Πανεπιστήμιο Πελοποννήσου
Σχολή Οικονομίας, Διοίκησης και Πληροφορικής
Τμήμα Πληροφορικής και Τηλεπικοινωνιών
Π.Μ.Σ. στην Επιστήμη και Τεχνολογία Υπολογιστών

Μεταπτυχιακή Εργασία

Μεταγλώττιση MATLAB σε υψηλού επιπέδου αναπαράσταση

Λαμπίρης Νικόλαος
Α.Μ.: 2022201402007

Μασσέλος Κωνσταντίνος, Δημητρουλάκος Γρηγόρης

Τρίπολη, Μάρτιος 2016

Περιεχόμενα

Εκτεταμένη περίληψη . . . 3
Extended abstract . . . 3

1 Εισαγωγή (Introduction) **4**
1.1 Μοτίβα σχεδίασης (Design patterns) . . . 5
1.2 Μεταγλωττιστές - Μεταφραστές (Compilers - Translators) . . . 6
1.3 Διερμηνευτές (Interpreters) . . . 8

2 Λεκτική ανάλυση (Lexixal analysis) **9**
2.1 Αναλυτές (Recognisers) . . . 9
2.2 Κανονικές εκφράσεις (Regular expressions) . . . 14
2.3 Από τις κανονικές εκφράσεις στο σαρωτή . . . 16
2.3.1 Ισοτιμία των NFAs και DFAs . . . 18
2.4 Υλοποίηση σαρωτών (Scanner construction) . . . 19
2.4.1 Table-Driven σαρωτές . . . 20
2.4.2 Direct-Coded σαρωτές . . . 21
2.4.3 Hand-Coded σαρωτές . . . 22

3 Συντακτική ανάλυση (Syntactical analysis) **23**
3.1 Context-Free γραμματικές . . . 24
3.1.1 Ένα πιο περίπλοκο παράδειγμα . . . 25
3.2 Top-Down Συντακτική ανάλυση (Top-Down parsing) . . . 28
3.2.1 Κατηγορίες Context-free γραμματικών . . . 28
3.3 Bottom-Up Συντακτική ανάλυση (Bottom-Up parsing) . . . 29

4 Εργαλεία/Γλώσσες που χρησιμοποιήθηκαν (Tools/Languages used) **30**
4.1 Γλώσσα προγραμματισμού C# . . . 30
4.2 Γεννήτρια συντακτικών αναλυτών ANTLR . . . 32
4.3 Εργαλείο παραγωγής γράφων Graphviz . . . 33
4.4 Γλώσσα προγραμματισμού MATLAB . . . 35

5 Κατασκευή συντακτικού αναλυτή για την γλώσσα προγραμματισμού MATLAB (MATLAB parser construction) **38**
5.1 Λεκτικός αναλυτής (Lexer) . . . 38
5.1.1 Ενέργειες υπό μορφή συναρτήσεων . . . 39

5.1.2 Semantic predicates . . . 41
5.1.3 Inline ενέργειες . . . 42
5.2 Συντακτικός αναλυτής (Parser) . . . 43
5.2.1 Δηλώσεις (Statements) . . . 44
5.2.2 Συναρτήσεις (Functions) . . . 44
5.2.3 Εκφράσεις (Expressions) . . . 46

6 Μηχανισμός διαχείρισης της υψηλού επιπέδου αναπαράστασης (HLIR mechanism) 48
6.1 Μοτίβα σχεδίασης που εφαρμόστηκαν . . . 48
6.1.1 Μοτίβο σύνθεσης (Composite pattern) . . . 49
6.1.2 Μοτίβο επισκέπτη (Visitor pattern) . . . 51
6.1.3 Μοτίβο παρατηρητή (Observer pattern) . . . 53
6.1.4 Μοτίβο επαναλήπτη (Iterator pattern) . . . 55
6.1.5 Μοτίβο εργοστάσιο μεθόδων (Factory method pattern) . . . 57
6.1.6 Μοτίβο πρόσοψης (Façade pattern) . . . 58
6.2 Πίνακας συμβόλων (Symbol table) . . . 60

7 Διάσχιση της υψηλού επιπέδου αναπαράστασης (HLIR traversal) 64
7.1 Μετατροπή ΣΣΤ σε ΑΣΤ . . . 64
7.2 Γραφική αναπαράσταση των ΣΣΤ και ΑΣΤ . . . 65
7.3 Αναγνώριση συμβόλων . . . 67

8 Συμπεράσματα - Μελλοντικές κατευθύνσεις (Conclusions - Future Directions) 69
8.1 Συμπεράσματα . . . 69
8.2 Μελλοντικές κατευθύνσεις . . . 70

Βιβλιογραφία (References) 71

Παράρτημα (Appendix) 73

Εκτεταμένη περίληψη

Στην παρούσα εργασία θα μελετήσουμε την κατασκευή ενός μεταγλωττιστή οποίος θα δέχεται ως είσοδο ένα αρχείο γραμμένο στη γλώσσα προγραμματισμού MATLAB και θα παράγει μετά από διάφορες επεξεργασίες της αρχικής εισόδου μια υψηλού επιπέδου αναπαράσταση η οποία μπορεί να χρησιμοποιηθεί μετέπειτα με σκοπό την γραφική αναπαράσταση ή την μετάφραση της γλώσσας του αρχικού προγράμματος σε μία άλλη γλώσσα προγραμματισμού.

Η διαδικασία της μεταγλώττισης έχει πολλά στάδια και ξεκινάει αρχικά με την λεκτική και συντακτική ανάλυση που πραγματοποιείται στον πηγαίο κώδικα. Στη συνέχεια όλη αυτή η πληροφορία κωδικοποιείται σε μια δομή δεντρικής αναπαράστασης, μια υψηλού επιπέδου ενδιάμεση αναπαράσταση, η οποία έχει κατασκευαστεί με διάφορα μοτίβα σχεδίασης. Μέσω αυτής της δομής στη συνέχεια μπορούμε να προσπελάσουμε την επιθυμητή πληροφορία.

Extended abstract

In this work we will study the construction of a compiler which will receive a MATLAB file as input and will produce a higher level intermediate representation that can be used in order to print graphically the source code of the input file or at a later time, after much more processing in the input, the input source can be translated to another programming language.

The process of compilation has many stages and starts initially with a lexical and a syntactical analysis which is done at the source code. Subsequently, all this information produced by the lexer and parser is encoded as a tree representation, a higher level intermediate representation, which is built with help of several design patterns. Using now this representation we can access every aspect of the input source as well as continue processing.

Κεφάλαιο 1

Εισαγωγή (Introduction)

Η μηχανική λογισμικού ή τεχνολογία λογισμικού (software engineering) ονομάζεται η τυποποιημένη και συστηματική προσέγγιση στην ανάλυση, σχεδίαση, υλοποίηση και συντήρηση λογισμικού.[1] Ένας τυπικός ορισμός είναι 'η έρευνα, σχεδίαση, ανάπτυξη και δοκιμή λογισμικού στο επίπεδο του λειτουργικού συστήματος, μεταγλωττιστών (compilers), λογισμικού διανομής δικτύων για ιατρικές, βιομηχανικές, στρατιωτικές, επικοινωνιακές, διαστημικές, επιχειρησιακές, επιστημονικές και γενικού σκοπού εφαρμογές'.

Στις αρχές του 1940 έκαναν την εμφάνιση τους οι πρώτοι υπολογιστές[2]. Οι εντολές που ήταν απαραίτητες για τη λειτουργία τους ήταν καταχωρημένες στο ίδιο το σύστημα (hard-coded) και δεν ήταν δυνατό να μεταβληθούν. Ως εκ τούτου, έγινε αντιληπτό ότι ο σχεδιασμός δεν ήταν ευέλικτος και έτσι εμφανίστηκε η λογική της αρχιτεκτονικής ενός αποθηκευμένου προγράμματος, όπου δηλαδή ο υπολογιστής δέχεται εντολές από ένα πρόγραμμα. Αυτή είναι η λεγόμενη αρχιτεκτονική von Neumann[3]. Οι γλώσσες προγραμματισμού έκαναν σταδιακά την εμφάνιση τους τη δεκαετία του 1950 με τις κυριότερες να είναι οι FORTRAN, ALGOL και COBOL οι οποίες χρησιμοποιήθηκαν για την επίλυση επιστημονικών, αλγοριθμικών και οικονομικών προβλημάτων αντίστοιχα. Ο όρος 'μηχανική λογισμικού' (software engineering) που ανήκει στον Anthony Oettinger[4] και μετέπειτα στην Margaret Hamilton[5], χρησιμοποιήθηκε το 1968 ως τίτλος για το πρώτο συνέδριο στον κόσμο με θέμα τη μηχανική λογισμικού το όποιο υλοποιήθηκε από το ΝΑΤΟ. Εκεί τέθηκαν οι βάσεις για τις καλύτερες πρακτικές ανάπτυξης λογισμικού[6]. Αυτές οι πρακτικές δημιουργήθηκαν για να δείξουν την κακή ποιότητα του λογισμικού ως τότε, έτσι ώστε να εξασφαλιστεί η συστηματική παραγωγή λογισμικού μέσα στα χρονικά και οικονομικά περιθώρια που έχουν τεθεί.

Μερικές υποκατηγορίες της μηχανικής λογισμικού είναι οι παρακάτω:[1]

- Σχεδιασμός λογισμικού(Software design): Η διαδικασία ορισμού της αρχιτεκτονικής, των συστατικών ή άλλων χαρακτηριστικών στοιχείων ενός συστήματος.
- Κατασκευή λογισμικού(Software construction): Η διαδικασία παραγωγής

λογισμικού μέσω συγγραφής κώδικα, δοκιμών και αποσφαλμάτωσης (debugging).

- Δοκιμή λογισμικού(Software testing): Εκτενής διαδικασία δοκιμών με σκοπό την προμήθεια στοιχείων που αφορούν την καλή λειτουργία του λογισμικού.

- Συντήρηση λογισμικού(Software maintenance): Διαδικασίες υποστήριξης του παραγόμενου λογισμικού.

1.1 Μοτίβα σχεδίασης (Design patterns)

Ο σχεδιασμός ενός λογισμικού χρησιμοποιώντας αντικειμενοστραφή προγραμματισμό (object-oriented programming) είναι κάτι δύσκολο. Επαναχρησιμοποιώντας αντικειμενοστραφές λογισμικό είναι ακόμα δυσκολότερο. Ο σχεδιασμός θα πρέπει να είναι ακριβής και να αντιμετωπίζει το πρόβλημα αλλά και αρκετά γενικός έτσι ώστε να μπορεί να παραμετροποιηθεί στο μέλλον καλύπτοντας επιπλέον ανάγκες και απαιτήσεις. Συνήθως ο σχεδιασμός ενός τέτοιου λογισμικού δεν πετυχαίνει με την πρώτη φορά αλλά με συνεχείς δοκιμές μέχρι και την τελική μορφή του.

Γενικά ένα μοτίβο σχεδίασης έχει 4 απαραίτητα στοιχεία:[7]

- Το *όνομα του μοτίβου* είναι ένας τρόπος να περιγράψουμε το πρόβλημα, τις λύσεις και συνέπειες αυτού. Χρησιμοποιώντας ένα όνομα μπορούμε να περιγράψουμε καλύτερα το πρόβλημα που προσπαθούμε να επιλύσουμε καθώς και να προσθέσουμε ένα επίπεδο αφαιρετικότητας.

- Το *πρόβλημα* στο οποίο θα εφαρμοστεί το εκάστοτε μοτίβο. Έχοντας κατανοήσει το πρόβλημα σε βάθος μπορούμε να περιγράψουμε το μοτίβο σχεδίασης πιο συγκεκριμένα όπως για παράδειγμα το πως θα αναπαρασταθούν τα αντικείμενα ή οι αλγόριθμοι.

- Η *λύση* περιγράφει τα στοιχεία που αποτελούν το σχεδιασμό και τις σχέσεις μεταξύ τους. Η λύση δεν περιγράφει μια συγκεκριμένη υλοποίηση επειδή ένα μοτίβο είναι σαν ένα πρότυπο που μπορεί να εφαρμοστεί σε πολλές διαφορετικές καταστάσεις. Αντ' αυτού παρουσιάζει μια αφηρημένη περιγραφή του προβλήματος.

- Οι *συνέπειες* είναι τα αποτελέσματα και οι υποχωρήσεις που μπορεί να γίνουν εφαρμόζοντας ένα σχεδιαστικό μοτίβο. Είναι σημαντικές διότι με αυτό τον τρόπο μπορούν να βρεθούν εναλλακτικοί τρόποι σχεδίασης καθώς επίσης μπορεί να γίνει και καλύτερη κατανόηση του κόστους αλλά και του κέρδους από την υλοποίηση του μοτίβου.

Τα μοτίβα σχεδίασης χωρίζονται γενικά σε 3 κατηγορίες, δημιουργικά (creational), δομικά (structural) και συμπεριφοράς (behavioral). Τα δημιουργικά μοτίβα ασχολούνται με τη δημιουργία των αντικειμένων. Τα δομικά με τη σύνθεση των κλάσεων ή αντικειμένων. Τέλος, τα μοτίβα συμπεριφοράς καθορίζουν τους

τρόπους με τους οποίους οι κλάσεις ή τα αντικείμενα αλληλεπιδρούν. Παρακάτω φαίνεται παραστατικά η κατηγοριοποίηση των μοτίβων σχεδίασης.

Creational patterns	Structural patterns	Behavioral patterns
Abstract factory	Adapter	Chain of responsibility
Builder	Bridge	Command
Factory method	Composite	Interpreter
Lazy initialization	Decorator	Iterator
Prototype	Facade	Mediator
Singleton	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template method
		Visitor

Σχήμα 1.1: Μοτίβα σχεδίασης (design patterns)

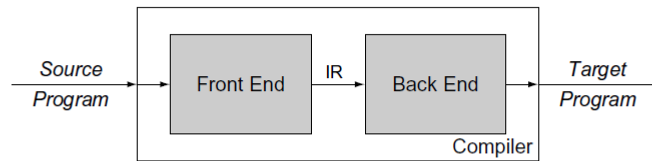
1.2 Μεταγλωττιστές - Μεταφραστές (Compilers - Translators)

Οι μεταγλωττιστές (compilers) είναι προγράμματα που μεταφράζουν ένα πρόγραμμα γραμμένο σε μια γλώσσα προγραμματισμού σε ένα άλλο πρόγραμμα γραμμένο σε μια άλλη γλώσσα. Ο μεταγλωττιστής είναι ένα σύστημα λογισμικού με πολλά εσωτερικά τμήματα και αλγορίθμους που επικοινωνούν μεταξύ τους. Κατά συνέπεια η μελέτη ενός τέτοιου συστήματος αποτελεί μια εισαγωγή σε τεχνικές και πρακτική εξάσκηση πάνω στη μηχανική λογισμικού (software engineering)[8].

Τυπικά ένας μεταγλωττιστής δέχεται μια πηγαία γλώσσα (source language) και παράγει μια γλώσσα στόχο (target language). Μια πηγαία γλώσσα μπορεί να είναι η C, C++, FORTRAN κλπ. ενώ η γλώσσα στόχος είναι συνήθως το σύνολο εντολών του εκάστοτε επεξεργαστή (instruction set).

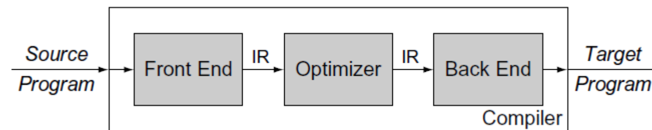
Υπάρχουν κάποιοι μεταγλωττιστές οι οποίοι δέχονται ως είσοδο μια γλώσσα προγραμματισμού και παράγουν ως έξοδο κώδικα σε κάποια άλλη γλώσσα προγραμματισμού. Αυτοί συνήθως ονομάζονται source-to-source μεταφραστές (translators). Πιο συγκεκριμένα η δομή ενός μεταγλωττιστή φαίνεται στο παρακάτω σχήμα.

Το εμπρός τμήμα (front end) του μεταγλωττιστή είναι αυτό που υποδέχεται το πηγαίο πρόγραμμα (source program) και το οποίο πρέπει να κατανοήσει και να κωδικοποιήσει αυτή τη γνώση σε μια ενδιάμεση αναπαράσταση (intermediate representation - IR). Όσο προχωράει η μεταγλώττιση του προγράμματος ο μεταγλωττιστής μπορεί να παράξει περισσότερες από μία IRs για να γίνει πιο αποδοτική η κωδικοποίηση του προγράμματος διατηρώντας την απαραίτητη πληροφορία. Πάντα όμως, μόνο μία IR είναι ενεργή κάθε φορά και η



Σχήμα 1.2: Μεταγλωττιστής ως ‘μαύρο κουτί’

οποία χρησιμοποιείται από το μεταγλωττιστή για την παραγωγή του τελικού προγράμματος. Η ύπαρξη μιας ενδιάμεσης αναπαράστασης μας επιτρέπει να προσθέσουμε περισσότερες από μία φάσεις στη μεταγλώττιση. Συνήθως μια ενδιάμεση φάση είναι η βελτιστοποίηση (optimization) μιας ενδιάμεσης αναπαράστασης (IR). Έτσι καταλήγουμε στο παρακάτω σχήμα.



Σχήμα 1.3: Μεταγλωττιστής 3 φάσεων ως ‘μαύρο κουτί’

Ο optimizer είναι ένας IR-to-IR μετατροπέας που δέχεται ως είσοδο μία IR και έχει ως έξοδο την βελτιστοποιημένη εκδοχή της. Μπορεί να κάνει περισσότερα από ένα περάσματα πάνω στην IR, έτσι ώστε να την αναλύσει και να την αποθηκεύσει ξανά. Ο optimizer μπορεί να διαμορφώσει την IR με τέτοιο τρόπο έτσι ώστε να παράγεται ένα πιο γρήγορο πρόγραμμα ή ένα που καταναλώνει λιγότερη μνήμη. Επίσης, θα μπορούσε να παράγει ένα τελικό πρόγραμμα που δεν θα καταναλώσει μεγάλη υπολογιστική ισχύ άρα μικρότερη κατανάλωση ενέργειας.

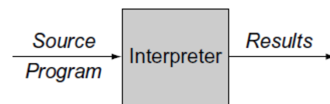
Το πίσω μέρος του μεταγλωττιστή, διασχίζει την ενδιάμεση αναπαράσταση (IR) και παράγει κώδικα για τη μηχανή-στόχο. Επιλέγει εντολές που υποστηρίζονται από την αρχιτεκτονική-στόχο για να υλοποιήσει τις λειτουργίες της IR. Διαλέγει την σειρά με την οποία θα εκτελεστούν και εισάγει επιπλέον κώδικα αν κριθεί απαραίτητο. Σε αντίθεση με το μεταγλωττιστή, ένας μεταφραστής θα διασχίσει την IR μορφή η οποία περιέχει συνοπτικά τη δομή της πηγαίας γλώσσας (source language) με στόχο την αντιστοίχηση των εντολών και δομών των 2 γλωσσών για την παραγωγή του τελικού κώδικα.

Διαισθητικά, έχουμε ένα τριφασικό μεταγλωττιστή (3-phase compiler) με κάθε φάση να αποτελείται από ένα ή περισσότερα περάσματα (passes). Το εμπρός μέρος (front end) κάνει 2 - 3 περάσματα έτσι ώστε να αναγνωρίσει την γλώσσα εισόδου σωστά και να την καταγράψει σε μια IR. Η ενδιάμεση φάση (optimizer) πραγματοποιεί με τη σειρά της κάποια περάσματα με σκοπό τη βελτιστοποίηση της IR χωρίς όμως να χάνεται η ουσία και η βασική δομής της. Η τρίτη φάση (back end) παράγει τον τελικό κώδικα (αν μιλάμε για translator) ή το τελικό πρόγραμμα (αν μιλάμε για compiler) το οποίο θα εκτελεστεί στην

εκάστοτε αρχιτεκτονική.

1.3 Διερμηνευτές (Interpreters)

Πολλά άλλα συστήματα μπορούν να θεωρηθούν μεταγλωττιστές (compilers). Ένας διερμηνευτής (interpreter) δέχεται ως είσοδο μια προδιαγραφή (specification) και παράγει ως έξοδο τα αποτελέσματα της εφαρμογής αυτής της προδιαγραφής. Μερικές γλώσσες όπως είναι η Perl, Scheme και η APL υλοποιούνται συνήθως με διερμηνευτές παρά με μεταγλωττιστές.



Σχήμα 1.4: Διερμηνευτής ως 'μαύρο κουτί'

Άλλες γλώσσες υιοθετούν και τις 2 μεθόδους, διερμηνεία και μεταγλώττιση. Η Java, για παράδειγμα, μεταγλωττίζεται από τον πηγαίο κώδικα σε μια συμπαγής μορφή που ονομάζεται bytecode. Οι εφαρμογές Java εκτελούνται φορτώνοντας τον bytecode στην JVM (Java Virtual Machine), ένα διερμηνευτή για τον bytecode. Πολλές υλοποιήσεις της JVM περιλαμβάνουν έναν μεταγλωττιστή ο οποίο εκκινεί κατά το χρόνο εκτέλεσης (runtime), καλείται και just-in-time μεταγλωττιστής (just-in-time compiler - JIT), ο οποίος μεταγλωττίζει τον bytecode για εκτέλεση από το σύστημα.

Οι διερμηνευτές (interpreters) και μεταγλωττιστές (compilers) έχουν πολλά κοινά.

- Και οι δύο αναλύουν την είσοδο και αποφαίνονται αν αποτελεί ένα έγκυρο πρόγραμμα.
- Και οι δύο κατασκευάζουν ένα εσωτερικό μοντέλο αναπαράστασης του προγράμματος(μια ενδιάμεση αναπαράσταση, intermidiate representation - IR).
- Και οι δύο αποφασίζουν που θα αποθηκευτούν οι μεταβλητές κατά την εκτέλεση.

Η διαφορά έγκειται στο γεγονός ότι ο διερμηνευτής 'διαβάζει το πρόγραμμα' έτσι ώστε να παράξει το επιθυμητό αποτέλεσμα, σε αντίθεση με τον μεταγλωττιστή που παράγει ένα τελικό πρόγραμμα το οποίο θα πρέπει να εκτελεστεί για παραχθεί αποτέλεσμα.

Κεφάλαιο 2

Λεκτική ανάλυση (Lexixal analysis)

Η λεκτική ανάλυση (lexical analysis ή αλλιώς scanning) είναι η διαδικασία αναγνώρισης του προγράμματος εισόδου. Ο λεκτικός αναλυτής (lexical analyzer ή lexer) δέχεται μια ροή από χαρακτήρες και παράγει μια ροή από λέξεις. Αθροίζει τους χαρακτήρες για να σχηματίσει λέξεις και εφαρμόζει ένα σύνολο κανόνων για να αποφανθεί αν η κάθε λέξη είναι έγκυρη στην πηγαία γλώσσα (source language). Αν η λέξη υπάρχει, τότε ο λεκτικός αναλυτής την αναθέτει σε μια συντακτική κατηγορία[1].

Ο σαρωτής (scanner) είναι το μόνο πέρασμα που διαχειρίζεται κάθε χαρακτήρα του προγράμματος εισόδου. Επειδή εκτελεί μια απλή εργασία, οι υλοποιήσεις του μπορεί να είναι γρήγορες και αποδοτικές. Αμφότεροι, παραγόμενοι (generated) και χειροποίητοι (hand-crafted) σαρωτές, βασίζονται στις ίδιες τεχνικές. Ενώ τα περισσότερα εγχειρίδια και μαθήματα συνηγορούν στη χρήση παραγόμενων σαρωτών, οι περισσότεροι εμπορικοί (commercial) ή ανοιχτού κώδικα (open-source) μεταγλωττιστές βασίζονται σε χειροποίητους σαρωτές. Ένας χειροποίητος σαρωτής μπορεί να είναι πιο γρήγορος από έναν παραγόμενο επειδή η υλοποίηση του μπορεί να βελτιστοποιήσει ένα μέρος του κώδικα το οποίο δεν μπορεί να αποφευχθεί από έναν παραγόμενο σαρωτή. Επειδή οι σαρωτές είναι απλοί και δεν επιδέχονται μεταβολές στον κώδικα τους συχνά, το κέρδος στην απόδοση από έναν χειροποίητο σαρωτή υπερτερεί έναντι της ευκολίας που διαθέτει ένας παραγόμενος σαρωτής.

2.1 Αναλυτές (Recognisers)

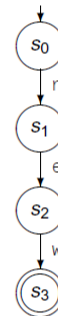
Ο πιο απλός τρόπος αναγνώρισης λέξεων είναι συνήθως χαρακτήρα-προς-χαρακτήρα. Για παράδειγμα, έστω ότι θέλουμε να αναγνωρίσουμε τη λέξη "ne-

[1] **Συντακτική κατηγορία**: Κατηγοριοποίηση των λέξεων βάση της γραμματικής τους χρήσης, σε ποιο μέρος του λόγου χρησιμοποιούνται.

w". Η συνάρτηση `NextChar` επιστρέφει τον επόμενο χαρακτήρα. Παρακάτω φαίνεται το κομμάτι κώδικα που χρησιμοποιήθηκε. Ο κώδικας δουλεύει με ένα χαρακτήρα κάθε φορά. Μπορούμε να αναπαραστήσουμε τον κώδικα με τη μορφή ενός απλού διαγράμματος μεταβάσεων όπως φαίνεται δεξιά του κώδικα. Το διάγραμμα μεταβάσεων αναπαριστά έναν αναλυτή. Κάθε κύκλος αναπαριστά μια αφηρημένη κατάσταση. Κάθε διπλός κύκλος αναπαριστά μια αποδεκτή κατάσταση.

```
c ← NextChar();
if c = 'n' then
    begin;
    c ← NextChar();
    if c = 'e' then
        begin;
        c ← NextChar();
        if c = 'w' then
            report success;
        else
            try something else;
        end if
    else
        try something else;
    end if
else
    try something else;
end if
```

(α΄) Αλγόριθμος

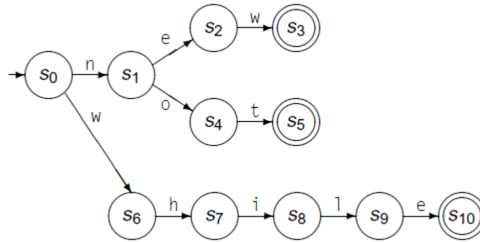


(β΄) Πεπερασμένο αυτόματο

Σχήμα 2.1: Αλγόριθμος για την αναγνώριση της λέξης 'new'

Η αρχική κατάσταση (initial state, start state) είναι η s_0. Η κατάσταση s_3 είναι η αποδεκτή κατάσταση (accept state). Ο recongiser φτάνει στην κατάσταση s_3 μόνο όταν αναγνωρίσει την λέξη 'new'. Τα βελάκια δείχνουν τις μεταβάσεις από την μία κατάσταση στην άλλη. Επίσης, υπάρχει και η κατάσταση λάθους (error state) s_e στην οποία μεταβαίνουμε όταν η είσοδος είναι διαφορετική από την επιθυμητή. Κάθε κατάσταση έχει μια μετάβαση στην κατάσταση λάθους.

Ένα συνδυαστικό παράδειγμα φαίνεται παρακάτω αναγνωρίζοντας τις λέξεις new, not και while.



Σχήμα 2.2: Πεπερασμένο αυτόματο για τις λέξεις 'new', 'not' και 'while'

Ο μαθηματικός φορμαλισμός των διαγραμμάτων μετάβασης είναι τα πεπερασμένα αυτόματα. Τα πεπερασμένα αυτόματα θα χρησιμοποιηθούν στην συνέχεια για την μοντελοποίηση των λεκτικών αναλυτών.

Ο ορισμός ενός πεπερασμένου αυτόματου[2] είναι η πεντάδα $(S, \Sigma, \delta, s_0, S_A)$ όπου:

- S είναι οι πεπερασμένες καταστάσεις του αναλυτή μαζί με την κατάσταση λάθους
- Σ είναι το πεπερασμένο αλφάβητο που χρησιμοποιείται από τον αναλυτή
- δ είναι η συνάρτηση μετάβασης του λεκτικού αναλυτή. Καταγράφει κάθε κατάσταση $s \in S$ και κάθε χαρακτήρα $c \in S$ σε μια επόμενη κατάσταση $s_i \xrightarrow{c} \delta(s_i, c) == s_f \in S$ όπου s_i η αρχική κατάσταση
- $s_0 \in S$ είναι η αρχική κατάσταση
- S_A είναι το σύνολο των αποδεκτών καταστάσεων. Κάθε αποδεκτή κατάσταση συμβολίζεται με έναν διπλό κύκλο στο διάγραμμα μεταβάσεων

Χρησιμοποιώντας τον παραπάνω φορμαλισμό έχουμε:

- $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$
- $\Sigma = \{e, h, i, l, n, o, t, w\}$
- $\delta = \left\{ \begin{array}{l} s_0 \xrightarrow{n} s_1, s_0 \xrightarrow{w} s_6, s_1 \xrightarrow{e} s_2, s_1 \xrightarrow{o} s_4, s_2 \xrightarrow{w} s_3, \\ s_4 \xrightarrow{t} s_5, s_6 \xrightarrow{h} s_7, s_7 \xrightarrow{i} s_8, s_8 \xrightarrow{l} s_9, s_9 \xrightarrow{e} s_{10} \end{array} \right\}$
- $s_0 = s_0$
- $S_A = \{s_3, s_5, s_{10}\}$

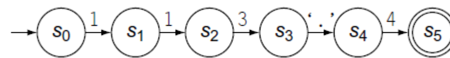
[2]**Πεπερασμένο αυτόματο**: Ένας μαθηματικός φορμαλισμός για αναλυτές οι οποίοι έχουν ένα πεπερασμένο σύνολο καταστάσεων, αλφάβητο, συνάρτηση μετάβασης, μία αρχική κατάσταση και μία ή περισσότερες αποδεκτές καταστάσεις.

Για οποιονδήποτε άλλο συνδυασμό της κατάστασης s_i και του χαρακτήρα c ορίζεται η κατάσταση λάθους $\delta(s_i, c) = s_e$

Ένα πεπερασμένο αυτόματο αναγνωρίζει ένα αλφαριθμητικό αν και μόνο αν, εκκινώντας από την κατάσταση s_0, η ακολουθία των χαρακτήρων στο αλφαριθμητικό οδηγεί το FA σε μια σειρά από μεταβάσεις που καταλήγει σε μια αποδεκτή κατάσταση όταν έχει διαπεραστεί ολόκληρο το αλφαριθμητικό. Μόλις περάσει σε κατάσταση λάθους το πεπερασμένο αυτόματο, τότε παραμένει σε αυτή μέχρι και το πέρας της διαπέρασης του αλφαριθμητικού. Αυτό αποτελεί συντακτικό λάθος (lexical error).

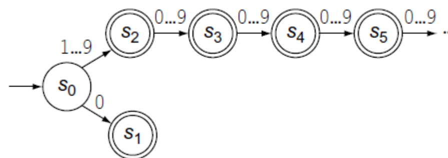
- Η είσοδος έχει διαπεραστεί όσο δεν βρισκόμαστε σε αποδεκτή κατάσταση
- Ο χαρακτήρας μπορεί να οδηγήσει τον λεκτικό αναλυτή στην κατάσταση λάθους (error state)

Το μοντέλο χαρακτήρα-χαρακτήρα που είδαμε παραπάνω μπορεί να επεκταθεί για χρήση μιας αυθαίρετης συλλογής λέξεων. Παρακάτω γίνεται αναγνώριση ενός συγκεκριμένου αριθμού, του 113,4.



Σχήμα 2.3: Πεπερασμένο αυτόματο για τον αριθμό '113,4'

Το άνωθεν διάγραμμα μεταβάσεων αφορά ένα συγκεκριμένο αριθμό. Πρέπει να κατασκευάσουμε ένα διάγραμμα μεταβάσεων που να αναγνωρίζει οποιοδήποτε αριθμό. Ας περιοριστούμε μόνο σε μη προσημασμένους ακεραίους αριθμούς (unsigned integers). Γενικά, ένας αριθμός μπορεί να είναι 0 ή μια σειρά από ένα ή περισσότερα ψηφία όπου το πρώτο ψηφίο κυμαίνεται από ένα 1 έως 9 και τα υπόλοιπα από 0 έως 9.



Σχήμα 2.4: Πεπερασμένο αυτόματο για την αναγνώριση μη προσημασμένων ακεραίων αριθμών (γενική περίπτωση)

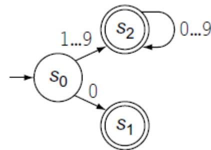
Η μετάβαση s_0 σε s_1 χειρίζεται την περίπτωση που ο αριθμός είναι 0. Οι καταστάσεις s_2, s_3 κ.ο.κ. χειρίζονται την περίπτωση που ο αριθμός είναι μεγαλύτερος του 1. Όπως φαίνεται και από το σχήμα, το διάγραμμα μεταβάσεων είναι ατέρμονο συνεπώς παραβιάζει την αρχή ότι δηλαδή ένα πεπερασμένο αυτόματο είναι πράγματι πεπερασμένο. Επίσης οι καταστάσεις μετά την s_2 είναι ίδιες και είναι όλες αποδεκτές καταστάσεις.

Παρακάτω φαίνεται ένας αλγόριθμος αναγνώρισης μη προσημασμένων ακεραίων αριθμών:

$char \leftarrow NextChar();$
$state \leftarrow s_0;$
while $char \neq eof$ and $state \neq s_e$ **do**
    $state \leftarrow \delta(state, char);$
    $char \leftarrow NextChar();$
end while
if $state \in S_A$ **then**
    report acceptance;
else
    report failure;
end if;

Σχήμα 2.5: Αλγόριθμος αναγνώρισης μη προσημασμένων ακεραίων αριθμών

Μπορούμε να απλοποιήσουμε το πεπερασμένο αυτόματο αν επιτρέψουμε στο διάγραμμα μεταβάσεων να κάνει κύκλους. Ομαδοποιούμε όλες τις καταστάσεις μετά την s_2 και έτσι υφίσταται μόνο μία κατάσταση στη περίπτωση που έχουμε παραπάνω από ένα ψηφία.



Σχήμα 2.6: Πεπερασμένο αυτόματο για την αναγνώριση μη προσημασμένων ακεραίων αριθμών (σωστός τρόπος)

Ο παρακάτω πίνακας δείχνει την ακολουθία των μεταβάσεων μεταξύ όλων των πιθανών ψηφίων.

δ	0	1	2	3	4	5	6	7	8	9	Other
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

Σχήμα 2.7: Πίνακας μεταβάσεων

Ο παραπάνω πίνακας μπορεί να συμπιεστεί διότι οι στήλες 1 έως 9 έχουν το ίδιο περιεχόμενο συνεπώς μπορούν ενωθούν σε μία στήλη.

δ	0	1 ... 9	Other
s_0	s_1	s_2	s_e
s_1	s_e	s_e	s_e
s_2	s_2	s_2	s_e
s_3	s_e	s_e	s_e

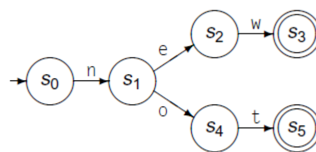
Σχήμα 2.8: Πίνακας μεταβάσεων (συμπιεσμένη μορφή)

2.2 Κανονικές εκφράσεις (Regular expressions)

Το σύνολο των λέξεων που γίνεται αποδεκτό από ένα πεπερασμένο αυτόματο (finite automaton - FA) F, διαμορφώνουν μία γλώσσα που δηλώνεται ως L(F). Το διάγραμμα μεταβάσεων του FA καθορίζει τη δομή της γλώσσας. Για οποιοδήποτε FA μπορούμε επίσης να περιγράψουμε τη γλώσσα χρησιμοποιώντας μια άλλη μέθοδο που ονομάζεται κανονική έκφραση (regular expression - RE). Μια γλώσσα που περιγράφεται από μια κανονική έκφραση ονομάζεται κανονική γλώσσα (regular language).

- Η γλώσσα που αποτελείται μόνο από τη λέξη ‘new’ μπορεί να περιγραφεί από την RE γραμμένη ως ‘new’. Γράφοντας τους χαρακτήρες το έναν δίπλα στον άλλο, υποδηλώνει ότι θα εμφανιστούν με αυτή τη σειρά.
- Η γλώσσα που αποτελείται μόνο από τις λέξεις ‘new’ ή ‘while’ μπορεί να γραφεί ως ‘new | while’, όπου το ‘ή’ συμβολίζεται με το χαρακτήρα ‘|’.
- Η γλώσσα που αποτελείται μόνο από τις λέξεις ‘new’ ή ‘not’ μπορεί να γραφεί ως ‘new | not’. Μια εναλλακτική RE είναι η εξής: ‘n(ew|ot)’.

Παρακάτω φαίνεται σχηματικά το FA.



Σχήμα 2.9: Πεπερασμένο αυτόματο των λέξεων ‘new’, ‘not’

Μια RE κατασκευάζεται από 3 βασικές λειτουργίες:

1. Διάζευξη (Alternation): Η διάζευξη μεταξύ 2 συνόλων αλφαριθμητικών R και S, συμβολίζεται με R|S και είναι $x|x \in R$ ή $x \in S$.
2. Σύζευξη (Concatenation): Η σύζευξη 2 συνόλων R και S συμβολίζεται με RS και περιέχει όλα τα αλφαριθμητικά που σχηματίζονται προσθέτοντας στην αρχή ενός στοιχείου του R ένα από το S, δηλαδή $xy|x \in R$ και $y \in S$.

3. Κλειστότητα (Closure): Η κλειστότητα του Kleene για ένα σύνολο R, συμβολίζεται με R^*, είναι $\bigcup_{i=0}^{\infty} R^i$. Αυτή είναι απλά η ένωση όλων των συνδυασμών του R με τον εαυτό του, καμία ή περισσότερες φορές.

Χρησιμοποιώντας τους τρεις βασικούς τελεστές, διάζευξη (alternation), σύζευξη (concatenation) και κλειστότητα (closure), μπορούμε να ορίσουμε το σύνολο των κανονικών εκφράσεων (regular expressions) βάσει ενός αλφαβήτου Σ όπως φαίνεται παρακάτω:

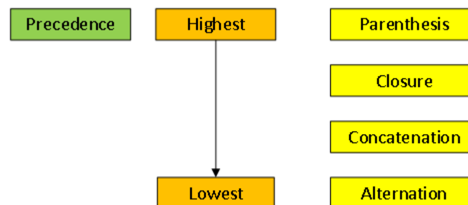
- Αν το $\alpha \in \Sigma$ τότε το α είναι επίσης μια κανονική έκφραση που υποδηλώνει ότι το σύνολο περιέχει μόνο το α

- Αν τα r και s ή οι κανονικές εκφράσεις υποδηλώνουν τα σύνολα L(r) και L(s) αντίστοιχα τότε

  - $r|s$ είναι μια κανονική έκφραση η οποία υποδηλώνει ένωση (union) ή διάζευξη (alternation) των L(r) και L(s)
  - rs είναι μια κανονική έκφραση η οποία υποδηλώνει τη σύζευξη των L(r) και L(s)
  - r^* είναι μια κανονική έκφραση η οποία υποδηλώνει την κλειστότητα του L(r)

- ϵ είναι μια κανονική έκφραση που υποδηλώνει ένα άδειο σύνολο που περιέχει ένα άδειο αλφαριθμητικό

Παρακάτω απεικονίζονται οι προτεραιότητες, από την υψηλότερη προς τη χαμηλότερη, με βάση τα σύμβολα που υπάρχουν στις κανονικές εκφράσεις.



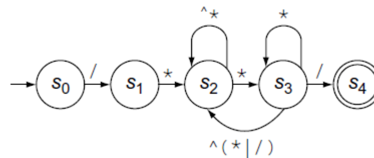
Σχήμα 2.10: Προτεραιότητες κανονικών εκφράσεων

Στη συνέχεια, θα παρουσιάσουμε πως μπορούμε να χρησιμοποιήσουμε τεχνικές για να αυτοματοποιήσουμε την κατασκευή υψηλής ποιότητας σαρωτών (scanners).

Μπορούμε να καθορίσουμε ένα εύρος από χαρακτήρες με το πρώτο και τελευταίο στοιχείο να συνδέονται με αποσιωπητικά ‘ ... ’, π.χ. $[0 \ldots 9] \equiv (0|1|2|3|4|5|6|7|8|9)$. Μπορούμε να καθορίσουμε ένα σύνολο από όλους του χαρακτήρες εξαιρώντας μερικούς χρησιμοποιώντας τον τελεστή του συμπληρώματος ‘^’ [^εξαιρούμενοι χαρακτήρες] π.χ. [^’] που σημαίνει ‘ αναγνώρισε όλους τους χαρακτήρες εκτός από το ‘ ’. Με την τελεία ‘.’ ταυτοποιούμε οποιοδήποτε χαρακτήρα εκτός από το χαρακτήρα αλλαγής γραμμής ‘\n’. Ακολουθούν

διάφορα παραδείγματα κανονικών εκφράσεων για χρήση σε πραγματικές γλώσσες προγραμματισμού:

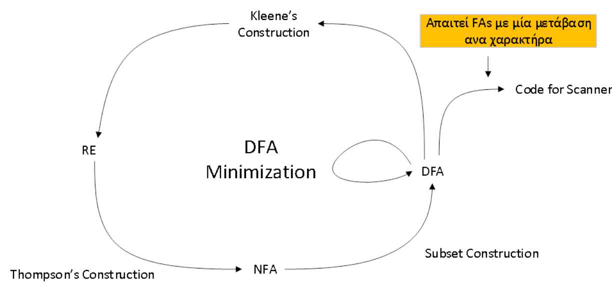
- Identifier: $([A \ldots Z]|[a \ldots z])([A \ldots Z]|[a \ldots z]|[0 \ldots 9])^*$
- Μη προσημασμένος ακέραιος: $0|[1 \ldots 9][0 \ldots 9]^*$
- Μη προσημασμένοι πραγματικοί αριθμοί: $(0|[1 \ldots 9][0 \ldots 9]^*)(e|.[0 \ldots 9]^*)E(e|+|-)(0[1 \ldots 9][0 \ldots 9]^*)$
- Αλφαριθμητικό: "(^("|$\backslash n$))*"
- Σχόλια μονής γραμμής: //(^$\backslash n)^*$ \n. Ο τελευταίος χαρακτήρας αλλαγής γραμμής \n δεν είναι και τόσο απαραίτητος.
- Σχόλια πολλαπλών γραμμών: Αν γράψουμε την κανονική έκφραση /*.*/ τότε ένα αλφαριθμητικό δεν μπορεί να αναγνωριστεί γιατί η '.' αφομοιώνει το τελευταίο '*'. Κατά συνέπεια ο οριοθέτης '*/' δεν μπορεί να αναγνωριστεί από τον σαρωτή. Αν αποδεχτούμε το χαρακτήρα '*' στο κέντρο του αλφαριθμητικού θα πρέπει να γράψουμε: /* (^*|*+^/)* */. Το πεπερασμένο αυτόματο (FA) που υλοποιεί αυτή την κανονική έκφραση (RE) φαίνεται παρακάτω.



Σχήμα 2.11: Πεπερασμένο αυτόματο για τα σχόλια πολλαπλής γραμμής (/* */)

2.3 Από τις κανονικές εκφράσεις στο σαρωτή

Σε αυτό το τμήμα θα αναφερθούμε στην αυτοματοποίηση της παραγωγής σαρωτών από μία συλλογή κανονικών εκφράσεων. Θα περιγράψουμε το αλγοριθμικό μονοπάτι που ακολουθείται από μία κανονική έκφραση για τη δημιουργία ενός πεπερασμένου αυτόματου που είναι κατάλληλο για άμεση υλοποίηση και παράλληλα έναν αλγόριθμο που παράγει μια κανονική έκφραση για μια γλώσσα που είναι αποδεκτή από το πεπερασμένο αυτόματο. Το παρακάτω σχήμα παρουσιάζει τις σχέσεις μεταξύ όλων αυτών των αλγορίθμων.



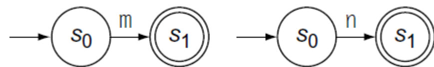
Σχήμα 2.12: Σχέσεις μεταξύ κανονικών εκφράσεων - NFA - DFA

Για να κατανοήσουμε αυτούς τους αλγορίθμους, πρέπει να γίνει διάκριση μεταξύ των ντετερμινιστικών πεπερασμένων αυτόματων (deterministic FAs or DFAs) και των μη-ντετερμινιστικών πεπερασμένων αυτόματων (nondeterministic FAs or NFAs). Στη συνέχεια παρουσιάζουμε 3 αλγορίθμους:

- Thompson construction: Παράγει ένα NFA από μια RE
- Subset construction: Κατασκευάζει ένα DFA το οποίο προσομοιώνει ένα NFA
- Αλγόριθμος του Hopcroft: Ελαχιστοποιεί ένα NFA

Για να εξακριβώσουμε την ισοτιμία μεταξύ των REs και DFAs πρέπει να δείξουμε ότι οποιοδήποτε DFA είναι ίσο με μια RE. O αλγόριθμος του Κλεενε παράγει μία RE από ένα DFA

Από τον ορισμό μια κανονικής έκφρασης ορίσαμε το κενό αλφαριθμητικό ϵ σαν μια κανονική έκφραση (RE). Στην πράξη, το κενό αλφαριθμητικό ϵ χρησιμοποιείται για τον συνδυασμό πεπερασμένων αυτόματων για πιο πολύπλοκες κανονικές εκφράσεις. Για παράδειγμα, έστω ότι έχουμε τα 2 παρακάτω FAs για τις REs, m και n που ονομάζεται FA_m και FA_n.

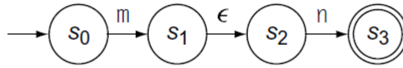


Σχήμα 2.13: Πεπερασμένα αυτόματα για τις εκφράσεις m και n

Μπορούμε να κατασκευάσουμε ένα FA για την κανονική έκφραση mn προσθέτοντας μια μετάβαση στην ϵ[3] από την αποδεκτή κατάσταση του FA_m στην αρχική κατάσταση του FA_n, αλλάζοντας τα νούμερα στις καταστάσεις και χρησιμοποιώντας την αποδεκτή κατάσταση του FA_n's ως την αποδεκτή κατάσταση του νέου FA.

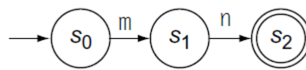
Προσθέτοντας αυτή την ϵ-μετάβαση, ο ορισμός της αποδοχής πρέπει να αλλάξει έτσι ώστε να επιτρέπει περισσότερες από μία ϵ-μεταβάσεις μεταξύ 2 χαρακτήρων της εισόδου. Για παράδειγμα, στην κατάσταση s_1 το FA περιέχει τη

[3] ϵ-**μετάβαση**: Μια μετάβαση στο κενό αλφαριθμητικό ϵ η οποία δεν προωθεί την είσοδο



Σχήμα 2.14: Πεπερασμένο αυτόματο για την έκφραση mn χρησιμοποιώντας την ϵ-μετάβαση

μετάβαση $s_1 \xrightarrow{\epsilon} s_2$ χωρίς να καταναλώνει κάποιο χαρακτήρα της εισόδου. Συνεπώς το FA μπορεί να απλοποιηθεί εξαλείφοντας την ϵ-μετάβαση.



Σχήμα 2.15: Πεπερασμένο αυτόματο για την έκφραση mn

Παρακάτω αναφέρουμε 2 μοντέλα που περιγράφουν τη συμπεριφορά του NFA:

- Κάθε φορά που το NFA πρέπει να κάνει μία μη-ντετερμινιστική επιλογή ακολουθεί τη μετάβαση που οδηγεί σε μια αποδεκτή κατάσταση για μια είσοδο, αν υπάρχει μια τέτοια μετάβαση. Πρακτικά, το NFA μαντεύει τη σωστή μετάβαση σε κάθε σημείο.
- Κάθε φορά που το NFA πρέπει να κάνει μία μη-ντετερμινιστική επιλογή δημιουργεί έναν κλώνο του εαυτού του για να ακολουθήσει όλες τις πιθανές μεταβάσεις. Συνεπώς για κάθε χαρακτήρα που έρχεται ως είσοδος το NFA έχει ένα συγκεκριμένο σύνολο από καταστάσεις που προέρχεται από όλους τους κλώνους του. Σε αυτό το μοντέλο, το NFA ακολουθεί όλα τα μονοπάτια ταυτόχρονα. Οποιαδήποτε στιγμή καλούμε το συγκεκριμένο σύνολο από καταστάσεις στο οποίο το NFA έχει ενεργή διαμόρφωση (active configuration). Όταν το NFA φτάσει σε μια διαμόρφωση στην οποία έχει διαπεράσει όλη την είσοδο και ένας ή περισσότεροι κλώνοι έχουν φτάσει σε μια αποδεκτή κατάσταση, το NFA αποδέχεται το αλφαριθμητικό.

2.3.1 Ισοτιμία των NFAs και DFAs

NFAs[4] και DFAs[5] είναι ισότιμα όσον αφορά την εκφραστική τους δύναμη. Οποιοδήποτε DFA είναι μια ειδική περίπτωση ενός NFA. Συνεπώς, ένα NFA είναι τουλάχιστον τόσο ισχυρό όσο ένα DFA.

Για να προσομοιώσουμε τη συμπεριφορά του NFA, χρειαζόμαστε ένα DFA με μία κατάσταση για κάθε configuration του NFA. Σαν αποτέλεσμα, το DFA μπορεί να έχει εκθετικά περισσότερες καταστάσεις από το NFA. Το σύνολο των

[4] **Μη-ντετερμινιστικό FA**: ένα FA το οποίο επιτρέπει μεταβάσεις στο κενό αλφαριθμητικό ϵ και καταστάσεις οι οποίες περιέχουν πολλαπλές μεταβάσεις στον ίδιο χαρακτήρα

[5] **Ντετερμινιστικό FA**: ένα FA όπου η συνάρτηση μετάβασης είναι μονότιμη. Τα DFAs δεν επιτρέπουν ϵ-μεταβάσεις

καταστάσεων του DFA μπορεί να είναι μεγάλο αλλά πεπερασμένο. Η προσομοίωση ενός NFA σε ένα DFA αντιμετωπίζει ένα πιθανό πρόβλημα χώρου όχι όμως και χρόνου.

2.4 Υλοποίηση σαρωτών (Scanner construction)

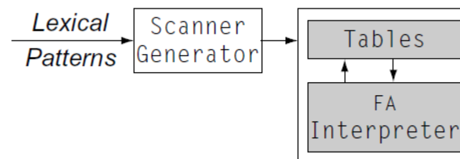
Για τις περισσότερες γλώσσες, μπορούμε να παράγουμε ένα αποδεκτά γρήγορο σαρωτή απευθείας από ένα σύνολο κανονικών εκφράσεων. Δημιουργούμε μια κανονική έκφραση για κάθε συντακτική κατηγορία και δίνουμε ως είσοδο τις κανονικές εκφράσεις στη γεννήτρια σαρωτών (scanner generator). Η γεννήτρια κατασκευάζει ένα NFA για κάθε κανονική έκφραση, τα ενώνει χρησιμοποιώντας ϵ-μεταβάσεις, δημιουργεί ένα αντίστοιχο DFA και το ελαχιστοποιεί. Σε αυτό το σημείο πρέπει η scanner generator να μετατρέψει το DFA σε εκτελέσιμο κώδικα. Σε αυτό το κεφάλαιο θα αναλύσουμε 3 στρατηγικές υλοποίησης για την μετατροπή ενός DFA σε εκτελέσιμο κώδικα:

- table-driven σαρωτής
- direct-coded σαρωτής
- hand-coded σαρωτής

Όλοι αυτοί οι σαρωτές λειτουργούν με τον ίδιο τρόπο, προσομοιώνοντας το DFA. Διαβάζουν επαναλαμβανόμενα τον επόμενο χαρακτήρα της εισόδου και προσομοιώνουν τη μετάβαση στο DFA που προκαλείται από το χαρακτήρα. Αυτή η διαδικασία σταματάει το DFA να αναγνωρίσει μία λέξη. Αυτό συμβαίνει όταν η τρέχουσα κατάσταση s δεν έχει εξερχόμενη ακμή στον τρέχων χαρακτήρα της εισόδου.

- Αν η s είναι αποδεκτή κατάσταση, ο σαρωτής αναγνωρίζει τη λέξη, επιστρέφει το lexeme[6] της καθώς και τη συντακτική κατηγορία στην καλούσα διαδικασία
- Αν η s δεν είναι αποδεκτή κατάσταση, ο σαρωτής πρέπει να εξακριβώσει αν πέρασε ή όχι από μία αποδεκτή κατάσταση περνώντας από την κατάσταση s. Αν ο σαρωτής πράγματι συνάντησε μια αποδεκτή κατάσταση, θα πρέπει να γυρίσει πίσω (backtrack) την εσωτερική του κατάσταση (internal state) καθώς και το ρεύμα εισόδου του (input stream) σε εκείνο το σημείο και ανακοινώνει την επιτυχή αναγνώριση
- Αν δεν έφτασε ποτέ σε μια αποδεκτή κατάσταση τότε ο σαρωτής θα πρέπει να ανακοινώσει ότι υπάρχει λάθος

[6]**lexeme**: Το ακριβές κείμενο για μία λέξη η οποία έχει αναγνωριστεί από ένα FA



Σχήμα 2.16: Παραγωγή ενός Table-driven σαρωτή

2.4.1 Table-Driven σαρωτές

Η table-driven προσέγγιση χρησιμοποιεί ένα πρότυπο σαρωτή για έλεγχο και ένα σύνολο από παραγόμενους πίνακες οι οποίοι κωδικοποιούν την εκάστοτε γλώσσα. Στο παρακάτω σχήμα παρουσιάζονται τα βασικά τμήματα ενός table-driven σαρωτή.

Δοθέντος των κανονικών εκφράσεων η γεννήτρια σαρωτών (scanner generator) παράγει ένα πρότυπο σαρωτή και τους πίνακες που κωδικοποιούν το συντακτικό με λεπτομέρειες. Ο πρότυπος σαρωτής χωρίζεται σε 4 μέρη: το πρώτο είναι οι προαπαιτούμενες αρχικοποιήσεις, ένα scanning loop το οποίο μοντελοποιεί τη συμπεριφορά του DFA, ένα loop το οποίο θα γυρίζει πίσω (roll-back) στην περίπτωση που ο σαρωτής ξεπεράσει την λέξη και ένα τελευταίο τμήμα στο οποίο ερμηνεύει και ανακοινώνει τα αποτελέσματα. Το scanning loop επαναλαμβάνει τις 2 βασικές ενέργειες του σαρωτή: διάβασμα ενός χαρακτήρα και προσομοίωση της ενέργειας του DFA. Το loop σταματάει όταν εισέλθει στην κατάσταση λάθους s_e. Ένα παράδειγμα table-driven σαρωτή φαίνεται παρακάτω.

```
state ← s_0;
lexeme ← '';
clear stack;
push(bad);
while state ≠ s_e do
    NextChar(char);
    lexeme ← lexeme + char;
    if state ∈ S_A then
        clear stack;
    end if;
    push(state);
    cat ← CharCat[char];
    state ← δ[state, cat];
end while
while char ∉ S_A and state ≠ bad do
    state ← pop();
    trucate lexeme;
    RollBack();
end while
if state ∈ S_A then
    return Type[state];
else
    return invalid;
end if;
```

Σχήμα 2.17: Αλγόριθμος Table-driven σαρωτών

Έχοντας ένα DFA, η γεννήτρια σαρωτών (scanner generator) μπορεί να γεννήσει τους πίνακες με άμεσο τρόπο. Ο αρχικός πίνακας έχει μία στήλη για κάθε χαρακτήρα του αλφαβήτου εισόδου και μια γραμμή για κάθε κατάσταση στο DFA. Για κάθε κατάσταση, η γεννήτρια εξετάζει με τη σειρά τις εξερχόμενες μεταβάσεις και γεμίζει τη γραμμή με τις κατάλληλες καταστάσεις.

Για να μοντελοποιήσουμε ένα διαφορετικό DFA, για μία άλλη γλώσσα, το μόνο που έχουμε να κάνουμε είναι να προμηθεύσουμε τη γεννήτρια σαρωτών με νέους πίνακες.

2.4.2 Direct-Coded σαρωτές

Για να βελτιώσουμε την απόδοση ενός table-driven σαρωτή πρέπει να μειώσουμε το κόστος της μίας ή και των 2 βασικών του ενεργειών: διάβασμα ενός χαρακτήρα και υπολογισμός της επόμενης μετάβασης του DFA. Οι Direct-coded σαρωτές μειώνουν το κόστος του υπολογισμού των μεταβάσεων του DFA αντικαθιστώντας την άμεση αναπαράσταση του γράφου καταστάσεων και μεταβάσεων του DFA με μια πιο έμμεση αναπαράσταση, όπως είναι ο κώδικας. Αυτή η έμμεση αναπαράσταση απλοποιεί το δεύτερο βήμα, που είναι ο υπολογισμός των μεταβάσεων με βάση τους πίνακες (table-lookup). Ο παραγόμενος σαρωτής

έχει την ίδια λειτουργικότητα με έναν table-driven σαρωτή αλλά με μικρότερο επιπλέον φόρτο ανά χαρακτήρα.

Ένας table-driven σαρωτής περνά τον περισσότερο χρόνο του μέσα σε ένα κεντρικό while loop. Συνεπώς, η καρδιά ενός table-driven σαρωτή είναι ένα while loop όπως φαίνεται παρακάτω στην πιο απλή μορφή του:

while $state \neq s_e$ **do**
  NextChar(char);
  $cat \leftarrow CharCat[char]$;
  $state \leftarrow \delta[state, cat]$;
end while

Σχήμα 2.18: Ένα απλό while loop ενός table-driven σαρωτή

2.4.3 Hand-Coded σαρωτές

Οι παραγόμενοι σαρωτές, είτε table-driven είτε direct-coded, χρησιμοποιούν ένα μικρό, σταθερό κομμάτι χρόνου ανά χαρακτήρα. Παρόλα αυτά, πολλοί compilers χρησιμοποιούν hand-coded σαρωτές. Ο direct-coded σαρωτής μείωσε το φόρτο της προσομοίωσης του DFA. Ο hand-coded σαρωτής μπορεί να μειώσει το φόρτο που υπάρχει ανάμεσα στο σαρωτή και στο υπόλοιπο σύστημα. Πιο συγκεκριμένα, μια προσεκτική υλοποίηση μπορεί να βελτιώσει τους μηχανισμούς που χρησιμοποιούνται για το διάβασμα και το χειρισμό των χαρακτήρων της εισόδου καθώς και τις λειτουργίες που χρειάζονται για την παραγωγή ενός αντιγράφου του ακριβούς κειμένου στην έξοδο.

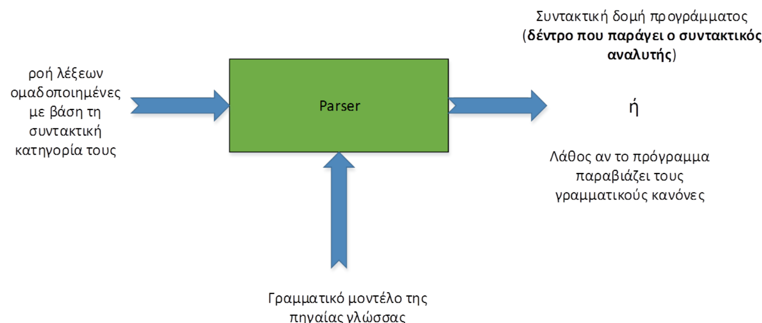
Παρόλο που η ανάγνωση ανά χαρακτήρα οδηγεί σε αλγοριθμικές τυποποιήσεις, ο επιπλέον φόρτος της κλήσης μια διαδικασίας ανά χαρακτήρα είναι σημαντικός σε σχέση με το κόστος της προσομοίωσης του DFA είτε μιλάμε για table-driven σαρωτή ή για direct-coded σαρωτή. Για να μειώσουμε το κόστος διαβάσματος ανά χαρακτήρα, μπορούμε να χρησιμοποιήσουμε έναν buffer στον οποίο θα επιστρέφεται ένα μεγαλύτερο κομμάτι της εισόδου και όχι μεμονωμένοι χαρακτήρες. Έτσι ο σαρωτής θα προσπελαύνει το buffer μέσω ενός δείκτη και όχι το αρχείο.

Το κόστος διαβάζοντας ένα γεμάτο buffer από χαρακτήρες έχει 2 τμήματα, ένα μεγάλο σταθερό κόστος και ένα μικρό ανά χαρακτήρα. Η μέθοδος με το δείκτη και τον buffer εξισορροπεί αυτά τα σταθερά κόστη σε σχέση με την ανάκτηση κατευθείαν από την είσοδο ένα - ένα χαρακτήρα. Ένας μεγαλύτερος buffer ελαχιστοποιεί αυτό το κόστος.

Κεφάλαιο 3

Συντακτική ανάλυση (Syntactical analysis)

Η *συντακτική ανάλυση* (parsing)[1] είναι η δεύτερη φάση στο εμπρός τμήμα του μεταγλωττιστή (compiler). Ένας συντακτικός αναλυτής (syntax analyzer ή parser) δουλεύει με το πρόγραμμα που έλαβε ως είσοδο όπως αυτό έχει διαμορφωθεί από το σαρωτή (scanner). Χρησιμοποιεί μια ροή λέξεων όπου κάθε λέξη περιγράφεται από τη συντακτική κατηγορία που ανήκει. Ο συντακτικός αναλυτής παράγει τη συντακτική δομή του προγράμματος ταιριάζοντας τις λέξεις στο μοντέλο γραμματικής της γλώσσας εισόδου.



Σχήμα 3.1: Συντακτικός αναλυτής

Το τυπικό πρόβλημα της ανάλυσης έχει μελετηθεί εκτενώς ως μέρος της θεωρίας τυπικών γλωσσών. Σε όλες τις τεχνικές, ο χρόνος είναι ανάλογος του μεγέθους του προγράμματος, παρόμοια με τη σάρωση (scanning).Σε αντίθεση με τους σαρωτές όπου το hand-coding είναι κοινό εργαλείο, οι παραγόμενοι συντακτικοί αναλυτές (generated parsers) είναι περισσότερο κοινοί από τους hand-

[1] **Parsing**: Δοθέντος μια ροής s από λέξεις και μιας γραμματικής G, βρίσκουμε μια παραγωγή στη G η οποία παράγει την s

coded συντακτικούς αναλυτές. Η βασική εργασία του αναλυτή (parser) είναι να αποφανθεί αν το πρόγραμμα εισόδου είναι ή όχι συντακτικά σωστό στην πηγαία γλώσσα. Για να το πετύχουμε αυτό, χρειαζόμαστε ένα μηχανισμό για τον καθορισμό του συντακτικού της γλώσσας εισόδου.

3.1 Context-Free γραμματικές

Για να περιγράψουμε τη σύνταξη μιας γλώσσας προγραμματισμού, χρειαζόμαστε μια πιο δυνατή σημειογραφία από τις κανονικές εκφράσεις (regular expressions). Η παραδοσιακή λύση είναι να χρησιμοποιήσουμε μια context-free γραμματική (context-free grammar - CFG)[2]. Μια context-free γραμματική, G, είναι ένα σύνολο κανόνων που περιγράφουν πως να σχηματίσουμε προτάσεις. Η συλλογή των προτάσεων που μπορεί να παραχθεί από τη γραμματική ορίζεται ως η γλώσσα που παράγεται από τη G. Το σύνολο των γλωσσών που παράγονται από context-free γραμματικές ονομάζονται context-free γλώσσες. Ας θεωρήσουμε την ακόλουθη γραμματική, η οποία ονομάζεται SN:

$$\begin{array}{rcl} \textit{SheepNoise} & \rightarrow & \texttt{baa}\ \textit{SheepNoise} \\ & | & \texttt{baa} \end{array}$$

Σχήμα 3.2: Γραμματική SN

Ο πρώτος κανόνας, ή *παραγωγή* (*production*)[3] διαβάζεται ως εξής: 'Ο κανόνας *SheepNoise* μπορεί να αναπτυχθεί στη λέξη baa ακολουθούμενη από περισσότερα *SheepNoise*.' Εδώ η λέξη *SheepNoise* είναι μια συντακτική μεταβλητή η οποία αναπαριστά ένα σύνολο από αλφαριθμητικά το οποίο μπορεί να παραχθεί από τη γραμματική. Μια τέτοια συντακτική μεταβλητή ονομάζεται μη τερματικό σύμβολο (*nonterminal symbol*)[4]. Κάθε λέξη στη γλώσσα που ορίζεται από τη γραμματική είναι ένα τερματικό σύμβολο (*terminal symbol*)[5]. Ο δεύτερος κανόνας διαβάζεται ως εξής: 'Η λέξη *SheepNoise* μπορεί επίσης να αναλυθεί στο αλφαριθμητικό baa.'

Η γραμματική που αναφέρθηκε παραπάνω είναι γραμμένη στη μορφή Backus-Naur Form (BNF).και χρησιμοποιείται συνήθως για την περιγραφή του συντακτικού γλωσσών προγραμματισμού. Αναπτύχθηκε από τον John Backus για την περιγραφή της γλώσσας προγραμματισμού ALGOL[24]. Επίσης, υπάρχει και μια πιο εκτεταμένη έκδοση της BNF σημειολογίας, η Extended Backus-Naur Form (EBNF) η οποία δημιουργήθηκε από τον Niklaus Wirth[25].

Για να κατανοήσουμε τη σχέση μεταξύ της γραμματικής SN και της γλώσσας L(SN), πρέπει να καθορίσουμε πως θα εφαρμόσουμε τους κανόνες στη γραμματική για να παράγουμε προτάσεις στην L(SN). Πρώτα πρέπει να ορίσουμε το

[2] **Context-free γραμματική**: Για μία γλώσσα L, η context-free γραμματική της ορίζει ένα σύνολο από αλφαριθμητικά σύμβολα τα οποία είναι έγκυρες προτάσεις στην L

[3] **Παραγωγή**: Κάθε κανόνας σε μία CFG ονομάζεται *παραγωγή*

[4] **Μη τερματικό σύμβολο**: Αναπαριστά μία πρόταση ή μέρος μιας πρότασης της γλώσσας η οποία περιγράφεται από τη γραμματική

[5] **Τερματικό σύμβολο**: Μια λέξη η οποία μπορεί να εμφανιστεί σε μία πρόταση

αρχικό σύμβολο (*goal symbol - start symbol*) της γραμματικής SN. Το αρχικό σύμβολο αναπαριστά το σύνολο με όλα τα αλφαριθμητικά στην L(SN). Συνεπώς δεν μπορεί να είναι κάποιο τερματικό σύμβολο. Πρέπει να είναι ένα από τα τερματικά σύμβολα που εισήχθησαν για να προσθέσουν αφαιρετικότητα και συγκεκριμένη δομή στη γλώσσα. Μιας και η SN έχει μόνο ένα μη τερματικό σύμβολο, το *SheepNoise* είναι το αρχικό σύμβολο (goal symbol).

Για να παραχθεί μία πρόταση πρέπει να ξεκινήσουμε με ένα αλφαριθμητικό που περιέχει μόνο το αρχικό σύμβολο της γραμματικής, *SheepNoise*. Επιλέγουμε ένα μη τερματικό σύμβολο *α*, στο αλφαριθμητικό, διαλέγουμε ένα γραμματικό κανόνα $\alpha \rightarrow \beta$ και ξαναγράφουμε το *α* με το *β*. Επαναλαμβάνουμε αυτή τη διαδικασία μέχρι το αλφαριθμητικό να περιέχει μόνο τερματικά σύμβολα. Ακολουθεί ένα μικρό παράδειγμα βάσει της γραμματικής που μόλις είδαμε.

Μια context-free γραμματική G αποτελείται από την τετράδα (T,NT,S,P) όπου:

- T είναι το σύνολο των τερματικών στοιχείων τα οποία εμφανίζονται στη γλώσσα L(G). Αυτά επιστρέφονται από το σαρωτή.
- NT είναι το σύνολο των μη-τερματικών στοιχείων τα οποία εμφανίζονται στις παραγωγές της γλώσσας. Τα μη-τερματικά στοιχεία είναι συντακτικές μεταβλητές που εμφανίστηκαν για να παρέχουν αφαιρετικότητα και σωστή δομή στις παραγωγές
- S είναι ένα μη-τερματικό στοιχείο που ονομάζεται και goal σύμβολο ή αρχικό σύμβολο της γραμματικής. Ουσιαστικά είναι ο αρχικός κανόνας της γραμματικής. Το S αναπαριστά το σύνολο των προτάσεων στη γλώσσα L(G)
- P είναι το σύνολο των παραγωγών της γραμματικής G. Κάθε κανόνας έχει τη μορφή $NT \rightarrow (T \bigcup NT)^{+}$ δηλαδή αντικαθιστά έναν απλό μη-τερματικό χαρακτήρα με ένα αλφαριθμητικό από ένα ή περισσότερα γραμματικά σύμβολα. Τα T, NT μπορούν να παραχθούν απευθείας από το P

3.1.1 Ένα πιο περίπλοκο παράδειγμα

Η γραμματική SheepNoise είναι αρκετά απλή για να παρουσιάσει την δύναμη και πολυπλοκότητα των CFG γραμματικών. Αντί αυτού, ας δούμε ένα παράδειγμα μιας αριθμητικής γραμματικής.

Έστω ότι έχουμε την εξής έκφραση: $(a+b)\times c$. Με τη βοήθεια της γραμματικής κάνουμε ανάπτυξη της έκφρασης με βάση την προτεραιότητα των κανόνων. Ο πίνακας στα αριστερά παρουσιάζει σε κάθε βήμα την αντικατάσταση των μη-τερματικών συμβόλων με την κατάλληλη εναλλακτική, σταδιακά από δεξιά προς τα αριστερά (rightmost derivation[6]). Η δενδροειδής μορφή που φαίνεται στα δεξιά ονομάζεται συντακτικό δέντρο (parse tree ή Concrete Syntax Tree - CST) και αναπαριστά την ανάπτυξη των κανόνων γραφικά.

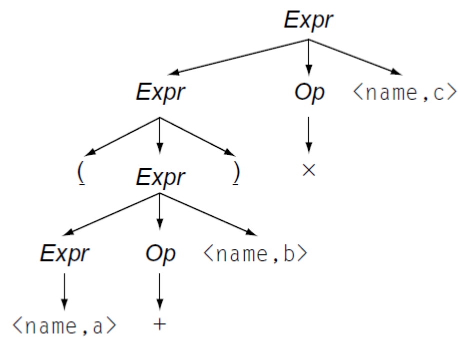
[6]**Rightmost derivation**: Μια παραγωγή η οποία αναπτύσσει σε κάθε βήμα το δεξιότερο μη-τερματικό σύμβολο

1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Σχήμα 3.3: Γραμματική αριθμητικών εκφράσεων

Rule	Sentential Form
	Expr
2	Expr Op name
6	Expr × name
1	(Expr) × name
2	(Expr Op name) × name
4	(Expr + name) × name
3	(name + name) × name

(α′) Ανάπτυξη κανόνων



(β′) Συντακτικό δέντρο

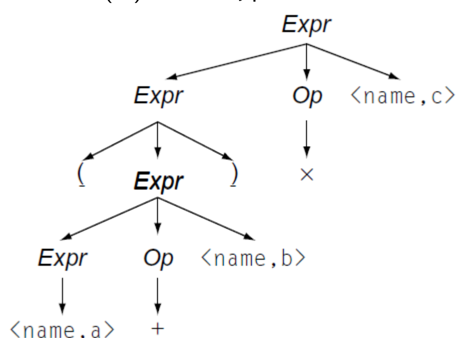
Σχήμα 3.4: Ανάπτυξη δεξιότερης παραγωγής

Η ανάπτυξη του $(a + b) \times c$ ξεκινάει από την ανάπτυξη του δεξιότερου μη τερματικού συμβόλου. Υπάρχουν όμως και άλλες επιλογές ανάπτυξης. Μια προφανής εναλλακτική είναι η ανάπτυξη του αριστερότερου μη τερματικού σύμβολου (leftmost derivation[7]). Χρησιμοποιώντας μια τέτοια μέθοδο παράγεται μια διαφορετική ακολουθία για την ίδια έκφραση. Παρακάτω φαίνεται το αποτέλεσμα αυτής της μεθόδου στο παράδειγμα που είδαμε προηγουμένως.

[7]**Leftmost derivation**: Μια παραγωγή η οποία αναπτύσσει σε κάθε βήμα το αριστερότερο μη-τερματικό σύμβολο

Rule	Sentential Form
	Expr
2	Expr Op name
1	(Expr) Op name
2	(Expr Op name) Op name
3	(name Op name) Op name
4	(name + name) Op name
6	(name + name) × name

(α′) Ανάπτυξη κανόνων



(β′) Συντακτικό δέντρο

Σχήμα 3.5: Ανάπτυξη αριστερότερης παραγωγής

Η αριστερότερες και δεξιότερες παραγωγές χρησιμοποιούν το ίδιο σύνολο κανόνων, απλά εφαρμόζουν αυτούς τους κανόνες με διαφορετική σειρά. Επειδή το συντακτικό δέντρο παρουσιάζει τους κανόνες που εφαρμόστηκαν αλλά όχι τη σειρά με την οποία εφαρμόστηκαν, και τα 2 συντακτικά δέντρα που είδαμε είναι ίδια. Από την οπτική του μεταγλωττιστή τώρα, είναι σημαντικό κάθε έκφραση να ορίζεται από μία και μοναδική παραγωγή (είτε αριστερότερη είτε δεξιότερη). Αν σε μια γραμματική υπάρχουν πολλαπλές παραγωγές για μία έκφραση τότε αυτή η γραμματική ονομάζεται *διφορούμενη* (ambiguous[8]). Μια διφορούμενη γραμματική μπορεί να παράγει πολλές διαφορετικές παραγωγές ή συντακτικά δέντρα.

Υπάρχουν 2 μέθοδοι κατασκευής του συντακτικού δέντρου:

- Top-down συντακτικοί αναλυτές: Ξεκινούν από τη ρίζα και φτάνουν έως τα φύλλα του δέντρου. Σε κάθε βήμα, ο συντακτικός αναλυτής (parser) διαλέγει έναν μη-τερματικό κόμβο τον οποίο αναπτύσσει σχηματίζοντας έτσι ένα υποδέντρο το οποίο αναπαριστά το δεξί μέρος μιας παραγωγής του μη-τερματικού συμβόλου.

- Bottom-up συντακτικοί αναλυτές: Ξεκινούν από τα φύλλα και αναπτύσσουν το δέντρο ως τη ρίζα του. Σε κάθε βήμα, ο συντακτικός αναλυτής

[8] **Διφορούμενη γραμματική**: Η γραμματική G είναι διφορούμενη αν μια πρόταση στη γλώσσα L(G) έχει περισσότερες από μία δεξιότερες ή αριστερότερες παραγωγές

(parser) αναγνωρίζει ένα αλφαριθμητικό το οποίο ταιριάζει με το δεξί μέρος μιας παραγωγής. Με αυτόν τον τρόπο δημιουργεί έναν κόμβο τον οποίο συνδέει στο δέντρο και έτσι συνεχίζει παραπάνω.

3.2 Top-Down Συντακτική ανάλυση (Top-Down parsing)

Ένας top-down συντακτικός αναλυτής ξεκινά από τη ρίζα του συντακτικού δέντρου και επεκτείνει το δέντρο προς τα κάτω μέχρις ότου τα φύλλα του ταιριάζουν με τις λέξεις που επιστρέφει ο λεκτικός αναλυτής (scanner). Σε κάθε σημείο ο συντακτικός αναλυτής επιλέγει ένα μη-τερματικό σύμβολο στο κατώτερο όριο του δέντρου και το επεκτείνει προσθέτοντας παιδιά τα οποία αντιστοιχούν στο δεξί μέρος μιας παραγωγής για ένα μη-τερματικό σύμβολο. Δεν μπορεί να επεκτείνει αυτό το όριο από ένα τερματικό σύμβολο. Η διαδικασία συνεχίζεται μέχρις ότου:

- Τα φύλλα του δέντρου αποτελούνται μόνο από τερματικά σύμβολα και επίσης η είσοδος έχει εξαντληθεί.
- Μια πλήρης αναντιστοιχία εμφανίζεται μεταξύ του συντακτικού δέντρου και της εισόδου.

Στην πρώτη περίπτωση, έχουμε επιτυχία. Στη δεύτερη περίπτωση, 2 καταστάσεις είναι πιθανές. Ο συντακτικός αναλυτής διάλεξε λάθος παραγωγή να αναπτύξει σε κάποιο προηγούμενο βήμα στη διαδικασία και έτσι πρέπει να γυρίσει πίσω (backtrack), επανεξετάζοντας τις προηγούμενες αποφάσεις του. Αν η είσοδος που δόθηκε είναι έγκυρη τότε με την οπισθοδρόμηση, ο συντακτικός αναλυτής θα αναπτύξει τη σωστή παραγωγή και θα κατασκευάσει το σωστό συντακτικό δέντρο. Αν τώρα η είσοδος δεν είναι έγκυρη τότε η οπισθοδρόμηση θα αποτύχει και ο αναλυτής θα ανακοινώσει το συντακτικό λάθος στο χρήστη. Η οπισθοδρόμηση αυξάνει το κόστος της συντακτικής ανάλυσης. Στην πράξη είναι ένας ακριβός δρόμος ανακάλυψης συντακτικών λαθών.

Το ANTLR είναι μια γεννήτρια παραγωγής top-down recursive descent LL(k) συντακτικών αναλυτών. Περισσότερες λεπτομέρειες για την αρχιτεκτονική και λειτουργία του ANTLR αναφέρονται στην παράγραφο 4.2.

3.2.1 Κατηγορίες Context-free γραμματικών

Ένα μεγάλο υποσύνολο των context-free γραμματικών μπορεί να αναλυθεί χωρίς οπισθοδρόμηση.

Οι LR(1) γραμματικές περιλαμβάνουν ένα μεγάλο σύνολο από διφορούμενες CFGs. Αναλύονται από κάτω προς τα πάνω (bottom-up) κοιτώντας το πολύ 1 λέξη μπροστά από το τρέχων σύμβολο.

Οι LL(1) γραμματικές είναι ένα υποσύνολο των LR(1) γραμματικών. Αναλύονται από πάνω προς τα κάτω κοιτώντας το πολύ 1 λέξη μπροστά από το

τρέχων σύμβολο. Οι LL(1) γραμματικές μπορούν να αναλυθούν είτε με hand-coded αναδρομικούς-καθοδικούς (recursive-descent) συντακτικούς αναλυτές είτε με παραγόμενους συντακτικούς αναλυτές.

3.3 Bottom-Up Συντακτική ανάλυση (Bottom-Up parsing)

Οι bottom-up συντακτικοί αναλυτές χτίζουν ένα συντακτικό δέντρο ξεκινώντας από τα φύλλα και καταλήγοντας στη ρίζα του. Ο αναλυτής κατασκευάζει ένα κόμβο φύλλο για κάθε λέξη που επιστρέφεται από τον σαρωτή. Για να κατασκευάσει μια παραγωγή, ο αναλυτής προσθέτει επίπεδα από μη τερματικά σύμβολα πάνω από τα φύλλα τα οποία καθορίζονται από τη γραμματική και από το μερικώς υλοποιημένο δέντρο.

Οι κόμβοι πάνω από τα φύλλα κωδικοποιούν όλη τη γνώση που έχει συλλέξει ο συντακτικός αναλυτής. Για να προχωρήσει ο αναλυτής κοιτάει αν κάποιο μέρος ενός αλφαριθμητικού ταιριάζει με το δεξί μέρος μιας παραγωγής. Η διαδικασία συνεχίζεται μέχρις ότου:

- Να φτάσει στη ρίζα της γραμματικής (goal symbol).
- Να εντοπίσει λάθος.

Στην πρώτη περίπτωση ο αναλυτής κατανάλωσε όλες τις λέξεις της εισόδου οπότε έχουμε επιτυχία. Στη δεύτερη περίπτωση ο αναλυτής δεν μπορεί να κατασκευάσει μια παραγωγή και θα πρέπει να ανακοινώσει την αποτυχία.

Ο bison είναι μιας γεννήτρια παραγωγής bottom-up LALR(1) συντακτικών αναλυτών. Τα LALR σημαίνουν ότι ο bison παράγει ένα parser ο οποίος ξεκινάει την ανάλυση από αριστερά προς τα δεξιά, χρησιμοποιώντας δεξιότερες παραγωγές κοιτώντας μπροστά κατά 1 χαρακτήρα. Ο συντακτικός αναλυτής που παράγεται είναι table-driven, περισσότερα στην παράγραφο 1.1 και χρησιμοποιεί τι λεγόμενες ενέργειες shift και reduce.

Ο parser κατασκευάζει ένα συντακτικό δέντρο σταδιακά, από κάτω προς τα πάνω, αριστερά προς τα δεξιά χωρίς να μαντεύει ή να γυρίζει πίσω. Σε κάθε σημείο της τους περάσματος, ο συντακτικός αναλυτής έχει συσσωρεύσει μία λίστα από υποδέντρα ή φράσεις της εισόδου τα οποία έχουν αναλυθεί.

Ένα βήμα shift μετακινεί το δείκτη του ρεύματος εισόδου κατά ένα σύμβολο. Το σύμβολο που μόλις μετακινήθηκε γίνεται ένας νέος κόμβος στο συντακτικό δέντρο. Ένα βήμα reduce εφαρμόζει έναν ολοκληρωμένο γραμματικό κανόνα σε ένα από τα υποδέντρα ενώνοντάς τα μαζί σε ένα δέντρο με μία κοινή ρίζα.

Αν η είσοδος δεν έχει συντακτικά λάθη, τότε ο parser συνεχίζει με αυτά τα βήματα μέχρι να καταναλωθεί όλη η είσοδος και όλα τα συντακτικά δέντρα έχει γίνει reduce σε ένα μοναδικό δέντρο το οποίο αναπαριστά όλο το ρεύμα εισόδου.

Κεφάλαιο 4

Εργαλεία/Γλώσσες που χρησιμοποιήθηκαν (Tools/Languages used)

Για αυτή την εργασία χρησιμοποιήθηκαν διάφορα εργαλεία, από την κατασκευή της γραμματικής μέχρι την γραφική αναπαράσταση των δεδομένων. Πιο συγκεκριμένα αυτά είναι:

- Γλώσσα προγραμματισμού C#
- Γεννήτρια συντακτικών αναλυτών ANTLR
- Εργαλείο παραγωγής γράφων Graphviz (DOT)
- Γλώσσα προγραμματισμού MATLAB

Στα παρακάτω τμήματα όλα τα εργαλεία θα αναλυθούν διεξοδικά αναφορικά με τις δυνατότητες τους και τη χρήση τους.

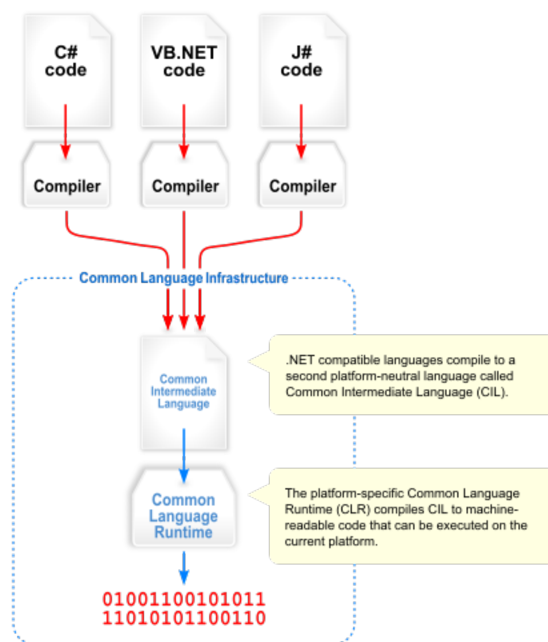
4.1 Γλώσσα προγραμματισμού C#

Πρώτα ας ξεκινήσουμε από το βασικό εργαλείο το οποίο είναι η γλώσσα προγραμματισμού που χρησιμοποιήθηκε για την ανάπτυξη η οποία είναι η C#. Είναι μια αντικειμενοστραφής γλώσσα προγραμματισμού η οποία αναπτύχθηκε από τη Microsoft πριν από 16 χρόνια και η τελευταία σταθερή της έκδοση είναι η 6.0[9]. Μερικά από τα χαρακτηριστικά της γλώσσας είναι:

Portability

Σχεδιαστικά, η C# χρησιμοποιεί το Common Language Infrastructure[10] το οποίο περιγράφει τον εκτελέσιμο κώδικα και το περιβάλλον εκτέλεσης

επιτρέποντας έτσι σε διαφορετικές γλώσσες προγραμματισμού υψηλού επιπέδου να χρησιμοποιηθούν σε διαφορετικά συστήματα χωρίς να ξαναγραφτούν ανάλογα με την αρχιτεκτονική του συστήματος. Το παρακάτω σχήμα περιγράφει αυτή τη διαδικασία.



Σχήμα 4.1: Επισκόπηση του CLI[11]

Meta programming
: Η χρήση attributes επιτρέπει αντίστοιχη λειτουργικότητα με τις directives του προ επεξεργαστή της C.

Property
: Οι properties είναι κάτι αντίστοιχο με το ζευγάρι μεθόδων get και set της Java. Έτσι γίνεται πιο εύκολα η ανάθεση τιμής σε μία μεταβλητή καθώς και η ανάγνωση αυτής.

Namespace
: Το αντίστοιχο package της Java.

Memory access
: Η C# επιτρέπει την χρήση δεικτών μόνο όταν βρίσκονται μέσα στο block `unsafe` .

Polymorphism
: Δεν υποστηρίζει πολλαπλή κληρονομικότητα (multiple inheritance) όπως η

C++. Αντίθετα μια κλάση μπορεί να υλοποιήσει περισσότερα από ένα interfaces. Επίσης υποστηρίζει τη δυνατότητα operator overloading.

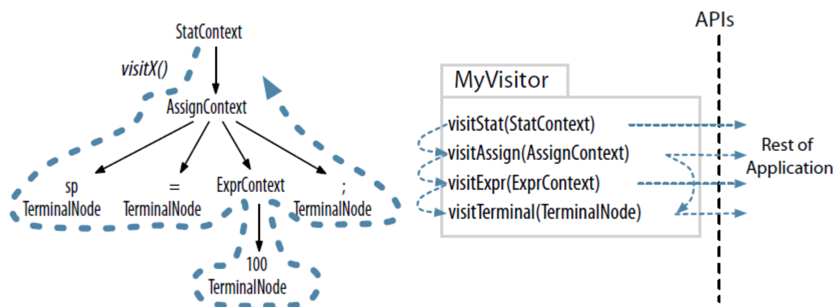
4.2 Γεννήτρια συντακτικών αναλυτών ANTLR

Το ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition)[12] είναι μία δυνατή γεννήτρια συντακτικών αναλυτών (parser generator) για ανάγνωση, επεξεργασία, εκτέλεση ή μετάφραση δομημένου κειμένου ή δυαδικών αρχείων. Χρησιμοποιείται ευρέως για την κατασκευή γλωσσών, εργαλείων και programming frameworks. Από μια γραμματική, το ANTLR παράγει έναν συντακτικό αναλυτή ο οποίος μπορεί να κατασκευάσει ένα συντακτικό δέντρο το οποίο στη συνέχεια μπορούμε να διασχίσουμε για να ανακτήσουμε όποια πληροφορία θελήσουμε. Η διάσχιση του δέντρου γίνετε με τη χρήση 2 μοτίβων:

- Ακροατή (Listener)
- Επισκέπτη (Visitor)

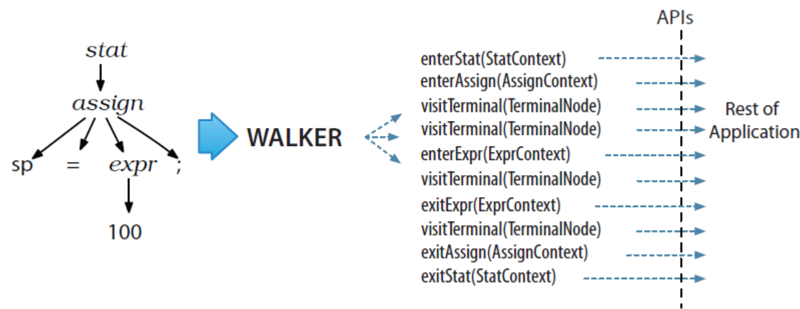
Το ANTLR παράγει top-down συντακτικούς αναλυτές οι οποίοι χρησιμοποιούν τη στρατηγική Adaptive LL(*) (ALL(*))[13] που μπορεί να χρησιμοποιήσει όλα τα υπόλοιπα σύμβολα για να πάρει τις αποφάσεις που χρειάζονται για μια επιτυχή αναγνώριση. Οι top-down συντακτικοί αναλυτές, ξεκινούν από τη ρίζα του δέντρου (goal symbol) και προχωρούν προς τα φύλλα του δέντρου.

Ξεκινώντας με το μοτίβο επισκέπτη, οι κόμβοι του δέντρου είναι αντικείμενα κλάσεων προερχόμενες από μια κύρια γονική αφηρημένη κλάση (abstract class). Βάση της κληρονομικότητας/πολυμορφίας επιτυγχάνεται σάρωση του δέντρου και επεξεργασία της πληροφορίας του καλώντας την αντίστοιχη με την επεξεργασία ρουτίνα της κύριας αφηρημένης κλάσης. Έτσι μια εικονική ρουτίνα (virtual method) που υπάρχει στην κύρια αφηρημένη κλάση μπορεί να παρακαμφθεί (overridden) στις υποκλάσεις (subclasses) και να καλείται από τον κώδικα του κάθε κόμβου εξασφαλίζοντας πάντα ότι θα καλείται η ρουτίνα που αντιστοιχεί στον τύπο της τρέχουσας κλάσης.



Σχήμα 4.2: Διάσχιση δέντρου με το μοτίβο ακροατή (listener pattern)

Ο μηχανισμός σάρωσης του συντακτικού δέντρου από το μοτίβο ακροατή και το γεγονός ότι δεν επιστρέφεται τιμή από τις μεθόδους της κλάσης του δεν δίνει την δυνατότητα διατήρησης πληροφορίας που συσσωρεύεται κατά την διαγραφή του δέντρου. Το ANTLR παράγει εξ' ορισμού ένα interface βάσει του μοτίβου ακροατή το οποίο ανταποκρίνεται σε διάφορα γεγονότα (events) που ενεργοποιούνται από τον εσωτερικό αλγόριθμο διάσχισης του δέντρου, οποίος είναι ο DFS (Depth First Search) αλγόριθμος (Διάσχιση κατά βάθος)[12].



Σχήμα 4.3: Διάσχιση δέντρου με το μοτίβο επισκέπτη (visitor pattern)

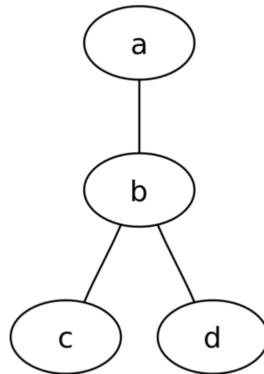
Η έκδοση του ANTLR που χρησιμοποιήθηκε είναι η **4.4.1-alpha001**.

4.3 Εργαλείο παραγωγής γράφων Graphviz

Το Graphviz είναι ένα εργαλείο ανοικτού κώδικα το οποίο ξεκίνησε από τα εργαστήρια AT&T για το σχεδιασμό γράφων στη γλώσσα DOT. Η DOT[14] είναι μια γλώσσα περιγραφής γράφων. Επιτρέπει κατευθυνόμενους και μη κατευθυνόμενους γράφους. Μερικά παραδείγματα φαίνονται παρακάτω:

Μη κατευθυνόμενος γράφος

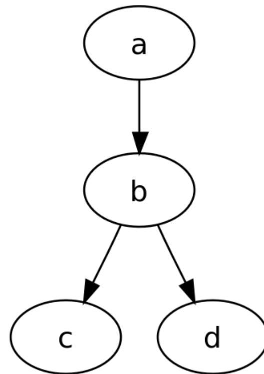
```
// The graph name and the semicolons are optional
graph graphname {
        a -- b -- c;
        b -- d;
}
```



Σχήμα 4.4: Μη κατευθυνόμενος γράφος[15]

Κατευθυνόμενος γράφος

```
digraph graphname {
        a -> b -> c;
        b -> d;
}
```

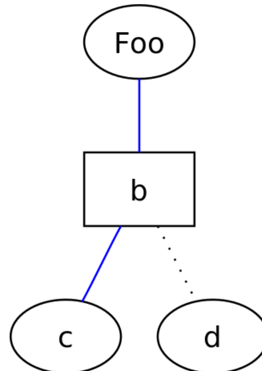


Σχήμα 4.5: Κατευθυνόμενος γράφος[15]

Επίσης παρέχει και διάφορα attributes τα οποία επηρεάζουν και την εμφάνιση του γράφου, όπως είναι το χρώμα, σχήμα και η εμφάνιση των ακμών και κόμβων:

```
graph graphname {
        // This attribute applies to the graph itself
        size="1,1";
        // The label attribute can be used to change the label of a node
        a [label="Foo"];
        // Here, the node shape is changed.
```

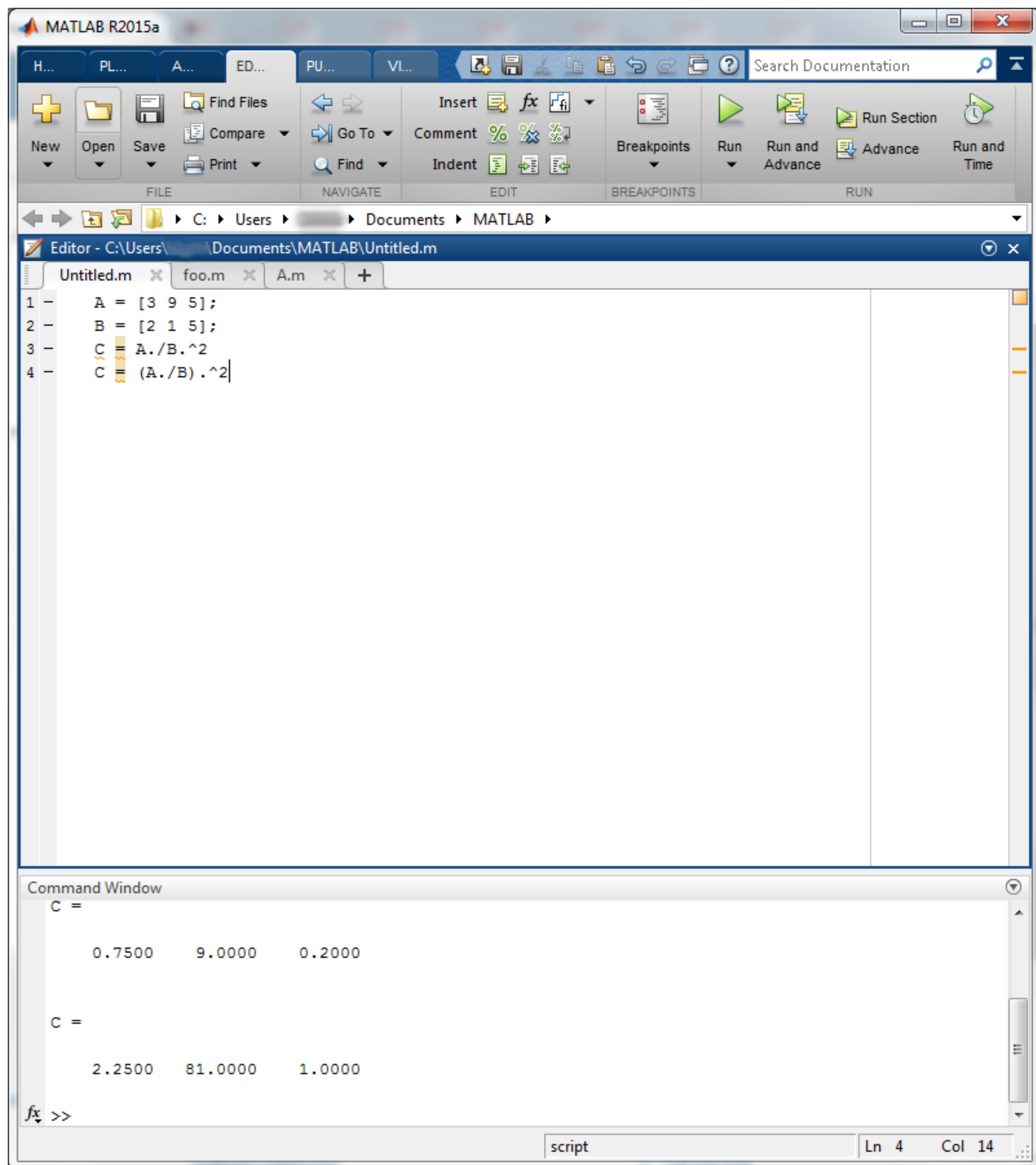
```
        b [shape=box];
        // These edges both have different line properties
        a -- b -- c [color=blue];
        b -- d [style=dotted];
}
```



Σχήμα 4.6: Γράφος με attributes[15]

4.4 Γλώσσα προγραμματισμού MATLAB

Η γλώσσα προγραμματισμού MATLAB είναι μία scripting γλώσσα που χρησιμοποιείται στην ομώνυμη εφαρμογή. Το MATLAB (**mat**rix **lab**oratory) (εφαρμογή)[16] είναι ένα περιβάλλον υπολογιστικής αριθμητικής μέσω το οποίου ο χρήστης μπορεί να πραγματοποιήσει υπολογισμούς μεταξύ πινάκων, γραφική αναπαράσταση δεδομένων, υλοποίηση διαφόρων αλγορίθμων ακόμη και δημιουργία γραφικών περιβαλλόντων. Επιτρέπει επίσης την επικοινωνία με προγράμματα γραμμένα σε άλλες γλώσσες προγραμματισμού συμπεριλαμβανομένων των C, C++, Java, Fortran και Python.



Σχήμα 4.7: Γραφικό περιβάλλον (GUI) του MATLAB R2015a

Ο Cleve Moler, πρόεδρος του τμήματος επιστήμης υπολογιστών του πανεπιστημίου του New Mexico ξεκίνησε να αναπτύσσει το MATLAB στα τέλη της δεκαετίας του 1970[17]. Αναγνωρίζοντας την εμπορική του δυναμική, αργότερα σε συνεργασία με τον Steve Bangert ξανά έγραψαν το MATLAB σε C και ίδρυσαν την εταιρία MathWorks το 1984.

Αν και το MATLAB στοχεύει κυρίως στους αριθμητικούς υπολογισμούς, χάρη στην επεκτασιμότητα του μέσω εργαλειοθηκών toolboxes) αποκτά τεράστιες δυνατότητες και σε άλλα επιστημονικά πεδία. Για παράδειγμα μπορεί να χρησιμοποιηθεί για ψηφιακή επεξεργασία σήματος, εικόνας και βίντεο, δημιουργία νευρωνικών δικτύων καθώς και ρομποτικής. Ακόμη, χρησιμοποιείται εκτενώς για εκπαιδευτικούς σκοπούς, πιο συγκεκριμένα για την εκμάθηση γραμμικής άλγεβρας.

Η πιο απλή χρήση του MATLAB γίνεται μέσω της γραμμής εντολών, που βρίσκεται μέσα στο κεντρικό παράθυρο, και λειτουργεί σαν ένα ενεργό κέλυφος (shell) ή μέσω της εκτέλεσης αρχείων που περιέχουν κώδικα MATLAB και ονομάζονται script αρχεία. Αυτά τα αρχεία έχουν κατάληξη **.m**[18].

Η γλώσσα προγραμματισμού MATLAB είναι μια weakly typed γλώσσα που σημαίνει ότι οι τύποι δεδομένων μετατρέπονται εμμέσως[19] επειδή στις μεταβλητές μπορούν να ανατεθούν τιμές χωρίς να δηλώνεται ο τύπος τους και συνεπώς ο τύπος μπορεί να αλλάζει στη συνέχεια του προγράμματος. Ακολουθεί ένα παράδειγμα.

```
>> g = 25
g =
    25

>> g = 'cat'
g =
    cat
```

Ένας απλός πίνακας ορίζεται χρησιμοποιώντας την ακόλουθη σύνταξη:

```
start : increment : end
```

Το παρακάτω παράδειγμα ορίζει έναν πίνακα, που ονομάζεται matrix, **από** 1 **έως και** 10. Το βήμα αύξησης είναι 1.

```
>> matrix = 1:1:10
matrix =
    1 2 3 4 5 6 7 8 9 10
```

Η δεικτοδότηση (indexing) του πίνακα ξεκινάει από το 1, σε αντίθεση με πολλές γλώσσες προγραμματισμού (C, C++, C#, Java) που ξεκινάνε από το 0.

```
>> matrix = 1:1:10
matrix =
    1 2 3 4 5 6 7 8 9 10

>> matrix(1)
matrix =
    1
```

Η τελευταία έκδοση του προγράμματος είναι η **R2015b**. Η έκδοση που χρησιμοποιήθηκε στην παρούσα εργασία είναι η **R2015a**.

Κεφάλαιο 5

Κατασκευή συντακτικού αναλυτή για την γλώσσα προγραμματισμού MATLAB (MATLAB parser construction)

Τα προγράμματα που αναγνωρίζουν γλώσσες ονομάζονται συντακτικοί αναλυτές (syntax analyzers - parsers). Ως συντακτικό αναφερόμαστε στους κανόνες που ορίζουν τη γλώσσα. Η γραμματική είναι απλά ένα σύνολο από κανόνες και κάθε ένας εκφράζει τη δομής μιας φράσης και γενικά ολόκληρης της γλώσσας. Το ANTLR μεταφράζει μια γραμματική σε ένα συντακτικό αναλυτή.

Σε αυτό το κεφάλαιο θα γίνει μια ανάλυση σχετικά με τη γραμματική που κατασκευάστηκε για την αναγνώριση της γλώσσας προγραμματισμού MATLAB. Για την κατασκευή της γραμματικής χρησιμοποιήθηκε ένα και μόνο αρχείο και δεν έγινε χρήση της δυνατότητας που δίνει το ANTLR, της διάσπασης της γραμματικής στο κομμάτι του parser και του lexer.

5.1 Λεκτικός αναλυτής (Lexer)

Η διαδικασία ομαδοποίησης χαρακτήρων σε λέξεις ή σύμβολα (tokens) ονομάζεται λεξική ανάλυση lexical analysis ή πιο απλά tokenizing. Το πρόγραμμα που κάνει αυτή τη διαδικασία ονομάζεται λεκτικός αναλυτής (lexical analyzer - lexer). O lexer χωρίζει παρόμοια token σε κατηγορίες, π.χ. INT, DOUBLE, IDENTIFIER κλπ. Τα tokens αποτελούνται από τουλάχιστον 2 κομμάτια πληροφορίας:

- Ο τύπος του token

• Το κείμενο το οποίο αναγνωρίζεται από τον lexer γι΄ αυτό το token

Στόχος ενός λεκτικού αναλυτή είναι να μας επιστρέψει μία ακολουθία από tokens.

Θα ξεκινήσουμε πρώτα από τον λεκτικό αναλυτή καθώς εδώ γίνεται μια αρκετά μεγάλη προ επεξεργασία της εισόδου έτσι ώστε στη συνέχεια να έχουμε επιτυχή αναγνώριση συγκεκριμένων δομών της γλώσσας όπως είναι για παράδειγμα οι πίνακες (matrices) ή ακόμα και οι δεκαδικοί αριθμοί.

Στο κομμάτι του λεκτικού αναλυτή πραγματοποιείται η δήλωση όλων των δεσμευμένων λέξεων της γλώσσας, των τελεστών καθώς και των αριθμών (int, double, float κλπ) ή αλφαριθμητικών. Επίσης δηλώνονται οι χαρακτήρες του κενού (whitespace) και της αλλαγής γραμμής (CRLF ή LF). Ειδικά τα τελευταία είναι πολύ σημαντικά και αποτελούσαν το μείζων πρόβλημα δυσκολίας αναγνώρισης συγκεκριμένων δομών της γλώσσας.

Οι δηλώσεις στον λεξερ έχουν την εξής μορφή:

```
ALIAS : ‘regular expression or actual value’;
```

Το ALIAS είναι ένα συμβολικό όνομα το οποίο μπορεί να χρησιμοποιηθεί ξανά μέσα στον lexer ή στον parser. Αυτό το όνομα αντιστοιχεί σε μία κανονική έκφραση ή σε μία τιμή και είναι υπεύθυνο για την αναγνώριση ενός token. Για την αναγνώριση αριθμών χρησιμοποιήθηκαν κανονικές εκφράσεις σε συνδυασμό με διάφορες συναρτήσεις οι οποίες αναφέρονται παρακάτω.

Πέρα από την απλή δήλωση όλων των tokens, το ANTLR μας δίνει τη δυνατότητα να εισάγουμε διάφορες ενέργειες (actions) οι οποίες συνοδεύουν κάθε κανόνα και χρησιμοποιούνται για προ επεξεργασία στο επίπεδο του lexer πριν περάσουν τα δεδομένα στον parser. Αυτές είναι συνήθως λίγων γραμμών κυρίως για λόγους αναγνωσιμότητας αλλά και δυσκολίας κάποιες φορές. Για μεγαλύτερης έκτασης κώδικα, υπάρχουν ειδικές περιοχές μέσα στη γραμματική.

Καθότι η γραμματική μας είναι συνδυαστική πρέπει να μπορούμε να ξεχωρίσουμε ποιες ενέργειες αφορούν τον parser και ποιες τον lexer. Αυτό γίνετε με τις παρακάτω 2 περιοχές:

```
// Parser actions
@parser::members {

}

// Lexer actions
@lexer::members {

}
```

5.1.1 Ενέργειες υπό μορφή συναρτήσεων

Όσον αφορά την δεύτερη περιοχή (του λεκτικού αναλυτή), εκεί περιέχεται ένας μεγάλος όγκος συναρτήσεων ο οποίος αφορά:

1. Έλεγχος τελεστών μέσα σε πίνακα
2. Διαχωρισμός δεκαδικών αριθμών και τελεστών που περιέχουν το χαρακτήρα '.' (τελεία)
3. Έλεγχος μοναδιαίων τελεστών στην αρχή ενός πίνακα
4. Συναρτήσεις pragma[1]

Στην περίπτωση **1** υπάρχει μία συνάρτηση η οποία καλείται αμέσως μόλις συναντήσει κάποιον αριθμό ή ένα αναγνωριστικό το οποίο βρίσκεται μέσα σε πίνακα με σκοπό να διαχειριστεί τα κενά (whitespace). Για παράδειγμα, αν υπάρχουν μοναδιαίοι τελεστές πρέπει να χειριστούμε τα κενά διαφορετικά.

```
[1 + 1]    1 element  (toss whitespace)
[1 +1]     2 elements (preserve whitespace)
[1+ 1]     1 element  (toss whitespace)

[1 + 1 + 1]   1 element  (toss whitespace)
[1 +1+1]      2 elements (preserve whitespace)
[1 +1 +1]     3 elements (preserve whitespace)

[1 + 1 1 + 1]  2 elements (toss whitespace between operators)
[1 +1 1 + 1]   3 elements (preserve whitespace between the first
               two elements then toss whitespace between operator)
[1 +1 1 +1]    4 elements (preserve whitespace)
```

Στην περίπτωση **2** γίνεται διαχωρισμός μεταξύ ενός τελεστή που περιέχει το χαρακτήρα '.' (τελεία) και των δεκαδικών αριθμών. Υπάρχουν περιπτώσεις όπου υπήρχε πρόβλημα στην αναγνώριση ενός δεκαδικού αριθμού όταν εμφανιζόταν ένας τέτοιος τελεστής. Τέτοιου είδους τελεστές φαίνονται παρακάτω:

```
Power           (.^)
Multiplication  (.*)
Right division  (./)
Left division   (.\)
Transpose       (.')
```

Το συγκεκριμένο πρόβλημα εμφανίζεται είτε είμαστε μέσα σε πίνακα είτε όχι. Για παράδειγμα, έστω ότι έχουμε την παρακάτω έκφραση η οποία είναι ένας πίνακας:

```
[1. ^ 1. +1]
```

[1] Δεν έχουν κάποια σχέση με τις αντίστοιχες συναρτήσεις pragma της γλώσσας MATLAB. Χρησιμοποιούνται κυρίως στο type-checking για να διευκολυνθεί το code generation

Ο πρώτος αριθμός είναι δεκαδικός, ακολουθεί ο τελεστής ύψωσης σε δύναμη '^', ένας δεύτερος δεκαδικός αριθμός και τέλος ένας τρίτος αριθμός ο οποίος όμως είναι μια μοναδιαία έκφραση καθώς προηγείται ο τελεστής της πρόσθεσης '+'.

Από την άλλη έχουμε και μια έκφραση εκτός πίνακα:

```
.0 ^ .0 +1
```

Εδώ τα κενά αγνοούνται από προεπιλογή, συνεπώς εδώ το μόνο που πρέπει να προσέξουμε είναι η διαχείριση του τελεστή της ύψωσης σε δύναμη.

Στην περίπτωση **3** ελέγχουμε αν υπάρχουν κενοί χαρακτήρες μεταξύ ενός μοναδιαίου τελεστή και μιας έκφρασης μόνο στην αρχή του πίνακα έτσι ώστε να εξακριβώσουμε αν υφίσταται κάποια μοναδιαία έκφραση. Ένα παράδειγμα είναι η παρακάτω περίπτωση:

```
[+ 4 15i]     (2 elements) Between the + and 4 the space is skipped
```

Τέλος στην περίπτωση **4** ψάχνουμε, αν υπάρχουν, συναρτήσεις pragma οι οποίες χρησιμοποιούνται επικουρικά για την καλύτερη αναγνώριση της εισόδου. Για παράδειγμα, ο χρήστης μπορεί να δηλώσει συγκεκριμένες εκφράσεις μέσω των συναρτήσεων pragma δίνοντας τη δυνατότητα στον μεταγλωττιστή να γνωρίζει εκ των προτέρων τι τύπου είναι συγκεκριμένες μεταβλητές χωρίς να κάνει περισσότερα περάσματα για να κατανοήσει την έκφραση. Χρησιμοποιούμε 2 διαφορετικούς τρόπους για την συγκεκριμένη υλοποίηση.

Ο πρώτος χρησιμοποιεί μια παρόμοια σύνταξη με αυτή του MATLAB για τις δικές του συναρτήσεις `pragma` :

```
%#pragma_identifier(arg1, arg2, arg3,... )
```

Ο δεύτερος τρόπος είναι μια κανονική συνάρτηση μόνο που χρησιμοποιεί το πρόθεμα "`pragma`" για να ξεχωρίζει από τις κανονικές συναρτήσεις. Οι συναρτήσεις `pragma` χρησιμοποιούνται από τον μεταγλώττιστη και μόνον από αυτόν.

```
pragma_identifier(arg1, arg2, arg3,... )
```

5.1.2 Semantic predicates

Πέρα από αυτές τις ενέργειες που εφαρμόζονται κυρίως σε πίνακες και έχουν μεγάλη έκταση από άποψη κώδικα, υπάρχουν και άλλες υπό τη μορφή διακοπτών που ονομάζονται *semantic predicates* οι οποίες μας επιτρέπουν να διακόψουμε επιλεκτικά συγκεκριμένα κομμάτια της γραμματικής. Είναι λογικές εκφράσεις (boolean expressions) που έχουν την δυνατότητα να περιορίζουν τον αριθμό των επιλογών που βλέπει ένας συντακτικός αναλυτής. Αυτό εν τέλει αυξάνει την αποδοτικότητα του parser.

Υπάρχουν 2 κοινές χρήσεις των semantic predicates:

- Να διευκολύνει τον συντακτικό αναλυτή να διαχειριστεί πολλαπλές, λίγο διαφορετικές διαλέκτους της ίδιας γλώσσας.
- Να λύσει τυχόν ασάφειες μέσα στη γραμματική. Σε μερικές γλώσσες, μία συντακτική δομή μπορεί να μεταφραστεί σε πολλά διαφορετικά πράγματα. Κατά συνέπεια τα predicates μας δίνουν τη δυνατότητα να επιλέξουμε ανάμεσα σε πολλές εναλλακτικές για μία φράση της εισόδου.

Ένα παράδειγμα χρήσης των semantic predicates είναι το πως θα γίνει ο διαχωρισμός του τελεστή ανάστροφου πίνακα (transpose operator (')) και ενός αλφαριθμητικού ("this is a string"). Για παράδειγμα:

```
[1 2 3]'              Transpose operator

'this is a string'    String

1 '                   String (!)
                      (there is a space between the number and the ')
```

5.1.3 Inline ενέργειες

Εκτός από τις παραπάνω περιπτώσεις, έχουμε χρησιμοποιήσει κώδικα υπό την μορφή ενεργειών (actions) οι οποίος εφαρμόζεται μετά την κανονική έκφραση ή τιμή που αντιστοιχεί στο εκάστοτε ALIAS. Για να εκτελεστούν αυτές οι ενέργειες πρέπει να έχουμε επιτυχή αναγνώριση. Για παράδειγμα έχουμε μερικές τέτοιες περιπτώσεις:

1. Χειρισμός κενών χαρακτήρων
2. Αρχικοποίηση flags και κλήση συναρτήσεων
3. Διαχείριση εμφωλευμένων δομών

Τα κενά παίζουν σημαντικό ρόλο κυρίως μέσα σε πίνακες όπου επηρεάζουν τον τρόπο αναγνώρισης μιας έκφρασης. Χρησιμοποιούνται συνήθως για τον διαχωρισμό στοιχείων σε ένα πίνακα. Συνεπώς, χρειαζόμαστε έναν τρόπο έτσι ώστε να γνωρίζουμε πότε πρέπει να διατηρούμε τους κενούς χαρακτήρες και πότε όχι. Αυτό γίνεται με τη χρήση ειδικών σημαιών (flags) οι οποίες ενεργοποιούνται όταν είμαστε μέσα σε ένα πίνακα ή όταν έχουμε συναντήσει μια έκφραση μέσα σε ένα πίνακα. Όταν μια σημαία ενεργοποιείται συνήθως σημαίνει ότι πρέπει να πετάξουμε ό,τι κενά συναντήσουμε μέχρι το σημείο όπου θα απενεργοποιηθεί ξανά.

Η δεύτερη περίπτωση, αφορά κυρίως πρωταρχικούς τύπους δεδομένων (integer, float κλπ) όπου εκεί συνήθως γίνεται η αρχικοποίηση ορισμένων flags και η κλήση διάφορων συναρτήσεων (όπως αναφέρθηκαν παραπάνω) σχετικά με τη διαχείριση κενών χαρακτήρων.

Τέλος, είναι σημαντικό να γνωρίζουμε το πότε μπαίνουμε ή βγαίνουμε από μία έκφραση που ξεκινάει με παρενθέσεις, αγκύλες ή άγκιστρα. Για την

αναγνώριση ενός συνδυασμού τέτοιων χαρακτήρων χρησιμοποιούμε ένα δείκτη ο οποίος είναι σε θέση να χειριστεί τα ζευγάρια τέτοιων εκφράσεων. Αν συναντήσουμε έστω, άνοιγμα παρένθεσης '(' τότε αυξάνουμε ένα σχετικό δείκτη ενώ όταν συναντήσουμε το κλείσιμο μιας παρένθεσης ')' τότε μειώνουμε τον δείκτη. Με αυτόν τον τρόπο είμαστε σε θέση να διαχειριστούμε και τους κενούς χαρακτήρες μέσα σε κάθε έκφραση και τη σωστή διαχείριση των εμφωλευμένων δομών (αν υπάρχουν).

5.2 Συντακτικός αναλυτής (Parser)

Εδώ χτίζεται ουσιαστικά η γραμματική που χρησιμοποιήθηκε[20] για την αναγνώριση της γλώσσας προγραμματισμού MATLAB. Ξεκινάμε με έναν κανόνα που είναι η ρίζα του συντακτικού δέντρου που θα κατασκευαστεί αργότερα και συνεχίζουμε με τους υπόλοιπους κανόνες για τις διάφορες δομές της γλώσσας.

Επί του παρόντος, υποστηρίζουμε ένα υποσύνολο της γλώσσας. Μερικές από τις δομές που υποστηρίζονται είναι:

- `if` statements
- `while` loops
- `for` loops
- `switch` statements
- `try-catch` statements
- `break`, `continue`, `return` statements
- structures
- matrices
- anonymous functions
- function definitions
  - `global` declarations
  - `persistent` declarations
- line comments/multiline comments
- support for all MATLAB primitive operations

Οι παρακάτω δομές δεν υποστηρίζονται:

- Tables
- Cell arrays

- Objects and classes

Συνήθως ένα πρόγραμμα μπορεί να αποτελείται από διάφορα αρχεία, τα οποία ονομάζονται M-files, που μπορεί να περιέχουν βοηθητικές συναρτήσεις (οι οποίες μπορεί να δέχονται κάποιες παραμέτρους εισόδου ή/και να έχουν παραμέτρους εξόδου) ή περισσότερα από ένα (scripts) τα οποία ούτε δέχονται ούτε επιστρέφουν παραμέτρους και κάποιο από τα οποία μπορεί να καλεί κάποιες από αυτές τις συναρτήσεις.

5.2.1 Δηλώσεις (Statements)

Ένα πρόγραμμα MATLAB αποτελείται από μία ακολουθία από δηλώσεις (statements). Τα statements χωρίζονται από διαχωριστικά (delimiters). Αυτά τα διαχωριστικά είναι ένα μείγμα από τους χαρακτήρες *κόμμα* (,), *ερωτηματικό* (;) και *αλλαγή γραμμής* (\n). Ένα statement μπορεί να είναι μια έκφραση (expression), μια ανάθεση σε μια μεταβλητή καθώς και άλλες δομές που αναφέρθηκαν παραπάνω. Όλα τα statements είναι απαραίτητο να τελειώνουν με ένα διαχωριστικό. Εξαίρεση αποτελεί το statement στην τελευταία γραμμή του αρχείου.

Επειδή πολλές φορές είναι δυνατόν ένα πρόγραμμα να αποτελείται από περισσότερα από ένα αρχεία, έχουμε ενσωματώσει κάτω από την ίδια δομή, είτε μιλάμε για το συντακτικό δέντρο είτε για την υψηλού επιπέδου αναπαράσταση, όλα τα αρχεία από τα οποία αποτελείται το πρόγραμμα. Κατά συνέπεια, στην τελική αναπαράσταση διαθέτουμε όλα τα αρχεία συνολικά. Πρακτικά είναι ο μόνος τρόπος για να γίνει αυτό με το ANTLR4. Στο ANTLR3[23] δίνεται η δυνατότητα, να πραγματοποιηθεί ο συνδυασμός των δέντρων που επιστρέφονται μετά τη συντακτική ανάλυση κάνοντας χρήση συγκεκριμένων κλάσεων.

Η ρίζα του δέντρου αναπαρίσταται με τον κανόνα:

```
translation_unit :
                 | statementlist (FILE_SEPARATOR statementlist)*;
```

Όταν ξεκινάει ο συντακτικός αναλυτής, κοιτάει στο δεξί μέρος, από αριστερά προς τα δεξιά, ποιους κανόνες πρέπει να αναπτύξει για να σταματήσει τελικά όταν φτάσει σε τερματικά σύμβολα. Το token `FILE_SEPARATOR` είναι ένα αναγνωριστικό που έχει εισαχθεί έτσι ώστε να ξεχωρίζουμε τα περισσότερα από ένα αρχεία που μπορεί να έχει δώσει ο χρήστης ως είσοδο.

5.2.2 Συναρτήσεις (Functions)

Αναφορικά με τις συναρτήσεις, υπάρχουν διάφοροι τρόποι δηλώσεων. Κάθε συνάρτηση μπορεί να περιέχει κάποια από τα ακόλουθα στοιχεία:

function keyword (required)	
Output arguments (optional)	**output arguments:** `function [a,b,c] = func(x)` **no output arguments:** `function func(x)` **empty square brackets:** `function [] = func(x)`
Function name (required)	Valid function names follow the same rules as variable names
Input arguments (optional)	**input arguments:** `function y = func(a,b,c)` **no input arguments:** `function y = func` **empty parenthesis:** `function y = func()`

Σχήμα 5.1: Δηλώσεις συναρτήσεων (Function definitions)

Το σώμα μιας συνάρτησης μπορεί να περιέχει όλες τις δομές που αναφέρθηκαν παραπάνω. Κάθε μεταβλητή που δηλώνεται μέσα στο σώμα της συνάρτησης έχει τοπικό χαρακτήρα εκτός αν δηλωθεί διαφορετικά. Το τέλος μιας συνάρτησης σημειώνεται με τη δεσμευμένη λέξη end. Αν ένα αρχείο περιέχει μόνο μία συνάρτηση τότε το end είναι προαιρετικό (δεν υπάρχει υποστήριξη για κάτι τέτοιο προς το παρόν).

Σχετικά, με τις ανώνυμες συναρτήσεις (anonymous functions) αυτές έχουν διαφορετική μορφή. Ο ορισμός μιας τέτοια συνάρτησης έχει ως εξής:

```
expression : FUNCTION_HANDLE arguments expression
```

Μια ανώνυμη συνάρτηση ξεκινάει με το χαρακτήρα '@', ακολουθούν τα ορίσματα μέσα σε ένα ζευγάρι παρενθέσεων και μετά η έκφραση η οποία θα υλοποιεί η συνάρτηση. Ουσιαστικά η συνάρτηση αυτή αποτελείται μόνο από μία γραμμή, η οποία υπολογίζει μια έκφραση και επιστρέφει ένα αποτέλεσμα. Για παράδειγμα, έστω ότι ορίζουμε μια ανώνυμη συνάρτηση:

```
sqr = @(x) x.^2;
```

Η συνάρτηση υλοποιεί τον υπολογισμό του τετραγώνου ενός αριθμού και επιστρέφει ένα δείκτη σε μία μεταβλητή. Έστω ότι υπολογίζουμε το τετράγωνο του αριθμού 5 περνώντας σαν παράμετρο την τιμή στο δείκτη της συνάρτησης sqr, ακριβώς όπως περνάμε μια παράμετρο σε μία κανονική συνάρτηση.

```
a = sqr(5)
a =
    25
```

Πρακτικά μια ανώνυμη συνάρτηση δημιουργεί ένα δείκτη όπου μέσω αυτού καλούμε την συνάρτηση. Όπως είδαμε και παραπάνω, οι συναρτήσεις σε ένα

πρόγραμμα MATLAB αποθηκεύονται σε ξεχωριστά αρχεία, τα λεγόμενα m-files, και καλούνται όταν χρειάζεται από ένα script. Το βασικό πλεονέκτημα των ανώνυμων συναρτήσεων είναι ότι δεν αποθηκεύονται σε ένα ξεχωριστό αρχείο, αντίθετα σχετίζονται με μία μεταβλητή τύπου function_handle. Κατά συνέπεια, αν μια συνάρτηση μπορεί να υλοποιηθεί άμεσα σε μία μόνο γραμμή, οι ανώνυμες συναρτήσεις έχουν πλεονέκτημα έναντι των κλασσικών.

5.2.3 Εκφράσεις (Expressions)

Οι εκφράσεις στο MATLAB συνθέτονται ως πράξεις πάνω σε υπό εκφράσεις όπου τελικά αυτές αναλύονται σε τερματικά σύμβολα, όπως είναι ακέραιοι, δεκαδικοί, αλφαριθμητικά ή φανταστικοί αριθμοί. Ο lexer μας αναγνωρίζει ένα μεγάλο εύρος αριθμών και σταθερών.

- decimal $(0.1, .1, 1.0, 1.)$
- integer $(0, 5, 54)$
- identifier (`var_5`)
- floating $(1.0001e + 308, 55E + 308)$
- string literal (`‘this is a string’`)
- imaginary unit $(55i, 24j, 5 + 15i)$

Επιπλέον, όσον αφορά τους φανταστικούς αριθμούς, ο lexer αναγνωρίζει οποιονδήποτε αριθμό που ακολουθείται αμέσως από μία φανταστική μονάδα ως φανταστικό αριθμό. Οι φανταστικοί αριθμοί ορίζονται από του χαρακτήρες ‘`i`’ ή ‘`j`’. Η συγκεκριμένη αναγνώριση αφορά το λεξερ και κατα συνέπεια δεν επηρεάζει απλές μεταβλητές με όνομα `i` ή `j`.

Εξίσου σημαντικό είναι οι προτεραιότητες μεταξύ των τελεστών που υπάρχουν (operator precedence)[21] αλλά και η εξάλειψη της αριστερής αναδρομής (left recursion). Για να επιλύσει τις όποιες ασάφειες μεταξύ των εναλλακτικών ενός κανόνα, το ANTLR δίνει μεγαλύτερη προτεραιότητα σε αυτές που βρίσκονται ψηλότερα. Συνεπώς, σε συνδυασμό με της προτεραιότητες των τελεστών εξαλείφονται οι όποιες αμφιβολίες που μπορεί να συναντήσει ο parser κατά την ανάπτυξη των κανόνων.

Ακόμη, το ANTLR μας επιτρέπει να δηλώσουμε χειροκίνητα το associativity για κάποιο τελεστή. Από προεπιλογή, το ANTLR ορίζει όλους τους τελεστές από αριστερά (left associative). Για παράδειγμα, ο τελεστής ύψωσης σε δύναμη ‘^’ είναι right associative. Έστω ότι έχουμε τον κανόνα expr ο οποίος αναγνωρίζει μια πράξη ύψωσης σε δύναμη, π.χ. 2^3. Συνεπώς ο κανόνας γίνεται ως εξής.

```
expr : expr ‘^’<assoc=right> expr
     | INT
     ;
```

Τέλος όσον αφορά την αριστερή αναδρομή (left recursion), το ANTLR4 μπορεί και τις εξαλείφει εσωτερικά (σε αντίθεση με το ANTLR3) χωρίς την παρέμβαση του χρήστη όπως γίνεται σε άλλους συντακτικούς αναλυτές όπως είναι ο bison.

Ως κανόνας αριστερής αναδρομής (left recursive rule) ορίζεται αυτός ο οποίος καλεί τον εαυτό του, είτε άμεσα είτε έμμεσα, στο αριστερό μέρος μιας εναλλακτικής. Ο κανόνας expr που είδαμε παραπάνω είναι άμεσης αριστερής αναδρομής (direct left recursion) διότι όλες οι εναλλακτικές εκτός από την 'INT' ξεκινούν με μια αναφορά στον ίδιο τον κανόνα expr.

Παρ' όλο που το ANTLR4 μπορεί να διαχειριστεί την άμεση αναδρομή, δεν ισχύει το ίδιο και για την έμμεση αριστερή αναδρομή (indirect left recursion).

```
expr : expo // indirectly invokes expr left recursively via expo
     | ...
     ;
expo : expr '^'<assoc=right> expr;
```

Όπως φάνηκε παραπάνω, η κατασκευή της γραμματικής είναι σχετικά απλή, ωστόσο η σωστή αναγνώριση της εισόδου απαιτεί διάφορες προκαταρκτικές ενέργειες. Στη συνέχεια θα παρουσιάσουμε το βασικό μηχανισμό πάνω στον οποίο στηρίζεται και αποθηκεύεται η υψηλού επιπέδου αναπαράσταση που μπορεί να χρησιμοποιηθεί εν τέλει για ποικίλους σκοπούς.

Κεφάλαιο 6

Μηχανισμός διαχείρισης της υψηλού επιπέδου αναπαράστασης (HLIR mechanism)

Στο παρών κεφάλαιο θα παρουσιάσουμε τις τεχνικές που χρησιμοποιήθηκαν σχετικά με το σύστημα διαχείρισης του *Αφαιρετικού Συντακτικού Δέντρου* (Abstract Syntax Tree - AST). Σε γενικές γραμμές ο σχεδιασμός είναι ανεξάρτητος από τη γλώσσα προγραμματισμού και μπορεί να προσαρμοστεί εύκολα στις εκάστοτε ανάγκες λόγω της εκτεταμένης χρήσης του πολυμορφισμού (polymorphism) και της κληρονομικότητας (inheritance) καθώς και διάφορων μοτίβων σχεδίασης (design patterns).

Τέλος θα παρουσιαστεί και ένα απλό πέρασμα στο ΑΣΤ με στόχο την γραφική απεικόνιση του η οποία γίνεται με χρήση της γλώσσας αναπαράστασης γράφων DOT[1].

6.1 Μοτίβα σχεδίασης που εφαρμόστηκαν

Για να μπορέσουμε να κατασκευάσουμε ένα σύστημα το οποίο θα είναι προσαρμόσιμο, ευέλικτο αλλά και επεκτάσιμο χρησιμοποιήσαμε κάποια από τα μοτίβα σχεδίασης που αναφέρθηκαν στην παράγραφο 1.1. Αυτά είναι:

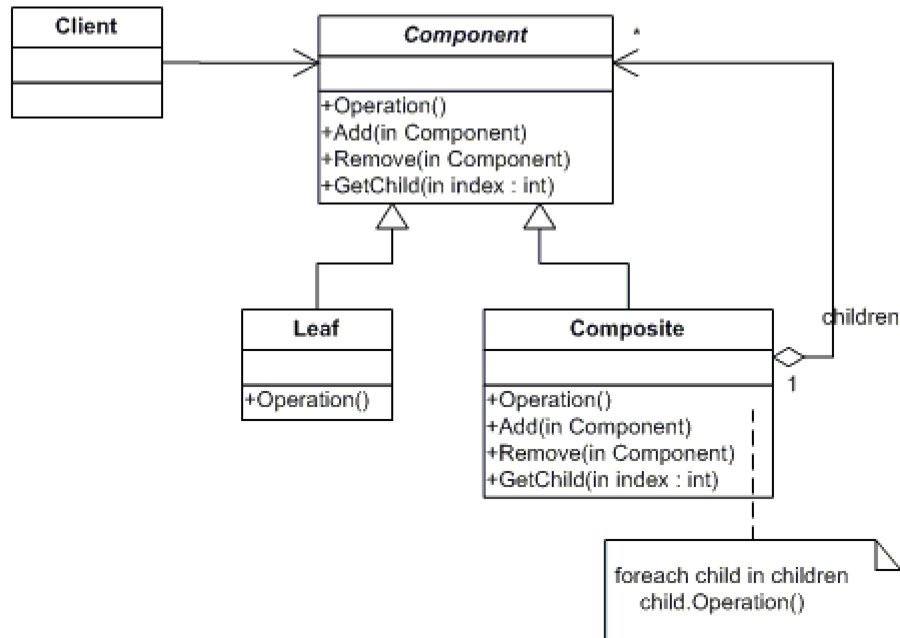
- Composite
- Visitor
- Observer

[1] `http://www.graphviz.org/doc/info/lang.html`

- Iterator
- Factory Method
- Facade

6.1.1 Μοτίβο σύνθεσης (Composite pattern)

Το μοτίβο σύνθεσης (composite pattern) περιγράφει ότι μία ομάδα από αντικείμενα (objects) θα αντιμετωπίζεται με τον ίδιο τρόπο όπως ένα απλό στιγμιότυπο ενός αντικειμένου. Σκοπός του composite είναι να συνθέσει αντικείμενα σε δεντρικές αναπαραστάσεις για να αναπαραστήσει μια ιεραρχία. Υλοποιώντας αυτό το μοτίβο μας επιτρέπει να διαχειριστούμε απλά αντικείμενα και άλλες συνθέσεις ομοιόμορφα.



Σχήμα 6.1: Μοτίβο σχεδίασης Composite

Κάθε φορά που θέλουμε να πραγματοποιήσουμε κάποιο πέρασμα στους κόμβους του ΑΣΤ πρέπει να δημιουργήσουμε μερικές βασικές κλάσεις οι οποίες με τη σειρά τους χρησιμοποιούν το μηχανισμό που έχουμε δημιουργήσει για τη διαχείριση της πληροφορίας όλης της αναπαράστασης.

Πιο συγκεκριμένα, όσον αφορά τη δεντρική αναπαράσταση, έχουν κατασκευαστεί 4 κλάσεις οι οποίες προσδιορίζουν τη δομή του δέντρου και είναι υπεύθυνες για την αποθήκευση της πληροφορίας.

ASTElement class

Όλα ξεκινούν από την κλάση ASTElement όπου εδώ πρακτικά αποθηκεύονται κάποιες βασικές πληροφορίες όπως είναι οι κόμβοι του ΑΣΤ, καθώς και βοηθητικές συναρτήσεις για την ανάκτηση ή προσθήκη δεδομένων στην δομή.

ASTComposite class

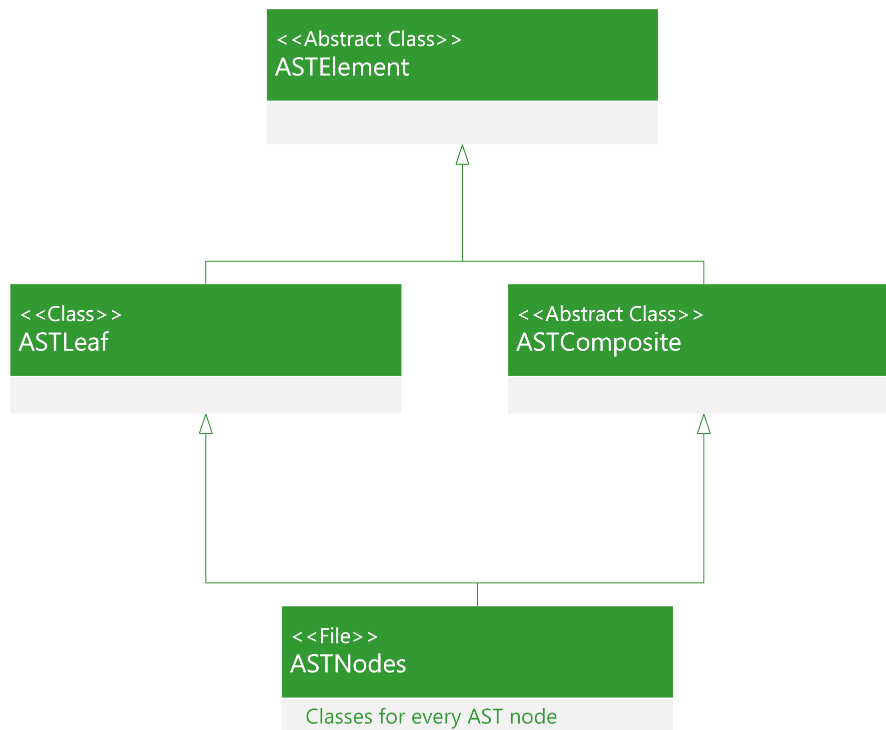
Αναπαριστά ένα σύνθετο κόμβο του ΑΣΤ ο οποίος μπορεί να αποτελείται και από άλλους κόμβους.

ASTLeaf class

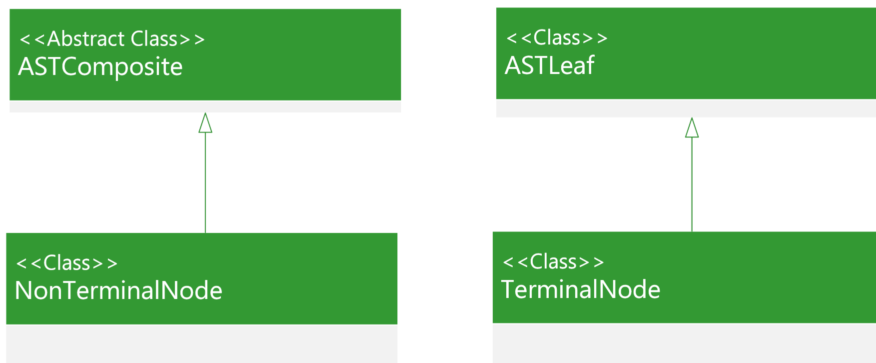
Αναπαριστά ένα κόμβο που βρίσκεται στα σύνορα του ΑΣΤ και συνήθως αναπαριστά τερματικά σύμβολα της γραμματικής.

ASTNodes file

Ένα αρχείο που περιλαμβάνει ξεχωριστές κλάσεις για κάθε κόμβο του ΑΣΤ οι οποίες είναι υπεύθυνες για την σωστή τοποθέτηση του εκάστοτε κόμβου στο ΑΣΤ καθώς και την αποθήκευση της όποιας πληροφορίας προκύπτει από τα περάσματα που γίνονται.



Σχήμα 6.2: Βασικές κλάσεις του ΑΣΤ



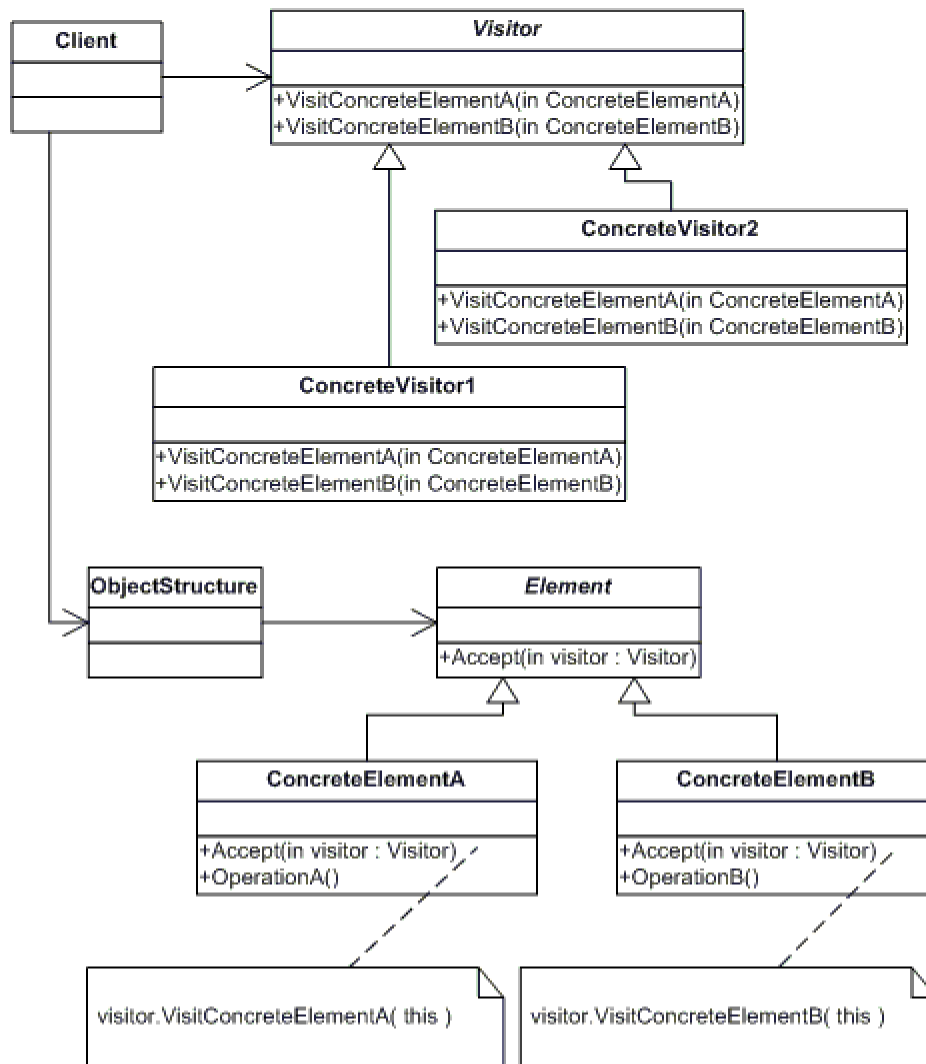
Σχήμα 6.3: Τερματικός και μή-τερματικός κόμβος ΑΣΤ

Από μόνη της η παραπάνω δομή δεν προσφέρει κάτι. Σε συνδυασμό με το μοτίβο επισκέπτη (visitor pattern) που θα δούμε παρακάτω γίνεται εφικτή η ανάκτηση ή αποθήκευση δεδομένων στην αναπαράσταση.

6.1.2 Μοτίβο επισκέπτη (Visitor pattern)

Έχοντας υλοποιήσει τις απαραίτητες κλάσεις και τους κόμβους, συνέχεια έχει ένας μηχανισμός οποίος θα διασχίζει το ΑΣΤ. Το μοτίβο επισκέπτη είναι ένας τρόπος διαχωρισμού ενός αλγορίθμου από μία δομή πάνω στην οποία επιδρά. Η διάσχιση του δέντρου γίνεται καλώντας την αντίστοιχη μέθοδο του κόμβου που επιθυμούμε να προσπελάσουμε. Έτσι μια εικονική μέθοδος (virtual method) μπορεί να παρακαμφθεί (override) και να καλείται από τον κώδικα του κάθε κόμβου εξασφαλίζοντας πάντα ότι θα καλείται η ρουτίνα που αντιστοιχεί στον τύπο της τρέχουσας κλάσης.

Στην ουσία, με αυτό το μοτίβο κατασκευάζουμε ένα μηχανισμό ο οποίος ανάλογα με το πέρασμα που επιθυμούμε να κάνουμε δίνουμε τη δυνατότητα σε κάποιον να παρακάμψει μια εικονική μέθοδο και σε μια καινούρια κλάση να την υλοποιήσει κατάλληλα ανάλογα με τις ανάγκες του.



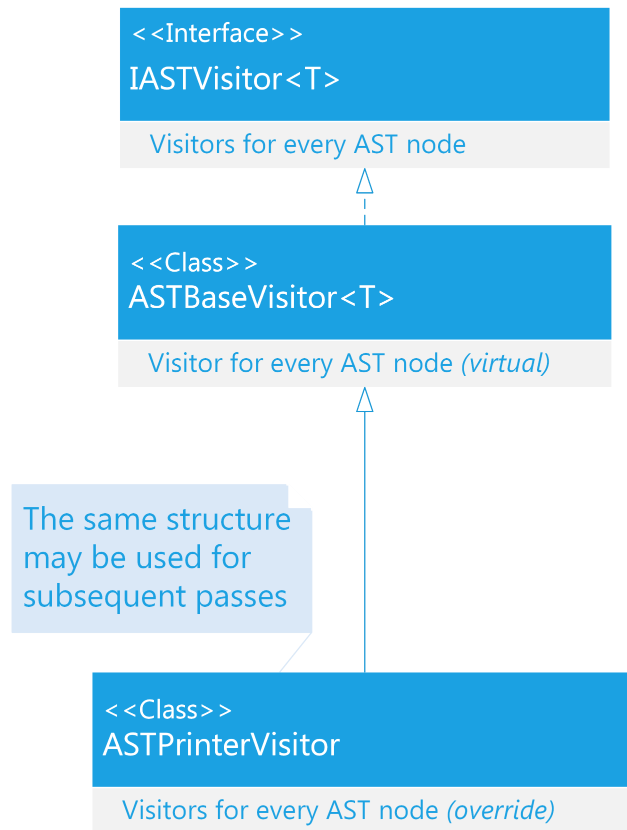
Σχήμα 6.4: Μοτίβο σχεδίασης Visitor

Για την κατασκευή, βάσει του μοτίβου, υλοποιήθηκαν ένα interface και μια κλάση.

IASTVisitor interface
: Περιέχει απλά τα prototypes όλων των visitor μεθόδων για κάθε κόμβο του ΑΣΤ.

ASTBaseVisitor class
: Περιέχει τις εικονικές υλοποιήσεις των μεθόδων οι οποίες μπορούν να παρακαμφθούν για να χρησιμοποιηθούν με διαφορετικό τρόπο.

Πέρα από τις μεθόδους για κάθε κόμβο, υλοποιήθηκαν και 2 ακόμα μέθοδοι που χρησιμοποιούνται για γενικές περιπτώσεις, ασχέτως κόμβου. Το μοτίβο επισκέπτη χρησιμοποιεί τον αλγόριθμο DFS (Depth First Search) για να διαγράψει το δέντρο.

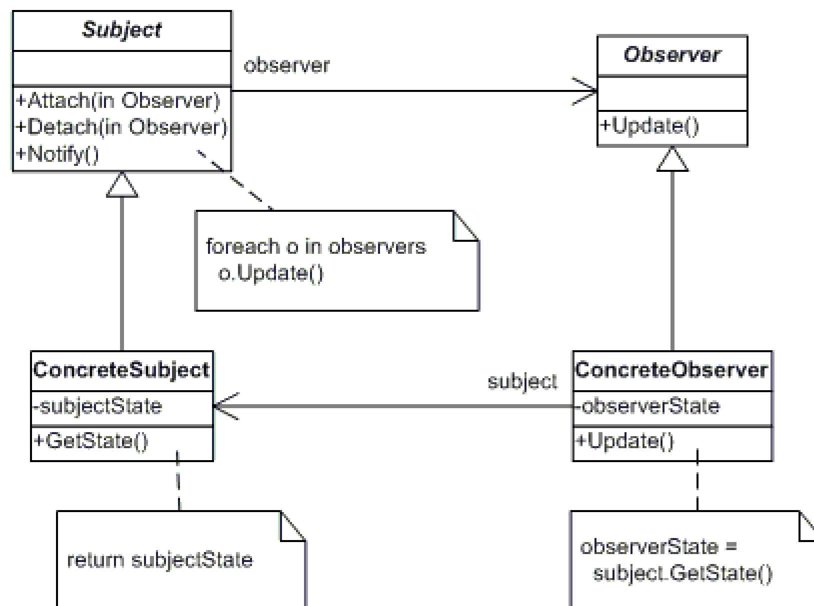


Σχήμα 6.5: Σχέση μεταξύ ενός περάσματος και του μοτίβου Visitor

6.1.3 Μοτίβο παρατηρητή (Observer pattern)

Σε συνδυασμό με το μοτίβο επισκέπτη, χρησιμοποιούμε το μοτίβο παρατηρητή (observer pattern). Ένα αντικείμενο (object), που ονομάζεται υποκείμενο (subject) διατηρεί μια λίστα από παρατηρητές (observers) και τους ενημερώνει αυτόματα με οποιαδήποτε αλλαγή κατάστασης συνήθως καλώντας μία από τις μεθόδους τους. Χρησιμοποιείται κυρίως για τη κατασκευή κατανεμημένων συστημάτων διαχείρισης γεγονότων.

Στην περίπτωση μας, υλοποιήσαμε ένα σύστημα όπου κάθε φορά που εισερχόμαστε ή εξερχόμαστε σε έναν κόμβο, ενεργοποιείται ένα γεγονός. Αυτό σημαίνει ότι μπορούμε να χρησιμοποιήσουμε τα γεγονότα έτσι ώστε να πραγματοποιήσουμε οποιαδήποτε ενέργεια σε οποιοδήποτε σημείο του ΑΣΤ.



Σχήμα 6.6: Μοτίβο σχεδίασης Observer

Η ουσία εδώ είναι η σωστή διαχείριση των γεγονότων καθώς και το περιεχόμενο τους ανάλογα τι επιθυμούμε να γίνετε.

ASTAbstractVisitorEvents class
Αυτή η αφηρημένη κλάση υλοποιεί ως εικονικές μεθόδους τα γεγονότα που θα χρησιμοποιηθούν. Κατά συνέπεια, αυτά μπορούν να παρακαμφθούν στις κλάσεις των κόμβων του ΑΣΤ.

ASTVisitorEventArgs class
Κάθε γεγονός είναι δυνατόν να μεταφέρει κάποια δεδομένα, π.χ. ο κόμβος από τον οποίο ήρθαμε (parent node) κλπ. Αυτό γίνεται υπό την μορφή των ορισμάτων (arguments).

NodeVisitorEvents file
Εδώ περιέχονται όλες οι κλάσεις ο οποίες αναπαριστούν τους κόμβους του ΑΣΤ και είναι υπεύθυνες για την υλοποίηση και τη σωστή διαχείριση των γεγονότων. Μέσα στα εκάστοτε γεγονότα μπορούμε να τοποθετήσουμε διάφορες ενέργειες που επιθυμούμε να πραγματοποιούνται κατά τη διάσχιση του δέντρου.

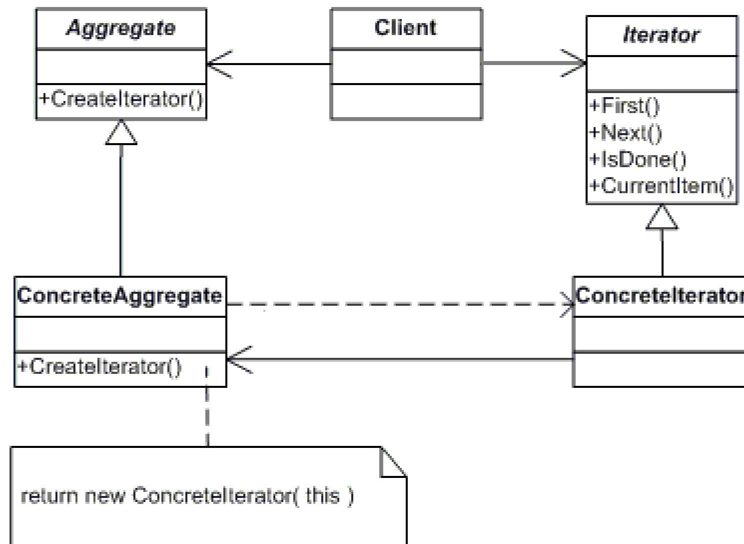
Σχήμα 6.7: Υλοποίηση και διαχείριση γεγονότων με τη βοήθεια του μοτίβου Observer

Για να εφαρμοστούν με τη σωστή ακολουθία τα γεγονότα χρειαζόμαστε ένα ακόμα μοτίβο σχεδίασης το οποίο θα αναλυθεί παρακάτω και το οποίο διευκολύνει τη διάσχιση του ΑΣΤ. Τέλος, μέσω αυτού γίνεται και η ενεργοποίηση των γεγονότων.

6.1.4 Μοτίβο επαναλήπτη (Iterator pattern)

Σε συνδυασμό με το μοτίβο παρατηρητή, χρησιμοποιούμε και το μοτίβο επαναλήπτη. Ένας επαναλήπτης (iterator) χρησιμοποιείται για να διασχίσουμε ένα context καθώς και να αποκτήσουμε πρόσβαση στα στοιχεία αυτού.

Χρησιμοποιήθηκαν αρκετές κλάσεις και interfaces έτσι ώστε να μπορεί ο κώδικας μας να είναι επεκτάσιμος με τη βοήθεια βέβαια το πολυμορφισμού (polymorphism) και της κληρονομικότητας (inheritance). Πρέπει να τονιστεί εδώ ότι γίνεται και χρήση ενός ακόμα μοτίβου το οποίο θα αναλυθεί σε επόμενη παράγραφο.



Σχήμα 6.8: Μοτίβο σχεδίασης Iterator

Iterator iterface

Περιέχονται 2 σύνολα από μεθόδους που χρησιμοποιούνται για τη διάσχιση του ΑΣΤ. Στο πρώτο, οι μέθοδοι επιστρέφουν τον κόμβο που επισκεφθήκαμε ενώ στο δεύτερο δεν επιστρέφουν κάτι.

ASTContextIterator_Sequential class

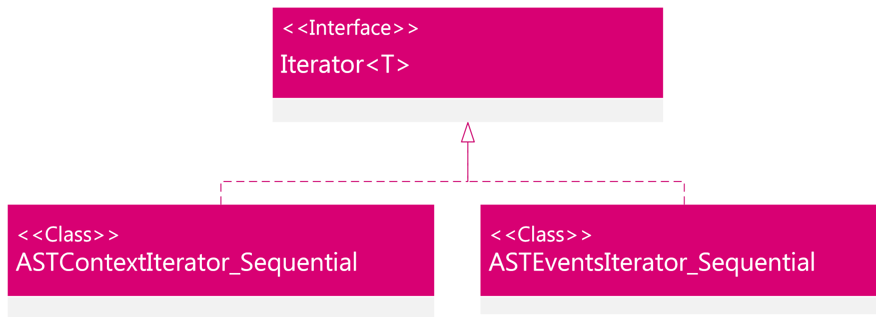
Η συγκεκριμένη κλάση υλοποιεί το Iterator interface χρησιμοποιώντας κανονικούς iterators (regular iterators)[2]. Εδώ, υλοποιούνται οι μέθοδοι που χρησιμοποιούνται για τη διάσχιση του ΑΣΤ.

ASTEventsIterator_Sequential class

Σε αντίθεση με την προηγούμενη κλάση, εδώ κάνουμε χρήση των iterators γεγονότων (events iterators)[3].

[2] **Regular iterator**: Αυτή είναι η προεπιλεγμένη υλοποίηση του μοτίβου σχεδίασης Iterator.

[3] **Event iterator**: Συνδυασμός των μοτίβων σχεδίασης Observer και Iterator. Για κάθε κόμβο ή context που επισκεπτόμαστε ενεργοποιείται και ένα γεγονός (event).

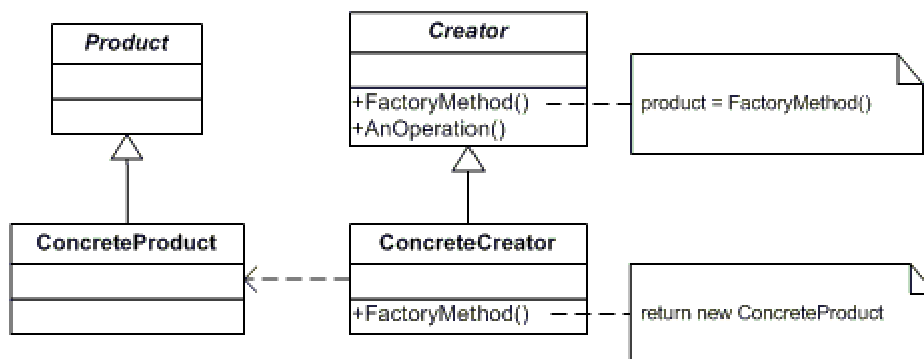


Σχήμα 6.9: Regular και events iterators

6.1.5 Μοτίβο εργοστάσιο μεθόδων (Factory method pattern)

Ορίζουμε ένα interface για τη δημιουργία ενός αντικειμένου αλλά επιτρέπουμε στις υποκλάσεις να επιλέξουν ποια κλάση θα το δημιουργήσει. Χρησιμοποιεί μεθόδους που αναλαμβάνουν να δημιουργήσουν αντικείμενα χωρίς να χρειάζεται να καθορίσουν την ακριβή κλάση του αντικειμένου που θα δημιουργηθεί. Το μοτίβο Factory method μας επιτρέπει να διαχωρίσουμε τη δημιουργία των αντικειμένων σε υποκλάσεις.

Ουσιαστικά αυτό είναι μια επέκταση η οποία μας παρέχει ευελιξία πάνω στον έλεγχο των κόμβων του δέντρου. Πρακτικά, έχουν υλοποιηθεί 2 interfaces και 2 κλάσεις οι οποίες είναι υπεύθυνες για τα 2 είδη iterators που είδαμε στην παράγραφο 6.1.4.



Σχήμα 6.10: Μοτίβο σχεδίασης Factory method

ContextIteratorFactory iterface

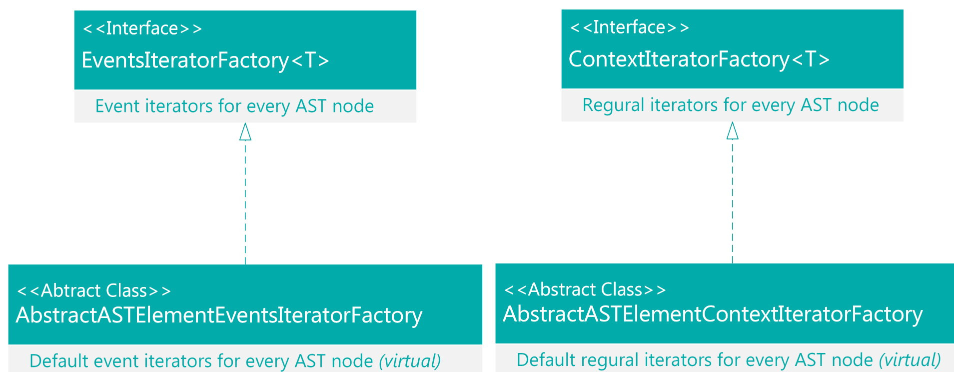
Αυτό το interface μας παρέχει ένα σύνολο από υπογραφές μεθόδων των regular iterators για κάθε κόμβο του ΑΣΤ (εκτός από τους τερματικούς κόμβους). Αυτό μας δίνει τη δυνατότητα να δημιουργήσουμε ένα διαφορετικό

iterator ανά κόμβο.

EventsIteratorFactory interface
Όπως και πριν, έτσι και εδώ έχουμε ένα δεύτερο interface το οποίο περιέχει ένα σύνολο από υπογραφές μεθόδων των event iterators αυτή τη φορά.

AbstractASTElementContextIteratorFactory abstract class
Υλοποιεί το interface ContextIteratorFactory και περιέχει τις προεπιλεγμένες υλοποιήσεις για τους regular iterators.

AbstractASTElementEventsIteratorFactory abstract class
Υλοποιεί το interface EventsIteratorFactory και περιέχει τις προεπιλεγμένες υλοποιήσεις για τους event iterators.

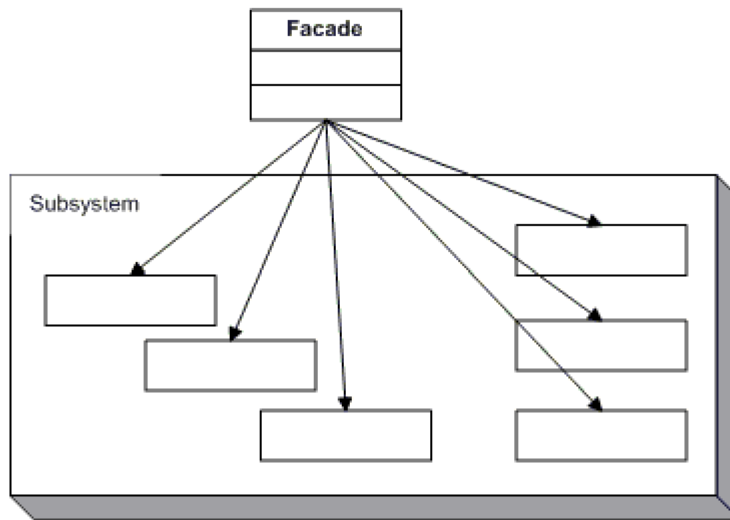


Σχήμα 6.11: Εργοστάσιο παραγωγής iterators

6.1.6 Μοτίβο πρόσοψης (Façade pattern)

Αυτό το μοτίβο ορίζει ένα interface σε υψηλότερο επίπεδο έτσι ώστε να κάνει τη χρήση ενός υποσυστήματος ευκολότερη. Το μοτίβο σχεδίασης Façade χρησιμοποιείται όταν ένα σύστημα είναι πολύπλοκο ή δύσκολο ως προς την κατανόηση της λειτουργίας του επειδή αποτελείται από ένα μεγάλο αριθμό από ανεξάρτητες κλάσεις ή ο πηγαίος κώδικας του δεν είναι διαθέσιμος.

Στην ουσία, αποκρύπτεται η πολυπλοκότητα ενός μεγαλύτερου συστήματος και παρέχει ένα απλούστερο περιβάλλον για τον χρήστη/προγραμματιστή. Τυπικά περιλαμβάνει μία κλάση που λειτουργεί σαν περιτύλιγμα (wrapper) η οποία περιέχει ένα σύνολο από μέλη και μεθόδους. Αυτά τα μέλη έχουν πρόσβαση στο σύστημα εκ μέρους της κλάσης αυτής κρύβοντας έτσι τις λεπτομέρειες της υλοποίησης.



Σχήμα 6.12: Μοτίβο σχεδίασης Façade

Στην περίπτωση μας, χρησιμοποιήσαμε μόνο μία κλάση η οποία καλύπτει όλα τα υποσυστήματα διαχείρισης της αναπαράστασης.

MATLABCompiler_Facade class

Αυτή η κλάση χειρίζεται την ακολουθία εκτέλεσης του συστήματος. Περιέχει όλα τα δεδομένα που χρησιμοποιούνται από τις μεθόδους και η αρχικοποίηση τους γίνεται εδώ. Η μέθοδος `MatlabCompiler(string[] args)` είναι υπεύθυνη για την εκκίνηση της διαδικασίας μεταγλώττισης. Ακόμη, εδώ γίνεται η διαχείριση των αρχείων που δίνονται ως είσοδος καθώς και των ορισμάτων (arguments) που καθορίζουν την λειτουργία της μεταγλώττισης.

<<Class>>
MATLABCompiler_Facade

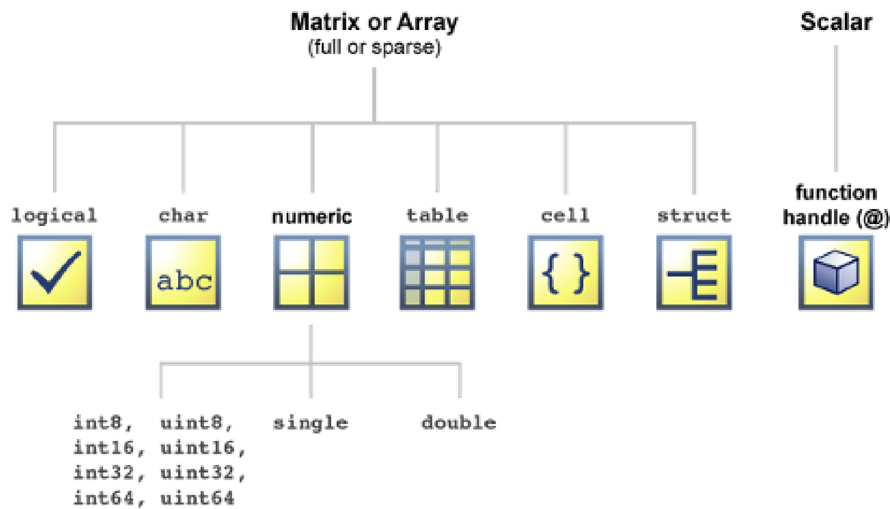
void MatlabCompiler(string[] args)
void MultipleFileJoiner(string[] args)
void ParseTreeGeneration()
void PrintTokens()
void PrintParseTree()
void PrintAST()
void ASTGeneration()
void SymbolTableLoader()
string[] ParseArgs(string[] args)
void Help()

Σχήμα 6.13: Η κλάση που λειτουργεί ως ένα wrapper του κύριου συστήματος

6.2 Πίνακας συμβόλων (Symbol table)

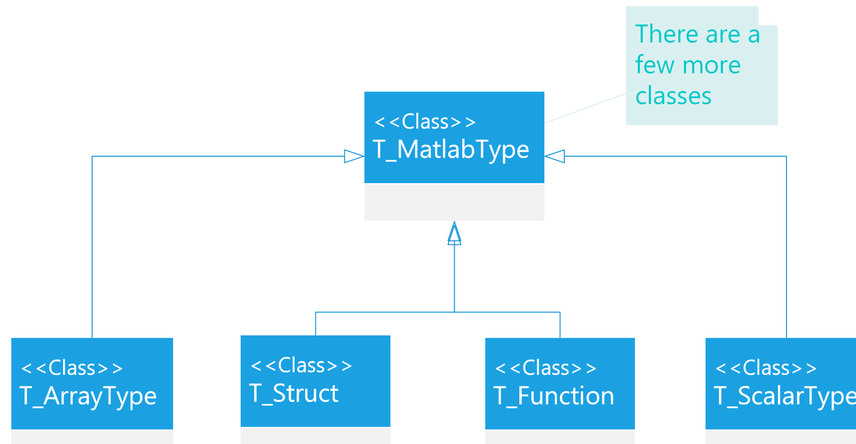
Για την περαιτέρω ανάπτυξη του μεταγλωττιστή είναι απαραίτητο να γνωρίζουμε τα σύμβολα του προγράμματος έτσι ώστε να τα διαχειριστούμε αναλόγως. Έχουμε κατασκευάσει μια ομάδα από κλάσεις οι οποίες είναι υπεύθυνες για τη διαχείριση των συμβόλων που αναγνωρίζονται κατά τη διαδικασία της συντακτικής ανάλυσης του προγράμματος. Σύμφωνα με το documentation του MATLAB[22], αρχικά υπάρχουν 2 είδη τύπων. Matrix και Scalar. Εξ' ορισμού, το MATLAB αποθηκεύει όλες τις αριθμητικές τιμές ως αριθμού κινητής υποδιαστολής, διπλής ακρίβειας (double).

Παρακάτω θα γίνει αναφορά στις δομές που αφορούν τους τύπους που υποστηρίχθηκαν και στη συνέχεια θα περιγράψουμε τα scopes που αποτελούν ένα πρόγραμμα MATLAB.



Σχήμα 6.14: Οι θεμελιώδεις τύποι του MATLAB

Ξεκινώντας πρώτα από τις κλάσεις που αφορούν την περιγραφή των τύπων, όλοι ξεκινούν από μια βασική κλάση που περιέχει τα κοινά χαρακτηριστικά κάθε τύπου την οποία και κληρονομούν οι υπόλοιπες κλάσεις.



Σχήμα 6.15: Μερικοί από τους θεμελιώδεις τύπους του MATLAB

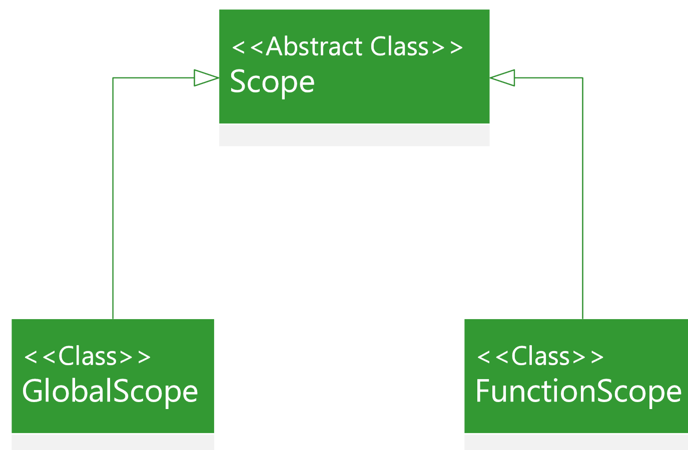
Κάθε κλάση χρησιμοποιείται για να αρχικοποιήσει διάφορα στοιχεία του κάθε συμβόλου ανάλογα με τον τύπο του.

Όσον αφορά τα scopes μέσα στα οποία περιέχονται τα όποια σύμβολα, αυτά είναι 2: ένα global scope το οποίο είναι το κεντρικό πρόγραμμα (script) και ένα local ή function scope το οποίο αφορά το σώμα διαφόρων συναρτήσεων που έχουν υλοποιηθεί.

Ένα σύμβολο αποτελείται συνήθως από 3 μέρη:

- Όνομα συμβόλου
- Τύπος συμβόλου
- Scope στο οποίο ανήκει το σύμβολο

Αυτές οι 3 ιδιότητες ορίζονται για κάθε σύμβολο κατά τη διάσχιση του συντακτικού δέντρου (ΣΣΤ) κάθε φορά που συναντούμε ένα αναγνωριστικό (identifier). Πρώτα απ' όλα, έχουμε υλοποιήσει 2 κλάσεις, `Symbol` και `Scope` οι οποίες συνεργάζονται προκειμένου να διατηρηθεί όλη η πληροφορία που αφορά τα σύμβολα αλλά και που ακριβώς βρίσκονται αυτά. Η κλάση `Scope` είναι μια αφηρημένη κλάση (abstract class) η οποία περιέχει βοηθητικές μεθόδους για την καταχώρηση ή αναζήτηση συμβόλων και η οποία αποτελεί τη βάση για τα 2 βασικά scopes που αναφέρθηκαν παραπάνω.



Σχήμα 6.16: Τα 2 βασικά scopes του MATLAB

Η κλάση `FunctionScope` αφορά τα σύμβολα που εμφανίζονται μέσα στο σώμα συναρτήσεων οι οποίες συναρτήσεις συνήθως ανήκουν στο `GlobalScope`. Για να γίνει αποδοτικά η διαχείριση των scopes και η αποθήκευση των συμβόλων, έχουμε υλοποιήσει μια κλάση η οποία χρησιμοποιείται από το πέρασμα που αφορά την ανακάλυψη των συμβόλων του προγράμματος. Είναι υπεύθυνη για την:

- Είσοδο και έξοδο από το εκάστοτε scope
- Αποθήκευση συμβόλων
- Αναζήτηση συμβόλων

Φτάσαμε στο τέλος της κατασκευής μιας δομής η οποία είναι υπεύθυνη για την διαχείριση του ΑΣΤ. Με τη βοήθεια της είναι δυνατό να πραγματοποιηθούν διάφορες ενέργειες πάνω στην είσοδο όπως είναι για παράδειγμα ή αναγνώριση των συμβόλων του προγράμματος και αποθήκευση αυτών, η γραφική αναπαράσταση του προγράμματος εισόδου υπό μορφή δέντρου ή ακόμα και η μετάφραση της γλώσσας του αρχικού προγράμματος σε μία άλλη γλώσσα. Αυτό είναι δυνατόν γιατί το ΑΣΤ διατηρεί την πληροφορία του αρχικού προγράμματος δίνοντάς μας τη δυνατότητα διαφόρων υλοποιήσεων.

Πρέπει να τονιστεί εδώ ότι όλες οι κλάσεις και δομές που αναφέρθηκαν παραπάνω δεν αντιπροσωπεύουν πλήρως το μηχανισμό διαχείρισης της υψηλού επιπέδου αναπαράστασης.

Στο επόμενο κεφάλαιο θα αναφερθούμε στα διάφορα περάσματα που εφαρμόζονται στην αναπαράσταση που αναφέραμε για την απόκτηση συγκεκριμένων πληροφοριών.

Κεφάλαιο 7

Διάσχιση της υψηλού επιπέδου αναπαράστασης (HLIR traversal)

Προτού περάσουμε στα βασικά περάσματα που έχουμε εφαρμόσει, πρέπει πρώτα να δημιουργήσουμε το *Αφαιρετικό Συντακτικό Δέντρο* (ΑΣΤ) από το *Συνεκτικό Συντακτικό Δέντρο* (ΣΣΤ). Πάνω σε αυτό θα κάνουμε όλα τα περάσματα που χρειαζόμαστε. Όπως είδαμε και σε προηγούμενο κεφάλαιο, το ANTLR είναι αυτό που παράγει το συντακτικό δέντρο καθώς και τις δομές που το διέπουν. Ως εκ το τούτου, χρησιμοποιούμε αυτές τις δομές για να προσπελάσουμε το συντακτικό δέντρο. Πιο συγκεκριμένα, η διάσχιση του ΣΣΤ γίνεται με χρήση του μοτίβου Visitor.

7.1 Μετατροπή ΣΣΤ σε ΑΣΤ

Αυτό είναι το βασικό πέρασμα που αφορά την κατασκευή του ΑΣΤ από το ΣΣΤ. Συνεπώς, η συγκεκριμένη διάσχιση περιλαμβάνει ένα πέρασμα από όλους τους κόμβους του δέντρου έτσι ώστε να συμπεριλάβουμε όλη την πληροφορία της εισόδου. Αυτό θα μας επιτρέψει να ασχοληθούμε στη συνέχεια με το ΑΣΤ και μόνο με αυτό.

Για να πραγματοποιηθεί αυτό το πέρασμα, κατασκευάζουμε μία κλάση (`PTtoASTGeneration class`) η οποία περιέχει visitor μεθόδους για κάθε κόμβο του ΣΣΤ. Μέσω αυτών δημιουργούμε τους κόμβους για το ΑΣΤ, τερματικούς και μη-τερματικούς. Συνοπτικά, ακολουθεί ως ψευδό κώδικας μια μέθοδος επισκέπτη ενός κόμβου του ΣΣΤ.

```
1 // Create an AST node
2 ASTNode node = factory.SomeASTNode();
3
4 // Add to the parent's list
```

```
5 // Make the connection between
6 // the parent node (where we came)
7 // and the child node (where we are now)
8 parent.AddChild(node);
9
10 // Update parent stack on entering
11 ParentStack.Push(node);
12
13 // Visit children using ANTLR's visit method
14 VisitChildren(context);
15
16 // Update on leaving
17 ParentStack.Pop();
```

Στη γραμμή **2** γίνετε η δημιουργία ενός κόμβου του ΑΣΤ.

Μετά το πέρας της διάσχισης του ΣΣΤ με την βοήθεια των visitor μεθόδων όλη η πληροφορία σχετικά με τη δομή του ΑΣΤ έχει αποθηκευθεί στη δομή που έχουμε κατασκευάσει και είναι διαθέσιμη για περαιτέρω επεξεργασία.

7.2 Γραφική αναπαράσταση των ΣΣΤ και ΑΣΤ

Η οπτικοποίηση των ΣΣΤ και ΑΣΤ είναι ένας εύκολος τρόπος να αναπαραστήσουμε την πληροφορία του προγράμματος έτσι ώστε να είναι πιο κατανοητή. Για να γίνει αυτό πρέπει να εφαρμοστεί ένα ξεχωριστό πέρασμα και στις 2 δομές. Για να το πετύχουμε αυτό κατά τη διάρκεια των περασμάτων προσθέτουμε τον απαραίτητο κώδικα για την δημιουργία των κόμβων. Χρησιμοποιήθηκε το εργαλείο παραγωγής γράφων Graphviz στο οποίο αναφερθήκαμε στην παράγραφο 4.3.

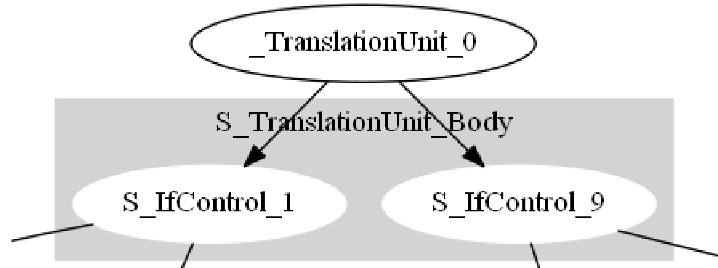
Το αποτέλεσμα του περάσματος είναι η παραγωγή αρχείων στη γλώσσα DOT τα οποία μετά μεταγλωττίζονται με τη βοήθεια του Graphviz παράγωντας ως εικόνες τις 2 δομές που αναφέραμε. Ακολουθεί μία μέθοδος visitor ενός κόμβου ΑΣΤ (ψευδό κώδικας).

```
1 // Create a visitor event for the
2 // CompoundStatement node
3 // Pass as parameter the current visitor
4 ASTNodeVisitorEvents visitorEvents = new
    ASTNodeVisitorEvents();
5
6 // Print contexts
7 VisitChildrenEvents(context, visitorEvents);
8
9 // Print edge from the parent to the child
10 ASTstream.WriteLine("\"{0}\" -> \"{1}\";",
    parent.Name, child.Name);
```

```
11
12 // Push current rulename
13 ruleStack.Push(child.Name);
14
15 // Continue traversal
16 VisitChildren(context);
17
18 // Pop current rulename
19 ruleStack.Pop();
```

Παραπάνω, παράγουμε τον απαραίτητο κώδικα για την μετατροπή της δομής υψηλού επιπέδου σε μια δενδροειδή μορφή. Αφορά την γραφική αναπαράσταση του ΑΣΤ. Όπως παρατηρείται στον κώδικα υπάρχουν 2 μέθοδοι επίσκεψης των κόμβων του ΑΣΤ. Η `VisitChildrenEvents` χρησιμοποιεί το μοτίβο σχεδίασης παρατηρητή (observer design pattern) χάρη στο οποίο με τη χρήση γεγονότων διαβάζουμε τη λίστα με τα παιδιά του εκάστοτε κόμβου και παράγουμε τον επιθυμητό κώδικα. Η δεύτερη μέθοδος `VisitChildren` είναι υπεύθυνη για τη διάσχιση του ΑΣΤ.

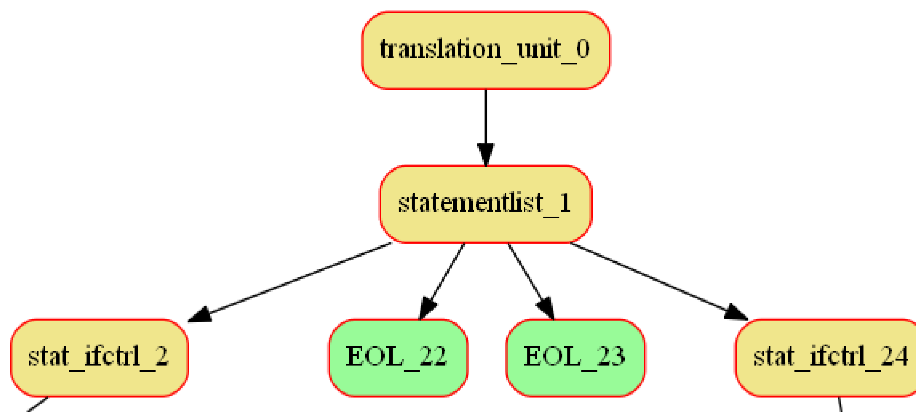
Επίσης πρέπει να διευκρινιστεί η διαφορά μεταξύ ενός *πλαισίου* (context) και ενός *κόμβου* (node). Ένα context περιγράφει όλα όσα ξέρουμε για την αναγνώριση μιας έκφρασης από έναν κανόνα. Περιέχει όλους κόμβους (nodes) που αφορούν αυτόν τον κανόνα.



Σχήμα 7.1: 2 δομές if σε ένα context

Παρατηρούμε ότι ο κόμβος `TranslationUnit` είναι η ρίζα του δέντρου και συνδέεται με 2 `if` δομές. Το context `TranslationUnit_Body` περιέχει τους κόμβους που εκφέρονται από τον συγκεκριμένο κανόνα της γραμματικής.

Πέρα από την γραφική αναπαράσταση του ΑΣΤ, έχουμε και το ΣΣΤ το οποίο προσδιορίζει επακριβώς την δομή της εισόδου και περιέχει όλη την πληροφορία με κάθε λεπτομέρεια. Το συντακτικό δέντρο καταγράφει πως ο συντακτικός αναλυτής (parser) αναγνωρίζει τη δομή της εισόδου με βάση τη γραμματική που έχει κατασκευαστεί. Οι εσωτερικοί κόμβοι του δέντρου είναι τα ονόματα των κανόνων της γραμματικής που αναπτύσσονται ενώ τα φύλλα είναι τερματικοί κόμβοι, τα σύμβολα εισόδου (input tokens).



Σχήμα 7.2: Συντακτικό δέντρο: 2 δομές if

Είναι εμφανές ότι η δομή εδώ έχει αλλάξει αισθητά καθώς προστέθηκαν επιπλέον κανόνες αλλά και τερματικά σύμβολα τα οποία έχουν αφαιρεθεί από την υψηλού επιπέδου αναπαράσταση.

7.3 Αναγνώριση συμβόλων

Ένα ακόμα πέρασμα στο οποίο θα αναφερθούμε είναι αυτό της αναγνώρισης των συμβόλων του προγράμματος. Αυτή η πληροφορία είναι χρήσιμη για την περαιτέρω επεξεργασία του πηγαίου κώδικα καθώς μας επιτρέπει να γνωρίζουμε την χρήση του εκάστοτε συμβόλου έτσι ώστε να το χειριστούμε αναλόγως. *Αυτό το πέρασμα πραγματοποιείται στο ΣΣΤ.* Εδώ γίνεται το γέμισμα του πίνακα συμβόλων (symbol table)

Πρώτα απ' όλα, έχουμε το global scope το οποίο περιέχει όλες τις μεταβλητές και τις κλήσεις συναρτήσεων του προγράμματος. Δεύτερον, έχουμε το function scope το οποίο είναι και το σώμα μιας συνάρτησης. Κρατάμε τα σύμβολα και των 2 scopes. Για την αναγνώριση των συμβόλων χρησιμοποιούμε τη μέθοδο VisitTerminal που μας παρέχει το ANTLR. Αν το αναγνωριστικό (identifier) που συναντήσαμε δεν ανήκει ήδη στον πίνακα συμβόλων, το προσθέτουμε.

Ο πίνακας συμβόλων διατηρεί:

- Σύμβολα που βρίσκονται στο global scope
- Παραμέτρους συναρτήσεων (όχι κλήσεων αλλά δηλώσεων) (local scope)
- Παραμέτρους συναρτήσεων pragma

Σχετικά με τις συναρτήσεις pragma αναφερθήκαμε στην παράγραφο 5.1. Έτσι ώστε να διευκολύνουμε τη διαδικασία παραγωγής κώδικα (code generation) χρησιμοποιούμε συγκεκριμένες συναρτήσεις που ονομάζονται συναρτήσεις pragma (pragma functions) οι οποίες περιγράφουν με ποιον τρόπο θα ερμηνευθεί μια

μεταβλητή. Στόχος μιας συνάρτησης pragma είναι η προσθήκη στον πίνακα συμβόλων μιας τέτοιας μεταβλητής εκ των προτέρων καθώς και η ενημέρωση επιπλέον πληροφοριών που υπάρχουν ανάλογα με τον τύπο της μεταβλητής.

Σε γενικές γραμμές υπάρχουν 2 ξεχωριστές διαδικασίες. Αρχικά εισερχόμαστε στο global scope και από εκεί επισκεπτόμαστε τους υπόλοιπους κόμβους του ΣΣΤ.

```
1  scopeSystem.EnterGlobalScope();
2  VisitChildren(context);
3  scopeSystem.LeaveGlobalScope();
```

Στη συνέχεια, επισκεπτόμαστε συναρτήσεις, αν υπάρχουν, και αποθηκεύουμε τις παραμέτρους τους καθώς και τις μεταβλητές μέσα στο σώμα της κάθε συνάρτησης. Οι παράμετροι και οι μεταβλητές ανήκουν στο local scope.

```
1  // Create a function type and save its function name
2  // at the symbol table
3  type = new T_Function();
4  scopeSystem.InstallIdentifier(functionName, type);
5
6  scopeSystem.EnterFunctionScope();
7
8  // Initialize parameters...
9
10 VisitChildren(context);
11
12 scopeSystem.LeaveFunctionScope();
```

Κεφάλαιο 8

Συμπεράσματα - Μελλοντικές κατευθύνσεις (Conclusions - Future Directions)

Φτάνοντας στο τέλος της εργασίας, κάνουμε μια αναφορά στην πορεία που ακολουθήσαμε για την υλοποίηση της υψηλού επιπέδου αναπαράστασης, μερικά καινούρια χαρακτηριστικά που προστέθηκαν σε επίπεδο λεκτικής και συντακτικής ανάλυσης καθώς την πορεία που μπορεί να ακολουθήσει μελλοντικά και τις προοπτικές που διαθέτει.

8.1 Συμπεράσματα

Στο μεγαλύτερο μέρος της εργασίας ασχοληθήκαμε με την κατασκευή μιας υψηλού επιπέδου αναπαράστασης η οποία χρησιμοποιείται για την αποθήκευση πηγαίου κώδικα στη γλώσσα προγραμματισμού MATLAB. Χρησιμοποιήθηκαν διάφορες τεχνικές για να κατασκευαστεί και αποτελεί το βασικό κορμό του μεταγλωττιστή. Χάρη στο ANTLR ξεκινήσαμε την κατασκευή ενός συντακτικού αναλυτή (parser) που ουσιαστικά είναι αυτός που αναγνωρίζει επιτυχώς την είσοδο βάσει της γραμματικής που δημιουργήσαμε η οποία έχει αρκετά νέα χαρακτηριστικά κυρίως στο τμήμα του λεκτικού αναλυτή όπου ήταν απαραίτητη η χρήση κώδικα για την επιτυχή αναγνώριση συγκεκριμένων δομών.

Το κλειδί για την σωστή διαχείριση της εισόδου είναι η επιτυχής αναγνώριση της στα πρώτα στάδια του μεταγλωττιστή, στο επίπεδο του λεκτικού αναλυτή. Οι δεκαδικοί αριθμοί, τα αλφαριθμητικά, οι μοναδιαίοι τελεστές αλλά και τα κενά (κενοί χαρακτήρες) ήταν μερικά από τα εμπόδια που έπρεπε να επιλυθούν για να μπορέσουμε να συνεχίσουμε. Ακόμη, και οι πίνακες όπου χρησιμοποιο-

ύνται εκτενώς έχουν διαφορετική διαχείριση λόγω των κενών χαρακτήρων που επηρεάζουν την σημασία τους κατά περίπτωση. Εδώ οι δυνατότητες του ANTLR βοήθησαν αρκετά έτσι ώστε να επιλύσουμε αυτά τα προβλήματα (παρόλο που ακόμα το εργαλείο βρίσκεται σε κατάσταση alpha, είναι αρκετά σταθερό).

Από την άλλη μεριά, για την διαχείριση της ενδιάμεσης αναπαράστασης γράφτηκαν αρκετές γραμμές κώδικα. Με τη βοήθεια των μοτίβων σχεδίασης (design patterns) δεν θα ήταν εφικτή η κατασκευή της λόγω του μεγάλου μεγέθους καθώς από ένα σημείο και μετά θα ήταν δύσκολη η συντήρηση ή ακόμα και η ανάγνωση του κώδικα. Η σωστή σχεδίαση σε συνδυασμό με τα χαρακτηριστικά της γλώσσας που χρησιμοποιήσαμε όπως είναι ο πολυμορφισμός, η κληρονομικότητα και γενικότερα το αντικειμενοστραφές μοντέλο προγραμματισμού (object oriented programming - OOP) μας βοήθησε στο να χωρίσουμε όλες τις κλάσεις βάσει του μοτίβου που εφαρμόσαμε και έτσι να ξεχωρίσουμε σε κομμάτια την ανάπτυξη της εφαρμογής.

8.2 Μελλοντικές κατευθύνσεις

Η συγκεκριμένη εργασία περιλαμβάνει μόνο ένα κομμάτι ενός μεταγλωττιστή, για την ακρίβεια ενός μεταφραστή (translator). Όπως είδαμε και σε προηγούμενα κεφάλαια ένας μεταγλωττιστής αποτελείται από πολλά μέρη. Γενικά μιλώντας έχουμε 3 επίπεδα: το εμπρός μέρος (front end), την ενδιάμεση αναπαράσταση (intermediate representation) και το πίσω μέρος (back end). Εμείς ασχοληθήκαμε με το εμπρός μέρος και την ενδιάμεση αναπαράσταση σε μεγάλο βαθμό. Συνήθως το τρίτο μέρος, είτε μιλάμε για μεταγλωττιστή που παράγει εκτελέσιμο κώδικα είτε για ένα μεταφραστή, περιλαμβάνει την διαδικασία παραγωγής κώδικα (code generation) η οποία έχει διευκολυνθεί και με τεχνικές που αναφέρθηκαν κυρίως στο τελευταίο κεφάλαιο.

Συνοψίζοντας, έχοντας υλοποιήσει ένα κομμάτι του μεταγλωττιστή, η εργασία ανοίγει την πόρτα σε επιπλέον εφαρμογές οι οποίες μπορούν να εφαρμοστούν όπως είναι το code generation σε μία άλλη γλώσσα προγραμματισμού καθώς και η εφαρμογή διαφορετικών αλγορίθμων περασμάτων (traversal algorithms) για ποιο αποδοτική πρόσβαση στα δεδομένα.

Βιβλιογραφία

[1] Abran, Alain; Moore, James W.; Bourque, Pierre; Dupuis, Robert; Tripp, Leonard L. (2004). Guide to the Software Engineering Body of Knowledge. IEEE.

[2] Leondes (2002). intelligent systems: technology and applications. CRC Press. p. I-6. ISBN 978-0-8493-1121-5. "1.4 Computers and a First Glimpse at AI (1940s)"ISBN 0-7695-2330-7.

[3] von Neumann, John (1945), First Draft of a Report on the EDVAC (PDF), archived from the original (PDF) on March 14, 2013, retrieved August 24, 2011

[4] Oettinger, Anthony (1967). "The hardware-software complementarity". Communications of the ACM 10 (10): 604–606.

[5] Rayl, A.J.S. (October 16, 2008). "NASA Engineers and Scientists-Transforming Dreams Into Reality". NASA.

[6] Peter, Naur; Brian Randell (7-11 October 1968). Report on a conference sponsored by the NATO SCIENCE COMMITTEE. Garmisch, Germany: Scientific Affairs Division, NATO.

[7] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

[8] Cooper, Keith D.; Torczon, Linda (2012). Engineering a Compiler, Second Edition. Morgan Kaufmann.

[9] Lander, Rich (20 July 2015). "Announcing .NET Framework 4.6". .NET Blog. Microsoft.

[10] ISO/IEC 23271:2012 - Information technology – Common Language Infrastructure (CLI).

[11] Piiroinen, Jarkko (23 February 2008). Visual overview of the Common CLR Language Infrastructure, and how the components relate to each other.

[12] Parr, Terence (2013). The Definitive ANTLR 4 Reference. ANTLR.

[13] Parr, Terence; Harwell, Sam; Fisher, Kathleen (2014). Adaptive LL(*) parsing: The power of dynamic analysis. In Proceedings of the 2014 ACM SIGPLAN International Conference on Object-Oriented Programming Systems Languages and Applications.

[14] The DOT Language (http://graphs.grevian.org/reference).

[15] Graphs images are courtesy of Wikipedia. DOT (graph description language).

[16] MATLAB website. MathWorks.

[17] Cleve Moler (December 2004). "The Origins of MATLAB".

[18] MATLAB Documentation. MathWorks.

[19] "Comparing MATLAB with Other OO Languages". MATLAB. MathWorks.

[20] Pramod G. Joisha; Abhay Kanhere; Prithviraj Banerjee; U. Nagaraj Shenoy; Alok Choudhary (1999). The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler. Technical Report No. CPDC–TR–9909–017. Center for Parallel and Distributed Computing.

[21] Operator precedence. MATLAB Documentation. MathWorks.

[22] Fundamental MATLAB Classes. MATLAB Documentation. MathWorks.

[23] CommonTree class. ANTLR 3 Runtime 3.5.2 API.

[24] Backus, J.W. (1959). "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". Proceedings of the International Conference on Information Processing. UNESCO. pp. 125–132.

[25] Niklaus Wirth (1996). Extended BNF Standard (ISO/IEC 14977:1996)

Παράρτημα (Appendix)

Κατάλογος Σχημάτων

1.1 Μοτίβα σχεδίασης (design patterns) 6
1.2 Μεταγλωττιστής ως 'μαύρο κουτί' 7
1.3 Μεταγλωττιστής 3 φάσεων ως 'μαύρο κουτί' 7
1.4 Διερμηνευτής ως 'μαύρο κουτί' 8

2.1 Αλγόριθμος για την αναγνώριση της λέξης 'new' 10
2.2 Πεπερασμένο αυτόματο για τις λέξεις 'new', 'not' και 'while' . . . 11
2.3 Πεπερασμένο αυτόματο για τον αριθμό '113,4' 12
2.4 Πεπερασμένο αυτόματο για την αναγνώριση μη προσημασμένων ακεραίων αριθμών (γενική περίπτωση) 12
2.5 Αλγόριθμος αναγνώρισης μη προσημασμένων ακεραίων αριθμών . 13
2.6 Πεπερασμένο αυτόματο για την αναγνώριση μη προσημασμένων ακεραίων αριθμών (σωστός τρόπος) 13
2.7 Πίνακας μεταβάσεων 13
2.8 Πίνακας μεταβάσεων (συμπιεσμένη μορφή) 14
2.9 Πεπερασμένο αυτόματο των λέξεων 'new', 'not' 14
2.10 Προτεραιότητες κανονικών εκφράσεων 15
2.11 Πεπερασμένο αυτόματο για τα σχόλια πολλαπλής γραμμής (/* */) 16
2.12 Σχέσεις μεταξύ κανονικών εκφράσεων - NFA - DFA 17
2.13 Πεπερασμένα αυτόματα για τις εκφράσεις m και n 17
2.14 Πεπερασμένο αυτόματο για την έκφραση mn χρησιμοποιώντας την ϵ-μετάβαση 18
2.15 Πεπερασμένο αυτόματο για την έκφραση mn 18
2.16 Παραγωγή ενός Table-driven σαρωτή 20
2.17 Αλγόριθμος Table-driven σαρωτών 21
2.18 Ένα απλό while loop ενός table-driven σαρωτή 22

3.1 Συντακτικός αναλυτής 23
3.2 Γραμματική SN 24
3.3 Γραμματική αριθμητικών εκφράσεων 26
3.4 Ανάπτυξη δεξιότερης παραγωγής 26
3.5 Ανάπτυξη αριστερότερης παραγωγής 27

4.1 Επισκόπηση του CLI[11] 31
4.2 Διάσχιση δέντρου με το μοτίβο ακροατή (listener pattern) 32

4.3 Διάσχιση δέντρου με το μοτίβο επισκέπτη (visitor pattern) 33
4.4 Μη κατευθυνόμενος γράφος[15] 34
4.5 Κατευθυνόμενος γράφος[15] 34
4.6 Γράφος με attributes[15] 35
4.7 Γραφικό περιβάλλον (GUI) του MATLAB R2015a 36

5.1 Δηλώσεις συναρτήσεων (Function definitions) 45

6.1 Μοτίβο σχεδίασης Composite 49
6.2 Βασικές κλάσεις του ΑΣΤ 50
6.3 Τερματικός και μή-τερματικός κόμβος ΑΣΤ 51
6.4 Μοτίβο σχεδίασης Visitor 52
6.5 Σχέση μεταξύ ενός περάσματος και του μοτίβου Visitor 53
6.6 Μοτίβο σχεδίασης Observer 54
6.7 Υλοποίηση και διαχείριση γεγονότων με τη βοήθεια του μοτίβου Observer 55
6.8 Μοτίβο σχεδίασης Iterator 56
6.9 Regular και events iterators 57
6.10 Μοτίβο σχεδίασης Factory method 57
6.11 Εργοστάσιο παραγωγής iterators 58
6.12 Μοτίβο σχεδίασης Façade 59
6.13 Η κλάση που λειτουργεί ως ένα wrapper του κύριου συστήματος . 60
6.14 Οι θεμελιώδεις τύποι του MATLAB 61
6.15 Μερικοί από τους θεμελιώδεις τύπους του MATLAB 61
6.16 Τα 2 βασικά scopes του MATLAB 62

7.1 2 δομές if σε ένα context 66
7.2 Συντακτικό δέντρο: 2 δομές if 67