλαμβάνουν βελτιστοποίηση του παραγόμενου κώδικα. Το χαρακτηριστικό αυτό δεν επιτρέπει την γραμμική απεικόνισή τους.

Διάγραμμα 6.2: Η έκφραση "$6*x*x+6*x+7$" ως Κατευθυνόμενος Ακυκλικός Γράφος



6.2.3 Παραγωγή Ενδιάμεσου Κώδικα

Για τον πειραματικό Μεταγλωττιστή της Γλώσσας PCL επιλέχθηκαν οι Τετράδες, ως Γλώσσα απόδοσης του Ενδιάμεσου Κώδικα.

Τελούμενα

Το είδος, αλλά και το πλήθος (πόσα από τα τρία), των τελουμένων ορίζονται βάση του τελεστή της τετράδας και χωρίζονται σε δύο κύριες ομάδες, τα απλά και τα σύνθετα τελούμενα.

Τα **απλά τελούμενα** για την Γλώσσα PCL είναι:

- Οι σταθερές όλων των τύπων της PCL
- Τα ονόματα μεταβλητών
- Οι προσωρινές μεταβλητές για τις ανάγκες της μετάφρασης ($\$n$ όπου n Φυσικός αριθμός)
- Τα αποτελέσματα των συναρτήσεων.

Τα **σύνθετα τελούμενα** είναι:

- Η αποδεικτοδότηση (dereferenciation), όπου, αν έχουμε μία μεταβλητή x τύπου δείκτη σε τύπο t, τότε γράφοντας $[x]$, αποδεικτοδοτούμε την x και παίρνουμε το περιεχόμενο της θέσης μνήμης, στην οποία αυτή έδειχνε, πάντα του τύπου t.
- Η διεύθυνση (referenciation) που λειτουργεί αντίστροφα από την αποδεικτοδότηση όπου αν έχουμε μία μεταβλητή x τύπου t, τότε γράφοντας $\{x\}$ παίρνουμε την διεύθυνση της μεταβλητής x στη μνήμη.

Οι παραπάνω ομάδες τελουμένων αναφέρονται σε αντικείμενο/δεδομένα που βρίσκονται στη μνήμη. Υπάρχει όμως και μία σειρά τελουμένων τα οποία έχουν βοηθητικό χαρακτήρα. Τα τελούμενα αυτά είναι:

- Η ετικέτα μίας τετράδας που μπορεί να χρησιμοποιηθεί, λόγου χάρη, σε ένα άλμα.

- Η ετικέτα της PCL η οποία χρησιμοποιείται ως όρισμα στην εντολή `goto`.

- Η δήλωση του τρόπου περάσματος μίας μεταβλητής σε μία υπορουτίνα, `V` για πέρασμα με τιμή, `R` για πέρασμα κατ' αναφορά και `RET` για δήλωση της μεταβλητής επιστροφής μίας συνάρτησης.

- Η άγνωστη ετικέτα η οποία συμβολίζεται με `*`, στη θέση ενός τελούμενου, η οποία αντικαθίσταται με μία πραγματική ετικέτα, όταν ο Μεταγλωττιστής είναι σε θέση να την προσδιορίσει, όταν λόγου χάρη, βρεθεί η δήλωση `end if` μετά από ένα `if`.

Γραμματική Τετράδων

Η Γραμματική της Γλώσσας των Τετράδων, που χρησιμοποιήσαμε για τον πειραματικό Μεταγλωττιστή της PCL είναι η ακόλουθη:

```
<program>  ::=(<quadruple>)*
<quadruple>::=<label>":"<opname>","<operand>","<operand>"
               ,"<operand>
<label>    ::=<integer-const>
<opname>   ::= "unit" | "endu" | "+" | "-" | "*" | "/" | "%"
             | ":=" | "array" | "=" | "<>" | ">" | "<"
             | ">=" | "<=" | "ifb" | "label" | "jumpl"
             | "call" | "par" | "ret"
<operand>  ::= <simple> | "[" <simple> "]" | "{" <simple> "}"
```

```
              | <label> | <pass-mode> | "*" | "-"
<simple>    ::= <constant> | <object> | <temporary> | "$$"
<pass-mode>::= "V" | "R" | "RET"
<constant>  ::= <integer-const> | <real-const> | "true"
              | "false" | <char-const> | <string-literal> | "nil"
<object>    ::= <id>
<temporary>::=  "$" <integer-const>
```

Τα μη-τερματικά σύμβολα που δεν ορίζονται στην παραπάνω Γραμματική, συμπίπτουν με αυτά της Γραμματικής για την Γλώσσα PCL. Επιπλέον, τα σύμβολα `unit` και `endu` χρησιμοποιούνται για την δήλωση της αρχής και του τέλους ενός υποπρογράμματος, ενώ το σύμβολο `$$` είναι η μεταβλητή επιστροφής μίας συνάρτησης.

Υλοποίηση της Μετάφρασης σε Ενδιάμεση Γλώσσα

Για την υλοποίηση της Μετάφρασης σε ενδιάμεση Γλώσσα ανατίθενται σε κάθε σύμβολο της αρχική Γραμματικής ιδιότητες, που αφορούν, τόσο το περιεχόμενό του, όσο και θέσεις μετάβασης, σε περίπτωση διακλαδώσεων της ροής του προγράμματος. Αυτό συμβαίνει στις περιπτώσεις εντολών σαν την `if` και `while`.

Ταυτόχρονα, για την αποφυγή επανάληψης κώδικα, καθώς και για την ευαναγνωσημότητα του κώδικα, ορίζονται και υλοποιούνται υπορουτίνες, οι οποίες επιφορτίζονται με την διαχείριση, τόσο των ιδιοτήτων των συμβόλων, όσο και με την παραγωγή των τετράδων, δηλαδή του ενδιάμεσου κώδικα.

6.3 Τελικός Κώδικας

Η φάση παραγωγής του Τελικού Κώδικα (code generation) υλοποιείται από το τελευταίο και σπουδαιότερο τμήμα του Μεταγλωττιστή. Όπως είναι άλλωστε φυσικό, αυτή η φάση της Μεταγλώττισης, εμπεριέχει τον μεγαλύτερο βαθμό πολυπλοκότητας. Η εξαγωγή αυτού του τμήματος αποτελεί το τελικό πρόγραμμα μεταγλώττισης, εκφρασμένο στην Τελική Γλώσσα.

6.3.1 Γεννήτρια Τελικού Κώδικα

Το εν λόγω τμήμα του Μεταγλωττιστή ονομάζεται Γεννήτρια Τελικού Κώδικα (code generator). Κατά την εκτέλεσή του, τροφοδοτείται με τον ενδιάμεσο κώδικα, την ορθότητα του οποίου έχουμε ως δεδομένη, και με τον πίνακα συμβόλων, όπου τα ονόματα που περιλαμβάνει, αντιστοιχίζονται σε θέσεις μνήμης, παράγοντας τον τελικό κώδικα. Για να θεωρηθεί αποδεκτή μία υλοποίησή του θα πρέπει ο κώδικας που παράγει να είναι σωστός, σύμφωνα με τον ορισμό της τελικής Γλώσσας, να είναι υψηλής ποιότητας, βάση των χαρακτηριστικών του Υπολογιστικού Συστήματος, για το οποίο προορίζεται και φυσικά ο αλγόριθμος κατασκευής του τελικού κώδικα, θα πρέπει να είναι αποδοτικός, όσον αφορά τις απαιτήσεις του σε πόρους, όπως ο χρόνος επεξεργασίας και οι απαιτήσεις σε μνήμη.

Η διαδικασία μετατροπής του ενδιάμεσου κώδικα, στον αντίστοιχο βέλτιστο τελικό κώδικα, είναι πρακτικά αδύνατη, δεδομένου, ότι αφενός πρόκειται για ένα μη αποφασιστό[2] πρόβλημα (undecidable), όπως είναι άλλωστε κάθε

[2]Αποφασιστός/μη αποφασιστός, από την αρχαία ελληνική και την καθαρεύουσα είναι ρηματικό επίθετο σε -τος και σημαίνει το μπορεί. π.χ. διαβατός ποταμός (=το ποτάμι μπορεί να το διαβούμε).

είδους βελτιστοποίηση, αφετέρου, τα προβλήματα επιλογής εντολών, αλλά και αξιοποίησης των καταχωρητών, είναι και αυτά με τη σειρά τους ανεπίλυτα/δυσεπίλυτα (NP-hard) προβλήματα.

Εύκολα, λοιπόν, κατανοεί κανείς το λόγο για τον οποίο η όλη διαδικασία υλοποιείται με ευρετικές (heuristic) τεχνικές, οι οποίες εκ φύσεως δεν δίνουν την βέλτιστη λύση, αλλά μία ικανοποιητική προσέγγισή της. Φυσικά οι αλγόριθμοι αυτοί θα πρέπει να είναι σχεδιασμένοι με μεγάλη προσοχή, αφού μικρές διαφοροποιήσεις μπορούν να προκαλέσουν μεγάλες μεταβολές στο μήκος αλλά και την ταχύτητα εκτέλεσης του παραγώμενου προγράμματος.

Ένα σημαντικό χαρακτηριστικό της εξαγωγής του τμήματος αυτού, είναι η Γλώσσα που χρησιμοποιείται για την απόδοση του τελικού κώδικα. Τρεις είναι οι βασικές επιλογές:

- Η Γλώσσα Μηχανής στην απόλυτη μορφή της (absolute machine language), δηλαδή σε δυαδική μορφή έτοιμη για τροφοδότηση τον επεξεργαστή, φορτώνοντας το πρόγραμμα σε προκαθορισμένη θέση της μνήμης και εκτελώντας, το χωρίς περαιτέρω παρεμβάσεις. Επιτρέπει την γρήγορη μετάφραση μικρών προγραμμάτων και προτιμάται για υλοποιήσεις εκπαιδευτικών μεταγλωττιστών.

- Η Γλώσσα Μηχανής στην επανατοποθετήσιμη και διασυνδέσιμη μορφή της (relocatable and linkable machine language), η οποία διαφέρει από την προηγούμενη στην μη οριστικοποίηση των διευθύνσεων μνήμης του εξαγομένου από τον Μεταγλωττιστή πρόγραμματος. Μεγάλο πλεονέκτημα είναι η δυνατότητα ξεχωριστής μεταγλώττισης των υποπρογραμμάτων, πράγμα το οποίο επιτρέπει την χρήση προμεταγλωττισμένων βι-

βλιοθηκών, με χρήση του συνδέτη προγραμμάτων (linker). Επιπλέον είναι δυνατή η φόρτωση του προγράμματος σε οποιαδήποτε θέση της μνήμης, αφού οι τελικές διεύθυνσης μνήμης οριστικοποιούνται από τον φορτωτή (loader) που αποτελεί αναπόσπαστο κομμάτι των σύγχρονων λειτουργικών συστημάτων και δημιουργεί την τελική μορφή του προγράμματος στη μνήμη. Πρόκειται για υλοποίηση που προτιμάται από την πλειονότητα των εμπορικών υλοποιήσεων.

- Η Συμβολική Γλώσσα (assembly language), η οποία είναι μία αναγνώσιμη από τον άνθρωπο μορφή και χρησιμοποιείται μάλιστα από τους προγραμματιστές, ως μία Γλώσσα αρκετά κοντά, αλλά υψηλότερου επιπέδου από την Γλώσσα Μηχανής (συμβολική γλώσσα μηχανής). Η χρήση της, απλουστεύει σε μεγάλο βαθμό, την διαδικασία παραγωγής τελικού κώδικα. Το αρνητικό, είναι ότι προσθέτει ένα ακόμα βήμα για την παραγωγή εκτελέσιμου κώδικα, αυτό της ύπαρξης της κλήσης ενός συμβολομετφραστή (assembler). Απαιτεί πολύ λιγότερη μνήμη και μπορεί να ενσωματωθεί σε Μεταγλωττιστές πολλών περασμάτων.

6.3.2 Υπολογιστικό Σύστημα Στόχος

Το υπολογιστικό σύστημα στόχος (target-machine) είναι ένας προσωπικός υπολογιστής (PC) με επεξεργαστή συμβατό με τον 8086, ο οποίος συνοδεύεται από μαθηματικό συνεπεξεργαστή ομότιμο του 8087. Τέτοιοι είναι όλοι οι επεξεργαστές που ακολουθούν την αρχιτεκτονική x86 και οι μαθηματικοί συνεπεξεργαστές της αρχιτεκτονικής x87. Φυσικά οι λειτουργίες των επεξεργαστών, που ακολουθούν τις παραπάνω αρχιτεκτονικές, έχουν εμπλουτιστεί με

τον καιρό. Τα σχετικά αυτά προθέματα δεν επηρεάζουν τον μεταγλωττιστή μας για την PCL (abwere compatibility).

Οι καταχωρητές του 8086

Πρόκειται για καταχωρητές μήκους 16 bit και ομαδοποιούνται ως εξής:

Οι καταχωρητές γενικής φύσεως είναι οι ax, bx, cx και dx. Μπορούμε να αναφερθούμε στα 8 περισσότερο σημαντικά bit (8 πρώτα) αυτών αντικαθιστώντας το x του ονόματος με h (high) και στα 8 λιγότερο σημαντικά bit (8 τελευταία), αντικαθιστώντας το x με l (low). Για παράδειγμα, μπορούμε να αναφερθούμε στον ax ως ah για τα 8 περισσότερο σημαντικά bit και ως al για τα 8 λιγότερο σημαντικά. Επιπλέον ο ax λειτουργεί ως συσσωρευτής (accumulator) για τις αριθμητικές πράξεις ακεραίων.

Οι καταχωρητές δείκτες που προορίζονται για την δεικτοδότηση της μνήμης. Αυτοί είναι ο δείκτης στοίβας (stack pointer) sp, ο οποίος ενημερώνεται αυτόματα από τον 8086 όταν εκτελεί εντολές στοίβας[3] και ο bp ο οποίος, παρόλο που δεν έχει προορισμένη χρήση, λειτουργεί ως δείκτης στοίβας εκτέλεσης, που κρατάει τα πλαίσια υποπρογραμμάτων από τους εμπορικούς μεταγλωττιστές καθώς και από την πειραματική υλοποίησή μας για την PCL.

Οι καταχωρητές αναφοράς (index registers) είναι οι si (source index) και di (destination index) οι οποίοι, παρότι μπορούν να χρησιμοποιηθούν σαν καταχωρητές γενικής φύσεως, είθισται να χρησιμοποιούνται για τον

[3]Υπάρχει η δυνατότητα απευθείας τιμοδότησής του από τον προγραμματιστή, κάτι το οποίο δεν συνηθίζεται, για ευνόητους λόγους.

υπολογισμό αντιστάθμισης (offset) μνήμης.

Οι καταχωρητές τμημάτων (segments) ρόλος των οποίων είναι η δεικτοδότηση περιοχών της μνήμης. Είναι οι:

- cd — code segment, ο οποίος αφορά το τμήμα κώδικα του προγράμματος.
- ds — data segment, ο οποίος αφορά το τμήμα δεδομένων.
- ss — stack segment, ο οποίος αφορά το τμήμα της μνήμης, όπου βρίσκεται η στοίβα του προγράμματος.
- es — extra segment, ο οποίος δεν έχει προκαθορισμένη χρήση.

Οι ειδικοί καταχωρητές από τους οποίους μας ενδιαφέρουν δύο. Ο μετρητής προγράμματος ip (instruction counter ή program counter - pc), ο οποίος περιέχει την διεύθυνση της επόμενης, προς εκτέλεση, εντολής και μεταβάλλεται, είτε αυτόματα, με μοναδιαία αύξηση, μετά την εκτέλεση μίας εντολής, είτε άμεσα, με χρήση κάποιας από τις εντολές άλματος. Δεύτερος, που έχει σημασία για την δική μας υλοποίηση, είναι ο καταχωρητής σημαιών (flag register) μήκους 9 bit (9 σημαιών) από τα οποία θα χρησιμοποιήσουμε τα εξής 4:

- zf (zero flag), το οποίο παίρνει την τιμή 1, όταν η τελευταία αριθμητική πράξη είχε μηδενικό αποτέλεσμα.
- cf (carry flag), το οποίο παίρνει την τιμή 1, όταν η τελευταία αριθμητική πράξη είχε κρατούμενο.
- of (overflow flag), το οποίο παίρνει την τιμή 1, όταν προκαλείται υπερχείληση από την τελευταία αριθμητική πράξη.

- sf (sign flag), το οποίο παίρνει την τιμή 1, όταν το αποτέλεσμα της τελευταίας αριθμητικής πράξης έχει αρνητικό πρόσημο.

Η διαχείριση μνήμης από τον 8086

Ο 8086 είναι σε θέση να υποστηρίξει μνήμη έως 1 MB (2^{20} bytes). Αυτό σημαίνει πως για την παράσταση μίας τυχαίας διεύθυνσης μνήμης απαιτούνται 20 bit. Όμως ο 8086 είναι ένας 16μπιτος επεξεργαστής, με αποτέλεσμα οι καταχωρητές του να υπολείπονται κατά 4 bit σχετικά με το έργο της δεικτοδότησης της κύριας μνήμης. Η λύση σε αυτό το πρόβλημα δόθηκε με την εισαγωγή της έννοιας των τμημάτων (segments). Έτσι, η κεντρική μνήμη χωρίστηκε σε 2^{16} τμήματα των 64 KB (2^{16} byte) με κάθε τμήμα να ξεκινά 16 byte μετά την αρχή του προηγουμένου. Είναι προφανές πως ένας καταχωρητής του 8086 έχει το σωστό μέγεθος για να μπορεί να αναφερθεί σε όλες τις διευθύνσεις μνήμης εντός ενός τμήματος. Εδώ έρχονται οι τέσσερις καταχωρητές τμημάτων να συμπληρώσουν την λύση δίνοντας την δυνατότητα ορισμού τμήματος. Έτσι μία φυσική διεύθυνση δίνεται από τον τύπο:

$$address = segment * 16 + \text{offset}$$

και παρίστανται ως:

$$segment : \text{offset}$$

όπου τα segment και offset συνηθίζεται να γράφονται στην δεκαεξαδική τους μορφή.

Χαρακτηριστικό αυτής της αρχιτεκτονικής είναι πως μία φυσική διεύθυνση μνήμης μπορεί να γραφεί με περισσότερους του ενός τρόπους. Για παράδειγμα η φυσική διεύθυνση FF_{16} μπορεί να γραφεί στην μορφή $address = segment * 16 + \mathrm{offset}$ ως:

- $FF_{16} = 0_{16} * 16_{10} + FF_{16}$ (με segment = 0)
- $FF_{16} = 1_{16} * 16_{10} + EF_{16}$ (με segment = 1)
- $FF_{16} = 2_{16} * 16_{10} + DF_{16}$ (με segment = 2)
- $FF_{16} = A_{16} * 16_{10} + 5F_{16}$ (με segment = 10)
- $FF_{16} = F_{16} * 16_{10} + F_{16}$ (με segment = 15)

Ειδικότερα ο καταχωρητής τμήματος κώδικα cs χρησιμοποιείται σε συνδυασμό με τον ip (instruction pointer). Ο καταχωρητής τμήματος δεδομένων ds χρησιμοποιείται, όταν έχουμε αναφορά στη μνήμη με σταθερή αντιστάθμιση (offset), μέσω των καταχωρητών αναφοράς si και di ή μέσω των καταχωρητών γενικής χρήσεως. Ο καταχωρητής τμήματος στοίβας ss χρησιμοποιείται, όταν έχουμε αναφορά στη μνήμη μέσω των καταχωρητών στοίβας sp ή bp. Τέλος ο καταχωρητής τμήματος γενικής χρήσεως, μπορεί να χρησιμοποιηθεί χωρίς κάποιον περιορισμό.

Οι εντολές Συμβολικής Γλώσσας για τον 8086

Οι εντολές Συμβολικής Γλώσσας για τον 8086 είναι της μορφής:

```
[label] opname [operand_1[, operand_2]]
```

Όπου η ετικέτα είναι προαιρετική όμως αν λείπει στην θέση της θα πρέπει να υπάρχει τουλάχιστον ένα κενό ενώ το πλήθος των τελεσταίων (operands) εξαρτάται από τον τελεστή.

Όπως συμβαίνει με τους καταχωρητές, έτσι και οι εντολές, χωρίζονται με την σειρά τους σε ομάδες. Αυτές αφορούν:

Εντολές Μεταφοράς από τις οποίες χρησιμοποιούμαι δύο. Πρώτη είναι η "`mov destination, source`" όπου τα source και destination μπορούν να είναι αναφορές, με οποιονδήποτε τρόπο, στην μνήμη ή σε καταχωρητή, ενώ το source μπορεί να είναι ακόμη και μία σταθερά. Επιπλέον, όταν θέλουμε να ορίσουμε το μήκος τον μεταφερόμενων δεδομένων, μπορούμε να το κάνουμε προσθέτοντας μπροστά από την αναφορά τον όρο `byte ptr` για 8 bit και `word ptr` για 16 bit. Η δεύτερη είναι η "`lea destination, source`" η οποία λειτουργεί παρόμοια με την `mov` διαφέροντας στο ότι δεν μεταφέρει το περιεχόμενο της θέσης μνήμης αλλά την διεύθυνσή της. Προφανώς η πηγή (source) πρέπει πάντα να είναι αναφορά στη μνήμη και όχι καταχωρητής.

Αριθμητικές πράξεις ακεραίων που υλοποιούνται από τις εντολές:

- "`add op1, op2`" πρόσθεση με τελούμενα των 16 bit και αποθήκευση του αποτελέσματος στο πρώτο τελούμενο.
- "`sub op1, op2`" αφαίρεση με τελούμενα των 16 bit και αποθήκευση του αποτελέσματος στο πρώτο τελούμενο.
- "`neg op`" υπολογισμός αντίθετου με τελούμενο 16 bit και αποθήκευση του στην ίδια θέση.

- "`imul op`" πολλαπλασιασμός με τελούμενο μήκους 16 bit, το οποίο πολλαπλασιάζεται με το περιεχόμενο του καταχωρητή ax, και δίνει αποτέλεσμα 32 bit, το οποίο μοιράζεται στους καταχωρητές ax και dx.
- "`idiv op`" ακέραια διαίρεση με τελούμενο τον διαιρέτη μήκους 16 bit, και διαιρετέο 32 bit μοιρασμένο στους ax και dx και αποθήκευση του πηλίκου στον ax και του υπολοίπου στον dx.
- "`cmp op1, op2`" σύγκριση των τελουμένων χωρίς αλλοίωσή τους, αλλά μόνο ενημέρωση του καταχωρητή σημαιών.

Ας σημειωθεί πως η αρχιτεκτονική x86 έχει υιοθετήσει για τους αρνητικούς αριθμούς την παράσταση "συμπλήρωμα ως προς 2".

Λογικές πράξεις που υλοποιούνται από τις εντολές:

- "`not op1, op2`" η λογική άρνηση
- "`and op1, op2`" η λογική σύζευξη
- "`or op1, op2`" η λογική διάζευξη
- "`xor op1, op2`" η αποκλειστική διάζευξη

Σε όλες τις παραπάνω εντολές οι τελεστές εφαρμόζονται bit προς bit, ενώ το αποτέλεσμα αποθηκεύεται πάντα στον πρώτο τελούμενο. Επιπλέον υπάρχει και η εντολή "`test op1, op2`" η οποία εκτελεί την πράξη της λογικής σύζευξης με την διαφορά ότι δεν αλλοιώνει κάποιο από τα τελούμενα, αλλά ενημερώνει μόνο τον καταχωρητή σημαιών.

Εντολές Αλμάτων οι οποίες χωρίζονται σε άλματα χωρίς συνθήκη και άλματα υπό συνθήκη. Η πρώτη κατηγορία αντιπροσωπεύεται από την εντολή "`jmp address`" ενώ η δεύτερη αποτελείται από τις εντολές, όπου η εφαρμοζόμενη συνθήκη για την πραγματοποίηση του άλματος είναι κάθε φορά:

- `jz` ή `je` ισότητα
- `jnz` ή `jne` διάφορο
- `jl` μικρότερο
- `jle` μικρότερο ή ίσο
- `jg` μεγαλύτερο
- `jge` μεγαλύτερο ή ίσο

Όλες οι εντολές ακολουθούνται από ένα τελούμενο, την διεύθυνση στη οποία πρόκειται να γίνει το άλμα. Ειδικότερα οι εντολές της δεύτερης κατηγορίας, για την εκτέλεσή τους, ελέγχουν τον καταχωρητή σημαιών. Επιπλέον, δεν έχουν την δυνατότητα άλματος πέραν των -128 έως +127 θέσεων μνήμης, πράγμα το οποίο αντιμετωπίζεται αντιστρέφοντας τον έλεγχο και συνδυάζοντάς τον με ένα άλμα χωρίς συνθήκη που δεν υπόκειται στον εν λόγω περιορισμό.

Εντολές στοίβας οι οποίες θεωρούν ότι η κορυφή της στοίβας βρίσκεται στην διεύθυνση ss:sp και είναι οι:

- "`push operand`" η οποία τοποθετεί το τελούμενο στην στοίβα και μειώνει τον sp κατά δύο

- "`pop operand`" η οποία αφαιρεί και επιστρέφει στο τελούμενο το κορυφαίο στοιχείο στην στοίβα και αυξάνει τον sp κατά 2

Τα τελούμενα των εντολών έχουν μήκος 16 bit, γι αυτό και αυξομειώνουν τον sp κατά 2 θέσεις κάθε φορά.

Κλήσεις υποπρογραμμάτων που υλοποιούνται με τις δύο εντολές:

- "`call address`" η οποία αποθηκεύει το περιεχόμενο του ip στην στοίβα και τοποθετεί σε αυτόν το περιεχόμενο του τελουμένου
- "`ret`" η οποία ανακτά την διεύθυνση επιστροφής από την κορυφή της στοίβας και την τοποθετεί πίσω στον ip

Πράξεις κινητής υποδιαστολής που υλοποιούνται από τον μαθηματικό συνεπεξεργαστή αρχιτεκτονικής x87. Εσωτερικά οι αριθμοί κινητής υποδιαστολής αποθηκεύονται με χρήση της παράστασης **διπλής επεκτεταμένης ακρίβειας** (double extended precision) βασιζόμενη στην τυποποίηση IEEE 754, μήκους 10 byte (80 bit). Οι περισσότερες Γλώσσες Προγραμματισμού υποστιρίζουν συνήθως δύο τύπους μικρότερης ακρίβειας, τους μονής ακρίβειας (single precition) των 4 byte (32 bit) και διπλής ακρίβειας των 8 byte (64 bit). Ο συνεπεξεργαστής διαθέτει 8 καταχωρητές μήκους 80 bit ο καθένας, οι οποίοι χρησιμοποιούνται ως μία στοίβα. Η στοίβα αυτή ονομάζεται `ST` και η αναφορά στα στοιχεία της γίνεται με τα `ST(0)`, `ST(1)` ...

Επιπλέον, όπως είδαμε στην εντολή mov μπορούμε να ορίζουμε το μήκος των τελουμένων με τους όρους `byte` και `word`, σε αυτούς προστίθεται ο όρος `tbyte` ο οποίος ορίζει μήκος 10 byte.

Οι εντολές που υποστηρίζονται από τον μαθηματικό συνεπεξεργαστή είναι:

- "`fld tbyte ptr[real1]`" η οποία λειτουργεί ως push στη στοίβα των καταχωρητών φορτώνοντας το τελούμενο μήκους 10 byte το οποίο ξεκινά στην θέση `real1`.
- "`fild word ptr[integer1]`" η οποία μετατρέπει τον ακέραιο της θέσης `integer1` σε αριθμό κινητής υποδιαστολής διπλής επεκτεταμένης ακρίβειας και τον τοποθετεί στην στοίβα καταχωρητών.
- "`fstp tbyte ptr[real2]`" είναι η αντίστροφη της fld και λειτουργεί ως εντολή pop για την στοίβα καταχωρητών.
- "`fistp tbyte ptr[integer2]`" είναι η αντίστροφη της `fild`. Ως προεπιλογή στρογγυλοποιεί τον αριθμό, επιλέγοντας στην περίπτωση της μέσης (.5) τον πλησιέστερο άρτιο. Πρόκειται για συμπεριφορά που προβλέπεται από το πρότυπο IEEE 754, πράγμα που μπορεί να αλλάξει, μεταβάλλοντας την τιμή καταχωρητών ελέγχου του συνεπεξεργαστή.
- οι εντολές μαθηματικών πράξεων `faddp`, `fsubp`, `fmulp`, `fdivp`, οι οποίες παίρνουν από τη στοίβα (pop) τους δύο τελεσταίους και επανατοποθετούν (push) σε αυτή το αποτέλεσμα.
- η εντολή `fchs` αντιστρέφει το πρόσημο του αριθμού που βίσκεται στην θέση `ST(0)` της στοίβας.
- η εντολή `fcompp` συγκρίνει τα δύο πρώτα στοιχεία της στοίβας, τα οποία αφαιρεί από τη στοίβα ενώ παράλληλα ενημερώνει τον καταχωρητή σημαιών του μαθηματικού συνεπεξεργαστή.

- η εντολή `fstsw` μεταφέρει τον καταχωρητή σημαιών του συνεπεξεργαστή, είτε στον καταχωρητή `ax` είτε σε κάποια θέση μνήμης, ανάλογα με το τελούμενο. Είναι ο μόνος τρόπος να ελεγχθεί ο καταχωρητής σημαιών του μαθηματικού συνεπεξεργαστή.

6.3.3 Το περιβάλλον εκτέλεσης

Το περιβάλλον εκτέλεσης (runtime environment) είναι κώδικας που προστίθεται από τον Μεταγλωττιστή σε όλα τα προγράμματα μίας δεδομένης Αρχικής Γλώσσας και υποστηρίζει λειτουργίες που παρέχει η Γλώσσα, σύμφωνα με τον ορισμό της, στους χρήστες της. Παραδείγματα τέτοιων λειτουργιών, είναι η δέσμευση ή η απελευθέρωση μνήμης, η αναφορά σε μή τοπικές μεταβλητές, η κλήση υποπρογραμμάτων και το πέρασμα μεταβλητών.

Δεδομένου ότι η PCL ανήκει στην οικογένεια των προστακτικών Γλωσσών προγραμματισμού, η περιγραφή των ζητημάτων που αφορούν το περιβάλλον εκτέλεσης γίνεται σύμφωνα με τις επιταγές του συγκεκριμένου μοντέλου.

Η παράσταση δεδομένων

Όπως η Αρχική έτσι και η Τελική Γλώσσα, ορίζουν κάποιες δομές δεδομένων, τις οποίες θα πρέπει αντιστοιχίσουμε. Είναι φυσικό οι δομές δεδομένων της Τελικής Γλώσσας, στην περίπτωση του Μεταγλωττιστή, να καθορίζονται από το υλικό (hardware) της μηχανής - στόχου.

Ειδικότερα, για τους ακέραιους και τους αριθμούς κινητής υποδιαστολής υιοθετούνται οι παραστάσεις που υποστηρίζονται από τις αρχιτεκτονικές x86 και x87 αντίστοιχα. Ο τύπος boolean παριστάνεται με έναν ακέραιο ορίζοντας

το 0 ως "ψευδές" και οποιονδήποτε άλλο αριθμό ως "αληθές". Οι συμβολοσειρές παριστάνονται ως μία ακολουθία χαρακτήρων μήκους 8 bit, όταν χρησιμοποιείται η κωδικοποίηση χαρακτήρων ASCII, ή 16 bit, όταν χρησιμοποιείται η κωδικοποίηση χαρακτήρων Unicode. Οι δείκτες έχουν την παράσταση θέσης μνήμης που χρησιμοποιείται από το υλικό, ενώ οι μή έγκυρες διευθύνσεις (nil), ορίζονται ως η θέση μνήμης 0, για την αρχιτεκτονική x86.

Η εγγραφή/πλαίσιο δραστηριοποίησης

Για κάθε ενότητα του προγράμματος ορίζεται η εγγραφή δραστηριοποίησης (activation record ή frame). Σε αυτό αποθηκεύονται:

- οι παράμετροι
- το αποτελέσμα
- πληροφορίες κατάστασης της μηχανής
- οι τοπικές μεταβλητές (της Αρχικής Γλώσσας)
- οι προσωρινες μεταβλητές (της ενδιάμεσης Γλώσσας)

Η προσπέλαση των παραπάνω πληροφοριών γίνεται μέσω ενός δείκτη πλαισίου, ο οποίος, όταν αναφερόμαστε στην αρχιτεκτονική x86, συνηθίζεται να τοποθετείται στον καταχωρητή `bp`.

Η οργάνωση της μνήμης

Σε ένα μεταγλωττισμένο πρόγραμμα της PCL η μνήμη χωρίζεται σε δύο τμήματα. Το πρώτο περιέχει το κώδικα και παραμένει σταθερό καθ' όλη τη διάρκεια της εκτέλεσης. Το δεύτερο περιέχει τις εγγραφές δραστηριοποίησης και

έχει μεταβλητό μήκος. Στην αρχή του τοποθετείται η εγγραφή δραστηριοποίησης του κυρίως προγράμματος, το οποίο μένει εκεί όσο εκτελείται το πρόγραμμα, και μετά από αυτό προστίθενται οι εγγραφές δραστηριοποίησης των υπόλοιπων ενοτήτων, όταν αυτές καλούνται και μέχρι το πέρας της εκτέλεσής τους. Με αυτό τον τρόπο μπορεί να υποστηριχτεί εύκολα και η αναδρομή, χωρίς να έχουμε προβλήματα ανάθεσης των τοπικών μεταβλητών. Η διαχείριση του χώρου αυτού γίνεται ως μία στοίβα (stack). Στο ίδιο τμήμα τοποθετείται μία σωρός (heap) η οποία ξεκινά στο αντίθετο άκρο και αναπτύσσεται προς την στοίβα. Αυτή χρησιμοποιείται για την δυναμική παραχώρηση μνήμης. Όταν η σωρός και η στοίβα συναντηθούν, τότε είμαστε σε μία κατάσταση, όπου έχει τελειώσει η μνήμη και συνήθως εμφανίζεται μήνυμα σφάλματος.

Αναφορά σε μη τοπικά δεδομένα

Σε αντίθεση με τα τοπικά δεδομένα όπου διαθέτουμε άμεσα τον δείκτη βάσης της εγγραφής δραστηριοποίησης και μπορούμε να τα βρούμε προσθέτοντας σε αυτόν την ανάλογη απόκλιση, για τα μη τοπικά δεδομένα, θα πρέπει προηγουμένως να βρεθεί η βάση της ανάλογης εγγραφής δραστηριοποίησης. Υπάρχουν δύο τεχνικές που μπορούν να χρησιμοποιηθούν για το σκοπό αυτό.

- Η χρήση **συνδέσμων προσπέλασης** οι οποίοι είναι μία ακόμα καταχώρηση στην εγγραφή δραστηριοποίησης και αποτελούν δείκτες στη βάση της αμέσως προηγούμενης (υψηλότερου επιπέδου) εγγραφής.

- Η χρήση ενός **πίνακα δεικτών** του οποίου στοιχεία είναι οι βάσεις, των εγγραφών δραστηριοποίησης και διάστασή τους το βάθος φωλιάσματος της κάθε εγγραφής.

Προφανώς η χρήση της δεύτερης τεχνικής μας δίνει αμεσότερα την λύση σε κάθε περίπτωση, αφού δεν απαιτεί αναδρομική αναζήτηση, απαιτεί όμως μεγαλύτερη φροντίδα, όσον αφορά την συντήρηση του πίνακα δεικτών.

Πέρασμα παραμέτρων

Πέρα από τις μη τοπικές μεταβλητές που εξετάστηκαν παραπάνω, τα υποπρογράμματα επικοινωνούν μεταξύ τους και με το πέρασμα παραμέτρων. Υπάρχουν αρκετοί τρόποι περάσματος παραμέτρων, με τον καθένα να κατέχει διαφορετικές ιδιότητες. Οι τρόποι αυτοί είναι:

- Το πέρασμα **κατ' αξία** (by value) ή **με αντίγραφο** (by copy) όπου δίνεται στο υποπρόγραμμα ένα αντίγραφο της πρωτότυπης μεταβλητής. Προφανώς οι αλλαγές, που τυχών γίνουν σε αυτό το αντίγραφο δεν μεταφέρονται ποτέ στο πρωτότυπο.

- Το πέρασμα **κατ' αναφορά** (by reference) ή με διευθύνσεις (by address) όπου στο υποπρόγραμμα δίνεται η θέση της αρχικής μεταβλητής, με αποτέλεσμα οποιαδήποτε τροποποίησή της, να γίνεται απευθείας σε αυτήν.

- Η κλήση **κατ' όνομα** όπου κάθε αναφορά στην παράμετρο εντός του υποπρογράμματος αντικαθίσταται από μία αναφορά στην πρωτότυπη μεταβλητή. Χρησιμοποιείται όταν μία κλήση στο υποπρόγραμμα αντικαθίσταται από το σώμα του υποπρογράμματος (inline υλοποίηση), όπως γίνεται στην περίπτωση που μία συνάρτηση της C δηλωθεί με την λέξη-κλειδί `inline` ή στην περίπτωση βελτιστοποιήσεων σχετικών με την ταχύτητα εκτέλεσης του προγράμματος.

- Το πέρασμα **κατ' ανάγκη** το οποίο είναι παραλαγή του περάσματος κατ' όνομα με τη διαφορά ότι η πραγματική διεύθυνση της παραμέτρου δεν υπολογίζεται εξ' αρχής αλλά κατά την πρώτη χρήση της.

- Το πέρασμα **κατ' αξία και αποτέλεσμα** (ή αντιγραφής-αποκατάστασης) το οποίο είναι ένας συνδυασμός των περασμάτων κατ' αξία και κατ' αναφορά. Περνιούνται τόσο το αντίγραφο όσο και η θέση της μεταβλητής. Κατά την διάρκεια της εκτέλεσης οποιεσδήποτε αλλαγές γίνονται στο αντίγραφο, περνούν στον καλούντα με το τέλος του υποπρογράμματος.

6.3.4 Η παραγωγή του τελικού κώδικα

Για την παραγωγή του τελικού κώδικα έχουμε στόχο την Γλώσσα Asembly για την Αρχιτεκτονική x86, όπως αυτή υλοποιείται από τον συμβολομεταφραστή MASM. Ειδικότερα το μοντέλο μνήμης που θα χρησιμοποιηθεί, είναι το .COM βάση του οποίου οι καταχωρητές τμήματος κώδικα (cs), δεδομένων (ds) και στοίβας (ss) δείχνουν στο ίδιο τμήμα (segment). Το γεγονός αυτό περιορίζει το συνολικό μήκος του προγράμματος μαζί με τα δεδομένα κατά την εκτέλεση στα 64 KB. Επιπλέον, η πρώτη εντολή του προγράμματος τοποθετείται στην θέση μνήμης 100_{16} (`100h`) ενώ η τιμή του καταχωρητή στοίβας (sp) ορίζεται ως $FFFE_{16}$ (`FFFEh`).

Η βιβλιοθήκη χρόνου εκτέλεσης

Η βιβλιοθήκη χρόνου εκτέλεσης (runtime library) περιέχει τα υποπρογράμματα που παρέχονται από την Γλώσσα βάση του ορισμού της. Για την PCL, παραδείγματος χάρην, παρέχονται οι συναρτήσεις εισαγωγής/εξαγωγής όπως

η `writeString` και η `readString`. Συνήθως, είναι ήδη υλοποιημένες στη Τελική Γλώσσα και συνδέονται με τα προγράμματα στο τελικό στάδιο της Μεταγλώττισης.

Η γεννήτρια τελικού κώδικα από την Γλώσσα των τετράδων

Η γεννήτρια τελικού κώδικα από την Γλώσσα των τετράδων αναλαμβάνει να μεταφράσει το πρόγραμμα που έχει αποδοθεί σε Τετράδες στην Τελική Γλώσσα. Για την διευκόλυνση του έργου της συγγραφής της γεννήτριας τελικού κώδικα ορίζονται μία σειρά από βοηθητικές ρουτίνες οι οποίες αναλαμβάνουν την παραγωγή κώδικα Assemly x86, για στενά ορισμένες λειτουργίες των προγραμμάτων.

Οι ρουτίνες αυτές είναι:

- `getAR(a)` φορτώνει την διεύθυνση πλαισίου του a στον καταχωρητή `si`.
- `updateAL` ενημερώνει τον σύνδεσμο προσπέλασης (όταν αυτος χρησιμοποιείται για αναφορά σε μη τοπικά δεδομένα).
- `load(R,a)` αποθηκεύει το a στον καταχωρητή `R`. Το a μπορεί να είναι οποιοσδήποτε τύπος δεδομένων εκτός από αριθμός κινητής υποδιαστολής.
- `loadReal(a)` εισάγει τον a στην στοίβα του μαθηματικού συνεπεξεργαστή.
- `store(R,a)` αντίστροφη της `load(R,a)`.
- `storeReal(a)` λειτουργεί αντίστροφα από την `loadReal(R)`, εξάγει από την στοίβα έναν αριθμό και τον τοποθετεί στο a.

- `name(p)` δημιουργεί μία μοναδική ετικέτα για την έναρξη της δομικής μονάδας `p`.

- `endof(p)` δημιουργεί μία μοναδική ετικέτα για την λήξη της δομικής μονάδας `p`.

- `label` δημιουργεί μοναδικές ετικέτες από συνόνυμες ετικέτες που βρίσκονται σε διαφορετικά τμήματα προγραμμάτων της Ενδιάμεσης ή της Αρχικής Γλώσσας.

Η παραγωγή του τελικού κώδικα υλοποιείται μέσω της γεννήτριας, αντιστοιχίζοντας τις τετράδες της ενδιάμεσης Γλώσσας, βάση του τελεστή τους, με μία αλληλουχία από άμεσες εντολές Γλώσσας Assembly x86, και κλήσεων στις βοηθητικές ρουτίνες, που προαναφέρθηκαν. Φυσικά για να μπορεί να εκτελεστεί το τελικό πρόγραμμα θα πρέπει προηγουμένως, να μεταφραστεί σε Γλώσσα μηχανής από τον MASM και να συνδεθεί από τον linker του συστήματος.

Κεφάλαιο 7

ΕΠΙΛΟΓΟΣ

7.1 Ανασκόπηση

Γίνεται εύκολα αντιληπτό, πως η κατασκευή ενός Μεταγλωττιστή, είναι ένα πολύπλοκο αλλά ταυτόχρονα, πολύ ενδιαφέρον εγχείρημα. Ο σχεδιαστής καλείται να εμβαθύνει στην Επιστήμη της Γλωσσολογίας, έτσι ώστε να καταφέρει να περιγράψει την γλώσσα, τόσο σε επίπεδο λεκτικών μονάδων, όσο και σε συντακτικό, αλλά και εννοιολογικό, επίπεδο. Μόνο τότε είναι σε θέση να ξεκινήσει την κατασκευή των τριών πρώτων τμημάτων του Μεταγλωττιστή, όπου αναλύεται το πηγαίο πρόγραμμα, το οποίο ακόμα βρίσκεται στην Αρχική Γλώσσα. Στην συνέχεια απαιτείται καλή αντίληψη της λειτουργίας της μηχανής - στόχου, πράγμα το οποίο επιτυγχάνεται με την βοήθεια των εργαλείων που προσφέρει ο τομέας της Αρχιτεκτονικής Υπολογιστών.

Έχοντας λοιπόν την ανάλυση τους αρχικού προγράμματος και γνωρίζοντας τον τελικό στόχο, σχεδιάζεται η μετάφραση του προγράμματος στην Τελική Γλώσσα, συνήθως χρησιμοποιώντας μία ή περισσότερες ενδιάμεσες Γλώσσες, κάθε μία από τις οποίες, πλησιάζει όλο και περισσότερο την Τελική. Σε καθένα από αυτά τα βήματα, όπου αυτό είναι επιθυμητό, εισάγονται διαδικασίες βελτιστοποίησης του κώδικα, έχοντας συνήθως ως στόχο, την επιτάχυνση της εκτέλεσης ή τη μείωση των απαιτήσεων σε κύρια μνήμη. Βέβαια υπάρχουν αρκετές περιπτώσεις, όπου η Τελική Γλώσσα, δεν είναι η Γλώσσα της μηχανής στόχου αλλά μία Γλώσσα που την πλησιάζει πολύ και είναι αναγνώσιμη από τον άνθρωπο. Πρόκειται για μία πρακτική που συνηθίζεται στην περίπτωση των πειραματικών Μεταγλωττιστών και έχει υιοθετηθεί για τον Μεταγλωττιστή που συνοδεύει το παρόν πόνημα.

Με χρήση κατάλληλων παραμέτρων στην εντολή μεταγλώττισης μπορεί σε

ξεχωριστούς φακέλους, πέρα από το εκτελέσιμο πρόγραμμα, που θα παραχθεί, να έχουμε τόσο τον ενδιάμεσο κώδικα (κώδικα τετράδων), όσο και τον x86 assembly κώδικα.

7.2 Γνώσεις και Δεξιότητες

Φυσικά, όταν κανείς καταπιάνεται με ένα έργο σχεδιασμού και υλοποίησης ενός Μεταγλωττιστή, είναι αναγκασμένος να κατακτήσει τις απαιτούμενες γνώσεις, ενώ ταυτόχρονα πρέπει να αναπτύξει τις ανάλογες δεξιότητες. Αποκτάται μία οικειότητα με την χρήση τόσο των Κανονικών Εκφράσεων όσο και των Κανονικών Γλωσσών. Παράλληλα, μπαίνουν σε εφαρμογή οι γνώσεις πάνω στις δομές δεδομένων, αφού είναι αναγκαίες για την διαχείριση των πληροφοριών, που δίνονται από το αρχικό πρόγραμμα και την σύνταξη του εξαγόμενου κώδικα. Επιπλέον, χρησιμοποιείται σε βάθος η Γλώσσα υλοποίησης και πολλές φορές εξερευνούνται τα όριά της, στην προσπάθεια της σύνταξης του κώδικα, που αποτελεί τον Μεταγλωττιστή. Το ίδιο γίνεται και με τις γεννήτριες μεταγλωττιστών, όπως είναι οι flex και bison, αφού και οι ίδιες υλοποιούν ειδικού τύπου Γλώσσες Προγραμματισμού υψηλότερου, βέβαια, επιπέδου.

Θα μπορούσε κανείς να ισχυριστεί πως πρόκειται για μία σειρά εξωτικών γνώσεων, αφού στην πράξη της αγοράς ή της έρευνας[1] είναι αρκετά σπάνιο να βρεθεί κάποιος αντιμέτωπος με ένα τέτοιο έργο. Στην πραγματικότητα, όμως, οι γνώσεις και οι δεξιότητες που αποκτούνται, μπορούν να χρησιμοποιηθούν σε άλλους τομείς της πληροφορικής, διευκολύνοντας και ταυτόχρονα επιταχύνοντας την λύση των προβλημάτων που εμφανίζονται σε αυτούς.

[1] Δεν μας είναι γνωστός κανένας εμπορικός μεταγλωττιστής, που υλοποιήθηκε στην πατρίδα μας.

Ένα κλασικό παράδειγμα είναι ο έλεγχος ορθότητας των δεδομένων που εισάγει ο χρήστης σε μία εφαρμογή. Είναι γνωστό πώς όσο το εύρος των χρηστών Ηλεκτρονικών Υπολογιστών μεγαλώνει, τόσο μειώνεται η εξοικείωση του μέσου χρήστη για τον τρόπο λειτουργίας, τόσο του υλικού, όσο και του λογισμικού. Αυτό δημιουργεί την απαίτηση οι εφαρμογές να είναι όλο και περισσότερο ανεκτικές σε εσφαλμένα δεδομένα εισαγωγής και θα πρέπει όχι απλώς να μην αποτυγχάνουν, αλλά να καθοδηγούν τον χρήστη στην διόρθωσή τους.

Βιβλιογραφία

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, Cambridge, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[GBJL02] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design.* John Wiley, 2002.

[Gyi96] Tibor Gyimothy, editor. *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, Linköping, Sweden, 24–26 April 1996. Springer.

[HEk08] Kenneth Hoste and Lieven Ee khout. Cole: compiler optimization level exploration. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, New York, NY, USA, 2008. ACM.

[Hol95] Jim Holmes. *Object-Oriented Compiler Construction.* Prentice-Hall, 1995.

[Kos98] Kai Koskimies, editor. *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1383 of *Lecture Notes in Computer Science*. Springer, 1998.

[KP92] Uwe Kastens and Peter Pfahler, editors. *Compiler Construction, 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, Paderborn, Germany, 5–7 October 1992. Springer.

[KR90] Brian W. Kerninghan and Dennis M. Ritchie. *Η Γλώσσα Προγραμματισμού C (The C Programming Language - Second Edition).* Κλειδάριθμος, Αθήνα, 1990. (Μετάφ.: Θωμάς Μωραΐτης Μεταλλειολόγος Μηχ. Ε.Μ.Π.).

[Lev00] Levine. *Linkers and Loaders.* Morgan Kaufmann, San Francisco, 2000.

[Mog00] Torben Æ. Mogensen. *Basics of Compiler Design*, volume 5 of *Kursusbog for Datalogi 1E.* DIKU, University of Copenhagen, Denmark, Universitetsparken 1, 2 edition, September 2000.

[Mor98] Robert Morgan. *Building an Optimizing Compiler.* Digital Press, 1998.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[oai94] Compiler construction, 5th international conference, CC'94, edinburgh, U.K., april 7-9, 1994, proceedings, 1994.

[oai99] Compiler construction, 8th international conference, CC'99, held as part of the european joint conferences on the theory and practice of software, ETAPS'99, amsterdam, the netherlands, 22-28 march, 1999, proceedings, 1999.

[Par92] Thomas W. Parsons. *Introduction to Compiler Construction.* Computer Science Press, 1992.

[PP92] T. Pittman and J. Peters. *The art of compiler design theory and practice.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Rüt98] Oliver Rüthing. *Interacting Code Motion Transformations: Their Impact and Their Complexity*, volume 1539 of *Lecture Notes in Computer Science.* Springer, 1998.

[SS07] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition.* CRC Press, Inc., Boca Raton, FL, USA, 2007.

[Ter00] P. D. Terry. Compilers and compiler generators: An introduction with c++, 2000.

[Wik09a] Wikipedia. Automata theory — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Automata_theory`, 2009. [Online; λήψη 20-Απριλίου-2009].

[Wik09b] Wikipedia. Computer programming — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Computer_programming`, 2009. [Online; λήψη 16-Απριλίου-2009].

[ΠΑ03] ΠΑΝΑΓΙΩΤΗΣ ΠΙΝΤΕΛΑΣ and ΠΑΝΑΓΙΩΤΗΣ ΑΛΕΦΡΑΓΚΗΣ. *Μεταγλωττιστές (Τόμος Α)*. Ελληνικό Ανοικτό Πανεπιστήμιο, Πάτρα, 2003.

[ΠΣ02] Νικόλαος Σ. Παπασπύρου and Εμμανουήλ Στ. Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, Αθήνα, 2002.

[Χει08] Θεοδόσιος Χειμωνίδης. *Υλικό μαθήματος Γλώσσες και Μεταγλωττιστές (Η Εξαμ.)*. Τμήμα Πληροφορικής, Σχολή Τ.Εφ., Αλεξ. Τ.Ε.Ι., Θεσσαλονίκη, 2008.

Παράθεμα Α

Η Γλώσσα PCL

Σε αυτό το παράρτημα περιγράφεται πλήρως η Σύνταξη και η Σημασιολογία της Γλώσσας PCL που χρησιμοποιείται ως παράδειγμα για την κατασκευή του πειραματικού Μεταγλωττιστή. Η Γλώσσα PCL είναι βασισμένη σε ένα γνήσιο υποσύνολο της ISO Pascal, είναι όμως εμπλουτισμένη με μερικές μικρές παραλλαγές. Λόγω των πολλών ομοιοτήτων της PCL με την Pascal, τόσο από πλευράς σύνταξης όσο και από πλευράς σημασιολογίας, η περιγραφή θα είναι σύντομη με εξαίρεση τα σημεία που οι δύο Γλώσσες διαφέρουν.

Α.1 Λεκτικές Μονάδες

Οι λεκτικές μονάδες της Γλώσσας PCL χωρίζονται στις παρακάτω κατηγορίες:

- Τις **λέξεις κλειδιά** οι οποίες είναι οι παρακάτω:

and	array	begin	boolean	char	dispose	div
do	else	end	false	forward	function	goto
if	integer	label	mod	new	nil	not
of	or	procedure	program	real	result	return
then	true	var	while			

- Τα **ονόματα**, τα οποία αποτελούνται από ένα πεζό ή κεφαλαίο γράμμα του λατινικού αλφαβήτου, πιθανώς ακολουθούμενο μία σειρά πεζών ή κεφαλαίων γραμμάτων, δεκαδικών ψηφίων ή χαρακτήρων υπογράμμισης (underscore). Τα πεζά γράμματα θεωρούνται διαφορετικά από τα αντίστοιχα κεφαλαία, επομένως τα ονόματα foo, Foo και FOO είναι διαφορετικά. Επίσης, τα ονόματα δεν πρέπει να συμπίπτουν με τις λέξεις κλειδιά που αναφέρθηκαν παραπάνω.

- Τις **ακέραιες σταθερές** χωρίς πρόσημο, που αποτελούνται από ένα ή περισσότερα δεκαδικά ψηφία. Παραδείγματα ακέραιων σταθερών είναι τα ακόλουθα:

  0 42 1284 00200

- Τις **πραγματικές σταθερές** χωρίς πρόσημο, που αποτελούνται από ένα ακέραιο μέρος, ένα κλασματικό μέρος και ένα προαιρετικό εκθετικό μέρος. Το ακέραιο μέρος αποτελείται από ένα ή περισσότερα δεκαδικά ψηφία. Το κλασματικό μέρος αποτελείται από το χαρακτήρα . της υποδιαστολής, ακολουθούμενο από ένα ή περισσότερα δεκαδικά ψηφία. Τέλος, το εκθετικό μέρος αποτελείται από το πεζό η κεφαλαίο γράμμα E, ένα προαιρετικό πρόσημο $+$ ή $-$ και ένα ή περισσότερα δεκαδικά ψηφία. Παραδείγματα πραγματικών σταθερών είναι τα ακόλουθα:

$$42.0 \quad 4.2e1 \quad 0.420e+2 \quad 42000.0e-3$$

Πίνακας Α.1: Ακολουθείες διαφυγής (escape sequences)

Ακολουθεία διαφυγής	Περιγραφή
\n	ο χαρακτήρας αλλαγής γραμμής (line feed)
\t	ο χαρακτήρας στηλοθέτησης (tab)
\r	ο χαρακτήρας επιστροφής στην αρχή της γραμμής (carriege return)
\0	ο χαρακτήρας με ASCII κωδικό 0
\\	ο χαρακτήρας \ (backslash)
\'	ο χαρακτήρας ' (απλό εισαγωγικό)
\"	ο χαρακτήρας " (διπλό εισαγωγικό)

- Τους **σταθερούς χαρακτήρες**, που αποτελούνται από ένα χαρακτήρα μέσα σε απλά εισαγωγικά. Ο χαρακτήρας αυτός μπορεί να είναι οποιοσδήποτε κοινός χαρακτήρας ή ακολουθία διαφυγής (escape sequence). Κοινοί χαρακτήρες είναι όλοι οι εκτυπώσιμοι χαρακτήρες πλην των απλών και διπλών εισαγωγικών και του χαρακτήρα \(backslash). Οι ακολουθίες διαφυγής ξεκινούν με το χαρακτήρα \(backslash) και περιγράφονται στον πίνακα Α.1. Παραδείγματα σταθερών χαρακτήρων είναι οι ακόλουθες:

  'a' '1' '@' '\n' '\''

- Τις **σταθερές συμβολοσειρές**, που αποτελούνται από μια ακολουθία κοινών χαρακτήρων ή ακολουθιών διαφυγής μέσα σε διπλά εισαγωγικά. Οι συμβολοσειρές δεν μπορούν να εκτείνονται σε περισσότερες από μια γραμμές προγράμματος. Παραδείγματα σταθερών συμβολοσειρών είναι οι ακόλουθες:

  "abc" "Route 66" "Hello world!\n"
  "Name:\t\" Douglas Adams\"nValue:\t42\n"

- Τους **τελεστές**, οι οποίοι είναι οι παρακάτω:

  `=  >  <  <>  >=  <=  +  -  *  /  ^  @`

- Τους **διαχωριστές** οι οποίοι είναι οι παρακάτω:

  `:=  ;  .  (  )  :  ,  [  ]`

Εκτός από τις λεκτικές μονάδες που προαναφέρθηκαν, ένα πρόγραμμα PCL μπορεί επίσης να περιέχει τα παρακάτω, τα οποία αγνοούνται:

- **Κενούς χαρακτήρες**, δηλαδή ακολουθίες αποτελούμενες από κενά διαστήματα (space), χαρακτήρες στηλοθέτησης (tab), χαρακτήρες αλλαγής γραμμής (line feed) ή χαρακτήρες επιστροφής στην αρχή της γραμμής (carriage return).
- **Σχόλια**, τα οποία αρχίζουν με την ακολουθία χαρακτήρων `(*` και τερματίζονται με την πρώτη μετέπειτα εμφάνιση της ακολουθίας χαρακτήρων `*)` Κατά συνέπεια τα σχόλια δεν επιτρέπετε να είναι φωλιασμένα. Στο εσωτερικό τους επιτρέπεται η εμφάνιση οποιουδήποτε χαρακτήρα.

Α.2 Τύποι δεδομένων

Η PCL υποστηρίζει τέσσερις βασικούς τύπους δεδομένων:

- `integer`: ακέραιοι αριθμοί,
- `boolean`: λογικές τιμές,
- `char`: χαρακτήρες, και
- `real`: πραγματικοί αριθμοί.

Εκτός από τους βασικού τύπους, η PCL υποστιρίζει επίσης τους παρακάτω σύνθετους τύπους, η κατασκευή των οποίων βασίζεται σε άλλους βασικού ή σύνθετους τύπους:

- `array [n] of t`: πίνακες (arrays), αποτελούμενοι από n στοιχεία τύπου t. Το n θα πρέπει να είναι ακέραια σταθερά με θετική τιμή και το t έγκυρος τύπος.
- `array of t`: πίνακες (arrays), αποτελούμενοι από άγνωστο αριθμό στοιχείων τύπου t. Το t πρέπει να είναι έγκυρος τύπος.
- `^t`: δείκτες (pointers) σε ένα στοιχείο τύπου t. Το t πρέπει να είναι έγκυρος τύπος.

Παραδείγματα σύνθετων τύπων είναι:

```
array[10] of integer
array of ^integer
^array of array [10] of booean
```

Οι τύποι `integer` και `real` ονομάζονται **αριθμητικοί** τύποι. Οι βασικοί τύποι, οι τύποι πινάκων δεδομένων μεγέθους και οι τύποι δεικτών ονομάζονται **πλήρεις** τύποι. Αντίθετα οι τύποι πινάκων άγνωστου μεγέθους ονομάζονται **ημιτελείς** τύποι. Κατά το σχηματισμό ενός τύπου πίνακα, είτε πλήρους είτε ημιτελούς, ο τύπος του στοιχείου t θα πρέπει να είναι πλήρης. Η αρίθμηση των στοιχείων ενός πίνακα ακολουθεί τη σύμβαση της C: το πρώτο στοιχείο έχει δείκτη 0, το δεύτερο 1, ενώ το τελευταίο έχει δείκτη κατά ένα μικρότερο από την πραγματική διάσταση του πίνακα.

Ο αριθμός των bytes που καταλαμβάνουν τα δεδομένα κάθε τύπου στην μνήμη του υπολγιστή, καθώς και ο ακριβής τρόπος παράστασης αυτών εξαρτώνται από την υλοποίηση της PCL. Στο πλαίσιο του παρόντος πονήματος κάνουμε τις ακόλουθες παραδοχές που διευκολίνουν την κατασκευή μεταγλωτιστών της PCL για υπολογιστές βασισμένους στον επεξεργαστή 8086 της Intel, χρησιμοποιώντας το μοντέλο προγραμμάτων `.COM`.

- `integer`: μέγεθος 2 bytes, παράσταση συμπληρώματος ως προς 2.
- `boolean`: μέγεθος 1 byte, τιμές `false` $= 0$ και `true` $= 1$.
- `char`: μέγεθος ένα byte, παράσταση σύμφωνα με τον πίνακα ASCII.
- `real`: μέγεθος 10 bytes, παράσταση σε μορφή διπλής επεκτεταμένης ακριβείας (double extended precision) σύμφωνα με το πρώτυπο IEEE 754.
- `array [n] of t`: μέγεθος ίσον με n επί το μέγεθος του τύπου t, τοποθέτηση σε αύξουσα σειρά δειυθύνσεων.
- `array of t`: οι τύποι αυτοί είναι ημιτελείς και η χρήση τους επιτρέπεται μόνο σε δείκτες ή παραμέτους κατ' αναφορά, οπότε παριστάνονται με την διεύθυνση του πρώτου στοιχείου ενός πίνακα και το μέγεθός τους είναι 2 bytes.
- `^t`: μέγεθος 2 bytes.

Α.3 Δομή του προγράμματος

Ένα πρόγραμμα PCL αποτελείται από την επικεφαλίδα και το σώμα της κύρια δομικής μονάδας. Η επικεφαλίδα αυτής είναι της μορφής:

```
program p;
```

όπου `p` το όνομα του προγράμματος. Το σώμα κάθε δομικής μονάδας μπορεί να περιέχει προαιρετικά:

- Δηλώσεις ετικετών.
- Δηλώσεις μεταβλητών.
- Ορισμούς υποπρογραμμάτων.
- Δηλώσεις υποπρογραμμάτων, οι ορισμοί των οποίων θα ακολουθήσουν.

Τελευταίο συστατικό του σώματος μιας δομικής μονάδας είναι μία σύνθετη εντολή, η οποία καθορίζει την λειτουργία της δομικής μονάδας. Στην περίπτωση της κυρίας δομικής μονάδας, η εκτέλεση του προγράμματος ξεκινά από αυτή τη σύνθετη εντολή.

Η PCL ακολουθεί τους κανόνες εμβέλειας της ISO Pascal, όσον αφορά στην ορατότητα των ονομάτων μεταβλητών, υποπρογραμμάτων και παραμέτρων.

Α.3.1 Μεταβλητές

Οι δηλώσεις μεταβλητών γίνονται με την λέξη κλειδί `var`. Ακολουθούν ένα ή περισσότερα ονόματα μεταβλητών και ένας τύπος δεδομένων, ο οποίος πρέπει να είναι πλήρης. Περισσότερες συνεχόμενες δηλώσεις μεταβλητών μπορούν να γίνουν παραλείποντας την λέξη κλειδί `var`. Παραδείγματα δηλώσεων είναι:

```
var i     : integer;
    x, y  : real;
var s     : array [80] of char;
```

Α.3.2 Υποπρογράμματα

Τα υποπρογράμματα διακρίνονται σε **διαδικασίες** (procedures) και **συναρτήσεις** (functions). Κάθε υποπρόγραμμα είναι μία δομική μονάδα και αποτελείται από την επικεφαλίδα του και το σώμα του. Η δομή του σώματος εχει ήδη περιγραφεί. Στην επικεφαλίδα αναφέρεται κατ' αρχήν αν το υπρόγραμμα είναι διαδικάσία ή συνάρτηση, το όνομά του, η τυπικές του παράμετροι μέσα σε παρενθέσεις και, αν πρόκειτα για συνάρτηση, ο τύπος του αποτελέσματος ο οποίος πρέπει να είναι πλήρης. Οι παρενθέσεις είναι υποχρεωτιοκές ακόμα και αν ένα υποπρόγραμμα δεν έχει τυπικές παραμέτρους.

Κάθε τυπική παράμετρος χαρακτηρίζεται από το όνομά της, τον τύπο της και τον τρόπο περάσματος. Η PCL υποστηρίζει πέρασμα παραμέτων κατ' αξία (by value) και κατ' αναφορά (by reference). Αν στη δήλωση μίας τυπικής παραμέτρου έχει προηγηθεί η λέξη κλειδί `var`, τότε αυτή περνά κατ' αναφορά, διαφορετικά περνά κατ' αξία. Οι τύποι των τυπικών παραμέτρων που περνούν κατ' αξία πρέπει να είναι πλήρεις.

Ακολουθούν παραδείγματα επικεφαλίδων συναρτήσεων.

```
procedure p1 ();
procedure p2 (n : integer);
procedure p3 (a, b : integer; var b : boolean);
funtion f1 (x : real) : real;
funtion f2 (var s : array of char) : integer;
funtion f3 (x : real) : array [10] of real;
funtion f4 (n : integer; x : real) : ^array of real;
```

Στην περίπτωση αμοιβαία αναδρομικών υποπρογραμμάτων, το όνομα ενός υποπρογράμματος χρειάζεται να εμφανιστεί πριν τον ορισμό του. Στην περίπτωση αυτή, για να μην παραβιαστούν οι κανόνες εμβέλειας, πρέπει να έχει προηγηθεί μία δήλωση της επικεφαλίδας αυτού του υποπρογράμματος, χωρίς το σώμα του. Αυτό γίνεται με την λέξη κλειδί `forward`.

Α.4 Εκφράσεις

Κάθε έκφραση της PCL διαθέτει ένα μοναδικό τύπο[1] και μπορεί να αποτιμηθεί δίνοντας ως αποτέλεσμα μία τιμή αυτού του τύπου. Οι εκφράσεις διακρίνονται σε δύο κατηγορίες: αυτές που προκείπτουν από l-values, οι οποίες περιγράφονται στην ενότητα Α.4.1 και αυτές που προκύπτουν από r-values, που περιγαφονται στις ενότητες Α.4.2 ως Α.4.4. Τα δύο αυτά είδη τιμών έχουν πάρει το όνομά τους από την θέση τους σε μία εντολή ανάθεσης: οι l-values εμφανίζονται στο αριστερό μέλος της ανάθεσης ενώ τα r-values στο δεξιό.

Τόσο οι l-values όσο και οι r-values μπορούν να εμφανίζονται μέσα σε παρενθέσεις, που χρησιμοποιούνται για λόγους ομαδοποίησης.

Α.4.1 L-values

Οι l-values αντιπροσωπεύουν αντικείμενα που καταλαμβάνουν χώρο στην μνήμη του υπολογιστή κατά την εκτέλεση του προγράμματος και τα οποία μπορούν να περιέχουν τιμές. Τέτοια αντικείμενα είναι οι μεταβλητές, οι παράμετροι των υποπρογραμμάτων, οι μεταβλητές που κατασκευάζονται με δυναμική παραχώρηση μνήμης και οι θέσεις όπου φυλάσσονται τα αποτελέσματα των συναρτήσεων. Συγκεκριμένα:

- Το όνομα μιας μεταβλητής ή μιας παραμέτρου υποπρογράμματος είναι l-value και αντιστοιχεί στο εν λόγω αντικείμενο. Ο τύπος της l-value είναι ο τύπος του αντίστοιχου αντικειμένου.

[1] Εξαίρεση αποτελεί η σταθερά μηδενικού δείκτη *nil*, η οποία δεν έχει μοναδικό τύπο. Περιγαφεται στην ενότητα Α.4.2

- Οι σταθερές συμβολοσειρές είναι l-values. Έχουν τύπο `array [n] of char` όπου n είναι ο αριθμός χαρακτήρων που περιέχονται στην συμβολοσειρά προσαυξημένος κατά ένα. Κάθε τέτοια l-value αντιστοιχεί σε ένα αντικείμενο τύπου πίνακα όπου βρίσκονται αποθηκευμένοι με τη σειρά οι χαρακτήρες της συμβολοσειράς. Στο τέλος του πίνακα αποθηκεύεται αυτόματα ο χαρακτήρας `'\0'`, σύμφωνα με τη σύμβαση που ακολουθεί η γλώσσα C για τις συμβολοσειρές. Οι σταθερές συμβολοσειρές είναι το μόνο είδος σταθεράς τύπου πίνακα που επιτρέπεται στην PCL.

- Αν l είναι μία l-value τύπου `array [n] of t` ή τύπου `array of t` και e είναι μία έκφραση τύπου `integer`, τότε $l[e]$ είναι μία l-value με τύπο t. Αν η τιμή της έκφρασης e είναι ο μη αρνητικός ακέραιος n τότε αυτή η l-value αντιστοιχεί στο στοιχείο με δείκτη n του πίνακα που αντιστοιχεί στην l. Η τιμή του n δεν πρέπει να υπερβαίνει τα πραγματικά όρια του πίνακα, ο έλεγχος αυτός όμως επαφίεται στον προγραμματιστή.

- Αν e είναι μία έκφραση τύπου `^t` τότε `e^` είναι μία l-value με τύπο t, που αντιστοιχεί στο αντικείμενο όπου δείχνει η τιμή της e.

- Στο σώμα μιας συνάρτησης που επιστρέφει αποτέλεσμα τύπου t, η λέξη κλειδί `result` είναι μία l-value με τύπο t και αντιστοιχεί στο αντικείμενο όπου φυλάσσεται το αποτέλεσμα της συνάρτησης. Το αντικείμενο αυτό είναι προσωρινό και δεν πρέπει να χρησιμοποιείται μετά την επιστροφή από την συνάρτηση.

Αν μία l-value χρησιμοποιηθεί ως έκφραση, η τιμή αυτής της έκφρασης είναι ίση με την τιμή η οποία περιέχεται στο αντικείμενο που αντιστοιχεί στη l-value.

Α.4.2 Σταθερές

Στις r-values της γλώσσας PCL συγκαταλέγονται οι ακόλουθες σταθερές:

- Οι ακέραιες σταθερές χωρίς πρόσημο, όπως περιγράφονται στην ενότητα Α.1. Έχουν τύπο `integer` και η τιμή τους με τον μη αρνητικό ακέραιο αριθμό που παριστάνουν.

- Οι λογικές σταθερές `true` και `false`, με τύπο `boolean` και προφανείς τιμές.

- Οι πραγματικές σταθερές χωρίς πρόσημο, όπως περιγαφονται στην ενότητα Α.1. Έχουν τύπο `char` και η τιμή τους είναι ίση με τον χαρακτήρα που παριστάνουν.

- Η λέξη κλειδί `nil` που έχει τύπο `^t`, για κάθε έγκυρο τύπο t, και η τιμή της οποίας είναι ο μηδενικό δείκτης. Είναι η μόνη έκφραση της PCL που δεν έχει μοναδικό τύπο. Ο μηδενικός δείκτης απαγορεύεται να αποδεικτοδοτηθεί με χρήση του τελεστή `^`.

Α.4.3 Τελεστές

Οι τελεστές της PCL διακρίνονται σε τελεστές με ένα τελούμενο και τελεστές με δύο τελούμενα. Από τους πρώτους, ορισμένου γράφονται πριν το τελούμενο (prefix) και ορισμένοι μετά (postfix), ενώ οι δεύτεροι γράφονται πάντα μεταξύ των τελούμενων (infix). Η αποτίμηση των τελουμένων των τελεστών με δύο τελούμενα γίνεται από αριστερά προς τα δεξιά.

Ήδη στην ενότητα Α.4.1 περιγράφηκε ο τελεστής αναφοράς σε στοιχείο πίνακα, η σύνταξη του οποίου αποτελεί εξαίρεση στα παραπάνω, και ο τελεστής αποδεικτοδότησης `^`. Οι δύο αυτοί τελεστές είναι οι μοναδικοί που έχουν ως αποτέλεσμα l-value. Στη συνέχεια περιγράφονται οι υπόλοιποι τελεστές της PCL, που έχουν ως αποτέλεσμα r-value.

- Ο τελεστής `@` επιστρέφει την διεύθυνση ενός αντικειμένου. Αν l είναι μία l-value τύπου t, τότε `@l` είναι μία r-value τύπου `^t`. Η τιμή της είναι η διεύθυνση του αντικειμένου που αντιστοιχεί στην l και είναι είναι πάντα διαφορετική του μηδενικού δείκτη.

- Οι τελεστές με ένα τελούμενο `+` και `-` υλοποιούν τους τελεστές προσήμου. Το τελούμενο πρέπει να είναι αριθμητικού τύπου και το αποτέλεσμα είναι του ίδιου τύπου με το τελούμενο.

- Ο τελεστής `not` υλοποιεί τη λογική άρνηση. Το τελούμενο του πρέπει τύπου `boolean` και τον ίδιο τύπο έχει και το αποτέλεσμα.

- Οι τελεστές με δύο τελούμενα `+`, `-`, `*`, `/`, `div` και `mod` υλοποιούν αντίστοιχα τις αριθμητικες πράξεις της πρόσθεσης, της αφαίρεσης, του πολλαπλασιάσμού, της πραγματικής διαίρεσης, του πηλοίκου και του υπολοίπου της ακέραιας διαίρεσης.Τα τελούμενά τους πρέπει να είναι εκφράσεις αριθμητιων τύπων. Στην περίπτωση των τελεστών `+`, `-` και `*`, αν και τα δύο τελούμενα είναι τύπου `integer` τότε και το αποτέλεσμα είναι τύπου `integer`, διαφορετικά το αποτέλεσμα είναι τύπου `real`. Στην περίπτωση του τελεστή `/`, το αποτέλεσμα είναι πάντα τύπου `real`. Τέλος στην περίπτωση των τελεστών `div` και `mod`, τα τελούμενα πρέπει να είναι τύπου `integer` και το αποτέλεσμα είναι τύπου `integer`.

- Οι τελεστές `=` και `<>` υλοποιούν αντίστοιχα την ισότητα και την ανισότητα. Το αποτέλεσμα είναι τύπου `boolean`. Τα τελούμενα πρέπει να

είναι είτε και τα δύο αριθμητικά, οπότε συγκρίνονται οι αριθμητικές τιμές τους, είτε του ίδιου τύπου, ο οποίος δεν πρέπει να είναι τύπος πίνακα. Στη δεύτερη περίπτωση, συγκρίνονται οι δυαδικές αναπαραστάσεις των τιμών των τελουμένων.

- Οι τελεστές `<`, `>`, `<=` και `>=` υλοποιούν τις σχέσεις ανισότητας. Τα τελούμενα πρέπει να είναι και τα δύο αριθμητικά ή και τα δύο τύπου `char`. Το αποτέλεσμα είναι τύπου `boolean`. Η σύγκριση μεταξύ των χαρακτήρων γίνεται βάσει των κωδικών στον πίνακα ASCII.
- Οι τελεστές `and` και `or` υλοποιούν αντίστοιχα τις πράξεις της λογικής σύξευξης και διάζευξης. Τα τελούμενα πρέπει να είναι τύπου `boolean` και τον ίδιο τύπο έχει και το αποτέλεσμα. Η αποτίμηση εκφράσεων που χρησιμοποιούν αυτούς τους τελεστές γίνεται με **βραχυκύκλωση** (short-circuit). Δηλαδή, αν το αποτέλεσμα της έκφρασης είναι γνωστό από την αποτίμηση και μόνο του πρώτου τελούμενου, το δεύτερο δεν αποτιμάται καθόλου.

Στον πίνακα Α.2 ορίζεται η προτεραιότητα και η προσεταιριστικότητα των τελεστών της PCL.

Πίνακας Α.2: Προτεραιότητα και προσεταιριστικότητα των τελεστών της PCL.

Τελεστές	Περιγραφή	Αριθμός τελουμένων	Θέση και προσεταιριστικότητα
[ ]	Αναφορά σε στοιχείο πίνακα	2	ειδική
@	Διεύθυνση αντικειμένου	1	prefix
^	Αποδεικτοδότηση	1	prefix
+ -	Πρόσημα	1	prefix
not	Λογική άρνηση	1	prefix
* / div mod and	Πολλαπλασιαστοικοί τελεστές	2	infix, αριστερή
+ - or	Προσθετικοί τελεστές	2	infix, αριστερή
= > < <= >= <>	Σχεσιακοί τελεστές	2	infix, καμία

Α.4.4 Κλήση συναρτήσεων

Αν f είναι το όνομα μιας συνάρτησης με αποτέλεσμα τύπου t, τότε η έκφραση $f(e_1, ..., e_n)$ είναι μία r-value με τύπο t. Ο αριθμός των πραγματικών παραμέτρων n πρέπει να συμπίπτει με τον αριθμό των τυπικών παραμέτρων της f.

Επίσης ο τύπος και το είδος κάθε πραγματικής παραμέτρου πρέπει να συμπίπτει με τον τύπο και τον τρόπο περάσματος της αντίστοιχης τυπικής παραμέτρου, σύμφωνα με τους παρακάτω κανόνες. Η έννοια της **συμβατότητας για ανάθεση** περιγράφεται στην ενότητα Α.5.

- Αν η τυπική παράμετρος είναι τύπου t και περνά κατ' αξία, τότε ο τύπος t' της αντίστοιχης πραγματικής παραμέτρου πρέπει να είναι συμβατός για ανάθεση στον τύπο t.
- Αν η τυπική παράμετρος είναι τύπου t και περνά κατ' αναφορά, τότε η αντίστοιχη πραγματική παράμετρος πρέπει να είναι l-value τύπου t', όπου ο τύπος `^t'` πρέπει να είναι συμβατός για ανάθεση στον τύπο `^t`.

Κατά την κλήση μίας συνάρτησης, οι πραγματικές παράμετροι αποτιμώνται από αριστερά προς τα δεξιά.

Α.5 Εντολές

Οι εντολές που υποστιρίζει η γλώσσα PCL είναι οι ακόλουθες:

- Η κενή εντολή, που δεν κάνει καμία ενέργεια.
- Η εντολή ανάθεσης `l := e`, όπου l μία l-value τύπου t και e μία έκφραση τύπου t'. Ο τύπος t' πρέπει να είναι **συμβατός για ανάθεση** στον τύπο t. Αυτή η σχέση συμβατότητας, που πρέπει να τονιστέι ότι δεν είναι συμμετρική ορίζετια ως εξής:
  - Κάθε πλήρης τύπος είναι συμβατός για ανάθεση στον εαυτό του.
  - Ο τύπος `integer` είναι συμβατός για ανάθεση στον τύπο `real`.
  - Ο τύπος `^array [n] of t` είναι συμβατός για ανάθεση στον τύπο `^array of t`.
- Η σύνθετη εντολή, που αποτελείται από μία σειρά έγκυρων εντολών χωρισμένων με το διαχωριστεί `;` ανάμεσα στις λέξεις `begin` και `end`. Οι εντολές αυτές εκτελούνται διαδοχικά, εκτός αν κάποια από αυτές είναι εντολή άλματος.
- Η εντολή ελέγχου `if` e `then` s_1 `else` s_2. Η έκφραση e πρέπει να έχει τύπο `boolean` και τα s_1, s_2 να είναι έγκυρες εντολές. Το τμήμα `else` είναι προαιρετικό.
- Η εντολή βρόχου `while` e `do` s. Η έκφραση e πρέπει να έχει τύπο `boolean` και το s να είναι έγκυρη εντολή

- Η εντολή με ετικέτα I : s, όπου I το όνομα μιας ετικέτας και s μια έγκυρη εντολή. Κάθε ετικέτα πρέπει να ορίζεται το πολύ μια φόρα στην συνθετη εντολή που ορίζει το σώμα μιας συγκεκριμένης δομικής μονάδας

- Η εντοη αλματος `goto` I, όπου I το όνομα μιας ετικέτας που πρέπει να εμφανίζεται στην ίδια δομική μονάδα. Πέραν αυτού, δεν υπάρχουν αλλοι περιορισμοί ως προς την θέση των εντολών αλμάτων ή των ετικετών όπου αυτα οδηγούν.

- Η εντολή `return`, που επιστρέφει τερματίζοντας την εκτέλεση της δομικής μοναδας.

- Η κλήση μιας διαδικασίας. Συντακτικά και σημασιολογικά συμπιπτει με την κληση μιας συνάρτησης, με τη διαφορά ότι δεν επιστρέφεται αποτέλεσμα.

- Η εντολη `new` με την οποία γίνεται δυναμική παραχώρηση μνήμης, και η οποία εμφανίζεται σε δυο μορφές:

  - `new` l, όπου l πρέπει να είναι μία l-value τύπου `^`t και t πρέπει να είναι πλήρης τύπος. Μετά την εκτέλεση της εντολής, ο δείκτης που αντιστοιχεί στην l τοποθετείται να δείχνει προς ένα νεό δυναμικό αντικείμενο τύπου t.
  - `new [`e`]` l, όπου l πρέπει να είναι μία l-value `^array of` t και e μία έκφραση τύπου `integer`. Η τιμη της e πρέπει να είναι ένας θετικός αριθμός n. Μετά την εκτέλεση της εντολής, ο δείκτης που αντιστοιχεί στην l τοποθετείται να δείχνει προς ένα νέο δυναμικό αντικείμενο τύπου `array [`n`] of` t.

- Η εντολή `dispose` με την οποία γίνεται η αποδεύσμευση της μνήμης που έχει παραχωρηθή δυναμικά με την ετνολή `new`. Εμφανίζεται και αυτή σε δύο μοφες:

  - `dispose` l, όπου l πρέπει να είναι μία l-value τύπου `^`t και t πρέπει να είναι πλήρης τύπος. Η τρέχουσα τιμή της l πρέπει να είναι ένας δείκτης προς ένα δυναμικό αντικείμενο που έχει κατασκευαστεί με χρήση της εντολής `new`. Μετά την εκτέλεση της εντολής, η l περιέχει τον μηδενικό δείκτη.
  - `dispose []` l, όπου l πρέπει να είναι μία l-value τύπου `^array of` t. Η τρέχουσα τιμή της l πρέπει να είναι ένας δείκτης προς ένας δείκτης προς ένα δυναμικό αντικείμενο που έχει κατασκευαστεί με χρήση της εντολής `new`. Μετά την εκτέλεση της εντολής, η l περιέχει τον μηδενικό δείκτη.

Α.6 Βιβλιοθήκη χρόνου εκτέλεσης

Η βιβλιοθήκη χρόνου εκτέλεσης (run-time library) της PCL υποστηρίζει ένα σύνολο προκαθορισμένων υποπρογραμμάτων, τα οποία βρίσκονται στην διάθεση του προγραμματιστή. Τα υποπρογράμματα αυτά είναι ορατά σε κάθε δομική μονάδα, εκτός αν επισκιάζονται από μεταβλητές, παραμέτρους ή υποπρογράμματα με το ίδιο όνομα. Παρακάτω δίνονται οι επικεφαλίδες τους, όπως θα γράφονταν αν τα ορίζαμε σε ένα πρόγραμμα PCL. Επίσης εξηγείται η λειτουργία τους.

Α.6.1 Είσοδος και έξοδος

```
procedure writeInteger(n : integer);
procedure writeBoolean(b : boolean);
procedure writeChar   (c : char);
procedure writeReal   (r : real);
procedure writeString (var s : array of char);
```

Οι διαδικασίες αυτές χρησιμοποιούνται για την εκτύπωση τιμών που ανοίκουν στους βασικούς τύπους της PCL, καθώς και για την εκτύπωση συμβολοσειρών.

```
function  readInteger() : integer;
function  readBoolean() : boolean;
function  readChar   () : char;
function  readReal   () : real;
procedure readString (size : integer; var s : array of
   char);
```

Αντίστοιχα τα παραπάνω υποπρογράμματα χρησιμοποιούνταιο για την εισαγωγή τιμών που ανήκουν στους βασικούς τύπους της PCL και για την εισαγωγή συμβολοσειρών. Η διαδικασία `readString` χρησιμοποιείται για την ανάγνωση μιας συμβολοσειράς μέχρι τον επόμενο χαρακτήρα αλλαγής γραμμής. Οι παράμετροί της καθορίζουν το μέγιστο αριθμό χαρατήρων (συμπεριλαμβανομένου του τελικού `'\0'`) που επιτρέπεται να διαβαστούν και τον πίνακα χαρακτήρων στον οποίο αυτοί θα τοποθετηθούν. Ο χαρακτήρας αλλαγής γραμμής δεν αποθηκεύεται. Αν το μέγεθος του πίνακα εξαντληθεί πριν συναντηθεί χαρακτήρας αλλαγής γραμμής, η ανάγνωση θα συνεχιστέι αργότερα από το σημείο όπου διακόπηκε.

Α.6.2 Μαθηματικές συναρτήσεις

```
function abs (n : integer) : integer;
function fabs(r : real)    : real;
```

Η απόλυτη τιμή ενός ακεραίου ή πραγματικού αριθμού.

```
function sqrt  (r : real) : real;
function sin   (r : real) : real;
function cos   (r : real) : real;
function tan   (r : real) : real;
function arctan(r : real) : real;
function exp   (r : real) : real;
function ln    (r : real) : real;
function pi    ()         : real;
```

Βασικές μαθηματικές συναρτήσεις: τετραγωγνική ρίζα, τριγωνομετρικές συναρτήσεις, εκθετική συνάρτηση, φυσικός λογάριθμος, ο αριθμός π.

Α.6.3 Συναρτήσεις μετατροπής

```
function trunc(r : real) : integer;
function round(r : real) : integer;
```

Η `trunc` επιστρέφει τον πλησιέστερο ακέραιο αριθμό, η απόλυτη τιμή του οποίου είναι μικρότερη από την απόλυτη τιμή του r. Η `round` επιστρέφει το πλησιέστερο ακέραιο αριθμό. Σε περίπτωση αμφιβολίας, προτιμάται ο αριθμός με την μεγαλύτερη απόλυτη τιμή.

```
function ord(c : char)    : integer;
function chr(n : integer) : char;
```

Μετατρέπουν από ένα χαρακτήρα στον αντίστοιχο κωδικό ASCII και αντίστροφα.

Α.7 Πλήρης γραμματική της PCL

Η σύνταξη της γλώσσας PCL δίνεται παρακάτω σε μορφή EBNF. Η γραμματική που ακολουθεί είναι **διφορούμενη**, οι αμφισημίες όμως μπορούν να ξεπεραστούν αν λάβει κανείς υπόψη τους κανόντες προτεραιότητας και προσεταιριστικότητας των τελεστών όπως περιγράφονται στην ενότητα Α.4.3. Τα σύμβολα `id`, `integer-const`, `real-const`, `char-const` και `string-literal` είναι τερματικά σύμβολα της γραμματικής.

$< program >$ ::= “**program**” $< id >$ “**;**” $< body >$ “**.**”
$< body >$ ::= $(< local >)^* < block >$
$< local >$ ::= “**var**” $(< id > (\text{“,”} < id >)^* \text{“:”} < type > \text{“;”})^+$
| “**label**” $< id > (\text{“,”} < id >)^* \text{“;”}$
| $< header > \text{“;”} < body > \text{“;”}$
| “**forward**” $< header > \text{“;”}$
$< header >$ ::= “**procedure**” $< id >$ “(” $[< formal > (\text{“,”} < formal >)^*]$ “)”
| “**function**” $< id >$ “(” $[< formal > (\text{“,”} < formal >)^*]$ “)” “**:**” $< type >$
$< formal >$ ::= [“**var**”] $< id > (\text{“,”} < id >)^* \text{“:”} < type >$
$< type >$ ::= “**integer**” | “**real**” | “**boolean**” | “**char**”
| “**array**” [“[” $< integer - const >$ “]”] “**of**” $< type >$ | “^” $< type >$
$< block >$ ::= “**begin**” $< stmt > (\text{“;”} < stmt >)^*$ “**end**”
$< stmt >$ ::= ε | $< l - value >$ “**:=**” $< expr >$ | $< block >$ | $< call >$
| “**if**” $< expr >$ “**then**” $< stmt >$ [“**else**” $< stmt >$]
| “**while**” $< expr >$ “**do**” $< stmt >$
| $< id >$ “**:**” $< stmt >$ | “**goto**” $< id >$ | “**return**”
| “**new**” [“[” $< expr >$ “]”] $< l - value >$
| “**dispose**” [“[” “]”] $< l - value >$
$< expr >$ ::= $< l - value >$ | $< r - value >$
$< l - value >$ ::= $< id >$ | “**result**” | $< string - literal >$ | $< l - value >$ “[” $< expr >$ “]”
| $< expr >$ “^” | “(” $< l - value >$ “)”
$< r - value >$ ::= $< integer - const >$ | “**true**” | “**false**” | $< real - const >$
| $< char - const >$ | “**nil**” | “(” $< r - value >$ “)” | $< call >$
| “**@**” $< lvalue >$ | $< unop >< expr >$ | $< expr >< binop >< expr >$
$< call >$ ::= $< id >$ “(” $[< expr > (\text{“,”} < expr >)^*]$ “)”
$< unop >$ ::= “**not**” | “+” | “−”
$< binop >$ ::= “+” | “−” | “∗” | “/” | “**div**” | “**mod**” | “**or**”
| “**and**” | “=” | “<>” | “<” | “<=” | “>” | “>=”

Παράθεμα Β

Μεταγλωττιστής για την Γλώσσα PCL

Ο πειραματικός Μεταγλωττιστής για την Γλώσσα PCL δέχεται στην είσοδό του προγράμματα γραμμένα στην Γλώσσα PCL και εξάγει τα ίδια προγράμματα σε Γλώσσα assembly x86, όπως αυτή υλοποιείται από το συμβολομεταφραστή MASM της Microsoft®.

Στην συνέχεια παρουσιάζεται η δομή του πειραματικού μεταγλωττιστή για την Γλώσσα PCL. Ακολουθεί μία περιγραφή του τρόπου κλήσης του μεταγλωττιστή. Το παράρτημα ολοκληρώνεται με την παρουσίαση παραδειγμάτων προγραμμάτων, γραμμένων σε PCL.

Β.1 Η δομή του Μεταγλωττιστή

Για την κατασκευή των δύο πρώτων τμημάτων του Μεταγλωττιστή, του Λεξικού και του Συντακτικού Αναλυτή, χρησιμοποιήθηκαν τα εργαλεία flex και bison, αντίστοιχα. Στον φάκελο bison, που υλοποιεί την Συντακτική Ανάλυση, προστέθηκαν, σε κατάλληλα σημεία, τμήματα κώδικα που υλοποιούν την Σημαντική Ανάλυση και την παράλληλη συμπλήρωση του Πίνακα Συμβόλων. Στην συνέχεια υλοποιήθηκαν οι ομάδες συναρτήσεων που παράγουν τον Ενδιάμεσο και τον Τελικό Κώδικα, κλήσεις των οποίων τοποθετήθηκαν σε συγκεκριμένα σημεία του φακέλου bison.

Γίνεται προφανές, από την παραπάνω περιγραφή, πως το τμήμα του Μεταγλωττιστή που οδηγεί την διαδικασία της μεταγλώττισης, το δεσπόζον τμήμα του Μεταγλωττιστή, δεν είναι άλλο από τον Συντακτικό Αναλυτή. Επιπλέον ο Πίνακας Συμβόλων υλοποιήθηκε και αυτός, ως μία ομάδα συναρτήσεων που συνοδεύεται από τις ανάλογες δομές δεδομένων.

Τόσο ο παραγόμενος κώδικας από τα μεταεργαλεία flex και bison, όσο και ο κώδικας που γράφτηκε για την υλοποίηση των υπόλοιπων τμημάτων του Μεταγλωττιστή, γράφτηκαν σε ANSI C.

Β.1.1 Λεξικός Αναλυτής

Ο Λεξικός Αναλυτής υλοποιείται ως ένας φάκελος (file)[1] flex με όνομα **lex_pcl.l**. Ο φάκελος αυτός δίνεται ως είσοδος στον flex από τον οποίο παίρνουμε την υλοποίηση του Λεξικού Αναλυτή ως ο φάκελος C με όνομα **lex_pcl.c**.

Β.1.2 Συντακτικός Αναλυτής

Ο Συντακτικός Αναλυτής υλοποιείται ως ένας φάκελος bison με όνομα **parse_pcl.y** ο οποίος δίνεται ως είσοδος στον bison και εξάγονται οι φάκελοι **parse_pcl.h**

[1] Στο παρόν πόνημα χρησιμοποιήθηκε η ορθολογικότερη μετάφραση των όρων: file → φάκελος, directory → κατάλογος, archive → αρχείο.

και **parse_pcl.c**.

Β.1.3 Σημαντικός Αναλυτής

Ο Σημαντικός[2] Αναλυτής υλοποιείται εν μέρη με την βοήθεια του bison και κατά κύριο λόγο, από μία σειρά συναρτήσεων που βρίσκονται στους φακέλους **SemanticAnalysis.h** και **SemanticAnalysis.c** και καλούνται σε κατάλληλα σημεία της Συντακτικής Ανάλυσης.

Β.1.4 Πίνακας Συμβόλων

Ο Πίνακας Συμβόλων χρησιμοποιείται σχεδόν από όλα τα τμήματα του Μεταγλωττιστή και υλοποιείται από τα τρία ζεύγη σχετικών φακέλων **SymbolTable.h**, **SymbolTable.c**, **HashTable.h**, **HashTable.c**, **Stack.h** και **Stack.c**.

Β.1.5 Παραγωγή Ενδιάμεσου Κώδικα

Η Παραγωγή Ενδιάμεσου Κώδικα υλοποιείται από μία σειρά συναρτήσεων οι οποίες καλούνται σε κατάλληλα σημεία της Συντακτικής Ανάλυσης και βρίσκονται στους φακέλους **intermediateCodeGenerator.h** και **intermediateCodeGenerator.c**.

Β.1.6 Παραγωγή Τελικού Κώδικα

Η Παραγωγή Τελικού Κώδικα υλοποιείται από μία σειρά συναρτήσεων που καλούνται στο τέλος κάθε block κώδικα της πηγαίας Γλώσσας και βρίσκονται στους φακέλους **x86CodeGenerator.h** και **x86CodeGenerator.c**.

Β.1.7 Αναφορά Σφαλμάτων

Η Αναφορά Σφαλμάτων υλοποιείται ως μία συνάρτηση της Γλώσσας C, η οποία αναφέρει τόσο το μήνυμα του σφάλματος, όσο και τον αριθμό της γραμμής, όπου εντοπίστηκε το σφάλμα στο πρόγραμμα που μεταγλωττίζεται.

[2] Στο παρόν πόνημα χρησιμοποιήθηκε η μετάφραση του όρου **semantic** → **Σημαντικός/Σημαντική** αντί του ευρέως χρησιμοποιούμενου **σημασιολογική**, που είναι μεν συγγενείς αλλά όχι ακριβείς.

Β.2 Παραγωγή εκτελεσίμου κώδικα Μεταγλωττιστή

Η παραγωγή του εκτελέσιμου κώδικα του Μεταγλωττιστή γίνεται με την εκτέλεση του φακέλου **BuildCompiler.bat** ο οποίος αποτελείται από μία αλληλουχία εκτέλεσης των μεταεργαλείων flex και bison και στην συνέχεια την μεταγλώττιση και σύνδεση των επιμέρους τμημάτων του Μεταγλωττιστή.

pcl.l]../compiler/FinalCodeProduction/SyntacticaAndSemanticalAnalyser/syntax.y

Β.3 Κλήσεις στην γραμμή εντολών

Ο Μεταγλωττιστής της PCL καλείται με τρεις τρόπους:

```
<prompt>pcl[.exe] program.pcl ↲
<prompt>pcl[.exe] ↲
<prompt>cat program.pcl | pcl[.exe] ↲
```

Στον πρώτο δίνεται ως μοναδική παράμετρος το όνομα του φακέλου με τον πηγαίο κώδικα. Στον δεύτερο ο pcl καλείται χωρίς παραμέτρους οπότε το πρόγραμμα διαβάζεται από την τυπική είσοδο (standard input), που είναι το πληκτρολόγιο και έχει EOF το CTRL-Z στα Windows/DOS και CTRL-D στα LINUX/UNIX. Στην τρίτη περίπτωση ο pcl διαβάσει τον πηγαίο κώδικα επίσης από την καθιερωμένη είσοδο που είναι, αυτή τη φορά, το ρεύμα των δεδομένων που περιέχει ο φάκελος program.pcl (pipeline). Σε κάθε περίπτωση δημιουργούνται δύο νέοι φάκελοι, ο ένας με κατάληξη **.imm**, που περιέχει τον Ενδιάμεσο Κώδικα και ο δεύτερος με κατάληξη **.asm** που περιέχει τον Τελικό x86 Κώδικα. Το όνομα των φακέλων αυτών δημιουργείται από το βασικό όνομα (όνομα χωρίς προέκταμα) και τα προεκτάματα (extensions) `.imm`, `.asm` ή, στην περίπτωση που χρησιμοποιείται η τυπική είσοδος, το **a**.

Για παράδειγμα αν το πρόγραμμα που πρόκειται να μεταγλωττιστή βρίσκεται στον φάκελο **program.pcl** τότε στην πρώτη περίπτωση καλούμε των Μεταγλωττιστή με την εντολή:

```
<prompt>pcl[.exe] program.pcl ↲
```

και παράγονται οι φάκελοι **program.imm** με τον Ενδιάμεσο Κώδικα και **program.asm** με τον Τελικό x86 Κώδικα.

Αντίστοιχα στη δεύτερη και τρίτη περίπτωση χρησιμοποιούμε ένα τρίτο πρόγραμμα το οποία μπορεί να διαβάσει έναν φάκελο και να εξάγει τα περιεχόμενα του στην τυπική έξοδο την οποία διασωληνώνουμε (pipeline) με την

τυπική είσοδο του Μεταγλωττιστή. Τέτοιο πρόγραμμα είναι το cat (από την συλλογή προγραμμάτων GNU) ή το type που συνοδεύει το Microsoft®DOS και τον εξομοιωτή του που υπάρχει στα Microsoft®Windows®. Οι εντολές κλήσης είναι οι εξής:

```
<prompt>pcl[.exe] ↲
<prompt>cat program.pcl | pcl[.exe] ↲
```

και παράγονται οι φάκελοι **a.imm** με τον Ενδιάμεσο Κώδικα και **a.asm** με τον Τελικό Κώδικα.

Β.4 Παραδείγματα Προγραμμάτων PCL

Ακολουθούν μερικά αντιπροσωπευτικά παραδείγματα που παρουσιάζουν την χρήση της PCL ως Γλώσσας Προγραμματισμού. Περισσότερα παραδείγματα μπορούν να βρεθούν στον κατάλογο **examples** ο οποίος βρίσκεται στον ψηφιακό δίσκο (CD) που συνοδεύει το παρόν πόνημα.

Παράθεση Β.1: Υπολογισμός κυβικής ρίζας (qroot.pcl)

```
program qroot;
        var number: real;
    label done;

        function getNumber():real;
        begin
        result := -1.0;

        while result < 0.0 do
        begin
            writeString("\r\nGive a positive number or 0 to quit
                :");
            result := readReal();
            if result < 0.0 then
                writeString("\r\nExcepted non-negative value!\r\
                    n");
        end;
        end;

    procedure display(n, r : real);
    begin
        writeString("\r\nThe qubic root of ");
        writeReal(n);
        writeString(" is ");
        writeReal(r);
        writeString(".\r\n");
```

```
    end;

    function calculateQubicRoot(num : real) : real;
    var fault : real;

    function qube(r: real) : real;
    begin
        result := r * r * r;
    end;

    function searchRoot(num, from, to, fault : real) : real;
    var mid, test : real;
    label found;
    begin
        while true do
        begin
            mid := from + (to - from) / 2.0;

            test := qube(mid);

            if fabs(test - num) <= fault then (*we found it*)
                goto found
            else if test > num then
                to := mid
            else
                from := mid;
        end;

        found:
        result := mid;
    end;
    begin
        fault := 0.5e-4;
        result := searchRoot(num, 0.0, num, fault);
    end;
begin

    while true do
    begin
        number := getNumber();

        if number = 0.0 then
            goto done;

        display(number, calculateQubicRoot(number));
    end;

    done:
```

```
end.
```

Παράθεση Β.2: Ο αλγόριθμος Κρυπογράφησης ROT13 (rot13.pcl)

```
program rot13_program;
        var myString : ^ array of char;
    function getString(maxLength : integer) : ^array of char;
        var str : ^ array of char;
    begin
        new [maxLength + 1] str;

        writeString("Give me a sentece (maximum ");
        writeInteger(maxLength);
        writeString(" caharacters):\r\n");


        readString(maxLength, str^);

        result := str;
    end;

    procedure rot13(s : ^ array of char);
        var i : integer;
        function rot13_c(c : char) : char;
        begin
            if (c >= 'a' and c <= 'z') then
                result := chr((ord(c) - ord('a') + 13) mod 26 +
                    ord('a'))
            else if (c >= 'A' and c <= 'Z') then
                result := chr((ord(c) - ord('A') + 13) mod 26 +
                    ord('A'))
            else
                result := c;

        end;
    begin
        i := 0;

        while s^[i] <> '\0' do
        begin
            s^[i] := rot13_c(s^[i]);
            i := i + 1;
        end;

    end;
begin

    myString := getString(50);
```

```
    writeString("Original:\t");
    writeString(myString^);

    rot13(myString);
    writeString("\r\nRot13 x1:\t");
    writeString(myString^);

    rot13(myString);
    writeString("\r\nRot13 x2:\t");
    writeString(myString^);

    rot13(myString);
    writeString("\r\nRot13 x3:\t");
    writeString(myString^);

    rot13(myString);
    writeString("\r\nRot13 x4:\t");
    writeString(myString^);

    writeString("\r\n");

end.
```

Παράθεση Β.3: Μετατροπή του πρώτου γράμματος κάθε λέξης σε κεφαλαίο και των υπολοίπων σε μικρά (wordCap.pcl)

```
program wordCap;
        var myString : ^ array of char;
    function getString(maxLength : integer) : ^array of char;
        var str : ^ array of char;
    begin
        new [maxLength + 1] str;

        writeString("Give me a sentece (maximum ");
        writeInteger(maxLength);
        writeString(" caharacters):\r\n");


        readString(maxLength, str^);

        result := str;
    end;

    procedure formatWordCap(str : ^ array of char);
        var i : integer;
            newWord : boolean;

        function toLower(x : char) : char;
        begin
```

```
            if x >= 'A' and x <= 'Z' then
                result := chr(ord(x) - ord('A') + ord('a'))
            else
                result := x;
        end;

        function toUpper(x : char) : char;
        begin
            if x >= 'a' and x <= 'z' then
                result := chr(ord(x) - ord('a') + ord('A'))
            else
                result := x;
        end;
    begin
        i := 0;

        newWord := true;

        while str^[i] <> '\0' do (*Until the end ot the string*)
        begin

            if newWord then
                str^[i] := toUpper(str^[i])
            else
                str^[i] := toLower(str^[i]);

            newWord := str^[i] = '␣' or str^[i] = '.' or str^[i]
                = ',';

            i := i + 1;
        end;
    end;
begin
        myString := getString(35);

    formatWordCap(myString);

    writeString("To␣wordCap:\r\n");
    writeString(myString^);

    writeString("\r\n");
end.
```