Σκέψεις και περιγραφή για την εργασία

Σε γενικές γραμμές

Τα θέματα της εργασίας υλοποιούνται στην γλώσσα C.

Δουλεύω με GCC και GDB για compiling και debugging αντίστοιχα και έχει φροντιστεί τα προγράμματα για κάθε θέμα να διαχειρίζονται με ασφάλεια τη μνήμη ώστε να μην υπάρξει θέμα διαχείρισης μνήμης.

Εκτός του βιβλίου της κα. Βίρβου, χρησιμοποιώ και ένα δωρεάν pdf πάνω στους μεταγλωττιστές που καλύπτει πάνω-κάτω την ίδια ύλη που κάνουμε στη σχολή.

- Basics Of Compiler Design, Torben Mogensen (ISBN 978-87-993154-0-6)

  (http://hjemmesider.diku.dk/~torbenm/Basics/basics_lulu2.pdf)

Θέμα 1

Το θέμα 1 είναι αρκετά απλό και δεν του έδωσα πολύ έμφαση. Παρακάτω βρίσκονται οι σκέψεις μου προτού αρχίσω την επίλυση προγραμματικά.

Σκέψεις: Η γραμματική μπορεί εύκολα να εκφραστεί μέσω προγράμματος με συναρτήσεις (functions).

π.Χ. Ο κανόνας παραγωγής:

<Z> ::= (<K>),

μπορεί να γραφεί στην C ως:

```
int parse_Z() {
```

```
        cat('(');
      parse_K();
        cat(')');
       return 0;
}
```

Όπου cat(char) είναι η συνένωση ενός char με τη συμβολοσειρά την όποια κατασκευάζουμε και το 0 που επιστρέφεται επικοινωνεί με το πρόγραμμα και στην ουσία λέει «Όλα καλά!».

π.Χ. Αν συμβολοσειρά = “abc”, τότε η κλήση cat(‘d’) μας δίνει συμβολοσειρά = “abcd”.

Το μόνο που μας λείπει τώρα λοιπόν είναι να λάβουμε μερίμνα ώστε η διαδικασία να τερματίζει, έχω τις εξής ιδέες:

- Τερματισμός ύστερα από δοσμένο αριθμό βημάτων[1].
- Τερματισμός ύστερα από δοσμένο αριθμό χαρακτήρων.
- Τερματισμός ύστερα από δοσμένο αριθμό χρόνου.

Το πιο απλό από αυτά φαίνεται να είναι το πρώτο άρα θα κάνουμε το πρόγραμμα με αυτό!

Τέλος, οι διαζεύξεις ($X_1|X_2$) στους κανόνες παραγωγής κάνουν την λήψη αποφάσεων του προγράμματος μη-ντετερμινιστική. Για να «προσομοιώσουμε» την λήψη απόφασης για αυτές τις διαζεύξεις, θα χρησιμοποιήσουμε random.

<u>Εξήγηση υλοποίησης και σκέψεις</u>:

Είμαι αρκετά ικανοποιημένος με το τελικό αποτέλεσμα, ειδικά με το πως εκτυπώνεται η διαδικασία στην οθόνη. Ο κώδικας θα ήθελα να ήταν λίγο πιο καθαρός αλλά είναι δύσκολο με όλα τα printf και με τους ελέγχους parse() == -1.

Το κύριο μέρος του προγράμματος, η συναρτήσεις parse, λειτουργούν ως εξής:

[1] «Βήμα» εδώ θεωρούμε την εκτέλεση οποιουδήποτε κανόνα παραγωγής.

- Return:
    - 0: Η συνάρτηση επιστρέφει 0, δίνοντας το «σήμα» ότι όλα λειτουργούν σωστά χωρίς προβλήματα.
    - -1: Η συνάρτηση επιστρέφει -1, δίνοντας το «σήμα» ότι κάτι πήγε λάθος. Εδώ δεν το έχω φτιάξει σχεδόν καθόλου το error handling, καθώς επιστρέφουν -1 οι συναρτήσεις μονό αν υπάρξει πρόβλημα με την ανάθεση random τιμής.
- step++:
    - Πριν από κάθε κλήση συνάρτησης parse αύξησε τον αριθμό βημάτων κατά 1.

Η δομή των συναρτήσεων είναι χονδρικά η εξής:

- Έλεγχος υπέρβασης βημάτων, αν υπέρβαση, τότε επέστρεψε αναδρομικά στην main και τερμάτισε.
- Εκτύπωσε πληροφορίες στην οθόνη («Parsing X», «Using production rule <X> ::= y»)
- Κάλεσε την συνάρτηση cat() εάν πρόκειται για προσθήκη συμβόλου στην συμβολοσειρά. Δηλαδή: καθώς κάνεις αριστερότερη παραγωγή, όταν συναντάς τερματικό σύμβολο κάλεσε cat().
- Κάλεσε την κατάλληλη συνάρτηση parse() όταν συναντάς μη-τερματικό σύμβολο στην αριστερότερη παραγωγή.
- Όταν τερματίσει η συνάρτηση parse() που κλήθηκε (και πολύ πιθανόν οι συναρτήσεις που κάλεσε αυτή), επέστρεψε 0 και τερμάτισε.

Πιστεύω ότι αξίζει μια σύντομη αναφορά και για την main().

Αρχικά, στην εκτέλεση του κώδικα, κάνουμε seed την τυχαία γεννήτρια αριθμών έτσι ώστε οι συναρτήσεις parse_M() και parse_G() να λαμβάνουν διαφορετικές τιμές στην μεταβλητή τους «decision». Χωρίς το seed, θα έπρεπε να κάνουμε compile ξανά το πρόγραμμα κάθε φορά που θα το τρέχαμε!

Επομένως, ελέγχουμε αν ο χρήστης έχει δώσει αυστηρά ένα command-line argument, το οποίο ορίζει το αριθμό βημάτων.

Μετά, κατανέμουμε δυναμικά μνήμη για τη συμβολοσειρά που θα κατασκευαστεί. Το * 2 φροντίζει να μην υπάρξει υπερχείλιση.

Τέλος, καλείτε η συνάρτηση parse_Z() και εκτελείται το κύριο μέρος του προγράμματος. Όταν τερματίσει η parse_Z(), εκτυπώνεται το αποτέλεσμα στην οθόνη, απελευθερώνεται η μνήμη για την str και τερματίζει το πρόγραμμα.

Θέμα 2

Σκέψεις: Δεν υπάρχουν αρχικές σκέψεις εδώ, καθώς το τελικό πρόγραμμα δεν έχει καμία σχέση με αυτό που φαντάστηκα προτού αρχίσω να το φτιάχνω.

Την γραφική αναπαράσταση του δέντρου θα την υλοποιήσω χρησιμοποιώντας dot, όπου θα παραχθεί μια όμορφη png εικόνα αντί για ascii χαρακτήρες σε ένα terminal.

- (Σύνδεσμος για την γλώσσα dot από την GraphViz: **https://graphviz.org/doc/info/lang.html**)
- (Σύνδεσμος για λήψη GraphViz: **https://graphviz.org/download/)**

Πριν αρχίσει ο προγραμματισμός θα πρέπει να βρω τι είδος γραμματική είναι.

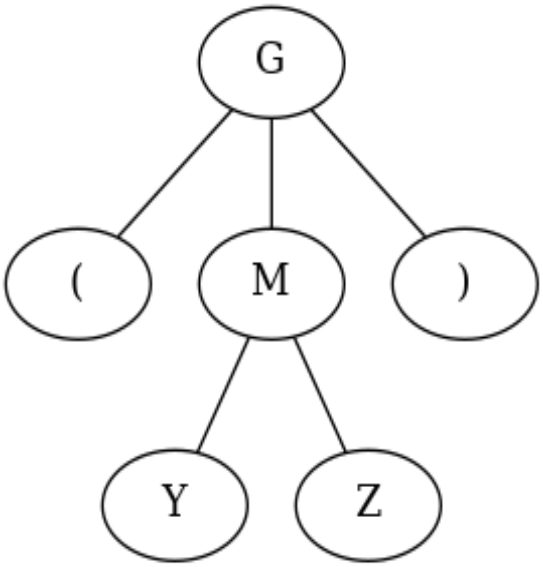
Αποδεικνύεται τελικά ότι η γραμματική είναι LL(1).

Σε δοκιμαστικό πρόγραμμα, αποφάσισα να υλοποιήσω αναλυτή για μια γραμματική παρόμοια με το θέμα που δίνεται ως παράδειγμα στις διαφάνειες συντακτικής ανάλυσης του μαθήματος.

Από την προσπάθεια μου στο δοκιμαστικό, έχω τα εξής σχόλια όσον αφορά τη συντακτική ανάλυση:

- Η συντακτική ανάλυση θα γίνει με συντακτικό πίνακα και στοίβα, καθώς η γραμματική είναι LL(1).
- Χρήση δυο δομών δεδομένων: Στοίβα και δισδιάστατος πίνακας (για το συντακτικό πίνακα).
- Αυτές οι δομές θα βρίσκονται σε ξεχωριστά .c και .h αρχεία για να υπάρχει έμφαση στην συντακτική ανάλυση και όχι στην υλοποίηση των δομών.

Ύστερα από την υλοποίηση:



Εικόνα 1: Παράδειγμα χρήσης dot για γραφική αναπαράσταση

Πιστεύω ότι «υπερπαραγωγή» περιγράφει αρκετά καλά το πως έφτιαξα τελικά αυτό το θέμα. Αν κάτσω να περιγράψω λεπτομερής κάθε κομμάτι του προγράμματος δεν νομίζω να τελειώσουμε ποτέ, επομένως θα προσπαθήσω όσο καλύτερα να εξάγω την ουσία από αυτό το τέρας. Σε περίπτωση που θέλετε να μου κάνετε ερωτήσεις, είμαι πολύ πρόθυμος να επικοινωνήσω μαζί σας μέσω e-mail ή ακόμα καλύτερα από κοντά! Συγχωρείστε με για τη περιπλοκότητα του :,).

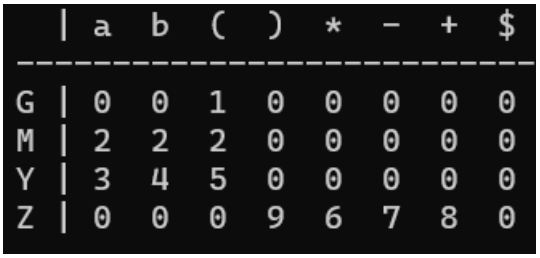
Την επεξήγηση του θέματος θα την χωρίσω σε δύο κομμάτια:

- **Συντακτική ανάλυση** με χρήση συντακτικού πίνακα και στοίβας.
- **Παραγωγή κώδικα dot** για την απεικόνιση του συντακτικού δέντρου.

Συντακτική Ανάλυση

Όπως αναφέρθηκε πριν, η ανάλυση γίνεται με συντακτικό πίνακα (εφαρμόζεται στο syntaxTable.c) και με στοίβα (εφαρμόζεται στο stack.c). Ο αλγόριθμος ανάλυσης γίνεται ακριβώς όπως και στο βιβλίο και στις διαφάνειες.

Ο συντακτικός πίνακας που έχω φτιάξει είναι απλός 2D πίνακας με αρκετές επιπλέον ιδιότητες



	a	b	(	)	*	-	+	$
G	0	0	1	0	0	0	0	0
M	2	2	2	0	0	0	0	0
Y	3	4	5	0	0	0	0	0
Z	0	0	0	9	6	7	8	0

Εικόνα 2: Συντακτικός πίνακας τυπωμένος στο τερματικό με τη κλήση της printSyntaxTable()

Το σημαντικότερο κομμάτι του είναι η συνάρτηση productionExists(), καθώς είναι μέρος του αλγόριθμου αυτής της συντακτικής ανάλυσης, και ελέγχει αν υπάρχει κανόνας παραγωγής σε οποιοδήποτε [x,a] κελί του πίνακα.

Η κανόνες παραγωγής είναι κωδικοποιημένοι με αριθμούς από 1 έως 9 και με χρήση map αντιστοιχίζονται στους κατάλληλους κανόνες σε μορφή string (δηλαδή π.Χ. “(M)”).

Όσον αφορά την στοίβα, είναι τελείως by-the-book η εφαρμογή της τόσο ως δομή δεδομένων όσο κι ως το κύριο εργαλείο της συντακτικής ανάλυσης. Πολύ απλά, οπού λέει στο βιβλίο ότι η στοίβα στο σημείο X πρέπει να κάνει το Y, έτσι γίνεται και στο πρόγραμμα.

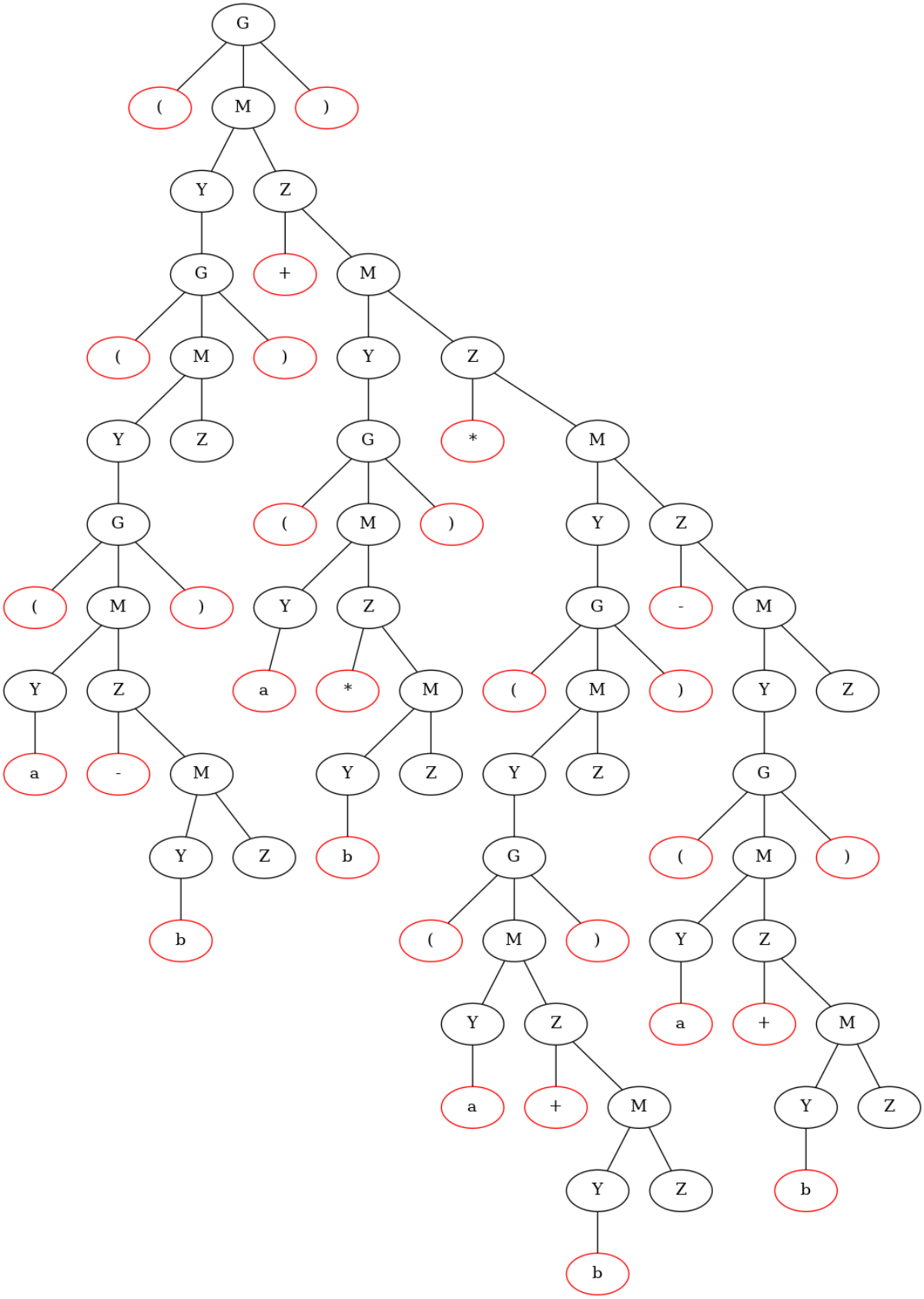
Παραγωγή κώδικα dot

Εδώ εξηγείται το πως παράγεται η τελική εικόνα του συντακτικού δέντρου όπως αυτό στην Εικόνα 3.

Συνοπτικά γίνεται το εξής:

1. Κάθε φορά που αναλύεται μη-τερματικό σύμβολο από την κορυφή της στοίβας συμβόλων, προστίθεται ένα κομμάτι κώδικα dot στο dotString, το όποιο περιέχει όλο το κώδικα dot.
2. Στο τέλος, δηλαδή όταν αναλυθεί το τερματικό σύμβολο $ από την κορυφή της στοίβας συμβόλων, το dotString γράφεται ολόκληρο σε ένα αρχείο με κατάληξη .dot. Ύστερα καλείται η εντολή dot από το τερματικό μέσω τη κλήση της system()[2] και παράγεται η τελική εικόνα .png.

[2] Η κλήση της system() λειτουργεί σε οποιοδήποτε λειτουργικό σύστημα, αρά δεν υπάρχει ανησυχία για platform compatibility.



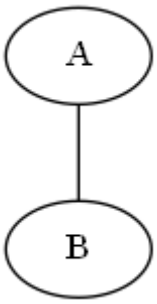
*Εικόνα 3: Παράγωγη συντακτικού δέντρου για την έκφραση "(((a-b)+(a*b))*((a+b)-(a+b)))$"*

Εδώ αξίζει να γίνει περαιτέρω περιγραφή του **1.**

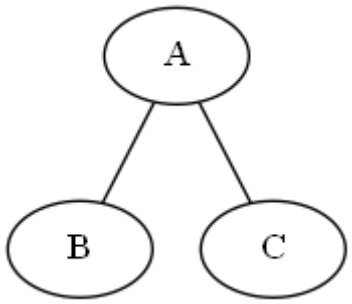
Όταν λοιπόν αναλύεται μη-τερματικό σύμβολο του οποίου ο κανόνας παραγωγής υπάρχει στη θέση του πίνακα M(x,a), εκτελείται ένας «αλγόριθμος» που παράγει κώδικα dot.

Κάθε τέτοια παραγωγή κώδικα προσθέτει στο dotString τα εξής:

1. Ένα σχόλιο (//) που δείχνει ποιος κανόνας παραγωγής αναλύεται.
2. Μια ή περισσότερες αναθέσεις νέων ή παλιών μεταβλητών. Η ανάθεση μεταβλητών έχει μορφή X – Y;

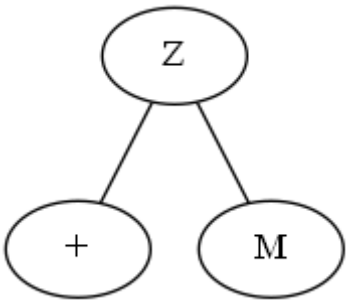


Εικόνα 4: Γραφική αναπαράσταση της A—B;



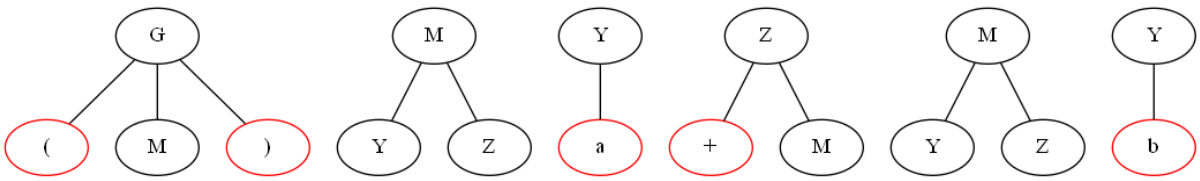
Εικόνα 5: Γραφική αναπαράσταση της: A—B; B—A;

3. Αντίστοιχες αναθέσεις label για κάθε μεταβλητή.



Εικόνα 6: Ανάθεση label για κάθε μια από τις μεταβλητές της εικόνας 5.

Η περιπλοκότητα βρίσκεται στο βήμα 2. Σε μια πρώτη ματιά, όλα φαίνονται ότι θα είναι καλά εάν εφαρμόσουμε τον αλγόριθμο έτσι χωρίς κάτι επιπλέον. Το πρόγραμμα όμως, κάθε φορά που θα αναλυθεί ένα καινούργιο μη-τερματικό, δεν θα θυμάται ποιες μεταβλητές έχουν χρησιμοποιηθεί προηγουμένως, με άλλα λόγια δεν θα έχει μνήμη. Το πρόβλημα της ανύπαρκτης μνήμης γίνεται φανερό με την τελική εικόνα που παράγεται.



Εικόνα 7: Τελική εικόνα συντακτικού δέντρου της πρότασης "(a+b)$" χωρίς μνήμη

Αμέσως φανερώνεται το πρόβλημα και η λύση του.

Το πρόβλημα: Παράγεται ένα ξεχωριστό δέντρο για κάθε ανάλυση μη-τερματικού συμβόλου.

Στόχος: Να συνδεθούν αυτά τα ξεχωριστά δέντρα και να παραχθεί ΈΝΑ που περιέχει όλα τα σύμβολα.

Η λύση: Μνήμη για τις προηγούμενες αναλύσεις μη-τερματικών.

Υλοποίηση της λύσης

Για να αποθηκεύονται οι προηγούμενες αναλύσεις μη-τερματικών, θα χρησιμοποιηθεί μια ουρά (queue) η οποία θα έχει τις βασικές λειτουργίες enqueue, dequeue καθώς και μια ειδική λειτουργία για το πρόγραμμα, την swapFront().

Αποθήκευση (enqueue)

Κάθε μη-τερματικό που αναλύεται θα αποθηκεύεται στην ούρα ως μια συμβολοσειρά μήκους 3. Όπου ο πρώτος χαρακτήρας είναι το σύμβολο και ο δεύτερος και ο τρίτος είναι η μεταβλητή στο κώδικα dot που αντιστοιχεί στο σύμβολο. Για παράδειγμα, έστω ότι έχουμε το σύμβολο A και για αυτό δόθηκε όνομα μεταβλητής dot το G0, το σύμβολο θα αποθηκευτεί στην ουρά ως AG0.

Προσπέλαση (dequeue)

Για κάθε μη-τερματικό σύμβολο που αναλύεται θα γίνεται ο εξής έλεγχος: Είναι το μπροστινό μέρος της ουράς άδειο; Αν όχι, ταιριάζει το μη-τερματικό σύμβολο που περιέχει με το σύμβολο το οποίο αναλύεται (**Συνθήκη Ι**);

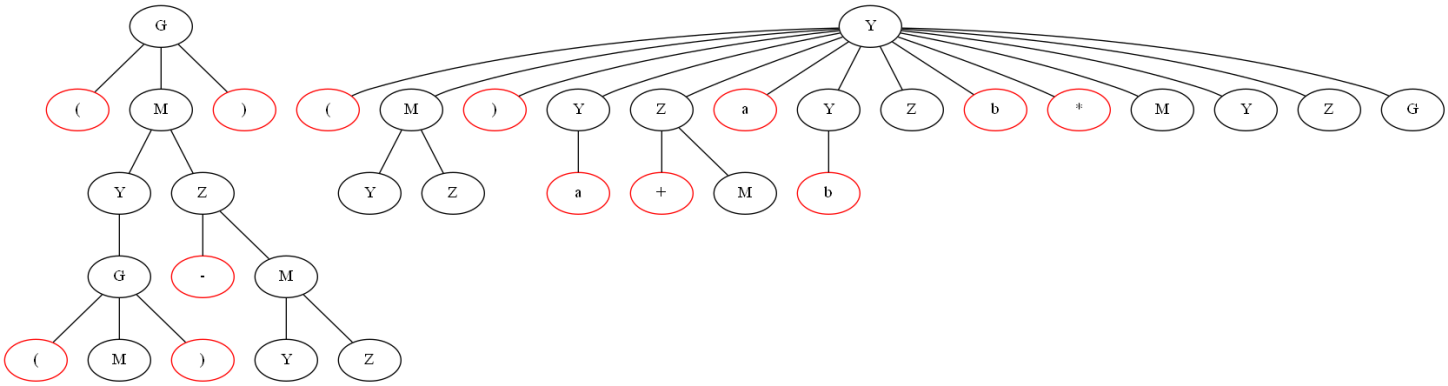
Αν η συνθήκη αυτή είναι ψευδής, δηλαδή το μπροστά μέρος είναι άδειο τότε δεν πραγματοποιείται προσπέλαση του μπροστινού μέρους της ουράς και ο αλγόριθμος συνεχίζεται κανονικά.

Αν η συνθήκη είναι αληθής, τότε γίνεται προσπέλαση στο μπροστά της ουράς και η αριστερή μεταβλητή dot στο βήμα 2 της

παραγωγής κώδικα θα είναι η μεταβλητή η οποία βρισκόταν στη μνήμη.

Κάτι μυρίζει καμένο…

Στην εκτέλεση της προσπέλασης παρουσιάζεται και ένα ακόμα τελευταίο πρόβλημα. Ας το εξετάσουμε βλέποντας την εικόνα που θα παραχθεί.



Εικόνα 8: Τελική εικόνα συντακτικού δέντρου της πρότασης "((a-b)(a+b))$" με χρήση μνήμης αλλά χωρίς κάποιους ελέγχους*

Τελικά παράχθηκε μια σκέτη φρίκη. Το πρόβλημα εδώ βρίσκεται στη σειρά αποθήκευσης των δεδομένων. Εφόσον χρησιμοποιούμε ουρά, έχουμε LIFO (last in first out). Υπάρχει πρόβλημα με αυτό το τρόπο προσπέλασης επειδή έχουμε τον κανόνα M -> YZ. Όταν αναλυθεί αυτός ο κανόνας θα αποθηκευτούν **ΔΥΟ** δεδομένα στην ουρά. Στην ακριβώς επόμενη ανάλυση που θα αναλυθεί το μη-τερματικό Y, το πρόγραμμα θα βρει το Y στο μπροστά της ουράς και όλα καλά. Τώρα όμως το Y παράγει το G υπάρχει πρόβλημα, επειδή στο μπροστά της ουράς υπάρχει το Z. Όταν αναλυθεί το G, το πρόγραμμα δεν θα κάνει dequeue, επειδή G != Z, και έτσι χάλασε το δέντρο και παράγεται η παραπάνω φρίκη.

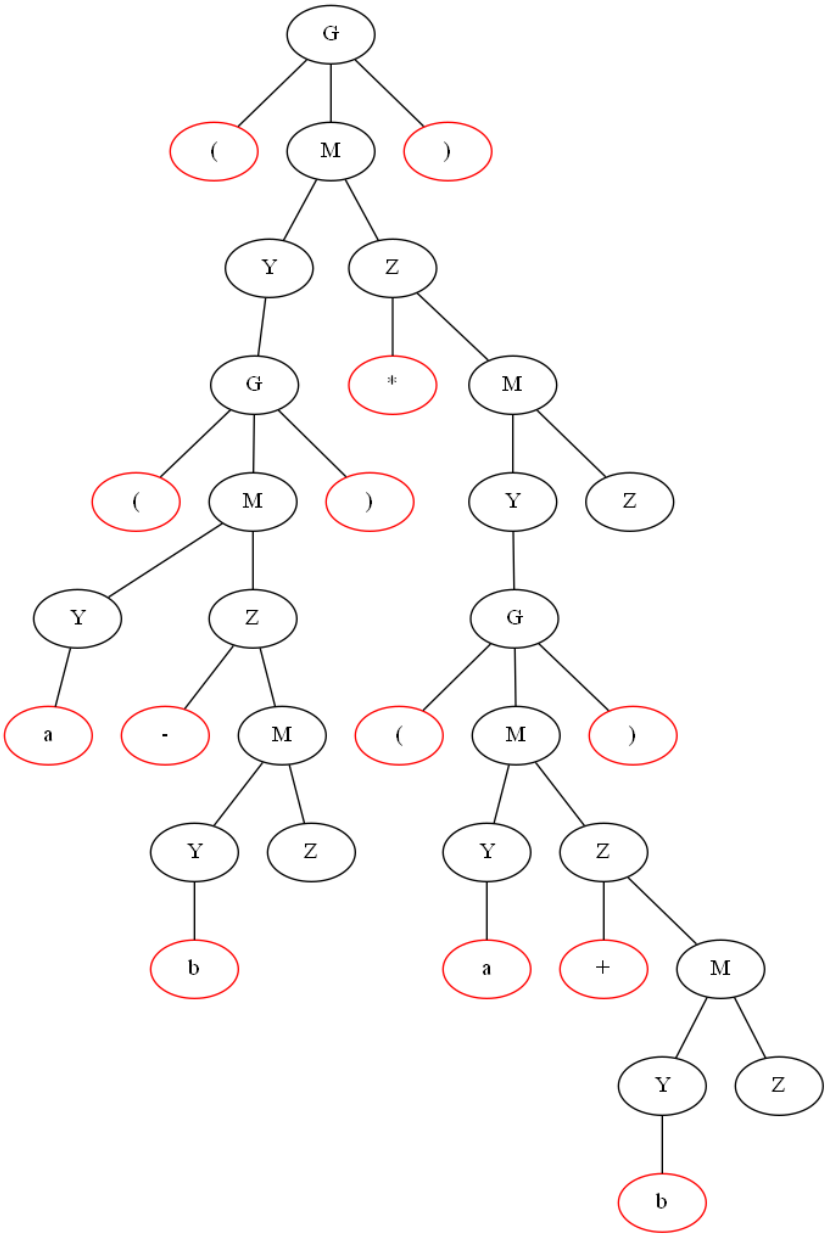
Εικόνα 9: Γραφική αναπαράσταση της κατάστασης της ουράς στην παραπάνω περιγραφή του προβλήματος

Για να λυθεί αυτό το πρόβλημα θα κλέψουμε!

Θα χρησιμοποιήσουμε την swapFront() για να βρούμε το στοιχείο της ουράς το οποίο θα ικανοποιεί την συνθήκη Ι.

Εικόνα 10: Γραφική αναπαράσταση της swapFront() στο παράδειγμα μας.

That's all, folks!



Εικόνα 11: Τελική εικόνα συντακτικού δέντρου της πρότασης "((a-b)(a+b))$" με χρήση μνήμης και swapFront()*

Θέμα 3

Επιστρέφουμε στην απλότητα με το τελευταίο θέμα, το οποίο το χωρίζω σε δύο κομμάτια:

- Pattern Matching (Regex) στο patterns.l
- Ανάλυση ορθότητας στο tokenizer.c

Και τα δύο είναι πολύ απλά στην υλοποίηση και ευτυχώς δεν θα χρειαστούμε 8 σελίδες!

Pattern Matching

Αυτό συνέβη στο patterns.l, όπου με τη χρήση υπερβολικά απλού regex, εφαρμόζονται οι κανόνες αναγνώρισης μιας λεκτικής μονάδας (token). Εδώ για κάθε μονάδα που αναγνωριστεί το πρόγραμμα επιστρέφει έναν αριθμό που αντιστοιχεί στη μονάδα αυτή (βλ. Tokens.h). Οι τιμές που επιστρέφει το πρόγραμμα δίνονται ως έξοδος της συνάρτησης yylex().

Ανάλυση ορθότητας

Σκοπός εδώ είναι να αναλύσουμε τις τιμές που επιστρέφουν οι yylex() που καλούμε. Για να καλυφθούν οι ανάγκες του θέματος αρκεί να κληθεί η yylex() δύο φορές. Η πρώτη φόρα για το σχήμα (τρίγωνο, τετράγωνο) και η δεύτερη για τη δήλωση σημείων (ABC, BCDA).

Για να αναλύσουμε την ορθότητα του συνδυασμού αυτών των δυο τιμών που επιστρέφουν οι yylex(), αρκεί να κάνουμε μερικούς γενικούς ελέγχους και κάποιους που ορίζει το θέμα (όπως η επανάληψη σημείων)

Αν περάσουμε όλους τους ελέγχους το πρόγραμμα επιστρέφει 0 και το σχήμα μας αναγνωρίστηκε επιτυχώς!