**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:

<Ζ> ::= (<Κ>),

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]](#footnote-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 (X1|X2) στους κανόνες παραγωγής κάνουν την λήψη αποφάσεων του προγράμματος μη-ντετερμινιστική. Για να «προσομοιώσουμε» την λήψη απόφασης για αυτές τις διαζεύξεις, θα χρησιμοποιήσουμε random.

Εξήγηση υλοποίησης και σκέψεις:

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:

* 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 visual αναπαράσταση του δέντρο 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/)

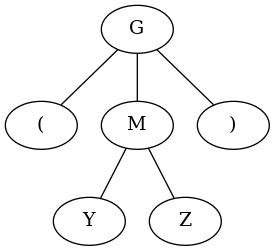
Πριν αρχίσει ο προγραμματισμός θα πρέπει να βρω τι είδος γραμματική είναι.

Αποδεικνύεται τελικά ότι η γραμματική είναι LL(1).

Σε δοκιμαστικό πρόγραμμα, αποφάσισα να υλοποιήσω αναλυτή για παρόμοια με το θέμα γραμματική 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).



Παράδειγμα χρήσης dot για γραφική αναπαράσταση συντακτικού δέντρου

* 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:

Πιστεύω ότι «υπερπαραγωγή» περιγράφει αρκετά καλά το πως το 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 :,).

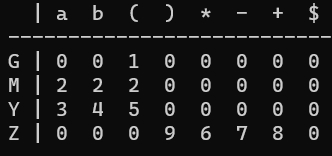
Την επεξήγηση του θέματος θα την χωρίσω σε δύο κομμάτια:

* 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



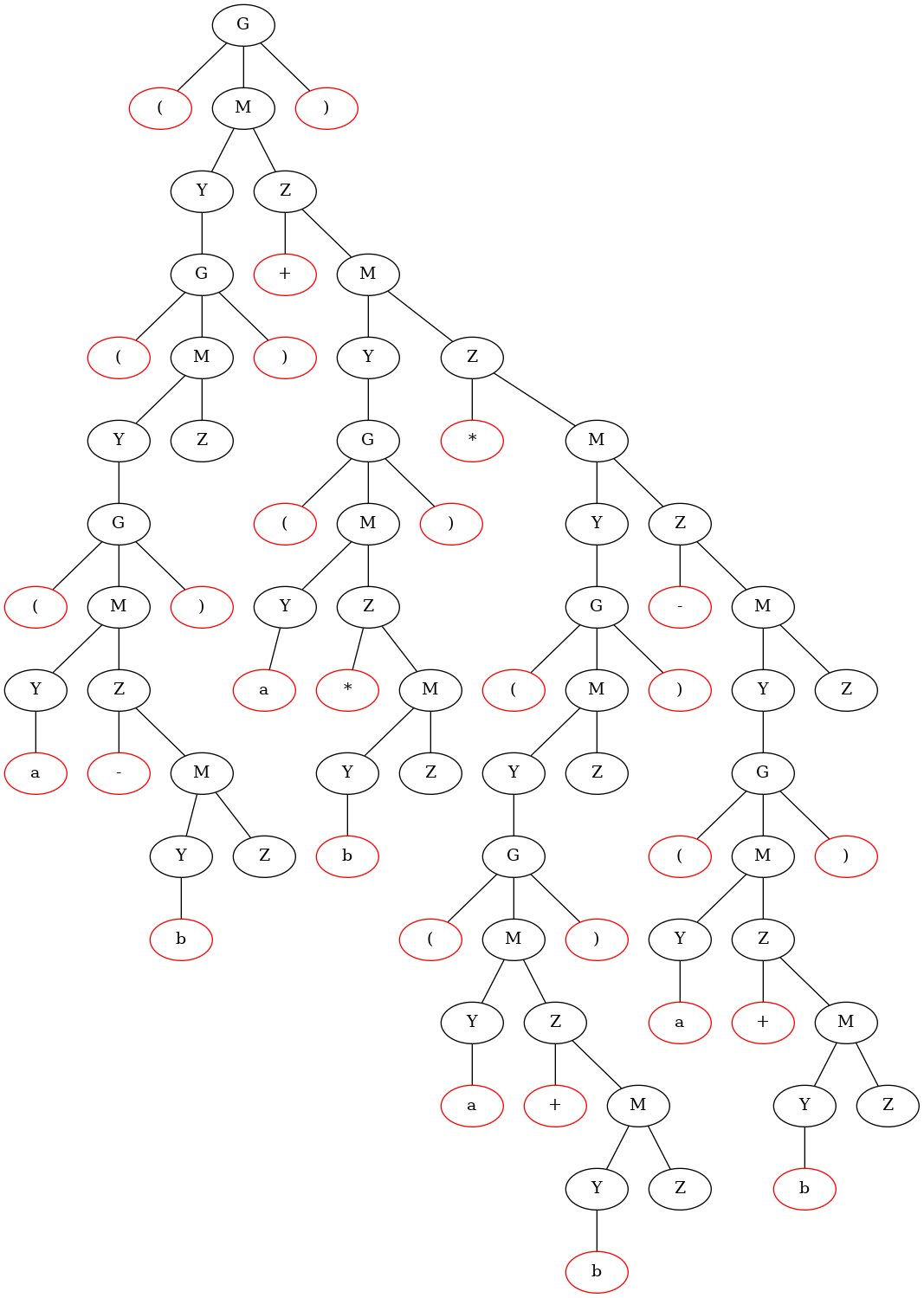
Συντακτικός πίνακας εκτυπωμένος χρησιμοποιώντας τη συνάρτηση printSyntaxTable()

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 Χ πρέπει να κάνει το Υ, έτσι και στο πρόγραμμα γίνεται.

**Παραγωγή κώδικα .dot**



Παράγωγη συντακτικού δέντρου για την έκφραση (((a-b)+(a\*b))\*((a+b)-(a+b)))$

1. «Βήμα» εδώ θεωρούμε την εκτέλεση οποιουδήποτε κανόνα παραγωγής. [↑](#footnote-ref-1)