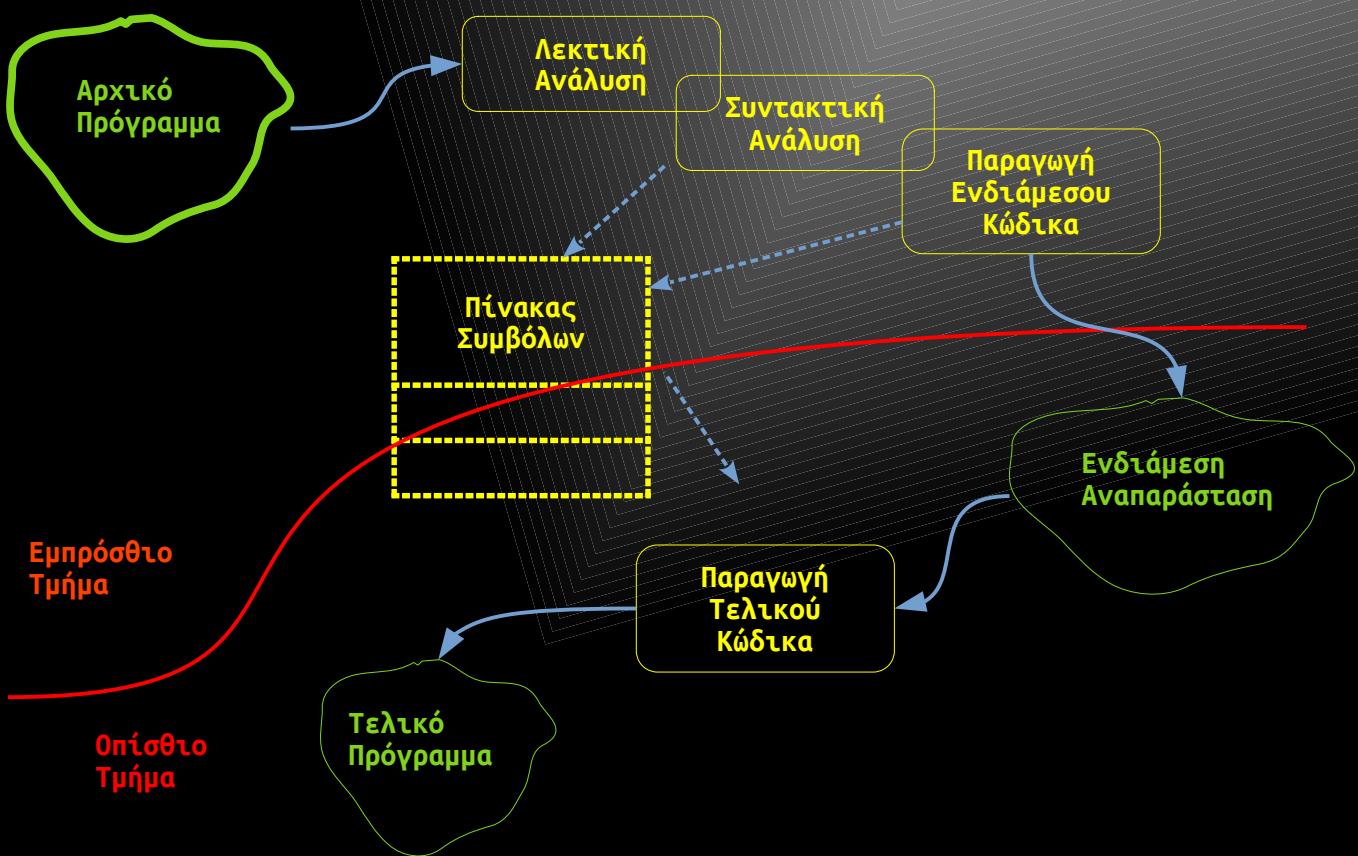


# ΕΓΧΕΙΡΙΔΙΟ ΣΧΕΔΙΑΣΗΣ ΚΑΙ ΑΝΑΠΤΥΞΗΣ ΜΕΤΑΓΛΩΤΤΙΣΤΩΝ





# Εγχειρίδιο Σχεδίασης και Ανάπτυξης Μεταγλωττιστών

---

Γεώργιος Μανής  
Αναπληρωτής Καθηγητής  
Πανεπιστήμιο Ιωαννίνων  
Πολυτεχνική Σχολή  
Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Τίτλος πρωτότυπου: «Εγχειρίδιο Σχεδίασης και Ανάπτυξης Μεταγλωττιστών»  
Copyright © 2023, ΚΑΛΛΙΠΟΣ, ΑΝΟΙΚΤΕΣ ΑΚΑΔΗΜΑΪΚΕΣ ΕΚΔΟΣΕΙΣ



(ΣΕΑΒ + ΕΛΚΕ-ΕΜΠ)

Το παρόν έργο διατίθεται με τους όρους της άδειας Creative Commons Αναφορά Δημιουργού – Μη Εμπορική Χρήση – Παρόμοια Διανομή 4.0. Για να δείτε τους όρους της άδειας αυτής επισκεφτείτε τον ιστότοπο <https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.el>

Αν τυχόν κάποιο τμήμα του έργου διατίθεται με διαφορετικό καθεστώς αδειοδότησης, αυτό αναφέρεται ρητά και ειδικώς στην οικεία θέση.

#### Συντελεστές έκδοσης

Γλωσσική επιμέλεια:

Αλεξάνδρα Χιώτη

Τεχνική επεξεργασία:

Γεώργιος Μανής

## ΚΑΛΛΙΠΟΣ

Εθνικό Μετσόβιο Πολυτεχνείο

Ηρώων Πολυτεχνείου 9 15780

Ζωγράφου

[www.kallipos.gr](http://www.kallipos.gr)

Βιβλιογραφική αναφορά:

Γεώργιος Μανής, (2023). *Εγχειρίδιο Σχεδίασης και Ανάπτυξης Μεταγλωττιστών*. [Προπτυχιακό Εγχειρίδιο] Κάλλιπος, Ανοικτές Ακαδημαϊκές Εκδόσεις

Διαθέσιμο στο:

<http://dx.doi.org/10.57713/kallipos-372>

ISBN:

978-618-228-141-3



Αφιερώνεται  
στους γονείς μου  
και στην οικογένειά μου

# ΠΕΡΙΕΧΟΜΕΝΑ

---

Εισαγωγή	xi
<b>1 Τυπικές γραμματικές και τεχνολογία μεταγλωττιστών</b>	<b>1</b>
1.1 Ορισμοί . . . . .	3
1.2 Ιεραρχία Chomsky . . . . .	4
1.2.1 Κανονικές γραμματικές . . . . .	5
1.2.2 Γραμματικές χωρίς συμφραζόμενα . . . . .	6
1.2.3 Γραμματικές με συμφραζόμενα . . . . .	6
1.2.4 Γενικές γραμματικές . . . . .	7
1.3 Περιγραφική ικανότητα τυπικών γραμματικών και μεταγλωττιστές . . . . .	8
1.4 Γραμματικές LL(1) . . . . .	10
1.5 Γραμματικές LR(1) . . . . .	11
1.6 Ο συμβολισμός EBNF . . . . .	12
Βιβλιογραφία . . . . .	13
<b>2 Γραμματικές LL(1) και συντακτική ανάλυση</b>	<b>15</b>
2.1 Το σύνολο <i>first</i> . . . . .	16
2.2 Το σύνολο <i>follow</i> . . . . .	18
2.3 Σύνολα <i>first</i> , <i>follow</i> και γραμματικές LL(1) . . . . .	20
2.4 Κατασκευή πίνακα ανίχνευσης/συντακτικής ανάλυσης LL(1) . . . . .	21
2.5 Συντακτική ανάλυση με πίνακα ανίχνευσης . . . . .	22
Βιβλιογραφία . . . . .	24
<b>3 Η γλώσσα προγραμματισμού <i>C-implement</i></b>	<b>25</b>
3.1 Λεκτικές μονάδες . . . . .	26
3.2 Μορφή προγράμματος . . . . .	27
3.3 Τύποι και δηλώσεις μεταβλητών . . . . .	27
3.4 Τελεστές και εκφράσεις . . . . .	27
3.5 Δομές της γλώσσας . . . . .	28
3.5.1 Εκχώρηση . . . . .	28
3.5.2 Απόφαση “if” . . . . .	28

3.5.3	Επανάληψη “while” . . . . .	28
3.5.4	Επιλογή “switchcase” . . . . .	28
3.5.5	Επανάληψη “forcase” . . . . .	28
3.5.6	Επανάληψη “incase” . . . . .	29
3.5.7	Επιστροφή τιμής συνάρτησης . . . . .	29
3.5.8	Έξοδος δεδομένων . . . . .	29
3.5.9	Είσοδος δεδομένων . . . . .	29
3.5.10	Κλήση διαδικασίας . . . . .	29
3.6	Συναρτήσεις και διαδικασίες . . . . .	29
3.7	Μετάδοση παραμέτρων . . . . .	30
3.8	Κανόνες εμβέλειας . . . . .	30
3.9	Η γραμματική της <i>C-imple</i> . . . . .	31
3.10	Παραδείγματα κώδικα . . . . .	34
3.11	Γραμματικές δημοφιλών γλωσσών προγραμματισμού . . . . .	37
	Βιβλιογραφία . . . . .	37
<b>4</b>	<b>Λεκτικός αναλυτής</b>	<b>39</b>
4.1	Ο λεκτικός αναλυτής στη διαδικασία της μετάφρασης . . . . .	40
4.2	Η εσωτερική λειτουργία ενός λεκτικού αναλυτή . . . . .	42
4.3	Διαχείριση σφαλμάτων . . . . .	44
4.4	Παράδειγμα εκτέλεσης . . . . .	46
4.5	Διάγραμμα κλάσεων λεκτικού αναλυτή . . . . .	47
	Βιβλιογραφία . . . . .	48
<b>5</b>	<b>Συντακτικός αναλυτής</b>	<b>51</b>
5.1	Ενεργοποίηση κανόνων με πρόβλεψη . . . . .	52
5.2	Υλοποίηση συναρτήσεων αναδρομικής κατάβασης . . . . .	54
5.3	Ο συντακτικός αναλυτής στο διάγραμμα κλάσεων . . . . .	58
	Βιβλιογραφία . . . . .	59
<b>6</b>	<b>Παραγωγή ενδιάμεσου κώδικα</b>	<b>61</b>
6.1	Η ενδιάμεση αναπαράσταση . . . . .	63
6.2	Βοηθητικές συναρτήσεις . . . . .	68
6.3	Αριθμητικές παραστάσεις . . . . .	69
6.4	Λογικές συνθήκες . . . . .	78
6.5	Αρχή και τέλος ενότητας . . . . .	90
6.6	Είσοδος και έξοδος δεδομένων . . . . .	91
6.7	Τερματισμός εκτέλεσης . . . . .	91
	Βιβλιογραφία . . . . .	91
<b>7</b>	<b>Ενδιάμεσος κώδικας για τις δομές της γλώσσας, συναρτήσεις και διαδικασίες</b>	<b>93</b>
7.1	Οι δομές της γλώσσας . . . . .	94
7.1.1	Η δομή επανάληψης “while” . . . . .	94
7.1.2	Η δομή απόφασης ”if” . . . . .	97
7.1.3	Η δομή επιλογής ”switchcase” . . . . .	100
7.1.4	Η δομή επιλογής-επανάληψης ”forcase” . . . . .	103
7.1.5	Η δομή πολλαπλής επιλογής-επανάληψης ”incase” . . . . .	105
7.2	Συναρτήσεις και διαδικασίες . . . . .	108
7.3	Ένα ολοκληρωμένο, απλό, παράδειγμα μετατροπής κώδικα σε ενδιάμεση αναπαράσταση	110

7.4	Ένα πιο σύνθετο παράδειγμα, με φωλιασμένες δομές . . . . .	113
7.5	Διάφορες ενδιάμεσες αναπαραστάσεις . . . . .	117
7.6	Οι κλάσεις της φάσης της παραγωγής του ενδιάμεσου κώδικα . . . . .	118
	Βιβλιογραφία . . . . .	119
<b>8</b>	<b>Πίνακας συμβόλων</b>	<b>121</b>
8.1	Οι εγγραφές στον πίνακα συμβόλων . . . . .	122
8.1.1	Μεταβλητή . . . . .	123
8.1.2	Παράμετρος . . . . .	124
8.1.3	Συνάρτηση και διαδικασία . . . . .	124
8.1.4	Τυπική παράμετρος . . . . .	125
8.1.5	Προσωρινή μεταβλητή . . . . .	126
8.1.6	Συμβολική σταθερά . . . . .	126
8.2	Η δομή του πίνακα συμβόλων . . . . .	126
8.3	Οι κανόνες εμβέλειας . . . . .	127
8.4	Το εγγράφημα δραστηριοποίησης . . . . .	128
8.5	Οι λειτουργίες του πίνακα συμβόλων . . . . .	130
8.6	Σημασιολογική ανάλυση . . . . .	131
8.7	Αντικειμενοστραφής σχεδίαση . . . . .	132
8.8	Παράδειγμα λειτουργίας του πίνακα συμβόλων . . . . .	133
	Βιβλιογραφία . . . . .	138
<b>9</b>	<b>Παραγωγή τελικού κώδικα</b>	<b>139</b>
9.1	Η συμβολική γλώσσα μηχανής του RISC-V . . . . .	141
9.1.1	Οι καταχωρητές του RISC-V . . . . .	141
9.1.2	Πράξεις μεταξύ ακέραιων αριθμών . . . . .	142
9.1.3	Η πρόσβαση στη μνήμη . . . . .	142
9.1.4	Εντολές διακλαδώσεων . . . . .	143
9.1.5	Κλήση συνάρτησης ή διαδικασίας . . . . .	144
9.1.6	Είσοδος και έξοδος δεδομένων . . . . .	144
9.1.7	Τερματισμός προγράμματος . . . . .	145
9.1.8	Παράδειγμα ταξινόμησης . . . . .	146
9.1.9	Το εγγράφημα δραστηριοποίησης . . . . .	148
9.2	Βοηθητικές συναρτήσεις . . . . .	151
9.2.1	Η συνάρτηση <code>gnlvcode()</code> . . . . .	151
9.2.2	Η συνάρτηση <code>loadvr()</code> . . . . .	154
9.2.2.1	Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή ή προσωρινή μεταβλητή . . . . .	154
9.2.2.2	Παράμετρος που έχει περαστεί με αναφορά . . . . .	154
9.2.2.3	Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή η οποία ανήκει σε πρόγονο . . . . .	155
9.2.2.4	Παράμετρος που έχει περαστεί με αναφορά, η οποία ανήκει σε πρόγονο	156
9.2.2.5	Καθολική μεταβλητή . . . . .	158
9.2.2.6	Εκχώρηση αριθμητικής σταθεράς . . . . .	159
9.2.3	Η συνάρτηση <code>storerv()</code> . . . . .	160
9.2.3.1	Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή ή προσωρινή μεταβλητή . . . . .	160
9.2.3.2	Παράμετρος που έχει περαστεί με αναφορά . . . . .	160

9.2.3.3	Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή η οποία ανήκει σε πρόγονο . . . . .	161
9.2.3.4	Παράμετρος που έχει περαστεί με αναφορά, η οποία ανήκει σε πρόγονο	161
9.2.3.5	Καθολική μεταβλητή . . . . .	161
9.2.3.6	Αριθμητική σταθερά . . . . .	161
9.3	Εκχώρηση και αριθμητικές πράξεις . . . . .	162
9.4	Διακλαδώσεις . . . . .	163
9.5	Αρχή προγράμματος, κυρίως πρόγραμμα και τέλος προγράμματος . . . . .	163
	Βιβλιογραφία . . . . .	165
<b>10</b>	<b>Πέρασμα παραμέτρων, κλήση συναρτήσεων και διαδικασιών στον τελικό κώδικα</b>	<b>167</b>
10.1	Πέρασμα παραμέτρων . . . . .	168
10.1.1	Πέρασμα παραμέτρων με τιμή . . . . .	168
10.1.2	Πέρασμα παραμέτρων με αναφορά . . . . .	170
10.1.2.1	Παράμετρος με αναφορά, όταν στη στοίβα είναι αποθηκευμένη η τιμή της παραμέτρου . . . . .	170
10.1.2.2	Παράμετρος με αναφορά, όταν στη στοίβα βρίσκεται η διεύθυνση της παραμέτρου . . . . .	171
10.1.2.3	Παράδειγμα περάσματος παραμέτρου με τη χρήση της <code>gnlvcode()</code> .	172
10.1.3	Επιστροφή τιμής συνάρτησης . . . . .	173
10.2	Κλήση συνάρτησης ή διαδικασίας . . . . .	174
10.2.1	Σύνδεσμος προσπέλασης . . . . .	174
10.2.2	Δείκτης στοίβας . . . . .	176
10.2.3	Μεταβίβαση ελέγχου . . . . .	177
10.3	Παράδειγμα παραγωγής τελικού κώδικα . . . . .	178
	Βιβλιογραφία . . . . .	190
<b>11</b>	<b>Βελτιστοποίηση κώδικα</b>	<b>193</b>
11.1	Τεχνικές αποτίμησης . . . . .	194
11.1.1	Διάδοση σταθερών . . . . .	194
11.1.2	Υπολογισμοί σταθερών εκφράσεων . . . . .	195
11.1.3	Διάδοση αντιγράφων . . . . .	195
11.1.4	Διάδοση υποκατάστατων . . . . .	195
11.1.5	Αλγεβρικές απλοποιήσεις . . . . .	195
11.1.6	Υποβίβασμός ισχύος . . . . .	196
11.2	Απαλοιφή κώδικα . . . . .	196
11.2.1	Απαλοιφή μη προσβάσιμου κώδικα . . . . .	196
11.2.2	Απαλοιφή μη χρήσιμου κώδικα . . . . .	196
11.2.3	Απαλοιφή μη χρήσιμων μεταβλητών . . . . .	197
11.2.4	Απαλοιφή κοινών υποεκφράσεων . . . . .	197
11.3	Βελτιστοποιητικοί μετασχηματισμοί στις κλήσεις διαδικασιών και συναρτήσεων . . . . .	197
11.3.1	Βελτιστοποίηση κλήσης σε διαδικασία φύλλο . . . . .	198
11.3.2	Εκχώρηση καταχωρητών κατά τις κλήσεις . . . . .	198
11.3.3	Ενσωμάτωση διαδικασίας . . . . .	198
11.3.4	Απαλοιφή αναδρομής ουράς . . . . .	199
11.4	Απομνημόνευση κλήσεων . . . . .	199
11.5	Μετασχηματισμοί βρόχων . . . . .	200
11.5.1	Υποβίβασμός ισχύος βασιζόμενος σε βρόχο . . . . .	200
11.5.2	Απαλοιφή επαγωγικών μεταβλητών . . . . .	201

11.5.3 Μετακίνηση κώδικα ανεξάρτητου της εκτέλεσης του βρόχου . . . . .	202
11.5.4 Εναλλαγή σε βρόχο . . . . .	203
11.5.5 Κυκλική συρρίκνωση . . . . .	203
11.5.6 Διαχωρισμός και ένωση βρόχων . . . . .	204
Βιβλιογραφία . . . . .	204
<b>12 Εργαλεία αυτοματοποιημένης ανάπτυξης μεταγλωττιστών</b>	<b>207</b>
12.1 Ανάπτυξη μεταγλωττιστών με τα εργαλεία lex και yacc/bison . . . . .	208
12.1.1 Το μετα-εργαλείο lex . . . . .	208
12.1.2 Το μετα-εργαλείο yacc/bison . . . . .	211
12.1.3 Ένα απλό μετα-πρόγραμμα yacc . . . . .	214
12.1.4 Παράδειγμα υλοποίησης συντακτικού και λεκτικού αναλυτή με τα μετα-εργαλεία lex και yacc . . . . .	216
12.2 Το μετα-εργαλείο ANTLR . . . . .	221
12.2.1 Συντακτική ανάλυση με το ANTLR . . . . .	221
12.2.2 Σύνταξη γραμματικών ANTLR . . . . .	224
12.2.3 Προγραμματίζοντας με listeners . . . . .	226
Βιβλιογραφία . . . . .	227
<b>ΠΑΡΑΡΤΗΜΑΤΑ</b>	
<b>A Αναλυτικό παράδειγμα μεταγλώττισης</b>	<b>229</b>
<b>B Λεκτικός και συντακτικός αναλυτής με τα μετα-εργαλεία Lex-Yacc/Bison</b>	<b>257</b>
<b>Γ Μορφωτής κώδικα με το μετα-εργαλείο ANTLR και γλώσσα προγραμματισμού Python</b>	<b>263</b>



# ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

---

1	Είσοδος και έξοδος ενός μεταγλωττιστή . . . . .	xii
2	Η εσωτερική αρχιτεκτονική ενός μεταγλωττιστή . . . . .	xii
1.1	Γραμματικές στη λεκτική και συντακτική ανάλυση. . . . .	2
1.2	Η ιεραρχία του Chomsky. . . . .	4
4.1	Ο λεκτικός αναλυτής. . . . .	40
4.2	Το αυτόματο λειτουργίας του λεκτικού αναλυτή. . . . .	42
4.3	Διαχείριση σχολίων από τον λεκτικό αναλυτή. . . . .	44
4.4	Διαχείριση σφαλμάτων από τον λεκτικό αναλυτή. . . . .	45
4.5	Διάγραμμα κλάσεων του λεκτικού αναλυτή. . . . .	48
5.1	Ο συντακτικός αναλυτής στη διαδικασία της μεταγλώττισης. . . . .	52
5.2	Διάγραμμα κλάσεων λεκτικού-συντακτικού αναλυτή. . . . .	59
6.1	Η ενδιάμεση αναπαράσταση αποτελεί μέσο επικοινωνίας ανάμεσα στο εμπρόσθιο και το οπίσθιο τμήμα του μεταγλωττιστή. . . . .	62
6.2	Παραγωγή κώδικα. . . . .	63
7.1	Παραγωγή ενδιάμεσου κώδικα για το while. . . . .	95
7.2	Εσφαλμένη παραγωγή κώδικα για το if. . . . .	97
7.3	Παραγωγή κώδικα για το if. . . . .	98
7.4	Παραγωγή κώδικα για το switchcase. . . . .	102
7.5	Παραγωγή κώδικα για το forcase. . . . .	104
7.6	Παραγωγή κώδικα για το incase. . . . .	107
7.7	Η παραγωγή ενδιάμεσου κώδικα στο διάγραμμα κλάσεων. . . . .	118
8.1	Ο πίνακας συμβόλων. . . . .	123
8.2	Η δομή του πίνακα συμβόλων. . . . .	127
8.3	Η δομή του εγγραφήματος δραστηριοποίησης. . . . .	130
8.4	Η ιεραρχία των κλάσεων των εγγραφών του πίνακα συμβόλων. . . . .	133
8.5	Οι σχέσεις ανάμεσα στις κλάσεις Table, Scope και Entity. . . . .	133
8.6	Αρχικός κώδικας παραδείγματος λειτουργίας του πίνακα συμβόλων. . . . .	135

8.7	Παράδειγμα μετάφρασης, στιγμιότυπο του πίνακα συμβόλων . . . . .	135
8.8	Παράδειγμα μετάφρασης, στιγμιότυπο του πίνακα συμβόλων . . . . .	137
8.9	Παράδειγμα μετάφρασης, στιγμιότυπο του πίνακα συμβόλων . . . . .	137
8.10	Παράδειγμα μετάφρασης, στιγμιότυπο του πίνακα συμβόλων . . . . .	137
9.1	Η παραγωγή τελικού κώδικα στη διαδικασία της μεταγλώττισης. . . . .	140
9.2	Εγγράφημα δραστηριοποίησης του κυρίως προγράμματος στη στοίβα. . . . .	148
9.3	Νέο εγγράφημα δραστηριοποίησης στη στοίβα. . . . .	149
9.4	Εγγραφήματα δραστηριοποίησης στη στοίβα από φωλιασμένες κλήσεις. . . . .	149
9.5	Ολοκλήρωση συνάρτησης και επιστροφή δεσμευμένου χώρου στη στοίβα. . . . .	150
9.6	Εγγράφημα δραστηριοποίησης. . . . .	150
9.7	Πρόσβαση σε δεδομένα που βρίσκονται σε συναρτήσεις προγόνους, μέσω της <code>gnv1code()</code> . . . . .	153
9.8	Ανάγνωση μιας τοπικής μεταβλητής ή παραμέτρου που έχει περαστεί με τιμή ή προσωρινής μεταβλητής. . . . .	155
9.9	Προσπέλαση μιας παραμέτρου που έχει περαστεί με αναφορά. . . . .	156
9.10	Προσπέλαση τιμής μιας τοπικής μεταβλητής ή παραμέτρου με τιμή και βρίσκεται σε κάποιον πρόγονο. . . . .	157
9.11	Προσπέλαση τυπικής παραμέτρου που έχει περαστεί με αναφορά και βρίσκεται σε κάποιον πρόγονο. . . . .	158
9.12	Προσπέλαση καθολικής μεταβλητής. . . . .	159
10.1	Πέρασμα παραμέτρων με τιμή . . . . .	169
10.2	Πέρασμα παραμέτρων με αναφορά . . . . .	171
10.3	Πέρασμα παραμέτρων με αναφορά, όταν στο εγγράφημα δραστηριοποίησης είναι αποθηκευμένη η διεύθυνση της μεταβλητής . . . . .	172
10.4	Πέρασμα με αναφορά και σύνδεσμος προσπέλασης . . . . .	173
10.5	Επιστροφή τιμής συνάρτησης . . . . .	174
10.6	Σύνδεσμος προσπέλασης . . . . .	176
10.7	Ο δείκτης στοίβας κατά την κλήση συνάρτησης . . . . .	177
11.1	Κυκλική συρρίκνωση . . . . .	204
12.1	Παραγωγή λεκτικού αναλυτή με το <code>lex</code> . . . . .	208
12.2	Παράδειγμα κανονικής έκφρασης που αναγνωρίζει ο <code>lex</code> . . . . .	209
12.3	Παραγωγή συντακτικού αναλυτή με το <code>yacc</code> . . . . .	212

# ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

---

2.1	Τα σύνολα <i>first</i> παραδείγματος γραμματικής αριθμητικών εκφράσεων. . . . .	18
2.2	Τα σύνολα <i>follow</i> παραδείγματος γραμματικής αριθμητικών εκφράσεων. . . . .	20
2.3	Πίνακας ανίχνευσης για τη γραμματική των αριθμητικών εκφράσεων. . . . .	22
2.4	Αναγνώριση της συμβολοσειράς $(\alpha + \beta)$ από τη γραμματική των αριθμητικών εκφράσεων.	23
9.1	Μονάδες υλικού/λογισμικού που χρησιμοποιούνται από τις <code>loadvr()</code> και <code>storerv()</code> . . .	162
9.2	Αντιστοίχιση λογικών αλμάτων στον ενδιάμεσο και στον τελικό κώδικα. . . . .	164



## ΕΙΣΑΓΩΓΗ

---

Σκοπός του παρόντος συγγράμματος είναι η εισαγωγή του αναγνώστη στην επιστήμη των μεταγλωττιστών. Η προσέγγιση που επιλέγεται είναι αυτή της καθοδήγησης της γνώσης μέσα από τη διαδικασία ανάπτυξης. Σε αντίθεση με άλλα βιβλία που είναι διαθέσιμα στα ελληνικά και στα οποία δίνεται το βάρος στο θεωρητικό υπόβαθρο, εδώ το θεωρητικό υπόβαθρο γίνεται όχημα και η ανάπτυξη το κίνητρο και ο οδηγός.

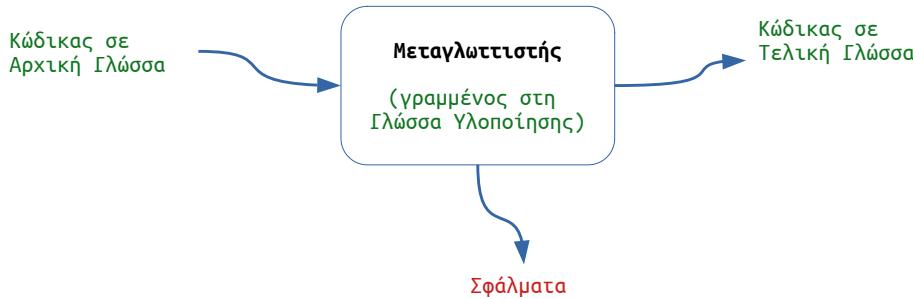
Έτσι, στο βιβλίο αυτό ορίζεται μία εκπαιδευτική γλώσσα προγραμματισμού για την οποία θα υλοποιηθεί ένας πλήρως λειτουργικός μεταγλωττιστής ο οποίος θα δέχεται ως είσοδο προγράμματα στη γλώσσα αυτή και θα παράγει κώδικα σε γλώσσα μηχανής, τον οποίο θα μπορούν οι αναγνώστες στη συνέχεια να εκτελούν.

Η πορεία του βιβλίου συμπίπτει με αυτή της ανάπτυξης του μεταγλωττιστή. Η ανάπτυξη ενός μεταγλωττιστή χωρίζεται σε φάσεις. Ο συγγραφέας θεωρεί ότι οι αναγνώστες έχουν γνώσεις θεωρίας υπολογισμού και το βιβλίο λειτουργεί αθροιστικά, προσθέτοντας μόνο τις γνώσεις εκείνες, οι οποίες συνήθως δεν αποτελούν τμήμα της θεωρίας υπολογισμού και σχετίζονται στενά με την επιστήμη των μεταφραστών. Μετά τις απαραίτητες θεωρητικές γνώσεις, για κάθε φάση συζητείται η μεθοδολογία ανάπτυξής της σε βαθμό αρκετά λεπτομερή, ώστε να μπορεί κάποιος να υλοποιήσει τον μεταγλωττιστή με τη βοήθεια του βιβλίου και μόνο.

Οι φάσεις της ανάπτυξης είναι η λεκτική ανάλυση, η συντακτική ανάλυση, η παραγωγή ενδιάμεσου κώδικα, η κατασκευή του πίνακα συμβόλων και η παραγωγή τελικού κώδικα. Για καθεμία από τις φάσεις αυτές το βιβλίο αφιερώνει ένα ή δύο κεφάλαια. Κάποια εισαγωγικά κεφάλαια στην αρχή, ένα κεφάλαιο που διαπραγματεύεται θέματα βελτιστοποίησης κώδικα και ένα κεφάλαιο που περιγράφει εργαλεία με τα οποία μπορούμε να αυτοματοποιήσουμε διαδικασίες ανάπτυξης μεταγλωττιστών, ολοκληρώνουν το βιβλίο. Στα παραρτήματά του μπορεί κανείς να βρει ένα ολοκληρωμένο παράδειγμα μεταγλωττισης και κάποια παραδείγματα ανάπτυξης μικρών εφαρμογών με εργαλεία αυτοματοποιημένης ανάπτυξης.

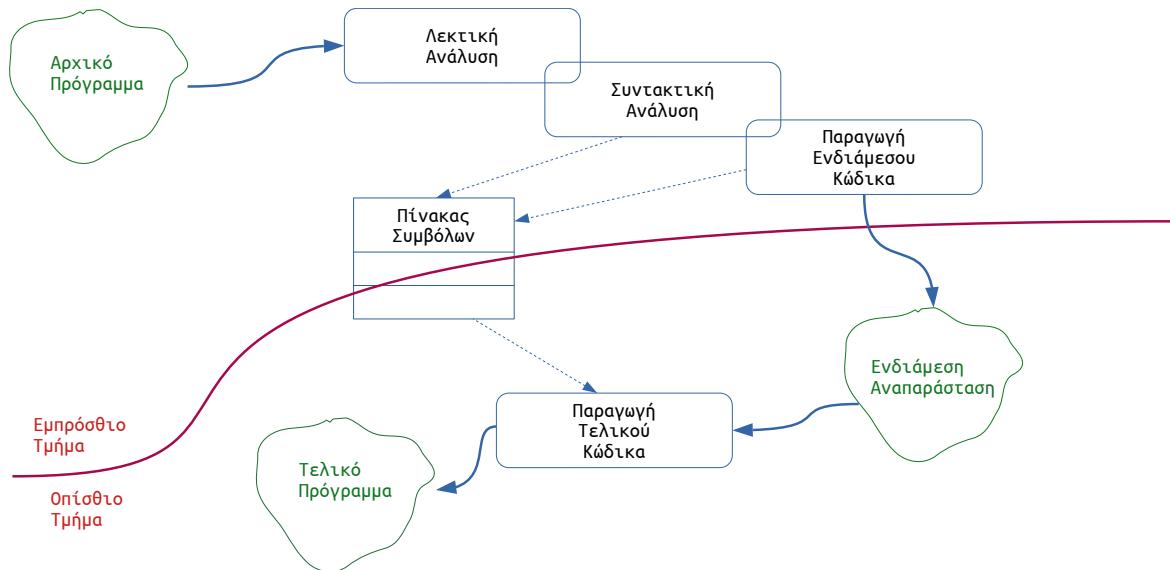
Ένας μεταγλωττιστής μεταφράζει ένα πρόγραμμα από την αρχική γλώσσα στην τελική γλώσσα. Η αρχική γλώσσα είναι η γλώσσα του μεταφραστή της οποίας υλοποιούμε. Τελική γλώσσα είναι η γλώσσα, στην οποία μετατρέπουμε το πρόγραμμα. Συνήθως πρόκειται για τη γλώσσα μηχανής κάποιου επεξεργαστή. Γλώσσα υλοποίησης είναι η γλώσσα στην οποία είναι γραμμένος ο κώδικας του μεταγλωττιστή. Στο σχήμα 1 εικονίζονται η είσοδος και η έξοδος ενός μεταγλωττιστή, όπου φαίνονται και οι γλώσσες που ονομάσαμε παραπάνω.

Η εσωτερική δομή ενός μεταγλωττιστή φαίνεται στο σχήμα 2, όπου εικονίζονται οι βασικές του φάσεις. Θα μπορούσαμε να εμπλουτίσουμε το σχήμα αυτό και με άλλες λειτουργίες, όπως η σημασιολογική ανάλυση και η βελτιστοποίηση. Ας κρατήσουμε το σχήμα αυτό και ας δούμε μία προς μία τις φάσεις της μεταγλωττισης που εικονίζονται στο σχήμα 2, δίνοντας μία σύντομη περιγραφή για καθεμία από αυτές και σκιαγραφώντας τον ρόλο της στον μεταγλωττιστή.



Σχήμα 1: Είσοδος και έξοδος ενός μεταγλωττιστή

Κατά τη λεκτική ανάλυση, το υπό μεταγλωττιση πρόγραμμα διαβάζεται χαρακτήρα-χαρακτήρα από το αρχείο εισόδου και, αξιοποιώντας μία κατάλληλα σχεδιασμένη μηχανή πεπερασμένων καταστάσεων, σχηματίζονται οι λεκτικές μονάδες. Λεκτική μονάδα μπορεί να είναι μία λέξη κλειδί της γλώσσας, μία μεταβλητή, ένα μαθηματικό σύμβολο, κάθε σύμβολο της γλώσσας. Οι λεκτικές μονάδες επιστρέφονται ως αποτέλεσμα από τον λεκτικό αναλυτή και αποτελούν την είσοδο στον συντακτικό αναλυτή. Αν ο λεκτικός αναλυτής εντοπίσει κάποιο σφάλμα, τότε εμφανίζει το κατάλληλο μήνυμα και τερματίζει τη μεταγλωττιση.



Σχήμα 2: Η εσωτερική αρχιτεκτονική ενός μεταγλωττιστή

Ο συντακτικός αναλυτής, με τη σειρά του, παίρνει ως είσοδο τις λεκτικές μονάδες που εντοπίζει ο λεκτικός αναλυτής και, με βάση τη γραμματική της γλώσσας, ελέγχει αν το υπό μεταγλωττιση πρόγραμμα είναι συντακτικά ορθό. Η περιγραφική ικανότητα της συντακτικής ανάλυσης είναι αυτής μίας γραμματικής χωρίς συμφραζόμενα, αφού μία τέτοια γραμματική επιλέγουμε να χρησιμοποιήσουμε για την περιγραφή των γλωσσών προγραμματισμού.

Αν χρειαζόμαστε περισσότερη περιγραφική δύναμη, και τη χρειαζόμαστε, επιλέγουμε να μετακινήσουμε τους απαιτούμενους ελέγχους σε κάτι που ονομάζουμε σημασιολογική ανάλυση. Στη σημασιολογική ανάλυση χρησιμοποιούμε ευρετικές τεχνικές προκειμένου να ελέγχουμε απαιτήσεις που αδυνατεί να περιγράψει μία γραμματική χωρίς συμφραζόμενα. Πέρα από τον λεκτικό αναλυτή, σφάλματα μπορούν να επιστρέψουν ο συντακτικός αναλυτής και η σημασιολογική ανάλυση. Οι υπόλοιπες φάσεις δεν εντοπίζουν σφάλματα. Όταν εντοπιστεί κάποιο σφάλμα, εμφανίζεται το κατάλληλο μήνυμα και τερματίζεται η μεταγλωττιση.

Η παραγωγή του ενδιάμεσου κώδικα μετατρέπει το υπό μεταγλωττιση πρόγραμμα σε μία γλώσσα που αποκαλούμε ενδιάμεση αναπαράσταση και λειτουργεί ως ενδιάμεσο βήμα ανάμεσα στο πρόγραμμα που θα μεταγλωττιστεί και στον τελικό κώδικα που θα παραχθεί. Η ύπαρξη μιας ενδιάμεσης γλώσσας, ως σκαλοπάτι

ανάμεσα στην αρχική και την τελική γλώσσα, έχει πολλά πλεονεκτήματα. Η σχεδίαση του μεταγλωττιστή απλοποιείται σημαντικά και τη χωρίζει σε δύο ανεξάρτητες φάσεις:

- Τη φάση πριν την παραγωγή του ενδιάμεσου κώδικα. Καλούμε αυτό το μέρος του μεταγλωττιστή εμπρόσθιο τμήμα (*front end*). Το εμπρόσθιο τμήμα είναι υπεύθυνο για τη μετατροπή της αρχικής γλώσσας σε ενδιάμεσο κώδικα.
- Τη φάση μετά την παραγωγή του ενδιάμεσου κώδικα. Καλούμε αυτό το μέρος του μεταγλωττιστή οπίσθιο τμήμα (*back end*). Το οπίσθιο τμήμα είναι υπεύθυνο για τη μετατροπή του ενδιάμεσου κώδικα σε τελικό κώδικα.

Το εμπρόσθιο και το οπίσθιο τμήμα φαίνονται στο σχήμα 2 χωρισμένα με μία κόκκινη γραμμή. Ο διαχωρισμός αυτός, πέρα από την απλοποίηση που επιφέρει στη σχεδίαση του μεταγλωττιστή, έχει ακόμα ένα πολύ σημαντικό πλεονέκτημα: η ενδιάμεση γλώσσα αποτελεί ένα στρώμα επικοινωνίας ανάμεσα στο εμπρόσθιο και το οπίσθιο τμήμα το οποίο είναι ανεξάρτητο, τόσο της αρχικής, όσο και της τελικής γλώσσας. Έτσι, το εμπρόσθιο τμήμα θα είναι το ίδιο για κάθε μεταγλωττιστή της αρχικής γλώσσας, ενώ το οπίσθιο τμήμα θα είναι το ίδιο για κάθε μεταγλωττιστή που παράγει κώδικα στην ίδια τελική γλώσσα, ανεξάρτητα ποια θα είναι η αρχική. Αυτό δίνει ευκαιρία για επαναχρησιμοποίηση κώδικα και ευκολότερη ανάπτυξη μεταγλωττιστών, αν έχει ήδη αναπτυχθεί μεταγλωττιστής της αρχικής γλώσσας ή μεταγλωττιστής που παράγει κώδικα στην τελική γλώσσα που μας ενδιαφέρει.

Ο πίνακας συμβόλων δεν μπορεί να τοποθετηθεί σε αυτό το σχήμα, κάπου ανάμεσα σε δύο άλλες φάσεις. Συγκεντρώνει πληροφορία κατά τη συντακτική ανάλυση και την παραγωγή του ενδιάμεσου κώδικα και τη διαθέτει στην παραγωγή τελικού κώδικα. Χρονικά η υλοποίηση του τμήματος λογισμικού του τελικού κώδικα γίνεται μετά την υλοποίηση της παραγωγής ενδιάμεσου κώδικα και πριν την υλοποίηση της παραγωγής του τελικού. Οι πληροφορίες που διατηρεί ο πίνακας συμβόλων σχετίζονται με οτιδήποτε έχει όνομα σε ένα πρόγραμμα, δηλαδή μεταβλητές, συναρτήσεις, παραμέτρους κλπ.

Τέλος, ακολουθεί η παραγωγή τελικού κώδικα, όπου ο ενδιάμεσος κώδικας, με τη βοήθεια του πίνακα συμβόλων, μετατρέπεται στην τελική γλώσσα, που συνήθως είναι η γλώσσα μηχανής.

Βελτιστοποίηση μπορεί να επιλεγεί να γίνει σε διάφορα σημεία της ανάπτυξης. Συνήθως εφαρμόζουμε τεχνικές βελτιστοποίησης πάνω στον παραγόμενο ενδιάμεσο και τελικό κώδικα.

Για καθευμία από τις παραπάνω φάσεις έχουν αφιερωθεί ένα ή περισσότερα κεφάλαια στο βιβλίο αυτό.

Η διάρθρωση του βιβλίου σε κεφάλαια είναι ως εξής:

- *Κεφάλαιο 1: Τυπικές γραμματικές και τεχνολογία μεταγλωττιστών*

Το βιβλίο αυτό θεωρεί ως δεδομένο το θεωρητικό υπόβαθρο που αφορά τη θεωρία αυτομάτων, τυπικών γραμματικών και τυπικών γλωσσών. Θεωρεί, όμως, χρήσιμο να συνοψίσει τις βασικές έννοιες και τους σημαντικότερους ορισμούς, οι οποίοι θα φανούν χρήσιμοι στη συνέχεια. Έτσι, στην αρχή του κεφαλαίου παρουσιάζονται οι ορισμοί των τυπικών γραμματικών και των τυπικών γλωσσών, αλλά εισάγονται και άλλες χρήσιμες έννοιες, όπως το τερματικό και μη τερματικό σύμβολο, ο κανόνας παραγωγής, η πρόταση και ο προτασιακός τύπος.

Στη συνέχεια παρουσιάζεται η οργάνωση των τυπικών γραμματικών, των τυπικών γλωσσών και των αυτομάτων με βάση την *ιεραρχία Chomsky*. Ο Noam Chomsky ιεράρχησε τις τυπικές γραμματικές, τις τυπικές γλώσσες και τα αυτόματα με βάση την περιγραφική ισχύ τους. Κάθε επίπεδο μπορεί να παράγει ένα σύνολο από συμβολοσειρές, το οποίο είναι υποσύνολο του συνόλου συμβολοσειρών που μπορεί να παράγει το ανώτερό του επίπεδο.

Αφού παρουσιαστεί αναλυτικά κάθε επίπεδο της ιεραρχίας Chomsky, θα γίνει αντιστοίχιση του κάθε επιπέδου με τις διάφορες φάσεις ανάπτυξης ενός μεταγλωττιστή. Οι τυπικές γραμματικές χρησιμοποιούνται στη φάση της λεκτικής και της συντακτικής ανάλυσης, ενώ συνδέονται και με τη σημασιολογική ανάλυση.

Ενισχύοντας το θεωρητικό υπόβαθρο του αναγνώστη, θα εισαχθούν μερικές ακόμα χρήσιμες έννοιες, όπως η γραμματική LL(1). Μία γραμματική LL(1) επιτρέπει την επιλογή του κανόνα που θα ενεργοποιηθεί με βάση το επόμενο τερματικό σύμβολο στην υπό αναγνώριση συμβολοσειρά. Με τη βοήθεια μιας γραμματικής LL(1) μπορούμε εύκολα να κατασκευάσουμε συντακτικούς αναλυτές.

Τέλος, θα δούμε τον συμβολισμό EBNF. Ο συμβολισμός αυτός είναι πιο περιεκτικός από τον συμβολισμό που χρησιμοποιούμε για τις απλές γραμματικές και παρουσιάζει κάποια πρόσθετα πλεονεκτήματα τα οποία διευκολύνουν στην ανάπτυξη ενός μεταγλωττιστή.

- *Κεφάλαιο 2: Γραμματικές LL(1) και συντακτική ανάλυση*

Στο κεφάλαιο αυτό θα δούμε πως χρησιμοποιούνται οι γραμματικές LL(1) στη συντακτική ανάλυση. Η σύνταξη μίας γλώσσας προγραμματισμού σε γραμματική τύπου LL(1) επιτρέπει την αυτοματοποιημένη μετατροπή της γραμματικής σε κώδικα, είτε μέσω της τεχνικής της προβλέπουσας αναδρομικής κατάβασης, την οποία θα δούμε σε επόμενο κεφάλαιο, είτε μέσω του πίνακα ανίχνευσης (ή άλλις πίνακα συντακτικής ανάλυσης), τεχνική που θα μελετήσουμε στο κεφάλαιο αυτό. Για τον σχηματισμό του πίνακα ανίχνευσης θα χρειαστεί να ορίσουμε και να υπολογίσουμε κάποια σύνολα από τερματικά σύμβολα.

Έτσι, θα οριστούν τα σύνολα *first* και *follow*. Το σύνολο *first* αποτελείται από τα τερματικά σύμβολα από τα οποία μπορεί να ξεκινήσουν οι παραγωγές που παράγονται από κάποιον κανόνα. Το σύνολο *follow(A)* αποτελείται από τα τερματικά σύμβολα που μπορεί να βρεθούν μετά από το μη τερματικό σύμβολο A σε έναν κανόνα. Ο υπολογισμός των συνόλων *first* και *follow* μπορεί να γίνει με μηχανιστικό τρόπο και θα παρουσιαστεί ένας αλγόριθμος υπολογισμού για το συνόλο *first* και ένας για το σύνολο *follow*.

Τα σύνολα *first* και *follow* βοηθούν στην κατασκευή του πίνακα ανίχνευσης/συντακτικής ανάλυσης. Ο πίνακας ανίχνευσης είναι μία δισδιάστατη δομή, η οποία στη μία διάσταση έχει τα τερματικά σύμβολα, στην άλλη τα μη τερματικά σύμβολα και στα κελιά τον κανόνα που πρέπει να ακολουθηθεί για το συγκεκριμένο ζευγάρι τερματικού και μη τερματικού συμβόλου. Για την κατασκευή του πίνακα θα περιγραφεί ένας αλγόριθμος, αφού και αυτή η διαδικασία είναι μηχανιστική.

Τέλος, θα παρουσιαστεί ο αλγόριθμος συντακτικής ανάλυσης, ο οποίος χρησιμοποιεί τον πίνακα συντακτικής ανάλυσης και ένα αυτόματο στοίβας για να αποφανθεί αν μία συμβολοσειρά παράγεται ή όχι από τη γραμματική.

- *Κεφάλαιο 3: Η γλώσσα προγραμματισμού C-implement*

Η C-implement είναι μια μικρή, εκπαιδευτική, γλώσσα προγραμματισμού. Θυμίζει τη γλώσσα C, από την οποία αντλεί ιδέες και δομές, αλλά είναι αρκετά πιο μικρή, τόσο στις υποστηριζόμενες δομές, όσο φυσικά και σε προγραμματιστικές δυνατότητες.

Παρόλο που οι προγραμματιστικές της δυνατότητες είναι μικρές, η εκπαιδευτική αυτή γλώσσα προγραμματισμού περιέχει πλούσια στοιχεία και η κατασκευή του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον, αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται από δημοφιλείς γλώσσες προγραμματισμού, καθώς και κάποιες πρωτότυπες. Η C-implement υποστηρίζει δομές όπως οι δημοφιλείς while και η if-else, αλλά και τις πρωτότυπες και ενδιαφέρουσες στην υλοποίηση forcase και incase. Επίσης, υποστηρίζει συναρτήσεις και διαδικασίες, μετάδοση παραμέτρων με αναφορά και τιμή, αναδρομικές κλήσεις. Επιτρέπει φώλιασμα στη δήλωση συναρτήσεων και διαδικασιών.

Από την άλλη όμως πλευρά, η C-implement δεν προσφέρει βασικά προγραμματιστικά εργαλεία, όπως η δομή for ή τύπους δεδομένων όπως οι πραγματικοί αριθμοί και οι συμβολοσειρές ή οι πίνακες. Οι παραλείψεις αυτές έχουν γίνει για εκπαιδευτικούς λόγους, ώστε να απλουστευτεί η διαδικασία κατασκευής του μεταγλωττιστή. Είναι μία απλοποίηση που έχει να κάνει μόνο με τη μείωση των

γραμμών κώδικα που πρέπει να γραφεί και όχι με τη δυσκολία της κατασκευής ή την εκπαιδευτική αξία της διαδικασίας ανάπτυξης.

Στο κεφάλαιο αυτό θα οριστεί η γλώσσα *C-imple*. Θα παρουσιαστούν η δομή ενός προγράμματος σε γλώσσα *C-imple*, οι λεκτικές μονάδες που συνθέτουν ένα πρόγραμμα, οι λέξεις κλειδιά που είναι δεσμευμένες, οι τελεστές που είναι διαθέσιμοι, οι προγραμματιστικές δομές που υποστηρίζονται, οι κανόνες φωλιάσματος συναρτήσεων και διαδικασιών, οι κανόνες εμβέλειας και ο τρόπος περάσματος παραμέτρων.

Στη συνέχεια του κεφαλαίου θα παρουσιαστεί η γραμματική της γλώσσας σε μορφή EBNF και θα δοθούν κάποια προγράμματα-παραδείγματα συνταγμένα με τους κανόνες της *C-imple*.

Τέλος, θα δοθούν σύνδεσμοι και αναφορές προς γραμματικές δημοφιλών γλωσσών προγραμματισμού, όπως η C, η Pascal, η Java και η Python.

- **Κεφάλαιο 4: Λεκτικός αναλυτής**

Εδώ θα παρουσιαστεί η διαδικασία ανάπτυξης ενός λεκτικού αναλυτή. Αποτελεί την πρώτη φάση της μεταγλωττισης κατά την οποία το αρχικό πρόγραμμα αποσυντίθεται σε μία σειρά από λεκτικές μονάδες.

Θα μελετήσουμε τις βασικές έννοιες και τους κύριους μηχανισμούς που χρησιμοποιούνται στην ανάπτυξη του λεκτικού αναλυτή, έχοντας ως κύριο στόχο την κατανόηση όλων των λεπτομερειών της υλοποίησης. Έτσι, χωρίς να χρησιμοποιηθεί κανένα εργαλείο αυτοματοποιημένης ανάπτυξης, αλλά χρησιμοποιώντας μια γλώσσα προγραμματισμού υψηλού επιπέδου ως το μόνο μέσο υλοποίησης, θα δούμε βήμα-βήμα την ανάπτυξη ενός λεκτικού αναλυτή, χωρίς να θεωρήσουμε ως δεδομένες κάποιες ενότητες κώδικα.

Θα δούμε τη θέση που έχει ο λεκτικός αναλυτής μέσα στον κώδικα του μεταγλωττιστή και τι πληροφορία μεταφέρει στον συντακτικό αναλυτή. Θα μελετήσουμε και θα κατανοήσουμε την εσωτερική του λειτουργία και τον αλγόριθμο αναγνώρισης λεκτικών μονάδων που χρησιμοποιεί. Έτσι, θα κατασκευάσουμε και θα έχουμε ως βάση στη μελέτη μας ένα αυτόματο το οποίο θα αναγνωρίζει τις λεκτικές μονάδες ενός προγράμματος *C-imple*. Θα εξηγήσουμε αναλυτικά τις καταστάσεις και τις μεταβάσεις του και θα στοχεύσουμε σε λεπτομέρειες της υλοποίησης που θέλουν προσοχή. Τέλος, θα δούμε πώς ένας λεκτικός αναλυτής αντιμετωπίζει τα σχόλια, πώς ανιχνεύει και αντιμετωπίζει τα σφάλματα.

- **Κεφάλαιο 5: Συντακτικός αναλυτής**

Η συντακτική ανάλυση ελέγχει την ορθότητα ενός προγράμματος βάσει της γραμματικής της γλώσσας και προσφέρει το περιβάλλον το οποίο θα καθοδηγήσει την παραγωγή του κώδικα στις επόμενες φάσεις της ανάπτυξης. Εκκινώντας από τη γραμματική, ο συντακτικός αναλυτής διαβάζει μία-μία τις λεκτικές μονάδες από τον λεκτικό αναλυτή, ελέγχοντας ότι η σειρά με την οποία αυτές εμφανίζονται, είναι συνεπής με τη δεδομένη γραμματική.

Στο κεφάλαιο αυτό θα δούμε τον τρόπο με τον οποίο αναπτύσσεται ένας συντακτικός αναλυτής.

Η γραμματική της *C-imple* είναι γραμμένη, έτσι ώστε να είναι κατάλληλη για την υλοποίηση ενός συντακτικού αναλυτή με τη μέθοδο της αναδρομικής κατάβασης. Θα δούμε τι είναι η μέθοδος της αναδρομικής κατάβασης, πότε μία γραμματική είναι κατάλληλη για υλοποίηση με τη μέθοδο της αναδρομικής κατάβασης και πώς από μία τέτοια γραμματική μπορούμε να κατασκευάσουμε με τρόπο μηχανιστικό, έναν συντακτικό αναλυτή. Θα χρησιμοποιήσουμε παραδείγματα στα οποία θα φαίνεται, πώς από έναν κανόνα της γραμματικής της *C-imple* θα φτάσουμε σε κώδικα Python, ο οποίος θα αναγνωρίζει τις συμβολοσειρές που περιγράφει ένας κανόνας.

Στη φάση της συντακτικής ανάλυσης εντοπίζονται τα περισσότερα από τα σφάλματα που μπορεί να εντοπίσει ένας μεταγλωττιστής. Έτσι, η διαχείριση των σφαλμάτων θα αποτελέσει σημαντικό τμήμα του κεφαλαίου.

- **Κεφάλαιο 6: Παραγωγή ενδιάμεσου κώδικα**

Το πρόγραμμα σε τελική γλώσσα δεν παράγεται απευθείας από το αρχικό πρόγραμμα. Μεσολαβεί η μετατροπή του αρχικού προγράμματος σε μία ενδιάμεση αναπαράσταση, ένα βήμα που διευκολύνει τη διαδικασία, δίνει τη δυνατότητα βελτιστοποιητικών μετασχηματισμών και επιτρέπει την απεξάρτηση μεταξύ των φάσεων ανάπτυξης του μεταγλωττιστή.

Το κεφάλαιο εντάσσεται στη φάση της παραγωγής τελικού κώδικα, όπως και το επόμενο κεφάλαιο του βιβλίου αυτού. Για λόγους καλύτερης δόμησης η φάση της παραγωγής ενδιάμεσου κώδικα χωρίστηκε σε δύο κεφάλαια. Θα μπορούσε να έχει ως υπότιτλο *Ενδιάμεση αναπαράσταση, αριθμητικές και λογικές παραστάσεις*.

Στην αρχή θα παρουσιαστεί η ενδιάμεση αναπαράσταση που θα χρησιμοποιήσουμε. Πρόκειται για μία γλώσσα που βασίζεται σε τετράδες. Σκοπός της είναι να σχηματίσει ένα ενδιάμεσο επίπεδο ανάμεσα στην αρχική και την τελική γλώσσα. Η ενδιάμεση γλώσσα εξακολουθεί να είναι μία γλώσσα υψηλού επιπέδου, αφού περιέχει μεταβλητές και κλήσεις συναρτήσεων, οι δομές ελέγχου όμως της γλώσσας έχουν απλοποιηθεί και από δομές μιας δομημένης γλώσσας προγραμματισμού έχουν γίνει δομές απλών αλμάτων και αλμάτων υπό συνθήκη προς μία ετικέτα.

Στη συνέχεια θα δούμε πώς μεταφράζονται οι αριθμητικές και λογικές παραστάσεις. Με την ίδια φιλοσοφία, κάθε αριθμητική παράσταση θα αποσυντεθεί σε απλές αριθμητικές πράξεις (πρόσθεση, αφαίρεση, πολλαπλασιασμός και διαίρεση), καθεμία από τις οποίες έχει έναν τελεστή, δύο τελούμενα και εκχωρεί το αποτέλεσμα σε ένα άλλο τελούμενο.

Οι λογικές παραστάσεις αποσυντίθενται επίσης σε απλούστερες συγκρίσεις, καθεμία από τις οποίες δημιουργεί έναν αριθμό από άλματα και άλματα υπό συνθήκη. Κάθε άλμα υπό συνθήκη έχει ακριβώς έναν τελεστή σύγκρισης, δύο τελούμενα που συγκρίνονται μεταξύ τους και μία ετικέτα στην οποία θα μεταβεί ο έλεγχος, αν η συνθήκη είναι αληθής.

- **Κεφάλαιο 7: Ενδιάμεσος κώδικας για τις δομές της γλώσσας**

Μετά τη μετατροπή των αριθμητικών και λογικών παραστάσεων σε μία γλώσσα τετράδων, ακολουθεί η μετατροπή των δομών ελέγχου σε απλούστερες. Οι πολύπλοκες δομές μιας τυπικής γλώσσας δομημένου προγραμματισμού αποσυντίθενται σε απλές δομές, στις οποίες κυριαρχούν απλά και υπό συνθήκη άλματα.

Έτσι, στο κεφάλαιο αυτό θα περιγραφούν με λεπτομέρεια οι ενέργειες που σχετίζονται με τη μετατροπή σε ενδιάμεση γλώσσα δομών, όπως η εντολή απόφασης *if*, η εντολή επανάληψης *while* και η εντολή επιλογής *switchcase*. Θα μελετήσουμε και περισσότερο ενδιαφέρουσες στην υλοποίηση, αν και λιγότερο χρήσιμες στον προγραμματισμό, δομές, όπως οι εντολές απόφασης-επανάληψης *forcase* και *incase*.

Στη συνέχεια ακολουθεί η μετάφραση σε ενδιάμεση αναπαράσταση των συναρτήσεων και των διαδικασιών. Για τις συναρτήσεις και τις διαδικασίες δεν επιτυγχάνεται στη φάση αυτή ανάλογα σημαντική αποσύνθεση των δομών τους. Οι κλήσεις τους μετατρέπονται κατάλληλα, ώστε να συμβολίστούν σε ενδιάμεση γλώσσα. Η περαιτέρω αποδόμηση της πολυπλοκότητάς τους μεταφέρεται για τη φάση της παραγωγής του τελικού κώδικα.

Ένα ολοκληρωμένο, απλό παράδειγμα, μετατροπής κώδικα σε ενδιάμεση αναπαράσταση θα παρουσιαστεί με αναλυτικά βήματα, ώστε να καλύψει πιθανά σημεία που τυχόν δυσκόλευγαν.

Τέλος, θα συζητήσουμε το πώς θα μπορούσαμε, με βάση τις αρχές του αντικειμενοστραφούς προγραμματισμού, να ιεραρχήσουμε πιθανές κλάσεις που εμφανίζονται στη σχεδίαση του μεταγλωττιστή.

- **Κεφάλαιο 8: Πίνακας συμβόλων**

Στον πίνακα συμβόλων αποθηκεύουμε, κατά τη φάση της συντακτικής ανάλυσης και της παραγωγής ενδιάμεσου κώδικα, πληροφορίες σχετικά με τα συμβολικά ονόματα που χρησιμοποιούμε σε ένα πρόγραμμα: μεταβλητές, συναρτήσεις, διαδικασίες, παράμετροι, συμβολικές σταθερές κλπ. Η πληροφορία αυτή χρησιμοποιείται στη φάση της σημασιολογικής ανάλυσης αλλά και της παραγωγής του τελικού κώδικα.

Θα εξετάσουμε τη δομή που έχει ένας τέτοιος πίνακας, καθώς και το πώς εισάγουμε και αφαιρούμε εγγραφές σε αυτόν, ώστε σε κάθε σημείο της μετάφρασης ο πίνακας να έχει πρόσβαση στις εγγραφές, και μόνο σε αυτές, στις οποίες εκείνη τη στιγμή της μετάφρασης έχει δικαίωμα να προσπελάσει το υπό μετάφραση πρόγραμμα.

Για να δούμε πώς συμπληρώνουμε την πληροφορία που απαιτείται σε κάθε εγγραφή, θα θυμηθούμε τι είναι το εγγράφημα δραστηριοποίησης και πώς υπολογίζουμε το πού και πώς βρίσκεται μια μεταβλητή μέσα σε αυτό.

Θα απαριθμήσουμε και θα περιγράψουμε, με λεπτομέρεια, κάθε λειτουργία του πίνακα συμβόλων.

’ Θα αναζητήσουμε τις ευκαιρίες για αντικειμενοστραφή σχεδίαση και θα προτείνουμε έναν τρόπο ιεράρχησης κλάσεων με βάση την αντικειμενοστραφή φιλοσοφία.

Τέλος, θα δούμε ένα παράδειγμα μετάφρασης ενός προγράμματος *C-imperative*, με το οποίο θα μπορέσετε να εμπεδώσετε όλες τις έννοιες του κεφαλαίου αυτού.

- **Κεφάλαιο 9: Παραγωγή τελικού κώδικα**

Η παραγωγή του τελικού κώδικα είναι το στάδιο της μεταγλώττισης, όπου παράγεται ο κώδικας σε γλώσσα μηχανής. Η φάση της παραγωγής του τελικού κώδικα έχει χωριστεί σε δύο κεφάλαια στο βιβλίο αυτό, λόγω της έκτασής τους. Σε αυτό το κεφάλαιο, που είναι το πρώτο, παρουσιάζεται ότι αφορά την παραγωγή του τελικού κώδικα, αλλά δεν εντάσσεται στη θεματολογία της μεταγλώττισης συναρτήσεων και διαδικασιών. Το δεύτερο κεφάλαιο, αυτό που ακολουθεί, ασχολείται κυρίως με τη μεταγλώττιση συναρτήσεων και διαδικασιών.

Θα μελετήσουμε τα κυριότερα σημεία της μετάφρασης κώδικα από ενδιάμεση σε τελική γλώσσα. Θα δούμε τον τρόπο με τον οποίο παράγεται η γλώσσα μηχανής για καθεμία από τις εντολές της γλώσσας που χρησιμοποιούμε για την ενδιάμεση αναπαράσταση. Θα δώσουμε ιδιαίτερη έμφαση στην απεικόνιση των μεταβλητών στη μνήμη και στον τρόπο αναζήτησής τους, ανάλογα με το είδος της κάθε μεταβλητής. Ο τρόπος αποθήκευσης μιας μεταβλητής και ο τρόπος πρόσβασης διαφέρουν αρκετά αν έχουμε τοπικές μεταβλητές, καθολικές μεταβλητές ή μεταβλητές που ανήκουν στους προγόνους μιας συνάρτησης ή διαδικασίας, χωρίς να είναι καθολικές.

Στην αρχή του κεφαλαίου παρουσιάζεται η γλώσσα μηχανής του επεξεργαστή RISC-V. Η περιγραφή περιορίζεται στα στοιχεία της γλώσσας που θα φανούν χρήσιμα στην παραγωγή του κώδικα. Στη συνέχεια, θα περιγράψουμε τις βοηθητικές συναρτήσεις `gnlncode()` για την πρόσβαση σε μεταβλητές που ανήκουν σε προγόνους, `loadvr()` για τη φόρτωση μεταβλητών από τη μνήμη σε καταχωρητές και `storevr()` για την αποθήκευση μεταβλητών στη μνήμη.

Τέλος, περιγράφεται η παραγωγή τελικού κώδικα για τις εκχωρήσεις, τις αριθμητικές πράξεις, τις διακλαδώσεις, την αρχή και το τέλος προγράμματος και τις εντολές εισόδου-εξόδου.

- **Κεφάλαιο 10: Πέρασμα παραμέτρων, κλήση συναρτήσεων και διαδικασιών στον τελικό κώδικα**

Στο δεύτερο αυτό κεφάλαιο αφιερωμένο στον τελικό κώδικα, θα περιγράψουμε με λεπτομέρεια πώς γίνονται η κλήση μιας συνάρτησης ή διαδικασίας, το πέρασμα παραμέτρων με αναφορά και τιμή, η επιστροφή αποτελέσματος των συναρτήσεων, αλλά και η επιστροφή του ελέγχου ροής όταν μία συνάρτηση ή διαδικασία τερματιστεί.

Πρώτα θα δούμε τις εντολές που θα δημιουργηθούν προκειμένου να περάσουμε μία παράμετρο με τιμή, αντιγράφοντας στο νέο εγγράφημα δραστηριοποίησης την τιμή της. Στη συνέχεια θα κάνουμε τις αντίστοιχες ενέργειες, ώστε να αντιγραφεί στο νέο εγγράφημα δραστηριοποίησης η διεύθυνση μίας μεταβλητής που περνάει με αναφορά. Η επιστροφή τιμής μιας συνάρτησης ακολουθεί έναν παρόμοιο μηχανισμό με τον μηχανισμό περάσματος παραμέτρου με αναφορά.

Η κλήση μιας συνάρτησης ή διαδικασίας προϋποθέτει την ενεργοποίηση του νέου εγγραφήματος δραστηριοποίησης και της μεταβίβασης του ελέγχου στη νέα συνάρτηση. Μετά το πέρας της εκτέλεσης, ο έλεγχος πρέπει να επιστρέψει στην καλούσα συνάρτηση ή διαδικασία και να ενεργοποιηθεί ξανά το εγγράφημα δραστηριοποίησης της καλούσας.

- **Κεφάλαιο 11: Τεχνικές βελτιστοποίησης κώδικα**

Η βελτιστοποίηση του κώδικα είναι πολύ σημαντικό τμήμα της διαδικασίας της μεταγλώττισης. Ο κώδικας, ο οποίος παράγεται τόσο στη φάση της παραγωγής ενδιάμεσου, όσο και στη φάση της παραγωγής τελικού κώδικα, έχει πολλά περιθώρια βελτίωσης, όσον αφορά την ταχύτητα του τελικού εκτελέσιμου και το μέγεθός του. Θα ασχοληθούμε με δυνατότητες που μας δίνονται για βελτιστοποίηση, τόσο στον ενδιάμεσο όσο και στον τελικό κώδικα.

Στο κεφάλαιο αυτό θα παρουσιαστούν, αρχικά, μετασχηματισμοί που ανήκουν στην κατηγορία των τεχνικών αποτίμησης, όπως η διάδοση σταθερών, ο υπολογισμός σταθερών εκφράσεων, η διάδοση αντιγράφων, η διάδοση υποκατάστατων, οι αλγεβρικές απλοποιήσεις και ο υποβιβασμός ισχύος. Στη συνέχεια θα παρουσιαστούν μετασχηματισμοί που ανήκουν στην κατηγορία της απαλοιφής κώδικα όπως η απαλοιφή μη προσβάσιμου κώδικα, η απαλοιφή μη χρήσιμου κώδικα, η απαλοιφή μη χρήσιμων μεταβλητών και η απαλοιφή κοινών υποεκφράσεων.

Σε μία άλλη κατηγορία μετασχηματισμών, στους βελτιστοποιητικούς μετασχηματισμούς στις κλήσεις συναρτήσεων και διαδικασιών, εντάσσονται η βελτιστοποίηση κλήσης σε διαδικασία φύλλο, η εκχώρηση καταχωρητών κατά τις κλήσεις, η ενσωμάτωση διαδικασίας, η απαλοιφή αναδρομής ουράς και η απομνημόνευση κλήσεων. Τέλος, στην πολύ σημαντική κατηγορία των μετασχηματισμών βρόχων θα εξετάσουμε τον υποβιβασμό ισχύος βασιζόμενο σε επαναλήψεις βρόχου, την απαλοιφή επαγωγικών μεταβλητών, τη μετακίνηση κώδικα ανεξάρτητου από την εκτέλεση του βρόχου, την εναλλαγή σε βρόχο, την κυκλική συρρίκνωση, τον διαχωρισμό και την ένωση βρόχων.

- **Κεφάλαιο 12: Εργαλεία αυτοματοποιημένης ανάπτυξης μεταγλωττιστών**

Η αυτοματοποιημένη ανάπτυξη τμημάτων ενός μεταγλωττιστή είναι μία συνηθισμένη προσέγγιση και έχουν κατασκευαστεί μετα-εργαλεία κατάλληλα για τον σκοπό αυτόν. Τα εργαλεία αυτόματης ανάπτυξης επικεντρώνονται στη λεκτική και τη συντακτική ανάλυση, αφού σε αυτές τις φάσεις έχουμε αρκετά αυτοματοποιημένες διαδικασίες. Η είσοδος σε αυτά περιγράφεται με κανονικές εκφράσεις για τον λεκτικό αναλυτή και με μία γραμματική χωρίς συμφραζόμενα για τον συντακτικό αναλυτή.

Θα ασχοληθούμε με δύο τέτοια εργαλεία. Στη βιβλιογραφία υπάρχει εκτενές υλικό που τα περιγράφει αναλυτικά και ο αναγνώστης μπορεί να απευθυνθεί εκεί για μία πλήρη περιγραφή τους. Στο κεφάλαιο αυτό θα γίνει μία σύντομη παρουσίαση, μέσα από παραδείγματα για κάθε περίπτωση. Ο αναγνώστης θα διδαχθεί τις βασικές δυνατότητες των εργαλείων αυτών, ώστε να γνωρίζει πότε πρέπει να τα επιλέξει, αλλά και εύκολα να εμπλουτίσει τις γνώσεις του, αν το θελήσει ή το χρειαστεί.

Η πρώτη ομάδα εργαλείων αποτελείται από το ζευγάρι *lex* και *yacc/bison*. Είναι η κλασικότερη προσέγγιση στην αυτοματοποιημένη κατασκευή λεκτικών και συντακτικών αναλυτών, η οποία απλοποιεί πολύ το έργο του προγραμματιστή. Ένα παράδειγμα λεκτικής και συντακτικής ανάλυσης μιας απλής γλώσσας προγραμματισμού με ορισμούς και εκχωρήσεις μεταβλητών θα μας φανεί πολύ χρήσιμο στην κατανόηση.

Το δεύτερο μετα-εργαλείο είναι το *ANTLR*, το οποίο αποτελεί μια εξελιγμένη και πιο σύγχρονη μορφή ενός τέτοιου εργαλείου. Το ίδιο παράδειγμα, εμπλουτισμένο, θα παρουσιαστεί για το *ANTLR*, ώστε να μπορεί να γίνει σύγκριση ανάμεσα στα εργαλεία.

- *Παραρτήματα*

Στο τέλος του βιβλίου ο αναγνώστης μπορεί να βρει τρία παραρτήματα:

- Στο πρώτο παράρτημα παρουσιάζεται ένα πλήρες παράδειγμα μεταγλώττισης, στο οποίο ο αναγνώστης μπορεί να δει ολοκληρωμένη τη διαδικασία και ιδιαίτερα τη χρονική αλληλουχία των πραγμάτων, κάτι που δεν είναι τόσο εύκολο να φανεί, όταν κάποιος εξετάζει τις φάσεις χωριστά την καθεμία.
- Στο δεύτερο παράρτημα δίνεται ο κώδικας του λεκτικού και συντακτικού αναλυτή, ο οποίος περιγράφεται στο κεφάλαιο 12 και έχει υλοποιηθεί με τα μετα-εργαλεία *lex/yacc*.
- Στο τρίτο παράρτημα θα παρουσιαστεί ένα παράδειγμα ανάπτυξης ενός μορφωτή προγραμμάτων αναπτυγμένου με το μετα-εργαλείο *ANTLR*, ως συνέχεια της παρουσίασής του στο κεφάλαιο 12.



## ΚΕΦΑΛΑΙΟ 1

---

# ΤΥΠΙΚΕΣ ΓΡΑΜΜΑΤΙΚΕΣ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑ ΜΕΤΑΓΛΩΤΤΙΣΤΩΝ

---

### Σύνοψη:

Το βιβλίο αυτό θεωρεί ως δεδομένο το θεωρητικό υπόβαθρο που αφορά τη θεωρία αυτομάτων, τυπικών γραμματικών και τυπικών γλωσσών. Θεωρεί, όμως, χρήσιμο να συνοψίσει τις βασικές έννοιες και τους σημαντικότερους ορισμούς, οι οποίοι θα φανούν χρήσιμοι στη συνέχεια. Έτσι, στην αρχή του κεφαλαίου παρουσιάζονται οι ορισμοί των τυπικών γραμματικών και των τυπικών γλωσσών, αλλά εισάγονται και άλλες χρήσιμες έννοιες, όπως το τερματικό και μη τερματικό σύμβολο, ο κανόνας παραγωγής, η πρόταση και ο προτασιακός τύπος.

Στη συνέχεια παρουσιάζεται η οργάνωση των τυπικών γραμματικών, των τυπικών γλωσσών και των αυτομάτων με βάση την *ιεραρχία Chomsky*. Ο Noam Chomsky ιεράρχησε τις τυπικές γραμματικές, τις τυπικές γλώσσες και τα αυτόματα με βάση την περιγραφική ισχύ τους. Κάθε επίπεδο μπορεί να παράγει ένα σύνολο από συμβολοσειρές, το οποίο είναι υποσύνολο του συνόλου συμβολοσειρών που μπορεί να παράγει το ανώτερό του επίπεδο.

Αφού παρουσιαστεί αναλυτικά κάθε επίπεδο της ιεραρχίας Chomsky, θα γίνει αντιστοίχιση του κάθε επίπεδου με τις διάφορες φάσεις ανάπτυξης ενός μεταγλωττιστή. Οι τυπικές γραμματικές χρησιμοποιούνται στη φάση της λεκτικής και της συντακτικής ανάλυσης, ενώ συνδέονται και με τη σημασιολογική ανάλυση.

Ενισχύοντας το θεωρητικό υπόβαθρο του αναγνώστη, θα εισαχθούν μερικές ακόμα χρήσιμες έννοιες, όπως η γραμματική LL(1). Μία γραμματική LL(1) επιτρέπει την επιλογή του κανόνα που θα ενεργοποιηθεί με βάση το επόμενο τερματικό σύμβολο στην υπό αναγνώριση συμβολοσειρά. Με τη βοήθεια μιας γραμματικής LL(1) μπορούμε εύκολα να κατασκευάσουμε συντακτικούς αναλυτές.

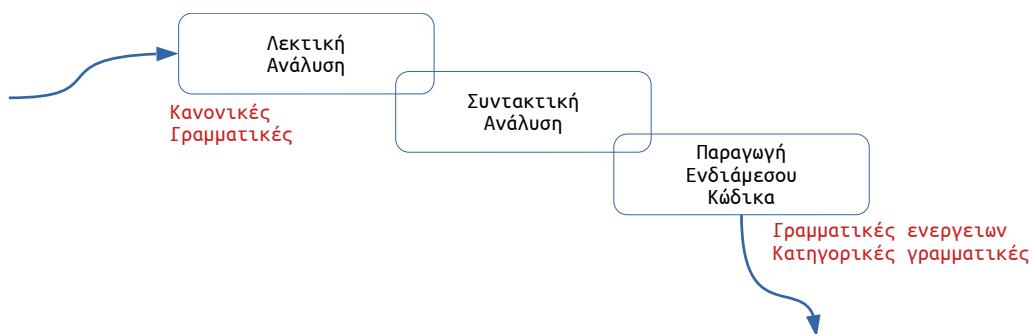
Τέλος, θα δούμε τον συμβολισμό EBNF. Ο συμβολισμός αυτός είναι πιο περιεκτικός από τον συμβολισμό που χρησιμοποιούμε για τις απλές γραμματικές και παρουσιάζει κάποια πρόσθετα πλεονεκτήματα, τα οποία διευκολύνουν στην ανάπτυξη ενός μεταγλωττιστή.

### Προαπαιτούμενη γνώση:

- θεωρία αυτομάτων
- θεωρία τυπικών γραμματικών
- θεωρία τυπικών γλωσσών

Mία τυπική γραμματική αποτελείται από ένα αλφάβητο από τερματικά σύμβολα, ένα αρχικό σύμβολο και ένα σύνολο από κανόνες που περιγράφουν τις συμβολοσειρές που είναι νόμιμο να σχηματιστούν από το αλφάβητο αυτό ή, με άλλα λόγια, τις συμβολοσειρές που ανήκουν στη γλώσσα που περιγράφει.

Οι τυπικές γραμματικές αποτελούν ένα χρήσιμο εργαλείο για τη σχεδίαση και ανάπτυξη μεταγλωττιστών. Οι φάσεις στις οποίες οι τυπικές γραμματικές εμπλέκονται άμεσα στη σχεδίαση ενός μεταγλωττιστή είναι οι φάσεις της λεκτικής και της συντακτικής ανάλυσης. Κατά τη λεκτική ανάλυση χρησιμοποιούμε κανονικές γραμματικές ή τις ισοδύναμες και περισσότερο αναγνώσιμες κανονικές εκφράσεις, για να περιγράψουμε τις λεκτικές μονάδες της αρχικής γλώσσας. Στη φάση της συντακτικής ανάλυσης εκμεταλλευόμαστε την αυξημένη περιγραφικότητα και συνάμα επιθυμητή απλότητα των τυπικών γραμματικών χωρίς συμφραζόμενα, για να περιγράψουμε τη σύνταξη της αρχικής γλώσσας, όπως φαίνεται στο σχήμα 1.1. Μία γραμματική χωρίς συμφραζόμενα είναι η γραμματική της *C-imple* που περιγράφεται στο κεφάλαιο 2.



Σχήμα 1.1: Γραμματικές στη λεκτική και συντακτική ανάλυση.

Οι τυπικές γραμματικές σχετίζονται και με τη φάση της σημασιολογικής ανάλυσης, τουλάχιστον θεωρητικά. Για να περιγράψουμε τις ιδιότητες της αρχικής γλώσσας που συνήθως ελέγχουμε κατά τη σημασιολογική ανάλυση, απαιτείται να εμπλέξουμε στην περιγραφή μία γραμματική με συμφραζόμενα. Όμως, λόγω της αυξημένης πολυπλοκότητας που εμφανίζει η περιγραφή με μιας τέτοια γραμματική, στην πράξη επιλέγουμε να υλοποιήσουμε τη σημασιολογική ανάλυση, βασισμένοι σε ευρετικές τεχνικές προσαρμοσμένες στις εκάστοτε ανάγκες.

Οι γραμματικές ενεργειών και οι κατηγορικές γραμματικές χρησιμοποιούνται για να αναπαραστήσουν το σχέδιο ενδιάμεσου κώδικα. Σε κάθε σημείο της γραμματικής μπορούμε να αντιστοιχίσουμε μία ενέργεια, οδηγούμενοι έτσι σε συντακτικά καθοδηγούμενη μετάφραση.

Θα μελετήσουμε και θα εμβαθύνουμε όσο χρειάζεται στους τύπους γραμματικών τους οποίους θα χρειαστούμε παρακάτω στη σχεδίαση και την υλοποίηση του μεταγλωττιστή της *C-imple*. Θα επιχειρήσουμε μία περισσότερη τυπική περιγραφή τους, αρχίζοντας από τον ορισμό μιας τυπικής γραμματικής.

Χρήσιμες πηγές για εμπλουτισμό των γνώσεων και ενδυνάμωση του θεωρητικού υποβάθρου που σχετίζεται με το υλικό του κεφαλαίου αυτού παρατίθενται στη βιβλιογραφία, στο τέλος του κεφαλαίου [1, κεφ.3-7], [2, 3, κεφ.1-2], [4, κεφ.1-3].

## 1.1 Ορισμοί

Μία τυπική γραμματική  $G$  ορίζεται [5, 6] από την τετράδα  $G = (N, T, S, P)$ , όπου:

- $N$  είναι το σύνολο των μη τερματικών συμβόλων.
- $T$  είναι το σύνολο των τερματικών συμβόλων.
- $S$  είναι το αρχικό σύμβολο, όπου  $S \in N$ .
- $P$  είναι οι κανόνες παραγωγής.

Το σύνολο των τερματικών συμβόλων αποτελεί το αλφάβητο της γραμματικής. Τα τερματικά σύμβολα είναι σύμβολα που εμφανίζονται στις παραγωγές της γραμματικής ή, αν μιλάμε για μεταγλωττιστές, στο αρχικό πρόγραμμα. Τα τερματικά σύμβολα δεν αντικαθίστανται από άλλα κατά την παραγωγή των συμβολοσειρών.

Μη τερματικά σύμβολα είναι τα σύμβολα τα οποία αντικαθίστανται κατά τη διαδικασία της παραγωγής.

Αρχικό σύμβολο είναι το μη τερματικό σύμβολο από το οποίο ξεκινά κάθε παραγωγή της γραμματικής. Ισχύει ότι:  $S \in N$ .

Ο συμβολισμός  $\underset{G}{\overset{*}{\Rightarrow}}$  χρησιμοποιείται για να δείξει ότι κάτι παράγεται από τη γραμματική  $G$  σε μηδέν ή περισσότερα βήματα.

Προτασιακός τύπος είναι κάθε παραγωγή της γραμματικής που προκύπτει σε πεπερασμένο αριθμό βημάτων:

$$\alpha \in (N \cup T)^*, \quad S \underset{G}{\overset{*}{\Rightarrow}} \alpha.$$

Πρόταση είναι κάθε παραγωγή της γραμματικής που προκύπτει σε πεπερασμένο αριθμό βημάτων και περιέχει μόνο τερματικά σύμβολα:

$$\alpha \in T^*, \quad S \underset{G}{\overset{*}{\Rightarrow}} \alpha.$$

Κάθε κανόνας παραγωγής περιγράφει μία αντικατάσταση συμβόλων, ώστε να παραχθούν νέες συμβόλοσειρές. Κάθε κανόνας παραγωγής έχει τη μορφή:

$$P : \alpha \rightarrow \beta$$

όπου  $\alpha, \beta \in (N \cup T)^*$ , αλλά τουλάχιστον ένα σύμβολο από το  $\alpha$  πρέπει υποχρεωτικά να ανήκει στο  $N$ .

Ισχύει, επίσης, ότι τα σύνολα των τερματικών και μη τερματικών συμβόλων πρέπει να είναι ξένα μεταξύ τους:

$$N \cap T = \emptyset.$$

Ακολουθεί ένα παράδειγμα γραμματικής η οποία περιγράφει μη αρνητικές ακέραιες σταθερές. Η γραμματική δεν επιτρέπει μία τέτοια σταθερά να ξεκινάει με το τερματικό σύμβολο 0, εκτός αν η σταθερά αυτή είναι το 0. Τα τερματικά σύμβολα σημειώνονται με πράσινο, ενώ τα συντακτικά σύμβολα της γραμματικής με κόκκινο. Πέρα από τα χρώματα, για ακόμα πιο ευκρινή διαχωρισμό και για να ακολουθήσουμε την κοινή πρακτική, θα συμβολίσουμε με κεφαλαία γράμματα τα μη τερματικά σύμβολα και με μικρά τα τερματικά.

$$\begin{array}{l} S \rightarrow 0 \\ | \\ D' D \\ D' \rightarrow 1 \mid 2 \mid 3 \mid \dots \mid 9 \\ D \rightarrow ( \ 0 \mid D' ) \ D \\ D \rightarrow \epsilon \end{array}$$

Ο χαρακτήρας  $\epsilon$  συμβολίζει την κενή συμβολοσειρά.

Η ακόλουθη γραμματική εξασφαλίζει την ορθή σύνταξη μιας μαθηματικής παράστασης, στην οποία ο αριθμός των παρενθέσεων που ανοίγουν είναι ίδιος με τον αριθμό των παρενθέσεων που κλείνουν. Η γραμματική δεν εξασφαλίζει την προτεραιότητα των πράξεων ανάμεσα σε προσθέσεις και αφαιρέσεις και σε πολλαπλασιασμούς και διαιρέσεις:

$$\begin{aligned} S &\rightarrow S \ ( \ + \ | \ - \ | \ * \ | \ / \ ) \ S \\ S &\rightarrow ( \ S \ ) \\ S &\rightarrow \dots \end{aligned}$$

Ας δούμε, τώρα τον ορισμό μιας τυπικής γλώσσας. Κάθε γραμματική  $G = (T, N, P, S)$  ορίζει μία γλώσσα:  $L(G) \subseteq T^*$

Λέμε ότι η γραμματική παράγει τη γλώσσα και ορίζεται ως:

$$L(G) = \alpha \in T^*, \quad S \xrightarrow{+} \alpha$$

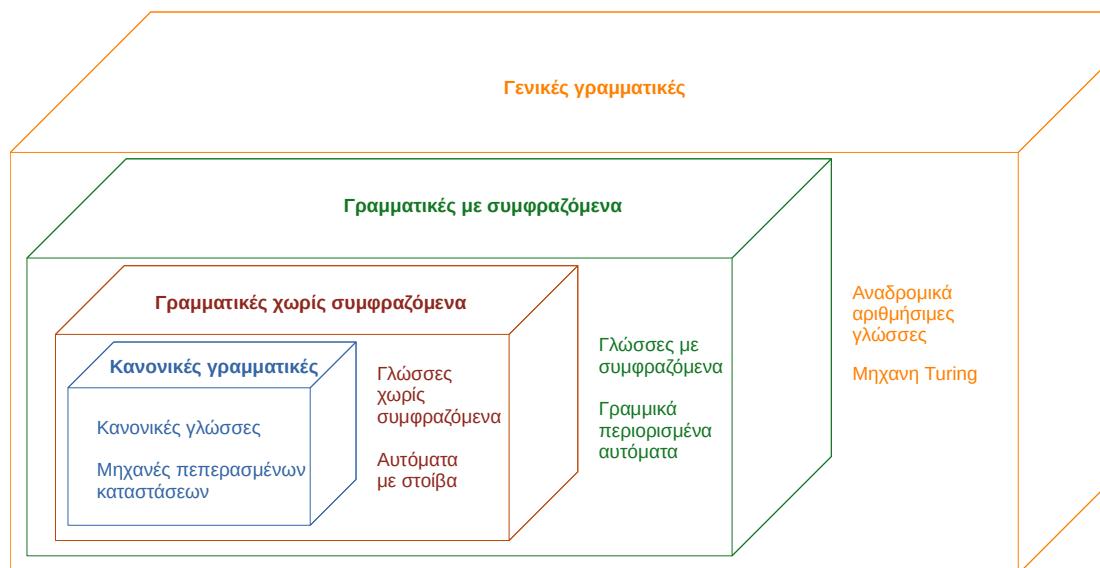
Το σύμβολο  $\xrightarrow{+}$  το χρησιμοποιούμε για να δείξουμε ότι μία συμβολοσειρά παράγεται σε ένα ή περισσότερα βήματα.

Μία γλώσσα λέγεται τυπική γλώσσα, όταν υπάρχει γραμματική που την παράγει.

## 1.2 Ιεραρχία Chomsky

Στην εργασία [5] o Noam Chomsky το 1956 πρότεινε μία ιεράρχηση των γραμματικών, που έχει επικρατήσει να ονομάζεται Ιεραρχία Chomsky (Chomsky hierarchy). Με βάση αυτή την ιεράρχηση, οι γραμματικές ταξινομούνται σε τέσσερις κατηγορίες. Κάθε κατηγορία έχει διαφορετική περιγραφική ικανότητα και διαφορετική πολυπλοκότητα στη διαχείρισή της. Όσο μεγαλύτερη περιγραφική ικανότητα έχει μία γραμματική, τόσο δυσκολότερη είναι η κατασκευή ενός συντακτικού αναλυτή για τη γραμματική αυτή και τόσο περισσότερος χρόνος απαιτείται για να ολοκληρωθεί η συντακτική ανάλυση, χρόνος που για κάποιες κατηγορίες γραμματικών είναι απαγορευτικός.

Στην ιεραρχία Chomsky κάθε τύπος γραμματικής αντιστοιχίζεται σε έναν τύπο τυπικής γλώσσας και σε έναν τύπο αυτομάτου.



Σχήμα 1.2: Η ιεραρχία του Chomsky.

Η ιεραρχία του Chomsky φαίνεται στο σχήμα 1.2. Κάθε γραμματική υψηλότερα στην ιεραρχία μπορεί να περιγράψει όλες τις συμβολοσειρές που παράγονται από γραμματικές χαμηλότερα στην ιεραρχία. Σε κάθε κύβο, πέρα από τη γραμματική που αντιστοιχεί σε αυτόν και φαίνεται με έντονα γράμματα στο πάνω μέρος του, έχουν σημειωθεί και η ονομασία της γλώσσας την οποία παράγει η γραμματική αυτή, καθώς και ο τύπος μηχανής που με τον οποίο ισοδυναμεί.

Έτσι, χαμηλότερα στην ιεραρχία του Chomsky είναι οι κανονικές γραμματικές, οι οποίες παράγουν τις

κανονικές γλώσσες και ισοδυναμούν με τα αυτόματα πεπερασμένων καταστάσεων. Αμέσως υψηλότερα έχουμε τις γραμματικές χωρίς συμφραζόμενα, οι οποίες παράγουν τις γλώσσες χωρίς συμφραζόμενα και ισοδυναμούν με αυτόματα με στοίβα. Παραπάνω στην ιεραρχία βρίσκονται οι γραμματικές με συμφραζόμενα, οι οποίες παράγουν τις γλώσσες με συμφραζόμενα και αντιστοιχίζονται στα γραμμικά περιορισμένα αυτόματα. Τέλος, υψηλότερα στην ιεραρχία βρίσκονται οι γενικές γραμματικές, οι οποίες παράγουν τις αναδρομικά απαριθμίσιμες γλώσσες και αντιστοιχίζονται στις μηχανές Turing.

Στη συνέχεια θα δούμε και θα συζητήσουμε τον κάθε τύπο γραμματικής χωριστά, ξεκινώντας από τις γραμματικές με τη μικρότερη περιγραφική ικανότητα και τη μικρότερη πολυπλοκότητα, οι οποίες βρίσκονται χαμηλότερα στην ιεραρχία και έχουν τις περισσότερες πρακτικές εφαρμογές.

### 1.2.1 Κανονικές γραμματικές

Οι κανονικές γραμματικές ή γραμματικές τύπου 3 έχουν τη μορφή:

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow aB \end{aligned}$$

Τα κεφαλαία γράμματα αναπαριστούν μη τερματικά σύμβολα ( $A, B \in N$ ), ενώ τα μικρά γράμματα αναπαριστούν τερματικά σύμβολα ( $a \in T$ ).

Οι κανονικές γραμματικές παράγουν κανονικές γλώσσες. Ένα παράδειγμα κανονικής γλώσσας είναι το ακόλουθο:

$$L = a^n, \quad n \geq 0.$$

Η περιγραφική ισχύς των κανονικών γραμματικών ισοδυναμεί με την περιγραφική ισχύ των αυτομάτων πεπερασμένων καταστάσεων.

Ένα παράδειγμα γραμματικής τύπου 3 από τον χώρο των μεταφραστών είναι η αναγνώριση συμβολοσειράς που εκκινεί με γράμμα και στη συνέχεια μπορεί να ακολουθήσει γράμμα ή ψηφίο. Τέτοιες συμβολοσειρές αναπαριστούν συνήθως ονομασίες μεταβλητών, συμβολικών σταθερών, συναρτήσεων ή διαδικασιών. Ο μεταγλωττιστής αναγνωρίζει τέτοιες συμβολοσειρές κατά τη φάση της λεκτικής ανάλυσης.

```
S → a A      # first symbol is a letter
S → b A
...
S → z A
A → a A      # a letter
A → b A
...
A → z A
A → 0 A      # or a digit follows
A → 1 A
...
A → 9 A
A → a         # last symbol is a letter
A → b
...
A → z
A → 0         # or a digit
A → 1
...
A → 9
```

Οι κανονικές εκφράσεις αποτελούν έναν τρόπο περιγραφής των κανονικών γλωσσών. Στις κανονικές εκφράσεις χρησιμοποιείται το άστρο του Kleene (\*). Ο Stephen Kleene, μαθηματικός, το 1951, εισήγαγε τον συμβολισμό αυτόν ο οποίος χρησιμοποιείται εκτενώς σήμερα [7]. Θα χρησιμοποιήσουμε το άστρο του

Kleene ώστε να περιγράψουμε την παραπάνω γραμματική.

$$\begin{aligned} S &\rightarrow L \ ( D \mid L )^* \\ L &\rightarrow a \mid b \mid c \mid \dots \mid z \\ D &\rightarrow \emptyset \mid 1 \mid 2 \mid \dots \mid 9 \end{aligned}$$

Η βελτίωση της αναγνωσιμότητας είναι αξιοσημείωτη.

### 1.2.2 Γραμματικές χωρίς συμφραζόμενα

Οι γραμματικές χωρίς συμφραζόμενα ή γραμματικές τύπου 2 έχουν τη μορφή:

$$A \rightarrow \alpha$$

Τα κεφαλαία γράμματα αναπαριστούν τερματικά σύμβολα:  $A \in N$ . Τα μικρά ελληνικά γράμματα αποτελούν συμβολοσειρές που περιέχουν τερματικά και μη τερματικά σύμβολα:  $\alpha \in (T \cup N)^*$ .

Οι γραμματικές χωρίς συμφραζόμενα παράγουν γλώσσες χωρίς συμφραζόμενα. Ένα παράδειγμα γλώσσας χωρίς συμφραζόμενα είναι το ακόλουθο:

$$L = \alpha^n \beta^n, \quad n > 0.$$

Η περιγραφική τους ισχύς ισοδυναμεί με αυτή των αυτομάτων στοίβας.

Ένα παράδειγμα γραμματικής τύπου 2 από τον χώρο των μεταφραστών είναι η γραμματική που δόθηκε ως περιγραφή της γλώσσας *C-imple* (δείτε παρακάτω, κεφάλαιο 3).

Ένα ακόμα παράδειγμα από τον χώρο των μεταγλωττιστών είναι η αναγνώριση ορισμού πινάκων σε μία γλώσσα που μοιάζει με τη C:

$$\begin{aligned} S &\rightarrow \text{Id Tail} \\ \text{Id} &\rightarrow \text{Letter } ( \text{Digit} \mid \text{Letter} )^* \\ \text{Int} &\rightarrow ( 1 \mid 2 \mid \dots \mid 9 ) \text{ Digit}^* \\ \text{Tail} &\rightarrow [ (\text{Id} \mid \text{Int}) ] \text{ Tail} \\ \text{Tail} &\rightarrow \varepsilon \\ \text{Letter} &\rightarrow a \mid b \mid c \mid \dots \mid z \\ \text{Digit} &\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9 \end{aligned}$$

Το αρχικό σύμβολο  $S$  παράγει το τερματικό σύμβολο  $\text{Id}$ , το οποίο είναι το όνομα του πίνακα, και στη συνέχεια το μη τερματικό σύμβολο  $\text{Tail}$ , το οποίο περιγράφει τις διαστάσεις του πίνακα. Αναγνωρίζονται δηλαδή συμβολοσειρές όπως η ακόλουθη:  $c[2][a]...[k]$ .

Στη γραμματική το  $\text{Id}$  παράγει προτάσεις που ξεκινούν από γράμμα και συνεχίζονται από κανένα ή περισσότερα γράμματα ή ψηφία. Το  $\text{Tail}$  παράγει συμβολοσειρές που ξεκινούν με το άνοιγμα αγκύλης, ακολουθεί ένα  $\text{Id}$  ή ένας θετικός ακέραιος αριθμός ( $\text{Int}$ ) και ολοκληρώνονται με το κλείσιμο της αγκύλης. Παρατηρήστε ότι ο αριθμός  $\text{INT}$  ξεκινάει από μη μηδενικό ψηφίο και συνεχίζει με κανένα ή περισσότερα ψηφία. Η αναδρομική κλήση της  $\text{Tail}$  που τερματίζεται με τον κανόνα στον οποίο το  $\text{Tail}$  παράγει το κενό, είναι υπεύθυνη για την περιγραφή των διαστάσεων του πίνακα.

### 1.2.3 Γραμματικές με συμφραζόμενα

Οι γραμματικές με συμφραζόμενα ή γραμματικές τύπου 1 έχουν τη μορφή:

$$\alpha \beta \rightarrow \alpha \gamma \beta$$

Τα κεφαλαία γράμματα συμβολίζουν μη τερματικά σύμβολα:  $A \in N$ . Τα μικρά ελληνικά γράμματα είναι συμβολοσειρές που περιέχουν τερματικά και μη τερματικά σύμβολα:  $\alpha, \beta, \gamma \in (T \cup N)^*$ , ενώ  $\gamma \neq \varepsilon$ .

Οι γραμματικές με συμφραζόμενα παράγουν γλώσσες με συμφραζόμενα. Ένα παράδειγμα γλώσσας με συμφραζόμενα είναι η ακόλουθη:

$$L = \alpha^n \beta^n c^n, \quad n > 0.$$

Η περιγραφική τους ισχύς ισοδυναμεί με αυτή των γραμμικών πεπερασμένων αυτομάτων.

Θα κατασκευάσουμε γραμματική για τη γλώσσα:  $a^n b^n c^n$ . Αποδεικνύεται ότι δεν μπορεί να κατασκευαστεί γραμματική τύπου 2, χωρίς συμφραζόμενα δηλαδή, για τη γλώσσα αυτή.

Αρχίζουμε από την απλή περίπτωση:

$$S \rightarrow a \ b \ c$$

Αν προσθέσουμε ένα  $a$  στην αρχή θα πρέπει να προσθέσουμε κάτι και στο τέλος:

$$S \rightarrow a \ S \ Q$$

Στο τέλος πρέπει να υπάρχει κάτι με  $b$  και  $c$  αλλά δεν μας αρκεί απλά να προσθέσουμε  $bc$ , αφού όλα τα  $b$  πρέπει να βρίσκονται πριν τα  $c$ . Έτσι, προσθέσαμε αυτό το  $Q$ , το οποίο θα θέλαμε να αντικαταστήσουμε κάπως με  $bc$ . Αυτό θα μπορούσε να γίνει με τον ακόλουθο κανόνα:

$$b \ Q \ c \rightarrow b \ b \ c \ c$$

Το πρόβλημα είναι ότι για να το κάνουμε αυτό, τα  $Q$  πρέπει να βρίσκονται αριστερά από τα  $c$ , ενώ με τις αντικαταστάσεις των  $S$  τα  $c$  τοποθετούνται δεξιά των  $Q$ . Αυτό μπορεί να διορθωθεί με τον κανόνα:

$$c \ Q \rightarrow Q \ c$$

Η γραμματική ολοκληρώθηκε. Συγκεντρώνοντας τους παραπάνω κανόνες έχουμε:

$$\begin{array}{ll} S & \rightarrow a \ b \ c \\ S & \rightarrow a \ S \ Q \\ b \ Q \ c & \rightarrow b \ b \ c \ c \\ c \ Q & \rightarrow Q \ c \end{array}$$

Οι γραμματικές χωρίς συμφραζόμενα καλούνται και μονότονα αύξουσες γραμματικές (*increasing monotonically*). Ο λόγος είναι ότι μετά από κάθε αντικατάσταση το μέγεθος της παραγωγής μένει ίδιο ή αυξάνεται. Ο ορισμός που δόθηκε παρακάτω μπορεί να αποδειχθεί ότι είναι ισοδύναμος με τον ακόλουθο:

$$\alpha \rightarrow \beta, |\alpha| \leq |\beta|$$

όπου  $|\alpha|$  είναι το μήκος της συμβολοσειράς  $\alpha$ .

Εξαίρεση αποτελεί ο κανόνας:

$$S \rightarrow \varepsilon$$

ο οποίος επιτρέπεται να υπάρχει στη γραμματική για να μπορεί να δημιουργηθεί η κενή συμβολοσειρά. Όμως το  $S$  δεν επιτρέπεται να βρίσκεται σε δεξί μέλος ενός κανόνα, έτσι ώστε να μην μπορεί να αντικατασταθεί σε κάποιο δεξί μέλος του κανόνα.

Παρατηρήστε στη γραμματική του παραδείγματος ότι στους δύο πρώτους κανόνες αντικαθίσταται μία συμβολοσειρά μήκους 1 με μία συμβολοσειρά μήκους 3. Στον τρίτο κανόνα έχουμε αντικατάσταση μιας συμβολοσειράς μήκους 3 με μία μήκους 4, ενώ στον τελευταίο κανόνα το μήκος της συμβολοσειράς δεν μεταβάλλεται με την αντικατάσταση.

#### 1.2.4 Γενικές γραμματικές

Οι γενικές γραμματικές ή γραμματικές χωρίς περιορισμούς ή γραμματικές τύπου 0 έχουν τη μορφή:

$$\gamma \rightarrow \alpha$$

όπου  $\alpha, \gamma \in (T \cup N)^*$ , ενώ  $\gamma \neq \varepsilon$ .

Οι γραμματικές χωρίς περιορισμούς παράγουν αναδρομικά απαριθμήσιμες γλώσσες. Η περιγραφική τους ισχύς ισοδυναμεί με μία μηχανή Turing.

Ένα παράδειγμα μιας γραμματικής χωρίς περιορισμούς είναι το ακόλουθο:

$$\begin{array}{ll} 1 & S \rightarrow X \ a \ A \ c \\ 2 & A \ c \rightarrow a \ c \ A \end{array}$$

- 3      X a → D A
- 4      D A → a D
- 5      c A → D
- 6      D a → a a
- 7      D → d a

### 1.3 Περιγραφική ικανότητα τυπικών γραμματικών και μεταγλωττιστές

Η χρήση γραμματικών στην ανάπτυξη μεταγλωττιστών συνδέεται κυρίως με τη λεκτική και τη συντακτική ανάλυση. Στη λεκτική ανάλυση χρησιμοποιούμε γραμματικές επιπέδου 3 και στη συντακτική ανάλυση γραμματικές επιπέδου 2. Το επίπεδο 1 σχετίζεται με τη σημασιολογική ανάλυση, αν και στην πράξη δεν χρησιμοποιούνται γραμματικές με συμφραζόμενα για τη σημασιολογική ανάλυση, αλλά καταφεύγουμε σε κατάλληλες, γρήγορες και ευκολότερα υλοποιήσιμες ευρετικές τεχνικές.

Ας δούμε μερικές προτάσεις που ισχύουν για την περιγραφική ικανότητα των τυπικών γραμματικών σε σχέση με τις ανάγκες της τεχνολογίας των μεταγλωττιστών και τις φάσεις ανάπτυξης.

Το σύνολο όλων των λεκτικά ορθών προγραμμάτων μπορεί να περιγραφεί από μία κανονική γραμματική.

Στη λεκτική ανάλυση χρησιμοποιούμε κανονικές γραμματικές ή τις ισοδύναμες και περισσότερο ευέλικτες περιγραφικά κανονικές εκφράσεις, με σκοπό να αναγνωρίσουμε λεκτικές μονάδες ή σφάλματα.

Για παράδειγμα, στη λεκτική ανάλυση μπορούμε να ελέγχουμε ότι:

- Οι αριθμητικές σταθερές έχουν τη σωστή μορφή που ορίζεται από την περιγραφή της γλώσσας.
- Δεν συναντάμε end-of-file ενώ έχουν ανοίξει σχόλια χωρίς να έχουν κλείσει.
- Τα αναγνωριστικά ξεκινούν από γράμμα και ακολουθεί μία σειρά γραμμάτων και αριθμών.
- Μία λεκτική μονάδα που ξεκινάει από ψηφίο και ακολουθείται από χαρακτήρες ή ψηφία είναι μία μη νόμιμη λεκτική μονάδα.
- Δεν υπάρχει μέσα σε κάποιο πρόγραμμα παράνομος χαρακτήρας, δηλαδή χαρακτήρας που δεν ανήκει στο αλφάβητο της γλώσσας.
- Όλες οι λεκτικές μονάδες που σχηματίζονται έχουν νόημα με βάση την περιγραφή της γλώσσας.

Το σύνολο των συντακτικά ορθών προγραμμάτων περιγράφεται από μία γραμματική χωρίς συμφραζόμενα.

Στη συντακτική ανάλυση ελέγχουμε αν ένα πρόγραμμα είναι συντακτικά ορθό. Με τον όρο συντακτικά ορθό εννοούμε ότι ακολουθεί τη γραμματική της γλώσσας. Αν κάποια ιδιότητα της γλώσσας δεν περιγράφεται από τη γραμματική, τότε δεν θεωρείται σύνταξη. Παράβαση αυτής της ιδιότητας και δεν θα οδηγήσει σε αυτό που θεωρούμε συντακτικό σφάλμα.

Για παράδειγμα, με βάση τη γραμματική μιας γλώσσας μπορούμε να εντοπίσουμε:

- Αν σε μία αριθμητική ή λογική παράσταση, ή οπουδήποτε άλλού έχει νόημα, ο αριθμός των παρενθέσεων που άνοιξαν είναι ο ίδιος με τον αριθμό των παρενθέσεων που έκλεισαν.
- Αν μετά το while ή το if ακολουθεί άνοιγμα παρένθεσης.
- Αν βρέθηκε else το οποίο δεν αντιστοιχεί σε κάποιο if που προηγήθηκε.
- Αν έγινε εκχώρηση σε συμβολική αριθμητική σταθερά, (πχ. 3.14:=pi).
- Αν ακολουθείται ο κανόνας των διαχωριστών εντολών ( ; ).

- Αν κατά τη δήλωση των μεταβλητών, αυτές χωρίζονται με κόμματα μεταξύ τους.
- Αν η πρώτη εντολή ενός προγράμματος είναι η δεσμευμένη λέξη `program` και η τελευταία η `τελεία`.

Δεν μπορούμε, όμως, με τη συντακτική ανάλυση να αναγνωρίσουμε, αν μία μεταβλητή έχει δηλωθεί ή όχι. Αυτό θα απαιτούσε γραμματική υψηλότερου επιπέδου (ή κάποια ευρετική τεχνική).

Το σύνολο των σημασιολογικά ορθών προγραμμάτων περιγράφεται από μία γραμματική με συμφραζόμενα.

Η σημασιολογική ανάλυση εντοπίζει σφάλματα στο αρχικό πρόγραμμα τα οποία δεν μπορούν να περιγραφούν από μία γραμματική χωρίς συμφραζόμενα.

Η σημασιολογική ανάλυση μπορεί, για παράδειγμα, να απαντήσει στα ερωτήματα:

- Η μεταβλητή έχει δηλωθεί;
- Πρόκειται για απλή μεταβλητή ή πίνακα;
- Ο τύπος της μεταβλητής είναι έγκυρος σε σχέση με το πως χρησιμοποιείται η μεταβλητή μέσα στην αριθμητική παράσταση;
- Μήπως γράφουμε/διαβάζουμε έξω από τα όρια ενός πίνακα;
- Μήπως γράφουμε/διαβάζουμε σε έναν πίνακα χωρίς να προσδιορίσουμε κάποιο κελί του;
- Μήπως διαβάζουμε την τιμή μίας μεταβλητής χωρίς πριν να έχουμε αναθέσει τιμή σε αυτήν;
- Η εντολή `return` βρίσκεται πράγματι μέσα σε μία συνάρτηση;
- Υπάρχει τουλάχιστον ένα `return` μέσα σε μία συνάρτηση;
- Μήπως καλούμε μία συνάρτηση όπως μία διαδικασία ή το αντίστροφο;
- Οι παράμετροι μιας συνάρτησης ή διαδικασίας καλούνται ακριβώς με τον τρόπο που έχουν δηλωθεί (σειρά, τύπος, τρόπος περάσματος);

Η περιγραφή ενός προβλήματος με γραμματικές με συμφραζόμενα είναι αρκετά δυσκολότερο έργο από το να περιγράψουμε ένα πρόβλημα με γραμματικές χωρίς συμφραζόμενα. Αυτός είναι ο λόγος που στην συντακτική ανάλυση περιορίζομαστε σε μία γραμματική χωρίς συμφραζόμενα, ακόμα κι αν με αυτόν τον τρόπο δεν καταφέρνουμε να περιγράψουμε δλες τις λεπτομέρειες του προβλήματος που θα επιθυμούσαμε.

Πέρα από τη δυσκολία στην περιγραφή, υπάρχει και η δυσκολία στην ανάπτυξη αυτοματοποιημένων μεθόδων κατασκευής μεταγλωττιστών ή εργαλείων που δημιουργούν έναν συντακτικό αναλυτή αυτόματα από μία γραμματική.

Αλλά, ίσως το σημαντικότερο από όλα, είναι ότι ένας συντακτικός αναλυτής μίας γλώσσα με συμφραζόμενα έχει αυξημένη υπολογιστική πολυπλοκότητα και χρειάζεται απαγορευτικό χρόνο για να εκτελεστεί.

Λαμβάνοντας υπόψη όλα τα παραπάνω, οι κατασκευαστές μεταγλωττιστών έχουν επιλέξει να περιγράφουν με μία γραμματική χωρίς συμφραζόμενα τη σύνταξη της γλώσσας. Όσα δεν μπορούν να περιγραφούν από μία γραμματική χωρίς συμφραζόμενα, δεν περιγράφονται από κάποια τυπική γραμματική, αλλά χρησιμοποιούμε ευρετικές μεθόδους ανάλογα με την περίπτωση.

Για παράδειγμα, όταν θέλουμε να βεβαιωθούμε ότι σε κάθε συνάρτηση υπάρχει τουλάχιστον ένα `return`, θα χρησιμοποιήσουμε μία μεταβλητή που θα αρχικοποιείται σε `false` στην αρχή της μεταγλωττισης της συνάρτησης. Στη μεταβλητή αυτή θα εκχωρείται η τιμή `true` κάθε φορά που συναντάται κάποιο `return`. Τέλος, θα ελέγχεται στο τέλος της μεταγλωττισης της συνάρτησης αν η μεταβλητή αυτή έχει γίνει `true` ή όχι. Αν δεν έχει γίνει `true`, τότε θα εμφανίζεται το κατάλληλο προειδοποιητικό μήνυμα ή το ανάλογο σημασιολογικό σφάλμα και ο μεταγλωττιστής θα δρα αντίστοιχα για τη συνέχεια. Αν θέλαμε, μάλιστα, να

βεβαιωθούμε ότι έστω ένα `return` θα εκτελεστεί, τότε η ανάλυση είναι περισσότερο δύσκολη και απαιτεί ανάλυση των δομών απόφασης και των συνθηκών εκτέλεσης των βρόχων.

Τελειώνοντας ας δούμε τι ακόμα έχει δυνατότητα να περιγράψει μία γραμματική και τι όχι.

Το σύνολο των προγραμμάτων που τερματίζουν σε πεπερασμένο χρόνο με συγκεκριμένη είσοδο μπορεί να περιγραφεί από μία γραμματική χωρίς περιορισμούς.

Το σύνολο των προγραμμάτων που λύνουν ένα συγκεκριμένο πρόβλημα δεν μπορεί να περιγραφεί από καμία γραμματική.

#### 1.4 Γραμματικές LL(1)

Ο συμβολισμός  $LL(1)$  συντίθεται από τα παρακάτω:

- Το πρώτο  $L$  είναι το αρχικό γράμμα του όρου *left to right* και σημαίνει ότι η είσοδος της γραμματικής διαβάζεται από τα αριστερά προς τα δεξιά.
- Το δεύτερο  $L$  είναι το αρχικό γράμμα της λέξης *leftmost* και υποδηλώνει ότι το συντακτικό δέντρο που παράγεται αποτελεί την αριστερότερη δυνατή παραγωγή.
- Το  $(1)$  είναι ίσως το πιο ενδιαφέρον από όλα και υποδηλώνει ότι όταν ένας κανόνας έχει περισσότερες από μία εναλλακτικές επιλογές να ακολουθήσει, τότε την επιλογή του θα την καθορίσει το επόμενο τερματικό σύμβολο στην είσοδο. Ο αριθμός  $(1)$  σημαίνει ότι ένα τερματικό σύμβολο στην είσοδο αρκεί για να καθοριστεί η επιλογή αυτή.

Οι γραμματικές  $LL(1)$  δεν περιέχουν αμφισημίες (not ambiguous) και ούτε παρουσιάζονται σε αυτές αριστερές αναδρομές.

Ας δούμε ένα παράδειγμα γραμματικής  $LL(1)$ :

1	$S \rightarrow A \ c$
2	$A \rightarrow a \ A$
3	b   A
4	ε

η οποία αναγνωρίζει συμβολοσειρές από  $a$  και  $b$  οι οποίες τελειώνουν με  $c$ .

Ας θεωρήσουμε, επίσης, ότι έχουμε την είσοδο:  $ababc$ .

Η αναγνώριση θα ξεκινήσει από τον κανόνα  $(1)$  όπου θα ενεργοποιηθεί ο κανόνας  $A$ . Ο  $A$  έχει δύο επιλογές, την  $(2)$  και  $(3)$ . Το ποια από τις δύο διαδρομές θα ακολουθήσει, θα το καθορίσει το επόμενο τερματικό σύμβολο στην ακολουθία των τερματικών συμβόλων της εισόδου. Στο παράδειγμά μας αυτό είναι το  $a$ . Έτσι, θα ενεργοποιηθεί ο κανόνας  $(2)$ , αφού αυτός ξεκινά με το τερματικό  $a$ . Όταν αναγνωριστεί το τερματικό σύμβολο  $a$ , τότε σύμφωνα με τον κανόνα  $(2)$ , θα ενεργοποιηθεί πάλι ο κανόνας  $A$ . Εμφανίζεται το ίδιο δίλημμα. Τώρα, όμως, στην είσοδο έχουμε διαθέσιμο ένα  $b$ . Έτσι, θα ενεργοποιηθεί ο κανόνας  $(3)$  και θα αναγνωριστεί το  $b$ .

Με την ίδια ακριβώς διαδικασία ο κανόνας  $(2)$  θα αναγνωρίσει το επόμενο τερματικό  $a$  και ο κανόνας  $(3)$  το  $b$ , ο οποίος θα ενεργοποιήσει για μία ακόμα φορά τον κανόνα  $A$ . Τώρα, όμως, φτάσαμε στο τέλος της σειράς των  $a$  και  $b$  στην είσοδο και στο τέλος της δυνατότητας αναγνώρισης τερματικών συμβόλων από τους κανόνες  $A$  και  $B$ . Το επόμενο τερματικό σύμβολο στην είσοδο είναι το  $c$  και οι εναλλακτικές  $(2)$  και  $(3)$  δεν μπορούν να ενεργοποιηθούν, οπότε η μόνη εναπομένουσα εναλλακτική είναι ο κανόνας  $(4)$ . Ο  $(4)$  θα ενεργοποιηθεί, θα τερματιστεί η αναδρομική κλήση της  $A$  και η γραμματική θα επιστρέψει στον κανόνα  $(1)$ , όπου θα αναγνωριστεί το τερματικό σύμβολο  $c$  και μαζί, επιτυχώς, δήλη η συμβολοσειρά.

Αν αντί για την παραπάνω γραμματική είχαμε την ακόλουθη:

1	$S \rightarrow A \ c$
2	$A \rightarrow a \ A$
3	 a    B
4	 ε
5	$B \rightarrow b \ A$
6	 b    B
7	 ε

και την ίδια είσοδο,  $ababc$ , τότε θα μπορούσαμε να διαπιστώσουμε ότι η γραμματική εξακολουθεί να αναγνωρίζει την προς εξέταση συμβολοσειρά. Όμως η γραμματική δεν είναι πια LL(1), αφού όταν ενεργοποιηθεί ο κανόνας  $A$  και στην είσοδο υπάρχει το τερματικό σύμβολο  $a$ , τότε δεν μας αρκεί ένα τερματικό σύμβολο στην είσοδο για να επιλέξουμε αν θα ενεργοποιηθεί ο κανόνας (2) ή (3). Όμοιο πρόβλημα μπορούμε να διαπιστώσουμε ότι είναι δυνατόν να δημιουργηθεί στους κανόνες (5) και (6), όταν το επόμενο τερματικό σύμβολο προς αναγνώριση στην είσοδο είναι το  $b$ .

Οι γραμματικές LL(1) είναι πολύ σημαντικές στην τεχνολογία των μεταγλωττιστών, διότι διευκολύνουν την ανάπτυξη συντακτικών αναλυτών που βασίζονται στην τεχνική της προβλέπουσας αναδρομικής κατάβασης, κάτι που θα μελετήσουμε στο κεφάλαιο της συντακτικής ανάλυσης (κεφ. 5), παρακάτω στο βιβλίο αυτό.

## 1.5 Γραμματικές LR(1)

Εκτός από τις γραμματικές LL(1), υπάρχουν και οι γραμματικές LR(1). Όπως γίνεται αντιληπτό από τον συμβολισμό, σε αντίθεση με τις LL(1), οι LR(1) γραμματικές παράγουν το δεξιότερο δέντρο.

- Το  $L$  σημαίνει ότι η είσοδος της γραμματικής διαβάζεται από τα αριστερά προς τα δεξιά, όπως στις γραμματικές LL(1).
- Το  $R$  είναι το αρχικό γράμμα της λέξης *rightmost* και υποδηλώνει ότι το συντακτικό δέντρο που παράγεται αποτελεί τη δεξιότερη δυνατή παραγωγή.
- Το (1), όπως και στις γραμματικές LL(1), υποδηλώνει ότι όταν ένας κανόνας έχει περισσότερες από μία εναλλακτικές επιλογές να ακολουθήσει, τότε την επιλογή του θα την καθορίσει το επόμενο τερματικό σύμβολο στην είσοδο.

Οι γραμματικές LR(1) είναι προσεγγίσεις από κάτω προς τα πάνω (*bottom up*), δηλαδή το δέντρο σχηματίζεται από τα φύλλα προς τη ρίζα. Ανήκουν, επίσης, στην κατηγορία των γλωσσών χωρίς συμφραζόμενα.

Δεν θα ασχοληθούμε εδώ με τις γραμματικές αυτές, αλλά θα δούμε ένα απλό παράδειγμα. Περισσότερα παραδείγματα χρήσης τους θα μπορέσει κανείς να δει αργότερα στο κεφάλαιο 12.

Στο παρακάτω παράδειγμα η γραμματική αναγνωρίζει συμβολοσειρές οι οποίες αρχίζουν με κανένα ή περισσότερα  $a$ , συνεχίζουν με ένα  $b$  περισσότερα  $b$  και τερματίζουν με ακριβώς ένα  $c$ . Παρατηρήστε τη δεξιά αναδρομή η οποία είναι επιτρεπτή σε μία LR(1) γραμματική.

$S \rightarrow A \ b \ B \ c$
$A \rightarrow A \ a$
 ε
$B \rightarrow B \ β$
 ε

## 1.6 Ο συμβολισμός EBNF

Ο συμβολισμός *Extented Backus-Naur Form (EBNF)* αναπτύχθηκε από τον Backus το 1959 [8] και αργότερα με τη συμβολή του Naur εξελίχθηκε και αποτέλεσε έναν ιδιαίτερα ευέλικτο και περιγραφικό τρόπο συμβολισμού γραμματικής χωρίς συμφράζόμενα.

Στον συμβολισμό EBNF οι κανόνες:

$$\begin{aligned} A &\rightarrow \alpha \\ A &\rightarrow \beta \end{aligned}$$

μπορούν να γραφούν ως:

$$\begin{aligned} A &\rightarrow \alpha \\ &| \beta \end{aligned}$$

Το προαιρετικό μέλος ένος κανόνα μπορεί να συμβολιστεί με [ ]. Μία γραμματική της δομής if με το μέλος του else να είναι προαιρετικό θα γραφόταν κάπως έτσι:

$$\begin{aligned} S &\rightarrow \text{if } (\text{condition}) \\ &\quad S \\ &[ \text{else } S ] \end{aligned}$$

Τα σύμβολα + και \* χρησιμοποιούνται για να περιγράψουν επαναλήψεις. Μία λίστα μεταβλητών, που χωρίζονται μεταξύ τους με κόμματα, μπορεί να περιγραφεί ως εξής:

$$V \rightarrow \text{id } ( , \text{id})^*$$

ενώ ένας πίνακας στη γλώσσα προγραμματισμού C, αφού κάθε μεταβλητή που έχει δηλωθεί ως πίνακας ακολουθείται από τουλάχιστον έναν δείκτη:

$$A \rightarrow \text{id } ([\text{exp}])^+$$

Ένα πολύ σημαντικό πλεονέκτημα του συμβολισμού EBNF είναι ότι μας επιτρέπει να δημιουργήσουμε μικρότερες σε αριθμό κανόνων γραμματικές, κάθε κανόνας των οποίων περιγράφει περισσότερη πληροφορία. Το χαρακτηριστικό αυτό επιτρέπει την ανάπτυξη συντακτικών αναλυτών με μικρότερο αριθμό συναρτήσεων, κάτι που μειώνει την ανάγκη ανταλλαγής πληροφορίας ανάμεσα στις συναρτήσεις, απλοποιώντας τη σχεδίαση και την ανάπτυξη του κώδικα.

Ακολουθεί ένα παράδειγμα γραμματικής εκφρασμένης στο συμβολισμό EBNF. Πρόκειται για μία γραμματική που περιγράφει μαθηματικές πράξεις ανάμεσα σε ακέραιους αριθμούς:

```
# arithmetic expression consists of one of more terms
# sepatated by addition operators
expression → optionalSign term ( ADD_OP term )*
# term consists of one of more factors sepatated by multiplication operators
term → factor ( MUL_OP factor )*
# factor can be an integer
factor → INTEGER
# or an expression in parenthesis to declare priority
# addition and multiplication operators
ADD_OP → +
MUL_OP → *
```

Κάθε έκφραση (expression) αποτελείται από ένα προαιρετικό πρόσημο (optionalSign), ενώ στη συνέχεια ακολουθεί τουλάχιστον ένας όρος (term). Αν υπάρχουν περισσότεροι του ενός όροι, τότε αυτοί χωρίζονται μεταξύ τους με έναν προσθετικό τελεστή (ADD\_OP), δηλαδή με έναν εκ των: + και -. Κάθε όρος αποτελείται από έναν ή περισσότερους παράγοντες (factor). Αν οι παράγοντες είναι περισσότεροι από ένας, τότε χωρίζονται μεταξύ τους με έναν πολλαπλασιαστικό τελεστή (MUL\_OP), δηλαδή με έναν εκ των:

\* και /. Κάθε παράγοντας μπορεί να είναι είτε ένας ακέραιος αριθμός (INTEGER) είτε μία έκφραση μέσα σε παρενθέσεις.

Με αυτήν τη δομή των κανόνων ορίζεται η προτεραιότητα ανάμεσα στους αριθμητικούς τελεστές και τις παρενθέσεις. Το γεγονός ότι ο κανόνας *term* ενεργοποιεί τους κανόνες *factor* δίνει μεγαλύτερη προτεραιότητα στους πολλαπλασιαστικούς τελεστές έναντι των προσθετικών, αφού πρέπει πρώτα να τελεστούν όλες οι πράξεις ανάμεσα στους πολλάπλασιαστικούς τελεστές, προκειμένου να ολοκληρωθεί ο κανόνας *term* και να επιτραπεί η συνέχεια της αποτίμησης του κανόνα *expression*, όπου και βρίσκονται οι πράξεις ανάμεσα στους προσθετικούς τελεστές. Με το ίδιο σκεπτικό, ο κανόνας *factor* δημιουργεί μεγαλύτερη προταιραιότητα στην τέλεση των πράξεων μέσα στις παρενθέσεις από αυτές που βρίσκονται έξω από αυτές.

Ένα μεγαλύτερο παράδειγμα γραμματικής σε συμβολισμό EBNF μπορείτε να βρείτε στο κεφάλαιο 3, όπου παρουσιάζεται η γραμματική της *C-imperative*, γραμμένη στον συμβολισμό αυτόν.

## Βιβλιογραφία

- [1] John E. Hopcroft, Rajeev Motwani και Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd. Pearson/Addison Wesley, 2007. ISBN: 9780321455369.
- [2] Michael Sipser. *Introduction to the Theory of Computation*. 3rd. Boston, MA: Course Technology, 2013. ISBN: 9781133187790.
- [3] Michael Sipser. *Εισαγωγή στη Θεωρία Υπολογισμού*. Ξενόγλωσσος τίτλος: *Introduction to the Theory of Computation*, Επιστημονική επιμέλεια έκδοσης: Αριστείδης Παγουρτζής. Πανεπιστημιακές Εκδόσεις Κρήτης, 2018. ISBN: 9789605245580.
- [4] Harry R. Lewis και Christos H. Papadimitriou. *Elements of the theory of computation* (2. ed.). Prentice Hall, 1998, σσ. 1–361. ISBN: 9780132624787.
- [5] Noam Chomsky. “Three models for the description of language”. Στο: *IRE Transactions on Information Theory* 2 (1956), σσ. 113–124.
- [6] Noam Chomsky. *Syntactic Structures*. Reprinted 1985 by Springer, Berlin and New York. The Hague: Mouton & Co., 1957. ISBN: 3-11-017279-8.
- [7] S. C. Kleene. “Representation of events in nerve nets and finite automata”. Στο: *Automata Studies*. Επιμέλεια υπό Claude Shannon και John McCarthy. Princeton, NJ: Princeton University Press, 1956, σσ. 3–41.
- [8] John W. Backus. “The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.” Στο: *IFIP Congress*. Butterworths, London, 1959, σσ. 125–131.



## ΚΕΦΑΛΑΙΟ 2

---

### ΓΡΑΜΜΑΤΙΚΕΣ LL(1) ΚΑΙ ΣΥΝΤΑΚΤΙΚΗ ΑΝΑΛΥΣΗ

---

#### Σύνοψη:

Στο κεφάλαιο αυτό θα δούμε πώς χρησιμοποιούνται οι γραμματικές LL(1) στη συντακτική ανάλυση. Η σύνταξη μιας γλώσσας προγραμματισμού σε γραμματική τύπου LL(1) επιτρέπει την αυτοματοποιημένη μετατροπή της γραμματικής σε κώδικα, είτε μέσω της τεχνικής της προβλέπουσας αναδρομικής κατάβασης, που θα δούμε σε επόμενο κεφάλαιο, είτε μέσω του πίνακα ανίχνευσης (ή αλλιώς πίνακα συντακτικής ανάλυσης), τεχνική που θα μελετήσουμε στο κεφάλαιο αυτό. Για τον σχηματισμό του πίνακα ανίχνευσης θα χρειαστεί να ορίσουμε και να υπολογίσουμε κάποια σύνολα από τερματικά σύμβολα.

Έτσι, θα οριστούν τα σύνολα *first* και *follow*. Το σύνολο *first* αποτελείται από τα τερματικά σύμβολα από τα οποία μπορεί να ξεκινήσουν οι παραγωγές που παράγονται από κάποιον κανόνα. Το σύνολο *follow(A)* αποτελείται από τα τερματικά σύμβολα που μπορεί να βρεθούν μετά από το μη τερματικό σύμβολο *A* σε έναν κανόνα. Ο υπολογισμός των συνόλων *first* και *follow* μπορεί να γίνει με μηχανιστικό τρόπο και θα παρουσιαστεί ένας αλγόριθμος υπολογισμού για το συνόλο *first* και ένας για το σύνολο *follow*.

Τα σύνολα *first* και *follow* βοηθούν στην κατασκευή του πίνακα ανίχνευσης/συντακτικής ανάλυσης. Ο πίνακας ανίχνευσης είναι μία δισδιάστατη δομή, η οποία στη μία διάσταση έχει τα τερματικά σύμβολα, στην άλλη τα μη τερματικά σύμβολα και στα κελιά τον κανόνα που πρέπει να ακολουθηθεί για το συγκεκριμένο ζευγάρι τερματικού και μη τερματικού συμβόλου. Για την κατασκευή του πίνακα θα περιγραφεί ένας αλγόριθμος, αφού και αυτή η διαδικασία είναι μηχανιστική.

Τέλος, θα παρουσιαστεί ο αλγόριθμος συντακτικής ανάλυσης, ο οποίος χρησιμοποιεί τον πίνακα συντακτικής ανάλυσης και ένα αυτόματο στοίβας για να αποφανθεί αν μία συμβολοσειρά παράγεται ή όχι από τη γραμματική.

### Προαπαιτούμενη γνώση:

- θεωρία τυπικών γραμματικών
  - κεφάλαιο 1
- 

Με τη βοήθεια των γραμματικών LL(1) μπορούμε εύκολα να κατασκευάσουμε συντακτικούς αναλυτές. Στο κεφάλαιο αυτό θα μελετήσουμε πώς μπορούμε να κατασκευάσουμε έναν συντακτικό αναλυτή με τη βοήθεια του πίνακα συντακτικής ανάλυσης/ανίχνευσης. Με τον πίνακα ανίχνευσης μπορούμε να διαπιστώσουμε αν μία γραμματική είναι LL(1) ή όχι, αλλά και να βασιστούμε σε αυτόν κατά τη συντακτική ανάλυση. Όλες οι διαδικασίες είναι μηχανιστικές, αφού βασίζονται σε αλγορίθμους. Το γεγονός αυτό, επιτρέπει την εύκολη δημιουργία συντακτικών αναλυτών από γραμματικές LL(1). Βασικά εργαλεία στη διαδικασία αυτή αποτελούν τα σύνολα *first* και *follow*.

Περισσότερες πληροφορίες σχετικά με τα θέματα του κεφαλαίου αυτού μπορείτε να βρείτε στα ακόλουθα: [1, 2, κεφ.3.3], [3, 4, κεφ.4.4], [5, κεφ.4.2], [6, κεφ.3], [7, κεφ.3.8].

## 2.1 Το σύνολο *first*

Έστω η γραμματική  $G = (T, N, P, S)$  και  $\alpha \in (T \cup N)^*$ .

Το σύνολο  $first(\alpha) \subseteq (T \cup \{ \epsilon \})$  ορίζεται ως εξής:

- αν υπάρχει δυνατή παραγωγή της μορφής:

$$\alpha \xrightarrow{*} c\beta, \quad c \in T, \quad \beta \in (T \cup N)^*$$

τότε:

$$c \in first(\alpha)$$

- αν υπάρχει δυνατή παραγωγή της μορφής:

$$\alpha \xrightarrow{*} \epsilon$$

τότε:

$$\epsilon \in first(\alpha)$$

με άλλα λόγια το  $first(\alpha)$  περιέχει όλα τα τερματικά σύμβολα με τα οποία είναι δυνατόν να αρχίζουν οι συμβολοσειρές που παράγονται από το  $\alpha$ . Αν η  $\alpha$  παράγει την κενή συμβολοσειρά, τότε το σύνολο  $first(\alpha)$  περιέχει και το  $\epsilon$ :

$$first(\alpha) = \{ t \mid \alpha \xrightarrow{*} t\beta \} \cup \{ \epsilon \mid \alpha \xrightarrow{*} \epsilon \}$$

όπου  $t$  τερματικό σύμβολο και  $\alpha, \beta$  προτασιακοί τύποι.

Για να υπολογίσουμε το σύνολο  $first(\alpha)$  ακολουθούμε τα εξής βήματα, συμβολισμένα σε έναν python-like ψευδοκώδικα:

```

1      if α = x and α ∈ T:
2          x ∈ first(α)
3      if α = ε:
4          ε ∈ first(α)
5      if α ∈ N and α → β1 | β2 | ... | βk:
6          first(β1) ∪ first(β2) ∪ ... ∪ first(βk) ⊆ first(α)
7      if α ∈ (T ∪ N)* and α = X1X2 ... Xk:
8          j=0
9          while True:

```

```

10          j = j + 1
11          x ∈ first(α), ∀x ∈ (first(Xj) - {ε})
12          if not Xj * ⇒ ε or j=k:
13              break
14          if X1X2 ... Xk * ⇒ ε:
15              ε ∈ first(α)

```

Ας δούμε γραμμή-γραμμή τον παραπάνω αλγόριθμο.

Αν το  $\alpha$  είναι ένα τερματικό σύμβολο, έστω το  $x$  (γραμμή 1), τότε το  $x$  ανήκει στο  $first(\alpha)$  (γραμμή 2).

Αν το  $\alpha$  είναι η κενή συμβολοσειρά (γραμμή 3), τότε και η κενή συμβολοσειρά ανήκει στο  $first(\alpha)$  (γραμμή 4).

Αν το  $\alpha$  είναι ένα μη τερματικό σύμβολο (γραμμή 5), τότε τα στοιχεία όλων των συνόλων  $first$  των κανόνων που παράγει ανήκουν στο  $first(\alpha)$  (γραμμή 6).

Αν το  $\alpha$  είναι μία σειρά από τερματικά και μη τερματικά σύμβολα  $\alpha \in (T \cup N)^*$  και  $\alpha = X_1X_2 \dots X_k$  (γραμμή 7), τότε δημιουργούμε έναν ατέρμονο βρόχο (γραμμή 9), ώστε να εξομοιώσουμε μία δομή repeat-until που δεν υποστηρίζεται από την Python. Ο δείκτης του βρόχου είναι η μεταβλητή  $j$ , η οποία αρχικοποιείται στο μηδέν (γραμμή 8) και αυξάνεται κατά 1 (γραμμή 10), ώστε στην πρώτη επανάληψη να δείχνει το πρώτο τερματικό ή μη τερματικό σύμβολο  $X_1$  και σε κάθε επανάληψη ένα από τα  $X_j$ . Σε κάθε επανάληψη τα στοιχεία που ανήκουν στο  $first$  του εξεταζόμενου τερματικού ή μη τερματικού συμβόλου  $X_j$  μεταφέρονται στο  $first(\alpha)$  (γραμμή 11).

Η κενή συμβολοσειρά δεν μας χρειάζεται πια, οπότε την εξαιρούμε από τη μεταφορά αυτή (γραμμή 11). Η κενή συμβολοσειρά είχε τοποθετηθεί εκεί προκειμένου να γνωρίζουμε αν το εξεταζόμενο σύμβολο  $X_j$  μπορεί να παραγάγει την κενή συμβολοσειρά. Πού χρειαζόταν αυτό; Στη γραμμή 12. Αν το εξεταζόμενο σύμβολο  $X_j$  μπορεί να παραγάγει την κενή συμβολοσειρά, τότε μπορεί να αντικατασταθεί με αυτή. Στην περίπτωση αυτή το  $first(X_{j+1})$  πρέπει να συνεισφέρει και αυτό στο  $first(a)$ .

Με άλλα λόγια, ξεκινώντας από το  $X_1$  προσθέτουμε νέα στοιχεία στο  $first(a)$ , όσο το  $X_j$  παράγει την κενή συμβολοσειρά. Το πρώτο  $X_j$  που δεν θα έχει αυτή την ιδιότητα θα μας οδηγήσει έξω από τον βρόχο (γραμμές 12-13). Φυσικά, έξω από τον βρόχο θα οδηγηθούμε και αν δεν υπάρχουν άλλα  $X_j$  για εξέταση, όταν δηλαδή  $j=k$  (γραμμές 12-13).

Τέλος, αν το  $\alpha$  παράγει την κενή συμβολοσειρά, τότε η κενή συμβολοσειρά ανήκει στο  $first(a)$  (γραμμές 14-15).

Ένα παράδειγμα είναι σίγουρα χρήσιμο για να κατανοήσουμε τον αλγόριθμο.

Δίνεται η ακόλουθη γραμματική, η οποία περιγράφει αριθμητικές παραστάσεις που μπορεί να περιέχουν προσθέτεις και πολλαπλασιασμούς, ενώ επιτρέπει την ομαδοποίηση με τα σύμβολα των παρενθέσεων:

```

1      E → T E'
2      E' → ε
3      E' → + T E'
4      T → F T'
5      T' → ε
6      T' → * F T'
7      F → ( E )
8      F → id

```

Από τον κανόνα (1) έχουμε:

$$first(E) = first(T).$$

Το  $T$  δεν παράγει το  $E$ , άρα δεν προχωράμε στην εξέταση του  $E'$ .

Από τον κανόνα (2), το  $E'$  παράγει το κενό, ενώ από τον κανόνα (3) φαίνεται ότι και το  $+$  ανήκει στο  $first(E')$ . Άρα για το  $first(E')$  έχουμε:

$$first(E') = \{+, \epsilon\}.$$

Από τον κανόνα (4) συμπεραίνουμε ότι:

$$\text{first}(T) = \text{first}(F).$$

Ούτε το  $F$  παράγει την κενή συμβολοσειρά, οπότε συνεχίζουμε με το επόμενο σύμβολο, το  $T'$ . Από τους κανόνες (5) και (6) συμπεραίνουμε ότι:

$$\text{first}(T') = \{ *, \varepsilon \}$$

Τέλος, από τους κανόνες (7) και (8) έχουμε ότι:

$$\text{first}(F) = \{ (, \text{id} \} \}$$

Κάνοντας αντικαταστάσεις έχουμε:

$$\text{first}(E) = \text{first}(T) = \text{first}(F) = \{ (, \text{id} \}$$

Τα σύνολα  $\text{first}$  του παραδείγματος φαίνονται στον πίνακα 2.1

Πίνακας 2.1: Τα σύνολα  $\text{first}$  παραδείγματος γραμματικής αριθμητικών εκφράσεων.

Μη τερματικό σύμβολο	Στοιχεία συνόλου $\text{first}$
$E$	$(, \text{id}$
$T$	$(, \text{id}$
$F$	$(, \text{id}$
$E'$	$+ , \varepsilon$
$T'$	$* , \varepsilon$

## 2.2 Το σύνολο follow

Έστω η γραμματική  $G = (T, N, P, S)$  και  $A \in N$ .

Το σύνολο  $\text{follow}(A) \subseteq (T \cup \{ \text{eof} \})$  ορίζεται ως εξής:

- $\text{eof} \in \text{follow}(S)$
- αν υπάρχει δυνατή παραγωγή της μορφής:

$$S \xrightarrow[G]{*} \alpha A c \beta, \quad c \in T, \quad \alpha, \beta \in (T \cup N)^*$$

τότε:

$$c \in \text{follow}(A)$$

- αν υπάρχει δυνατή παραγωγή της μορφής:

$$S \xrightarrow[G]{*} \alpha A, \quad \alpha \in (T \cup N)^*$$

τότε:

$$\text{eof} \in \text{follow}(A)$$

με άλλα λόγια το  $\text{follow}(A)$  περιέχει όλα τα τερματικά σύμβολα τα οποία μπορούν να ακολουθούν το  $A$  σε έναν προτασιακό τύπο. Αν το  $A$  μπορεί να είναι το τελευταίο σύμβολο σε έναν προτασιακό τύπο, τότε το σύνολο  $\text{follow}(A)$  περιέχει και το  $\text{eof}$ :

$$\text{follow}(A) = \{ t \mid S a A t \beta \}$$

με  $t$  τερματικό,  $A$  μη τερματικό και  $\alpha, \beta$  προτασιακούς τύπους.

Για να υπολογίσουμε το σύνολο  $\text{follow}(A)$  ακολουθούμε τα εξής βήματα, συμβολισμένα σε έναν python-like ψευδοκώδικα:

- 1       $eof \in follow(S)$
- 2       $\forall B, B \rightarrow \alpha A \gamma, A, B \in N, \alpha, \gamma \in (T \cup N)^*$ :  
 $\forall x \in (first(\gamma) - \{\epsilon\}), x \in follow(A)$
- 3       $\forall B, B \rightarrow \alpha A, A, B \in N, \alpha \in (T \cup N)^*$ :  
 $\forall x \in follow(B), x \in follow(A)$
- 4       $\forall B, B \rightarrow \alpha A \gamma, A, B \in N, \alpha, \gamma \in (T \cup N)^* \text{ and } \epsilon \in first(\gamma)$ :  
 $\forall x \in follow(B), x \in follow(A)$

Ως παράδειγμα θα υπολογίσουμε τα σύνολα  $follow$  για τα μη τερματικά σύμβολα της ίδιας γραμματική που χρησιμοποιήσαμε για τα σύνολα  $first$ , εκμεταλλευόμενοι ότι έχουμε ήδη υπολογίσει τα σύνολα αυτά. Τα σύνολα  $first$  απαιτούνται για τον υπολογισμό των συνόλων  $follow$ . Παραθέτουμε πάλι τη γραμματική για ευκολία.

- 1       $E \rightarrow T E'$
- 2       $E' \rightarrow \epsilon$
- 3       $E' \rightarrow + T E'$
- 4       $T \rightarrow F T'$
- 5       $T' \rightarrow \epsilon$
- 6       $T' \rightarrow * F T'$
- 7       $F \rightarrow ( E )$
- 8       $F \rightarrow id$

Επειδή το  $E$  είναι το αρχικό σύμβολο της γραμματικής, πρέπει στο σύνολο  $follow(E)$  να τοποθετήσουμε το  $eof$ :

$$follow(E) = \{eof\}$$

Από τον κανόνα (7) συμπεραίνουμε ότι στο σύνολο  $follow(E)$  πρέπει να τοποθετηθεί και το σύμβολο  $,$ , αφού το  $,$  έπεται του  $E$  στον κανόνα. Οπότε τώρα έχουμε:

$$follow(E) = \{eof, ,\}$$

Από τον κανόνα (1) συμπεραίνουμε ότι τα στοιχεία του συνόλου  $first(E')$  ανήκουν στο  $follow(T)$ , αφού το  $E'$  έπεται του  $T$  στον κανόνα.

$$x \in follow(T), \forall x \in first(E')$$

Επειδή το  $E'$  μπορεί να παράγει την κενή συμβολοσειρά, το  $T$  μπορεί να βρεθεί ως τελευταίο σύμβολο στον κανόνα (1), άρα τα στοιχεία του συνόλου  $follow(E)$  πρέπει να προστεθούν σε αυτά του  $follow(T)$ :

$$x \in follow(T), \forall x \in follow(E)$$

Επειδή το  $E'$  μπορεί να παράγει την κενή συμβολοσειρά, το  $T$  μπορεί να βρεθεί ως τελευταίο σύμβολο στον κανόνα (3), άρα τα στοιχεία του συνόλου  $follow(E')$  πρέπει να προστεθούν σε αυτά του  $follow(T)$ :

$$x \in follow(T), \forall x \in follow(E')$$

Άρα, συνολικά για το  $follow(T)$  έχουμε:

$$follow(T) = first(E') \cup follow(E) \cup follow(E')$$

Από τον κανόνα (1), αφού το  $E'$  είναι το τελευταίο σύμβολο στο δεξί μέλος του κανόνα, τα στοιχεία του συνόλου  $follow(E)$  πρέπει να τοποθετηθούν και στο  $follow(E')$ :

$$x \in follow(E'), \forall x \in follow(E)$$

ή αλλιώς:

$$follow(E') = follow(E) = \{eof, ,\}$$

Αντικαθιστώντας στο  $follow(T)$  έχουμε:

$$follow(T) = first(E') \cup follow(E) \cup follow(E') = \{eof, , +\}$$

Από τον κανόνα (4), αφού το  $T'$  είναι το τελευταίο σύμβολο στο δεξί μέλος του κανόνα, τα στοιχεία του συνόλου  $follow(T)$  πρέπει να τοποθετηθούν και στο  $follow(T')$ :

$x \in follow(T'), \forall x \in follow(T)$

ή αλλιώς:

$$follow(T') = follow(T) = \{eof, ), +\}$$

Τέλος, από τους κανόνες (4) και (6) το  $follow(F)$  παίρνει τα στοιχεία από το  $first(T')$ . Από τους διοικητικούς κανόνες, και επειδή το  $T'$  παράγει την κενή συμβολοσειρά, το  $T'$  παίρνει και τα  $follow(T)$  και  $follow(T')$ . Άρα έχουμε:

$$follow(F) = first(T') \cup follow(T) \cup follow(T') = \{*, eof, ), +\}$$

Τα σύνολα  $follow$  του παραδείγματος φαίνονται στον πίνακα 2.2.

Πίνακας 2.2: Τα σύνολα  $follow$  παραδείγματος γραμματικής αριθμητικών εκφράσεων.

Μη τερματικό σύμβολο	Στοιχεία συνόλου $follow$
$E$	$eof, )$
$T$	$eof, ), +$
$F$	$*, eof, ), +$
$E'$	$eof, )$
$T'$	$eof, ), +$

### 2.3 Σύνολα $first$ , $follow$ και γραμματικές LL(1)

Μία γραμματική είναι γραμματική LL(1), όταν  $\forall A \in N$ , με  $A \Rightarrow \alpha \in (T \cup N)^*$  και  $A \Rightarrow \beta \in (T \cup N)^*$  ισχύουν:

- $first(\alpha) \cap first(\beta) = \emptyset$
- if  $\alpha \stackrel{*}{\Rightarrow} \varepsilon$  then  $first(\beta) \cap follow(A) = \emptyset$

Ας εφαρμόσουμε τον παραπάνω ορισμό στη γραμματική των αριθμητικών εκφράσεων.

1	$E \rightarrow T E'$
2	$E' \rightarrow \varepsilon$
3	$E' \rightarrow + T E'$
4	$T \rightarrow F T'$
5	$T' \rightarrow \varepsilon$
6	$T' \rightarrow * F T'$
7	$F \rightarrow ( E )$
8	$F \rightarrow id$

Στη γραμματική αυτή υπάρχουν τρία σημεία τα οποία πρέπει να ελεγχθούν. Στις γραμμές 2 και 3 υπάρχουν δύο εναλλακτικές παραγωγές για το  $E'$ , στις γραμμές 5 και 6 υπάρχουν δύο εναλλακτικές παραγωγές για το σύμβολο  $T'$  και στις γραμμές 7 και 8 υπάρχουν δύο εναλλακτικές παραγωγές για το σύμβολο  $T'$ .

Θα ξεκινήσουμε από τις γραμμές 7 και 8 όπου το μη τερματικό σύμβολο  $F$  έχει δύο εναλλακτικές παραγωγές.

Για τη γραμμή 7 έχουμε ότι:

$$first'((E')) = \{( ( \}$$

Για τη γραμμή 8 έχουμε ότι:

$$first(id) = \{ (id \}$$

ενώ ισχύει ότι:

$$\{( \} \cap \{ (id \} = \emptyset$$

οπότε ικανοποιούνται οι συνθήκες για να είναι η γραμματική LL(1), όσον αφορά το σύμβολο  $F$ .

Στις γραμμές 2 και 3 το μη τερματικό σύμβολο  $E'$  έχει δύο εναλλακτικές παραγωγές.

Επειδή στη γραμμή 2 το  $E'$  παράγει την κενή συμβολοσειρά, πρέπει με βάση τη γραμμή 3 να ισχύει ότι:

$$first(+TE) \cap follow(E') = \emptyset$$

Πράγματι:

$$first(+TE) = \{ + \} \text{ και } follow(E') = \{ eof, ) \}$$

των οποίων η τομή είναι το κενό, οπότε και για το σύμβολο  $E'$  ικανοποιούνται οι συνθήκες για να είναι η γραμματική LL(1).

Όμοια, στις γραμμές 5 και 6 το μη τερματικό σύμβολο  $T'$  έχει δύο εναλλακτικές παραγωγές. Σε αναλογία με το σύμβολο  $E'$  διαπιστώνουμε ότι και εδώ ικανοποιούνται οι απαιτούμενες συνθήκες για να είναι η γραμματική LL(1).

Αφού και στα τρία σημεία της γραμματικής που ένα μη τερματικό σύμβολο έχει εναλλακτικές επιλογές ικανοποιούνται οι απαιτούμενες συνθήκες, μπορούμε να συμπεράνουμε ότι η δοθείσα γραμματική είναι LL(1).

## 2.4 Κατασκευή πίνακα ανίχνευσης/συντακτικής ανάλυσης LL(1)

Έχοντας υπολογίσει τα σύνολα  $first$  και  $follow$  μπορούμε να κατασκευάσουμε τον πίνακα ανίχνευσης για τη συντακτική ανάλυση, τον οποίο θα αποκαλούμε και ως πίνακα συντακτικής ανάλυσης. Ο πίνακας ανίχνευσης είναι μία δισδιάστατη δομή, ένας τετραγωνικός πίνακας, που στη μία του διάσταση έχει τα μη τερματικά σύμβολα της γραμματικής, στη δεύτερη τα τερματικά σύμβολα, ενώ στα κελιά του έχει κανόνες της γραμματικής. Η πληροφορία η οποία αποθηκεύει είναι το ποιος κανόνας ενεργοποιείται, για κάθε ζευγάρι μη τερματικού συμβόλου και τερματικού συμβόλου.

Για να είναι μία γραμματική LL(1) πρέπει κάθε κελί του πίνακα ανίχνευσης να περιέχει το πολύ έναν κανόνα της γραμματικής. Σε αντίθετη περίπτωση ερχόμαστε σε σύγκρουση με τη βασική αρχή της συντακτικής ανάλυσης LL(1) ότι η γραμματική είναι κατασκευασμένη με τέτοιον τρόπο ώστε αν ένας κανόνας έχει δύο εναλλακτικές, τότε αρκεί το επόμενο διαθέσιμο σύμβολο στην είσοδο για να καθορίσει ποιος κανόνας είναι ο επόμενος που θα ενεργοποιηθεί.

Αν κατά την κατασκευή του πίνακα ανίχνευσης προκύψει από τον αλγόριθμο ότι σε ένα κελί του πίνακα πρέπει να τοποθετηθούν περισσότεροι από ένας κανόνες, τότε συμπεραίνουμε ότι η γραμματική δεν είναι LL(1).

Αν η διαπέραση της συμβολοσειράς εισόδου με βάση τον πίνακα ανίχνευσης μας οδηγήσει σε κάποιο κελί το οποίο είναι κενό, τότε έχουμε οδηγηθεί σε κατάσταση σφάλματος.

Η κατασκευή του πίνακα ανίχνευσης μπορεί να γίνει με τρόπο μηχανιστικό, βασισμένοι στον ακόλουθο αλγόριθμο:

- 1  $\forall A, A \rightarrow \alpha \quad B \in N, \alpha \in (T \cup N)^*$ :
- 2  $\forall x \in first(a), \text{ make entry } (A, \alpha) : A \rightarrow \alpha$
- 3  $\text{if } \varepsilon \in first(\alpha), \forall x \in follow(a) \text{ make entry } (A, \alpha) : A \rightarrow \varepsilon$

Χρησιμοποιώντας τον παραπάνω αλγόριθμο, ας δούμε πώς κατασκευάζουμε τον πίνακα ανίχνευσης της γραμματικής των αριθμητικών εκφράσεων. Έχουμε ήδη υπολογίσει τα σύνολα  $first$  και  $follow$  της γραμματικής, τα οποία βρίσκονται στους πίνακες 2.1 και 2.2.

Για τον κανόνα στη γραμμή 1 της γραμματικής:

$$E \rightarrow T \quad E'$$

αναζητούμε με τη βοήθεια του πίνακα 2.1 το σύνολο  $first(TE')$ , όπου και βλέπουμε ότι  $first(T) = \{ (, id \}$ . Άρα στις θέσεις  $(E, "(")$  και  $(E, id)$  του πίνακα ανίχνευσης τοποθετούμε τον κανόνα  $E \rightarrow TE'$ . Το  $\varepsilon$  δεν ανήκει στο  $first(E)$ , οπότε προχωρούμε στον επόμενο κανόνα.

Για τον κανόνα στη γραμμή 3 της γραμματικής:

$$E' \rightarrow + T E'$$

έχουμε ότι  $first(+TE') = \{ + \}$ . Άρα στη θέση  $(E', +)$  του πίνακα ανίχνευσης τοποθετούμε τον κανόνα  $E' \rightarrow +TE'$ . Το  $\epsilon$  ανήκει στο  $first(E')$ , άρα για κάθε τερματικό σύμβολο που ανήκει στο  $follow(E')$  πρέπει να εισάγουμε στον πίνακα ανίχνευσης τον κανόνα  $E' \rightarrow \epsilon'$ . Από τον πίνακα 2.2 έχουμε ότι  $follow(E') = \{ eof, \} \}$ . Άρα στις θέσεις  $(E', eof)$  και  $(E, ")$  του πίνακα ανίχνευσης τοποθετούμε τον κανόνα  $E' \rightarrow \epsilon$ .

Για τον κανόνα στη γραμμή 4 της γραμματικής:

$$T \rightarrow F T'$$

αναζητούμε με τη βοήθεια του πίνακα 2.1 το σύνολο  $first(FT')$ , όπου και βλέπουμε ότι  $first(F) = \{ (, id) \}$ . Άρα στις θέσεις  $(F, ")$  και  $(F, id)$  του πίνακα ανίχνευσης τοποθετούμε τον κανόνα  $T \rightarrow FT'$ . Το  $\epsilon$  δεν ανήκει στο  $first(T)$ , οπότε προχωρούμε στον επόμενο κανόνα.

Για τον κανόνα στη γραμμή 6 της γραμματικής:

$$T' \rightarrow * F T'$$

έχουμε ότι  $first(*FT') = \{ * \}$ . Άρα στη θέση  $(T', *)$  του πίνακα ανίχνευσης τοποθετούμε τον κανόνα  $T' \rightarrow *FT'$ . Το  $\epsilon$  ανήκει στο  $first(T')$ , άρα για κάθε τερματικό σύμβολο που ανήκει στο  $follow(T')$  πρέπει να εισάγουμε στον πίνακα ανίχνευσης τον κανόνα  $T' \rightarrow \epsilon'$ . Από τον πίνακα 2.2 έχουμε ότι  $follow(T') = \{ eof, , + \}$ . Άρα στις θέσεις  $(T', eof)$ ,  $(T, ")$  και  $(T, +)$  του πίνακα ανίχνευσης τοποθετούμε τον κανόνα  $T' \rightarrow \epsilon$ .

Για τον κανόνα στη γραμμή 7 της γραμματικής:

$$F \rightarrow ( E )$$

έχουμε ότι  $first(" ( E )") = \{ ( \}$ . Άρα στη θέση  $(F, ")$  του πίνακα ανίχνευσης τοποθετούμε τον κανόνα  $F' \rightarrow (E)$ . Το  $\epsilon$  δεν ανήκει στο  $first(F)$ , οπότε προχωρούμε στον επόμενο κανόνα.

Για τον κανόνα στη γραμμή 8 της γραμματικής:

$$F \rightarrow id$$

έχουμε ότι  $first(id) = \{ id \}$ . Άρα στη θέση  $(F, id)$  του πίνακα ανίχνευσης τοποθετούμε τον κανόνα  $F' \rightarrow id$ . Το  $\epsilon$  δεν ανήκει στο  $first(F)$ , οπότε έχουμε ολοκληρώσει με όλους τους κανόνες της γραμματικής και μπορούμε τώρα να προχωρήσουμε στον σχηματισμό του πίνακα. Ο πίνακας ανίχνευσης ακολουθεί:

Πίνακας 2.3: Πίνακας ανίχνευσης για τη γραμματική των αριθμητικών εκφράσεων.

	+	*	(	)	<i>id</i>	eof
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow +TE'$		$E' \rightarrow \epsilon$
T			$T \rightarrow *FT'$		$T \rightarrow *FT'$	
T'	$T' \rightarrow \epsilon$	$T' * \rightarrow FT'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

## 2.5 Συντακτική ανάλυση με πίνακα ανίχνευσης

Έχοντας κατασκευάσει τον πίνακα ανίχνευσης μπορούμε να εξετάσουμε εάν μία συμβολοσειρά αναγνωρίζεται από τη γραμματική που εκφράζει ο πίνακας. Πρακτικά έχουμε στα χέρια μας έναν συντακτικό αναλυτή για τη γραμματική αυτή. Ο τρόπος με τον οποίο αυτό γίνεται είναι φυσικά μηχανιστικός. Χρειαζόμαστε, πέρα από τον πίνακα, μια στοίβα, η οποία αποτελεί τη μνήμη του συντακτικού αναλυτή. Να θυμίσουμε

εδώ την ιεραρχία Chomsky, την οποία εξετάσαμε στο κεφ. 1, σύμφωνα με την οποία υπάρχει ισοδυναμία ανάμεσα στις γραμματικές και τις γλώσσες χωρίς συμφραζόμενα και στα αυτόματα με στοίβα.

Πίνακας 2.4: Αναγνώριση της συμβολοσειράς  $(\alpha + \beta)$  από τη γραμματική των αριθμητικών εκφράσεων.

Στοίβα	Συμβολοσειρά Εισόδου	
E	$(\alpha + \beta) \$$	αρχικοποίηση
E'T	$(\alpha + \beta) \$$	$E \rightarrow TE'$
E'T'F	$(\alpha + \beta) \$$	$T \rightarrow FT'$
E'T')E(	$(\alpha + \beta) \$$	$F \rightarrow (E)$
E'T')E	$\alpha + \beta \$$	αναγνώριση "("
E'T')E'T	$\alpha + \beta \$$	$E \rightarrow TE'$
E'T')E'T'F	$\alpha + \beta \$$	$T \rightarrow FT'$
E'T')E'T'id	$\alpha + \beta \$$	$F \rightarrow id$
E'T')E'T'	$+ \beta \$$	αναγνώριση "α"
E'T')E'	$+ \beta \$$	$T' \rightarrow \epsilon$
E'T')E'T+	$+ \beta \$$	αναγνώριση "+"
E'T')E'T	$\beta \$$	$T \rightarrow FT'$
E'T')E'T'F	$\beta \$$	$F \rightarrow id$
E'T')E'T'id	$\beta \$$	αναγνώριση "β"
E'T')E'T'	$\$$	$T' \rightarrow \epsilon$
E'T')E'	$\$$	$E' \rightarrow \epsilon$
E'T')	$\$$	αναγνώριση ")"
E'T'	$\$$	$T' \rightarrow \epsilon$
E'	$\$$	$E' \rightarrow \epsilon$
	$\$$	επιτυχής αναγνώριση συμβολοσειράς

Η περιγραφή του αλγορίθμου ακολουθεί:

- Αρχικοποιούμε μία στοίβα με το αρχικό σύμβολο της γραμματικής:
- Σε κάθε βήμα ελέγχουμε την κορυφή της στοίβας και τον επόμενο χαρακτήρα στη συμβολοσειρά εισόδου.
  - Αν στην κορυφή της στοίβας βρίσκεται το τερματικό σύμβολο  $a$  και το επόμενο σύμβολο της συμβολοσειράς εισόδου είναι επίσης το τερματικό σύμβολο  $a$ , τότε θεωρούμε ότι το  $a$  αναγνωρίστηκε, αφαιρείται από τη στοίβα και καταναλώνεται από τη συμβολοσειρά εισόδου.
  - Αν στην κορυφή της στοίβας βρίσκεται το μη τερματικό σύμβολο  $A$ , το επόμενο σύμβολο της συμβολοσειράς εισόδου είναι το  $a$  και ο πίνακας ανίχνευσης στη θέση  $M(A, a)$  περιέχει τον κανόνα  $A \rightarrow \beta$ , τότε ο κανόνας  $A \rightarrow \beta$  θεωρείται ότι επιλέγεται προς ενεργοποίηση, το  $A$  αφαιρείται από τη στοίβα και τοποθετούνται εκεί τα σύμβολα που αποτελούν τη συμβολοσειρά  $\beta$  με αντίστροφη σειρά, δηλαδή από το δεξιότερο στο αριστερότερο, ώστε πάνω πάνω στη στοίβα να βρίσκεται το αριστερότερο.
  - Αν κανένα από τα παραπάνω δεν μπορεί να εφαρμοστεί, τότε αναγνωρίζεται συντακτικό σφάλμα και αποφαινόμαστε ότι η συμβολοσειρά δεν αναγνωρίζεται από τη γραμματική.
  - Αν αδειάσει η στοίβα και δεν έχει εξαντληθεί η συμβολοσειρά εισόδου, τότε αναγνωρίζεται πάλι συντακτικό σφάλμα και αποφαινόμαστε ότι η συμβολοσειρά δεν αναγνωρίζεται από τη γραμματική.

- Αν αδειάσει η στοίβα και έχει εξαντληθεί η συμβολοσειρά εισόδου, τότε έχουμε επιτυχή αναγνώριση της συμβολοσειράς.

Η εκτέλεση του αλγορίθμου για τη γραμματική των αριθμητικών εκφράσεων και είσοδο τη συμβολοσειρά ( $\alpha + \beta$ ) φαίνεται στον πίνακα 2.4.

### Βιβλιογραφία

- [1] Keith D. Cooper και Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2004. ISBN: 1558606998.
- [2] Keith D. Cooper και Linda Torczon. *Σχεδίαση και Κατασκευή Μεταγλωττιστών*. Ξενόγλωσσος τίτλος: Engineering a Compiler, Επιστημονική επιμέλεια έκδοσης: Γεώργιος Μανής, Νικόλαος Παπασπύρου και Αντώνιος Σαββίδης. Πανεπιστημιακές Εκδόσεις Κρήτης, 2018. ISBN: 9789605245191.
- [3] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 0321486811.
- [4] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Μεταγλωττιστές: Αρχές, τεχνικές και εργαλεία, 2η αμερικανική έκδοση*. Επιστημονική επιμέλεια έκδοσης: Π. Αλεφραγκής, Α. Βώρος, Ν. Βώρος, Κ. Μασσέλος. Εκδόσεις Νέων Τεχνολογιών, 2011. ISBN: 9789606759727.
- [5] Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.
- [6] Ζαφείρης Καραϊσκος. *Μεταγλωττιστές*. Εκδόσεις DaVinci, 2016. ISBN: 9789609732185.
- [7] Κωνσταντίνος Λάζος, Παναγιώτης Κατσαρός και Ζαφείρης Καραϊσκος. *Μεταγλωττιστές Γλωσσών Προγραμματισμού: Θεωρία & Πράξη*. Ζυγός, 2004. ISBN: 9608772346.

## ΚΕΦΑΛΑΙΟ 3

---

### Η ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ *C-impe*

---

#### Σύνοψη:

Η *C-impe* είναι μια μικρή, εκπαιδευτική, γλώσσα προγραμματισμού. Θυμίζει τη γλώσσα C, από την οποία αντλεί ιδέες και δομές, αλλά είναι αρκετά πιο μικρή, τόσο στις υποστηριζόμενες δομές, όσο φυσικά και σε προγραμματιστικές δυνατότητες.

Παρόλο που οι προγραμματιστικές της δυνατότητες είναι μικρές, η εκπαιδευτική αυτή γλώσσα προγραμματισμού περιέχει πλούσια στοιχεία και η κατασκευή του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον, αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται από δημοφιλείς γλώσσες προγραμματισμού, καθώς και κάποιες πρωτότυπες. Η *C-impe* υποστηρίζει δομές όπως οι δημοφιλείς `while` και η `if-else`, αλλά και τις πρωτότυπες και ενδιαφέρουσες στην υλοποίηση `forcase` και `incase`. Επίσης, υποστηρίζει συναρτήσεις και διαδικασίες, μετάδοση παραμέτρων με αναφορά και τιμή, αναδρομικές κλήσεις. Επιτρέπει φώλιασμα στη δήλωση συναρτήσεων και διαδικασιών.

Από την άλλη όμως πλευρά, η *C-impe* δεν προσφέρει βασικά προγραμματιστικά εργαλεία, όπως η δομή `for`, ή τύπους δεδομένων, όπως οι πραγματικοί αριθμοί και οι συμβολοσειρές ή οι πίνακες. Οι παραλήψεις αυτές έχουν γίνει για εκπαιδευτικούς λόγους, ώστε να απλουστευτεί η διαδικασία κατασκευής του μεταγλωττιστή. Είναι μία απλοποίηση που έχει να κάνει μόνο με τη μείωση των γραμμών κώδικα που πρέπει να γραφεί και όχι με τη δυσκολία της κατασκευής ή την εκπαιδευτική αξία της διαδικασίας ανάπτυξης.

Στο κεφάλαιο αυτό θα οριστεί η γλώσσα *C-impe*. Θα παρουσιαστεί η δομή ενός προγράμματος σε γλώσσα *C-impe*, οι λεκτικές μονάδες που συνθέτουν ένα πρόγραμμα, οι λέξεις κλειδιά που είναι δεσμευμένες, οι τελεστές που είναι διαθέσιμοι, οι προγραμματιστικές δομές που υποστηρίζονται, οι κανόνες φωλιάσματος συναρτήσεων και διαδικασιών, οι κανόνες εμβέλειας και ο τρόπος περάσματος παραμέτρων.

Στη συνέχεια του κεφαλαίου θα παρουσιαστούν η γραμματική της γλώσσας σε μορφή EBNF και θα δοθούν κάποια προγράμματα-παραδείγματα συνταγμένα με τους κανόνες της *C-impe*.

**Προαπαιτούμενη γνώση:**

- θεωρία τυπικών γραμματικών
  - κεφάλαιο 1
  - κεφάλαιο 2
- 

Η γλώσσα προγραμματισμού *C-imple* θα αποτελέσει τη βάση του ταξιδιού μας στην τεχνολογία των μεταγλωττιστών. Το όνομά της είναι συμβολικό. Έχει ως βάση τη γλώσσα προγραμματισμού C, γι' αυτό και δίνεται έμφαση στη γραφή της στο γράμμα C. Μετά το C ακολουθεί το -imple, από τη λέξη implementation. Η λέξη *C-imple*, ηχεί όπως και η λέξη simple, θυμίζοντας ότι πρόκειται για μία απλή γλώσσα, σχεδιασμένη για εκπαιδευτικούς σκοπούς.

Έτσι, θα κατασκευάσουμε και θα παρακολουθήσουμε βήμα-βήμα την ανάπτυξη ενός μεταγλωττιστή *C-imple*, χωρίς τη χρήση εργαλείων ανάπτυξης, αλλά μόνο μίας γλώσσας προγραμματισμού. Ο μεταγλωττιστής θα είναι πλήρως λειτουργικός και θα παράγει, ως τελική γλώσσα, τη γλώσσα μηχανής (assembly) του επεξεργαστή RISC-V [1, 2]. Χρησιμοποιώντας έναν εξομοιωτή του επεξεργαστή RISC-V θα είναι δυνατόν να τρέξουμε τη γλώσσα μηχανής που θα δημιουργεί ο μεταγλωττιστής. Η επιλογή της γλώσσας μηχανής του RISC-V ως γλώσσα στόχο, έχει να κάνει με εκπαιδευτικούς λόγους, αφού η επιλογή ενός εμπορικού επεξεργαστή δεν έχει να προσθέσει κάτι εκπαιδευτικά.

Τα αρχεία της *C-imple* έχουν κατάληξη .ci

### 3.1 Λεκτικές μονάδες

Το αλφάριθμο της *C-imple* αποτελείται από:

- τα μικρά και κεφαλαία γράμματα της λατινικής αλφαριθμητικής (A, ..., Z και a, ..., z),
- τα αριθμητικά ψηφία (0, ..., 9),
- τα σύμβολα των αριθμητικών πράξεων (+, -, \*, /),
- τους τελεστές συσχέτισης (<, >, =, <=, >=, <>)
- το σύμβολο ανάθεσης (:=),
- τους διαχωριστές (;, ;),
- τα σύμβολα ομαδοποίησης ([, ], (, ), {, }),
- το σύμβολο τερματισμού του προγράμματος (. ),
- και τα σύμβολα διαχωρισμού σχολίων (#).

Τα σύμβολα [ , ] χρησιμοποιούνται στις λογικές παραστάσεις όπως τα σύμβολα ( , ) στις αριθμητικές παραστάσεις.

Οι δεσμευμένες λέξεις είναι:

```
program      declare
if           else        while
switchcase   forcase    incase   case     default
not          and         or
function     procedure  call     return   in       inout
input        print
```

Οι λέξεις αυτές δεν μπορούν να χρησιμοποιηθούν ως μεταβλητές. Οι σταθερές της γλώσσας είναι ακέραιες σταθερές που αποτελούνται από προαιρετικό πρόσημο και από μία ακολουθία αριθμητικών ψηφίων. Οι ακέραιες σταθερές πρέπει να έχουν τιμές από  $-(2^{32} - 1)$  έως  $2^{32} - 1$ .

Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία, αρχίζοντας όμως από γράμμα. Ο μεταγλωττιστής λαμβάνει υπόψη του μόνο τα τριάντα πρώτα γράμματα. Αναγνωριστικά με περισσότερους από 30 χαρακτήρες θεωρούνται λανθασμένα.

Οι λευκοί χαρακτήρες (tab, space, return) αγνοούνται και μπορούν να χρησιμοποιηθούν με οποιονδήποτε τρόπο χωρίς να επηρεάζεται η λειτουργία του μεταγλωττιστή, αρκεί, βέβαια, να μην βρίσκονται μέσα σε δεσμευμένες λέξεις, αναγνωριστικά, σταθερές.

Το ίδιο ισχύει και για τα σχόλια, τα οποία πρέπει να βρίσκονται μέσα σε σύμβολα #

### 3.2 Μορφή προγράμματος

Κάθε πρόγραμμα ξεκινάει με τη λέξη κλειδί program. Στη συνέχεια ακολουθεί ένα αναγνωριστικό (όνομα) για το πρόγραμμα αυτό. Μέσα σε άγκιστρα τοποθετούνται τα τρία βασικά μπλοκ του προγράμματος: οι δηλώσεις των μεταβλητών (declarations), οι συναρτήσεις και διαδικασίες (subprograms), οι οποίες μπορούν και να είναι φωλιασμένες μεταξύ τους, και οι εντολές του κυρίως προγράμματος (statements). Η δομή ενός προγράμματος C-implement φαίνεται παρακάτω:

```
program id
{
    declarations
    subprograms
    statements
}.
```

### 3.3 Τύποι και δηλώσεις μεταβλητών

Ο μοναδικός τύπος δεδομένων που υποστηρίζει η C-implement είναι οι ακέραιοι αριθμοί. Η δήλωση γίνεται με την εντολή declare. Ακολουθούν τα ονόματα των αναγνωριστικών χωρίς καμία άλλη δήλωση, αφού γνωρίζουμε ότι πρόκειται για ακέραιες μεταβλητές και χωρίς να είναι αναγκαίο να βρίσκονται στην ίδια γραμμή. Οι μεταβλητές χωρίζονται μεταξύ τους με κόμματα. Το τέλος της δήλωσης αναγνωρίζεται με το ελληνικό ερωτηματικό. Επιτρέπεται να έχουμε περισσότερες των μία συνεχόμενες χρήσεις της declare.

### 3.4 Τελεστές και εκφράσεις

Η προτεραιότητα των τελεστών από τη μεγαλύτερη στη μικρότερη είναι:

- πολλαπλασιαστικοί: \*, /
- προσθετικοί: +, -
- σχεσιακοί =, <, >, <>, <=, >=
- λογικοί: not
- λογικό and
- λογικό or

### 3.5 Δομές της γλώσσας

#### 3.5.1 Εκχώρηση

Χρησιμοποιείται για την ανάθεση της τιμής μιας μεταβλητής ή μιας σταθεράς, ή μιας έκφρασης σε μία μεταβλητή.

Σύνταξη:

```
ID := expression
```

#### 3.5.2 Απόφαση ``if''

Η εντολή απόφασης `if` εκτιμά εάν ισχύει η συνθήκη `condition` και, εάν πράγματι ισχύει, τότε εκτελούνται οι εντολές `statements1` που το ακολουθούν. Το `else` δεν αποτελεί υποχρεωτικό τμήμα της εντολής και γι' αυτό βρίσκεται σε αγκύλη. Οι εντολές `statements2` που ακολουθούν το `else` εκτελούνται εάν η συνθήκη `condition` δεν ισχύει.

Σύνταξη:

```
if (condition)
    statements1
[else
    statements2]
```

#### 3.5.3 Επανάληψη ``while''

Η εντολή επανάληψης `while` επαναλαμβάνει τις εντολές `statements`, όσο η συνθήκη `condition` ισχύει. Αν την πρώτη φορά που θα αποτιμηθεί η `condition`, το αποτέλεσμα της αποτίμησης είναι ψευδές, τότε οι `statements` δεν εκτελούνται ποτέ.

Σύνταξη:

```
while (condition)
    statements
```

#### 3.5.4 Επιλογή ``switchcase''

Η δομή `switchcase` ελέγχει τις `condition` που βρίσκονται μετά τα `case`. Μόλις μία από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες `statements1` (που ακολουθούν το `condition`). Μετά ο έλεγχος μεταβαίνει έξω από την `switchcase`. Αν, κατά το πέρασμα, καμία από τις `case` δεν ισχύσει, τότε ο έλεγχος μεταβαίνει στην `default` και εκτελούνται οι `statements2`. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την `switchcase`.

Σύνταξη:

```
switchcase
( case (condition) statements1 )*
default statements2
```

#### 3.5.5 Επανάληψη ``forcase''

Η δομή επανάληψης `forcase` ελέγχει τις `condition` που βρίσκονται μετά τα `case`. Μόλις μία από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες `statements1` (που ακολουθούν το `condition`). Μετά ο έλεγχος μεταβαίνει στην αρχή της `forcase`. Αν καμία από τις `case` δεν ισχύει, τότε ο έλεγχος μεταβαίνει στη `default` και εκτελούνται οι `statements2`. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την `forcase`.

Σύνταξη:

```
forcase
  ( case (condition) statements1 )*
  default statements2
```

### 3.5.6 Επανάληψη ``incase''

Η δομή επανάληψης `incase` ελέγχει τις *condition* που βρίσκονται μετά τα `case`, εξετάζοντάς τες κατά σειρά. Για καθεμία από αυτές που η αντίστοιχη *condition* ισχύει, εκτελούνται οι αντίστοιχες *statements* (που ακολουθούν το *condition*). Θα εξεταστούν όλες οι *condition* και θα εκτελεστούν όλες οι *statements* των οποίων οι *condition* ισχύουν. Αφότου εξεταστούν όλες οι `case`, ο έλεγχος μεταβαίνει έξω από τη δομή `in-case` εάν καμία από τις *statements* δεν έχει εκτελεστεί ή μεταβαίνει στην αρχή της `incase`, εάν έστω και μία από τις *statements* έχει εκτελεστεί.

Σύνταξη:

```
incase
  ( case (condition) statements )*
```

### 3.5.7 Επιστροφή τιμής συνάρτησης

Χρησιμοποιείται μέσα σε συναρτήσεις για να επιστραφεί το αποτέλεσμα της συνάρτησης, το οποίο είναι το αποτέλεσμα της αποτίμησης του *expression*.

Σύνταξη:

```
return (expression)
```

### 3.5.8 Έξοδος δεδομένων

Εμφανίζει στην οθόνη το αποτέλεσμα της αποτίμησης του *expression*.

Σύνταξη:

```
print (expression)
```

### 3.5.9 Είσοδος δεδομένων

Ζητάει από τον χρήστη να δώσει μία τιμή μέσα από το πληκτρολόγιο. Η τιμή που θα δώσει θα μεταφερθεί στη μεταβλητή *ID*.

Σύνταξη:

```
input (ID)
```

### 3.5.10 Κλήση διαδικασίας

Καλεί μία διαδικασία.

Σύνταξη:

```
call functionName(actualParameters)
```

## 3.6 Συναρτήσεις και διαδικασίες

Η *C-impe* υποστηρίζει συναρτήσεις και διαδικασίες.

Για τις συναρτήσεις η σύνταξη είναι:

```
function ID(formalPars)
{
    declarations
    subprograms
    statements
}
```

ενώ για τις διαδικασίες:

```
procedure ID(formalPars)
{
    declarations
    subprograms
    statements
}
```

Η *formalPars* είναι η λίστα των τυπικών παραμέτρων. Οι συναρτήσεις και οι διαδικασίες μπορούν να φωλιάσουν η μία μέσα στην άλλη και οι κανόνες εμβέλειας είναι όπως της PASCAL. Η επιστροφή της τιμής μιας συνάρτησης γίνεται με την *return*.

Η κλήση μιας συνάρτησης, γίνεται από τις αριθμητικές παραστάσεις ως τελούμενο, π.χ.

```
D = a + f(in x)
```

όπου *f* η συνάρτηση και *x* παράμετρος που περνάει με τιμή.

Η κλήση της διαδικασίας, γίνεται με την *call*, π.χ.

```
call f(inout x)
```

όπου *f* η διαδικασία και *x* παράμετρος που περνάει με αναφορά.

### 3.7 Μετάδοση παραμέτρων

Η *C-impl* υποστηρίζει δύο τρόπους μετάδοσης παραμέτρων:

- *Με τιμή*. Δηλώνεται με τη λεκτική μονάδα *in*. Άλλαγές στην τιμή της δεν επιστρέφονται στο πρόγραμμα που κάλεσε τη συνάρτηση.
- *Με αναφορά*. Δηλώνεται με τη λεκτική μονάδα *inout*. Κάθε αλλαγή στην τιμή της μεταφέρεται αμέσως στο πρόγραμμα που κάλεσε τη συνάρτηση.

Στην κλήση μίας συνάρτησης οι πραγματικοί παράμετροι συντάσσονται μετά από τις λέξεις κλειδιά *in* και *inout*, ανάλογα με το αν περνούν με τιμή ή αναφορά.

### 3.8 Κανόνες εμβέλειας

Καθολικές ονομάζονται οι μεταβλητές που δηλώνονται στο κυρίως πρόγραμμα και είναι προσβάσιμες σε όλους. Τοπικές είναι οι μεταβλητές που δηλώνονται σε μία συνάρτηση ή διαδικασία και είναι προσβάσιμες μόνο μέσα από τη συγκεκριμένη συνάρτηση ή διαδικασία.

Ισχύουν οι ακόλουθοι κανόνες εμβέλειας:

- Κάθε συνάρτηση ή διαδικασία, εκτός των τοπικών μεταβλητών, των παραμέτρων της και των καθολικών μεταβλητών, έχει επίσης πρόσβαση και στις μεταβλητές που έχουν δηλωθεί σε συναρτήσεις ή διαδικασίες προγόνους ή και ως παράμετροι αυτών.

- Ισχύει ο δημοφιλής κανόνας ότι, αν δύο (ή περισσότερες) μεταβλητές ή παράμετροι έχουν το ίδιο όνομα και έχουν δηλωθεί σε διαφορετικό επίπεδο φωλιάσματος, τότε οι τοπικές μεταβλητές και παράμετροι υπερκαλύπτουν τις μεταβλητές και παραμέτρους των προγόνων, οι οποίες με τη σειρά τους υπερκαλύπτουν τις καθολικές μεταβλητές.
- Ίδιοι κανόνες εμβέλειας με τη δήλωση μεταβλητών ακολουθούνται και στη δήλωση συναρτήσεων και διαδικασιών.
- Μία συνάρτηση ή διαδικασία μπορεί να καλέσει αναδρομικά τον εαυτό της, όποια συνάρτηση ή διαδικασία έχει οριστεί πριν από αυτήν στο ίδιο βάθος φωλιάσματος (είναι δηλαδή αδελφός), καθώς και τα παιδιά της.
- Μία συνάρτηση ή διαδικασία δεν μπορεί να καλέσει τα εγγόνια της. Αυτός, άλλωστε, είναι ο λόγος που έχουμε “κρύψει” τα εγγόνια μέσα στα παιδιά, ώστε να μην υπάρχει πρόσβαση παρά μόνο από τον γονέα και τα αδέλφια τους.
- Μία συνάρτηση δεν μπορεί να καλέσει τον γονέα της. Σε επίπεδο υλοποίησης, όταν μεταφράζεται μία συνάρτηση, ο γονέας της υπάρχει ως εγγραφή στον πίνακα συμβόλων, αλλά δεν έχουν συμπληρωθεί ακόμα όλα τα πεδία του.

### 3.9 Η γραμματική της *C-imple*

Ακολουθεί η γραμματική της *C-imple* η οποία δίνει και την ακριβή περιγραφή της γλώσσας:

```

# "program" is the starting symbol
# followed by its name and a block
# Every program ends with a fullstop
program      →  program ID
               block
               .
               .

# a block consists of declarations, subprograms and statements
block        →  {
               declarations
               subprograms
               blockstatements
               }

# declaration of variables
# Kleene star implies zero or more "declare" statements
declarations →  ( declare varlist ; )*
              

# a list of variables following the declaration keyword
varlist      →  ID
               ( , ID )*
               |
               ε

# zero or more subprograms
subprograms  →  ( subprogram )*

```

```

# a subprogram is a function or a procedure,
# followed by parameters and block
subprogram → function ID ( formalparlist )
    block
    | procedure ID ( formalparlist )
    block

# list of formal parameters
# one or more parameters are allowed
formalparlist → formalparitem
    ( , formalparitem )*
    |
    ε

# a formal parameter
# "in": by value, "inout" by reference
formalparitem → in ID
    | inout ID

# one or more statements
# more than one statements should be grouped with brackets
statements → statement ;
    |
    {
        statement
        ( ; statement )*
    }

# statements considered as block (used in program and subprogram)
blockstatements → statement
    ( ; statement )*

# one statement
statement → assignStat
    |
    ifStat
    |
    whileStat
    |
    switchcaseStat
    |
    forcaseStat
    |
    incaseStat
    |
    callStat
    |
    returnStat
    |
    inputStat
    |
    printStat
    |
    ε

# assignment statement
assignStat → ID := expression

# if statement
ifStat → if ( condition )
    statements
    elsepart

```

```

# else part is optional
elsepart → else
    statements
|   ε

# while statement
whileStat → while ( condition )
    statements

# switch statement
switchcaseStat → switchcase
    ( case ( condition ) statements )*
    default statements

# forcase statement
forcaseStat → forcase
    ( case ( condition ) statements )*
    default statements

# incase statement
incaseStat → incase
    ( case ( condition ) statements )*

# return statement
returnStat → return( expression )

# call statement
callStat → call ID( actualparlist )

# print statement
printStat → print( expression )

# input statement
inputStat → input( ID )

# list of actual parameters
actualparlist → actualparitem
    ( , actualparitem )*
|   ε

# an actual parameter
# "in": by value, "inout" by reference
actualparitem → in expression
|   inout ID

# boolean expression
condition → boolterm
    ( or boolterm )*

# term in boolean expression
boolterm → boolfactor
    ( and boolfactor )*

```

```

# factor in boolean expression
boolfactor → not [ condition ]
| [ condition ]
| expression REL_OP expression

# arithmetic expression
expression → optionalSign term
( ADD_OP term )*

# term in arithmetic expression
term → factor
( MUL_OP factor )*

# factor in arithmetic expression
factor → INTEGER
| ( expression )
| ID idtail

# follows a function or procedure
# describes parentheses and parameters
idtail → ( actualparlist )
| ε

# symbols "+" and "-" (are optional)
optionalSign → ADD_OP
| ε

#####
# lexer rules: relational, arithmetic operations,
# integer values and ids

REL_OP → = | <= | >= | > | < | <>

ADD_OP → + | -

MUL_OP → * | /

INTEGER → [0-9]++

ID → [a-zA-Z][a-zA-Z0-9]*

```

### 3.10 Παραδείγματα κώδικα

Παρακάτω δίνονται τρία μικρά παραδείγματα σε γλώσσα C-imple: ο υπολογισμός του παραγοντικού, η ακολουθία Fibonacci και η μέτρηση των ψηφίων ενός αριθμού.

Υπολογισμός παραγοντικού:

```
program factorial
{
    # declarations #
    declare x;
    declare i,fact;

    # main #
    input(x);
    fact:=1;
    i:=1;
    while (i<=x)
    {
        fact:=fact*i;
        i:=i+1;
    };
    print(fact);
}.
```

H ακολούθια Fibonacci:

```
program fibonacci
{
    declare x;

    function fibonacci(in x)
    {
        return (fibonacci(in x-1)+fibonacci(in x-2));
    }

    # main #
    input(x);
    print(fibonacci(in x));
}.
```

Μέτρηση των ψηφίων ενός αριθμού:

```
program countDigits
{
    declare x, count;

    # main #
    input(x);
    count := 0;
    while (x>0)
    {
        x := x/10;
        count := count+1;
    };
    print(count);
}.
```

Υπολογισμός των αθροίσματος  $1 + 2 + \dots + N$ , με την *forcase*:

```
program summation
{
    declare x,sum;

    # main #
    input(x);
    sum := 0;

    forcase
        case(x>0)
        {
            sum := sum+x;
            x := x-1
        }
        default
            print(sum);
}.

```

Υπολογισμός των πρώτων 30 πρώτων αριθμών:

```
program primes
{
    # declarations for main #
    declare i;

    function isPrime(in x)
    {
        # declarations for isPrime #
        declare i;

        function divides(in x, in y)
        {
            # body of divides #
            if (y = (y/x)*x)
                return (1);
            else
                return (0);
        }

        # body of isPrime #
        i:=2;
        while (i<x)
        {
            if (divides(in i, in x)=1)
                return(0);
            i := i + 1
        };
        return(1)
    }
}
```

```

# body of main #
i := 2;
while (i<=30)
    if (isPrime(in i)=1)
        print(i);
}.

```

Παρατηρήστε ότι σε δύο σημεία του κώδικα η *C-imple* απαίτησε από τον προγραμματιστή να τοποθετήσει δύο συνεχόμενα ελληνικά ερωτηματικά (μετά από το δεύτερο *return* και μετά από το *print*). Ο κανόνας στη *C-imple* λέει ότι κάθε εκτελούμενη εντολή τερματίζει με ερωτηματικό. Δεν είναι απαραίτητο το ερωτηματικό όταν η εντολή αυτή είναι η τελευταία ενός μπλοκ. Στην περίπτωση της *return* στο σημείο αυτό τερματίζουν οι εντόλες *return*, *if* και *while*. Η *while* ως τελευταία στο μπλοκ δεν απαιτεί ερωτηματικό (αν βάλουμε δεν είναι συντακτικό λάθος). Άρα χρειαζόμαστε τουλάχιστον δύο ερωτηματικά εκεί. Στην περίπτωση της *print* στο σημείο αυτό τερματίζουν οι εντόλες *print*, *if* και *while*. Η *while* ως τελευταία στο μπλοκ δεν απαιτεί ερωτηματικό. Άρα και εδώ χρειαζόμαστε τουλάχιστον δύο ερωτηματικά.

### 3.11 Γραμματικές δημοφιλών γλωσσών προγραμματισμού

Αν αναζητήσει κανείς στο διαδίκτυο γραμματικές γλωσσών προγραμματισμού, χρησιμοποιώντας τις κατάλληλες λέξεις κλειδιά, θα του επιστραφούν σύνδεσμοι για γραμματικές από πολλές γλώσσες προγραμματισμού, ακόμα και εκπαιδευτικές.

Η γραμματική της C, την οποία σας προτείνεται να αναζητήσετε και να μελετήσετε λόγω της ιδιαίτερης, ελεύθερης δομής που έχει η γλώσσα, είναι η γραμματική που έχει γραφεί από τους D. Ritchie και B. Kernighan [3]. Είναι ένα από τα περισσότερο διαβασμένα βιβλία. Προς το τέλος του βιβλίου τους, οι Ritchie και Kernighan παραθέτουν τη γραμματική της ANSI C.

Αν θέλετε να αναζητήσετε γραμματική για την Pascal, δείτε το πρότυπο ISO7185, στο οποίο περιγράφεται η γλώσσα και παρατίθεται η γραμματική της [4]. Πρότυπα άλλων γλωσσών έχουν επίσης περιγραφεί σε ISO.

Για την Java σας προτείνεται να αναζητήσετε τη γραμματική της στο site της Oracle [5], ενώ για την Python στο python.org [6].

### Βιβλιογραφία

- [1] Andrew Waterman και Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Αδημοσίευτη ερευνητική εργασία. EECS Department, University of California, Berkeley, 2019.
- [2] Andrew Waterman, Krste Asanović και John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Αδημοσίευτη ερευνητική εργασία. EECS Department, University of California, Berkeley, 2021.
- [3] D.M. Ritchie και B.W. Kernighan. *The C programming language*. Bell Laboratories, 1988.
- [4] *Pascal*, ISO 7185:1990. <https://www.iso.org/standard/13802.html>. τελευταία πρόσβαση: Σεπτέμβριος 2022.
- [5] *Oracle: Java Language Specification*. <https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html>. τελευταία πρόσβαση: Σεπτέμβριος 2022.
- [6] *Python: Full Grammar Specification*. <https://docs.python.org/3/reference/grammar.html>. τελευταία πρόσβαση: Σεπτέμβριος 2022.



## ΚΕΦΑΛΑΙΟ 4

---

### ΛΕΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

---

#### Σύνοψη:

Στο κεφάλαιο αυτό θα παρουσιαστεί η διαδικασία ανάπτυξης ενός λεκτικού αναλυτή. Αποτελεί την πρώτη φάση της μεταγλωττισης κατά την οποία το αρχικό πρόγραμμα αποσυντίθεται σε μία σειρά από λεκτικές μονάδες.

Θα μελετήσουμε τις βασικές έννοιες και τους κύριους μηχανισμούς που χρησιμοποιούνται στην ανάπτυξη του λεκτικού αναλυτή, έχοντας ως κύριο στόχο την κατανόηση όλων των λεπτομερειών της υλοποίησης. Έτσι, χωρίς να χρησιμοποιηθεί κανένα εργαλείο αυτοματοποιημένης ανάπτυξης, αλλά χρησιμοποιώντας μια γλώσσα προγραμματισμού υψηλού επιπέδου ως το μόνο μέσο υλοποίησης, θα δούμε βήμα-βήμα την ανάπτυξη ενός λεκτικού αναλυτή, χωρίς να θεωρήσουμε ως δεδομένες κάποιες ενότητες κώδικα.

Θα δούμε τη θέση που έχει ο λεκτικός αναλυτής μέσα στον κώδικα του μεταγλωττιστή και τι πληροφορία μεταφέρει στον συντακτικό αναλυτή. Θα μελετήσουμε και θα κατανοήσουμε την εσωτερική του λειτουργία και τον αλγόριθμο αναγνώρισης λεκτικών μονάδων που χρησιμοποιεί. Έτσι, θα κατασκευάσουμε και θα έχουμε ως βάση στη μελέτη μας ένα αυτόματο το οποίο θα αναγνωρίζει τις λεκτικές μονάδες ενός προγράμματος *C-imperative*. Θα εξηγήσουμε αναλυτικά τις καταστάσεις και τις μεταβάσεις του και θα στοχεύσουμε σε λεπτομέρειες της υλοποίησης που θέλουν προσοχή. Τέλος, θα δούμε πώς ένας λεκτικός αναλυτής αντιμετωπίζει τα σχόλια, πώς ανιχνεύει και αντιμετωπίζει τα σφάλματα.

#### Προαπαιτούμενη γνώση:

- θεωρία αυτομάτων
  - κεφάλαιο 1
  - κεφάλαιο 3
- 

Η λεκτική ανάλυση αποτελεί την πρώτη φάση της μεταγλωττισης. Όλες οι φάσεις της μεταγλωττισης πα-

Γεώργιος Μανής (2023). «Εγχειρίδιο Σχεδίασης και Ανάπτυξης Μεταγλωττιστών».

Αθήνα: Κάλλιπος, Ανοικτές Ακαδημαϊκές Εκδόσεις. <http://dx.doi.org/10.57713/kallipos-372>

 Creative Commons Αναφορά Δημιουργού - Μη Εμπορική Χρήση - Παρόμοια Διανομή 4.0

ρουσιάστηκαν στην εισαγωγή του συγγράμματος αυτού. Για περισσότερη ανάγνωση πάνω στις φάσεις στις οποίες χωρίζεται ένας μεταγλωττιστής μπορείτε να ανατρέξετε στα κεφάλαια των βιβλίων [1, κεφ.1], [2, κεφ.1], [3, κεφ.1], [4, κεφ.1], [5, κεφ.1], [6, κεφ.1], [7, κεφ.1], [8, κεφ.3], αλλά και στην εργασία/σύνοψη [9].

Κατά τη φάση αυτή διαβάζεται το αρχικό πρόγραμμα (το οποίο συνηθίζεται να ονομάζεται και πηγαίο πρόγραμμα) και παράγονται οι λεκτικές μονάδες. Χρησιμοποιούμε τον όρο λεκτική μονάδα για να αναπαραστήσουμε οτιδήποτε έχει νόημα να θεωρηθεί ως αυτόνομο σύνολο συνεχόμενων χαρακτήρων που μπορεί να συναντηθεί σε ένα πρόγραμμα και βρίσκει σημασιολογία στην υπό υλοποίηση γλώσσα. Θα τις ονομάζαμε ίσως λέξεις σε μία φυσική γλώσσα, αλλά σε ένα πρόγραμμα μας ενδιαφέρει να εντοπίσουμε και ακολουθίες χαρακτήρων, όπως ==, +, (, }, άρα ο όρος λέξη δεν είναι δόκιμος.

Η λεκτική ανάλυση συνήθως γίνεται με τη χρήση αυτοματοποιημένων εργαλείων. Πολύ γνωστό εργαλείο ανάπτυξης λεκτικών αναλυτών, και από τα πρώτα που έγιναν διαθέσιμα και χρησιμοποιήθηκαν ευρέως, είναι το μετα-εργαλείο *lex*. Το μεταγενέστερο *antlr* είναι, επίσης, μία πολύ καλή επιλογή, περισσότερο ώριμη προσέγγιση και ευκολότερη για τον προγραμματιστή. Θα ασχοληθούμε αρκετά με αυτό στο αντίστοιχο κεφάλαιο, παρακάτω (κεφ. 12), όταν θα εξετάσουμε εργαλεία αυτοματοποιημένης ανάπτυξης. Σε σημερινές γλώσσες προγραμματισμού, όπως η Java και η Python, υπάρχουν διαθέσιμες βιβλιοθήκες που απλοποιούν σημαντικά την υλοποίηση ενός λεκτικού αναλυτή.

Στο κεφάλαιο αυτό μας ενδιαφέρει να μελετήσουμε τις βασικές έννοιες και μηχανισμούς της λεκτικής ανάλυσης, έχοντας ως κύριο στόχο την κατανόηση όλων των λεπτομερειών της ανάπτυξης ενός λεκτικού αναλυτή. Έτσι, χωρίς να χρησιμοποιηθεί κανένα εργαλείο αυτοματοποιημένης ανάπτυξης, αλλά χρησιμοποιώντας μια γλώσσα προγραμματισμού υψηλού επιπέδου ως το μόνο μέσο υλοποίησης, θα δούμε βήμα-βήμα την ανάπτυξη ενός λεκτικού αναλυτή, χωρίς να θεωρήσουμε ως δεδομένη κάποια ενότητα κώδικα.

Αξιοσημείωτη βιβλιογραφία για τη λεκτική ανάλυση περιλαμβάνει τα ακόλουθα κεφάλαια σε βιβλία [1, κεφ.3], [2, κεφ.3], [3, κεφ.3], [4, κεφ.2], [5, κεφ.2], [6, κεφ.2], [7, κεφ.2].

#### 4.1 Ο λεκτικός αναλυτής στη διαδικασία της μετάφρασης

Στο εισαγωγικό κεφάλαιο είδαμε την αλληλουχία των φάσεων της ανάπτυξης ενός μεταγλωττιστή. Εκεί, η λεκτική ανάλυση αποτελούσε χρονικά την πρώτη φάση της ανάπτυξης. Εάν, όμως, σχεδιάζαμε ένα διάγραμμα το οποίο να δείχνει τη δομή ενός μεταγλωττιστή με βάση τη διάρθρωση των μονάδων λογισμικού που το αποτελούν, τότε θα εμφανίζόταν ως μία μονάδα λογισμικού η οποία καλείται από τον συντακτικό αναλυτή.

Ο λεκτικός αναλυτής υλοποιείται ως μία συνάρτηση, η οποία διαβάζει έναν-έναν τους χαρακτήρες από την είσοδο (το αρχικό πρόγραμμα) και κάθε φορά που καλείται επιστρέφει την επόμενη διαθέσιμη λεκτική μονάδα. Στην περίπτωση που ο λεκτικός αναλυτής αναγνωρίσει κάποιο σφάλμα, τότε οφείλει να πληροφορήσει σχετικά τον συντακτικό αναλυτή ή να διακόψει τη μεταγλωττιση και να επιστρέψει στον χρήστη ένα διαφωτιστικό μήνυμα λάθους. Η αλληλεπίδραση του λεκτικού αναλυτή με το περιβάλλον του φαίνεται στο σχήμα 4.1, όπου έχει επιλεγεί η δεύτερη λύση, όπως και στο υπόλοιπο κεφάλαιο.



Σχήμα 4.1: Ο λεκτικός αναλυτής.

Όταν ο λεκτικός αναλυτής αναγνωρίσει μία λεκτική μονάδα, τότε την επιστρέφει στον συντακτικό ανα-

λυτή, επιστρέφοντάς του και τον έλεγχο. Μία λεκτική μονάδα είναι ένα αντικείμενο το οποίο περιέχει τρία πεδία. Το πρώτο είναι η συμβολοσειρά την οποία αναγνώρισε. Ας την ονομάσουμε `recognized_string`. Το δεύτερο εντάσσει τη συμβολοσειρά αυτή σε μία κατηγορία η οποία έχει νόημα για τη συντακτική ανάλυση. Ας της δώσουμε το όνομα `family`. Τέτοιες κατηγορίες μπορεί να είναι οι ακόλουθες:

- *identifier*: ονόματα μεταβλητών, συναρτήσεων, διαδικασιών, σταθερών και ό,τι άλλο μπορεί να έχει ονομασία σε ένα πρόγραμμα. Στη φάση αυτή είναι πολύ νωρίς για να μπορέσουμε να διαχωρίσουμε αν η συμβολοσειρά που αναγνωρίσαμε αποτελεί όνομα μεταβλητής, συνάρτησης, διαδικασίας ή σταθεράς, ακόμα κι αν είναι νόμιμη (π.χ. δηλωμένη μεταβλητή ή όχι) και για τον λόγο αυτό δημιουργούμε μία γενική κατηγορία, κάτω από την οποία τοποθετούμε όλες αυτές τις περιπτώσεις.
- *number*: αριθμητικές σταθερές
- *keyword*: περιέχει τις λέξεις κλειδιά της γλώσσας, π.χ.: `while`, `if`, `else`
- *addOperator*: προσθετικοί αριθμητικοί τελεστές: `+`, `-`
- *mulOperator*: πολλαπλασιαστικοί αριθμητικοί τελεστές: `*`, `/`
- *relOperator*: λογικοί τελεστές, π.χ.: `==`, `>=`, `<`, `<>`
- *assignment*: τελεστής εκχώρησης (`:=`)
- *delimiter*: διαχωριστές, π.χ.: `,`, `.`, `;`
- *groupSymbol*: σύμβολα ομαδοποίησης, π.χ.: `(`, `)`, `{`, `}`, `[`, `]`

Θα μπορούσαμε, φυσικά, να ορίσουμε πολλές ακόμα κατηγορίες, ανάλογα με τις ανάγκες της γλώσσας, όπως, η συμβολοσειρά, οι τελεστές σε bit (bitwise operators), μαθηματικές συναρτήσεις, κ.λ.π.

Ένα ακόμα πεδίο του αντικειμένου που αναπαριστά μια λεκτική μονάδα είναι ο αριθμός γραμμής στην οποία αναγνωρίστηκε. Η πληροφορία αυτή είναι χρήσιμη στον συντακτικό αναλυτή, έτσι ώστε να μπορεί να επιστρέψει εύστοχα μηνύματα σφάλματος, αν εντοπίσει κάποιο σφάλμα κατά τη συντακτική ανάλυση. Αν αυτή η πληροφορία δεν διατηρηθεί μέσα στο αντικείμενο που επιστρέφεται στον συντακτικό αναλυτή, τότε θα χαθεί, αφού ο συντακτικός αναλυτής δεν χρησιμοποιεί άλλη πληροφορία, πέρα από αυτήν που του επιστρέφεται από τον συντακτικό αναλυτή. Ας το ονομάσουμε `line_number`.

Έτσι, μια λεκτική μονάδα μπορεί να αναπαρασταθεί με ένα αντικείμενο της κλάσης `Token`:

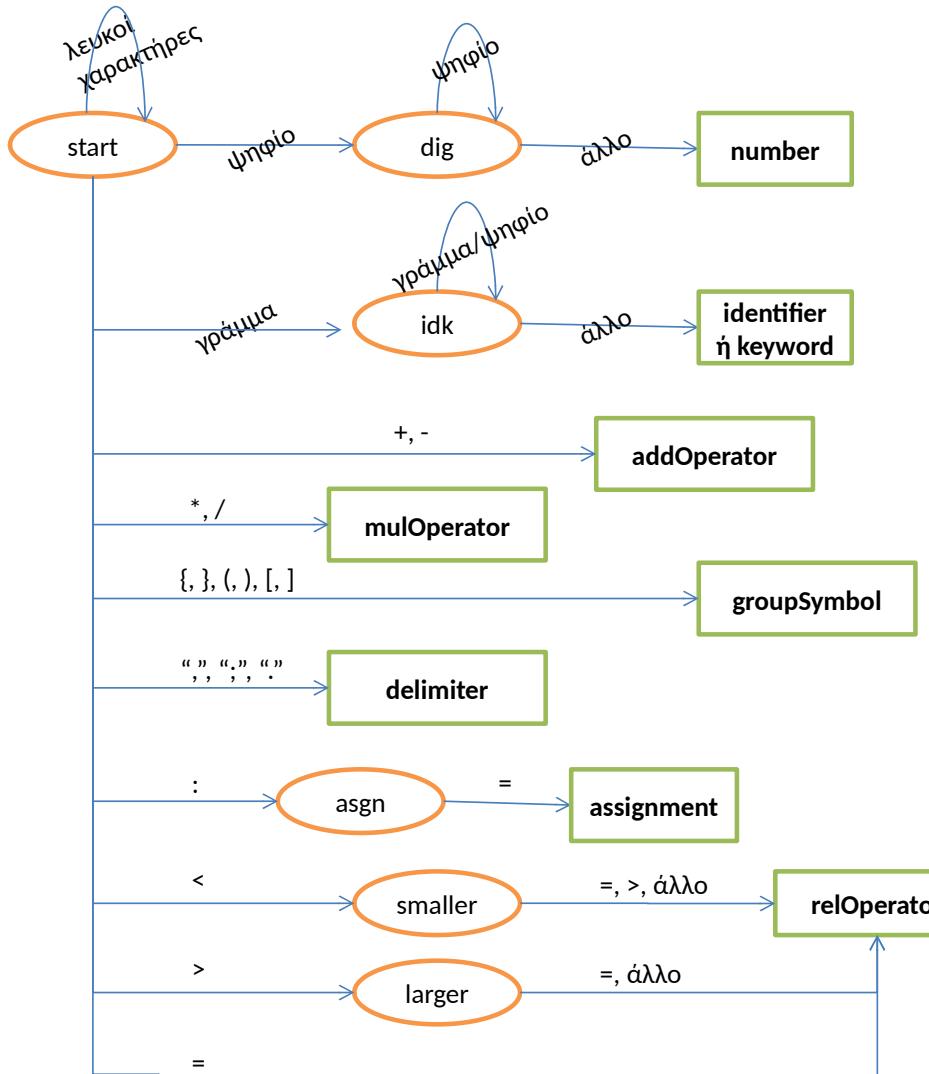
```
class Token:
    # properties: recognized_string, family, line_number
    def __init__(recognized_string, family, line_number):
        self.recognized_string = recognized_string
        self.family = family
        self.line_number = line_number
```

Ένα αντικείμενο της κλάσης `Token` περιέχει την πληροφορία που μεταφέρει ο λεκτικός αναλυτής στον συντακτικό αναλυτή. Για να δούμε πώς ο λεκτικός αναλυτής δημιουργεί και επιστρέφει στον συντακτικό αναλυτή τα `token`, τα αντικείμενα της κλάσης `Token` δηλαδή, ας μελετήσουμε αναλυτικά την εσωτερική του λειτουργία.

## 4.2 Η εσωτερική λειτουργία ενός λεκτικού αναλυτή

Ο λεκτικός αναλυτής υλοποιείται από μία συνάρτηση η οποία καλείται από τον συντακτικό αναλυτή και κάθε φορά που καλείται επιστρέφει την επόμενη διαθέσιμη στην είσοδο λεκτική μονάδα. Αυτό σημαίνει ότι σε κάθε του κλήση, ο λεκτικός αναλυτής σημειώνει το σημείο στο οποίο σταματάει την ανάγνωση του αρχικού προγράμματος και την επόμενη φορά που καλείται συνεχίζει την ανάγνωση από το σημείο αυτό.

Η λειτουργία του βασίζεται σε ένα αυτόματο. Για την *C-implement* το αυτόματο που αναγνωρίζει τις λεκτικές μονάδες φαίνεται στο σχήμα 4.2. Κάθε τέτοιο αυτόματο αποτελείται από την αρχική κατάσταση, κάποιες ενδιάμεσες και κάποιες τελικές. Στο αυτόματο του σχήματος η αρχική κατάσταση είναι η κατάσταση *start*, οι μη τελικές καταστάσεις συμβολίζονται με έλλειψη, ενώ οι τελικές με παραλληλόγραμμο. Ας ακολουθήσουμε το αυτόματο να δούμε με ποιον τρόπο αναγνωρίζει τις λεκτικές μονάδες της *C-implement*.



Σχήμα 4.2: Το αυτόματο λειτουργίας του λεκτικού αναλυτή.

Με την κλήση του λεκτικού αναλυτή το αυτόματο αρχικοποιείται στην κατάσταση *start*. Εάν στην είσοδο εμφανιστεί κάποιο ψηφίο, τότε το αυτόματο μεταβαίνει στην κατάσταση *dig*, η οποία είναι μη τελική. Όσο στην είσοδο εμφανίζονται ψηφία, τότε για κάθε ψηφίου που διαβάζεται, γίνεται μία μετάβαση, η οποία όμως δεν αλλάζει την κατάσταση του αυτομάτου, αφού είναι μετάβαση από την κατάσταση *dig* στην κατάσταση *dig*. Από την *dig* θα φύγει μόλις έρθει κάτι διαφορετικό, κάτι άλλο που δεν είναι ψηφίο. Όταν συμβεί αυτό, το αυτόματο θα μεταβεί στην τελική κατάσταση *number* και θα έχει αναγνωρίσει μία αριθμητική (ακέραια) σταθερά. Στον συντακτικό αναλυτή θα επιστραφεί:

- η αριθμητική σταθερά ως `recognized_string`,
- ως `family` η οικογένεια στην οποία ανήκει το `token`, που την περίπτωση αυτή είναι η οικογένεια `number`,
- και φυσικά ο αριθμός γραμμής στον οποίο βρέθηκε η αριθμητική σταθερά, το `line_number`.

Αν από την αρχική κατάσταση, αντί για ψηφίο έρθει γράμμα, τότε σύμφωνα με το αυτόματο θα μεταβούμε στην κατάσταση `idk`. Όμοια με την προηγούμενη περίπτωση, το αυτόματο θα παραμείνει στην κατάσταση `idk`, όσο στην είσοδο εμφανίζεται γράμμα ή ψηφίο. Μόλις εμφανιστεί κάτι διαφορετικό, κάτι που δεν είναι ούτε γράμμα ούτε ψηφίο, τότε το αυτόματο μεταβαίνει στην τελική κατάσταση `identifier/keyword`.

Από το όνομα της κατάστασης μπορούμε να καταλάβουμε ότι εδώ δεν έχουμε αναγνωρίσει (τουλάχιστον όχι ακόμα) κάποιο αναγνωριστικό (`identifier`), δηλαδή το όνομα μιας μεταβλητής, μίας συνάρτησης, κλπ., αφού στην τελική κατάσταση αυτή θα καταλήξει το αυτόματο και στην περίπτωση που θα συναντήσει κάποια λέξη κλειδί της γλώσσας. Έτσι, πριν βιαστούμε να επιστρέψουμε στον συντακτικό αναλυτή το αποτέλεσμα, πρέπει πρώτα να γίνει ο έλεγχος αν αυτό που αναγνωρίσαμε είναι λέξη κλειδί, που ανήκει στην κατηγορία `keyword`, ή είναι πράγματι ένας `intentifier` και ως τέτοιον πρέπει να τον χαρακτηρίσουμε.

Στο υπόλοιπο αυτόματο τα πράγματα είναι πιο ομαλά. Αν από την αρχική κατάσταση έρθει κάποιος από τους χαρακτήρες `+`, `-` τότε θα μεταβούμε αμέσως στην τελική κατάσταση `addOperator`. Αν έρθει κάποιος από τους χαρακτήρες `*`, `/` τότε πηγαίνουμε στην κατάσταση `mulOperator`. Αν έρθει κάποιο από τα σύμβολα `(`, `)`, `{`, `}`, `[`, `]`, τότε θα μεταβούμε στην τελική κατάσταση `groupSymbol`, ενώ αν έρθει ένα από τα σύμβολα `,,`, `;`, `..`, τότε καταλήγουμε στην `delimiter`.

Ξεχωριστή κατηγορία αποτελεί το σύμβολο εκχώρησης, το οποίο απαιτεί μία μη τελική κατάσταση (την `asgn`) πριν φτάσει στην τελική `assignment`, αφού αποτελείται από δύο ακριβώς χαρακτήρες.

Παρόμοια περίπτωση είναι αυτή με τους λογικούς τελεστές. Εκτός από τον τελεστή `isoteta`, η εμφάνιση του οποίου μας οδηγεί αμέσως στην τελική κατάσταση `relOperator`, για τους υπόλοιπους λογικούς τελεστές πρέπει να είμαστε πιο προσεκτικοί. Αν αναγνωρίσουμε, για παράδειγμα, το σύμβολο `<`, αυτό δεν σημαίνει ότι μπορούμε να πάμε με ασφάλεια σε τελική κατάσταση, διότι είναι πιθανό να ακολουθεί το σύμβολο `=` ή το σύμβολο `>`, οπότε η λεκτική μονάδα που θα πρέπει να αναγνωρίστε θα είναι `<=` την πρώτη φορά και `>` τη δεύτερη.

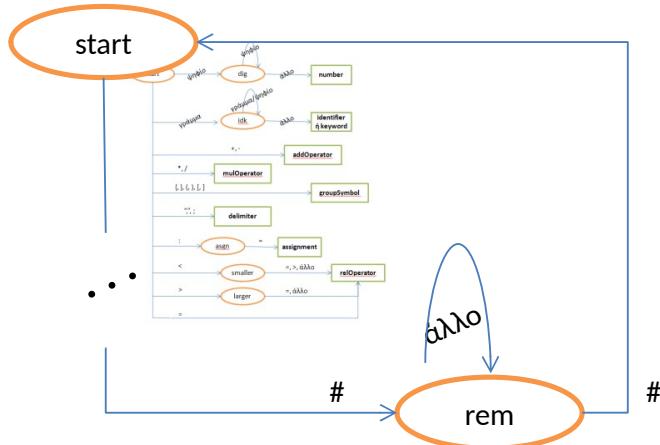
Μάλιστα, στην περίπτωση αυτή θα πρέπει να προσέξουμε και κάτι ακόμα. Ενώ στις περισσότερες περιπτώσεις, όποιο σύμβολο διαβάζουμε το ενσωματώνουμε σε κάποια λεκτική μονάδα, εδώ για να καταλάβουμε αν το σύμβολο που αναγνωρίσαμε ήταν το `<` και όχι το `<>` ή το `<=`, θα πρέπει να διαβάσουμε έναν χαρακτήρα ο οποίος ή είναι λευκός χαρακτήρας ή ανήκει στην επόμενη λεκτική μονάδα. Αν είναι λευκός χαρακτήρας, δεν δημιουργείται πρόβλημα. Αν όμως ανήκει στην επόμενη λεκτική μονάδα, πρέπει να είμαστε προσεκτικοί ώστε να μην τον χάσουμε.

Στο παράδειγμα `a>b<c`, για να είμαστε βέβαιοι ότι ο λογικός τελεστής είναι το `>` και όχι κάποιος άλλος, για παράδειγμα το σύμβολο `>=` πρέπει να διαβάσουμε και το `b`. Όταν όμως ξανακληθεί ο λεκτικός αναλυτής, η ανάγνωση της εισόδου πρέπει να αρχίσει από το `b`, το οποίο έχει ήδη διαβαστεί, και όχι από το `c`, το οποίο είναι το επόμενο σύμβολο στην είσοδο. Πρέπει, δηλαδή, όταν χρησιμοποιήσουμε το `b`, με κάποιον τρόπο να το επιστρέψουμε πάλι στη θέση του στην είσοδο ή να βρούμε μία εναλλακτική ισοδύναμη λύση.

Εκτός από τους λογικούς τελεστές που είδαμε παραπάνω, το ίδιο συμβαίνει στην αναγνώριση των κατηγοριών `identifier`, `keyword` και `number`, αφού και εκεί, για να αναγνωρίσουμε μία λεκτική μονάδα, είμαστε υποχρεωμένοι να κρυφοκοιτάζουμε σε χαρακτήρα που δεν ανήκει στη λεκτική μονάδα, έτσι ώστε να γνωρίζουμε ότι η υπό αναγνώριση λεκτική μονάδα ολοκληρώθηκε. Στο αυτόματο του σχήματος 4.2 μπορείτε να εντοπίσετε τις περιπτώσεις αυτές εκεί που εμφανίζεται η λέξη `άλλο`.

Το μόνο σημείο το οποίο δεν έχουμε συζητήσει καθόλου ακόμα είναι οι λευκοί χαρακτήρες. Αν βρισκόμαστε στην αρχική κατάσταση και στην είσοδο βρεθεί ένας λευκός χαρακτήρας, τότε παραμένουμε στην αρχική κατάσταση. Πρακτικά, ο λευκός χαρακτήρας αγνοείται και συνεχίζουμε την ανάγνωση της εισόδου. Αφού δεν έχουμε φτάσει σε τελική κατάσταση, ο λεκτικός αναλυτής δεν τερματίζεται και δεν επιστρέφει

αποτέλεσμα. Αυτό θα γίνει όταν αναγνωριστεί λεκτική μονάδα. Όταν ο λευκός χαρακτήρας που συναντήσαμε είναι η αλλαγή γραμμής, τότε πρέπει να ενημερωθεί ο μετρητής γραμμών, ο οποίος μας χρησιμεύει για να επιστέψουμε στον συντακτικό αναλυτή το πεδίο `line_number` της κλάσης `Token`.



Σχήμα 4.3: Διαχείριση σχολίων από τον λεκτικό αναλυτή.

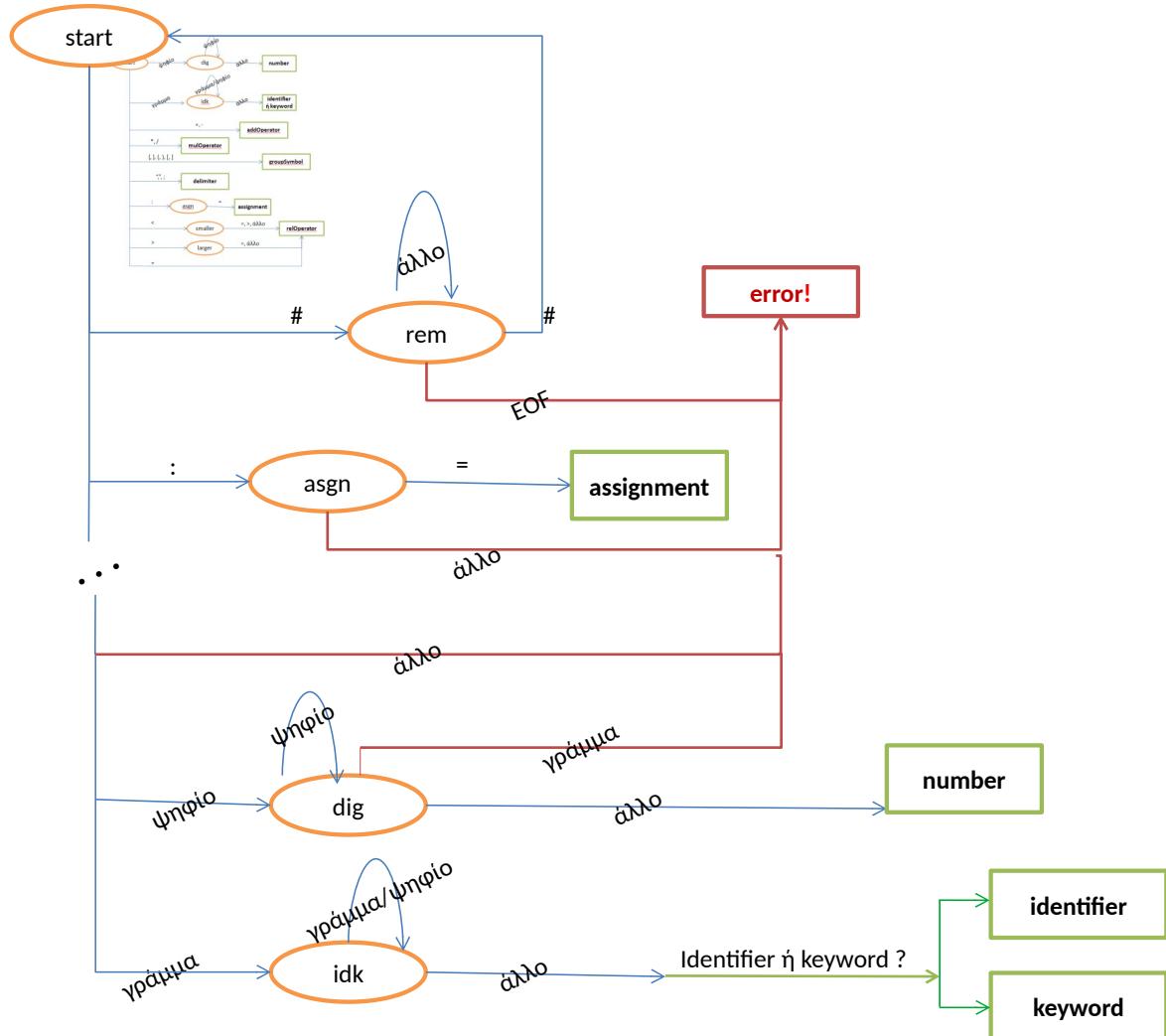
Πριν κλείσουμε την περιγραφή της λειτουργίας του λεκτικού αναλυτή, ας αφιερώσουμε και μία παράγραφο στα σχόλια. Ο σχολιασμός του κώδικα πρέπει να είναι αναπόσπαστο τμήμα της συγγραφής του. Τα σχόλια τα διαχειρίζεται αποκλειστικά ο λεκτικός αναλυτής, τα αναγνωρίζει και τα αφαιρεί από το πρόγραμμα. Ο συντακτικός αναλυτής δεν χρειάζεται καν να πληροφορηθεί την ύπαρξή τους. Στην `C-implement` τα σχόλια αρχίζουν και τελειώνουν με τον χαρακτήρα `#`. Όταν βρισκόμαστε στην αρχική κατάσταση και εμφανιστεί ο χαρακτήρας `#`, τότε μεταβαίνουμε στην κατάσταση `rem`. Εκεί παραμένουμε όσο στην είσοδο δεν εμφανίζεται πάλι ο χαρακτήρας `#`. Μόλις εμφανιστεί ο χαρακτήρας `#`, μεταβαίνουμε πάλι στην αρχική κατάσταση, όπου συνεχίζουμε με την αναγνώριση της επόμενης λεκτικής μονάδας. Αφού δεν έχουμε φτάσει σε τελική κατάσταση, ο λεκτικός αναλυτής δεν τερματίζεται και δεν επιστρέφει αποτέλεσμα. Πρέπει πάλι να προσέξουμε με την αλλαγή γραμμής, αφού όταν βρισκόμαστε μέσα στα σχόλια, παρόλο που πρακτικά δεν αναγνωρίζουμε λεκτικές μονάδες, πρέπει να ενημερώνουμε τον μετρητή γραμμών όταν συναντάμε αλλαγή γραμμής. Στο σχήμα 4.3 φαίνεται πώς θα ενσωματώσουμε τα σχόλια στο αυτόματο του σχήματος 4.2.

### 4.3 Διαχείριση σφαλμάτων

Στη φάση της λεκτικής ανάλυσης αναγνωρίζονται τα πρώτα από τα σφάλματα που μπορεί να ανακαλύψει ένας μεταγλωττιστής. Φυσικά, πρόκειται για σφάλματα τα οποία σχετίζονται με λεκτικές μονάδες. Σφάλματα που σχετίζονται με σύνταξη (π.χ. παράλειψη παρένθεσης) ή σημασιολογικά (π.χ. μία μεταβλητή δεν έχει δηλωθεί) θα εντοπιστούν σε φάσεις που θα ακολουθήσουν.

Τα σφάλματα αυτής της φάσης προκύπτουν από συνθήκες που μπορεί να εμφανιστούν κατά τη διάσχιση του αυτομάτου. Θα χρησιμοποιήσουμε το αυτόματο για την `C-implement` (σχήμα 4.2) για να εντοπίσουμε τέτοιες περιπτώσεις. Κάθε γλώσσα, φυσικά, έχει τις δικές της ιδιαιτερότητες, οπότε η διαχείριση σφαλμάτων θα πρέπει να προσαρμοστεί ανάλογα. Ας δούμε, όμως, ποια σφάλματα είναι δυνατόν να εντοπίσουμε με βάση το αυτόματο του σχήματος 4.2. Το εμπλουτισμένο αυτόματο που προκύπτει φαίνεται στο σχήμα 4.4, το οποίο και θα αναλύσουμε.

- Ενώ βρισκόμαστε στην κατάσταση `start` εμφανίζεται στην είσοδο ένας χαρακτήρας που δεν ταιριάζει σε καμία από τις επιλογές που φαίνεται να υπάρχουν για την κατάσταση αυτή. Ο μόνος λόγος που μπορεί να συμβεί αυτό είναι γιατί εμφανίστηκε χαρακτήρας που δεν ανήκει στη γλώσσα. Στην περίπτωση αυτή, όχι μόνο έχουμε εντοπίσει σφάλμα, αλλά ξέρουμε ακριβώς και ποιο είναι αυτό. Μπορούμε να εμφανίζουμε ένα αρκετά διαφωτιστικό μήνυμα, κάτι σαν: `Εμφανίστηκε ο χαρακτήρας xx`



Σχήμα 4.4: Διαχείριση σφαλμάτων από τον λεκτικό αναλυτή.

στη γραμμή  $\gamma$ , ο οποίος δεν ανήκει στη γλώσσα. Παρατηρήστε ότι όταν εμφανίζεται μη νόμιμος χαρακτήρας, αυτό θα συμβεί πάντοτε όταν είμαστε στην αρχική κατάσταση **start**. Δείτε το ακόλουθο παράδειγμα, στο οποίο ως είσοδος εμφανίζεται η συμβολοσειρά  $w!1e$ . Η πρώτη κλήση του λεκτικού αναλυτή θα εντοπίσει τη λεκτική μονάδα **w**, κρυφοκοιτάζοντας, αλλά όχι καταναλώνοντας τον χαρακτήρα **!**. Η δεύτερη κλήση του λεκτικού αναλυτή είναι αυτή που θα εντοπίσει το **!** και θα επιστρέψει το μήνυμα σφάλματος.

- Ενώ βρισκόμαστε στην κατάσταση **asgn** και αναμένουμε να έρθει ο χαρακτήρας **=**, εμφανίζεται κάποιος άλλος χαρακτήρας. Σε κάποια άλλη γλώσσα ίσως αυτό ήταν επιτρεπτό, οπότε θα έπρεπε να προσαρμοστούμε ανάλογα, όμως στην *C-imple* δεν είναι. Άρα εδώ εντοπίζουμε ένα ακόμα πιθανό σφάλμα, για το οποίο είμαστε πάλι σε θέση να εμφανίσουμε ένα διαφωτιστικό μήνυμα σφάλματος.
- Ενώ βρισκόμαστε στην κατάσταση **rem**, έχουν ανοίξει δηλαδή σχόλια, εμφανίζεται τέλος του αρχείου εισόδου. Έχουμε δηλαδή σχόλια τα οποία ανοίγουν, αλλά δεν κλείνουν ποτέ.
- Τέλος, στη λεκτική ανάλυση θα κατατάσσαμε και την αναγνώριση λεκτικών μονάδων που, ενώ ξεκινούν από ψηφίο, μέσα στη λεκτική μονάδα εμφανίζονται γράμματα. Ένα τέτοιο παράδειγμα είναι η συμβολοσειρά  $123abc$  η οποία δεν έχει κάποιο νόημα για μία γλώσσα όπως η *C-imple*. Αυτό είναι κάτι που θα μπορούσε να αναγνωριστεί και να οδηγήσει σε σφάλμα και κατά τη διάρκεια της συντακτικής ανάλυσης. Είναι φανερό, όμως, ότι ως σφάλμα είναι λεκτικό και πρέπει να ενταχθεί εδώ.

Μάλιστα, το μήνυμα το οποίο έχουμε τη δυνατότητα να εμφανίσουμε τώρα είναι πολύ περισσότερο διαφωτιστικό.

- Σύμφωνα με την περιγραφή της *C-imple*, υπάρχουν δύο περιορισμοί, οι οποίοι θα πρέπει να ελεγχθούν: α) εάν το μήκος ενός *identifier* είναι το πολύ 30 χαρακτήρες και εάν μία ακέραια αριθμητική σταθερά βρίσκεται μέσα στο επιτρεπτό εύρος τιμών. Έτσι, στις καταστάσεις *identifier*, *keyword* και *number* του σχήματος 4.2 θα γίνουν οι απαραίτητοι έλεγχοι. Στην περίπτωση που οι έλεγχοι είναι επιτυχημένοι, τότε θα έχουμε επιτυχή αναγνώριση και επιστροφή του αποτελέσματος στον συντακτικό αναλυτή. Εάν ο έλεγχος δεν είναι επιτυχημένος, τότε ο λεκτικός αναλυτής θα οδηγείται σε κατάσταση σφάλματος.

#### 4.4 Παράδειγμα εκτέλεσης

Το πρόγραμμα *factorial.ci*:

```
program factorial
{
    # declarations #
    declare x;
    declare i,fact;

    # main #
    input(x);
    fact:=1;
    i:=1;
    while (i<=x)
    {
        fact:=fact*i;
        i:=i+1;
    };
    print(fact);
}.
```

επιστρέφει τις ακόλουθες λεκτικές μονάδες:

```
program      family:"keyword",  line: 1
factorial   family:"id",       line: 1
{           family:"groupSymbol", line: 2
declare     family:"keyword",  line: 4
x          family:"id",       line: 4
;          family:"delimiter", line: 4
declare     family:"keyword",  line: 5
i          family:"id",       line: 5
,          family:"delimiter", line: 5
fact       family:"id",       line: 5
;          family:"delimiter", line: 5
input      family:"keyword",  line: 8
(           family:"groupSymbol", line: 8
x          family:"id",       line: 8
)           family:"groupSymbol", line: 8
;           family:"delimiter", line: 8
fact       family:"id",       line: 9
:=         family:"assignment", line: 9
```

```

1      family:"number",  line: 9
;
family:"delimiter",  line: 9
i      family:"id",  line: 10
:=     family:"assignment",  line: 10
1      family:"number",  line: 10
;
family:"delimiter",  line: 10
while   family:"keyword",  line: 11
(
family:"groupSymbol",  line: 11
i      family:"id",  line: 11
<=     family:"relOperator",  line: 11
x      family:"id",  line: 11
)
family:"groupSymbol",  line: 11
{
family:"groupSymbol",  line: 12
fact    family:"id",  line: 13
:=     family:"assignment",  line: 13
fact    family:"id",  line: 13
*
family:"mulOperator",  line: 13
i      family:"id",  line: 13
);
family:"delimiter",  line: 13
i      family:"id",  line: 14
:=     family:"assignment",  line: 14
i      family:"id",  line: 14
+
family:"addOperator",  line: 14
1      family:"number",  line: 14
;
family:"delimiter",  line: 14
}
family:"groupSymbol",  line: 15
;
family:"delimiter",  line: 15
print   family:"keyword",  line: 16
(
family:"groupSymbol",  line: 16
fact    family:"id",  line: 16
)
family:"groupSymbol",  line: 16
;
family:"delimiter",  line: 16
}
family:"groupSymbol",  line: 17
.
family:"delimiter",  line: 17

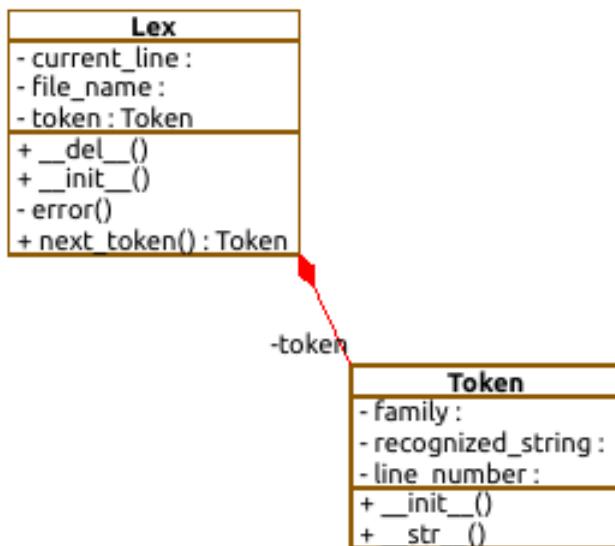
```

#### 4.5 Διάγραμμα κλάσεων λεκτικού αναλυτή

Θα δοκιμάσουμε να σχεδιάσουμε έναν λεκτικό αναλυτή με βάση την αντικειμενοστραφή φιλοσοφία. Ένα πιθανό διάγραμμα κλάσεων του λεκτικού αναλυτή θα μπορούσε να περιέχει δύο κλάσεις. Η πρώτη από αυτές θα περιγράφει την κλάση Token, η οποία θα δημιουργεί αντικείμενα που θα αναπαριστούν λεκτικές μονάδες. Τα πεδία της κλάσης Token είναι τα:

- `recognized_string`, το οποίο θα αποθηκεύει τη λεκτική μονάδα που αναγνωρίστηκε,
- `family`, το οποίο θα υποδηλώνει την οικογένεια στην οποία ανήκει η λεκτική μονάδα,
- και το `line_number`, που θα μεταφέρει στον συντακτικό αναλυτή τη γραμμή στην οποία αναγνωρίστηκε η λεκτική μονάδα.

Η δεύτερη κλάση στο διάγραμμα κλάσεων είναι αυτή που θα δημιουργήσει το αντικείμενο του λεκτικού αναλυτή. Θα την ονομάσουμε Lex και θα έχει τρία πεδία:



Σχήμα 4.5: Διάγραμμα κλάσεων του λεκτικού αναλυτή.

- Το πρώτο από τα πεδία αποθηκεύει το όνομα του αρχείου του αρχικού προγράμματος, όπως θα το δώσει ο προγραμματιστής στη γραμμή εντολών.
- Το δεύτερο πεδίο περιέχει τη γραμμή στην οποία βρισκόμαστε σε κάθε στιγμή της μετάφρασης.
- Το τρίτο πεδίο αποτελεί τη συσχέτιση με την κλάση **Token**, περιέχει τη λεκτική μονάδα που επιστρέφεται ως αποτέλεσμα στον συντακτικό αναλυτή.

Η κλάση **Lex** έχει δύο μεθόδους πέρα από τον κατασκευαστή και κάποιον πιθανό καταστροφέα:

- Η μέθοδος **next\_token()** είναι δημόσια μέθοδος μέσα από την οποία ο λεκτικός αναλυτής θα επιστρέψει την επόμενη λεκτική μονάδα που θα αναγνωρίσει, ένα αντικείμενο τύπου **Token** δηλαδή.
- Η δεύτερη μέθοδος είναι η **error()** και υλοποιεί τον διαχειριστή σφαλμάτων. Ο διαχειριστής σφαλμάτων θα κληθεί από τον ίδιο τον λεκτικό αναλυτή, κάθε φορά που ο λεκτικός αναλυτής εντοπίζει ένα από τα σφάλματα της αρμοδιότητάς του. Αφού η μέθοδος καλείται από τον ίδιο τον λεκτικό αναλυτή, τότε πρόκειται για μία ιδιωτική μέθοδο.

Το διάγραμμα κλάσεων του λεκτικού αναλυτή φαίνεται στο σχήμα 4.5. Η σχέση ανάμεσα στον λεκτικό αναλυτή και τις λεκτικές μονάδες είναι σχέση σύνθεσης. Ο λεκτικός αναλυτής δημιουργεί πολλά αντικείμενα τύπου **Token**, ένα από τα οποία είναι χρήσιμο ανά πάσα στιγμή.

Σε κάθε φάση ανάπτυξης που θα μελετάμε στη συνέχεια του βιβλίου, θα εμπλουτίζουμε το διάγραμμα αυτό, ώστε να φτάσουμε στο τέλος να έχουμε το συνολικό διάγραμμα του μεταφραστή.

## Βιβλιογραφία

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 0321486811.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Μεταγλωττιστές: Αρχές, τεχνικές και εργαλεία, 2η αμερικανική έκδοση*. Επιστημονική επιμέλεια έκδοσης: Π. Αλεφραγκής, Α. Βώρος, Ν. Βώρος, Κ. Μασσέλος. Εκδόσεις Νέων Τεχνολογιών, 2011. ISBN: 9789606759727.
- [3] Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.

- [4] Keith D. Cooper και Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2004. ISBN: 1558606998.
- [5] Keith D. Cooper και Linda Torczon. *Σχεδίαση και Κατασκευή Μεταγλωττιστών*. Ξενόγλωσσος τίτλος: Engineering a Compiler, Επιστημονική επιμέλεια έκδοσης: Γεώργιος Μανής, Νικόλαος Παπασπύρου και Αντώνιος Σαββίδης. Πανεπιστημιακές Εκδόσεις Κρήτης, 2018. ISBN: 9789605245191.
- [6] Κωνσταντίνος Λάζος, Παναγιώτης Κατσαρός και Ζαφείρης Καραϊσκος. *Μεταγλωττιστές Γλωσσών Προγραμματισμού: Θεωρία & Πράξη*. Ζυγός, 2004. ISBN: 9608772346.
- [7] Ζαφείρης Καραϊσκος. *Μεταγλωττιστές*. Εκδόσεις DaVinci, 2016. ISBN: 9789609732185.
- [8] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996. ISBN: 0201403536.
- [9] David F. Bacon, Susan L. Graham και Oliver J. Sharp. “Compiler Transformations for High-Performance Computing.” Στο: *ACM Computing Surveys* 26.4 (1994), σσ. 345–420.



## ΚΕΦΑΛΑΙΟ 5

---

### ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

---

#### Σύνοψη:

Η συντακτική ανάλυση ελέγχει την ορθότητα ενός προγράμματος βάσει της γραμματικής της γλώσσας και προσφέρει το περιβάλλον το οποίο θα καθοδηγήσει την παραγωγή του κώδικα στις επόμενες φάσεις της ανάπτυξης. Εκκινώντας από τη γραμματική, ο συντακτικός αναλυτής διαβάζει μία-μία τις λεκτικές μονάδες από τον λεκτικό αναλυτή, ελέγχοντας ότι η σειρά με την οποία αυτές εμφανίζονται, είναι συνεπής με τη δεδομένη γραμματική.

Στο κεφάλαιο αυτό θα δούμε τον τρόπο με τον οποίο αναπτύσσεται ένας συντακτικός αναλυτής.

Η γραμματική της *C-imple* είναι γραμμένη, έτσι ώστε να είναι κατάλληλη για την υλοποίηση ενός συντακτικού αναλυτή με τη μέθοδο της αναδρομικής κατάβασης. Θα δούμε τι είναι η μέθοδος της αναδρομικής κατάβασης, πότε μία γραμματική είναι κατάλληλη για υλοποίηση με τη μέθοδο της αναδρομικής κατάβασης και πώς από μία τέτοια γραμματική μπορούμε να κατασκευάσουμε με τρόπο μηχανιστικό, έναν συντακτικό αναλυτή. Θα χρησιμοποιήσουμε παραδείγματα στα οποία θα φαίνεται πώς από έναν κανόνα της γραμματικής της *C-imple* θα φτάσουμε σε κώδικα Python, ο οποίος θα αναγνωρίζει τις συμβολοσειρές που περιγράφει ένας κανόνας.

Στη φάση της συντακτικής ανάλυσης εντοπίζονται τα περισσότερα από τα σφάλματα που μπορεί να εντοπίσει ένας μεταγλωττιστής. Έτσι, η διαχείριση των σφαλμάτων θα αποτελέσει σημαντικό τμήμα του κεφαλαίου.

#### Προαπαιτούμενη γνώση:

- θεωρία αυτομάτων
  - θεωρία τυπικών γλωσσών
  - κεφάλαιο 1
  - κεφάλαιο 2
  - κεφάλαιο 3
  - κεφάλαιο 4
-

Τη φάση της λεκτικής ανάλυσης ακολουθεί η φάση της συντακτικής ανάλυσης. Κατά τη συντακτική ανάλυση ελέγχεται εάν η ακολουθία των λεκτικών μονάδων που σχηματίζεται από τον λεκτικό αναλυτή, αποτελεί μία νόμιμη ακολουθία, με βάση τη γραμματική της γλώσσας. Όποια ακολουθία δεν αναγνωρίζεται από τη γραμματική, αποτελεί μη νόμιμο κώδικα και οδηγεί στον εντοπισμό συντακτικού σφάλματος.

Έτσι, ο συντακτικός αναλυτής παίρνει ως είσοδο μία ακολουθία από λεκτικές μονάδες. Στην έξοδο μπορούμε να θεωρήσουμε ότι δίνει το συντακτικό δέντρο που αντιστοιχεί στο αρχικό πρόγραμμα. Στην πραγματικότητα δεν επιστρέφει κανένα δέντρο, αλλά διαπερνά το αρχικό πρόγραμμα και, πέρα από την αναγνώριση σφαλμάτων, δίνει το περιβάλλον πάνω στο οποίο θα βασιστεί η επόμενη φάση, η παραγωγή ενδιάμεσου κώδικα, αλλά και όλη η υπόλοιπη μεταγλώττιση. Η λειτουργία της συντακτικής ανάλυσης εικονίζεται στο σχήμα 5.1.



Σχήμα 5.1: Ο συντακτικός αναλυτής στη διαδικασία της μεταγλώττισης.

Η γραμματική η οποία χρησιμοποιείται είναι μία γραμματική χωρίς συμφραζόμενα. Μία γραμματική με συμφραζόμενα θα μπορούσε να περιγράψει ανάγκες που μία γραμματική χωρίς συμφραζόμενα αδυνατεί να κάνει. Όπως για παράδειγμα, να ξεχωρίσει αν μία μεταβλητή έχει δηλωθεί ή όχι. Όμως μία γραμματική χωρίς συμφραζόμενα έχει αυξημένη πολυπλοκότητα στη συγγραφή και την υλοποίησή της, αλλά και απαιτεί υπερβολικά μεγάλο χρόνο για να ολοκληρώσει τη συντακτική ανάλυση. Έτσι, περιορίζόμαστε να ορίσουμε τη γλώσσα μας με μία γραμματική χωρίς συμφραζόμενα και να κάνουμε συντακτική ανάλυση με αυτήν. Όποια πληροφορία χρειάζεται γραμματική με συμφραζόμενα για να περιγραφεί, την αποκόπτουμε από τη συντακτική ανάλυση και εφαρμόζουμε ευρετικές τεχνικές για να εξασφαλίσουμε τις απαιτήσεις μας, σε μεταγενέστερο στάδιο ανάπτυξης, κατά τη διάρκεια της σημασιολογικής ανάλυσης.

Για περισσότερη πληροφορία γύρω από την υλοποίηση ενός συντακτικού αναλυτή μπορείτε να ανατρέξετε στα: [1, 2, κεφ.4.4] και [3, κεφ.4].

## 5.1 Ενεργοποίηση κανόνων με πρόβλεψη

Η γραμματική της *C-imple* είναι μία γραμματική χωρίς συμφραζόμενα, η οποία όμως έχει και ένα επιπλέον χαρακτηριστικό: είναι κατάλληλη για υλοποίηση με την τεχνική της προβλέποντας αναδρομικής κατάβασης. Έτσι, κατά τη συντακτική ανάλυση, όταν ένας κανόνας έχει την εναλλακτική να ενεργοποιήσει περισσότερους από έναν κανόνες ή με άλλα λόγια βρίσκεται σε δίλημμα ποιον από τους διαθέσιμους κανόνες να ενεργοποιήσει, τότε η επιλογή του θα καθοριστεί από την επόμενη λεκτική μονάδα που υπάρχει διαθέσιμη στην είσοδο. Έτσι, στους κανόνες της εντολής απόφασης:

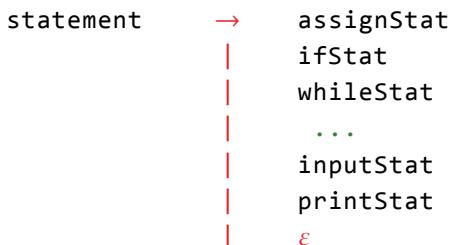
```

# if statement
ifStat   →   if ( condition )
            statements(1)
            elsepart
# else part is optional
elsepart →   else
            statements(2)
            |
            ε
  
```

και αφού έχει αποτιμηθεί η συνθήκη του `if` ως ψευδής και τα `statements(1)` δεν έχουν εκτελεστεί, η γραμματική βρίσκεται στο δίλημμα αν θα ενεργοποιήσει τον κανόνα `elsepart` ή αν θα επιλέξει τον εναλλακτικό

δρόμο της κενής συμβολοσειράς. Αν η λεκτική μονάδα `else` είναι πράγματι η επόμενη λεκτική μονάδα προς αναγνώριση στην είσοδο, τότε ο συντακτικός αναλυτής θα επιλέξει να ενεργοποιήσει τον κανόνα `elsepart`. Σε αντίθετη περίπτωση θα θεωρήσει ότι αναγνώρισε την κενή συμβολοσειρά.

Ένα άλλο σημείο της γραμματικής στο οποίο φαίνεται η χρησιμότητα αυτής της ιδιότητας της γραμματικής είναι ο κανόνας `statement`.



Η ενεργοποίηση του `statement` θέτει τον συντακτικό αναλυτή μπροστά από τις επιλογές αν θα ακολουθηθεί ο δρόμος της ενεργοποίησης του κανόνα `assignStat` ή του κανόνα `ifStat` ή του κανόνα `whileStat` και ούτω καθεξής. Το ποιος από τους κανόνες θα ενεργοποιηθεί θα καθοριστεί πάλι από την επόμενη λεκτική μονάδα που εμφανίζεται στην είσοδο. Αν, για παράδειγμα, η επόμενη λεκτική μονάδα στην είσοδο είναι το `if`, τότε θα ενεργοποιηθεί ο κανόνας `ifStat`.

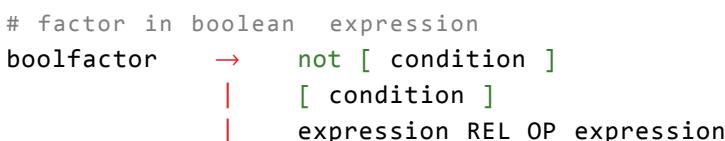
Σε κάποιες περιπτώσεις, το ποια λεκτική μονάδα πρέπει να αναζητηθεί για να ακολουθηθεί ο καταλληλότερος δρόμος δεν είναι πάντοτε άμεσα εμφανές. Αν και θα μπορούσαμε να γράψουμε τη γραμματική μας με κατάλληλο τρόπο, ώστε αυτό να είναι τελείως φανερό, επιλέξαμε να συντάξουμε τη γραμματική της *C-implement* ώστε να είναι πιο ευανάγνωστη, ευκολότερο να υλοποιηθεί και όχι να είναι προσαρμοσμένη στο να αναγνωρίζεται εύκολα η επιθυμητή λεκτική μονάδα.

Ας δούμε ένα παράδειγμα και ας θυμηθούμε τον κανόνα `term`:



Στον κανόνα `term`, αρχικά αναγνωρίζεται ένα `factor`. Στη συνέχεια, το άστρο του Kleene επιτρέπει την ύπαρξη ενός ή περισσοτέρων ακόμα `factor` χωρισμένων από τα σύμβολα `*` ή `/`. Από τον κανόνα `term` αυτό δεν είναι άμεσα φανερό. Για να το δούμε θα πρέπει να κοιτάξουμε μέσα στον λεκτικό κανόνα `MUL_OP`. Έχοντας αναγνωρίσει τον πρώτο `factor`, η επιλογή αν θα αναζητήσουμε δεύτερο `factor` ή όχι θα καθοριστεί από το αν η επόμενη προς αναγνώριση λεκτική μονάδα στην είσοδο είναι μία από τις `*` ή `/`. Όμοια, η έξοδος από τον βρόχο που δημιουργεί το άστρο του Kleene καθορίζεται επίσης από την επόμενη στην είσοδο λεκτική μονάδα, αφού μετά από την αναγνώριση ενός `factor`, αν στην είσοδο έχουμε ένα εκ των `*` και `/` θα παραμείνουμε μέσα στο βρόχο και θα αναζητήσουμε ακόμα έναν `factor`.

Εδώ είναι το καταλλήλοτερο σημείο για να σχολιάσουμε μία ασυνήθιστη επιλογή που κάναμε κατά τον ορισμό της γλώσσας. Ας θυμηθούμε τον κανόνα `boolfactor`:



Ενώ το συνηθέστερο στις γλώσσες προγραμματισμού είναι να χρησιμοποιούνται τα ίδια σύμβολα για τον καθορισμό των προτεραιοτήτων πράξεων στις λογικές και στις αριθμητικές παραστάσεις, και πιο συγκεκριμένα τα σύμβολα παρενθέσεων ( και ), στην περιγραφή της γλώσσας *C-implement* επιλέξαμε να χρησιμοποιήσουμε τις αγκύλες, δηλαδή τα σύμβολα [ και ], για τις λογικές παραστάσεις και τις παρενθέσεις για τις αριθμητικές. Ο λόγος που οδηγηθήκαμε στην επιλογή αυτή σχετίζεται με τη δυνατότητα πρόβλεψης ενεργοποίησης κανόνων με βάση την επόμενη διαθέσιμη προς αναγνώριση λεκτική μονάδα.

Ας δούμε ποιο θα ήταν το πρόβλημα, αν επιλέγαμε τα σύμβολα που ορίζουν την προτεραιότητα των πράξεων στις λογικές και στις αριθμητικές παραστάσεις να είναι τα ίδια, και δη οι παρενθέσεις. Ο κανόνας boolfactor θα ήταν ο ακόλουθος:

```
# factor in boolean expression
boolfactor → not ( condition )
          | ( condition )
          | expression REL_OP expression
```

Παρατηρήστε ότι όταν βρισκόμαστε μέσα στον boolfactor και, ενώ έχουμε να επιλέξουμε ανάμεσα σε τρεις δρόμους, αν η επόμενη προς αναγνώριση λεκτική μονάδα στην είσοδο είναι το άνοιγμα παρένθεσης, τότε δεν μπορούμε να αποφασίσουμε αν θα επιλέξουμε να ακολουθήσουμε το (condition) ή το expression REL\_OP expression, αφού και οι δυο επιλογές μπορούν να δημιουργήσουν συμβολοσειρές που ξεκινούν με το μη τερματικό σύμβολο “(”. Αντίθετα, αν επιλέξουμε για τον καθορισμό της προτεραιότητας στις λογικές παραστάσεις να χρησιμοποιήσουμε τις αγκύλες, τότε το πρόβλημα λύνεται, αφού, όταν η επόμενη λεκτική μονάδα στην είσοδο είναι το άνοιγμα αγκύλης, τότε θα ακολουθήσουμε τον δρόμο [condition], ενώ όταν είναι το άνοιγμα της παρένθεσης, θα αναζητήσουμε ένα expression REL\_OP expression.

Φυσικά, με τον κατάλληλο μετασχηματισμό της γραμματικής της *C-implement* μπορούμε να καταλήξουμε σε μία γραμματική η οποία χρησιμοποιεί τις παρενθέσεις ως σύμβολα ομαδοποίησης και προτεραιότητας, τόσο για τις λογικές, όσο και για τις αριθμητικές παραστάσεις. Προτιμήσαμε όμως να κρατήσουμε τη γραμματική σε μια πιο ευανάγνωστη μορφή και να αδράξουμε την ευκαιρία να σχολιάσουμε την επιλογή αυτή, δίνοντας ένα διαφωτιστικό παράδειγμα της ιδιότητας της γραμματικής που την κάνει κατάλληλη για υλοποίησή με την τεχνική της αναδρομικής κατάβασης.

## 5.2 Υλοποίηση συναρτήσεων αναδρομικής κατάβασης

Η τεχνική, σύμφωνα με την οποία θα μεταβούμε από την περιγραφή με τη μορφή γραμματικής στον κώδικα λέγεται τεχνική της υλοποίησης με τη μέθοδο της αναδρομικής κατάβασης. Πρόκειται για έναν εύκολο τρόπο να μετατρέψουμε τους κανόνες της γραμματικής σε κώδικα και, κυρίως, πρόκειται για έναν τρόπο αυτοματοποιημένο, μηχανιστικό. Η μετατροπή γίνεται σύμφωνα με τον παρακάτω αλγόριθμο, ο οποίος βρίσκει εφαρμογή σε γραμματικές χωρίς συμφραζόμενα, κατάλληλες για υλοποίηση με αναδρομική κατάβαση:

- Για κάθε κανόνα της γραμματικής υλοποιούμε μία συνάρτηση. Δίνουμε στη συνάρτηση το ίδιο όνομα με τον κανόνα ή τουλάχιστον έναν όνομα που να αντιστοιχίζεται μονοσήμαντα στον κανόνα. Στην παρούσα φάση η συνάρτηση αυτή δεν χρειάζεται να παίρνει κάτι ως όρισμα ή να επιστρέφει κάποιο αποτέλεσμα στη συνάρτηση που την κάλεσε.
- Γράφουμε τον κώδικα που αντιστοιχεί στο δεξιό μέλος του κανόνα.
  - Για κάθε μη τερματικό σύμβολο, καλούμε την αντίστοιχη συνάρτηση που έχουμε υλοποιήσει.
  - Για κάθε τερματικό σύμβολο ελέγχουμε ότι πράγματι το τερματικό σύμβολο αυτό συμπίπτει με την επόμενη λεκτική μονάδα στην είσοδο.
- \* Αν πράγματι το τερματικό σύμβολο που συναντάμε στη γραμματική συμπίπτει με την επόμενη λεκτική μονάδα στην είσοδο, τότε καταναλώνουμε τη λεκτική μονάδα, θεωρούμε την αναγνώριση μέχρι το σημείο αυτό επιτυχής και προχωρούμε στην αναγνώριση της επόμενης λεκτικής μονάδας.
- \* Αν το τερματικό σύμβολο που συναντάμε στη γραμματική δεν συμπίπτει με την επόμενη λεκτική μονάδα στην είσοδο τότε:

- Αν μία από τις εναλλακτικές που δίνει η γραμματική είναι η κενή συμβολοσειρά, ακολουθούμε αυτήν την επιλογή, ελπίζοντας το σύμβολο που ζητούμε να αναγνωριστεί από τον κανόνα που θα ακολουθήσει.
- Αν η γραμματική δεν δίνει ως επιλογή το κενό, τότε έχουμε φτάσει σε κατάσταση σφάλματος, εμφανίζεται το κατάλληλο μήνυμα λάθους και τερματίζεται η μετάφραση.

Ίσως τα παραδείγματα διευκολύνουν να κατανοήσουμε καλύτερα την παραπάνω περιγραφή και θα δούμε κάποια στη συνέχεια. Ας διευκρινίσουμε πρώτα τη λέξη καταναλώνω που χρησιμοποιήσαμε παραπάνω. Να θυμίσουμε ότι κάθε κλήση του λεκτικού αναλυτή επιστρέφει στον συντακτικό αναλυτή ένα αντικείμενο της κλάσης Token με τα πεδία recognized\_string και family συμπληρωμένα (όπως και το πεδίο αριθμού γραμμών line\_number, αλλά δεν το χρειαζόμαστε τη στιγμή αυτή). Ας θεωρήσουμε μία μέθοδο get\_token, μέσα στην οποία ο συντακτικός αναλυτής καλεί τον λεκτικό αναλυτή και πάίρνει το token με την επόμενη λεκτική μονάδα.

Με την εκκίνηση του συντακτικού αναλυτή, και πριν κληθεί ο κώδικας του αρχικού κανόνα (του program στην *C-imple*), καλείται η get\_token για να γίνει διαθέσιμη η πρώτη λεκτική μονάδα. Μόλις κάποιος κανόνας αναγνωρίσει μία λεκτική μονάδα, τότε η λεκτική μονάδα αυτή καταναλώνεται, καλείται δηλαδή μέσω της get\_token ο λεκτικός αναλυτής ώστε το αντικείμενο token να αποκτήσει την επόμενη λεκτική μονάδα στην είσοδο.

Το σχέδιο κώδικα ενός συντακτικού αναλυτή μπορεί να είναι κάπως έτσι:

```
def syntax_analyzer():
    global token
    token = self.get_token()
    self.program()
    print('compilation successfully completed')
```

Ας δούμε κάποια παραδείγματα κατασκευής συναρτήσεων βασισμένοι στους κανόνες της γραμματικής. Θα ξεκινήσουμε με την program:

```
# "program" is the starting symbol
# followed by its name and a block
# Every program ends with a fullstop
program → program ID
    block
    .
```

Σύμφωνα με τον αλγόριθμο που παρουσιάστηκε παραπάνω, η μέθοδος θα ονομαστεί program(). Με την εκκίνηση κάθε μεθόδου η επόμενη λεκτική μονάδα προς αναγνώριση είναι ήδη διαθέσιμη. Η πρώτη ενέργεια που πρέπει να γίνει από τον συντακτικό αναλυτή μέσα στη μέθοδο program() είναι να ελέγξει αν το token.recognized\_string αντιστοιχεί στο program. Αν πράγματι το token.recognized\_string αντιστοιχεί στο program, τότε το καταναλώνουμε, διαβάζουμε δηλαδή την επόμενη λεκτική μονάδα που ακολουθεί στην είσοδο, καλώντας την get\_token(), και συνεχίζουμε την αναγνώριση αναζητώντας το ID, πάντα σύμφωνα με τη γραμματική.

Αν αναγνωρίσουμε επιτυχώς και το ID, τότε το καταναλώνουμε και αυτό, διαβάζουμε την επόμενη λεκτική μονάδα και καλούμε τη μέθοδο block(), αφού στη γραμματική, μετά το ID συναντούμε το μη τερματικό σύμβολο block. Μέσα στην block(), θα γίνει επιτυχής αναγνώριση μιας συμβολοσειράς που μπορεί να αναγνωρίσει η block() ή η μεταγλώττιση θα οδηγηθεί σε σφάλμα και θα διακοπεί η εκτέλεση. Αν συμβεί το δεύτερο, είναι ευθύνη του προγραμματιστή να διορθώσει το λάθος και να επανεκκινήσει τη μεταγλώττιση. Αν συμβεί το πρώτο, είναι ευθύνη της block() να καλέσει σωστά την get\_token(), ώστε να έχουμε ενημερωμένες τις token.recognized\_string και token.family, έτσι ώστε η program() να μπορεί να συνεχίσει την εκτέλεσή της. Η ενέργεια που έχει απομείνει στην program() είναι να αναγνωρίσει ότι στο τέλος του προγράμματος υπάρχει πράγματι η τελεία που απαιτεί ο ορισμός της γλώσσας, άρα αναμένει να βρει στο token.recognized\_string το τερματικό σύμβολο της τελείας.

Σε τρία σημεία του κανόνα `program` μπορούμε να αναγνωρίσουμε κατάσταση σφάλματος:

- Όταν ζητάμε να αναγνωρίσουμε τη δεσμευμένη λέξη `program`, αλλά κάτι άλλο εμφανίζεται αντ' αυτής.

Εδώ μπορούμε να εμφανίζουμε ένα αρκετά περιγραφικό μήνυμα: *keyword "program" expected in line 1. All programs should start with the keyword "program". Instead, the word '...' appeared.*

- Όταν μετά το `program` δεν ακολουθήσει ID.

Εδώ μπορούμε να έχουμε το εξής, επίσης αρκετά περιγραφικό μήνυμα: *The name of the program expected after the keyword "program" in line 1. The illegal program name ... appeared.*

- Όταν το τελευταίο σύμβολο του προγράμματος δεν είναι η τελεία.

Εδώ μπορούμε να έχουμε ένα μήνυμα: *Every program should end with a fullstop, fullstop at the end is missing.*

- Όταν μετά την τελεία ακολουθεί κάτι άλλο, εκτός από το τέλος του αρχείου.

Το μήνυμα *No characters are allowed after the fullstop indicating the end of the program.* μπορεί να υποδείξει με σαφήνεια το σφάλμα που προέκυψε.

Η επιλογή του κατάλληλου μηνύματος δεν είναι πάντοτε εύκολη. Πολλές γλώσσες προτιμούν να εμφανίζουν ένα γενικό μήνυμα τύπου *syntax error*, αλλά είναι βέβαιο ότι μπορεί να γίνει κάτι καλύτερο από αυτό σε πολλές περιπτώσεις. Περιγραφικά μηνύματα τύπου το λάθος μπορεί να οφείλεται στο ... ή στο ... είναι περισσότερο χρήσιμα στον προγραμματιστή από το *syntax error*.

Ο κώδικας σε Python που αντιστοιχεί στην `program()` ακολουθεί:

```
def program():
    global token
    if token.recognized_string == 'program':
        token = self.get_token()
        if token.family == 'id':
            token = self.get_token()
            self.block()
            if token.recognized_string == '.':
                token = self.get_token()
                if token.recognized_string == 'eof':
                    token = self.get_token()
                else:
                    self.error(...)
            else:
                self.error(...)
        else:
            self.error(...)
    else:
        self.error(...)
```

Στον παραπάνω κώδικα, σε όλες τις περιπτώσεις που γίνεται έλεγχος αν η λεκτική μονάδα που επιστρέφει ο λεκτικός αναλυτής αντιστοιχεί στη λεκτική μονάδα που αναμένεται από τη γραμματική, χρησιμοποιούμε το `token.recognized_string`. Εξαίρεση αποτελεί ο έλεγχος για το όνομα του προγράμματος. Εκεί το `token.recognized_string` περιέχει την ονομασία που δόθηκε στο συγκεκριμένο πρόγραμμα από τον προγραμματιστή, ενώ η γραμματική γνωρίζει μόνο ότι εκεί αναμένεται ένα ID. Για τον λόγο αυτόν χρησιμοποιούμε το `token.family` για τον έλεγχο, το οποίο περιέχει αυτή ακριβώς την πληροφορία. Το `token.recognized_string` μπορεί να χρησιμοποιηθεί στην εμφάνιση του μηνύματος σφάλματος, ώστε να διευκολύνει τον προγραμματιστή να εντοπίσει το σφάλμα.

Στον κανόνα του if το ενδιαφέρον σημείο είναι το προαιρετικό else, κάτι που εκφράζεται μέσα στον κανόνα elsepart. Ο κανόνας if αναγνωρίζει με τη σειρά τη λεκτική μονάδα if, το άνοιγμα της παρένθεσης, καλεί την condition() για να αναγνωριστεί η συνθήκη, αναγνωρίζει το κλείσιμο της παρένθεσης, καλεί την statements για να μεταφραστούν οι εντολές που θέλουμε να εκτελεστούν όταν η συνθήκη ισχύει και τέλος καλεί το elsepart.

```
# if-else statement
ifStat → if ( condition )
          statements
          elsepart
elsepart → else
          statements
          | ε
```

To elsepart καλείται πάντοτε, σε κάθε μετάφραση δομής if. Μέσα στην elsepart και ανάλογα με το αν η επόμενη λεκτική μονάδα που έχει αναγνωστεί στην είσοδο είναι το else, ακολουθεί την εναλλακτική να αναγνωρίσει το κενό ή να προχωρήσει να αναγνωρίσει το else και στη συνέχεια, φυσικά, το statements.

Τα σφάλματα που μπορούν να αναγνωριστούν από αυτόν τον κανόνα είναι η έλλειψη ανοίγματος παρένθεσης μετά τη λεκτική μονάδα if, η έλλειψη κλεισίματος παρένθεσης μετά την κλήση της condition() και τίποτε άλλο. Ο κανόνας elsepart δεν αναγνωρίζει κάποιο σφάλμα, αφού υπάρχει η εναλλακτική επιλογή της αναγνώρισης της κενής συμβολοσειράς.

Να σημειώσουμε κάτι ακόμα. Ίσως προσέξετε ότι στα πιθανά σφάλματα δεν αναφέρθηκε η μη αναγνώριση της λεκτικής μονάδας if, παρότι η γραμματική το ζητάει ως πρώτη λεκτική μονάδα του κανόνα. Ο λόγος που έγινε αυτό, είναι ότι από τον κανόνα statement (από εκεί ενεργοποιείται ο κανόνας if) θα οδηγηθούμε στον κανόνα if μονάχα αν η επόμενη λεκτική μονάδα στην είσοδο είναι το if. Έτσι, δεν υπάρχει εδώ λόγος να κάνουμε κάποιον έλεγχο, αφού έχουμε σίγουρο το θετικό αποτέλεσμα. Σκεφτείτε και αλλιώς. Έχετε δει ποτέ κάποιον μεταγλωττιστή να εμφανίζει διαγνωστικό μήνυμα if expected; Τι νόημα θα είχε αυτό;

Η αναγνώριση, λοιπόν, του if φαίνεται να μοιράζεται ανάμεσα στην statement και την ifStat. Για την ακρίβεια δεν συμβαίνει κάτι τέτοιο. Η αναγνώριση ανήκει στην ifStat. Εκεί θα γίνει και η κατανάλωση του if. Η statement κρυφοκοιτάζει την ύπαρξη του if και προχωρεί στην επιλογή της. Δεύτερος έλεγχος για το if είναι περιττός και παραλείπεται.

Ο κώδικας Python που υλοποιεί τα παραπάνω, ακολουθεί:

```
def ifStat():
    global token
    token = self.get_token() #consume if
    if token.recognized_string == '(':
        token = self.get_token()
        self.condition()
        if token.recognized_string == ')':
            token = self.get_token()
            self.statements()
            self.elsepart()
    else:
        self.error(...)
else:
    self.error(...)

def elsepart():
    if token.recognized_string == 'else':
        token = self.get_token()
        self.statements()
```

Τέλος, ας δούμε την υλοποίηση ενός ακόμα κανόνα. Εδώ το ενδιαφέρον σημείο είναι η εμφάνιση του άστρου του Kleene μέσα σε αυτόν. Επιλέγουμε τον κανόνα `boolterm`, τη σύνταξη του οποίου θυμίζουμε παρακάτω.

```
# term in boolean expression
boolterm → boolfactor(1)
          ( and boolfactor(2) )*
```

Με την έναρξη του κανόνα καλείται η `boolfactor` προκειμένου να αναγνωριστεί ο πρώτος λογικός παράγοντας. Στη συνέχεια αναμένουμε κανέναν, έναν, ή περισσότερους λογικούς παράγοντες χωρισμένους με τη λεκτική μονάδα `and`. Έτσι, αφού αναγνωρίσουμε το `boolfactor(1)`, ελέγχουμε αν στην είσοδο ακολουθεί προς αναγνώριση το `and`. Αν όχι, τερματίζεται ο κανόνας. Αν ναι, ζητάμε να αναγνωρίσουμε το `boolfactor(2)`. Αφού το επιτύχουμε, ελέγχουμε αν στην είσοδο ακολουθεί προς αναγνώριση και άλλο `and`. Αν όχι, πάλι, τερματίζεται ο κανόνας. Αν ναι, αναζητούμε και άλλο `boolfactor(2)`. Αυτό συνεχίζεται μέχρι να μην εμφανιστεί άλλο `and`. Η όλη διαδικασία μας θυμίζει προγραμματιστικά τη δομή `while` και αυτή είναι που θα επιστρατεύσουμε εδώ.

Ο κώδικας σε Python που υλοποιεί τον `boolterm` ακολουθεί:

```
def boolterm():
    global token
    self.boolfactor()
    while token.recognized_string == 'and':
        token = self.get_token()
        self.boolfactor()
```

Ο κανόνας δεν αναγνωρίζει σφάλματα. Αν κάπου υπάρχουν σφάλματα, αυτά θα αναγνωριστούν μέσα από τις δύο εμφανίσεις του `boolfactor`.

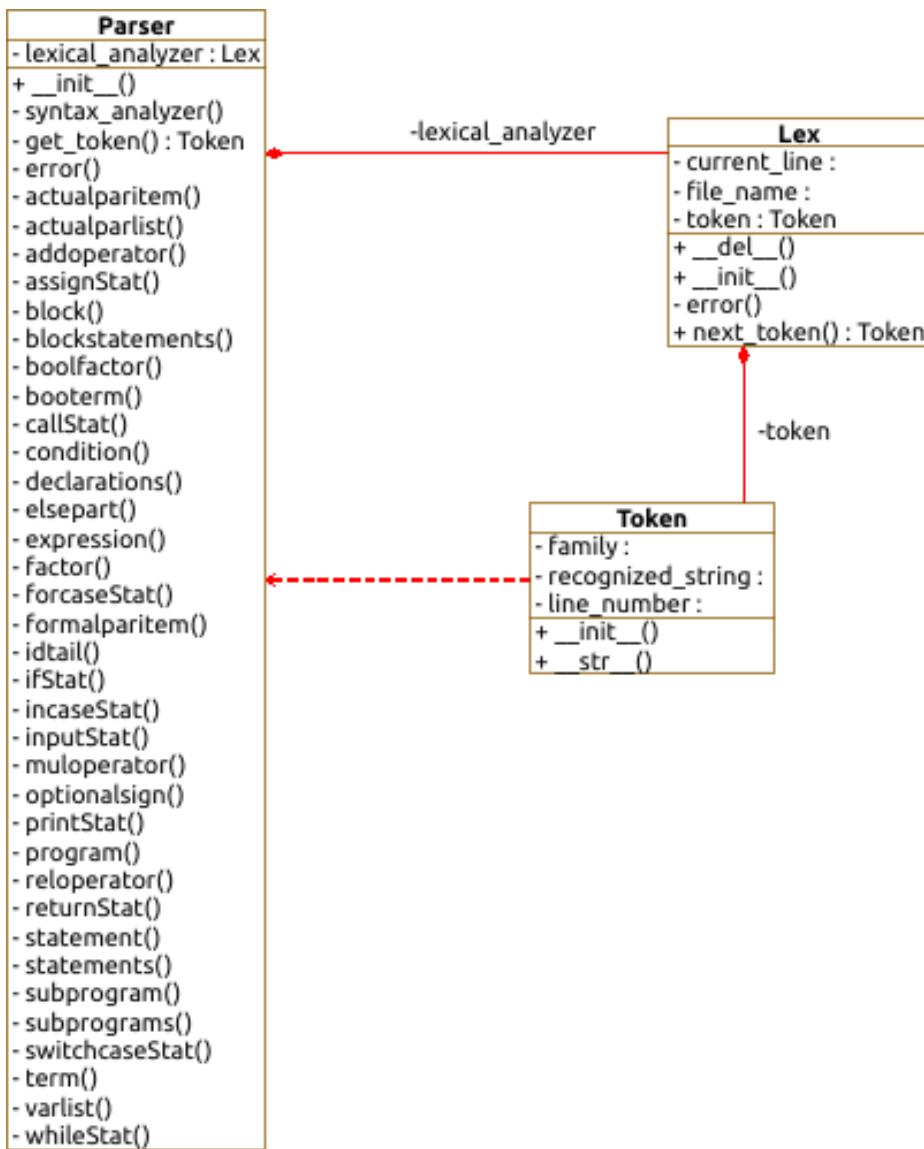
### 5.3 Ο συντακτικός αναλυτής στο διάγραμμα κλάσεων

Ο συντακτικός αναλυτής θα υλοποιηθεί ως μία κλάση η οποία σχετίζεται με τον λεκτικό αναλυτή και την κλάση `Token`. Ο λεκτικός αναλυτής δημιουργείται και χρησιμοποιείται από τον συντακτικό αναλυτή, οπότε μία σχέση ανάμεσά τους θα μπορούσε να χαρακτηριστεί ως συναρμολόγηση, ενώ άλλες σχεδιαστικές επιλογές μπορεί να είναι επίσης σωστές. Ο λεκτικός αναλυτής επιστρέφει αντικείμενα της κλάσης `Token` στον συντακτικό αναλυτή, οπότε διαγιγνώσκεται εδώ μία σχέση εξάρτησης.

Σύμφωνα με τη μέθοδο ανάπτυξης που βασίζεται στην αναδρομική κατάβαση, κάθε κανόνας της γραμματικής θα αποτελέσει και μία μέθοδο της κλάσης. Οι μέθοδοι αυτές δεν έχουν κάποιο λόγο να μην είναι ιδιωτικές, το αντίθετο μάλιστα, είναι καλά παραδείγματα λειτουργικότητας που κελυφοποιείται μέσα στην κλάση και ο εξωτερικός κόσμος δεν ενδιαφέρεται για τον τρόπο υλοποίησής τους.

Εκτός από τις μεθόδους που υλοποιούν τους κανόνες της γραμματικής, ο συντακτικός αναλυτής περιέχει τη μέθοδο `get_token` που αντλεί από τον λεκτικό αναλυτή την επόμενη λεκτική μονάδα, μία `error` για τα συντακτικά σφάλματα και την `syntax_analyzer` που υλοποιεί την αναδρομική κατάβαση.

Στο πεδίο `lexical_analyzer` αποθηκεύεται το αντικείμενο που υλοποιεί τον λεκτικό αναλυτή.



Σχήμα 5.2: Διάγραμμα κλάσεων λεκτικού-συντακτικού αναλυτή.

## Βιβλιογραφία

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison Wesley, 2006. ISBN: 0321486811.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Μεταγλωττιστές: Αρχές, τεχνικές και εργαλεία*, 2η αμερικανική έκδοση. Επιστημονική επιμέλεια έκδοσης: Π. Αλεφαραγκής, Α. Βώρος, Ν. Βώρος, Κ. Μασσέλος. Εκδόσεις Νέων Τεχνολογιών, 2011. ISBN: 9789606759727.
- [3] Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.



## ΚΕΦΑΛΑΙΟ 6

---

### ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ

---

#### Σύνοψη:

Το πρόγραμμα σε τελική γλώσσα δεν παράγεται απευθείας από το αρχικό πρόγραμμα. Μεσολαβεί η μετατροπή του αρχικού προγράμματος σε μία ενδιάμεση αναπαράσταση, ένα βήμα που διευκολύνει τη διαδικασία, δίνει τη δυνατότητα βελτιστοποιητικών μετασχηματισμών και επιτρέπει την απεξάρτηση μεταξύ των φάσεων ανάπτυξης του μεταγλωττιστή.

Το κεφάλαιο εντάσσεται στη φάση της παραγωγής τελικού κώδικα, όπως και το επόμενο κεφάλαιο του βιβλίου αυτού. Για λόγους καλύτερης δόμησης η φάση της παραγωγής ενδιάμεσου κώδικα χωρίστηκε σε δύο κεφάλαια. Θα μπορούσε να έχει ως υπότιτλο *Ενδιάμεση αναπαράσταση, αριθμητικές και λογικές παραστάσεις*.

Στην αρχή θα παρουσιαστεί η ενδιάμεση αναπαράσταση που θα χρησιμοποιήσουμε. Πρόκειται για μία γλώσσα που βασίζεται σε τετράδες. Σκοπός της είναι να σχηματίσει ένα ενδιάμεσο επίπεδο ανάμεσα στην αρχική και την τελική γλώσσα. Η ενδιάμεση γλώσσα εξακολουθεί να είναι μία γλώσσα υψηλού επιπέδου, αφού περιέχει μεταβλητές και κλήσεις συναρτήσεων, οι δομές ελέγχου όμως της γλώσσας έχουν απλοποιηθεί και από δομές μιας δομημένης γλώσσας προγραμματισμού έχουν γίνει δομές απλών αλμάτων και αλμάτων υπό συνθήκη προς μία ετικέτα.

Στη συνέχεια θα δούμε πώς μεταφράζονται οι αριθμητικές και λογικές παραστάσεις. Με την ίδια φιλοσοφία, κάθε αριθμητική παράσταση θα αποσυντεθεί σε απλές αριθμητικές πράξεις (πρόσθεση, αφαίρεση, πολλαπλασιασμός και διαίρεση), καθεμία από αυτές έχει έναν τελεστή, δύο τελούμενα και εκχωρεί το αποτέλεσμα σε ένα άλλο τελούμενο.

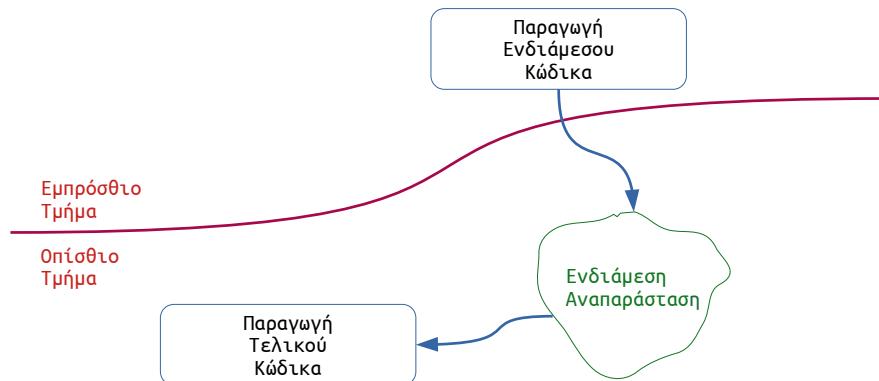
Οι λογικές παραστάσεις αποσυντίθενται επίσης σε απλούστερες συγκρίσεις, καθεμία από τις οποίες δημιουργεί έναν αριθμό από αλμάτα και αλμάτα υπό συνθήκη. Κάθε αλμάτα υπό συνθήκη έχει ακριβώς έναν τελεστή σύγκρισης, δύο τελούμενα που συγκρίνονται μεταξύ τους και μία ετικέτα στην οποία θα μεταβεί ο έλεγχος, αν η συνθήκη είναι αληθής.

### Προαπαιτούμενη γνώση:

- κεφάλαιο 1
  - κεφάλαιο 2
  - κεφάλαιο 3
  - κεφάλαιο 5
- 

Η μετατροπή του κώδικα από την αρχική γλώσσα στην τελική δεν γίνεται απευθείας. Μεσολαβεί η μετατροπή του σε μία ενδιάμεση γλώσσα, την οποία συνηθίζουμε να λέμε και ενδιάμεση αναπαράσταση, η οποία εξακολουθεί να θεωρείται γλώσσα υψηλού επιπέδου, αλλά οι δομές της δεν είναι τόσο σύνθετες, όσο αυτές της αρχικής γλώσσας, ενώ η συντακτική της ανάλυση είναι τετριμμένη.

Θα παρουσιάσουμε εδώ μια τέτοια γλώσσα και θα δούμε πώς από τη φάση της συντακτικής ανάλυσης θα οδηγηθούμε στην παραγωγή του ενδιάμεσου κώδικα.



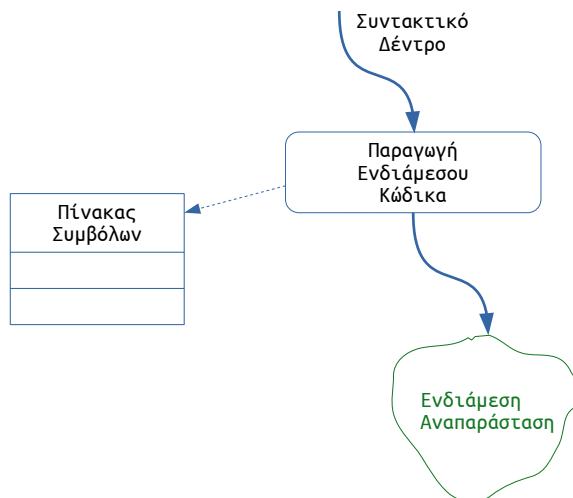
Σχήμα 6.1: Η ενδιάμεση αναπαράσταση αποτελεί μέσο επικοινωνίας ανάμεσα στο εμπρόσθιο και το οπίσθιο τμήμα του μεταγλωττιστή.

Η ύπαρξη μιας ενδιάμεσης γλώσσας, ως σκαλοπάτι ανάμεσα στην αρχική και την τελική γλώσσα, έχει πολλά πλεονεκτήματα. Η σχεδίαση του μεταγλωττιστή απλοποιείται σημαντικά και χωρίζεται σε δύο ανεξάρτητες φάσεις, όπως είχε συζητηθεί στο κεφάλαιο της εισαγωγής και εικονίζεται στο σχήμα 6.1

- τη φάση πριν την παραγωγή του ενδιάμεσου κώδικα, το εμπρόσθιο τμήμα (*front end*),
- τη φάση μετά την παραγωγή του ενδιάμεσου κώδικα, το οπίσθιο τμήμα (*back end*).

Ο διαχωρισμός αυτός, πέρα από την απλοποίηση που επιφέρει στη σχεδίαση του μεταγλωττιστή, έχει ακόμα ένα πολύ σημαντικό πλεονέκτημα. Η ενδιάμεση γλώσσα αποτελεί ένα στρώμα επικοινωνίας ανάμεσα στο εμπρόσθιο και το οπίσθιο τμήμα το οποίο είναι ανεξάρτητο τόσο της αρχικής, όσο και της τελικής γλώσσας. Έτσι, το εμπρόσθιο τμήμα θα είναι το ίδιο για κάθε μεταγλωττιστή της αρχικής γλώσσας, ενώ το οπίσθιο τμήμα θα είναι το ίδιο για κάθε μεταγλωττιστή που παράγει κώδικα στην ίδια τελική γλώσσα, ανεξάρτητα ποια θα είναι η αρχική. Αυτό δίνει την ευκαιρία για επαναχρησιμοποίηση κώδικα και ευκολότερη ανάπτυξη μεταγλωττιστών, αν έχει ήδη αναπτυχθεί μεταγλωττιστής της αρχικής γλώσσας ή μεταγλωττιστής που παράγει κώδικα στην τελική γλώσσα που μας ενδιαφέρει.

Η παραγωγή ενδιάμεσου κώδικα ως ενότητα λογισμικού μπορούμε να θεωρήσουμε ότι παίρνει ως είσοδο το δέντρο της συντακτικής ανάλυσης. Πιο ξεκάθαρο είναι τι δημιουργεί ως αποτέλεσμα, το οποίο είναι το αρχικό πρόγραμμα μεταφρασμένο σε ενδιάμεση αναπαράσταση. Εισάγει εγγραφές και γενικά προσθέτει πληροφορία στον πίνακα συμβόλων, αν και δεν θα ασχοληθούμε καθόλου με αυτό σε αυτό το κεφάλαιο, αφού θα παρουσιαστεί αναλυτικά και ολοκληρωμένα ως νοηματική ενότητα στο επόμενο. Σχηματικά η παραγωγή ενδιάμεσου κώδικα ως ενότητα λογισμικού φαίνεται στο σχήμα 6.2.



Σχήμα 6.2: Παραγωγή κώδικα.

Περισσότερο υλικό για την παραγωγή ενδιάμεσου κώδικα μπορείτε να αναζητήσετε στα ακόλουθα κεφάλαια βιβλίων: [1, 2, κεφ.5], [3, 4, κεφ.6], [5, κεφ.7].

## 6.1 Η ενδιάμεση αναπαράσταση

Ένα πρόγραμμα συμβολισμένο στην ενδιάμεση γλώσσα αποτελείται από μία σειρά από τετράδες, οι οποίες είναι αριθμημένες έτσι ώστε σε κάθε τετράδα να μπορούμε να αναφερθούμε χρησιμοποιώντας τον αριθμό της ως ετικέτα. Για παράδειγμα, να μπορούμε να κάνουμε άλμα σε αυτήν χρησιμοποιώντας της ετικέτα της τετράδας. Δεν έχουμε κάποιο λόγο οι ετικέτες των τετράδων να είναι αριθμοί, αρκεί το σύνολο των τετράδων να είναι διατεταγμένο, δηλαδή, να γνωρίζουμε για κάθε τετράδα ποια είναι η επόμενή της. Ο ευκολότερος και πιο αυτονόητος τρόπος για να το πετύχουμε αυτό είναι με την αρίθμηση.

Κάθε τετράδα αποτελείται από έναν τελεστή και τρία τελούμενα. Αν μετρήσουμε και την ετικέτα, πρόκειται για μία τετράδα που αποτελείται από ... πέντε πράγματα. Από τις ονομασίες τελεστής και τελούμενα μπορεί κανείς να συμπεράνει ότι ο τελεστής καθορίζει την ενέργεια που πρόκειται να γίνει και τα τελούμενα είναι εκείνα πάνω στα οποία θα εφαρμοστεί η ενέργεια.

Για να απλοποιήσουμε όσο τον δυνατόν τους συμβολισμούς, θεωρούμε ότι έχουμε έναν τελεστή και τρία τελούμενα, πάντοτε, ανεξάρτητα του πόσα τελούμενα χρειάζεται ο τελεστής. Αν ο τελεστής είναι μία ενέργεια που έχει νόημα να γίνει πάνω σε λιγότερα από τρία τελούμενα, τότε πρέπει ένα ή δύο ή και τρία από αυτά να μείνουν κενά. Αν ο τελεστής απαιτεί περισσότερα από τρία τελούμενα, τότε πρέπει να βρούμε κάποια λύση ώστε να μπορέσουμε να περιγράψουμε αυτό που θέλουμε με κάποιον άλλον τρόπο. Ο πιο προφανής τρόπος είναι να χρησιμοποιήσουμε περισσότερες της μίας τετράδες είτε τη μία ως συνέχεια της άλλης, είτε δύο αυτόνομες που αν εκτελεστούν η μία μετά την άλλη, θα δώσουν το επιθυμητό αποτέλεσμα.

Οι τετράδες του ενδιάμεσου κώδικα αποθηκεύονται σε μία κατάλληλη δομή στη μνήμη. Στην Python ο καταλληλότερος τρόπος είναι να δημιουργήσουμε μία κλάση Quad η οποία να έχει τα πεδία που αναφέραμε παραπάνω. Κάθε πρόγραμμα μπορεί να είναι ένα αντικείμενο που να ενθυλακώνει κατάλληλα μία διατεταγμένη λίστα από αντικείμενα των τετράδων αυτών.

Όπως θα παρατηρήσετε και εσείς στη συνέχεια, η επιλογή της αποθήκευσης ενός προγράμματος ενδιάμεσης γλώσσας σε αρχείο κειμένου δεν είναι η καταλληλότερη επιλογή. Χρειαζόμαστε την ευχέρεια όχι μόνο να μπορούμε να παράγουμε νέες τετράδες, αλλά να επιστρέφουμε, και να τροποποιούμε σε μεταγενέστερο στάδιο, τετράδες που έχουν ήδη παραχθεί, κάτι το οποίο ένα αρχείο κειμένου δεν είναι σχεδιασμένο να μπορεί να κάνει με τρόπο εύκολο.

Η λίστα με τις τετράδες αποτελεί το μέσο επικοινωνίας της φάσης της παραγωγής ενδιάμεσου κώδικα και

της παραγωγής τελικού κώδικα, αφού ο τελικός κώδικας παράγεται με βάση αυτή τη λίστα και πληροφορία που αντλεί από τον πίνακα συμβόλων.

Παρακάτω θα περιγράψουμε τις εντολές της ενδιάμεσης γλώσσας που θα χρησιμοποιήσουμε για τη μετατροπή αρχικού κώδικα *C-imple* σε ενδιάμεση αναπαράσταση. Αν η αρχική γλώσσα ήταν περισσότερο πολύπλοκη από τη *C-imple*, για παράδειγμα αν υποστήριζε πίνακες, τότε θα ήταν προτιμότερο να προσθέσουμε μερικές ακόμα εντολές στο σύνολο των εντολών της ενδιάμεσης γλώσσας.

Γενικά, η ενδιάμεση γλώσσα είναι ανεξάρτητη και από την αρχική γλώσσα και από την τελική γλώσσα. Είμαστε ελεύθεροι να τη σχεδιάσουμε όπως εμείς επιθυμούμε. Η ενδιάμεση αναπαράσταση δεν είναι ορατή στον έξω κόσμο, αλλά παράγεται από ένα τμήμα και διαβάζεται από κάποιο άλλο τμήμα του μεταγλωττιστή που υλοποιούμε.

Ας δούμε τις εντολές της ενδιάμεσης γλώσσας κατηγοριοποιημένες:

Ομαδοποίηση κώδικα:

```
begin_block, name, _, _
end_block, name, _, _
```

Οι εντολές `begin_block` και `end_block` χρησιμοποιούνται για να ομαδοποιήσουμε εντολές ενδιάμεσου κώδικα. Βασικά, αυτό που θέλουμε να οριοθετήσουμε με τις `begin_block` και `end_block` είναι η αρχή και το τέλος του ενδιάμεσου κώδικα που παρήχθη για μια συνάρτηση, διαδικασία ή για το κυρίως πρόγραμμα. Έτσι, στην αρχή του κώδικα μιας συνάρτησης, διαδικασίας ή του κυρίως προγράμματος και πριν την πρώτη εκτελέσιμη εντολή, τοποθετούμε μία `begin_block` με το όνομα της συνάρτησης, της διαδικασίας ή του κυρίως προγράμματος και στο τέλος μία `end_block`.

Παράδειγμα:

Ο κώδικας:

```
function f()
{
    ...
}
```

Θα δημιουργήσει τον κώδικα:

```
xxx: begin_block, f, _, _
      ...
xxx: end_block, f, _, _
```

Εκχώρηση:

```
:=, source, _, target
```

Εκχωρεί την τιμή του `source` στο `target`. Το `source` μπορεί να είναι μεταβλητή, αριθμητική ή συμβολική σταθερά, ενώ το `target` μεταβλητή (ή και συμβολική σταθερά στη γενικότερη περίπτωση - στην *C-imple* δεν χρειαζόμαστε να κάνουμε εκχώρηση σε συμβολική σταθερά).

Παράδειγμα:

Η εντολή ενδιάμεσου κώδικα:

```
:=, a, _, b
```

αντιστοιχεί στην ακόλουθη εντολή της αρχικής γλώσσας:

```
b := a
```

Αριθμητική πράξη:

```
op, operand1, operand2, target
```

Το ορ είναι ένα εκ των συμβόλων της πρόσθεσης, της αφαίρεσης, του πολλαπλασιασμού ή της διαίρεσης. Τα operand1 και operand2 είναι τα δύο τελούμενα πάνω στα οποία θα εφαρμοστεί η πράξη και το target το τελούμενο στο οποίο θα αποθηκευτεί το αποτέλεσμα.

Παράδειγμα:

Η εντολή ενδιάμεσου κώδικα:

```
+, a, b, c
```

αντιστοιχεί στην ακόλουθη εντολή της αρχικής γλώσσας:

```
c := a + b
```

Εντολή άλματος:

```
jump, _, _, label
```

Μεταφέρει τον έλεγχο στην εντολή με ετικέτα label.

Παράδειγμα:

Το πρόγραμμα:

```
100: :=, x, _, y
101: jump, _, _, 100
```

δημιουργεί ατέρμονο βρόχο, επιστρέφοντας τον έλεγχο της ροής μετά την εκτέλεση της 101 στην 100.

Εντολή λογικού άλματος:

```
conditional_jump, a, b, label
```

Το conditional\_jump είναι ένα από τα ακόλουθα λογικά άλματα και ανάλογα με το λογικό άλμα η λειτουργία του ορίζεται ως εξής:

=	:	αν a==b τότε εκτελείται άλμα στην ετικέτα label
<	:	αν a<b τότε εκτελείται άλμα στην ετικέτα label
>	:	αν a>b τότε εκτελείται άλμα στην ετικέτα label
<=	:	αν a≤b τότε εκτελείται άλμα στην ετικέτα label
>=	:	αν a≥b τότε εκτελείται άλμα στην ετικέτα label
<>	:	αν a≠b τότε εκτελείται άλμα στην ετικέτα label

Παράδειγμα:

Το παρακάτω πρόγραμμα σε ενδιάμεση αναπαράσταση μεταφέρει τον έλεγχο σε τρία διαφορετικά σημεία του προγράμματος, ανάλογα με το αν η διακρίνουσα είναι θετική, αρνητική ή μηδέν.

```
200: =, diakrinousa, 0, 300
201: >, diakrinousa, 0, 400
202: <, diakrinousa, 0, 500
```

Κλήση συνάρτησης ή διαδικασίας:

```
call, name, _, _
```

Ο τρόπος κλήσης μιας συνάρτησης και μίας διαδικασίας δεν διαφέρουν. Σημειώνεται με την call και μετά ακολουθεί το όνομά της. Οι απαραίτητοι διαχωρισμοί θα γίνουν σε άλλο σημείο και, πιο συγκεκριμένα, στο πέρασμα των παραμέτρων.

Πέρασμα πραγματικής παραμέτρου:

```
par, name, mode, _
```

Το πατείται το όνομα της παραμέτρου, ενώ το mode ο τρόπος περάσματος. Το mode έχει τρεις επιλογές που έχουν νόημα στην *C-imple*:

- cv: πέρασμα με τιμή.
- ref: πέρασμα με αναφορά.
- ret: επιστροφή τιμής συνάρτησης.

Έχει ενδιαφέρον να σταθούμε στο τελευταίο. Χειρίζόμαστε την επιστροφή τιμής της συνάρτησης ως παράμετρο. Αλήθεια, γιατί όχι; Αυτό που στην ουσία συμβαίνει είναι ότι υπολογίζεται μία τιμή και αυτή η τιμή επιστρέφεται στην καλούσα συνάρτηση. Γιατί θα έπρεπε να αναζητήσουμε πολύ διαφορετικό τρόπο για να το υλοποιήσουμε από ότι στο πέρασμα με αναφορά;

Έτσι, όταν έχουμε μια συνάρτηση, θα πρέπει να δεσμεύουμε μία μεταβλητή η οποία θα εμφανιστεί ως παράμετρος ret της συνάρτησης:

`par, name, ret, _`

Όμως ποιο είναι το όνομα που θα επιλέξουμε και θα ονομάσουμε την παράμετρο;

Εδώ πρέπει να εισαγάγουμε την έννοια της προσωρινής μεταβλητής. Οι μεταβλητές που έχουμε χρησιμοποιήσει μέχρι τώρα ήταν μεταβλητές που ο προγραμματιστής είχε δηλώσει με κάποιο τρόπο στο πρόγραμμά του. Δημιουργείται, όμως, πολλές φορές η ανάγκη να αποθηκεύσουμε τιμές οι οποίες χρειάζονται, είτε ως ενδιάμεσα αποτελέσματα υπολογισμών είτε για άλλους λόγους, όπως αυτός της επιστροφής τιμής της συνάρτησης. Οι μεταβλητές αυτές δεν διαφέρουν από τις μεταβλητές που θα δηλωθούν μέσα στο πρόγραμμα από τον προγραμματιστή: χρειάζονται κάποιο όνομα για να αναφέρεσαι σε αυτές, πρέπει να έχουν κάποιον τύπο αν η γλώσσα το απαιτεί, χρειάζονται χώρο στη μνήμη για να αποθηκευτούν.

Στο επίπεδο της μετάφρασης που είμαστε τη στιγμή αυτή, μας αρκεί να επιλέξουμε και να τους δώσουμε το κατάλληλο όνομα και τα υπόλοιπα θα μας απασχολήσουν αργότερα. Άλλωστε είπαμε, ότι η ενδιάμεση αναπαράσταση είναι γλώσσα υψηλού επιπέδου. Άρα η έννοια της μεταβλητής υπάρχει ακριβώς όπως στην αρχική γλώσσα και δεν απαιτεί κάποια ιδιαίτερη διαχείριση, όπως για παράδειγμα η τοποθέτηση στη μνήμη.

Με τον όρο κατάλληλο όνομα, στον οποίο αναφερθήκαμε παραπάνω, εννοούμε κάποιο όνομα που δεν έχει χρησιμοποιηθεί μέχρι στιγμής, είτε ως προσωρινή μεταβλητή, είτε από τον προγραμματιστή ως μεταβλητή ή ονομασία συνάρτησης ή διαδικασίας στο αρχικό πρόγραμμα. Επίσης, θέλουμε να εξασφαλίσουμε ότι δεν θα χρησιμοποιηθεί και στο μέλλον. Για να τα εξασφαλίσουμε αυτά, επιλέγουμε να κατασκευάζουμε το όνομα της μεταβλητής ως εξής:

- Για να συμβαδίζει με τον κανόνα ότι μία μεταβλητή ξεκινάει από γράμμα, κάθε προσωρινή μεταβλητή ξεκινάει με το γράμμα T. Η αλήθεια είναι ότι, αν παραβαίναμε τον κανόνα αυτόν, δεν θα δημιουργούνταν κάποια δυσεπίλυτα προβλήματα.
- Για να εξασφαλίσουμε ότι δεν έχει χρησιμοποιηθεί από τον προγραμματιστή, μετά το T θα ακολουθεί μία κάτω παύλα (“\_”). Αφού η κάτω παύλα δεν ανήκει στο αλφάριθμο της *C-imple*, μπορούμε να εξασφαλίσουμε ότι το όνομα της μεταβλητής που κατασκευάζουμε, δεν έχει χρησιμοποιηθεί από τον προγραμματιστή.
- Για να εξασφαλίσουμε ότι το ίδιο όνομα προσωρινής μεταβλητής δεν έχει δημιουργηθεί πάλι ως προσωρινή μεταβλητή, χρησιμοποιούμε έναν μετρητή, τον οποίο αυξάνουμε κατά 1, κάθε φορά που δημιουργείται μία νέα προσωρινή μεταβλητή. Ο μετρητής αυτός ακολουθεί την κάτω παύλα και ολοκληρώνει το όνομα της μεταβλητής.

Οι προσωρινές μεταβλητές που θα κατασκευαστούν θα είναι, με τη σειρά, οι ακόλουθες:

T\_1, T\_2, T\_3, ... .

Επιστρέφουμε στην περιγραφή της `par` και της `call` δίνοντας δύο παραδείγματα μετάφρασης. Έστω ο κώδικας:

```
call f(in x, in y)
```

Θα δημιουργήσει τον ακόλουθο ενδιάμεσο κώδικα:

```
150: par, x, cv, _
151: par, y, ref, _
152: call, f, _, _
```

Ενώ ο κώδικας:

```
... := f(in x, in y)
```

Θα δημιουργήσει τον ακόλουθο ενδιάμεσο κώδικα:

```
150: par, x, cv, _
151: par, y, ref, _
152: par T_1, ret, _
153: call, f, _, _
```

και θα ακολουθήσει η εντολή της εκχώρησης, η οποία δεν φαίνεται εδώ ποια θα είναι.

Χρειαζόμαστε ακόμα μία εντολή για την επιστροφή τιμής. Είναι αυτό που θα αντιστοιχίζαμε στο return μίας συνάρτησης. Μας αρκεί μία εντολή με μία παράμετρο, τη μεταβλητή που θα επιστραφεί από την return:

```
ret, source, _, _
```

*Είσοδος-έξοδος:*

Χρειαζόμαστε επίσης δύο εντολές, μία για την εισαγωγή από το πληκτρολόγιο και μία για την εμφάνιση της τιμής μιας μεταβλητής στην οθόνη. Αυτές θα μπορούσαν να είναι οι ακόλουθες:

```
in, x, _, _
```

και:

```
out, x, _, _
```

όπου x η μεταβλητή η οποία θα διαβαστεί από το πληκτρολόγιο ή θα τυπωθεί στην οθόνη, αντίστοιχα.

*Τερματισμός προγράμματος:*

Τέλος, έχουμε ακόμα μία εντολή. Πρόκειται για την:

```
halt, _, _, _
```

η οποία τερματίζει το πρόγραμμα και τοποθετείται πριν από την end\_block του κυρίως προγράμματος.

Παράδειγμα:

Ο κώδικας:

```
program test
{ ...
}
```

Θα δημιουργήσει τον κώδικα:

```
xxx: begin_block, main_test, _, _
...
xxx: halt, _, _, _
xxx: end_block, main_test, _, _
```

## 6.2 Βοηθητικές συναρτήσεις

Κατά τη σχεδίαση της ενότητας λογισμικού που θα παράγει τον ενδιάμεσο κώδικα, υπάρχουν κάποιες ενέργειες που επαναλαμβάνονται συχνά και είναι κατάλληλες για να τις υλοποιήσουμε με τη μορφή συναρτήσεων. Πέρα από το ότι κάτι τέτοιο αποτελεί ορθή απόφαση στη σχεδίαση του λογισμικού, διευκολύνει πάρα πολύ και στην περιγραφή του σχεδίου του ενδιάμεσου κώδικα, διότι καθεμία θα αποτελεί μία καλά ορισμένη λειτουργία την οποία θα μπορούμε να επικαλούμαστε στην περιγραφή του σχεδίου.

Έτσι οι βοηθητικές συναρτήσεις που θα ορίσουμε είναι οι ακόλουθες:

- **genQuad(operator, operand1, operand2, operand3):**  
Δημιουργεί μία νέα τετράδα, το πρώτο πεδίο της οποίας είναι το operator και τα τρία επόμενα τα operand1, operand2 και operand3. Ο αριθμός της τετράδας που δημιουργείται προκύπτει αυτόματα από τον αριθμό της τελευταίας τετράδας που δημιουργήθηκε, συν ένα
- **nextQuad():**  
Επιστρέφει την ετικέτα της επόμενης τετράδας που θα δημιουργηθεί, όταν κληθεί η genQuad
- **newTemp():**  
Επιστρέφει το όνομα της επόμενης προσωρινής μεταβλητής. Αν η τελευταία προσωρινή μεταβλητή που δημιουργήθηκε είναι η T\_2, τότε θα δημιουργήσει και θα επιστρέψει την T\_3,
- **emptyList():**  
Δημιουργεί και επιστρέφει μία νέα κενή λίστα στην οποία στη συνέχεια θα τοποθετηθούν ετικέτες τετράδων
- **makeList(label):**  
Δημιουργεί και επιστρέφει μία νέα λίστα η οποία έχει ως μοναδικό στοιχείο της την ετικέτα τετράδας label
- **mergeList(list1, list2):**  
Δημιουργεί μία λίστα και συνενώνει τις list1 και list2 σε αυτήν
- **backpatch(list,label):**  
Διαβάζει μία μία τις τετράδες που σημειώνονται στη λίστα list και για την τετράδα που αντιστοιχεί στην ετικέτα αυτή, συμπληρώνουμε το τελευταίο πεδίο της με το label. Όταν συμπληρωθούν όλες οι τετράδες που σημειώνονται στη λίστα αυτή, η λίστα δεν χρειάζεται άλλο και μπορεί να αποδεσμεύσει τη μνήμη που κατέχει

Ένα παράδειγμα ίσως βοηθήσει στην κατανόηση της λειτουργίας των παραπάνω συναρτήσεων:

Έστω το παρακάτω πρόγραμμα:

```
x1 = makeList(nextQuad())
genQuad('jump', '_ ', '_ ', '_ ')
genQuad('+', 'a', '1', 'a')
x2 = makeList(nextQuad())
genQuad('jump', '_ ', '_ ', '_ ')
x = mergeList(x1,x2)
genQuad('+', 'a', '2', 'a')
backpatch(x,nextQuad())
```

Ας υποθέσουμε ότι η πρώτη τετράδα που θα δημιουργηθεί είναι η 100. Στην πρώτη γραμμή του κώδικα θα δημιουργηθεί η λίστα x1, η οποία θα έχει μέσα της την ετικέτα 100, αφού την ετικέτα 100 θα επιστρέψει η nextQuad(). Χρονικά, αμέσως μετά θα δημιουργηθεί η τετράδα 100, η οποία θα είναι το μη συμπληρωμένο

jump. Στη συνέχεια, στην ετικέτα 101 θα δημιουργηθεί η τετράδα που θα αυξάνει το a κατά 1. Έτσι μέχρι στιγμής, έχουμε σημειώσει την τετράδα 100 ως μη συμπληρωμένη και έχουμε δημιουργήσει τις τετράδες:

```
100: jump, _, _, _
101: +, a, 1, a
```

Με όμοιο τρόπο, θα δημιουργηθεί η τετράδα 102 και θα σημειωθεί στη λίστα x. Στη συνέχεια οι λίστες x1 και x2 συνενώνονται στην λίστα x. Ως το σημείο αυτό έχουμε δημιουργήσει τον κώδικα:

```
100: jump, _, _, _
101: +, a, 1, a
102: jump, _, _, _
103: +, a, 2, a
```

και έχουμε τη λίστα x η οποία έχει μέσα τις τετράδες 100 και 102:

```
x = [100, 102]
```

Καλώντας την backpatch(x, nextQuad()) κάθε τετράδα που είναι σημειωμένη στη λίστα x θα συμπληρωθεί με το nextQuad(), δηλαδή με το 104. Η λίστα x θα επιστρέψει τον χώρο που είχε δεσμεύσει στη μνήμη. Ο κώδικας που τελικά θα παραχθεί ακολουθεί ολοκληρωμένος:

```
100: jump, _, _, 104
101: +, a, 1, a
102: jump, _, _, 104
103: +, a, 2, a
```

### 6.3 Αριθμητικές παραστάσεις

Σύμφωνα με τη γραμματική της γλώσσας, υποστηρίζονται αριθμητικές παραστάσεις με τις τέσσερις αριθμητικές πράξεις, καθώς και η ομαδοποίηση και προτεραιότητα ανάμεσα σε αυτές που ορίζεται με τη χρήση παρενθέσεων. Θα χρησιμοποιήσουμε μία γραμματική, απλούστερη από αυτήν της C-implement, η οποία όμως γενικεύεται πολύ εύκολα. Πρόκειται για μία γραμματική η οποία υποστηρίζει προσθέσεις, πολλαπλασιασμούς και προτεραιότητα με παρενθέσεις. Η γενίκευση για αφαίρεση και διαίρεση είναι προφανής. Η γραμματική ακολουθεί:

```
# addition
E → T(1) ( + T(2) )*
# multiplication
T → F(1) ( * F(2) )*
# priority with parentheses
F → ( E )
# terminal symbols
F → ID
```

Στη γραμματική έχει χρησιμοποιηθεί ο συμβολισμός του δείκτη σε παρένθεση στη θέση του εκθέτη, προκειμένου να διαχωρίσουμε τις διαφορετικές εμφανίσεις του κάθε μη τερματικού συμβόλου. Τα T<sup>(1)</sup> και T<sup>(2)</sup> δεν αποτελούν διαφορετικό κανόνα, απλά διαφορετική εμφάνιση του κανόνα στη γραμματική και χρειαζόμαστε έναν συμβολισμό για να αναφερόμαστε και να διαχωρίζουμε εύκολα τις εμφανίσεις αυτές. Αν τις θεωρήσουμε δηλαδή ως κλήσεις συναρτήσεων, όπως τελικά θα υλοποιηθούν στον κώδικα του μεταγλωττιστή, πρόκειται για διαφορετικές κλήσεις της ίδιας συνάρτησης.

Ας περάσουμε, όμως, σιγά σιγά στο να οργανώσουμε στο μυαλό μας τη μεθοδολογία που θα ακολουθήσουμε. Θα πρέπει να αναζητήσουμε τα σημεία εκείνα της γραμματικής, και κατ' επέκταση του κώδικα του συντακτικού αναλυτή, στα οποία πρέπει να εισαχθούν σημασιολογικές ρουτίνες που θα παραγάγουν

ενδιάμεσο κώδικα ισοδύναμο με την αριθμητική παράσταση που θέλουμε να μετατρέψουμε σε ενδιάμεσο κώδικα.

Με τον τρόπο που έχει συνταχθεί η γραμματική, είναι εφικτό να θεωρήσουμε ότι υπάρχει ανεξαρτησία ανάμεσα στους κανόνες. Αυτό μας επιτρέπει να σχεδιάσουμε την παραγωγή ενδιάμεσου κώδικα για κάθε κανόνα ξεχωριστά. Θα θεωρήσουμε ότι οι κανόνες που ενεργοποιούνται στο δεξί μέλος του κανόνα λειτουργούν σωστά, όπως ακριβώς πράττουμε όταν γράφουμε τον κώδικα μιας συνάρτησης η οποία καλεί μέσα της άλλες συναρτήσεις. Άλλωστε, τα μη τερματικά σύμβολα στο δεξί μέλος ενός κανόνα δεν είναι τίποτε άλλο παρά συναρτήσεις που θα κληθούν, θα εκτελεστούν και θα παραγάγουν κάποιο αποτέλεσμα που η καλούσα συνάρτηση θα χρησιμοποιήσει.

Έτσι, κάθε κανόνας, με βάση τα δεδομένα που θα συλλέξει από τα μη τερματικά σύμβολα και με βάση τα τερματικά σύμβολα που θα αναγνωρίσει, θα κάνει τα εξής:

- Θα παραγάγει ενδιάμεσο κώδικα, όπου και εάν απαιτείται. Διαισθητικά μπορούμε να φανταστούμε ότι μία αριθμητική παράσταση πρέπει να παραγάγει κώδικα όταν εκτελείται μία πρόσθεση, ένας πολλαπλασιασμός ή εκχωρείται τιμή σε μία μεταβλητή, είτε λόγω κάποιου υπολογισμού, είτε λόγω αναγνώρισης κάποιου τερματικού συμβόλου.
- Θα προετοιμάσει και θα πρωθήσει πληροφορία στον κανόνα που τον κάλεσε. Την πληροφορία αυτήν την αναμένει ο κανόνας που τον κάλεσε προκειμένου να συνθέσει τον δικό του ενδιάμεσο κώδικα ή να συνθέσει την πληροφορία που αυτός θα προετοιμάσει και θα πρωθήσει με τη σειρά του στον κανόνα που τον κάλεσε.

Έτσι, κάθε κανόνας θα χρησιμοποιήσει πληροφορία που συγκεντρώθηκε και του μεταφέρθηκε από τα μη τερματικά σύμβολα που συναντήθηκαν στο δεξί του μέλος, και με βάση τα τερματικά σύμβολα που αναγνώρισε, θα δημιουργήσει ενδιάμεσο κώδικα. Με τη σειρά του θα επιστρέψει το δικό του αποτέλεσμα στον κανόνα που τον ενεργοποίησε, ο οποίος και θα τον χρησιμοποιήσει για να δημιουργήσει τον δικό του τελικό κώδικα.

Τι είναι όμως αυτό που πρέπει να μεταφερθεί από κανόνα σε κανόνα; Ποια ανάγκη επικοινωνίας υπάρχει ανάμεσα στα μη τερματικά σύμβολα της γραμματικής;

Μπορούμε να φανταστούμε κάθε κανόνα της γραμματικής που περιγράφει τις αριθμητικές εκφράσεις ως ένα υποσύνολο υπολογισμών. Οι υπολογισμοί αυτοί υλοποιούνται από τον ενδιάμεσο κώδικα που τελικά παράγεται. Το αποτέλεσμα αυτών των υπολογισμών θα βρεθεί αποθηκευμένο σε κάποιες από τις μεταβλητές που χρησιμοποιεί ο μεταγλωττιστής, είτε πρόκειται για μεταβλητές που έχει δηλώσει ο προγραμματιστής, είτε πρόκειται για προσωρινές μεταβλητές που έχει δημιουργήσει και χρησιμοποιήσει ο μεταγλωττιστής. Η μεταβλητή, η οποία όταν θα τρέξει ο ενδιάμεσος κώδικας θα περιέχει το αποτέλεσμα της παράστασης που περιγράφει ο κανόνας (και όλο το συντακτικό δέντρο των κανόνων που έχουν ενεργοποιηθεί κάτω από αυτόν), αποτελεί το αποτέλεσμα του κανόνα. Καθένας από τους τέσσερις κανόνες της γραμματικής επιστρέφει στον κανόνα που τον ενεργοποίησε μία μεταβλητή ως αποτέλεσμα. Αυτό ισχύει για όλους τους κανόνες, ανεξάρτητα αν πρόκειται για κανόνα που θα ενεργοποιηθεί βαθιά στο δέντρο της συντακτικής ανάλυσης ή τον κανόνα που θα εκκινήσει το δέντρο της αριθμητικής παράστασης και τελικά θα επιστρέψει αποτέλεσμα στον κανόνα από τον οποίο ενεργοποιήθηκε η αναγνώριση της αριθμητικής παράστασης.

Συνηθίζουμε να δίνουμε το όνομα *place* στις μεταβλητές αυτές. Έτσι, ο πρώτος κανόνας θα παραλάβει από την  $T^{(1)}$  το  $T^{(1)}.place$ , από την  $T^{(2)}$  το  $T^{(2)}.place$  και θα δημιουργήσει (ενδιάμεσο κώδικα και) την  $E.place$ .

Ο πρώτος κανόνας:

$$E \rightarrow T^{(1)} ( + T^{(2)} )^*$$

είναι υπεύθυνος για την τέλεση της πρόσθεσης. Διαισθητικά πάλι, θα περιμέναμε όταν συναντηθεί κάπου το σύμβολο  $+$  ο κανόνας να είναι υπεύθυνος να δημιουργήσει τον κώδικα που απαιτείται σε ενδιάμεση γλώσσα. Πράγματι έτσι είναι. Ας δούμε πώς γίνεται.

Ας ξεκινήσουμε τη σκέψη μας με την περίπτωση που ο κανόνας ενεργοποιείται, αλλά δεν εμφανίζεται στην αριθμητική παράσταση κάποιο σύμβολο πρόσθεσης. Για παράδειγμα, ο κώδικας `a := 1`, θα ενεργοποιήσει τον κανόνα E, στη συνέχεια τον κανόνα T και ακολούθως τον κανόνα F, χωρίς να αναγνωρίζονται πουθενά τερματικά σύμβολα για πρόσθεση ή πολλαπλασιασμό. Αν δεν αναγνωρίζονται τερματικά σύμβολα για πρόσθεση ή πολλαπλασιασμό, δεν υπάρχει λόγος να παραχθεί κώδικας από τους κανόνες αυτούς. Το τερματικό σύμβολο, όμως, που έχει αναγνωρίσει ο F, πρέπει να περαστεί στον T και στη συνέχεια στον E.

Αν μείνουμε στον κανόνα E, το αποτέλεσμα αυτό θα επιστρέψει από τον κανόνα  $T^{(1)}$ , μέσα από τη μεταβλητή  $T^{(1)}.\text{place}$ . Πράγματι, αν το σκεφτούμε περισσότερο, ο  $T^{(1)}$  δίνει ως αποτέλεσμα στον E τη μεταβλητή που έχει το αποτέλεσμα των υπολογισμών του T. Αφού δεν γίνονται άλλοι υπολογισμοί, η ίδια μεταβλητή είναι αυτή που περιέχει και το αποτέλεσμα των υπολογισμών του E.

Ας δούμε πώς θα συμβολίσουμε τα παραπάνω στη γραμματική μας. Το `{p1}` υποδηλώνει σε ποιο σημείο του κανόνα θα γίνει η σημασιολογική ενέργεια, ενώ παρακάτω περιγράφεται η ενέργεια αυτή. Έτσι, επειδή κατά τη μετάφραση της παράστασης δεν θα μπούμε καθόλου μέσα στο  $(+T^{(2)})^*$  μπορούμε να τοποθετήσουμε την ενέργεια με την εκχώρηση του  $T^{(1)}.\text{place}$  στο E.place πριν ή μετά αυτό. Θα επιλέξουμε να τοποθετήσουμε το `{p1}` στο τέλος του κανόνα. Είναι πιο λογικό, αφού το E.place είναι το αποτέλεσμα που τελικά θα επιστραφεί από τον κανόνα:

$$E \rightarrow T^{(1)} (+ T^{(2)})^* \{p1\}$$

$$\{p1\} : E.place = T^{(1)}.place$$

Η παραπάνω γραμματική αντιστοιχεί σε κώδικα. Ο συμβολισμός `{p1}` σημαίνει ότι στον κώδικα του συντακτικού αναλυτή, μετά το τέλος του βρόχου `while` που εκφράζει το άστρο του Kleene, πρέπει να τοποθετήσουμε την εκχώρηση του  $T^{(1)}$  στην E.

Προχωρούμε ένα βήμα περισσότερο και πάμε να εξετάσουμε την περίπτωση που ο κανόνας αναγνωρίζει ένα ακριβώς τερματικό σύμβολο, το +. Αφού αναγνωριστεί το +, θα κληθεί ο κανόνας  $T^{(2)}$  ο οποίος θα επιστρέψει τη μεταβλητή  $T^{(2)}.\text{place}$  ως αποτέλεσμα. Έχουμε λοιπόν το  $T^{(1)}.\text{place}$  από τον  $T^{(1)}$  και το  $T^{(2)}.\text{place}$  από τον  $T^{(2)}$ . Πρέπει αυτά τα δύο να προστεθούν. Το αποτέλεσμα θα πρέπει να εκχωρηθεί σε μία μεταβλητή που θα επιστραφεί ως αποτέλεσμα από την E, δηλαδή στην E.place.

Πρέπει, λοιπόν, να παραχθεί μία νέα τετράδα η οποία, όταν θα εκτελεστεί, θα προσθέτει τη μεταβλητή που βρίσκεται στο  $T^{(1)}.\text{place}$  και τη μεταβλητή που βρίσκεται στο  $T^{(2)}.\text{place}$  και θα εκχωρεί το αποτέλεσμα σε μία μεταβλητή που δεν θα επηρεάσει τη λειτουργία του προγράμματος. Δεν μπορούμε δηλαδή να χρησιμοποιήσουμε μία ήδη υπάρχουσα μεταβλητή ή μία μεταβλητή που πιθανόν να χρησιμοποιηθεί αργότερα. Άρα θα επιστρατεύσουμε μία προσωρινή μεταβλητή.

Ας τα συμβολίσουμε όλα αυτά στη γραμματική, όπου θα καλέσουμε την `genQuad()` στο κατάλληλο σημείο:

$$E \rightarrow T^{(1)} (+ T^{(2)} \{p1\})^* \{p2\}$$

$$\begin{aligned} \{p1\} : & w = \text{newTemp}() \\ & \text{genQuad}('+', T^{(1)}.place, T^{(2)}.place, w) \\ & T^{(1)}.place = w \\ \{p2\} : & E.place = T^{(1)}.place \end{aligned}$$

Στο τέλος του `{p1}` τοποθετήσαμε τη νέα μεταβλητή στο  $T^{(1)}.\text{place}$ , αφού τελικά αυτό είναι που στο `{p2}` θα χρησιμοποιηθεί ως αποτέλεσμα. Θα μπορούσαμε να είχαμε κάνει την τοποθέτηση απευθείας στο E.place, αλλά, όπως θα δούμε στη συνέχεια, αυτό δεν βολεύει στην περίπτωση που ο κανόνας τελικά θα αναγνωρίσει περισσότερες από μία προσθέσεις. Αφού βεβαιωθήκαμε ότι κατανοήσαμε ότι το σχέδιο ενδιάμεσου κώδικα λειτουργεί όταν αναγνωρίζει ακριβώς μία πρόσθεση, πάμε να διαπιστώσουμε ότι λειτουργεί σωστά και για την περίπτωση που θα αναγνωριστούν περισσότερες από μία προσθέσεις.

Ας θεωρήσουμε την παράσταση `a+b+c`. Το a θα επιστραφεί από το  $T^{(1)}$  και το b από το  $T^{(2)}$ . Η πρόσθεση τους θα δημιουργήσει την τετράδα `+, a, b, T_1` και το `T_1` θα τοποθετηθεί στο  $T^{(1)}.\text{place}$ . Στον επόμενο

κύκλο, που περιγράφεται από το αστεράκι του Kleene, το  $T^{(2)}.place$  θα πάρει την τιμή c. Θα δημιουργηθεί η τετράδα +, T\_1, c, T\_2 και το T\_2 είναι αυτό που θα τοποθετηθεί στο  $T^{(1)}.place$ . Δεν υπάρχει άλλος κύκλος, αφού δεν αναγνωρίζεται άλλο +. Βγαίνουμε από το αστεράκι του Kleene και εκτελείται το {p2}, το οποίο θέτει ως αποτέλεσμα την τελευταία τιμή του T\_1, δηλαδή την τελευταία προσωρινή μεταβλητή που δημιουργήθηκε, η οποία πράγματι κρατάει τη μεταβλητή που περιέχει το αποτέλεσμα της παράστασης. Άρα η γραμματική μας είναι σωστή, τουλάχιστον για την περίπτωση που αναγνωρίζονται δύο προσθέσεις στον κανόνα.

Είναι πολύ εύκολο να διαπιστώσουμε ότι η παραπάνω λογική λειτουργεί για οποιονδήποτε αριθμό προσθέσεων. Κάθε φορά που εμφανίζεται νέα πρόσθεση, δημιουργούμε μία νέα μεταβλητή και τοποθετούμε εκεί το άθροισμα του μέχρι στιγμής αποτελέσματος και του νέου όρου που εμφανίστηκε μετά το +. Αν ο κανόνας ολοκληρωθεί, τότε έχουμε τελειώσει και η τελευταία προσωρινή μεταβλητή αποτελεί το αποτέλεσμα που θα επιστραφεί από τον κανόνα, σύμφωνα με {p2}. Αν όχι, θα επαναλαμβάνεται ο ίδιος κύκλος μέχρι να εξαντληθούν οι προσθέσεις τις οποίες ο κανόνας μπορεί να αναγνωρίσει.

Παρακάτω δίνεται ολοκληρωμένος ο ενδιάμεσος κώδικας που παράγεται από την παράσταση του παραδείγματος: (a+b+c).

```
100: +, a, b, T_1
101: +, T_1, c, T_2
```

Περνώντας στον δεύτερο κανόνα της γραμματικής:

```
# multiplication
T → F(1) (* F(2))*
```

μας περιμένει μια ευχάριστη έκπληξη. Η λογική του είναι ακριβώς ίδια με τον κανόνα της πρόσθεσης. Άλλαζον μόνο τα τερματικά και τα μη τερματικά σύμβολα. Χωρίς να χρειάζεται να κουραστούμε πολύ, μετατρέπουμε κατάλληλα τον προηγούμενο κανόνα και έχουμε:

```
T → F(1) (* F(2) {p1})* {p2}
```

```
{p1} : w = newTemp()
genQuad('*', F(1).place, F(2).place, w)
F(1).place = w
{p2} : T.place = F(1).place
```

Οι ευχάριστες εκπλήξεις συνεχίζονται και στον επόμενο κανόνα:

```
# priority
F → ( E )
```

Παρόλο που μοιάζει, και είναι, ένας σημαντικός κανόνας, το σημαντικό μέρος της ύπαρξής του απορροφάται από τον συντακτικό αναλυτή. Εκεί υλοποιείται η προτεραιότητα των πράξεων, ενώ εδώ περιοριζόμαστε στο να διαχειριστούμε τα δεδομένα που έρχονται από τον κανόνα E, τα οποία και δεν πρέπει να χαθούν. Το E.place περιέχει τη μεταβλητή με το αποτέλεσμα των πράξεων που έχουν δημιουργηθεί από τον κανόνα αυτόν και όσους βρίσκονται κάτω από αυτόν στο συντακτικό δέντρο. Αυτή η ίδια μεταβλητή είναι που θα αποτελέσει το F.place, αφού καμία αριθμητική πράξη δεν αναγνωρίζεται και δεν δημιουργείται από τον κανόνα F.

Συνεπώς, το σχέδιο ενδιάμεσου κώδικα για τον κανόνα F είναι το ακόλουθο:

```
F → ( E ) {p1}
```

```
{p1} : F.place = E.place
```

Τέλος, παραθέτουμε το σχέδιο ενδιάμεσου κώδικα για τον κανόνα που αναγνωρίζει τα τερματικά σύμβολα.

```
F → ID
```

Το μόνο που πρέπει να κάνει ο κανόνας είναι να μεταφέρει το τερματικό σύμβολο που αναγνώρισε στη μεταβλητή F.place. Το σχέδιο ενδιάμεσου κώδικα για τον κανόνα F ακολουθεί:

```
F → ID {p1}
{p1} : F.place = ID.place
```

Ολοκληρώνοντας, παραθέτουμε ένα παράδειγμα μετατροπής μίας αριθμητικής παράστασης σε ενδιάμεσο κώδικα.

Έστω η παράσταση:  $a+b*c+a*(b+c)$ . Ο κώδικας που αντιστοιχεί στην παράσταση αυτή είναι ο ακόλουθος:

```
100: *, b, c, T_1
101: +, a, T_1, T_2
102: +, b, c, T_3
103: *, a, T_3, T_4
104: +, T_2, T_4, T_5
```

Η παράσταση θα επιστρέψει το T\_5 ως αποτέλεσμα στον κανόνα από τον οποίο ενεργοποιήθηκε, δηλαδή το E.place θα έχει την τιμή “T\_5”.

Δεν θα αναλύσουμε βήμα βήμα πώς φτάσαμε ως εδώ. Θα το αφήσουμε ως άσκηση στον αναγνώστη, αφού είναι αρκετά απλό.

Θα δώσουμε άλλο ένα παράδειγμα, επίσης χωρίς αναλυτικό σχολιασμό. Θα δημιουργήσουμε τον ενδιάμεσο κώδικα για την παράσταση  $3+(c*(a+b)*d)$ :

```
100: +, a, b, T_1
101: *, c, T_1, T_2
102: *, T_2, d, T_3
103: +, 3, T_3, T_4
```

Το E.place θα επιστρέψει το T\_4.

Συνοψίζοντας, ας παραθέσουμε ολοκληρωμένο και συγκεντρωμένο το σχέδιο ενδιάμεσου κώδικα για τις αριθμητικές παραστάσεις:

```
E → T(1) ( + T(2) {p1} )* {p2}
{p1} : w = newTemp()
       genQuad( '+', T(1).place, T(2).place, w)
       T(1).place = w
{p2} : E.place = T(1).place

T → F(1) ( * F(2) {p1} )* {p2}
{p1} : w = newTemp()
       genQuad( '*', F(1).place, F(2).place, w)
       F(1).place = w
{p2} : T.place = F(1).place

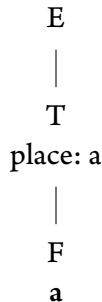
F → ( E ) {p1}
{p1} : F.place = E.place

F → ID {p1}
{p1} : F.place = ID.place
```

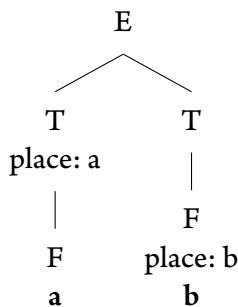
Θα δώσουμε ένα ακόμα παράδειγμα, αρκετά αναλυτικό αυτή τη φορά. Έστω ότι θέλουμε να μετατρέψουμε σε ενδιάμεσο κώδικα την παράσταση:

$a+b^*(c+d+1)$

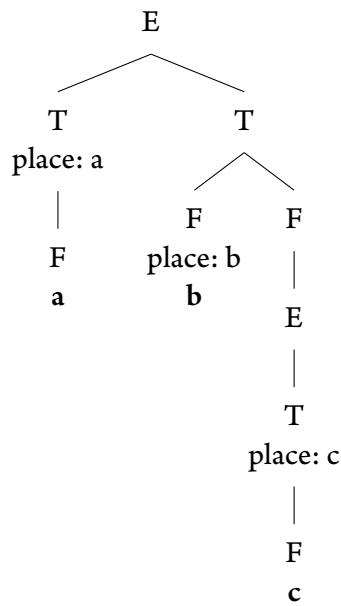
Η συντακτική ανάλυση θα εκκινήσει από τον κανόνα E. Θα κληθεί ο κανόνας T και στη συνέχεια ο κανόνας F, ώστε να αναγνωριστεί το a. Το F.place θα πάρει την τιμή a, η οποία θα περάσει και στο T.place. Έτσι, τη στιγμή αυτή το δέντρο που έχει σχηματιστεί είναι το ακόλουθο:



Το επόμενο σύμβολο που θα αναγνωριστεί στην είσοδο είναι το +, το οποίο θα αναγνωριστεί μέσα στον κανόνα E. Θα δημιουργηθεί ένα νέο T, το οποίο με τη σειρά του θα καλέσει το F, ώστε να φτάσουμε στο τερματικό σύμβολο b. Το δέντρο τώρα έχει ως εξής:



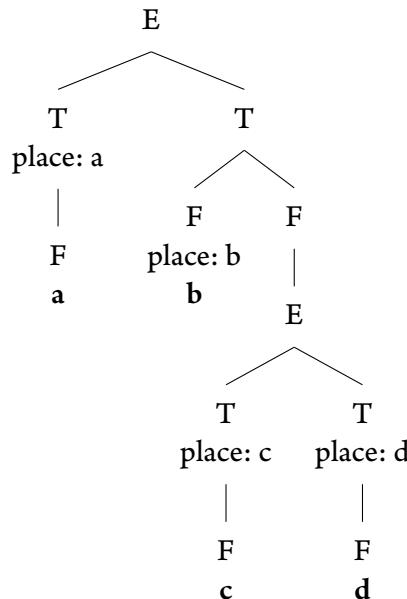
Στο σημείο αυτό το \* δημιουργεί νέο F, ενώ η παρένθεση που το ακολουθεί θα δημιουργήσει E. Η ανάλυση θα συνεχιστεί, το E θα δημιουργήσει T και το T ένα νέο F το οποίο θα αναγνωρίσει το c. Το F.place θα περάσει το c στο T.place, οπότε η εικόνα του δέντρου τώρα είναι:



Σε κάθε κόμβο του δέντρου φαίνεται ο κανόνας που ενεργοποιήθηκε στο σημείο αυτό. Όταν είμαστε σε φύλλο σημειώνεται το τερματικό σύμβολο που αναγνωρίστηκε. Στα σημεία που υπάρχει ενδιαφέρον φαίνεται η τιμή της μεταβλητής place. Η τιμή της μεταβλητής place δεν σημειώνεται σε όλους τους κόμβους,

για να μην υπερφορτωθεί το δέντρο και να μπορεί κανείς εύκολα να εντοπίσει το σημείο ενδιαφέροντος για το κάθε σχήμα. Ας επιστρέψουμε στη διαδικασία διαπέρασης και δημιουργίας του συντακτικού δέντρου.

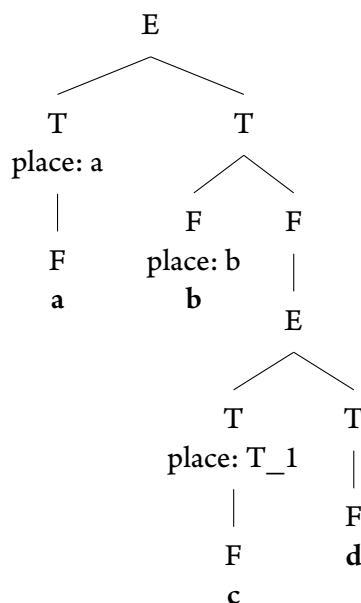
Το τερματικό σύμβολο + που ακολουθεί θα δημιουργήσει έναν νέο όρο (κανόνας T), δίπλα στο τελευταίο μη τερματικό σύμβολο T και με την ίδια ακολουθία κλήσεων και μεταφοράς αποτελεσμάτων από κανόνα σε κανόνα, θα φέρει το d στο T.place. Το δέντρο τώρα θα γίνει:



Είναι αξιοσημείωτο ότι μέχρι στιγμής δεν έχει δημιουργηθεί καμία τετράδα ενδιάμεσου κώδικα. Στον κανόνα E έχουν δημιουργηθεί δύο T τα οποία έχουν ανάμεσά τους ένα τερματικό σύμβολο +. Εδώ θα δημιουργηθεί η πρώτη τετράδα κώδικα:

100: +, c, d, T\_1

Η προσωρινή μεταβλητή που θα δημιουργηθεί, σύμφωνα με το σχέδιο ενδιάμεσου κώδικα, θα τοποθετηθεί στο T\_1.place, ώστε να χρησιμοποιηθεί από έναν πιθανό νέο όρο που θα εμφανιστεί ή για να επιστραφεί ως E.place. Έτσι το δέντρο τώρα γίνεται:

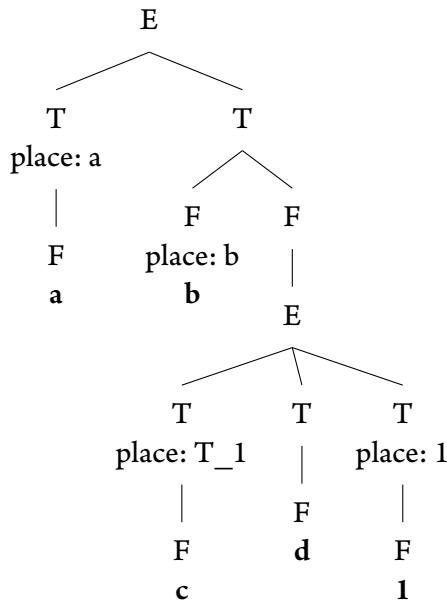


Στη συνέχεια ακολουθεί το επόμενο τερματικό σύμβολο +, το οποίο θα δημιουργήσει την ίδια σειρά κλήσεων και επιστροφών τιμών, ώστε το τερματικό σύμβολο 1, να ανέβει και να τοποθετηθεί στο T.place.

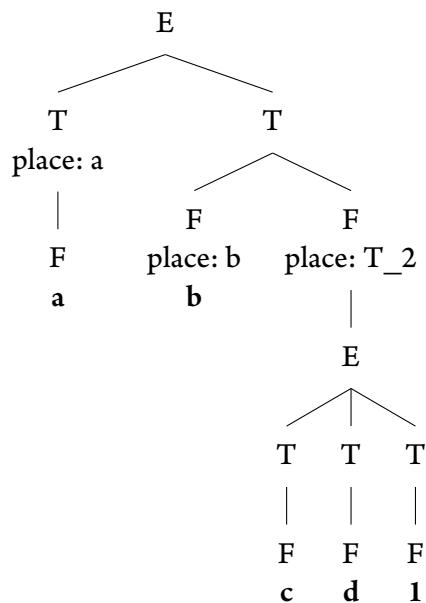
Στο σημείο αυτό θα παραχθεί νέα τετράδα ενδιάμεσου κώδικα, η:

101: +, T\_1, 1, T\_2

Το δέντρο μετά και την αναγνώριση του τελευταίου τερματικού συμβόλου (του 1) γίνεται:



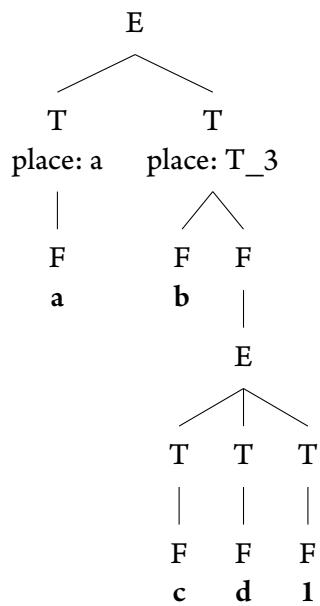
Το T\_2 είναι το αποτέλεσμα που θα μεταφερθεί στο E και στη συνέχεια στο F, όπως φαίνεται παρακάτω:



Η νέα εντολή ενδιάμεσου κώδικα θα πολλαπλασιάσει τα place από τα δύο F και θα δημιουργήσει το αποτέλεσμα για το T. Δημιουργείται, λοιπόν, η τετράδα:

102: \*, b, T\_2, T\_3

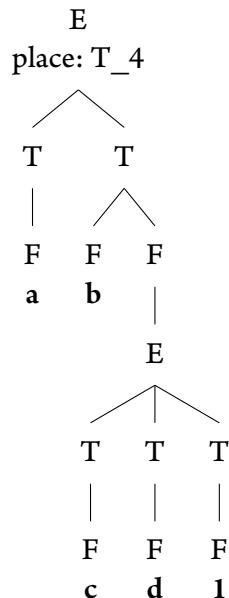
και το δέντρο γίνεται:



Τέλος, με τον ίδιο ακριβώς τρόπο θα παραχθεί η τετράδα που θα κάνει την πρόσθεση στο ανώτερο επίπεδο:

103: +, a, T\_3, T\_4

Και θα σχηματιστεί και το place για όλη την παράσταση, η οποία θα διαβιβαστεί στη δομή που ανήκει η αριθμητική παράσταση, ώστε να τη χρησιμοποιήσει:



και ο κώδικας που παρήγθη είναι ο ακόλουθος:

100: +, c, d, T\_1  
 101: +, T\_1, 1, T\_2  
 102: \*, b, T\_2, T\_3  
 103: +, a, T\_3, T\_4

## 6.4 Λογικές συνθήκες

Σε ένα πρόγραμμα αρχικής γλώσσας μία λογική παράσταση μπορεί να εμφανιστεί είτε στον δεξί μέλος μίας εικώρησης σε μία λογική μεταβλητή είτε μέσα σε μία λογική συνθήκη. Η πρώτη περίπτωση δεν εμφανίζεται μέσα σε ένα πρόγραμμα *C-impe*. Η δεύτερη μπορεί να εμφανιστεί σε πολλές περιπτώσεις, για παράδειγμα στις εντολές *if* ή *while*.

Οι λογικές παραστάσεις έχουν αρκετές ομοιότητες και αρκετές διαφορές, όσον αφορά τον τρόπο που τις χειρίζεται ένας μεταγλωττιστής προκειμένου να παραγάγει τον ενδιάμεσο κώδικα. Είδαμε ότι στις αριθμητικές παραστάσεις, κάθε κανόνα τον χειριζόμαστε ανεξάρτητα από τους υπόλοιπους, θεωρώντας ότι οι κανόνες που αυτός ενεργοποιεί στο δεξί του μέλος θα παραγάγουν τον ενδιάμεσο κώδικα που απαιτείται, έτσι ώστε, όταν ο ενδιάμεσος κώδικας εκτελεστεί, το αποτέλεσμα της παράστασης που αντιστοιχεί σε αυτόν να τοποθετηθεί σε μία συγκεκριμένη μεταβλητή. Η μεταβλητή αυτή είναι το αποτέλεσμα του κανόνα το οποίο θα περαστεί στον κανόνα που τον ενεργοποίησε. Κάθε κανόνας, λοιπόν, συγκεντρώνει τις μεταβλητές που λαμβάνει ως αποτέλεσμα από τους κανόνες που ενεργοποιεί, παράγει τον ενδιάμεσο κώδικα που αντιστοιχεί στον κανόνα αυτόν, συνδυάζοντας τις τιμές των μεταβλητών που έλαβε ως αποτέλεσμα, και επιστρέφει και αυτός ως δικό του αποτέλεσμα τη μεταβλητή που έχει το αποτέλεσμα των πράξεων που εκείνος παρήγαγε.

Στις λογικές παραστάσεις έχουμε πάλι κανόνες που μπορούμε να θεωρήσουμε ότι είναι ανεξάρτητοι μεταξύ τους. Έτσι, θα πρέπει να διαχειριστούμε τα αποτελέσματα από τους κανόνες που θα ενεργοποιήσει ο κάθε κανόνας και να δημιουργήσουμε το νέο αποτέλεσμα. Όμως τώρα, τα αποτελέσματα δεν είναι μεταβλητές όπως πριν. Επίσης, μόνο σε έναν κανόνα παράγεται ενδιάμεσος κώδικας, ενώ δύο οι υπόλοιποι κανόνες διαχειρίζονται πληροφορία και την επεξεργάζονται πριν περάσουν το δικό τους αποτέλεσμα προς τα πάνω. Τι είναι όμως αυτή η πληροφορία που κινείται από κανόνα σε κανόνα και δεν είναι μεταβλητές;

Ας χρησιμοποιήσουμε ένα παράδειγμα, για να κάνουμε λίγο πιο απλά τα πράγματα. Δίνεται η παρακάτω έκφραση σε *C-impe*. Αυτή μπορεί να αποτελεί το condition ενός *if* ή ενός *while*. Ο κανόνας που θα αναγνωρίσει την έκφραση δεν το γνωρίζει αυτό. Μπορεί μόνο να κάνει συντακτική ανάλυση στην έκφραση:

`a > b and c > d or e > f`

Φυσικά ισχύουν οι κανόνες προτεραιότητας για τα *and* και *or*. Δεν θα σας είναι δύσκολο να διαπιστώσετε ότι ο παραπάνω κώδικας είναι ισοδύναμος με τον παρακάτω:

```
100: >, a, b, 102
101: jump, _, _, 104
102: >, c, d, ...
103: jump, _, _, 104
104: >, e, f, ...
105: jump, _, _, ...
```

Δεν θα μας απασχολήσει ακόμα το πώς βγήκε ο κώδικας αυτός, μας αρκεί που γνωρίζουμε ότι οι δύο παραπάνω κώδικες είναι ισοδύναμοι. Παρατηρήστε ότι τα λογικά άλματα στις εντολές 102,104,105 δεν έχουν συμπληρωθεί. Αυτό δεν συμβαίνει διότι κάνουμε άλμα στο πουθενά, αλλά γιατί δεν γνωρίζουμε ακόμα πού πρέπει να γίνει αυτό το άλμα. Αν η λογική συνθήκη είναι μέρος μίας *if*, το 104 πρέπει να συμπληρωθεί με την πρώτη εκτελέσιμη εντολή μέσα στο σώμα της *if*. Αν υπάρχει *else*, τότε οι 102 και 105 πρέπει να κάνουν άλμα στην πρώτη εντολή που περικλείεται από το *else* και, εάν δεν υπάρχει *else*, στην πρώτη εκτελέσιμη εντολή έξω από τη δομή της *if*. Αν η λογική συνθήκη είναι μέρος μίας *while*, τότε το 104 πρέπει να συμπληρωθεί με την πρώτη εκτελέσιμη εντολή μέσα στο σώμα της *while*, ενώ οι 102 και 105 πρέπει να κάνουν άλμα στην πρώτη εκτελέσιμη εντολή έξω από τη δομή της *while*. Το σίγουρο είναι ότι τη στιγμή αυτή δεν γνωρίζουμε πώς θα συμπληρωθούν οι τετράδες αυτές, αφού δεν γνωρίζουμε τι υπάρχει παραπάνω. Η λύση στο πρόβλημα αυτό είναι η πληροφορία αυτή να περάσει ως αποτέλεσμα στον κανόνα που ενεργοποίησε τον κανόνα της λογικής συνθήκης και εκείνος να διαχειριστεί τις τετράδες αυτές, να τις συμπληρώσει δηλαδή κατάλληλα με τα σημεία στα οποία πρέπει να γίνουν τα λογικά άλματα.

Το θέμα αυτό δεν περιορίζεται στον κανόνα που βρίσκεται υψηλότερα στην ιεραρχία ενός δέντρου κανόνων που αποτιμούν λογικές παραστάσεις, στον κανόνα δηλαδή που καλείται από μία `if`, `while`, κλπ. Η ίδια ανάγκη παρουσιάζεται σε καθέναν από τους κανόνες που συμμετέχουν στην αποτίμηση μιας λογικής έκφρασης. Έτσι, όταν ο μεταγλωττιστής συνάντησε την πρώτη σύγκριση μέσα στην παράσταση του παραδείγματός μας (`a>b`), τότε παρήγαγε τις εντολές 100 και 101. Οι εντολές 100 και 101 στο σημείο εκείνο ήταν ασυμπλήρωτες. Ακόμα δεν έχει διαβαστεί η υπόλοιπη παράσταση για να γνωρίζει η μεταγλωττιση που θα τοποθετηθούν οι εντολές που τελικά τοποθετήθηκαν στα 102 και 104. Ο κανόνας αυτός πρέπει να επιστρέψει ως αποτέλεσμα τις τετράδες που έχουν μείνει ασυμπλήρωτες, έτσι ώστε να συμπληρωθούν αργότερα, από άλλους κανόνες, όταν γνωρίζουμε το πού πρέπει να γίνει το λογικό άλμα. Κάθε κανόνας που καλείται λαμβάνει, αλλά και προετοιμάζει για τον κανόνα που τον κάλεσε, δύο λίστες:

- Τη λίστα `true` η οποία αποτελείται από όλες εκείνες τις τετράδες, που έχουν μείνει ασυμπλήρωτες διότι ο κανόνας αδυνατεί να συμπληρώσει. Οι τετράδες πρέπει να συμπληρωθούν με την ετικέτα εκείνης της τετράδας στην οποία πρέπει να μεταβεί ο έλεγχος αν η λογική συνθήκη ισχύει.
- Τη λίστα `false` η οποία αποτελείται από όλες εκείνες τις τετράδες οι οποίες έχουν μείνει ασυμπλήρωτες διότι ο κανόνας αδυνατεί να συμπληρώσει. Οι τετράδες πρέπει να συμπληρωθούν με την ετικέτα εκείνης της τετράδας στην οποία πρέπει να μεταβεί ο έλεγχος αν η λογική συνθήκη δεν ισχύει.

Ας θυμηθούμε τους κανόνες που σχετίζονται με την αναγνώριση και αποτίμηση λογικών εκφράσεων:

```
# boolean expression
B → Q ( or Q )*
# term in boolean expression
Q → R ( and R )*
# factor in boolean expression
R → not [ B ]
| [ B ]
| E rel_op E
```

Όπως κάναμε και στις αριθμητικές εκφράσεις, θα μελετήσουμε τους κανόνες έναν έναν και θα εκμεταλλευτούμε την ανεξαρτησία τους. Ο πρώτος εξ αυτών περιγράφει τη συνένωση λογικών συνθηκών με τον τελεστή `or`.

$$B \rightarrow Q^{(1)} ( \text{ or } Q^{(2)} )^*$$

Ο `B` ενεργοποιεί τους κανόνες `Q` και διαχειρίζεται τις ασυμπλήρωτες τετράδες που έρχονται σημειωμένες στις λίστες `Q(1).true`, `Q(1).false`, `Q(2).true`, `Q(2).false`. Πρέπει να συμπληρώσει όσες από αυτές μπορεί και να δημιουργήσει τις λίστες `B.true`, `B.false`, για να μεταφέρει στον κανόνα που τον ενεργοποίησε, όσες από αυτές δεν μπόρεσε να συμπληρώσει ή όσες μη συμπληρωμένες τετράδες χρειάστηκε να δημιουργήσει αυτός.

Θα σκεφτούμε με έναν τρόπο ανάλογο που σκεφτήκαμε στον κανόνα `E`. Ας θεωρήσουμε ότι η ενεργοποίηση του κανόνα δεν θα αναγνωρίσει κανένα `and`. Τότε ο κανόνας `Q(2)`, δεν θα ενεργοποιηθεί ποτέ και η `B` θα έχει να διαχειριστεί μόνο τις `Q(1).true`, `Q(1).false`, τις οποίες θα περάσει ανέπαφες στις αντίστοιχες `B.true`, `B.false`. Άρα η γραμματική μέχρι στιγμής γίνεται:

$$B \rightarrow Q^{(1)} \{p1\} ( \text{ or } Q^{(2)} )^*$$

$$\{p1\} : B.\text{true} = Q^{(1)}.true \\ B.\text{false} = Q^{(1)}.false$$

Ας θεωρήσουμε τώρα την περίπτωση που ο κανόνας αναγνωρίζει ένα `or`. Για να εμφανίζεται ένα `or` σημαίνει ότι η συνθήκη που ακολουθεί το `or` θα αποτιμηθεί εάν η συνθήκη πριν το `or` αποτύχει. Γνωρίζουμε ότι οι τετράδες που βρίσκονται μέσα στη λίστα `Q(1).false` πρέπει να συμπληρωθούν με την τετράδα που

Θα δημιουργηθεί στη συνέχεια. Βέβαια, λόγω της  $\{p1\}$ , αντί για την  $Q^{(1)}$ .`false` μπορεί να χρησιμοποιηθεί και η `B.false`, αφού είναι ίδιες. Θα προτιμήσουμε τη δεύτερη λύση, η οποία θα μας διευκολύνει παρακάτω. Η ετικέτα της τετράδας που θα δημιουργηθεί αμέσως μετά, μας δίνεται με την κλήση της `nextquad()`, ενώ η κλήση που θα χρησιμοποιήσουμε για να συμπληρωθούν οι τετράδες είναι η `backpatch()`. Συμπληρώνοντας με τα παραπάνω το σχέδιο ενδιάμεσου κώδικα, έχουμε:

`B → Q(1) {p1} ( or {p2} Q(2) )*`

```
{p1} : B.true = Q(1).true
      B.false = Q(1).false
{p2} : backpatch(B.false, nextquad())
```

Όταν ολοκληρωθεί η αναγνώριση από τον κανόνα  $Q^{(2)}$ , τότε θα έχουμε ακόμα δύο λίστες που πρέπει να χειριστούμε, την  $Q^{(2)}$ .`true` και την  $Q^{(2)}$ .`false`. Οι τετράδες από τις λίστες αυτές δεν μπορούν ακόμα να συμπληρωθούν. Από τη στιγμή που θεωρούμε ότι υπάρχει μόνο ένα `or` το οποίο θα αναγνωριστεί, οι τετράδες αυτές πρέπει να εισαχθούν στις λίστες `B.true` και `B.false` για να μπορέσουν να συμπληρωθούν σε επόμενο στάδιο της μεταγλώττισης. Ποια λίστα όμως θα πάει πού; Η λίστα  $Q^{(2)}$ .`true` περιέχει τετράδες οι οποίες θα κάνουν λογικό άλμα στο ίδιο σημείο που θα κάνουν άλμα και οι τετράδες της  $Q^{(1)}$ .`true` ή της `B.true`, αφού είπαμε ότι είναι η ίδια λίστα, λόγω του  $\{p1\}$ . Είναι λογικό, άλλωστε. Όλες οι τετράδες που προέρχονται από τις συνθήκες που χωρίζονται με το `or` πρέπει να κάνουν άλμα στο ίδιο σημείο, όταν μία από αυτές αποτιμήθει ως αληθής. Άρα συνενώνουμε την `B.true` με την  $Q^{(2)}$ .`true`. Η `B.false` πρακτικά δεν υπάρχει, αφού μόλις έγινε `backpatch()`. Η `B.false` δεν μπορεί παρά να αποτελείται από τις τετράδες που έχουν απομείνει και που πρέπει να κάνουν άλμα στο σημείο που θέλουμε να μεταβεί ο κώδικας, όταν η συνθήκη που εξετάζει ο συγκεκριμένος κανόνας δεν ισχύει. Αυτή διατηρείται στη λίστα  $Q^{(2)}$ .`false`. Άρα, αρκεί να μετονομάσουμε την  $Q^{(2)}$ .`false` σε `B.false`. Μετά και από αυτές τις παρατηρήσεις, το σχέδιο ενδιάμεσου κώδικα γίνεται:

`B → Q(1) {p1} ( or {p2} Q(2) {p3} )*`

```
{p1} : B.true = Q(1).true
      B.false = Q(1).false
{p2} : backpatch(B.false, nextquad())
{p3} : B.true = mergeList(B.true, Q(2).true)
      B.false = Q(2).false
```

Μας απομένει να δούμε τι συμβαίνει όταν έχουμε περισσότερα από ένα `or`. Στην περίπτωση αυτή ενεργοποιείται το αστεράκι του Kleene. Σε κάθε βήμα, έχοντας ως βάση τις λίστες `B.true` και `B.false` από την προηγούμενη επανάληψη, κάνουμε `backpatch()` τη λίστα `B.false`, για τον ίδιο λόγο που σχολιάσαμε παραπάνω και κάνουμε `mergeList()` τις νέες τετράδες που θα έρθουν από το  $Q^{(2)}$ .`true`. Έτσι, η λίστα `B.true` πληθαίνει, χωρίς να αλλάζει ο σκοπός της. Το σχέδιο ενδιάμεσου κώδικα, όπως φαίνεται παραπάνω, δεν απαιτεί κάποια τροποποίηση για να λειτουργήσει για την περίπτωση που ο κανόνας αναγνωρίζει περισσότερα του ενός `or`.

Ένα παράδειγμα εκτέλεσης ακολουθεί. Έστω ο κώδικας:

`a>b or c>d or e>f`

Ο ισοδύναμος κώδικας σε ενδιάμεση γλώσσα είναι ο ακόλουθος:

```
100: >, a, b, ...
101: jump, _, _, 102
102: >, c, d, ...
103: jump, _, _, 104
104: >, e, f, ...
105: jump, _, _, ...
```

Όταν ο μεταγλωττιστής συναντήσει το `a>b` θα δημιουργήσει τις εντολές 100 και 101. Δεν έχουμε δει πως θα γίνει αυτό, θα το δούμε παρακάτω. Αυτό που μας ενδιαφέρει τώρα είναι ότι αυτές οι τετράδες είναι ασυμπλήρωτες. Έτσι έχουμε:

```
B.true = Q(1).true = [100]
B.false = Q(1).false = [101]
```

Στη συνέχεια, μόλις ο μεταγλωττιστής συναντήσει το `or`, τότε αμέσως γνωρίζει ότι η 101 μπορεί να συμπληρωθεί με την τετράδα η οποία θα δημιουργηθεί αμέσως μετά, αφού εκεί πρέπει να μεταβεί ο έλεγχος. Χρησιμοποιώντας την `backpatch()` τη συμπληρώνει. Άρα για τις λίστες `B` τώρα έχουμε:

```
B.true = [100]
B.false = []
```

και η τετράδα 101 είναι συμπληρωμένη.

Στη συνέχεια ακολουθεί η αναγνώριση της `c>d`. Τότε ενεργοποιείται το `{p2}` και οι τετράδες που ήρθαν από την `Q(2)` ενσωματώνονται στις `B`.

```
B.true = mergeList(B.true,Q(2).true) = [100, 102]
B.false = Q(2).false = [103]
```

Και ο κύκλος επαναλαμβάνεται για μία ακόμη φορά. Μόλις ο μεταγλωττιστής συναντήσει το `or`, τότε αμέσως γνωρίζει ότι η 103 μπορεί να συμπληρωθεί με την τετράδα η οποία θα δημιουργηθεί αμέσως μετά. Άρα για τις λίστες `B` τώρα έχουμε:

```
B.true = [100, 102]
B.false = []
```

και η τετράδα 103 είναι συμπληρωμένη.

Στη συνέχεια ακολουθεί η αναγνώριση της `e>f`. Τότε ενεργοποιείται πάλι το `{p2}` και οι τετράδες που ήρθαν από την `Q(2)` ενσωματώνονται στις `B`.

```
B.true = mergeList(B.true,Q(2).true) = [100, 102, 104]
B.false = Q(2).false = [105]
```

Εδώ τελειώνει η αρμοδιότητα του κανόνα. Οι λίστες `B.true` και `B.false` είναι αδύνατον να συμπληρωθούν από τον κανόνα αυτόν και θα περαστούν στον κανόνα που τον ενεργοποίησε ως αποτέλεσμα, έτσι ώστε να συμπληρωθούν σε μεταγενέστερο στάδιο.

Κάπου εδώ τελειώσαμε με τον πρώτο κανόνα. Ο δεύτερος από τους κανόνες που χρησιμοποιούνται για την αναγνώριση λογικών εκφράσεων είναι αυτός που διαχειρίζεται τα `and`:

$Q \rightarrow R \ ( \text{and} \ R )^*$

Δεν θα αποτελούσε έκπληξη αν διαπιστώναμε ότι ο κανόνας αυτός και ο κανόνας που αναλύσαμε πριν από λίγο, για τα `or`, έχουν πολλές ομοιότητες. Για την ακρίβεια, το σημείο στο οποίο διαφέρουν είναι η λειτουργία των `and` και `or`. Ενώ, όταν έχουμε σε μία έκφραση το `and`, πρέπει να ισχύουν και οι δύο συνθήκες που βρίσκονται δεξιά και αριστερά του ώστε να ισχύει ο συνδυασμός τους, στο `or` αρκεί να ισχύει τουλάχιστον η μία. Άρα, όταν αποτυγχάνει μία συνθήκη που βρίσκεται αριστερά ενός `or`, τότε ο έλεγχος μεταβαίνει στη συνθήκη δεξιά του `or`. Όταν κρίνεται αληθής μία συνθήκη που βρίσκεται αριστερά ενός `or`, τότε ο έλεγχος μεταβαίνει έξω από τη συνθήκη, διότι δεν απαιτούνται περισσότεροι έλεγχοι. Αντίθετα, όταν αποτυγχάνει μία συνθήκη που βρίσκεται αριστερά ενός `and`, τότε ο έλεγχος μεταβαίνει έξω από τη συνθήκη, διότι δεν απαιτούνται περισσότεροι έλεγχοι. Όταν κρίνεται αληθής μία συνθήκη που βρίσκεται αριστερά ενός `and`, τότε ο έλεγχος μεταβαίνει στη συνθήκη δεξιά του `and`.

Αν δούμε πιο προσεκτικά την επίπτωση που έχει αυτό στο σχέδιο του ενδιάμεσου κώδικα, θα δούμε ότι υπάρχει μια εναλλαγή στον ρόλο που είχαν οι λίστες `B` και τώρα έχουν οι λίστες `Q`. Δηλαδή η λίστα που πρέπει να γίνει `backpatch()` είναι η λίστα `Q.true`, ενώ η λίστα που πρέπει να συσσωρεύσει τις τετράδες

που τελικά θα μας οδηγήσουν εκτός της λογικής συνθήκης, όταν αυτή δεν ισχύει, είναι η Q.false. Λαμβάνοντας υπόψη τα παραπάνω και κατ' αναλογία με το τι ακολουθήσαμε στον κανόνα με τα or, το σχέδιο ενδιάμεσου κώδικα για τον κανόνα των and θα είναι:

$$\begin{aligned} Q \rightarrow & R^{(1)} \{p1\} (\text{ or } \{p2\} R^{(2)} \{p3\})^* \\ \{p1\} : & Q.\text{true} = R^{(1)}.\text{true} \\ & Q.\text{false} = R^{(1)}.\text{false} \\ \{p2\} : & \text{backpatch}(Q.\text{true}, \text{nextquad}()) \\ \{p3\} : & Q.\text{false} = \text{mergeList}(Q.\text{false}, R^{(2)}.\text{false}) \\ & Q.\text{true} = R^{(2)}.\text{true} \end{aligned}$$

Ας περάσουμε στον επόμενο κανόνα:

$$\begin{aligned} R \rightarrow & \text{not } [B] \\ | & [B] \\ | & E \text{ rel\_op } E \end{aligned}$$

Πρακτικά εδώ έχουμε τρεις κανόνες. Ας ξεκινήσουμε με τον τελευταίο:

$$R \rightarrow E^{(1)} \text{ rel\_op } E^{(2)}$$

Πρόκειται για τον χαμηλότερο κανόνα, με την έννοια ότι περιγράφει τη σύγκριση δύο αριθμητικών εκφράσεων, δεν περιέχει δηλαδή συνδυασμό κάποιων λογικών εκφράσεων. Είναι και ο μόνος κανόνας ο οποίος δημιουργεί ενδιάμεσο κώδικα. Έτσι, λοιπόν, όταν συναντάμε τη σύγκριση δύο αριθμητικών εκφράσεων, πρέπει να παραγάγουμε την τετράδα που θα πραγματοποιεί το λογικό άλμα στην περίπτωση που η συνθήκη ισχύει, όπως και στην περίπτωση που δεν ισχύει. Έτσι, θα παραχθούν οι εξής τετράδες:

$$\begin{aligned} x: & \text{ rel\_op, } E^{(1)}.\text{place}, E^{(2)}.\text{place}, \dots \\ x+1: & \text{ jump, } \_, \_, \dots \end{aligned}$$

Η πρώτη τετράδα θα εκτελεστεί αν η συνθήκη ισχύει. Στην περίπτωση αυτή θα μεταβεί ο έλεγχος στην ετικέτα που ορίζει η τετράδα αυτή. Αν η συνθήκη αποτιμηθεί στο ψευδές, τότε το άλμα στην εντολή με την ετικέτα x δεν θα εκτελεστεί και ο έλεγχος θα μεταβεί στην επόμενη εντολή. Η επόμενη εντολή είναι αυτή με το jump. Το jump εδώ παίζει τον ρόλο του else σε μία δομή if, αφού εκτελείται όταν η συνθήκη δεν ισχύει.

Τόσο η τετράδα με το λογικό άλμα, όσο και η τετράδα με το άλμα θα μείνουν ασυμπλήρωτες. Από την πληροφορία ότι δύο αριθμητικές παραστάσεις θα συγκριθούν δεν προκύπτει και το πού θα μεταβεί ο έλεγχος στην περίπτωση που η σύγκριση θα δώσει ως αποτέλεσμα το αληθές ή το ψευδές. Έτσι, οι ετικέτες των δύο τετράδων θα περάσουν ως αποτέλεσμα του κανόνα R, μέσω των λιστών R.true και R.false, αντίστοιχα, ώστε να συμπληρωθούν σε υψηλότερο επίπεδο.

Οι λίστες R.true και R.false δεν προέρχονται από κάποιον άλλον κανόνα, όπως ίσχυε στους κανόνες B και R. Οι λίστες αυτές δημιουργούνται εδώ, αφού εδώ δημιουργούνται και οι τετράδες των αλμάτων. Για τον σκοπό αυτόν έχουμε προδιαγράψει την makeList(). Η makeList() πρέπει να κληθεί πριν την genQuad() και να πάρει ως παράμετρο το nextQuad(). Με τον τρόπο αυτόν θα δημιουργηθεί μία νέα λίστα, μοναδικό στοιχείο της οποίας θα είναι η ετικέτα της τετράδας που θα δημιουργηθεί αμέσως μετά και, φυσικά, θα πρόκειται για μία μη συμπληρωμένη τετράδα.

Το σχέδιο ενδιάμεσου κώδικα, που υλοποιεί τα παραπάνω, είναι το ακόλουθο:

$$\begin{aligned} R \rightarrow & E^{(1)} \text{ rel\_op } E^{(2)} \{p1\} \\ \{p1\}: & R.\text{true} = \text{makeList(nextQuad())} \\ & \text{genQuad(rel\_op, } E^{(1)}.\text{place}, E^{(2)}.\text{place}\}, \text{ '_'} \\ & R.\text{false} = \text{makeList(nextQuad())} \\ & \text{genQuad('jump', } '_', '_', '_') \end{aligned}$$

Ο επόμενος κανόνας που θα εξετάσουμε είναι ο:

$R \rightarrow [ B ]$

Είναι ανάλογος με τον κανόνα του ορισμού προτεραιοτήτων με τις παρενθέσεις στις αριθμητικές εκφράσεις. Στον κανόνα αυτόν δεν υπάρχει ανάγκη να δημιουργηθεί ενδιάμεσος κώδικας. Η συντακτική ανάλυση είναι αυτή που θα φροντίζει για την ορθή τήρηση. Οι τετράδες που έρχονται ασυμπλήρωτες από την  $B$  θα μεταφερθούν ως έχουν στην  $R$ . Άρα το σχέδιο ενδιάμεσου κώδικα που υλοποιεί τη μεταφορά αυτή είναι:

$R \rightarrow [ B ] \{p1\}$

```
{p1} : R.true = B.true
      R.false = B.false
```

Ολοκληρώνουμε με τον κανόνα:

$R \rightarrow \text{not } [ B ]$

Η διαφοροποίησή του από τον προηγούμενο κανόνα, που δεν περιέχει το  $\text{not}$ , έγκειται στο ότι όταν το  $B$  επιστρέφει αληθές, τότε το  $R$  πρέπει να επιστρέφει ψευδές. Όταν το  $B$  επιστρέφει ψευδές, τότε το  $R$  πρέπει να επιστρέφει αληθές. Άρα οι τετράδες που πρέπει να συμπληρωθούν με την ετικέτα στην οποία πρέπει να γίνει το λογικό άλμα όταν ισχύει ο  $B$ , είναι οι τετράδες που πρέπει συμπληρωθούν με την ετικέτα στην οποία πρέπει να γίνει το λογικό άλμα όταν δεν ισχύει ο  $R$ . Οι τετράδες από τη λίστα  $\text{true}$  του  $B$  γίνονται οι τετράδες της λίστας  $\text{false}$  του  $R$ . Αντίστοιχα, οι τετράδες από τη λίστα  $\text{false}$  του  $B$  γίνονται οι τετράδες της λίστας  $\text{true}$  του  $R$ . Το σχέδιο ενδιάμεσου κώδικα ακολουθεί:

$R \rightarrow \text{not } [ B ] \{p1\}$

```
{p1} : R.true = B.false
      R.false = B.true
```

Συνοψίζοντας, ας ενώσουμε το σχέδιο ενδιάμεσου κώδικα που κατασκευάσαμε για όλους τους κανόνες των λογικών συνθηκών σε ένα ενιαίο σχέδιο, ώστε να το έχουμε συγκεντρωμένο και ολοκληρωμένο:

$B \rightarrow Q^{(1)} \{p1\} (\text{ or } Q^{(2)})^*$

```
{p1} : B.true = Q^{(1)}.true
      B.false = Q^{(1)}.false
```

$Q \rightarrow R^{(1)} \{p1\} (\text{ or } \{p2\} R^{(2)} \{p3\})^*$

```
{p1} : Q.true = R^{(1)}.true
      Q.false = R^{(1)}.false
{p2} : \text{backpatch}(Q.true, \text{nextquad}())
{p3} : Q.false = \text{mergeList}(Q.false, R^{(2)}.false)
      Q.true = R^{(2)}.true
```

$R \rightarrow E^{(1)} \text{ rel\_op } E^{(2)} \{p1\}$

```
{p1} : R.true = \text{makeList}(\text{nextQuad}())
      \text{genQuad}(\text{rel\_op}, E^{(1)}.place, E^{(2)}.place), '_'
      R.false = \text{makeList}(\text{nextQuad}())
      \text{genQuad}('jump', '_', '_', '_')
```

$R \rightarrow [ B ] \{p1\}$

```
{p1} : R.true = B.true
      R.false = B.false
```

$R \rightarrow \text{not } [ B ] \{p1\}$

```
{p1} : R.true = B.false
      R.false = B.true
```

Ας δούμε τώρα αναλυτικά, βήμα βήμα, ένα παράδειγμα μετατροπής μίας λογικής έκφρασης σε ενδιάμεσο κώδικα. Έστω η λογική έκφραση:

```
a>b or a>c and (b>c or a>1) and b<>1
```

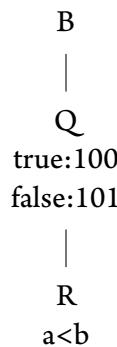
Η μεταγλώττιση θα ξεκινήσει από τον κανόνα B, ο οποίος θα καλέσει τον Q και στη συνέχεια θα κληθεί ο R ο οποίος (με τη βοήθεια του E, αλλά ας μην προχωρήσουμε περισσότερο σε βάθος) θα αναγνωρίσει το a>b. Θα δημιουργηθούν οι τετράδες:

```
100: <, a, b, _
101: jump, _, _, _
```

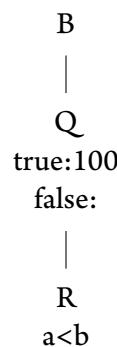
οι οποίες θα είναι ασυμπλήρωτες, άρα θα τοποθετηθούν στις λίστες true, false

```
true: [100]
false: [101]
```

για το R, οι οποίες θα περάσουν στο Q. Παρακάτω φαίνεται το δέντρο συντακτικής ανάλυσης, όπως έχει σχηματιστεί μέχρι τώρα, διακοσμημένο με τις ιδιότητες true και false



Όταν ο συντακτικός αναλυτής συναντήσει το or, τότε γνωρίζει ότι η false λίστα του Q πρέπει να γίνει backpatch() στην επόμενη τετράδα που θα δημιουργηθεί. Έτσι, η εικόνα του δέντρου θα γίνει:



ενώ ο κώδικας:

```
100: <, a, b, _
101: jump, _, _, 102
```

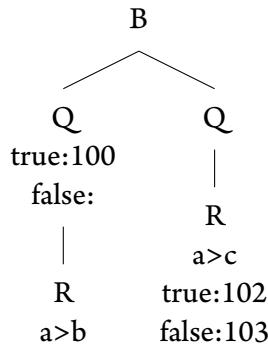
Στη συνέχεια θα ξεκινήσει η αναγνώριση του Q που ακολουθεί το or, μέσα στο οποίο θα κληθεί ο κανόνας R και θα γίνει η αναγνώριση του a>c. Θα δημιουργηθεί ο ενδιάμεσος κώδικας:

```
102: <, a, c, _
103: jump, _, _, _
```

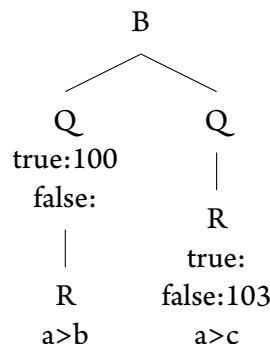
Οι τετράδες είναι ασυμπλήρωτες, άρα θα τοποθετηθούν στις λίστες true, false του R:

```
true: [102]
false: [103]
```

Το δέντρο τώρα γίνεται:



Στη συνέχεια ο μεταγλωττιστής συναντά το and, ενώ βρίσκεται ακόμα μέσα στον κανόνα Q. Το σχέδιο ενδιάμεσου κώδικα θα τον οδηγήσει στο να κάνει backpatch() το true του R στην επόμενη τετράδα που θα δημιουργηθεί. Άρα το δέντρο γίνεται:



και ο κώδικας:

```
100: <, a, b, _
101: jump, _, _, 102
102: <, a, c, 104
103: jump, _, _, _
```

Μετά το and ενεργοποιείται ο κανόνας R, ο οποίος λόγω της παρένθεσης που ακολουθεί το and, θα ενεργοποιήσει τον κανόνα B, εκείνος πάλι τον Q και με τη σειρά του τον R, ο οποίος και θα αναγνωρίσει την επόμενη σύγκριση b>c.

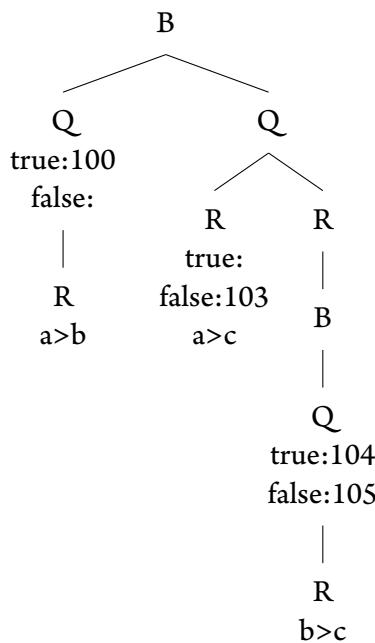
Θα δημιουργηθεί ο ενδιάμεσος κώδικας:

```
104: >, b, c, γίνεται_
105: jump, _, _, _
```

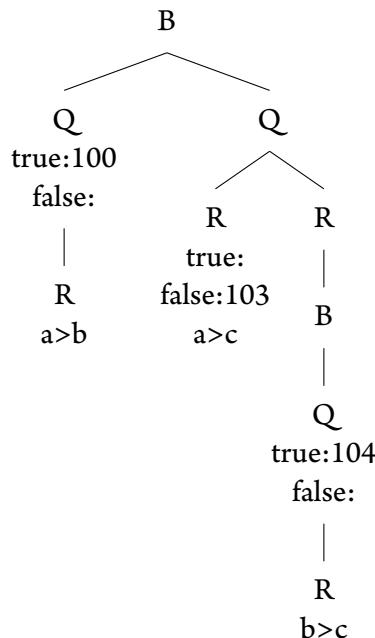
Οι παραπάνω τετράδες θα είναι ασυμπλήρωτες, άρα θα τοποθετηθούν στις λίστες true, false του R:

```
true: [104]
false: [105]
```

Οι λίστες από το R θα περάσουν στο Q. Δεν εξερχόμαστε ακόμα από τον κανόνα B. Το δέντρο τη στιγμή αυτή έχει γίνει ως εξής:



Στη συνέχεια ο μεταγλωττιστής συναντά το or. Ο συντακτικός αναλυτής, θυμίζοντες, βρίσκεται μέσα στο B, όπου μετά το or αναμένεται να αναγνωρίσει ακόμα ένα Q. Έτσι, το false του Q θα κάνει backpatch() στο nextquad() και το δέντρο θα γίνει:



και ο κώδικας:

```

100: <, a, b, _
101: jump, _, _, 102
102: <, a, c, 104
103: jump, _, _, -
104: >, b, c, _
105: jump, _, _, 106
  
```

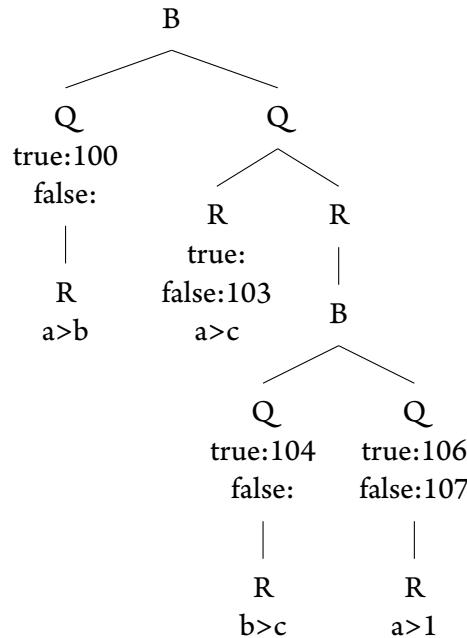
Μέτα αναγνωρίζεται το νέο Q, καλώντας, φυσικά, το R. Αναγνωρίζεται η σύγκριση a>1, δημιουργείται ο κώδικας:

```
106: >, a, 1, _
107: jump, _, _, _
```

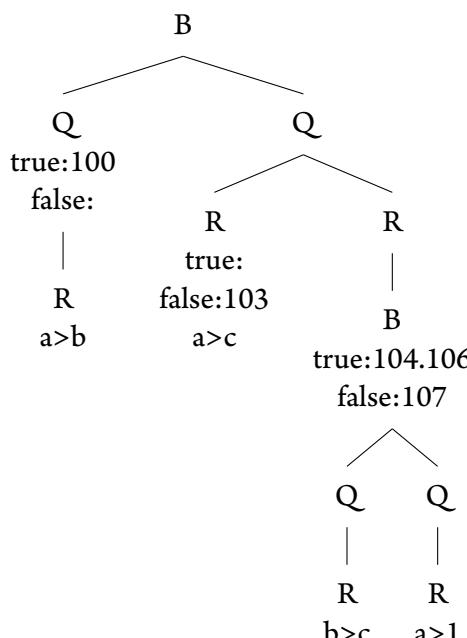
και οι ασυμπλήρωτες τετράδες μπαίνουν στις λίστες:

```
true: [106]
false: [107]
```

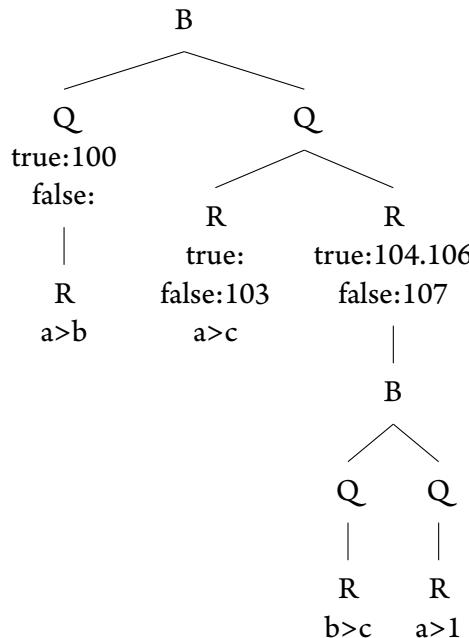
Το δέντρο τώρα έχει γίνει ως εξής:



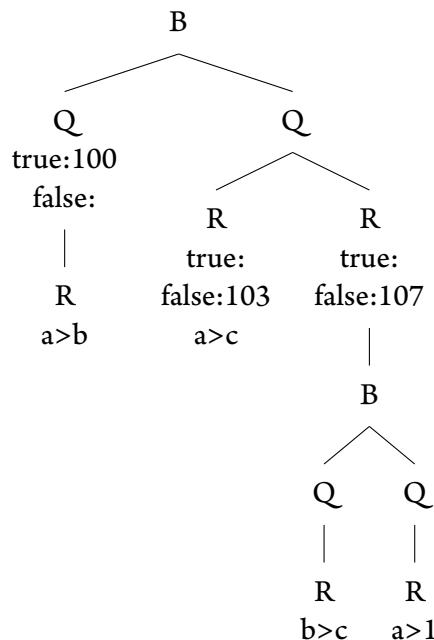
Ο κανόνας B έχει ολοκληρωθεί. Οι δύο κανόνες Q του έχουν δημιουργήσει τρεις λίστες τις οποίες πρέπει να διαχειριστεί. Μπόρεσε και έκανε backpatch() στην περίπτωση του false της πρώτης Q, αλλά τώρα πρέπει να συνδυνάσει τις τρεις λίστες σε δύο, ώστε να τις περάσει στον κανόνα που τον ενεργοποίησε, όπου και θα διαχειριστούν περαιτέρω. Κοιτώντας το σχέδιο ενδιάμεσου κώδικα που φτιάξαμε, αλλά και επιστρατεύοντας τη λογική, θα ενώσουμε τις δύο true λίστες σε μία και θα μεταφέρουμε την Q.false στην B.false. Έτσι, το δέντρο θα γίνει:



Οι δύο λίστες από το B θα μεταβιβαστούν στο R:



Η επόμενη σύγκριση που θα συναντήσει ο μεταγλωττιστής είναι το and έξω από την παρένθεση. Αυτό θα προκαλέσει ένα backpatch() στο true του R που μόλις ολοκληρώθηκε. Φυσικά το backpatch() θα γίνει στο nextQuad() ώστε να στοχεύσει τη συνθήκη μετά το and. Άρα το δέντρο θα γίνει:



και ο κώδικας:

```

100: <, a, b, _
101: jump, _, _, 102
102: <, a, c, 104
103: jump, _, _, -
104: >, b, c, 108
105: jump, _, _, 106
106: >, a, 1, 108
107: jump, _, _, -
  
```

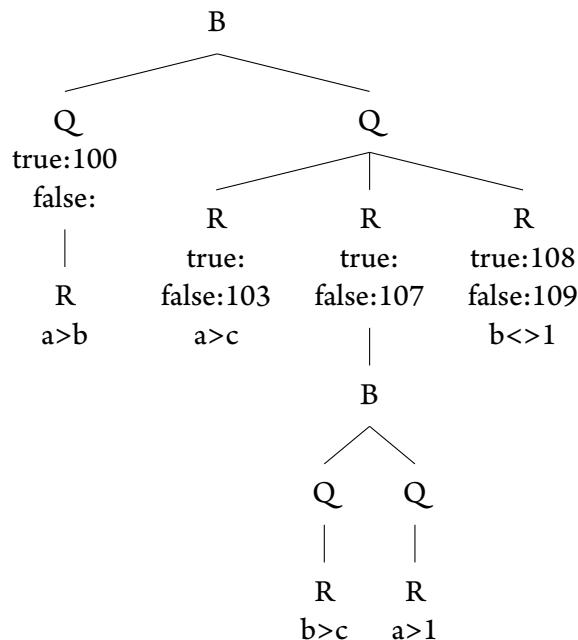
Στη συνέχεια θα αναγνωριστεί μέσω ενός νέου R το  $b <> 1$  και θα δημιουργηθεί ο κώδικας:

108: =, b, 1, \_  
109: jump, \_, \_, \_

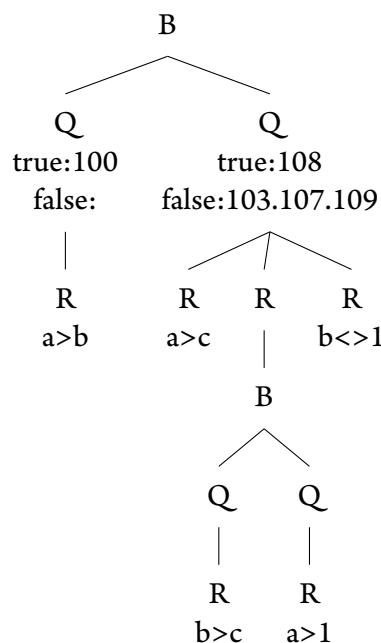
και οι λίστες:

true: [108]  
false: [109]

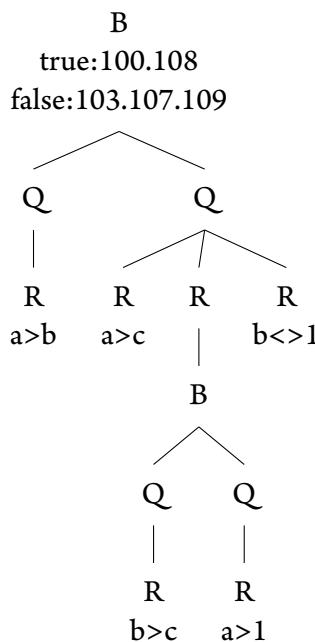
ενώ το δέντρο γίνεται:



Σε αυτό το σημείο ολοκληρώνεται ο κανόνας Q, οπότε από τις λίστες των τριών R, δημιουργούνται οι δύο λίστες για τα Q, κάνοντας `mergeList()` στην περίπτωση των false που είναι τρεις και μεταφέροντας την true. Έτσι το δέντρο γίνεται:



Τέλος, από τις λίστες των δύο Q θα πάρουμε τις λίστες του B:



Οι δύο αυτές λίστες θα μεταφερθούν στον κανόνα που ενεργοποίησε τον B, ώστε να διαχειριστούν εκεί, ενώ ο κώδικας που δημιουργήθηκε μέσα από τη διαπέραση της λογικής συνθήκης είναι ο ακόλουθος:

```

100: <, a, b, _                      #true
101: jump, _, _, 102
102: <, a, c, 104
103: jump, _, _, _                     #false
104: >, b, c, 108
105: jump, _, _, 106
106: >, a, 1, 108
107: jump, _, _, _                     #false
108: =, b, 1, _                         #true
109: jump, _, _, _                     #false
  
```

## 6.5 Αρχή και τέλος ενότητας

Στο κεφάλαιο που ορίσαμε τη γλώσσα του ενδιάμεσου κώδικα, αναφερθήκαμε σε δύο εντολές, την begin\_block και την end\_block. Οι δύο αυτές εντολές περικλείουν τον ενδιάμεσο κώδικα που δημιουργείται από μία συνάρτηση ή μία διαδικασία ή από το κυρίως πρόγραμμα.

Οι εντολές αυτές δεν επιτρέπεται να είναι εμφωλευμένες. Δηλαδή, πρέπει κάθε begin\_block να τερματίζεται με end\_block πριν ανοίξει μία νέα ενότητα με ένα begin\_block.

Αν, για παράδειγμα, έχουμε το ακόλουθο πρόγραμμα σε C-imple:

```

program fibonacci_numbers
  ...
  function fibonacci(in x)
  {
    ...
  }
  
```

```
# main #
{
    ...
}.
```

Η σειρά που θα εμφανιστούν στον ενδιάμεσο κώδικα τα `begin_block` και `end_block` είναι η εξής:

```
xxx : begin_block, fibonacci, _, _
      ... code for fibonacci
xxx : end_block, fibonacci, _, _
xxx : begin_block, main_fibonacci_numbers, _, _
      ... code for main
xxx : end_block, main_fibonacci_numbers, _, _
```

Παρατηρήστε τη σειρά με την οποία εμφανίστηκαν. Δεν είναι η σειρά εμφάνισης των συναρτήσεων, αλλά η σειρά με την οποία εμφανίστηκε το τμήμα των εκτελέσιμων εντολών τους. Και το γεγονός ότι η `fibonacci` είναι εμφωλευμένη μέσα στη `fibonacci_numbers` δεν οδήγησε σε εμφωλευμένα `begin_block` και `end_block`.

## 6.6 Είσοδος και έξοδος δεδομένων

Το ίδιο απλές με την `return` είναι οι δύο εντολές εισόδου-έξόδου: η `input` και η `print`. Και οι δύο δημιουργούν μία εντολή ενδιάμεσου κώδικα. Ή:

```
input (x)
```

αντιστοιχεί στη δημιουργία της:

```
in, _, _, _
```

και η:

```
print (x)
```

αντιστοιχεί στη δημιουργία της:

```
out, _, _, _
```

## 6.7 Τερματισμός εκτέλεσης

Η τελευταία εντολή ενός προγράμματος πρέπει να είναι η `halt`. Με την

```
halt, _, _, _
```

τερματίζεται το πρόγραμμα και επιστρέφεται ο χώρος στο λειτουργικό σύστημα. Η `halt` βοηθάει και στο να είμαστε βέβαιοι ότι όταν παράγουμε κώδικα για μία δομή και κάνουμε `backpatch()` έξω από αυτήν, θα υπάρξει μία εντολή η οποία θα δημιουργηθεί, ακόμα κι αν φαινομενικά η δομή που μεταφράζουμε είναι η τελευταία του προγράμματος.

Η `halt` εμφανίζεται μόνο στο κυρίως πρόγραμμα. Τοποθετείται πριν από το `end_block`.

## Βιβλιογραφία

- [1] Keith D. Cooper και Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2004. ISBN: 1558606998.

- [2] Keith D. Cooper και Linda Torczon. *Σχεδίαση και Κατασκευή Μεταγλωττιστών*. Ξενόγλωσσος τίτλος: Engineering a Compiler, Επιστημονική επιμέλεια έκδοσης: Γεώργιος Μανής, Νικόλαος Παπασπύρου και Αντώνιος Σαββίδης. Πανεπιστημιακές Εκδόσεις Κρήτης, 2018. ISBN: 9789605245191.
- [3] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 0321486811.
- [4] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Μεταγλωττιστές: Αρχές, τεχνικές και εργαλεία, 2η αμερικανική έκδοση*. Επιστημονική επιμέλεια έκδοσης: Π. Αλεφραγκής, Α. Βώρος, Ν. Βώρος, Κ. Μασσέλος. Εκδόσεις Νέων Τεχνολογιών, 2011. ISBN: 9789606759727.
- [5] Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.

## ΚΕΦΑΛΑΙΟ 7

---

# ΕΝΔΙΑΜΕΣΟΣ ΚΩΔΙΚΑΣ ΓΙΑ ΤΙΣ ΔΟΜΕΣ ΤΗΣ ΓΛΩΣΣΑΣ, ΣΥΝΑΡΤΗΣΕΙΣ ΚΑΙ ΔΙΑΔΙΚΑΣΙΕΣ

---

### Σύνοψη:

Μετά τη μετατροπή των αριθμητικών και λογικών παραστάσεων σε μία γλώσσα τετράδων, ακολουθεί η μετατροπή των δομών ελέγχου σε απλούστερες. Οι πολύπλοκες δομές μιας τυπικής γλώσσας δομημένου προγραμματισμού αποσυντίθενται σε απλές δομές, στις οποίες κυριαρχούν απλά και υπό συνθήκη άλματα.

Έτσι, στο κεφάλαιο αυτό θα περιγραφούν με λεπτομέρεια οι ενέργειες που σχετίζονται με τη μετατροπή δομών σε ενδιάμεση γλώσσα, όπως η εντολή απόφασης `if`, η εντολή επανάληψης `while` και η εντολή επιλογής `switchcase`. Θα μελετήσουμε και περισσότερο ενδιαφέρουσες στην υλοποίηση, αν και λιγότερο χρήσιμες στον προγραμματισμό, δομές, όπως οι εντολές απόφασης-επανάληψης `forcase` και `incase`.

Στη συνέχεια ακολουθεί η μετάφραση σε ενδιάμεση αναπαράσταση των συναρτήσεων και των διαδικασιών. Για τις συναρτήσεις και τις διαδικασίες δεν επιτυγχάνεται στη φάση αυτή ανάλογα σημαντική αποσύνθεση των δομών τους. Οι κλήσεις τους μετατρέπονται κατάλληλα, ώστε να συμβολιστούν σε ενδιάμεση γλώσσα. Η περαιτέρω αποδόμηση της πολυπλοκότητάς τους μεταφέρεται για τη φάση της παραγωγής του τελικού κώδικα.

Ένα ολοκληρωμένο, απλό παράδειγμα μετατροπής κώδικα σε ενδιάμεση αναπαράσταση θα παρουσιαστεί με αναλυτικά βήματα, ώστε να καλύψει πιθανά σημεία που τυχόν δυσκόλεψαν.

Τέλος, θα συζητήσουμε το πώς θα μπορούσαμε, με βάση τις αρχές του αντικειμενοστραφούς προγραμματισμού, να ιεραρχήσουμε πιθανές κλάσεις που εμφανίζονται στη σχεδίαση του μεταγλωττιστή.

### Προαπαιτούμενη γνώση:

- κεφάλαιο 1
  - κεφάλαιο 2
  - κεφάλαιο 3
  - κεφάλαιο 5
  - κεφάλαιο 6
- 

Η μετάφραση των δομών της γλώσσας είναι ένα ιδιαίτερα ενδιαφέρον σημείο, αφού σε αυτό διασπώνται πολύπλοκες δομές, κοντινές στον ανθρώπινο τρόπο σκέψης, σε απλές εντολές αλλαγής της ροής ελέγχου πολύ περισσότερο κοντινές στον τρόπο με τον οποίο το υλικό επιθυμεί να συνομιλούμε μαζί του. Έτσι, μία ανθρώπινη σκέψη:

όσο πεινάω τρώω

πρέπει να μετασχηματιστεί στο αρκετά λιγότερο φιλικό για εμάς:

έλεγχος πείνας: πεινάω?

αν όχι πηγαίνω στο "δεν πεινάω"  
τρώω μία μπουκιά  
πηγαίνω στον "έλεγχο πείνας"

δεν πεινάω: τέλος καλό.

Για τις συναρτήσεις και τις διαδικασίες δεν έχουμε στη φάση αυτή να επιδείξουμε τόσα πολλά επιτεύγματα. Ο μόνος μετασχηματισμός που εφαρμόζουμε περιλαμβάνει τον συμβολισμό της κλήση τους σε ενδιάμεση γλώσσα και απλά μεταθέτουμε το πρόβλημα για τη φάση του τελικού κώδικα, όπου εκεί έχουμε ό,τι εργαλεία χρειαζόμαστε για την πλήρη μεταγλώττιση σε γλώσσα μηχανής.

Περισσότερο υλικό για την παραγωγή ενδιάμεσου κώδικα μπορείτε να αναζητήσετε στα ακόλουθα κεφάλαια στη βιβλιογραφία: [1, 2, κεφ.5], [3, 4, κεφ.6], [5, κεφ.7].

## 7.1 Οι δομές της γλώσσας

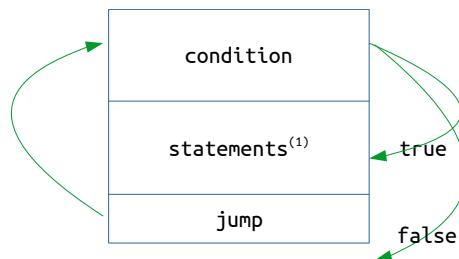
Ας ξεκινήσουμε, λοιπόν, με τη μετάφραση των δομών της γλώσσας και ας αφήσουμε για τη συνέχεια τις συναρτήσεις και τις διαδικασίες. Θα δούμε παρακάτω πώς θα κατασκευάσουμε σχέδιο ενδιάμεσου κώδικα για τις τρεις βασικές δομές: τη δομή επανάληψης `while`, τη δομή απόφασης `if` και τη δομή επιλογής `switch-case`. Ιδιαίτερο ενδιαφέρον έχουν και οι δύο ακόμα δομές της γλώσσας, η `incase` και η `forcase`, ενδιαφέροντα κυρίως για την υλοποίησή τους και όχι για τις προγραμματιστικές τους δυνατότητες. Ας ξεκινήσουμε με την πιο απλή.

### 7.1.1 Η δομή επανάληψης "while"

Η γραμματική `while` είναι η ακόλουθη:

`whileStat` → `while ( condition ) statements`

Η `condition` είναι μία λογική συνθήκη η οποία θα επιστρέψει τις δύο γνωστές λίστες με τα `condition.true` και `condition.false`. Οι ασυμπλήρωτες τετράδες της `condition.true` πρέπει να συμπληρωθούν με την πρώτη τετράδα των `statements`, διότι εκεί πρέπει να μεταβεί ο έλεγχος όταν η συνθήκη ισχύει. Οι ασυμπλήρωτες τετράδες της `condition.false` πρέπει να συμπληρωθούν με την πρώτη τετράδα της επόμενης εντολής, δηλαδή έξω από την `while`, διότι εκεί πρέπει να μεταβεί ο έλεγχος όταν η



Σχήμα 7.1: Παραγωγή ενδιάμεσου κώδικα για το while.

συνθήκη δεν ισχύει. Άρα ακριβώς πριν την statements πρέπει να τοποθετηθεί η backpatch(), με παράμετρο το nextQuad() για τη λίστα condition.true. Επίσης, ακριβώς στο τέλος, πρέπει να τοποθετηθεί η backpatch(), για τη λίστα condition.false. Ενσωματώνουμε τα παραπάνω στη γραμματική και έχουμε:

`whileStat → while ( condition ) {p1}  
statements {p2}`

`{p1} : backpatch(condition.true, nextQuad())  
{p2} : backpatch(condition.false, nextQuad())`

Πέρα από τη διαχείριση των δύο λιστών, για να λειτουργήσει το while χρειάζεται κάτι ακόμα. Όταν εκτελούνται οι εντολές των statements πρέπει ο έλεγχος να μεταβαίνει στην αρχή της λογικής συνθήκης, ώστε αυτή να επανεξετάζεται. Για να μπορέσει να γίνει αυτό, πρέπει να γίνουν δύο πράγματα. Το πρώτο είναι, την ώρα που δημιουργείται η πρώτη τετράδα της συνθήκης, η επικέτα της να σημειώνεται. Επειδή η δημιουργία της τετράδας θα γίνει μέσα στο condition και επειδή εμείς υλοποιούμε τώρα τον κανόνα για το while, θα σημειώσουμε την τετράδα αυτή ακριβώς πριν καλέσουμε την condition, εκμεταλλευόμενοι την nextQuad(). Ας το σημειώσουμε στη γραμματική:

`whileStat → while {p0} ( condition ) {p1}  
statements {p2}`

`{p0} : condQuad = nextQuad()  
{p1} : backpatch(condition.true, nextQuad())  
{p2} : backpatch(condition.false, nextQuad())`

Τέλος πρέπει να φροντίσουμε να δημιουργηθεί και η τετράδα η οποία θα κάνει το άλμα από το τέλος των statements στην αρχή της condition, δηλαδή στο condQuad. Αυτή πρέπει να είναι πριν το backpatch() που βρίσκεται στο {p2}, αφού το backpatch() είπαμε ότι πρέπει να είναι η τελευταία ενέργεια στο σχέδιο του ενδιάμεσου κώδικα. Το σχέδιο ολοκληρωμένο ακολουθεί:

`whileStat → while {p0} ( condition ) {p1}  
statements {p2}`

`{p0} : condQuad = nextQuad()  
{p1} : backpatch(condition.true, nextQuad())  
{p2} : genQuad('jump', '_', '_', condQuad)  
backpatch(condition.false, nextQuad())`

Αν θέλουμε σχηματικά να δούμε πώς λειτουργεί ο ενδιάμεσος κώδικας, μπορούμε να το κάνουμε με το σχήμα 7.1. Στο σχήμα 7.1 φαίνεται η σειρά με την οποία τοποθετήθηκαν τα τμήματα προγράμματος στη μνήμη, το πώς θα συμπληρωθούν οι λίστες true και false και η ανάγκη και το πώς θα υλοποιηθεί το άλμα για να επανεκτιμηθεί η λογική συνθήκη.

Στο σημείο αυτό ας κάνουμε κάποιες παρατηρήσεις πάνω στο σχέδιο του ενδιάμεσου κώδικα. Η πρώτη

αφορά αυτό το τελευταίο σημείο που συζητήσαμε. Τι θα συνέβαινε αν οι εντολές μέσα στο {p2} τοποθετούνταν με αντίθετη σειρά, δηλαδή πρώτα το backpatch() και μετά το genQuad(); Στην περίπτωση αυτή, το backpatch() θα συμπληρωνόταν με την ετικέτα της τετράδας που θα δημιουργήσει το nextQuad(). Άρα, όταν η condition αποτιμηθεί ως false, τότε το άλμα θα γίνει πάνω στην jump η οποία θα μεταφέρει τον έλεγχο στην αρχή της condition. Η condition θα αποτιμηθεί πάλι σε false, αφού δεν άλλαξε τίποτε από την προηγούμενη αποτίμησή της και θα έχουμε, έτσι, έναν ατέρμονο βρόχο. Σίγουρα δεν είναι αυτό που θέλαμε.

Η δεύτερη αφορά το σημείο που τοποθετήσαμε το {p0}. Αν το τοποθετούσαμε μετά το άνοιγμα της παρένθεσης, θα πείραζε; Η απάντηση είναι όχι. Είτε ο συντακτικός αναλυτής βρίσκεται πριν το άνοιγμα της παρένθεσης είτε μετά, η επόμενη τετράδα που θα δημιουργηθεί θα είναι η πρώτη τετράδα της condition. Αυτήν θέλουμε να στοχεύσουμε με το nextQuad() και το επιτυγχάνουμε και με τις δύο θέσεις.

Η τρίτη παρατήρηση έχει να κάνει με το statements. Αλήθεια, αν το statements είναι κενό, το σχέδιο ενδιάμεσου κώδικα εξακολουθεί να λειτουργεί σωστά; Για να δούμε τι θα συμβεί αν το statements είναι κενό. Το backpatch() στη λίστα true της condition θα συμπλήρωνε τις ασυμπλήρωτες τετράδες με την επόμενη εντολή που θα δημιουργηθεί, η οποία είναι το άλμα που θα επιστρέψει τον έλεγχο στην αρχή της συνθήκης. Η συνθήκη θα επαναπατιμηθεί ως αληθής και θα σχηματιστεί πάλι ένας ατέρμονος βρόχος. Όμως, αυτή τη φορά μας πειράζει; Μάλλον όχι, ή μάλλον καλύτερα, είναι αυτό που θα θέλαμε να συμβαίνει.

Τέλος, ας δώσουμε ένα παράδειγμα μετατροπής αρχικού κώδικα σε ενδιάμεσο, το οποίο να επικεντρώνει στη δομή while:

```
while (a>b)
    b:=b+1
```

Ξεκινώντας τη μετάφραση, θα εκτελεστεί το {p0} και θα σημειωθεί στη μεταβλητή condQuad η ετικέτα της πρώτης τετράδα της condition. Έστω ότι αυτή είναι το 100. Στη συνέχεια θα παραχθούν οι δύο τετράδες σύγκρισης από την condition:

```
100: >, a, b, _
101: jump, _, _, _
```

καθώς και οι λίστες condition.true=[100], condition.false=[101].

Το επόμενο που θα δραστηριοποιηθεί είναι το {p2}, το οποίο θα συμπληρώσει τα στοιχεία της λίστας true με το nextQuad(), δηλαδή το 102. Ο κώδικας γίνεται:

```
100: >, a, b, 102
101: jump, _, _, _
```

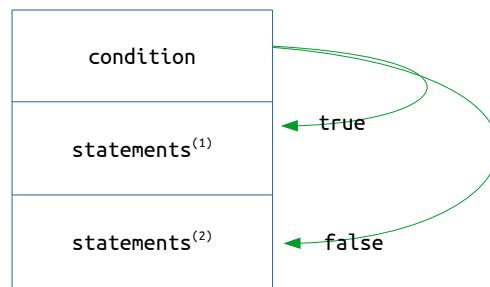
Στη συνέχεια θα δημιουργηθεί κώδικας από την statements:

```
100: >, a, b, 102
101: jump, _, _, _
102: +, b, 1, T_1
103: :=, T_1, _, b
```

Ακολουθεί το genQuad() που θα στείλει τον έλεγχο πίσω στη συνθήκη, δηλαδή στο condQuad, δηλαδή στο 100:

```
100: >, a, b, 102
101: jump, _, _, _
102: +, b, 1, T_1
103: :=, T_1, _, b
104: jump, _, _, 100
```

Ενώ η δημιουργία ενδιάμεσου κώδικα για την while θα τελειώσει με το backpatch() που θα συμπληρώσει τη λίστα false με το nextQuad(), δηλαδή το 105. Ο ολοκληρωμένος κώδικας ακολουθεί:



Σχήμα 7.2: Εσφαλμένη παραγωγή κώδικα για το if.

```

100: >, a, b, 102
101: jump, _, _, 105
102: +, b, 1, T_1
103: :=, T_1, _, b
104: jump, _, _, 100

```

Να σημειώσουμε ότι στο 105 υπάρχει σίγουρα μια εντολή, ακόμα κι αν αυτή είναι το halt,\_,\_,\_ το τέλος του προγράμματος.

### 7.1.2 Η δομή απόφασης "if"

Η γραμματική if είναι η ακόλουθη:

ifStat	$\rightarrow$	if ( condition ) statements <sup>(1)</sup> elsePart
elsePart	$\rightarrow$	else statements <sup>(2)</sup>
		$\epsilon$

Το elsepart είναι προαιρετικό, όπως φαίνεται από τον εναλλακτικό κανόνα που παράγει το κενό. Όπως και στη while, έτσι και στην if πρέπει να διαχειριστούμε τις δύο λίστες που επιστρέφει η condition. Οι τετράδες που έχουν αποθηκευτεί στη λίστα true πρέπει να οδηγηθούν στο statements<sup>(1)</sup>, άρα το backpatch() στο nextQuad() θα πρέπει να γίνει πριν την παραγωγή της πρώτης εντολής των statements<sup>(1)</sup>.

Το backpatch() του false πρέπει να γίνει πριν το else, αν υπάρχει else ή μετά το τέλος της if, μετά και από την τελευταία τετράδα που θα παραγάγει το if. Πρέπει να αναζητήσουμε το κατάλληλο σημείο στη γραμματική μας για να τοποθετήσουμε το backpatch() εκεί. Είμαστε τυχεροί διότι υπάρχει ένα τέτοιο. Αν τοποθετήσουμε το backpatch() στον κανόνα ifStat, ακριβώς πριν το elsePart, τότε (α) αν δεν υπάρχει else, οι τελευταίες τετράδες που θα παραχθούν θα είναι από το statements<sup>(1)</sup>, άρα το backpatch() πράγματι γίνεται μετά την τελευταία τετράδα που θα παραγάγει το if, (β) αν υπάρχει else, τότε η επόμενη τετράδα που θα παραχθεί θα είναι από το statements<sup>(2)</sup>, δηλαδή ακριβώς αυτό που θέλαμε.

Σύμφωνα με τα παραπάνω, η γραμματική γίνεται:

ifStat	$\rightarrow$	if ( condition ) {p1} statements <sup>(1)</sup> {p2}
		elsePart
elsePart	$\rightarrow$	else statements <sup>(2)</sup>
		$\epsilon$

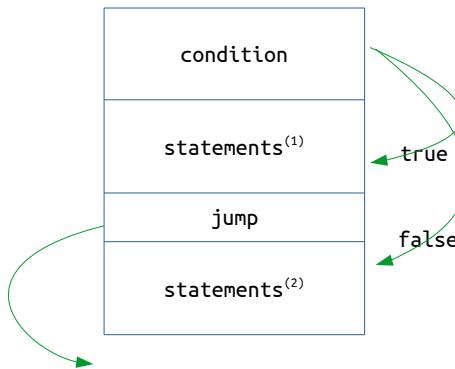
```

{p1} : backpatch(condition.true,nextquad())
{p2} : backpatch(condition.false,nextquad())

```

Στο while υπήρχε η ανάγκη να γίνει ένα άλμα προς τα πίσω, όταν έπρεπε να εξεταστεί πάλι η συνθήκη. Εδώ δεν έχουμε επανεξέταση της συνθήκης, οπότε με την πρώτη ματιά φαίνεται ότι, αφού εκτελούνται σωστά τα άλματα από τη συνθήκη, τότε εκτελείται κάθε φορά ο κώδικας που θέλουμε, οπότε έχουμε τελειώσει.

Ας δούμε τι θα συμβεί, αν χρησιμοποιήσουμε το παραπάνω σχέδιο κώδικα ώστε να παραγάγουμε κώδικα σε ένα παράδειγμα if, στο οποίο υπάρχει και else. Όπως φαίνεται στο σχήμα 7.2, πρώτα θα παρα-



Σχήμα 7.3: Παραγωγή κώδικα για το if.

χθεί και θα τοποθετηθεί στο αρχείο ο κώδικας για το `condition`, μετά για το `statements(1)` και μετά για το `statements(2)`. Έτσι, όταν ολοκληρωθούν οι εντολές για το `statements(1)`, μετά θα εκτελεστούν οι εντολές για το `statements(2)`. Χρειαζόμαστε λοιπόν έναν τρόπο για να παρακάμπτουμε τις εντολές `statements(2)`, όταν οι εντολές `statements(1)` έχουν εκτελεστεί.

Η λύση εικονίζεται στο σχήμα 7.3. Ανάμεσα στις `statements(1)` και `statements(2)` παρεμβάλονται ένα `jmp`, το οποίο θα στείλει την εκτέλεση στην τετράδα μετά την τελευταία τετράδα του `if`. Αν τώρα δεν υπάρχει καθόλου `else`, οπότε δεν υπάρχουν και `statements(2)`, τότε το `jmp` αυτό θα εκτελέσει ένα άλμα στην αμέσως επόμενη εντολή. Κακός κώδικας, αλλά δεν χρειάζεται να ανησυχούμε. Αυτό είναι παιχνιδάκι για τη φάση βελτιστοποίησης που θα ακολουθήσει.

Πώς όμως θα τοποθετήσουμε εκεί το `jmp`? Με μία `genQuad()`. Αλλά ποιο θα είναι το τελευταίο τελούμενο, η τετράδα στην οποία θα γίνει το `jmp`? Εδώ μας παρουσιάζεται ένα πρόβλημα όμοιο με αυτό που είχαμε όταν αφήναμε ασυμπλήρωτες τετράδες στις λογικές συνθήκες. Έτσι και εδώ, όταν δημιουργούμε την τετράδα του άλματος, δεν γνωρίζουμε πού θα γίνει το άλμα, αφού αυτό θα γίνει σε κώδικα που δεν έχει ακόμα παραχθεί. Στη `while` τα πράγματα ήταν εύκολα, διότι το `jmp` γινόταν προς τα πίσω, σε τετράδα που είχε ήδη παραχθεί και το μόνο που έπρεπε να κάνουμε ήταν να σημειώσουμε την ετικέτα αυτής της τετράδας τη στιγμή που την παραγάγαμε. Εδώ πρέπει να δημιουργήσουμε την τετράδα και να επιστρέψουμε να τη συμπληρώσουμε, αφού παραγάγουμε τον ενδιάμεσο κώδικα για τις εντολές `statements(2)`. Χρειαζόμαστε λοιπόν μία `makeList()` για να σημειώσουμε τη μη συμπληρωμένη τετράδα, μία `genQuad()` για να τη δημιουργήσουμε και μία `backpatch()` για να τη συμπληρώσουμε.

Ας βαφτίσουμε τη λίστα αυτή `ifList`. Η λίστα θα δημιουργηθεί στο `{p2}` πριν την `genQuad`, αφού η `makeList()` έχει φτιαχτεί ώστε να σημειώνει την επόμενη τετράδα που θα δημιουργηθεί. Όπως μόλις υπονοήσαμε, θα ακολουθήσει η δημιουργία της τετράδας ασυμπλήρωτου άλματος με την `genQuad()`. Οι δύο αυτές εντολές θα τοποθετηθούν πριν την `backpatch()` στο `false` που ήδη βρίσκεται από προηγουμένων στην `{p2}`. Είναι προφανές ότι, αν οι δύο αυτές εντολές τοποθετηθούν μετά την `backpatch()`, τότε η `backpatch()` θα συμπληρώσει τις τετράδες της να δείχνουν την `jmp`, `_`, `_`, `_`, κάτι που ασφαλώς δεν είναι αυτό που ζητούμε.

Το `backpatch()` της `ifList` πρέπει να δείχνει έξω από την `if`, άρα πρέπει να είναι η τελευταία ενέργεια που θα γίνει κατά τη μετάφραση της `if`. Το κατάλληλο σημείο είναι το τέλος του κανόνα `ifStat`, τοποθετώντας εκεί ένα `{p3}`.

Το σχέδιο ενδιάμεσου κώδικα, αν ενσωματώσουμε τα παραπάνω, γίνεται:

```

ifStat   →  if ( condition ) {p1} statements(1) {p2}
          |           elsePart {p3}
elsePart →  else statements(2)
          |           ε

{p1} : backpatch(condition.true,nextquad())
  
```

```

{p2} : ifList = makeList(nextQuad())
       genQuad('jump','_','_','_')
       backpatch(condition.false,nextquad())
{p3} : backpatch(ifList,nextquad())

```

Είχαμε πει για την backpatch() της condition() ότι είναι το τελευταίο πράγμα που πρέπει να κάνει η if. Το ίδιο είπαμε και για το backpatch() της ifList. Εδώ δημιουργείται με την πρώτη ματιά ένα πρόβλημα, αλλά αν προσέξουμε λίγο περισσότερο θα δούμε ότι και τα δύο backpatch() τελικά συμπληρώνουν τις τετράδες τους με την ίδια ετικέτα, την πρώτη της εντολής που ακολουθεί το if, όπως επιθυμούσαμε. Αυτό συμβαίνει διότι ανάμεσα στα δύο backpatch() δεν έχει δημιουργηθεί κάποια άλλη τετράδα. Άρα ο κώδικας μας είναι σωστός.

Στο σημείο αυτό θα πρέπει πάλι να κάνουμε έναν έλεγχο αν το σχέδιο ενδιάμεσου κώδικα λειτουργεί για τις περιπτώσεις που τα statements<sup>(1)</sup> ή/και τα statements<sup>(2)</sup> είναι το κενό. Αν τα statements<sup>(1)</sup> είναι κενά, τότε το backpatch() της true θα γίνει επάνω στην jump της ifList, οπότε ο έλεγχος θα βγει έξω από την if, χωρίς να συμβεί τίποτε, κάτι που το θέλουμε. Άρα, αν το statements<sup>(1)</sup> είναι κενό, ο κώδικας λειτουργεί. Αν τώρα το statements<sup>(2)</sup> είναι κενό, τότε η backpatch() του false θα κάνει άλμα σε ό,τι ακολουθεί το else, δηλαδή στην πρώτη τετράδα της εντολής που ακολουθεί το if, οπότε ο έλεγχος θα μεταβεί από την if, χωρίς να συμβεί πάλι τίποτε, κάτι που το θέλουμε. Έτσι, μπορούμε να συμπεράνουμε ότι το σχέδιο ενδιάμεσου κώδικα λειτουργεί ακόμα και αν τα statements<sup>(1)</sup> και statements<sup>(2)</sup> είναι κενά.

Ας ολοκληρώσουμε με ένα παράδειγμα μετατροπής αρχικού σε ενδιάμεσο κώδικα. Έστω το αρχικό πρόγραμμα:

```

if (a>b)
  b := b+1
else
  b := b - 1

```

Η παραγωγή κώδικα εκκινεί από τη μεταγλώττιση της συνθήκης a>b, οπότε θα δημιουργηθεί ο κώδικας:

```

100: >, a, b, _
101: jump, _, _, _

```

και οι λίστες condition.true=[100], condition.false=[101]. Στη συνέχεια, μετά το τέλος της συνθήκης μπορούμε να κάνουμε backpatch() στη λίστα true. Άρα ο κώδικας γίνεται:

```

100: >, a, b, 102
101: jump, _, _, _

```

και οι λίστες: condition.true=[], condition.false=[101]

Στη συνέχεια μεταφράζεται το σώμα της if, δηλαδή η ανάθεση b := b+1, και ο κώδικας γίνεται:

```

100: >, a, b, 102
101: jump, _, _, _
102: :=, b, 1, T_1
103: :=, T_1, _, b

```

Σύμφωνα με το σχέδιο ενδιάμεσου κώδικα, δημιουργείται στη συνέχεια η μη συμπληρωμένη jump η οποία και θα σημειωθεί στη λίστα ifList. Άρα ο κώδικας γίνεται:

```

100: >, a, b, 102
101: jump, _, _, _
102: +, b, 1, T_1
103: :=, T_1, _, b
104: jump, _, _, _

```

και οι λίστες έχουν τις τιμές: condition.true=[], condition.false=[101], ifList=[104].

Στη συνέχεια μεταφράζεται ο κώδικας μέσα στο else. Κατ' αναλογία με τις εντολές 101,102 θα δημιουργηθούν οι 105,106, ως εξής:

```

100: >, a, b, 102
101: jump, _, _, _
102: :=, b, 1, T_1
103: :=, T_1, _, b
104: jump, _, _, -
105: -, b, 1, T_1
106: :=, T_1, _, b

```

Η τελευταία ενέργεια είναι το `backpatch()`. Έχουμε δύο `backpatch()`, ένα για τη λίστα `condition.false=[101]` και ένα για την `ifList=[104]`. Τελικά ο κώδικας γίνεται:

```

100: >, a, b, 102
101: jump, _, _, 107
102: :=, b, 1, T_1
103: :=, T_1, _, b
104: jump, _, _, 108
105: -, b, 1, T_1
106: :=, T_1, _, b

```

ενώ όλες οι λίστες είναι κενές.

### 7.1.3 Η δομή επιλογής "switchcase"

Η γραμματική της `switchcase` είναι η ακόλουθη:

```

switchcaseStat → switchcase
                  ( case ( condition ) statements(1) )*
                  default statements(2)

```

Ελέγχονται ένα ένα `condition` και, όταν ένα `condition` αποτιμηθεί ως αληθές, τότε εκτελούνται τα αντίστοιχα `statements(1)` και ο έλεγχος στη συνέχεια μεταβαίνει έξω από τη δομή. Αν κανένα `condition` δεν αποτιμηθεί ως αληθές, τότε εκτελούνται τα `statements(2)` και πάλι ο έλεγχος βγαίνει έξω από τη δομή.

Κάθε `condition` θα επιστρέψει στην `switchcase` δύο λίστες, `true` και `false`, τις οποίες αυτή θα πρέπει να διαχειριστεί. Πρέπει να επιλεγούν τα σημεία που αυτές οι λίστες θα συμπληρωθούν. Κάθε `case`, καθώς και τα αντίστοιχα `condition` και `statements(1)`, βρίσκονται μέσα στο άστρο Kleene και αυτό μας κάνει να υποψιαστούμε ότι η διαχείριση όλων των `case` θα είναι κοινή και, μάλιστα, μέσα στο άστρο Kleene.

Όταν, λοιπόν, ένα `condition` είναι αληθές, ο έλεγχος πρέπει να μεταφερθεί στην αρχή των `statements(1)`. Άρα ακριβώς πριν το `statements(1)` πρέπει να τοποθετηθεί μία κλήση της `backpatch()` που θα συμπληρώνει την `condition.true` στο `nextQuad()`.

Όταν η συνθήκη αποτιμηθεί ως ψευδής, τότε ο έλεγχος πρέπει να μεταφερθεί στο επόμενο `condition` ή, αν δεν υπάρχει επόμενο `condition`, στην αρχή των `statements(2)`. Ένα κατάλληλο τέτοιο σημείο είναι ακριβώς μετά το `statements(1)` και ακριβώς πριν την παρένθεση. Αν υπάρχει επόμενο `condition`, τότε ο κώδικας του `condition` ακολουθεί το `statements(1)`. Αν δεν υπάρχει επόμενο `condition`, τότε ο κώδικας που ακολουθεί το `statements(1)` είναι το `statements(2)`, κάτι που συμπτωματικά είναι αυτό που θέλουμε. Αν και δεν έχει τελειώσει το σχέδιο ενδιάμεσου κώδικα, είμαστε σίγουροι ότι το κομμάτι το οποίο διαχειρίζεται τις λίστες `true` και `false` λειτουργεί σωστά.

Το τμήμα του σχεδίου ενδιάμεσου κώδικα που περιγράψαμε παραπάνω και αφορά τη διαχείριση των λιστών `true` και `false` είναι το ακόλουθο:

```

switchcaseStat → switchcase
                  ( case ( condition ) {p1}
                      statements(1) {p2} )*
                  default statements(2)

```

```
{p1} : backpatch(condition.true,nextQuad())
{p2} : backpatch(condition.false,nextQuad())
```

Δεν έχουμε τελειώσει, όμως. Μετά από κάθε εκτέλεση των statements<sup>(1)</sup> πρέπει να γίνει ένα άλμα έξω από την switchcase. Η λογική δεν μπορεί να είναι διαφορετική από όλα τα προηγούμενα άλματα σε τετράδες που θα δημιουργηθούν αργότερα. Σε καθεμία από τις προηγούμενες δύο περιπτώσεις (if και while) υπήρχε ένα άλμα έξω από τη δομή. Έτσι και εδώ, πρέπει να δημιουργήσουμε ασυμπλήρωτα jump, όταν θέλουμε να μεταβούμε έξω από τη δομή και να κάνουμε το γνώστο backpatch() στο τέλος.

Έχουμε, λοιπόν, έναν αριθμό από statements<sup>(1)</sup>, μετά από καθένα από τα οποία πρέπει να δημιουργηθεί ένα jump, \_, \_,\_. Όλα αυτά τα jump πρέπει να τοποθετηθούν σε μία λίστα η οποία θα συμπληρωθεί στο τέλος της δομής. Ας ονομάσουμε αυτή τη λίστα exitList. Σε κάθε ένα case λοιπόν, μετά την statements<sup>(1)</sup> θα τοποθετήσουμε μία t=makeList(nextQuad()), η οποία θα δημιουργήσει μία λίστα με το ασυμπλήρωτο άλμα, μία genQuad('jump','\_','\_','\_'), η οποία θα δημιουργήσει τη μη συμπληρωμένη τετράδα και μία mergeList(), η οποία θα συνενώσει την t με τη λίστα που τελικά θα γίνει στο τέλος backpatch(), την exitList.

Έτσι, στο σχέδιο ενδιάμεσου κώδικα πρέπει στην αρχή να δημιουργούμε μία κενή exitList, να δημιουργούμε μία μη συμπληρωμένη τετράδα μετά από κάθε statements<sup>(1)</sup> και να την τοποθετούμε στην exitList και στο τέλος να συμπληρώνουμε με backpatch() την exitList, ώστε να δείχνει στην πρώτη τετράδα μετά το τέλος της switchcase. Το ολοκληρωμένο σχέδιο ενδιάμεσου κώδικα ακολουθεί:

```
switchcaseStat → switchcase {p0}
    ( case ( condition ) {p1}
        statements(1) {p2} ) )*
    default statements(2)
    {p3}

{p0} : exitList = emptyList()
{p1} : backpatch(condition.true,nextQuad())
{p2} : t = makeList(nextQuad())
       genQuad('jump','_','_','_')
       exitList = mergeList(exitList,t)
       backpatch(condition.false,nextQuad())
{p3} : backpatch(exitList,nextQuad())
```

Σχηματικά, η λειτουργία του σχεδίου ενδιάμεσου κώδικα απεικονίζεται στο σχήμα 7.4.

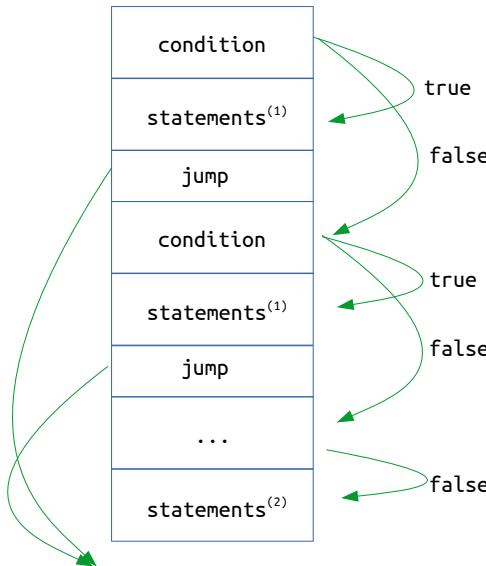
Στο σημείο αυτό θα πρέπει να ελέγξουμε αν το σχέδιο ενδιάμεσου κώδικα λειτουργεί σωστά και για τις περιπτώσει που τα statements<sup>(1)</sup> και statements<sup>(2)</sup> είναι κενά. Πράγματι, αν το statements<sup>(1)</sup> είναι κενό, τότε το backpatch() θα στείλει την condition.true πάνω στο άλμα που θα μεταφέρει τον έλεγχο έξω από τη δομή. Είναι πράγματι αυτό που θα θέλαμε να συμβεί. Αν το statements<sup>(2)</sup> είναι κενό, ο έλεγχος θα “κυλήσει” έξω από την switchcase. Και εδώ θα συμβεί αυτό που θα θέλαμε να συμβεί.

Ας δούμε και ένα παράδειγμα μετατροπής αρχικού κώδικα σε ενδιάμεσο:

```
switchcase
  case D>0:
    a:=1;
  case D<0:
    a:=2;
  default:
    a:=3;
```

Αρχικά αρχικοποιείται η λίστα exitList, η οποία δεν έχει κανένα στοιχείο μέσα της. Στη συνέχεια η πρώτη condition δημιουργεί τις τετράδες:

```
100: >, D, 0, _
```



Σχήμα 7.4: Παραγωγή κώδικα για το switchcase.

101: `jump, _, _, _`

και επιστρέφει τις λίστες `true=[100]`, `false=[101]`. Γίνεται `backpatch()` η λίστα `true` στο 102, που είναι η τετράδα που θα δημιουργηθεί για την `a:=1`. Δημιουργείται και αυτή η τετράδα και ο ενδιάμεσος κώδικας μέχρι τη στιγμή αυτή έχει γίνει:

```

100: >, D, 0, 102
101: jump, _, _, _
102: :=, 1, _, a
  
```

Στη συνέχεια, και σύμφωνα με το σχέδιο ενδιάμεσου κώδικα, σημειώνεται στη λίστα `t` και δημιουργείται η τετράδα:

103: `jump, _, _, _`

που θα μας βγάλει έξω από τη δομή, όταν συμπληρωθεί. Η λίστα `t` συνενώνεται με τη λίστα `exitList`, οπότε η `exitList` έχει μέσα της ένα μόνο στοιχείο: `exitList=[103]`.

Ακολουθεί το `backpatch()` του `false`. Αυτό θα μας στείλει στην επόμενη τετράδα που θα δημιουργηθεί, δηλαδή έξω από το άστρο του Kleene, στην 104. Έτσι, και οι δύο λίστες `true` και `false` έχουν συμπληρωθεί για την πρώτη `condition`, ενώ η λίστα είναι η μόνη λίστα που θα πρέπει να συμπληρωθεί παρακάτω. Ο κώδικας που έχει παραχθεί μέχρι στιγμής είναι ο ακόλουθος:

```

100: >, D, 0, 102
101: jump, _, _, 104
102: :=, 1, _, a
103: jump, _, _, _
  
```

Με όμοιο ακριβώς τρόπο θα παραχθεί και ο κώδικας για τη δεύτερη συνθήκη. Μετά την παράγωγή κώδικα και για τη δεύτερη συνθήκη ο ενδιάμεσος κώδικας θα είναι όπως φαίνεται παρακάτω:

```

100: >, D, 0, 102
101: jump, _, _, 104
102: :=, 1, _, a
103: jump, _, _, _
104: <, D, 0, 106
105: jump, _, _, 108
106: :=, 2, _, a
107: jump, _, _, _
  
```

και η λίστα exitList θα έχει μέσα της τις τετράδες: exitList=[103,107]

Στη συνέχεια, θα παραχθεί ο κώδικας που αντιστοιχεί στο default, δηλαδή οι εντολές του statements<sup>(2)</sup>. Άρα ο κώδικας θα γίνει:

```
100: >, D, 0, 102
101: jump, _, _, 104
102: :=, 1, _, a
103: jump, _, _, -
104: <, D, 0, 106
105: jump, _, _, 108
106: :=, 2, _, a
107: jump, _, _, -
108: :=, 3, _, a
```

Τελευταία ενέργεια είναι να συμπληρωθούν οι τετράδες της exitList με την τετράδα της πρώτης εντολής που ακολουθεί το switchcase, την nextQuad(), δηλαδή την 109. Ο ενδιάμεσος κώδικας, ολοκληρωμένος, ακολουθεί:

```
100: >, D, 0, 102
101: jump, _, _, 104
102: :=, 1, _, a
103: jump, _, _, 109
104: <, D, 0, 106
105: jump, _, _, 108
106: :=, 2, _, a
107: jump, _, _, 109
108: :=, 3, _, a
```

#### 7.1.4 Η δομή επιλογής-επανάληψης "forcase"

Η γραμματική της forcase μοιάζει με αυτήν της switchcase:

```
forcaseStat → forcase
              ( case ( condition ) statements(1) ) )*
              default statements(2)
```

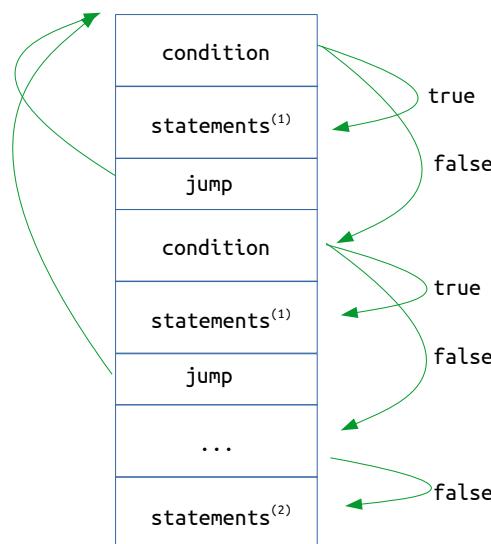
Η δομή επανάληψης forcase ελέγχει τις condition που βρίσκονται μετά τα case. Μόλις μία από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες statements<sup>(1)</sup> (που ακολουθούν το condition). Μετά ο έλεγχος μεταβαίνει στην αρχή της forcase. Αν καμία από τις case δεν ισχύει, τότε ο έλεγχος μεταβαίνει στη default και εκτελούνται οι statements<sup>(2)</sup>. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την forcase.

Σκεπτόμενοι όμοια με τις προηγούμενες δομές που αναλύσαμε, πρέπει να διαχειριστούμε τις λίστες true και false και πρέπει να υλοποιήσουμε το άλμα προς τα πίσω στην πρώτη συνθήκη. Πρέπει, επίσης, να προσέξουμε και το default.

Αφού έχουμε άλμα προς την πρώτη συνθήκη, πρέπει, πριν δημιουργηθεί η πρώτη τετράδα της συνθήκης, πριν δηλαδή κληθεί ο κανόνας condition και όσο φυσικά είμαστε ακόμα μέσα στην forCaseStat, να σημειώσουμε την πρώτη τετράδα της συνθήκης. Αυτό γίνεται με την κλήση της nextQuad() ακριβώς πριν αρχίσουν οι συνθήκες, δηλαδή θα μπορούσε να τοποθετηθεί μετά το forcase:

```
forcaseStat → forcase {p1}
              ( case ( condition ) statements(1) ) )*
              default statements(2)

{p1} : firstCondQuad = nextQuad()
```



Σχήμα 7.5: Παραγωγή κώδικα για το forcase.

Θα χειριστούμε τα condition ακριβώς όπως στην switchcase. Και εδώ όταν ένα condition είναι αληθές, ο έλεγχος πρέπει να μεταφερθεί στην αρχή των statements<sup>(1)</sup>. Όταν η συνθήκη αποτιμηθεί ως φευδής, τότε ο έλεγχος πρέπει να μεταφερθεί στο επόμενο condition ή, αν δεν υπάρχει επόμενο condition, στην αρχή των statements<sup>(2)</sup>.

Ένα κατάλληλο σημείο για να τοποθετηθεί η backpatch() που θα συμπληρώνει την condition.true στο nextQuad() είναι ακριβώς πριν το statements<sup>(1)</sup>. Αντίστοιχα, ένα κατάλληλο σημείο για το backpatch() της false είναι ακριβώς μετά το statements<sup>(1)</sup> και ακριβώς πριν την παρένθεση. Αν υπάρχει επόμενη condition, τότε ο κώδικας του condition ακολουθεί το statements<sup>(1)</sup>. Αν δεν υπάρχει επόμενο condition, τότε ο κώδικας που ακολουθεί το statements<sup>(1)</sup> είναι το statements<sup>(2)</sup>.

```

forcaseStat → forcase {p1}
                  ( case ( condition ) {p2}
                      statements(1) {p3} )*
                  default statements(2)

{p1} : firstCondQuad = nextQuad()
{p2} : backpatch(condition.true,nextQuad())
{p3} : backpatch(condition.false,nextQuad())

```

Αφού σημειώσαμε την τετράδα στην οποία πρέπει να κάνουμε άλμα, συμπληρώσαμε κατάλληλα τις λίστες που μας έδωσε ασυμπλήρωτες η condition, ελέγξαμε ότι η default λειτουργεί όπως θα θέλαμε, μας απομένει να υλοποιήσουμε το άλμα προς τα πίσω. Αυτό θα γίνει κάθε φορά που αποτιμάται μία συνθήκη αληθής και αφού εκτελεστούν τα statements<sup>(1)</sup>. Άρα ακριβώς μετά την statements<sup>(1)</sup>, στην αρχή δηλαδή του {p3}. Το ολοκληρωμένο σχέδιο ενδιάμεσου κώδικα είναι το ακόλουθο:

```

forcaseStat → forcase {p1}
                  ( case ( condition ) {p2}
                      statements(1) {p3} )*
                  default statements(2)

{p1} : firstCondQuad = nextQuad()
{p2} : backpatch(condition.true,nextQuad())
{p3} : genQuad('jump','_','_',firstCondQuad)
      backpatch(condition.false,nextQuad())

```

Σχηματικά, η λειτουργία του σχεδίου ενδιάμεσου κώδικα απεικονίζεται στο σχήμα 7.5. Ακολουθεί παράδειγμα μετατροπής αρχικού κώδικα σε ενδιάμεσο για την `forcase`, η εξήγησή του όμως αφήνεται στον αναγνώστη:

```
forcase
  case a>0:
    a:=a-1;
  case a<0:
    a:=a+1;
  default:
;
```

Το πρόγραμμα αυτό μειώνει το `a` κατά 1, όσο το `a` είναι θετικό, αυξάνει το `a` κατά 1, όσο το `a` είναι αρνητικό και δεν κάνει τίποτε όταν αυτό γίνει 0. Ο ενδιάμεσος κώδικας που παράγεται ακολουθεί:

```
100: >, a, 0, 102
101: jump, _, _, 105
102: -, a, 1, T_1
103: :=, T_1, _, a
104: jump, _, _, 100
105: <, a, 0, 107
106: jump, _, _, 110
107: +, a, 1, T_2
108: :=, T_2, _, a
109: jump, _, _, 100
```

### 7.1.5 Η δομή πολλαπλής επιλογής-επανάληψης "incase"

Ας θυμηθούμε τη γραμματική της `incase`:

```
incaseStat → incase
            ( case ( condition ) statements(1) )*
            default statements(2)
```

Η δομή επανάληψης `incase` ελέγχει τις `condition` που βρίσκονται μετά τα `case`, εξετάζοντάς τες κατά σειρά. Για καθεμία από αυτές που η αντίστοιχη `condition` ισχύει, εκτελούνται οι αντίστοιχες `statements` (που ακολουθούν το `condition`). Θα εξεταστούν όλες οι `condition` και θα εκτελεστούν όλες οι `statements` των οποίων οι `condition` ισχύουν. Αφότου εξεταστούν όλες οι `case`, ο έλεγχος μεταβαίνει έξω από τη δομή `incase` εάν καμία από τις `statements` δεν έχει εκτελεστεί ή μεταβαίνει στην αρχή της `incase`, εάν, έστω και μία από τις `statements` έχει εκτελεστεί.

Η δομή μοιάζει σε εξαιρετικά μεγάλο βαθμό με τις `switchcase` και `forcase`. Στην πραγματικότητα όμως η μετατροπή της σε ενδιάμεσο κώδικα ακολουθεί πολύ διαφορετική φιλοσοφία από αυτή των άλλων δύο.

Όλα αυτά τα οποία έχουμε συζητήσει μέχρι τώρα αφορούν ενέργειες που κάνει ο μεταγλωττιστής σε χρόνο μετάφρασης. Και έτσι πρέπει να είναι φυσικά, αφού ο μεταγλωττιστής δεν εμπλέκεται καθόλου στον χρόνο εκτέλεσης. Η αρμοδιότητά του είναι να παράγει (σε χρόνο μετάφρασης) τον τελικό κώδικα που απαιτείται, ώστε όταν αυτός εκτελεστεί (σε χρόνο εκτέλεσης) να έχουμε το αποτέλεσμα που επιθυμούμε και περιγράψαμε με τον αρχικό μας κώδικα. Έτσι, σε χρόνο μετάφρασης δημιουργούμε εντολές και σε χρόνο εκτέλεσης τις εκτελούμε.

Σε ό,τι έχουμε δει μέχρι τώρα, ο μεταγλωττιστής δημιουργεί όλες τις εναλλακτικές διαδρομές και κατά την εκτέλεση του προγράμματος ακολουθείται μία από αυτές. Φέρτε στο μυαλό σας ως πιο χαρακτηριστικό παράδειγμα το `if` με την επιλογή του `else`. Όταν κάποια μέσα στο σχέδιο ενδιάμεσου κώδικα εμφανίζεται η εντολή `firstQuad=nextQuad()`, η `firstQuad` θα πάρει τιμή κατά τη μετάφραση του προγράμματος.

Η `incase` ελέγχει όλες τις `condition` που βρίσκονται μετά τα `case`. Αφότου εξεταστούν όλες οι `case`, τότε καλούμαστε να λάβουμε μία απόφαση. Εάν καμία από τις `statements(1)` δεν έχει εκτελεστεί, τότε ο έλεγχος μεταβαίνει έξω από την `incase`. Άλλως μεταβαίνει στην αρχή της `incase`. Η πληροφορία, εάν, έστω και μία από τις `statements(1)` έχει εκτελεστεί, πρέπει να συλλεχθεί κατά τη διάρκεια της αποτίμησης των εκφράσεων ή της εκτέλεσης των `statements(1)` και να αξιολογηθεί όταν φτάσουμε στο τέλος της δομής, λίγο πριν την `default`. Στην `switchcase` και στην `forcase`, δεν υπήρχει τέτοιο πρόβλημα, αφού όταν αποτιμούσαμε μία `condition` η πορεία του κώδικα μέχρι το τέλος της επανάληψης ήταν μονόδρομος.

Η απόφαση για το αν θα εξέλθουμε από τη δομή ή αν θα επιστρέψουμε πίσω λαμβάνεται με την ίδια λογική που λαμβάνεται και στο `bubble sort`. Εκεί, χρησιμοποιούμε μια μεταβλητή, που έχει καθιερωθεί να ονομάζεται `flag`, η οποία σημειώνει αν έχει γίνει μία εναλλαγή στοιχείων κατά το τελευταίο πέρασμα ή όχι. Στο τέλος, αν στη μεταβλητή έχει σημειωθεί έστω και μία εναλλαγή, τότε ο πίνακας με τις προς ταξινόμηση τιμές διαπερνάται πάλι, σε αναζήτηση νέων πιθανών εναλλαγών. Θα χρησιμοποιήσουμε, λοιπόν, και εδώ μία `flag` η οποία θα σημειώνει αν έστω και μία από τις `statements(1)` έχει εκτελεστεί.

Δεν δείχνει δύσκολο, ούτε και είναι. Αρκεί να συνειδητοποιήσουμε ότι η μεταβλητή `flag` (θα την ονομάσουμε και εμείς έτσι, ως φόρο τιμής) δεν παίρνει τιμές σε χρόνο μετάφρασης αλλά σε χρόνο εκτέλεσης. Αν δηλαδή στο σχέδιο ενδιάμεσου κώδικα σημειώναμε `flag:=1` αυτό σημαίνει ότι κατά τη συντακτική ανάλυση του αρχικού κώδικα και την παραγωγή του ενδιάμεσου, η μεταβλητή `flag` θα γινόταν ίση με 1. Εμείς θέλουμε να γίνει ίση με 1 κατά την εκτέλεση του προγράμματος που παραγάγαμε, αφού τότε και μόνο τότε θα είναι γνωστό αν μία συνθήκη αποτιμήθηκε ως αληθής ή ψευδής. Συνεπώς, η μεταβλητή `flag` πρέπει να είναι μία προσωρινή μεταβλητή, την οποία ο μεταγλωτιστής θα δημιουργήσει για τον σκοπό αυτό.

Στο σχέδιο ενδιάμεσου κώδικα, τώρα, εφόσον υπάρχει πιθανότητα άλματος προς τα πίσω στην πρώτη τετράδα της πρώτης `condition`, πρέπει, όταν περνάμε από εκεί, πριν δημιουργηθεί, να τη σημειώνουμε, κατά την προσφιλή μας συνήθεια.

Πρέπει επίσης να διαχειριστούμε τις `condition.true` και `condition.false`. Μπορούμε εύκολα να παρατηρήσουμε ότι η διαχείριση αυτή δεν διαφέρει από αυτήν που απαιτήθηκε στις `switchcase` και `incase`. Έτσι, το σχέδιο ενδιάμεσου κώδικα που έχουμε ως βάση μας είναι το εξής:

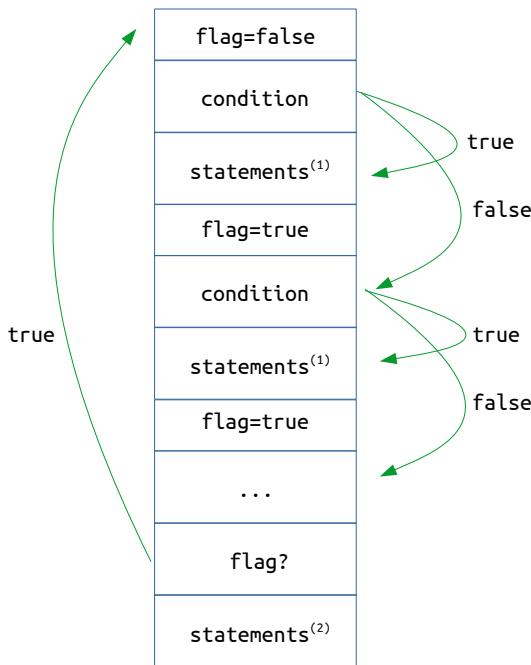
```
incaseStat → incase {p1}
              ( case ( condition ) {p2}
                statements(1) {p3} )*
              default statements(2)

{p1} : firstCondQuad = nextQuad()
{p2} : backpatch(condition.true,nextQuad())
{p3} : backpatch(condition.false,nextQuad())
```

Η μεταβλητή `flag` πρέπει να δημιουργηθεί στην αρχή. Μην ξεχνάμε ότι πρόκειται για προσωρινή μεταβλητή, οπότε πρέπει να δημιουργηθεί με την `newTemp()`. Στη συνέχεια, πάλι στην αρχή πρέπει να αρχικοποιηθεί. Ας την αρχικοποιήσουμε σε `false` (0), δηλαδή ας θεωρήσουμε ότι η μεταβλητή `flag` απαντά στο ερώτημα αν έχει εκτελεστεί έστω και ένα από τα `statements(1)`. Το σημείο στο οποίο πρέπει να αλλάξει τιμή σε `true` (1) είναι αμέσως μετά το `statements(1)`. Ο έλεγχος αν πρέπει να γίνει το άλμα προς τα πίσω είναι ακριβώς πριν το `default`. Αν, λοιπόν, το `flag` είναι `false` (0), τότε θα γίνει το άλμα, αλλιώς η εκτέλεση θα κυλήσει μόνη της στο `default`.

Ένα άλλο σημείο το οποίο θέλει λίγο προσοχή, είναι ότι το άλμα προς τα πίσω δεν πρέπει να στοχεύει την πρώτη τετράδα της `condition`, όπως λέγαμε παραπάνω και γινόταν τόσο στην `switchcase` όσο και στην `forcase`. Σε κάθε επανάληψη της `incase` η τιμή του `flag` πρέπει να επαναρχικοποιείται στο `false` (0). Άρα το άλμα πρέπει να στοχεύει την αρχικοποίηση αυτή. Αυτό δεν είναι καθόλου δύσκολο να γίνει, απλά πρέπει να τοποθετήσουμε την αρχικοποίηση μετά την `firstCondQuad = nextQuad()`. Έτσι το σχέδιο ενδιάμεσου κώδικα γίνεται:

```
incaseStat → incase {p1}
              ( case ( condition ) {p2}
```



Σχήμα 7.6: Παραγωγή κώδικα για το `incase`.

```
statements(1) {p3} )*
default statements(2)
```

```
{p1} : flag = newTemp()
       firstCondQuad = nextQuad()
       genQuad(':=', 0, _, flag)
{p2} : backpatch(condition.true, nextQuad())
{p3} : genQuad(':=', 1, _, flag)
       backpatch(condition.false, nextQuad())
{p4} : genQuad('=', 1, flag, firstQuad)
```

Σχηματικά το σχέδιο ενδιάμεσου κώδικα `incase` απεικονίζεται στο σχήμα 7.6.

Θα ολοκληρώσουμε με ένα παράδειγμα μετατροπής αρχικού κώδικα που περιέχει μία `incase` σε ενδιάμεσο κώδικα. Η εξήγηση του παραδείγματος αφήνεται στον αναγνώστη.

```
incase
  case a>0:
    a:=a-1;
  case a<0:
    a:=a+1;
  default:
    a:=0;
```

Ο ενδιάμεσος κώδικας που προκύπτει είναι ο ακόλουθος:

```
100: :=, 0, _, T_1
101: >, a, 0, 103
102: jump, _, _, 106
103: -, a, 1, T_2
104: :=, T_2, _, a
105: :=, 0, _, T_1
106: <, a, 0, 108
107: jump, _, _, 111
```

```

108: +, a, 1, T_3
109: :=, T_3, _, a
110: :=, 0, _, T_1
111: =, T_1, 1, 100
112: :=, 0, _, a

```

## 7.2 Συναρτήσεις και διαδικασίες

Η ενδιάμεση αναπαράσταση είναι μία γλώσσα, η οποία δεν χρησιμοποιεί σύνθετες προγραμματιστικές δομές. Θα τη χαρακτηρίζαμε, όμως, ως γλώσσα υψηλού επιπέδου, για δύο κυρίως λόγους:

- χρησιμοποιεί μεταβλητές, σε αντίθεση με την assembly που η έννοια της μεταβλητής έχει πια απαλειφθεί,
- χρησιμοποιεί συναρτήσεις. Ο συμβολισμός του τρόπου κλήσεων των συναρτήσεων απλοποιείται, αλλά οι κλήσεις συναρτήσεων δεν εξαλείφονται στο στάδιο αυτό.

Με τις συναρτήσεις, και κύρια την κλήση τους, αφού η δήλωσή τους έχει συζητηθεί στην προηγούμενη ενότητα με τα begin\_block και end\_block, θα ασχοληθούμε στην ενότητα αυτή. Αντίθετα με το τι θα περιμενει κανείς, η κλήση των συναρτήσεων δεν απαιτεί πολύπλοκο χειρισμό, αλλά έναν απλό μετασχηματισμό, ώστε να είναι δυνατόν να συμβολιστεί στον ενδιάμεσο κώδικα. Η πολλή δουλειά μεταφέρεται για τη φάση του τελικού κώδικα.

Θα λειτουργήσουμε με παραδείγματα, γιατί είναι ευκολότερο. Ας θεωρήσουμε ότι πρέπει να μετασχηματίσουμε σε ενδιάμεσο κώδικα την:

```
call proc(in x, inout y)
```

Πρώτα συμβολίζουμε σε ενδιάμεσο κώδικα τις παραμέτρους και στη συνέχεια καλούμε τη διαδικασία:

```

100: par, x, cv, _
101: par, y, ref, _
102: call, proc, _, _

```

Έτσι, όταν μία διαδικασία ή συνάρτηση θέλει να δει τις παραμέτρους της, θα ξεκινήσει από το σημείο που γίνεται η κλήση της (call) και θα κινείται προς τα πίσω στον κώδικα μέχρι να τελειώσουν τα par. Ο μετασχηματισμός αυτός είναι μονοσήμαντος. Μπορούμε από την C-implement να πάμε στον ενδιάμεσο κώδικα και από τον ενδιάμεσο κώδικα πίσω στην C-implement.

Ας δούμε τι αλλάζει αν αντί για κλήση διαδικασίας έχουμε κλήση συνάρτησης. Η συνάρτηση, μόλις ολοκληρωθεί, επιστρέφει ένα αποτέλεσμα και το αποτέλεσμα αυτό πρέπει κάπου να αποθηκευτεί. Χρειαζόμαστε μια μεταβλητή, η οποία δεν θα έχει ήδη χρησιμοποιηθεί, ούτε πρόκειται να χρησιμοποιηθεί παρακάτω στο πρόγραμμα. Για τον σκοπό αυτόν, έχουμε τον μηχανισμό των προσωρινών μεταβλητών και αυτόν τον μηχανισμό θα χρησιμοποιήσουμε. Θυμηθείτε ότι όταν περιγράφαμε την ενδιάμεση αναπαράσταση μιλήσαμε για έναν ακόμα τύπο παραμέτρων, το ret. Για κάθε κλήση συνάρτησης, λοιπόν, θα ορίζουμε μία προσωρινή μεταβλητή ως ret και εκεί θα θεωρούμε (και θα υλοποιηθεί στον τελικό κώδικα) ότι θα αποθηκεύεται το αποτέλεσμα της συνάρτησης. Ας θεωρήσουμε μία κλήση που η συνάρτηση αποτελεί το δεξιό μέλος μιας εκχώρησης:

```
x := 1 + func(in x, inout y)
```

Ο ενδιάμεσος κώδικας που αντιστοιχεί στο παραπάνω, ακολουθεί, με το τμήμα της κλήσης της συνάρτησης να αντιστοιχεί στις εντολές 100-103:

```

100: par, x, cv, _
101: par, y, ref, _

```

```

102: par, T_1, ret, _
103: call, func, _, _
104: +, 1, T_1, T_2
105: :=, T_2, _, x

```

Ας δούμε ένα ακόμα παράδειγμα στο οποίο τα πράγματα είναι λίγο πιο μπλεγμένα:

```
x := max (in max(in a, in b), in max(in c, in d))
```

Ο κώδικας που αντιστοιχεί στην παραπάνω κλήση συναρτήσεων είναι ο ακόλουθος:

```

100: par, a, cv, _
101: par, b, cv, _
102: par, T_1, ret, _
103: call, max, _, _
104: par, c, cv, _
105: par, d, cv, _
106: par, T_2, ret, _
107: call, max, _, _
108: par, T_1, cv, _
109: par, T_2, cv, _
110: par, T_3, ret, _
111: call, max, _, _
112: :=, T_3, _, x

```

Μία ακόμα εντολή, της οποίας το καταλληλότερο σημείο για να συζητηθεί είναι εδώ, είναι η `return`. Με την `return` μία συνάρτηση επιστρέφει την τιμή της σε αυτόν που την κάλεσε. Ή:

```
return (x)
```

αντιστοιχεί στη δημιουργία μιας εντολής ενδιάμεσου κώδικα, της:

```
ret, _, _, _
```

Ως παράδειγμα ας δούμε τη μετατροπή του παρακάτω κώδικα σε ενδιάμεσο:

```

program blocks
  ...
  function block1()
  {
    ...
    function block2()
    {
      ...
    }
    # main #
    {
      ...
    }.

```

Η σειρά που θα εμφανιστούν στον ενδιάμεσο κώδικα τα `begin_block` και `end_block`, καθώς και η θέση του `halt` είναι η εξής:

```

xxx : begin_block, block1, _, _
      ... code for block1
xxx : end_block, block1, _, _
xxx : begin_block, block2, _, _
      ... code for block2
xxx : end_block, block2, _, _
xxx : begin_block, main_blocks, _, _
      ... code for main_blocks
xxx : halt, _, _, _
xxx : end_block, main_blocks, _, _

```

### 7.3 Ένα ολοκληρωμένο, απλό, παράδειγμα μετατροπής κώδικα σε ενδιάμεση αναπαράσταση

Θα παρακολουθήσουμε στην ενότητα αυτή, βήμα-βήμα, τη μετατροπή ενός ολοκληρωμένου προγράμματος *C-imple* σε ενδιάμεσο κώδικα. Ισως ο κώδικας αυτός να μην κάνει κάτι χρήσιμο, αλλά αυτό δεν πειράζει, ούτε εμάς, ούτε τον μεταγλωττιστή. Έχουμε, λοιπόν, το παρακάτω πρόγραμμα γραμμένο σε *C-imple*.

```
program small()
{
    const A:=1;
    declare b,g,f;

    function P1(in X, inout Y)
    {       declare e,f;

        function P11(inout X)
        {       declare e;
            e:=A;
            X:=Y;
            f:=b;
            return(e);
        }

        # code for P1 #
        b:=X;
        e:=P11(inout X);
        e:=P1(in X,inout Y);
        X:=b;
        return(e);
    }

    # code for main #
    if (b>1 and f<2 or g+1<f+b)
    {
        f:=P1(in g);
    }
    else
    {
        f:=1;
    }
}.
```

Η συντακτική ανάλυση θα ξεκινήσει από το `program`, θα συνεχίσει στο `const` και το `declare`, θα μπει μέσα στη συνάρτηση `P1`, θα περάσει από το `declare`, θα μπει μέσα στη συνάρτηση `P11`, θα περάσει από το `declare` και θα φτάσει στο `e := A`. Ως εδώ δεν θα δημιουργηθεί καθόλου ενδιάμεσος κώδικας. Η πρώτη εντολή ενδιάμεσου κώδικα που θα δημιουργηθεί θα είναι η `begin_block` του `P11` και η δεύτερη η εκχώρηση στο `e`:

```
1 :     begin_block, P11, _, _
2 :     :=, A, _, e
```

Ακολουθούν οι εκχωρήσεις στο `X` και στο `f`:

```
3 :     :=, Y, _, X
4 :     :=, b, _, f
```

Τελευταία εντολή είναι η `return`, η οποία θα δημιουργήσει το αντίστοιχο `ret`. Επειδή όμως τελειώνει η μετάφραση της P11, θα παραχθεί και μία `end_block`:

```
5 :   ret, _, _, e
6 :   end_block, P11, _, _
```

Άρα, ο κώδικας για την P11 είναι ο ακόλουθος:

```
1 :   begin_block, P11, _, _
2 :   :=, A, _, e
3 :   :=, Y, _, X
4 :   :=, b, _, f
5 :   ret, _, _, e
6 :   end_block, P11, _, _
```

Στη συνέχεια ακολουθεί η μετάφραση της P1. Παρόλο ότι η P11 είναι παιδί της P1 και παρόλο που δήλωση της P1 προηγείται αυτής της P11, ο ενδιάμεσος κώδικας που παράγεται για την P1, έπειτα αυτού της P11, διότι η εμφάνιση του κώδικα της P1 έπειτα αυτής της P11. Αρχικά δημιουργείται το `begin_block` και στη συνέχεια η εκχώρηση στο b:

```
7 :   begin_block, P1, _, _
8 :   :=, X, _, b
```

Έπειτα εμφανίζονται δύο κλήσεις συναρτήσεων, μία για την P11 και μία για την P1. Η P11 έχει ως παράμετρο την `inout X`, οπότε δημιουργείται η τετράδα:

```
9 :   par, X, ref, _
```

και επειδή πρόκειται για συνάρτηση, πρέπει να δημιουργήσουμε μία προσωρινή μεταβλητή για να κρατήσει την τιμή που θα επιστρέψει:

```
10 :   par, T_1, ret, _
```

Ακολουθεί η κλήση της P11 και η εκχώρηση στο e:

```
11 :   call, P11, _, _
12 :   :=, T_1, _, e
```

Όμοια, για την P1, έχουμε δύο παραμέτρους, τις X και Y, μία ακόμα για την επιστροφή τιμής, την κλήση και την εκχώρηση στο e:

```
13 :   par, X, cv, _
14 :   par, Y, ref, _
15 :   par, T_2, ret, _
16 :   call, P1, _, _
17 :   :=, T_2, _, e
```

Για να κλείσει η συνάρτηση, απαιτείται ακόμα μία εκχώρηση στο X και μία `return`:

```
18 :   :=, b, _, X
19 :   ret, e, _, _
20 :   end_block, P1, _, _
```

Ακολουθεί ολοκληρωμένος ο ενδιάμεσος κώδικας για την P1:

```
7 :   begin_block, P1, _, _
8 :   :=, X, _, b
9 :   par, X, ref, _
10 :  par, T_1, ret, _
11 :  call, P11, _, _
12 :  :=, T_1, _, e
```

```

13 :   par, X, cv, _
14 :   par, Y, ref, _
15 :   par, T_2, ret, _
16 :   call, P1, _, _
17 :   :=, T_2, _, e
18 :   :=, b, _, X
19 :   ret, e, _, _
20 :   end_block, P1, _, _

```

Τελευταίος ακολουθεί ο κώδικας για το κυρίως πρόγραμμα. Το τμήμα που παρουσιάζει περισσότερο ενδιαφέρον είναι το if με τη λογική συνθήκη.

Η πρώτη τετράδα που θα δημιουργηθεί, μετά το begin\_block, είναι αυτή της λογικής συνθήκης. Δημιουργούνται δύο άλματα, με το τελευταίο τελούμενο να μην είναι συμπληρωμένο, ένα λογικό άλμα για το true και απλό jump για το false:

```

21 :   begin_block, main_small, _, _
22 :   >, b, 1, _
23 :   jump, _, _, _

```

Με το που συναντάμε το and γνωρίζουμε ότι πρέπει να γίνει στην εντολή 22 μία backpatch() στο nextQuad(). Το κάνουμε:

```

22 :   >, b, 1, 24
23 :   jump, _, _, _

```

Ακολουθούν τα δύο μη συμπληρωμένα άλματα για την  $f < 2$ :

```

24 :   <, f, 2, _
25 :   jump, _, _, _

```

Φτάνοντας στο or, γνωρίζουμε ότι σε αυτό που θα το ακολουθήσει πρέπει να στραφούν οι μη συμπληρωμένες τετράδες από το false της λογικής έκφρασης  $b > 1$  and  $f < 2$ . Πρόκειται για τις τετράδες 23, 25 που θα κάνουν άλμα στο 26. Ας δούμε μέχρι στιγμής πώς έχει γίνει ο κώδικας:

```

21 :   begin_block, main_small, _, _
22 :   >, b, 1, 24
23 :   jump, _, _, 26
24 :   <, f, 2, _
25 :   jump, _, _, 26

```

Μένει μονάχα ασυμπλήρωτη η τετράδα 24. Στη συνέχεια θα μεταφραστεί ο κώδικας για το  $g+1 < f+b$ . Εδώ, πριν γίνει η σύγκριση ανάμεσα στα δύο μέλη της ανισότητας, θα γίνουν οι πράξεις στο δεξί και το αριστερό μέλος. Θα χρειαστούμε, λοιπόν, δύο προσωρινές μεταβλητές που θα αποθηκεύσουν τα αποτελέσματα των πράξεων  $g+1$  και  $f+b$ :

```

26 :   +, g, 1, T_3
27 :   +, f, b, T_4

```

και μετά θα ακολουθήσει η σύγκριση με τις μη συμπληρωμένες τετράδες:

```

28 :   <, T_3, T_4, _
29 :   jump, _, _, _

```

Ο επόμενος κώδικας που θα μεταφραστεί είναι το  $f := P1(\text{in } g)$  και η πρώτη του τετράδα θα τοποθετηθεί στο 30. Πρόκειται για τον κώδικα που θέλουμε να εκτελεστεί αν το if αποτιμηθεί ως αληθές. Άρα θα συμπληρώσουμε με το 30 τις τετράδες 24 και 28.

```

24 :   <, f, 2, 30
28 :   <, T_3, T_4, 30

```

Στη θέση 30 και παρακάτω, λοιπόν, θα έχουμε δύο παραμέτρους, την `g` που περνάει με τιμή και την `T_5` που είναι για επιστροφή της τιμής της συνάρτησης, μία `call` και μία εκχώρηση:

```
30 :  par, g, cv, _
31 :  par, T_5, ret, _
32 :  call, P1, _, _
33 :  :=, T_5, _, f
```

Αφού τελειώσει το σώμα της `if` που εκτελείται όταν η συνθήκη ισχύει, μετά μεταφράζεται το τμήμα του `else`, αν υπάρχει. Εδώ υπάρχει. Ανάμεσα στον κώδικα που εκτελείται στην περίπτωση της αληθούς και της ψευδούς συνθήκης, εισάγεται ένα `jump`, το οποίο δεν αφήνει τον κώδικα της ψευδούς συνθήκης να εκτελεστεί, αν εκτελεστεί ο αντίστοιχος κώδικας της αληθούς. Άρα πριν μεταφράσουμε το `else` εισάγουμε ένα κενό `jump` που θα συμπληρωθεί με την πρώτη τετράδα έξω από τη δομή `if`.

```
34 :  jump, _, _, _
```

Ακολουθεί ο κώδικας για το `else` που αποτελείται από μία μόνο εκχώρηση:

```
35 :  :=, 1, _, f
```

Εκεί θα κάνει `backpatch()` και η λίστα `false` της συνθήκης, η οποία έχει μέσα της μόνο την τετράδα 29:

```
29 :  jump, _, _, 35
```

Ο κώδικας μας τελείωσε. Προσθέτουμε την `halt` και την `end_block`.

```
36 :  halt, _, _, _
37 :  end_block, main_small, _, _
```

Στην 36 πρέπει να γίνει `backpatch()` η 34, που έχει μείνει ασυμπλήρωτη και πρέπει να οδηγήσει την εκτέλεση έξω από το `if`:

```
34 :  jump, _, _, 36
```

Ο κώδικας για το κυρίως πρόγραμμα, ολοκληρωμένος, ακολουθεί:

```
21 :  begin_block, main_small, _, _
22 :  >, b, 1, 24
23 :  jump, _, _, 26
24 :  <, f, 2, 30
25 :  jump, _, _, 26
26 :  +, g, 1, T_3
27 :  +, f, b, T_4
28 :  <, T_3, T_4, 30
29 :  jump, _, _, 35
30 :  par, g, cv, _
31 :  par, T_5, ret, _
32 :  call, P1, _, _
33 :  :=, T_5, _, f
34 :  jump, _, _, 36
35 :  :=, 1, _, f
36 :  halt, _, _, _
37 :  end_block, main_small, _, _
```

## 7.4 Ένα πιο σύνθετο παράδειγμα, με φωλιασμένες δομές

Ας δούμε και ένα πιο σύνθετο παράδειγμα, όπου οι φωλιασμένες δομές θέλουν πολλή προσοχή. Έστω ότι έχουμε τον αρχικό κώδικα:

```

program ifWhile
{
    declare c,a,b,t;

    a:=1;
    while (a+b<1 and b<5)
    {
        if (t=1)
            c:=2;
        else
            if (t=2)
                c:=4;
            else
                c:=0;
        while (a<1)
            if (a=2)
                while(b=1)
                    c:=2;
    }
}

```

Και αυτό το πρόγραμμα δεν κάνει κάτι χρήσιμο. Αυτό που μας νοιάζει είναι ότι έχει ενδιαφέρονσα πολυπλοκότητα.

Η συντακτική ανάλυση θα ξεκινήσει και πάλι από το `program` και θα περάσει στο `declare`, χωρίς να δημιουργηθεί ενδιάμεσος κώδικας. Η πρώτη τετράδα θα παραχθεί όταν συναντήσουμε το `a:=1;` και θα είναι μία `begin_block`. Στη συνέχεια θα παραχθεί η τετράδα για το `a:=1;`:

```

1 :      begin_block, main_ifWhile, _, _
2 :      :=, 1, _, a

```

Μετά ξεκινάει η μετάφραση του `while`. Πρώτα θα αποτιμηθεί η έκφραση `a+b` και θα τοποθετηθεί στην προσωρινή μεταβλητή `T_1`. Αυτή η εντολή είναι και η πρώτη του `while`, οπότε πρέπει κάπου να φυλάξουμε τον αριθμό της τετράδας για να μπορέσουμε να κάνουμε το άλμα προς τα πίσω, όταν φτάσουμε στο τέλος του `while`. Αφού υπολογιστεί το `T_1`, μετά θα γίνει η σύγκριση:

```

3 :      +, a, b, T_1
4 :      <, T_1, 1, _
5 :      jump, _, _, -

```

Στο 6 θα μεταφραστεί το `b<5` και θα γίνει `backpatch()` το `true` της `a+b<1`, δηλαδή η 4. Ο κώδικας ως τώρα έχει γίνει:

```

1 :      begin_block, main_ifWhile, _, _
2 :      :=, 1, _, a
3 :      +, a, b, T_1
4 :      <, T_1, 1, 6
5 :      jump, _, _, -
6 :      <, b, 5, 8
7 :      jump, _, _, -

```

Έχουν μείνει ασυμπλήρωτες οι τετράδες του `false`, δηλαδή οι 5 και 7.

Η τετράδα 8 είναι η πρώτη μέσα στον βρόχο της `while` και η πρώτη της `if` και πιο συγκεκριμένα της συνθήκης `t=1`. Άρα έχουμε τις μη συμπληρωμένες τετράδες:

```

8 :      =, t, 1, _
9 :      jump, _, _, -

```

Η 8 αντιστοιχεί στο true και θα κάνει εδώ backpatch(), στο σημείο που θα τοποθετηθεί ο κώδικας για το `c := 2`.

```
8 :      =, t, 1, 10
```

Αμέσως μετά θα τοποθετηθεί το jump που θα μας βγάλει έξω από το if και θα είναι, φυσικά, ακόμα ασυμπλήρωτο. Αργότερα θα δείξει στο while(`a < 1`), όταν γνωρίσουμε τον αριθμό της τετράδας που θα δημιουργηθεί για το `a < 1`. Ο νέος κώδικας που παράγεται στο σημείο αυτό είναι οι τετράδες 10 και 11:

```
10 :      :=, 2, _, c
11 :      jump, _, _, _
```

Συνεχίζουμε με το else το οποίο έχει μέσα τον άλλο ένα if. Μέσα στο else κάνει backpatch() το false του `t=1`:

```
9 :      jump, _, _, 12
```

ενώ δημιουργούνται οι τετράδες για το `t=1`:

```
12 :      =, t, 2, _
13 :      jump, _, _, _
```

και αμέσως συμπληρώνεται το 12, αφού ακολουθεί το κυρίως σώμα του if:

```
12 :      =, t, 2, 14
```

Το κυρίως σώμα του if και το ασυμπλήρωτο jump που θα παρακάμψει το else μας δίνουν τις εξής τετράδες:

```
14 :      :=, 4, _, c
15 :      jump, _, _, _
```

Αμέσως μετά ακολουθεί ο κώδικας του else:

```
16 :      :=, 0, _, c
```

Μην ξεχάσουμε ότι στον κώδικα του else, δηλαδή στην 16, πρέπει να κάνει backpatch() το false του if που βρίσκεται στην 13:

```
13 :      jump, _, _, 16
```

Μετά ο έλεγχος κυλάει στο while(`a < 1`). Εκεί είχαμε αφήσει κάποιες τετράδες που έπρεπε να γίνουν backpatch(). Πρόκειται για τις 11 και 15 που αφορούν τετράδες που παρακάμπτουν το else:

```
11 :      jump, _, _, 17
15 :      jump, _, _, 17
```

Σημειώνουμε τη 17, διότι αφού έχουμε while θα γίνει αργότερα κάποιο άλμα προς τα πίσω στο σημείο αυτό και θα παραχθούν οι τετράδες για τη συνθήκη:

```
17 :      <, a, 1, _
18 :      jump, _, _, _
```

Στη συνέχεια γίνεται ο έλεγχος αν `a=2`, που οφείλεται στο if, ενώ σε αυτό κάνει backpatch() το true στο 17, από το while. Έτσι η 17 γίνεται:

```
17 :      <, a, 1, 19
```

και οι δύο νέες τετράδες που δημιουργούνται:

```
19 :      =, a, 2, _
20 :      jump, _, _, _
```

To 19 ως true του if θα κάνει backpatch() στο nextQuad():

```
19 :      =, a, 2, 21
```

όπου θα τοποθετηθούν οι μη συμπληρωμένες τετράδες που αντιστοιχούν στη λογική συνθήκη `b=1` του `while`.

```
21 :      =, b, 1, _
22 :      jump, _, _, _
```

Το 21 θα σημειωθεί ως πρώτη εντολή του `while` για να μπορέσουμε να κάνουμε το άλμα της επανεξέτασης της λογικής συνθήκης. Το 21 θα συμπληρωθεί με την ετικέτα 23, ως `true` της `if`:

```
21 :      =, b, 1, 23
```

ενώ ο κώδικας για το `c:=2` θα τοποθετηθεί στην 23:

```
23 :      :=, 2, _, c
```

Στη συνέχεια θα εμφανιστούν μια σειρά από `jump`. Επίσης, θα πρέπει να εκτελεστεί και μία σειρά από `backpatch()`, τόσο για τα `jump` που έχουν ήδη παραχθεί και είναι ασυμπλήρωτα, όσο και για τα ασυμπλήρωτα `jump` που θα παραχθούν παρακάτω.

Το πρώτο από αυτά τα `jump` είναι αυτό που θα επιτρέψει, μετά το σώμα του `while` (`b=1`), να επιστρέψουμε στη συνθήκη για νέο έλεγχο. Η συνθήκη ξεκινάει στο 21, άρα έχουμε:

```
24 :      jump, _, _, 21
```

Επειδή ο κώδικας έχει μεγαλώσει αρκετά και έχει γίνει και αρκετά πιο πολύπλοκος, λόγω των πολλών φωλιασμένων βρόχων, ας θυμηθούμε τι έχουμε αφήσει ασυμπλήρωτο στον κώδικα που έχουμε ήδη παραγάγει.

Η τετράδα 5 έχει ένα μη συμπληρωμένο `jump`. Το `jump` αυτό οφείλεται στο `false` του `while` (`a+b<1 and b<5`). Όμοια, στην τετράδα 7, έχουμε ένα μη συμπληρωμένο `jump` λόγω του `while` (`a+b<1 and b<5`). Η τετράδα 18 έχει για τον ίδιο λόγο ένα ασυμπλήρωτο `jump` από την την `while` (`a<1`). Το ασυμπλήρωτο `jump` της 26 οφείλεται στο `false` του `if` (`a=2`), ενώ της 22, στο `false` της `while(b=1)`.

Επιστρέφουμε στην παραγωγή νέων τετράδων. Στο σημείο της μεταγλώττισης που βρισκόμαστε ολοκληρώθηκε η μετάφραση για το σώμα της `if` (`a=2`). Αναζητώντας στον υπάρχοντα ενδιάμεσο κώδικα, μπορούμε να βρούμε ποιος πρέπει να κάνει άλμα μετά το σώμα του `if`. Δεν είναι δύσκολο να δούμε ότι το άλμα αυτό έχει προκύψει από την ψευδή αποτίμηση της συνθήκης του `if`, δηλαδή το άλμα που βρίσκεται στην εντολή 22. Έτσι, κάνουμε το κατάλληλο `backpatch()` σε αυτό:

```
22 :      jump, _, _, 25
```

Θα χρειαστεί ένα `jump` για να μην εκτελεστεί πιθανό `else`. Εδώ μπορεί να μην υπάρχει κάποιο `else`, αφού αυτό είναι προαιρετικό, αλλά το `jump` σύμφωνα με το σχέδιο ενδιάμεσου κώδικα θα παραχθεί. Και αφού δεν υπάρχει `else` θα παραχθεί `jump` στην επόμενη εντολή. Άρα:

```
25 :      jump, _, _, 26
```

Η επόμενη τετράδα που θα δημιουργηθεί είναι η 26. Βρισκόμαστε στο τέλος του βρόχου `while` (`a<1`). Χρειάζεται, δηλαδή, ένα άλμα προς τα πίσω στη συνθήκη `a<1`, η οποία βρίσκεται στο 26, άρα:

```
26 :      jump, _, _, 17
```

Αμέσως μετά τελειώνει ο βρόχος του `while` (`a+b<1 and b<5`). Άρα και εδώ, ακριβώς όπως και προηγουμένως, χρειάζεται άλμα προς τα πίσω στη συνθήκη `a+b<1 and b<5` η οποία ξεκινάει στην εντολή 3:

```
27 :      jump, _, _, 3
```

Κάπου εδώ τελειώνουν οι εντολές του προγράμματος. Για τέλος, χρειαζόμαστε μία `halt`, αφού μιλάμε για το κυρίως πρόγραμμα και μία `end_block`, όπως κάθε άλλο `block`:

```
28 :      halt, _, _, _
29 :      end_block, main_ifWhile, _, _
```

Μας έχουν μείνει δύο ασυμπλήρωτα jump, στην 5 και την 7, τα οποία οδηγούν την εκτέλεση έξω από το while ( $a+b < 1$  and  $b < 5$ ). Εκεί βρίσκεται το halt, στη γραμμή 28.

```
5 :      jump, _, _, 28
7 :      jump, _, _, 28
```

Η μετάφραση του κώδικα τελείωσε με επιτυχία. Ο ενδιάμεσος κώδικας συγκεντρωμένος ακολουθεί:

```
1 :      begin_block, main_ifWhile, _, _
2 :      :=, 1, _, a
3 :      +, a, b, T_1
4 :      <, T_1, 1, 6
5 :      jump, _, _, 28
6 :      <, b, 5, 8
7 :      jump, _, _, 28
8 :      =, t, 1, 10
9 :      jump, _, _, 12
10 :     :=, 2, _, c
11 :     jump, _, _, 17
12 :     =, t, 2, 14
13 :     jump, _, _, 16
14 :     :=, 4, _, c
15 :     jump, _, _, 17
16 :     :=, 0, _, c
17 :     <, a, 1, 19
18 :     jump, _, _, 27
19 :     =, a, 2, 21
20 :     jump, _, _, 26
21 :     =, b, 1, 23
22 :     jump, _, _, 25
23 :     :=, 2, _, c
24 :     jump, _, _, 21
25 :     jump, _, _, 26
26 :     jump, _, _, 17
27 :     jump, _, _, 3
28 :     halt, _, _, _
29 :     end_block, main_ifWhile, _, _
```

## 7.5 Διάφορες ενδιάμεσες αναπαραστάσεις

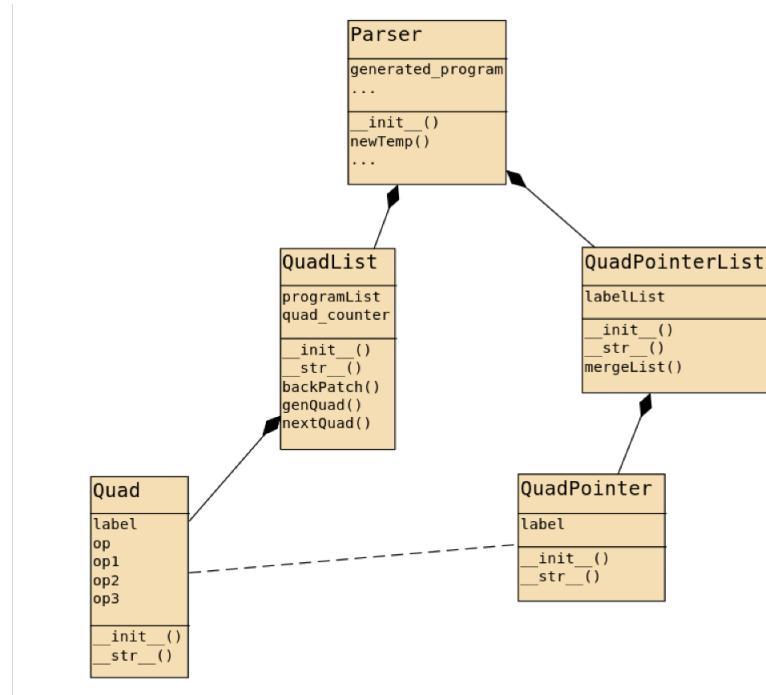
Η γλώσσα που βρίσκεται ανάμεσα στο εμπρόσθιο και το οπίσθιο τμήμα ενός μεταγλωττιστή πρέπει να είναι ανεξάρτητη από τα δύο αυτά μέρη του μεταγλωττιστή και κατάλληλα σχεδιασμένη ώστε να είναι τόσο απλή και τόσο σύνθετη όσο χρειάζεται για να καλύψει τις απαιτήσεις μας.

Κάθε γλώσσα μιας εικονικής μηχανής μπορεί να θεωρηθεί ως μία ενδιάμεση αναπαράσταση. Παραδείγματα αποτελούν οι Java bytecode για τη γλώσσα προγραμματισμού Java και Common Intermediate Language, ως κοινό επίπεδο των μεταγλωττιστών του πλαισίου .NET.

Ως ενδιάμεση αναπαράσταση μπορεί να χρησιμοποιηθεί και ένα υποσύνολο ή κάποια παραλλαγή της γλώσσας C. Η γλώσσα C είναι γλώσσα που βρίσκεται κοντύτερα στο υλικό και έχει τα χαρακτηριστικά που χρειάζονται για να χρησιμοποιηθεί ως ενδιάμεση γλώσσα. Παραδείγματα γλωσσών που το κάνουν αυτό είναι η Eiffel και η Haskell, ενώ έχουν προταθεί παραλλαγές της C ως γλώσσες ενδιάμεσης αναπαράστασης, όπως η C- και η C Intermediate Language.

## 7.6 Οι κλάσεις της φάσης της παραγωγής του ενδιάμεσου κώδικα

Η παραγωγή του ενδιάμεσου κώδικα, παρότι είναι το πιο δύσκολο τμήμα της μεταγλώττισης, του λάχιστον σε σύγκριση με ό,τι έχουμε δει μέχρι τώρα, δεν προσθέτει πολλές κλάσεις στο διάγραμμα κλάσεων του μεταγλωττιστή.



Σχήμα 7.7: Η παραγωγή ενδιάμεσου κώδικα στο διάγραμμα κλάσεων.

Δύο προφανείς κλάσεις που πρέπει να προστεθούν είναι οι κλάσεις για την τετράδα (ας την ονομάσουμε Quad) και η κλάση για τον δείκτη σε τετράδα (ας την ονομάσουμε QuadPointer, με βάση την ονομασία Quad που δώσαμε στην προηγούμενη κλάση).

Δύο ακόμα προφανώς αναγκαίες κλάσεις περιέχουν τα αντικείμενα των δύο παραπάνω κλάσεων, υλοποιώντας κάποια λίστα από αυτά τα αντικείμενα. Διατηρώντας την ίδια σύμβαση στην ονοματολογία, ας τις βαφτίσουμε QuadList και QuadPointerList, αντίστοιχα.

Δεν απαιτούνται άλλες κλάσεις για τον σχεδιασμό. Οι κλάσεις και οι σχέσεις ανάμεσα στις κλάσεις φαίνονται στο σχήμα 7.7.

Στο σχήμα αυτό, η κλάση Parser υλοποιεί τον συντακτικό αναλυτή που είδαμε εκτενώς στο κεφάλαιο 5. Εδώ σημειώνονται μόνο τα πεδία και οι μέθοδοι που σχετίζονται με την παραγωγή του ενδιάμεσου κώδικα. Έτσι, στην Parser προστίθεται το πεδίο generated\_program, το οποίο είναι τύπου QuadList και αποθηκεύει το παραγόμενο σε ενδιάμεση γλώσσα πρόγραμμα. Επίσης προστίθεται και η μέθοδος newTemp() για τη δημιουργία προσωρινών μεταβλητών.

Η κλάση QuadList είναι μία λίστα από τετράδες, από αντικείμενα της κλάσης Quad, δηλαδή. Έχει δύο πεδία: το ένα είναι λίστα με τις τετράδες (programList), ενώ το δεύτερο κρατά τον αριθμό των τετράδων που έχουν ήδη δημιουργηθεί (quad\_counter). Ο αριθμός αυτός δεν είναι ίσος με το μήκος της λίστας programList. Μετά τη δημιουργία τελικού κώδικα για κάποια συνάρτηση, από την programList θα αφαιρεθούν πεδία. Στην κλάση QuadList τοποθετούνται σχεδιαστικά και οι μέθοδοι backpatch, genQuad και nextQuad. Η σχέση ανάμεσα στον Parser και την QuadList είναι σχέση σύνθεσης.

Η κλάση Quad αναπαριστά τετράδα. Έχει τα γνωστά τέσσερα πεδία, τα οποία είναι πέντε: label, op, op1, op2, label. Η σχέση ανάμεσα στην κλάση Quad και την κλάση QuadList είναι σχέση σύνθεσης.

Στη δεξιά πλευρά του σχήματος 5, εικονίζονται οι κλάσεις QuadPointer και QuadPointerList. Ένας QuadPointer είναι ένας δείκτης σε μία τετράδα, υλοποιημένος με το πεδίο label το οποίο κρατά την ετι-

κέτα της τετράδας. Έχει σχέση συσχέτισης με την κλάση Quad και σχέση σύνθεσης με την QuadPointerList.

Η κλάση QuadPointerList είναι μία λίστα από τετράδες. Έχει σχέση σύνθεσης με την κλάση Parser, αν και η σχέση τους είναι 1:1.

## Βιβλιογραφία

- [1] Keith D. Cooper και Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2004. ISBN: 1558606998.
- [2] Keith D. Cooper και Linda Torczon. *Σχεδίαση και Κατασκευή Μεταγλωττιστών*. Ξενόγλωσσος τίτλος: Engineering a Compiler, Επιστημονική επιμέλεια έκδοσης: Γεώργιος Μανής, Νικόλαος Παπασπύρου και Αντώνιος Σαββίδης. Πανεπιστημιακές Εκδόσεις Κρήτης, 2018. ISBN: 9789605245191.
- [3] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 0321486811.
- [4] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Μεταγλωττιστές: Αρχές, τεχνικές και εργαλεία, 2η αμερικανική έκδοση*. Επιστημονική επιμέλεια έκδοσης: Π. Αλεφραγκής, Α. Βώρος, Ν. Βώρος, Κ. Μασσέλος. Εκδόσεις Νέων Τεχνολογιών, 2011. ISBN: 9789606759727.
- [5] Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.



## ΚΕΦΑΛΑΙΟ 8

---

### ΠΙΝΑΚΑΣ ΣΥΜΒΟΛΩΝ

---

#### Σύνοψη:

Στον πίνακα συμβόλων αποθηκεύουμε, κατά τη φάση της συντακτικής ανάλυσης και της παραγωγής ενδιάμεσου κώδικα, πληροφορίες σχετικά με τα συμβολικά ονόματα που χρησιμοποιούμε σε ένα πρόγραμμα: μεταβλητές, συναρτήσεις, διαδικασίες, παράμετροι, συμβολικές σταθερές κλπ. Η πληροφορία αυτή χρησιμοποιείται στη φάση της σημασιολογικής ανάλυσης αλλά και της παραγωγής του τελικού κώδικα.

Θα εξετάσουμε τη δομή που έχει ένας τέτοιος πίνακας, καθώς και το πώς εισάγουμε και αφαιρούμε εγγραφές σε αυτόν, ώστε σε κάθε σημείο της μετάφρασης ο πίνακας να έχει πρόσβαση στις εγγραφές, και μόνο σε αυτές, τις οποίες εκείνη τη στιγμή της μετάφρασης έχει δικαίωμα να προσπελάσει το υπό μετάφραση πρόγραμμα.

Για να δούμε πώς συμπληρώνουμε την πληροφορία που απαιτείται σε κάθε εγγραφή, θα θυμηθούμε τι είναι το εγγράφημα δραστηριοποίησης και πώς υπολογίζουμε το πού και πώς βρίσκεται μια μεταβλητή μέσα σε αυτό.

Θα απαριθμήσουμε και θα περιγράψουμε, με λεπτομέρεια, κάθε λειτουργία του πίνακα συμβόλων.

Με τη βοήθεια του πίνακα συμβόλων, μπορούμε να κάνουμε περισσότερους ελέγχους για την ορθότητα ενός προγράμματος από αυτούς που περιγράφηκαν με τη γραμματική της *C-imper*, όπως για παράδειγμα εάν μία μεταβλητή έχει δηλωθεί ή όχι, εάν μία μεταβλητή έχει δηλωθεί περισσότερες από μία φορές ή εάν μία εντολή *return* βρίσκεται ή όχι μέσα στο σώμα μίας συνάρτησης. Οι έλεγχοι αυτοί εντάσσονται στο πλαίσιο της σημασιολογικής ανάλυσης.

Θα αναζητήσουμε τις ευκαιρίες για αντικειμενοστραφή σχεδίαση και θα προτείνουμε έναν τρόπο ιεράρχησης κλάσεων με βάση την αντικειμενοστραφή φιλοσοφία.

Τέλος, θα δούμε ένα παράδειγμα μετάφρασης ενός προγράμματος *C-imper*, με το οποίο θα μπορέσετε να εμπεδώσετε όλες τις έννοιες του κεφαλαίου αυτού.

### Προαπαιτούμενη γνώση:

- θεωρία δομών δεδομένων
  - κεφάλαιο 5
  - κεφάλαιο 6
  - κεφάλαιο 7
- 

Ο πίνακας συμβόλων είναι δυναμική δομή στην οποία αποθηκεύεται πληροφορία σχετιζόμενη με τα συμβολικά ονόματα που χρησιμοποιούνται στο υπό μεταγλώττιση πρόγραμμα. Η δομή αυτή παρακολουθεί τη μεταγλώττιση και μεταβάλλεται δυναμικά, με την προσθήκη ή αφαίρεση πληροφορίας σε και από αυτήν, ώστε σε κάθε σημείο της διαδικασίας της μεταγλώττισης να περιέχει ακριβώς την πληροφορία που εκείνη τη στιγμή πρέπει να έχει. Λέγοντας πρέπει να έχει, εννοούμε τις εγγραφές εκείνες και μόνο αυτές, στις οποίες σύμφωνα με τους κανόνες εμβέλειας της γλώσσας, το υπό μεταγλώττιση πρόγραμμα έχει δικαίωμα να έχει πρόσβαση τη συγκεκριμένη στιγμή.

Σε έναν πίνακα συμβόλων διατηρούμε πληροφορία για τα συμβολικά ονόματα που εμφανίζονται στο πρόγραμμα. Έτσι, σε έναν πίνακα συμβόλων αποθηκεύεται πληροφορία που σχετίζεται με τις μεταβλητές του προγράμματος, τις διαδικασίες και τις συναρτήσεις, τις παραμέτρους και με τα ονόματα των σταθερών. Για καθένα από αυτά υπάρχει διαφορετική εγγραφή στον πίνακα και στην εγγραφή αυτή αποθηκεύεται διαφορετική πληροφορία, ανάλογα με το είδος του συμβολικού ονόματος. Η πληροφορία αυτή είναι χρήσιμη για τον έλεγχο σφαλμάτων, αλλά και είναι διαθέσιμη να ανακτηθεί κατά τη φάση της παραγωγής του τελικού κώδικα.

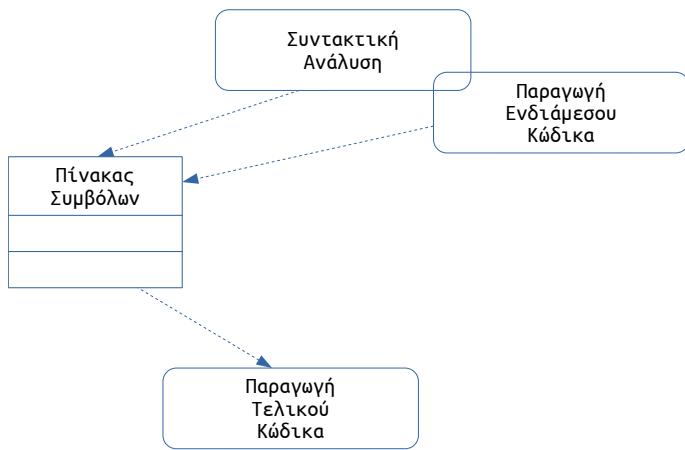
Η μορφή του πίνακα συμβόλων δεν είναι αυτή την οποία φανταζόμαστε κρίνοντας από τη λέξη πίνακας. Συνήθως ως πίνακα θεωρούμε κάποια τετραγωνική απεικόνιση πληροφορίας, κάτι δηλαδή καλά περιγεγραμμένο από μία δισδιάστατη, ομοιόμορφη αναπαράσταση. Ανάλογα με τη γλώσσα που υλοποιούμε, ο πίνακας συμβόλων μπορεί να είναι από μία απλή δομή, ως μία οργάνωση δεδομένων σε επίπεδα, με (πιθανά φωλιασμένες) λίστες και λεξικά ή αντικείμενα σε κάθε επίπεδο. Η *C-impe* έχει αυξημένες απαιτήσεις σε αυτό το σημείο και ο πίνακας συμβόλων παρουσιάζει ενδιαφέρουσα πολυπλοκότητα.

Πέρα από την αρωγή στην παραγωγή του τελικού κώδικα, με τον πίνακα συμβόλων είναι συνυφασμένη η σημασιολογική ανάλυση. Ο πίνακας συμβόλων παρέχει την πληροφορία που απαιτείται για τη σημασιολογική ανάλυση, ενώ μέρος της σημασιολογικής ανάλυσης υλοποιείται μέσα στον πίνακα συμβόλων.

Σε σχέση με τις φάσεις ανάπτυξης ενός μεταγλωττιστή, τις οποίες περιγράψαμε στο εισαγωγικό κεφάλαιο, ο πίνακας συμβόλων αντλεί πληροφορία από τις φάσεις της συντακτικής ανάλυσης και της παραγωγής ενδιάμεσου κώδικα και διαθέτει την πληροφορία που έχει συλλέξει για την παραγωγή τελικού κώδικα. Στο σχήμα 8.1 εικονίζεται η αλληλεπίδραση του πίνακα συμβόλων με τις φάσεις της συντακτικής ανάλυσης και της παραγωγής ενδιάμεσου και τελικού κώδικα.

## 8.1 Οι εγγραφές στον πίνακα συμβόλων

Ο πίνακας συμβόλων διατηρεί διαφορετική πληροφορία για κάθε είδος συμβολικού ονόματος. Σκοπός του είναι να συγκεντρώσει όλη την πληροφορία που θα απαιτηθεί να αντληθεί για το κάθε συμβολικό όνομα μέχρι το τέλος της μεταγλώττισης ή μέχρι το τέλος της ζωής του αντικειμένου. Έτσι, για παράδειγμα, στην εγγραφή που αντιστοιχίζεται σε μία μεταβλητή θα αποθηκεύσει, πέρα από το όνομά της και τον τύπο της. Στην εγγραφή μιας διαδικασίας ένα πεδίο για τον τύπο δεν έχει καμία αξία, έχει όμως αντίθετα αξία να αποθηκεύσουμε, πέρα από οτιδήποτε άλλο, τη θέση της μνήμης στην οποία είναι αποθηκευμένος ο κώδικας της διαδικασίας. Στην εγγραφή μιας παραμέτρου χρειαζόμαστε να αποθηκεύσουμε, πάλι μεταξύ άλλων, εάν η παράμετρος είναι παράμετρος που περνάει με τιμή ή με αναφορά. Ας δούμε όμως καλύτερα, για κάθε περίπτωση χωριστά, ποια είναι η πληροφορία που αποθηκεύουμε για κάθε τύπο συμβολικού ονόματος.



Σχήμα 8.1: Ο πίνακας συμβόλων.

Πριν ξεκινήσουμε να εξετάζουμε μία-μία τις εγγραφές, θα πρέπει να σημειώσουμε ότι οι εγγραφές εισάγονται στον πίνακα συμβόλων κατά τη δήλωσή τους στο αρχικό πρόγραμμα, ενώ τα πεδία τους συμπληρώνονται όταν η πληροφορία που απαιτείται για τη συμπλήρωσή τους γίνει γνωστή. Για παράδειγμα, η εγγραφή για μία συνάρτηση δημιουργείται και εισάγεται στον πίνακα συμβόλων όταν συναντούμε τη δήλωση συνάρτησης στον κώδικα του αρχικού προγράμματος, αλλά το πεδίο `startingQuad` της εγγραφής συμπληρώνεται όταν μεταφραστεί η πρώτη εκτελέσιμη τετράδα ενδιάμεσου κώδικα της υπό μετάφρασης συνάρτησης.

Οι οντότητες που θα αποτελέσουν τον πίνακα συμβόλων της *C-implied* είναι οι μεταβλητές, οι παράμετροι, οι προσωρινές μεταβλητές, οι συναρτήσεις και οι διαδικασίες. Ας τις δούμε μία-μία.

### 8.1.1 Μεταβλητή

Ας ξεκινήσουμε με τον πιο συχνά χρησιμοποιούμενο τύπο, αυτόν της μεταβλητής. Κάθε μεταβλητή έχει το όνομα που τη χαρακτηρίζει, καθώς και τον τύπο της. Δεν απαιτούν όλες οι γλώσσες προγραμματισμού από τον προγραμματιστή να ορίσει τον τύπο μιας μεταβλητής. Ανάλογα με τη φιλοσοφία της γλώσσας, ο ορισμός του τύπου της μεταβλητής μπορεί να γίνει αυτόματα, μπορεί να οριστεί και να τροποποιηθεί αργότερα τόσο από τον προγραμματιστή όσο και από τον μεταγλωττιστή. Ο πίνακας συμβόλων που υλοποιούμε οφείλει να παρακολουθήσει τις ανάγκες της γλώσσας και να προσαρμοστεί κατάλληλα. Στην *C-implied* οι τύποι των μεταβλητών ορίζονται στην αρχή του προγράμματος ή ενός υποπρογράμματος και διατηρούνται χωρίς μεταβολή όσο διαρκεί η ζωή της μεταβλητής.

Μία ακόμα πληροφορία πολύ σημαντική για κάθε μεταβλητή είναι η θέση στην οποία αυτή θα βρίσκεται στη μνήμη. Η πληροφορία αυτή είναι απαραίτητη και θα αναζητηθεί στον πίνακα συμβόλων κατά την παραγωγή τελικού κώδικα. Παρότι φαίνεται νωρίς να ασχοληθεί ο μεταγλωττιστής με κάτι τέτοιο, στην πραγματικότητα αυτή είναι η κατάλληλη στιγμή για να σχεδιαστεί ο τρόπος με τον οποίο θα γίνει η τοποθέτηση των μεταβλητών στη μνήμη.

Η φυσική θέση της μεταβλητής στη μνήμη δεν μπορεί να προσδιοριστεί με ακρίβεια, αλλά ούτε και θέλουμε να κάνουμε κάτι τέτοιο. Αρκεί να σκεφτούμε ότι, κάθε φορά που καλείται μια συνάρτηση ή μία διαδικασία, η ίδια τοπική μεταβλητή τοποθετείται σε διαφορετική θέση της μνήμης. Στην περίπτωση, μάλιστα, αναδρομικών ή φωλιασμένων κλήσεων υπάρχουν περισσότερα από ένα στιγμιότυπα της μεταβλητής ταυτόχρονα στη μνήμη.

Τότε, τελικά, τι είναι αυτό που αποθηκεύεται στον πίνακα συμβόλων; Στον πίνακα συμβόλων αποθηκεύεται η θέση της μεταβλητής μέσα στο εγγράφημα δραστηριοποίησης της συνάρτησης ή της διαδικασίας, η απόσταση δηλαδή από την αρχή του. Το εγγράφημα δραστηριοποίησης είναι ο χώρος που δίνεται σε μία συνάρτηση ή διαδικασία για να τοποθετήσει τα δεδομένα της στη μνήμη. Ο τρόπος με τον οποίο γίνεται ο υπολογισμός της απόστασης της μεταβλητής από την αρχή του θα συζητηθεί λίγο αργότερα στο κεφάλαιο

αυτό.

Τώρα ας ορίσουμε την κλάση `Variable` με την οποία μπορούμε να δημιουργούμε εισαγωγές στον πίνακα συμβόλων που αναπαριστούν μεταβλητές. Τα πεδία της κλάσης θα μπορούσαν να είναι τα ακόλουθα:

- `name`: το όνομα της μεταβλητής,
- `datatype`: ο τύπος δεδομένων της μεταβλητής,
- `offset`: η απόσταση τής μεταβλητής από την αρχή του εγγραφήματος δραστηριοποίησης.

### 8.1.2 Παράμετρος

Ένας άλλος τύπος εγγραφών σε πίνακα συμβόλων είναι η `παράμετρος`. Πρόκειται για τις παραμέτρους που περνούμε στις διαδικασίες και τις συναρτήσεις. Η παράμετρος μοιάζει πολύ με τη μεταβλητή. Ο τρόπος με τον οποίο τελικά θα διαχειριστούμε τις μεταβλητές και τις παραμέτρους είναι πολύ διαφορετικός, αλλά όσον αφορά τον πίνακα συμβόλων η πληροφορία που διατηρούμε είναι παρόμοια. Έτσι, κάθε παράμετρος έχει το όνομά της, τον τύπο της και τον τρόπο με τον οποίο περνάει. Τα δύο πρώτα πεδία είναι τα ίδια με αυτά της μεταβλητής και δεν χρειάζεται να τα αναλύσουμε περισσότερο.

Οι τρόποι περάσματος διαφέρουν από γλώσσα σε γλώσσα, αλλά οι περισσότερο συνηθισμένοι τύποι περάσματος είναι το `πέρασμα με τιμή` και το `πέρασμα με αναφορά`, οι τύποι δηλαδή που υποστηρίζονται από την `C-imple`. Θα ονομάσουμε λοιπόν ένα πεδίο `mode` το οποίο θα μπορεί να παίρνει τις τιμές `in`, `ref` και `ret`. Το πρώτο αντιστοιχεί στο πέρασμα με τιμή, το δεύτερο στο πέρασμα με αναφορά, ενώ το τρίτο είναι για επιστροφή τιμής συνάρτησης. Θυμάστε ότι είχαμε διαχειριστεί την επιστροφή τιμής συνάρτησης στον ενδιάμεσο κώδικα ως μία παράμετρο. Δεν έχουμε λόγο να το διαχειριστούμε διαφορετικά εδώ. Πρόκειται για έναν άλλο τρόπο να διαχειριστούμε επικοινωνία δεδομένων ανάμεσα στην κληθείσα και την καλούσα συνάρτηση ή διαδικασία.

Το τελευταίο πεδίο που χρειαζόμαστε είναι το `offset`. Όπως και στις μεταβλητές, πρέπει να γνωρίζουμε την απόσταση της παραμέτρου από την αρχή του εγγραφήματος δραστηριοποίησης, ώστε να μπορούμε να εντοπίζουμε την παράμετρο στη μνήμη. Το τι πληροφορία κρατάμε στη μνήμη για μια μεταβλητή ή κάθε είδος παραμέτρου δεν μας ενδιαφέρει ακόμα, αν και μπορούμε να τη φανταστούμε. Αυτό που μας ενδιαφέρει τώρα είναι ότι μέσα από το `offset` μπορούμε να εντοπίσουμε την πληροφορία αυτή.

Έτσι, συνοψίζοντας, θα ορίσουμε την κλάση `Parameter`, η οποία περιγράφει μία παράμετρο και θα μπορούσε να έχει τα ακόλουθα πεδία:

- `name`: το όνομα της παραμέτρου,
- `datatype`: ο τύπος δεδομένων της παραμέτρου,
- `mode`: ο τρόπος περάσματος της παραμέτρου,
- `offset`: η απόσταση τής παραμέτρου από την αρχή του εγγραφήματος δραστηριοποίησης.

Πολύ εύκολα μπορεί να παρατηρήσει κανείς ότι εδώ παρουσιάζονται ευκαιρίες οργάνωσης των κλάσεων σε ιεραρχία που εκφράζει κληρονομικότητα. Ας αναβάλουμε όμως αυτή τη συζήτηση έως ότου αναλύσουμε όλους τους πιθανούς τύπους εγγραφών, οπότε και θα δούμε τις δυνατότητες ιεράρχησης των κλάσεων συνολικά.

### 8.1.3 Συνάρτηση και διαδικασία

Δύο ακόμα τύποι εγγραφής στον πίνακα συμβόλων είναι η `συνάρτηση` και η `διαδικασία`. Χρησιμοποιούνται για να σημειώσουν την ύπαρξη ενός υποπρογράμματος στον κώδικα. Όπως και στις προηγούμενες εγγραφές, αποθηκεύονται στον πίνακα συμβόλων όλη την πληροφορία που σχετίζεται με το υποπρόγραμμα αυτό.

Έτσι, στο πρώτο πεδίο θα αποθηκεύσουμε το όνομα του υποπρογράμματος. Το δεύτερο πεδίο είναι μία λίστα με τις τυπικές παραμέτρους του υποπρογράμματος. Η σειρά με την οποία συναντούμε τις τυπικές παραμέτρους στο υποπρόγραμμα είναι και η σειρά με την οποία εμφανίζονται αυτές στη λίστα. Χρειαζόμαστε, λοιπόν, μία ακόμα κλάση, την κλάση των τυπικών παραμέτρων. Ας δώσουμε στην κλάση αυτή το προφανές όνομα: `FormalParameter` και ας τη μελετήσουμε αμέσως μετά.

Ένα ακόμα πεδίο που κάθε υποπρόγραμμα χρειάζεται είναι το `startingQuad`. Στο πεδίο αυτό αποθηκεύουμε την ετικέτα της πρώτης εκτελέσιμης τετράδας του ενδιάμεσου κώδικα που αντιστοιχεί στη συνάρτηση ή τη διαδικασία αυτή. Με άλλα λόγια, εκεί σημειώνουμε την τετράδα στην οποία πρέπει να καλούσα συνάρτηση να κάνει άλμα προκειμένου να εκκινήσει η εκτέλεση της κληθείσας.

Το τελευταίο κοινό πεδίο ανάμεσα σε μία συνάρτηση και μία διαδικασία είναι το `framelength`. Πρόκειται, όπως ίσως υποψιάστηκε κανείς από την ονομασία του, για το μήκος (σε bytes) του εγγραφήματος δραστηριοποίησης της συνάρτησης. Ο τρόπος υπολογισμού του μήκους του εγγραφήματος δραστηριοποίησης θα συζητηθεί σε λίγο, παρακάτω, σε επόμενη ενότητα του κεφαλαίου αυτού.

Αν πρόκειται για συνάρτηση και όχι για διαδικασία, χρειαζόμαστε ακόμα ένα πεδίο: αυτό που περιγράφει τον τύπο δεδομένων που επιστρέφει η συνάρτηση. Βέβαια, στη `C-implement` έχουμε μόνο έναν τύπο δεδομένων, οπότε το πεδίο αυτό θα έχει πάντοτε την τιμή `integer`, αλλά στη γενικότερη περίπτωση το πεδίο αυτό θα μπορεί να πάρει κάθε τύπο δεδομένων που η περιγραφή της γλώσσας ορίζει και επιτρέπει.

Έτσι, η κλάση `Procedure` θα έχει τα εξής πεδία:

- `name`: το όνομα της διαδικασίας,
- `startingQuad`: η ετικέτα της πρώτης εκτελέσιμης τετράδας της διαδικασίας,
- `framelength`: το μήκος του εγγραφήματος δραστηριοποίησης,
- `formalParameters`: η λίστα με τις τυπικές παραμέτρους της διαδικασίας.

Και η κλάση `Function` ακόμα ένα:

- `name`: το όνομα της συνάρτησης,
- `datatype`: ο τύπος δεδομένων της συνάρτησης,
- `startingQuad`: η ετικέτα της πρώτης εκτελέσιμης τετράδας της συνάρτησης,
- `framelength`: το μήκος του εγγραφήματος δραστηριοποίησης,
- `formalParameters`: η λίστα με τις τυπικές παραμέτρους της συνάρτησης.

#### 8.1.4 Τυπική παράμετρος

Αφήσαμε ένα υπόλοιπο, την κλάση `FormalParameter`, για τις τυπικές παραμέτρους. Όπως είπαμε, πρόκειται για την κλάση, στιγμιότυπα της οποίας αποτελούν αντικείμενα που περιγράφουν τη δήλωση παραμέτρων της συνάρτησης ή της διαδικασίας. Αυτά θα τοποθετηθούν σε μία λίστα, πεδίο ενός στιγμιότυπου της κλάσης `Procedure` ή `Function`. Η σειρά που θα έχει στη λίστα αυτή θα υποδηλώνει και τη σειρά την οποία θα έχει αυτή η παράμετρος στη συνάρτηση. Άρα δεν χρειαζόμαστε πεδίο της `FormalParameter` που να κρατά την πληροφορία αυτή.

Έτσι, η κλάση `FormalParameter` θα πρέπει να έχει μόνο δύο πεδία τα:

- `datatype`: ο τύπος δεδομένων της τυπικής παραμέτρου,
- `mode`: ο τρόπος περάσματος της τυπικής παραμέτρου.

Ο προσεκτικός αναγνώστης μπορεί να παρατηρήσει ότι στην κλάση αυτή δεν χρειάζεται να αποθηκεύσουμε το όνομα της παραμέτρου. Ο λόγος που διατηρούμε τη λίστα με τις τυπικές παραμέτρους είναι για να μπορέσουμε να ελέγξουμε, αργότερα, ότι μία συνάρτηση ή διαδικασία κλήθηκε όπως ακριβώς δηλώθηκε. Ως γνωστόν, το όνομα της τυπικής παραμέτρου δεν είναι υποχρεωτικό να είναι το ίδιο με το όνομα της πραγματικής παραμέτρου. Για λόγους ομοιομορφίας, αλλά και πληρότητας της πληροφορίας που τοποθετούμε στον πίνακα συμβόλων, θα επιλέξουμε να κρατήσουμε και το όνομα της τυπικής παραμέτρου σε αυτόν.

### 8.1.5 Προσωρινή μεταβλητή

Θα ορίσουμε ακόμα μία κλάση, την `TemporaryVariable`. Αυτή η κλάση, δεν διαφέρει στη βασική λειτουργικότητά της από την `Variable`, ίσως διαφέρει όσον αφορά τον κατασκευαστή της. Σχεδιαστικά, είναι σωστό να τη διαφοροποιήσουμε από την `Variable`, και θα το κάνουμε, όπως είπαμε, για πληρότητα. Τα πεδία της είναι τα ακόλουθα:

- `name`: το όνομα της προσωρινής μεταβλητής,
- `datatype`: ο τύπος δεδομένων της προσωρινής μεταβλητής,
- `offset`: η απόσταση της προσωρινής μεταβλητής από την αρχή του εγγραφήματος δραστηριοποίησης.

### 8.1.6 Συμβολική σταθερά

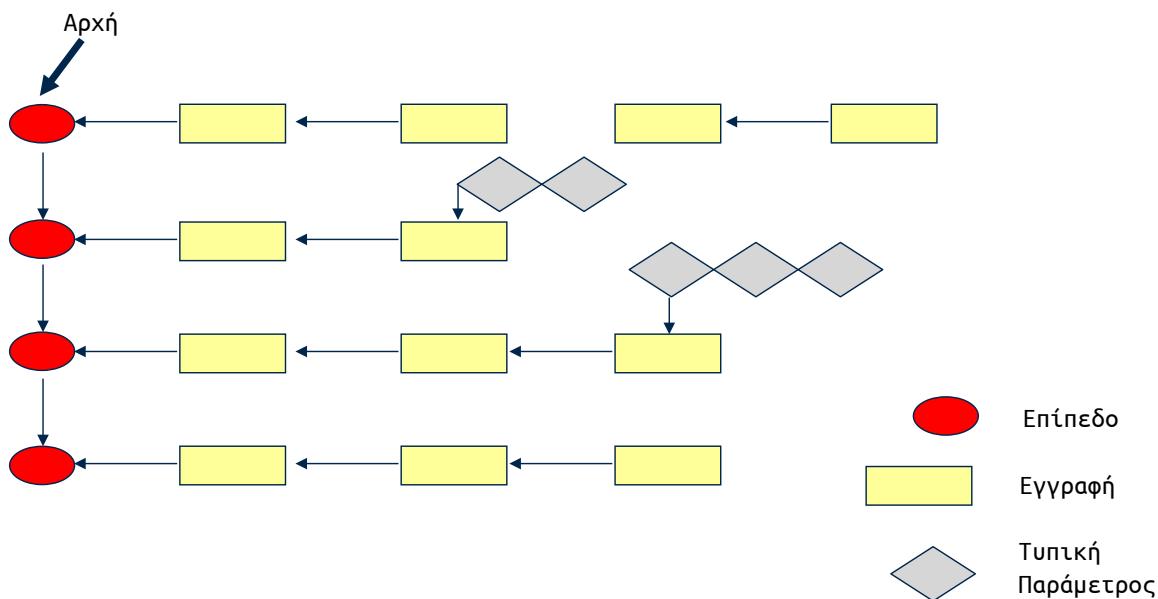
Τέλος χρειαζόμαστε ακόμα μία κλάση, την `SymbolicConstant`, για να κρατήσουμε εκεί τις τιμές των συμβολικών σταθερών. Θυμίζουμε ότι η `C-imple` δεν υποστηρίζει συμβολικές σταθερές. Οι συμβολικές σταθερές πρέπει να είναι διαθέσιμες στη φάση της μετάφρασης. Όταν ολοκληρωθεί η μετάφραση, αυτές δεν χρειάζονται πια, αφού στον τελικό κώδικα τα ονόματα των σταθερών έχουν αντικατασταθεί με τις τιμές τους. Συνεπώς δεν υπάρχει κάποιος λόγος να τοποθετηθούν στη στοίβα και να καταλάβουν άσκοπα χώρο εκεί. Για τον λόγο αυτό, οι συμβολικές σταθερές δεν χρειάζεται να έχουν πεδίο `offset`. Έτσι, τα πεδία της κλάσης `SymbolicConstant` είναι τρία:

- `name`: το όνομα της συμβολικής σταθεράς,
- `datatype`: ο τύπος δεδομένων της συμβολικής σταθεράς,
- `value`: η τιμή της συμβολικής σταθεράς.

## 8.2 Η δομή του πίνακα συμβόλων

Η δομή του πίνακα συμβόλων εξαρτάται από τη γλώσσα που υλοποιούμε. Εδώ θα δούμε τη δομή ενός πίνακα συμβόλων κατάλληλο για την `C-imple`. Η `C-imple` υποστηρίζει φωλιασμένες συναρτήσεις και διαδικασίες, κάτι που κάνει τη δομή του πίνακα συμβόλων πολύπλοκη και ενδιαφέρουσα. Αν κατανοήσετε τις απαιτήσεις και τον τρόπο που υλοποιούμε και χρησιμοποιούμε τον πίνακα συμβόλων της `C-imple`, εύκολα θα μπορέσετε να υλοποιήσετε πίνακα συμβόλων για οποιαδήποτε γλώσσα, ενώ το πιθανότερο είναι οι λειτουργίες αυτές και η δομή του πίνακα να είναι κάποιο υποσύνολο αυτών της `C-imple`.

Ο πίνακας συμβόλων αποτελείται από επίπεδα, τα οποία ας ονομάσουμε `scope`. Ο όρος σημαίνει εμβέλεια και κάθε τέτοιο επίπεδο αντιστοιχεί στη μετάφραση μίας συνάρτησης. Όταν ξεκινάει η μετάφραση μίας συνάρτησης, τότε δημιουργείται ένα νέο επίπεδο (`scope`). Όταν τερματίζεται η μετάφραση μίας συνάρτησης, τότε αφαιρείται το επίπεδό της από τον πίνακα συμβόλων. Όταν δύο συναρτήσεις `f1`, `f2` είναι φωλιασμένες η μία μέσα στην άλλη με εσωτερικότερη την `f2`, τότε κατά τη μετάφραση της `f2` στον πίνακα συμβόλων υπάρχουν τα επίπεδα και για την `f1` και για την `f2`.



Σχήμα 8.2: Η δομή του πίνακα συμβόλων.

Αν θεωρήσουμε ότι η  $f_2$  ανήκει στο κυρίως πρόγραμμα, τότε θα δημιουργηθεί πρώτα ένα επίπεδο για το κυρίως πρόγραμμα,. 'Όταν ξεκινήσει η μετάφραση της  $f_1$  θα δημιουργηθεί ένα επίπεδο για αυτήν.' Όταν ξεκινήσει η μετάφραση της  $f_2$  θα δημιουργηθεί και ένα επίπεδο για την  $f_2$ . Εποι, τη στιγμή της μετάφρασης της  $f_2$  θα υπάρχουν τρία επίπεδα στον πίνακα συμβόλων. 'Όταν τελειώσει η μετάφραση της  $f_2$ , θα αφαιρεθεί ένα επίπεδο από τον πίνακα συμβόλων, αυτό που αντιστοιχούσε στην  $f_2$ . Τη στιγμή της μετάφρασης της  $f_1$  θα υπάρχουν δύο επίπεδα στον πίνακα συμβόλων. 'Όταν τελειώσει η μετάφραση της  $f_1$ , θα αφαιρεθεί ένα επίπεδο από τον πίνακα συμβόλων, αυτό που αντιστοιχεί στην  $f_1$ . Τη στιγμή της μετάφρασης του κυρίως προγράμματος θα υπάρχει μόνο ένα επίπεδο στον πίνακα συμβόλων. Με την προσθαφαίρεση επιπέδων επιτυγχάνουμε να έχουμε μέσα στον πίνακα συμβόλων, σε κάθε στιγμή της μετάφρασης, ακριβώς την πληροφορία που πρέπει να έχει, με βάση την περιγραφή και τους κανόνες της γλώσσας.

Η δομή ενός πίνακα συμβόλων εικονίζεται στο σχήμα 8.2.

'Όλα θα ξεκαθαρίσουν με το παράδειγμα μετατροπής του πίνακα συμβόλων που υπάρχει στο τέλος του κεφαλαίου. Αλλά πριν φτάσουμε ως εκεί, έχουμε ακόμα κάποια πράγματα που πρέπει να συζητήσουμε, αλλά και να δούμε πώς μπορούμε να εκμεταλλευτούμε την αντικειμενοστρεφή σχεδίαση για να υλοποιήσουμε έναν πίνακα συμβόλων με βάση τις αρχές της.

### 8.3 Οι κανόνες εμβέλειας

Οι κανόνες εμβέλειας μιας γλώσσας υλοποιούνται στον πίνακα συμβόλων ή, αν προτιμάτε, ο πίνακας συμβόλων καθορίζει τους κανόνες εμβέλειας της γλώσσας. Για να καταλάβουμε τι εννοούμε με αυτό, πρέπει πρώτα να περιγράψουμε πώς γίνεται η αναζήτηση μίας εγγραφής στον πίνακα συμβόλων.

Η αναζήτηση εγγραφών γίνεται κυρίως στη φάση της παραγωγής του τελικού κώδικα, όπου ο μεταγλωττιστής χρειάζεται να ανακτήσει την αποθηκευμένη πληροφορία. Η αναζήτηση γίνεται με βάση το όνομα της εγγραφής και εκκινεί από το ανώτερο επίπεδο του πίνακα συμβόλων. Οι εγγραφές του επιπέδου αυτού διαπερνιούνται μία προς μία και συγκρίνονται ως προς το όνομα με την υπό αναζήτηση ονομασία. Αν η προς αναζήτηση ονομασία βρεθεί, τότε η αντίστοιχη εγγραφή επιστρέφεται ως το αποτέλεσμα της αναζήτησης. Αν στο επίπεδο αυτό δεν βρεθεί η ονομασία που αναζητείται, τότε η αναζήτηση συνεχίζεται στο αμέσως επόμενο επίπεδο. Η διαδικασία αυτή συνεχίζεται μέχρι να βρεθεί η ζητούμενη εγγραφή σε κάποιο επίπεδο ή να εξαντληθούν όλα τα επίπεδα στον πίνακα συμβόλων. Στην περίπτωση που εξαντληθούν όλα τα επίπεδα και δεν βρεθεί η ζητούμενη ονομασία, έχουμε εντοπίσει αδήλωτη μεταβλητή ή μεταβλητή εκτός

εμβέλειας και πρέπει να καλέσουμε τον διαχειριστή σφαλμάτων.

Με βάση τον παραπάνω μηχανισμό αναζήτησης μπορούμε να παρατηρήσουμε ότι ορίζονται οι παρακάτω κανόνες εμβέλειας:

- Οι τοπικές μεταβλητές και παράμετροι υπερκαλύπτουν τις μεταβλητές ή παραμέτρους με το ίδιο όνομα οι οποίες είναι δηλωμένες σε υψηλότερα επίπεδα. Εγγραφές δηλωμένες σε βαθύτερα εμφωλευμένες συναρτήσεις ή διαδικασίες υπερκαλύπτουν εγγραφές δηλωμένες σε λιγότερο βαθύτερα εμφωλευμένες συναρτήσεις ή διαδικασίες. Η iεράρχηση αυτή σταματάει στις καθολικές μεταβλητές, στις οποίες θα ανατρέξουμε αν σε κανένα από τα προηγούμενα επίπεδα δεν υπάρχει η προς αναζήτηση εγγραφή.
- Μία συνάρτηση ή διαδικασία μπορεί να καλέσει αναδρομικά τον εαυτό της, όποια συνάρτηση ή διαδικασία έχει οριστεί πριν από αυτήν στο ίδιο βάθος φωλιάσματος (είναι δηλαδή αδελφός), καθώς και τα παιδιά της. Αυτό συμβαίνει διότι για αυτές τις συναρτήσεις ή διαδικασίες ο πίνακας συμβόλων έχει συμπληρώσει όλη την πληροφορία που χρειάζεται για να κληθούν.
- Μία συνάρτηση ή διαδικασία δεν μπορεί να καλέσει τα εγγόνια της. Αυτός, άλλωστε, είναι ο λόγος που τις έχουμε “κρύψει” τα εγγόνια μέσα στα παιδιά, ώστε να μην έχουν πρόσβαση σε αυτές πέρα από τον γονέα και τα αδέλφια τους. Σε επίπεδο υλοποίησης, όταν μεταφράζεται μία συνάρτηση, τα εγγόνια της έχουν ήδη μεταφραστεί και η εγγραφή τους έχει αφαιρεθεί από τον πίνακα συμβόλων.
- Μία συνάρτηση δεν μπορεί να καλέσει τον γονέα της. Σε επίπεδο υλοποίησης, όταν μεταφράζεται μία συνάρτηση, ο γονέας της υπάρχει ως εγγραφή στον πίνακα συμβόλων, αλλά δεν έχουν συμπληρωθεί ακόμα όλα τα πεδία του.

Φυσικά, αν επιλέγαμε διαφορετικό τρόπο εισαγωγής και αναζήτησης πληροφορίας στον πίνακα συμβόλων, θα μπορούσαμε να υλοποιήσουμε μία διαφορετική πολιτική, ανάλογα με τις απαιτήσεις της υπό υλοποίησης γλώσσας.

#### 8.4 Το εγγράφημα δραστηριοποίησης

Το εγγράφημα δραστηριοποίησης (activation record) δημιουργείται για κάθε συνάρτηση ή διαδικασία που πρόκειται να κληθεί και καταστρέφεται με την ολοκλήρωση της εκτέλεσής της. Πρόκειται για τον χώρο ο οποίος παραχωρείται για τη συνάρτηση στη στοίβα. Εκεί τοποθετούνται όλες οι ζωτικές πληροφορίες για την εκτέλεσή της, συμπεριλαμβανομένων των πραγματικών παραμέτρων (actual parameters), των τοπικών μεταβλητών και των προσωρινών μεταβλητών. Ας δούμε αναλυτικά πώς είναι οργανωμένη η πληροφορία αυτή μέσα στο εγγράφημα δραστηριοποίησης.

Στην πρώτη θέση του εγγραφήματος δραστηριοποίησης αποθηκεύεται η διεύθυνση επιστροφής της συνάρτησης. Όχι βέβαια με κάποιο μαγικό τρόπο. Εμείς είμαστε υπεύθυνοι για να το κάνουμε αυτό. Στο κεφάλαιο αυτό δεν μας ενδιαφέρει πώς θα βρεθεί εκεί η διεύθυνση επιστροφής, πότε και με ποιον τρόπο θα αξιοποιηθεί. Το μόνο που μας ενδιαφέρει είναι πόσος χώρος πρέπει να δεσμευτεί για αυτήν. Έτσι, η πρώτη θέση του εγγραφήματος δραστηριοποίησης έχει το μέγεθος που απαιτείται για να αποθηκεύσουμε μία διεύθυνση στη μνήμη.

Στο σημείο αυτό θα εκμεταλλευτούμε τον επεξεργαστή που θα χρησιμοποιήσουμε και τις προδιαγραφές της *C-impl*e για να απλοποιήσουμε λίγο τις περιγραφές. Χωρίς βλάβη της γενικότητας, θα παρατηρήσουμε ότι στον επεξεργαστή RISC-V μίας διεύθυνση είναι 4 bytes και ο μοναδικός τύπος της *C-impl*e, ο ακέραιος, είναι επίσης 4 bytes. Έτσι κάθε θέση στο εγγράφημα δραστηριοποίησης θα αποτελείται από 4 bytes. Αυτό μας βοηθάει μόνο στην απλοποίηση της περιγραφής και το θεωρούμε χωρίς βλάβη της γενικότητας, αφού αν χρειαζόμασταν θέσεις στο εγγράφημα δραστηριοποίησης μικρότερες ή μεγαλύτερες των 4 bytes, αυτό δεν

Θα ήταν πρόβλημα. Απλά θα έπρεπε να το λαμβάνουμε υπόψη κάθε φορά όταν υπολογίζουμε τη διεύθυνση μιας μεταβλητής.

Στη δεύτερη θέση του εγγραφήματος δραστηριοποίησης τοποθετείται ο σύνδεσμος προσπέλασης. Ο σύνδεσμος προσπέλασης είναι ένας δείκτης ο οποίος δείχνει στο εγγράφημα δραστηριοποίησης στο οποίο πρέπει να αναζητήσει η συνάρτηση μία μεταβλητή ή παράμετρο η οποία δεν της ανήκει, αλλά στην οποία όμως, σύμφωνα με τους κανόνες εμβέλειας της γλώσσας, έχει πρόσβαση. Όταν λέμε δεν της ανήκει, εννοούμε ότι η μεταβλητή ή η παράμετρος δεν βρίσκεται στο δικό της εγγράφημα δραστηριοποίησης αλλά σε κάποιο άλλο, στο οποίο όμως, ξανατονίζουμε, έχει δικαίωμα πρόσβασης με βάση την περιγραφή της γλώσσας. Η αναζήτηση γίνεται αναδρομικά και θα τη δούμε στο κεφάλαιο 9. Να σημειώσουμε μόνο ότι στη *C-imple* κάθε συνάρτηση ή διαδικασία έχει πρόσβαση στις δικές της μεταβλητές και παραμέτρους, στις τοπικές μεταβλητές και παραμέτρους του γονέα της και σε κάθε άλλη συνάρτηση ή διαδικασία που ανήκει στο γενεαλογικό της δέντρο, συμπεριλαμβανομένων και των καθολικών μεταβλητών, κάτι που προφανώς μας προϊδεάζει για αναδρομή.

Στην τρίτη θέση του εγγραφήματος δραστηριοποίησης δεσμεύουμε χώρο για την επιστροφή τιμής της συνάρτησης. Εκεί θα αποθηκευτεί η διεύθυνση της μεταβλητής στην οποία θα επιστραφεί η τιμή της συνάρτησης. Αν το εγγράφημα δραστηριοποίησης ανήκει σε μία διαδικασία, τότε η τρίτη θέση του εγγραφήματος δραστηριοποίησης θα μείνει κενή. Αν θέλαμε να κάνουμε οικονομία στη μνήμη, θα μπορούσαμε να αναζητήσουμε σχεδιασμό στον οποίο δεν θα δεσμεύαμε καθόλου χώρο στη στοίβα για την επιστροφή τιμής, όταν μεταφράζεται (άρα και όταν εκτελείται) μια διαδικασία. Θα προτιμήσουμε να διατηρήσουμε την ομοιομορφία στη σχεδίαση και όχι την οικονομία στη μνήμη, αν και κάθε λύση έχει τα δικά της πλεονεκτήματα.

Στις τρεις θέσεις που έχουμε συνήτησει μέχρι τώρα έχουμε αποθηκεύσει τρεις διευθύνσεις, συνολικά 12 bytes. Μετά από αυτά τα 12 bytes αρχίζουμε να τοποθετούμε τις παραμέτρους και τις μεταβλητές. Πρώτα τις παραμέτρους.

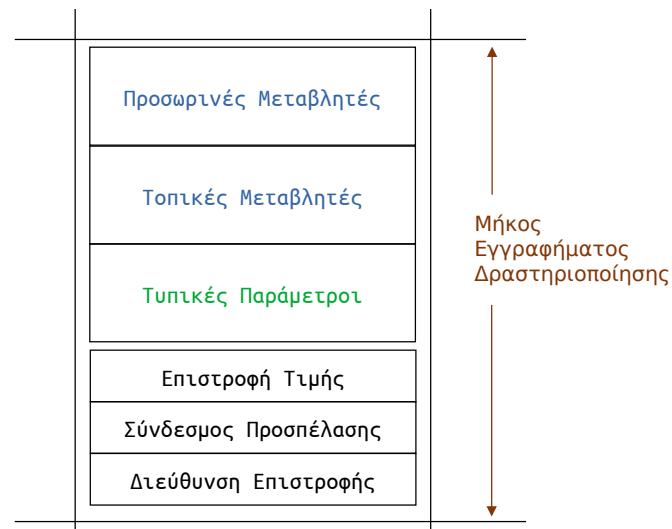
Για καθεμία παράμετρο δεσμεύουμε 4 bytes. Ανάλογα με το αν θα τοποθετηθεί εκεί παράμετρος με τιμή ή με αναφορά, στη δεσμευμένη για την παράμετρο θέση θα τοποθετηθεί η τιμή της ή η διεύθυνσή της. Αυτό όμως δεν μας απασχολεί στη φάση αυτή. Αυτό που χρειαζόμαστε τώρα να γνωρίζουμε είναι ο χώρος που θα καταλάβει η παράμετρος, ποιος θα είναι και πόσος θα είναι. Οι παράμετροι τοποθετούνται στη στοίβα με τη σειρά εμφάνισής τους στις τυπικές παραμέτρους της συνάρτησης ή της διαδικασίας και για καθεμία παράμετρο απαιτούνται 4 bytes στη *C-imple*, αφού η *C-imple* υποστηρίζει μόνο ακέραιους αριθμούς. Στη γενική περίπτωση, θα έπρεπε να δεσμεύσουμε τόσα bytes, όσα απαιτούνται κάθε φορά ανάλογα με τον τύπο δεδομένων της παραμέτρου. Ένας χαρακτήρας στη γλώσσα προγραμματισμού C, για παράδειγμα, χρειάζεται μόνο ένα byte.

Τα ίδια ισχύουν και στη δέσμευση χώρου για τις μεταβλητές. Οι μεταβλητές τοποθετούνται στη στοίβα με τη σειρά εμφάνισής τους στη δήλωση των τοπικών μεταβλητών της συνάρτησης ή της διαδικασίας και για κάθε μία μεταβλητή απαιτούνται 4 bytes, όταν πρόκειται για τη *C-imple*, ή λιγότερα ή περισσότερα αν κάποια άλλη γλώσσα έχει άλλες απαιτήσεις.

Η τελευταία ομάδα μεταβλητών που καταλαμβάνει χώρο στη στοίβα είναι οι προσωρινές μεταβλητές. Μπορεί να είναι μεταβλητές που δεν υπάρχουν μέσα στο πρόγραμμα που μεταφράζεται αλλά δημιουργούνται από τον μεταγλωττιστή για τις δικές του ανάγκες αποθήκευσης, χρειάζονται όμως χώρο, όπως όλες οι μεταβλητές. Ισχύει ό,τι και στις τοπικές μεταβλητές.

Στο σχήμα 8.3 μπορείτε να δείτε τη δομή του εγγραφήματος δραστηριοποίησης, με τα 12 δεσμευμένα bytes να βρίσκονται στην αρχή του εγγραφήματος και να ακολουθούν με σειρά οι παράμετροι, οι τοπικές και οι προσωρινές μεταβλητές.

Στο σημείο αυτό θα εισαγάγουμε δύο νέους όρους που θα χρησιμοποιήσουμε αργότερα. Ο πρώτος είναι το μήκος του εγγραφήματος δραστηριοποίησης. Πρόκειται για τον συνολικό χώρο, σε bytes, που καταλαμβάνει το εγγράφημα δραστηριοποίησης στη στοίβα. Το μήκος αυτό γίνεται γνωστό αφότου γνωρίσουμε τον αριθμό των παραμέτρων, των μεταβλητών και των προσωρινών μεταβλητών μίας συνάρτησης ή διαδικασίας, όταν δηλαδή τελεώσει η μετάφραση του ενδιάμεσου κώδικα για το υποπρόγραμμα αυτό. Το μήκος του εγγραφήματος δραστηριοποίησης σημειώνεται επίσης στο σχήμα 8.3.



Σχήμα 8.3: Η δομή του εγγραφήματος δραστηριοποίησης.

Ο δεύτερος όρος είναι η απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης (*offset*) και δεν είναι τίποτε άλλο από ότι λέει ο όρος. Πρόκειται για την απόσταση σε bytes μίας μεταβλητής ή παραμέτρου από την αρχή του εγγραφήματος. Αν πρόκειται για τη *C-implement*, για μία συνάρτηση με 2 παραμέτρους, 2 τοπικές μεταβλητές και μία προσωρινή, τα *offset* που αντιστοιχούν στις οντότητες αυτές είναι 12, 16, 20, 24 και 28 αντίστοιχα, ενώ το μήκος του εγγραφήματος δραστηριοποίησης είναι 32. Ο λόγος που η πρώτη παράμετρος τοποθετήθηκε στο 12, είναι ότι τα 12 πρώτα bytes είναι δεσμευμένα για τη διεύθυνση επιστροφής, τον σύνδεσμο προσπέλασης και την επιστροφή τιμής.

## 8.5 Οι λειτουργίες του πίνακα συμβόλων

Οι ενέργειες που γίνονται σε έναν πίνακα συμβόλων μπορούν να απαριθμηθούν εύκολα. Όπως είπαμε παραπάνω, στον πίνακα συμβόλων προσθαφαιρούμε πληροφορία, ώστε να πετύχουμε το ζητούμενο, σε κάθε σημείο της παραγωγής τελικού κώδικα ο πίνακας συμβόλων να περιέχει ακριβώς εκείνες τις εγγραφές τις οποίες το μεταφραζόμενο πρόγραμμα έχει δικαίωμα να προσπελάσει τη συγκεκριμένη χρονική στιγμή, με βάση τους κανόνες της γλώσσας.

Οι βασικές λειτουργίες του πίνακα συμβόλων είναι οι ακόλουθες:

- **Πρόσθεση νέας εγγραφής:** Προσθέτουμε μία νέα εγγραφή όταν συναντάμε τη δήλωση μιας σταθεράς, μεταβλητής, διαδικασίας ή συνάρτησης, παραμέτρου ή όταν δημιουργούμε μια νέα προσωρινή μεταβλητή. Η νέα εγγραφή προστίθεται στην τελευταία θέση του ανώτερου επιπέδου του πίνακα συμβόλων, στο επίπεδο δηλαδή της συνάρτησης ή της διαδικασίας που μεταφράζεται τη στιγμή αυτή.
- **Πρόσθεση νέου επιπέδου:** Ένα νέο επίπεδο δημιουργείται όταν ξεκινάει η μετάφραση μιας συνάρτησης, διαδικασίας ή του κυρίως προγράμματος. Το νέο επίπεδο δημιουργείται με τη λογική της στοίβας, τοποθετείται δηλαδή στον πίνακα συμβόλων πάνω από τα επίπεδα που μέχρι στιγμής έχουν δημιουργηθεί και κατά την αναζήτηση (που θα δούμε παρακάτω) προσπελάζεται πρώτο.
- **Αφαίρεση επιπέδου:** Ένα επίπεδο αφαιρείται από τον πίνακα συμβόλων όταν ολοκληρωθεί η μετάφραση μιας συνάρτησης ή διαδικασίας ή του κυρίως προγράμματος. Αφαιρείται ολόκληρο το επίπεδο, μαζί φυσικά με όλες τις εγγραφές που βρίσκονται σε αυτό. Πάντοτε αφαιρείται το ανώτερο επίπεδο, αφού η συνάρτηση/διαδικασία της οποίας η μετάφραση ολοκληρώνεται πρώτη είναι η περισσότερο εμφαλευμένη.

- **Ενημέρωση πεδίων:** Με την ενημέρωση πεδίων έχουμε τη δυνατότητα να συμπληρώσουμε κάποια πληροφορία σε μία εγγραφή του πίνακα συμβόλων, η οποία πληροφορία δεν ήταν διαθέσιμη κατά τη δημιουργία της εγγραφής. Ένα καλό παράδειγμα αποτελούν τα πεδία `framelength` και `startingQuad`. Στο χρονικό σημείο στο οποίο δημιουργείται μία εγγραφή που αντιστοιχεί σε συνάρτηση, διαδικασία ή στο κυρίως πρόγραμμα, δεν γνωρίζουμε ούτε το `framelength`, ούτε το `startingQuad`. Το `framelength` θα το γνωρίζουμε όταν ολοκληρώσουμε τη μετάφραση της συνάρτησης, οπότε και θα είναι δυνατόν να συμπληρώσουμε το αντίστοιχο πεδίο της εγγραφής. Το ίδιο συμβαίνει και με το πεδίο `startingQuad`, το οποίο γίνεται γνωστό και μπορεί να συμπληρωθεί αμέσως πριν μεταφραστεί η πρώτη εκτελέσιμη εντολή της συνάρτησης.
- **Πρόσθεση τυπικής παραμέτρου:** Η πρόσθεση μίας τυπικής παραμέτρου θα μπορούσε να ανήκει και στην κατηγορία της ενημέρωσης πεδίων, αφού πρακτικά ενημερώνουμε κάποια εγγραφή. Προτιμούμε όμως για λόγους καλύτερης δόμησης να τη διαχωρίσουμε ως λειτουργία. Η χρονική στιγμή της μετάφρασης στην οποία πραγματοποιείται η πρόσθεση τυπικής παραμέτρου, είναι κατά τη δήλωση της συνάρτησης, αμέσως μετά την πρόσθεση της εγγραφής για τη συνάρτηση στον πίνακα συμβόλων. Στην εγγραφή για τη συνάρτηση/διαδικασία έχει προβλεφθεί ένα πεδίο το οποίο μπορεί να αποθηκεύει λίστα με παραμέτρους. Η εμφάνιση μίας νέας παραμέτρου της συνάρτησης/διαδικασίας θα προσθέσει την τυπική παράμετρο στο τέλος της λίστας αυτής.
- **Αναζήτηση εγγραφής:** Η αναζήτηση εγγραφής είναι η μόνη ενέργεια με την οποία ανακτούμε πληροφορία από τον πίνακα συμβόλων. Η αναζήτηση γίνεται με βάση το όνομα της εγγραφής. Ξεκινά από το υψηλότερο επίπεδο, του οποίου οι εγγραφές διατερνιούνται μία προς μία. Αν το προς αναζήτηση όνομα βρεθεί στο επίπεδο αυτό, τότε η αντίστοιχη εγγραφή επιστρέφεται ως το αποτέλεσμα της αναζήτησης. Αν οι εγγραφές σε ένα επίπεδο τελειώσουν και δεν βρεθεί η εγγραφή που αναζητήθηκε, τότε η διαδικασία συνεχίζεται και επαναλαμβάνεται στο αιμέσως επόμενο επίπεδο. Η διαδικασία αυτή συνεχίζεται μέχρις ότου βρεθεί η εγγραφή σε κάποιο επίπεδο ή μέχρις ότου εξαντληθούν τα επίπεδα του πίνακα. Στην περίπτωση που εξαντληθούν τα επίπεδα και δεν έχει βρεθεί η ζητούμενη πληροφορία, έχουμε αναγνωρίσει σφάλμα και πρέπει να καλέσουμε τον διαχειριστή σφαλμάτων.

## 8.6 Σημασιολογική ανάλυση

Στο πλαίσιο της σημασιολογικής ανάλυσης, και με τη βοήθεια του πίνακα συμβόλων, μπορούμε να εφαρμόσουμε πρόσθετους ελέγχους για την ορθότητα ενός προγράμματος, έλεγχοι που δεν έχουν περιγραφεί από τη γραμματική της γλώσσας που χρησιμοποιήθηκε κατά τη φάση της συντακτικής ανάλυσης.

Οι πρόσθετοι αυτοί έλεγχοι δεν είναι δυνατόν να περιγραφούν από μία γραμματική χωρίς συμφραζόμενα. Θα μπορούσαν ίσως να περιγραφούν από μία γραμματική με συμφραζόμενα, αλλά δεν επιθυμούμε να χρησιμοποιήσουμε γραμματικές με συμφραζόμενα, λόγω της πολυπλοκότητάς τους, τόσο σε επίπεδο σχεδίασης, όσο και σε απαιτούμενο χρόνο εκτέλεσης της συντακτικής ανάλυσης. Έτσι, καταφεύγουμε σε ευρετικές τεχνικές με τις οποίες κάνουμε τους απαραίτητους ελέγχους.

Παρακάτω υπάρχει μία λίστα από ελέγχους που σχετίζονται με τον πίνακα συμβόλων και εντάσσονται στη διαδικασία της σημασιολογικής ανάλυσης.

- Έλεγχος αν μία μεταβλητή, παράμετρος, συνάρτηση ή διαδικασία μπορεί να προσπελαστεί ή να κληθεί στο παρόν σημείο της μεταγλώττισης. Αναζητείται η οντότητα στον πίνακα συμβόλων και, εάν δεν βρεθεί, επιστρέφεται μήνυμα σφάλματος.
- Έλεγχος αν μία μεταβλητή, παράμετρος, συνάρτηση ή διαδικασία έχει ήδη δηλωθεί. Θυμίζουμε ότι μία μεταβλητή, παράμετρος, συνάρτηση ή διαδικασία μπορεί να δηλωθεί σε διαφορετικά επίπεδα και στις περιπτώσεις αυτές ισχύει ότι περισσότερο εμφωλευμένες εγγραφές υπερκαλύπτουν τις εγγραφές σε μικρότερο βάθος φωλιάσματος. Άρα, ο έλεγχος αν μία μεταβλητή, παράμετρος, συνάρτηση ή διαδικασία έχει ήδη δηλωθεί πρέπει να γίνεται στο υψηλότερο επίπεδο του πίνακα συμβόλων μόνο.

- Έλεγχος τύπων. Οι τύποι δεδομένων που εμφανίζονται σε μία παράσταση πρέπει να συμφωνούν μεταξύ τους. Δεν μπορούμε, για παράδειγμα, να προσθέσουμε δύο πραγματικούς αριθμούς και το αποτέλεσμα να εκχωρείται σε ακέραιο ή δεν μπορούμε να προσθέσουμε έναν ακέραιο με μία συμβολοσειρά. Οι περιορισμοί αυτοί μπορούν να διαφέρουν ανάλογα με την περιγραφή της γλώσσας.
- Έλεγχος αν μία εντολή `return` βρίσκεται μέσα σε μία συνάρτηση. Αν βρίσκεται μέσα στο κυρίως πρόγραμμα ή σε διαδικασία, καλούμε τον διαχειριστή σφαλμάτων.
- Έλεγχος ότι υπάρχει τουλάχιστον μία εντολή `return` μέσα σε μία συνάρτηση.
- Έλεγχος ότι μια διαδικασία που καλείται έχει πράγματι δηλωθεί ως διαδικασία, μια συνάρτηση που καλείται έχει δηλωθεί πράγματι ως συνάρτηση και μάλιστα επιστρέφει τον σωστό τύπο δεδομένων.
- Έλεγχος ότι οι παράμετροι μιας συνάρτησης ή διαδικασίας έχουν οριστεί ακριβώς όπως καλούνται. Για τον σκοπό αυτόν θα ανατρέξουμε στο πεδίο `parameters` της συνάρτησης ή διαδικασίας και θα ελέγξουμε ένα τα αντικείμενα της λίστας των παραμέτρων αν συμφωνούν στη σειρά κλήσης, στον τρόπο περάσματος παραμέτρων και στον τύπο δεδομένων της παραμέτρου. Φυσικά, δεν μας ενδιαφέρει το όνομα της παραμέτρου, αφού πρόκειται για τυπικές παραμέτρους και δεν υπάρχει απαίτηση συμφωνίας ανάμεσα στο όνομα με το οποίο δηλώνεται και στο όνομα με το οποίο καλείται μία τυπική παράμετρος.

## 8.7 Αντικειμενοστραφής σχεδίαση

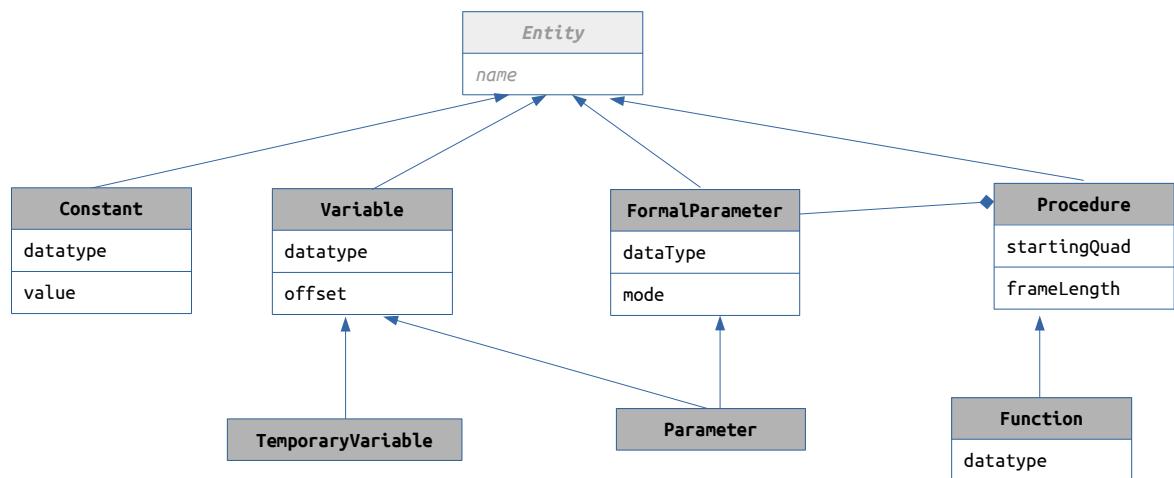
Όπως υπονοήθηκε νωρίτερα στο κεφάλαιο αυτό, αλλά και όπως σίγουρα σκεφτήκατε και εσείς διαβάζοντας τις προηγούμενες ενότητες, οι κλάσεις των διαφόρων εγγραφών του πίνακα συμβόλων δίνουν ευκαιρίες ιεραρχικής σχεδίασης και εκμετάλλευσης της κληρονομικότητας. Οι λύσεις που μπορούν να δοθούν είναι πολλές, αλλά θα περιοριστούμε στο να παρουσιάσουμε μία, η οποία μας φαίνεται πιο καλά δομημένη. Κάθε άλλη πιθανή λύση μπορεί να έχει θετικά και αρνητικά στοιχεία.

Μία αφηρημένη κλάση, η `Entity` μπορεί να τοποθετηθεί υψηλότερα στην ιεραρχία, ώστε να ενοποιήσει εννοιολογικά όλες τις εγγραφές. Υπάρχει μόνο ένα πεδίο κοινό σε όλες τις κλάσεις και αυτό είναι το `name`, οπότε η αφηρημένη κλάση `Entity` θα έχει μόνο ένα πεδίο. Κάθε εγγραφή που αντιστοιχεί σε κάποια βασική έννοια είναι λογικό να κληρονομήσει από την `Entity`. Δύο τέτοιες έννοιες είναι οι `Variable` αλλά και τα υποπρογράμματα. Κοιτώντας προσεκτικά τα πεδία των κλάσεων `Function` και `Procedure`, θα αντιληφθούμε ότι η `Function` περιέχει όλα τα πεδία της `Procedure` και ακόμα ένα. Άρα είναι λογικό η `Procedure` να είναι αυτή που θα κληρονομήσει από την `Entity` και στη συνέχεια η `Function` από την `Procedure`. Αν το δούμε σε υψηλότερο επίπεδο, μία συνάρτηση είναι μια διαδικασία η οποία μπορεί να επιστρέφει τιμή.

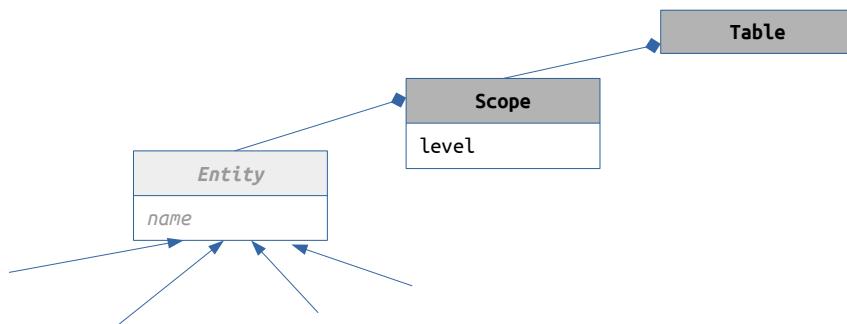
Στη συνέχεια, πρέπει να τοποθετήσουμε κάπου στο δέντρο της ιεραρχίας των κλάσεων τις κλάσεις `Parameter` και `FormalParameter`. Τα πεδία της `FormalParameter` περιέχονται στην `Parameter`, άρα η δεύτερη είναι αυτή που φαίνεται να πρέπει να κληρονομήσει από την πρώτη, η οποία με τη σειρά της θα κληρονομήσει από την `Entity`.

Ιδιαίτερο ενδιαφέρον έχει η κλάση `Parameter`. Μία παράμετρος έχει όλα τα χαρακτηριστικά μίας μεταβλητής. Επιπρόσθετα, έχει τον τρόπο περάσματος. Μία καλή σχεδιαστική επιλογή είναι να χρησιμοποιήσουμε πολλαπλή κληρονομικότητα και να δώσουμε στην κλάση `Parameter` τα πεδία της `FormalParameter` και της `Variable`. Αν τώρα η γλώσσα μας δεν υποστηρίζει πολλαπλή κληρονομικότητα, θα πρέπει η `Parameter` να κληρονομήσει την `FormalParameter` και στη συνέχεια να προστεθούν σε αυτήν τα πεδία που λείπουν.

Όπως είχαμε αποφασίσει στην προηγούμενη ενότητα, θα ορίσουμε μία κλάση για τις προσωρινές μεταβλητές, την `TemporaryVariable`. Οι προσωρινές μεταβλητές θα μπορούσαν να ανήκουν στην κλάση



Σχήμα 8.4: Η ιεραρχία των κλάσεων των εγγραφών του πίνακα συμβόλων.



Σχήμα 8.5: Οι σχέσεις ανάμεσα στις κλάσεις Table, Scope και Entity.

**Variable**, αλλά για λόγους καλύτερης σχεδίασης δεν το κάναμε αυτό. Μία εύκολη και σχεδιαστικά σωστή επιλογή είναι οι προσωρινές μεταβλητές να κληρονομούν από την κλάση **Variable**.

Μας απομένει ακόμα η **SymbolicConstant**. Είναι κάτι αρκετά διαφορετικό από τις υπόλοιπες κλάσεις, αφού ναι μεν θυμίζει μεταβλητή, αλλά δεν τοποθετείται στη στοίβα. Η τιμή της είναι γνωστή σε χρόνο μετάφρασης και θα τη χρησιμοποιήσουμε σε χρόνο μετάφρασης, ώστε να αντικαταστήσουμε το συμβολικό της όνομα με την τιμή του συμβολικού ονόματος. Όπως μας οδηγούν τα πεδία των κλάσεων **SymbolicConstant** και **Variable** αλλά και οι προδιαγραφές κάθε κλάσης, η **SymbolicConstant** θα θεωρηθεί κληρονομικά μη σχετιζόμενη με κάθε άλλη μη αφηρημένη κλάση και θα κληρονομήσει απευθείας από την **Entity**.

Η προτεινόμενη σχεδίαση απεικονίζεται στο σχήμα 8.4.

Αν θέλουμε να προχωρήσουμε λίγο περισσότερο στη σχεδίαση, θα διακρίναμε μία κλάση για τον πίνακα, την οποία θα ονομάζαμε **Table** και μία ακόμα για τα επίπεδα, την **Scope**. Η **Scope** και η **Entity** έχουν σχέση συναρμολόγησης/σύνθεσης, αφού ένα επίπεδο αποτελείται από πολλές εγγραφές, ενώ για τον ίδιο λόγο υπάρχει η ίδια σχέση ανάμεσα στην **Table** και την **Scope**. Στην **Scope** χρειαζόμαστε ακόμα ένα πεδίο το οποίο θα κρατά τον αύξοντα αριθμό του επιπέδου. Θα μπορούσαμε, βέβαια, να υλοποιήσουμε το πρόγραμμά μας και χωρίς αυτό. Οι σχέσεις των κλάσεων εικονίζονται στο σχήμα 8.5.

## 8.8 Παράδειγμα λειτουργίας του πίνακα συμβόλων

Στην ενότητα αυτή θα δούμε ένα πλήρες παράδειγμα λειτουργίας ενός πίνακα συμβόλων. Ως είσοδος θα χρησιμοποιηθεί το παράδειγμα σε γλώσσα *C-impl* που φαίνεται στο σχήμα 8.6. Το παράδειγμα είναι κατάλληλα κατασκευασμένο ώστε να γίνουν κατανοητές οι λεπτομέρειες λειτουργίας, κυρίως ο τρόπος και ο

χρόνος που εισάγονται οι εγγραφές, ο τρόπος και ο χρόνος που αφαιρούνται τα επίπεδα, ο τρόπος και ο χρόνος που συμπληρώνεται η πληροφορία στον πίνακα συμβόλων και οι χρονικές στιγμές κατά τις οποίες γίνεται αναζήτηση πληροφορίας. Το παράδειγμα δεν υλοποιεί κάτι χρήσιμο, αλλά αυτό δεν μας ενδιαφέρει καθόλου.

Ξεκινώντας από την αρχή του αρχικού προγράμματος του σχήματος 8.6, ο συντακτικός αναλυτής περνάει από το `program`, όπου και αντιλαμβάνεται ότι πρέπει να δημιουργήσει το πρώτο επίπεδο, το επίπεδο 0.

Προχωρώντας, θα συναντήσει το `const` το οποίο θα εισαγάγει στον πίνακα συμβόλων ως την πρώτη εγγραφή του επιπέδου 0. Μία διαφορετική προσέγγιση μπορεί να συναντήσει κανείς στη γλώσσα προγραμματισμού C, όπου οι `statheres` (`#define`) διαχειρίζονται από έναν προεπεξεργαστή, ο οποίος διαπερνά όλο το πρόγραμμα και αντικαθιστά τα ονόματα των σταθερών με τις τιμές τους, πριν ξεκινήσει η μεταγλώττιση του κώδικα.

Στο δικό μας παράδειγμα, μία εγγραφή σταθεράς, με όνομα A και τιμή 1, θα εισαχθεί στον πίνακα συμβόλων. Οι σταθερές δεν τοποθετούνται στη στοίβα, για τον λόγο αυτόν δεν έχουν offset. Οι σταθερές διαχειρίζονται από τον μεταγλωττιστή σε χρόνο μετάφρασης και δεν υπάρχει κανένας λόγος να τοποθετηθούν στη στοίβα. Όσο διαρκεί η μετάφραση, ο πίνακας συμβόλων είναι διαθέσιμος για να ανατρέξουμε σε αυτόν και να αναζητήσουμε την τιμή (και τον τύπο για τις γλώσσες που έχει νόημα) της σταθεράς.

Στη συνέχεια συναντώνται οι δηλώσεις για τις μεταβλητές a, b και c. Τοποθετούνται στον πίνακα συμβόλων και υπολογίζονται για τις θέσεις με offset 12, 16 και 20, οι οποίες σημειώνονται και στο αντίστοιχο πεδίο.

Με τη βοήθεια του σχήματος 8.7 μπορούμε να παρακολουθήσουμε τον σχηματισμό του πίνακα συμβόλων. Μέχρι στιγμής έχουμε δημιουργήσει ένα επίπεδο, το επίπεδο 0 και έχουμε τοποθετήσει σε αυτό τέσσερις εγγραφές. Η συντομογραφία που ακολουθείται για τις μεταβλητές είναι `name/offset` και θα τη βρείτε μέσα στο κίτρινο παραλληλόγραμμο που απεικονίζει την εγγραφή. Η συντομογραφία που χρησιμοποιούμε για τις παραμέτρους είναι `name/offset/mode`, ενώ για τις συναρτήσεις και τις διαδικασίες `name` ή `name/framelength`, όταν γνωρίζουμε το `framelength`. Στο σχήμα αυτό δεν σημειώνουμε το `startingQuad`, για λόγους διαχείρισης χώρου.

Πάνω δεξιά από το παραλληλόγραμμο που συμβολίζει τις συναρτήσεις και τις διαδικασίες, τοποθετούμε με έναν ρόμβο τις τυπικές παραμέτρους της συνάρτησης και μέσα στον ρόμβο σημειώνουμε τον τρόπο περάσματος. Για λόγους επίσης διαχείρισης χώρου, δεν σημειώνουμε το όνομα της τυπικής παραμέτρου, το οποίο άλλωστε δεν χρειάζεται. Η σειρά με την οποία εμφανίζονται οι ρόμβοι αντιστοιχεί στη σειρά με την οποία εμφανίζονται οι τυπικές παράμετροι στη δήλωση της συνάρτησης, πληροφορία που θα χρησιμοποιηθεί κατά τη σημασιολογική ανάλυση.

Στη συνέχεια συναντάμε τη διαδικασία P1. Με το που συναντάμε την έναρξη της διαδικασίας, προσθέτουμε την εγγραφή για τη διαδικασία στο τρέχον επίπεδο και δημιουργούμε ένα νέο επίπεδο για τη μετάφραση της διαδικασίας αυτής.

Έπειτα, συναντάμε την παράμετρο `in` x. Για την παράμετρο αυτή, όπως και για κάθε άλλη παράμετρο, εισάγουμε μία εγγραφή στο ανώτερο επίπεδο που έχουμε δημιουργήσει, το επίπεδο 1 στη συγκεκριμένη περίπτωση, και τοποθετούμε την τυπική παράμετρο στην εγγραφή που αντιστοιχεί στη διαδικασία. Στο παράδειγμά μας πρόκειται για τον πρώτο από τους δύο ρόμβους στην εγγραφή P1, αυτόν με την ένδειξη `in` ως τρόπο περάσματος. Συμπληρώνουμε και τα υπόλοιπα πεδία στην εγγραφή, άρα έχουμε τις τιμές `x/12/cv` για την εγγραφή αυτή. Όμοια για την παράμετρο `y` θα εισάγουμε την εγγραφή για την παράμετρο, θα τοποθετήσουμε τις τιμές `y/16/ref` σε αυτήν και θα προσθέσουμε και τον ρόμβο για την τυπική παράμετρο, στη λίστα με τις τυπικές παραμέτρους της συνάρτησης.

Μετά ακολουθεί η δήλωση της τοπικής μεταβλητής a. Οι δύο παράμετροι έχουν καταλάβει τις θέσεις 12 και 16, άρα η επόμενη διαθέσιμη θέση, την οποία και θα καταλάβει η a, είναι η 20. Εισάγουμε στον πίνακα και την εγγραφή αυτή και έχουμε τελειώσει με τις τοπικές μεταβλητές. Αν κοιτάξουμε στο σχήμα 8.7, έχουμε φτάσει στο σημείο που έχουν εισαχθεί οι πέντε εγγραφές του επιπέδου 0 και οι τρεις πρώτες εγγραφές του επιπέδου 1.

```

program symbol
{   const A=1;
    declare a,b,c;

procedure P1(in x, inout y)
{   declare a;

    function F11(in x);
    {
        declare a;
        # body of F11 #
        b = a;
        a = x;
        c = F11(in x);
        return (c);
    }

    function F12(in x)
    {
        # body of F12 #
        c = F11(in x);
        return (c);
    }

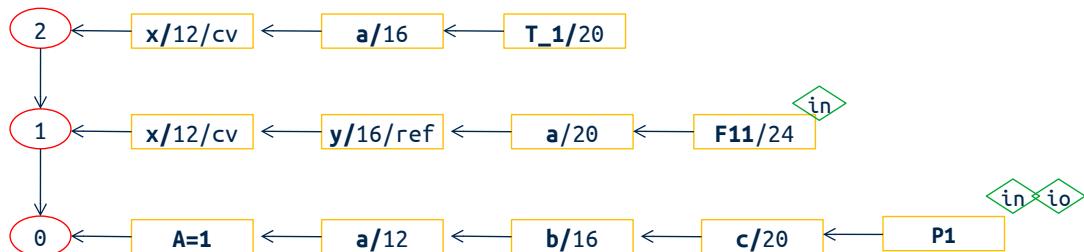
    # body of P1 #
    y = x;
}

procedure P2(inout x)
{   declare x;
    # body of P2 #
    y = A;
    call P1(in x, inout y);
}

# main program #
call P1(in a, inout b);
call P2(in c);
}

```

Σχήμα 8.6: Αρχικός κώδικας παραδείγματος λειτουργίας των πίνακα συμβόλων.



Σχήμα 8.7: Στιγμιότυπο των πίνακα συμβόλων μετά την ολοκλήρωση της μετάφρασης του ενδιάμεσου κώδικα για την F11.

Φωλιασμένη μέσα στη P1 βρίσκεται η συνάρτηση F11. Θα ακολουθήσουμε την ίδια διαδικασία. Θα δημιουργηθεί η εγγραφή για την F11 στο τρέχον επίπεδο του πίνακα συμβόλων, θα δημιουργηθεί ένα νέο επίπεδο για τη μετάφραση της συνάρτησης F11, θα δημιουργηθεί στο νέο επίπεδο η εγγραφή για την παράμετρο  $x$ , με offset=12 και mode=cv, και θα τοποθετηθεί και στη λίστα με τις παραμέτρους στην εγγραφή για τη συνάρτηση F11. Η μία και μοναδική τοπική μεταβλητή θα έχει offset=16 και θα πάρει και αυτή τη θέση της στον πίνακα συμβόλων.

Αφού ο συντακτικός αναλυτής περάσει και από τη δήλωση της a, τότε θα συναντήσει την πρώτη εκτελέσιμη εντολή της συνάρτησης F11. Τώρα είναι η κατάλληλη χρονική στιγμή για να συμπληρωθεί το startingQuad της F11. Για οικονομία χώρου δεν το σημειώνουμε στο σχήμα 8.7, αλλά η ενέργεια αυτή πρέπει να γίνει. Ακολουθώντας γραμμή προς γραμμή τον κώδικα της C-implement παράγεται ο αντίστοιχος ενδιάμεσος κώδικας για τη συνάρτηση F11. Κάθε φορά που απαιτείται η δημιουργία μιας νέας μεταβλητής, αυτή εισάγεται στην επόμενη διαθέσιμη θέση του πίνακα συμβόλων. Στο παράδειγμά μας, υπάρχει ανάγκη μονάχα για μία τέτοια στη συνάρτηση F11, την T\_1, η οποία προκύπτει από την ανάγκη της επιστροφής της τιμής της συνάρτησης, στη γραμμή με την αναδρομική κλήση. Αφού η a κατέλαβε τη θέση 16, η T\_1 αντιστοιχίζεται με τη θέση 20.

Με την ολοκλήρωση της μετάφρασης του ενδιάμεσου κώδικα για την F11, γνωρίζουμε και το μήκος του εγγραφήματος δραστηριοποίησής της. Με 12 bytes για τη διεύθυνση επιστροφής, τον σύνδεσμο προσπέλασης και την επιστροφή της τιμής, με 4 bytes για την παράμετρο, 4 για την τοπική μεταβλητή και 4 για την προσωρινή, σχηματίζεται ένα εγγράφημα δραστηριοποίησης μήκους 24 bytes. Άρα σημειώνουμε για την F11 ότι framelength=24. Η εικόνα που έχει τη στιγμή αυτή ο πίνακας συμβόλων είναι ακριβώς αυτή του σχήματος 8.7.

Το σημείο αυτό της μετάφρασης είναι πολύ χαρακτηριστικό για τη συνάρτηση F11. Αν παρατηρήσουμε τον πίνακα συμβόλων, τη στιγμή αυτή έχει όλη την πληροφορία που έχει δικαίωμα να δει η F11 και μόνο αυτήν. Έτσι, μπορεί να δει τη δική της παράμετρο  $x$ , τη δική της τοπική μεταβλητή a και την προσωρινή μεταβλητή T\_1. Από τον γονέα της μπορεί να δει την παράμετρο y που έχει περαστεί εκεί ως παράμετρος με αναφορά. Θα μπορούσε να δει και την παράμετρο  $x$  και την τοπική, στον γονέα, μεταβλητή a, αν αυτές δεν είχαν το ίδιο όνομα με τις δηλωμένες στο δικό της χώρο  $x$  και a, οι οποίες σύμφωνα με τους κανόνες εμβέλειας υπερκαλύπτουν τις ομώνυμες εγγραφές δηλωμένες στον γονέα. Φυσικά, προσωρινές μεταβλητές δηλωμένες στον γονέα δεν έχει δικαίωμα να δει η συνάρτηση F11. Άλλωστε αυτές δεν έχουν ακόμα εισαχθεί στον πίνακα συμβόλων. Τέλος, σύμφωνα πάντα με τον πίνακα συμβόλων, η συνάρτηση F11 έχει δικαίωμα να δει και τις μεταβλητές στο επίπεδο 0, τις καθολικές μεταβλητές δηλαδή. Η καθολική μεταβλητή a υπερκαλύπτεται από την τοπική μεταβλητή a, όμως, οι καθολικές μεταβλητές b και c μπορούν να προσπελαστούν κανονικά. Το ίδιο ισχύει και για τη σταθερά A, την τιμή της οποίας μπορεί να διαβάσει χωρίς κανένα πρόβλημα η συνάρτηση F11

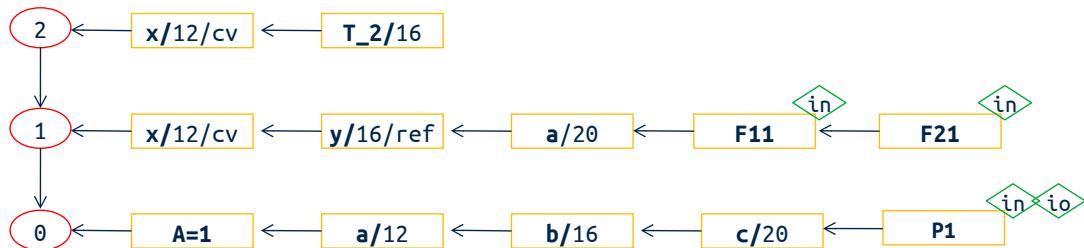
Αφήσαμε έξω από τη συζήτηση τις εγγραφές για τις F11 και P1. Σύμφωνα με τον πίνακα, η συνάρτηση F11 μπορεί να δει ότι αυτές έχουν δηλωθεί. Όμως, στο σημείο αυτό της μετάφρασης, ο πίνακας συμβόλων είναι πλήρως συμπληρωμένος μόνο για την F11, όπου και το framelength αλλά και το startingQuad είναι ήδη γνωστά. Αυτό είναι λογικό, αφού μόνο για την F11 έχει παραχθεί ο ενδιάμεσος κώδικας. Έτσι, η μόνη συνάρτηση ή διαδικασία που μπορεί να κληθεί από τη συνάρτηση F11 είναι η ίδια η F11, με αναδρομική κλήση.

Να σημειώσουμε εδώ, αν και αφορά επόμενο κεφάλαιο, ότι στο σημείο αυτό θα παραχθεί ο τελικός κώδικας για τη συνάρτηση F11. Αφού παραχθεί ο τελικός κώδικας για τη συνάρτηση F11, το επίπεδο 2, το οποίο έχει δηλαδή δημιουργηθεί για τη μετάφραση της F11, αφαιρείται από τον πίνακα συμβόλων, αφού δεν χρειάζεται πια. Άρα καμία άλλη συνάρτηση ή διαδικασία που θα μεταφραστεί αργότερα δεν θα έχει δικαίωμα να δει τις παραμέτρους και τις μεταβλητές, τοπικές ή προσωρινές, της F11, κάτι που είναι σε αρμονία με την περιγραφή της γλώσσας.

Μετά την ολοκλήρωση της μετάφρασης της F11 συνεχίζουμε με τη μετάφραση τη διαδικασία P1. Σύμφωνα με τον κώδικα, μέσα στην P1 ακολουθεί η δήλωση της συνάρτησης F12, αδελφό της συνάρτησης F11. Δημιουργούμε, λοιπόν, μία εγγραφή για τη συνάρτηση F12 και ένα επίπεδο μετάφρασης για αυτήν.

Έχει πάλι το νούμερο 2, αφού το επίπεδο με το ίδιο νούμερο αφαιρέθηκε μετά την ολοκλήρωση της F11. Η συνάρτηση F12 έχει μία παράμετρο την x, η οποία θα τοποθετηθεί στο επίπεδο 12, ως παράμετρο cv. Επίσης θα τοποθετηθεί και ως τυπική παράμετρος στην εγγραφή της συνάρτησης στο επίπεδο 1. Από το σώμα, και πιο συγκεκριμένα για την επιστροφή τιμής της F11, θα χρειαστούμε μία προσωρινή μεταβλητή, η οποία επίσης θα εισαχθεί στον πίνακα συμβόλων, στο επίπεδο 2 και με το επόμενο διαθέσιμο offset, το οποίο είναι το 16.

Τη χρονική στιγμή αυτή, μετά την παραγωγή του ενδιάμεσου κώδικα για τη διαδικασία F12, πάλι βρισκόμαστε στο σημείο που ο πίνακας συμβόλων είναι συμπληρωμένος για την F12, άρα και μπορεί να παραχθεί ο τελικός κώδικας για αυτήν. Η εικόνα του πίνακα συμβόλων κατά την παραγωγή του τελικού κώδικα είναι αυτή που φαίνεται στο σχήμα 8.8.



Σχήμα 8.8: Στιγμιότυπο του πίνακα συμβόλων μετά την ολοκλήρωση της μετάφρασης του ενδιάμεσου κώδικα για την F12.



Σχήμα 8.9: Στιγμιότυπο του πίνακα συμβόλων μετά την ολοκλήρωση της μετάφρασης του ενδιάμεσου κώδικα για την P2.



Σχήμα 8.10: Στιγμιότυπο του πίνακα συμβόλων μετά την ολοκλήρωση της μετάφρασης του ενδιάμεσου κώδικα για το κυρίως πρόγραμμα.

Με το τέλος της μετάφρασης της F12, αφαιρείται το επίπεδο 2 από τον πίνακα συμβόλων. Παρατηρήστε ότι, παρότι έχουν αφαιρεθεί τα επίπεδα που αντιστοιχούν στη μετάφραση των συναρτήσεων F11 και F12, οι εγγραφές τους από το επίπεδο 1 δεν έχουν αφαιρεθεί. Συνεπώς, αν η διαδικασία P1 θελήσει να τις καλέσει, θα μπορεί να βρει ό,τι πληροφορία χρειάζεται για αυτές.

Η μετάφραση θα συνεχιστεί με την P1. Δεν χρειαζόμαστε προσωρινές μεταβλητές εδώ. Όταν ολοκληρωθεί η μετάφραση του ενδιάμεσου και του τελικού κώδικά της, το επίπεδο 1 θα αφαιρεθεί από τη μνήμη. Γυρίζοντας στις συναρτήσεις F11 και F12 βλέπουμε ότι κανείς δεν μπορεί πια να τις καλέσει, σύμφωνα με την περιγραφή της γλώσσας.

Η μετάφραση της διαδικασίας P2 θα δημιουργήσει νέα εγγραφή στο επίπεδο 0 και νέο επίπεδο 1, για τη μετάφραση. Στο τέλος της παραγωγής του ενδιάμεσου κώδικα για την P2, η εικόνα του πίνακα συμβόλων θα είναι αυτή του σχήματος 8.9.

Με την ολοκλήρωση της μετάφρασης της διαδικασίας P2, θα αφαιρεθεί το επίπεδό της από τον πίνακα. Η μετάφραση του κυρίως προγράμματος δεν θα δημιουργήσει προσωρινές μεταβλητές.

Στο τέλος της μετάφρασης του ενδιάμεσου κώδικα για το κυρίως πρόγραμμα, ο πίνακας συμβόλων αποτελείται από ένα μόνο επίπεδο το οποίο φαίνεται στο σχήμα 8.10.

Παρατηρήστε ότι το κυρίως πρόγραμμα μπορεί να δει και να καλέσει μόνο τις διαδικασίες P1 και P2. Με την αφαιρέση των υψηλότερων επιπέδων αφαιρέθηκαν και οι εγγραφές με τις συναρτήσεις F11, F12 και το κυρίως πρόγραμμα δεν έχει τη δυνατότητα να τις δει, κάτι που είναι συμβατό με τους κανόνες εμβέλειας που ορίζει η *C-imple*.

Με το τέλος της μετάφρασης του τελικού κώδικα του κυρίως προγράμματος, θα αφαιρεθεί και το τελευταίο επίπεδο από τον πίνακα και ο πίνακας συμβόλων θα μείνει κενός.

Για τον πίνακα συμβόλων μπορείτε επίσης να ανατρέξετε στο [1, κεφ.5], ενώ για σημασιολογική ανάλυση στο [1, κεφ.6]

## Βιβλιογραφία

- [1] Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.

## ΚΕΦΑΛΑΙΟ 9

---

### ΠΑΡΑΓΩΓΗ ΤΕΛΙΚΟΥ ΚΩΔΙΚΑ

---

#### Σύνοψη:

Η παραγωγή του τελικού κώδικα είναι το στάδιο της μεταγλώττισης, όπου παράγεται ο κώδικας σε γλώσσα μηχανής. Η φάση της παραγωγής του τελικού κώδικα έχει χωριστεί σε δύο κεφάλαια στο βιβλίο αυτό, λόγω της έκτασής τους. Σε αυτό το κεφάλαιο, που είναι το πρώτο, παρουσιάζεται ότι αφορά την παραγωγή του τελικού κώδικα, αλλά δεν εντάσσεται στη θεματολογία της μεταγλώττισης συναρτήσεων και διαδικασιών. Το δεύτερο κεφάλαιο, αυτό που ακολουθεί, ασχολείται κυρίως με τη μεταγλώττιση συναρτήσεων και διαδικασιών.

Θα μελετήσουμε τα κυριότερα σημεία της μετάφρασης κώδικα από ενδιάμεση σε τελική γλώσσα. Θα δούμε τον τρόπο με τον οποίο παράγεται η γλώσσα μηχανής για καθεμία από τις εντολές της γλώσσας που χρησιμοποιούμε για την ενδιάμεση αναπαράσταση. Θα δώσουμε ιδιαίτερη έμφαση στην απεικόνιση των μεταβλητών στη μνήμη και στον τρόπο αναζήτησής τους, ανάλογα με το είδος της κάθε μεταβλητής. Ο τρόπος αποθήκευσης μιας μεταβλητής και ο τρόπος πρόσβασης διαφέρουν αρκετά αν έχουμε τοπικές μεταβλητές, καθολικές μεταβλητές ή μεταβλητές που ανήκουν στους προγόνους μιας συνάρτησης ή διαδικασίας, χωρίς να είναι καθολικές.

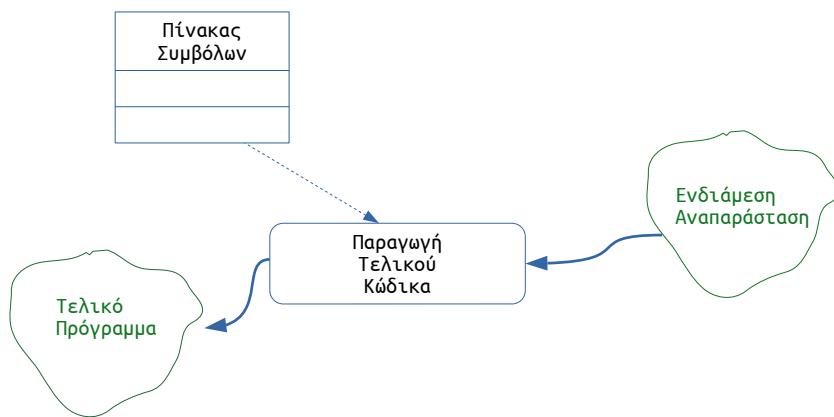
Στην αρχή του κεφαλαίου παρουσιάζεται η γλώσσα μηχανής του επεξεργαστή RISC-V. Η περιγραφή περιορίζεται στα στοιχεία της γλώσσας που θα φανούν χρήσιμα στην παραγωγή του κώδικα. Στη συνέχεια, θα περιγράψουμε τις βοηθητικές συναρτήσεις `gn1vcode()` για την πρόσβαση σε μεταβλητές που ανήκουν σε προγόνους, `loadvr()` για τη φόρτωση μεταβλητών από τη μνήμη σε καταχωρητές και `storevr()` για την αποθήκευση μεταβλητών στη μνήμη.

Τέλος, περιγράφεται η παραγωγή τελικού κώδικα για τις εκχωρήσεις, τις αριθμητικές πράξεις, τις διακλαδώσεις, την αρχή και το τέλος προγράμματος και τις εντολές εισόδου-εξόδου.

### Προαπαιτούμενη γνώση:

- στοιχεία γλώσσας μηχανής
  - κεφάλαιο 6
  - κεφάλαιο 7
  - κεφάλαιο 8
- 

Η τελευταία φάση της παραγωγής κώδικα είναι η παραγωγή του τελικού κώδικα. Ο τελικός κώδικας προκύπτει από τον ενδιάμεσο κώδικα με τη βοήθεια του πίνακα συμβόλων. Συγκεκριμένα, από κάθε εντολή ενδιάμεσου κώδικα προκύπτει μία σειρά εντολών τελικού κώδικα, η οποία για να παραχθεί ανακτά πληροφορίες από τον πίνακα συμβόλων. Η λειτουργία της φάσης της παραγωγής τελικού κώδικα φαίνεται στο σχήμα 9.1.



Σχήμα 9.1: Η παραγωγή τελικού κώδικα στη διαδικασία της μεταγλωττισης.

Η τελική γλώσσα που δημιουργείται από έναν μεταγλωττιστή είναι συνήθως η γλώσσα μηχανής ενός επεξεργαστή. Για τη *C-imple* θα παραγάγουμε τελικό κώδικα σε συμβολική γλώσσα μηχανής (*assembly code*) του επεξεργαστή RISC-V [1, 2].

Ο RISC-V είναι ένας αρκετά αντιπροσωπευτικός επεξεργαστής της τεχνολογίας RISC. Επιλέχθηκε ως επεξεργαστής στόχος για την ανάπτυξη ενός εκπαιδευτικού μεταγλωττιστή, διότι εκτός από αντιπροσωπευτικός, είναι και αρκετά απλός, υποστηρίζοντας όλες τις λειτουργίες που θα ζητούσαμε για την ανάπτυξη του μεταγλωττιστή της *C-imple*. Χρησιμοποιείται σε πολλά Πανεπιστημιακά Τμήματα για μαθήματα κυρίως αρχιτεκτονικής υπολογιστών και αποτελεί ένα επαρκές υποκείμενο σύστημα υλικού για μαθήματα μεταγλωττιστών.

Στο παρόν σύγγραμμα δεν μας ενδιαφέρει να μελετήσουμε σε βάθος τον επεξεργαστή, ούτε και να προσαρμόσουμε την παραγωγή τελικού κώδικα σε αυτόν. Ο τελικός κώδικας θα σχεδιαστεί βασισμένος σε όσο το δυνατόν λιγότερη εξάρτηση από το υλικό και τις ιδιαιτερότητες του RISC-V. Οι εντολές της συμβολικής γλώσσας μηχανής που θα επιστρατεύσουμε από το διαθέσιμο σύνολο εντολών του RISC-V, υποστηρίζονται από όλους τους επεξεργαστές, με μικρές και μη σημαντικές διαφοροποιήσεις. Η γλώσσα *C-imple* υποστηρίζει μόνο ακέραιους αριθμούς και αυτό έχει το πρόσθετο πλεονέκτημα ότι περιορίζει το σύνολο των εντολών που είναι αναγκαίες.

Παρακάτω θα κάνουμε μία παρουσίαση των εντολών της συμβολικής γλώσσας μηχανής του RISC-V τις οποίες θα χρησιμοποιήσουμε. Φυσικά, δεν αποτελεί παρουσίαση της γλώσσας μηχανής του RISC-V. Περισσότερα για τη γλώσσα μηχανής του RISC-V μπορείτε να βρείτε εδώ [1], ενώ οι αναγνώστες που ενδιαφέρονται περισσότερο μπορούν βρουν χρήσιμη πληροφορία και εδώ [2]. Για πρόσθετη βιβλιογραφία στην παραγωγή τελικού κώδικα μπορείτε να ανατρέξετε και εδώ: [3, 4, κεφ.6-7], [5, 6, κεφ.8], [7, κεφ.9].

## 9.1 Η συμβολική γλώσσα μηχανής του RISC-V

### 9.1.1 Οι καταχωρητές του RISC-V

Ο RISC-V διαθέτει 32 καταχωρητές για ακέραιους αριθμούς και 32 καταχωρητές για αριθμούς κινητής υποδιαστολής. Οι καταχωρητές των ακεραίων αριθμών ονομάζονται  $x_0, x_1, \dots, x_{31}$ , αλλά καθένας από αυτούς έχει ένα μνημονικό όνομα, ανάλογα με τη λειτουργία που συνηθίζεται να κάνει. Την αντιστοιχία, αν σας ενδιαφέρει, μπορείτε εύκολα να τη βρείτε, εμείς θα τους παρουσιάσουμε με τα μνημονικά τους ονόματα:

- **zero** (μηδενικός καταχωρητής): Καταχωρητής που έχει μόνιμα την τιμή 0. Η συχνή χρήση της σταθεράς 0 οδήγησε στην απόφαση ένας καταχωρητής να αφιερωθεί για τον σκοπό αυτόν και να έχει πάντοτε την τιμή 0.
- **sp** (stack pointer - δείκτης στοίβας): Καταχωρητής που δείχνει στη στοίβα. Σκοπός του είναι να σημειώνει την αρχή του εγγραφήματος δραστηριοποίησης της συνάρτησης ή της διαδικασίας που κάθε στιγμή εκτελείται.
- **fp** (frame pointer - δείκτης πλαισίου): Καταχωρητής που δείχνει στη στοίβα. Θα τον χρησιμοποιήσουμε για να δείξουμε την αρχή ενός εγγραφήματος δραστηριοποίησης, το οποίο εκείνη τη στιγμή δημιουργείται.
- **t0-t6** (temporary registers - προσωρινοί καταχωρητές): Καταχωρητές που μπορούν να αξιοποιηθούν για οποιονδήποτε σκοπό. Το γράμμα  $t$  προέρχεται από τη λέξη *temporary*. Ως προσωρινοί καταχωρητές θα χρησιμοποιηθούν και στο σύγγραμμα αυτό, για πολλαπλούς σκοπούς.
- **s1-s11** (saved registers - καταχωρητές διατηρούμενων τιμών): Οι τιμές των καταχωρητών αυτών διατηρούνται ανάμεσα σε κλήσεις συναρτήσεων και διαδικασιών.
- **a0-a7** (function arguments registers - καταχωρητές ορισμάτων συναρτήσεων): Χρησιμοποιούνται προκειμένου να περαστούν παράμετροι ανάμεσα σε συναρτήσεις ή διαδικασίες. Μπορούν ακόμα να χρησιμοποιηθούν για επιστροφή τιμών.
- **ra** (return address - διεύθυνσης επιστροφής): Στον καταχωρητή αυτόν αποθηκεύεται η διεύθυνση στην οποία πρέπει να επιστρέψει ο έλεγχος του προγράμματος, όταν ολοκληρωθεί η εκτέλεση μιας συνάρτησης ή διαδικασίας. Η διεύθυνση τοποθετείται στον καταχωρητή από τον επεξεργαστή με την κλήση της συνάρτησης ή της διαδικασίας.
- **pc** (program counter - μετρητής προγράμματος): Περιέχει τη διεύθυνση της εντολής που εκτελείται.
- **gp** (global pointer - καθολικός δείκτης): Δείχνει στην αρχή των μεταβλητών που είναι καθολικές σε ένα πρόγραμμα, ώστε να μπορούμε να έχουμε ευκολότερη και ταχύτερη πρόσβαση σε αυτές.
- **tp** (thread pointer - δείκτης στα δεδομένα ενός νήματος, δεν θα τον χρειαστούμε).

Οι καταχωρητές μπορούν να πάρουν τιμή είτε με απευθείας εκχώρηση αριθμητικής σταθεράς σε αυτούς, είτε με μεταφορά δεδομένων από έναν καταχωρητή σε έναν άλλον, είτε με τέλεση κάποιας αριθμητικής πράξης.

Η απευθείας εκχώρηση αριθμητικής σταθεράς σε έναν καταχωρητή γίνεται με την ψευδοεντολή **li**:

```
li reg, int    # reg = int
    # reg: destination register
    # int: arithmetic integer constant
```

Για παράδειγμα η εντολή:

```
li t0,3
```

εκχωρεί το 3 στον καταχωρητή t0. Το γεγονός ότι πρόκειται για ψευδοεντολή και υλοποιείται στην πραγματικότητα μέσω κάποιας άλλης εντολής του RISC-V έχει ενδιαφέρον σε επίπεδο αρχιτεκτονικής υπολογιστών, αλλά δεν μας απασχολεί σε επίπεδο μεταγλωττιστών.

Η μεταφορά τιμής από έναν καταχωρητή σε έναν άλλο γίνεται με την ψευδοεντολή mv:

```
mv reg1,reg2    # reg1 = reg2
# reg1: destination register
# reg2: source register
```

Στο παρακάτω πρόγραμμα οι καταχωρητές t1 και t2 εναλλάσσουν τις τιμές τους, μέσω του καταχωρητή t0:

```
mv t0,t1
mv t1,t2
mv t2,t0
```

### 9.1.2 Πράξεις μεταξύ ακέραιων αριθμών

Για τις τέσσερις αριθμητικές πράξεις ανάμεσα σε καταχωρητές είναι διαθέσιμες οι εξής εντολές:

```
add reg, reg1, reg2    # reg = reg1 + reg2
sub reg, reg1, reg2    # reg = reg1 - reg2
mul reg, reg1, reg2    # reg = reg1 * reg2
div reg, reg1, reg2    # reg = reg1 / reg2
# reg: destination register
# reg1,reg2: registers (operands)
```

όπου τα reg1,reg2,reg3 είναι καταχωρητές.

Πρόσθεση μίας ακέραιας σταθεράς σε έναν καταχωρητή γίνεται με την addi:

```
addi target_reg, source_reg, int    # target_reg = source_reg + int
# target_reg, source_reg: registers
# int: arithmetic integer constant
```

Η διαίρεση μεταξύ ακέραιων επιστρέφει ακέραιο αριθμό, ενώ υπάρχει και η rem η οποία επιστρέφει το υπόλοιπο της διαίρεσης.

Στο παρακάτω πρόγραμμα οι καταχωρητές t1 και t2 εναλλάσσουν τις τιμές τους, χωρίς τη χρήση του καταχωρητή t0:

```
add t1, t1, t2
sub t2, t1, t2
sub t1, t1, t2
```

### 9.1.3 Η πρόσβαση στη μνήμη

Για την πρόσβαση στη μνήμη χρησιμοποιούμε τις εντολές lw και sw. Το lw συμβολίζει την ανάγνωση, το s την εγγραφή, ενώ το w ότι πρόκειται να διαβάσουμε ή να γράψουμε δεδομένα μεγέθους μίας λέξης (4 bytes). Υποστηρίζονται εντολές για εγγραφή ή ανάγνωση ενός byte (lb,sb), μισής λέξης, δηλαδή δύο bytes, (lh,sh), δύο λέξεων, δηλαδή οκτώ bytes (ld, sd), τις οποίες, όμως, δεν θα χρειαστούμε.

Η πρόσβαση στη μνήμη γίνεται μέσω ενός καταχωρητή. Για τον σκοπό αυτό χρησιμοποιούμε συνήθως τον καταχωρητή sr ή τον fp. Πολλές φορές θα χρησιμοποιήσουμε και τον t0. Όπως είπαμε, με τις εντολές lw και sw διαβάζουμε και γράφουμε μία λέξη στη μνήμη. Μας ενδιαφέρει η πρόσβαση μέσω καταχωρητή. Στα σχόλια, παρακάτω, τα σύμβολα [ ] αναπαριστούν έμμεση αναφορά, δηλαδή ο συμβολισμός [x] υποδηλώνει ότι θα προσπελάσουμε το περιεχόμενο της μνήμης στη θέση x:

```

lw reg1,offset(reg2)    # reg1 = [reg2 + offset]
# reg1: destination register
# reg2: base register
# offset: distance from reg2
sw reg1,offset(reg2)    # [reg2 + offset] = reg1
# reg1: source register
# reg2: base register
# offset: distance from reg2

```

Η πρόσβαση στη μνήμη γίνεται ως εξής: λαμβάνοντας ως βάση τον καταχωρητή reg2, μετακινούμαστε κατά offset θέσεις και στο σημείο που μετακινηθήκαμε γράφουμε (αν πρόκειται για sw) ή διαβάζουμε (στην περίπτωση της lw) την τιμή του reg1.

Το παρακάτω πρόγραμμα προσθέτει τους ακεραίους που είναι τοποθετημένοι στις θέσεις 12 και 16 bytes κάτω από τον δείκτη στοίβας sp και τοποθετεί το αποτέλεσμα στη θέση 20 bytes κάτω από τον sp:

```

lw t1,-12(sp)
lw t2,-16(sp)
add t1,t1,t2
sw t1,-20(sp)

```

Ένας άλλος τρόπος πρόσβασης είναι η πρόσβαση μέσω καταχωρητή, χωρίς να δηλωθεί κάποιο offset:

```

lw reg1,(reg2)    # reg1 = [reg2]
# reg1: destination register
# reg2: base register
sw reg1,(reg2)    # [reg2] = reg1
# reg1: source register
# reg2: base register

```

Ο συμβολισμός ( $t\theta$ ) είναι ισοδύναμος με τον συμβολισμό  $\theta(t\theta)$ .

#### 9.1.4 Εντολές διακλαδώσεων

Στη συνέχεια θα περιγραφούν οι εντολές άλματος που θα χρησιμοποιήσουμε. Η εντολή για άλμα χωρίς συνθήκη είναι η b (ή ισοδύναμα η j):

```

b label
# label: an address

```

Το ακόλουθο πρόγραμμα υλοποιεί ατέρμονο βρόχο, αν στο τμήμα με τις τελείες δεν υπάρχει εντολή άλματος ή τερματισμού:

```

mylabel:
...
b mylabel

```

Ενδιαφέρον έχουν οι εντολές άλμάτων υπό συνθήκη, αφού με αυτές υλοποιούμε τις λογικές παραστάσεις.

```

beq reg1,reg2,label    # branch if equal
bne reg1,reg2,label    # branch if not equal
blt reg1,reg2,label    # branch if less than
bgt reg1,reg2,label    # branch if greater than
ble reg1,reg2,label    # branch if less or equal than
bge reg1,reg2,label    # branch if greater or equal than
# reg1,reg2: the registers to be compared
# label: the address to jump to

```

Κάθε μία από τις παραπάνω εντολές άλματος υπό συνθήκη ελέγχει τους δύο καταχωρητές reg1, reg2 στο πρώτο και στο δεύτερο όρισμά της, και ανάλογα με το αποτέλεσμα της σύγκρισης και το είδος της συνθήκης που εκφράζει η εντολή, πραγματοποιείται το άλμα στην ετικέτα label ή όχι.

Ας δούμε ένα παράδειγμα. Αν θεωρήσουμε ότι στον καταχωρητή t0 έχουμε την τιμή της διακρίνουσας (και κάνουμε την παραδοχή ότι είναι ακέραια) ενός τριωνύμου, τότε το παρακάτω πρόγραμμα υπολογίζει τις ρίζες του.

```
beq t0,zero,equal
bgt t0,zero,positive
negative:
...
b exit
equal:
...
b exit
positive:
...
exit:
...
```

Θα αναφέρουμε ακόμα ένα είδος άλματος, αυτό που γίνεται μέσω καταχωρητή και αντιστοιχεί στην εντολή jr.

```
jr reg # jump [reg]
# reg: a register
```

Ας υποθέσουμε ότι στον καταχωρητή t0 βρίσκεται αποθηκευμένη μία διεύθυνση, για παράδειγμα η διεύθυνση της πρώτης εντολής μίας συνάρτησης ή διαδικασίας. Τότε με την εκτέλεση της:

```
jr t0
```

ο έλεγχος θα μεταφερθεί σε αυτήν τη συνάρτηση ή διαδικασία.

### 9.1.5 Κλήση συνάρτησης ή διαδικασίας

Για την κλήση μίας συνάρτησης ή μίας διαδικασίας υποστηρίζεται η εντολή jal. Η jal παίρνει ως όρισμα μία διεύθυνση και εκτελεί άλμα στη διεύθυνση αυτή. Ταυτόχρονα τοποθετεί στον καταχωρητή ra τη διεύθυνση της εντολής που ακολουθεί την jal στον υπό μετάφραση κώδικα.

Η εντολή jal διευκολύνει στην κλήση συναρτήσεων και διαδικασιών. Αν καλέσουμε την jal με την πρώτη εντολή μίας συνάρτησης ή διαδικασίας, τότε ο έλεγχος θα μεταφερθεί σε αυτή τη συνάρτηση ή τη διαδικασία και στον ra θα υπάρχει έτοιμη η διεύθυνση στην οποία θα επιστρέψουμε όταν ολοκληρωθεί η εκτέλεση της κληθείσας. Υπάρχουν, βέβαια, και άλλα βήματα που πρέπει να γίνουν κατά την κλήση συναρτήσεων ή διαδικασιών, αλλά η τοποθέτηση της διεύθυνσης επιστροφής στο ra είναι σημαντική διευκόλυνση.

Θα ήταν καλύτερο να περιμένουμε για να δούμε κάποιο παράδειγμα στο αντίστοιχο κεφάλαιο (κεφ. 10) για την παραγωγή τελικού κώδικα για την κλήση συναρτήσεων και διαδικασιών.

### 9.1.6 Είσοδος και έξοδος δεδομένων

Η είσοδος δεδομένων από το πληκτρολόγιο γίνεται μέσω των καταχωρητών ορισμάτων a0 και a7. Μόλις τα ορίσματα τοποθετηθούν στους δύο καταχωρητές, τότε καλείται η εντολή ecall, ώστε να διαβαστούν τα δεδομένα από το πληκτρολόγιο. Στον a7 τοποθετείται η τιμή 5, η οποία ορίζει ότι πρόκειται να διαβαστεί ακέραιος αριθμός. Ο ακέραιος που θα διαβαστεί τοποθετείται στον καταχωρητή a0. Δίνεται ένα παράδειγμα ανάγνωσης ενός ακέραιου αριθμού, ο οποίος μετά την ecall θα βρίσκεται στον a0:

```
li a7,5
ecall
```

Για την εμφάνιση στην οθόνη χρησιμοποιούνται πάλι οι ίδιοι καταχωρητές ορισμάτων με διαφορετικό τρόπο, βέβαια. Αν θέλουμε να εμφανίσουμε στην οθόνη έναν ακέραιο αριθμό, τότε τοποθετούμε στον καταχωρητή a7 τον αριθμό 1 και στον καταχωρητή aθ τον ακέραιο που θέλουμε να εμφανίσουμε στην οθόνη. Στη συνέχεια καλούμε την ecall. Ακολουθεί παράδειγμα κώδικα το οποίο εμφανίζει το 44 στην οθόνη:

```
li a0,44
li a7,1
ecall
```

Θα παρατηρήσει όμως κανείς ότι μετά τον ακέραιο αριθμό δεν υπάρχει αλλαγή γραμμής, κάτι το οποίο μάλλον είναι λογικό. Αν θέλουμε να αλλάξουμε γραμμή, αυτό θα πρέπει να γίνει με τον χαρακτήρα αλλαγής γραμμής: “\n”. Ετσι, θα ορίσουμε ένα συμβολικό όνομα για τον χαρακτήρα αλλαγής γραμμής. Αυτό γίνεται στο χώρο .data στην αρχή του προγράμματος:

```
.data
str_nl: .asciz "\n"
```

Στη συνέχεια τοποθετούμε στον καταχωρητή a7 τον αριθμό 4 και στον καταχωρητή aθ το συμβολικό όνομα. Ο παρακάτω κώδικας τυπώνει μόνο μία αλλαγή γραμμής:

```
.data
str_nl: .asciz "\n"
.text
la a0,str_nl
li a7,4
ecall
```

Θα περιμένουμε λίγο για κάποιο μικρό παράδειγμα, για να ενσωματώσουμε σε αυτό και τον τερματισμό του προγράμματος.

### 9.1.7 Τερματισμός προγράμματος

Για τον τερματισμό του προγράμματος χρησιμοποιούμε πάλι τους ίδιους καταχωρητές ορισμάτων aθ και a7. Στον καταχωρητή a7 τοποθετούμε την τιμή 93, ενώ στον καταχωρητή aθ αυτό που θέλουμε να επιστρέψουμε στο λειτουργικό σύστημα ως αποτέλεσμα. Ένα παράδειγμα τερματισμού εκτέλεσης το οποίο επιστρέφει στο λειτουργικό σύστημα τον αριθμό 0 φαίνεται στη συνέχεια:

```
li a0,0
li a7,93
ecall
```

Ας δούμε, τώρα ένα ολοκληρωμένο παράδειγμα στο οποίο διαβάζουμε έναν ακέραιο αριθμό και τυπώνουμε το διπλάσιο του αριθμού στην οθόνη.

Στον κώδικα, αντί για σχόλια, έχουν τοποθετηθεί ετικέτες, οι οποίες στην πραγματικότητα δεν χρειάζονται, αφού κάνεις δεν κάνει εκεί κάποιο άλμα. Μας φάνηκε αρκετά πιο περιγραφικό να τον οργανώσουμε έτσι. Το πρόγραμμα χωρίζεται στο τμήμα .data στο οποίο ορίστηκε το συμβολικό όνομα για την αλλαγή γραμμής και το τμήμα .text με το κυρίως πρόγραμμα. Η είσοδος του ακεραίου γίνεται στον χώρο που ορίζεται με την ετικέτα input. Στην ετικέτα double\_it μεταφέρεται από τον καταχωρητή aθ στον καταχωρητή το tθ ο αριθμός που διαβάστηκε. Εκεί ο αριθμός διπλασιάζεται. Κατά την εκτύπωση, στην ετικέτα print ο διπλασισμένος αριθμός μεταφέρεται πίσω στον καταχωρητή aθ από όπου και εκτυπώνεται ακολουθούμενος από μία αλλαγή γραμμής. Στην ετικέτα exit γίνεται η έξοδος από το πρόγραμμα:

```

main:
    input:
        li a7,5
        ecall

    double_it:
        mv t0,a0
        addi t0,t0,t0

    print:
        mv a0,t0
        li a7,1
        ecall
        la a0,str_nl
        li a7,4
        ecall

    exit:
        li a0,0
        li a7,93
        ecall

```

### 9.1.8 Παράδειγμα ταξινόμησης

Θα παρουσιάσουμε ένα ακόμα πρόγραμμα σε assembly, ως περισσότερο αντιπροσωπευτικό, αλλά επίσης και αρκούντως σύντομο. Το πρόγραμμα αυτό δέχεται ως είσοδο έναν ακέραιο αριθμό  $N$  και ταξινομεί τους  $N$  πρώτους ακέραιους που βρίσκονται τοποθετημένοι στις  $N$  θέσεις πάνω από τον δείκτη στοίβας. Κάθε ακέραιος καταλαμβάνει 4 bytes.

Αρχικά έχουμε την εισαγωγή του ακέραιου  $N$ . Τον τοποθετούμε στην ετικέτα `input`. Μετά την ολοκλήρωση της `ecall` η είσοδος από τον χρήστη βρίσκεται στον καταχωρητή  $a0$ :

```

input:
    li a7,5                  # input N --> a0
    ecall

```

Θα υλοποιηθεί μία έκδοση του αλγόριθμου φυσαλίδας (bubble sort). Ο αλγόριθμος υλοποιείται με δύο βρόχους. Ο πρώτος δείκτης υλοποιεί τον εξωτερικό βρόχο και τοποθετείται στον καταχωρητή  $s1$ . Θα διασχίσει τον πίνακα από το τέλος προς την αρχή. Ξεκινώντας θέλουμε να δείχνει το  $N-2$  (προτελευταίο) στοιχείο του πίνακα. Έτσι, θα τον αρχικοποιήσουμε πολλαπλασιάζοντας τον αριθμό  $N$  με το μήκος του ακέραιου (4 bytes), ώστε να δείχνει στο τέλος του πίνακα και θα αφαιρέσουμε 8 bytes για να δείξει στο προτελευταίο στοιχείο. Ας δώσουμε σε αυτήν τη διαδικασία το όνομα `init`:

```

init:
    li t0,4                  # initialization of index1 in register s1
    mul s1, a0, t0
    addi s1,s1,-8

```

Ο δείκτης του εσωτερικού βρόχου τοποθετείται στον καταχωρητή  $s2$ . Σε κάθε επανάληψη του εξωτερικού βρόχου αρχικοποιείται στο 0. Ας δώσουμε στο σημείο αυτό την ετικέτα `main_loop`:

```

main_loop:
    li s2,0                  # initialization of index2 in register s2

```

Ο έλεγχος αν οι βρόχοι έχουν τερματιστεί γίνεται στο τέλος των επαναλήψεων. Ας τοποθετήσουμε εκεί την ετικέτα `loop_conditions`. Στην αρχή γίνεται ο έλεγχος για τον εσωτερικότερο βρόχο, όπου έχουμε

αύξηση του `s2` κατά το μήκος ενός ακεραίου, και αμέσως μετά ακολουθεί ο έλεγχος αν οι δύο καραχωρητές είναι ίσοι, αν δηλαδή έχουν ολοκληρωθεί οι εσωτερικές επαναλήψεις. Στην περίπτωση που δεν ολοκληρώθηκαν, θα κάνει άλμα πίσω στην ετικέτα στην οποία θα τοποθετήσουμε το σώμα του βρόχου. Ας την ονομάσουμε `inner_loop`.

Αλλιώς θα μεταβεί στον έλεγχο για τον εξωτερικό βρόχο. Θα γίνει η μείωση του δείκτη κατά το μήκος ενός ακεραίου και θα ακολουθήσει ο έλεγχος που θα το στείλει πίσω στο `main_loop` για να συνεχιστούν οι επαναλήψεις ή θα το αφήσει να κυλήσει έξω από τον βρόχο, αν όλες οι επαναλήψεις έχουν ολοκληρωθεί.

```
loop_conditions:
    addi s2,s2,4          # check loop conditions
    ble s2,s1,inner_loop
    addi s1,s1,-4
    bge s1,zero,main_loop
```

Στην ετικέτα `inner_loop` βρίσκεται το σώμα του βρόχου. Εκεί φορτώνονται δύο συνεχόμενες θέσεις μνήμης στους καταχωρητές `t1` και `t2` και ελέγχεται αν ο `t1` είναι μεγαλύτερος ή ίσος με τον `t2`. Αν ισχύει αυτό, τότε γίνεται άλμα το οποίο αποφεύγει την εναλλαγή των τιμών των δύο αυτών θέσεων μνήμης και οδηγεί στους ελέγχους για πιθανή επόμενη επανάληψη:

```
inner_loop:
    add t0,sp,s2          # check if swap is necessary
    lw t1,(t0)
    lw t2,4(t0)
    bge t1,t2,loop_conditions
```

Η εναλλαγή, αν χρειαστεί, θα γίνει στην ετικέτα `swap` που βρίσκεται κάτω από την `inner_loop`:

```
swap:
    sw t1,4(t0)          # swap
    sw t2,(t0)
```

Οι εντολές στην ετικέτα `exit` είναι υπεύθυνες για τον ομαλό τερματισμό του προγράμματος. Το πλήρες πρόγραμμα ακολουθεί:

```
entry:
    input:
        li a7,5            # input N --> a0
        ecall

    init:
        li t0,4            # initialization of index1 in register s1
        mul s1, a0, t0
        addi s1,s1,-8

    main_loop:
        li s2,0            # initialization of index2 in register s2

    inner_loop:
        add t0,sp,s2      # check if swap is necessary
        lw t1,(t0)
        lw t2,4(t0)
        bge t1,t2,loop_conditions
    swap:
        sw t1,4(t0)        # swap
```

```

sw t2,(t0)

loop_conditions:
    addi s2,s2,4      # check loop conditions
    ble s2,s1,inner_loop
    addi s1,s1,-4
    bge s1,zero,main_loop

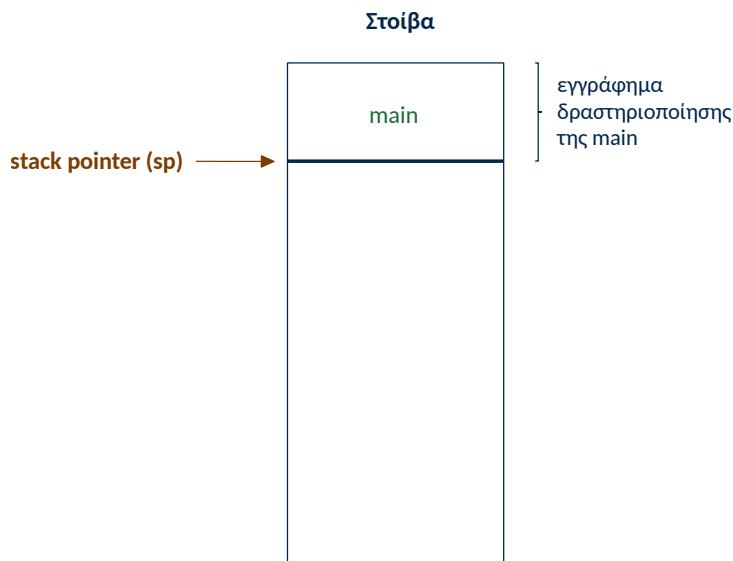
exit:
    li a0,0            # the end
    li a7,93
    ecall

```

### 9.1.9 Το εγγράφημα δραστηριοποίησης

Κάθε συνάρτηση ή διαδικασία που εκτελείται, συμπεριλαμβανομένης και του κυρίως προγράμματος, δεσμεύει έναν χώρο στη στοίβα προκειμένου να τοποθετήσει εκεί δεδομένα ζωτικής σημασίας για τη λειτουργία της, καθώς και τιμές ή διευθύνσεις διαφόρων μεταβλητών. Μία πρώτη συζήτηση για το εγγράφημα δραστηριοποίησης, κυρίως για τις μεταβλητές οι οποίες αποθηκεύονται σε αυτό, είχαμε στο κεφάλαιο του πίνακα συμβόλων (κεφ. 8). Στην ενότητα αυτή θα δούμε τη δομή ενός εγγραφήματος δραστηριοποίησης, πού τοποθετείται στη στοίβα, ποια η διάρκεια ζωής του και πώς συνεισφέρει στη λειτουργία ενός προγράμματος.

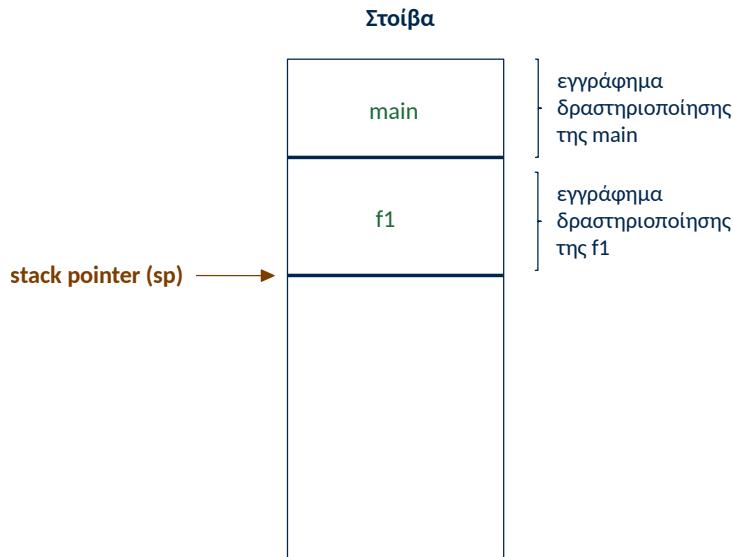
Με την εκκίνηση της εκτέλεσης ενός προγράμματος το λειτουργικό σύστημα τού δεσμεύει χώρο ο οποίος θα λειτουργήσει ως στοίβα. Στο σημείο αυτό τοποθετείται ο δείκτης στοίβας *sp* (*stack pointer*). Στη συνέχεια αναλαμβάνει ο κώδικας που παρήχθη από τον μεταγλωττιστή. Ο δείκτης στοίβας μετατοπίζεται τόσες θέσεις, όσες θέσεις μνήμης θέλουμε να καταλάβει το εγγράφημα δραστηριοποίησης, δεσμεύοντας, έτσι, χώρο για το κυρίως πρόγραμμα, μέσα στον χώρο που έδωσε για το σκοπό αυτόν στην εφαρμογή το λειτουργικό σύστημα. Θυμίζουμε ότι ο δείκτης στοίβας πρέπει ανά πάσα στιγμή να δείχνει στην αρχή του εγγραφήματος δραστηριοποίησης της συνάρτησης που τη στιγμή αυτή εκτελείται. Η εικόνα της στοίβας είναι τη στιγμή αυτή όπως εικονίζεται στο σχήμα 9.2.



Σχήμα 9.2: Εγγράφημα δραστηριοποίησης του κυρίως προγράμματος στη στοίβα.

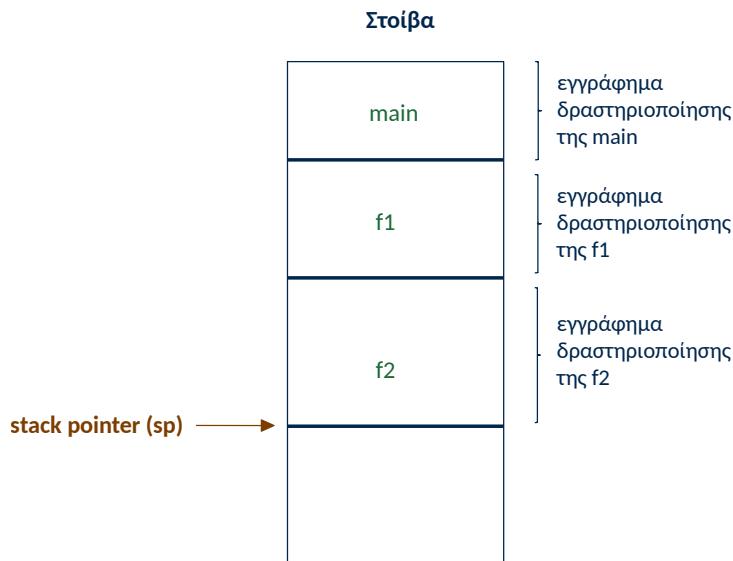
Αν το κυρίως πρόγραμμα καλέσει τη συνάρτηση *f1*, τότε αυτή θα τοποθετηθεί μετά το εγγραφημα δραστηριοποίησης του κυρίως προγράμματος. Η δέσμευση του χώρου θα γίνει με τη μετατόπιση του *sp*. Το

εγγράφημα δραστηριοποίησης του κυρίως προγράμματος εξακολουθεί να υπάρχει στη στοίβα, αφού η εκτέλεσή του δεν έχει ολοκληρωθεί και θα επιστραφεί σε αυτό ο έλεγχος μετά την ολοκλήρωση της εκτέλεσης της `f1`. Η στοίβα έχει τώρα την εικόνα που φαίνεται στο σχήμα 9.3.



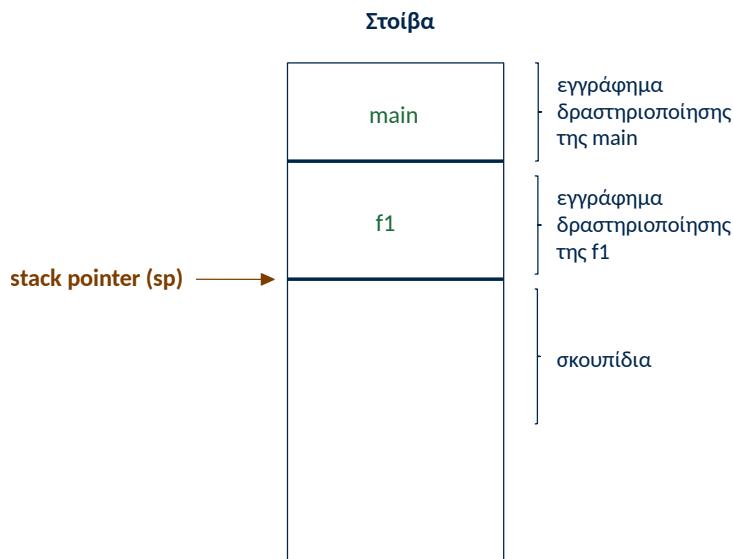
Σχήμα 9.3: Νέο εγγράφημα δραστηριοποίησης στη στοίβα.

Ας προσθέσουμε ακόμα ένα επίπεδο και ας θεωρήσουμε ότι η συνάρτηση `f1` καλεί τη συνάρτηση `f2`. Με παρόμιοι τρόπο, όπως συζητήσαμε παραπάνω για την `f1`, θα φτάσουμε σε μία είκονα της στοίβας, όπως αυτή εικονίζεται στο σχήμα 9.4.

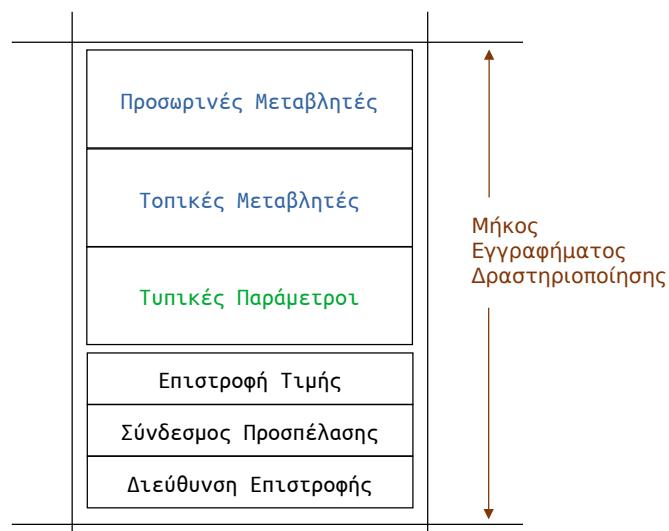


Σχήμα 9.4: Εγγραφήματα δραστηριοποίησης στη στοίβα από φωλιασμένες κλήσεις.

Ας υποθέσουμε ότι η `f3` δεν καλεί κάποια συνάρτηση ή διαδικασία, οπότε με την ολοκλήρωσή της ο έλεγχος της εκτέλεσης πρέπει να επιστρέψει στην `f2`. Όσον αφορά τη στοίβα, αυτό σημαίνει ότι ο δείκτης στοίβας θα επιστρέψει στην προηγούμενη θέση του και θα δείχνει στην αρχή του εγγραφήματος δραστηριοποίησης της `f2`. Ο χώρος που κατείχε η `f3` θεωρείται πια αποδεσμευμένος. Τα περιεχόμενά του δεν έχουν σβηστεί, απλά θεωρούνται από το σύστημα άχρηστα και επιτρέπεται ο χώρος αυτός να εκχωρηθεί σε άλλη συνάρτηση που πιθανά να κληθεί και να τον χρειαστεί στη συνέχεια, ώστε να τοποθετήσει εκεί δικά της δεδομένα. Χρησιμοποιούμε για τα δεδομένα αυτά τον όρο *σκουπίδια*. Η κατάσταση της στοίβας, όπως είναι μετά την ολοκλήρωση της `f3`, φαίνεται στο σχήμα 9.5.



Σχήμα 9.5: Ολοκλήρωση συνάρτησης και επιστροφή δεσμευμένου χώρου στη στοίβα.



Σχήμα 9.6: Εγγράφημα δραστηριοποίησης.

Με παρόμοιο τρόπο θα προστεθούν εγγραφήματα δραστηριοποίησης στη στοίβα κάθε φορά που μία συνάρτηση καλεί μία άλλη και θα επιστρέφεται ο δεσμευμένος χώρος κάθε φορά που μία συνάρτηση ολοκληρώνει την εκτέλεσή της. Όταν ολοκληρωθεί η εκτέλεση του κυρίως προγράμματος, τότε και ο τελευταίος δεσμευμένος χώρος αποδεσμεύεται και στη συνέχεια όλος ο χώρος που καταλάμβανε η εφαρμογή στη στοίβα επιστρέφεται στο λειτουργικό σύστημα για μελλοντική χρήση.

Όλα αυτά δεν γίνονται αυτόματα, αλλά από κώδικα που εμείς θα δημιουργήσουμε στη συνέχεια. Προς το παρόν ας δούμε τι περιέχει μέσα ένα εγγράφημα δραστηριοποίησης.

Δεν είναι η πρώτη φορά που ασχολούμαστε με τα περιεχόμενα ενός εγγραφήματος δραστηριοποίησης. Το έχουμε μελετήσει στο κεφάλαιο του πίνακα συμβόλων (κεφ. 8). Επαναλαμβάνουμε εδώ το αντίστοιχο σχήμα για διευκόλυνση (Σχήμα 9.6). Στο κεφάλαιο εκείνο είχαμε δώσει κύρια βάση στις θέσεις από 12 και επάνω, όπου τοποθετούνται μεταβλητές και παράμετροι. Ανάλογα με το είδος της παραμέτρου στο εγγράφημα δραστηριοποίησης μπορεί να είναι τοποθετημένη είτε η τιμή της μεταβλητής που περνάει ως παράμετρος είτε η διεύθυνσή της. Τι περιέχουν, όμως, οι τρεις πρώτες θέσεις του εγγραφήματος δραστηριοποίησης; Περιέχουν πληροφορίες για την επιστροφή της συνάρτησης, μετά την ολοκλήρωσή της, για τη σύνδεση με συναρτήσεις προγόνους και για την επιστροφή του αποτελέσματος της συνάρτησης. Κάθε μία τέτοια θέση

καταλαμβάνει 4 bytes και για τον λόγο αυτόν απαιτούνται συνολικά 12 bytes. Συγκεκριμένα:

- **Διεύθυνση επιστροφής:** η διεύθυνση στην οποία πρέπει να μεταβεί το πρόγραμμα μετά την ολοκλήρωση της εκτέλεσης της συνάρτησης ή της διαδικασίας. Καταλαμβάνει τα 4 πρώτα bytes του εγγραφήματος δραστηριοποίησης.
- **Σύνδεσμος προσπέλασης:** η διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα της συνάρτησης ή της διαδικασίας. Μέσα από αυτόν τον σύνδεσμο η συνάρτηση ή η διαδικασία μπορεί να προσπελάσει δεδομένα που ανήκουν σε πρόγονούς της. Καταλαμβάνει τα bytes από τη θέση 4 μέχρι και τη θέση 7 του εγγραφήματος δραστηριοποίησης (συνολικά 4 bytes).
- **Επιστροφή τιμής:** η διεύθυνση της μεταβλητής στην οποία επιθυμούμε να γραφεί το αποτέλεσμα της συνάρτησης. Αν πρόκειται για διαδικασία, η θέση αυτή στο εγγράφημα δραστηριοποίησης μένει αχρησιμοποίητη (με σκουπίδια). Καταλαμβάνει τα bytes από τη θέση 8 μέχρι και τη θέση 11 του εγγραφήματος δραστηριοποίησης (συνολικά 4 bytes).

Τα περιεχόμενα των τριών πρώτων θέσεων του εγγραφήματος δραστηριοποίησης συμπληρώνονται κατά την εκτέλεση του προγράμματος από κώδικα τον οποίο παράγει ο μεταγλωττιστής.

## 9.2 Βοηθητικές συναρτήσεις

Νωρίτερα, στο κεφάλαιο της παραγωγής του ενδιάμεσου κώδικα (κεφ. 6), ορίσαμε κάποιες βοηθητικές συναρτήσεις οι οποίες μας διευκόλυναν τόσο στον σχεδιασμό του ενδιάμεσου κώδικα, κάνοντας το σχέδιο περισσότερο σαφές και ευανάγνωστο, όσο και στην υλοποίησή του, μειώνοντας τον όγκο του κώδικα που απαιτείται να αναπτυχθεί.

Στον τελικό κώδικα θα κάνουμε ακριβώς το ίδιο. Ο σκοπός των βοηθητικών συναρτήσεων παραμένει ο ευκολότερος σχεδιασμός, η ευανάγνωστη περιγραφή και η απλούστευση του τελικού κώδικα. Μόνο που τώρα ο κώδικας που θα γραφεί είναι αρκετά πιο πολύπλοκος, μεγαλύτερος σε όγκο και κρύβει περισσότερη λειτουργικότητα.

Θα αναφέρουμε εδώ τις βοηθητικές συναρτήσεις που χρειαζόμαστε και θα τις δούμε μία προς μία αναλυτικά στη συνέχεια.

- **`gnlvcod()`:** δημιουργεί τελικό κώδικα για την προσπέλαση πληροφορίας που βρίσκεται αποθηκευμένη στο εγγράφημα δραστηριοποίησης κάποιου προγόνου της συνάρτησης ή της διαδικασίας που αυτή τη στιγμή μεταφράζεται.
- **`loadvr()`:** παράγει τελικό κώδικα ο οποίος διαβάζει μία μεταβλητή που είναι αποθηκευμένη στη μνήμη και την μεταφέρει σε έναν καταχωρητή.
- **`storerv()`:** κάνει την αντίστροφη διαδικασία από το `loadvr()`, παράγει τελικό κώδικα ο οποίος αποθηκεύει στη μνήμη την τιμή μιας μεταβλητής η οποία βρίσκεται σε έναν καταχωρητή.

Θα χρησιμοποιήσουμε ακόμα τη βοηθητική συνάρτηση `produce()` η οποία, σε αντιστοιχία με την `genQuad()` που δημιουργούσε μία νέα τετράδα ενδιάμεσου κώδικα, δημιουργεί μία νέα γραμμή (εντολή) τελικού κώδικα. Η δημιουργία του τελικού κώδικα δεν γίνεται σε κάποια δομή στη μνήμη, όπως συνέβαινε με τον ενδιάμεσο κώδικα, αλλά ο κώδικας που παράγεται γράφεται απευθείας σε αρχείο.

### 9.2.1 Η συνάρτηση `gnlvcod()`

Η πρώτη βοηθητική συνάρτηση είναι η `gnlvcod()`. Η `gnlvcod()` παράγει τελικό κώδικα για την προσπέλαση μεταβλητών ή διευθύνσεων που είναι αποθηκευμένες σε κάποιο εγγράφημα δραστηριοποίησης

διαφορετικό από της συνάρτησης που αυτή τη στιγμή μεταφράζεται. Σύμφωνα με τον ορισμό της *C-imple*, κάθε συνάρτηση έχει δικαίωμα να προσπελάσει, πέρα από τις μεταβλητές και διευθύνσεις που είναι αποθηκευμένες στο δικό της εγγράφημα δραστηριοποίησης, μεταβλητές και διευθύνσεις που ανήκουν στο εγγράφημα δραστηριοποίησης κάποιου προγόνου της.

Μέσα στους προγόνους της συνάρτησης βρίσκεται και το κυρίως πρόγραμμα. Συνεπώς, η `gnlvcode()` μπορεί να προσπελάσει και τις καθολικές μεταβλητές. Παρόλο που μπορεί να το κάνει, δεν θα της ζητηθεί ποτέ να το κάνει, αφού, όπως θα δούμε αργότερα, υπάρχει γρηγορότερος τρόπος για να γίνει αυτό. Επειδή η ανάγκη προσπέλασης καθολικών μεταβλητών είναι αρκετά συχνή σε ένα πρόγραμμα, θα προτιμήσουμε να μην χρησιμοποιήσουμε την `gnlvcode()` για την προσπέλασή τους, αλλά θα το κάνουμε με διαφορετικό μηχανισμό, που θα δούμε στις δύο επόμενες βοηθητικές συναρτήσεις, αμέσως στη συνέχεια.

Η `gnlvcode()` παίρνει ως όρισμα μία μεταβλητή, τη μεταβλητή της οποίας την τιμή ή τη διεύθυνση θέλουμε να προσπελάσουμε:

```
def gnlvcode(v):
```

όπου `v` το όνομα της μεταβλητής.

Το αποτέλεσμα της εκτέλεσης της `gnlvcode()` είναι:

- αν αναζητείται η τιμή μιας μεταβλητής, τότε θα μεταφερθεί στον καταχωρητή `t0` η διεύθυνση της μεταβλητής που αναζητείται.
- αν αναζητείται η διεύθυνση μιας μεταβλητής, τότε θα μεταφερθεί στον καταχωρητή `t0` η διεύθυνση μνήμης η οποία περιέχει τη διεύθυνση της μεταβλητής που αναζητείται.

Η `gnlvcode()` λειτουργεί ως εξής:

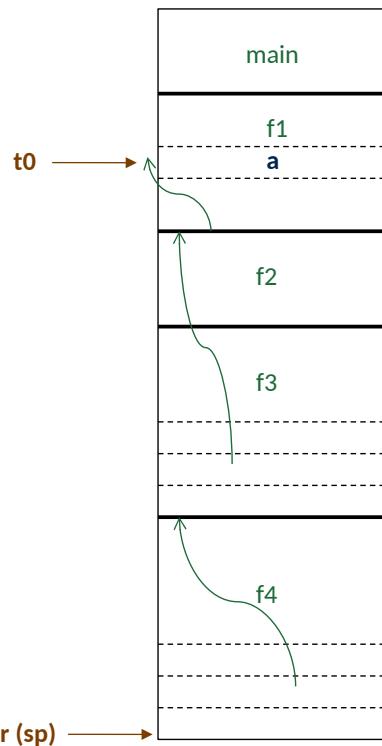
- Αναζητεί στον πίνακα συμβόλων το όνομα της μεταβλητής που της δίνεται ως παράμετρος (αν το όνομα της μεταβλητής δεν βρεθεί στον πίνακα συμβόλων, τότε ο πίνακας συμβόλων θα επιστρέψει στον χρήστη το κατάλληλο μήνυμα σφάλματος και θα τερματίσει τη μεταγλώττιση).
- Από το επίπεδο στο οποίο βρέθηκε η μεταβλητή η `gnlvcode()` θα συμπεράνει πόσα επίπεδα επάνω στο γενεαλογικό δέντρο της συνάρτησης θα πρέπει να ανέβει προκειμένου να φτάσει στο εγγράφημα δραστηριοποίησης που έχει την πληροφορία που αναζητεί.
- Το πρώτο βήμα είναι να μεταβεί στον γονέα της συνάρτησης. Η διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα βρίσκεται αποθηκευμένη στον σύνδεσμο προσπέλασης της συνάρτησης, στη θέση `-8(sp)` δηλαδή, αφού ο `sp` δείχνει την αρχή του εγγραφήματος δραστηριοποίησης της συνάρτησης που κάθε στιγμή εκτελείται. Θα χρησιμοποιηθεί ένας καταχωρητής, ως καταχωρητής που θα βοηθήσει την `gnlvcode()` να ανέβει τα επίπεδα. Έστω ότι αυτός είναι ο `t0`. Η `gnlvcode()` θα παραγάγει για τον σκοπό αυτόν τον κώδικα:

```
lw mv t0, -4(sp)
```

- Αν η `gnlvcode()` από τον πίνακα συμβόλων έχει συμπεράνει ότι πρέπει να ανέβει *n* επίπεδα για να εντοπίσει το εγγράφημα δραστηριοποίησης που αναζητεί, τότε έχει ακόμα *n*−1 επίπεδα να ανέβει. Με τον ίδιο τρόπο, όπως και προηγουμένως, θα διαβάσει τον σύνδεσμο προσπέλασης από το εγγράφημα δραστηριοποίησης στο οποίο δείχνει ο `t0` και για κάθε επίπεδο που πρέπει να ανέβει θα παραγάγει την εντολή:

```
lw mv t0, -4(t0)
```

- Μετά την ολοκλήρωση των μεταβάσεων μέσω των συνδέσμων προσπέλασης ο καταχωρητής `t0` δείχνει την αρχή του εγγραφήματος δραστηριοποίησης το οποίο ο πίνακας συμβόλων υπέδειξε.



Σχήμα 9.7: Πρόσβαση σε δεδομένα που βρίσκονται σε συναρτήσεις προγόνους, μέσω της `gnvlcode()`.

- Το τελευταίο βήμα είναι να κατέβει ο `t0` κατά offset θέσεις ώστε να δείξει στη θέση μνήμης στην οποία βρίσκεται η πληροφορία που αναζητείται.
- Το αποτέλεσμα που επιστρέφει η `gnvlcode()` είναι το περιεχόμενο του `t0`

Αν συγκεντρώσουμε τον κώδικα που παράγει η `gnvlcode()` έχουμε:

```
lw t0,-4(sp)
lw t0,-4(t0)
...
lw t0,-4(t0)
addi t0,t0,offset
```

Σχηματικά η διαδικασία εικονίζεται στο σχήμα 9.7. Στο σχήμα αυτό η `f4` αναζητεί την τιμή του `a`, η οποία είναι αποθηκευμένη στον παππού της, την `f1`. Μέσω του δικού της συνδέσμου προσπέλασης `4(sp)`, μεταβαίνει στον γονέα της `f3` και τοποθετεί στο εκεί εγγράφημα δραστηριοποίησης τον `t0`. Στη συνέχεια, μέσω του συνδέσμου προσπέλασης του γονέα `4(t0)`, βρίσκει τον παππού `f1` και τοποθετεί τον `t0` στην αρχή του εγγραφήματος δραστηριοποίησης του παππού `f3`. Τέλος, μετατοπίζει τον `t0` κατά offset θέσεις, ώστε ο `t0` να λάβει την τελική του θέση δείχνοντας τελικά το `a`.

Στο ίδιο σχήμα θα παρατηρήσετε την ύπαρξη της συνάρτησης `f2`, η οποία δεν εμπλέκεται στην αναζήτηση της `a`. Ο λόγος είναι ότι η `f2` δεν αποτελεί πρόγονο της `f4`. Γιατί τότε βρίσκεται στη στοίβα; Μία συνάρτηση δεν καλείται μόνο από τον γονέα της αλλά μπορεί να κληθεί από αδερφό ή και την ίδια τη συνάρτηση, αναδρομικά. Η κλήση των συναρτήσεων στο παράδειγμά μας έχει γίνει ως εξής: Το κυρίως πρόγραμμα κάλεσε την `f1`, η οποία είναι παιδί του κυρίως προγράμματος, η `f1` κάλεσε την `f2`, η οποία είναι παιδί της, η `f2` κάλεσε την `f3` η οποία είναι αδελφός της και τέλος η `f3` κάλεσε την `f4`, η οποία είναι παιδί της. Άρα, ενώ η `f2` υπάρχει στην ιεραρχία των κλήσεων, η `f4` δεν έχει δικαίωμα πρόσβασης στις μεταβλητές της, κάτι που είναι σε συμφωνία και με την περιγραφή της *C-imple*.

Στον παρακάτω κώδικα:

```
function f1()
```

```
{
    declare x enddeclare;
    function f2()
    {
        function f3()
        {
            x := 1
        }
    }
}
```

η `gnlvcode(x)` θα τοποθετήσει στον `t0` τη διεύθυνση του `x`. Ο πίνακας συμβόλων θα δείξει ότι η `x` βρίσκεται δύο επίπεδα επάνω από το επίπεδο της `f3` και ότι έχει offset ίσο με 12. Η `gnlvcode(x)` θα παραγάγει τον ακόλουθο κώδικα:

```
lw t0,-4(sp)
lw t0,-4(t0)
addi t0,t0,-12
```

### 9.2.2 Η συνάρτηση `loadvr()`

Η `loadvr()` είναι η συνάρτηση η οποία παράγει τον κώδικα για να διαβαστεί η τιμή μιας μεταβλητής από τη μνήμη, δηλαδή από μία θέση στη στοίβα, και να μεταφερθεί σε έναν καταχωρητή.

Η σύνταξη της `loadvr()` είναι η ακόλουθη:

```
def loadvr(v, reg)
    # v: source variable
    # reg: target register
```

όπου `v` το όνομα της μεταβλητής την τιμή της οποίας θέλουμε να διαβάσουμε και `reg` το όνομα του καταχωρητή στον οποίο θέλουμε να τοποθετηθεί.

Η `loadvr()` βασίζεται στην πληροφορία που της επιστρέφει ο πίνακας συμβόλων και, ανάλογα με την περίπτωση, παράγει και τον αντίστοιχο κώδικα. Πρόκειται, δηλαδή, για μία μεγάλη δομή πολλαπλής επιλογής (`if...elif...else`). Θα διακρίνουμε τις περιπτώσεις και θα εξετάσουμε τον κώδικα που παράγεται σε κάθε περίπτωση χωριστά.

#### 9.2.2.1 Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή ή προσωρινή μεταβλητή

Πρόκειται για τις περιπτώσεις που η τιμή της μεταβλητής που ζητάμε βρίσκεται αποθηκευμένη στο εγγράφημα δραστηριοποίησης της συνάρτησης που μεταφράζεται. Για να μεταφερθεί μία τέτοια μεταβλητή στον καταχωρητή `reg` παράγεται ο ακόλουθος κώδικας:

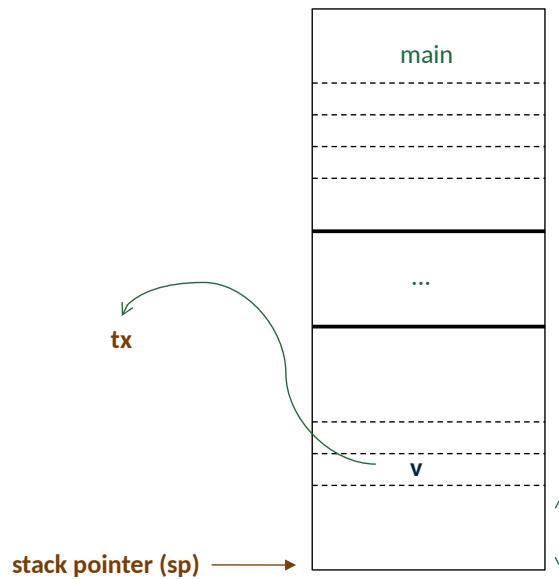
```
lw reg,-offset(sp)
```

όπου `offset`, το `offset` της μεταβλητής που επιστρέφει ο πίνακας συμβόλων.

Σχηματικά, η μεταφορά της τιμής μιας τοπικής μεταβλητής ή παραμέτρου που έχει περαστεί με τιμή ή προσωρινής μεταβλητής εικονίζεται στο σχήμα 9.8.

#### 9.2.2.2 Παράμετρος που έχει περαστεί με αναφορά

Όταν μία παράμετρος έχει περαστεί με αναφορά, τότε στο εγγράφημα δραστηριοποίησης της συνάρτησης έχει τοποθετηθεί η διεύθυνσή της. Αυτό δεν γίνεται αυτόματα. Θα δούμε πώς γίνεται παρακάτω στην κλήση



Σχήμα 9.8: Ανάγνωση μιας τοπικής μεταβλητής ή παραμέτρου που έχει περαστεί με τιμή ή προσωρινής μεταβλητής.

των συναρτήσεων και των διαδικασιών. Για να μεταφερθεί η τιμή μίας τέτοιας μεταβλητής στον καταχωρητή reg, απαιτείται ακόμα ένα βήμα. Πρέπει πρώτα να μεταφερθεί η διεύθυνση της μεταβλητής από τη στοίβα σε έναν καταχωρητή, για παράδειγμα τον tθ, και στη συνέχεια να χρησιμοποιηθεί ο tθ ως καταχωρητής δείκτης ώστε να μεταφερθεί στον reg η τιμή της μεταβλητής. Για να γίνει αυτό θα πρέπει να παραχθεί ο ακόλουθος κώδικας:

```
lw tθ, -offset($p)
lw reg, (tθ)
```

όπου offset το offset της μεταβλητής που επιστρέφει ο πίνακας συμβόλων.

Σχηματικά, η μεταφορά της τιμής μιας παραμέτρου που έχει περαστεί με αναφορά εικονίζεται στο σχήμα 9.9.

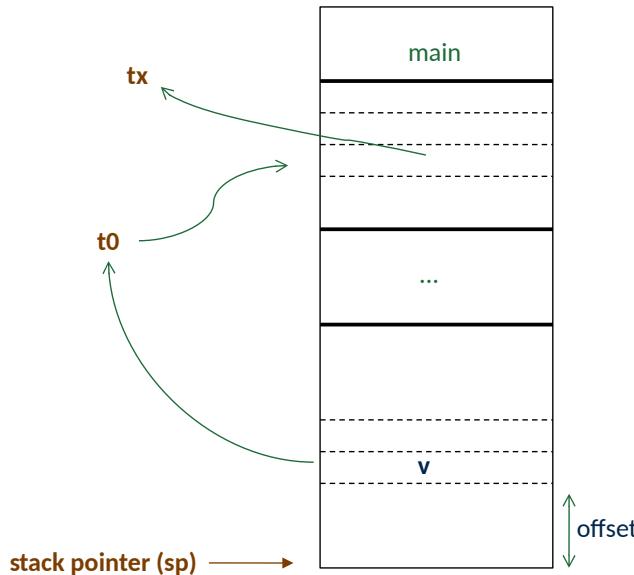
### 9.2.2.3 Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή η οποία ανήκει σε πρόγονο

Πρόκειται για περιπτώσεις που η τιμή της μεταβλητής που ζητάμε βρίσκεται αποθηκευμένη στο εγγράφημα δραστηριοποίησης κάποιας συνάρτησης προγόνου της συνάρτησης που μεταφράζεται. Αυτό σημαίνει ότι η πληροφορία που αναζητούμε βρίσκεται σε κάποιο εγγράφημα δραστηριοποίησης στο οποίο μπορούμε να έχουμε πρόσβαση μέσα από μία σειρά ανακτήσεων συνδέσμων προσπέλασης.

Να σημειώσουμε εδώ ότι όλα τα εγγραφήματα δραστηριοποίησης των προγόνων της συνάρτησης βρίσκονται στη στοίβα, αφού η ολοκλήρωση της εκτέλεσής τους δεν έχει ολοκληρωθεί, δεδομένου ότι κάθε συνάρτηση μπορεί να κληθεί μόνο από γονέα ή από αδελφό. Η με άλλα λόγια, για να κληθεί μία συνάρτηση, πρέπει να έχει κληθεί ο γονέας της. Ολοκλήρωση της εκτέλεσης του γονέα σημαίνει ότι όλα τα παιδιά του θα έχουν ολοκληρώσει τη δική τους εκτέλεση.

Στο σημείο αυτό μπορούμε να εκμεταλλευτούμε τη συνάρτηση gnlvcode() που φτιάξαμε νωρίτερα. Η gnlvcode() θα πάρει ως παράμετρο τη μεταβλητή που θέλουμε να διαβάσουμε και θα τοποθετήσει στον tθ τη διεύθυνση στην οποία βρίσκεται αποθηκευμένη η μεταβλητή. Στη συνέχεια, δεν έχουμε παρά να διαβάσουμε το περιεχόμενο της θέσης μνήμης:

```
gnlvcode()
produce('lw reg,(tθ)')
```



Σχήμα 9.9: Προσπέλαση μιας παραμέτρου που έχει περαστεί με αναφορά.

Στον παραπάνω συμβολισμό εννοούμε ότι γίνεται κλήση της `gnlvcode()` και στη συνέχεια παράγεται η εντολή:

```
lw reg,(t0)
```

Ας δούμε ένα παράδειγμα. Στον παρακάτω κώδικα:

```
function f1(in x)
{
    function f2()
    {
        function f3()
        {
            declare a enddeclare;
            a := x
        }
    }
}
```

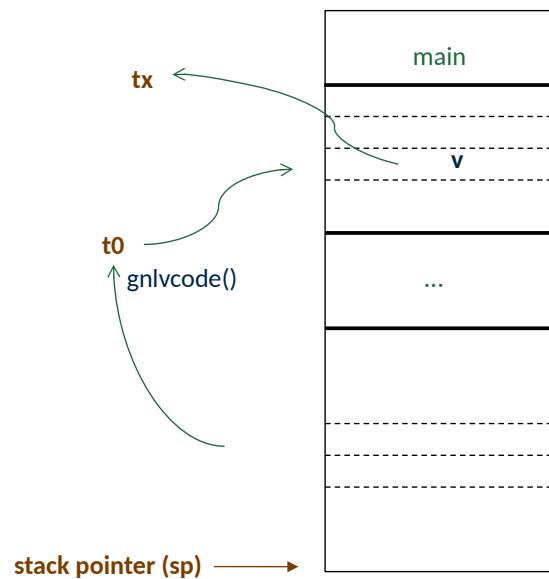
με την κλήση της `loadvr(x,t1)` η `gnlvcode()` θα τοποθετήσει στον `t0` τη διεύθυνση του `x`. Ο πίνακας συμβόλων θα δείξει ότι η `x` βρίσκεται δύο επίπεδα επάνω από το επίπεδο της `f3` και ότι έχει `offset` ίσο με 12. Στη συνέχεια θα μεταφερθεί το περιεχόμενο της θέσης μνήμης που δείχνει ο `t0` στον καταχωρητή `t1`:

```
lw t0,-4(sp)
lw t0,-4(t0)
addi t0,t0,-12
lw t1,(t0)
```

Σχηματικά, η μεταφορά της τιμής μιας τοπικής μεταβλητής ή παραμέτρου με τιμή η οποία βρίσκεται σε πρόγονο της υπό μετάφρασης συνάρτησης ή διαδικασίας εικονίζεται στο σχήμα 9.10.

#### 9.2.2.4 Παράμετρος που έχει περαστεί με αναφορά, η οποία ανήκει σε πρόγονο

Πρόκειται για περιπτώσεις που ζητάμε την τιμή μιας μεταβλητής η οποία όμως έχει περαστεί με αναφορά σε κάποιον πρόγονο της συνάρτησης. Άρα, στο εγγράφημα δραστηριοποίησης του προγόνου βρίσκεται



Σχήμα 9.10: Προσπέλαση τιμής μιας τοπικής μεταβλητής ή παραμέτρου με τιμή και βρίσκεται σε κάποιον πρόγονο.

αποθηκευμένη η διεύθυνση της μεταβλητής. Σε αυτό το εγγράφημα δραστηριοποίησης έχουμε πρόσβαση μέσα από μία σειρά ανακτήσεων συνδέσμων προσπέλασης.

Όπως και στην περίπτωση της τοπικής μεταβλητής ή της παραμέτρου που έχει περαστεί με τιμή η οποία ανήκει σε κάποιον πρόγονο και την συζητήσαμε νωρίτερα, έτσι και εδώ μπορούμε να εκμεταλλευτούμε τη συνάρτηση `gnlvcode()`. Η `gnlvcode()` θα πάρει ως παράμετρο τη μεταβλητή που θέλουμε να διαβάσουμε και θα τοποθετήσει στον `t0` τη διεύθυνση στην οποία βρίσκεται αποθηκευμένη η διεύθυνση της μεταβλητής (και όχι η τιμή της, όπως στην προηγούμενη περίπτωση). Χρειαζόμαστε δηλαδή ακόμα ένα βήμα, αφού διαβάσουμε το περιεχόμενο της θέσης μνήμης που τοποθέτησε τον `t0` η `gnlvcode()`, να τη χρησιμοποιήσουμε για να φτάσουμε στην τιμή της μεταβλητής που ζητούμε:

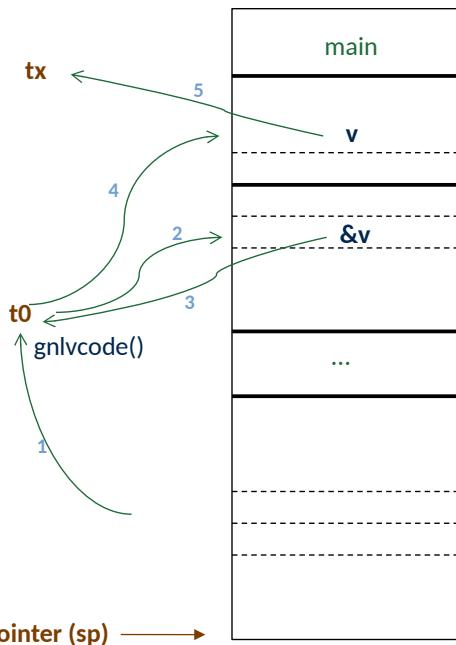
```
gnlvcode()
produce('lw t0,(t0)')
produce('lw reg,(t0)')
```

Ας δούμε ένα παράδειγμα. Στον παρακάτω κώδικα:

```
function f1(inout x)
{
    function f2()
    {
        function f3()
        {
            declare a enddeclare;
            a := x
        }
    }
}
```

Το `x` έχει περαστεί στην `f1` με αναφορά. Το εγγόνι της, η `f3`, θέλει να τη διαβάσει. Ο πίνακας συμβόλων δίνει την πληροφορία ότι πρόκειται για παράμετρο με αναφορά, δύο επίπεδα επάνω από το τρέχον βάθος φωλιάσματος και έχει `offset=12`. Άρα μέσω της `gnlvcode()` θα ανέβουμε δύο επίπεδα, ώστε μετά τον γονέα να φτάσουμε στον παππού και στη συνέχεια θα μετατοπίσουμε τον `t0` κατά 12 θέσεις:

```
lw t0,-4(sp)
```



Σχήμα 9.11: Προσπέλαση τυπικής παραμέτρου που έχει περαστεί με αναφορά και βρίσκεται σε κάποιον πρόγονο.

```
lw t0,-4(t0)
addi t0,t0,-12
```

Μας μένει να διαβάσουμε τα περιεχόμενα της θέσης μνήμης που μας υπέδειξε η `gnlvcode()` και, επειδή εκεί είναι αποθηκευμένη η διεύθυνση της μεταβλητής που ζητάμε, να κάνουμε ακόμα ένα βήμα και μέσω της διεύθυνσης να φτάσουμε στη ζητούμενη τιμή:

```
lw t0,(t0)
lw t1,(t0)
```

Ο ολοκληρωμένος κώδικας που θα παραχθεί είναι ο ακόλουθος:

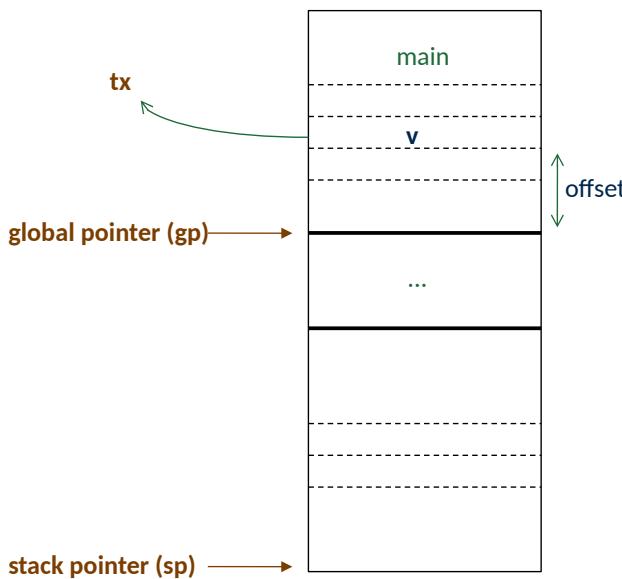
```
lw t0,-4(sp)
lw t0,-4(t0)
addi t0,t0,-12
lw t0,(t0)
lw t1,(t0)
```

Σχηματικά τα παραπάνω εικονίζονται στο σχήμα 9.11.

### 9.2.2.5 Καθολική μεταβλητή

Οι καθολικές μεταβλητές είναι μεταβλητές που έχουν δηλωθεί στο κυρίως πρόγραμμα. Τουλάχιστον στη γλώσσα *C-imple*. Έτσι, μέσω της `gnlvCode()`, μπορούμε να ανέβουμε όσα επίπεδα χρειαστεί και να φτάσουμε στο κυρίως πρόγραμμα, το οποίο είναι πρόγονος κάθε άλλης συνάρτησης στο πρόγραμμα.

Η πρόσβαση στις καθολικές μεταβλητές είναι συνήθως συχνή, ενώ η διαδικασία με τα επίπεδα έχει κάποιο κόστος, ιδιαίτερα όταν πρέπει να ανέβουμε αρκετά επίπεδα. Μια λύση, η οποία επιταχύνει τον χρόνο πρόσβασης στις καθολικές μεταβλητές, είναι να αφιερώσουμε έναν καταχωρητή, ο οποίος θα είναι υπεύθυνος να σημειώνει την αρχή του εγγραφήματος δραστηριοποίησης του κυρίως προγράμματος και δεν θα χρησιμοποιείται για κάποιον άλλο σκοπό όσο εκτελείται το πρόγραμμα. Θα επιλέξουμε τον καταχωρητή `gp` (global pointer). Έτσι, όταν σε μία ερώτηση προς τον πίνακα συμβόλων επιστραφεί η απάντηση ότι μία



Σχήμα 9.12: Προσπέλαση καθολικής μεταβλητής.

μεταβλητή είναι καθολική και ότι αυτή έχει ένα συγκεκριμένο offset, τότε η πρόσβαση στη μεταβλητή γίνεται ως εξής:

```
lw reg,-offset(gp)
# reg: register
```

Σχηματικά, ο τρόπος πρόσβασης στις καθολικές μεταβλητές φαίνεται στο σχήμα 9.12. Η μεταβλητή να χαρακτηρίζεται ως καθολική από τον πίνακα συμβόλων και προσπελάζεται μέσω του καταχωρητή gp και βρίσκεται offset θέσεις πάνω από αυτόν.

Στην περίπτωση που έχουμε μία γλώσσα η οποία διαχωρίζει τις μεταβλητές του κυρίως προγράμματος από τις καθολικές μεταβλητές, η παραπάνω λύση μπορεί να εφαρμοστεί για την πρόσβαση στις καθολικές μεταβλητές, ενώ για την πρόσβαση στις μεταβλητές του κυρίως προγράμματος θα χρησιμοποιηθεί ο μηχανισμός πρόσβασης μέσω των συνδέσμων προσπέλασης.

### 9.2.2.6 Εκχώρηση αριθμητικής σταθεράς

Μία απλή περίπτωση, η οποία ίσως και να μην ανήκει εδώ, αφού δεν πρόκειται για μετακίνηση από τη μνήμη, αλλά θα τη συζητήσουμε καταχρηστικά και θα την εντάξουμε στην loadvr(), είναι η φόρτωση μιας αριθμητικής σταθεράς σε έναν καταχωρητή. Η C-implement υποστηρίζει μόνο ακέραιες σταθερές, άρα η μόνη εντολή που χρειαζόμαστε είναι η li:

```
li reg, integer
# reg: register
# integer: integer arithmetic value
```

Κάπου εδώ ολοκληρώνονται όλες οι περιπτώσεις που πρέπει να φροντίσει η gnv1Code(). Ας δοκιμάσουμε να γράψουμε έναν υποτυπώδη ψευδοκώδικα για την loadvr(), σε υψηλό επίπεδο:

```
def loadvr(v,reg):
    if v is integer_constant:
        ...
    else:
        retrieve information for v from symbol table
        if v is global_variable:
            ... # using gp
```

```

        elif v is local_variable or parameter_by_value
            or temporary_variable:
                ...
                # using sp
        elif v is parameter_by_reference:
                ...
                # using sp
        elif v is (local_variable or parameter_by_value)
            in ancestor_function
                ...
                # using gnlvCode()
        elif v is parameter_by_reference in ancestor_function
                ...
                # using gnlvCode()

```

### 9.2.3 Η συνάρτηση storerv()

Η συνάρτηση storerv() δεν διαφέρει πολύ στην υλοποίησή της από την loadvr(). Κάθε μεταβλητή θα βρεθεί στη μνήμη με παρόμοιους μηχανισμούς με αυτούς που χρησιμοποιήθηκαν στην loadvr(). Η διαφοροποίηση βρίσκεται στις τελευταίες εντολές που παράγει η κάθε κλήση της storerv(), όπου αντί για εντολή ανάγνωσης `lw`, έχουμε εντολή αποθήκευσης `sw`.

Η σύνταξη της storerv() είναι η ακόλουθη:

```

def storerv(reg,v)
    # reg: source register
    # v: target variable

```

όπου `v` το όνομα της μεταβλητής, την τιμή της οποίας θέλουμε να διαβάσουμε, και `reg` το όνομα του καταχωρητή, στον οποίο θέλουμε να τοποθετηθεί.

Θα δούμε πάλι μία μία τις περιπτώσεις, μόνο που αυτή τη φορά θα είμαστε λιγότερο περιγραφικοί και για καθεμία περίπτωση θα παραθέσουμε μόνο τον κώδικα που θα παραχθεί, συνοδευόμενο από σχόλια μέσα στον κώδικα.

#### 9.2.3.1 Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή ή προσωρινή μεταβλητή

Εγγραφή στο τοπικό εγγράφημα δραστηριοποίησης, μέσω του `sp` και του `offset` της μεταβλητής:

```

sw reg,-offset(sp)
    # reg: source register
    # offset (from symbol table - distance from stack pointer)

```

Στο σημείο αυτό ίσως αναρωτηθεί κανείς για ποιο λόγο συμπεριλαμβάνουμε στην κατηγορία αυτή την παράμετρο που έχει περαστεί με τιμή. Το πέρασμα παραμέτρου με τιμή χρησιμοποιείται για να περαστούν δεδομένα προς τη συνάρτηση, ενώ μέσα από αυτήν την παράμετρο δεν είναι δυνατόν να επιστραφεί οποιαδήποτε πληροφορία στην καλούσα. Η τιμή της παραμέτρου αυτής βρίσκεται στη στοίβα της κληθείσας συνάρτησης και μετατρέπεται σε σκουπίδια μόλις ολοκληρωθεί η εκτέλεσή της και ο δείκτης στοίβας `sp` μεταφερθεί στο εγγράφημα δραστηριοποίησης της καλούσας. Για όσο καιρό όμως εκτελείται η συνάρτηση, η θέση μνήμης της τυπικής παραμέτρου εξακολουθεί να βρίσκεται δεσμευμένη στη στοίβα και όποιος έχει δικαίωμα πρόσβασης σε αυτή μπορεί να τη διαβάσει ή να την τροποποιήσει. Κατά τη διάρκεια, δηλαδή, της εκτέλεσης μιας συνάρτησης ή διαδικασίας η τυπική παράμετρος που έχει περαστεί με αναφορά έχει τη συμπεριφορά μιας τοπικής μεταβλητής η οποία είναι αρχικοποιημένη.

#### 9.2.3.2 Παράμετρος που έχει περαστεί με αναφορά

Ανάγνωση της διεύθυνσης και εγγραφή μέσω της διεύθυνσης:

```

lw t0,-offset($p)
    # t0 used as temporary register
    # offset (from symbol table - distance from stack pointer)
sw reg,(t0)
    # reg: source register
    # t0 used as index register

```

### 9.2.3.3 Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή η οποία ανήκει σε πρόγονο

Πρόσβαση μέσω της `gnlvcode()` και εγγραφή μέσω του `t0`, δηλαδή του αποτελέσματος της `gnlvcde()`:

```

call gnlvcde()
    # t0 has the address of the target variable
produce('sw reg,(t0)')
    # reg: source register
    # t0 used as index register

```

Το ίδιο σχόλιο που έγινε για την εγγραφή μίας τιμής σε μία τυπική παράμετρο, που έχει περαστεί με τιμή στην κληθείσα συνάρτηση, ισχύει και για την περίπτωση που η τυπική παράμετρος έχει περαστεί με τιμή σε κάποιο πρόγονο.

### 9.2.3.4 Παράμετρος που έχει περαστεί με αναφορά, η οποία ανήκει σε πρόγονο

Πρόσβαση μέσω της `gnlvcde()`, ανάγνωση της διεύθυνσης και εγγραφή μέσω της διεύθυνσης:

```

call gnlvcde()
    # t0 has the address of the address of the target variable
produce('lw t0,(t0)')
    # t0 has the address pf the target variable
produce('sw reg,(t0)')
produce('sw reg,(t0)')
    # reg: source register
    # t0 used as index register

```

### 9.2.3.5 Καθολική μεταβλητή

Εγγραφή μέσω του `gp` και του `offset` της μεταβλητής:

```

sw reg,-offset(gp)
    # reg: source register
    # offset (from symbol table - distance from global pointer)

```

### 9.2.3.6 Αριθμητική σταθερά

Εδώ υπάρχει μια μικρή διαφοροποίηση με την αντίστοιχη περίπτωση της `loadvr()`, αφού η αριθμητική σταθερά πρέπει πρώτα να φορτωθεί σε έναν καταχωρητή (π.χ. τον `t0`) και στη συνέχεια από εκεί να τοποθετηθεί στη μεταβλητή στόχο:

```

loadvr(1,t0)
storerv(t0,z)

```

Πίνακας 9.1: Μονάδες υλικού/λογισμικού που χρησιμοποιούνται από τις `loadvr()` και `storerv()`.

	χρήση βασικού δείκτη	χρήση συνδέσμου προσπέλασης	χρήση συνάρτησης <code>gnlvcde()</code>	χρήση καταχωρητή δείκτη <code>tθ</code>
τοπική μεταβλητή	<code>sp</code>	όχι	όχι	όχι
παράμετρος με τιμή	<code>sp</code>	όχι	όχι	όχι
προσωρινή μεταβλητή	<code>sp</code>	όχι	όχι	όχι
παράμετρος με αναφορά	<code>sp</code>	όχι	όχι	ναι
τοπική μεταβλητή σε πρόγονο	<code>sp</code>	ναι	ναι	ναι
παράμετρος με τιμή σε πρόγονο	<code>sp</code>	ναι	ναι	ναι
παράμετρος με αναφορά σε πρόγονο	<code>sp</code>	ναι	ναι	ναι
καθολική μεταβλητή	<code>gp</code>	όχι	όχι	όχι
αριθμητική σταθερά	όχι	όχι	όχι	όχι

Στον πίνακα 9.1 έχουν συγκεντρωθεί για κάθε περίπτωση τα αγαθά που χρησιμοποιεί η κάθε μεταβλητή ώστε να προσπελαστεί από τις `loadvr()` και `storerv()`. Χρησιμοποιήστε τον πίνακα για να βεβαιωθείτε ότι έχετε κατανοήσει την υλοποίηση της κάθε περίπτωσης και χωρίς να γυρίσετε πίσω στο διάβασμα, προσπαθήστε μέσα από αυτόν να επαναφέρετε τις υλοποιήσεις στη μνήμη σας.

Να σημειώσουμε ότι στην `storerv()` ο καταχωρητής `tθ` χρησιμοποιείται για την προσωρινή αποθήκευση της αριθμητικής σταθεράς, αλλά όχι ως καταχωρητής δείκτης. Για τον λόγο αυτό δεν σημειώνεται στην αντίστοιχη στήλη του πίνακα 9.1.

### 9.3 Εκχώρηση και αριθμητικές πράξεις

Στην ενότητα αυτή θα χρησιμοποιήσουμε τις βοηθητικές συναρτήσεις που φτιάξαμε παραπάνω για να παράγουμε τελικό κώδικα για τις αριθμητικές πράξεις και τις εκχωρήσεις.

Η εντολή του ενδιάμεσου κώδικα που εκχωρεί μία μεταβλητή σε μία άλλη θυμίζουμε ότι είναι η εξής:

```
:=, x, _, z      # z := x
```

Έχοντας διαθέσιμες τις `loadvr()` και `storerv()` είναι πολύ εύκολο να δημιουργήσουμε τον κώδικα που απαιτείται. Αρκεί να φορτώσουμε τη μεταβλητή `x` σε έναν καταχωρητή (π.χ. τον `tθ`) και στη συνέχεια από εκεί να τον μεταφέρουμε στη θέση μνήμης που βρίσκεται η μεταβλητή `z`. Τόσο η `loadvr()` όσο και η `storerv()` είναι φτιαγμένες έτσι, ώστε να καλύπτουν όλες τις περιπτώσεις και δεν υπάρχει λόγος να προβληματίζόμαστε πια για την ορθή μεταφορά της τιμής μιας μεταβλητής σε έναν καταχωρητή ή το αντίστροφο. Έτσι για να παραχθεί ο ζητούμενος κώδικας πρέπει να καλέσουμε αρχικά την `loadvr()` και μετά την `storerv()` με τις κατάλληλες παραμέτρους:

```
loadvr(x, tθ)
storerv(tθ, z)
```

Στην περίπτωση που έχουμε εκχώρηση αριθμητικής σταθεράς σε μεταβλητή, τότε απλά αντικαθιστούμε τη μεταβλητή `x` με την αριθμητική σταθερά. Έχουμε φροντίσει στην `storerv()` ώστε να λειτουργεί αυτό σωστά:

```
loadvr(integer, tθ)
storerv(tθ, z)
```

Η *C-imple* υποστηρίζει τις τέσσερις αριθμητικές πράξεις:

```
op, x, y, z      # z = x op y
```

```
# op: +, -, *, /
# x, y, z: variables
```

Για καθεμία από αυτές, οι δύο μεταβλητές x και y μεταφέρονται αντίστοιχα στους καταχωρητές t1 και t2, μέσω της `loadvr()`. Στη συνέχεια γίνεται η πράξη `add,sub,mul,div`, ανάλογα την πράξη που θέλουμε να εκτελεστεί. Σαν παράμετροι στις εντολές αυτές θα χρησιμοποιηθούν οι t1 και t2, ως καταχωρητές πάνω στους οποίους θα εκτελεστεί η πράξη, και ο t1 ως καταχωρητής στον οποίο θα τοποθετηθεί το αποτέλεσμα. Απομένει μόνο η μεταφορά του αποτελέσματος από τον καταχωρητή t1 στη μεταβλητή z. Για την εντολή ενδιάμεσου κώδικα:

```
+, x, y, z      # z = x + y
```

Θα γίνουν οι εξής κλήσεις:

```
loadvr(x,t1)
loadvr(y,t2)
produce('add t1,t2,t1')
storerv(t1,z)
```

## 9.4 Διακλαδώσεις

Η αποσύνθεση ενός προγράμματος σε τμήματα κώδικα και διακλαδώσεις που μεταφέρουν τον κώδικα από τμήμα σε τμήμα έγινε κατά την παραγωγή του ενδιάμεσου κώδικα. Στη φάση της παραγωγής του τελικού κώδικα καθεμία από τις εντολές αυτές πρέπει να παραγάγει έναν σχετικά απλό κώδικα ο οποίος θα εκτελεί ένα απλό άλμα στην περίπτωση της εντολής `jmp` του ενδιάμεσου κώδικα, ενώ ένα λογικό άλμα θα ακολουθεί τη σύγκριση δύο μεταβλητών στις επιλογές των λογικών αλμάτων.

Έτσι, για την εντολή ενδιάμεσου κώδικα:

```
jmp, _, _, label
```

Θα παραχθεί η εντολή:

```
j label
```

Εάν πρόκειται για το λογικό άλμα:

```
cond_jump_int_code, x, y, label
```

Θα εκτελεστούν οι παρακάτω βοηθητικές συναρτήσεις:

```
loadvr(x,t1)
loadvr(y,t2)
produce('cond_jump_fin_code t1,t2,label')
```

όπου η προφανής αντιστοιχία ανάμεσα στα `cond_jump_int_code` και `cond_jump_fin_code` φαίνεται στον πίνακα 9.2

## 9.5 Αρχή προγράμματος, κυρίως πρόγραμμα και τέλος προγράμματος

Εκκινώντας ένα πρόγραμμα, αυτό που πρέπει να εκτελεστεί είναι η πρώτη εντολή του κυρίως προγράμματος. Αυτή όμως δεν συμπίπτει με την πρώτη εντολή του τελικού κώδικα που έχει παραχθεί, αφού η μετάφραση ενός προγράμματος ακολουθεί τη σειρά εμφάνισης των εντολών στο αρχικό πρόγραμμα. Έτσι, οι εντολές του κυρίως προγράμματος θα μεταφραστούν τελευταίες, μετά τη μετάφραση όλων των συναρτήσεων και διαδικασιών.

Με την εκκίνηση του προγράμματος ο κώδικας πρέπει να εκτελέσει ένα άλμα στην πρώτη εντολή του κυρίως προγράμματος. Άρα, η πρώτη εντολή που θα παραχθεί, θα είναι μία `jmp`. Η ετικέτα της πρώτης

Πίνακας 9.2: Αντιστοίχιση λογικών αλμάτων στον ενδιάμεσο και στον τελικό κώδικα.

cond_jump_int_code	cond_jump_fin_code
==	beq
<>	bne
<	blt
>	bgt
<=	ble
>=	bge

εντολής του κυρίως προγράμματος δεν είναι ακόμα γνωστή, μπορούμε όμως να τοποθετήσουμε μία ετικέτα πριν από αυτήν και να κάνουμε το άλμα στην ετικέτα αυτή. Έτσι, αν ονομάσουμε την ετικέτα `main`, η πρώτη εντολή κάθε προγράμματος θα είναι:

L0:

```
j main
```

Όταν μεταβούμε στην ετικέτα `main`, τότε θα πρέπει να κάνουμε δύο ενέργειες για να αρχικοποιήσουμε δύο καταχωρητές, τον δείκτη στοίβας `sp` και τον καταχωρητή για τις καθολικές μεταβλητές `gp`.

Ο καταχωρητής `sp` δείχνει στην αρχή του χώρου που μας παραχωρήθηκε για τη στοίβα από το λειτουργικό σύστημα. Πρέπει να τον τοποθετήσουμε στην αρχή του εγγραφήματος δραστηριοποίησης του κυρίως προγράμματος. Άρα πρέπει να μεταφερθεί προς τα πάνω, τόσα bytes, όσο το εγγράφημα δραστηριοποίησης του κυρίως προγράμματος. Εκεί θα τοποθετηθούν οι καθολικές μεταβλητές, άρα εκεί θα τοποθετηθεί και ο `gp`.

Στην ετικέτα `main` θα υπάρχουν οι εξής δύο εντολές:

main:

```
addi sp,sp,framelength_main
mv gp,sp
```

L... :

Μετά ακολουθεί η ετικέτα της πρώτης εκτελέσιμης εντολής του κυρίως προγράμματος `L...`, όποια κι αν είναι αυτή.

Όταν ολοκληρωθεί η παραγωγή κώδικα για όλες τις εντολές του προγράμματος, θα φτάσουμε και στην τελευταία εντολή, η οποία είναι η `halt`. Με την `halt` θα επιστραφεί ο έλεγχος στο λειτουργικό σύστημα. Είδαμε στην ενότητα 9.1.7 ότι ο τερματισμός προγράμματος, επιστρέφοντας στο λειτουργικό σύστημα τον αριθμό 0, γίνεται με τις εντολές:

```
li a0,0
li a7,93
ecall
```

Συνεπώς, το παράδειγμα `fibonacci.ci` του κεφαλαίου 3 θα παραγάγει τον εξής κώδικα:

L0:

```
j main
```

L1:

```
# code for fibonacci function
```

```
...
```

main:

```
addi sp,sp,framelength_main
mv gp,sp
```

L... :

```

# code for main program
...
# code for halt,_,_,_
li a0,0
li a7,93
ecall

```

## Βιβλιογραφία

- [1] Andrew Waterman και Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Αδημοσίευτη ερευνητική εργασία. EECS Department, University of California, Berkeley, 2019.
- [2] Andrew Waterman, Krste Asanović και John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Αδημοσίευτη ερευνητική εργασία. EECS Department, University of California, Berkeley, 2021.
- [3] Keith D. Cooper και Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2004. ISBN: 1558606998.
- [4] Keith D. Cooper και Linda Torczon. *Σχεδίαση και Κατασκευή Μεταγλωττιστών*. Ξενόγλωσσος τίτλος: Engineering a Compiler, Επιστημονική επιμέλεια έκδοσης: Γεώργιος Μανής, Νικόλαος Παπασπύρου και Αντώνιος Σαββίδης. Πανεπιστημιακές Εκδόσεις Κρήτης, 2018. ISBN: 9789605245191.
- [5] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 0321486811.
- [6] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Μεταγλωττιστές: Αρχές, τεχνικές και εργαλεία, 2η αμερικανική έκδοση*. Επιστημονική επιμέλεια έκδοσης: Π. Αλεφραγκής, Α. Βώρος, Ν. Βώρος, Κ. Μασσέλος. Εκδόσεις Νέων Τεχνολογιών, 2011. ISBN: 9789606759727.
- [7] Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.



## ΚΕΦΑΛΑΙΟ 10

---

# ΠΕΡΑΣΜΑ ΠΑΡΑΜΕΤΡΩΝ, ΚΛΗΣΗ ΣΥΝΑΡΤΗΣΕΩΝ ΚΑΙ ΔΙΑΔΙΚΑΣΙΩΝ ΣΤΟΝ ΤΕΛΙΚΟ ΚΩΔΙΚΑ

---

### Σύνοψη:

Στο πρώτο κεφάλαιο του τελικού κώδικα είδαμε κυρίως πώς προσπελάζουμε τις μεταβλητές που βρίσκονται τοποθετημένες στη στοίβα. Στο δεύτερο αυτό κεφάλαιο θα περιγράψουμε με λεπτομέρεια πώς γίνονται η κλήση μιας συνάρτησης ή διαδικασίας, το πέρασμα παραμέτρων με αναφορά και τιμή, η επιστροφή αποτελέσματος των συναρτήσεων, αλλά και η επιστροφή του ελέγχου ροής όταν μία συνάρτηση ή διαδικασία τερματιστεί.

Πρώτα θα δούμε τις εντολές που θα δημιουργηθούν προκειμένου να περάσουμε μία παράμετρο με τιμή, αντιγράφοντας στο νέο εγγράφημα δραστηριοποίησης την τιμή της. Στη συνέχεια θα κάνουμε τις αντίστοιχες ενέργειες ώστε να αντιγραφεί στο νέο εγγράφημα δραστηριοποίησης η διεύθυνση μίας μεταβλητής που περνάει με αναφορά. Η επιστροφή τιμής μιας συνάρτησης ακολουθεί έναν παρόμοιο μηχανισμό με τον μηχανισμό περάσματος παραμέτρου με αναφορά.

Η κλήση μίας συνάρτησης ή διαδικασίας προϋποθέτει την ενεργοποίηση του νέου εγγραφήματος δραστηριοποίησης και της μεταβίβασης του ελέγχου στη νέα συνάρτηση. Μετά το πέρας της εκτέλεσης, ο έλεγχος πρέπει να επιστρέψει στην καλούσα συνάρτηση ή διαδικασία και να ενεργοποιηθεί ξανά το εγγράφημα δραστηριοποίησης της καλούσας.

### Προαπαιτούμενη γνώση:

- στοιχεία γλώσσας μηχανής
  - κεφάλαιο 6
  - κεφάλαιο 7
  - κεφάλαιο 8
  - κεφάλαιο 9
-

Όπως και στο προηγούμενο κεφάλαιο, η μεταγλώττιση θα γίνει οδηγούμενη από τις εντολές του ενδιάμεσου κώδικα, εντολή προς εντολή.

Η κλήση μιας συνάρτησης ή διαδικασίας προϋποθέτει τη δημιουργία ενός νέου εγγραφήματος δραστηριοποίησης και την τοποθέτηση σε αυτό όποιας πληροφορίας απαιτείται για τη λειτουργία του, συμπεριλαμβανομένης και της πληροφορίας που σχετίζεται με το πέρασμα παραμέτρων και επιστροφή αποτελέσματος. Προϋποθέτει επίσης τη μετάβαση του ελέγχου σε αυτήν, αλλά και την επιστροφή του ελέγχου στην καλούσα συνάρτηση, όταν η κληθείσα ολοκληρωθεί.

## 10.1 Πέρασμα παραμέτρων

Οι παράμετροι μιας συνάρτησης θα τοποθετηθούν στο εγγράφημα δραστηριοποίησης αμέσως επάνω από τα 12 δεσμευμένα bytes για τη διεύθυνση επιστροφής, τον σύνδεσμο προσπέλασης και τη διεύθυνση επιστροφής τιμής της συνάρτησης. Η σειρά τοποθέτησης είναι αυτή της εμφάνισής τους.

Για διευκόλυνση θα χρησιμοποιήσουμε τον καταχωρητή `fp` ως δείκτη στο νέο εγγράφημα δραστηριοποίησης. Θα τοποθετήσουμε, δηλαδή, τον `fp` στην αρχή του εγγραφήματος δραστηριοποίησης της νέας συνάρτησης ή διαδικασίας, εκεί που θα τοποθετηθεί ο `sp` όταν ξεκινήσει η εκτέλεσή της. Έτσι, θα δημιουργήσουμε εύκολη πρόσβαση στο υπό δημιουργία εγγράφημα δραστηριοποίησης.

Πριν κάνουμε οποιεσδήποτε ενέργειες για το πέρασμα της πρώτης παραμέτρου, τοποθετούμε τον `fp` στη θέση του. Προσθέτουμε στον `sp` τόσα bytes, από όσα αποτελείται το εγγράφημα δραστηριοποίησης της καλούσας, το σημείο δηλαδή στο οποίο θα τοποθετηθεί το εγγράφημα δραστηριοποίησης της κληθείσας. Για τον σκοπό αυτόν παράγουμε την εντολή:

```
addi fp,sp,framelength
# framelength is the size of the activation record
# of the calling procedure/function
```

Η παραπάνω εντολή θα δημιουργηθεί όταν συναντάται η πρώτη `par` στον ενδιάμεσο κώδικα. Αν η κληθείσα δεν έχει παραμέτρους, τότε δεν υπάρχει κάποια `par` που να προηγείται της `call`, και η εντολή `addi` θα δημιουργηθεί όταν εμφανιστεί η `call`.

Μετά την τοποθέτηση του `fp` στην αρχή του υπό δημιουργία εγγραφήματος δραστηριοποίησης, για κάθε παράμετρο που συναντάμε (`par`), και ανάλογα με το αν αυτή περνά με τιμή ή αναφορά, κάνουμε τις ανάλογες ενέργειες, οι οποίες αναλύονται στις ακόλουθες ενότητες.

### 10.1.1 Πέρασμα παραμέτρων με τιμή

Κατά το πέρασμα μιας παραμέτρου με τιμή, η τιμή της παραμέτρου αντιγράφεται στο εγγράφημα δραστηριοποίησης της υπό δημιουργία συνάρτησης, στη θέση που έχει δεσμευτεί για την παράμετρο αυτή.

Αρχικά, η τιμή της παραμέτρου θα αναζητηθεί και θα τοποθετηθεί προσωρινά σε έναν καταχωρητή. Η αναζήτηση και η τοποθέτηση της παραμέτρου σε καταχωρητή μπορεί να γίνει χρησιμοποιώντας τη βοηθητική συνάρτηση `loadvr()`. Η `loadvr()` θα αντλήσει την απαιτούμενη πληροφορία από τον πίνακα συμβόλων και θα παραγάγει τον κώδικα που θα αναζητήσει την τιμή της παραμέτρου, είτε τοπικά, είτε στο επίπεδο των καθολικών μεταβλητών, είτε σε επίπεδα προγόνων της υπό δημιουργία συνάρτησης. Με την ολοκλήρωση της `loadvr()` ο κώδικας που θα έχει παραχθεί θα τοποθετεί, όταν εκτελεστεί, την τιμή της ζητούμενης παραμέτρου σε κάποιον καταχωρητή. Έστω ότι ο καταχωρητής που επιλέξαμε ως ενδιάμεσος είναι ο καταχωρητής `t0`.

Το επόμενο βήμα είναι η αντιγραφή της τιμής της παραμέτρου από τον `t0` στην κατάλληλη θέση στη στοίβα. Για τον υπολογισμό της θέσης της στη στοίβα υπολογίζεται ο χώρος που έχουν καταλάβει οι παράμετροι που έχουν ήδη τοποθετηθεί εκεί. Η νέα παράμετρος θα καταλάβει τις επόμενες διαθέσιμες θέσεις.

Επειδή στη `C-impl` έχουμε μόνο ακέραιες μεταβλητές, κάθε παράμετρος καταλαμβάνει 4 bytes. Άρα η

*i*-οστή παράμετρος μίας συνάρτησης θα έχει offset:

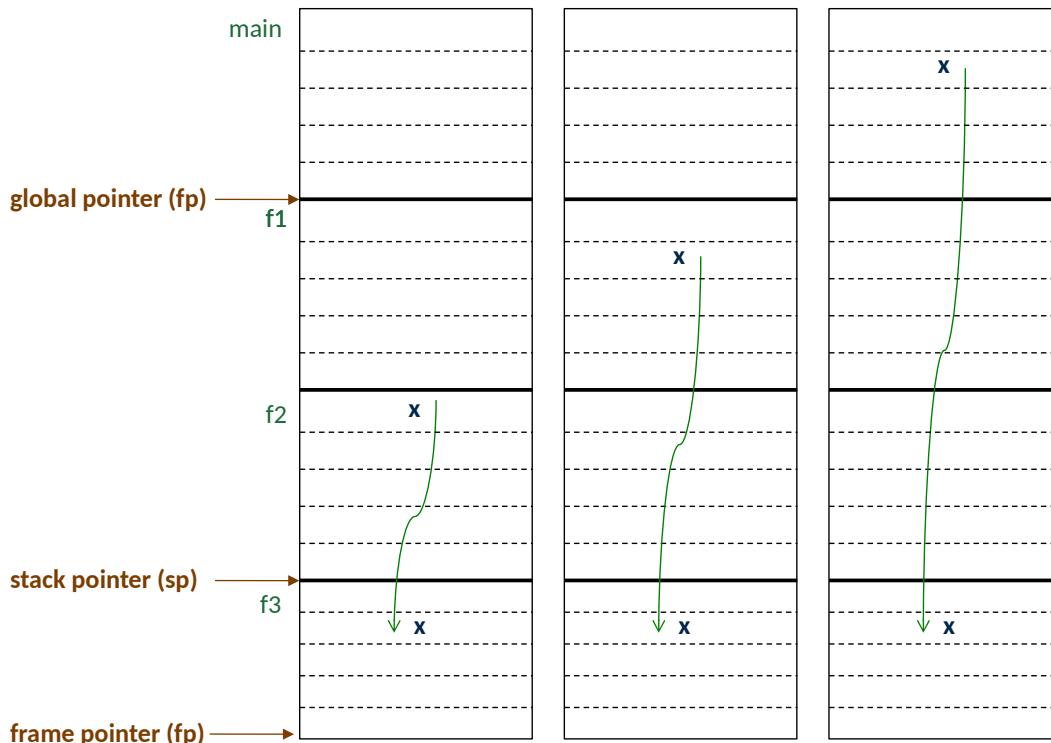
$$d = 12 + (i - 1) * 4 \text{ bytes} \quad (10.1)$$

Θυμίζουμε ότι στα 12 πρώτα bytes του εγγραφήματος δραστηριοποίησης αποθηκεύεται πληροφορία απαραίτητη για τη λειτουργία της συνάρτησης. Θα δούμε πιο αναλυτικά παρακάτω, στο κεφάλαιο αυτό, τι ακριβώς αποθηκεύεται εκεί και για ποιο λόγο.

Η εντολή που θα αντιγράψει την τιμή της παραμέτρου από τον καταχωρητή *t0* στην κατάλληλη θέση στη στοίβα είναι η εξής:

```
sw t0, -d(fp)
```

όπου το *d* δίνεται από εξίσωση 10.1 και είναι ακέραιος αριθμός. Θυμίζουμε ότι έχουμε ήδη μεταφέρει τον *fp* στην αρχή του εγγραφήματος δραστηριοποίησης της καλούσας συνάρτησης.



**Σχήμα 10.1:** Πέρασμα παραμέτρων με τιμή. Αριστερά, όταν η παράμετρος βρίσκεται στο εγγράφημα δραστηριοποίησης της καλούσας. Στη μέση, όταν βρίσκεται σε κάποιον πρόγονο. Δεξιά, όταν αποτελεί καθολική μεταβλητή.

Στο σχήμα 10.1 φαίνονται τρεις περιπτώσεις περάσματος παραμέτρου με τιμή, ανάλογα με το πού βρίσκεται η παράμετρος που θα περαστεί. Η συνάρτηση *f2* είναι αυτή που τη στιγμή αυτή εκτελείται. Γι' αυτό και ο stack pointer δείχνει στο εγγράφημα δραστηριοποίησης της *f2*. Η *f2* έχει κληθεί από την *f1*, η οποία με τη σειρά της έχει κληθεί από την *main*. Η *f2* δημιουργεί την *f3*, ώστε να της περάσει τον έλεγχο μόδις ολοκληρωθεί η δημιουργία του εγγραφήματος δραστηριοποίησης. Για τον λόγο αυτό ο *fp* έχει τοποθετηθεί στην αρχή του εγγραφήματος δραστηριοποίησης της *f3* και μέσω του *fp* έχουμε εύκολη πρόσβαση σε αυτό. Υποθέτουμε ότι θέλουμε να περάσουμε ως παράμετρο την τιμή *x* και ότι η *x* είναι η πρώτη παράμετρος της *f3*. Άρα η *x* θα τοποθετηθεί στη θέση *d* = 12 του εγγραφήματος δραστηριοποίησης της *f3*.

Όπως φαίνεται στις τρεις περιπτώσεις του σχήματος 10.1 η παράμετρος μπορεί να βρίσκεται στο εγγράφημα δραστηριοποίησης της καλούσας συνάρτησης (αριστερά στο σχήμα), σε κάποιον πρόγονο (στη μέση) ή μπορεί να αποτελεί καθολική μεταβλητή (δεξιά). Μία μεταβλητή μπορεί να βρεθεί στο εγγράφημα δραστηριοποίησης της συνάρτησης που εκτελείται αν πρόκειται για τοπική μεταβλητή, παράμετρο που έχει

περαστεί με τιμή στη συνάρτηση που εκτελείται ή αποτελεί προσωρινή μεταβλητή. Μία μεταβλητή μπορεί να βρεθεί στο εγγράφημα δραστηριοποίησης μίας συνάρτησης προγόνου, αν αποτελεί τοπική μεταβλητή ή παράμετρο που έχει περαστεί με τιμή στη συνάρτηση πρόγονο. Μία μεταβλητή είναι καθολική όταν έχει δηλωθεί στο κυρίως πρόγραμμα.

### 10.1.2 Πέρασμα παραμέτρων με αναφορά

Κατά το πέρασμα παραμέτρων με τιμή αντιγράφεται στο εγγράφημα δραστηριοποίησης της υπό δημιουργία συνάρτησης η τιμή της μεταβλητής που περνάμε ως παράμετρο. Κατά το πέρασμα παραμέτρων με αναφορά αντιγράφεται στο εγγράφημα δραστηριοποίησης της υπό δημιουργίας συνάρτησης η διεύθυνση της μεταβλητής αυτής. Η πρόσβαση στη μεταβλητή που περάστηκε ως παράμετρος γίνεται μέσω αυτής της διεύθυνσης.

Ο μηχανισμός περάσματος είναι πρακτικά αυτός που ακολουθούμε για να περάσουμε μία παράμετρο με αναφορά στη γλώσσα προγραμματισμού C. Το πέρασμα με αναφορά γίνεται με τον ίδιο τρόπο σε όλες τις γλώσσες προγραμματισμού, με τη μόνη διαφορά ότι στις υπόλοιπες γλώσσες προγραμματισμού οι λεπτομέρειες υλοποίησης κρύβονται από τον προγραμματιστή, ενώ στη C η διαχείριση/υλοποίηση είναι ευθύνη του προγραμματιστή.

Η παραγωγή κώδικα για το πέρασμα παραμέτρου με τιμή είναι κάτι σχετικά απλό, όσον αφορά την υλοποίησή της, δεδομένου ότι είχε ήδη υλοποιηθεί η συνάρτηση `loadvr()`, η οποία παρήγαγε μέρος του κώδικα και απλοποίησε σημαντικά τη διαδικασία. Στο πέρασμα παραμέτρου με αναφορά, η υλοποίηση απαιτεί περισσότερη προσπάθεια, αφού δεν έχουμε υλοποιήσει κάποια αντίστοιχη συνάρτηση που να μας τοποθετεί τη διεύθυνση μιας μεταβλητής σε κάποιον καταχωρητή. Έτσι, θα πρέπει να υλοποιήσουμε ολόκληρη την παραγωγή του κώδικα και να διαχωρίσουμε περιπτώσεις ανάλογα με αυτό που μας επιστρέφει ο πίνακας συμβόλων.

Θα διαχωρίσουμε δύο βασικές περιπτώσεις. Στην πρώτη, στη θέση της στοίβας που μας παραπέμπει ο πίνακας συμβόλων βρίσκεται η τιμή της μεταβλητής που θέλουμε να περάσουμε ως παράμετρο. Στη δεύτερη, στη θέση αυτή, βρίσκεται η διεύθυνσή της.

Όταν περνάμε μία παράμετρο, στη στοίβα της νέας συνάρτησης ή διαδικασίας τοποθετείται είτε η τιμή της παραμέτρου, είτε η διεύθυνσή της. Αν πρόκειται για την τιμή, επειδή στην C-implement έχουμε μόνο ακέραιες μεταβλητές, κάθε παράμετρος καταλαμβάνει στη στοίβα 4 bytes. Αν πρόκειται για διεύθυνση χρειαζόμαστε πάλι 4 bytes, αφού και μία διεύθυνση χρειάζεται 4 bytes για να αποθηκευτεί. Έτσι, η i-οστή παράμετρος μίας συνάρτησης ή διαδικασίας θα απέχει από την αρχή του εγγραφήματος δραστηριοποίησης (offset):

$$d = 12 + (i - 1) * 4 \text{ bytes} \quad (10.2)$$

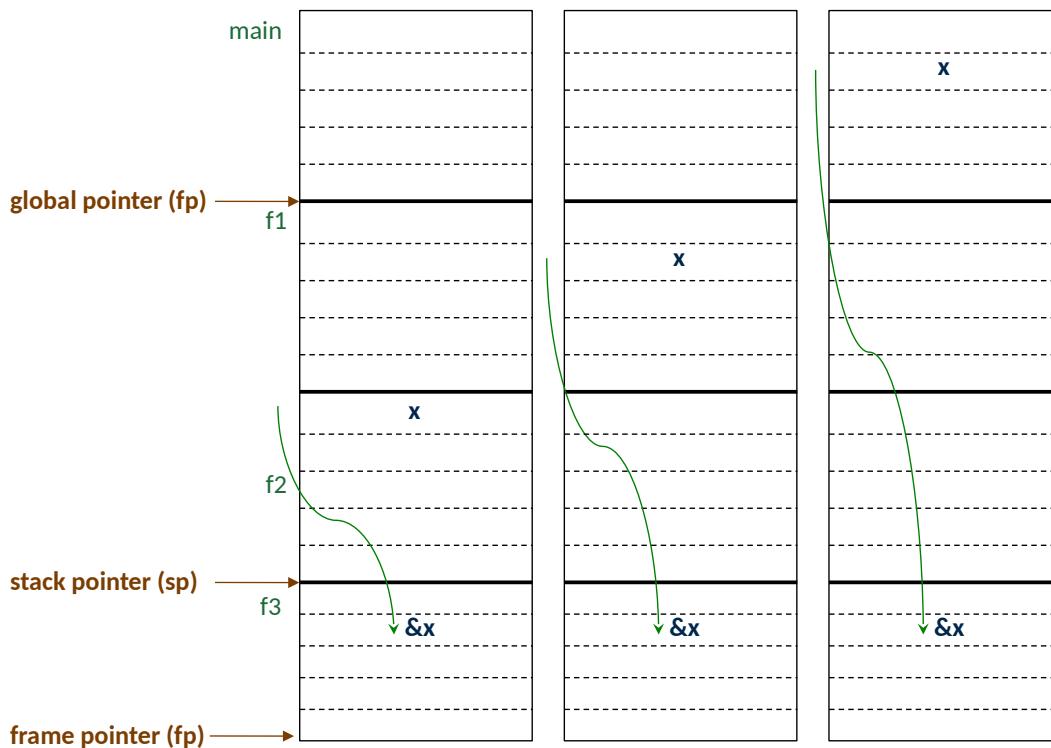
#### 10.1.2.1 Παράμετρος με αναφορά, όταν στη στοίβα είναι αποθηκευμένη η τιμή της παραμέτρου

Πρόκειται για τις περιπτώσεις που η μεταβλητή που περνιέται ως παράμετρος είναι α) τοπική μεταβλητή ή προσωρινή μεταβλητή ή παράμετρος που έχει περαστεί με τιμή στη συνάρτηση, β) τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή σε συνάρτηση πρόγονο ή γ) καθολική μεταβλητή. Για καθεμία από τις περιπτώσεις αυτές δημιουργούμε διαφορετικό κώδικα, αλλά το ζητούμενο είναι πάντα το ίδιο: η διεύθυνση της μεταβλητής που περνιέται ως παράμετρος θα πρέπει να αντιγραφεί στην κατάλληλη θέση του εγγραφήματος δραστηριοποίησης που δημιουργείται. Ας δούμε τις περιπτώσεις αυτές περισσότερο αναλυτικά:

α) τοπική μεταβλητή ή προσωρινή μεταβλητή ή παράμετρος που έχει περαστεί με τιμή στη συνάρτηση:

Σε όποια από τις τρεις αυτές περιπτώσεις ανήκει η μεταβλητή, αυτή βρίσκεται offset bytes πάνω από τον sp. Άρα στη θέση d bytes πάνω από τον fp θα τοποθετήσουμε το offset(sp):

```
addi t0, sp, -offset
sw t0, - d(fp)
```



Σχήμα 10.2: Πέρασμα παραμέτρων με αναφορά, όταν στο εγγράφημα δραστηριοποίησης είναι αποθηκευμένη η τιμή της μεταβλητής. Αριστερά, όταν η παράμετρος βρίσκεται στο εγγράφημα δραστηριοποίησης της καλούσας. Στη μέση, όταν βρίσκεται σε κάποιον πρόγονο. Δεξιά, όταν αποτελεί καθολική μεταβλητή.

β) τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή σε συνάρτηση πρόγονο στην οποία ανήκει:

Η κλήση της `gnlvcode()` θα μεταφέρει τη διεύθυνση της μεταβλητής που θέλουμε να περάσουμε ως παράμετρο στον καταχωρητή `t0`. Από τον `t0` θα αντιγραφεί στη θέση `d` bytes πάνω από τον `fp`, η οποία είναι δεσμευμένη για την παράμετρο αυτή:

```
call gnlvcode(x)
sw t0, -d(fp)
```

γ) καθολική μεταβλητή:

Αφού πρόκειται για καθολική μεταβλητή, αυτή βρίσκεται τοποθετημένη offset bytes πάνω από τον `gp`. Άρα στη θέση `d` bytes πάνω από τον `fp` θα τοποθετήσουμε το `offset(gp)`.

```
addi t0, gp, -offset
sw t0, -d(gp)
```

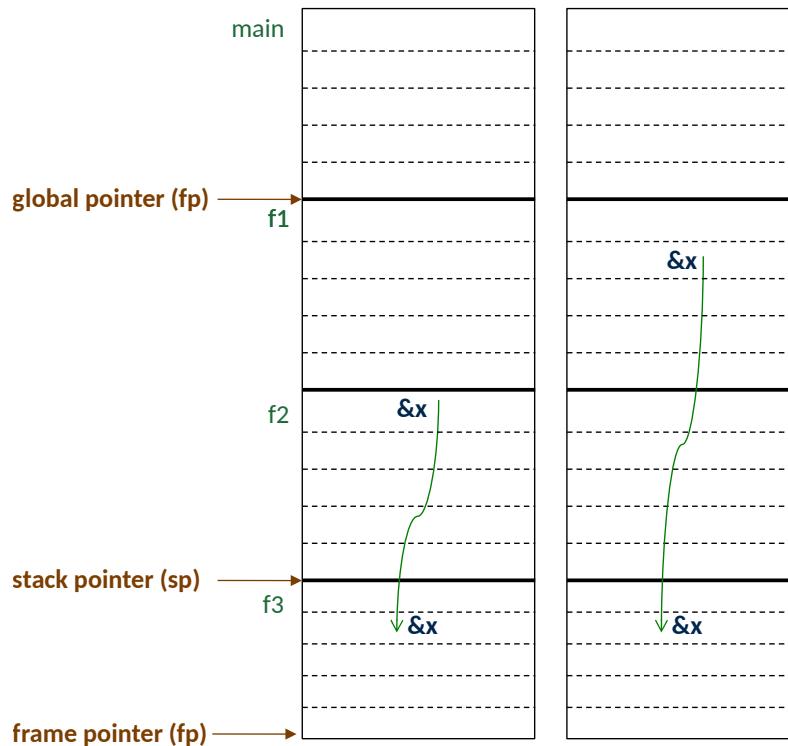
Οι τρεις αυτές περιπτώσεις συνοψίζονται στο σχήμα 10.2.

### 10.1.2.2 Παράμετρος με αναφορά, όταν στη στοίβα βρίσκεται η διεύθυνση της παραμέτρου

Για να υπάρχει στη στοίβα αποθηκευμένη η διεύθυνση της μεταβλητής που θέλουμε να περάσουμε ως παράμετρο και όχι η τιμή της, σημαίνει ότι η μεταβλητή που θέλουμε να περάσουμε ως παράμετρο με αναφορά είναι α) παράμετρος που έχει περαστεί με αναφορά στην καλούσα συνάρτηση ή διαδικασία β) παράμετρος που έχει περαστεί με αναφορά σε κάποιον πρόγονο. Θα εξετάσουμε τις δύο περιπτώσεις χωριστά, τις οποίες μπορείτε να δείτε στο σχήμα 10.3.

α) παράμετρος που έχει περαστεί με αναφορά στην καλούσα συνάρτηση ή διαδικασία:

Στην περίπτωση αυτή, στη θέση `offset` πάνω από τον `sp`, υπάρχει τοποθετημένη η διεύθυνση της μεταβλητής που έχει περαστεί ως παράμετρος με αναφορά στην καλούσα. Τοποθετήθηκε εκεί κατά τη δημιουργία του εγγραφήματος δραστηριοποίησης της καλούσας. Από εκεί, πρέπει να αντιγραφεί στη θέση `d` που



Σχήμα 10.3: Πέρασμα παραμέτρων με αναφορά, όταν στο εγγράφημα δραστηριοποίησης είναι αποθηκευμένη η διεύθυνση της μεταβλητής. Αριστερά, όταν η παράμετρος βρίσκεται στο εγγράφημα δραστηριοποίησης της καλούντας. Δεξιά, όταν βρίσκεται σε κάποιον πρόγονο.

έχουμε δεσμεύσει για την παράμετρο αυτή στην κληθείσα, την οποία προσπελάζουμε μέσω του fp:

```
lw t0, -offset(sp)
sw t0, -d(fp)
```

β) παράμετρος που έχει περαστεί με αναφορά σε συνάρτηση ή διαδικασία πρόγονο:

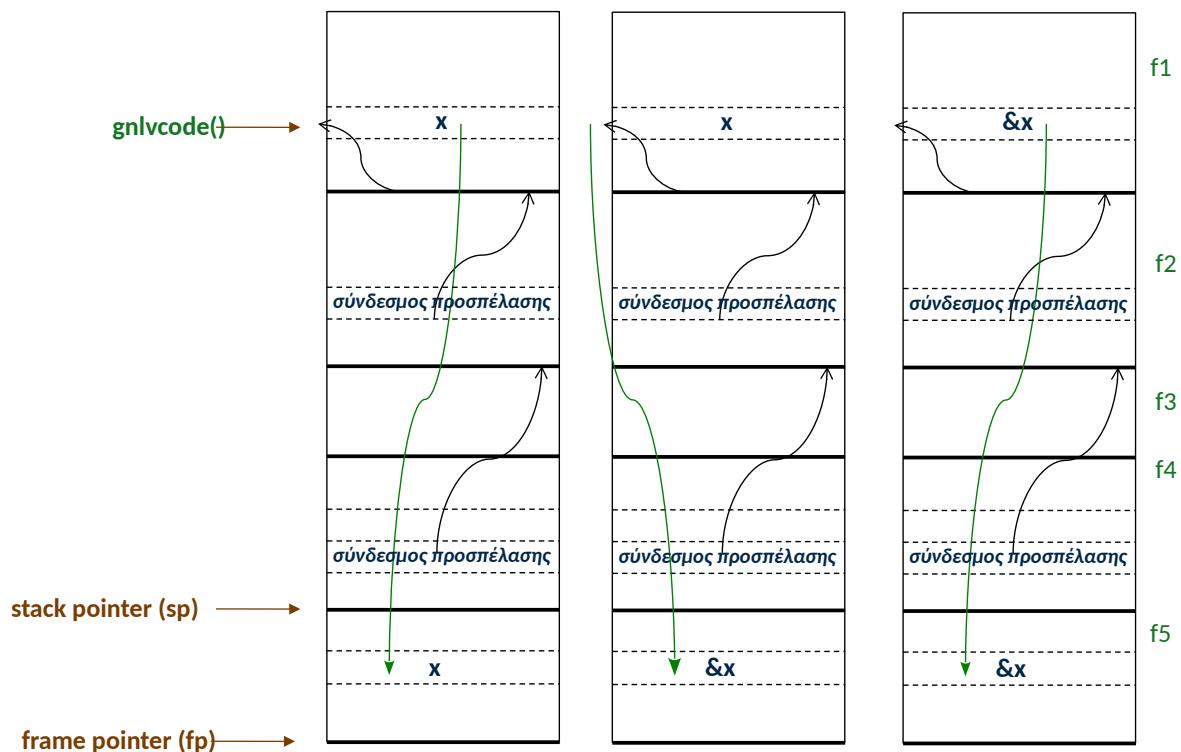
Η πρόσβαση σε θέση της στοίβας ενός προγόνου γίνεται με την `gnlvcode()`. Η `gnlvcode()` τοποθετεί στον καταχωρητή t0 τη διεύθυνση της πληροφορίας που επιστρέφει ο πίνακας συμβόλων. Εκεί βρίσκεται η διεύθυνση της μεταβλητής που μας ενδιαφέρει. Δεν έχουμε παρά να την αντιγράψουμε στη θέση μνήμης που έχουμε δεσμεύσει στο νέο εγγράφημα δραστηριοποίησης. Σημειώστε ότι χρειαζόμαστε ένα βήμα περισσότερο από την προηγούμενη περίπτωση, αφού η `gnlvcode()` θα τοποθετήσει στον καταχωρητή t0 τη διεύθυνση στη στοίβα στην οποία βρίσκεται η διεύθυνση που αναζητούμε. Ο κώδικας που αντιστοιχεί στα παραπάνω ακολουθεί:

```
call gnlvcode(x)
lw t0, (t0)
sw t0, -d(fp)
```

#### 10.1.2.3 Παράδειγμα περάσματος παραμέτρου με τη χρήση της `gnlvcode()`

Η `gnlvcode()` έχει χρησιμοποιηθεί σε τρεις από τις περιπτώσεις που εξετάσαμε παραπάνω και οι οποίες συγκεντρώνονται στο σχήμα 10.4. Και στις τρεις περιπτώσεις η πληροφορία που χρειαζόμαστε βρίσκεται αποθηκευμένη σε κάποιον πρόγονο. Αριστερά στο σχήμα έχουμε πέρασμα με τιμή, στη μέση πέρασμα με αναφορά όταν περνάμε τοπική μεταβλητή ή παράμετρο που έχει περαστεί με αναφορά στη συνάρτηση πρόγονο και δεξιά όταν περνάμε παράμετρο που έχει περαστεί με αναφορά στη συνάρτηση πρόγονο.

Και στις τρεις περιπτώσεις θα ανεβάσουμε με την `gnlvcode()` τον t0 να δείχνει στην πληροφορία που μας ενδιαφέρει. Στο παράδειγμα αυτό έχουμε τέσσερις συναρτήσεις, οι οποίες έχουν εκκινήσει την εκτέλεσή



Σχήμα 10.4: Κατά το πέρασμα παραμέτρου με αναφορά, η τιμή ή η διεύθυνση της οποίας είναι αποθηκευμένη σε κάποιον πρόγονο, εκμεταλλευόμαστε τους συνδέσμους προσπέλασης για να εντοπίσουμε τη διεύθυνση της μεταβλητής που θα περαστεί ως παράμετρος.

τους ( $f_1, f_2, f_3, f_4$ ), με την  $f_4$  να είναι αυτή που αυτή τη στιγμή εκτελείται και η οποία δημιουργεί την  $f_5$  για να της περάσει αργότερα τον έλεγχο. Η  $f_1$  έχει καλέσει την  $f_2$ , η οποία έχει καλέσει την  $f_3$ , και με τη σειρά της την  $f_4$ . Γονέας της  $f_4$  είναι η  $f_2$  και παππούς η  $f_1$ , όπως μπορεί να συμπεράνει κανείς από τους συνδέσμους προσπέλασης. Έτσι, η αναζήτηση εκκινεί από την  $f_4$ , από τον σύνδεσμο προσπέλασης της οποίας μεταβαίνουμε στην  $f_2$ . Επαναλαμβάνοντας το ίδιο βήμα, και χρησιμοποιώντας τον σύνδεσμο προσπέλασης της  $f_2$ , φτάνουμε στο εγγράφημα δραστηριοποίησης της  $f_1$ . Αφαιρώντας το offset φτάνουμε στην πληροφορία που αναζητούμε.

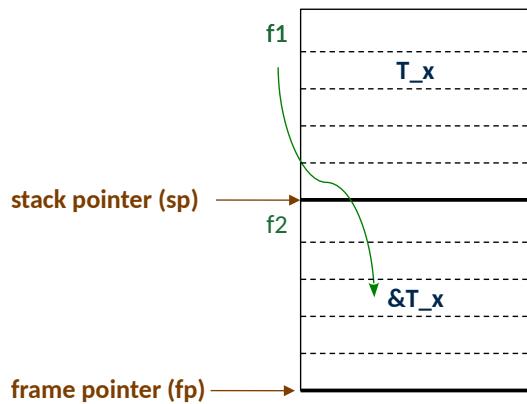
Αν πρόκειται για την πρώτη ή την τρίτη περίπτωση του σχήματος 10.4, πρέπει να αντιγράψουμε στο νέο εγγράφημα δραστηριοποίησης την πληροφορία που υπάρχει στη θέση μνήμης που δείχνει ο  $t\theta$ . Απλά, στη μία περίπτωση αυτή θα είναι τιμή, ενώ στην άλλη θα είναι διεύθυνση.

Μας μένει ακόμα η μεσαία περίπτωση του σχήματος 10.4, όπου εκεί αυτό που θέλουμε να αντιγράψουμε είναι η διεύθυνση του  $x$ . Αυτή βρίσκεται τοποθετημένη από την  $gnlvcde()$  στον  $t\theta$  και από εκεί αντιγράφεται στην  $f_5$ .

### 10.1.3 Επιστροφή τιμής συνάρτησης

Όπως είδαμε κατά την παραγωγή ενδιάμεσου κώδικα, η επιστροφή της τιμής μιας συνάρτησης συμβολίζεται στον ενδιάμεσο κώδικα πάλι με την εντολή `ret`, κάτι που δεν εκπλήσσει αφού κάλιστα η επιστροφή της τιμής μιας συνάρτηση μπορεί να γίνει με έναν παραπλήσιο μηχανισμό περάσματος με τους μηχανισμούς περάσματος παραμέτρων που είδαμε παραπάνω.

Αν το σκεφτούμε περισσότερο, θα διαπιστώσουμε ότι η επιστροφή τιμής θα μπορούσε να γίνει με τη χρήση ενός δείκτη, όπως ακριβώς κάναμε κατά το πέρασμα παραμέτρων με αναφορά. Η σημαντικότερη διαφορά είναι ότι, ενώ στο πέρασμα παραμέτρων με αναφορά η μεταβλητή που περνιέται ως παράμετρος μπορεί να είναι οτιδήποτε, η τιμή μίας συνάρτησης επιστρέφεται πάντοτε σε μία τοπική (προσωρινή) μεταβλητή της καλούσας.



Σχήμα 10.5: Κατά την κλήση συνάρτησης, η διεύθυνση μιας προσωρινής μεταβλητής αποθηκεύεται στη διεύθυνση επιστροφής τιμής συνάρτησης, ώστε να μπορεί να επιστραφεί το αποτέλεσμα της κληθείσας στην καλούσα.

Έτσι, όταν μεταφράζουμε την εντολή:

```
par, ret, x, _
```

πρέπει να μεταφέρουμε τη διεύθυνση της προσωρινής μεταβλητής  $x$  της καλούσας στην τρίτη θέση του εγγραφήματος δραστηριοποίησης της κληθείσας.

```
add t0, sp, -offset      # store return value address to t0
sw t0, -8(fp)           # store t0 to the reserved address in the stack
```

Σχηματικά ο μηχανισμός επιστροφής τιμής μίας συνάρτησης φαίνεται στο σχήμα 10.5.

Όταν, τώρα, συναντάται η εντολή ενδιάμεσου κώδικα:

```
ret, z, _, _
```

τότε η τιμή του  $z$  θα αντιγραφεί, μέσω του δείκτη που είναι αποθηκευμένος στην τρίτη θέση του εγγραφήματος δραστηριοποίησης της κληθείσας, στην προσωρινή μεταβλητή που έχει δημιουργηθεί για τον σκοπό αυτόν στην καλούσα.

Έτσι, αρχικά μεταφέρεται από τη θέση  $-8(sp)$  στον  $t0$  η διεύθυνση της προσωρινής μεταβλητής της καλούσας, η τιμή του  $x$  στον  $t1$  και στη συνέχεια, χρησιμοποιώντας τον  $t0$ , αποθηκεύονται την τιμή του  $t1$ .

```
lw t0, -8(sp)          # return value address to t0
lw t1, -offset(sp)     # return value to t1
sw t1, (t0)             # write t1 through t0
```

## 10.2 Κλήση συνάρτησης ή διαδικασίας

Μετά την ολοκλήρωση του περάσματος των παραμέτρων μίας συνάρτησης ή διαδικασίας, ακολουθεί η εντολή `call`, η οποία θα παραγάγει τον κώδικα για την ολοκλήρωση του εγγραφήματος δραστηριοποίησης της μεταβίβασης του ελέγχου από την καλούσα συνάρτηση ή διαδικασία στην κληθείσα και την επαναφορά του ελέγχου από την κληθείσα στην καλούσα, όταν η εκτέλεση της κληθείσας ολοκληρωθεί.

### 10.2.1 Σύνδεσμος προσπέλασης

Η πρώτη ενέργεια που θα κάνει η εντολή `call` είναι να συμπληρώσει το δεύτερο πεδίο του εγγραφήματος δραστηριοποίησης, τον σύνδεσμο προσπέλασης.

Ο σύνδεσμος προσπέλασης πρέπει να δείχνει στο εγγράφημα δραστηριοποίησης του γονέα της συνάρτησης ή της διαδικασίας, δηλαδή της συνάρτησης ή της διαδικασίας μέσα στην οποία είναι αυτή εμφωλευμένη. Στο εγγράφημα δραστηριοποίησης του γονέα θα πρέπει να ανατρέξουμε ώστε να ξεκινήσει η αναζήτηση μίας εγγραφής του πίνακα συμβόλων, την οποία η κληθείσα συνάρτηση ή διαδικασία έχει το δικαίωμα να δει, αλλά η πληροφορία που σχετίζεται με αυτήν την εγγραφή δεν βρίσκεται στο εγγράφημα δραστηριοποίησης της κληθείσας. Θα διαχωρίσουμε δύο περιπτώσεις, ανάλογα με το αν η συνάρτηση καλείται από συνάρτηση/διαδικασία αδελφό ή από συνάρτηση/διαδικασία γονέα.

Πρέπει να είμαστε προσεκτικοί και να μην συγχέουμε τα επίπεδα φωλιάσματος με το ποιος καλεί μία συνάρτηση ή διαδικασία. Μία συνάρτηση ή διαδικασία είναι πάντοτε εμφωλευμένη μέσα στον γονέα της, μπορεί όμως να κληθεί από τον γονέα της, από τον αδελφό της ή και από την ίδια τη συνάρτηση ή διαδικασία, αναδρομικά.

Στο παρακάτω παράδειγμα:

```
procedure parent()
{
    procedure son()
    {
        ...
        call son()
        ...
    }

    procedure daughter()
    {
        ...
        call son()
        ...
    }

    call son()
    call daughter()
    call parent()
}
```

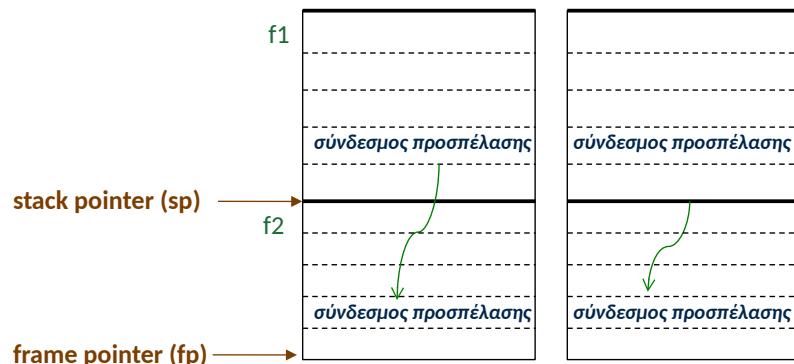
οι διαδικασίες `son()` και `daughter()` είναι εμφωλευμένες μέσα την `parent()`, αποτελούν, δηλαδή, παιδιά της. Οι διαδικασίες `son()` και `daughter()` έχουν σχέση αδελφών, αφού έχουν το ίδιο βάθος φωλιάσματος. Οι κλήσεις δεν ακολουθούν το ίδιο σχήμα. Η `son()` καλεί τον εαυτό της, αναδρομικά. Η `daughter()` καλεί την `son()`, και η `parent()` καλεί τις `son()` και `daughter()` ενώ στη συνέχεια τον αδελφό της αναδρομικά.

Έτσι διαχωρίζουμε τις εξής περιπτώσεις, οι οποίες εμφανίζονται στο σχήμα 10.6:

- Μία συνάρτηση ή διαδικασία γονέας καλεί τη συνάρτηση ή διαδικασία παιδί της. Ο σύνδεσμος προσπέλασης πρέπει να δείχνει στο εγγράφημα δραστηριοποίησης του γονέα, άρα στον `sp`, αφού τη στιγμή της δημιουργίας του εγγραφήματος δραστηριοποίησης της κληθείσας ο έλεγχος δεν έχει μεταβεί ακόμα στην κληθείσα και ο δείκτης στοίβας δείχνει ακόμα στην καλούσα. Μεταφέρουμε, λοιπόν, την τιμή του δείκτη στοίβας στη δεύτερη θέση του υπό δημιουργία εγγραφήματος δραστηριοποίησης:

```
sw sp, -4(fp)
```

- Μία συνάρτηση καλεί τη συνάρτηση αδελφό της ή τον εαυτό της, αναδρομικά. Ο σύνδεσμος προσπέλασης πρέπει και πάλι να δείχνει στο εγγράφημα δραστηριοποίησης του γονέα. Τη στιγμή της



**Σχήμα 10.6:** Αριστερά: στην κλήση συνάρτησης ή διαδικασίας από αδελφή συνάρτηση ή διαδικασία η καλούσα και η κληθείσα μοιράζονται τον ίδιο γονέα, άρα και ίδιο σύνδεσμο προσπέλασης. Δεξιά: στην κλήση συνάρτησης ή διαδικασίας από τον γονέα της, ο σύνδεσμος προσπέλασης της κληθείσας δείχνει το εγγράφημα δραστηριοποίησης της καλούσας.

δημιουργίας του εγγραφήματος δραστηριοποίησης της κληθείσας, ο έλεγχος δεν έχει μεταβεί ακόμα στην κληθείσα και ο δείκτης στοίβας δείχνει στην καλούσα. Σε κάθε περίπτωση, όμως, η καλούσα και η κληθείσα έχουν τον ίδιο γονέα. Μεταφέρουμε, λοιπόν, στη δεύτερη θέση του υπό δημιουργία εγγραφήματος δραστηριοποίησης, τη διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα, η οποία βρίσκεται αποθηκευμένη στη δεύτερη θέση του εγγραφήματος δραστηριοποίησης της καλούσας:

```
lw t0, -4(sp)
sw t0, -4(fp)
```

### 10.2.2 Δείκτης στοίβας

Το επόμενο βήμα είναι η μετακίνηση του δείκτη στοίβας. Ο δείκτης στοίβας σε κάθε στιγμή δείχνει το εγγράφημα δραστηριοποίησης της συνάρτησης ή της διαδικασίας η οποία εκείνη τη στιγμή εκτελείται. Έτσι, πριν τη μεταβίβαση του ελέγχου ροής, πρέπει να τοποθετηθεί ο δείκτης στοίβας στο εγγράφημα δραστηριοποίησης της κληθείσας.

Αφού το εγγράφημα δραστηριοποίησης της κληθείσας τοποθετείται στη στοίβα κάτω από αυτό της καλούσας, η απόσταση ανάμεσα στην παλαιότερη και στη νεότερη θέση του δείκτη στοίβας ισούται με το μήκος του εγγραφήματος δραστηριοποίησης της κληθείσας.

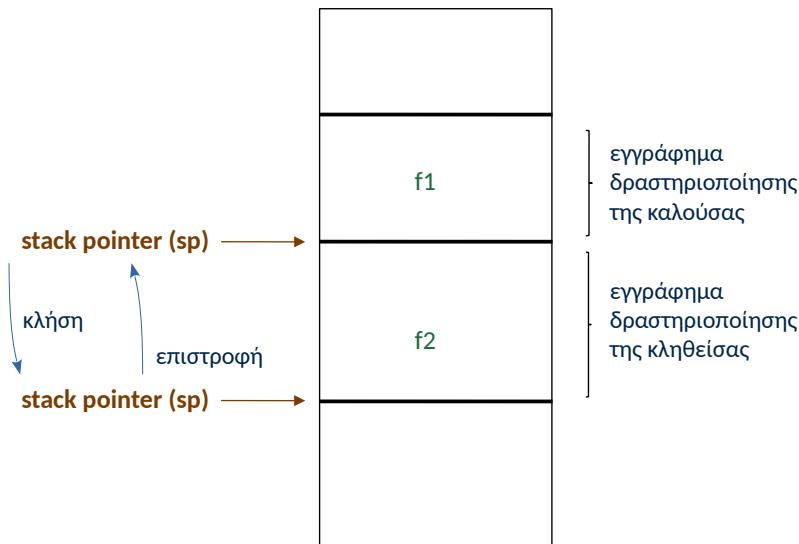
Το μήκος του εγγραφήματος δραστηριοποίησης είναι πληροφορία που είναι διαθέσιμη στον πίνακα συμβόλων. Έτσι, κατά την κλήση της συνάρτησης ή της διαδικασίας παράγεται η ακόλουθη εντολή τελικού κώδικα:

```
add sp, sp, frame_length
```

Μόλις ολοκληρωθεί η εκτέλεση της κληθείσας και ο έλεγχος ροής επιστραφεί στην καλούσα (θα μιλήσουμε στην επόμενη ενότητα για αυτό), ο δείκτης στοίβας πρέπει να επιστραφεί στην αρχική του θέση, να δείχνει, δηλαδή, στο εγγράφημα δραστηριοποίησης της καλούσας. Αυτό θα γίνει με την ίδια εντολή που χρησιμοποιήθηκε για τη μεταφορά του στο εγγράφημα δραστηριοποίησης της κληθείσας, μόνο που τώρα έχουμε αφαίρεση και όχι πρόσθεση του μήκους του εγγραφήματος δραστηριοποίησης στον δείκτη στοίβας:

```
add sp, sp, -frame_length
```

Σχηματικά μπορεί να δει κανείς αυτή την αλληλουχία στο σχήμα 10.7.



Σχήμα 10.7: Κατά την κλήση συνάρτησης, ο δείκτης στοίβας μεταφέρεται στο εγγράφημα δραστηριοποίησης της κληθείσας. Με τον τερματισμό τη κληθείσας μεταφέρεται πίσω στο εγγράφημα δραστηριοποίησης της καλούσας.

### 10.2.3 Μεταβίβαση ελέγχου

Στο τελευταίο βήμα της κλήσης μιας συνάρτησης ή διαδικασίας μεταφέρεται ο έλεγχος ροής στην κληθείσα. Στον επεξεργαστή RISC-V, καθώς και στους περισσότερους επεξεργαστές, υπάρχει μία εντολή που μας βοηθά σε αυτό.

Πρόκειται για την `jal` η οποία παίρνει ως όρισμα τη διεύθυνση μνήμης στην οποία βρίσκεται ο κώδικας της συνάρτησης ή διαδικασίας που θέλουμε να καλέσουμε, υπολογίζει και τοποθετεί στον καταχωρητή `ra` τη διεύθυνση της επόμενης εντολής από την `jal`, δηλαδή τη διεύθυνση της εντολής στην οποία πρέπει να επιστρέψει ο έλεγχος όταν η κληθείσα τερματίσει τη λειτουργία της. Να θυμίσουμε ότι εκεί τοποθετήσαμε στην προηγούμενη ενότητα την εντολή:

```
add sp, sp, -frame_length
```

Αν η τιμή του καταχωρητή `ra` δεν τροποποιηθεί κατά τη διάρκεια της εκτέλεσης της κληθείσας, τότε με την ολοκλήρωσή της θα χρησιμοποιήσουμε το περιεχόμενό του για να επιστρέψουμε στην καλούσα:

```
jr ra
```

Το περιεχόμενο του `ra` μπορεί να τροποποιηθεί στην περίπτωση που η συνάρτηση καλέσει άλλη συνάρτηση. Κατά την κλήση της δεύτερης συνάρτησης, η `jal` θα τοποθετήσει στον `ra` τη νέα διεύθυνση επιστροφής.

Για να αποφύγουμε την επικάλυψη της παλιάς διεύθυνσης από την καινούργια, θα πρέπει, πριν ξεκινήσει η εκτέλεση την κληθείσας, η διεύθυνση που έχει αποθηκεύσει η `jal` στον `ra` να μεταφερθεί στην πρώτη θέση του εγγραφήματος δραστηριοποίησης της κληθείσας, εκεί που έχουμε δεσμεύσει χώρο για τον σκοπό αυτόν:

```
sw ra, (sp)
```

Στο τέλος της κληθείσας πρέπει η διεύθυνση επιστροφής να επανακτηθεί από την πρώτη θέση του εγγραφήματος δραστηριοποίησης και να μεταφερθεί στον `ra`, πριν εκτελεστεί η εντολή για το άλμα της επιστροφής:

```
lw ra, (sp)
jr ra
```

### 10.3 Παράδειγμα παραγωγής τελικού κώδικα

Θα δούμε ένα απλό παράδειγμα παραγωγής τελικού κώδικα από ένα πρόγραμμα C-imperative, για να βεβαιωθούμε ότι έχουμε κατανοήσει τη διαδικασία παραγωγής του τελικού κώδικα. Ένα σύνθετο παράδειγμα, στο οποίο φαίνεται όλη η χρονική ακολουθία παραγωγής κώδικα σε κάθε φάση μεταγλώττισης μπορεί κανείς να αναζητήσει στο παράρτημα A.

```
program finalCodeExample
{
    declare A,B;

    procedure proc(in a, inout b)
    {   declare c;
        function func()
        {
            declare d;
            d:=4;
            A:=B;
            b:=a;
            return(c+d)
        }

        # body of proc #
        c:=3;
        print(A);
        print(B);
        B:=func();
        print(A);
        print(B)
    }

    # body of finalCodeExample #
    A:=1;
    B:=2;
    call proc(in A, inout B);
    print(A);
    print(B)
}.
```

Θα θεωρήσουμε εύκολο το βήμα της παραγωγής του ενδιάμεσου κώδικα, οπότε θα τον παραθέσουμε απλά:

```
# intermediate code for init
0: jump, _, _, main
# final code for init is generated here

# intermediate code for func
1: begin_block, func, _, _
2: :=, 4, _, d
3: :=, B, _, A
4: :=, a, _, b
5: +, c, d, T_1
6: return, T_1, _, _
```

```

7: end_block, func, _, _
# final code for func is generated here

# intermediate code for proc
8: begin_block, proc, _, _
9: :=, 3, _, c
10: print, A, _, _
11: print, B, _, _
12: par, T_2, ret, func
13: call, func, _, _
14: :=, T_2, _, B
15: print, A, _, _
16: print, B, _, _
17: end_block, proc, _, _
# final code for proc is generated here

# intermediate code for main
18: begin_block, main, _, _
19: :=, 1, _, A
20: :=, 2, _, B
21: par, A, in, proc
22: par, B, inout, proc
23: call, proc, _, _
24: print, A, _, _
25: print, B, _, _
26: halt, _, _, -
27: end_block, main, _, _
# final code for main is generated here

```

Ο τελικός κώδικας παράγεται μετά τη γραμμή 0 για το αρχικό τμήμα του προγράμματος, μετά τη γραμμή 7 για τη συνάρτηση func, μετά τη γραμμή 17 για τη διαδικασία proc και μετά τη γραμμή 27 για το κυρίως πρόγραμμα.

Αρχικά παράγεται τελικός κώδικας για τη γραμμή 0. Ορίζεται ένα μνημονικό όνομα για την αλλαγή γραμμής που θα χρησιμοποιηθεί από την εντολή print και στη συνέχεια μεταφράζεται το áλμα στην αρχή του κυρίως προγράμματος.

```

.data
str_nl: .asciz "\n"
.text

# 0: jump, _, _, main
L0:
    j Lmain

```

Το δεύτερο σημείο στο οποίο παράγεται τελικός κώδικας είναι αφού ολοκληρωθεί η μετάφραση του ενδιάμεσου κώδικα για τη func. Στο σημείο αυτό έχει ολοκληρωθεί η πληροφορία που πρέπει να εισαχθεί στον πίνακα συμβόλων για την func, όπως το start\_quad το οποίο συμπληρώθηκε με την παραγωγή της εντολής 1 του ενδιάμεσου κώδικα και είναι ίσο με 1, και το frame\_length, το οποίο έγινε γνωστό και συμπληρώθηκε μετά την παραγωγή της γραμμής 8 του ενδιάμεσου κώδικα, όταν είμαστε σίγουροι ότι δεν θα χρειαστούν περισσότερες προσωρινές μεταβλητές. Το frame\_length της func είναι ίσο με 20.

Στο σημείο αυτό της μετάφρασης ο πίνακας συμβόλων έχει την ακόλουθη μορφή:

```
-----
func :: d(12), T_1(16)
```

```
proc :: a(in,12), b(inout,16), c(20), func(sq:1,f1:20,arg:[])
finalCodeExample :: A(12), B(16), proc(arg:[in,inout])
-----
```

Οι εντολές του ενδιάμεσου κώδικα μεταφράζονται σε τελικό κώδικα μία προς μία. Ξεκινάμε με την εντολή με ετικέτα L1.

Στην αρχή της κλήσης μίας συνάρτησης ή διαδικασίας η τιμή του ra, ο οποίος έχει τη διεύθυνση επιστροφής και έχει τοποθετηθεί εκεί από την jal που εκτέλεσε η καλούσα, μεταφέρεται στην πρώτη θέση του εγγραφήματος δραστηριοποίησης της func, η οποία είναι δεσμευμένη για τον σκοπό αυτόν:

```
# 1: begin_block, func, _, _
L1:
    sw ra,(sp)
```

Στην ετικέτα L2, η σταθερά 4 μεταφέρεται στο καταχωρητή t1. Το d αναζητείται στον πίνακα συμβόλων, ο οποίος επιστρέφει ότι πρόκειται για τοπική μεταβλητή, 12 bytes πάνω από τον sp. Έτσι, η τιμή του t1 θα μεταφερθεί στο -12(sp):

```
# 2: :=, 4, _, d
L2:
    li t1,4
    sw t1,-12(sp)
```

Στην ετικέτα L3, ο πίνακας συμβόλων επιστρέφει για τις A και B ότι είναι καθολικές μεταβλητές στις θέσεις 12 και 16. Άρα η πρόσβαση σε αυτές θα γίνει με βάση τον gp. Αρχικά θα μεταφερθεί η τιμή της B από τη θέση -16(gp) στον t1 και από εκεί στη θέση -12(gp) στη μεταβλητή A:

```
# 3: :=, B, _, A
L3:
    lw t1,-16(gp)
    sw t1,-12(gp)
```

Στην ετικέτα L4, την οντότητα α ο πίνακας συμβόλων θα την εντοπίσει ένα επίπεδο επάνω από το τρέχον. Άρα πρόκειται για μία οντότητα που ανήκει στον γονέα της υπό μετάφρασης συνάρτησης. Μάλιστα, για την οντότητα αυτή, θα μας πληροφορήσει ότι στο επίπεδο που έχει δηλωθεί είναι τυπική παράμετρος που έχει περαστεί με τιμή και έχει offset 12.

Έτσι, ο κώδικας που θα παραχθεί θα ανακτήσει από το εγγράφημα δραστηριοποίησης της υπό μετάφρασης συνάρτησης τον σύνδεσμό προσπέλασης, ο οποίος είναι αποθηκευμένος στη θέση -4(sp) και θα τον τοποθετήσει στον καταχωρητή t0. Τώρα ο t0 δείχνει το εγγράφημα δραστηριοποίησης της γονικής διαδικασίας. Αφαιρώντας τον το offset, που στη συγκεκριμένη περίπτωση είναι 12, ο t0 θα δείχνει τη θέση που είναι αποθηκευμένο το a και, επειδή το a είναι μία παράμετρος με τιμή, στη θέση αυτή βρίσκεται αποθηκευμένη η τιμή της. Χρησιμοποιώντας τον καταχωρητή t0 ως δείκτη, διαβάζουμε την τιμή της μεταβλητής a και την καταχωρούμε στον t1.

Το πρώτο μέρος της εντολής στην ετικέτα 4, το οποίο αντιστοιχεί στις 3 πρώτες γραμμές του παραπάνω κώδικα και υλοποιεί τη μεταφορά της τιμής του a στον t1, έχει ολοκληρωθεί. Απομένει η μεταφορά του περιεχομένου του t1 στην b.

Για την b ο πίνακας συμβόλων επιστρέφει ότι πρόκειται για τυπική παράμετρο που έχει περαστεί με αναφορά και ανήκει ένα επίπεδο πάνω από την υπό μετάφραση συνάρτηση, δηλαδή πρόκειται για την ίδια γονική διαδικασία στην οποία βρέθηκε και το a. Ο κώδικας που θα παραχθεί θα ανακτήσει από το εγγράφημα δραστηριοποίησης της υπό μετάφραση συνάρτησης τον σύνδεσμο προσπέλασης, ο οποίος είναι αποθηκευμένος στη θέση -4(sp) και στη συνέχεια θα τον τοποθετήσει στον καταχωρητή t0. Ο t0 δείχνει το εγγράφημα δραστηριοποίησης της γονικής διαδικασίας, οπότε αν αφαιρέσουμε το offset, που στη συγκεκριμένη περίπτωση είναι 16, ο t0 θα δείχνει τη θέση που είναι αποθηκευμένο το b. Μόνο που αυτή τη φορά εκεί δεν θα βρίσκεται αποθηκευμένη η τιμή του b, όπως έγινε προηγουμένως με το a, αλλά η διεύθυνσή του,

αφού το b έχει περαστεί με αναφορά. Έτσι, χρειαζόμαστε ένα βήμα παραπάνω, το οποίο θα χρησιμοποιήσει τον καταχωρητή t0 ως δείκτη, προκειμένου στον t0 να μεταφερθεί η πραγματική διεύθυνση του b, εκεί που είναι αποθηκευμένη η τιμή του. Τώρα, με τον καταχωρητή t0 να έχει τη διεύθυνση στην οποία θέλουμε να γράψουμε, και τον καταχωρητή t1 να έχει την τιμή την οποία θέλουμε να γράψουμε εκεί, η αποθήκευση της τιμής στο b είναι ένα απλό sw.

```
# 4: :=, a, _, b
L4:
    lw t0,-4(sp)
    addi t0,t0,-12
    lw t1,(t0)
    lw t0,-4(sp)
    addi t0,t0,-16
    lw t0,(t0)
    sw t1,(t0)
```

Στην ετικέτα L5 προσθέτονται δύο μεταβλητές και τοποθετούνται σε μία προσωρινή μεταβλητή. Για την c ο πίνακας συμβόλων επιστρέφει ότι πρόκειται για μία τοπική μεταβλητή που έχει δηλωθεί στη γονική διαδικασία και έχει offset 20. Η d είναι τοπική μεταβλητή στην func με offset 12, ενώ η T\_1 προσωρινή μεταβλητή στην func με offset 16.

Για την c ακολουθούμε την ίδια διαδικασία που ακολουθήσαμε νωρίτερα για τις a και b, αφού και αυτή ανήκει στη γονική διαδικασία. Έτσι, ανακτούμε τον σύνδεσμο προσπέλασης, αφαιρούμε το offset και τοποθετούμε στον t1 την τιμή της c.

Η d είναι τοπική μεταβλητή στη θέση 12 bytes πάνω από τον sp, άρα μεταφέρουμε στον t2 το περιεχόμενο της θέσης -12(sp).

Στη συνέχεια με τη χρήση της add, προσθέτουμε τους καταχωρητές t1 και t2 και τοποθετούμε το αποτέλεσμα στον καταχωρητή t1.

Το τελευταίο βήμα είναι η μεταφορά του περιεχομένου του t1 στη θέση -16(sp), αφού το offset της προσωρινής μεταβλητής T\_1 είναι 16.

```
# 5: +, c, d, T_1
L5:
    lw t0,-4(sp)
    addi t0,t0,-20
    lw t1,(t0)
    lw t2,-12(sp)
    add t1,t2,t1
    sw t1,-16(sp)
```

Η τελευταία εντολή της func είναι η return, με την οποία επιστρέφεται η τιμή της συνάρτησης. Στην ετικέτα L6, η προσωρινή μεταβλητή T\_1, από τη θέση 16 που βρίσκεται στο εγγράφημα δραστηριοποίησης, μεταφέρεται στον καταχωρητή t1. Η διεύθυνση στην οποία θέλουμε να γραφεί το αποτέλεσμα έχει τοποθετηθεί κατά την κλήση της συνάρτησης στο -8(sp) και από εκεί μεταφέρεται στον t0. Δεν απομένει παρά, χρησιμοποιώντας τον t0 ως δείκτη, να επιστρέψουμε στην καλούσα την τιμή του t1.

```
# 6: return, T_1, _, _
L6:
    lw t1,-16(sp)
    lw t0,-8(sp)
    sw t1,(t0)
```

Τελευταίο βήμα είναι η επιστροφή του ελέγχου στην καλούσα, η οποία γίνεται στην ετικέτα L7. Από την πρώτη θέση του εγγραφήματος δραστηριοποίησης, από εκεί δηλαδή που στην ετικέτα L1 είχαμε αποθηκεύσει τη διεύθυνση επιστροφής, θα ανακτήσουμε τη διεύθυνση αυτή και θα την τοποθετήσουμε στον

καταχωρητή `ra`. Χρησιμοποιώντας τον `ra` και την εντολή `jr` θα εκτελέσουμε το áλμα της επιστροφής.

```
# 7: end_block, func, _, _
L7:
    lw ra,(sp)
    jr ra
```

Στη συνέχεια καλείται η `close_scope`, αφαιρείται το επίπεδο της `func` από τον πίνακα συμβόλων και εισάγεται πληροφορία στο επίπεδο για την `proc`. Ο πίνακας συμβόλων μετά την ολοκλήρωση της μετάφρασης του ενδιάμεσου κώδικα της `proc`, και πριν τη δημιουργία του τελικού, είναι ο ακόλουθος:

```
-----  
proc :: a(in,12), b(inout,16), c(20), func(sq:1,f1:20,arg:[]), T_2(24)  
finalCodeExample :: A(12), B(16), proc(arg:[in,inout])  
-----
```

Ξεκινάει η μετάφραση του τελικού κώδικα για την `proc`, με την ετικέτα `L8`. Το `frame_length` της `proc` είναι 24.

Η τιμή του `ra` που έχει τη διεύθυνση επιστροφής και έχει τοποθετηθεί εκεί από την εντολή `jal` της καλούσας, μεταφέρεται στην πρώτη θέση του εγγραφήματος δραστηριοποίησης.

```
# 8: begin_block, proc, _, _
L8:
    sw ra,(sp)
```

Στη συνέχεια, στην ετικέτα `L9`, γίνεται η εκχώρηση του 3 στην `c`. Η σταθερά 3 φορτώνεται στον καταχωρητή `t1` και από εκεί αποθηκεύεται στην 20η θέση του εγγραφήματος δραστηριοποίησης: `-20(sp)`, στην οποία σύμφωνα με τον πίνακα συμβόλων βρίσκεται το `c`.

```
# 9: :=, 3, _, c
L9:
    li t1,3
    sw t1,-20(sp)
```

Στην ετικέτα `L10` η εντολή `print` εμφανίζει το αποτέλεσμα της `A` στην οθόνη. Πρόκειται για καθολική μεταβλητή, άρα την αναζητούμε με βάση τον `gp`, σε απόσταση 12 bytes από αυτόν, σύμφωνα με τον πίνακα συμβόλων: `-12(gp)`. Από εκεί μεταφέρεται στον καταχωρητή `a0`, από όπου θα μεταφερθεί στην οθόνη. Για να γίνει αυτό, ο RISC-V ζητά να τοποθετηθεί στον `a7` η τιμή 1 και στη συνέχεια να κληθεί η `ecall`. Κάθε καταχωρητής έχει τον δικό του τρόπο με βάση τον οποίο το περιεχόμενο ενός καταχωρητή εμφανίζεται στην οθόνη, δεν διαφέρει óμως ως φιλοσοφία.

Στη συνέχεια πρέπει να ακολουθήσει μία αλλαγή γραμμής. Έχουμε ορίσει στην αρχή του προγράμματος ένα μνημονικό όνομα για την αλλαγή γραμμής: `str_nl`. Το τοποθετούμε στον `a0`, εκεί που προηγουμένως είχαμε τοποθετήσει τον ακέραιο, μόνο που τώρα στον `a7` τοποθετούμε την τιμή 4. Ολοκληρώνουμε την `print` καλώντας εκ νέου την `ecall`.

```
# 10: print, A, _, _
L10:
    lw a0,-12(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall
```

Όμοια διαδικασία ακολουθείται και στην `L11`, μόνο που το `B`, το οποίο είναι πάλι καθολική μεταβλητή, βρίσκεται τέσσερα bytes κάτω από το `A`: στη θέση `-16(gp)`.

```
# 11: print, B, _, _
L11:
    lw a0,-16(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall
```

Στην ετικέτα L12 ξεκινάει η κλήση της συνάρτησης func. Στο πρώτο βήμα γίνεται το πέρασμα παραμέτρων. Η func δεν έχει κάποια εμφανή παράμετρο. Όπως όλες οι συναρτήσεις, όμως, επιστρέφει αποτέλεσμα, και αυτό το κάνει μέσω της:

```
par, T_2, ret, func
```

Επειδή είναι η πρώτη παράμετρος της κλήσης, είναι υπεύθυνη για την τοποθέτηση του fp στην αρχή του εγγραφήματος δραστηριοποίησης της func. Σύμφωνα με τον πίνακα συμβόλων το μήκος του εγγραφήματος δραστηριοποίησης της func είναι 20, άρα ο fp τοποθετείται 20 θέσεις πάνω από τον sp.

Στη συνέχεια τοποθετούμε στον t0 τη διεύθυνση της προσωρινής μεταβλητής στην οποία θα επιστραφεί το αποτέλεσμα. Αφού η απόστασή της από την αρχή του εγγραφήματος δραστηριοποίησης είναι, σύμφωνα με τον πίνακα συμβόλων, ίση με 24, θα αφαιρέσουμε από τον sp το 24, ώστε να φτάσουμε στη θέση που βρίσκεται η T\_2.

Η διεύθυνση αυτή πρέπει, από τον t0, να μεταφερθεί στη θέση της στοίβας 8 bytes από την αρχή του εγγραφήματος δραστηριοποίησης της func, όπου έχει δεσμευτεί ο χώρος για τον σκοπό αυτόν. Η μετάφραση της par σε τελικό κώδικα ολοκληρώθηκε:

```
# 12: par, T_2, ret, func
L12:
    addi fp,sp,20
    addi t0,sp,-24
    sw t0,-8(fp)
```

Στη συνέχεια έχουμε την κλήση της func με την call, η οποία συμβαίνει στην L13. Μία call αποτελείται από τέσσερις ενέργειες.

Αρχικά συμπληρώνουμε τη δεύτερη θέση του εγγραφήματος δραστηριοποίησης με τον σύνδεσμο προσπέλασης. Εδώ έχουμε κλήση ενός παιδιού από έναν γονέα, άρα ως σύνδεσμος προσπέλασης πρέπει να σημειωθεί ο δείκτης στοίβας του γονέα. Έτσι, μεταφέρεται ο sp στη θέση -4(fp).

Μετά, μεταφέρουμε τον δείκτη στοίβας να δείχνει, αντί του εγγραφήματος δραστηριοποίησης της καλούσας, στην αρχή του εγγραφήματος δραστηριοποίησης της κληθείσας. Σύμφωνα με τον πίνακα συμβόλων το μήκος του εγγραφήματος δραστηριοποίησης είναι 20, άρα μετατοπίζουμε τον sp κατά 20 θέσεις.

Ακολουθεί η jal η οποία θα εκχωρήσει τον έλεγχο στην L1. L1 είναι η ετικέτα της πρώτης εκτελέσιμης εντολής της func, πληροφορία που βρίσκεται και αυτή αποθηκευμένη στον πίνακα συμβόλων.

Όταν τελειώσει η εκτέλεση της func, ο έλεγχος θα μεταβεί στην εντολή τελικού κώδικα που ακολουθεί την jal. Το πώς, δεν αφορά την call και έχει υλοποιηθεί νωρίτερα μέσα στην func. Αυτό που μας ενδιαφέρει εδώ είναι να τοποθετήσουμε μετά την jal μία εντολή η οποία θα επαναφέρει τον δείκτη στοίβας να δείχνει στην αρχή του εγγραφήματος δραστηριοποίησης της καλούσας. Είναι η ίδια εντολή που χρησιμοποιήσαμε για να μεταφέρουμε τον δείκτη στοίβας στο εγγράφημα της func, μόνο που αντί να προσθέτουμε bytes, τώρα αφαιρούμε.

Με την εντολή που επαναφέρει τον δείκτη στοίβας στην καλούσα, έχει ολοκληρωθεί η μετάφραση της jal σε τελικό κώδικα:

```
# 13: call, func, _, _
L13:
    sw sp,-4(fp)
```

```

addi sp,sp,20
jal L1
addi sp,sp,-20

```

Το αποτέλεσμα της func έχει επιστραφεί στην προσωρινή μεταβλητή T\_2. Σύμφωνα με τον αρχικό κώδικα, πρέπει τελικά να τοποθετηθεί στην καθολική μεταβλητή B και γίνεται στην εντολή ενδιάμεσου κώδικα με ετικέτα L14. Από τη θέση στοίβας -24(sp), στην οποία βρίσκεται τοποθετημένη η τιμή της T\_2, θα μεταφερθεί στον καταχωρητή t1 και από εκεί στη θέση -16(gp), όπου βρίσκεται η καθολική μεταβλητή B:

```

# 14: :=, T_2, _, B
L14:
    lw t1,-24(sp)
    sw t1,-16(gp)

```

Στην εντολή με ετικέτα L15 έχουμε μία print. Η διαδικασία που ακολουθείται στην L15 είναι όμοια με αυτήν στις L10 και L11. Τα A και B είναι πάλι καθολικές μεταβλητές, στις θέσεις -12(gp) και -16(gp). Για την L10 έχουμε:

```

# 15: print, A, _, _
L15:
    lw a0,-12(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall

```

Για την L11 έχουμε:

```

# 16: print, B, _, _
L16:
    lw a0,-16(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall

```

Τελευταίο βήμα είναι η επιστροφή του ελέγχου στην καλούσα, η οποία γίνεται στην ετικέτα L17. Όπως και στην ετικέτα L7, ανακτούμε από την πρώτη θέση του εγγραφήματος δραστηριοποίησης τη διεύθυνση επιστροφής και την τοποθετούμε στον καταχωρητή ra. Χρησιμοποιώντας τον ra και την jr εκτελούμε το άλμα της επιστροφής.

```

# 17: end_block, proc, _, _
L17:
    lw ra,(sp)
    jr ra

```

Στο σημείο αυτό έχει ολοκληρωθεί η μετάφραση της proc. Έτσι, καλείται πάλι η close\_scope και αφαιρείται το επίπεδο της proc από τον πίνακα συμβόλων ενώ εισάγεται πληροφορία στον πίνακα συμβόλων για τις προσωρινές μεταβλητές του κυρίως προγράμματος. Στο συγκεκριμένο παράδειγμα δεν υπάρχει ανάγκη από προσωρινές μεταβλητές. Ο πίνακας συμβόλων, μετά την ολοκλήρωση της μετάφρασης του τελικού κώδικα της proc και πριν ξεκινήσει η μετάφραση του τελικού κώδικα του κυρίως προγράμματος, αποτελείται από ένα μόνο επίπεδο το οποίο φαίνεται παρακάτω. Παρατηρήστε ότι η func δεν εμφανίζεται εδώ, και αυτό είναι λογικό, αφού το κυρίως πρόγραμμα δεν έχει δικαίωμα να την καλέσει.

-----

```
finalCodeExample :: A(12), B(16), proc(arg:[in,inout])
-----
```

Στο τέλος μετατρέπεται ο ενδιάμεσος κώδικας του κυρίως προγράμματος σε τελικό. Πρόκειται για τις εντολές με ετικέτες από L18 έως και L27. Μπορεί ο κώδικας αυτός να μεταφράζεται τελευταίος, εκτελείται όμως πρώτος. Για αυτό φροντίζει ο κώδικας στην ετικέτα L0 που παρήχθη νωρίτερα. Πριν μεταφράσουμε την ετικέτα L18, ορίζουμε την ετικέτα Lmain για να μπορέσει να γίνει το απαραίτητο αρχικό άλμα.

Στην ετικέτα L18 έχουμε την αρχή του κώδικα του κυρίως προγράμματος. Μετά από τη δήλωση της ετικέτας Lmain, ορίζονται οι θέσεις των sp και gp. Ο δείκτης sp δείχνει στην αρχή του χώρου που μας δόθηκε για στοίβα από το λειτουργικό σύστημα. Πρέπει να μεταφερθεί κατά τόσες θέσεις παρακάτω, όσες και το μήκος του εγγραφήματος δραστηριοποίησης του κυρίως προγράμματος. Η πληροφορία αυτή δεν είναι αποθηκευμένη στον πίνακα συμβόλων, αλλά μπορεί να εξαχθεί από τη θέση που έχει τοποθετηθεί η τελευταία μεταβλητή του κυρίως προγράμματος. Στον πίνακα συμβόλων που εμφανίζεται παραπάνω, η καθολική μεταβλητή B βρίσκεται στη θέση 16, καταλαμβάνει χώρο 4 bytes, άρα το μήκος του εγγραφήματος δραστηριοποίησης του κυρίως προγράμματος είναι 20. Προσθέτουμε λοιπόν το 20 στον δείκτη στοίβας για να σημειώσει την αρχή του εγγραφήματος δραστηριοποίησης του κυρίως προγράμματος.

Στο ίδιο σημείο θέλουμε να δείξει και ο gp. Ο δείκτης αυτός θέλουμε να δείχνει τη θέση του εγγραφήματος δραστηριοποίησης που περιέχονται οι καθολικές μεταβλητές, δηλαδή το σημείο στο οποίο αυτή τη στιγμή δείχνει ο sp. Τοποθετούμε τον gp να δείχνει όπου και ο sp. Η τιμή του gp δεν θα μεταβληθεί καθ' όλη τη διάρκεια της εκτέλεσης.

```
L18:
Lmain:
# 18: begin_block, main, _, _
    addi sp,sp,20
    mv gp,sp
```

Στην ετικέτα L19 τοποθετούμε το 1 στην καθολική μεταβλητή A. Στο πρώτο βήμα το 1 τοποθετείται στον καταχωρητή t1 και από εκεί μεταφέρεται στη θέση -12(gp), όπου, σύμφωνα με τον πίνακα συμβόλων, βρίσκεται το A:

```
# 19: :=, 1, _, A
L19:
    li t1,1
    sw t1,-12(gp)
```

Η ίδια διαδικασία ακολουθείται και με την εκχώρηση του 2 στο B. Το B βρίσκεται στη θέση -16(gp)

```
# 20: :=, 2, _, B
L20:
    li t1,2
    sw t1,-16(gp)
```

Στην ετικέτα L21 ξεκινάει η κλήση της proc, με το πέρασμα της πρώτης παραμέτρου, η οποία είναι η A. Αρχικά τοποθετούμε τον δείκτη fp, 28 θέσεις κάτω από τον sp, όσο είναι και το μήκος του εγγραφήματος δραστηριοποίησης της proc. Η A είναι καθολική μεταβλητή, άρα η τιμή της βρίσκεται στο -12(gp). Από εκεί μεταφέρεται στον καταχωρητή t0 και στη συνέχεια στο νέο εγγράφημα δραστηριοποίησης, στην πρώτη θέση των παραμέτρων, δηλαδή στο -12(fp). Η μεταφορά της τιμής της παραμέτρου στο περιβάλλον της proc ολοκληρώθηκε:

```
# 21: par, A, in, proc
L21:
    addi fp,sp,28
    lw t0,-12(gp)
    sw t0,-12(fp)
```

Στη συνέχεια, στην ετικέτα L22 περνάει η δεύτερη παράμετρος. Πρόκειται πάλι για καθολική μεταβλητή, όμως τώρα δεν περνάει με τιμή, αλλά με αναφορά. Άρα στο εγγράφημα δραστηριοποίησης της proc, στη δεύτερη θέση των παραμέτρων (στην -16(fp)) πρέπει να τοποθετηθεί η διεύθυνση της B. Σχηματίζουμε τη διεύθυνση της B αφαιρώντας από τον sp (ή τον gp, αυτή τη στιγμή δείχνουν στο ίδιο σημείο) το offset της B, δηλαδή το 16. Μέσω του καταχωρητή t0, η διεύθυνση της καθολικής μεταβλητής τοποθετείται στη δεύτερη θέση του εγγραφήματος δραστηριοποίησης της proc, 16 bytes από την αρχή του:

```
# 22: par, B, inout, proc
L22:
    addi t0,sp,-16
    sw t0,-16(fp)
```

Ακολουθεί η κλήση της proc με την call, στην ετικέτα L23. Αρχικά συμπληρώνουμε τη δεύτερη θέση του εγγραφήματος δραστηριοποίησης με τον σύνδεσμο προσπέλασης. Εδώ έχουμε κλήση ενός παιδιού από έναν γονέα, άρα ως σύνδεσμος προσπέλασης πρέπει να σημειωθεί ο δείκτης στοίβας του γονέα. Έτσι μεταφέρεται ο sp στη θέση -4(fp).

Μετά, μεταφέρουμε τον δείκτη στοίβας να δείχνει, αντί του εγγραφήματος δραστηριοποίησης της καλούσας, στην αρχή του εγγραφήματος δραστηριοποίησης της κληθείσας. Σύμφωνα με τον πίνακα συμβόλων το μήκος του εγγραφήματος δραστηριοποίησης είναι 28, άρα μεταφέρουμε τον sp κατά 28 θέσεις.

Ακολουθεί η jal η οποία θα μεταφέρει τον έλεγχο στην L8, στην ετικέτα της πρώτης εκτελέσιμης εντολής της proc.

Όταν τελειώσει η εκτέλεση της proc, ο έλεγχος θα μεταβεί στην εντολή τελικού κώδικα που ακολουθεί την jal. Εκεί βρίσκεται η εντολή που θα επαναφέρει τον δείκτη στοίβας στην αρχή του εγγραφήματος δραστηριοποίησης του κυρίως προγράμματος:

```
# 23: call, proc, _, _
L23:
    sw sp,-4(fp)
    addi sp,sp,28
    jal L8
    addi sp,sp,-28
```

Στη συνέχεια έχουμε δύο κλήσεις της print για τις μεταβλητές A και B. Πρόκειται για καθολικές μεταβλητές, άρα ο κώδικας που παράγεται δεν διαφέρει από αυτόν των εντολών στις ετικέτες L15 και L16.

Για την L24 έχουμε:

```
# 24: print, A, _, _
L24:
    lw a0,-12(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall
```

Και για την L25:

```
# 25: print, B, _, _
L25:
    lw a0,-16(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall
```

Φτάσαμε στο τέλος, όπου συναντάμε την εντολή `halt`. Αυτή παράγει την κλήση ενός `ecall` που τερματίζει το πρόγραμμα.

```
# 26: halt, _, _, _
L26:
    li a0,0
    li a7,93
    ecall
```

Επειδή βρισκόμαστε στο κυρίως πρόγραμμα η `begin_block` δεν παράγει κώδικα, αφού η `halt`, αμέσως πριν παρήγαγε τον κώδικα που απαιτούνταν.

```
# 27: end_block, main, _, _
L27:
```

Ο κώδικας που παρήχθη ακολουθεί ολοκληρωμένος:

```
.data
str_nl: .asciz "\n"
.text

L0:
# 0: jump, _, _, main
j Lmain

L1:
# 1: begin_block, func, _, _
sw ra,(sp)

L2:
# 2: :=, 4, _, d
li t1,4
sw t1,-12(sp)

L3:
# 3: :=, B, _, A
lw t1,-16(gp)
sw t1,-12(gp)

L4:
# 4: :=, a, _, b
lw t0,-4(sp)
addi t0,t0,-12
lw t1,(t0)
lw t0,-4(sp)
addi t0,t0,-16
lw t0,(t0)
sw t1,(t0)

L5:
# 5: +, c, d, T_1
lw t0,-4(sp)
addi t0,t0,-20
lw t1,(t0)
lw t2,-12(sp)
```

```
add t1,t2,t1
sw t1,-16(sp)
```

L6:

```
# 6: return, T_1, _, _
lw t1,-16(sp)
lw t0,-8(sp)
sw t1,(t0)
```

L7:

```
# 7: end_block, func, _, _
lw ra,(sp)
jr ra
```

L8:

```
# 8: begin_block, proc, _, _
sw ra,(sp)
```

L9:

```
# 9: :=, 3, _, C
li t1,3
sw t1,-20(sp)
```

L10:

```
# 10: print, A, _, _
lw a0,-12(gp)
li a7,1
ecall
la a0,str_nl
li a7,4
ecall
```

L11:

```
# 11: print, B, _, _
lw a0,-16(gp)
li a7,1
ecall
la a0,str_nl
li a7,4
ecall
```

L12:

```
# 12: par, T_2, ret, func
addi fp,sp,20
addi t0,sp,-24
sw t0,-8(fp)
```

L13:

```
# 13: call, func, _, _
sw sp,-4(fp)
addi sp,sp,20
```

```

jal L1
addi sp,sp,-20

L14:
# 14: :=, T_2, _, B
lw t1,-24(sp)
sw t1,-16(gp)

L15:
# 15: print, A, _, _
lw a0,-12(gp)
li a7,1
ecall
la a0,str_nl
li a7,4
ecall

L16:
# 16: print, B, _, _
lw a0,-16(gp)
li a7,1
ecall
la a0,str_nl
li a7,4
ecall

L17:
# 17: end_block, proc, _, _
lw ra,(sp)
jr ra

L18:
Lmain:
# 18: begin_block, main, _, _
addi sp,sp,20
mv gp,sp

L19:
# 19: :=, 1, _, A
li t1,1
sw t1,-12(gp)

L20:
# 20: :=, 2, _, B
li t1,2
sw t1,-16(gp)

L21:
# 21: par, A, in, proc
addi fp,sp,28

```

```
lw t0,-12(gp)
sw t0,-12(fp)
```

L22:

```
# 22: par, B, inout, proc
    addi t0,sp,-16
    sw t0,-16(fp)
```

L23:

```
# 23: call, proc, _, _
    sw sp,-4(fp)
    addi sp,sp,28
    jal L8
    addi sp,sp,-28
```

L24:

```
# 24: print, A, _, _
    lw a0,-12(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall
```

L25:

```
# 25: print, B, _, _
    lw a0,-16(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall
```

L26:

```
# 26: halt, _, _, _
    li a0,0
    li a7,93
    ecall
```

L27:

```
# 27: end_block, main, _, _
```

Περισσότερα για τη γλώσσα μηχανής του RISC-V μπορείτε να βρείτε στην αναφορά [1], ενώ οι αναγνώστες που ενδιαφέρονται περισσότερο μπορούν να βρουν χρήσιμη πληροφορία και στην αναφορά [2]. Για πρόσθετη βιβλιογραφία στην παραγωγή τελικού κώδικα μπορείτε να ανατρέξετε και στα: [3, 4, κεφ.6-7], [5, 6, κεφ.8], [7, κεφ.9]

## Βιβλιογραφία

- [1] Andrew Waterman και Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Αδημοσίευτη ερευνητική εργασία. EECS Department, University of California, Berkeley, 2019.
- [2] Andrew Waterman, Krste Asanović και John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Αδημοσίευτη ερευνητική εργασία. EECS Department, University of California, Berkeley, 2021.
- [3] Keith D. Cooper και Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2004. ISBN: 1558606998.
- [4] Keith D. Cooper και Linda Torczon. *Σχεδίαση και Κατασκευή Μεταγλωττιστών*. Ξενόγλωσσος τίτλος: Engineering a Compiler, Επιστημονική επιμέλεια έκδοσης: Γεώργιος Μανής, Νικόλαος Παπασπύρου και Αντώνιος Σαββίδης. Πανεπιστημιακές Εκδόσεις Κρήτης, 2018. ISBN: 9789605245191.
- [5] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 0321486811.
- [6] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Μεταγλωττιστές: Αρχές, τεχνικές και εργαλεία, 2η αμερικανική έκδοση*. Επιστημονική επιμέλεια έκδοσης: Π. Αλεφραγκής, Α. Βώρος, Ν. Βώρος, Κ. Μασσέλος. Εκδόσεις Νέων Τεχνολογιών, 2011. ISBN: 9789606759727.
- [7] Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.



## ΚΕΦΑΛΑΙΟ 11

---

### ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ ΚΩΔΙΚΑ

---

#### Σύνοψη:

Η βελτιστοποίηση του κώδικα είναι πολύ σημαντικό τμήμα της διαδικασίας της μεταγλωττισης. Ο κώδικας, ο οποίος παράγεται τόσο στη φάση της παραγωγής ενδιάμεσου, όσο και στη φάση της παραγωγής τελικού κώδικα, έχει πολλά περιθώρια βελτίωσης, όσον αφορά την ταχύτητα του τελικού εκτελέσιμου και το μέγεθός του. Θα ασχοληθούμε με δυνατότητες που μας δίνονται για βελτιστοποίηση τόσο στον ενδιάμεσο, όσο και στον τελικό κώδικα.

Στο κεφάλαιο αυτό θα παρουσιαστούν, αρχικά, μετασχηματισμοί που ανήκουν στην κατηγορία των τεχνικών αποτίμησης, όπως η διάδοση σταθερών, ο υπολογισμός σταθερών εκφράσεων, η διάδοση αντιγράφων, η διάδοση υποκατάστατων, οι αλγεβρικές απλοποιήσεις και ο υποβιβασμός ισχύος. Στη συνέχεια θα παρουσιαστούν μετασχηματισμοί που ανήκουν στην κατηγορία της απαλοιφής κώδικα, όπως η απαλοιφή μη προσβάσιμου κώδικα, η απαλοιφή μη χρήσιμου κώδικα, η απαλοιφή μη χρήσιμων μεταβλητών και η απαλοιφή κοινών υποεκφράσεων.

Σε μία άλλη κατηγορία μετασχηματισμών, στους βελτιστοποιητικούς μετασχηματισμούς στις κλήσεις συναρτήσεων και διαδικασιών, εντάσσονται η βελτιστοποίηση κλήσης σε διαδικασία φύλλο, η εκχώρηση καταχωρητών κατά τις κλήσεις, η ενσωμάτωση διαδικασίας, η απαλοιφή αναδρομής ουράς και η απομνημόνευση κλήσεων. Τέλος, στην πολύ σημαντική κατηγορία των μετασχηματισμών βρόχων θα εξετάσουμε τον υποβιβασμό ισχύος βασιζόμενο με επαναλήψεις βρόχου, την απαλοιφή επαγωγικών μεταβλητών, τη μετακίνηση κώδικα ανεξάρτητου από την εκτέλεση του βρόχου, την εναλλαγή σε βρόχο, την κυκλική συρρίκνωση, τον διαχωρισμό και την ένωση βρόχων.

### Προαπαιτούμενη γνώση:

- στοιχεία γλώσσας μηχανής
  - κεφάλαιο 6
  - κεφάλαιο 7
  - κεφάλαιο 9
  - κεφάλαιο 10
- 

Κατά τη βελτιστοποίηση του κώδικα, μπορούμε να επιτύχουμε τόσο τη μείωση του αριθμού των εντολών ενός προγράμματος, όσο και του χώρου που απαιτείται για την αποθήκευση των μεταβλητών στη στοίβα και γενικότερα των δεδομένων. Το πιο σημαντικό, όμως, κέρδος από τους βελτιστοποιητικούς μετασχηματισμούς είναι η μείωση του χρόνου που απαιτείται για την εκτέλεση του προγράμματος, μείωση η οποία μπορεί να είναι εντυπωσιακά αξιόλογη. Θα δούμε αναλυτικά τους πιο σημαντικούς από τους βελτιστοποιητικούς μετασχηματισμούς που χρησιμοποιούνται στην τεχνολογία των μεταγλωττιστών.

Στη βιβλιογραφία μπορεί ο αναγνώστης να αναζητήσει και να βρει ιδιαίτερα σημαντικό υλικό για περαιτέρω ανάγνωση και εμβάθυνση στο θέμα, εάν το επιθυμήσει. Βιβλιογραφικές αναφορές στους επί μέρους μετασχηματισμούς θα δοθούν κατά την περιγραφή τους. Κεφάλαια σε βιβλία, σχετικά με τη βελτιστοποίηση κώδικα μπορεί κανείς να αναζητήσει στα [1, 2, κεφ.8-11], [3, 4, κεφ.8.5-11], [5, κεφ.8]. Μία πολύ ενδιαφέρουσα εργασία που περιέχει συγκεντρωμένους όλους τους μετασχηματισμούς που περιγράφονται παρακάτω, αλλά δίνει και μία αντιπροσωπευτική εικόνα της περιοχής είναι η [6].

## 11.1 Τεχνικές αποτίμησης

Στην κατηγορία αυτή θα εντάξουμε βελτιστοποιήσεις που σχετίζονται με την αποτίμηση ή τη διαδικασία της αποτίμησης εκφράσεων. Θα εξετάσουμε τη διάδοση σταθερών, τον υπολογισμό σταθερών εκφράσεων, τη διάδοση αντιγράφων, τη διάδοση υποκατάστατων, τις αλγεβρικές απλοποιήσεις και τον υποβιβασμό ισχύος.

### 11.1.1 Διάδοση σταθερών

Η διάδοση σταθερών (*constant propagation*) [7, 8] είναι μια απλή αλλά αποτελεσματική τεχνική. Η πρακτική των προγραμματιστών να γράφουν ευανάγνωστο κώδικα είναι κάτι απαραίτητο όταν αυτοί λειτουργούν σε ομάδα, αλλά εξίσου χρήσιμο και για την περίπτωση που αναπτύσσουν μικρές εφαρμογές στις οποίες δεν συνεργάζονται με άλλους. Δεν θα αναλύσουμε τη στιγμή αυτή τα οφέλη, αλλά θα πάρουμε ως δεδομένο ότι ο ακόλουθος κώδικας:

```
dim_x = 100;
dim_y = 100;
for (x=0; x<dim_x; x++)
    for (y=0; j<dim_y; y++)
        ...

```

είναι περισσότερο περιγραφικός από έναν κώδικα ο οποίος δεν χρησιμοποιεί τα `dim_x`, `dim_y`. Σε αντίθεση, για τον μεταγλωττιστή ο καταλληλότερος κώδικας είναι ο ακόλουθος:

```
for (x=0; x<100; x++)
    for (y=0; j<100; y++)
        ...

```

Η τεχνική της βελτιστοποίησης της διάδοσης σταθερών μετατρέπει τον πρώτο κώδικα στον δεύτερο, εξυπηρετώντας τις ανάγκες και των προγραμματιστών και του μεταγλωττιστή.

### 11.1.2 Υπολογισμοί σταθερών εκφράσεων

Ο αγγλικός όρος του υπολογισμού σταθερών εκφράσεων είναι *constant folding*. Η τεχνική αυτή αποτελεί μία ιδιαίτερα χρήσιμη τεχνική που μεταφέρει υπολογισμούς από τον χρόνο εκτέλεσης στον χρόνο μετάφρασης. Οι υπολογισμοί στον χρόνο μετάφρασης γίνονται μία φορά, και στον χρόνο εκτέλεσης απλά χρησιμοποιείται το αποτέλεσμά τους [9, 10].

Σε συνδυασμό με τη διάδοση σταθερών, ο παρακάτω κώδικας μπορεί να απλοποιηθεί σημαντικά όσον αφορά την παράμετρο μέσα στο `malloc()`.

```
ptr = (int *)malloc(dim_x*dim_y*sizeof(int));
```

Η διάδοση σταθερών θα αντικαταστήσει τις τιμές των `dim_x`, `dim_y`, ενώ σε συνδυασμό με το γεγονός ότι το μέγεθος ενός ακεραίου είναι 4 bytes, η τέλεση των πράξεων μέσα στην παρένθεση μπορεί να μας οδηγήσει στο πολύ προτιμότερο για τον μεταγλωττιστή:

```
ptr = (int *)malloc(40000);
```

το οποίο εξαλείφει την ανάγκη της εκτέλεσης των πολλαπλασιασμών σε χρόνο εκτέλεσης.

### 11.1.3 Διάδοση αντιγράφων

Τεχνικές αυτόματης παραγωγής κώδικα ή τεχνικές βελτιστοποίησης μπορούν να οδηγήσουν στην ύπαρξη αντιγράφων μέσα στον κώδικα, τα οποία δεν έχουν κάποια ιδιαίτερη χρησιμότητα. Η τεχνική της διάδοσης αντιγράφων (*copy propagation*) εξαλείφει τα περιττά αντίγραφα [3, 4].

Ο κώδικας:

```
x = y;
printf("%d", x)
```

μπορεί απλά να αντικατασταθεί με τον κώδικα:

```
printf("%d", y)
```

### 11.1.4 Διάδοση υποκατάστατων

Η διάδοση υποκατάστατων [6] είναι μια γενίκευση της διάδοσης αντιγράφων, κατά την οποία έχουμε αντικατάσταση μιας ολόκληρης έκφρασης και όχι μιας μεταβλητής. Ο αγγλικός όρος είναι *forward substitution*. Έτσι, παραφράζοντας το προηγούμενο παράδειγμα, ο κώδικας:

```
x = y*y+1;
printf("%d", x)
```

μπορεί απλά να αντικατασταθεί με τον κώδικα:

```
printf("%d", y*y+1)
```

### 11.1.5 Αλγεβρικές απλοποιήσεις

Οι αλγεβρικές απλοποιήσεις (*algebraic simplification*) [6] εφαρμόζονται σε ακέραιους αριθμούς, αφού λόγω του τρόπου συμβολισμού και της ακρίβειας των πραγματικών αριθμών, το αποτέλεσμα μπορεί να είναι σε κάποιες περιπτώσεις ελαφρώς διαφορετικό από το επιθυμητό. Οι πιο συνηθισμένες αλγεβρικές απλοποιήσεις, οι οποίες είναι και ασφαλείς για ακέραιους αριθμούς, φαίνονται παρακάτω:

$$\begin{aligned} a * 0 &= 0 \\ 0 / a &= 0 \\ a + 0 &= a \end{aligned}$$

$$a / 1 = a$$

### 11.1.6 Υποβιβασμός ισχύος

Με τον όρο *υποβιβασμός ισχύος* (*strength reduction*) αναφερόμαστε σε αντικαταστάσεις περισσότερων ακριβών εκφράσεων με ισοδύναμες εκφράσεις που υπολογίζονται γρηγορότερα [11].

Παραδείγματα ακολουθούν:

$$\begin{aligned} x * 2 &\rightarrow x + x \\ x^2 &\rightarrow x * x \\ a * 2^x &\rightarrow a \ll c \text{ (ολίσθηση κατά 2)} \end{aligned}$$

## 11.2 Απαλοιφή κώδικα

Στην απαλοιφή κώδικα αφαιρούμε από τον κώδικα τμήματα τα οποία δεν είναι απαραίτητα σε αυτόν ή τα αντικαθιστούμε με άλλα υπολογιστικά λιγότερο ακριβά. Θα εξετάσουμε την απαλοιφή μη προσβάσιμου κώδικα, την απαλοιφή μη χρήσιμου κώδικα, την απαλοιφή μη χρήσιμων μεταβλητών και την απαλοιφή κοινών υποεκφράσεων.

### 11.2.1 Απαλοιφή μη προσβάσιμου κώδικα

Ως περιπτώσεις απαλοιφής μη προσβάσιμου κώδικα (*unreachable code*) σε ένα πρόγραμμα δομημένης γλώσσας προγραμματισμού μπορούμε να αναφέρουμε την αποτίμηση μιας λογικής έκφραστης, η οποία έχει πάντα την ίδια τιμή (*true* ή *false*). Στην περίπτωση που έχουμε έναν βρόχο του οποίου η συνθήκη γνωρίζουμε εκ των προτέρων ότι δεν θα αποτιμηθεί ποτέ ως αληθής, μπορούμε να απαλείψουμε ολόκληρο τον βρόχο [3, 12].

Ένα παράδειγμα αποτίμησης μιας έκφραστης που θα μπορούσε να αποτιμηθεί σε χρόνο μετάφραστης ως αληθής ή ως ψευδής είναι το ακόλουθο, στο οποίο χρησιμοποιείται η δημοφιλής τεχνική να ορίζουμε μια μεταβλητή με την οποία να διαχωρίζουμε αν βρισκόμαστε σε φάση αποσφαλμάτωσης ή όχι:

```
int debug;
int x;

debug = 0; // no debugging now
x = 1
if (debug)
{
    printf("%d", x)
}
```

Όλο το παραπάνω *if* θα ενταχθεί στην κατηγορία του μη προσβάσιμου κώδικα και θα απαλειφθεί.

### 11.2.2 Απαλοιφή μη χρήσιμου κώδικα

Μη χρήσιμος κώδικας (*useless code*) [3] είναι ο κώδικας ο οποίος δεν έχει λόγο να βρίσκεται μέσα στο πρόγραμμα. Μη χρήσιμος κώδικας μπορεί να προκύψει κυρίως από άλλους μετασχηματισμούς. Ένα παράδειγμα μη χρήσιμου κώδικα είναι ο κώδικας που δημιουργήθηκε από τον μετασχηματισμό που εφαρμόσαμε στην απαλοιφή μη προσβάσιμου κώδικα.

Έχοντας απαλείψει το:

```

if (debug)
{
    printf("%d", x)
}

```

ο κώδικας που απέμεινε:

```

debug = 0;
x = 1

```

δεν έχει πια κάποια χρησιμότητα και μπορεί να απαλειφθεί ως μη χρήσιμος

### 11.2.3 Απαλοιφή μη χρήσιμων μεταβλητών

Κατά την απαλοιφή μη χρήσιμων μεταβλητών (*dead variable elimination*) [3, 4] απαλείφουμε μεταβλητές οι οποίες έπαψαν να είναι χρήσιμες, κυρίως λόγω κάποιων μετασχηματισμών που προηγήθηκαν. Στο παράδειγμα που χρησιμοποιήσαμε στους δύο προηγούμενους μετασχηματισμούς και ύστερα από την εφαρμογή τους, οι μεταβλητές:

```

int debug;
int x;

```

ανήκουν στην κατηγορία των μη χρήσιμων μεταβλητών (*dead variables*), δεν έχουν πια κάποια χρησιμότητα και μπορούν να απαλειφθούν.

### 11.2.4 Απαλοιφή κοινών υποεκφράσεων

Η απαλοιφή κοινών υποεκφράσεων (*common subexpression elimination*) [3, 4] είναι μία τεχνική που στοχεύει στην επαναχρησιμοποίηση υπολογισμών που έχουν ήδη γίνει νωρίτερα στον κώδικα, ώστε να αποφύγει τον εκ νέου υπολογισμό των εκφράσεων αυτών. Θα αποκαλέσουμε κοινές υποεκφράσεις τα τμήματα στα οποία εμφανίζονται ομοιότητες που μπορούμε να εκμεταλλευτούμε για τον σκοπό αυτόν.

Τέτοιες υποεκφράσεις μπορεί να σχηματιστούν κατά την παραγωγή του ενδιάμεσου κώδικα. Αυτό, θα γίνει περισσότερο κατανοητό με ένα παράδειγμα. Ας θεωρήσουμε τον κώδικα:

```

a[i] = a[i] + a[i] * a[i]
b[k] = a[i] + 1

```

Δεν έχουμε μιλήσει καθόλου για τον τρόπο που διαχειρίζόμαστε πίνακες κατά την παραγωγή ενδιάμεσου ή και τελικού κώδικα. Μπορεί, όμως, κανείς να φανταστεί ότι και στις πέντε φορές που εμφανίζεται στον παραπάνω κώδικα το `a[i]` θα παραχθεί κώδικας ο οποίος αρχικά θα τοποθετήσει έναν δείκτη στην αρχή του πίνακα `a` και στη συνέχεια θα μετακινήσει τον δείκτη κατά `i*sizeof(int)` θέσεις, αν θεωρήσουμε ότι έχουμε πίνακα ακεραίων. Στο τέλος αυτής της διαδικασίας η διεύθυνση του `a[i]` θα βρίσκεται αποθηκευμένη σε κάποιον καταχωρητή. Αντί να επανα-υπολογίσουμε πέντε φορές τη διεύθυνση του `a[i]`, μπορούμε να διατηρήσουμε και να επαναχρησιμοποιήσουμε την ήδη υπολογισμένη διεύθυνση.

## 11.3 Βελτιστοποιητικοί μετασχηματισμοί στις κλήσεις διαδικασιών και συναρτήσεων

Οι μετασχηματισμοί αυτοί στοχεύουν στο να μειώσουν επιβαρύνσεις στον χρόνο εκτέλεσης που οφείλονται στην κλήση διαδικασιών και συναρτήσεων. Η μείωση στην επιβάρυνση μπορεί να επιτευχθεί είτε με την απαλοιφή της κλήσης, είτε με την παράλειψη κάποιων από τους μηχανισμούς της κλήσης, εκμεταλλευόμενοι ιδιαιτερότητες που εμφανίζονται στον κώδικα.

Θα δούμε τη βελτιστοποίηση κλήσης σε διαδικασία φύλλο, την εκχώρηση καταχωρητών κατά την κλήση, την ενσωμάτωση διαδικασίας και την απαλοιφή της αναδρομής ουράς.

### 11.3.1 Βελτιστοποίηση κλήσης σε διαδικασία φύλλο

Μία διαδικασία φύλλο (*leaf procedure*) είναι μία διαδικασία η οποία δεν καλεί κάποια άλλη. Μία διαδικασία φύλλο δεν χρειάζεται να σώσει στη στοίβα τη διεύθυνση επιστροφής, αλλά μπορεί να την κρατήσει στον καταχωρητή ρα ή σε οποιονδήποτε άλλον καταχωρητή χρησιμοποιεί το εκάστοτε υλικό για τον σκοπό αυτόν [6].

Στο ακόλουθο παράδειγμα, η διεύθυνση επιστροφής της `f()` δεν είναι απαραίτητο να αντιγραφεί στο εγγράφημα δραστηριοποίησης της `f()` στη στοίβα.

```
int f(int x)
{
    return x*x;
}

int main()
{
    printf("%d\n", f(4));
}
```

### 11.3.2 Εκχώρηση καταχωρητών κατά τις κλήσεις

Όταν η μεταγλωττιση λάβει υπόψη τις ανάγκες σε καταχωρητές και της καλούσας και της κληθείσας συνάρτησης ή διαδικασίας, τότε μπορεί να γίνει μία καλύτερη διαχείριση των διαθέσιμων καταχωρητών.

Καλούμε αυτήν την πρακτική εκχώρηση καταχωρητών κατά την κλήση (*cross call register allocation*) [6]. Αν η κληθείσα συνάρτηση ή διαδικασία δεν χρειάζεται να χρησιμοποιήσει όλους τους διαθέσιμους από το υλικό καταχωρητές ή αν γίνει ο κατάλληλος σχεδιασμός ώστε να μην χρειαστεί να τους χρησιμοποιήσει, τότε η καλούσα είναι σε θέση να θεωρήσει ότι κάποιοι καταχωρητές δεν μεταβλήθηκαν κατά την κλήση. Έτσι, μπορεί να αποφύγει μετακινήσεις τιμών από τους καταχωρητές στη μνήμη και αντίστροφα, οι οποίοι κανονικά θα έπρεπε να γίνουν αμέσως πριν και αμέσως μετά την κλήση..

### 11.3.3 Ενσωμάτωση διαδικασίας

Κατά την ενσωμάτωση διαδικασίας (*procedure inlining ή procedure integration*) [12, 13] η κλήση μιας διαδικασίας αντικαθίσταται με τον σώμα αυτής, εξαλείφοντας το πρόσθετο κόστος που απαιτείται για την κλήση. Ενώ ο προγραμματιστής μπορεί να προτιμήσει να φτιάξει μία συνάρτηση που να υπολογίζει το τετράγωνο ενός αριθμού:

```
int square(int x)
{
    return x*x;
}

int x = 2;
int y = square(x);
```

ο μεταγλωττιστής μπορεί με την τεχνική της ενσωμάτωσης διαδικασίας να θεωρήσει τον πιο απλό, μικρό και γρήγορο κώδικα:

```
int x = 2;
int y = x * x;
```

Παρατηρήστε ότι, στο παράδειγμά μας, με τον μετασχηματισμό αυτόν, δημιουργήθηκαν και άλλες περιπτώσεις βελτιστοποίησης, αφού ο υπολογισμός του `y` μπορεί πια να γίνει σε χρόνο μετάφρασης.

```
int y = 4;
```

#### 11.3.4 Απαλοιφή αναδρομής ουράς

Η απαλοιφή της αναδρομής ουράς (*tail recursion elimination*) μπορεί να εφαρμοστεί σε μία αναδρομική κλήση όταν η αναδρομική κλήση είναι η τελευταία ενέργεια που γίνεται στην αναδρομική συνάρτηση.

Στην παρακάτω συνάρτηση παραγοντικού η αναδρομική κλήση γίνεται στο τέλος της αναδρομής κλήσης:

```
int factorial(int x)
{
    if (x==0) return 1;
    else return x*factorial(x-1);
}
```

Όμως, μετά την ολοκλήρωση της αναδρομικής κλήσης απαιτείται ένας πολλαπλασιασμός με το x. Άρα στην κλήση αυτή δεν μπορεί να εφαρμοστεί απαλοιφή αναδρομής ουράς.

Στην παρακάτω συνάρτηση, όμως, η αναδρομική κλήση δεν ακολουθείται από κάτι άλλο:

```
char search(int * A, int N, int i, int x)
{
    if (i == N)
        return -1;
    else if (A[i] == x)
        return 1;
    else
        return search(A,N,i+1,x);
}
```

Όταν σε μία αναδρομική συνάρτηση η αναδρομική κλήση γίνεται στο τέλος της συνάρτησης, τότε με την ολοκλήρωση της εκτέλεσης το εγγράφημα δραστηριοποίησης δεν χρειάζεται πια. Αντί να δημιουργηθεί ένα νέο εγγράφημα δραστηριοποίησης και να τοποθετηθεί κάτω από το υπάρχον, μπορεί το παλιό εγγράφημα δραστηριοποίησης να μετατραπεί και να χρησιμοποιηθεί για τις ανάγκες της νέας συνάρτησης.

Ο παραπάνω κώδικας είναι ισοδύναμος με τον κώδικα που ακολουθεί, ο οποίος δεν είναι αναδρομικός.

```
char search(int * A, int N, int i, int x)
{
    label: start
    if (i == N)
        return -1;
    else if (A[i] == x)
        return 1;
    else
    {
        i++;
        goto start;
    }
}
```

Η απαλοιφή της αναδρομής ουράς μπορεί να γίνει αυτόματα σε ένα πρόγραμμα [14].

#### 11.4 Απομνημόνευση κλήσεων

Η απομνημόνευση κλήσεων (*function memoization*) [15] εφαρμόζεται σε συναρτήσεις και διαδικασίες οι οποίες δεν έχουν παρενέργειες. Με τον όρο *παρενέργειες* (*side effects*) εννοούμε τροποποιήσεις τιμών σε χώρο έξω από τη συνάρτηση/διαδικασία, πέρα από αυτές που προκαλούνται με δομημένο τρόπο, δηλαδή

μέσα από την επιστροφή τιμής της συνάρτησης και μέσα από τις παραμέτρους της.

Όταν μια διαδικασία ή συνάρτηση έχει αυτή την ιδιότητα, τότε κάθε φορά που καλείται μπορούμε να αποθηκεύουμε τα αποτελέσματα από τις κλήσεις της, ώστε, αν κληθεί πάλι με τις ίδιες παραμέτρους στην είσοδο, να μην είναι απαραίτητη η εκτέλεσή της.

## 11.5 Μετασχηματισμοί βρόχων

Οι βελτιστοποιητικοί μετασχηματισμοί που σχετίζονται με βρόχους είναι ιδιαίτερα σημαντικοί, αφού ένα μέσο πρόγραμμα θα δαπανήσει πολύ χρόνο μέσα σε βρόχους. Αν αναλογιστεί κανείς ότι οι βρόχοι μπορεί να απαιτούν χιλιάδες επαναλήψεις για να εκτελεστούν και ότι οι φωλιασμένοι βρόχοι είναι κάτι αρκετά σύνηθες σε ένα πρόγραμμα, τότε είναι εύκολο να συλλάβει την έκταση του οφέλους που υπάρχει αν μειωθεί ο χρόνος της εκτέλεσης, έστω και μίας εντολής που βρίσκεται μέσα στο σώμα ενός βρόχου.

Τόσο η ερευνητική κοινότητα, όσο και οι εταιρείες ανάπτυξης μεταγλωττιστών έχουν επενδύσει πολύ στο σημείο αυτό και τα αποτελέσματα είναι εντυπωσιακά. Θα δούμε παρακάτω τις σημαντικότερες τεχνικές που χρησιμοποιούνται και, πιο συγκεκριμένα, θα δούμε τον βασιζόμενο σε βρόχο υποβιβασμό ισχύος, την απαλοιφή επαγωγικών μεταβλητών, τη μετακίνηση κώδικα, την αντικατάσταση βρόχων, την εναλλαγή στους βρόχους, την αντιστροφή βρόχου, την ανίχνευση λωρίδων, τη συρρίκνωση κύκλων, τον τεμαχισμό βρόχων, την ένωση και τον διαχωρισμό βρόχων. Οι μετασχηματισμοί, φυσικά, δεν τελειώνουν εδώ. Υπάρχουν ακόμα πολλοί, πολύ ενδιαφέροντες.

### 11.5.1 Υποβιβασμός ισχύος βασιζόμενος σε βρόχο

Η ιδέα του υποβιβασμού ισχύος έχει ήδη συζητηθεί. Στην υποενότητα αυτή θα σημειώσουμε περιπτώσεις υποβιβασμού ισχύος που δημιουργούνται μέσα σε έναν βρόχο (*loop based strength reduction*) [16].

Μία περίπτωση ενός τέτοιου υποβιβασμού ισχύος μέσα σε έναν βρόχο εμφανίζεται όταν ένας πολλαπλασιασμός εξαρτάται από τη μεταβλητή του βρόχου *i*:

```
for i in range(N):
    ... = c * i
```

Η ιδέα είναι να αντικαταστήσουμε τον “ακριβό” πολλαπλασιασμό με μία γρηγορότερη πράξη, για παράδειγμα, μία πρόσθεση. Ο παραπάνω κώδικας είναι ισοδύναμος με τον ακόλουθο:

```
t = c
for i in range(N):
    ... = t
    t += c
```

Αφού σε κάθε επανάληψη το *c* πολλαπλασιάζεται με το *i*, τότε μπορούμε στην τιμή *c\*(i-1)* που υπολογίσαμε στην προηγούμενη επανάληψη, να προσθέσουμε το *c*. Οι υπολογισμοί που απαιτούνται στη δεύτερη περίπτωση είναι πολύ λιγότερο κοστούμοι για τον υπολογιστή.

Άλλες πιθανές περιπτώσεις υποβιβασμού ισχύος σε έναν βρόχο φαίνονται παρακάτω. Η ύψωση σε δύναμη μπορεί να αντικατασταθεί με πολλαπλασιασμό. Ο κώδικας:

```
for i in range(N):
    ... = c ^ i
```

γίνεται:

```
t = c
for i in range(N):
    ... = t
    t = t * c
```

Η ύψωση σε δύναμη του  $-1$  μπορεί να εκμεταλλευτεί την αλλαγή προσήμου. Ο κώδικας:

```
for i in range(N):
    ... = (-1) ^ i
```

γίνεται:

```
t = c
for i in range(N):
    ... = t
    t = -t
```

### 11.5.2 Απαλοιφή επαγωγικών μεταβλητών

Μία επαγωγική μεταβλητή (*induction variable*) είναι μία μεταβλητή της οποίας η τιμή εξαρτάται από την επανάληψη του βρόχου. Κατά την απαλοιφή επαγωγικών μεταβλητών (*induction variable elimination*) [3, 4], και βασιζόμενοι στην ιδέα της μείωσης ισχύος, ένας μεταγλωττιστής μπορεί να εξαλείψει τη μεταβλητή του βρόχου. Στον κώδικα:

```
for i in range(x):
    sum += A[i]
```

θεωρούμε έναν πίνακα ακεραίων, με μέγεθος ακεραίου ίσο με 4 bytes. Ο ενδιάμεσος κώδικας για τον παραπάνω βρόχο ακολουθεί. Με &Α συμβολίζουμε τη διεύθυνση του Α, οπότε στη συνέχεια χρησιμοποιούμε το  $T_2$  ως δείκτη ( $^T_2$ ).

```
100: :=, 0, _, i
110: <, i, x, 130
120: jump, _, _, 190
130: *, 4, i, T_1
140: +, &A, T_1, T_2
150: :=, ^T_2, _, T_3
160: +, sum, T_3, sum
170: +, i, 1, i
180: jump, _, _, 110
190: ...
```

Στη γραμμή 130 υπάρχει ένας πολλαπλασιασμός τον οποίο μπορούμε να μετατρέψουμε σε πρόσθεση, βασιζόμενοι στην τεχνική του υποβιβασμού ισχύος. Παράλληλα πρέπει να αρχικοποιήσουμε τη μεταβλητή  $T_1$  και για τον σκοπό αυτόν προσθέτουμε τη γραμμή 101:

```
100: :=, 0, _, i
101: :=, i, _, T_1
110: <, i, x, 130
120: jump, _, _, 190
130: +, 4, T_1, T_1
140: +, &A, T_1, T_2
150: :=, ^T_2, _, T_3
160: +, sum, T_3, sum
170: +, i, 1, i
180: jump, _, _, 110
190: ...
```

Τώρα έχουμε έναν αρκετά οικονομικότερο κώδικα σε χρόνο εκτέλεσης. Μπορούμε, όμως, να προχωρήσουμε σε περισσότερες βελτιστοποιήσεις και να εξαλείψουμε τελείως τη μεταβλητή  $i$ :

- Το  $i$  μπορεί να εξαλειφθεί από τη γραμμή 100, ενοποιώντας τις γραμμές 100 και 101:

```
100: :=, 0, _, T_1
```

- Στη γραμμή 110 χρησιμοποιείται το `i` για την έξοδο από τον βρόχο. Μπορούμε πάλι να χρησιμοποιήσουμε την `T_1` και να αναπροσαρμόσουμε το όριο για την έξοδο από το βρόχο σε  $x^*4$ . Αυτό πρέπει να γίνει με καινούργια μεταβλητή σε κάποια νέα τετράδα, έξω από τον βρόχο. Επιλέγουμε την ετικέτα 102 και τροποποιούμε κατάλληλα την 110:

```
102: *, 4, x, T_4
110: <, T_1, T_4, 130
```

- Η γραμμή 170 δεν χρειάζεται πια, οπότε αφαιρείται.

Ο μετασχηματισμένος κώδικας ακολουθεί:

```
100: :=, 0, _, T_1
101: :=, i, _, T_1
102: *, 4, x, T_4
110: <, T_1, T_4, 130
120: jump, _, _, 190
130: +, 4, T_1, T_1
140: +, &A, T_1, T_2
150: :=, ^T_2, _, T_3
160: +, sum, T_3, sum
180: jump, _, _, 110
190: ...
```

### 11.5.3 Μετακίνηση κώδικα ανεξάρτητου της εκτέλεσης του βρόχου

Κατά την τεχνική της μετακίνησης κώδικα ανεξάρτητου από την εκτέλεση του βρόχου (*loop invariant code motion*) [3, 4], κώδικας μπορεί να μεταφερθεί εκτός βρόχου, στην περίπτωση που δεν εξαρτάται από τις επαναλήψεις του βρόχου. Παρακάτω υπάρχει ένα παράδειγμα στο οποίο η πολύ ακριβή λειτουργία του υπολογισμού της τετραγωνικής ρίζας είναι εφικτό να μετακινηθεί πριν την εκτέλεση του βρόχου:

```
for i in range(N):
    a[i] = sqrt(x)
```

Με μια πρώτη ματιά, το `sqrt(x)` μπορεί να βγει έξω από τον βρόχο και να υπολογιστεί μία φορά αντί για  $N$ .

```
t = sqrt(x)
for i in range(N):
    a[i] = t
```

Αν γίνουμε όμως λίγο πιο προσεκτικοί, θα προσέξουμε ότι στο αρχικό πρόγραμμα ο υπολογισμός της τετραγωνικής ρίζας θα γίνει μόνο αν ο έλεγχος του προγράμματος εισέλθει μέσα στον βρόχο. Αν η εκτέλεση δεν εισέλθει μέσα στον βρόχο, η μεταβλητή `t` όχι μόνο υπολογίζεται άσκοπα, αλλά ενέχει κινδύνους να δημιουργήσει σφάλμα στην εκτέλεση. Ποιος μας διαβεβαιώνει ότι ο προγραμματιστής δεν έχει εξασφαλίσει νωρίτερα στον κώδικά του ότι το `x` δεν θα έχει αρνητική τιμή αν η εκτέλεση εισέλθει μέσα στον βρόχο;

Έτσι, ο σωστός μετασχηματισμός είναι ο ακόλουθος, ο οποίος μας εγγυάται ότι η τετραγωνική ρίζα θα υπολογιστεί, μόνο αν η εκτέλεση του προγράμματος τελικά εισέλθει μέσα στον βρόχο.

```
if N>0:
    t = sqrt(x)
for i in range(N):
    a[i] = t
```

#### 11.5.4 Εναλλαγή σε βρόχο

Η εναλλαγή σε βρόχο (*loop interchange*) [17, 18] επιτρέπει την εναλλαγή της θέσης δύο βρόχων οι οποίοι αποτελούν έναν τέλεια φωλιασμένο βρόχο. Ένας τέλεια φωλιασμένος βρόχος (*perfectly nested loop*) είναι μία δομή από βρόχους στην οποία ο ένας βρόχος βρίσκεται μέσα στον άλλο και στην οποία δομή όλες οι εκχωρήσεις τελούνται μέσα στον εσωτερικότερο βρόχο.

Ο παρακάτω διπλός βρόχος:

```
for i in range(n):
    for j in range(m):
        a[i,j] += b[i,j]
```

ο οποίος είναι τέλεια φωλιασμένος, μπορεί με τον μετασχηματισμό της εναλλαγής βρόχου να γίνει:

```
for j in range(m):
    for i in range(n):
        a[i,j] += b[i,j]
```

Ο μετασχηματισμός μπορεί να δημιουργήσει ευκαιρίες εφαρμογής άλλων βελτιστοποιητικών μηχανισμών, να οδηγήσει σε πιο ομοιόμορφη προσπέλαση στη μνήμη, να δημιουργήσει βρόχο με βήμα 1 ή και να βοηθήσει στον αυτόματο παραλληλισμό του βρόχου. Αν και ο αυτόματος παραλληλισμός είναι ένα αντικείμενο που δεν απασχολεί το βιβλίο αυτό, να σημειώσουμε ότι οι τέλεια φωλιασμένοι βρόχοι μπορούν πολλές φορές να επιτύχουν σημαντική επιτάχυνση αν εκτελεστούν σε παράλληλες μηχανές και η αυτόματη εξαγωγή παραλληλισμού είναι πολλές φορές εφικτή.

#### 11.5.5 Κυκλική συρρίκνωση

Κατά την κυκλική συρρίκνωση (*circle shrinking*) [19] ένας βρόχος διασπάται σε  $k$  βρόχους, οι οποίοι είναι ανεξάρτητοι μεταξύ τους. Το  $k$  είναι η εξάρτηση που έχει η επανάληψη του βρόχου. Με άλλα λόγια, μία επανάληψη, η οποία γράφει σε μία θέση  $i$  του πίνακα, διαβάζει το στοιχείο του πίνακα που γράφηκε  $k$  επαναλήψεις πριν, στη θέση  $i - k$ .

Για παράδειγμα, ο βρόχος:

```
for x in range(2, 10):
    a[x] = a[x-2]
```

στον οποίο  $k = 2$ , αφού κάθε στοιχείο  $a[i]$  του πίνακα  $a$  εξαρτάται από το στοιχείο  $a[i-2]$ , μπορεί να γίνει:

```
for x in range(2, 10, 2):
    a[x] = a[x-2]
for x in range(3, 10, 2):
    a[x] = a[x-2]
```

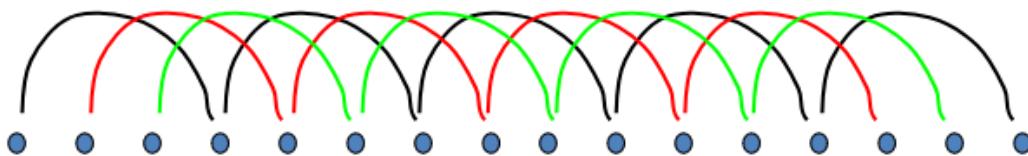
Στη γενικότερη περίπτωση ο βρόχος:

```
for x in range(k, N):
    a[x] = a[x-k]
```

μετασχηματίζεται σε  $k$  βρόχους:

```
for x in range(k, N, k):
    a[x] = a[x-2]
for x in range(k+1, N, k):
    a[x] = a[x-2]
...
for x in range(2*k-1, N, k):
    a[x] = a[x-2]
```

Σχηματικά ο μετασχηματισμός εικονίζεται στο σχήμα 11.1, όπου  $k = 3$ .



Σχήμα 11.1: Κυκλική συρρίκνωση για  $k = 3$ . Κάθε χρώμα δείχνει και έναν διαφορετικό ανεξάρτητο βρόχο.

### 11.5.6 Διαχωρισμός και ένωση βρόχων

Ο διαχωρισμός βρόχων (*loop distribution*) [20] και η ένωση βρόχων (*loop fusion*) [21] είναι δύο μετασχηματισμοί που είναι αντίθετοι μεταξύ τους. Ο ένας διαχωρίζει έναν βρόχο σε δύο ή περισσότερους βρόχους, ενώ ο άλλος ενώνει βρόχους σε έναν.

Έστω ότι έχουμε τον αρχικό βρόχο:

```
for i in range(N):
    a[i] += i
    b[i] = 6 * b[i] + 1
```

Ο παραπάνω βρόχος μπορεί μέσω του μετασχηματισμού του διαχωρισμού βρόχου να γίνει:

```
for i in range(N):
    a[i] += i
    for i in range(N):
        b[i] = 6 * b[i] + 1
```

ενώ η εφαρμογή του μετασχηματισμού της ένωσης βρόχων στους παραπάνω διαχωρισμένους βρόχους θα έδινε πάλι τον αρχικό βρόχο.

Ο διαχωρισμός βρόχων μπορεί να δημιουργήσει τέλεια φωλιασμένους βρόχους ή μικρότερους βρόχους με μικρότερες εξαρτήσεις ανάμεσα στις επαναλήψεις, και να επιτύχει πιο ομοιόμορφη πρόσβαση στη μνήμη, αφού οι επαναλήψεις θα γίνονται σε μικρότερο αριθμό πινάκων σε κάθε μικρότερο βρόχο.

### Βιβλιογραφία

- [1] Keith D. Cooper και Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2004. ISBN: 1558606998.
- [2] Keith D. Cooper και Linda Torczon. *Σχεδίαση και Κατασκευή Μεταγλωττιστών*. Ξενόγλωσσος τίτλος: *Engineering a Compiler*, Επιστημονική επιμέλεια έκδοσης: Γεώργιος Μανής, Νικόλαος Παπαστύρου και Αντώνιος Σαββίδης. Πανεπιστημιακές Εκδόσεις Κρήτης, 2018. ISBN: 9789605245191.
- [3] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 0321486811.
- [4] Alfred V. Aho, Monica S. Lam, Ravi Sethi και Jeffrey D. Ullman. *Μεταγλωττιστές: Αρχές, τεχνικές και εργαλεία, 2η αμερικανική έκδοση*. Επιστημονική επιμέλεια έκδοσης: Π. Αλεφραγκής, Α. Βώρος, Ν. Βώρος, Κ. Μασσέλος. Εκδόσεις Νέων Τεχνολογιών, 2011. ISBN: 9789606759727.
- [5] Νικόλαος Παπαστύρου και Εμμανουήλ Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002. ISBN: 9789602661352.

- [6] David F. Bacon, Susan L. Graham και Oliver J. Sharp. "Compiler Transformations for High-Performance Computing." Στο: *ACM Computing Surveys* 26.4 (1994), σσ. 345–420.
- [7] Gary A. Kildall. "A Unified Approach to Global Program Optimization." Στο: *POPL*. Επιμέλεια υπό Patrick C. Fischer και Jeffrey D. Ullman. ACM Press, 1973, σσ. 194–206.
- [8] Mark N. Wegman και F. Kenneth Zadeck. "Constant Propagation with Conditional Branches." Στο: *ACM Trans. Program. Lang. Syst.* 13 (1991), σσ. 181–210.
- [9] William D. Clinger. "How to Read Floating-Point Numbers Accurately." Στο: *PLDI*. Επιμέλεια υπό Bernard N. Fischer. ACM, 1990, σσ. 92–101. ISBN: 0-89791-364-7.
- [10] Guy L. Steele Jr. και Jon L. White. "How to Print Floating-Point Numbers Accurately." Στο: *PLDI*. Επιμέλεια υπό Bernard N. Fischer. ACM, 1990, σσ. 112–126. ISBN: 0-89791-364-7.
- [11] Keith D. Cooper, L. Taylor Simpson και Christopher A. Vick. "Operator strength reduction." Στο: *ACM Trans. Program. Lang. Syst.* 23.5 (2001), σσ. 603–625.
- [12] E. Allen F. και J. Cocke. "A Catalogue of Optimizing Transformations". Στο: *Design and Optimization on Compilers*. Επιμέλεια υπό R. Rustin. Prentice Hall, 1971, σσ. 3–41.
- [13] Robert Scheifler. "An Analysis of Inline Substitution for a Structured Programming Language." Στο: *Commun. ACM* 20.9 (1977), σσ. 647–654.
- [14] R. M. Burstall και John Darlington. "A Transformation System for Developing Recursive Programs". Στο: *Journal of the ACM* 24.1 (1977), σσ. 44–67.
- [15] Harold Abelson, Gerald J. Sussman και Julie Sussman. *Structure and Interpretation of Computer Programs*. Second. MIT Press, 2010. ISBN: 0262011530.
- [16] E. Allen F. "Program Optimization". Στο: *Anunual Review in Automatic Programming*. Τόμ. 18. Pergmont Press, 1969, σσ. 239–307.
- [17] John R. Allen και Ken Kennedy. "Automatic loop interchange." Στο: *SIGPLAN Symposium on Compiler Construction*. Επιμέλεια υπό Mary S. Van Deusen και Susan L. Graham. ACM, 1984, σσ. 233–246. ISBN: 0-89791-139-3.
- [18] Michael Wolfe. "Optimizing supercompilers for supercomputers." Στο: *ICS*. Επιμέλεια υπό Eduard Ayguadé, Wen mei W. Hwu, Rosa M. Badia και H. Peter Hofstee. ACM, 2020, 32:1. ISBN: 978-1-4503-7983-0.
- [19] Constantine D. Polychronopoulos. "Advanced Loop Optimizations for Parallel Computers." Στο: *ICS*. Επιμέλεια υπό Elias N. Houstis, Theodore S. Papatheodorou και Constantine D. Polychronopoulos. Lecture Notes in Computer Science. Springer, 1987, σσ. 255–277. ISBN: 3-540-18991-2.
- [20] David J. Kuck. "A Survey of Parallel Machine Organization and Programming." Στο: *ACM Comput. Surv.* 9.1 (1977), σσ. 29–59.
- [21] A. P. Yershov. "ALPHA - An Automatic Programming System of High Efficiency." Στο: *J. ACM* 13.1 (1966), σσ. 17–24.



## ΚΕΦΑΛΑΙΟ 12

---

# ΕΡΓΑΛΕΙΑ ΑΥΤΟΜΑΤΟΠΟΙΗΜΕΝΗΣ ΑΝΑΠΤΥΞΗΣ ΜΕΤΑΓΛΩΤΤΙΣΤΩΝ

---

### Σύνοψη:

Η αυτοματοποιημένη ανάπτυξη τμημάτων ενός μεταγλωττιστή είναι μία συνηθισμένη προσέγγιση και έχουν κατασκευαστεί μετα-εργαλεία κατάλληλα για τον σκοπό αυτόν. Τα εργαλεία αυτόματης ανάπτυξης επικεντρώνονται στη λεκτική και τη συντακτική ανάλυση, αφού σε αυτές τις φάσεις έχουμε αρκετά αυτοματοποιημένες διαδικασίες. Η είσοδος σε αυτά περιγράφεται με κανονικές εκφράσεις για τον λεκτικό αναλυτή και με μία γραμματική χωρίς συμφραζόμενα για τον συντακτικό αναλυτή.

Θα ασχοληθούμε με δύο τέτοια εργαλεία. Στη βιβλιογραφία υπάρχει εκτενές υλικό που τα περιγράφει αναλυτικά και ο αναγνώστης μπορεί να απευθυνθεί εκεί για μία πλήρη περιγραφή τους. Στο κεφάλαιο αυτό θα γίνει μία σύντομη παρουσίαση, μέσα από παραδείγματα για κάθε περίπτωση. Ο αναγνώστης θα διδαχθεί τις βασικές δυνατότητες των εργαλείων αυτών, ώστε να γνωρίζει πότε πρέπει να τα επιλέξει, αλλά και εύκολα να εμπλουτίσει τις γνώσεις του, αν το θελήσει ή το χρειαστεί.

Η πρώτη ομάδα εργαλείων αποτελείται από το *ζευγάρι lex και yacc/bison*. Είναι η κλασικότερη προσέγγιση στην αυτοματοποιημένη κατασκευή λεκτικών και συντακτικών αναλυτών, η οποία απλοποιεί πολύ το έργο του προγραμματιστή. Ένα παράδειγμα λεκτικής και συντακτικής ανάλυσης μιας απλής γλώσσας προγραμματισμού με ορισμούς και εκχωρήσεις μεταβλητών θα μας φανεί πολύ χρήσιμο στην κατανόηση.

Το δεύτερο μετα-εργαλείο είναι το ANTLR, το οποίο αποτελεί μια εξελιγμένη και πιο σύγχρονη μορφή ενός τέτοιου εργαλείου. Το ίδιο παράδειγμα, εμπλουτισμένο, θα παρουσιαστεί για το ANTLR, ώστε να μπορεί να γίνει σύγκριση ανάμεσα στα εργαλεία.

Ως συνέχεια του κεφαλαίου αυτού, θα παρουσιαστεί στα παραρτήματα του βιβλίου ένα παράδειγμα ανάπτυξης ενός μορφωτή προγραμμάτων αναπτυγμένου με το ANTLR.

### Προαπαιτούμενη γνώση:

- κεφάλαια 1 έως 10

---

Η ανάπτυξη των εργαλείων αυτόματης παραγωγής μεταγλωττιστών αποτελεί ένα σημαντικό βήμα που διευκολύνει την ανάπτυξη γλωσσών προγραμματισμού και άλλων εφαρμογών που απαιτούν λεκτική ή συντακτική ανάλυση. Τα εργαλεία αυτά παίρνουν ως είσοδο μία περιγραφή των λεκτικών μονάδων της γλώσσας σε μορφή κανονικών εκφράσεων, μία γραμματική χωρίς συμφραζόμενα που περιγράφει τη σύνταξη και ένα σύνολο σημασιολογικών κανόνων οι οποίοι προσθέτουν την απαραίτητη λειτουργικότητα. Η φιλοσοφία αυτή είναι κοινή σε όλα τα εργαλεία που θα εξετάσουμε, διαφέρει μόνο ο τρόπος υλοποίησής τους και η ευκολία με την οποία μπορούμε να το πετύχουμε.

Θα εξετάσουμε τα εργαλεία lex-yacc (ή flex-bison αν προτιμάτε) και το ANTLR.

#### 12.1 Ανάπτυξη μεταγλωττιστών με τα εργαλεία lex και yacc/bison

Τα lex και yacc αποτελούν εργαλεία αυτοματοποιημένης ανάπτυξης λεκτικών και συντακτικών αναλυτών, αντίστοιχα. Μπορούν να χαρακτηριστούν ως μετα-μεταγλωττιστές, αφού παίρνουν ως είσοδο μία περιγραφή ενός τμήματος ενός μεταγλωττιστή και παράγουν κώδικα που υλοποιεί το τμήμα αυτό του μεταγλωττιστή.

Είναι τα πρώτα, χρονικά, πολύ δημοφιλή εργαλεία, για τον σκοπό αυτόν. Οι νεότερες εκδόσεις τους είναι τα flex και bison, αν και οι ονομασίες σήμερα είναι πια συνώνυμες.

Η ονομασία yacc προέρχεται από τα αρχικά *Yet Another Compiler Compiler*. Η λέξη yacc ηχεί παρόμοια με τη λέξη yak, ένα ζώο το οποίο μοιάζει και συχνά συγχέεται με βίσωνα (bison), αν και ζουν σε διαφορετικές περιοχές.

##### 12.1.1 Το μετα-εργαλείο lex

Το μετα-εργαλείο lex (flex) [1] είναι ένας γεννήτορας λεκτικών αναλυτών. Μπορεί να συνεργάζεται με το μετα-εργαλείο yacc (bison), αλλά μπορεί να λειτουργήσει και αυτόνομα, όταν οι απαιτήσεις μας δεν συνεπάγονται συντακτική ανάλυση. Δέχεται ως είσοδο ένα μετα-πρόγραμμα που περιγράφει τις προς αναγνώριση λεκτικές μονάδες. Για κάθε λεκτική μονάδα που αναγνωρίζεται, ο προγραμματιστής μπορεί να καθορίσει ένα σύνολο ενεργειών που θα εκτελεστούν.

Η έξοδος είναι ένα πρόγραμμα σε γλώσσα προγραμματισμού υψηλού επιπέδου, το οποίο πρέπει να μεταγλωττιστεί για να εκτελεστεί. Περιέχει τη συνάρτηση `yylex()` η οποία υλοποιεί τον λεκτικό αναλυτή. Η σκιαγράφηση της διαδικασίας παραγωγής του λεκτικού αναλυτή εικονίζεται στο σχήμα 12.1.



Σχήμα 12.1: Παραγωγή λεκτικού αναλυτή με το lex.

Η συνάρτηση `yylex()` υλοποιεί τον λεκτικό αναλυτή. Για κάθε λεκτική μονάδα που αναγνωρίζεται, εκτελείται ο κώδικας που έχει οριστεί και, εάν υπάρχει συντακτικός αναλυτής, του επιστρέφεται το αποτέλεσμα της λεκτικής ανάλυσης. Αν δεν υπάρχει συντακτικός αναλυτής, τότε όλες οι ενέργειες γίνονται μέσα στον λεκτικό αναλυτή.

Το μετα-εργαλείο lex αναγνωρίζει λεκτικές μονάδες με τη χρήση κανονικών εκφράσεων. Η γλώσσα που περιγράφονται οι λεκτικές μονάδες δεν χρησιμοποιεί πάντοτε τους πιο εύχρηστους συμβολισμούς, είναι όμως αρκετά περιγραφική, δυναμική και ευέλικτη, ώστε να καλύπτει όλες τις ανάγκες. Ένα παράδειγμα κανονικής έκφρασης, υπό μορφή αυτομάτου, αλλά και με τον τρόπο που θα την συμβολίζαμε για να γίνει κατανοητή από τον lex, εικονίζεται στο σχήμα 12.2. Στο παράδειγμα του σχήματος αναγνωρίζεται ένας αριθμός που ανήκει στο σύνολο  $N^*$ , ο οποίος δεν ξεκινά με 0. Δηλαδή ο αριθμός 124 αναγνωρίζεται από την κανονική αυτή έκφραση, αλλά δεν αναγνωρίζεται ο αριθμός 0124. Φυσικά, δεν αναγνωρίζεται και το 0, αφού δεν ανήκει στο σύνολο  $N^*$ .



κανονική έκφραση: [1-9] [0-9]\*

Σχήμα 12.2: Παράδειγμα κανονικής έκφρασης που αναγνωρίζει ο lex.

Ας δούμε μερικούς συμβολισμούς που κατανοεί ο lex. Δεν αποτελεί σκοπό του κεφαλαίου η πλήρης περιγραφή της γλώσσας αυτής. Ο αναγνώστης μπορεί να βρει πολύ εύκολα αρκετή πληροφορία για αυτό αναρτημένη στο διαδίκτυο ή στη βιβλιογραφία.

[abc]: οποιοσδήποτε από τους χαρακτήρες a b ή c

[a-z]: οποιοσδήποτε από τους χαρακτήρες 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

[ -0-9]: προσημασμένος μονοψήφιος αριθμός

[ ^ a-zA-Z]: οτιδήποτε δεν είναι γράμμα

Ο χαρακτήρας <> υποδηλώνει οποιονδήποτε χαρακτήρα εκτός από την αλλαγή γραμμής

a?: καμία ή μία εμφάνιση του a

a\*: καμία ή περισσότερες εμφανίσεις του a

a+: μία ή περισσότερες εμφανίσεις του a

Παραδείγματα:

ab?c -> ac ή abc

[a-z]+ -> μη κενές συμβολοσειρές από μικρά γράμματα

[a-zA-Z] [a-zA-Z0-9]\* -> συμβολοσειρές από γράμματα και αριθμούς που ξεκινούν από γράμμα

Πολύ χρήσιμη είναι η δυνατότητα να ορίσουμε μνημονικά ονόματα. Αυτό κάνει την περιγραφή των κανονικών εκφράσεων πολύ πιο ευανάγνωστη. Στο ακόλουθο παράδειγμα ορίζουμε τα μνημονικά ονόματα `digit` και `letter`, τα οποία χρησιμοποιούμε στη συνέχεια για να εκφράσουμε μία μεταβλητή η οποία ξεκινάει από γράμμα και συνεχίζεται με μία σειρά από γράμματα ή ψηφία.

`digit` [0-9]

`letter` [a-zA-Z]

{`letter`} ( {`letter`} | {`digit`} )\*

Ένα πρόγραμμα lex αποτελείται από τρία τμήματα.

- Μέρος Α: περιέχει σχόλια, δήλωση μνημονικών ονομάτων και αρχικών καταστάσεων
- Μέρος Β: περιέχει την περιγραφή των προς αναγνώριση κανονικών εκφράσεων
- Μέρος Γ: ορισμοί συναρτήσεων

Πιο αναλυτικά, το Μέρος Α περιέχει:

- Σχόλια με τη σύμβαση της C

```
/* This is a comment */
```

- Μνημονικά ονόματα: χρησιμοποιούνται στο Β μέρος ως συντομογραφίες για κανονικές εκφράσεις, π.χ.

```
letter [A-Za-z]
digit [0-9]
```

- Δηλώσεις αρχικών καταστάσεων (δεν θα ασχοληθούμε με αυτές)
- Κώδικας C - περικλείεται από %{ και %}. Συνήθως περιέχει δηλώσεις μακροεντολών, τύπων δεδομένων και μεταβλητών που χρησιμοποιούνται από τον λεκτικό αναλυτή

Ένα παράδειγμα ακολουθεί:

```
%{
#define MAX_LEN 256

typedef struct {
    char lexeme[MAX_LEN];
    int lineNumber, charPosition;
} tokenInfo

int currentLine=1, currentChar=1;
%}
```

Το Μέρος Β αποτελείται από κανόνες που περιγράφουν ομάδες λεκτικών μονάδων. Σε κάθε κανόνα μπορεί να αντιστοιχίζονται κάποιες ενέργειες:

Κανονική έκφραση 1 ενέργεια 1

Κανονική έκφραση 2 ενέργεια 2

...

Κανονική έκφραση N ενέργεια N

Διαβάζονται χαρακτήρες από το αρχείο εισόδου έως ότου αναγνωριστεί το μακρύτερο πρόθεμα από μία από τις παραπάνω κανονικές εκφράσεις. Αν το πρόθεμα αυτό περιγράφεται από περισσότερες της μίας κανονικής έκφρασης, τότε επιλέγεται το πρώτο από αυτά. Ο κώδικας περιέχεται ανάμεσα στα σύμβολα %%.

Παράδειγμα:

```
%%
.
    charcount++;
\n
    { charcount++; linecount++ }
%%
```

Θυμίζουμε ότι η τελεία σημαίνει οτιδήποτε εκτός από αλλαγή γραμμής. Το \n σημαίνει αλλαγή γραμμής. Άρα ο παραπάνω κώδικας μετράει αριθμό χαρακτήρων και αριθμό γραμμών. Μπορεί να ξενίζει λίγο, αλλά πρέπει να παραδεχτούμε ότι αυτό που επιθυμούμε το περιγράφει πολύ εύκολα και συνοπτικά.

Το Μέρος Γ αποτελείται από κώδικα που μεταφέρεται αυτούσιος στο πρόγραμμα που παράγεται. Εδώ μπορούμε να βάλουμε συναρτήσεις που θα κληθούν στο Μέρος Β ή κάποια συνάρτηση main(), αν η εφαρμογή μας χρειάζεται μόνο λεκτική ανάλυση και όχι συντακτική (οπότε δεν χρησιμοποιείται καθόλου το yacc/bison). Στην περίπτωση που χρησιμοποιούμε συντακτική ανάλυση και επιστρατεύεται ο yacc/bison, τότε η συνάρτηση main() θα βρίσκεται στον συντακτικό αναλυτή.

Ένα παράδειγμα ακολουθεί:

```

int main()
{
    yylex();
    printf("There were %d characters in %d lines\n",
           charcount, linecount);
    return 0;
}

```

Στο παράδειγμα έχουμε κλήση της `yylex()` η οποία θα υπολογίσει τα `charcount` και `linecount`. Μετά την ολοκλήρωσή της, η `main()` θα τυπώσει τα `charcount` και `linecount` και θα επιστρέψει τον έλεγχο στο λειτουργικό σύστημα.

Ένα ολοκληρωμένο παράδειγμα μέτρησης αναγνωριστικών και αριθμών ακολουθεί:

```

digit [0-9]
letter [a-zA-Z]
ID {letter}({letter}|{digit})*
NUM {digit}+
NOTHING {digit}({letter}|{digit})*

%{
    #include<stdio.h>
    int count_id, count_num;
}

%%
{NUM}      count_num++;
{ID}       count_id++;
{NOTHING} ;
"\n"        ;
"."         ;
%%

int yywrap(void)
{
    // redefine yywrap()
    // called whenever the scanner reaches at the end of file
    return 0;
}

int main()
{
    yylex();
    printf("%d %d\n", count_id, count_num);
}

```

### 12.1.2 Το μετα-εργαλείο yacc/bison

Το μετα-εργαλείο *yacc*(*bison*) [1] είναι ένας γεννήτορας συντακτικών αναλυτών. Δέχεται ως είσοδο ένα μετα-πρόγραμμα το οποίο περιγράφει τη γραμματική που αναγνωρίζει την υπό υλοποίηση γλώσσα. Σε διάφορα σημεία της αναγνώρισης, κατά τη διάσχιση δηλαδή του συντακτικού δέντρου που περιγράφει το υπό ανάλυση πρόγραμμα, ο προγραμματιστής μπορεί να καθορίσει ένα σύνολο ενεργειών οι οποίες θα εκτε-

λεστούν, όταν η γραμματική περάσει από το σημείο αυτό. Η φιλοσοφία δεν διαφέρει από την προσθήκη σημασιολογικών κανόνων, όπως την είδαμε και την εφαρμόσαμε σε προηγούμενα κεφάλαια.

Η έξοδος είναι ένα πρόγραμμα σε γλώσσα προγραμματισμού υψηλού επιπέδου, το οποίο πρέπει να μεταγλωττιστεί για να εκτελεστεί. Περιέχει τη συνάρτηση `yyparse()`, η οποία υλοποιεί τον συντακτικό αναλυτή. Η συνάρτηση αναγνωρίζει ένα ορθό συντακτικά πρόγραμμα ή επιστρέφει μήνυμα λάθους. Η σκιαγράφηση του yacc/bison εικονίζεται στο σχήμα 12.3.



Σχήμα 12.3: Παραγωγή συντακτικού αναλυτή με το yacc.

Η σύνταξή του μετα-προγράμματος που δίνεται ως είσοδος στο yacc/bison αποτελείται πάλι από τρία μέρη:

- Μέρος Α: δίνονται ορισμοί.
- Μέρος Β: ορίζονται οι κανόνες της γλώσσας και οι σημασιολογικές ενέργειες που σχετίζονται με αυτούς.
- Μέρος Γ: ορίζονται εξωτερικές συναρτήσεις.

Στο Μέρος Α ορίζονται οι λεκτικές μονάδες της γλώσσας. Για παράδειγμα μπορούμε να ορίσουμε:

```
// keywords
%token program, begin, end, for, while
```

Μπορεί επίσης να περιέχει ορισμούς, οι οποίοι θα μεταφερθούν στον συντακτικό αναλυτή που θα παραχθεί. Ο κώδικας αυτός πρέπει να περιέχεται μέσα στα σύμβολα %{ και %}. Για παράδειγμα:

```
%
// definitions
# include <stdio.h>
int yylex();
void yyerror();
```

Στο τμήμα αυτό μπορεί να καθοριστεί το αρχικό σύμβολο της γραμματικής. Στο παράδειγμα καθορίζουμε ότι το αρχικό σύμβολο της γραμματικής είναι το `starting_symbol`:

```
%start starting_symbol
```

Στο Μέρος Β ορίζονται οι κανόνες της γραμματικής και οι ενέργειες που τους συνοδεύουν. Στο παράδειγμα, όταν ολοκληρώνεται επιτυχώς η μετάφραση ενός προγράμματος, τότε εμφανίζεται το ανάλογο μήνυμα επιτυχίας. Το `program_rule` είναι το αρχικό σύμβολο της γραμματικής:

```
program_rule
: program_tk id_tk
  programblock_rule
  { printf("program compiled sucessfully\n"); }
;
```

Το Μέρος Β περικλείεται ανάμεσα στα σύμβολα `%%` και `%%%`.

Στο Μέρος Γ ορίζονται βιοηθητικές συναρτήσεις. Οι συναρτήσεις αυτές μπορούν να κληθούν σε διάφορα σημεία μέσα στη γραμματική. Στο σημείο αυτό θα οριστεί και η συνάρτηση `main()`, η οποία θα καλέσει και την `yyparse()`. Στο παρακάτω απλό παράδειγμα ορίζεται μία `main()` στην απλούστερή της μορφή:

```
// definition of functions
...
// main program
int main()
{
    // syntactic analysis starts
    yyparse();
    // return success to the operating system
    return 1;
}
```

Όταν ορίζεται και υλοποιείται η `main()` με τον τρόπο αυτό, τότε η είσοδος στον μεταγλωττιστή που θα παραχθεί δίνεται από το πληκτρολόγιο ή με ανακατεύθυνση. Παρακάτω στο κεφάλαιο αυτό, Θα δούμε έναν τρόπο για να δίνεται η είσοδος από τη γραμμή εντολών.

Για να μεταφράσουμε ένα μετα-πρόγραμμα `lex` και ένα μετα-πρόγραμμα `yacc` εκτελούμε τα ακόλουθα βήματα:

- Αρχικά, βέβαια, συντάσσουμε σε έναν εκδότη κειμένου το αρχείο που θα αποτελέσει την είσοδο για τον `lex`. Ας υποθέσουμε ότι το ονομάζουμε: `test.lex`.
- Συντάσσουμε επίσης σε έναν εκδότη κειμένου το αρχείο που θα αποτελέσει την είσοδο για τον `yacc`. Ας υποθέσουμε ότι το ονομάζουμε: `test.yacc`.
- εκτελούμε στη γραμμή εντολών το εξής:

```
yacc -d test.yacc
```

Δημιουργείται το αρχείο `y.tab.c`. Η παράμετρος `-d` δημιουργεί το αρχείο `y.tab.h` με το οποίο μεταφέρονται δηλώσεις (π.χ. τα token που έχουν δηλωθεί στο μετα-πρόγραμμα για τον `yacc`) στον λεκτικό αναλυτή. Για τον λόγο αυτόν το μετα-πρόγραμμα του `lex` πρέπει να κάνει `include` το header `y.tab.h`:

```
#include "y.tab.h"
```

- Στη συνέχεια μεταφράζουμε το μετα-πρόγραμμα για τον `lex` και δημιουργείται το αρχείο `lex.yy.c`:

```
lex test.lex
```

- Τέλος κάνουμε μεταγλώττιση για να παραγάγουμε από τα αρχεία `lex.yy.c` και `y.tab.c`:

```
gcc -o complier lex.yy.c y.tab.c
```

Σε μερικά συστήματα πρέπει να γίνουν link στη γραμμή εντολών και οι βιβλιοθήκες `ll`, `ly`:

```
gcc -o complier lex.yy.c y.tab.c -ly -ll
```

Σε κάποια συστήματα αρκεί το `-ll`.

### 12.1.3 Ένα απλό μετα-πρόγραμμα yacc

Παρακάτω θα δούμε ένα απλό πρόγραμμα yacc, σε συνδυασμό με το αντίστοιχο πρόγραμμα για τον λεκτικό αναλυτή.

Ο λεκτικός αναλυτής αναγνωρίζει τις λεκτικές μονάδες που ανήκουν στην οικογένεια INTEGER και οι οποίες είναι ακέραιοι αριθμοί, καθώς και τις λεκτικές μονάδες “+” και “-”, οι οποίες αντιστοιχούν στα σύμβολα της πρόσθεσης και της αφαίρεσης, αντίστοιχα. Ο συντακτικός αναλυτής αναγνωρίζει αριθμητικές εκφράσεις που περιέχουν πρόσθεση και αφαίρεση. Η γραμματική ακολουθεί:

```
program
  : program expression '\n'
  |
  ;

expression
  : INTEGER
  | expression '+' expression
  | expression '-' expression
  ;
```

Για να μπορέσει να υποστηριχθεί η παραπάνω γραμματική, θα πρέπει ο λεκτικός αναλυτής να αντιστοιχίσει τις ακόλουθες ενέργειες στις λεκτικές μονάδες που αναγνωρίζει:

```
[0-9]+      {    yyval = atoi(yytext);
                  return INTEGER;
}

[ \t]        ;      // skip white characters

"."
{    // anything but new line
    yyerror();
}
```

Όποτε αναγνωρίζεται η κανονική έκφραση [0-9]+ στον συντακτικό αναλυτή επιστρέφεται μέσω της return η πληροφορία ότι αυτό που αναγνωρίστηκε είναι ένας INTEGER. Επίσης, μέσω της μεταβλητής yyval=atoi(yytext), επιστρέφεται η αριθμητική τιμή του ακεραίου που εντοπίστηκε. Το yytext είναι μια μεταβλητή στην οποία ο lex τοποθετεί τη συμβολοσειρά που αναγνωρίστηκε, ενώ μέσω της atoi() η συμβολοσειρά γίνεται ακέραιος και προωθείται στον συντακτικό αναλυτή. Εξ ορισμού, το yyval έχει ακέραια τιμή, αλλά έχουμε τη δυνατότητα να το επανα-օρίσουμε σε ό,τι τύπο θέλουμε μέσω της YYSTYPE ή της union.

Αν θέλουμε να δηλώσουμε την yyval ως συμβολοσειρά, τότε θα πρέπει να δηλώσουμε μέσα στο μετα-πρόγραμμα του yacc το:

```
#define YYSTYPE char*
```

Το union λειτουργεί κάπως διαφορετικά. Ένα παράδειγμα με union θα δούμε στην επόμενη ενότητα του κεφαλαίου αυτού.

Πριν περάσουμε στον συντακτικό αναλυτή, να σχολιάσουμε τις τρεις τελευταίες κανονικές εκφράσεις του μετα-προγράμματος του lex. Το:

```
[ \t]        return *yytext;
```

σημαίνει ότι αν αναγνωριστεί ένα από τα '+', '-', '\n', να επιστραφεί ο χαρακτήρας που αναγνωρίστηκε ως αποτέλεσμα στον συντακτικό αναλυτή.

Η έκφραση:

```
[ \t] ;
```

λέει ότι, αν βρεθεί ένα tab ή ένας κενός χαρακτήρας, ο λεκτικός αναλυτής πρέπει να το αγνοήσει.

Τέλος, η έκφραση:

```
". " yyerror();
```

καλεί τον διαχειριστή σφαλμάτων αν εμφανιστεί κάποιος άγνωστος χαρακτήρας στην είσοδο.

Ας περάσουμε τώρα στο τμήμα της συντακτικής ανάλυσης. Η γραμματική, όπως έχει οριστεί παραπάνω, έχει κάποιες αμφισημίες. Ενώ για εμάς είναι πολύ βολικό και εύκολο να ορίζουμε κανόνες με τον τρόπο που ορίσαμε παραπάνω τους κανόνες για την πρόσθεση και την αφαίρεση, για τη γραμματική δεν είναι προφανές ούτε ποιος κανόνας έχει προτεραιότητα ούτε αν οι πράξεις τελούνται από τα αριστερά στα δεξιά ή το αντίστροφο.

Αν η γλώσσα μας περιείχε και τις τέσσερις αριθμητικές πράξεις, θα μπορούσαμε απλά να ορίσουμε:

```
expression
: INTEGER
| expression '+' epxression
| expression '-' epxression
| expression '*' epxression
| expression '/' epxression
;
```

Θα έπρεπε όμως στο τμήμα των δηλώσεων να ορίζαμε την προτεραιότητα και την προσεταιριστικότητα. Αυτό μπορεί να γίνει ως εξής:

```
%left '+' '-'
%left '*' '/'
```

Η παραπάνω δήλωση σημαίνει ότι υπάρχει αριστερή προσεταιριστικότητα στις πράξεις '+', '-' και '\*', '/'; ενώ προτεραιότητα μικρότερη έχουν τα '+' και '-' έναντι των '\*' και '/'; αφού η δήλωσή τους προηγείται στο μετα-πρόγραμμα.

Αν αναρωτηθεί κανείς αν υπάρχουν πράξεις με δεξιά προσεταιριστικότητα, μπορεί να αναλογιστεί την εκχώρηση, αφού στην έκφραση

```
a=b=c;
```

το c είναι αυτό που δίνει τιμές στα a και b.

Στη γραμματική του παραδείγματος έχουμε μόνο πρόσθεση και αφαίρεση, άρα αρκεί να δηλώσουμε:

```
%left '+' '-'
```

Θα κάνουμε το πρόγραμμα περισσότερο ενδιαφέρον, προσθέτοντας κανόνες που προσδίδουν σημασιολογία στη γραμματική. Ο κανόνας `program` δημιουργεί μία σειρά από αριθμητικές εκφράσεις, οι οποίες αποτιμώνται από τον κανόνα `expression`. Ο κανόνας `expression` επιστρέφει το αποτέλεσμα της δικής του αποτίμησης στον κανόνα `program`, ο οποίος το τυπώνει.

Πώς όμως γίνεται η επικοινωνία ανάμεσα στους κανόνες; Στο παράδειγμά μας δεν έχουμε χρησιμοποιήσει ούτε την `YYSTYPE`, ούτε την `union`. Άρα αυτό που ανταλλάσσεται μεταξύ λεκτικού και συντακτικού αναλυτή, αλλά και μεταξύ των κανόνων του συντακτικού αναλυτή, είναι ακέραιος αριθμός. Για τις ανάγκες της εφαρμογής μας αυτό είναι ιδανικό, αφού θέλουμε να φτιάξουμε έναν αποτιμητή ακέραιων παραστάσεων.

Έχουμε συμφωνήσει στην ακόλουθη σύμβαση. Για τον κανόνα:

```
rule
: rule1 rule2 ... ruleN
;
```

το \$\$ θα αντιστοιχεί σε ό,τι επιστρέφει ο κανόνας rule, το \$1 σε ό,τι επιστρέφει ο κανόνας rule1, το \$2 σε ό,τι επιστρέφει ο κανόνας rule2 και το \$N, σε ό,τι επιστρέφει ο κανόνας ruleN.

Έτσι, στον κανόνα της γραμματικής:

```
program
  : program expression '\n'
  |
  ;
```

αν θέλουμε να τυπώσουμε ό,τι επιστρέφει κάθε expression, πρέπει να τοποθετήσουμε ένα printf() για κάθε expression. Αφού το expression είναι δεύτερο στο δεξί μέλος του κανόνα, τότε αυτό που θα τυπώσουμε θα είναι το \$2.

Έτσι, ο κανόνας θα γίνει:

```
program
  : program expression '\n'           { printf("%d\n",$2); }
  |
  ;
```

Το αποτέλεσμα του κανόνα expression θα πρέπει να υπολογιστεί μέσα στον κανόνα expression. Αν το expression δώσει ακέραιο αριθμό, τότε αυτός ο ακέραιος είναι που θα επιστραφεί ως αποτέλεσμα από το expression. Αν το expression αναγνωρίσει κάποια πράξη, τότε το αποτέλεσμα του expression θα είναι το αποτέλεσμα της πράξης που αναγνωρίστηκε πάνω στα τελούμενά της:

```
expression
  : INTEGER                         { $$ = $1; }
  | expression '+' expression      { $$ = $1 + $3; }
  | expression '-' expression    { $$ = $1 - $3; }
  ;
;
```

Παρατηρήστε ότι το expression θεωρείται ότι βρίσκεται στις θέσεις 1 και 3 στο δεξί μέλος του κανόνα, αφού στη θέση 2 βρίσκονται τα '+' και '-'.

Στην επόμενη ενότητα αυτού του κεφαλαίου θα δούμε τη δημιουργία ενός λεκτικού και συντακτικού αναλυτή μιας μίνι γλώσσας προγραμματισμού με τη βοήθεια των lex και yacc.

#### 12.1.4 Παράδειγμα υλοποίησης συντακτικού και λεκτικού αναλυτή με τα μετα-εργαλεία lex και yacc

Σκοπός του κεφαλαίου αυτού είναι να σας δείξει τις δυνατότητες των lex-yacc/bison και όχι να σας τα διδάξει αναλυτικά. Ο καταλληλότερος τρόπος για να γίνει αυτό εύκολα και ευχάριστα είναι με τη χρήση παραδειγμάτων.

Αρχικά θα κατασκευάσουμε έναν απλό συντακτικό αναλυτή, συνοδευόμενο από τον απαραίτητο λεκτικό αναλυτή. Ας ονομάσουμε τη γλώσσα cimpler. Για να κάνουμε περισσότερο ενδιαφέροντα τον μεταγλωττιστή της cimpler, θα προσθέσουμε σε αυτόν ενέργειες που θα παρακολουθούν τη δήλωση και τη χρήση των μεταβλητών σε ένα πρόγραμμα.

Η γραμματική της cimpler, γραμμένη ως μέρος ενός μετα-προγράμματος για τον yacc, ακολουθεί. Κάθε όνομα μη τερματικού συμβόλου αποτελεί κανόνα, οπότε προς διευκόλυνση, το όνομά του τελειώνει με “\_rule”. Κάθε τερματικό σύμβολο αποτελεί μία λεκτική μονάδα, οπότε του έχει δοθεί κατάληξη “\_tk” (από τη λέξη “token”):

```
program_rule
  : program Tk id Tk
    programblock_rule
  ;
```

```

programblock_rule
: declarations_rule
begin_tk
assignments_rule
end_tk
;

declarations_rule
: declare_tk
list_of_variables_rule
enddeclare_tk
;

list_of_variables_rule
: id_tk
| list_of_variables_rule comma_tk id_tk
;

assignments_rule
: assignments_rule assignment_rule
| assignment_rule
;

assignment_rule
: id_tk assign_tk expression_rule semicolon_tk
;

expression_rule
: term_rule
| expression_rule plus_tk term_rule
| expression_rule minus_tk term_rule
;

term_rule
: factor_rule
| term_rule times_tk factor_rule
| term_rule over_tk factor_rule
;

factor_rule
: id_tk
| num_tk
| left_par_tk expression_rule right_par_tk
;

```

Εύκολα θα παρατηρήσετε ότι η cimpler υποστηρίζει μόνο δηλώσεις μεταβλητών και εκχωρήσεις σε αυτές, έπειτα από αποτιμήσεις εκφράσεων. Δεν θα δυσκολευτείτε, επίσης, να παρακολουθήσετε τη γραμματική, οπότε μπορούμε να παραλείψουμε τον σχολιασμό της. Αυτό που έχει περισσότερο ενδιαφέρον είναι να δούμε τι χρειάζεται να δηλωθεί στα υπόλοιπα σημεία του μετα-προγράμματος του yacc ώστε να γίνει αυτή η γραμματική λειτουργική και να μπορεί να συνεργαστεί με το μετα-πρόγραμμα του lex.

Στο Μέρος A, πρέπει να εισάγουμε τους ορισμούς:

```
%{
    // definitions
    # include <stdio.h>
    int yylex();
    void yyerror();
%}
```

Το `include` μας επιτρέπει να χρησιμοποιήσουμε την `printf()`, ενώ οι υπόλοιποι δύο ορισμοί έχουν να κάνουν με συναρτήσεις των lex-yacc που θα επανα-οριστούν παρακάτω.

Στη συνέχεια ακολουθεί η επικοινωνία με τον λεκτικό αναλυτή. Θα εκμεταλλευτούμε το `union`, το οποίο είναι ο περισσότερο ευέλικτος και δυναμικός τρόπος επικοινωνίας ανάμεσα στον λεκτικό και τον συντακτικό αναλυτή και γ' αυτόν τον λόγο το επιλέξαμε για το παράδειγμά μας.

Το `union` το δανειζόμαστε ως ιδέα από τη γλώσσα προγραμματισμού C. Είναι μία δομή η οποία μοιάζει με τη δομή `struct`, αλλά έχει διαφορετική υλοποίηση και χρησιμότητα.

'Όταν ορίζουμε στη C:

```
struct {
    int my_integer;
    char a_string[50];
};
```

τότε ο μεταγλωττιστής δεσμεύει χώρο για έναν ακέραιο και 50 ακόμα bytes για το string. Σε κάθε τέτοιο `struct` μπορούμε να αποθηκεύσουμε έναν ακέραιο αριθμό και μία συμβολοσειρά μήκους, το πολύ, 49 χαρακτήρων.

'Όταν ορίζουμε:

```
union {
    int my_integer;
    char a_string[50];
};
```

τότε ο μεταγλωττιστής θα δεσμεύσει χώρο συνολικά 50 χαρακτήρων. Σε κάθε τέτοιο `union` μπορούμε να αποθηκεύσουμε έναν ακέραιο ή μία συμβολοσειρά μήκους το πολύ 49 χαρακτήρων, όχι και τα δύο μαζί.

'Ετσι, όταν έχουμε ορίσει το `union u`, και θέλουμε να εκχωρήσουμε τον ακέραιο 1 στο u, τότε θα πρέπει να κάνουμε το εξής:

```
u.my_integer = 1;
```

'Όταν θέλουμε να εκχωρήσουμε στο u μία συμβολοσειρά, τότε χρησιμοποιούμε το άλλο πεδίο και κάνουμε την εκχώρηση ως εξής:

```
strcpy(u.a_string,"this is a string");
```

Στο παράδειγμά μας θα καθορίσουμε μια ειδική απαίτηση. Θα ζητήσουμε ο λεκτικός αναλυτής να επιστρέψει στον συντακτικό αναλυτή και τη συμβολοσειρά που αναγνωρίστηκε, αλλά και τη γραμμή στην οποία βρέθηκε. Αυτό είναι μια τεχνητή απαίτηση, η οποία ζητείται για εκπαιδευτικούς λόγους, αφού ο αριθμός γραμμής συνήθως επιστρέφεται με τη μεταβλητή `yyline`.

Προκειμένου το `union` να μην δεσμεύσει χώρο μόνο για το μεγαλύτερο από τα πεδία του, κάτι που σημαίνει ότι μόνο ένα από τα πεδία του μπορεί να επιστραφεί κάθε φορά, εμείς θα ορίσουμε ένα `struct` μέσα στο `union`, ώστε να δεσμεύεται τόσος χώρος, όσος απαιτείται για ολόκληρο το `struct`:

```
%union {
    struct {
        int line_number;
        char token_string[50];
    }token;
}
```

Ονομάσαμε το struct “token” για να μπορούμε να αναφερόμαστε σε αυτό.

Στον τρόπο επικοινωνίας ανάμεσα σε συντακτικό και λεκτικό αναλυτή συγκαταλέγονται και οι λεκτικές μονάδες τις οποίες αναγνωρίζει ο λεκτικός αναλυτής και τις προωθεί ως είσοδο στον συντακτικό αναλυτή. Αυτές οι λεκτικές μονάδες θα δηλωθούν μέσα στο μετα-πρόγραμμα του yacc και θα περαστούν στον lex μέσα από το αρχείο y.tab.h. Το αρχείο y.tab.h θα παραχθεί κατά την παραγωγή του συντακτικού αναλυτή και θα πρέπει να γίνει #include στον λεκτικό αναλυτή.

Η δήλωση των λεκτικών μονάδων της γλώσσας ακολουθεί:

```
%token id_tk num_tk program_tk declare_tk enddeclare_tk
begin_tk end_tk comma_tk assign_tk semicolon_tk
plus_tk minus_tk times_tk over_tk left_par_tk right_par_tk
```

Τέλος θα ορίσουμε πιθανές βοηθητικές συναρτήσεις, καθώς και το main():

```
// definition of functions
void yyerror (char const *s)
{
    // called when an error occurs
    // define your own function here
    // this one simply prints a "syntax error" message
    printf ("%s\n", s);
}

// main program
int main(int argc, char** argv)
{
    extern FILE * yyin;
    // read input from the file defined in the command line
    if(argc==2)
    {
        yyin = fopen(argv[1], "r");
        if(!yyin)
        {
            fprintf(stderr, "can't read file %s\n", argv[1]);
            return 0;
        }
    }

    // syntactic analysis begins
    yyparse();
    // return success to the operating system
    return 1;
}
```

Στο κυρίως πρόγραμμα χρησιμοποιούμε τη μεταβλητή yyin στην οποία εκχωρούμε το πρώτο όρισμα από τη γραμμή εντολών. Με τον παραπάνω κώδικα η είσοδος στον μεταγλωττιστή που παρήθη δίνεται από τη γραμμή εντολών:

```
cimpler input.cimpler
```

Αν δηλώσουμε απλά

```
int main()
{ ... }
```

όπως κάναμε μέχρι τώρα, η είσοδος στον μεταγλωττιστή πρέπει να δοθεί από το πληκτρολόγιο ή με ανακατεύθυνση:

```
cimpler < input.cimpler
```

Τέλος, ας προσθέσουμε την παραγωγή κάποιων μηνυμάτων στη γραμματική του συντακτικού αναλυτή, έτσι ώστε να δούμε πώς χρησιμοποιούμε την πληροφορία που μεταφέρεται από κανόνα σε κανόνα μέσω του union. Θα υλοποιήσουμε την απαίτηση, όταν μία μεταβλητή δηλώνεται, εκχωρείται ή διαβάζεται, να τυπώνεται ένα κατάλληλο μήνυμα.

Πρέπει, δηλαδή, να εντοπίσουμε μέσα στη γραμματική τα σημεία στα οποία γίνονται οι παραπάνω ενέργειες, να διαβάσουμε το όνομα της μεταβλητής που εντοπίστηκε, τον αριθμό γραμμής που εμφανίστηκε και να τα τυπώσουμε. Και οι δύο αυτές πληροφορίες δίνονται μέσα από το union και δεν έχουμε παρά να τις χρησιμοποιήσουμε.

Η δήλωση των μεταβλητών γίνεται μέσα στο list\_of\_variables\_rule. Εκεί εμφανίζεται το id\_tk δύο φορές: μία στην αρχή και μία έπειτα από κάθε κόμμα. Σε αυτά τα δύο σημεία πρέπει να εισάγουμε κώδικα που θα εμφανίζει το μήνυμα που επιθυμούμε:

```
list_of_variables_rule
: id_tk
{   // the first variable has been declared
    printf("Variable '%s' was the first variable declared ",
           $<token.token_string>1);
    printf("in line %d\n", $<token.line_number>1);
}
| list_of_variables_rule comma_tk id_tk
{   // variable declared after comma
    printf("Variable '%s' declared after comma ",
           $<token.token_string>3);
    printf("in line %d\n", $<token.line_number>3);
}
;
```

Το ενδιαφέρον σημείο είναι το \$<token.token\_string>1 και τα αντίστοιχα. Ας δούμε τι σημαίνει. Ο χαρακτήρας \$ είναι ο ίδιος που είχαμε χρησιμοποιήσει στο προηγούμενο παράδειγμα, με τις αποτιμήσεις των προσθέσεων και των αφαιρέσεων, προκειμένου να αποκτήσουμε πρόσβαση στην πληροφορία που επιστρέφει ένας κανόνας. Εκεί, όμως, καθετί που επιστρέφοταν ήταν ακέραιος αριθμός, άρα δεν είχαμε παρά να τοποθετήσουμε έναν αριθμό δίπλα από το \$, ο οποίος έδειχνε σε ποιο κατά σειρά σύμβολο στο δεξί μέρος του κανόνα αναφερόμασταν. Η ανάγκη να εντοπίσουμε το σύμβολο ακόμα υπάρχει, οπότε εξακολουθούμε να τοποθετούμε έναν τέτοιο αριθμό στο τέλος της αναφοράς σε ένα σύμβολο.

Παρατηρήστε τα 1 και 3 στις παρακάτω αναφορές στο token:

```
$<token.token_string>1, $<token.token_string>3
```

τα οποία αντιστοιχούν στα δύο id\_tk, στην πρώτη και τρίτη θέση της πρώτης και της δεύτερης εναλλακτικής του κανόνα list\_of\_variables\_rule.

Μένει να δούμε τι είναι τα <token.token\_string> και τα <token.line\_number>. Το token είναι το όνομα που έχουμε δώσει στο struct μέσα στο union, ενώ τα token\_string και line\_number είναι τα ονόματα που έχουμε δώσει στα πεδία μέσα στο struct.

Ένα δεύτερο σημείο στο οποίο οφείλουμε να εισάγουμε κώδικα είναι στους κανόνες της εκχώρησης. Εκεί το id\_tk βρίσκεται στην πρώτη θέση του δεξιού μέλους του κανόνα, άρα η πληροφορία θα ανακτηθεί ως:

```
$<token.token_string>1, // string
$<token.line_number>1 // integer
```

Ο κανόνας, τώρα, γίνεται:

```
assignment_rule
: id_tk assign_tk expression_rule semicolon_tk
```

```

    { // assignment of a variable
        printf("Variable '%s' assigned ",
               $<token.token_string>1);
        printf("in line %d\n",$<token.line_number>1);
    }
;

```

Για να τυπώσουμε μήνυμα με το σημείο στο οποίο γίνεται χρήση μιας μεταβλητής, πρέπει να προσθέσουμε κώδικα στον κανόνα factor\_rule, ο οποίος, σε αναλογία με ό,τι έχει συζητηθεί προηγουμένως, γίνεται:

```

factor_rule
: id_tk
{ // use of a variable
printf("Variable '%s' used ",$<token.token_string>1);
printf("in line %d\n",$<token.line_number>1);
}
| num_tk
| left_par_tk expression_rule right_par_tk
;

```

Ολόκληρο το μετα-πρόγραμμα του συντακτικού αναλυτή καθώς και το αντίστοιχο του λεκτικού, μπορεί να αναζητηθεί στα παραρτήματα του βιβλίου.

## 12.2 Το μετα-εργαλείο ANTLR

Ένα ακόμα σημαντικό και δημοφιλές εργαλείο ανάπτυξης μεταγλωττιστών είναι το ANTLR (*ANother Tool for Language Recognition*) [2]. Όπως και τα lex, yacc/bison, το ANTLR παίρνει ως είσοδο την περιγραφή των λεκτικών μονάδων ως κανονικών εκφράσεων και τη γραμματική της γλώσσας εκφρασμένη ως γραμματική χωρίς συμφραζόμενα. Ο διαχωρισμός της λεκτικής και της συντακτικής ανάλυσης παύει να αποτελεί είσοδο από διαφορετικά αρχεία και ενοποιείται σε ένα ενιαίο συμβολισμό. Αποτελεί μία περισσότερο σύγχρονη προσέγγιση και υλοποίηση της ίδιας ιδέας, η οποία εμπλουτίστηκε και έγινε εξαιρετικά ενδιαφέροντα. Υποστηρίζει Java, C#, Python και άλλες γλώσσες προγραμματισμού.

Ο Terence Parr βρίσκεται πίσω από το ANTLR και δουλεύει πάνω σε τέτοια θέματα από το 1989. Είναι Καθηγητής Πληροφορικής στο University του San Francisco.

### 12.2.1 Συντακτική ανάλυση με το ANTLR

Θα ακολουθήσουμε την ίδια λογική με την παρουσίαση των lex και yacc και θα στοχεύσουμε στην κατανόηση των βασικών εννοιών, ώστε να γνωρίζουμε τις δυνατότητες του εργαλείου, αλλά και να αποκτηθεί το κατάλληλο υπόβαθρο, ώστε εύκολα και γρήγορα ο αναγνώστης να μπορέσει να εμβαθύνει σε αυτό, αν το χρειαστεί. Το ANTLR υποστηρίζει, μεταξύ άλλων, τη γλώσσα Python. Αν και το ANTLR είναι συνυφασμένο με τη γλώσσα Java, εμείς θα χρησιμοποιήσουμε στο σύγγραμμα αυτό τη γλώσσα Python, στην ίδια φιλοσοφία που ακολουθήθηκε στο υπόλοιπο βιβλίο.

Αφού εγκαταστήσουμε το ANTLR και το run-time της Python, έμαστε έτοιμοι να συντάξουμε το αρχείο που θα αποτελέσει την είσοδο στο ANTLR και θα περιέχει την περιγραφή των λεκτικών μονάδων και των κανόνων της γραμματικής. Ας ονομάσουμε αυτό το αρχείο Grammar.g4.

Η γραμματική περιγράφει την ίδια γλώσσα που χρησιμοποιήσαμε στο παράδειγμα για τα lex/yacc. Δεν θα την ακολουθήσουμε κανόνα-κανόνα, αφού δεν υπάρχει κάποιος λόγος να δυσκολέψει τον αναγνώστη. Θα σημειώσουμε, μόνο, τις τελευταίες τρεις γραμμές που ορίζουν τις κανονικές εκφράσεις για τα αναγνωριστικά, τις αριθμητικές σταθερές και τους λευκούς χαρακτήρες. Όταν αναγνωρίζεται ένας λευκός χαρακτή-

ρας, τότε αυτός αγνοείται. Παρατηρήστε τον εύχρηστο και ευανάγνωστο τρόπο με τον οποίο συμβολίζονται τα τερματικά σύμβολα / λέξεις-κλειδιά μέσα στη γραμματική.

```

grammar Grammar;

startRule                               // starting symbol
:   'program' ID
    programblock
;

programblock                            // program structure
:   declarations
    'begin'
        assignments
    'end'
;

declarations                           // declaration of variables
:   'declare'
    list_of_variables
    'enddeclare'
;

list_of_variables                      // comma separated variables
:   ID
| list_of_variables ',' ID
;

assignments                            // a sequence of assignments
:   assignments assignment
|   assignment
;

assignment                             // a single assignment
:   ID ':=' expression ';'
;

expression                            // a mathematical expression
:   term
|   expression '+' term
|   expression '-' term
;

term                                  // expressions consist of terms
:   factor
|   term '*' factor
|   term '/' factor
;

factor                                // terms consist of factors
:   ID
|   INTEGER
;
```

```

| '(' expression ')'
;

// definitions for terminal symbols
INTEGER      : [0-9]+ ('.' [0-9]+)? ([eE] [+ -]? [0-9]+)?;
ID           : [a-zA-Z]+;

WS           : [ \t\r\n] -> skip;      // skip whitespaces

```

Ας δοκιμάσουμε να τοποθετήσουμε κώδικα ο οποίος θα εμφανίζει μηνύματα που σχετίζονται με τη χρήση των μεταβλητών, όπως ακριβώς κάναμε και στο παράδειγμα με τα lex/yacc.

Η δήλωση των μεταβλητών γίνεται στον κανόνα declarations. Το σημείο όμως της γραμματικής που εμφανίζεται το ID βρίσκεται μέσα στον κανόνα list\_of\_variables, ο οποίος καλείται από τον declarations. Το ANTLR μας προσφέρει έτοιμο το \$ID.text και \$ID.line, τα οποία μας επιστρέφουν τη συμβολοσειρά που αναγνωρίστηκε και τη γραμμή στην οποία έγινε αυτό.

Δεν έχουμε, λοιπόν, παρά στον κανόνα list\_of\_variables να τοποθετήσουμε τον παρακάτω κώδικα:

```

list_of_variables
:   ID
        {print($ID.text, 'declared in line',$ID.line)}
| list_of_variables ',' ID
        {print($ID.text, 'declared in line',$ID.line)}
;

```

Στον κανόνα αυτόν, παρότι εμφανίζεται δύο φορές το ID, δεν δημιουργείται σύγχυση για το σε ποιο από τα δύο ID αναφέρεται κάθε \$ID.text και \$ID.line, αφού το καθένα από αυτά αντιστοιχίζεται εμφανώς σε μία από τις δύο εναλλακτικές του κανόνα list\_of\_variables. Αν, όμως, ο κανόνας μας ήταν κάπως έτσι:

```

addition
:   ID '+' ID
        {print($ID.text, '+', $ID.text)}
;

```

τότε δεν είναι ξεκάθαρο τα \$ID.text και \$ID.line σε ποια από τις δύο εμφανίζεις του ID αναφέρονται. Στην περίπτωση αυτή θα πρέπει να ονοματοδοτήσουμε καθεμία από τις εμφανίσεις αυτές, ώστε να μην υπάρχει αμφισημία. Η παραπάνω γραμματική θα γίνει:

```

addition
:   a = ID '+' b = ID
        {print($a.text, '+', $b.text)}
;

```

Ας επιστρέψουμε όμως στο παράδειγμα. Η εκχώρηση σε μία μεταβλητή γίνεται στον κανόνα assignment:

```

assignment
:   ID ':=' expression ';' {print($ID.text, 'set in line',$ID.line)}
;

```

ενώ μία μεταβλητή χρησιμοποιείται στον κανόνα factor:

```

factor
:   ID
        {print($ID.text, 'read in line',$ID.line)}
| INTEGER
| '(' expression ')'
;

```

Το ακόλουθο πρόγραμμα:

```

program test
declare
    x, y,
    z
enddeclare
begin
    x := 1+2-3+4*(1-3)/x;
    z := x;
    y := z;
end

```

δίνει την παρακάτω έξοδο:

```

x declared in line 3
y declared in line 3
z declared in line 4
x read in line 7
x set in line 7
x read in line 8
z set in line 8
z read in line 9
y set in line 9

```

Η γραμματική χρειάζεται και ένα κυρίως πρόγραμμα για να λειτουργήσει. Το κυρίως πρόγραμμα έχει τη μορφή που φαίνεται παρακάτω:

```

import sys
from antlr4 import *
from GrammarLexer import GrammarLexer
from GrammarParser import GrammarParser

def main(argv):
    input_stream = FileStream(argv[1])
    lexer = GrammarLexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = GrammarParser(stream)
    tree = parser.startRule()

if __name__ == '__main__':
    main(sys.argv)

```

Τα GrammarLexer και GrammarParser παράγονται αυτόματα από τη γραμματική. Το `input_stream` είναι ιυπεύθυνο για την είσοδο, το `lexer` είναι ο λεκτικός αναλυτής, ο οποίος επικοινωνεί με τον συντακτικό αναλυτή `parser` μέσω του `stream`. Το συντακτικό δέντρο σχηματίζεται από τον κανόνα `startRule`, ο οποίος έχει μετατραπεί σε μια μέθοδο του `parser`.

Ένα περισσότερο ολοκληρωμένο παράδειγμα χρήσης του ANTLR μπορεί κανείς να βρει στα παραρτήματα του βιβλίου, όπου παρουσιάζεται ένας μορφωτής προγραμμάτων.

### 12.2.2 Σύνταξη γραμματικών ANTLR

Ας περάσουμε από το παράδειγμα σε μία περισσότερο αναλυτική περιγραφή της σύνταξης μίας γραμματικής ANTLR.

Το όνομα του αρχείου που περιέχει τη γραμματική πρέπει να είναι `X.g4`. Απαιτείται να περιέχει τουλάχιστον έναν κανόνα.

Τα ονόματα των λεκτικών μονάδων (lexer rules) μέσα στη γραμματική ξεκινούν πάντοτε με κεφαλαίο γράμμα, ενώ τα ονόματα των κανόνων (parser rules) με μικρό γράμμα. Η υπόλοιπη ονομασία μπορεί να αποτελείται από μικρά, κεφαλαία, ψηφία και κάτω παύλες:

```
ID, INTEGER      // tokens
expression, assignmnent, statements, special_statements // rules
```

Το ANTLR δεν διαχωρίζει ανάμεσα σε χαρακτήρες και συμβολοσειρές. Όλες οι συμβολοσειρές με έναν ή περισσότερους χαρακτήρες τοποθετούνται μέσα σε απλά εισαγωγικά:

```
';', 'if', 'else', '>='
```

Το ANTLR επίσης καταλαβαίνει τους συνηθισμένους χαρακτήρες διαφυγής:

```
'\n' (newline), '\r' (carriage return), '\t' (tab),
'\b' (backspace)
```

Οι γραμματικές στις οποίες στην επικεφαλίδα (header) δεν δηλώνεται κάτι πέρα από το όνομα της γραμματικής είναι συνδυασμένες γραμματικές (combined grammars) και μπορεί να περιέχουν και λεκτικούς και συντακτικούς κανόνες:

```
grammar myGrammar;
```

Όταν μία γραμματική θέλουμε να περιέχει μόνο συντακτικούς κανόνες, δηλώνουμε την κεφαλίδα ως εξής:

```
parser grammar myGrammar;
```

Όταν μία γραμματική θέλουμε να περιέχει μόνο λεκτικούς κανόνες, δηλώνουμε την κεφαλίδα ως εξής:

```
lexer grammar myGrammar;
```

Ο μηχανισμός εισαγωγών (grammar imports) επιτρέπει να τμηματοποιήσουμε μία γραμματική σε λογικά και επαναχρησιμοποιήσιμα τμήματα. Το ANTLR χρησιμοποιεί τον μηχανισμό αυτόν όπως γίνεται στις αντικειμενοστραφείς γλώσσες προγραμματισμού με την κληρονομικότητα: μία γραμματική κληρονομεί όλους τους κανόνες, τις λεκτικές μονάδες και τις ονοματισμένες ενέργειες (named actions) από την εισαγόμενη γραμματική. Οι κανόνες στην κύρια γραμματική υποσκελίζουν (override) τους κανόνες από την εισαγόμενη γραμματική, αν αυτοί έχουν το ίδιο όνομα. Το αποτέλεσμα όλων των εισαγωγών σε μία γραμματική δίνει πάντοτε μία συνδυασμένη γραμματική.

Σε μία γραμματική μπορούν να οριστούν με οποιαδήποτε σειρά:

- επιλογές (options),
- εισαγωγές (imports),
- λεκτικές μονάδες (tokens),
- ονοματισμένες ενέργειες (named actions).

καθένα από τα οποία, όμως, μπορεί να εμφανίζεται το πολύ μία φορά. Στις επιλογές καθορίζουμε διάφορες παραμέτρους, όπως ας πούμε η γλώσσα που θα παραχθεί ή το πόσα από τα πρώτα σύμβολα της εισόδου θα κοιτάξει προκειμένου να πάρει κάποια απόφαση (look ahead symbols). Στο παράδειγμα, η γλώσσα που επιλέγεται είναι η Java, ενώ θα ληφθούν υπόψη τα δύο πρώτα σύμβολα της εισόδου:

```
options {
    language = "Java";
    k = 2;
}
```

Οι ονοματισμένες ενέργειες μπορούν να βρεθούν μέσα στα τμήματα @header{...} και @member{...}. Ότι βρεθεί μέσα στο @header{...} θα τοποθετηθεί στην αρχή της κλάσης του συντακτικού αναλυτή που θα παραχθεί. Θα μπορούσαν εδώ να τοποθετηθούν εισαγωγές πακέτων. Μέσα στο @member{...} τοποθετούνται πεδία και μέθοδοι του συντακτικού αναλυτή. Το παράδειγμα στο παράρτημα III του συγγράμματος είναι αρκετά διαφωτιστικό για το πώς χρησιμοποιούμε το @member{...}.

### 12.2.3 Προγραμματίζοντας με listeners

Οι listeners είναι μέθοδοι οι οποίοι καλούνται όταν ενεργοποιείται και όταν τερματίζεται ένας κανόνας. Τις σημειώνουμε μέσα στη γραμματική με το σύμβολο #. Στην ραμματική του παραδείγματος, αν επιθυμούμε να τυπώσουμε μήνυμα για τη δήλωση, την εκχώρηση και τη χρήση μιας μεταβλητής, θα πρέπει να ορίσουμε listeners στα ακόλουθα σημεία:

```
list_of_variables
:   a=ID                                #printDeclaration
| list_of_variables ', ' a=ID             #printDeclaration
;

assignment
:   a=ID ':=' expression ';'              #printAssignment
;

factor
: a=ID                                     #printUse
| INTEGER                                    #pass
| '(' expression ')'                      #pass
;
```

Έχουμε ορίσει τέσσερις listeners. Ο πρώτος (#printDeclaration) είναι υπεύθυνος για να εμφανίσει το κατάλληλο μήνυμα για τη δήλωση μίας μεταβλητής, ο δεύτερος (#printAssignment) και ο τρίτος (#printUse) για την εκχώρηση και τη χρήση της μεταβλητής. Ο listener #pass δεν κάνει τίποτα, αλλά πρέπει να βρίσκεται εκεί, αφού όταν ορίσουμε listener για μία εναλλακτική κάποιου κανόνα, πρέπει να ορίσουμε για όλες.

Παρατηρήστε ότι ο listener #printDeclaration εμφανίζεται δύο φορές στη γραμματική, αφού στα δύο σημεία αυτά της γραμματικής οι ενέργειες που θέλουμε να γίνουν είναι οι ίδιες.

Μία γραμματική ενεργοποιείται από ένα κυρίως πρόγραμμα. Για τη γραμματική Grammar, το κυρίως πρόγραμμα που θα την ενεργοποιήσει φαίνεται παρακάτω. Μέσα στο κυρίως πρόγραμμα τοποθετείται και ο κώδικας που είναι υπεύθυνος για την ενεργοποίηση των listeners:

```
import sys
from antlr4 import *
from GrammarLexer import GrammarLexer
from GrammarParser import GrammarParser
from MyListener import MyListener

def main(argv):
    input_stream = FileStream(argv[1])
    lexer = GrammarLexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = GrammarParser(stream)
    tree = parser.startRule()
```

```

    listener = MyListener()
    walker = ParseTreeWalker()
    walker.walk(listener, tree)

if __name__ == '__main__':
    main(sys.argv)

```

Στο κυρίως πρόγραμμα, όπως το είχαμε αφήσει στην προηγούμενη ενότητα, προσθέτουμε την εισαγωγή και την ενεργοποίηση του listener. Για κάθε listener που ορίζεται, δημιουργούνται δύο μέθοδοι, η enter και η exit. Έτσι για τον listener #printAssignment θα δημιουργηθούν οι μέθοδοι #exitPrintAssignment και #exitPrintAssignment, το σώμα των οποίων θα είναι κενό.

Η προσφιλής και δομημένη προσέγγιση είναι να μην χρησιμοποιήσουμε τις μεθόδους αυτές, οι οποίες μάλιστα επαναδημιουργούνται κάθε φορά που μεταγλωττίζουμε τη γραμματική, αλλά να δημιουργήσουμε μία υποκλάση της, μέσα στην οποία θα επαναφεύγουμε. Αυτός είναι ο λόγος που στον παραπάνω κώδικα βλέπουμε να εισάγεται η κλάση MyListener.

Τι έχει μέσα αυτή η κλάση; Η κλάση κληρονομεί την κλάση GrammarListener που δημιούργησε το ANTLR και υλοποιεί τις μεθόδους που μας ενδιαφέρουν, δηλαδή τις exitPrintDeclaration, exitPrintAssignment, exitPrintUse. Όλες οι υπόλοιπες μέθοδοι της κλάσης GrammarListener, δηλαδή οι enterPrintDeclaration, enterPrintAssignment, enterPrintUse, exitPass, enterPass, παραμένουν κενές.

Ας δούμε και την υλοποίηση των μεθόδων:

```

from GrammarListener import GrammarListener
from GrammarParser import GrammarParser

class MyListener(GrammarListener):

    def exitPrintDeclaration(self, ctx:GrammarParser.PrintDeclarationContext):
        print(ctx.a.text, 'declared in line',ctx.a.line)

    def exitPrintAssignment(self, ctx:GrammarParser.PrintAssignmentContext):
        print(ctx.a.text, 'set in line',ctx.a.line)

    def exitPrintUse(self, ctx:GrammarParser.PrintUseContext):
        print(ctx.a.text, 'used in line',ctx.a.line)

```

Ενδιαφέρον έχει πώς προσπελάζουμε τα διάφορα μεταφραστικά πεδία. Δείτε παραπάνω στη γραμματική ότι έχουμε δώσει το όνομα a στα σημεία που μας ενδιαφέρει να χρησιμοποιήσουμε τη συμβολοσειρά που αναγνώρισε ο λεκτικός αναλυτής μέσα από το ID. Η συμβολοσειρά που αναγνώρισε επιστρέφεται από το ctx.a.text, ενώ η γραμμή στην οποία βρέθηκε η συμβολοσειρά από το ctx.a.line. Μόλις, λοιπόν, αναγνωρίζεται το όνομα της μεταβλητής, τότε, ανάλογα με τον κανόνα που τη στιγμή αυτή είναι ενεργοποιημένος, τυπώνουμε το κατάλληλο μήνυμα, το οποίο περιέχει το όνομα της μεταβλητής και τη γραμμή στην οποία βρίσκεται.

## Βιβλιογραφία

- [1] John R. Levine, Tony Mason και Doug Brown. *Lex & Yacc (2nd Ed.)* USA: O'Reilly & Associates, Inc., 1992. ISBN: 1565920007.
- [2] Terence Parr. *The Definitive ANTLR 4 Reference*. 2η έκδοση. Raleigh, NC: Pragmatic Bookshelf, 2013. ISBN: 978-1-93435-699-9.



## ΠΑΡΑΡΤΗΜΑ Α

---

## ΑΝΑΛΥΤΙΚΟ ΠΑΡΑΔΕΙΓΜΑ ΜΕΤΑΓΛΩΤΤΙΣΗΣ

---

Στο παράρτημα αυτό θα ακολουθήσουμε την πορεία μεταγλώττισης ενός πλήρους παραδείγματος.

Η παρουσίαση θα γίνει με τέτοιο τρόπο, ώστε να φαίνεται η χρονική αλληλουχία των βημάτων και σε ποια χρονική στιγμή γίνεται τι. Η εμφάνιση των λεκτικών μονάδων που αναγνωρίζονται, καθώς και η εμφάνιση της γραμμής στην οποία αναγνωρίστηκαν δίνουν την έννοια του χρόνου και επιτρέπουν την παρακολούθηση της διαδικασίας μεταγλώττισης.

Το πρόγραμμα το οποίο θα μεταγλωττιστεί είναι το ακόλουθο:

```
1     program appendix
2     {
3         # declarations for main #
4         declare a,x,y;
5
6         function Equal(in x, inout y)
7         {
8             procedure B(in b1, inout b2)
9             {
10                 # body of B
11                 if (b2>b1)
12                     while (b2>b1)
13                         b2 := b2 - 1;;
14                 else
15                     while (b1>b2)
16                         b2 := b2 + 1;;
17             }
18
19             # body of Equal
20             call B(in x, inout y);
21             if (x=y)
```

```

22         return(1);
23     else
24         return(0);
25 }
26
27 function AddLoop(in x, in y, in z)
28 {
29     declare i,j,k;
30     declare sum;
31     i := 1;
32     j := 1;
33     k := 1;
34     sum := 0;
35     while (i<=x)
36     {
37         while (j<=y)
38         {
39             while (k<=z)
40             {
41                 sum := sum + 1;
42                 k := k + 1;
43             };
44             j := j + 1;
45         };
46         i := i + 1;
47     };
48     return(sum);
49 }
50
51 # body of main #
52 x := 5;
53 y := 10;
54 a := Equal(in x, inout y);
55 print(a);
56 a := AddLoop(in x, in x, in x);
57 print(a);
58 }.
```

Ακολουθούν οι ενέργειες της μεταγλώττισης σε βήματα. Αρχικά εισάγεται πληροφορία στον πίνακα συμβόλων:

Αναγνωρίστηκε στη γραμμή 1 η λεκτική μονάδα: `program`  
 Αναγνωρίστηκε στη γραμμή 1 η λεκτική μονάδα: `appendix`  
 Αναγνωρίστηκε στη γραμμή 2 η λεκτική μονάδα: {  
 Ένα νέο επίπεδο - με αριθμό 0 - προστέθηκε στον πίνακα συμβόλων  
 Αναγνωρίστηκε στη γραμμή 3 η λεκτική μονάδα: `declare`  
 Αναγνωρίστηκε στη γραμμή 3 η λεκτική μονάδα: a  
 Η οντότητα a προστέθηκε στον πίνακα συμβόλων στη θέση 12  
 Αναγνωρίστηκε στη γραμμή 3 η λεκτική μονάδα: ,  
 Αναγνωρίστηκε στη γραμμή 3 η λεκτική μονάδα: x  
 Η οντότητα x προστέθηκε στον πίνακα συμβόλων στη θέση 16  
 Αναγνωρίστηκε στη γραμμή 3 η λεκτική μονάδα: ,  
 Αναγνωρίστηκε στη γραμμή 3 η λεκτική μονάδα: y

Η οντότητα γ προστέθηκε στον πίνακα συμβόλων στη θέση 20  
 Αναγνωρίστηκε στη γραμμή 3 η λεκτική μονάδα: ;  
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: function  
 Ένα νέο επίπεδο - με αριθμό 1 - προστέθηκε στον πίνακα συμβόλων  
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: Equal  
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: (   
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: in  
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: x  
 Η οντότητα x προστέθηκε στον πίνακα συμβόλων στη θέση 12  
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: ,  
 Η παράμετρος x προστέθηκε στη λίστα των παραμέτρων ως παράμετρος in  
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: inout  
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: y  
 Η οντότητα γ προστέθηκε στον πίνακα συμβόλων στη θέση 16  
 Αναγνωρίστηκε στη γραμμή 5 η λεκτική μονάδα: )  
 Η συνάρτηση/διαδικασία Equal προστέθηκε στον πίνακα συμβόλων στο τρέχον επίπεδο  
 Αναγνωρίστηκε στη γραμμή 6 η λεκτική μονάδα: {  
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: procedure  
 Ένα νέο επίπεδο - με αριθμό 2 - προστέθηκε στον πίνακα συμβόλων  
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: B  
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: (   
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: in  
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: b1  
 Η οντότητα b1 προστέθηκε στον πίνακα συμβόλων στη θέση 12  
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: ,  
 Η παράμετρος b1 προστέθηκε στη λίστα των παραμέτρων ως παράμετρος in  
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: inout  
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: b2  
 Η οντότητα b2 προστέθηκε στον πίνακα συμβόλων στη θέση 16  
 Αναγνωρίστηκε στη γραμμή 7 η λεκτική μονάδα: )  
 Η συνάρτηση/διαδικασία B προστέθηκε στον πίνακα συμβόλων στο τρέχον επίπεδο

Ακολουθεί η δημιουργία ενδιάμεσου κώδικα για την B:

Αναγνωρίστηκε στη γραμμή 8 η λεκτική μονάδα: {  
 Αναγνωρίστηκε στη γραμμή 9 η λεκτική μονάδα: if  
 Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:  
 1: begin\_block, B, \_, \_  
 Αναγνωρίστηκε στη γραμμή 9 η λεκτική μονάδα: (   
 Αναγνωρίστηκε στη γραμμή 9 η λεκτική μονάδα: b2  
 Αναγνωρίστηκε στη γραμμή 9 η λεκτική μονάδα: >  
 Αναγνωρίστηκε στη γραμμή 9 η λεκτική μονάδα: b1  
 Αναγνωρίστηκε στη γραμμή 9 η λεκτική μονάδα: )  
 Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:  
 2: >, b2, b1, \_  
 Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:  
 3: jump, \_, \_, #  
 Αναγνωρίστηκε στη γραμμή 10 η λεκτική μονάδα: while  
 Το άλμα στην ετικέτα 2 συμπληρώθηκε με την ετικέτα: 4  
 2: >, b2, b1, 4  
 Αναγνωρίστηκε στη γραμμή 10 η λεκτική μονάδα: (   
 Αναγνωρίστηκε στη γραμμή 10 η λεκτική μονάδα: b2  
 Αναγνωρίστηκε στη γραμμή 10 η λεκτική μονάδα: >

Αναγνωρίστηκε στη γραμμή 10 η λεκτική μονάδα: b1

Αναγνωρίστηκε στη γραμμή 10 η λεκτική μονάδα: )

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

4: >, b2, b1, \_

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

5: jump, \_, \_, #

Αναγνωρίστηκε στη γραμμή 11 η λεκτική μονάδα: b2

Το άλμα στην ετικέτα 4 συμπληρώθηκε με την ετικέτα: 6

4: >, b2, b1, 6

Αναγνωρίστηκε στη γραμμή 11 η λεκτική μονάδα: :=

Αναγνωρίστηκε στη γραμμή 11 η λεκτική μονάδα: b2

Αναγνωρίστηκε στη γραμμή 11 η λεκτική μονάδα: -

Αναγνωρίστηκε στη γραμμή 11 η λεκτική μονάδα: 1

Αναγνωρίστηκε στη γραμμή 11 η λεκτική μονάδα: ;

Η οντότητα T\_1 προστέθηκε στον πίνακα συμβόλων στη θέση 20

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

6: -, b2, 1, T\_1

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

7: :=, T\_1, \_, b2

Αναγνωρίστηκε στη γραμμή 11 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

8: jump, \_, \_, 4

Το άλμα στην ετικέτα 5 συμπληρώθηκε με την ετικέτα: 9

5: jump, \_, \_, 9

Αναγνωρίστηκε στη γραμμή 12 η λεκτική μονάδα: else

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

9: jump, \_, \_, #

Το άλμα στην ετικέτα 3 συμπληρώθηκε με την ετικέτα: 10

3: jump, \_, \_, 10

Αναγνωρίστηκε στη γραμμή 13 η λεκτική μονάδα: while

Αναγνωρίστηκε στη γραμμή 13 η λεκτική μονάδα: (

Αναγνωρίστηκε στη γραμμή 13 η λεκτική μονάδα: b1

Αναγνωρίστηκε στη γραμμή 13 η λεκτική μονάδα: >

Αναγνωρίστηκε στη γραμμή 13 η λεκτική μονάδα: b2

Αναγνωρίστηκε στη γραμμή 13 η λεκτική μονάδα: )

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

10: >, b1, b2, \_

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

11: jump, \_, \_, #

Αναγνωρίστηκε στη γραμμή 14 η λεκτική μονάδα: b2

Το άλμα στην ετικέτα 10 συμπληρώθηκε με την ετικέτα: 12

10: >, b1, b2, 12

Αναγνωρίστηκε στη γραμμή 14 η λεκτική μονάδα: :=

Αναγνωρίστηκε στη γραμμή 14 η λεκτική μονάδα: b2

Αναγνωρίστηκε στη γραμμή 14 η λεκτική μονάδα: +

Αναγνωρίστηκε στη γραμμή 14 η λεκτική μονάδα: 1

Αναγνωρίστηκε στη γραμμή 14 η λεκτική μονάδα: ;

Η οντότητα T\_2 προστέθηκε στον πίνακα συμβόλων στη θέση 24

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

12: +, b2, 1, T\_2

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

13: :=, T\_2, \_, b2

Αναγνωρίστηκε στη γραμμή 14 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
14: jump, _, _, 10
```

Το áλμα στην ετικέτα 11 συμπληρώθηκε με την ετικέτα: 15

```
11: jump, _, _, 15
```

Αναγνωρίστηκε στη γραμμή 14 η λεκτική μονάδα: ;

Το áλμα στην ετικέτα 9 συμπληρώθηκε με την ετικέτα: 15

```
9: jump, _, _, 15
```

Αναγνωρίστηκε στη γραμμή 15 η λεκτική μονάδα: }

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
15: end_block, B, _, _
```

Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: call

Το startQuad της διαδικασίας B ορίστηκε σε: 1

Το frameLength του υποπρογράμματος B ορίστηκε σε: 28

Ο πίνακας συμβόλων, με το πέρας της μετάφρασης για το B, είναι ο ακόλουθος:

```
-- level 0:
Variable: a :: offset: 12
Variable: x :: offset: 16
Variable: y :: offset: 20
Function Equal :: start_quad: None, frame_length: None,
    parameters: ['Parameter: x :: offset: 12,
      mode: in', 'Parameter: y :: offset: 16, mode: inout']
-- level 1:
Parameter: x :: offset: 12, mode: in
Parameter: y :: offset: 16, mode: inout
Procedure B :: start_quad: 1, frame_length: 28,
    parameters: ['Parameter: b1 :: offset: 12, mode: in',
      'Parameter: b2 :: offset: 16, mode: inout']
-- level 2:
Parameter: b1 :: offset: 12, mode: in
Parameter: b2 :: offset: 16, mode: inout
Temporary Variable: T_1 :: offset: 20
Temporary Variable: T_2 :: offset: 24
```

Στη συνέχεια δημιουργείται ο τελικός κώδικας για την B:

```
.data
str_nl: .asciz "\n"
.text
L0:
# 0: jump, _, _, main
    j Lmain
L1:
# 1: begin_block, B, _, _
    sw ra,(sp)
L2:
# 2: >, b2, b1, 4
    lw t0,-16(sp)
    lw t1,(t0)
    lw t2,-12(sp)
    bgt t1,t2,4
```

```

L3:
#   3: jump, _, _, 10
      j L10

L4:
#   4: >, b2, b1, 6
      lw t0,-16(sp)
      lw t1,(t0)
      lw t2,-12(sp)
      bgt t1,t2,6

L5:
#   5: jump, _, _, 9
      j L9

L6:
#   6: -, b2, 1, T_1
      lw t0,-16(sp)
      lw t1,(t0)
      li t2,1
      sub t1,t2,t1
      sw t1,-20(sp)

L7:
#   7: :=, T_1, _, b2
      lw t1,-20(sp)
      lw t0,-16(sp)
      sw t1,(t0)

L8:
#   8: jump, _, _, 4
      j L4

L9:
#   9: jump, _, _, 15
      j L15

L10:
# 10: >, b1, b2, 12
      lw t1,-12(sp)
      lw t0,-16(sp)
      lw t2,(t0)
      bgt t1,t2,12

L11:
# 11: jump, _, _, 15
      j L15

L12:
# 12: +, b2, 1, T_2
      lw t0,-16(sp)
      lw t1,(t0)
      li t2,1
      add t1,t2,t1
      sw t1,-24(sp)

L13:
# 13: :=, T_2, _, b2
      lw t1,-24(sp)
      lw t0,-16(sp)
      sw t1,(t0)

L14:

```

```
# 14: jump, _, _, 10
    j L10
L15:
# 15: end_block, B, _, _
    lw ra,(sp)
    jr ra
```

Το τελευταίο επίπεδο - με αριθμό 2 - αφαιρέθηκε από τον πίνακα συμβόλων.

Ο πίνακας συμβόλων, τώρα, είναι ο ακόλουθος:

```
--- level 0:
Variable: a :: offset: 12
Variable: x :: offset: 16
Variable: y :: offset: 20
Function Equal :: start_quad: None, frame_length: None,
    parameters: ['Parameter: x :: offset: 12, mode: in',
                  'Parameter: y :: offset: 16, mode: inout']
--- level 1:
Parameter: x :: offset: 12, mode: in
Parameter: y :: offset: 16, mode: inout
Procedure B :: start_quad: 1, frame_length: 28,
    parameters: ['Parameter: b1 :: offset: 12, mode: in',
                  'Parameter: b2 :: offset: 16, mode: inout']
```

Δημιουργείται ο ενδιάμεσος κώδικας για την Equal:

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
16: begin_block, Equal, _, _
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: B
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: (
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: in
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: x
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: ,
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: inout
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: y
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: )
Αναγνωρίστηκε στη γραμμή 18 η λεκτική μονάδα: ;
Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:
```

```
17: par, x, in, B
```

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
18: par, y, inout, B
```

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
19: call, B, _, _
```

```
Αναγνωρίστηκε στη γραμμή 19 η λεκτική μονάδα: if
Αναγνωρίστηκε στη γραμμή 19 η λεκτική μονάδα: (
Αναγνωρίστηκε στη γραμμή 19 η λεκτική μονάδα: x
Αναγνωρίστηκε στη γραμμή 19 η λεκτική μονάδα: =
Αναγνωρίστηκε στη γραμμή 19 η λεκτική μονάδα: y
Αναγνωρίστηκε στη γραμμή 19 η λεκτική μονάδα: )
Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:
```

```
20: =, x, y, _
```

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
21: jump, _, _, #
```

Αναγνωρίστηκε στη γραμμή 20 η λεκτική μονάδα: return

Το άλμα στην ετικέτα 20 συμπληρώθηκε με την ετικέτα: 22

20: =, x, y, 22

Αναγνωρίστηκε στη γραμμή 20 η λεκτική μονάδα: (

Αναγνωρίστηκε στη γραμμή 20 η λεκτική μονάδα: 1

Αναγνωρίστηκε στη γραμμή 20 η λεκτική μονάδα: )

Αναγνωρίστηκε στη γραμμή 20 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

22: return, 1, \_, \_

Αναγνωρίστηκε στη γραμμή 21 η λεκτική μονάδα: else

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

23: jump, \_, \_, #

Το άλμα στην ετικέτα 21 συμπληρώθηκε με την ετικέτα: 24

21: jump, \_, \_, 24

Αναγνωρίστηκε στη γραμμή 22 η λεκτική μονάδα: return

Αναγνωρίστηκε στη γραμμή 22 η λεκτική μονάδα: (

Αναγνωρίστηκε στη γραμμή 22 η λεκτική μονάδα: 0

Αναγνωρίστηκε στη γραμμή 22 η λεκτική μονάδα: )

Αναγνωρίστηκε στη γραμμή 22 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

24: return, 0, \_, \_

Αναγνωρίστηκε στη γραμμή 22 η λεκτική μονάδα: ;

Το άλμα στην ετικέτα 23 συμπληρώθηκε με την ετικέτα: 25

23: jump, \_, \_, 25

Αναγνωρίστηκε στη γραμμή 23 η λεκτική μονάδα: }

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

25: end\_block, Equal, \_, \_

Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: function

To startQuad της συνάρτησης Equal ορίστηκε σε: 16

To frameLength του υποπρογράμματος Equal ορίστηκε σε: 20

Ο πίνακας συμβόλων, με το πέρας της μετάφρασης για την Equal, είναι ο ακόλουθος:

```
--- level 0:
Variable: a :::: offset: 12
Variable: x :::: offset: 16
Variable: y :::: offset: 20
Function Equal :::: start_quad: 16, frame_length: 20,
    parameters: ['Parameter: x :::: offset: 12, mode: in',
                  'Parameter: y :::: offset: 16, mode: inout']
--- level 1:
Parameter: x :::: offset: 12, mode: in
Parameter: y :::: offset: 16, mode: inout
Procedure B :::: start_quad: 1, frame_length: 28,
    parameters: ['Parameter: b1 :::: offset: 12, mode: in',
                  'Parameter: b2 :::: offset: 16, mode: inout']
```

Δημιουργείται ο τελικός κώδικας για την Equal:

L16:

```
# 16: begin_block, Equal, _, _
      sw ra,(sp)
```

L17:

```
# 17: par, x, in, B
      addi fp,sp,28
```

```

        lw t0,-12(sp)
        sw t0,-12(fp)
L18:
# 18: par, y, inout, B
        lw t0,-16(sp)
        sw t0,-16(fp)

L19:
# 19: call, B, _, _
        sw sp,-4(fp)
        addi sp,sp,28
        jal L1
        addi sp,sp,-28

L20:
# 20: =, x, y, 22
        lw t1,-12(sp)
        lw t0,-16(sp)
        lw t2,(t0)
        beq t1,t2,22

L21:
# 21: jump, _, _, 24
        j L24

L22:
# 22: return, 1, _, _
        li t1,1
        lw t0,-8(sp)
        sw t1,(t0)

L23:
# 23: jump, _, _, 25
        j L25

L24:
# 24: return, 0, _, _
        li t1,0
        lw t0,-8(sp)
        sw t1,(t0)

L25:
# 25: end_block, Equal, _, _
        lw ra,(sp)
        jr ra

```

Το τελευταίο επίπεδο - με αριθμό 1 - αφαιρέθηκε από τον πίνακα συμβόλων.

Και ο πίνακας συμβόλων τώρα γίνεται:

```

--- level 0:
Variable: a :::: offset: 12
Variable: x :::: offset: 16
Variable: y :::: offset: 20
Function Equal :::: start_quad: 16, frame_length: 20,
    parameters: ['Parameter: x :::: offset: 12, mode: in',
    'Parameter: y :::: offset: 16, mode: inout']

```

Πριν την παραγωγή του ενδιάμεσου κώδικα, πληροφορία για την AddLoop προστίθεται στον πίνακα συμβόλων:

Ένα νέο επίπεδο - με αριθμό 1 - προστέθηκε στον πίνακα συμβόλων

Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: AddLoop  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: (   
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: in  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: x  
 Η οντότητα x προστέθηκε στον πίνακα συμβόλων στη θέση 12  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: ,  
 Η παράμετρος x προστέθηκε στη λίστα των παραμέτρων ως παράμετρος in  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: in  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: y  
 Η οντότητα y προστέθηκε στον πίνακα συμβόλων στη θέση 16  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: ,  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: in  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: z  
 Η οντότητα z προστέθηκε στον πίνακα συμβόλων στη θέση 20  
 Αναγνωρίστηκε στη γραμμή 25 η λεκτική μονάδα: )  
 Η συνάρτηση/διαδικασία AddLoop προστέθηκε στον πίνακα συμβόλων στο τρέχον επίπεδο  
 Αναγνωρίστηκε στη γραμμή 26 η λεκτική μονάδα: {  
 Αναγνωρίστηκε στη γραμμή 27 η λεκτική μονάδα: declare  
 Αναγνωρίστηκε στη γραμμή 27 η λεκτική μονάδα: i  
 Η οντότητα i προστέθηκε στον πίνακα συμβόλων στη θέση 24  
 Αναγνωρίστηκε στη γραμμή 27 η λεκτική μονάδα: ,  
 Αναγνωρίστηκε στη γραμμή 27 η λεκτική μονάδα: j  
 Η οντότητα j προστέθηκε στον πίνακα συμβόλων στη θέση 28  
 Αναγνωρίστηκε στη γραμμή 27 η λεκτική μονάδα: ,  
 Αναγνωρίστηκε στη γραμμή 27 η λεκτική μονάδα: k  
 Η οντότητα k προστέθηκε στον πίνακα συμβόλων στη θέση 32  
 Αναγνωρίστηκε στη γραμμή 27 η λεκτική μονάδα: ;  
 Αναγνωρίστηκε στη γραμμή 28 η λεκτική μονάδα: declare  
 Αναγνωρίστηκε στη γραμμή 28 η λεκτική μονάδα: sum  
 Η οντότητα sum προστέθηκε στον πίνακα συμβόλων στη θέση 36  
 Αναγνωρίστηκε στη γραμμή 28 η λεκτική μονάδα: ;

Στη συνέχεια παράγεται ο ενδιάμεσος κώδικας για την AddLoop:

Αναγνωρίστηκε στη γραμμή 29 η λεκτική μονάδα: i  
 Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:  
 26: begin\_block, AddLoop, \_, \_  
 Αναγνωρίστηκε στη γραμμή 29 η λεκτική μονάδα: :=  
 Αναγνωρίστηκε στη γραμμή 29 η λεκτική μονάδα: 1  
 Αναγνωρίστηκε στη γραμμή 29 η λεκτική μονάδα: ;  
 Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:  
 27: :=, 1, \_, i  
 Αναγνωρίστηκε στη γραμμή 30 η λεκτική μονάδα: j  
 Αναγνωρίστηκε στη γραμμή 30 η λεκτική μονάδα: :=  
 Αναγνωρίστηκε στη γραμμή 30 η λεκτική μονάδα: 1  
 Αναγνωρίστηκε στη γραμμή 30 η λεκτική μονάδα: ;  
 Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:  
 28: :=, 1, \_, j  
 Αναγνωρίστηκε στη γραμμή 31 η λεκτική μονάδα: k  
 Αναγνωρίστηκε στη γραμμή 31 η λεκτική μονάδα: :=  
 Αναγνωρίστηκε στη γραμμή 31 η λεκτική μονάδα: 1  
 Αναγνωρίστηκε στη γραμμή 31 η λεκτική μονάδα: ;  
 Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

29: `:=, 1, _, k`

Αναγνωρίστηκε στη γραμμή 32 η λεκτική μονάδα: `sum`

Αναγνωρίστηκε στη γραμμή 32 η λεκτική μονάδα: `:=`

Αναγνωρίστηκε στη γραμμή 32 η λεκτική μονάδα: `0`

Αναγνωρίστηκε στη γραμμή 32 η λεκτική μονάδα: `;`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

30: `:=, 0, _, sum`

Αναγνωρίστηκε στη γραμμή 33 η λεκτική μονάδα: `while`

Αναγνωρίστηκε στη γραμμή 33 η λεκτική μονάδα: `(`

Αναγνωρίστηκε στη γραμμή 33 η λεκτική μονάδα: `i`

Αναγνωρίστηκε στη γραμμή 33 η λεκτική μονάδα: `<=`

Αναγνωρίστηκε στη γραμμή 33 η λεκτική μονάδα: `x`

Αναγνωρίστηκε στη γραμμή 33 η λεκτική μονάδα: `)`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

31: `<, i, x, _`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

32: `jump, _, _, #`

Αναγνωρίστηκε στη γραμμή 34 η λεκτική μονάδα: `{`

Το άλμα στην ετικέτα 31 συμπληρώθηκε με την ετικέτα: 33

31: `<, i, x, 33`

Αναγνωρίστηκε στη γραμμή 35 η λεκτική μονάδα: `while`

Αναγνωρίστηκε στη γραμμή 35 η λεκτική μονάδα: `(`

Αναγνωρίστηκε στη γραμμή 35 η λεκτική μονάδα: `j`

Αναγνωρίστηκε στη γραμμή 35 η λεκτική μονάδα: `<=`

Αναγνωρίστηκε στη γραμμή 35 η λεκτική μονάδα: `y`

Αναγνωρίστηκε στη γραμμή 35 η λεκτική μονάδα: `)`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

33: `<, j, y, _`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

34: `jump, _, _, #`

Αναγνωρίστηκε στη γραμμή 36 η λεκτική μονάδα: `{`

Το άλμα στην ετικέτα 33 συμπληρώθηκε με την ετικέτα: 35

33: `<, j, y, 35`

Αναγνωρίστηκε στη γραμμή 37 η λεκτική μονάδα: `while`

Αναγνωρίστηκε στη γραμμή 37 η λεκτική μονάδα: `(`

Αναγνωρίστηκε στη γραμμή 37 η λεκτική μονάδα: `k`

Αναγνωρίστηκε στη γραμμή 37 η λεκτική μονάδα: `<=`

Αναγνωρίστηκε στη γραμμή 37 η λεκτική μονάδα: `z`

Αναγνωρίστηκε στη γραμμή 37 η λεκτική μονάδα: `)`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

35: `<, k, z, _`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

36: `jump, _, _, #`

Αναγνωρίστηκε στη γραμμή 38 η λεκτική μονάδα: `{`

Το άλμα στην ετικέτα 35 συμπληρώθηκε με την ετικέτα: 37

35: `<, k, z, 37`

Αναγνωρίστηκε στη γραμμή 39 η λεκτική μονάδα: `sum`

Αναγνωρίστηκε στη γραμμή 39 η λεκτική μονάδα: `:=`

Αναγνωρίστηκε στη γραμμή 39 η λεκτική μονάδα: `sum`

Αναγνωρίστηκε στη γραμμή 39 η λεκτική μονάδα: `+`

Αναγνωρίστηκε στη γραμμή 39 η λεκτική μονάδα: `1`

Αναγνωρίστηκε στη γραμμή 39 η λεκτική μονάδα: `;`

Η οντότητα  $T_3$  προστέθηκε στον πίνακα συμβόλων στη θέση 40

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

37: +, sum, 1, T\_3

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

38: :=, T\_3, \_, sum

Αναγνωρίστηκε στη γραμμή 40 η λεκτική μονάδα: k

Αναγνωρίστηκε στη γραμμή 40 η λεκτική μονάδα: :=

Αναγνωρίστηκε στη γραμμή 40 η λεκτική μονάδα: k

Αναγνωρίστηκε στη γραμμή 40 η λεκτική μονάδα: +

Αναγνωρίστηκε στη γραμμή 40 η λεκτική μονάδα: 1

Αναγνωρίστηκε στη γραμμή 40 η λεκτική μονάδα: ;

Η οντότητα  $T_4$  προστέθηκε στον πίνακα συμβόλων στη θέση 44

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

39: +, k, 1, T\_4

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

40: :=, T\_4, \_, k

Αναγνωρίστηκε στη γραμμή 41 η λεκτική μονάδα: }

Αναγνωρίστηκε στη γραμμή 41 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

41: jump, \_, \_, 35

Το άλμα στην ετικέτα 36 συμπληρώθηκε με την ετικέτα: 42

36: jump, \_, \_, 42

Αναγνωρίστηκε στη γραμμή 42 η λεκτική μονάδα: j

Αναγνωρίστηκε στη γραμμή 42 η λεκτική μονάδα: :=

Αναγνωρίστηκε στη γραμμή 42 η λεκτική μονάδα: j

Αναγνωρίστηκε στη γραμμή 42 η λεκτική μονάδα: +

Αναγνωρίστηκε στη γραμμή 42 η λεκτική μονάδα: 1

Αναγνωρίστηκε στη γραμμή 42 η λεκτική μονάδα: ;

Η οντότητα  $T_5$  προστέθηκε στον πίνακα συμβόλων στη θέση 48

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

42: +, j, 1, T\_5

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

43: :=, T\_5, \_, j

Αναγνωρίστηκε στη γραμμή 43 η λεκτική μονάδα: }

Αναγνωρίστηκε στη γραμμή 43 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

44: jump, \_, \_, 33

Το άλμα στην ετικέτα 34 συμπληρώθηκε με την ετικέτα: 45

34: jump, \_, \_, 45

Αναγνωρίστηκε στη γραμμή 44 η λεκτική μονάδα: i

Αναγνωρίστηκε στη γραμμή 44 η λεκτική μονάδα: :=

Αναγνωρίστηκε στη γραμμή 44 η λεκτική μονάδα: i

Αναγνωρίστηκε στη γραμμή 44 η λεκτική μονάδα: +

Αναγνωρίστηκε στη γραμμή 44 η λεκτική μονάδα: 1

Αναγνωρίστηκε στη γραμμή 44 η λεκτική μονάδα: ;

Η οντότητα  $T_6$  προστέθηκε στον πίνακα συμβόλων στη θέση 52

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

45: +, i, 1, T\_6

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

46: :=, T\_6, \_, i

Αναγνωρίστηκε στη γραμμή 45 η λεκτική μονάδα: }

Αναγνωρίστηκε στη γραμμή 45 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

47: `jump`, \_, \_, 31

Το áλμα στην ετικέτα 32 συμπληρώθηκε με την ετικέτα: 48

32: `jump`, \_, \_, 48

Αναγνωρίστηκε στη γραμμή 46 η λεκτική μονάδα: `return`

Αναγνωρίστηκε στη γραμμή 46 η λεκτική μονάδα: `(`

Αναγνωρίστηκε στη γραμμή 46 η λεκτική μονάδα: `sum`

Αναγνωρίστηκε στη γραμμή 46 η λεκτική μονάδα: `)`

Αναγνωρίστηκε στη γραμμή 46 η λεκτική μονάδα: `;`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

48: `return`, `sum`, \_, \_

Αναγνωρίστηκε στη γραμμή 47 η λεκτική μονάδα: `}`

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

49: `end_block`, `AddLoop`, \_, \_

Αναγνωρίστηκε στη γραμμή 51 η λεκτική μονάδα: `x`

To `startQuad` της συνάρτησης `AddLoop` ορίστηκε σε: 26

To `frameLength` του υποπρογράμματος `AddLoop` ορίστηκε σε: 56

Ο πίνακας συμβόλων στο τέλος της παραγωγής του ενδιάμεσου κώδικα για την `AddLoop`, είναι ο ακόλουθος:

```
-- level 0:
Variable: a :: offset: 12
Variable: x :: offset: 16
Variable: y :: offset: 20
Function Equal :: start_quad: 16, frame_length: 20,
    parameters: ['Parameter: x :: offset: 12, mode: in',
                  'Parameter: y :: offset: 16, mode: inout']
Function AddLoop :: start_quad: 26, frame_length: 56,
    parameters: ['Parameter: x :: offset: 12, mode: in',
                  'Parameter: y :: offset: 16, mode: in',
                  'Parameter: z :: offset: 20, mode: in']
-- level 1:
Parameter: x :: offset: 12, mode: in
Parameter: y :: offset: 16, mode: in
Parameter: z :: offset: 20, mode: in
Variable: i :: offset: 24
Variable: j :: offset: 28
Variable: k :: offset: 32
Variable: sum :: offset: 36
Temporary Variable: T_3 :: offset: 40
Temporary Variable: T_4 :: offset: 44
Temporary Variable: T_5 :: offset: 48
Temporary Variable: T_6 :: offset: 52
```

Η παραγωγή του τελικού κώδικα ακολουθεί:

L26:

```
# 26: begin_block, AddLoop, _, _
    sw ra,(sp)
```

L27:

```

# 27: :=, 1, _, i
    li t1,1
    sw t1,-24(sp)

L28:
# 28: :=, 1, _, j
    li t1,1
    sw t1,-28(sp)

L29:
# 29: :=, 1, _, k
    li t1,1
    sw t1,-32(sp)

L30:
# 30: :=, 0, _, sum
    li t1,0
    sw t1,-36(sp)

L31:
# 31: <=, i, x, 33
    lw t1,-24(sp)
    lw t2,-12(sp)
    ble t1,t2,33

L32:
# 32: jump, _, _, 48
    j L48

L33:
# 33: <=, j, y, 35
    lw t1,-28(sp)
    lw t2,-16(sp)
    ble t1,t2,35

L34:
# 34: jump, _, _, 45
    j L45

L35:
# 35: <=, k, z, 37
    lw t1,-32(sp)
    lw t2,-20(sp)
    ble t1,t2,37

L36:
# 36: jump, _, _, 42
    j L42

L37:
# 37: +, sum, 1, T_3
    lw t1,-36(sp)
    li t2,1
    add t1,t2,t1
    sw t1,-40(sp)

L38:
# 38: :=, T_3, _, sum
    lw t1,-40(sp)
    sw t1,-36(sp)

L39:
# 39: +, k, 1, T_4
    lw t1,-32(sp)

```

```

    li t2,1
    add t1,t2,t1
    sw t1,-44(sp)

L40:
# 40: :=, T_4, _, k
    lw t1,-44(sp)
    sw t1,-32(sp)

L41:
# 41: jump, _, _, 35
    j L35

L42:
# 42: +, j, 1, T_5
    lw t1,-28(sp)
    li t2,1
    add t1,t2,t1
    sw t1,-48(sp)

L43:
# 43: :=, T_5, _, j
    lw t1,-48(sp)
    sw t1,-28(sp)

L44:
# 44: jump, _, _, 33
    j L33

L45:
# 45: +, i, 1, T_6
    lw t1,-24(sp)
    li t2,1
    add t1,t2,t1
    sw t1,-52(sp)

L46:
# 46: :=, T_6, _, i
    lw t1,-52(sp)
    sw t1,-24(sp)

L47:
# 47: jump, _, _, 31
    j L31

L48:
# 48: return, sum, _, _
    lw t1,-36(sp)
    lw t0,-8(sp)
    sw t1,(t0)

L49:
# 49: end_block, AddLoop, _, _
    lw ra,(sp)
    jr ra

```

Το τελευταίο επίπεδο - με αριθμό 1 - αφαιρέθηκε από τον πίνακα συμβόλων.

Ο πίνακας συμβόλων μετά την ολοκλήρωση της παράγωγης κώδικα για την AddLoop, είναι ο ακόλουθος:

```

--- level 0:
Variable: a :: offset: 12
Variable: x :: offset: 16

```

```

Variable: y :: offset: 20
Function Equal :: start_quad: 16, frame_length: 20,
    parameters: ['Parameter: x :: offset: 12, mode: in',
                  'Parameter: y :: offset: 16, mode: inout']
Function AddLoop :: start_quad: 26, frame_length: 56,
    parameters: ['Parameter: x :: offset: 12, mode: in',
                  'Parameter: y :: offset: 16, mode: in',
                  'Parameter: z :: offset: 20, mode: in']

```

Παράγεται ενδιάμεσος κώδικας για το κυρίως πρόγραμμα:

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
50: begin_block, main, _, _
```

Αναγνωρίστηκε στη γραμμή 51 η λεκτική μονάδα: :=

Αναγνωρίστηκε στη γραμμή 51 η λεκτική μονάδα: 5

Αναγνωρίστηκε στη γραμμή 51 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
51: :=, 5, _, x
```

Αναγνωρίστηκε στη γραμμή 52 η λεκτική μονάδα: y

Αναγνωρίστηκε στη γραμμή 52 η λεκτική μονάδα: :=

Αναγνωρίστηκε στη γραμμή 52 η λεκτική μονάδα: 10

Αναγνωρίστηκε στη γραμμή 52 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
52: :=, 10, _, y
```

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: a

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: :=

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: Equal

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: (

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: in

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: x

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: ,

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: inout

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: y

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: )

Αναγνωρίστηκε στη γραμμή 53 η λεκτική μονάδα: ;

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
53: par, x, in, Equal
```

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
54: par, y, inout, Equal
```

Η οντότητα T\_7 προστέθηκε στον πίνακα συμβόλων στη θέση 24

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
55: par, T_7, ret, Equal
```

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
56: call, Equal, _, _
```

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
57: :=, T_7, _, a
```

Αναγνωρίστηκε στη γραμμή 54 η λεκτική μονάδα: print

Αναγνωρίστηκε στη γραμμή 54 η λεκτική μονάδα: (

Αναγνωρίστηκε στη γραμμή 54 η λεκτική μονάδα: a

Αναγνωρίστηκε στη γραμμή 54 η λεκτική μονάδα: )

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

```
58: print, a, _, _
```

Αναγνωρίστηκε στη γραμμή 54 η λεκτική μονάδα: ;  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: a  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: :=  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: AddLoop  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: (  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: in  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: x  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: ,  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: in  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: x  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: ,  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: in  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: x  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: )  
 Αναγνωρίστηκε στη γραμμή 55 η λεκτική μονάδα: ;  
 Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

59: par, x, in, AddLoop

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

60: par, x, in, AddLoop

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

61: par, x, in, AddLoop

Η οντότητα T\_8 προστέθηκε στον πίνακα συμβόλων στη θέση 28

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

62: par, T\_8, ret, AddLoop

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

63: call, AddLoop, \_, \_

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

64: :=, T\_8, \_, a

Αναγνωρίστηκε στη γραμμή 56 η λεκτική μονάδα: print

Αναγνωρίστηκε στη γραμμή 56 η λεκτική μονάδα: (

Αναγνωρίστηκε στη γραμμή 56 η λεκτική μονάδα: a

Αναγνωρίστηκε στη γραμμή 56 η λεκτική μονάδα: )

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

65: print, a, \_, \_

Αναγνωρίστηκε στη γραμμή 56 η λεκτική μονάδα: ;

Αναγνωρίστηκε στη γραμμή 57 η λεκτική μονάδα: }

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

66: halt, \_, \_, \_

Δημιουργήθηκε η τετράδα ενδιάμεσου κώδικα:

67: end\_block, main, \_, \_

Αναγνωρίστηκε στη γραμμή 57 η λεκτική μονάδα: .

Ο πίνακας συμβόλων με βάση τον οποίο θα δημιουργηθεί ο τελικός κώδικας είναι:

```
-- level 0:  
Variable: a :: offset: 12  
Variable: x :: offset: 16  
Variable: y :: offset: 20  
Function Equal :: start_quad: 16, frame_length: 20,  
    parameters: ['Parameter: x :: offset: 12, mode: in',  
                 'Parameter: y :: offset: 16, mode: inout']  
Function AddLoop :: start_quad: 26, frame_length: 56,  
    parameters: ['Parameter: x :: offset: 12, mode: in',
```

```

        'Parameter: y :: offset: 16, mode: in',
        'Parameter: z :: offset: 20, mode: in']
Temporary Variable: T_7 :: offset: 24
Temporary Variable: T_8 :: offset: 28

```

Τέλος, παράγεται και ο τελικός κώδικας για το κυρίως πρόγραμμα:

```

L50:
Lmain:
# 50: begin_block, main, _, _
    addi sp,sp,32
    mv gp,sp
L51:
# 51: :=, 5, _, x
    li t1,5
    sw t1,-16(gp)
L52:
# 52: :=, 10, _, y
    li t1,10
    sw t1,-20(gp)
L53:
# 53: par, x, in, Equal
    addi fp,sp,20
    lw t0,-16(gp)
    sw t0,-12(fp)
L54:
# 54: par, y, inout, Equal
    addi t0,sp,-20
    sw t0,-16(fp)
L55:
# 55: par, T_7, ret, Equal
    addi t0,sp,-24
    sw t0,-8(fp)
L56:
# 56: call, Equal, _, _
    sw sp,-4(fp)
    addi sp,sp,20
    jal L16
    addi sp,sp,-20
L57:
# 57: :=, T_7, _, a
    lw t1,-24(gp)
    sw t1,-12(gp)
L58:
# 58: print, a, _, _
    lw a0,-12(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall
L59:
# 59: par, x, in, AddLoop

```

```

        addi fp,sp,56
        lw t0,-16(gp)
        sw t0,-12(fp)

L60:
# 60: par, x, in, AddLoop
        lw t0,-16(gp)
        sw t0,-16(fp)

L61:
# 61: par, x, in, AddLoop
        lw t0,-16(gp)
        sw t0,-20(fp)

L62:
# 62: par, T_8, ret, AddLoop
        addi t0,sp,-28
        sw t0,-8(fp)

L63:
# 63: call, AddLoop, _, _
        sw sp,-4(fp)
        addi sp,sp,56
        jal L26
        addi sp,sp,-56

L64:
# 64: :=, T_8, _, a
        lw t1,-28(gp)
        sw t1,-12(gp)

L65:
# 65: print, a, _, _
        lw a0,-12(gp)
        li a7,1
        ecall
        la a0,str_nl
        li a7,4
        ecall

L66:
# 66: halt, _, _, _
        li a0,0
        li a7,93
        ecall

L67:
# 67: end_block, main, _, _

Το τελευταίο επίπεδο - με αριθμό 0 - αφαιρέθηκε από τον πίνακα συμβόλων.
```

Ολοκληρώθηκε η παραγωγή του κώδικα. Ο ενδιάμεσος κώδικας ολοκληρωμένος φαίνεται παρακάτω:

```

0: jump, _, _, main
1: begin_block, B, _, _
2: >, b2, b1, 4
3: jump, _, _, 10
4: >, b2, b1, 6
5: jump, _, _, 9
6: -, b2, 1, T_1
7: :=, T_1, _, b2
```

```

8: jump, _, _, 4
9: jump, _, _, 15
10: >, b1, b2, 12
11: jump, _, _, 15
12: +, b2, 1, T_2
13: :=, T_2, _, b2
14: jump, _, _, 10
15: end_block, B, _, _
16: begin_block, Equal, _, _
17: par, x, in, B
18: par, y, inout, B
19: call, B, _, _
20: =, x, y, 22
21: jump, _, _, 24
22: return, 1, _, _
23: jump, _, _, 25
24: return, 0, _, _
25: end_block, Equal, _, _
26: begin_block, AddLoop, _, _
27: :=, 1, _, i
28: :=, 1, _, j
29: :=, 1, _, k
30: :=, 0, _, sum
31: <=, i, x, 33
32: jump, _, _, 48
33: <=, j, y, 35
34: jump, _, _, 45
35: <=, k, z, 37
36: jump, _, _, 42
37: +, sum, 1, T_3
38: :=, T_3, _, sum
39: +, k, 1, T_4
40: :=, T_4, _, k
41: jump, _, _, 35
42: +, j, 1, T_5
43: :=, T_5, _, j
44: jump, _, _, 33
45: +, i, 1, T_6
46: :=, T_6, _, i
47: jump, _, _, 31
48: return, sum, _, _
49: end_block, AddLoop, _, _
50: begin_block, main, _, _
51: :=, 5, _, x
52: :=, 10, _, y
53: par, x, in, Equal
54: par, y, inout, Equal
55: par, T_7, ret, Equal
56: call, Equal, _, _
57: :=, T_7, _, a
58: print, a, _, _
59: par, x, in, AddLoop

```

```

60: par, x, in, AddLoop
61: par, x, in, AddLoop
62: par, T_8, ret, AddLoop
63: call, AddLoop, _, _
64: :=, T_8, _, a
65: print, a, _, _
66: halt, _, _, _
67: end_block, main, _, _

```

και ο τελικός κώδικας:

```

.data
str_nl: .asciz "\n"
.text

L0:
# 0: jump, _, _, main
j Lmain

L1:
# 1: begin_block, B, _, _
sw ra,(sp)

L2:
# 2: >, b2, b1, 4
lw t0,-16(sp)
lw t1,(t0)
lw t2,-12(sp)
bgt t1,t2,4

L3:
# 3: jump, _, _, 10
j L10

L4:
# 4: >, b2, b1, 6
lw t0,-16(sp)
lw t1,(t0)
lw t2,-12(sp)
bgt t1,t2,6

L5:
# 5: jump, _, _, 9
j L9

L6:
# 6: -, b2, 1, T_1
lw t0,-16(sp)
lw t1,(t0)
li t2,1
sub t1,t2,t1
sw t1,-20(sp)

```

L7:

```
# 7: :=, T_1, _, b2
    lw t1,-20(sp)
    lw t0,-16(sp)
    sw t1,(t0)
```

L8:

```
# 8: jump, _, _, 4
    j L4
```

L9:

```
# 9: jump, _, _, 15
    j L15
```

L10:

```
# 10: >, b1, b2, 12
      lw t1,-12(sp)
      lw t0,-16(sp)
      lw t2,(t0)
      bgt t1,t2,12
```

L11:

```
# 11: jump, _, _, 15
    j L15
```

L12:

```
# 12: +, b2, 1, T_2
      lw t0,-16(sp)
      lw t1,(t0)
      li t2,1
      add t1,t2,t1
      sw t1,-24(sp)
```

L13:

```
# 13: :=, T_2, _, b2
      lw t1,-24(sp)
      lw t0,-16(sp)
      sw t1,(t0)
```

L14:

```
# 14: jump, _, _, 10
    j L10
```

L15:

```
# 15: end_block, B, _, _
    lw ra,(sp)
    jr ra
```

L16:

```
# 16: begin_block, Equal, _, _
    sw ra,(sp)
```

```

L17:
# 17: par, x, in, B
    addi fp,sp,28
    lw t0,-12(sp)
    sw t0,-12(fp)

L18:
# 18: par, y, inout, B
    lw t0,-16(sp)
    sw t0,-16(fp)

L19:
# 19: call, B, _, _
    sw sp,-4(fp)
    addi sp,sp,28
    jal L1
    addi sp,sp,-28

L20:
# 20: =, x, y, 22
    lw t1,-12(sp)
    lw t0,-16(sp)
    lw t2,(t0)
    beq t1,t2,22

L21:
# 21: jump, _, _, 24
    j L24

L22:
# 22: return, 1, _, _
    li t1,1
    lw t0,-8(sp)
    sw t1,(t0)

L23:
# 23: jump, _, _, 25
    j L25

L24:
# 24: return, 0, _, _
    li t1,0
    lw t0,-8(sp)
    sw t1,(t0)

L25:
# 25: end_block, Equal, _, _
    lw ra,(sp)
    jr ra

L26:
# 26: begin_block, AddLoop, _, _

```

```
sw ra,(sp)
```

L27:

```
# 27: :=, 1, _, i
    li t1,1
    sw t1,-24(sp)
```

L28:

```
# 28: :=, 1, _, j
    li t1,1
    sw t1,-28(sp)
```

L29:

```
# 29: :=, 1, _, k
    li t1,1
    sw t1,-32(sp)
```

L30:

```
# 30: :=, 0, _, sum
    li t1,0
    sw t1,-36(sp)
```

L31:

```
# 31: <=, i, x, 33
    lw t1,-24(sp)
    lw t2,-12(sp)
    ble t1,t2,33
```

L32:

```
# 32: jump, _, _, 48
      j L48
```

L33:

```
# 33: <=, j, y, 35
    lw t1,-28(sp)
    lw t2,-16(sp)
    ble t1,t2,35
```

L34:

```
# 34: jump, _, _, 45
      j L45
```

L35:

```
# 35: <=, k, z, 37
    lw t1,-32(sp)
    lw t2,-20(sp)
    ble t1,t2,37
```

L36:

```
# 36: jump, _, _, 42
      j L42
```

L37:

```
# 37: +, sum, 1, T_3
    lw t1,-36(sp)
    li t2,1
    add t1,t2,t1
    sw t1,-40(sp)
```

L38:

```
# 38: :=, T_3, _, sum
    lw t1,-40(sp)
    sw t1,-36(sp)
```

L39:

```
# 39: +, k, 1, T_4
    lw t1,-32(sp)
    li t2,1
    add t1,t2,t1
    sw t1,-44(sp)
```

L40:

```
# 40: :=, T_4, _, k
    lw t1,-44(sp)
    sw t1,-32(sp)
```

L41:

```
# 41: jump, _, _, 35
    j L35
```

L42:

```
# 42: +, j, 1, T_5
    lw t1,-28(sp)
    li t2,1
    add t1,t2,t1
    sw t1,-48(sp)
```

L43:

```
# 43: :=, T_5, _, j
    lw t1,-48(sp)
    sw t1,-28(sp)
```

L44:

```
# 44: jump, _, _, 33
    j L33
```

L45:

```
# 45: +, i, 1, T_6
    lw t1,-24(sp)
    li t2,1
    add t1,t2,t1
    sw t1,-52(sp)
```

L46:

```

# 46: :=, T_6, _, i
    lw t1,-52(sp)
    sw t1,-24(sp)

L47:
# 47: jump, _, _, 31
    j L31

L48:
# 48: return, sum, _, _
    lw t1,-36(sp)
    lw t0,-8(sp)
    sw t1,(t0)

L49:
# 49: end_block, AddLoop, _, _
    lw ra,(sp)
    jr ra

L50:
Lmain:
# 50: begin_block, main, _, _
    addi sp,sp,32
    mv gp,sp

L51:
# 51: :=, 5, _, x
    li t1,5
    sw t1,-16(gp)

L52:
# 52: :=, 10, _, y
    li t1,10
    sw t1,-20(gp)

L53:
# 53: par, x, in, Equal
    addi fp,sp,20
    lw t0,-16(gp)
    sw t0,-12(fp)

L54:
# 54: par, y, inout, Equal
    addi t0,sp,-20
    sw t0,-16(fp)

L55:
# 55: par, T_7, ret, Equal
    addi t0,sp,-24
    sw t0,-8(fp)

L56:

```

```

# 56: call, Equal, _, _
    sw sp,-4(fp)
    addi sp,sp,20
    jal L16
    addi sp,sp,-20

L57:
# 57: :=, T_7, _, a
    lw t1,-24(gp)
    sw t1,-12(gp)

L58:
# 58: print, a, _, _
    lw a0,-12(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall

L59:
# 59: par, x, in, AddLoop
    addi fp,sp,56
    lw t0,-16(gp)
    sw t0,-12(fp)

L60:
# 60: par, x, in, AddLoop
    lw t0,-16(gp)
    sw t0,-16(fp)

L61:
# 61: par, x, in, AddLoop
    lw t0,-16(gp)
    sw t0,-20(fp)

L62:
# 62: par, T_8, ret, AddLoop
    addi t0,sp,-28
    sw t0,-8(fp)

L63:
# 63: call, AddLoop, _, _
    sw sp,-4(fp)
    addi sp,sp,56
    jal L26
    addi sp,sp,-56

L64:
# 64: :=, T_8, _, a
    lw t1,-28(gp)
    sw t1,-12(gp)

```

L65:

```
# 65: print, a, _, _
    lw a0,-12(gp)
    li a7,1
    ecall
    la a0,str_nl
    li a7,4
    ecall
```

L66:

```
# 66: halt, _, _, _
    li a0,0
    li a7,93
    ecall
```

L67:

```
# 67: end_block, main, _, _
```

## ΠΑΡΑΡΤΗΜΑ Β

---

# ΛΕΚΤΙΚΟΣ ΚΑΙ ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ ΜΕ ΤΑ ΜΕΤΑ-ΕΡΓΑΛΕΙΑ LEX-YACC/BISON

---

Παρακάτω δίνεται ο κώδικας σε lex-yacc ενός λεκτικού και συντακτικού αναλυτή, ο οποίος περιγράφεται στο κεφάλαιο 12. Ο μεταγλωττιστής έχει εκπαιδευτικούς στόχους, τεχνητές απαιτήσεις και έχει διατηρηθεί όσο το δυνατόν απλούστερος.

Παρατίθεται αρχικά ο κώδικας του μετα-προγράμματος για τον lex:

```
digit [0-9]
letter [a-zA-Z]
ID {letter}({letter}|{digit})*
NUM {digit}+
INVALID {digit}({letter}|{digit})*

%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"

int counter = 1;
%}

%%
"program"    return program_tk;
"declare"    return declare_tk;
"enddeclare" return enddeclare_tk;

"{"          return begin_tk;
"}"          return end_tk;
```

```

"(          return left_par_tk;
")"        return right_par_tk;

{NUM}       return num_tk;

{ID}        {
            yyval.token.line_number = counter;
            strcpy(yyval.token.token_string,yytext);
            return id_tk;
}

{INVALID}   {
            printf("lexical error\n");
            exit(-1);
}

","         return comma_tk;
":="        return assign_tk;
";"         return semicolon_tk;
"+"         return plus_tk;
"-:"        return minus_tk;
"*"         return times_tk;
"/"         return over_tk; // common slash

[ \t]+      ;
"\n"        counter++;
"."         printf("error\n");

%%

int yywrap()
{
    // nothing to do here
    // called whenever the scanner reaches at the end of file
}

```

Αντίστοιχα, ο κώδικας του μετα-προγράμματος για το yacc είναι ο ακόλουθος:

```

%{
// definitions
# include <stdio.h>
int yylex();
void yyerror();

// structure for the communication between
// lexical and syntactic analyzer
%union {
    struct {
        int line_number;
        char token_string[50];

```

```

        }token;
}

// definition of tokens
%token id_tk num_tk program_tk declare_tk enddeclare_tk begin_tk
    end_tk comma_tk assign_tk semicolon_tk
    plus_tk minus_tk times_tk over_tk left_par_tk right_par_tk

%%

/////////// rules

program_rule
: program_tk id_tk
  programblock_rule
;

programblock_rule
: declarations_rule
  begin_tk
  assignments_rule
  end_tk
;

declarations_rule
: declare_tk
  list_of_variables_rule
  enddeclare_tk
;

list_of_variables_rule
: id_tk
  {
    // the first variable has been declared
    printf("Variable '%s' was the first variable declared ",
           $<token.token_string>1);
    printf("in line %d\n",$<token.line_number>1);
  }
| list_of_variables_rule comma_tk id_tk
  {
    // variable declared after comma
    printf("Variable '%s' declared after comma ",
           $<token.token_string>3);
    printf("in line %d\n",$<token.line_number>3);
  }
;

assignments_rule
: assignments_rule assignment_rule
| assignment_rule
;

```

```

assignment_rule
: id_tk assign_tk expression_rule semicolon_tk
{   // assignment of a variable
    printf("Variable '%s' assigned ",$<token.token_string>1);
    printf("in line %d\n",$<token.line_number>1);
}
;

expression_rule
: term_rule
| expression_rule plus_tk term_rule
| expression_rule minus_tk term_rule
;

term_rule
: factor_rule
| term_rule times_tk factor_rule
| term_rule over_tk factor_rule
;

factor_rule
: id_tk
{   // use of a variable
    printf("Variable '%s' used ",$<token.token_string>1);
    printf("in line %d\n",$<token.line_number>1);
}
|
| num_tk
| left_par_tk expression_rule right_par_tk
;

%%

// definition of functions
void yyerror (char const *s)
{
    // called when an error occurs
    // define your own function here
    // this one simply prints a "syntax error" message
    printf ("%s\n", s);
}

// main program
int main(int argc, char** argv)
{
    extern FILE * yyin;
    // read input from the file defined in the command line
    if(argc==2)
    {
        yyin = fopen(argv[1], "r");
        if(!yyin)
        {
            fprintf(stderr, "can't read file %s\n", argv[1]);
        }
    }
}

```

```
    return 0;
}
}

// syntactic analysis starts
yyparse();
// return success to the operating system
return 1;
}
```



## ΠΑΡΑΡΤΗΜΑ Γ

---

# ΜΟΡΦΩΤΗΣ ΚΩΔΙΚΑ ΜΕ ΤΟ ΜΕΤΑ-ΕΡΓΑΛΕΙΟ ANTLR ΚΑΙ ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ PYTHON

---

Όπως είδαμε στο αντίστοιχο κεφάλαιο (κεφ. 12) οι δυνατότητες του ANTLR για να αναπτυχθούν, εύκολα και με τρόπο δομημένο και συστηματικό, εφαρμογές που απαιτούν συντακτική ανάλυση, είναι πολλές. Κατά την προσφιλή τεχνική του βιβλίου, θα βασιστούμε σε ένα παράδειγμα για να εμπεδώσουμε τις γνώσεις μας. Παρακάτω δίνεται ο κώδικας σε ANTLR ενός μορφωτή κώδικα προγράμματος γραμμένου σε μία απλή γλώσσα, η οποία υποστηρίζει εκχωρήσεις με αριθμητικές εκφράσεις και τις εντολές `while` και `if-else`. Ο μορφωτής έχει εκπαιδευτικούς στόχους και για τον λόγο αυτόν έχει διατηρηθεί όσο το δυνατόν απλούστερος. Ως γλώσσα προγραμματισμού επιλέχθηκε η γλώσσα Python.

Ο μορφωτής ακολουθεί τους εξής κανόνες:

- Να μην υπάρχουν κενές γραμμές στον κώδικα.
- Τα άγκιστρα να ανοίγουν στην επόμενη γραμμή που βρίσκονται τα `if`, `else` και `while` και στοιχισμένα κάτω από αυτά.
- Τα άγκιστρα να κλείνουν σε δική τους ξεχωριστή γραμμή, στοιχισμένα με τα `if`, `else`, `while` και το άνοιγμα των αγκίστρων.
- Το μπλοκ κώδικα που υπάρχει μέσα στα άγκιστρα να βρίσκεται ένα `tab` πιο μέσα σε σχέση με τα `if`, `else` και `while`. Το υπόλοιπο μπλοκ να έχει την ίδια στοίχιση. Αν μέσα στο μπλοκ εμφανιστούν φωλιασμένα άγκιστρα, τότε αυτά ακολουθούν τους ίδιους κανόνες, οπότε το φωλιασμένο μπλοκ βρίσκεται ακόμα ένα `tab` πιο μέσα.
- Ανάμεσα στο `if`, `else` και το `while` και το άνοιγμα της παρένθεσης που τα ακολουθεί πρέπει να υπάρχει ένας κενός χαρακτήρας. Πριν και μετά το σύμβολο `=` πρέπει να υπάρχει ένας κενός χαρακτήρας. Δεν υπάρχουν άλλοι κενοί χαρακτήρες στο πρόγραμμα.

Ως παράδειγμα δίνεται το ακόλουθο μη μορφοποιημένο πρόγραμμα:

```

program Beautiful
a := -2; b := 1;
if (a=5) {
    a := 6
}
else
{
    if      (b>1
)
    {
        a := 1 + a
    }
;
while
        (a<10)
{
    a := a*1;
    b := b/ 1
}
}; print(a)

```

το οποίο μετά τη μορφοποίηση θα μετατραπεί στο περισσότερο κομψό:

```

program Beautiful
a := -2;
b := 1;
if (a=5)
{
    a := 6
}
else
{
    if (b>1)
    {
        a := 1+a
    };
    while (a<10)
{
    a := a*1;
    b := b/1
}
};
print(a)

```

Αρχικά δίνεται η γραμματική της γλώσσας σε ANTLR και στη συνέχεια η γραμματική περιγράφεται με λόγια, για την περίπτωση που σας δυσκολέψει κάτι. Η γραμματική ξεκινάει με τη λέξη `grammar` και ακολουθείται από το όνομά της, το οποίο πρέπει να είναι το ίδιο με το όνομα του αρχείου που την περιέχει. Το αρχείο στη συγκεκριμένη περίπτωση πρέπει να έχει ονομαστεί `Beautifier.g4`. Η γραμματική εκκινεί από τον κανόνα `startRule`:

```

grammar Beautifier;

startRule
:   'program' ID statements

```

```

;

statements
:   statement ';' statements
|   statement
;

statement
:   assignment_stat
|   if_stat
|   while_stat
|   print_stat
|
;

assignment_stat
:   ID ':=' signed_expression
;

signed_expression
:   op=('+|-') expression
|   expression
;

if_stat
:   'if' '(' expression RELAT_OPER expression ')'
'{'
    statements
}'
else_part
;

else_part
:   'else'
'{'
    statements
}'
|
;

while_stat
:   'while' '(' expression RELAT_OPER expression ')'
'{'
    statements
}'
;

print_stat
:   'print' '(' expression ')'
;

```

```

expression
:   expression ('+'|'-') term
|   term
;

term
:   term ('*'|'/') factor
|   factor
;

factor
:   INTEGER
|   ID
|   '(' expression ')'
;

INTEGER      :   [0-9]+ ('.' [0-9]+)? ([eE] [+ -]? [0-9]+)?;
ID           :   [a-zA-Z]+;
RELAT_OPER   :   '<' | '>' | '<>' | '<=' | '>=' | '=';
WS            :   [\t\r\n] -> skip;

```

Μετά τη λέξη κλειδί `program` και το όνομα του προγράμματος ακολουθούν οι εντολές `statements`, οι οποίες είναι μία σειρά από `statement` χωριζόμενα από ελληνικό ερωτηματικό.

Κάθε `statement` μπορεί να είναι μία εκχώρηση (`assignment_stat`), μία δομή `if`, μία δομή `while` ή η εντολή `print`.

Στο δεξιό μέλος της εκχώρησης βρίσκεται μία προσημασμένη αριθμητική παράσταση που περιγράφεται με τον κανόνα `signed_expression`. Η `signed_expression` αποτελείται από προσθέσεις, αφαιρέσεις, πολλαπλασιασμούς και διαιρέσεις με τη συνήθη προτεραιότητα, τα οποία υλοποιούνται στους κανόνες `expression`, `term` και `factor`.

Η δομή `while` περιγράφεται στον κανόνα `while_stat` και ξεκινά με τη δεσμευμένη λέξη `while`. Ανάμεσα σε σύμβολα παρένθεσης τοποθετείται μία σύγκριση που συγκρίνει δύο `expression`. Δεν υποστηρίζονται περισσότερο πολύπλοκες παραστάσεις με τελεστές `and`, `or`, `not`. Ακολουθεί το σώμα της επανάληψης που είναι μία `statements` ανάμεσα σε σύμβολα { και }.

Αρκετά όμοια είναι και η δομή `if` η οποία περιγράφεται στον κανόνα `if_stat`. Ξεκινά με το υποχρεωτικό μέρος της εντολής, όπου η πρώτη λεκτική μονάδα είναι η δεσμευμένη λέξη `if`. Ανάμεσα σε σύμβολα παρένθεσης τοποθετείται, όπως και στη `while`, μία σύγκριση δύο `expression`. Δεν υποστηρίζονται περισσότερο πολύπλοκες παραστάσεις με τελεστές `and`, `or`, `not`. Το σώμα της εντολής είναι μία `statements` ανάμεσα στα σύμβολα { και }. Μετά ακολουθεί το προαιρετικό τμήμα το οποίο ενεργοποιείται όταν βρεθεί η λεκτική μονάδα `else`. Το `else` το ακολουθεί και αυτό μία `statements` ανάμεσα στα σύμβολα { και }.

Τέλος, τουλάχιστον όσον αφορά τους συντακτικούς κανόνες, ο κανόνας `print_stat` περιγράφει πώς θα τυπώσουμε κάτι στην οθόνη. Μετά τη δεσμευμένη λέξη `print` ακολουθεί η `expression` που θα τυπωθεί, τοποθετημένη μέσα σε παρενθέσεις.

Απομένουν οι λεκτικοί κανόνες οι οποίοι περιγράφουν ως απλές κανονικές εκφράσεις τι είναι ένα `ID`, ένας ακέραιος αριθμός, ένας σχεσιακός τελεστής και ένας λευκός χαρακτήρας. Μάλιστα, η οδηγία:

```
--> skip
```

δίπλα από αυτόν υποδηλώνει ότι, όταν βρεθεί ένας λευκός χαρακτήρας, τότε πρέπει αυτός να αγνοηθεί.

Ας δούμε, τώρα, τι παρεμβάσεις πρέπει να γίνουν ώστε η γραμματική να μετατραπεί σε έναν μορφωτή κώδικα. Το ANTLR μας δίνει τη δυνατότητα να ορίσουμε πεδία και μεθόδους μέσα στον συντακτικό μας αναλυτή. Αυτό γίνεται με την οδηγία `members`:

```
parser::members ...
```

Ο συντακτικός αναλυτής υλοποιεί έναν μορφωτή κειμένου. Βασική πληροφορία που πρέπει να αποθηκεύει ένας μορφωτής κειμένου είναι η τρέχουσα στοίχιση, πόσες στήλες, δηλαδή, από την αρχή της γραμμής πρέπει να βρίσκεται το κείμενο που θα τυπωθεί την επόμενη φορά που αυτό θα χρειαστεί. Αυτό απαιτεί ένα πεδίο, ας το ονομάσουμε `tabs` και ας το αρχικοποιήσουμε στο 0, θεωρώντας ότι στο 0 τοποθετούμε την αριστερότερη στήλη του προγράμματος.

Ακολουθούν τρεις μέθοδοι:

- Η μέθοδος `spaces()` καλείται όταν η επόμενη συμβολοσειρά που θα τυπωθεί πρέπει να είναι στοιχισμένη τόσες στήλες δεξιά, όσες υποδηλώνει το πεδίο `tabs`. Έτσι, κάθε φορά που καλείται η `spaces()` ο μορφωτής αλλάζει γραμμή και τυπώνει τόσες κενές θέσεις, όσες απαιτείται. Κάθε κενή θέση είναι ένας ορισμένος αριθμός από κενούς χαρακτήρες. Μετά τους κενούς χαρακτήρες δεν γίνεται αλλαγή γραμμής, αφού αυτό που θα ακολουθήσει ανήκει στην ίδια γραμμή. Αναφορά στο πεδίο `tabs` γίνεται με τον προσφιλή τρόπο της Python: `self.tabs`.
- Η μέθοδος `indent()` αυξάνει την τιμή του `tabs` κατά 1.
- Η μέθοδος `deindent()` μειώνει την τιμή του `tabs` κατά 1.

Τοποθετούμε το `members` κάτω από το `grammar Beautifier`; ως εξής:

```
@parser::members {

    # field with the current depth
    tabs = 0

    # line feed and print the necessary empty space
    def spaces(self):
        print()
        print(self.tabs*' ',end='')

    # move indent one position to the right
    def indent(self):
        self.tabs = self.tabs + 1

    # move indent one position to the left
    def deindent(self):
        self.tabs = self.tabs - 1
}
```

Εδώ ολοκληρώνονται τα πεδία και οι μέθοδοι της κλάσης του συντακτικού αναλυτή και μπορούμε να περάσουμε στη γραμματική. Θα δούμε μία μία τις ενέργειες που τη συνοδεύουν.

Μόλις ο αρχικός κανόνας αναγνωρίσει τη δεσμευμένη λέξη `program` και την ονομασία που του δόθηκε, τότε ο μορφωτής αφήνει 5 κενές γραμμές και τυπώνει στην έκτη γραμμή τη λέξη `program` και το όνομα του προγράμματος. Το όνομα του προγράμματος έχει αναγνωριστεί από τον κανόνα ID κατά τη λεκτική ανάλυση. Η συμβολοσειρά που αναγνωρίστηκε μας είναι διαθέσιμη μέσα από το `$ID.text`.

Στη συνέχεια ακολουθούν τα `statements`. Τα `statements` θέλουμε να είναι στοιχισμένα μία θέση πιο εσωτερικά. Άρα, πριν κληθεί ο κανόνας προετοιμάζουμε το έδαφος αυξάνοντας την τιμή της `tabs` κατά ένα, καλώντας τη μέθοδο `indent()`. Με την ολοκλήρωση του κανόνα `statements` έχει ολοκληρωθεί η συντακτική ανάλυση, οπότε ο μορφωτής επιλέγει να τυπώσει δύο κενές γραμμές. Ο κανόνας μαζί με τις ενέργειες φαίνεται παρακάτω. Η σύνταξη των ενεργειών είναι αυστηρή, και δεν δείχνει να είναι και βολική, έχει όμως εναλλακτικές, αν τις προτιμάτε. Μην ξεχνάμε, βέβαια, ότι σε κάθε πρόγραμμα Python η στοίχιση έχει ιδιαίτερη σημασία.

```

startRule
:   'program' ID           {print('\n\n\n\n\nprogram', $ID.text)}
                           {self.indent()}
statements            {print('\n\n')}          }
;

```

Στον κανόνα statements η μόνη ανάγκη που υπάρχει είναι να αναπαραχθεί το ερωτηματικό όπου βρεθεί. Η αλλαγή γραμμής θα γίνει από την επόμενη εντολή. Η γραμμή που ακολουθεί θα βρίσκεται στοιχισμένη με τη γραμμή που μόλις ολοκληρώθηκε, οπότε δεν απαιτούνται επεμβάσεις στην τιμή του tabs.

```

statements
:
statement ';'           {print(';', end=' ')}      }
statements
| statement
;

```

Ο κανόνας statement είναι καθαρά δομικός ώστε να οδηγήσει στην κατάλληλη εντολή. Δεν ορίζεται κάποια ενέργεια για αυτόν.

Ένας αρκετά χαρακτηριστικός κανόνας είναι ο κανόνας t. Το σώμα της επανάληψης πρέπει να βρίσκεται στοιχισμένο μία θέση πιο εσωτερικά από τη λεκτική μονάδα while. Όταν αναγνωρίζεται μία εντολή, προετοιμάζεται ο χώρος για να τυπωθεί από τον μορφωτή. Η μέθοδος spaces() αλλάζει γραμμή και στη συνέχεια η στοίχιση πηγαίνει όσο δεξιά δηλώνει η tabs. Ακολούθως, τυπώνεται η συμβολοσειρά while, το άνοιγμα της παρένθεσης και καλείται η expression, η οποία θα χειριστεί την έκφραση. Το τι θα κάνουν οι δύο expression που καλούνται δεν αφορά τη while. Η while οφείλει να διαχειριστεί μόνο το σύμβολο της σύγκρισης το οποίο, αφού αναγνωρίσει, το τυπώνει.

Η while οφείλει επίσης να προετοιμάσει τη στοίχιση των εντολών του σώματός της. Η πληροφορία ότι τα statements που ακολουθούν αποτελούν μέρος μιας while, οπότε πρέπει να βρίσκονται εσωτερικότερα, υπάρχει μόνο στο επίπεδο αυτό και εδώ πρέπει να αξιοποιηθεί. Ακριβώς, λοιπόν, πριν τη statements, καλείται η μέθοδος indent() για να μεταφέρει τη στοίχιση μία θέση δεξιότερα και αμέσως μετά τη statements καλείται η μέθοδος deindent() για να επαναφέρει τη στοίχιση στο επίπεδο του while.

Αυτό που δεν συζητήθηκε είναι το άνοιγμα και το κλείσιμο των αγκίστρων. Επιθυμούμε να βρίσκονται στοιχισμένα κάτω από τη λέξη while. Άρα, αφού αναγνωριστούν, πρέπει αμέσως πριν το indent() και αμέσως μετά το deindent() να τυπωθεί το άνοιγμα και το κλείσιμο των αγκίστρων και αφού, βέβαια, κληθεί και η spaces() για να τα στοιχίσει στο σωστό σημείο.

Έτσι ολοκληρώνονται οι ενέργειες που αφορούν τον κανόνα while και ο κώδικας που αναλογεί στο σημείο αυτό ακολουθεί:

```

while_stat
:   'while' '('           {self.spaces()}          }
                           {print('while (',end=' ')}    }
expression
op=RELAT_OPER
expression ')'
'{'
{
statements
'}'
{self.deindent()}
{self.spaces()}
{print(')',end=' ')}
;
```

Ακολουθεί ο κανόνας `if`, ο οποίος έχει τις ίδιες ανάγκες όσον αφορά τη στοίχιση με το `while`. Η διαφοροποίησή του έγκειται στην ύπαρξη του προαιρετικού μέρους, του `else`, το οποίο και αυτό έχει παρόμοιες ανάγκες στοίχισης. Ο κανόνας `if`, μαζί με τις απαιτούμενες ενέργειες για την ορθή την στοίχιση, ακολουθεί:

```

if_stat
:   'if' '('
    expression
    op=RELAT_OPER
    expression ')'
    '{'
    statements
    '}'

    else_part
;

else_part
:   'else'
    '{'
    statements
    '}'

;

```

Ο κανόνας `assignment` πρέπει να τυπώσει τα κενά που απαιτούνται και στη συνέχεια να τυπώσει το `print` τις παρενθέσεις και να αφήσει τον κανόνα `expression` να φροντίσει για την έκφραση. Και αυτός ο κανόνας και οι επόμενοι που ακολουθούν δεν αλληλεπιδρούν με το `tabs` και αφήνονται στο περιβάλλον στοίχισης που ήδη υπάρχει:

```

assignment_stat
:   ID
    {self.spaces()}
    {print($ID.text,end='')}
    ':='
    signed_expression
;

```

Ο κανόνας `signed_expression` πρέπει να διαχειριστεί το πρόσημο. Όλα τα υπόλοιπα τα διαχειρίζεται το `expression`:

```

signed_expression
:   op=( '+'|'-')
    expression
|
    expression
;

```

Ο κανόνας print δεν έχει να κάνει κάτι δύσκολο, μόνο να φροντίσει να κληθεί η spaces() για να τον τοποθετήσει στην κατάλληλη στήλη. Φυσικά, πέρα από τη λέξη print πρέπει να τυπώσει το άνοιγμα και το κλείσιμο της παρένθεσης. Το τι θα τοποθετηθεί μέσα στις παρενθέσεις θα το φροντίσει η expression.

```
print_stat
:   'print' '('
      {self.spaces()}
      {print('print ',end='')}
      expression ')'
      {print(')',end='')
}
;
```

Με τη σειρά του, ο κανόνας expression θα πρέπει να διαχειριστεί τα τερματικά του σύμβολα και να τυπώσει τον τελεστή που θα αναγνωρίσει στο op:

```
expression
:   expression op=( '+' | '-' )    {print($op.text,end='')}
      term
|   term
;
;
```

Όμοια λειτουργεί και ο κανόνας term:

```
term
:   term op=( '*' | '/' )
      factor
|   factor
;
;
```

Ο κανόνας factor πρέπει να τυπώσει τα τερματικά σύμβολα INTEGER και ID:

```
factor
:   INTEGER          {print($INTEGER.text,end='')}
|   ID               {print($ID.text,end='')}
|   '(' expression ')'
;
;
```

Η πλήρης γραμματική, μαζί με τις ενέργειες που υλοποιούν τον μορφωτή, ακολουθεί:

```
grammar Beautifier;

@parser::members {
    # field with the current depth
    tabs = 0

    # line feed and print the necessary empty space
    def spaces(self):
        print()
        print(self.tabs*' ',end='')

    # move indent one position to the right
    def indent(self):
        self.tabs = self.tabs + 1

    # move indent one position to the left
    def deindent(self):
        self.tabs = self.tabs - 1
}
```

```

# parsing starts here
startRule
:   'program' ID           {print('\n\n\n\n\nprogram',$ID.text)}
    statements             {self.indent()                      }
:   ;                      {print('\n\n')                         }

# statements is a sequence of statement
statements
:   statements ';'         {print(';',end='')}                  }
|   statement
|   statement
:   ;

# a statement can be an assignment, an if-else, a while or a print
statement
:   assignment_stat
|   if_stat
|   while_stat
|   print_stat
|   ;
:   ;

# an assignment
assignment_stat
:   ID                     {self.spaces()}                  }
|   ':='                   {print($ID.text,end='')}      }
:   signed_expression      {print(':=',end='')}            }

# the optional sign
signed_expression
:   op=( '+' | '-' )       {print($op.text,end='')}      }
|   expression
|   expression
:   ;

# if statement
if_stat
:   'if' '('               {self.spaces()}                  }
|   expression
|   op=RELAT_OPER          {print('while (',end='')}     }
|   expression ')'
|   '{'
|   statements
|   '}'
:   ;

```

```

        {self.spaces()}           }
        {print('}',end='')}      }

    else_part
    ;

# the optional part of if statement
else_part
:   'else'
    '{'
        statements
    '}'
    ;
|


# while statement
while_stat
:   'while' '('
    expression
    op=RELAT_OPER
    expression ')'
    '{'
        statements
    '}'
    ;
|


# the print statement
print_stat
:   'print' '('
    expression ')'
    ;
|


# an expression consists of a series of terms
expression
:   expression op=( '+' | '-' )   {print($op.text,end='')}   }
    term
|   term
;
|
```

```
# a term is a series of factors
term
:   term op=('*'|'/')           {print($op.text,end='')}      }
    factor
|   factor
;

# a factor is a ID, an INTEGER or an expression in parenthesis
factor
:   INTEGER                  {print($INTEGER.text,end='')}  }
|   ID                      {print($ID.text,end='')}        }
|   '('                     {print(''("',end='')}          }
    expression
) '
{print('")'',end='')}      }
;

# lexical rules
INTEGER     :   [0-9]+
ID          :   [a-zA-Z]+;
RELAT_OPER  :   '<' | '>' | '<>' | '<=' | '>=' | '=';
WS          :   [ \t\r\n] -> skip;
```





# ΕΓΧΕΙΡΙΔΙΟ ΣΧΕΔΙΑΣΗΣ ΚΑΙ ΑΝΑΠΤΥΞΗΣ ΜΕΤΑΓΛΩΤΤΙΣΤΩΝ

Σκοπός του παρόντος συγγράμματος είναι η εισαγωγή του αναγνώστη στην επιστήμη των μεταγλωττιστών. Η προσέγγιση που επιλέγεται είναι αυτή της καθοδήγησης της γνώσης μέσα από τη διαδικασία ανάπτυξης. Σε αντίθεση με άλλα βιβλία που είναι διαθέσιμα στα ελληνικά και στα οποία δίνεται το βάρος στο θεωρητικό υπόβαθρο, εδώ το θεωρητικό υπόβαθρο γίνεται όχημα και η ανάπτυξη το κίνητρο και ο οδηγός.

Η πορεία του βιβλίου συμπίπτει με αυτή της ανάπτυξης των μεταγλωττιστής. Η ανάπτυξη ενός μεταγλωττιστή χωρίζεται σε φάσεις. Οι φάσεις της ανάπτυξης είναι η λεκτική ανάλυση, η συντακτική ανάλυση, η παραγωγή ενδιάμεσου κώδικα, η κατασκευή του πίνακα συμβόλων και η παραγωγή τελικού κώδικα. Για καθεμία από αυτές το βιβλίο αφιερώνει ένα ή δύο κεφάλαια, ενώ συμπληρώνεται με εισαγωγικά κεφάλαια για τις γραμματικές και τον ρόλο τους, ένα κεφάλαιο για θέματα βελτιστοποίησης κώδικα και ένα κεφάλαιο για εργαλεία αυτοματοποιημένων διαδικασιών ανάπτυξης μεταγλωττιστών.

Στο βιβλίο ορίζεται μία εκπαιδευτική γλώσσα προγραμματισμού για την οποία υλοποιείται ένας πλήρως λειτουργικός μεταγλωττιστής, ο οποίος δέχεται ως είσοδο προγράμματα συνταγμένα στη γλώσσα αυτή και παράγει τον αντίστοιχο κώδικα σε γλώσσα μηχανής. Πρόκειται για μια γλώσσα προγραμματισμού που αντλεί ιδέες και δομές από τη C, αλλά είναι πιο μικρή τόσο στις υποστηριζόμενες δομές, όσο και σε προγραμματιστικές δυνατότητες. Παρόλο που οι προγραμματιστικές της δυνατότητες είναι μικρές, η εκπαιδευτική αυτή γλώσσα προγραμματισμού περιέχει πλούσια στοιχεία και η κατασκευή του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον, αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται από γνωστές γλώσσες προγραμματισμού, όπως είναι οι δημοφιλείς while και if-else, καθώς και κάποιες πρωτότυπες. Υποστηρίζει συναρτήσεις και διαδικασίες, μετάδοση παραμέτρων με αναφορά και τιμή, αναδρομικές κλήσεις, ενώ επιτρέπει φώλιασμα στη δήλωση συναρτήσεων και διαδικασιών.

Το παρόν σύγγραμμα δημοιουργήθηκε στο πλαίσιο του Έργου ΚΑΛΛΙΠΟΣ+	
Χρηματοδότης	Υπουργείο Παιδείας και Θρησκευμάτων, Προγράμματα ΠΔΕ, ΕΠΑ 2020-2025
Φορέας υλοποίησης	ΕΛΚΕ ΕΜΠ
Φορέας λειτουργίας	ΣΕΑΒ/Παράρτημα ΕΜΠΙ/Μονάδα Εκδόσεων
Διάρκεια 2ης Φάσης	2020-2023
Σκοπός	H δημιουργία ακαδημαϊκών ψηφιακών συγγραμμάτων ανοικτής πρόσβασης (περισσότερων από 700) <ul style="list-style-type: none"><li>• Προπτυχιακόν και μεταπτυχιακών εγχειριδίων</li><li>• Μονογραφών</li><li>• Μεταφράσεων ανοικτών textbooks</li><li>• Βιβλιογραφικών Οδηγών</li></ul>
Επιστημονικά Υπεύθυνος	Νικόλαος Μήτρου, Καθηγητής ΣΗΜΜΥ ΕΜΠ
ISBN: 978-618-228-141-3	DOI: <a href="http://dx.doi.org/10.57713/kallipos-372">http://dx.doi.org/10.57713/kallipos-372</a>

Το παρόν σύγγραμμα χρηματοδοτήθηκε από το Πρόγραμμα Δημοσίων Επενδύσεων του Υπουργείου Παιδείας.