ΚΕΦΑΛΑΙΟ 1

ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ

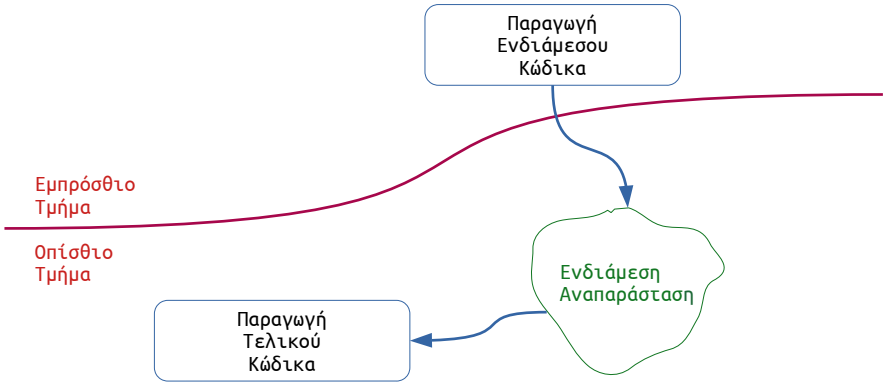
Παραγωγή ενδιάμεσου κώδικα

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

Μάρτιος 2022

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.

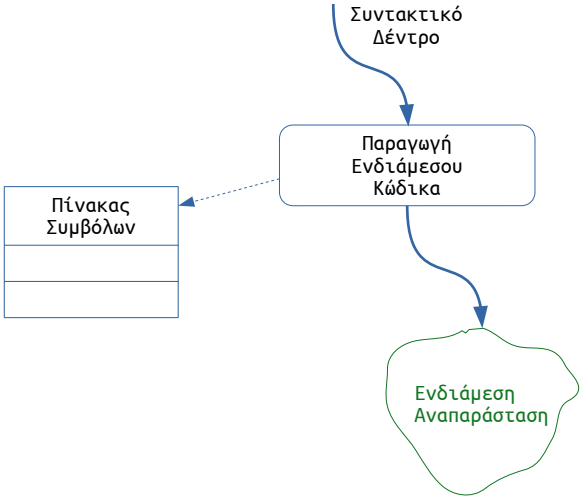
Θα παρουσιάσουμε εδώ μια τέτοια γλώσσα και θα δούμε πως από τη φάση της συντακτικής ανάλυσης θα οδηγηθούμε στην παραγωγή του ενδιάμεσου κώδικα.



Σχήμα 1.1: Η ενδιάμεση αναπαράσταση αποτελεί μέσο επικοινωνίας ανάμεσα στο εμπρόσθιο και το οπίσθιο τμήμα του μεταγλωττιστή.

Η ύπαρξη μιας ενδιάμεσης γλώσσας, σαν 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 1.1.

- 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)*.



Σχήμα 1.2: Παραγωγή κώδικα

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 σαν ενότητα λογισμικού μπορούμε να θεωρήσουμε ότι παίρνει ως είσοδο το δέντρο της συντακτικής ανάλυσης. Πιο ξεκάθαρο είναι τι δημιουργεί σαν 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 σαν νοηματική ενότητα στο επόμενο. Σχηματικά η παραγωγή ενδιάμεσου κώδικα σαν ενότητα λογισμικού φαίνεται στο σχήμα 1.2.

1.1 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 γα 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 είναι να

χρησιμοποιήσουμε περισσότερες της μίας τετράδες είτε η 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 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 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 ένα αρχείο κειμένου δεν είναι σχεδιασμένο για το σκοπό αυτό.

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.

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 κωρίως 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 op είναι ένα ακ 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 ελέγχο 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 όνομα της. Οι απαραίτητοι διαχωρισμοί θα γίνουν σε άλλο σημείο και, πιο συγκεκριμένα, στο πέρασμα των παραμάτρων.

Πέρασμα πραγματικής παραμέτρου:

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

Το `name` είναι 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. Χειριζόμαστε την επιστροφή τιμής της συνάρτησης σαν 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 σαν παράμετρος `ret` της συνάρτησης:

```
par, name, ret, _
```

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

Εδώ πρέπει να εισάγουμε 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 συνατησης 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 προβλήματά.

- 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 αλφάβητο 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 μια εντολή για την επιστροφή τιμής. Είναι αυτό που θα αντιστοιχίζαμε στο `return` μίας συνάρτησης. Μας αρκεί μία εντολή με μία παράμετρο, την 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.

Ο κώδικας:

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

Θα δημιουργήσει τον κώδικα:

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

1.2 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 κάθε μία θα αποτελεί μιά 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 σαν μοναδικό στοιχείο της την ετικέτα τετράδας `label`

- `mergeList(list1, list2)`:
  Δημιουργεί μία λίστα και συνενώσει τις `list1` και `list2` σε αυτήν.

- `backpatch(list,label)`:
  Διαβάζει μία μία της 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:
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 x2. 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 της 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(), δηλαδη με το 104. Η λίστα x θα επιστρέψει το 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
```

1.3 Αριθμητικές παραστάσεις

Σύμφωνα με τη γραμματική της γλώσσας, υποστηρίζονται αριθμητικές παραστάσεις με τις τέσσερις αριθμητικών πράξεων (πρόσθεση, αφαίρεση, πολλαπλασιασμός και διαίρεση) καθώς και η ομαδοποίηση/προτεραιότητα ανάμεσα σε αυτές που ορίζεται με τη χρήση παρενθέσεων. Θα χρησιμοποιήσουμε μία γραμματική, απλούστερη από αυτήν της *C-imple*, 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
```

Στην γραμματική αυτή έχει χρησιμοποιηθεί ο συμβολισμός του δείκτη σε παρένθεση στη θέση του εκθέτη, προκειμένου να διαχωρίσουμε τις διαφορετικές εμφανίσεις του κάθε μη τερματικού συμβόλου. Τα $T^{(1)}$ και $T^{(2)}$ 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 θεω-

ρήσουμε δηλαδή σαν 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 μέρος 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 προωθήσει 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 προωθήσει 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 μέλους, 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 σαν 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. Κάθε ένας από τους τέσσερις κανόνες της γραμματικής επιστρέφει στον κανόνα που τον ενεργοποίησε μία μεταβλητή σαν 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 $\mathtt{T}^{(1)}$ το $\mathtt{T}^{(1)}\mathtt{.place}$, από την $\mathtt{T}^{(2)}$ το $\mathtt{T}^{(2)}\mathtt{.place}$ και θα δημιουργήσει (ενδιάμεσο κώδικα και) την `E.place`.

Ο πρώτος κανόνας

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

είναι υπεύθυνος για την τέλεση της πρόσθεσης. Διαισθητικά πάλι, θα περιμέναμε όταν συναντηθεί κάπου το σύμβολο + ο κανόνας αυτός να είναι υπεύθυνος να δημιουργήσει τον κώδικα που απαιτείται σε ενδιάμεση γλώσσα. Πράγματι έτσι είναι. Ας δούμε πως 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 E, το αποτέλεσμα αυτό θα επιστρέφει από τον κανόνα $T^{(1)}$, μέσα από τη μεταβλητή $T^{(1)}$`.place`. Πράγματι, αν το σκεφτούμε περισσότερο, ο $T^{(1)}$ δίνει ως αποτέλεσμα στην 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 πως 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 $T^{(1)}$`.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   →   T(1) ( + T(2) )* {p1}

{p1} :  E.place = T(1).place
```

Η παραπάνω γραμματική αντιστοιχεί σε κώδικα. Ο συμβολισμός {p1} μέσα στον κανόνα σημαίνει ότι στον κώδικά 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 +. Αφού αναγνωριστεί το +, θα κληθεί ο κανόνας $T^{(2)}$ ο οποίος θα επιστρέψει τη μεταβλητή $T^{(2)}$`.place` ως αποτέλεσμα. Έχουμε λοιπόν το $T^{(1)}$`.place` από τον $T^{(1)}$ και το $T^{(2)}$`.place` από τον $T^{(2)}$. Πρέπει αυτά τα δύο να προστεθούν. Το αποτέλεσμα θα πρέπει να εκχωρηθεί σε μία μεταβλητή που θα επιστραφεί σαν 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 $T^{(1)}$`.place` και τη μεταβλητή που βρίσκεται στο $T^{(2)}$`.place` και θα εκχωρεί το αποτέλεσμα σε μία μεταβλητή που δεν θα επηρεάσει την λειτουργία του προγράμματος. Δεν μπορούμε δηλαδή να χρησιμοποιήσουμε μια 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   →   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
```

Στο τέλος του {p1}, τοποθετήσαμε τη νέα μεταβλητή στο $T^{(1)}$`.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 υην περίπτωση που θα αναγνωριστούν περισσότερες από μία.

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 $T^{(1)}$`.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`}, όπου 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 που ο κάνονας 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. Αλλάζουν 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 ένας σημαντικός κανόνας, και είναι, 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 που έχει το αποτέλεσμα των πράξεων που έχουν δημιουργηθεί από τον κανόνα αυτόν και όσους βρίσκονται κάτω από αυτόν στο συντακτικό δέντρο. Αυτή 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 είναι να μεταφέρει το τερματικό σύμβολο που αναγνώρισε στην μεταβλητή `F.place`. Το σχέδιο ενδιάμεσου κώδικα για τον κανόνα αυτόν ακολουθεί:

```
F   →   ID  {p1}

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

Ολοκληρώνοντας παραθέτουμε ένα ολοκληρωμένο 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 στον κανόνα από τον οποίο ενεργοποιήθηκε η εκτέλεσή της το `T_5`, σαν αποτέλεσμα, δηλαδή το `E.place` θα έχει την τιμή `T_5`.

Δεν θα αναλύσουμε βήμα βήμα πως φτάσαμε ως εδώ. Θα το αφήσουμε σαν 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`.

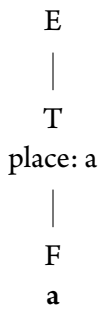
Συνοψίζοντας, ας παραθέσουμε το σχέδιο ενδιάμεσου κώδικα για τις αριθμητικές παραστάσεις, ολοκληρωμένο και συγκεντρωμένο:

```
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
```

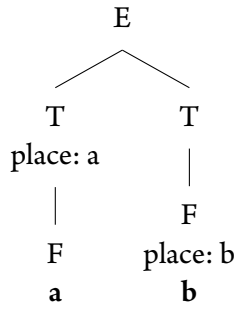
Θα δώσουμε ένα ακόμα παράδειγμα, αρκετά αναλυτικό αυτή τη φορά. Έστω ότι θέλουμε να μετατρέψουμε σε ενδιάμεσο κώδικα την παράσταση:

```
a+b*(c+d+1)
```

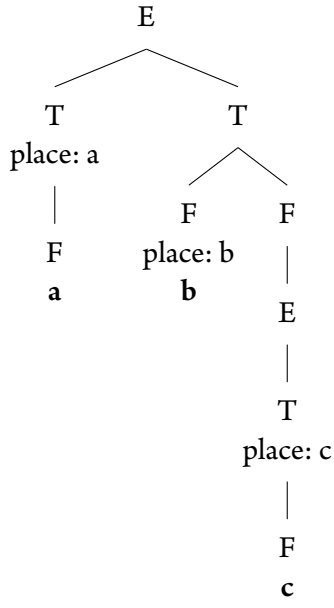
Η συντακτική ανάλυση θα εκκινήσει από τον κανόνα `E`. Θα κληθεί ο κανόνας `T` και στη συνέχεια ο κανόνας `F`, ώστε να αναγνωριστεί το `a`. Το `F.place` θα πάρει την τιμή `a` η οποία θα περάσει και στο `T.place`. Έτσι, τη στιγμή αυτή το δέντρο που έχει σχηματιστεί είναι το ακόλουθο:



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



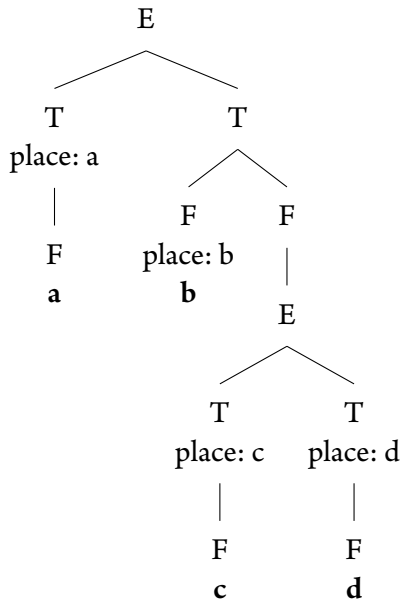
Στο σημείο αυτό, το `*` δημιουργεί νέο `F`, ενώ η παρένθεση που το ακολουθεί θα δημιουργήσει `E`. Η ανάλυση θα συνεχιστεί, το `E` θα δημιουργήσει `T` και το `T` ένα νεό `F` το οποίο θα αναγνωρίσει το `c`. Το `F.place` θα περάσει το `c` στο `T.place`, οπότε η εικόνα του δέντρου τώρα είναι:



Σε κάθε κόμβο του δέντρου φαίνεται ο κανόνας που ενεργοποιήθηκε στο σημείο αυτό. Όταν είμαστε σε φύλλο του δέντρου 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.

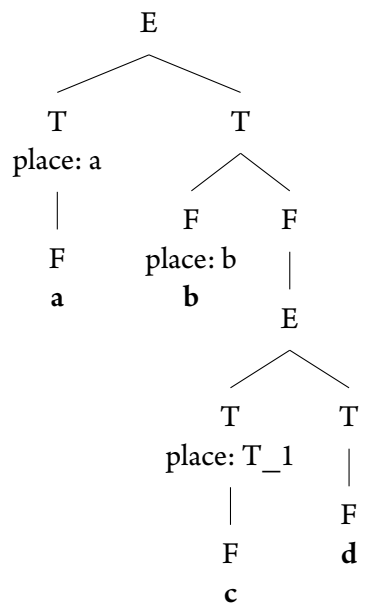
Το τερματικό σύμβολο + που ακολουθεί θα δημιουργήσει έναν νέο όρο (κανονας `T`), δίπλα στο τελευταίο μη τερματικό σύμβολο `T` και με την ίδια ακολουθία κλήσεων και μεταφοράς αποτελεσμάτων από κανόνα σε κανόνα, θα φέρει το `d` στο `T.place`. Το δέντρο τώρα θα γίνει:



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:

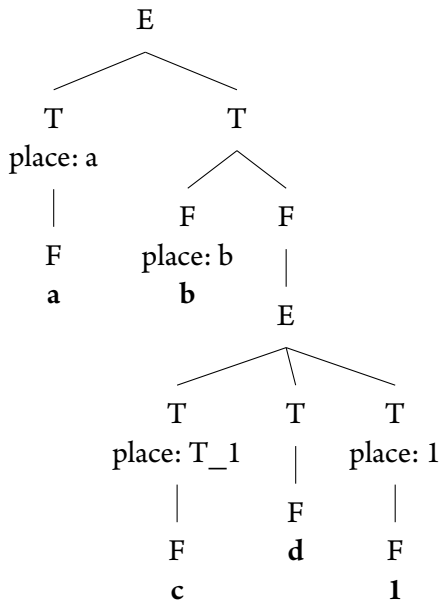


Στη συνέχεια ακολουθεί το επόμενο τερματικό σύμβολο +, το οποίο θα δημιουργήσει την ίδια σειρά κλήσεων και επιστροφών τιμών, ώστε το τερματικό σύμβολο 1, να ανέβει και να τοποθετηθεί στο `T.place`.

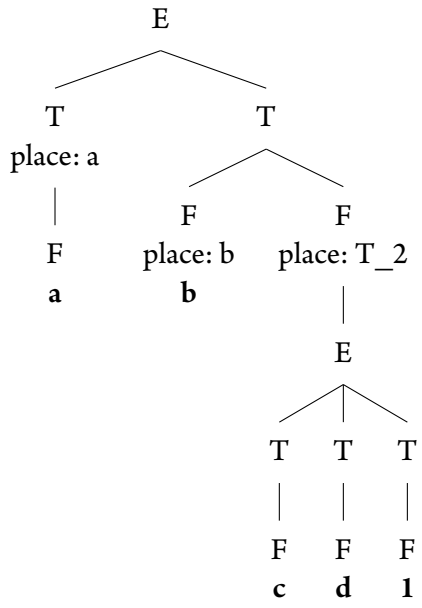
Στο σημείο αυτό θα παραχθεί νέα τετράδα ενδιάμεσου κώδικα, η:

```
101: +, T_1, 1, T_2
```

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



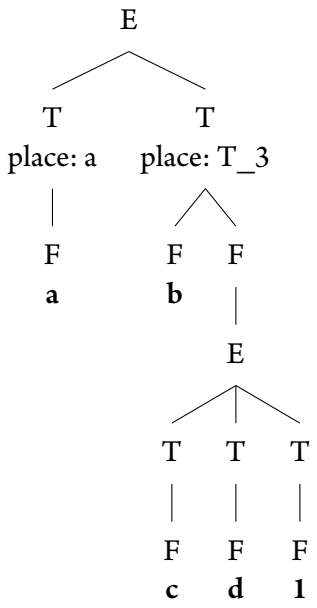
Το `T_2` είναι το αποτέλεσμα που θα μεταφερθεί στο `E` και στη συνέχεια στο `F`, όπως φαίνεται παρακάτω:



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
```

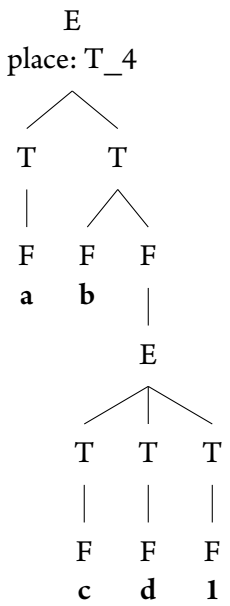
και το δέντρο γίνεται:



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 στην δομή που ανήκει η αριθμητική παράσταση, ώστε να τη χρησιμοποιήσει:



και ο κώδικας που παρήχθη 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
```

διαχειρι

1.4 Λογικές συνθήκες

Σε ένα πρόγραμμα αρχικής γλώσσας, μία λογική παράσταση μπορεί να εμφανιστεί, είτε στον δεξί μέλος μίας εκχώρησης σε μία λογική μεταβλητή ή μέσα σε μία λογική συνθήκη. Η πρώτη περίπτωση δεν εμφανίζεται μέσα σε ένα πρόγραμμα *C-imple*. 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 αυτός 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 σαν 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 σαν δικό του αποτέλεσμα την 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 όλοι 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-imple*. 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 πως 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 που 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 πως θα συμπληρωθούν οι τετράδες αυτές. Η λύση στο πρόβλημα αυτό είναι η πληροφορία αυτή να περάσει σαν 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 ιεραχία ενός δέντρου κανόνων

που αποτιμούν λογικές παραστάσεις, στον κανόνα δηλαδή που καλείται από μία `if`, `while`, κλπ. Η ίδια ανάγκη παρουσιάζεται σε κάθε έναν 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 μεταγλώττιση που θα τοποθετηθούν οι εντολές που τελικά τοποθετήθηκαν στα 102 και 104. Ο κανόνας αυτός πρέπει να επιστρέψει σαν αποτέλεσμα τις τετράδες που έχουν μείνει ασυμπλήρωτες, έτσι ώστε να συμπληρωθούν αργότερα, από άλλους κανόνες, όταν γνωρίζουμε το που 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 της εκείνης της τετράδας στην οποία πρέπει να μεταβεί ό 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 ό 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 την συνένωση λογικών συνθηκών με τον τελεστή `and`.

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

{p1} : B.true = Q(1).true
       B.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 λοιπόν 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 που; 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 αποτιμηθεί 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 σαν βάση τις λίστες `B.true` και `B.false` από την προηγούμενη επανάληψη, κάνουμε `backpatch()` την 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 νοιάζει 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, σαν 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  →  R ( 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 την συνθήκη, διότι δεν απαιτούνται περισσότεροι έλεγχοι. Αντίθετα, όταν αποτυγχάνει μία συνθήκη που βρίσκεται αριστερά ενός and, τότε ο έλεγχος μεταβαίνει έξω από την 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 στο 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:

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

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

Ας περάσουμε στον επόμενο κανόνα:

```
R → not [ B ]
  |   [ B ]
  |   E rel_op E
```

Πρακτικά εδώ έχουμε τρεις κανόνες, ας ξεκινήσουμε με τον τελευταίο:

```
R → E(1) 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:

```
x:   rel_op, E(1).place, E(2).place, ...
x+1: jump, _, _, ...
```

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 το 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 που 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 σαν 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`. Εδώ, οι λίστες αυτές δημιουργούνται, αφού εδώ δημιουργούνται και οι τετράδες των αλμάτων. Για το 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 θα φυσικά πρόκειται για μία μη συμπληρωμένη τετράδα.

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

```
R → E(1) rel_op E(2) {p1}

{p1}: R.true = makeList(nextQuad())
      genQuad(rel_op, E(1).place, E(2).place}, '_')
      R.false = makeList(nextQuad())
      genQuad('jump', '_', '_', '_')
```

Ο επόμενος κανόνας που θα εξετάσουμε είναι ο:

```
R →  [ 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 ώς έχουν στην `R`. Άρα το σχέδιο ενδιάμεσου κώδικα που υλοποιεί τη μεταφορά αυτή είναι:

```
R  →   [ B ] {p1}

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

Ολοκληρώνουμε με τον κανόνα:

```
R  →   not [ B ]
```

Η διαφοροποίησή του από τον προηγούμενο κανόνα, που δεν περιέχει το not, έγγειται 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 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 true του B γίνονται οι τετράδες της λίστας false του R. Αντίστοιχα, οι τετράδες από τη λίστα false του B γίνονται οι τετράδες της λίστας true του R. Το σχέδιο ενδιάμεσου κώδικα ακολουθεί:

```
R  →  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 εννιαιο σχέδιο, ώστε να το έχουμε συγκεντρωμένο και ολοκληρωμένο:

```
B  →   Q(1) {p1} ( or Q(2) )*
{p1} : B.true = Q(1).true
       B.false = Q(1).false

Q  →   R(1) {p1} ( or {p2} R(2) {p3} )*
{p1} : Q.true = R(1).true
       Q.false = R(1).false
{p2} : backpatch(Q.true, nextquad())
{p3} : Q.false = mergeList(Q.false,R(2).false)
       Q.true = R(2).true

R  →  E(1) rel_op E(2) {p1}
{p1} : R.true = makeList(nextQuad())
       genQuad(rel_op, E(1).place, E(2).place}, '_')
       R.false = makeList(nextQuad())
       genQuad('jump', '_', '_', '_')

R  →   [ B ] {p1}
{p1} : R.true = B.true
       R.false = B.false

R  →  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
```

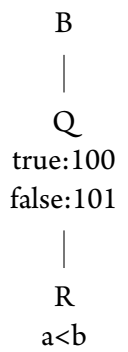
Η μεταγλώττιση θα ξεκινήσει από τον κανόνα B, ο οποίος θα καλέσει τον Q και στη συνέχεια θα κληθεί ο R ο οποίος (με τη βοήθεια του E, αλλά ας μην προχωρήσουμε περισσότερο σε βάθος) θα αναγνωρίσει το a>b. Θα δημιουργηθούν οι τετράδες:

```
100: <, a, b, _
101: jump, _, _, _
```

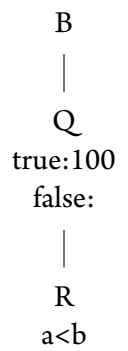
οι οποίες θα είναι ασυμπλήρωτες, άρα θα τοποθετηθούν στις λίστες true, false

```
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
```

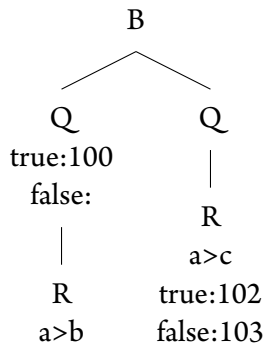
Στη συνέχεια θα ξεκινήσει η αναγνώριση του Q που ακολουθεί το or, μέσα στο οποίο θα κληθεί ο κανόνας R και θα γίνει η αναγνώριση του a>c. Θα δημιουργηθεί ο ενδιάμεσος κώδικας:

```
102: <, a, c, _
103: jump, _, _, _
```

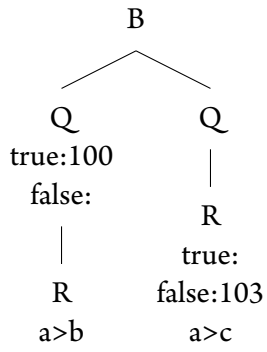
Οι τετράδες είναι ασυμπλήρωτες, άρα θα τοποθετηθούν στις λίστες true, false του R:

```
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, όπου 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.

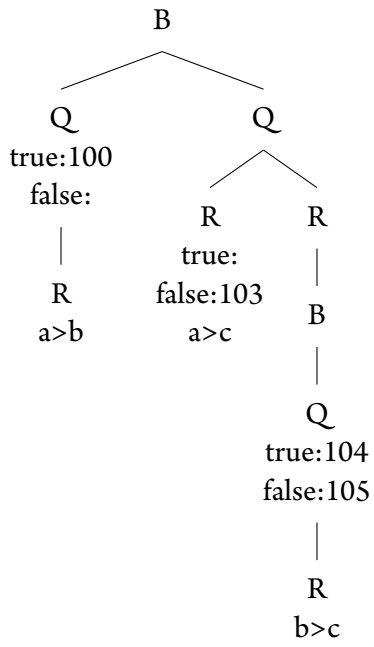
Θα δημιουργηθεί ο ενδιάμεσος κώδικας:

```
104: >, b, c, γίνεται_
105: jump, _, _, _
```

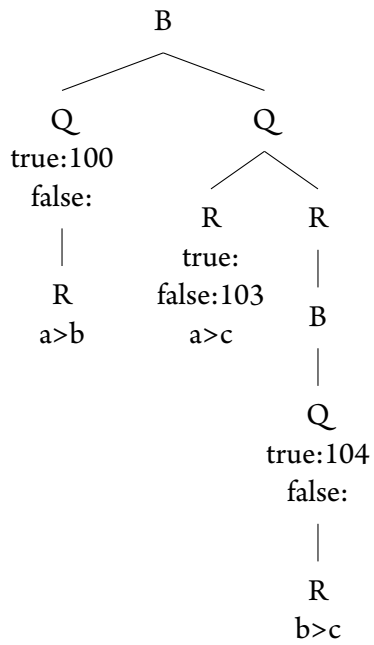
οι οποίες θα είναι ασυμπλήρωτες, άρα θα τοποθετηθούν στις λίστες true, false του R:

```
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 θυμίζουμε 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
```

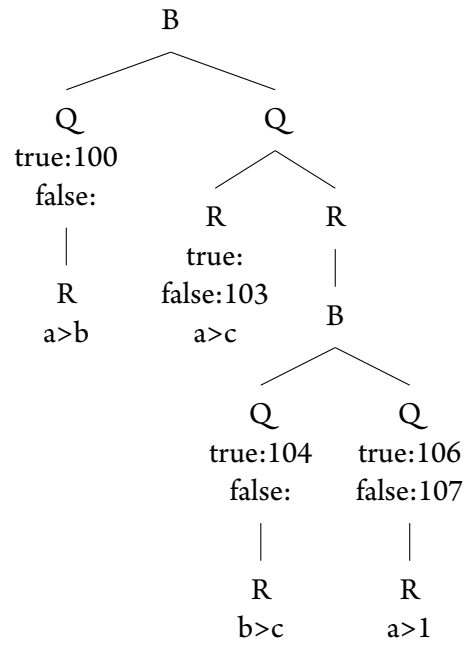
Μέτα αναγνωρίζεται το νέο `Q`, καλώντας φυσικά το `R`. Αναγνωρίζεται η σύγκριση `a>1`, δημιουργείται ο κώδικας:

```
106: >, a, 1, _
107: jump, _, _, _
```

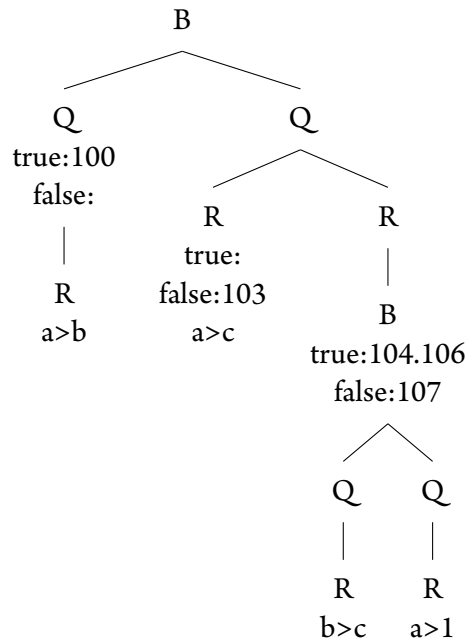
και οι ασυμπλήρωτες τετράδες μπαίνουν στις λίστες:

```
true: [106]
false: [107]
```

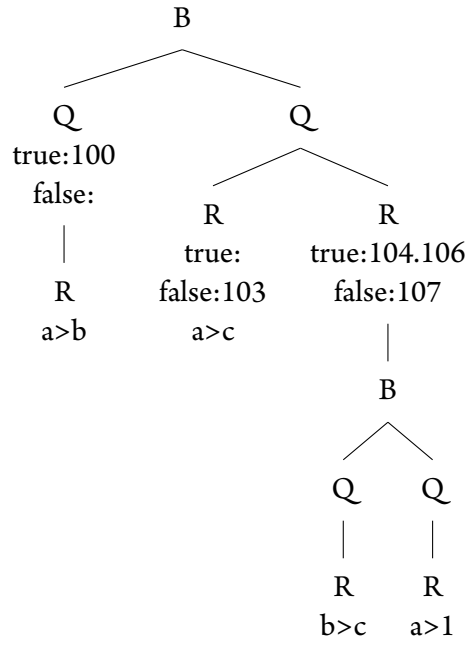
Το δέντρο τώρα έχει γίνει ως εξής:



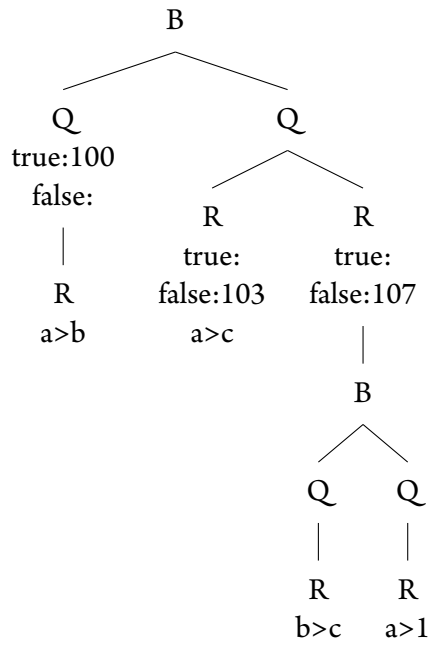
Ο κανόνας `B` έχει ολοκληρωθεί. Οι δύο κανόνες `Q` του έχουν δημιουργήσει τρεις λίστες τις οποίες πρέπει να διαχειριστεί. Μπόρεσε και έκανε `backpatch()` στην περίπτωση του `false` της πρώτης `Q`, αλλά τώρα πρέπει να συνδυάσει τις τρεις λίστες σε δύο ώστε να τις περάσει στον κανόνα που τον ενεργοποίησε, 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, _, _, _
```

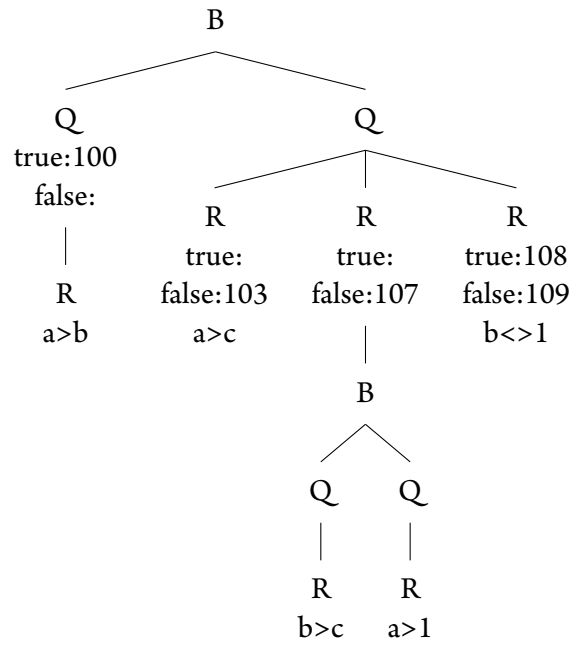
Στη συνέχεια θα αναγνωριστεί μέσω ενός νέου R το b<>1. θα δημιουργηθεί ο κώδικας:

```
108: =, b, 1, _
109: jump, _, _, _
```

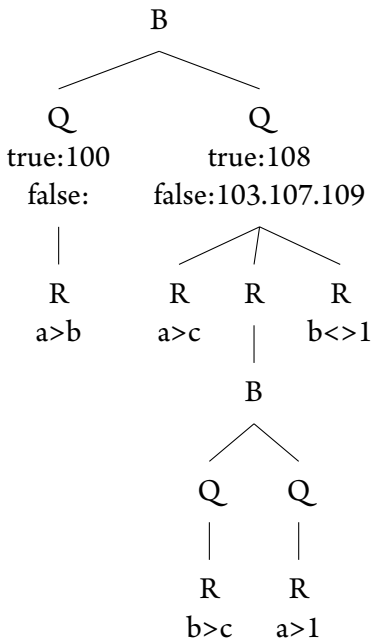
και οι λίστες:

```
true: [108]
false: [109]
```

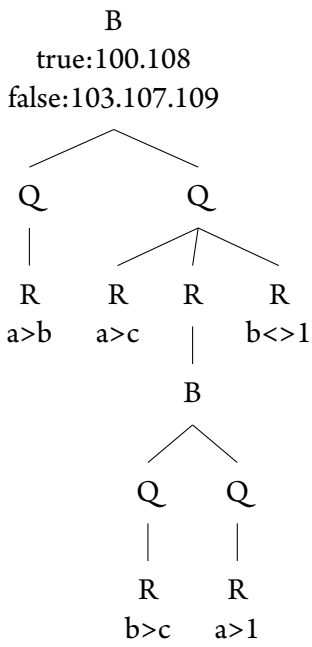
ενώ το δέντρο γίνεται:



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
```

1.5 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`.

1.6 Είσοδος και έξοδος δεδομένων

Το ίδιο απλή με την `return` είναι οι δύο εντολές εισόδου-εξόδου: η `input` και η `print`. Και οι δύο δημιουργούν μία εντολή ενδιάμεσου κώδικα. Η:

```
input (x)
```

αντιστοιχεί στη δημιουργία της:

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

και η :

```
print (x)
```

αντιστοιχεί στη δημιουργία της:

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

1.7 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`.

ΚΕΦΑΛΑΙΟ 2

ΕΝΔΙΑΜΕΣΟΣ ΚΩΔΙΚΑΣ ΓΙΑ ΤΙΣ ΔΟΜΕΣ ΤΗΣ ΓΛΩΣΣΑΣ

2.1 Οι δομές της γλώσσας

Θα δούμε παρακάτω πως θα κατασκευάσουμε σχέδιο ενδιάμεσου κώδικα για τις τρεις βασικές δομές της γλώσσας: τη δομή επανάληψης `while`, τη δομή απόφασης `if` και τη δομή επιλογής `switchcase`. Ιδιαίτερο ενδιαφέρουν έχουν και οι δύο ακόμα δομές της γλώσσας, η `incase` και η `forcase`, ενδιαφέρον κυρίως για την υλοποίησή τους και όχι για τις προγραμματιστικές τους δυνατότητες. Ας ξεκινήσουμε με την πιο απλή.

2.1.1 Η δομή επανάληψης *while*

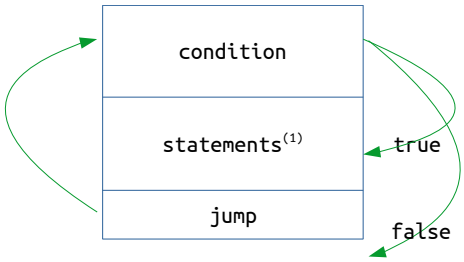
Η γραμματική της `while` είναι η ακόλουθη:

```
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 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 εκτε-



Σχήμα 2.1: Παραγωγή ενδιάμεσου κώδικα για το `while`

λούνται οι εντολές των `statements` πρέπει ο έλεγχος να μεταβαίνει στην αρχή της λογικής συνθήκες, 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 ετικέτα 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 πως λειτουργεί ο ενδιάμεσος κώδικας, μπορούμε να το κάνουμε με το σχήμα 2.1. Στο σχήμα αυτό φαίνεται η σειρά με την οποία τοποθετήθηκαν τα τμήματα προγράμματος στη μνήμη, το πως θα συμπληρωθούν οι λίστες `true` και `false` και η ανάγκη και το πως θα υλοποιηθεί το άλμα για να επαναεκτιμηθεί 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} τοποθετιόντουσαν 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 σαν `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 αναλυτή βρίσκεται πριν το άνοιγμα της παρένθεσης, είτε μετά, η επόμενη τετράδα που θα δημιουργηθεί θα είναι η πρώτη τετράδα της `condition`. Αυτή 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 επαναααποτιμηθεί 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
```

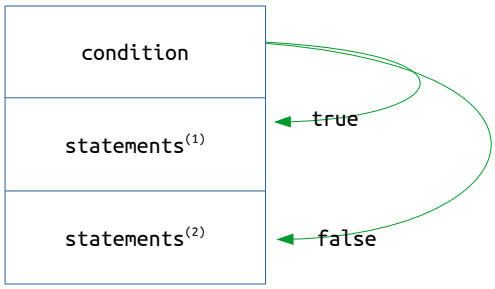
Ενώ η δημιουργία ενδιάμεσου κώδικα για την `while` θα τελειώσει με το `backpatch()` που θα συμπληρώσει τη λίστα `false` με το `nextQuad()`, δηλαδή το 105. Ο ολοκληρωμένος κώδικας ακολουθεί:

```
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.

2.1.2 Η δομή απόφασης *if*

Η γραμματική της `if` είναι η ακόλουθη:



Σχήμα 2.2: Εσφαλμένη παραγωγή κώδικα για το `if`

```
ifStat      →   if ( condition ) statements(1)  elsePart
elsePart    →   else statements(2)
            |   ε
```

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`$^{(1)}$, άρα το `backpatch()` στο `nextQuad()` θα πρέπει να γίνει πριν την παραγωγή της πρώτης εντολής των `statements`$^{(1)}$.

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`$^{(1)}$, 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`$^{(2)}$, ακριβώς αυτό που θέλαμε.

Σύμφωνα με τα παραπάνω, η γραμματική γίνεται:

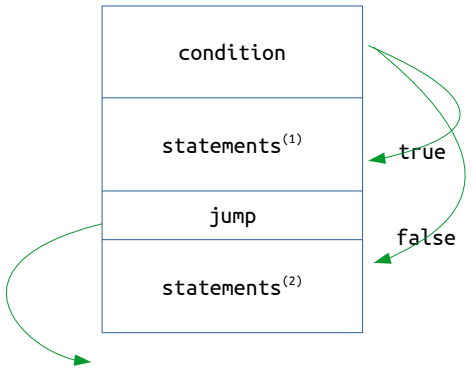
```
ifStat      →   if ( condition ) {p1} statements(1)  {p2}
                elsePart
elsePart    →   else statements(2)
            |   ε

{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 2.2, πρώτα θα παραχθεί και τα 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 ένα τρόπο για να παρακάμπτουμε τις εντολές `statements`$^{(2)}$, όταν οι εντολές `statements`$^{(1)}$ έχουν εκτελεστεί.

Η λύση εικονίζεται στο σχήμα 2.3. Ανάμεσα στις `statements`$^{(1)}$ και `statements`$^{(2)}$ παρεμβάλουμε ένα `jump`, 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 `jump` αυτό θα εκτελέσει ένα άλμα στην αμέσως επόμενη εντολή. Κακός κώδικας, αλλά δεν χρειάζεται να ανησυχούμε. Αυτό είναι παιχνιδάκι για τη φάση της βελτιστοποίησης που θα ακολουθήσει.

Πώς όμως θα τοποθετήσουμε εκεί το `jump`; Με μία `genQuad()`. Αλλά ποιο θα είναι το τελευταίο τελού-



Σχήμα 2.3: Παραγωγή κώδικα για το `if`

μενο, η τετράδα στην οποία θα γίνει το `jump`; 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 που θα γίνει το άλμα, αφού αυτό θα γίνει σε κώδικα που δεν έχει ακόμα παραχθεί. Στη `while` τα πράγματα ήταν εύκολα, διότι το `jump` 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 την 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 προηγουμένως 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 `jump,_,_,_`, 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 την μετάφραση της `if`. Το κατάλληλο σημείο είναι το τέλος του κανόνα `ifStat`, τοποθετώντας είναι 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 κώδικάς 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`$^{(1)}$ ή/και τα `statements`$^{(2)}$ είναι το κενό. Αν τα `statements`$^{(1)}$ είναι κενό 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`$^{(1)}$ είναι κενό ο κώδικας λειτουργεί. Αν τώρα το `statements`$^{(2)}$ είναι κενό, τότε η `backpatch()` του `false` θα κάνει άλμα σε ότι 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`$^{(1)}$ και `statements`$^{(2)}$ 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.

2.1.3 Η δομή επιλογής *switchcase*

Η γραμματική της `switchcase` είναι η ακόλουθη:

```
switchcaseStat  →   switchcase
                    ( case ( condition ) statements(1)  )*
                    default statements(2)
```

Ελέγχονται ένα ένα τα `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`, που 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()`.

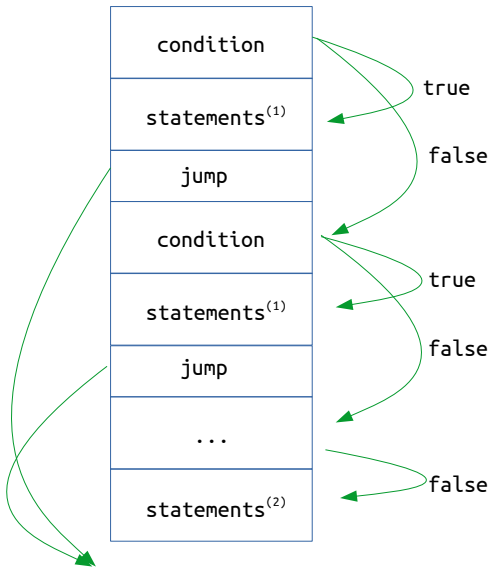
Όταν η συνθήκη αποτιμηθεί ως ψευδής, τότε ο έλεγχος πρέπει να μεταφερθεί στο επόμενο `condition` ή αν δεν υπάρχει επόμενο `condition` στην αρχή των `statements`$^{(2)}$. Ένα κατάλληλο τέτοιο σημείο είναι ακριβώς μετά το `statements`$^{(1)}$ και ακριβώς πριν την παρένθεση. Αν υπάρχει επόμενη `condition`, τότε ο κώδικάς 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`$^{(1)}$ 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 κάθε μία 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`$^{(1)}$, μετά από κάθε ένα από τα οποία πρέπει να δημιουργηθεί ένα `jump,_,_,_`. Όλα αυτά τα `jump` πρέπει να τοποθετηθούν σε μία λίστα η οποία θα συμπληρωθεί στο τέλος



Σχήμα 2.4: Παραγωγή κώδικα για το `switchcase`

της δομής. Ας ονομάσουμε αυτή τη λίστα `exitList`. Σε κάθε ένα `case` λοιπόν, μετά την `statements`$^{(1)}$ 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`$^{(1)}$ 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 2.4.

Στο σημείο αυτό θα πρέπει να ελέγξουμε αν το σχέδιο ενδιάμεσου κώδικα λειτουργεί σωστά και για τις περιπτώσει που τα `statements`$^{(1)}$ και `statements`$^{(2)}$ είναι κενά. Πράγματι, αν το `statements`$^{(1)}$ 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`$^{(2)}$ είναι κενό, ο έλεγχος θα "κυλήσει" έξω από την `switccase`. 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, _
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 στη συνέχεια δημιουργείται η τετράδαQ

```
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`(2). 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, θα 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
```

2.1.4 Η δομή επιλογής-επανάληψης *forcase*

Η γραμματική της `forcase` ακολουθεί:

```
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`$^{(1)}$ (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*$^{(2)}$. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την `forcase`.

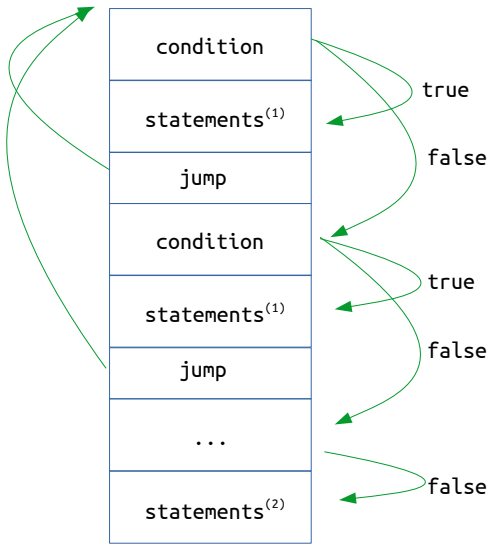
Σκεπτόμενοι όμοια με τις προηγούμενες δομές που αναλύσαμε, πρέπει να διαχειριστούμε τις λίστες `true` και `false` και πρέπει να υλοποιήσουμε το άλμα προς τα πίσω στην πρώτη συνθήκη. Πρέπει, επίσης, να προσέξουμε και το `default`.

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()
```

Θα χειριστούμε τα `condition` ακριβώς όπως στην `switchcase`. Και εδώ όταν ένα `condition` είναι αληθές, ο έλεγχος πρέπει να μεταφερθεί στην αρχή των `statements`$^{(1)}$. 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 σημείο για να τοποθετηθεί η `backpatch()` που θα συμπληρώνει την `condition.true` στο `nextQuad()` είναι ακριβώς πριν το `statements`$^{(1)}$. Αντίστοιχα, ένα κατάλληλο για το `backpatch()` της `false` είναι ακριβώς μετά το `statements`$^{(1)}$ και ακριβώς πριν την παρένθεση. Αν υπάρχει επόμενη `condition`, τότε ο κώδικάς του `condition` ακολουθεί το `statements`$^{(1)}$. Αν δεν υπάρχει επόμενο `condition`, τότε ο κώδικας που ακολουθεί το `statements`$^{(1)}$ είναι το `statements`$^{(2)}$.



Σχήμα 2.5: Παραγωγή κώδικα για το `forcase`

```
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`$^{(1)}$. Άρα ακριβώς μετά την `statements`$^{(1)}$, 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','_','_','_')
        backpatch(condition.false,nextQuad())
```

Σχηματικά, η λειτουργία του σχεδίου ενδιάμεσου κώδικα απεικονίζεται στο σχήμα 2.5.

Ακολουθεί παράδειγμα μετατροπής αρχικού κώδικα σε ενδιάμεσου 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
```

2.1.5 Η δομή πολλαπλής επιλογής-επανάληψης *incase*

Άς 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`, εξετάζοντας τες κατά σειρά. Για κάθε μία 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`, έαν 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 σαν πιο χαρακτηριστικό παράδειγμα το *if* με την επιλογή του `else`. Όταν κάπου 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`. Αλλιώς μεταβαίνει στην αρχή της `incase`. Η πληροφορία έαν 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 παράγαμε, 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 μεταγλωττιστής θα δημιουργήσει για το 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 κώδικας που έχουμε σαν βάση μας είναι το εξής:

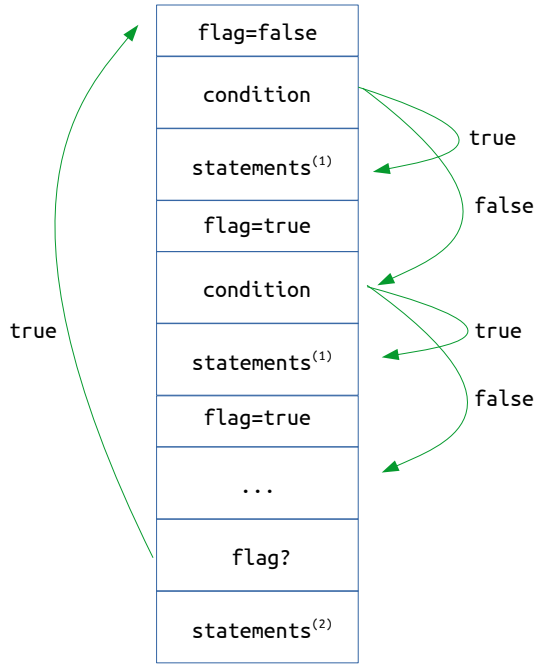
```
forcaseStat   →    incase  {p1}
                   ( case ( condition ) {p2}
                       statements(1)  {p3} )*
                   default statements(2)

{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 της 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 την `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:

```
forcaseStat   →    incase  {p1}
                   ( case ( condition ) {p2}
                       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 2.6.



Σχήμα 2.6: Παραγωγή κώδικα για το `incase`

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
```

2.2 Συναρτήσεις και διαδικασίες

Η ενδιάμεση αναπαράσταση είναι μία γλώσσα, η οποία δεν χρησιμοποιεί σύνθετες προγραμματιστικές δομές. Θα τη χαρακτηρίζαμε, όμως, σαν 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 εξαλείφεται 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 αναφερθούμε στην ενότητα αυτή. Αντίθετα με το τι θα περίμενε κανείς, η κλήση των συναρτήσεων απαιτεί πολύπλοκο χειρισμό, αλλά έναν απλό μετασχηματισμό, ώστε να είναι δυνατόν να συμβολιστεί στον ενδιάμεσο κώδικα. Η πολύ δουλειά μεταφέρεται για τη φάση του τελικού κώδικα.

Θα λειτουργήσουμε με παραδείγματα, 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 ξεκινήσεις από την κλήση της και θα επιστρέφει πίσω, μέχρι να τελειώσουν τα `par`. Ο μετασχηματισμός αυτός είναι μονοσήμαντος. Μπορούμε από την *C-imple* να πάμε στον ενδιάμεσο κώδικα και από τον ενδιάμεσο κώδικα πίσω στην *C-imple*.

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 το σκοπό αυτό, 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 σαν `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. Η:

```
return (x)
```

αντιστοιχεί στη δημιουργία μία 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, _, _
```

2.3 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. Ίσως 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 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, έπεται αυτόν της P11, διότι η εμφάνιση του κώδικα της P1 έπεται 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 σαν 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, την 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, έχουμε ακόμα μία εκχώρηση στο X και μία return:

```
18 :    :=, b, _, X
19 :    ret, e, _, _
20 :    end_block, P1, _, _
```

Ο ενδιάμεσος κώδικας για την P1 ολοκληρωμένος, ακολουθεί:

```
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 στο 22 `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 23, 25 που θα κάνουν άλμα στο 26. Ας δούμε μέχρι στιγμής πως 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(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 κώδικάς 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, _, _
```

2.4 Ένα πιο σύνθετο παράδειγμα, με φωλιασμένες δομές

Ας δούμε και ένα πιο σύνθετο παράδειγμα, όπου οι φωλιασμένες δομές θέλουν πολύ 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 ενδιαφέρουσα 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 ότι 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`. Όταν 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 είναι:

```
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 στο 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()` 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 του 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 παρακάμπτουν `else`:

```
11 :     jump, _, _, 17
15 :     jump, _, _, 17
```

Σημειώνουμε την 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, _, _, _
```

Το 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 στον κώδικά μας 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 που jump 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 πολλώς φωλιασμένων βρόχων, ας θυμηθούμε τι έχουμε αφήσει ασυμπλήρωτο στον κώδικα που έχουμε ήδη παράγει.

Η τετράδα 5 έχει ένα μη συμπληρωμένο jump]. Το jump αυτό οφείλεται στο false του while (a+b<1 and b<5). Όμοια κα 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 μετάφρασης μου 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 βρόχος 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, _, _
```

2.5 Οι κλάσεις της φάσης της παραγωγής του ενδιάμεσου κώδικα

Η παραγωγή του ενδιάμεσου κώδικα, παρότι είναι το πιο δύσκολο τμήμα της μεταγλώττισης από αυτά που μέχρι τώρα έχουμε δει, δεν προσθέτει πολλές κλάσεις στο διάγραμμα κλάσεων του μεταγλωττιστή.

Δύο προφανείς κλάσεις είναι οι κλάσεις για την τετράδα (`Quad`) και για τον δείκτη σε τετράδα (`QuadPointer`).

Η λίστα των τετράδων αποτελεί το πρόγραμμα σε ενδιάμεση αναπαράσταση και είναι τύπου `QuadList` ενώ οι λίστες ετικετών τετράδων `QuadPointerList`. Οι σχέσεις σύνθεσης ανάμεσα στις λίστες και στα στοιχεία τους είναι προφανής.