Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

# Μελέτη και βελτίωση της επίδοσης του συντακτικού αναλυτή Packrat

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΝΙΚΟΣ ΜΑΥΡΟΓΕΩΡΓΗΣ**

**Επιβλέπων :** Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

# Μελέτη και βελτίωση της επίδοσης του συντακτικού αναλυτή Packrat

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΝΙΚΟΣ ΜΑΥΡΟΓΕΩΡΓΗΣ**

**Επιβλέπων :** Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Ιουλίου 2020.

| ........................................... | ........................................... | ........................................... |
| --- | --- | --- |
| Νικόλαος Σ. Παπασπύρου<br>Καθηγητής Ε.Μ.Π. | Δημήτρης Φωτάκης<br>Αν. Καθηγητής Ε.Μ.Π. | Γεώργιος Γκούμας<br>Επικ. Καθηγητής Ε.Μ.Π. |

Αθήνα, Ιούλιος 2020

...........................................

**Νίκος Μαυρογεώργης**

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Νίκος Μαυρογεώργης, 2020.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

# Περίληψη

Πρακτικά όλες οι γλώσσες, είτε φυσικές είτε γλώσσες μηχανής, βασίζονται στην έκφραση της πληροφορίας με γραμμικό τρόπο. Συνήθως η αναπαράσταση γίνεται με τη μορφή μίας συμβολοσειράς, που είναι μια ακολουθία χαρακτήρων από ένα τυποποιημένο σύνολο. Οποιαδήποτε εφαρμογή επεξεργασίας γλώσσας πρέπει να μετατρέψει τις συμβολοσειρές σε πιο αφηρημένες δομές όπως λέξεις, φράσεις, προτάσεις, εκφράσεις ή εντολές. Συντακτική ανάλυση (parsing) είναι η διαδικασία που εξάγει χρήσιμη δομημένη πληροφορία από γραμμικό κείμενο.

Το packrat parsing είναι μία τεχνική συντακτικής ανάλυσης που βασίζεται στις *parsing expression grammars (PEGs)*, μία παραλλαγή των γραμματικών χωρίς συμφραζόμενα. Ένας packrat parser παρέχει την ισχύ και την απλότητα των καθοδικών συντακτικών αναλυτών, ωστόσο εγγυάται γραμμικό χρόνο εκτέλεσης. Οποιαδήποτε γλώσσα που ορίζεται από μία $LL(k)$ ή $LR(k)$ γραμματική μπορεί να αναγνωριστεί από έναν packrat parser, καθώς και πολλές άλλες γλώσσες που οι συμβατικοί αλγόριθμοι γραμμικού χρόνου δεν υποστηρίζουν.

Σκοπός της παρούσας εργασίας είναι αφενός η υλοποίηση ενός συντακτικού αναλυτή packrat στη κλασική του μορφή, αφετέρου η βελτίωση της επίδοσής του είτε τροποποιώντας τον αρχικό αλγόριθμο, είτε παραλληλοποιώντας τον ώστε να τρέξει αποδοτικότερα σε ένα πολυπύρηνο σύστημα.



## Λέξεις κλειδιά

Συντακτική Ανάλυση Packrat, Parsing Expression Grammars, Γεννήτορες συντακτικών αναλυτών, Παράλληλη εκτέλεση.

// Abstract

Practically all languages, either natural or artificial languages, are based on expressing information in a linear way. Usually, this representation is a string, which is a sequence of characters from a fixed set. Every language processing application must convert these strings into more abstract structures like words, phrases, sentences, expressions or instructions. Parsing is the process of extracting useful structured information from linear text.

Packrat parsing is a parsing technique based on *parsing expression grammars (PEGs)*, a variation of context-free grammars. A packrat parser provides the power and flexibility of top-down parsing with backtracking and unlimited lookahead, but nevertheless guarantees linear parse time. Any language defined by an $LL(k)$ or $LR(k)$ grammar can be recognized by a packrat parser, in addition to many languages that conventional linear-time algorithms do not support.

The purpose of this diploma dissertation is on one hand the implementation of a packrat parser in its original version, and on the other hand to improve its performance, either through modifications to the standard version of the algorithm, or through parallelizing it in order to improve execution time in a multicore machine.

## Key words

Packrat parsing, Parsing Expression Grammars, Parser generators, Parallel Execution.

# Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διατριβής, κ. Νίκο Παπασπύρου.

Νίκος Μαυρογεώργης,

Αθήνα, 3η Ιουλίου 2020

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-20, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2020.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

# Περιεχόμενα

**Περίληψη** . . . . . 5

**Abstract** . . . . . 7

**Ευχαριστίες** . . . . . 9

**Περιεχόμενα** . . . . . 11

**Κατάλογος σχημάτων** . . . . . 13

**1. Εισαγωγή** . . . . . 15
- 1.1 Ορισμός συντακτικού . . . . . 15
- 1.2 Parsing Expression Grammars . . . . . 16
- 1.3 Packrat Parsing . . . . . 16
- 1.4 Αυτόματη παραγωγή των Packrat Parsers . . . . . 16
- 1.5 Packrat Parsing με Ελαστικό Κυλιόμενο Παράθυρο . . . . . 16
- 1.6 Παράλληλο Packrat Parsing . . . . . 17
- 1.7 Η δομή της εργασίας . . . . . 17
- 1.8 Κώδικας . . . . . 17

**2. Parsing Expression Grammars** . . . . . 19
- 2.1 Ορισμός των Parsing Expression Grammars . . . . . 19
- 2.2 Ένα παράδειγμα μιας PEG . . . . . 21
- 2.3 Άπληστη και Μη Ντετερμινιστική επανάληψη . . . . . 22
- 2.4 Συντακτική ανάλυση ολόκληρων συμβολοσειρών . . . . . 22
- 2.5 Αριστερή Αναδρομή . . . . . 23

**3. Packrat Parsing** . . . . . 25
- 3.1 Θεμέλια . . . . . 25
- 3.2 Λειτουργία ενός packrat parser . . . . . 26
  - 3.2.1 Περιγραφή του συντακτικού αναλυτή με δυναμικό προγραμματισμό . . . . . 26
  - 3.2.2 Περιγραφή του Packrat Parser . . . . . 27
  - 3.2.3 Ενσωματωμένη λεκτική ανάλυση . . . . . 28
- 3.3 Υλοποίηση . . . . . 28
  - 3.3.1 Υλοποίηση της γραμματικής . . . . . 28
  - 3.3.2 Υλοποίηση του συντακτικού αναλυτή . . . . . 30

**4. Γεννήτορας Συντακτικών Αναλυτών Packrat** . . . . . 35
- 4.1 Τυπική Περιγραφή Γραμματικών . . . . . 35
- 4.2 Στιγμιότυπο μίας μετα-γραμματικής που περιγράφει Parsing Expresssion Grammars 36
- 4.3 Γεννήτορας συντακτικών αναλυτών packrat . . . . . 37

**5. Packrat Parsing με Ελαστικό Κυλιόμενο Παράθυρο** . . . . . 39
5.1 Εισαγωγή . . . . . 39
5.2 Κυλιόμενο παράθυρο . . . . . 39
5.3 Δυναμική απενεργοποίηση μη τερματικών συμβόλων . . . . . 40
5.4 Elastic Packrat Parsing . . . . . 41

**6. Παράλληλο Packrat Parsing** . . . . . 43
6.1 Παράλληλος DP packrat . . . . . 43
6.2 Παράλληλο Καθοδικό Packrat Parsing . . . . . 44
6.2.1 Παραλληλοποίηση της Διατεταγμένης Επιλογής . . . . . 44
6.2.2 Αναδρομική δημιουργία νημάτων . . . . . 47

**7. Πειραματικά Αποτελέσματα** . . . . . 51
7.1 Packrat με ελαστικό κυλιόμενο παράθυρο . . . . . 51
7.2 Παράλληλο packrat parsing . . . . . 52
7.3 Τελικά Αποτελέσματα . . . . . 53

**8. Συμπεράσματα** . . . . . 55

**Βιβλιογραφία** . . . . . 57

**Παράρτημα** 59

**Α. Μία PEG περιγράφει τυπικά την ίδια τη σύνταξή της** . . . . . 59

# Κατάλογος σχημάτων


2.1 Μία CFG για μία απλή αριθμητική γλώσσα . . . . . . . . . . . . . . . . . . . . . . . . 21
2.2 Μία PEG για μία απλή αριθμητική γλώσσα . . . . . . . . . . . . . . . . . . . . . . . . 21
2.3 Αναγνωρίζοντας το '$(12 - 3)$' με βάση τη γραμματική στο Σχήμα 2.2 [Ford02a] . . . 21
2.4 Επέκταση της γραμματικής ώστε να εξετάζει όλη την είσοδο μέχρι τέλους . . . . . . 23
2.5 Αριστερά και δεξιά προσεταιριστικοί τελεστές σε μία CFG . . . . . . . . . . . . . . 23
2.6 Ισοδύναμη PEG χωρίς αριστερή αναδρομή . . . . . . . . . . . . . . . . . . . . . . . 23

3.1 Μία PEG γραμματική για αριθμητικές εκφράσεις . . . . . . . . . . . . . . . . . . . . 26
3.2 Παράδειγμα εκτέλεσης Packrat Parsing [Ford02b] . . . . . . . . . . . . . . . . . . . . 28
3.3 Ενσωματωμένη λεκτική ανάλυση για την εντολή if στην Java . . . . . . . . . . . . . 29
3.4 Μοντελοποίηση των δομικών στοιχείων μίας PEG . . . . . . . . . . . . . . . . . . . 29
3.5 Μοντελοποίηση μίας PEG . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.6 Μοντελοποίηση του Packrat Parser . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.7 Συντακτική ανάλυση ενός μη τερματικού . . . . . . . . . . . . . . . . . . . . . . . . 32
3.8 Συντακτική ανάλυση ενός τερματικού . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.9 Συντακτική ανάλυση της ακολουθίας και της διατεταγμένης επιλογής . . . . . . . . 33

4.1 Η διαδικασία αυτόματης παραγωγής ενός packrat parser για Java προγράμματα . . . 36
4.2 Μία PEG περιγράφει τυπικά τη συντακτική δομή της . . . . . . . . . . . . . . . . . 36
4.3 Δημιουργία στιγμιοτύπου του πρώτου κανόνα της μετα-γραμματικής . . . . . . . . . 37
4.4 Ο κόμβος του AST που αφορά το Block . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.5 Διάσχιση ακολουθίας και δημιουργία στιγμιοτύπων για τη γραμματική-στόχο . . . . 38
4.6 Στιγμιότυπα για τον πρώτο κανόνα της μετα-γραμματικής . . . . . . . . . . . . . . . 38

5.1 Κυλιόμενο παράθυρο για τον πίνακα υπομνηματισμού [Kura15] . . . . . . . . . . . 40
5.2 Απενεργοποιήση μη τερματικού . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.3 Ένα απενεργοποιημένο μη τερματικό αναλύεται απευθείας χωρίς τη συμμετοχή του πίνακα ενδιάμεσων αποτελεσμάτων. . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.4 Αξιοποιήση ενός κελιού για ένα μη τερματικό προτού λήξουν οι ευκαιρίες και μόνιμη ενεργοποίησή του . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.5 Ελαστικό κυλιόμενο παράθυρο [Kura15] . . . . . . . . . . . . . . . . . . . . . . . . 42
5.6 Δημιουργία κλειδιού και δείκτη . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

6.1 Δημιουργία και τερματισμός νημάτων με βάση την ιεραρχία για κάθε υποέκφραση . 45
6.2 Έλεγχος της καθολικής μεταβλητής για πρόωρο τερματισμό . . . . . . . . . . . . . 45
6.3 Κανόνας με πολλές εναλλακτικές στη διατεταγμένη επιλογή . . . . . . . . . . . . . 46
6.4 Έλεγχος για την εκτέλεση ή μη της διατεταγμένης επιλογής με βάση το μέγεθος . . . 46
6.5 Κλείδωμα του κελιού . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.6 Αναμονή για υπολογισμό του κελιού από άλλο νήμα . . . . . . . . . . . . . . . . . . 47
6.7 Ορισμός της κλάσης του νήματος-εργάτη . . . . . . . . . . . . . . . . . . . . . . . . 47
6.8 Το νήμα-εργάτης δημιουργεί νήματα-εργάτες . . . . . . . . . . . . . . . . . . . . . . 48
6.9 Αναμονή και τερματισμός των νημάτων . . . . . . . . . . . . . . . . . . . . . . . . . 48
6.10 Έλεγχος της μεταβλητής του γονέα για πρόωρο τερματισμό . . . . . . . . . . . . . 49

# Κεφάλαιο 1

# Εισαγωγή

Κάθε γλώσσα προγραμματισμού από τη φάση σχεδιασμού της έχει ακριβείς κανόνες που καθορίζουν τη συντακτική δομή των σωστά διατυπωμένων προγραμμάτων της [Aho06]. Στη C, για παράδειγμα, ένα πρόγραμμα αποτελείται από συναρτήσεις, μία συνάρτηση από δηλώσεις και εντολές, μία εντολή από εκφράσεις και ούτω καθεξής. Πρακτικά όλες οι γλώσσες που χρησιμοποιούνται συχνά σήμερα, τόσο φυσικές γλώσσες όσο και γλώσσες μηχανής, βασίζονται στην έκφραση της πληροφορίας με γραμμικό τρόπο, όπως οι ακολουθίες συμβόλων [Ford02a]. Το κείμενο σε μία γραπτή γλώσσα συνήθως αναπαρίσταται ως μία *συμβολοσειρά*, δηλαδή μια ακολουθία χαρακτήρων που προέρχονται από ένα τυποποιημένο σύνολο. Το πρώτο πράγμα που πρέπει να κάνει οποιαδήποτε εφαρμογή επεξεργασίας γλώσσας, είναι να μετατρέψει αυτές τις συμβολοσειρές σε πιο αφηρημένες δομές όπως λέξεις, φράσεις, προτάσεις, εκφράσεις ή εντολές. Η διαδικασία που εξάγει τέτοια χρήσιμη δομημένη πληροφορία από γραμμικό κείμενο είναι γνωστή ως συντακτική ανάλυση ή parsing.

## 1.1 Ορισμός συντακτικού

Προκειμένου να κατασκευάσουμε έναν συντακτικό αναλυτή (ή parser) για μία γλώσσα, ή ακόμα και για να ορίσουμε τυπικά ποια είδη συμβολοσειρών έχουν νόημα σε αυτήν, πρέπει να έχουμε έναν τρόπο να για να εκφράσουμε και να κατανοήσουμε τη συντακτική δομή της. Για το σκοπό αυτό συνήθως χρησιμοποιούμε κάποια *γραμματική*, που είναι μία συμπαγής αναπαράσταση της δομής μίας γλωσσας, εκφρασμένη σε κάποια άλλη (ιδανικά μικρή και απλή) γλώσσα. Η γλώσσα της οποίας τη δομή προσπαθούμε να αναπαραστήσουμε είναι η *γλώσσα αντικείμενο*, ενώ η γλώσσα στην οποία *εκφράζεται* η συντακτική δομή ονομάζεται *γραμματική ορισμού γλώσσας*.

Ο πιο συνηθισμένος τύπος γραμματικής σήμερα, είναι οι *γραμμτικές χωρίς συμφραζόμενα ( context-free grammars - CFG)*, εκφρασμένες σε Backus-Naur Form (BNF). Μία γραμματική χωρίς συμφραζόμενα ουσιαστικά εκφράζει ένα σύνολο αμοιβαίως αναδρομικών κανόνων, οι οποίες περιγράφουν πώς μπορούν να γραφτούν οι συμβολοσειρές που περιγράφονται στη γλώσσα. Κάθε κανόνας ή *παραγωγή* σε μία CFG καθορίζει έναν τρόπο με τον οποίο μία συντακτική μεταβλητή ή *μη τερματικό* μπορεί να αντικατασταθεί σε μία συμβολοσειρά. Ένα μη τερματικό μπορεί να αντικατασταθεί σε μία συμβολοσειρά που μπορεί να περιέχει άλλα μη τερματικά, τα οποία θα αντικατασταθούν με τη σειρά τους, ώσπου να μην υπάρχουν άλλα. Επειδή υπάρχουν πολλοί τρόποι για να αντικατασταθεί ένα μη τερματικό, η γραμματική μπορεί να εκφράσει ένα άπειρο σύνολο καλώς ορισμένων συμβολοσειρών.

Η συντακτική ανάλυση μιας συμβολοσειράς, της οποίας η σύνταξη περιγράφεται από μία CFG, περιλαμβάνει την ανάποδη διαδικασία: να καθοριστεί από μία πλήρως ανεπτυγμένη συμβολοσειρά, η οποία περιέχει μόνο ατομικούς χαρακτήρες ή *τερματικά*, ποια ακολουθία (ή ακολουθίες βημάτων) αντικατάστασης, αν υπάρχουν, οδηγούν στην παραγωγή της συμβολοσειράς. Αυτή η εργασία περιπλέκεται καθώς οι CFGs συνήθως περιέχουν αμφισημίες: σε *τοπικό επίπεδο*, όπου η σωστή ερμηνεία ενός τμήματος της συμβολοσειράς μπορεί να καθοριστεί μόνο από τα συμφραζόμενα, και σε *καθολικό επίπεδο*, όπου ολόκληρη η συμβολοσειρά μπορεί να έχει πολλαπλές έγκυρες συντακτικές ερμηνείες.

## 1.2 Parsing Expression Grammars

Η θεωρία και η πράξη της συντακτικής ανάλυσης βασίζεται σε *παραγωγικά ( generative)* συστήματα, όπως οι κανονικές εκφράσεις και οι γραμματικές χωρίς συμφραζόμενα, στα οποία η γλώσσα ορίζεται τυπικά μέσα από κανόνες οι οποίοι όταν εφαρμοστούν αναδρομικά παράγουν συμβολοσειρές της γλώσσας. Αντίθετα, σε ένα *αναγνωριστικό σύστημα ( recognition-based system)* η γλώσσα ορίζεται μέσα από κανόνες ή κατηγορήματα που αποφασίζουν εάν η δοθείσα συμβολοσειρά ανήκει στη γλώσσα [Ford04]. Οι απλές γλώσσες μπορούν να εκφραστούν εξίσου εύκολα και στα δύο συστήματα. Για παράδειγμα, το $\{s \in \mathbf{a}^* | s = (\mathbf{aa})^n\}$ είναι ένας παραγωγικός ορισμός μιας γλώσσας με ένα μόνο γράμμα στο λεξιλόγιό της, της οποίας οι συμβολοσειρές κατασκευάζονται συνενώνοντας ζεύγη από **a**. Από την άλλη, το $\{s \in \mathbf{a}^* | (|s| mod 2 = 0)\}$, είναι ένας αναγνωριστικός ορισμός, όπου μία συμβολοσειρά από **a**'s γίνεται αποδεκτή μόνο αν το μήκος της είναι άρτιο.

Αν και το παραγωγικό μοντέλο χρησιμοποιείται ευρύτατα στη θεωρία γλωσσών, οι πιο πολλές πρακτικές γλωσσικές εφαρμογές περιλαμβάνουν την αναγνώριση και τη δομική ανάλυση συμβολοσειρών. Εμείς θα χρησιμοποιήσουμε το αναγνωριστικό μοντέλο για τη σύνταξη της γλώσσας, η οποία θα περιγράφεται από τις *Parsing Expression Grammars (PEGs)*. Αυτές μοιάζουν με τις γραμματικές χωρίς συμφραζόμενα με την προσθήκη χαρακτηριστικών από τις κανονικές εκφράσεις, όμοια με την Extended BNF σημειογραφία.

## 1.3 Packrat Parsing

Με βάση τις Parsing Expression Grammars και το αναγνωριστικό σχήμα, ο απλούστερος τρόπος να αναλυθεί (συντακτικά) μία συμβολοσειρά είναι μέσω ενός αναδρομικού καθοδικού συντακτικού αναλυτή, με δυνατότητα για οπισθαναχώρηση ( backtracking). Ωστόσο, σε πολλές PEGs ένας συνήθης καθοδικός αναλυτής μπορεί να κάνει εκθετικό χρόνο για να αναγνωρίσει μία συμβολοσειρά στην είσοδο. Και αυτό διότι η οπισθαναχώρηση μπορεί να οδηγήσει σε πλεονάζοντες υπολογισμούς ενδιάμεσων αποτελεσμάτων. Αν, όμως, χρησιμοποιηθεί *υπομνηματισμός ( memoisation)* για να αποφευχθούν τέτοιοι πλεονασμοί, μία PEG μπορεί να αναλυθεί σε γραμμικό χρόνο με το μήκος της εισόδου. Ένας τέτοιος συντακτικός αναλυτής ονομάζεται *Packrat Parser*.

## 1.4 Αυτόματη παραγωγή των Packrat Parsers

Αν και το packrat parsing είναι εύκολο να υλοποιηθεί με το χέρι, θα ήταν ακόμη καλύτερο να μπορούσαμε να κατασκευάσουμε τέτοιους συντακτικούς αναλυτές με αυτόματο τρόπο, όμοια με το YACC στον κόσμο της C. Ένας *γεννήτορας συντακτικών αναλυτών (parser generator)* παίρνει ως είσοδο μία τυπική περιγραφή της γραμματικής και "γεννάει" τον αντίστοιχο συντακτικό αναλυτή. Η περιγραφή που δέχεται ο δικός μας γεννήτορας βασίζεται στη σημειογραφία των PEGs και η υλοποίησή του είναι σε C++. Τέλος, παρέχει και τη δυνατότητα να επιστραφεί το συντακτικό δέντρο μετά την ανάλυση, ενώ επεκτείνεται εύκολα ώστε να κρατήσει και άλλες σημασιολογικές τιμές. Οι βασικές ιδέες για την υλοποίησή του πηγάζουν από πρότερη δουλειά που έχει γίνει σε Java [Fowl09] [Inst06].

## 1.5 Packrat Parsing με Ελαστικό Κυλιόμενο Παράθυρο

Αφού υλοποιήσουμε έναν packrat parser generator σε C++, η βασική συνεισφορά που θέλουμε να κάνουμε στα πλαίσια της διπλωματικής, είναι να εξετάσουμε πώς ένας αλγόριθμος packrat parser μπορεί να βελτιωθεί, τόσο σε θέμα χρόνου εκτέλεσης, όσο και στο αποτύπωμα της μνήμης που αφήνει.

Το βασικό μειονέκτημα του αλγορίθμου είναι το μεγάλο αποτύπωμα που αφήνει στη μνήμη, εξαιτίας του πίνακα ενδιάμεσου αποτελεσμάτων που χρησιμοποιεί. Αυτός ο πίνακας μπορεί να μειωθεί σε μέγεθος, κρατώντας ενδιάμεσα αποτελέσματα που είναι κοντά στο "μέτωπο" της συντακτικής ανάλυσης. Ουσιαστικά, είναι σαν να έχουμε ένα παράθυρο που να προχωράει μαζί με το μέτωπο της

συντακτικής ανάλυσης. Μάλιστα, αυτό θα είναι ”ελαστικό”, δηλαδή θα μας επιτρέπει να επιλέγουμε εν μέρει και ποια ενδιάμεσα αποτελέσματα θα κρατάμε. Η μέθοδος που αξιοποιεί ένα τέτοιο ελαστικό παράθυρο ονομάζεται εν συντομία elastic packrat parsing [Kura15].

## 1.6 Παράλληλο Packrat Parsing

Ένας τρόπος να βελτιώσουμε το χρόνο εκτέλεσης του αλγορίθμου είναι να τον παραλληλοποιήσουμε. Ειδικότερα, θα εξετάσουμε πώς θα μπορούσε να παραλληλοποιηθεί ώστε να τρέχει αποδοτικότερα σε ένα πολυπύρηνο σύστημα με αρχιτεκτονική κοινής μνήμης. Η πρώτη προσέγγιση που έχει δοκιμαστεί [Fowl09] είναι να χωρίζεται η είσοδος σε κομμάτια και να αναλαμβάνει ένα νήμα να υπολογίσει τα κελιά του συντακτικού αναλυτή που αφορούν το συγκεκριμένο κομμάτι.

Η πρώτη δική μας προσέγγιση είναι αντί για υπομνηματισμό, να χρησιμοποιήσουμε δυναμικό προγραμματισμό, τον οποίο έπειτα δοκιμάσαμε να παραλληλοποιήσουμε. Ακολούθως, εφαρμόσαμε παραλληλοποίηση στην πράξη της *διατεταγμένης επιλογής (ordered choice)* των PEGs, παραμετροποιώντας τον αριθμό των νημάτων που θα δημιουργούνται.

## 1.7 Η δομή της εργασίας

Το Κεφάλαιο 1 ήταν μία σύντομη εισαγωγή. Το Κεφάλαιο 2 εισάγει τις βασικές έννοιες για τις Parsing Expression Grammars, πάνω στις οποίες θα στηριχθούμε. Το Κεφάλαιο 3 περιγράφει τον κλασικό αλγόριθμο packrat και τον τρόπο υλοποίησής του σε μία αντικειμενοστραφή γλώσσα όπως η C++. Το Κεφάλαιο 4 αναλύει τον τρόπο με τον οποίο μπορούμε να κατασκευάσουμε έναν γεννήτορα αναλυτών packrat. Το Κεφάλαιο 5 παραθέτει τη βελτίωση του κλασικού αλγορίθμου, χρησιμοποιώντας το ελαστικό κυλιόμενο παράθυρο. Το Κεφάλαιο 6 εξετάζει τις παράλληλες εκδοχές του κλασικού packrat. Το Κεφάλαιο 7 παρουσιάζει τα αποτελέσματα συγκρίνοντας τους αλγορίθμους που υλοποιούμε με τον κλασικό αλγόριθμο. Το Κεφάλαιο 8 κλείνει τη διπλωματική με τα συμπεράσματα που προέκυψαν.

## 1.8 Κώδικας

Ο κώδικας που υλοποιήθηκε στα πλαίσια της διπλωματικής μπορεί να βρεθεί στο:

`https://github.com/blackgeorge-boom/parallel_packrat`

# Κεφάλαιο 2

# Parsing Expression Grammars

Οι δύο πιο συνηθισμένες μέθοδοι για να περιγραφεί η σύνταξη μίας γλώσσας σήμερα είναι οι κανονικές εκφράσεις και οι γραμματικές χωρίς συμφραζόμενα. Αυτοί οι φορμαλισμοί, ωστόσο, δεν είναι σε καμία περίπτωση ο μοναδικός τρόπος ορισμού της συντακτικής δομής μίας γλώσσας. Ένα ακόμη χρήσιμο πρότυπο περιγραφής της σύνταξης είναι οι *Parsing Expression Grammars (PEGs)* [Ford04], οι οποίες μοιάζουν με τις γραμματικές χωρίς συμφραζόμενα, αλλά έχουν και ορισμένες θεμελιώδεις διαφορές. Δαισθητικά, μια γραμματική χωρίς συμφραζόμενα μας περιγράφει το πώς *κατασκευάζεται* μία συμβολοσειρά που ανήκει σε κάποια γλώσσα, ενώ οι PEGs το πώς *αναλύεται* η συμβολοσειρά ώστε να προκύψει δομική πληροφορία για αυτή.

Δεδομένου ότι η περιγραφή των γλωσσών (συνήθως) γράφεται από ανθρώπους και διαβάζεται από μηχανές, οι Parsing Expression Grammars αποτελούν διαισθητικά ένα πιο κατάλληλο εργαλείο προσδιορισμού από τις γραμματικές χωρίς συμφραζόμενα. Ο σχεδιαστής της γραμματικής είναι ευκολότερο να σκέφτεται πώς αναλύεται μία δοσμένη συμβολοσειρά στα συστατικά της, παρά πώς θα γεννηθεί (generated) η συμβολοσειρά μέσα από τους κανόνες της γραμματικής.

Πολλά συντακτικά ιδιώματα των σύγχρονων γλωσσών προγραμματισμού εκφράζονται ευκολότερα και πιο ”φυσικά” σε Parsing Expression Grammars. Επιπρόσθετα, οι PEGs μπορούν να αναλυθούν συντακτικά σε γραμμικό χρόνο, χρησιμοποιώντας το Packrat Parsing που περιγράφεται σε επόμενη ενότητα, την ώρα που μόνο μία συγκεκριμένη υποκλάση των γλωσσών χωρίς συμφραζόμενα μπορεί να αναλυθεί σε γραμμμικό χρόνο.

## 2.1 Ορισμός των Parsing Expression Grammars

Όπως με τις Context Free, οι Parsing Expression Grammars χρησιμοποιούν τόσο τερματικά όσο και μη τερματικά σύμβολα και αποτελούνται από ένα σύνολο κανόνων που παρέχουν ορισμούς για τα μη τερματικά. Κάθε κανόνας μπορεί να αναφέρεται σε άλλους κανόνες της γραμματικής αναδρομικά. Θα ακολουθήσουμε το συμβολισμό ‘$n \leftarrow e$’, όπου το $n$ είναι ένα μη τερματικό και το $e$ είναι μία έκφραση που θα οριστεί ακολούθως. Η χρήση του αριστερού βέλους αντί του δεξιού εκφράζει την διασθητική διαφορά στην ”ροή της πληροφορίας” που διακρίνει τις PEGs από τις CFGs. Ενώ, οι κανόνες των CFGs εκφράζουν ”παραγωγές” από μη τερματικά στις αντίστοιχες εκφράσεις τους, οι κανόνες των PEGs αναπαριστούν ”αφαιρέσεις” από τις εκφράσεις στους αντίστοιχους κανόνες. Επιπλέον, οι παραγωγές εκφρασμένες σε CFGs αναπαριστούν πράξεις σε ολόκληρες συμβολοσειρές, ενώ οι αφαιρέσεις σε μία PEG αναπαριστούν πράξεις σε προθέματα της συμβολοσειράς στην είσοδο.

Σύμφωνα με τον συμβολισμό των Parsing Expression Grammars, οι εκφράσεις σχηματίζονται ως εξής:

- **Κενή συμβολοσειρά:** ‘()’ είναι μία έκφραση που υποδηλώνει την άδεια συμβολοσειρά. Η ερμηνεία της είναι ”Μην προσπαθήσεις να διαβάσεις τίποτα: απλά επίστρεψε επιτυχώς χωρίς να καταναλώσεις τίποτα από την είσοδο.”

- **Τερματικό:** Αν το $\alpha$ είναι ένα τερματικό σύμβολο (π.χ. ένας χαρακτήρας μόνος του), τότε το ‘$\alpha$’ είναι μία έκφραση της οποίας η ερμηνεία είναι: ”Αν το επόμενο τερματικό στην είσοδο είναι $\alpha$

τότε κατανάλωσε ένα τερματικό και επίστρεψε επιτυχώς. αλλιώς, απότυχε και μην καταναλώσεις τίποτα."

- **Μη Τερματικό:** Αν το $A$ είναι ένα μη τερματικό σύμβολο , τότε το '$A$' είναι μία έκφραση της οποίας η ερμηνεία είναι: "Προσπάθησε να διαβάσεις την είσοδο με βάση τον κανόνα που αντιστοιχεί στο $A$ και επίστρεψε επιτυχώς ή απότυχε αντίστοιχα."

- **Ακολουθία:** Αν $e_1, e_2, \ldots, e_n$ είναι εκφράσεις, τότε το '$(e_1 e_2 \ldots e_n)$' είναι μία έκφραση της οποίας η ερμηνεία είναι: "Πρώτα προσπάθησε να διαβάσεις μία συμβολοσειρά ώστε να επιτύχει η $e_1$. Αν η $e_1$ επιτύχει, τότε προσπάθησε να διαβάσεις μία συμβολοσειρά ώστε να επιτύχει η $e_2$, ξεκινώντας από το σημείο της εισόδου που δεν κατανάλωσε η $e_1$. Αν η $e_2$ επιτύχει τότε συνέχισε με την $e_3$ κ.ό.κ μέχρι την $e_n$. Αν και οι $n$ εκφράσεις αναγνωριστούν επιτυχώς διαδοχικά, τότε επίστρεψε επιτυχώς και κατανάλωσε όλα τα αντίστοιχα κομμάτια της εισόδου. Αν οποιαδήποτε υποέκφραση αποτύχει, τότε όλη η ακολουθία αποτυγχάνει συνολικά χωρίς να καταναλώσει τίποτα από την είσοδο."

- **Διατεταγμένη Επιλογή:** Αν $e_1, e_2, \ldots, e_n$ είναι εκφράσεις, τότε το '$(e_1/e_2/\ldots/e_n)$' είναι μία έκφραση της οποίας η ερμηνεία είναι η εξής: "Πρώτα προσπάθησε να διαβάσεις μία συμβολοσειρά ώστε να επιτύχει η $e_1$. Αν αυτό πετύχει τότε η επιλογή επιστρέφει επιτυχώς καταναλώνοντας το αντίστοιχο κομμάτι της εισόδου. Αλλιώς, προσπάθησε με την $e_2$ και την αρχική είσοδο, μετά με την $e_3$, κ.ό.κ, μέχρις ότου να φτάσεις στην $e_n$, σταματώντας στην πρώτη εναλλακτική που θα επιτύχει. Αν καμία από τις $n$ εναλλακτικές δεν πετύχουν, τότε απότυχε χωρίς να καταναλώσεις τίποτα από την είσοδο." Η έκφραση '$(e_1/e_2)$' μπορεί να διαβαστεί ως "$e_1$ ή αλλιώς $e_2$". Χρησιμοποιήσαμε το σύμβολο της καθέτου ('/') αντί της μπάρας ('|'), που χρησιμοποιείται στις γραμματικές χωρίς συμφραζόμενα, για να τονίσουμε την ουσιώδη διαφορά ότι η επιλογή στις PEGs δεν είναι συμμετρική, αλλά βασίζεται σε προτεραιότητα.

- **Άπληστη Επανάληψη:** Αν το '$e$' είναι μία έκφραση, τότε το '$(e^*)$' είναι μία έκφραση της οποίας η ερμηνεία είναι η εξής: "Εφάρμοσε την έκφραση $e$ επανειλλημένα στην είσοδο, καταναλώνοντας την είσοδο προοδευτικά με κάθε επανάληψη όσο συνεχίζει να επιτυγχάνει. Με την πρώτη αποτυχία, κατανάλωσε όλη την είσοδο που είχε αναγνωριστεί μέχρι τότε και επίστρεψε επιτυχώς. Αν το $e$ δεν πέτυχε ούτε μία φορά, τότε επίστρεψε όπως και να 'χει επιτυχώς χωρίς να καταναλώσεις τίποτα."

- **Άπληστη Θετική Επανάληψη:** Αν το '$e$' είναι μία έκφραση, τότε το '$(e^+)$' είναι μία έκφραση της οποίας η ερμηνεία είναι η εξής: "Εφάρμοσε την έκφραση $e$ επανειλλημένα στην είσοδο, και επίστρεψε επιτυχώς καταναλώνοντας όλη την είσοδο που είχε αναγνωριστεί όσο τουλάχιστον ένα στιγμιότυπο της $e$ πετύχαινε. Με την πρώτη αποτυχία, κατανάλωσε όλη την είσοδο που είχε αναγνωριστεί μέχρι τότε και επίστρεψε επιτυχώς. Αν το $e$ δεν πέτυχε ούτε μία φορά, τότε απότυχε χωρίς να καταναλώσεις τίποτα."

- **Προαιρετικό:** Αν το '$e$' είναι μία έκφραση, τότε το '$(e?)$' είναι μία έκφραση της οποίας η ερμηνεία είναι η εξής: "Προσπάθησε να εφαρμόσεις την έκφραση $e$ στην είσοδο. Αν πετύχεις, τότε κατανάλωσε το αναγνωρισμένο κείμενο και επίστρεψε επιτυχώς. Αν το $e$ αποτύχει, τότε επίστρεψε επιτυχώς όπως και να 'χει αλλά μην καταναλώσεις τίποτα από την είσοδο."

- **Ακουλουθείται-Από Κατηγόρημα:** Αν το '$e$' είναι μία έκφραση, τότε το '$\&(e)$' είναι μία έκφραση της οποίας η ερμηνεία είναι η εξής: "Προσπάθησε να εφαρμόσεις την έκφραση $e$ στην είσοδο. Αν πετύχεις με το '$e$', τότε επίστρεψε επιτυχώς με το '$\&(e)$' αλλά μην καταναλώσεις τίποτα από την είσοδο (δηλαδή επίστρεψε στην θέση της εισόδου που ήσουν πριν εφαρμοστεί το $e$). Αν το $e$ αποτύχει, τότε απότυχε".

- **Δεν-Ακολουθείται-Από Κατηγόρημα:** Αν το '$e$' είναι μία έκφραση, τότε το '$!(e)$' είναι μία έκφραση της οποίας η ερμηνεία είναι η εξής: "Προσπάθησε να εφαρμόσεις την έκφραση $e$ στην

είσοδο. Αν αποτύχεις με το '$e$', τότε πέτυχε με το '$!(e)$' αλλά μην καταναλώσεις τίποτα από την είσοδο. Αν το $e$ επιτύχει, τότε απότυχε αλλά μην καταναλώσεις τίποτα από την είσοδο."

Παρά την ποικιλία των παρεχόμενων εκφράσεων, όλες αυτές μπορούν να συρρικνωθούν σε έναν μικρό "πυρήνα" από στοιχειώδεις εκφράσεις.

## 2.2 Ένα παράδειγμα μιας PEG

Το Σχήμα 2.1 παρουσιάζει μία γραμματική χωρίς συμφραζόμενα για μία απλή αριθμητική γλώσσα.

$$\begin{array}{lcl} \mathrm{E} & \rightarrow & \mathrm{N} \\ & | & \text{'(' E '+' E ')'} \\ & | & \text{'(' E '−' E ')'} \\ \mathrm{N} & \rightarrow & \mathrm{D} \\ & | & \mathrm{D\ N} \\ \mathrm{D} & \rightarrow & \text{'0'} \mid \ldots \mid \text{'9'} \end{array}$$

**Σχήμα 2.1:** Μία CFG για μία απλή αριθμητική γλώσσα

Το Σχήμα 2.2 παρουσιάζει την αντίστοιχη PEG.

$$\begin{array}{lcl} \mathrm{E} & \leftarrow & \mathrm{N} \\ & / & \text{'(' E '+' E ')'} \\ & / & \text{'(' E '−' E ')'} \\ \mathrm{N} & \leftarrow & \mathrm{D\ N} \\ & / & \mathrm{D} \\ \mathrm{D} & \leftarrow & \text{'0'} \mid \ldots \mid \text{'9'} \end{array}$$

**Σχήμα 2.2:** Μία PEG για μία απλή αριθμητική γλώσσα

Πρακτικά, η περιγραφή της γραμματικής μας ορίζει νοηματικά τί θα έκανε ένας καθοδικός συντακτικός αναλυτής για να αναγνωρίσει μία συμβολοσειρά στην είσοδο.

Το Σχήμα 2.3 απεικονίζει πώς η συμβολοσειρά '$(12-3)$' μπορεί να αναγνωριστεί σύμφωνα με τη γραμματική στο Σχήμα 2.2.

| Position | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Expressions | E 1 (1–6) | | | | | |
| | | E 2 (2–3) | | | E 5 (5) | |
| Numbers | | N 2 (2–3) | | | | |
| | | | N 3 (3) | | N 5 (5) | |
| Digits | | D 2 | D 3 | | D 5 | |
| Input String | **(** | **1** | **2** | **−** | **3** | **)** |

**Σχήμα 2.3:** Αναγνωρίζοντας το '$(12-3)$' με βάση τη γραμματική στο Σχήμα 2.2 [Ford02a]

Ξεκινάμε προσπαθώντας να διαβάσουμε μία έκφραση ($E$), από την αρχή της συμβολοσειράς. Με βάση τον ορισμό της γραμματικής, για να αναγνωρίσει το μη τερματικό $E$, ο συντακτικός αναλυτής θα προσπαθούσε πρώτα να αναγνωρίσει την έκφραση $N$ που είναι η πρώτη εναλλακτική, οπότε προσπαθούμε και για τις δύο εναλλακτικές του $N$. Ωστόσο αποτυγχάνουμε καθώς στην είσοδο ο πρώτος χαρακτήρας είναι το '(' και όχι κάποιο ψηφίο. Ακολούθως, πηγαίνουμε στη δεύτερη εναλλακτική για το $E$, τον κανόνα της πρόσθεσης εκφράσεων. Αυτός αναγνωρίζεται επιτυχώς με την αριστερή παρένθεση και μας καθοδηγεί στο να διαβάσουμε μία (υπο-)έκφραση ξεκινώντας στη θέση 2 της εισόδου. Για να διαβάσουμε αυτή την υποέκφραση επιχειρούμε ξανά την $N$ εναλλακτική. Πλέον, η πρώτη εναλλακτική του $N$ επιτυγχάνει, διαβάζοντας ένα ψηφίο στη θέση 2, και αναδρομικά ελέγχει για ένα ψηφίο στη θέση 3. Η πρώτη εναλλακτική του $N$ (δηλαδή η $DN$) αποτυγχάνει γιατί το ψηφίο στη θέση 3 δεν ακολουθείται από άλλα ψηφία, ωστόσο η δεύτερη εναλλακτική ($D$) επιτυγχάνει να αναγνωρίσει το ψηφίο, οπότε παράγει το αποτέλεσμα $N_3$ στο σχήμα. Η επιτυχημένη προσπάθεια καθιστά επιτυχημένη την αναγνώριση του $N$ στη θέση 2, οπότε το $N$ πλέον έχει ως αποτέλεσμα δύο κολλητούς χαρακτήρες, δηλαδή το $N_2$. Αυτό οδηγεί στην έκφραση $E_2$ στο σχήμα. Επιστρέφοντας στο διάβασμα μίας έκφρασης στη θέση 1, η δεύτερη εναλλακτική αποτυγχάνει διότι η έκφραση $E_2$ ακολουθείται από ένα '$-$' αντί από ένα '$+$'. Όμως, αν χρησιμοποιήσουμε την τρίτη εναλλακτική του $E$, αυτή επιτυγχάνει αφού αναγνωρίζει την αριστερή παρένθεση και την $E_2$ όπως και πριν, ενώ τώρα πετυχαίνει το '$-$', το ψηφίο $E_5$ στη θέση 5, και τη δεξιά παρένθεση. Επομένως, η έκφραση $E_1$ γεννιέται, αναγνωρίζοντας όλη τη συμβολοσειρά εισόδου.

## 2.3 Άπληστη και Μη Ντετερμινιστική επανάληψη

Ο κανόνας για το μη τερματικό $N$ δείχνει μία από τις πιο σημαντικές διαφορές μεταξύ των PEGs και των CFG γραμματικών. Στη γραμματική του Σχήματος 2.1 η σειρά των δύο εναλλακτικών για το μη τερματικό δεν έχει σημασία, επειδή η επιλογή είναι μη ντετερμινιστική και προσανατολισμένη στο να γράφονται συμβολοσειρές και όχι να διαβάζονται. Στην PEG του Σχήματος 2.2, η σειρά με την οποία εξετάζονται οι εναλλακτικές έχει σημασία: αν επιλέγαμε τη συντομότερη εναλλακτική πρώτα ($D$), τότε η μακρύτερη εναλλακτική δεν θα χρησιμοποιούνταν ποτέ. Το αποτέλεσμα θα ήταν μία γραμματική που δεν θα μπορούσε να αναγνωρίσει τη συμβολοσειρά '$(12-3)$' διότι η ανάλυση του μη τερματικού στη θέση 2 θα κατανάλωνε μόνο το '1' στον αριθμό 12, αφήνοντας το '2' να το αναγνωρίσουν άλλοι κανόνες ξεκινώντας πάλι από τη θέση 1.

Εξαιτίας αυτής της διαφοράς, οι δομές με επανάληψη σε μία PEG είναι εκ των πραγμάτων περισσότερο "άπληστες" παρά μη ντετερμινιστικές: μια επαναληπτική δομή πάντα καταναλώνει όσο περισσότερο κείμενο μπορεί, ανεξάρτητα από τα συμφραζόμενα στα οποία βρίσκεται. Αν στο παράδειγμά μας θέλαμε ξεφορτωθούμε το μη τερματικό $N$, θα μπορούσαμε να αντικαταστήσουμε την πρώτη εναλλακτική του $E$ με το '$D+$'. Το αποτέλεσμα θα ήταν ακριβώς το ίδιο, γι' αυτό και ο τελεστής $+$ ονομάζεται "άπληστη θετική επανάληψη".

## 2.4 Συντακτική ανάλυση ολόκληρων συμβολοσειρών

Αν η γραμματική στο 2.2 χρησιμοποιηθεί για να διαβάσει τη συμβολοσειρά '$(12-3)XYZ$', με αρχικό σύμβολο το $E$, τότε το αποτέλεσμα θα είναι "επιτυχία", ωστόσο μόνο το '$(12-3)$' θα έχει καταναλωθεί, αφήνοντας το '$XYZ$' ως υπόλοιπο. Όταν η πρόθεσή μας είναι να αναλύσουμε συντακτικά μία συμβολοσειρά, συνήθως θέλουμε να το κάνουμε μέχρι τέλους, και όχι μόνο σε ένα τμήμα της. Ευτυχώς, η συμπεριφορά αυτή είναι εύκολο να υλοποιηθεί προσθέτοντας ένα νέο αρχικό σύμβολο $S$ όπως φαίνεται στο Σχήμα 2.4.

Το αρχικό σύμβολο ψάχνει για μία έκφραση $E$, και μετά χρησιμοποιεί τον τελεστή "δεν-ακολουθείται-από" για να διασφαλίσει ότι τίποτα δεν έπεται μετά την αναγνωρισμένη έκφραση στην είσοδο. Αν υπάρχει επιπλέον κείμενο που ακολουθεί την έκφραση, τότε το $C$ θα επιτύχει, κάνοντας το $S$ να αποτύχει. Αλλιώς, το $C$ αποτυγχάνει και το $S$ επιτυγχάνει.

Το $C$ σε αυτό το παράδειγμα είναι ένα μη τερματικό που αναπαριστά την *κλάση χαρακτήρων*.

$$
\begin{array}{lll}
\text{S} & \leftarrow & \text{E !(C)} \\
\text{E} & \leftarrow & \ldots \\
\ldots & & \\
\text{C} & \leftarrow & \text{any single character}
\end{array}
$$

**Σχήμα 2.4:** Επέκταση της γραμματικής ώστε να εξετάζει όλη την είσοδο μέχρι τέλους

## 2.5 Αριστερή Αναδρομή

Σε μία γραμματική χωρίς συμφραζόμενα, τόσο η *αριστερή* όσο και η *δεξιά* αναδρομή επιτρέπονται. Ένα αριστερά αναδρομικό μη τερματικό έχει την ιδιότητα, αφού αναπτυχθεί μία ή περισσότερες φορές, να δίνει συμβολοσειρές η οποίες ξεκινούν με αυτό το μη τερματικό. Ομοίως, ένα δεξιά αναδρομικό μη τερματικό έχει την ιδιότητα να αναπτύσσεται σε συμβολοσειρές που *τελειώνουν* με αυτό. Για παράδειγμα, είναι σύνηθες να εκφράζουμε τη σύνταξη αριστερά προσεταιριστικών τελεστών στα πλαίσια αριστερά αναδρομικών CFGs, και δεξιά προσεταιριστικούς τελεστές στα πλαίσια δεξιά αναδρομικών CFGs:

$$
\begin{array}{lll}
\text{Unary} & \rightarrow & \text{`+' Unary} \\
 & | & \text{`-' Unary} \\
 & | & \text{Primary} \\
\text{Additive} & \rightarrow & \text{Additive `+' Unary} \\
 & | & \text{Additive `-' Unary} \\
 & | & \text{Unary}
\end{array}
$$

**Σχήμα 2.5:** Αριστερά και δεξιά προσεταιριστικοί τελεστές σε μία CFG

Σε μία CFG ο δεξιά αναδρομικός ορισμός του $Unary$ υλοποιεί τους δεξιά προσεταιριστικούς μοναδιαίους τελεστές '$+$' και '$-$'. Ομοίως, το $Additive$ υλοποιεί τους αριστερά προσεταιριστικούς τελεστές '$+$' και '$-$'. Σε μία PEG, ενώ η δεξιά αναδρομή δουλεύει παρόμοια με τις CFGs, η αριστερή αναδρομή είναι εξ ορισμού λαθεμένη, καθώς η ερμηνεία της οδηγεί σε μία εκφυλισμένη αυτοαναφορά. Για παράδειγμα, αν πάμε να εφαρμόσουμε αυτολεξεί τον κανόνα $Additive$ στα πλαίσια μίας PEG, τότε η ερμηνεία θα ήταν η εξής: "Για να διαβάσεις μία $Additive$ έκφραση, πρώτα προσπάθησε να διαβάσεις μία $Additive$ έκφραση κ.ό.κ".

Σε μία CFG η αριστερή αναδρομή μπορεί να είναι βολική, ωστόσο δεν είναι απαραίτητη, καθώς οποιαδήποτε CFG που περιλαμβάνει αριστερή αναδρομή, μπορεί να γραφτεί σε μία ισοδύναμη CFG χωρίς αριστερή αναδρομή [Moor00]. Στις PEGs, συνήθως είναι πιο βολικό και ακριβές να χρησιμοποιούνται οι τελεστές '$*$' και '$+$', αντί της αριστερής ή της δεξιάς αναδρομής. Για παράδειγμα, η CFG του Σχήματος 2.5 μπορεί να γραφεί σε PEG ως:

$$
\begin{array}{lll}
\text{Unary} & \leftarrow & \text{(`+' / `-')* Primary} \\
\text{Additive} & \leftarrow & \text{Unary ((`+' / `-') Unary)*}
\end{array}
$$

**Σχήμα 2.6:** Ισοδύναμη PEG χωρίς αριστερή αναδρομή

Πάντως, αν είναι απαραίτητο, οι συντακτικοί αναλυτές των PEGs (τους οποίους θα περιγράψουμε παρακάτω), μπορούν να τροποποιηθούν ώστε να υποστηρίζουν αριστερή αναδρομή [Wart08].

# Κεφάλαιο 3

# Packrat Parsing

Το *Packrat Parsing* [Ford02a] είναι μία τεχνική για την υλοποίηση συντακτικών αναλυτών για Parsing Expression Grammars. Ένας συντακτικός αναλυτής packrat, ή αλλιώς packrat parser, προσφέρει την ισχύ και την ευελιξία ενός συντακτικού αναλυτή από πάνω προς τα κάτω με υπαναχώρηση (backtracking) και τη δυνατότητα ανάγνωσης άπειρων προπορευόμενων συμβόλων (infinite lookahead), αλλά εγγυάται γραμμικό χρόνο συντακτικής ανάλυσης. Οποιαδήποτε γλώσσα ορισμένη από μία LL($k$) ή LR($k$) γραμματική μπορεί να αναγνωριστεί από έναν packrat parser, όπως και άλλες γλώσσες οι οποίες δεν υποστηρίζονται από συμβατικούς γραμμικούς αλγορίθμους συντακτικής ανάλυσης.

Αυτή η επιπλέον ισχύς απλοποιεί: τη διαχείριση των συνηθισμένων συντακτικών ιδιωμάτων, όπως ο κανόνας της μακρύτερης αντιστοίχισης (longest-match rule), επιτρέπει τη χρήση συντακτικών και σημασιολογικών κατηγορημάτων για αποσαφήνιση (disambiguation), παρέχει καλύτερες ιδιότητες για σύνθεση γραμματικών (grammar composition), ενώ δίνει και τη δυνατότητα να ενσωματωθεί η λεκτική ανάλυση στη συντακτική. Παρόλα αυτά, το packrat parsing θυμίζει την απλότητα και την κομψότητα συντακτικών αναλυτών αναδρομικής κατάβασης.

## 3.1 Θεμέλια

Ο απλούστερος και διαισθητικά προφανής τρόπος να σχεδιάσουμε έναν συντακτικό αναλυτή είναι η από πάνω προς τα κάτω ανάλυση ή ανάλυση αναδρομικής κατάβασης ή καθοδική συντακτική ανάλυση. Σε αυτήν, τα στοιχεία της γλώσσας μεταφράζονται σχεδόν άμεσα σε ένα σύνολο από αμοιβαία αναδρομικές συναρτήσεις. Η καθοδική συντακτική ανάλυση μπορεί να θεωρηθεί ως το πρόβλημα της κατασκευής ενός συντακτικού δέντρου για μια συμβολοσειρά εισόδου, ξεκινώντας από τη ρίζα και δημιουργώντας τους κόμβους του συντακτικού δέντρου με πρωτοδιάταξη (preorder) [Aho06]. Ισοδύναμα, η καθοδική ανάλυση μπορεί να θεωρηθεί ως η εύρεση ενός αριστερότερου σχηματισμού παραγώγου για τη συμβολοσειρά εισόδου.

Οι καθοδικοί συντακτικοί αναλυτές μπορούν να διακριθούν σε δύο κατηγορίες. Οι *προβλέποντες (predictive) συντακτικοί αναλυτές* επιχειρούν να προβλέψουν ποιο στοιχείο της γλώσσας ακολουθεί, βλέποντας ορισμένα από τα προπορευόμενα σύμβολα στην είσοδο. Οι *συντακτικοί αναλυτές με οπισθαναχώρηση (backtracking)* παίρνουν αποφάσεις υποθετικά (speculatively) και δοκιμάζουν διαδοχικά διάφορες εναλλακτικές: αν μία αποτύχει, τότε ο αναλυτής οπισθαναχωρεί στη θέση της εισόδου που ήταν προτού δοκιμάσει την εναλλακτική, και μετά εξετάζει την επόμενη εναλλακτική.

Οι προβλέποντες συντακτικοί αναλυτές είναι γρήγοροι και εγγυώνται γραμμικό χρόνο στην ανάλυση (ως προς το μήκος της εισόδου), ενώ οι αναλυτές με οπισθαναχώρηση είναι πιο απλοί εννοιολογικά αλλά μπορεί να έχουν εκθετικό χρονο εκτέλεσης.

Το packrat parsing αποτελεί μία στρατηγική καθοδικής ανάλυσης που αξιοποιεί τα θετικά και από τις δύο προαναφερθείσες επιλογές. Αφενός προσφέρει απλότητα, κομψότητα και γενικότητα όπως ένας αναλυτής με οπισθαναχώρηση, αφετέρου εξοβελίζει τον εκθετικό χρόνο εκτέλεσης, αποθηκεύοντας τα ενδιάμεσα αποτελέσματα από τη συντακτική ανάλυση, ώστε κανένα αποτέλεσμα να μην υπολογιστεί παραπάνω από μία φορά.

Ένας packrat parser μπορεί εύκολα να κατασκευαστεί για οποιαδήποτε γλώσσα που περιγράφεται

$$
\begin{aligned}
Additive &\leftarrow Multitive\text{ ` + ' } Additive \mid Multitive \\
Multitive &\leftarrow Primary\text{ ` * ' } Multitive \mid Primary \\
Primary &\leftarrow \text{`(' } Additive \text{ `)'} \mid Decimal \\
Decimal &\leftarrow \text{`0'} \mid ... \mid \text{`9'}
\end{aligned}
\tag{3.1}
$$

**Σχήμα 3.1:** Μία PEG γραμματική για αριθμητικές εκφράσεις

από μία LL($k$) ή LR($k$) γραμματική, καθώς επίσης και για πολλές γλώσσες που απαιτούν infinite lookahead και δεν είναι, επομένως, LR. Επιπλέον, είναι πιο εύκολο να κατασκευαστεί από έναν LR αναλυτή (ανοδική ανάλυση), ακόμα και με το χέρι.

## 3.2 Λειτουργία ενός packrat parser

Είπαμε ότι το packrat parsing συνδυάζει τα πλεονεκτήματα της απλότητας του αναλυτή αναδρομικής κατάβασης με οπισθαναχώρηση και της ταχύτητας του προβλέποντα συντακτικού αναλυτή. Αυτό το πετυχαίνει κρατώντας τα ενδιάμεσα αποτελέσματα που υπολογίζει σε έναν *πίνακα υπομνηματισμού (memoisation table)*.

Ο πίνακας αυτός έχει σειρές που αντιστοιχούν σε ένα μη τερματικό της γραμματικής και στήλες που αντιστοιχούν σε μία συγκεκριμένη θέση στην είσοδο. Έτσι, το κελί $(i, j)$ περιέχει το αποτέλεσμα που θα πάρουμε αν προσπαθήσουμε με το μη τερματικό $i$ να αναγνωρίσουμε το κομμάτι της εισόδου που ξεκινάει από τη θέση $j$ και μετά. Αυτόν τον πίνακα, μπορούμε είτε να τον γεμίσουμε από πάνω προς τα κάτω, δηλαδή κάνοντας αναδρομική κατάβαση με υπομνηματισμό (memoisation), είτε από κάτω προς τα πάνω, στο πνεύμα του δυναμικού προγραμματισμού.

### 3.2.1 Περιγραφή του συντακτικού αναλυτή με δυναμικό προγραμματισμό

Στην εκδοχή του δυναμικού προγραμματισμού ξεκινάμε να γεμίζουμε τα κελιά από το δεξί άκρο της εισόδου προς τα αριστερά, και κινούμαστε από τα κατώτερα κελιά στα ανώτερα μέσα σε κάθε στήλη. Οποτεδήποτε συμπληρώνουμε ένα κελί, το αποτέλεσμά του αποθηκεύεται και μπορεί να χρησιμοποιηθεί έτοιμο στις κλήσεις άλλων κελιών που δεν έχουν υπολογιστεί ακόμα.

Έστω η γραμματική του Σχήματος 3.1.

Ο Πίνακας 3.1 παρουσιάζει έναν μερικώς συμπληρωμένο πίνακα για είσοδο τη συμβολοσειρά '$2 * (3 + 4)$'.

| column | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 |
|---|---|---|---|---|---|---|---|---|
| Additive | | | ↑ | (7,C7) | X | (4,C7) | X | X |
| Multitive | | | ⋮ | (3,C5) | X | (4,C7) | X | X |
| Primary | | ← ... | ? | (3,C5) | X | (4,C7) | X | X |
| Decimal | | | X | (3,C5) | X | (4,C7) | X | X |
| Input String | '2' | '*' | '(' | '3' | '+' | '4' | ')' | EOF |

Πίνακας 3.1: Ενδιάμεσα αποτελέσματα για την είσοδο $2 * (3 + 4)$

Κάθε στήλη $C_j$ αντιστοιχεί στο σημείο $j$ της εισόδου. Κάθε γραμμή ($Additive$, $Multitive$ κλπ) αντιστοιχεί στη συνάρτηση που αναλύει συντακτικά το μη τερματικό $i$. Για λόγους παρουσίασης, κάθε κελί παρουσιάζεται με δύο τιμές (η υλοποίηση θα είχε διαφορετικές). Η μία τιμή είναι η σημασιολογική τιμή που επιστρέφεται ως αποτέλεσμα της συντακτικής ανάλυσης. Η άλλη είναι η στήλη στην οποία θα πάει μετά η ανάλυση, μόλις καταναλώσει μέρος της εισόδου σε εκείνο το κελί.

Για παράδειγμα, στη στήλη C4, στη γραμμή Additive, η ερμηνεία είναι η εξής: Αν η συνάρτηση που αντιστοιχεί στο Additive, ξεκινήσει την συντακτική ανάλυση από τη θέση 4 (χαρακτήρας ‘3’), θα καταναλώσει 3 χαρακτήρες (την παράσταση ‘$3+4$’) και θα φτάσει μέχρι το σημείο 7 της εισόδου. Η σημασιολογική τιμή που θα επιστραφεί είναι το αποτέλεσμα της έκφρασης, δηλαδή το 7.

Το επόμενο κελί που θα πρέπει να υπολογιστεί είναι το Primary στη στήλη C3. Για να δείξουμε ένα παράδειγμα του πώς επαναχρησιμοποιούνται τα αποτελέσματα, εστιάζουμε σε αυτό το κελί.

Ο κανόνας για το Primary έχει δύο εναλλακτικές: μία έκφραση για το Additive μέσα σε παρενθέσεις ή ένα Decimal. Αν προσπαθήσουμε τις εναλλακτικές με τη σειρά που δίνονται στη γραμματική, το Primary πρώτα θα ελέγξει για Additive ανάμεσα σε παρενθέσεις. Για να γίνει αυτό, πρώτα αντιστοιχίζει την αριστερή παρένθεση στη στήλη C3, το οποίο πετυχαίνει, και επιστρέφει την υπόλοιπη συμβολοσειρά η οποία ξεκινάει στη στήλη C4, δηλαδή την ‘$3+4)$’. Στην έκδοση του αναλυτή αναδρομικής κατάβασης με οπισθαναχώρηση, το Primary θα έπρεπε να καλέσει αναδρομικά τη συνάρτηση Additive στην εναπομείνασα συμβολοσειρά. Ωστόσο, επειδή έχουμε αποθηκεύσει τα ενδιάμεσα αποτελέσματα στον πίνακα, μπορούμε απλά να κοιτάξουμε το αποτέλεσμα της κλήσης Additive στη στήλη C4, που είναι $(7, C7)$.

Οπότε, η σημασιολογική τιμή είναι το 7 και η νέα θέση που εξετάζουμε στη συμβολοσειρά ξεκινάει στη στήλη C7, όπου βρίσκεται η δεξιά παρένθεση. Εφόσον, η υποέκφραση της γραμματικής έχει και αυτή το μη τερματικό ‘)’, ο κανόνας πετυχαίνει στη θέση C7, καταναλώνει την παρένθεση και αφήνει το υπόλοιπο στη θέση C8. Τελικά, το αποτέλεσμα για το Primary στη θέση C3 είναι $(7, C8)$.

Το μέγεθος του πίνακα μεγαλώνει με το μήκος της συμβολοσειράς εισόδου αλλά μόνο γραμμικά, υποθέτοντας ότι η γραμματική έχει πεπερασμένο αριθμό από μη τερματικά σύμβολα. Επιπλέον, θεωρώντας ότι η γραμματική είναι εκφρασμένη σε Backus-Naur Normal Form, μόνο ένας συγκεκριμένος αριθμός από ενδιάμεσα αποτελέσματα χρειάζεται να προσπελαστεί για να υπολογιστεί ένα νέο αποτέλεσμα. Επομένως, υποθέτοντας ότι η προσπέλαση ενός κελιού παίρνει σταθερό χρόνο (π.χ. για υλοποίηση με δισδιάστατο πίνακα), η όλη διαδικασία είναι γραμμική ως προς το μήκος της εισόδου.

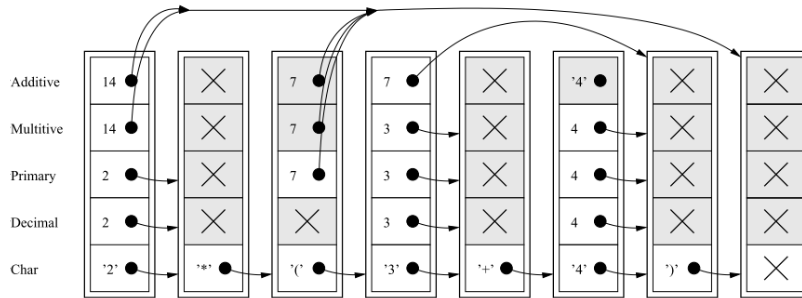
Ακριβώς επειδή κάθε κελί έχει έναν ”δείκτη” προς την επόμενη θέση της εισόδου που θα συνεχιστεί η συντακτική ανάλυση, μπορεί να συμβουλευτεί κελιά που βρίσκονται αυθαίρετα μακριά στον πίνακα. Για παράδειγμα, ο υπολογισμός του κελιού $[Primary, C3]$ χρειάστηκε αποτελέσματα από τις στήλες C3, C4 και C7. Αυτή η ικανότητα να προσπερνάει προπορευόμενα σύμβολα στην είσοδο είναι που του δίνει το infinite lookahead και τον κάνει πιο ισχυρό από έναν LR αναλυτή γραμμικού χρόνου.

### 3.2.2 Περιγραφή του Packrat Parser

Ένα προφανές πρακτικό πρόβλημα της προσέγγισης με το δυναμικό προγραμματισμό, όπου τα κελιά υπολογίζονται όλα από δεξιά προς τα αριστερά και από κάτω προς τα πάνω, είναι ο υπολογισμός πολλών ενδιάμεσων αποτελεσμάτων που δεν θα χρειαστούν. Μία επιπλέον δυσχέρεια είναι πως πρέπει εκ των προτέρων να καθορίσουμε τη σειρά με την οποία θα υπολογιστούν τα κελιά μιας στήλης. Εμείς είπαμε ότι υπολογίζονται από κάτω προς τα πάνω, όμως αυτό προϋποθέτει εξ αρχής να έχουμε βάλει τη σειρά των μη τερματικών με τέτοιο τρόπο, ώστε τα κάτω κελιά που υπολογίζονται πρώτα, να μην εξαρτώνται από τα πάνω. Για παράδειγμα, στο Σχήμα 3.1, οι κανόνες τοποθετήθηκαν έτσι ώστε παρουσιάζουν εξάρτηση από πάνω προς τα κάτω.

Το packrat parsing είναι πρακτικά ένας αναδρομικός αλγόριθμος με υπομνηματισμό που λύνει και τα δύο προβλήματα. Ένας packrat parser υπολογίζει αποτελέσματα μόνο όταν χρειάζονται, με την ίδια σειρά που θα ακολουθούσε και ένας αναλυτής αναδρομικής κατάβασης με οπισθαναχώρηση. Όμως, άπαξ και ένα ενδιάμεσο αποτέλεσμα υπολογιστεί, αποθηκεύεται για μελλοντική χρήση.

Το Σχήμα 3.2 παρουσιάζει τη δομή δεδομένων που δημιουργείται μετά από packrat parsing στην είσοδο ‘$2*(3+4)$’, χρησιμοποιώντας μία εννοιολογική αναπαράσταση με δείκτες που αναπαριστούν τη σχέση μεταξύ των κελιών. Τα γκρίζα κουτιά που περιέχουν τιμές στην πραγματικότητα δεν θα υπολογιστούν καθόλου. Τελικά, το μόνο που μας ενδιαφέρει να υπολογιστεί είναι το κελί στη γραμμή που αντιστοιχεί στο αρχικό σύμβολο της γραμματικής ($Additive$) και στη στήλη που αντιστοιχεί στην αρχή της εισόδου (χαρακτήρας ‘2’).




**Σχήμα 3.2:** Παράδειγμα εκτέλεσης Packrat Parsing [Ford02b]

Η αναπαράσταση καθιστά σαφές γιατί ο αλγόριθμος είναι πολυπλοκότητας $O(n)$ ως προς το μήκος $n$ της εισόδου. Η πρώτη συνάρτηση είναι αυτή η οποία ξεκινά τις αναδρομικές κλήσεις προς άλλες συναρτήσεις. Κάθε κελί υπολογίζεται το πολύ μία φορά, οπότε ο αλγόριθμος είναι γραμμικός ως προς την είσοδο. Προφανώς, η σειρά με την οποία υπολογίζονται τα ενδιάμεσα αποτελέσματα διαφέρει από τον προηγούμενο αλγόριθμο δυναμικού προγραμματισμού, που αναφέραμε νωρίτερα.

Τέλος, είναι προφανές ότι ο αλγόριθμος καταναλώνει μνήμη της τάξης $O(NT * n)$, όπου $NT$ ο αριθμός των μη τερματικών της εκάστοτε γραμματικής, και $n$ το μήκος της εισόδου. Αυτό αποτελεί εν δυνάμει ένα μειονέκτημα του αλγορίθμου, καθώς για μεγάλα αρχεία στην είσοδο, η δεσμευμένη μνήμη αυξάνεται γραμμικά. Στο Κεφάλαιο 5 θα περιγράψουμε μία βελτίωση του αλγορίθμου που βελτιώνει δραματικά τον απαιτούμενο χώρο εκτέλεσης.

### 3.2.3 Ενσωματωμένη λεκτική ανάλυση

Στους παραδοσιακούς συντακτικούς αναλυτές συνήθως ακολουθείται η παραδοχή ότι η είσοδος έχει ήδη υποστεί προεπεξεργασία από έναν λεκτικό αναλυτή. Η λεκτική ανάλυση, δηλαδή, είναι ξεχωριστή διαδικασία η οποία χωρίζει τη συμβολοσειρά εισόδου, μετατρέποντάς τη σε ένα ρεύμα (stream) από διακριτές μονάδες (tokens). Ο συντακτικός αναλυτής αντιμετωπίζει αυτές τις μονάδες σαν έτοιμα τερματικά, παρόλο που μπορεί να αντιπροσωπεύουν παραπάνω από έναν χαρακτήρες. Αυτός ο διαχωρισμός προσφέρει αρκετά πλεονεκτήματα όπως ευκολότερη αποσφαλμάτωση ή ευκολότερη ερμηνεία των κανόνων της γραμματικής χρησιμοποιώντας μη τερματικά ”υψηλότερου επιπέδου” από απλούς χαρακτήρες.

Στο packrat parsing, το άπειρο lookahead που προκύπτει από την αποθήκευση όλων των ενδιάμεσων αποτελεσμάτων, επιτρέπει άνετα την ενσωμάτωση της λεκτικής ανάλυσης στη συντακτική. Αρκεί, απλώς, να προστεθούν επιπλέον κανόνες στη γραμματική οι οποίοι θα είναι υπεύθυνοι για τη λεκτική ανάλυση. Για παράδειγμα, στο Σχήμα 3.3 η εντολή if στην Java, αφενός ορίζεται στη σύνταξη της γλώσσας ως ένα statement που ξεκινάει με τη λεκτική μονάδα ”if”, αφετέρου η μονάδα αυτή ορίζεται ως ακολουθία δύο χαρακτήρων και, ίσως μετά, κενών χαρακτήρων.

Έτσι, έχουμε συμπεριλάβει τόσο τη λεκτική, όσο και τη συντακτική ανάλυση της εντολής. Περισσότερα σχετικά με τα πλεονεκτήματα και μειονεκτήματα του Packrat Parsing, ιδιαίτερα σε σχέση με τις Context Free Grammars μπορούν να βρεθούν στο [Ford04].

## 3.3 Υλοποίηση

### 3.3.1 Υλοποίηση της γραμματικής

Το packrat parsing είναι θεμελιωδώς μία στρατηγική καθοδικής συντακτικής ανάλυσης, οπότε η υλοποίηση του σχετίζεται στενά με τους αναλυτές αναδρομικής κατάβασης. Θεμελιώδη ρόλο στην υλοποίηση και στους μετέπειτα πειραματισμούς μας παίζει και ο τρόπος με τον οποίο θα μοντελοποιήσουμε τον αναλυτή αλλά και όλη τη διαδικασία της ανάλυσης, ώστε να αποτυπωθούν σε κώ-

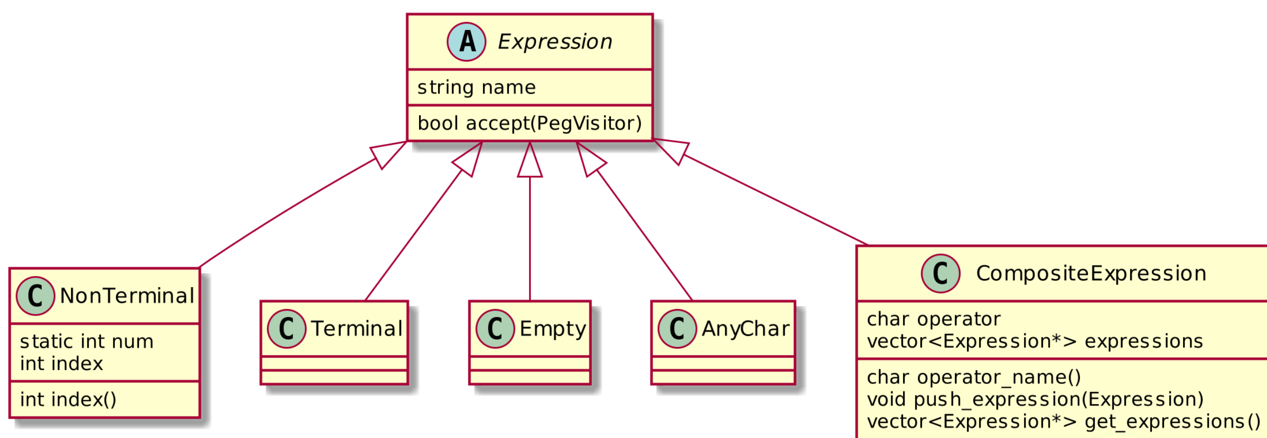
```
# Syntax Description
Statement
    <- Block
    / ASSERT Expression (COLON Expression)? SEMI
                            ...
    / IF ParExpression Statement (ELSE Statement)?

# Token Description
IF <- 'i' 'f' !LetterOrDigit Spacing?
```

**Σχήμα 3.3:** Ενσωματωμένη λεκτική ανάλυση για την εντολή if στην Java

δικα. Σε μία γλώσσα αντικειμενοστραφούς προγραμματισμού, όπως η C++, ένας τέτοιος αναλυτής θα μπορούσε να αναπαρασταθεί με ένα αντικείμενο. Το ίδιο και τα διάφορα επιμέρους κομμάτια που αποτελούν την εκάστοτε γραμματική.

Για καλύτερη οπτικοποίηση και κατανόηση, παραθέτουμε τα UML διαγράμματα που απεικονίζουν την αναπαράσταση των στοιχείων μίας PEG, με βάση τα όσα περιγράψαμε στη θεωρία που προηγήθηκε. Θεωρούμε ότι τα UML διαγράμματα παρουσιάζουν μια απλοποιημένη C++ σύνταξη.




**Σχήμα 3.4:** Μοντελοποίηση των δομικών στοιχείων μίας PEG

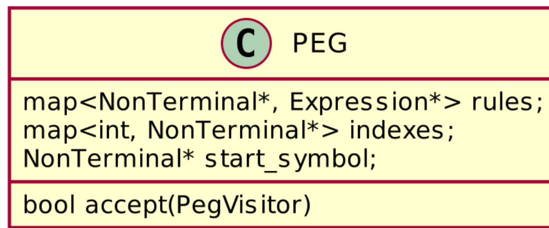
Όλες οι δομικές μονάδες της γραμματικής (τερματικά, μη τερματικά και οι σύνθετες εκφράσεις που τα περιέχουν) είναι εκφράσεις. Έτσι, προκύπτουν οι κλάσεις NonTerminal, Terminal και CompositeExpression που κληρονομούν την αφηρημένη κλάση Expression. Επιπλέον, χρειαζόμαστε και μία έκφραση που να αναπαριστά την κενή συμβολοσειρά (Empty), καθώς και μία που να αναπαριστά οποιονδήποτε χαρακτήρα (AnyChar). Κατ' ελάχιστο, όλες οι εκφράσεις πρέπει να έχουν ένα όνομα, καθώς και να "δέχονται" έναν "επισκέπτη" (visitor pattern). Όπως θα δούμε, ο επισκέπτης αυτός είναι ο συντακτικός αναλυτής που θα προσπαθήσει να τις αναγνωρίσει.

Επιπλέον, κάθε μη τερματικό θεωρούμε ότι αντιστοιχεί σε έναν μοναδικό ακέραιο index. Ακόμη, το CompositeExpression θεωρούμε ότι αποτελεί το σύνολο των Expressions που συνδέονται με έναν μόνο τελεστή. Για παράδειγμα:

$$A/(BC) \tag{3.2}$$

Εδώ υπάρχει ένα CompositeExpression (έστω $C_1$) με τελεστή την ακολουθία και εκφράσεις τα $B$ και $C$, καθώς και ένα CompositeExpression με τελεστή την διατεταγμένη επιλογή ('$/$') και εκφράσεις τα $A$ και $C_1$. Στο Σχήμα 3.4 φαίνονται ενδεικτικά οι βασικές μέθοδοι των κλάσεων με βάση τα πεδία που περιγράψαμε.

Με βάση αυτές τις δομικές μονάδες κατασκευάζουμε την κλάση μίας γραμματικής PEG, όπως φαίνεται στο Σχήμα 3.5.




**Σχήμα 3.5:** Μοντελοποίηση μίας PEG

Επομένως, τα βασικά πεδία μίας PEG είναι τα εξής:

- rules: Οι κανόνες θεωρούμε ότι είναι μία αντιστοίχιση από μη τερματικά σε σύνθετες εκφράσεις
- indexes: Η γραμματική ενσωματώνει την πληροφορία για την αντιστοιχία των μη τερματικών με μοναδικούς ακεραίους.
- start_symbol: Το αρχικό σύμβολο της γραμματικής

Όπως και τα επιμέρους στοιχεία της, η γραμματική δέχεται έναν visitor.

### 3.3.2 Υλοποίηση του συντακτικού αναλυτή

Αφού μοντελοποιήσαμε μία PEG, μπορούμε τώρα να κάνουμε το ίδιο και για έναν συντακτικό αναλυτή για αυτήν. Πώς όμως θα μοντελοποιήσουμε έναν τέτοιο συντακτικό αναλυτή? Είπαμε προηγουμένως ότι μία γραμματική PEG θα δέχεται έναν τέτοιον αναλυτή ως επισκέπτη (visitor). Επιπλέον, ο parser μας πρέπει να ενθυλακώσει (encapsulate):

- Τη συμβολοσειρά εισόδου
- Την τρέχουσα θέση όπου βρίσκεται η ανάλυση στη συμβολοσειρά εισόδου
- Τον πίνακα ενδιάμεσων αποτελεσμάτων
- Τη δυνατότητα να αναλύσει συντακτικά μία PEG, καθώς και τα μεμονωμένα συστατικά της

Με βάση τα παραπάνω, μια μοντελοποίηση για τον Packrat Parser θα ήταν η εξής:

Packrat

string in;
int pos;
PEG peg;
Cell** cells;

bool visit(NonTerminal& nt);
bool visit(Terminal& t);
bool visit(CompositeExpression& ce);
bool visit(Empty& e);
bool visit(AnyChar& ac);
bool visit(PEG& peg);

**Σχήμα 3.6:** Μοντελοποίηση του Packrat Parser

Παρατηρούμε ότι, αντίστοιχα με μία PEG και τα συστατικά της που δέχονται ως ”επισκέπτη” έναν συντακτικό αναλυτή, ο Packrat Parser υλοποιεί τις μεθόδους του επισκέπτη (visitor pattern).

Ενδεικτικά, η μέθοδος *visit(NonTerminal& nt)*, φαίνεται στο Σχήμα 3.7.

Αρχικά βρίσκουμε σε ποιο κελί του πίνακα αντιστοιχεί το μη τερματικό, με βάση το μοναδικό αριθμό του τερματικού (για τη γραμμή του πίνακα) και με βάση τη τρέχουσα θέση εισόδου (για τη στήλη του πίνακα) (γραμμές 3-5). Στη συνέχεια, αν το ενδιάμεσο αποτέλεσμα είναι ήδη υπολογισμένο, επιστρέφουμε ανάλογα με την περίπτωση είτε επιτυχώς, θέτοντας την αντίστοιχη θέση στην είσοδο (γρ. 9-13), είτε ανεπιτυχώς (γρ. 14-17). Αλλιώς, το υπολογίζουμε και αποθηκεύουμε το αντίστοιχο ενδιάμεσο αποτέλεσμα (γρ. 18-31).

Επιπλέον, η μέθοδος *visit(Terminal& t)*, φαίνεται στο Σχήμα 3.8.

Πρακτικά, ο αναλυτής κοιτάει αν ο χαρακτήρας στο τρέχον σημείο εισόδου ταυτίζεται με το χαρακτήρα του τερματικού. Αν ναι, τότε επιστρέφει επιτυχώς και ανεβάζει κατά 1 το δείκτη της εισόδου. Αλλιώς, επιστρέφει ανεπιτυχώς.

Τέλος, η μέθοδος *visit(CompositeExpression& ce)*, φαίνεται στο Σχήμα 3.9 για την περίπτωση της ακολουθίας και της διατεταγμένης επιλογής.

Προκειμένου να διαχωρίσει μεταξύ των διάφορων σύνθετων εκφράσεων, ο συντακτικός αναλυτής κοιτάει τον τελεστή της σύνθετης έκφρασης, ο οποίος ανάλογα με τη σύμβαση υποδηλώνει και κάτι διαφορετικό (π.χ. ο χαρακτήρας '/' αντιστοιχεί στη διατεταγμένη επιλογή).

Για την ακολουθία, ο αναλυτής εξετάζει όλες τις υποεκφράσεις και, αν έστω και μία αποτύχει, θέτει το δείκτη εισόδου στην αρχική θέση και επιστρέφει ανεπιτυχώς. Αλλιώς, αν πετύχουν όλες, επιστρέφει επιτυχώς (ο δείκτης εισόδου θα έχει προχωρήσει από τις εσωτερικές κλήσεις στις υποεκφράσεις).

Η δυαδική διαδικασία είναι στη διατεταγμένη επιλογή: ο αναλυτής κοιτάει μία μία τις υποεκφράσεις και, αν έστω και μία πετύχει, επιστρέφει επιτυχώς με την πρώτη που πετυχαίνει (που θα έχει προχωρήσει αυτή το δείκτη εισόδου). Αλλιώς, αν όλες αποτύχουν, επαναφέρει το δείκτη εισόδου και επιστρέφει ανεπιτυχώς.

```cpp
bool SerialPackrat::visit(NonTerminal& nt)
{
    int row = nt.index();
    Cell* cur_cell = &cells[row][pos];
    Result cur_res = cur_cell->res();

    switch (cur_res) {

        case Result::success:
        {
            pos = cur_cell->pos();
            return true;
        }
        case Result::fail:
        {
            return false;
        }
        case Result::unknown:
        {
            Expression* e = peg.get_expr(&nt);
            auto res = e->accept(*this);

            if (res) {
                cur_cell->set_res(Result::success);
                cur_cell->set_pos(pos); // pos has changed
                return true;
            } else {
                cur_cell->set_res(Result::fail);
                return false;
            }
        }
    }
    return false;
}
```

**Σχήμα 3.7:** Συντακτική ανάλυση ενός μη τερματικού

```cpp
bool SerialPackrat::visit(Terminal& t)
{
    int terminal_char = t.name()[0];
    ...
    if (pos < in.size() && terminal_char == this->cur_tok()) {
        pos++;
        return true;
    }
    return false;
}
```

**Σχήμα 3.8:** Συντακτική ανάλυση ενός τερματικού

```cpp
bool SerialPackrat::visit(CompositeExpression& ce)
{
    char op = ce.op_name();
    std::vector<Expression*> exprs = ce.expr_list();
    int orig_pos = pos;

    switch (op) {

        case '\b':  // sequence
        {
            for (auto expr : exprs)
                if (!expr->accept(*this)) {
                    pos = orig_pos;
                    return false;
                }
            return true;
        }
        case '/':   // ordered choice
        {
            for (auto expr : exprs) {
                pos = orig_pos;
                if (expr->accept(*this))
                    return true;
            }
            pos = orig_pos;
            return false;
        }
        ...
    }
}
```

**Σχήμα 3.9:** Συντακτική ανάλυση της ακολουθίας και της διατεταγμένης επιλογής

## Κεφάλαιο 4

# Γεννήτορας Συντακτικών Αναλυτών Packrat

Στο προηγούμενο κεφάλαιο, περιγράψαμε πώς λειτουργεί ένας packrat parser, καθώς και πώς θα μπορούσαμε να τον μοντελοποιήσουμε. Βέβαια, δεν μας ενδιαφέρει απλά να ξέρουμε πώς τον κατασκευάζουμε με το χέρι, αλλά και το πώς θα μπορούσαμε να κατασκευάσουμε ένα εργαλείο που θα παίρνει ως είσοδο μία τυπική περιγραφή της γραμματικής, και θα δίνει ως έξοδο έναν συντακτικό αναλυτή για αυτήν. Πρακτικά, θέλουμε έναν *μεταγλωττιστή-μεταγλωττιστή (compiler-compiler)*, όπως το εργαλείο YACC στον κόσμο της C. Δηλαδή, έναν γεννήτορα συντακτικών αναλυτών.

Το πλεονέκτημα που θα προσέφερε ένα τέτοιο εργαλείο, είναι πως θα μπορούσε να παίρνει ως είσοδο γραμματικές που έχουν, για παράδειγμα, αριστερή αναδρομή (η οποία δεν υποστηρίζεται από τις κλασικές PEGs), και να τις μετατρέπει εσωτερικά σε δεξιά αναδρομικές [Ford02a]. Ακόμη, επιτρέπει στον σχεδιαστή μίας γλώσσας να επικεντρωθεί στις υψηλού επιπέδου λεπτομέρειες της γραμματικής, χωρίς το φόρτο της διαρκούς υλοποίησης του αναλυτή με το χέρι. Τέλος, δίνει τη δυνατότητα με αυτόματο τρόπο να μπορούμε να επεξεργαζόμαστε τις γραμματικές σε υψηλό επίπεδο (π.χ. να ελέγξουμε με κάποιο εργαλείο ότι οι κανόνες δεν έχουν κάποια κυκλική αναφορά), ενώ επιτρέπει και την εύκολη επαναχρησιμοποίηση κανόνων σε διάφορες γραμματικές.

Αυτό που αλλάζει από γραμματική σε γραμματική για τον parser μας είναι η μοντελοποίηση της γραμματικής που ενθυλακώνει, όπως δείξαμε στο προηγούμενο κεφάλαιο. Δηλαδή, για να γεννήσουμε αυτόματα έναν packrat parser, αρκεί να γενήσσουμε τη μοντελοποίση της γραμματικής που θέλουμε να αναλύσει, οπότε είναι σαν να δημιουργήσαμε και τον ίδιο.

Η διαδικασία που θα περιγράψουμε σε αυτό το κεφάλαιο συνοψίζεται στο Σχήμα 4.1.

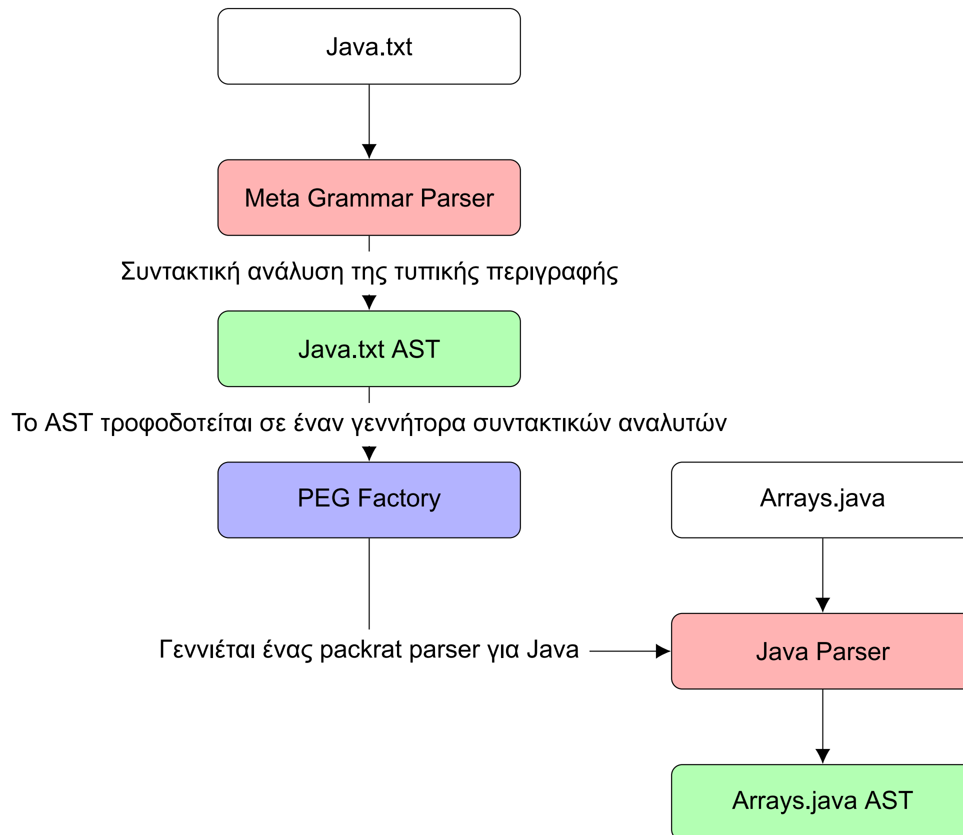
Έστω ότι θέλουμε να αναλύσουμε συντακτικά ένα πρόγραμμα Arrays.java. Αρχικά, θεωρούμε ότι διαθέτουμε μία τυπική περιγραφή της γραμματικής Java, εκφρασμένης ως parsing expression grammar, μέσα σε ένα .txt αρχείο. Αυτή η περιγραφή αναλύεται από έναν "Meta parser" και κατασκευάζεται το συντακτικό της δέντρο (AST). Ακολούθως, το δέντρο αυτό τροφοδοτείται σε έναν γεννήτορα γραμματικών PEG (PEG Factory), ο οποίος διασχίζει το δέντρο και γεννάει μία PEG για Java προγράμματα. Πρακτικά, είναι σαν να γεννάει και έναν packrat parser για Java προγράμματα, αφού είπαμε ότι ο parser ενθυλακώνει τη γραμματική που θα αναλύσει. Άρα, αν έχουμε ένα στιγμιότυπο της Java γραμματικής, σε μορφή PEG, μπορούμε να φτιάξουμε και το αντίστοιχο στιγμιότυπο και για τον συντακτικό αναλυτή. Πλέον, μπορούμε να τροφοδοτήσουμε το Arrays.java στον νέο packrat parser.

## 4.1 Τυπική Περιγραφή Γραμματικών

Είπαμε ότι ως είσοδο στο γεννήτορα συντακτικών αναλυτών θα δίνουμε μία τυπική περιγραφή μίας γραμματικής. Πώς, όμως, ορίζεται μία τέτοια περιγραφή? Στο Σχήμα 4.2, παρουσιάζεται η συντακτική περιγραφή μίας PEG εκφρασμένης σε τυπική περιγραφή PEG [Inst06].

Δηλαδή, μία PEG (κανόνας Grammar) αποτελείται από διαδοχικούς ορισμούς (κανόνας Definition). Ο κάθε ορισμός αποτελείται από ένα μη τερματικό (κανόνας Identifier), ένα αριστερό βέλος (LEFTARROW) και μία έκφραση (κανόνας Expression) κ.ό.κ.

Το Σχήμα 4.2 περιγράφει μόνο τη συντακτική δομή, και όχι το πώς κατασκευάζονται οι λέξεις (tokens) μίας γραμματικής PEG. Η πλήρης περιγραφή γίνεται στο Παράρτημα Α.




**Σχήμα 4.1:** Η διαδικασία αυτόματης παραγωγής ενός packrat parser για Java προγράμματα

```
Grammar    <- Spacing Definition+ EndOfFile                  # Type 0
Definition <- Identifier LEFTARROW Expression                # Type 1
Expression <- Sequence (SLASH Sequence)*                     # Type 2
Sequence   <- Prefix*                                        # Type 3
Prefix     <- (AND / NOT)? Suffix                            # Type 4
Suffix     <- Primary (QUESTION / STAR / PLUS)?              # Type 5
Primary    <- Identifier !LEFTARROW                          # Type 6
            / OPEN Expression CLOSE
            / Literal / DOT
```

**Σχήμα 4.2:** Μία PEG περιγράφει τυπικά τη συντακτική δομή της

Η είσοδος στο γεννήτορά μας είναι πλέον έτοιμη. Πρακτικά, είναι ένα αρχείο κειμένου που θα περιγράφει τη γραμματική μίας γλώσσας (Java, XML, κλπ.) σε PEG τυπική μορφή.

## 4.2 Στιγμιότυπο μίας μετα-γραμματικής που περιγράφει Parsing Expresssion Grammars

Έχοντας έτοιμη την είσοδο της τυπικής περιγραφής, χρειαζόμαστε έναν μετα-αναλυτή για να την αναλύσει, όπως περιγράψαμε στο Σχήμα 4.1. Το στιγμιότυπο αυτής της γραμματικής αναγκαστικά πρέπει να φτιαχτεί "με το χέρι". Δηλαδή, να δημιουργήσουμε τα κατάλληλα αντικείμενα που απαρτίζουν τις εκφράσεις και τους κανόνες μίας τέτοιας γραμματικής.

Ενδεικτικά, στο Σχήμα 4.3, παραθέτουμε τον κώδικα για τη δημιουργία του πρώτου κανόνα της μετα-γραμματικής, που λέει ότι μία PEG αποτελείται από ένα σύνολο διαδοχικών κανόνων (definitions).

Ο τελεστής της ”ακολουθίας” θεωρούμε ότι είναι ο χαρακτήρας ‘$\backslash b$’.

```
NonTerminal grammar("Grammar");
NonTerminal spacing("Spacing");
NonTerminal definition("Definition");
NonTerminal endOfFile("EndOfFile");

// Grammar <- Spacing Definition+ EndOfFile                    # Type 0
CompositeExpression grammarExp('\b');
grammarExp.push_expr(&spacing);
grammarExp.push_expr(new CompositeExpression('+', {&definition}));
grammarExp.push_expr(&endOfFile);
this->push_rule(&grammar, &grammarExp);
```

**Σχήμα 4.3:** Δημιουργία στιγμιοτύπου του πρώτου κανόνα της μετα-γραμματικής

## 4.3 Γεννήτορας συντακτικών αναλυτών packrat

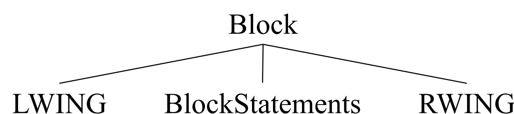
Συνεχίζοντας το σκεπτικό του Σχήματος 4.1, έχοντας κατασκευάσει τη μετα-γραμματική, της δίνουμε ως είσοδο μία τυπική περιγραφή της γραμματικής-στόχου (π.χ. ένα .txt αρχείο). Το αποτέλεσμα είναι το Αφηρημένο Συντακτικό Δέντρο (AST). Σε αυτή την ενότητα περιγράφουμε πώς ένας γεννήτορας γραμματικών peg (peg factory), παίρνει ως είσοδο το AST και δίνει ως έξοδο την αντίστοιχη peg.

Η βασική ιδέα είναι ότι οι κόμβοι του AST έχουν ονόματα τα οποία δείχνουν σε ποιο μη τερματικό της μετα-γραμματικής αντιστοιχούν. Για παράδειγμα, ένας κόμβος με το όνομα ”$Sequence$” αντιλαμβανόμαστε ότι αντιστοιχεί σε κάποια ακολουθία, οπότε τα παιδιά του θα είναι τα μέλη της ακολουθίας. Έτσι, θα επισκεπτούμε ένα-ένα τα παιδια του και θα τα βάλουμε σε ένα $CompositeExpression$ ακολουθίας.

Έστω, λοιπόν, ότι μία γραμμή στο Java.txt (που περιέχει την τυπική περιγραφή σε Java), δίνει τον εξής κανόνα:

$$Block \leftarrow LWING\ BlockStatements\ RWING \tag{4.1}$$

ο οποίος λέει πως ένα block εντολών σε Java αποτελείται από ένα σύνολο εντολών ανάμεσα σε παρενθέσεις. Ο meta-parser όταν αναλύσει αυτήν την πρόταση θα δώσει ως εσωτερικό κόμβο του AST το εξής:




**Σχήμα 4.4:** Ο κόμβος του AST που αφορά το Block

Σκοπός του γεννήτορα είναι να διασχίσει τους διάφορους κόμβους του AST, δημιουργώντας παράλληλα τα αντίστοιχα στιγμιότυπα των εκφράσεων και των κανόνων. Τελειώνοντας τη διάσχιση όλου του δέντρου, λοιπόν, θα έχει δημιουργήσει το στιγμιότυπο ολόκληρης της γραμματικής-στόχου (στο παράδειγμά μας, της Java).

Ενδεικτικά, ένας απλοποιημένος κώδικας για τη διάσχιση του κόμβου Block, το οποίο αποτελεί μία ακολουθία από εκφράσεις, θα ήταν όπως στο Σχήμα 4.5.

Θεωρούμε ότι η traverse(child) επισκέπτεται το κάθε παιδί-κόμβο και δημιουργεί το αντίστοιχο μη τερματικό, επιστρέφοντάς το. Οπότε, ο κώδικας λέει πως αν ο κόμβος έχει πάνω από ένα παιδί,

```
Expression* construct_sequence(TreeNode* node)
{
    if (node->children_num() > 1) {

        auto ce = new CompositeExpression('\b');

        for (auto child : node->get_children()) {
                Expression* e = traverse(child);
                ce->push_expr(e);
        }

        return ce;
    }
    else {
        return traverse(child);
    }
}
```

**Σχήμα 4.5:** Διάσχιση ακολουθίας και δημιουργία στιγμιοτύπων για τη γραμματική-στόχο

τότε δημιούργησε ένα CompositeExpression ακολουθίας με όλα αυτά τα παιδιά ως μη τερματικά. Αλλιώς, επίστρεψε απλά το μη τερματικό του μοναδικού παιδιού.

Τελικά, για το παράδειγμά μας, τα στιγμιότυπα που θα γεννιούνταν θα ήταν όπως στο Σχήμα 4.6.

```
NonTerminal parent("Block");
NonTerminal child1("LWING");
NonTerminal child2("BlockStatements");
NonTerminal child3("RWING");

// Block <- LWING BlockStatements RWING
CompositeExpression grammarExp('\b');
grammarExp.push_expr(&child1);
grammarExp.push_expr(&child2);
grammarExp.push_expr(&child3);
this->push_rule(&parent, &grammarExp);
```

**Σχήμα 4.6:** Στιγμιότυπα για τον πρώτο κανόνα της μετα-γραμματικής

Διασχίζοντας όλο το AST, γεννιέται μία γραμματική Java, εκφρασμένη ως PEG. Από αυτήν έχουμε αυτομάτως και τον αντίστοιχο Java packrat parser. Το τελευταίο βήμα για τη διαδικασία του Σχήματος 4.1, είναι να δώσουμε ως είσοδο στον Java packrat parser ένα αρχείο Java. Έτσι, παίρνουμε το AST για το .java αρχείο, οπότε ο στόχος μας επιτεύχθηκε.

## Κεφάλαιο 5

# Packrat Parsing με Ελαστικό Κυλιόμενο Παράθυρο

Όπως αναφέραμε, οι αναλυτές packrat, παρά την απλότητα και την γραμμική επίδοση (ως προς το μήκος της εισόδου) που προσφέρουν, έχουν το μειονέκτημα ότι καταναλώνουν πολύ χώρο για την αποθήκευση των ενδιάμεσων αποτελεσμάτων. Αυτό αποτελεί αποθαρρυντικό παράγοντα για τη χρησιμοποίησή τους σε πολλές εφαρμογές. Στην ενότητα αυτή περιγράφουμε μία ευριστική τεχνική για τη βελτίωση της κατανάλωσης μνήμης του packrat parser, χρησιμοποιώντας ένα *κυλιόμενο παράθυρο (packrat parsing with elastic sliding window)* [Kura15].

## 5.1 Εισαγωγή

Η ιδέα πίσω από το packrat parsing με ελαστικό κυλιόμενο παράθυρο (ή πιο απλά elastic packrat parsing) βασίζεται στην έννοια του μήκους της μέγιστης οπισθαναχώρησης (longest backtrack length). Ουσιαστικά, όταν προχωράει ο αναλυτής δεξιότερα ως προς την είσοδο, μπορεί να χρειαστεί να οπισθαναχωρήσει, όχι όμως αναγκαστικά μέχρι την αρχή της εισόδου, αλλά μέχρι ένα μικρότερο μήκος. Έστω, ότι είχαμε έναν μικρότερο πίνακα υπομνηματισμού (παράθυρο) το οποίο ”κυλάει” προς τα δεξιά ως προς την είσοδο και έχει επαρκές πλάτος ώστε να καλύπτει το μήκος της μέγιστης οπισθαναχώρησης. Τότε, ο πίνακας αυτός έχει τον απαραίτητο χώρο ώστε να αποθηκεύει όλα τα ενδιάμεσα αποτελέσματα, χωρίς να χρειαστεί οπισθαναχώρηση του αλγορίθμου.

Στην πράξη, βέβαια, είναι δύσκολο να ξέρουμε εκ των προτέρων πόση θα είναι η μέγιστη οπισθαναχώρηση. Εναλλακτικά, επιλέγουμε ένα προσεγγιστικό μέγεθος για το παράθυρο από εμπειρικά δεδομένα και, αν χρειαστεί, το επεκτείνουμε κατά τη διάρκεια της συντακτικής ανάλυσης.

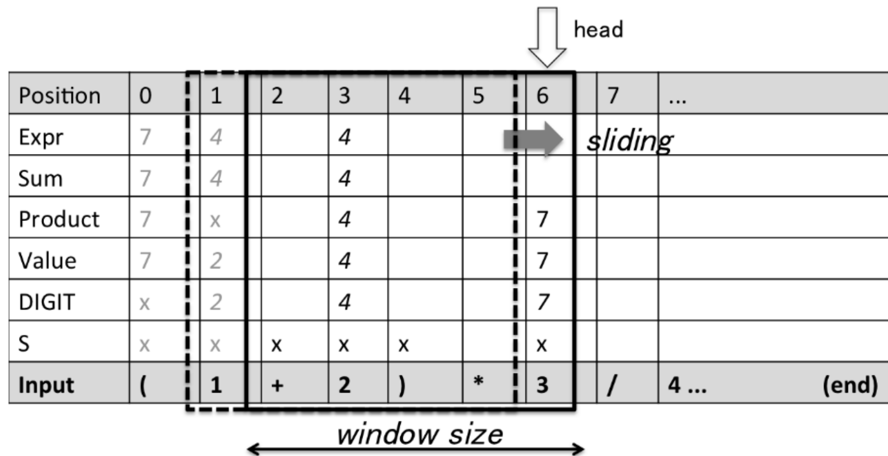
Όμως, η μέθοδος αυτή δεν περιορίζει μόνο τη χρήση κελιών ως προς τη διάσταση της εισόδου, αλλά και ως προς τη διάσταση των μη τερματικών συμβόλων. Συγκεκριμένα, κατά την εκτέλεση της συντακτικής ανάλυσης μετριέται δυναμικά κατά πόσο κάθε μη τερματικό αξίζει να αποθηκεύεται στον πίνακα (παράθυρο) ή όχι. Έτσι, ο χώρος που θα περισσέψει μπορεί να δοθεί ώστε να επεκταθεί το πλάτος του παραθύρου. Εξ ού, και η ”ελαστικότητα” του παραθύρου.

## 5.2 Κυλιόμενο παράθυρο

Το παράθυρο πρακτικά είναι ένας buffer σταθερού μεγέθους, ο οποίος κυλάει προς τα δεξιά ως προς την είσοδο ώστε να περιλάβει τα νέα δεδομένα, ενώ τα παλιά δεδομένα απορρίπτονται από αυτόν. Το Σχήμα 5.1 απεικονίζει έναν buffer πλάτους 5 θέσεων πάνω από έναν πίνακα υπομνηματισμού. Η δεξιότερη θέση του παραθύρου ταυτίζεται με το μέτωπο της συντακτικής ανάλυσης.

Αν το μέγεθος του παραθύρου δεν είναι επαρκώς μεγάλο, τότε θα ”εκτοπιστεί” κάποιο χρήσιμο κελί του πίνακα υπομνηματισμού το οποίο περιέχει ένα ενδιάμεσο αποτέλεσμα που στο μέλλον θα χρειαστεί. Αυτό σημαίνει ότι πρακτικά χάνεται η εγγύηση για γραμμικό χρόνο συντακτικής ανάλυσης, αφού το συγκεκριμένο κελί που χάθηκε θα πρέπει να υπολογιστεί ξανά, όπως θα έκανε ένας αναδρομικός αναλυτής με οπισθαναχώρηση.

Πόσο, όμως, πρέπει να είναι το πλάτος του παραθύρου, ώστε να αποφευχθεί αυτή η δυσχέρεια? Αυτή είναι μία δύσκολη ερώτηση. Θα μπορούσαμε να το κάνουμε όσο μεγάλη είναι και η είσοδος



| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|---|---|---|---|---|---|---|---|---|---|
| Expr | 7 | 4 | | 4 | | | | | |
| Sum | 7 | 4 | | 4 | | | | | |
| Product | 7 | x | | 4 | | | 7 | | |
| Value | 7 | 2 | | 4 | | | 7 | | |
| DIGIT | x | 2 | | 4 | | | 7 | | |
| S | x | x | x | x | x | | x | | |
| **Input** | **(** | **1** | **+** | **2** | **)** | ***** | **3** | **/** | **4 ... (end)** |



**Σχήμα 5.1:** Κυλιόμενο παράθυρο για τον πίνακα υπομνηματισμού [Kura15]

ώστε να μην ανησυχούμε για το αν θα χάσουμε κάποιο ενδιάμεσο αποτέλεσμα, όμως αυτό είναι ισοδύναμο με την αρχική έκδοση του packrat parsing. Το άλλο άκρο θα ήταν το πλάτος να είναι 1, που θα ισοδυναμούσε με έναν αναλυτή αναδρομικής κατάβασης με οπισθαναχώρηση.

Στο [Kura15] παρουσιάζονται πειραματικές μετρήσεις για μία ποικιλία προγραμμάτων σε διάφορες γλώσσες, όπου φαίνεται πως οι πιο πολλές περιπτώσεις οπισθαναχώρησης συμβαίνουν σχετικά κοντά στο μέτωπο της συντακτικής ανάλυσης. Συγκεκριμένα, ακόμα και ένα παράθυρο μήκους 16 bytes καταφέρνει να αποφύγει τον περιττό υπολογισμό κελιών στις περισσότερες περιπτώσεις οπισθαναχώρησης.

Το ξεκάθαρο πλεονέκτημα ενός παραθύρου είναι ότι διασφαλίζει ένα άνω φράγμα στη μνήμη που δεσμεύουμε στο σωρό. Αν μάλιστα το μήκος του παραθύρου είναι επαρκές, τότε εξακολουθεί να ισχύει και η εγγύηση του γραμμικού χρόνου εκτέλεσης. Ωστόσο, αν όχι, τότε υπάρχει ο κίνδυνος για ακόμα και εκθετικό χρόνο εκτέλεσης ως προς το μήκος της εισόδου, όπως θα έκανε ο αναλυτής αναδρομικής κατάβασης με οπισθαναχώρηση.

## 5.3 Δυναμική απενεργοποίηση μη τερματικών συμβόλων

Όπως αναφέραμε, στο elastic packrat parsing γίνεται προσπάθεια να περιοριστούν τα ενδιάμεσα αποτελέσματα που αποθηκεύονται, περιορίζοντας όχι μόνο το μήκος της εισόδου που καλύπτουμε, αλλά και τα μη τερματικά που εξυπηρετούμε. Θα μπορούσαμε να την κάνουμε αυτή την ανάλυση στατικά, όμως η προσέγγιση που επιλέγεται είναι η δυναμική ανάλυση.

Συγκεκριμένα, ξεκινάμε θεωρώντας ότι όλα τα μη τερματικά είναι ενεργά, δηλαδή ότι αποθηκεύουμε ενδιάμεσα αποτελέσμα που τα αφορούν. Δίνουμε σε όλα τα μη τερματικά έναν αριθμό ευκαιριών. Στη συνέχεια, αν μετά από έναν συγκεκριμένο αριθμό κλήσεων για ένα μη τερματικό δούμε ότι τα ενδιάμεσα αποτελέσματα που το αφορούν δε χρησιμοποιήθηκαν ούτε μία φορά, το ”απενεργοποιούμε” (Σχήμα 5.2).

Δηλαδή, παύουμε να κρατάμε αποτελέσματα για αυτό, και όταν πρέπει να το αναλύσουμε, χρησιμοποιούμε απευθείας συντακτική ανάλυση αναδρομικής κατάβασης, χωρίς να εμπλακεί ο πίνακας ενδιάμεσων αποτελεσμάτων (Σχήμα 5.3).

Αν, ωστόσο, αξιοποιήσουμε έστω και ένα ενδιάμεσο αποτέλεσμα του μη τερματικού προτού απενεργοποιηθεί, το κρατάμε μόνιμα ενεργοποιημένο, όπως φαίνεται στο Σχήμα 5.4.

Το ποιο θα είναι αυτό το ”κατώφλι”, δηλαδή ο αριθμός κλήσεων στο μη τερματικό που δεν αξιοποιεί ούτε ένα ενδιάμεσο αποτέλεσμα, είναι μια παράμετρος που πρέπει να μετρήσουμε.

```
...
if (nt_elapsed[row] >= 0)  // μετράμε αντίστροφα όταν συναντάμε ένα μη τερματικό
        nt_elapsed[row] = nt_elapsed[row] - 1;  // χωρίς να το χρησιμοποιήσουμε
...
if (nt_elapsed[row] == 0)  // αν έληξαν οι ευκαιρίες του
    if (!nt_utilized[row])  // και δε χρησιμοποιήθηκε ούτε μία φορά
        nt_activated[row] = false;  // το απενεργοποιούμε μόνιμα
...
```

**Σχήμα 5.2:** Απενεργοποιήση μη τερματικού

```
if (!nt_activated[row]) {
    Expression* e = peg.get_expr(&nt);
    return e->accept(*this);
}
```

**Σχήμα 5.3:** Ένα απενεργοποιημένο μη τερματικό αναλύεται απευθείας χωρίς τη συμμετοχή του πίνακα ενδιάμεσων αποτελεσμάτων.

```
...
case Result::success:  // έτοιμο αποτέλεσμα αποθηκευμένο
{
    if (nt_elapsed[row] > 0) // αν δεν έχουν περάσει οι ευκαιρίες το κρατάμε μόνιμα ενεργό
        nt_utilized[row] = true;
    pos = cur_cell->pos();
    return true;
}
case Result::fail:  // έτοιμο αποτέλεσμα αποθηκευμένο
{
    if (nt_elapsed[row] > 0) // αν δεν έχουν περάσει οι ευκαιρίες το κρατάμε μόνιμα ενεργό
        nt_utilized[row] = true;
    return false;
}
...
```

**Σχήμα 5.4:** Αξιοποιήση ενός κελιού για ένα μη τερματικό προτού λήξουν οι ευκαιρίες και μόνιμη ενεργοποίησή του

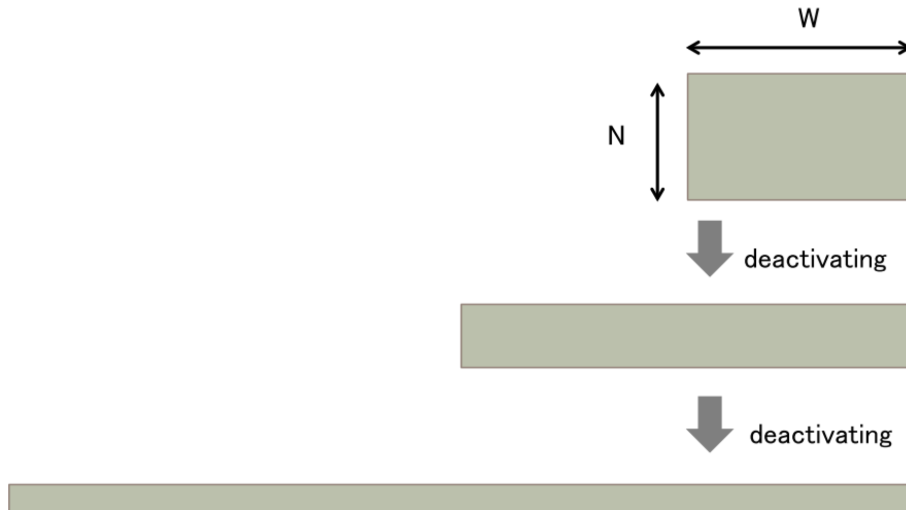
## 5.4 Elastic Packrat Parsing

Το elastic packrat parsing είναι μία μέθοδος που συνδυάζει τόσο το κυλιόμενο παράθυρο, όσο και τη δυναμική απενεργοποίηση μη τερματικών.

Το Σχήμα 5.5 απεικονίζει την ιδέα.

Για την υλοποίηση, χρησιμοποιούμε έναν μονοδιάστατο πίνακα $W * N$, όπου $W$ είναι το πλάτος του παραθύρου και $N$ ο αριθμός των ενεργών μη τερματικών. Οπότε, ένα ζεύγος $(position, nonterminal)$ αντιστοιχεί σε μία θέση στον μονοδιάστατο πίνακα. Ακολούθως, χρησιμοποιούμε έναν δείκτη κατακερματισμού (hasing-based index) για να εντοπίσουμε σε ποιο σημείο του μονοδιάστατου πίνακα θα αποθηκευτεί το ενδιάμεσο αποτέλεσμα, όπως στο Σχήμα 5.6. Φτιάχνουμε, αρχικά, ένα κλειδί μέσω της θέσης και του μη τερματικού, το οποίο μετά το ελαττώνουμε μέσω του modulo $(W * N)$.

Το πλεονέκτημα είναι ότι όταν απενεργοποιούμε ένα μη τερματικό, ο χώρος του στον πίνακα μπορεί να αξιοποιηθεί από άλλα μη τερματικά, διευρύνοντας ουσιαστικά το παράθυρο, όπως δείχνει




**Σχήμα 5.5:** Ελαστικό κυλιόμενο παράθυρο [Kura15]

```
1    long int key = (pos << shift) | row;
2    unsigned int index = hash(key) % (w * n);
3    ...
4    ElasticCell* cur_cell = &elastic_cells[index];    // πάρε το αντίστοιχο κελί
5    cur_cell->set_key(key);        // θέσε το κλειδί στο αντίστοιχο κελί
```

**Σχήμα 5.6:** Δημιουργία κλειδιού και δείκτη

το Σχήμα 5.5. Αυτή είναι και η ουσία της ελαστικότητας της μεθόδου. Στην ακραία περίπτωση που μόνο ένα μη τερματικό μείνει ενεργοποιημένο, τότε το παράθυρο έχει πρακτικά μέγεθος ίσο με $W * N$ και τη μορφή μίας γραμμής.

Το μειονέκτημα του hashing-based index είναι πως μπορεί να υπάρχουν συγκρούσεις (collisions) μεταξύ διαφορετικών κλειδιών, που όμως αντιστοιχίζονται στο ίδιο index. Αυτό αίρει την αυστηρότητα στην αποθήκευση ενδιάμεσων αποτελεσμάτων, αλλά τα οφέλη μίας τέτοιας απλής πρακτικής είναι μεγαλύτερα στη συνήθη περίπτωση.

Τέλος, η τεχνική αυτή δεν περιλαμβάνει ρητή κύλιση του παραθύρου, για πιο απλή υλοποίηση. Αντίθετα, η κύλιση προς τα δεξιά επιτυγχάνεται έμμεσα, καθώς τα νέα κλειδιά που αποθηκεύονται και αντιστοιχούν σε δεξιότερα σημεία της εισόδου, αντικαθιστούν τα παλιά. Αυτό προσεγγίζει την κύλιση, η οποία θα κόστιζε πολύ περισσότερο σε υλοποιήση για να γίνεται επ'ακριβώς.

Στο Κεφάλαιο 3 αναφέραμε ότι ο αλγόριθμος packrat καταναλώνει χώρο $O(NT * n)$, όπου $NT$ ο αριθμός των μη τερματικών της εκάστοτε γραμματικής, και $n$ το μήκος της εισόδου. Τώρα, φαίνεται ότι ο απαιτούμενος χώρος εκτέλεσης γίνεται $O(NT * W)$, όπου $W$ το μήκος του παραθύρου, το οποίο είναι σταθερό. Δηλαδή, πλέον, ο χώρος εκτέλεσης δεν είναι γραμμικός, αλλά σταθερός ως προς το μήκος της εισόδου, κάτι που αποτελεί μεγάλο πλεονέκτημα σε σχέση με τον κλασικό αλγόριθμο.

Κεφάλαιο 6

# Παράλληλο Packrat Parsing

Πέρα από την υλοποίηση του packrat parsing και του elastic packrat parsing, η βασική συνεισφορά που θέλουμε να κάνουμε στα πλαίσια της διπλωματικής, είναι να τροποποιήσουμε τη σειριακή έκδοση του packrat parser, ώστε να μπορεί να τρέξει αποδοτικότερα σε ένα πολυπύρηνο σύστημα.

Το πρόβλημα εύρεσης παραλληλισμού δεν είναι καθόλου τετριμμένο εν γένει. Ειδικά στην περίπτωση μας, θα πρέπει να εξετάσουμε διάφορα κομμάτια του αλγορίθμου που μπορούν να μοιραστούν μεταξύ των νημάτων, καθώς και πώς τα νήματα αυτά θα προσπελάσουν τη δομή δεδομένων με τα ενδιάμεσα αποτελέσματα. Εδώ, πρέπει να προσέξουμε ιδιαιτέρως δύο σημεία:

- Η πρόσβαση σε κάθε κελί της δομής δεδομένων πρέπει να γίνεται με *αμοιβαίο αποκλεισμό (mutual exlusion)* μεταξύ των νημάτων.

- Θέλουμε η *αναμονή (waiting)* των νημάτων να είναι όσο το δυνατόν μικρότερη.

Θα ξεκινήσουμε πρώτα με τον αλγόριθμο δυναμικού προγραμματισμού για το packrat parsing (DP packrat). Ακολούθως, θα επικεντρωθούμε στον αλγόριθμο με υπομνηματισμό.

## 6.1 Παράλληλος DP packrat

Η εκδοχή με τον από κάτω προς τα κάτω αναλυτή, μπορεί να παραλληλοποιηθεί σχετικά εύκολα (embarassingly parallel). Αναθέτοντας σε κάθε νήμα από μία στήλη του πίνακα για να υπολογίσει, μπορούμε να σπάσουμε εύκολα το αρχικό πρόβλημα σε επιμέρους υποπροβλήματα. Ο υπολογισμός γίνεται από κάτω προς τα πάνω, όπως φαίνεται στον Πίνακα 6.1.

Βέβαια, δεν αναμένουμε να είναι γραμμική η επιτάχυνση του αλγορίθμου καθώς τα αποτελέσματα της στήλης ενός νήματος μπορεί να εξαρτώνται από τις τιμές των κελιών που βρίσκονται στις στήλες άλλων νημάτων δεξιότερα. Έτσι, ένα νήμα θα πρέπει να κάνει τη ”δουλειά” κάποιου άλλου νήματος, οπότε ο φόρτος δεν μοιράζεται εξίσου.

Όπως και να ’χει, με δεδομένο ότι ο παράλληλος DP Packrat είναι πολύ αργός στη σειριακή περίπτωση, καθώς υπολογίζονται πολλά κελιά του πίνακα που δεν χρειάζονται, επικεντρωνόμαστε εφεξής στον καθοδικό packrat parser.

| | | | | | |
|---|---|---|---|---|---|
| ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| \| | \| | \| | \| | \| | \| |
| \| | \| | \| | \| | \| | \| |
| Thread 2 | Thread 1 | Thread 4 | Thread 3 | Thread 2 | Thread 1 |

**Πίνακας 6.1:** Μοίρασμα του υπολογισμού των κελιών για τον DP Packrat

## 6.2 Παράλληλο Καθοδικό Packrat Parsing

### 6.2.1 Παραλληλοποίηση της Διατεταγμένης Επιλογής

Η πρώτη ιδέα στη βελτίωση της απόδοσης του καθοδικού Packrat Parser είναι να προσπαθήσουμε να βρούμε κάποιο κομμάτι του αλγορίθμου το οποίο μπορεί να εκτελεστεί από πολλά νήματα ταυτόχρονα, ώστε να μοιραστεί ο φόρτος εργασίας. Έτσι, αυτό το κομμάτι του αλγορίθμου θα μπορεί να εκτελεστεί ταχύτερα, οδηγώντας και σε ταχύτερη εκτέλεση συνολικά.

Επαναλαμβάνουμε ότι ο αναλυτής μας ”επισκέπτεται” κάθε στοιχείο της γραμματικής για να την αναλύσει. Πρώτα, θα πάει στο αρχικό μη τερματικό της γραμματικής. Ακολούθως, θα πάει στη σύνθετη έκφραση που αντιστοιχεί σε αυτό το μη τερματικό. Μία ιδέα, λοιπόν, θα ήταν να προσπαθήσουμε να παραλληλοποιήσουμε τη συντακτική ανάλυση των σύνθετων εκφράσεων.

Προφανώς, για μία ακολουθία από εκφράσεις:

$$E \leftarrow E_1 E_2 E_3 \tag{6.1}$$

θα ήταν δύσκολο να αναθέσουμε μία υποέκφραση $E_i$ σε κάθε νήμα, καθώς πρέπει πρώτα να προσπαθήσουμε να αναλύσουμε την $E_1$ και, αν πετύχει, να αναλύσουμε την $E_2$ συνεχίζοντας από το σημείο της εισόδου που μας άφησε η $E_1$. Επειδή, δεν ξέρουμε εκ των προτέρων ποιο θα είναι αυτό, δεν μπορούμε έτσι απλά να δώσουμε την $E_2$ να την υπολογίσει άλλο νήμα.

Ωστόσο, στη διατεταγμένη επιλογή:

$$E \leftarrow E_1/E_2/E_3 \tag{6.2}$$

οι υποεκφράσεις αναλύονται ξεκινώντας από το ίδιο σημείο της εισόδου (αν αποτύχει η $E_1$, τότε ξαναπροσπαθεί από το ίδιο σημείο η $E_2$ κλπ.). Θεωρητικά, λοιπόν, θα μπορούσαμε να τις αναθέσουμε σε ξεχωριστά νήματα.

Επικεντρωνόμαστε στο Σχήμα 6.1.

Η ανάθεση των υποεκφράσεων σε ξεχωριστά νήματα φαίνεται στις γραμμές 8-16. Το κύριο νήμα καλεί νήματα-εργάτες για να αναλύσουν κάθε υποέκφραση χωριστά ξεκινώντας από την τρέχουσα θέση (γραμμή 11). Μετά κρατάει για κάθε νήμα το αποτέλεσμα της συντακτικής ανάλυσης (επιτυχία ή αποτυχία) (γραμμή 12), καθώς και το επόμενο σημείο της εισόδου, εφόσον πέτυχε η ανάλυση (γραμμή 13). Στις γραμμές 19-30 το κύριο νήμα περιμένει τα νήματα με τη σειρά να ολοκληρώσουν.

Ωστόσο, υπάρχει ένα σημείο που χρειάζεται προσοχή. Αν το νήμα που ανέλυσε την $E_2$, επιστρέψει επιτυχώς, πρέπει πάλι να περιμένουμε το νήμα που αναλύει την $E_1$, καθώς η υποέκφραση αυτή έχει προτεραιότητα. Συνεπώς, η κύρια διεργασία που έχει αναλάβει την ανάλυση της έκφρασης $E$ θα πρέπει να καλέσει ξεχωριστά νήματα-εργάτες για κάθε υποέκφραση, αλλά να τηρήσει την ιεραρχία:

- Πρώτα περιμένει να ολοκληρώσει το νήμα της $E_1$.
- Αν επιτύχει, τότε στέλνει σήμα στα υπόλοιπα νήματα να επιστρέψουν.
- Αλλιώς, περιμένει να τελειώσει το δεύτερο νήμα.
- Η διαδικασία συνεχίζεται είτε μέχρις ότου κάποιο νήμα επιστρέψει επιτυχώς (οπότε τερματίζονται τα επόμενα στην ιεραρχία), είτε όταν επιστρέψουν όλα τα νήματα ανεπιτυχώς.

Αν όλα επιστρέψουν ανεπιτυχώς, τότε και το κύριο νήμα που τα κάλεσε επιστρέφει ανεπιτυχώς και θέτει το δείκτη στην είσοδο όπως ήταν στην αρχή (reset) (γρ. 31-32). Αλλιώς, το πρώτο νήμα στην ιεραρχία που πέτυχε καθορίζει από ποιο σημείο της εισόδου θα συνεχίσει η ανάλυση (γρ. 23).

Για να έχει νόημα η παραπάνω παραλληλοποίηση πρέπει να διασφαλίζουμε ότι όταν ένα νήμα επιστρέφει επιτυχώς, τα επόμενα στην ιεραρχία θα τερματίζονται όσο το δυνατόν γρηγορότερα. Για το σκοπό αυτό, υπάρχει μία καθολική μεταβλητή $finished_rank$, η οποία κρατάει το νούμερο του νήματος που ολοκλήρωσε επιτυχώς, αρκεί να μην υπάρχουν νήματα πιο πάνω στην ιεραρχία που

```
finished_rank.store(-1);

int results[exprs.size()];
int positions[exprs.size()];
std::vector<std::thread> threads;

auto i = 0;
for (auto& expr : exprs) {
    threads.emplace_back([&, expr, i, this]()
    {
        SimpleWorker sw{in, peg, cells, pos, i};
        results[i] = expr->accept(sw);
        positions[i] = sw.cur_pos();
    });
    i++;
}

for (auto j = 0; j < i; ++j) {
    threads[j].join();
    if (results[j]) {
        finished_rank.store(j);
        pos = positions[j];
        for (auto k = j + 1; k < i; ++k) {
            threads[k].join();
        }
        return true;
    }
}
pos = orig_pos;
return false;
```

**Σχήμα 6.1:** Δημιουργία και τερματισμός νημάτων με βάση την ιεραρχία για κάθε υποέκφραση

επίσης έχουν επιστρέψει επιτυχώς. Για λόγους συγχρονισμού, η μεταβλητή αυτή είναι ατομική. Στην αρχή τίθεται ως $-1$ από το κύριο νήμα (γρ. 1).

Θεωρούμε ότι σε κάθε νήμα-εργάτη έχει δοθεί ένα μοναδικό νούμερο $rank$, όπου ψηλότερα στην ιεραρχία είναι το νούμερο 0 (γρ. 7, 11). Το κύριο νήμα δίνει το $rank$ στα παιδιά του και τα περιμένει μετά ένα ένα να επιστρέψουν βάσει ιεραρχίας. Αν κάποιο επιστρέψει επιτυχώς, το κύριο νήμα θέτει τη μεταβλητή $finished_rank$ με το rank του νήματος (γρ. 22).

Άρα, σε κάθε αναδρομική κλήση που κάνει το νήμα-εργάτης προτού επισκεφθεί κάθε έκφραση, κάνει τον έλεγχο του Σχήματος 6.2 για πρόωρο τερματισμό. Εάν το $finished_rank$ έχει τεθεί από νήμα μεγαλύτερης ιεραρχίας (μικρότερο $rank$), τότε το τρέχον νήμα πρέπει να επιστρέψει αμέσως (γρ. 2-3).

```
auto fr = finished_rank.load();
if (fr >= 0 && fr < rank) {
    return false;
}
```

**Σχήμα 6.2:** Έλεγχος της καθολικής μεταβλητής για πρόωρο τερματισμό

Βέβαια, θα υπάρχουν και περιπτώσεις όπου η διατεταγμένη επιλογή θα περιλαμβάνει πολλές εναλλακτικές, ιδιαίτερα στα μη τερματικά που αφορούν τη λεκτική ανάλυση του προγράμματος, όπως φαίνεται στο Σχήμα 6.3. Το $HexDigit$ αντιπροσωπεύει τα δεκαεξαδικά ψηφία και η συντακτική ανάλυσή του ισοδυναμεί πρακτικά με την αναγνώριση της κανονικής έκφρασης $[a-fA-FDigit]$, όπου $Digit$ είναι η κανονική έκφραση για τα ψηφία. Συνήθως, όταν υπάρχουν πολλές εναλλακτικές στη διατεταγμένη επιλογή, αυτές αποτελούν τερματικά (κυρίως για τους κανόνες που αφορούν λεκτική ανάλυση), τα οποία μπορούν να αναλυθούν άμεσα. Στην περίπτωση αυτή, μάλλον δεν είναι συμφέρον για το κύριο νήμα να καλέσει ένα νήμα για την ανάλυση της κάθε υποέκφρασης, καθώς το κόστος τόσο της κλήσης πολλών νημάτων, όσο και της αναμονής να τελειώσουν (join), θα είναι μεγάλο.

```
HexDigit <- 'a' / 'b' / 'c' / 'd' / 'e' / 'f' /
            'A' / 'B' / 'C' / 'D' / 'E' / 'F' / Digit
```

**Σχήμα 6.3:** Κανόνας με πολλές εναλλακτικές στη διατεταγμένη επιλογή

Τότε, συμφέρει καλύτερα το κύριο νήμα να αναλύσει τη διατεταγμένη επιλογή μόνο του, παρά να καλέσει άλλα νήματα-εργάτες. Ο κώδικας φαίνεται στο Σχήμα 6.4. Το $expr_limit$ είναι το όριο στον αριθμό των υποεκφράσεων πάνω από το οποίο δεν θα κληθούν νέα νήματα, αλλά η έκφραση θα αναλυθεί επί τόπου.

```
1 if (exprs.size() > expr_limit) {
2     for (auto expr : exprs) {
3         pos = orig_pos;
4         if (expr->accept(*this))
5             return true;
6     }
7     pos = orig_pos;
8     return false;
9 }
```

**Σχήμα 6.4:** Έλεγχος για την εκτέλεση ή μη της διατεταγμένης επιλογής με βάση το μέγεθος

Τέλος, πρέπει να αναφέρουμε ότι ο πίνακας των ενδιάμεσων αποτελεσμάτων είναι, πλέον, κοινόχρηστος από τα νήματα. Οπότε, για να αποφευχθούν συνθήκες ανταγωνισμού για τα κελιά του, πρέπει αυτά να προστατεύονται από κλειδώματα. Ένας απλός και αποτελεσματικός τρόπος είναι κάθε κελί να έχει το κλειδί του, οπότε για να το υπολογίσει κάποιος θα πρέπει να το κλειδώσει (Σχ. 6.5).

```
1 cur_cell->lock();
2 cur_cell->set_res(Result::pending);
3 cur_cell->unlock();
```

**Σχήμα 6.5:** Κλείδωμα του κελιού

Συγκεκριμένα το ”μαρκάρει” ως pending, οπότε αν κάποιο άλλο νήμα πάει να ζητήσει το αποτέλεσμά του, θα πρέπει να περιμένει μέχρι η κατάσταση pending να αλλάξει (Σχ. 6.6):

```cpp
case Result::pending:
{
    while (cur_cell->res() == Result::pending)
        std::this_thread::sleep_for(std::chrono::milliseconds(0));
    ...
}
```

**Σχήμα 6.6:** Αναμονή για υπολογισμό του κελιού από άλλο νήμα

### 6.2.2 Αναδρομική δημιουργία νημάτων

Στην προηγούμενη υποενότητα εξετάσαμε τη σχετικά απλή περίπτωση όπου ένα νήμα (master) καλούσε επιμέρους νήματα (workers), για να επιταχύνει τη συντακτική ανάλυση στην περίπτωση της διατεταγμένης επιλογής. Ωστόσο, αυτά τα νήματα-εργάτες λειτουργούσαν σειριακά. Δηλαδή, αν συναντούσαν με τη σειρά τους μία σύνθετη έκφραση διατεταγμένης επιλογής, δεν θα δημιουργούσαν καινούρια νήματα, αλλά θα ανέλυαν με τη σειρά τις υποεκφράσεις.

Τώρα, θα επιχειρήσουμε να επεκτείνουμε τη λογική μας, ώστε κάθε νήμα-εργάτης να μπορεί με τη σειρά του να καλέσει και άλλα νήματα-εργάτες. Αυτό εισάγει δύο (τουλάχιστον) δυσκολίες:

- Μία καθολική μεταβλητή συγχρονισμού δεν αρκεί, αλλά πρέπει να εισαχθεί μία μεταβλητή για κάθε νήμα-εργάτη, ώστε να ελέγχει τα νήματα που δημιουργεί στο επόμενο επίπεδο.

- Πλέον, δύναται να δημιουργηθούν πάρα πολλά νήματα, οπότε πρέπει να καθοριστούν αντίστοιχοι περιορισμοί, που θα φράσσουν την ανεξέλεγκτη δημιουργία νημάτων.

Όπως και πριν, κάθε γονικό νήμα μόλις καλέσει τα παιδιά-νήματα τα περιμένει ένα ένα να τελειώσουν βάσει ιεραρχίας. Ωστόσο, τώρα κάθε νήμα-γονέας έχει τη δική του $finished_rank$ ατομική μεταβλητή, όπως φαίνεται στο Σχήμα 6.7. Μέσω αυτής ελέγχει τα νήματα-παιδιά (γρ. 6). Επιπλέον, για να μπορεί να το ελέγχει και αυτό αντίστοιχα ο δικός του γονέας, διαθέτει και ένα δείκτη προς την ατομική μεταβλητή αυτού (γρ. 7).

```cpp
class SimpleWorker: public SerialPackrat {
    int expr_limit;
    int cur_tree_depth;
    int max_tree_depth;
    int rank;
    std::atomic<int> finished_rank{-1};
    std::atomic<int>* parent_finished_rank;
public:
    ...
};
```

**Σχήμα 6.7:** Ορισμός της κλάσης του νήματος-εργάτη

Στην προηγούμενη υποενότητα εισάγαμε την παράμετρο $expr_limit$ για να καθορίζουμε μέχρι πόσα παιδιά θα καλεί το αρχικό νήμα. Αν η σύνθετη έκφραση είχε περισσότερα από $expr_limit$ παιδιά, τότε το νήμα την ανέλυε σειρακά. Έτσι και τώρα, κάθε νήμα έχει ένα όριο στο πόσα παιδιά θα μπορεί να καλέσει (Σχ. 6.8, γρ. 11).

Επιπλέον, τώρα χρειαζόμαστε και μία παράμετρο που θα φράσσει το βάθος του δέντρου (fork-join tree). Η μεταβλητή $max_tree_depth$, λοιπόν, θα καθορίζεται άπαξ στο αρχικό νήμα, το οποίο θα τη στέλνει στα παιδιά-νήματα. Κάθε νήμα θα έχει μία μεταβλητή $cur_tree_depth$ που θα δείχνει το τρέχον βάθος, όπως φαίνεται στον ορισμό της κλάσης του. Όταν καλεί ένα νήμα-παιδί, θα του

δίνει $cur_tree_depth + 1$ ως βάθος (γρ. 12). Το νήμα-γονέας δίνει στα παιδιά και ένα δείκτη προς την ατομική μεταβλητή του, ώστε να μπορεί να τα ειδοποιήσει να τερματίσουν (γρ. 12). Πάντα το νήμα πρέπει να ξαναθέτει την ατομική μεταβλητή του σε -1, διότι μπορεί να επισκεπτεί πάνω από μία διατεταγμένες επιλογές (γρ. 1).

```
1  finished_rank.store(-1);
2
3  int results[exprs.size()];
4  int positions[exprs.size()];
5  std::vector<std::thread> threads;
6
7  auto i = 0;
8  for (auto& expr : exprs) {
9      threads.emplace_back([&, expr, i]()
10     {
11         SimpleWorker sw{in, peg, cells, pos, expr_limit,
12                         cur_tree_depth + 1, max_tree_depth, i, &finished_rank};
13         results[i] = expr->accept(sw);
14         positions[i] = sw.cur_pos();
15     });
16     i++;
17 }
```

**Σχήμα 6.8:** Το νήμα-εργάτης δημιουργεί νήματα-εργάτες

Το νήμα-γονέας με το πρώτο παιδί που επιστρέφει επιτυχώς βάζει το $rank$ του παιδιού στο $finished_rank$ του (Σχήμα 6.9, γρ. 4).

```
1  for (auto j = 0; j < i; ++j) {
2      threads[j].join();
3      if (results[j]) {
4          finished_rank.store(j);
5          pos = positions[j];
6          for (auto k = j + 1; k < i; ++k) {
7              threads[k].join();
8          }
9          return true;
10     }
11 }
12 pos = orig_pos;
13 return false;
```

**Σχήμα 6.9:** Αναμονή και τερματισμός των νημάτων

Οπότε, τα επόμενα παιδιά μετά θα διαβάσουν κάποια στιγμή ότι το $parent_finished_rank$ που αντιστοιχεί σε αυτά έχει τεθεί με νούμερο μεγαλύτερης ιεραρχίας, και θα επιστρέψουν ανεπιτυχώς. Ο έλεγχος για πρόωρο τερματισμό γίνεται χρησιμοποιώντας το δείκτη προς την ατομική μεταβλητή του γονέα (Σχήμα 6.10).

```
auto fr = parent_finished_rank->load();
if (fr >= 0 && fr < rank) {
    return false;
}
```

**Σχήμα 6.10:** Έλεγχος της μεταβλητής του γονέα για πρόωρο τερματισμό

## Κεφάλαιο 7

# Πειραματικά Αποτελέσματα

Θα αξιολογήσουμε τους αλγορίθμους μας με 3 αρχεία από τον πηγαίο κώδικα της Java [1]. Το μέγεθός τους είναι μέτριο προς μεγάλο, ώστε να μπορέσουν να φανούν οι διαφορές στους χρόνους εκτέλεσης των αλγορίθμων. Τα μεγέθη τους σε byte είναι:

- *Arrays.java* - 116K
- *BigDecimal.java* - 140K
- *Throwable.java* - 28K

Για τον παράλληλο αλγόριθμο, επικεντρωνόμαστε στην υλοποίηση με αναδρομική δημιουργία νημάτων της προηγούμενης ενότητας.

Οι μετρήσεις έγιναν με το *std::chrono::high_resolution_clock* της STL. Το μηχάνημα που χρησιμοποιήθηκε διαθέτει 12 X 2600 MHz CPUs και μνήμες cache: L1 Data 32 KiB (x6), L1 Instruction 32 KiB (x6), L2 Unified 256 KiB (x6) και L3 Unified 12288 KiB (x1).

## 7.1 Packrat με ελαστικό κυλιόμενο παράθυρο

Αρχικά, εκτελούμε διαδοχικά τον αλγόριθμο packrat με ελαστικό κυλιόμενο παράθυρο για διάφορα μήκη παραθύρου (w) και κατώφλια απενεργοποίησης μη τερματικών (thres). Ακολουθούμε ενδεικτικά τα όρια που προτείνονται στο [Kura15]. Τα αποτελέσματα που παίρνουμε είναι τα εξής, χρωματίζοντας ενδεικτικά όπου κρίνεται χρήσιμο:

| thres \ w | 256 | 512 | 1024 |
|---|---|---|---|
| 0 | 396 | 401 | 415 |
| 16 | 378 | 378 | 381 |
| 32 | 380 | 381 | 379 |
| 48 | 379 | 379 | 374 |

**Πίνακας 7.1:** Elastic Packrat - Arrays.java

Όπως αναμενόταν με βάση το [Kura15], το 32 αποτελεί αποτελεσματικό κατώφλι απενεργοποίησης των μη τερματικών. Επιπλέον, τα παράθυρα 256 και 512 θα έπρεπε να παρουσιάζουν την καλύτερη επίδοση, με μικρή πτώση της απόδοσης στα 1024, όπως επαληθεύεται. Στο μικρότερο αρχείο, το Throwable, δεν παρουσιάζονται σημαντικές μεταβολές στους χρόνους εκτέλεσης.

Για τις τελικές συγκρίσεις κρατάμε μήκος παραθύρου 256 και κατώφλι απενεργοποίησης ίσο με 32.

[1] `http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java`

| thres \ w | 256 | 512 | 1024 |
|---|---|---|---|
| 0 | 341 | 356 | 358 |
| 16 | 334 | 337 | 336 |
| 32 | 329 | 329 | 330 |
| 48 | 333 | 334 | 332 |

**Πίνακας 7.2:** Elastic Packrat - BigDecimal.java

| thres \ w | 256 | 512 | 1024 |
|---|---|---|---|
| 0 | 53 | 51 | 55 |
| 16 | 50 | 48 | 51 |
| 32 | 51 | 47 | 51 |
| 48 | 52 | 47 | 51 |

**Πίνακας 7.3:** Elastic Packrat - Throwable.java

## 7.2 Παράλληλο packrat parsing

Τώρα, εκτελούμε διαδοχικά τον παράλληλο αλγόριθμο και για τα τρία αρχεία, μεταβάλλοντας τις παραμέτρους: 1) του μέγιστου αριθμού υποεκφράσεων, πάνω από τον οποίο ένα νήμα δεν θα καλέσει άλλα νήματα, αλλά θα εκτελέσει την ανάλυση σειριακά (expr limit) και 2) του μέγιστου βάθους του δέντρου κλήσεων (max depth), για το fork-join tree. Κάθε στιγμή, δηλαδή, ο μέγιστος αριθμός νημάτων-εργατών που μπορεί να υπάρχουν είναι:

$$\frac{expr_limit^{max_depth+1} - 1}{expr_limit - 1} \tag{7.1}$$

Τα αποτελέσματα των μετρήσεων είναι τα εξής:

| max depth \ expr limit | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| 1 | 434 | 432 | 430 | 437 |
| 2 | 1278 | 451 | 446 | 446 |

**Πίνακας 7.4:** Παράλληλο Packrat - Arrays.java

| max depth \ expr limit | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| 1 | 368 | 369 | 367 | 369 |
| 2 | 2570 | 380 | 379 | 385 |

**Πίνακας 7.5:** Παράλληλο Packrat - BigDecimal.java

| max depth \ expr limit | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| 1 | 94 | 60 | 62 | 60 |
| 2 | 324 | 76 | 74 | 109 |

**Πίνακας 7.6:** Παράλληλο Packrat - Throwable.java

Παρατηρούμε ότι οι καλύτεροι χρόνοι προκύπτουν για μέγιστο βάθος ίσο με 1 και για όριο υποεκφράσεων 2 έως 6. Πρακτικά, δηλαδή, να υπάρχουν ενεργά 3 έως 7 νήματα κατά μέγιστο. Για μέγιστο βάθος 2 τα αποτελέσματα χειροτερεύουν αισθητά, ιδιαίτερα για expr_limit ίσο με 2. Για μεγαλύτερο βάθος τα αποτελέσματα ήταν πολύ χειρότερα, γι' αυτό δεν τα παραθέτουμε.

Σχετικά με τις τελικές συγκρίσεις, κρατάμε μέγιστο βάθος ίσο με 1 και όριο υποεκφράσεων ίσο με 4.

## 7.3 Τελικά Αποτελέσματα

Για τους συνδυασμούς παραμέτρων που επιλέχθηκαν στις προηγούμενες ενότητες, παρουσιάζουμε τις τελικές συγκρίσεις για τους τρεις αλγορίθμους και στα τρία αρχεία στον Πίνακα 7.7:

| Χρόνοι εκτέλεσης (ms) | | | |
|---|---|---|---|
| Αλγόριθμος Packrat | Arrays - 116K | BigDecimal - 140K | Throwable - 28K |
| Κλασικός | 404 | 350 | 46 |
| Elastic (256, 32) | 380 | 329 | 51 |
| Παράλληλος (1, 4) | 432 | 369 | 62 |

**Πίνακας 7.7:** Τελικά αποτελέσματα για τους τρεις αλγορίθμους

Είναι φανερό ότι ο αδιαφιλονίκητος νικητής στους χρόνους εκτέλεσης είναι ο packrat με ελαστικό κυλιόμενο παράθυρο, δεδομένου ότι στο Throwable οι διαφορές είναι μικρές. Μάλιστα, πετυχαίνει και την καλύτερη επίδοση σε μνήμη αφού, όπως είπαμε, καταναλώνει σταθερή μνήμη:

$$O(NT * W) \tag{7.2}$$

ενώ οι άλλοι δύο αλγόριθμοι απαιτούν μνήμη γραμμική ως προς το μήκος της εισόδου.

# Κεφάλαιο 8

# Συμπεράσματα

Οι Parsing Expression Grammars αποτελούν διαισθητικά ένα κατάλληλο εργαλείο προσδιορισμού γραμματικών, ιδιαίτερα αν συγκριθούν με τις γραμματικές χωρίς συμφραζόμενα. Αρχικά, ο σχεδιαστής της γραμματικής είναι ευκολότερο να σκέφτεται πώς αναλύεται μία δοσμένη συμβολοσειρά στα συστατικά της, σε σχέση με το πώς θα γεννηθεί η συμβολοσειρά μέσα από τους κανόνες της γραμματικής, στο πνεύμα των Context Free γραμματικών. Επιπλέον, ο ίδιος ο ορισμός των PEGs ορίζει απευθείας και τον αντίστοιχο συντακτικό αναλυτή της γραμματικής αυτής.

Το packrat parsing αποτελεί μία μέθοδο για τη συντακτική ανάλυση των PEGs, σε γραμμικό χρόνο. Καταφέρνει να συγκεράσει την κομψότητα συντακτικών αναλυτών αναδρομικής κατάβασης με την αποδοτικότητα που προσφέρουν οι προβλέποντες συντακτικοί αναλυτές. Ωστόσο, το κόστος της μνήμης σε ορισμένες εφαρμογές είναι απαγορευτικό.

Ένας γεννήτορας συντακτικών αναλυτών είναι ένα βολικό εργαλείο για να φτιάχνουμε αυτόματα αναλυτές για μία γραμματική PEG που έχουμε ορίσει τυπικά. Ο γεννήτορας παίρνει ως είσοδο μία τυπική περιγραφή της PEG και δίνει ως έξοδο ένα στιγμιότυπο της γραμματικής αυτής που μπορεί να αναλύσει ένας packrat parser. Ο γεννήτορας πρακτικά δημιουργεί γραμματικές αλλά, όπως είπαμε, η κατασκευή μίας PEG συνεπάγεται και την κατασκευή του αντίστοιχου συντακτικού αναλυτή. Επομένως, ο γεννήτορας κατασκευάζει συντακτικούς αναλυτές.

Ιδιαίτερα, λοιπόν, αν ο προκύπτων συντακτικός αναλυτής είναι και αποδοτικός στο χρόνο και στο χώρο, ένα τέτοιο εργαλείο θα ήταν βολικό για έναν προγραμματιστή ο οποίος πειραματίζεται με μία δική του γραμματική για κάποιον ειδικό σκοπό. Διότι, έχει τη δυνατότητα να ορίσει και να αλλάζει εύκολα τον ορισμό της γραμματικής του, αλλά και να παίρνει έναν γρήγορο συντακτικό αναλυτή για τις εφαρμογές του.

Από το προηγούμενο κεφάλαιο φαίνεται μάλλον ότι το packrat parsing με ελαστικό κυλιόμενο παράθυρο αποτελεί την καλύτερη βελτίωση του packrat, καθώς χρειάζεται αισθητά λιγότερο χρόνο εκτέλεσης, αλλά και σταθερή μνήμη, εξοβελίζοντας ένα κυρίαρχο μειονέκτημα του κλασικού αλγορίμου.

Σχετικά με τον παράλληλο αλγόριθμο, δεν παρατηρείται επιτάχυνση (speedup) σε σχέση με τη σειριακή περίπτωση. Ενώ υπάρχει βελτίωση για expr_limit από 2 σε 4 και, ίσως, σε 6 (που ισοδυναμεί εν δυνάμει με αντιστοίχως παραπάνω νήματα), δεν υπάρχει βελτίωση σε σχέση με τη σειριακή περίπτωση. Από την άλλη, μέγιστο βάθος του δέντρου των κλήσεων μεγαλύτερο του 1 μάλλον δυσχεραίνει την κατάσταση, καθώς αφήνει χώρο για παραπάνω νήματα από όσα μπορεί να διαθέσει ταυτόχρονα το μηχάνημα, οπότε αρκετά νήματα μένουν ανενεργά.

Στη ρίζα του, το πρόβλημα φαίνεται να είναι πως η πράξη της διατεταγμένης επιλογής, παρόλο που θεωρητικά αφήνει χώρο για παράλληλη εκτέλεση, δεν συμπεριφέρεται τόσο καλά στην πράξη. Συγκεκριμένα, φαίνεται ότι το κόστος της δημιουργίας και της αναμονής των νημάτων είναι μεγαλύτερο από το κέρδος του διαμοιρασμού του φόρτου εργασίας. Ειδικά στην αναμονή, όταν τα εναπομείναντα νήματα πρέπει να τερματιστούν, αυτό δεν μπορεί να γίνει άμεσα, αλλά πρέπει να τους σταλεί σήμα και να το λάβουν σε κάποιο σημείο του κώδικα. Αυτό συνεπάγεται ότι πιθανώς εκτελούνται και υπολογισμοί ενδιάμεσων αποτελεσμάτων από νήματα που δεν έχουν προλάβει να δουν το σήμα τερματισμού, οι οποίοι ούτε ωφελούν, αλλά αντίθετα καθυστερούν το αρχικό νήμα-γονέα.

Ακόμη, ενδέχεται στη γραμματική που εξετάσαμε στα πειράματα να είναι συχνό φαινόμενο στη διατεταγμένη επιλογή, να πετυχαίνει κάποια από τις πρώτες επιλογές (αν όχι η πρώτη). Σε αυτήν την

περίπτωση, η δημιουργία νημάτων και για τις επόμενες υποεκφράσεις είναι μάταια, ενώ προσθέτει και καθυστέρηση (overhead) στον συνολικό αλγόριθμο, οπότε η εκτέλεση γίνεται χειρότερη και από τη σειριακή.

Στο [Fowl09] περιγράφεται μία απόπειρα για παραλληλοποίηση του αλγορίθμου στη διάσταση της εισόδου. Δηλαδή, χωρίζεται η είσοδος σε μπλοκς που μοιράζονται μεταξύ νημάτων-εργατών, οι οποίοι υπολογίζουν με κάποια ευριστική μέθοδο διάφορα κελιά μέσα σε αυτά τα μπλοκς. Στο πρώτο μπλοκ, όμως, υπάρχει ένα κύριο νήμα που ξεκινάει από την αρχή να υπολογίζει κελιά με τον σειριακό αλγόριθμο. Ωστόσο, μόλις φτάσει στο σύνορο του μπλοκ με το γειτονικό νήμα-εργάτη, του "παραδίδει" μέχρι εκείνο το σημείο την υποέκφραση την οποία έχει καταφέρει να αναλύσει. Οπότε, το επόμενο νήμα, έχοντας υπολογίσει κάποια αποτελέσματα ήδη, ξεκινάει από την υποέκφραση και αρχίζει να την αναλύει με το σειριακό αλγόριθμο, ελπίζοντας ότι κάποια από τα νήματα που υπολόγισε ως τότε θα χρησιμεύσουν. Στη συνέχεια παραδίδει τη δική του υποέκφραση (όσο την έχει προχωρήσει) στο νήμα-εργάτη του επόμενου μπλοκ και ούτω καθεξής. Για μία απλή γραμματική (τη γραμματική που περιγράφει τις PEGs) ο ισχυρισμός είναι ότι επετεύχθη επιτάχυνση περίπου κατά 2.5, αν και η υλοποίηση ήταν μακράν πιο πολύπλοκη από τη δική μας ιδέα.

Πάντως, για τη βελτίωση με τη βοήθεια πολλών πυρήνων, στην περίπτωση που πρέπει να αναλυθεί συντακτικά ένα μεγάλο σύνολο αρχείων (π.χ. ενός μεγάλου project), θα μπορούσε να χρησιμοποιηθεί και ένας πυρήνας ανά αρχείο για συντακτική ανάλυση, ώστε να βελτιωθεί ο χρόνος εκτέλεσης. Όπως και να 'χει, συνολικά, για τη βελτίωση της επίδοσης του packrat μάλλον το ελαστικό κυλιόμενο παράθυρο αποτελεί την καλύτερη (και απλούστερη) επιλογή.

# Βιβλιογραφία


[Aho06] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[Ford02a] Bryan Ford, "Packrat Parsing : a Practical Linear-Time Algorithm with Backtracking by", *Proceedings of the International Conference on Functional Programming ICFP 2002*, 2002.

[Ford02b] Bryan Ford, "Packrat parsing: Simple, powerful, lazy, linear time - Functional pearl", *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, pp. 36–47, 2002.

[Ford04] Bryan Ford, "Parsing expression grammars: A recognition-based syntactic foundation", *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, vol. 31, pp. 111–122, 2004.

[Fowl09] S Fowler and J Paul, "Parallel Parsing: The Earley and Packrat Algorithms", 2009.

[Inst06] Instil, "Language Generator by Instil", `https://sourceforge.net/projects/instil-lang/`, 2006.

[Kura15] Kimio Kuramitsu, "Packrat parsing with elastic sliding window", *Journal of Information Processing*, vol. 23, no. 4, pp. 505–512, 2015.

[Moor00] Robert C. Moore, "Removing Left Recursion from Context-Free Grammars", in *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*, NAACL 2000, p. 249–255, USA, 2000, Association for Computational Linguistics.

[Wart08] Alessandro Warth, James R. Douglass and Todd Millstein, "Packrat Parsers Can Support Left Recursion", in *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '08, p. 103–110, New York, NY, USA, 2008, Association for Computing Machinery.

# Παράρτημα A

# Μία PEG περιγράφει τυπικά την ίδια τη σύνταξή της

```
# Hierarchical syntax

Grammar    <- Spacing Definition+ EndOfFile                   # Type 0
Definition <- Identifier LEFTARROW Expression                 # Type 1
Expression <- Sequence (SLASH Sequence)*                      # Type 2
Sequence   <- Prefix*                                         # Type 3
Prefix     <- (AND / NOT)? Suffix                             # Type 4
Suffix     <- Primary (QUESTION / STAR / PLUS)?               # Type 5
Primary    <- Identifier !LEFTARROW                           # Type 6
            / OPEN Expression CLOSE
            / Literal / DOT

# Lexical syntax

Identifier <- IdentifierStart IdentifierRest* Spacing
IdentifierStart <- ’a’ / ’b’ / ’c’ / ’d’ / ’e’ / ’f’ / ’g’ / ’h’ /
                   ’i’ / ’j’ / ’k’ / ’l’ / ’m’ / ’n’ / ’o’ / ’p’ /
                   ’q’ / ’r’ / ’s’ / ’t’ / ’u’ / ’v’ / ’w’ / ’x’ / ’y’ / ’z’ /
                   ’A’ / ’B’ / ’C’ / ’D’ / ’E’ / ’F’ / ’G’ / ’H’ /
                   ’I’ / ’J’ / ’K’ / ’L’ / ’M’ / ’N’ / ’O’ / ’P’ /
                   ’Q’ / ’R’ / ’S’ / ’T’ / ’U’ / ’V’ / ’W’ / ’X’ / ’Y’ / ’Z’ / ’_’

IdentifierRest <- ’a’ / ’b’ / ’c’ / ’d’ / ’e’ / ’f’ / ’g’ / ’h’ /
                  ’i’ / ’j’ / ’k’ / ’l’ / ’m’ / ’n’ / ’o’ / ’p’ /
                  ’q’ / ’r’ / ’s’ / ’t’ / ’u’ / ’v’ / ’w’ / ’x’ / ’y’ / ’z’ /
                  ’A’ / ’B’ / ’C’ / ’D’ / ’E’ / ’F’ / ’G’ / ’H’ /
                  ’I’ / ’J’ / ’K’ / ’L’ / ’M’ / ’N’ / ’O’ / ’P’ /
                  ’Q’ / ’R’ / ’S’ / ’T’ / ’U’ / ’V’ / ’W’ / ’X’ / ’Y’ / ’Z’ / ’_’ /
                  ’0’ / ’1’ / ’2’ / ’3’ / ’4’ / ’5’ / ’6’ / ’7’ / ’8’ / ’9’

Literal    <- ″’″ (!″’″ Character)* ″’″ Spacing               # Type 8
            / ’″’ (!’″’ Character)* ’″’ Spacing

Character  <- (’\\’ (’n’ / ’r’ / ’t’ / ’\\’ / ″’″ / ’″’ / UnicodeEscape)) / (!’\\’ .)

UnicodeEscape <- ’u’ UnicodeElement UnicodeElement UnicodeElement UnicodeElement

UnicodeElement <- ’0’ / ’1’ / ’2’/ ’3’ / ’4’ / ’5’ / ’6’ / ’7’ / ’8’ / ’9’ /
                  ’A’ / ’B’ / ’C’ / ’D’/ ’E’ / ’F’ / ’a’ / ’b’ / ’c’ / ’d’/ ’e’ / ’f’
```

```
LEFTARROW  <- '<-' Spacing                                # Type 10
SLASH      <- '/' Spacing                                 # Type 11
AND        <- '&' Spacing                                 # Type 12
NOT        <- '!' Spacing                                 # Type 13
QUESTION   <- '?' Spacing                                 # Type 14
STAR       <- '*' Spacing                                 # Type 15
PLUS       <- '+' Spacing                                 # Type 16
OPEN       <- '(' Spacing                                 # Type 17
CLOSE      <- ')' Spacing                                 # Type 18
DOT        <- '.' Spacing                                 # Type 19
Spacing    <- (Space / Comment)*                          # Type 20
Comment    <- '#' (!EndOfLine .)* EndOfLine               # Type 21
Space      <- ' ' / '\t' / EndOfLine                      # Type 22
EndOfLine  <- '\r\n' / '\n' / '\r'                        # Type 23
EndOfFile  <- !.                                          # Type 24
```