Συντακτικός αναλυτής

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

Φεβρουάριος 2022

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

Έτσι, ο συντακτικός αναλυτής παίρνει σαν είσοδο μία ακολουθία από λεκτικές μονάδες. Στην έξοδό μπορούμε να θεωρήσουμε ότι δίνει το συντακτικό δέντρο που αντιστοιχεί στο αρχικό πρόγραμμα. Στην πραγματικότητα δεν επιστρέφει κανένα δέντρο, αλλά διαπερνά το αρχικό πρόγραμμα και, πέρα από την αναγνώριση σφαλμάτων, δίνει το περιβάλλον πάνω στο οποίο θα βασιστεί η επόμενη φάση, η παραγωγή ενδιάμεσου κώδικα, αλλά και όλη η υπόλοιπη μεταγλώττιση. Η λειτουργία της συντακτικής ανάλυσης εικονίζεται στο σχήμα 1.1.

Σχήμα 1.1: Ο συντακτικός αναλυτής στη διαδικασία της μεταγλώττισης.

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

1.1 Ενεργοποίηση κανόνων με πρόβλεψη

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

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

και αφού έχει αποτιμηθεί η συνθήκη του `if` ως ψευδής, τα `statements`(1) δεν έχουν εκτελεστεί, η γραμματική βρίσκεται στο δίλημμα αν θα ενεργοποιήσει τον κανόνα `elsepart` ή αν θα επιλέξει τον εναλλακτικό δρόμο της κενής συμβολοσειράς. Αν η λεκτική μονάδα `else` είναι πράγματι η επόμενη λεκτική μονάδα προς αναγνώριση στην είσοδο, τότε ο συντακτικός αναλυτής θα επιλέξει να ενεργοποιήσει τον κανόνα `elsepart`. Σε αντίθετη περίπτωση θα θεωρήσει ότι αναγνώρισε την κενή συμβολοσειρά.

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

```
statement     →    assignStat
              |    ifStat
              |    whileStat
              |     ...
              |    inputStat
              |    printStat
              |    ε
```

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

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

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

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

Στον κανόνα `term`, αρχικά αναγνωρίζεται ένα `factor`. Στη συνέχεια, το άστρο του Kleene επιτρέπει την ύπαρξη ενός ή περισσοτέρων ακόμα `factor` χωρισμένους από τα σύμβολα * ή /. Από τον κανόνα `term` αυτό δεν είναι άμεσα φανερό. Για να το δούμε θα πρέπει να κοιτάξουμε μέσα στον λεκτικό κανόνα `MUL_OP`. Έχοντας αναγνωρίσει τον πρώτο `factor`, η επιλογή αν θα αναζητήσουμε δεύτερο `factor` ή όχι θα καθοριστεί από το

αν η επόμενη προς αναγνώριση λεκτική μονάδα στην είσοδο είναι μία από τις * ή /. Όμοια, η έξοδος από το βρόχο που δημιουργεί το άστρο του Kleene καθορίζεται επίσης από την επόμενη στην είσοδο λεκτική μονάδα, αφού μετά από την αναγνώριση ενός `factor`, αν στην είσοδο έχουμε ένα εκ των * και / θα ενεργοποιηθεί ο πάλι βρόχος και θα αναζητήσουμε ακόμα ένα `factor`.

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

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

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

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

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

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

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

1.2 Υλοποίηση συναρτήσεων αναδρομικής κατάβασης

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

- Για κάθε κανόνα της γραμματικής υλοποιούμε μία συνάρτηση. Δίνουμε στη συνάρτηση το ίδιο όνομα με τον κανόνα ή τουλάχιστον έναν όνομα που να αντιστοιχίζεται μονοσήμαντα στον κανόνα. Στην

παρούσα φάση η συνάρτηση αυτή δεν χρειάζεται να παίρνει κάτι σαν όρισμα ή να επιστέφει κάποιο αποτέλεσμα στη συνάρτηση που την κάλεσε.

- Γράφουμε τον κώδικα που αντιστοιχεί στο δεξί μέλος του κανόνα.
  - Για κάθε μη τερματικό σύμβολο, καλούμε την αντίστοιχη συνάρτηση που έχουμε υλοποιήσει.
  - Για κάθε τερματικό σύμβολο ελέγχουμε ότι πράγματι το τερματικό σύμβολο αυτό συμπίπτει με την επόμενη λεκτική μονάδα στην είσοδο.
    * Αν πράγματι το τερματικό σύμβολο που συναντάμε στη γραμματική συμπίπτει με την επόμενη λεκτική μονάδα στην είσοδο, τότε *καταναλώνουμε* τη λεκτική μονάδα, θεωρούμε την αναγνώριση μέχρι το σημείο αυτό επιτυχής και προχωρούμε στην αναγνώριση της επόμενης λεκτικής μονάδας.
    * Αν το τερματικό σύμβολο που συναντάμε στη γραμματική δεν συμπίπτει με την επόμενη λεκτική μονάδα στην είσοδο τότε:
      · Αν μία από τις εναλλακτικές που δίνει η γραμματική είναι το κενό ακολουθούμε αυτήν την επιλογή, ελπίζοντας το σύμβολο που ζητούμε να αναγνωριστεί από τον κανόνα που θα ακολουθήσει.
      · Αν η γραμματική δεν δίνει σαν επιλογή το κενό, τότε έχουμε φτάσει σε κατάσταση σφάλματος, εμφανίζεται το κατάλληλο μήνυμα λάθους και τερματίζεται η μετάφραση.

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

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

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

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

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

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

Σύμφωνα με τον αλγόριθμο που παρουσιάστηκε παραπάνω, η μέθοδος θα ονομαστεί `program()`. Με την εκκίνηση κάθε μεθόδου η επόμενη λεκτική μονάδα προς αναγνώριση είναι ήδη διαθέσιμη. Η πρώτη ενέργεια

που πρέπει να γίνει από τον συντακτικό αναλυτή μέσα στη μέθοδο `program()` είναι να ελέγξει αν το `token.recognized_string` αντιστοιχεί στο `program`. Αν πράγματι το `token.recognized_string` αντιστοιχεί στο `program`, τότε το καταναλώνουμε, διαβάζουμε δηλαδή την επόμενη λεκτική μονάδα που ακολουθεί στην είσοδο, καλώντας την `get_token()`, και συνεχίζουμε την αναγνώριση αναζητώντας το `ID`, πάντα σύμφωνα με τη γραμματική.

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

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

- όταν ζητάμε να αναγνωρίσουμε τη δεσμευμένη λέξη `program`, αλλά κάτι άλλο εμφανίζεται αντ' αυτής. Εδώ μπορούμε να εμφανίζουμε ένα αρκετά περιγραφικό μήνυμα: *keyword "program" expected in line 1. All programs should start with the keyword "program". Instead, the word '...' appeared*

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

- όταν το τελευταίο σύμβολο του προγράμματος δεν είναι η τελεία. Εδώ μπορούμε να έχουμε ένα μήνυμα: *Every program should end with a fullstop, fullstop at the end is missing.*

- όταν μετά την τελεία ακολουθεί κάτι άλλο, εκτός από το τέλος του αρχείου. Το μήνυμα *No characters are allowed after the fullstop indicating the end of the program.* μπορεί να υποδείξει με σαφήνεια το σφάλμα που προέκυψε.

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

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

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

```
            else:
              self.error(...)
         else:
            self.error(...)
      else:
         self.error(...)
```

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

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

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

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

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

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

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

Ο κώδικας Python που υλοποιεί τα παραπάνω, ακολουθεί:

```
def ifStat():
   global token
   token = self.get_token()  #consume if
   if token.recognized_string == '(':
      token = self.get_token()
```

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

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

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

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

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

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

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

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

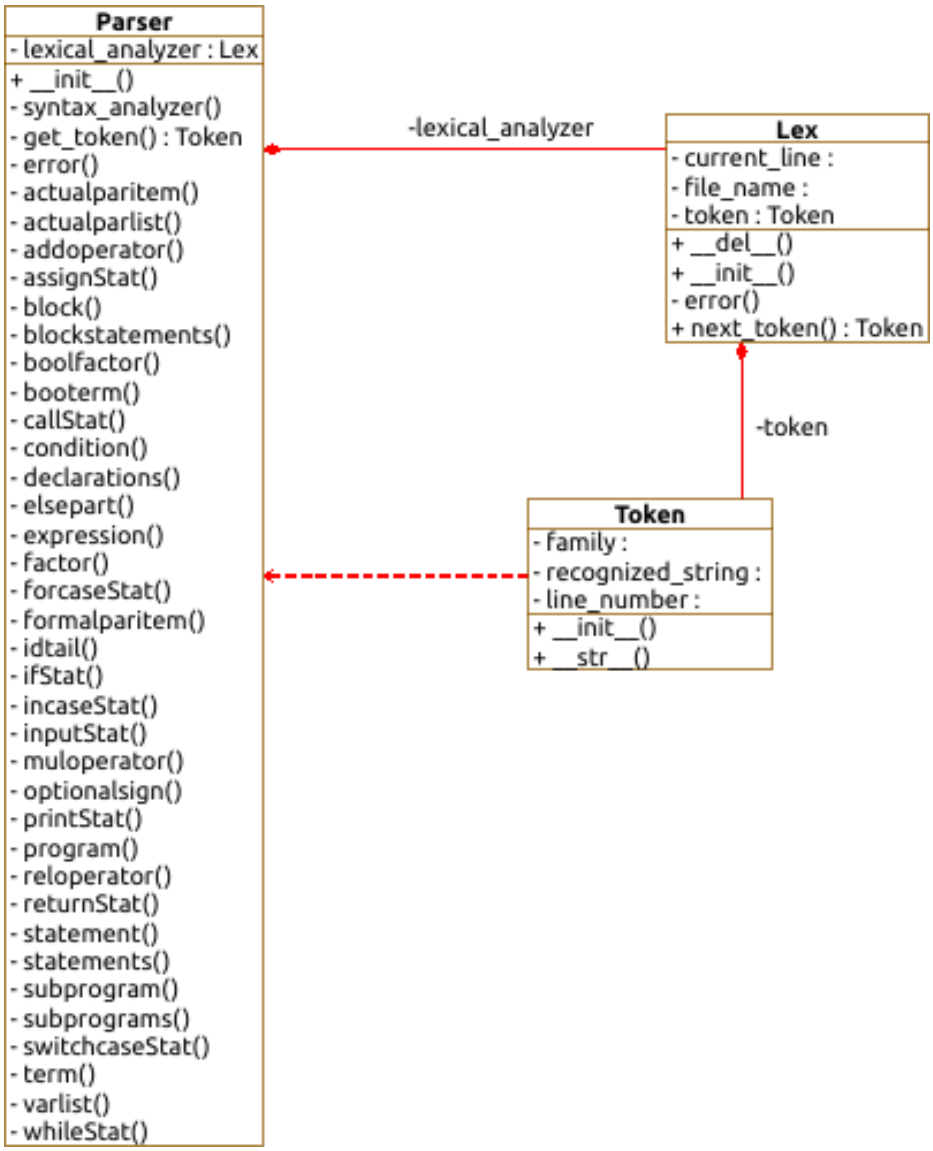
1.3 Ο συντακτικός αναλυτής στο διάγραμμα κλάσεων

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

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

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

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



Σχήμα 1.2: Διάγραμμα κλάσεων λεκτικού-συντακτικού αναλυτή