Μεταγλωττιστές 2021-22

Συντακτική Ανάλυση Αναδρομικής Κατάβασης
(τροποποιήσεις/προσαρμογές)

Αναδρομική Κατάβαση (Recursive Descent)

- Μέθοδος συντακτικής ανάλυσης για γραμματικές LL(1)
  - Συναρτήσεις μη τερματικών συμβόλων που καλούν η μία την άλλη αναδρομικά
    - Το θεωρητικό αυτόματο στοίβας υλοποιείται από το runtime stack των συναρτήσεων
  - Μέσα στις συναρτήσεις μπορούμε να βάλουμε οποιοδήποτε άλλο κομμάτι κώδικα
- Τροποποιήσεις και προσαρμογές
  - Στον κώδικα της αναδρομικής κατάβασης
  - Μπορούμε να επιτύχουμε custom συντακτική ανάλυση
  - Ή ακόμα και ανάλυση γραμματικών που δεν είναι LL(1)

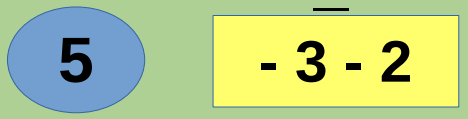
Παράδειγμα #1: Προσεταιριστικότητα (associativity) αριθμητικών τελεστών

```
Stmt_list   → Stmt Stmt_list | ε
Stmt        → id = Expr | print Expr
Expr        → Term Term_tail
Term_tail   → Addop Term Term_tail | ε
Term        → Factor Factor_tail
Factor_tail → Multop Factor Factor_tail | ε
Factor      → (Expr) | id | number
Addop       → + | -
Multop      → * | /
```

- Η κλασσική «by the book» υλοποίηση αναδρομικής κατάβασης δεν χειρίζεται σωστά εκφράσεις όπως 5 - 3 - 2 ή 20 / 4 / 2
  - Οι τελεστές - και / είναι αριστερά προσεταιριστικοί (left associative)
  - Οι πιο πάνω εκφράσεις πρέπει να υπολογιστούν ως (5 - 3) - 2 και (20 / 4) / 2
- Αντιθέτως, ο συντακτικός αναλυτής τις χειρίζεται ως 5 - (3 – 2) και 20 / (4 / 2)

5 - 3 - 2

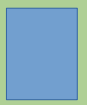
Expr → Term Term_tail



Term_tail → Addop Term Term_tail | ε

Term_tail → Addop Term Term_tail | ε

Term_tail → Addop Term Term_tail | ε



5 - 3 - 2

Expr → Term Term_tail

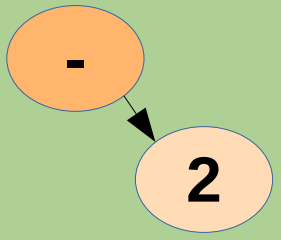
5 | -3-2

Term_tail → Addop Term Term_tail | ε

- | 3 | -2

Term_tail → Addop Term Term_tail | ε

- | 2 |



Term_tail → Addop Term Term_tail | ε

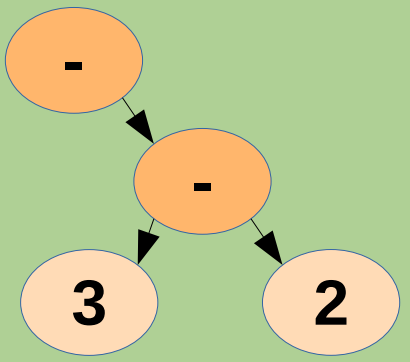
5 - 3 - 2

```
Expr        → Term Term_tail
```

5 | -3-2

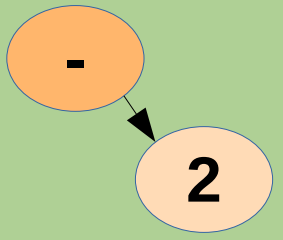
```
Term_tail   → Addop Term Term_tail | ε
```

- | 3 | -2



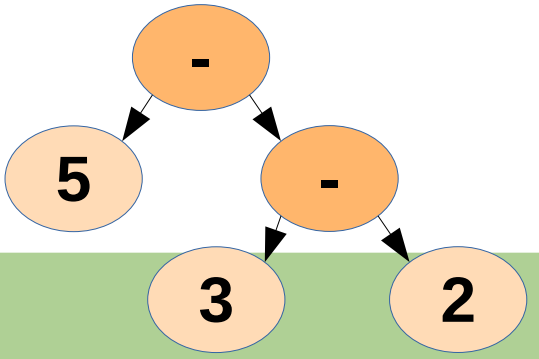

```
Term_tail   → Addop Term Term_tail | ε
```

- | 2 |




```
Term_tail   → Addop Term Term_tail | ε
```

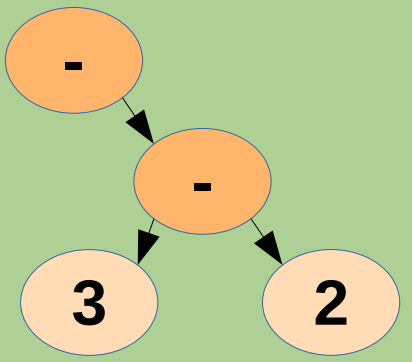
5 - 3 - 2



Expr → Term Term_tail

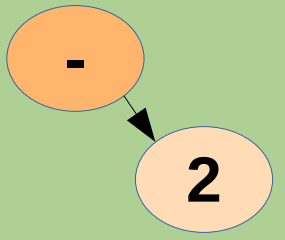
5 -3-2

Term_tail → Addop Term Term_tail | ε



- 3 -2

Term_tail → Addop Term Term_tail | ε



- 2

Term_tail → Addop Term Term_tail | ε

Ο αρχικός κώδικας για το Expr()

```
    def Expr(self):
        if self.next_token in ('(','id','number'):
            # Expr → Term Term_tail
            self.Term()
            self.Term_tail()
        else:
            raise ParseError
```

```
def Term_tail(self):
        if self.next_token in ('+','-'):
            # Term_tail → Addop Term Term_tail
            self.Addop()
            self.Term()
            self.Term_tail()
        elif self.next_token in ('id','print',')',None):
            # Term_tail → e
            return
        else:
            raise ParseError
```

Η ιδέα

```
Stmt_list    → Stmt Stmt_list | ε
Stmt         → id = Expr | print Expr
Expr         → Term (Addop Term)*
Term_tail    → Addop Term Term_tail | ε
Term         → Factor (Multop Factor)*
Factor_tail → Multop Factor Factor_tail | ε
Factor       → (Expr) | id | number
Addop        → + | -
Multop       → * | /
```

- Όσο (while) υπάρχουν τελεστές ίδιου επιπέδου προτεραιότητας, συνεχίζουμε την κατανάλωση εισόδου (συνεπώς, επεξεργασία από αριστερά προς τα δεξιά)
  - Επεκταμένη σύνταξη γραμματικής, (...)* = «μηδέν ή περισσότερες φορές»
  - Οι κανόνες ..._tail απορροφούνται

Ο νέος κώδικας για το Expr()

```
def Expr(self):
    if self.next_token in ('(','id','number'):
        # Expr → Term (Addop Term)*
        self.Term()
        while self.next_token in ('+','-'):
            self.Addop()
            self.Term()

    else:
        raise ParseError
```

- Ο «πειραγμένος» κώδικας μπορεί να εξυπηρετήσει τελεστές με αριστερή μόνο (-, /) ή με αριστερή/δεξιά προσεταιριστικότητα (+,*)
- Αν υπάρχουν τελεστές με δεξιά μόνο προσεταιριστικότητα (π.χ. η ύψωση σε δύναμη) θα πρέπει να χρησιμοποιηθεί ο αρχικός κώδικας

Παράδειγμα #2: “Dangling else”

```
Stmt        → if expr Stmt
            | if expr Stmt else Stmt
            | other
```

- Για συντομία τα expr και other εμφανίζονται ως τερματικά σύμβολα
  - Σε μια πραγματική γραμματική θα είναι μη τερματικά
- Είναι η γραμματική LL(1);
  - Αρχικά θα πρέπει να φύγει ο «κοινός παράγοντας» (κοινό πρόθεμα) από τους δύο κανόνες if του Stmt

Παράδειγμα #2: “Dangling else”

```
Stmt        → if expr Stmt Rest_if
            | other
Rest_if     → else Stmt | ε
```

- Είναι τώρα η γραμματική LL(1);
  - Βρείτε τα FIRST και FOLLOW sets

Σύνολα FOLLOW	Σύνολα FIRST	Κανόνες	
			→ Stmt #
#,else	if other	Stmt	→ if expr Stmt Rest_if | other
#,else	else ε	Rest_if	→ else Stmt | ε

FIRST/FOLLOW conflict!

Η γραμματική δεν είναι LL(1)!

```
def Rest_if(self):
    if self.next_token =='else':
        # Rest_if → else Stmt
        self.match('else')
        self.Stmt()
    elif self.next_token in ('else',None):
        # Rest_if → ε
        return
    else:
        raise ParseError
```

Σε ποιο if ανήκει το else;

```
if expr if expr other else other
```

```
if expr if expr other Rest_if Rest_if
```

Η ιδέα

```
Stmt -> if expr Stmt (else Stmt)?
      | other
```

- Εάν (if) μετά το if expr Stmt ακολουθεί else, τότε προσπαθούμε να ταιριάξουμε το else Stmt (συνεπώς το else θα συνδυαστεί με το τελευταίο if που έχουμε ήδη επεξεργαστεί, δηλ. το κοντινότερο if)
  - Επεκταμένη σύνταξη γραμματικής, (…)? = «προαιρετικά»

Ο νέος κώδικας για το Stmt()

```
def Stmt(self):
    if self.next_token=='if':
        # Stmt -> if expr Stmt (else Stmt)?
        self.match('if')
        self.match('expr')
        self.Stmt()
        # test if optional part (else Stmt) follows
        if self.next_token=='else':
            self.match('else')
            self.Stmt()

    elif self.next_token=='other':
        # Stmt -> other
        self.match('other')

    else:
        raise ParseError
```