ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

ΠΛΗ 402
ΘΕΩΡΙΑ ΥΠΟΛΟΓΙΣΜΟΥ

Εργασία Προγραμματισμού
Λεκτική και Συντακτική Ανάλυση της Γλώσσας Προγραμματισμού Lambda

Διδάσκων
Μιχαήλ Γ. Λαγουδάκης

Εργαστήριο
Γεώργιος Ανέστης
Νεκτάριος Μουμουτζής

Εαρινό Εξάμηνο 2024

Τελευταία ενημέρωση: 28.05.2024

1 Εισαγωγή

Η εργασία προγραμματισμού του μαθήματος **«ΠΛΗ 402 – Θεωρία Υπολογισμού»** έχει ως στόχο τη βαθύτερη κατανόηση της χρήσης και εφαρμογής θεωρητικών εργαλείων, όπως οι κανονικές εκφράσεις και οι γραμματικές χωρίς συμφραζόμενα, στο πρόβλημα της μεταγλώττισης (compilation) γλωσσών προγραμματισμού. Συγκεκριμένα, η εργασία αφορά στη σχεδίαση και υλοποίηση των αρχικών σταδίων ενός μεταγλωττιστή (compiler) για τη φανταστική γλώσσα προγραμματισμού **Lambda**, η οποία περιγράφεται αναλυτικά παρακάτω.

Πιο συγκεκριμένα, θα δημιουργηθεί ένας **source-to-source compiler** (trans-compiler ή transpiler), δηλαδή ένας τύπος μεταγλωττιστή ο οποίος παίρνει ως είσοδο τον πηγαίο κώδικα ενός προγράμματος σε μια γλώσσα προγραμματισμού και παράγει τον ισοδύναμο πηγαίο κώδικα σε μια άλλη γλώσσα προγραμματισμού. Στην περίπτωσή μας ο πηγαίος κώδικας εισόδου θα είναι γραμμένος στη φανταστική γλώσσα προγραμματισμού **Lambda** και ο παραγόμενος κώδικας θα είναι στη γνωστή γλώσσα προγραμματισμού **C**.

Για την υλοποίηση της εργασίας θα χρησιμοποιήσετε τα εργαλεία **flex** και **bison**, τα οποία είναι διαθέσιμα ως ελεύθερο λογισμικό, και τη γλώσσα προγραμματισμού **C**.

Η εργασία περιλαμβάνει δύο τμήματα:

- Υλοποίηση **λεκτικού αναλυτή** για τη γλώσσα **Lambda** με χρήση **flex**
- Υλοποίηση **συντακτικού αναλυτή** για τη γλώσσα **Lambda** με χρήση **bison**
  - Μετατροπή του κώδικα της **Lambda** σε κώδικα **C** με χρήση ενεργειών του **bison**

Παρατηρήσεις

Η εργασία θα εκπονηθεί **ατομικά**. Η τυφλή αντιγραφή (plagiarism), ακόμη και από παλαιότερες εργασίες, μπορεί να διαπιστωθεί πολύ εύκολα και οδηγεί σε μηδενισμό.

Για την εκπόνηση της εργασίας μπορούν να χρησιμοποιηθούν υπολογιστές του Μηχανογραφικού Κέντρου μέσω της υπηρεσίας VDI ή προσωπικοί υπολογιστές. Τα εργαλεία flex και bison είναι διαθέσιμα σε οποιαδήποτε διανομή Linux. Μπορείτε επίσης πολύ εύκολα να εγκαταστήσετε ένα Ubuntu Linux terminal σε Windows μέσω από το Microsoft Store (ψάξτε για την εφαρμογή Ubuntu) και στη συνέχεια να εγκαταστήσετε τα εργαλεία flex και bison.

Η παράδοση της εργασίας θα γίνει **ηλεκτρονικά** μέσα από την ιστοσελίδα του μαθήματος στο **eClass**. Το παραδοτέο αρχείο τύπου archive (`.zip` ή `.rar` ή `.tar`) θα πρέπει να εμπεριέχει όλα τα απαραίτητα αρχεία, σύμφωνα με τις προδιαγραφές της εργασίας.

Η εργασία πρέπει να παραδοθεί **εντός** της προθεσμίας. Εκπρόθεσμες εργασίες δεν γίνονται δεκτές. Μη παράδοση της εργασίας οδηγεί αυτόματα σε αποτυχία στο μάθημα.

Η αξιολόγηση της εργασίας θα περιλαμβάνει **εξέταση καλής λειτουργίας** του παραδοτέου προγράμματος, σύμφωνα με τις προδιαγραφές, καθώς και **προφορική εξέταση** κατά την οποία θα πρέπει να εξηγήσετε κάθε τμήμα του κώδικα που έχετε παραδώσει και να απαντήσετε στις σχετικές ερωτήσεις. Η εξέταση θα γίνει σε ημέρες και ώρες που θα ανακοινωθούν.

Υπενθύμιση: ο βαθμός της εργασίας θα πρέπει να είναι τουλάχιστον **40/100** για την επαρκή εκπόνησή της. Συνεπώς, δεν αρκεί να παραδώσετε μόνο ένα τμήμα της.

2 Η γλώσσα προγραμματισμού Lambda

Η περιγραφή της γλώσσας **Lambda** παρακάτω ακολουθεί τη γενική μορφή περιγραφής μιας γλώσσας προγραμματισμού. Πιθανότατα περιέχει και στοιχεία τα οποία δεν εντάσσονται στη λεκτική ή συντακτική ανάλυση. Είναι ευθύνη σας να αναγνωρίσετε αυτά τα στοιχεία και να τα αγνοήσετε κατά την ανάπτυξη του αναλυτή σας. Κάθε πρόγραμμα σε γλώσσα **Lambda** είναι ένα σύνολο από *λεκτικές μονάδες,* οι οποίες είναι διατεταγμένες βάσει *συντακτικών κανόνων*, όπως περιγράφονται παρακάτω.

2.1 Λεκτικές μονάδες

Οι λεκτικές μονάδες αποτελούν το λεξιλόγιο (vocabulary) της γλώσσας προγραμματισμού **Lambda** και χωρίζονται σε κατηγορίες, οι οποίες περιγράφονται αναλυτικότερα παρακάτω:

- Τα ***αναγνωριστικά*** *(identifiers)* που χρησιμοποιούνται για ονόματα μεταβλητών και συναρτήσεων και αποτελούνται από ένα πεζό ή κεφαλαίο γράμμα του λατινικού αλφαβήτου, ακολουθούμενο από μια σειρά μηδέν ή περισσοτέρων πεζών ή κεφαλαίων γραμμάτων του λατινικού αλφαβήτου, ψηφίων του δεκαδικού συστήματος ή χαρακτήρων υπογράμμισης (underscore). Τα αναγνωριστικά δεν πρέπει να συμπίπτουν με τις λέξεις κλειδιά.

  Παραδείγματα αναγνωριστικών: `x` `y1` `angle` `myValue` `Distance_02`

- Οι ***λέξεις κλειδιά*** *(keywords)*, οι οποίες είναι δεσμευμένες (reserved) λέξεις και δεν μπορούν να χρησιμοποιηθούν ως αναγνωριστικά ή να επανορισθούν, οι οποίες είναι οι παρακάτω:

integer	**scalar**	**str**	**bool**	**True**
False	**const**	**if**	**else**	**endif**
for	**in**	**endfor**	**while**	**endwhile**
break	**continue**	**not**	**and**	**or**
def	**enddef**	**main**	**return**	**comp**
endcomp	**of**			

  Οι λέξεις κλειδιά είναι case-sensitive, δηλαδή δεν μπορούν να γραφούν με κεφαλαία γράμματα ή με συνδυασμό πεζών και κεφαλαίων γραμμάτων, παρά μόνο όπως δίνονται παραπάνω. Σε διαφορετική περίπτωση, θεωρούνται identifiers.

- Οι **ακέραιες σταθερές** *(integer constants)*, που αποτελούνται από ένα ή περισσότερα ψηφία του δεκαδικού συστήματος χωρίς περιττά μηδενικά στην αρχή.

  Παραδείγματα ακέραιων σταθερών: `0` `42` `1233503` `10001`

- Οι ***πραγματικές σταθερές*** *(floating-point constants)*, που αποτελούνται από ένα ακέραιο μέρος, ένα κλασματικό μέρος και ένα προαιρετικό εκθετικό μέρος. Το ακέραιο μέρος αποτελείται από ένα ή περισσότερα ψηφία (του δεκαδικού συστήματος) χωρίς περιττά μηδενικά στην αρχή. Το κλασματικό μέρος αποτελείται από τον χαρακτήρα της υποδιαστολής (`.`) ακολουθούμενο από ένα ή περισσότερα ψηφία του δεκαδικού συστήματος. Τέλος, το εκθετικό μέρος αποτελείται από το πεζό ή κεφαλαίο γράμμα e ή E, ένα προαιρετικό πρόσημο **+** ή **−** και ένα ή περισσότερα ψηφία του δεκαδικού συστήματος χωρίς περιττά μηδενικά στην αρχή.

  Παραδείγματα πραγματικών σταθερών: `42.4` `4.2e1` `0.420E+2` `42000.0e-3`

- Οι ***λογικές σταθερές*** *(boolean constants)*, που είναι οι λέξεις-τιμές **`True`** και **`False`**.
- Οι ***σταθερές συμβολοσειρές*** *(constant strings)*, που αποτελούνται από μια ακολουθία κοινών χαρακτήρων ή χαρακτήρων διαφυγής (escape characters) μέσα σε διπλά εισαγωγικά. Κοινοί χαρακτήρες θεωρούνται η τελεία, το κόμμα, το κενό, οι χαρακτήρες a-z, A-Z, 0-9 και τα σύμβολα -, +, *, /, :, _, $, %, !, #, @, &, ~, ^, (,). Οι χαρακτήρες διαφυγής ξεκινούν με το \ (backslash) και είναι μόνο αυτοί που περιγράφονται στον εξής πίνακα.

Χαρακτήρας Διαφυγής	Περιγραφή
`\n`	χαρακτήρας αλλαγής γραμμής (line feed)
`\t`	χαρακτήρας στηλοθέτησης (tab)
`\r`	χαρακτήρας επιστροφής στην αρχή της γραμμής

\\	χαρακτήρας \ (backslash)
\"	χαρακτήρας " (διπλό εισαγωγικό)

Μια σταθερή συμβολοσειρά δεν μπορεί να εκτείνεται σε περισσότερες από μία γραμμές του αρχείου εισόδου. Πρέπει να περιέχεται πλήρως σε μία μόνο γραμμή.

Ακολουθούν παραδείγματα έγκυρων συμβολοσειρών:

```
"M"   "\n"   "\""   "abc" "Route 66"
"Hello world!\n"   "Item:\t\"Laser Printer\"\nPrice:\t$142\n"
```

- Τους ***τελεστές*** *(operators)*, οι οποίοι είναι οι παρακάτω:

  αριθμητικοί τελεστές: **+ - * / % ****

  σχεσιακοί τελεστές: **== != < <= > >=**

  λογικοί τελεστές: **and or not**

  τελεστές προσήμου: **+ -**

  τελεστής ανάθεσης: **= += -= *= /= %= :=**

- Τους ***διαχωριστές*** *(delimiters)*, οι οποίοι είναι οι παρακάτω:

  ; ( ) , [ ] : .

Εκτός από τις λεκτικές μονάδες που προαναφέρθηκαν, ένα πρόγραμμα **Lambda** μπορεί επίσης να περιέχει και στοιχεία που αγνοούνται (δηλαδή αναγνωρίζονται, αλλά δεν γίνεται κάποια ανάλυση):

Κενούς χαρακτήρες *(white space)*, δηλαδή ακολουθίες αποτελούμενες από κενά διαστήματα (space), χαρακτήρες στηλοθέτησης (tab), χαρακτήρες αλλαγής γραμμής (line feed) ή χαρακτήρες επιστροφής στην αρχή της γραμμής (carriage return).

Σχόλια γραμμής *(line comments),* το οποία ξεκινούν με τους χαρακτήρες **--** και εκτείνονται ως το τέλος της τρέχουσας γραμμής.

Το αρχείο εισόδου μπορεί να περιέχει προαιρετικά δηλώσεις ορισμού της μορφής

@defmacro <identifier> <string>

οι οποίες καταλαμβάνουν μία ξεχωριστή γραμμή αποκλειστικά η κάθε μία και μπορούν να βρίσκονται οπουδήποτε μέσα στην είσοδο. Για παράδειγμα, κάποιες δηλώσεις ορισμού θα ήταν οι εξής:

```
@defmacro my2pi (2*3.14159265)

@defmacro myname "John"

@defmacro N -100
```

Τέτοιες δηλώσεις ορισμού δεν μεταφέρονται ως λεκτικές μονάδες στη συντακτική ανάλυση, αλλά πρέπει να διεκπεραιωθούν πλήρως κατά τη λεκτική ανάλυση. Αυτό σημαίνει ότι αν στην πορεία αναγνωρισθεί κάποια λεκτική μονάδα που ταιριάζει με τον identifier κάποιου **defmacro** που έχει προηγηθεί, τότε αυτή θα πρέπει να αντικατασταθεί από το string που ορίζεται στα δεξιά του **defmacro** για το συγκεκριμένο identifier, να συνεχιστεί η λεκτική ανάλυση μ' αυτό το νέο string και μετά να επανέλθει στο αρχείο εισόδου. Εάν υπάρχουν πολλαπλές δηλώσεις ορισμού με τον ίδιο identifier, τότε σε κάθε σημείο του προγράμματος ισχύει η πιο πρόσφατη δήλωση σε σχέση με το εν λόγω σημείο.

2.2 Συντακτικοί κανόνες

Οι συντακτικοί κανόνες της γλώσσας **Lambda** ορίζουν την ορθή σύνταξη των λεκτικών μονάδων της.

i) Προγράμματα

Ένα πρόγραμμα **Lambda** μπορεί να βρίσκεται μέσα σε ένα αρχείο με κατάληξη `.la` και αποτελείται από τα παρακάτω συστατικά τα οποία παρατίθενται μ' αυτή τη σειρά και **τερματίζονται** με το χαρακτήρα `;`

Δηλώσεις σύνθετων τύπων	(προαιρετικά)
Δηλώσεις σταθερών	(προαιρετικά)
Δηλώσεις μεταβλητών	(προαιρετικά)
Ορισμοί συναρτήσεων	(προαιρετικά)
Κύρια δομική μονάδα	(υποχρεωτικά)

Η κύρια δομική μονάδα είναι η συνάρτηση `main`, η οποία δεν λαμβάνει ορίσματα και θεωρείται ότι δεν επιστρέφει κάποια τιμή. Αυτή η συνάρτηση είναι το σημείο εκκίνησης για την εκτέλεση του προγράμματος και είναι της μορφής:

```
def main():
    σώμα της main
enddef;
```

Ένα απλό παράδειγμα έγκυρου αρχείου `.la` είναι το παρακάτω:

```
def main():
    x: integer;
    x = 1 + 2 + 3 + 4;
    writeInt(x);
enddef;
```

ii) Τύποι δεδομένων

Η γλώσσα **Lambda** υποστηρίζει τους εξής τύπους δεδομένων (data types):

`integer`	ακέραιοι αριθμοί
`scalar`	πραγματικοί αριθμοί
`str`	σειρά από χαρακτήρες
`bool`	λογικές τιμές
`comp`	σύνθετος τύπος, ο οποίος σχηματίζεται με τη χρήση άλλων τύπων, καθώς και συναρτήσεων (αναλυτική περιγραφή στην ενότητα vi) παρακάτω)
`[arrayLength]:type`	τύπος πίνακα μεγέθους `arrayLength` με στοιχεία τύπου `type`, όπου το `type` είναι ένας από τους παραπάνω τύπους δεδομένων και το `arrayLength` θα πρέπει να είναι ακέραια σταθερά με θετική τιμή. Παράδειγμα: `sequenceOf10Ints[10]: int`;
`[]:type`	τύπος πίνακα, ως άνω, όπου το `arrayLength` μπορεί να παραλειφθεί, όταν για παράδειγμα δηλώνουμε τον τύπο της παραμέτρου μιας συνάρτησης.

iii) Μεταβλητές

Οι δηλώσεις μεταβλητών αποτελούνται από ένα ή περισσότερα αναγνωριστικά μεταβλητών, χωρισμένα με κόμμα, με το διαχωριστικό `:` στο τέλος, ακολουθούμενο από έναν τύπο δεδομένων. Ακολουθεί ένα παράδειγμα

δηλώσεων μεταβλητών, όπου χρησιμοποιείται και ένας σύνθετος τύπος, ο οποίος έχει οριστεί πριν τη δήλωση των μεταβλητών:

```
comp Coordinates:
      #latitude, #longitude: scalar;
endcomp;

i, j: scalar;
s1, s2: str;
n1: integer;
s, ss: str;
test: bool;
grades[5]: scalar;
directions[4]: str;
w, z: Coordinates;
```

iv) Σταθερές

Οι σταθερές δηλώνονται χρησιμοποιώντας τη λέξη κλειδί **const** και θα πρέπει υποχρεωτικά να τους δίνεται αρχική τιμή με χρήση του τελεστή ανάθεσης **=**. Παράδειγμα:

```
const pi = 3.14152: scalar;
```

v) Συναρτήσεις

Η συνάρτηση (function) είναι μια δομική μονάδα, η οποία αποτελείται από τα παρακάτω συστατικά, τα οποία παρατίθενται μ' αυτή τη σειρά:

```
def όνομα συνάρτησης ( δηλώσεις παραμέτρων ) -> τύπος επιστροφής :
      δηλώσεις τοπικών μεταβλητών και σταθερών      (προαιρετικά)
      εντολές                                       (υποχρεωτικά)
      return έκφραση                                (προαιρετικά)
enddef
```

Η δήλωση μιας συνάρτησης ξεκινά με τη λέξη κλειδί **def**, ακολουθούμενη από το όνομα της συνάρτησης, και ακολουθούν οι παράμετροί της μέσα σε παρενθέσεις. Προαιρετικά, εφόσον η συνάρτηση επιστρέφει κάποια τιμή, ακολουθούν οι χαρακτήρες **->** και ο τύπος του επιστρεφόμενου αποτελέσματος, **ο οποίος δεν μπορεί να είναι πίνακας**. Στη συνέχεια, υπάρχει ο χαρακτήρας **:** και το σώμα της συνάρτησης που τερματίζεται με τη λέξη κλειδί **enddef**.

Το σώμα μιας συνάρτησης περιέχει μία ακολουθία από δηλώσεις τοπικών μεταβλητών, δηλώσεις σταθερών και εντολές. Στο σώμα μπορεί να περιέχεται εντολή **return**, εφόσον η συνάρτηση επιστρέφει κάποια τιμή. Οι παρενθέσεις στις παραμέτρους είναι υποχρεωτικές, ακόμη κι αν μία συνάρτηση δεν έχει παραμέτρους. Ο τύπος επιστροφής μπορεί να παραλειφθεί, εάν η συνάρτηση δεν χρειάζεται να επιστρέφει κάποια συγκεκριμένη τιμή. Μία συνάρτηση μπορεί να περιέχει εντολή **return**, χωρίς να επιστρέφει κάποια τιμή, όταν δεν έχει δηλωθεί ο τύπος επιστροφής της. Ακολουθούν παραδείγματα ορισμού έγκυρων συναρτήσεων:

```
def f1(b: integer, e: integer) -> integer:
      return b ** e;
enddef;

def f2(s[]:str) -> integer:
      return 100;
enddef;

def f3():return; enddef;
```

```
def f4(prompt: str, msg:str):
      writeStr(prompt);
      writeStr(msg);
enddef;
```

Η γλώσσα **Lambda** υποστηρίζει ένα σύνολο προκαθορισμένων συναρτήσεων, οι οποίες βρίσκονται στη διάθεση του προγραμματιστή για χρήση οπουδήποτε μέσα στο πρόγραμμα. Παρακάτω, δίνονται οι επικεφαλίδες τους:

```
def readStr() -> str
def readInteger() -> integer
def readScalar() -> scalar
def writeStr(s: str)
def writeInteger(n: integer)
def writeScalar(n: scalar)
def write(fmt: str, ...)
```

vi) Δηλώσεις σύνθετων τύπων (`comp`)

Η γλώσσα **Lambda** παρέχει τη δυνατότητα δήλωσης σύνθετων τύπων με τη χρήση της δεσμευμένης λέξης `comp`. Η γενική μορφή της δήλωσης ενός σύνθετου τύπου έχει ως εξής:

`comp` *όνομα σύνθετου τύπου*`:`
δηλώσεις μεταβλητών (μέλη - τα ονόματα των μεταβλητών αρχίζουν υποχρεωτικά με τον χαρακτήρα #)
συναρτήσεις (μέθοδοι)
`endcomp;`

Παράδειγμα δήλωσης σύνθετου τύπου:

```
comp Circle:
      #x, #y: scalar;
      #radius: scalar;
      def area() -> scalar: return 3.14 * (#radius ** 2); enddef;
      def perimeter() -> scalar: return 2 * 3.14 * #radius; enddef;
      def move(dx: scalar, dy: scalar): #x=#x+dx; #y=#y+dx; enddef;
endcomp;
```

Οι συναρτήσεις ενός σύνθετου τύπου ονομάζονται *μέθοδοι*, θεωρούνται *μέρος* του τύπου και έχουν πρόσβαση στις μεταβλητές-μέλη του τύπου.

Αφού οριστεί ένας σύνθετος τύπος, μπορεί στη συνέχεια να χρησιμοποιηθεί όπως όλοι οι τύποι. Για παράδειγμα:

```
c1, c2: Circle;

def totalArea(c[]: Circle, size: integer) -> scalar:
      i: integer;
      sum: scalar;
      sum = 0;
      for i in [0:size-1]:
            sum = sum + c[i].area();
      endfor;
      return sum;
enddef;
```

vii) Εκφράσεις

Οι εκφράσεις (expressions) είναι ίσως το πιο σημαντικό κομμάτι μιας γλώσσας προγραμματισμού. Οι βασικές μορφές εκφράσεων είναι οι σταθερές, οι μεταβλητές οποιουδήποτε τύπου και οι κλήσεις συναρτήσεων. Σύνθετες μορφές εκφράσεων προκύπτουν με τη χρήση τελεστών και παρενθέσεων.

Οι τελεστές της **Lambda** διακρίνονται σε τελεστές με ένα όρισμα και τελεστές με δύο ορίσματα. Από τους πρώτους, ορισμένοι γράφονται πριν το όρισμα (prefix) και ορισμένοι μετά (postfix), ενώ οι δεύτεροι γράφονται πάντα μεταξύ των ορισμάτων (infix).

Η αποτίμηση των ορισμάτων των τελεστών με δύο ορίσματα γίνεται από αριστερά προς τα δεξιά. Στον παρακάτω πίνακα ορίζεται η προτεραιότητα και η προσεταιριστικότητα των τελεστών της **Lambda**. Οι τελεστές εμφανίζονται κατά **φθίνουσα** προτεραιότητα. Όσοι τελεστές βρίσκονται στην ίδια γραμμή έχουν την ίδια προτεραιότητα. Σημειώστε ότι μπορούν να χρησιμοποιηθούν παρενθέσεις σε μια έκφραση για να δηλωθεί η επιθυμητή προτεραιότητα.

Τελεστές	Περιγραφή	Προσεταιριστικότητα

`.`
`()`
`[]`
	πρόσβαση μέλους σύνθετου τύπου
	κλήση συνάρτησης ή μεθόδου
	πρόσβαση σε στοιχείο πίνακα
		από αριστερά προς τα δεξιά

`**`	ύψωση σε δύναμη	από δεξιά προς τα αριστερά
`+ -`	τελεστές προσήμου (μοναδιαίοι)	από δεξιά προς τα αριστερά
`* / %`	πολλαπλασιασμός, διαίρεση, υπόλοιπο διαίρεσης	από αριστερά προς τα δεξιά
`+ -`	πρόσθεση, αφαίρεση	από αριστερά προς τα δεξιά
`< <=` `> >=`	σχεσιακοί τελεστές ανισότητας	από αριστερά προς τα δεξιά
`== !=`	ισότητα, ανισότητα	από αριστερά προς τα δεξιά
`not`	λογικός τελεστής άρνησης (μοναδιαίος)	από δεξιά προς τα αριστερά
`and`	λογική σύζευξη	από αριστερά προς τα δεξιά

`or`	λογική διάζευξη	από αριστερά προς τα δεξιά

`=`
`+= -= *=`
`/= %= :=`
	τελεστές ανάθεσης	από δεξιά προς τα αριστερά

Ακολουθούν παραδείγματα σωστών εκφράσεων:

```
-a                      -- αντίθετος της μεταβλητής a
a + b * (b / a)         -- αριθμητική έκφραση
4 + 50.0*x / 2.45       -- αριθμητική έκφραση
(a+1) % cube(b+3)       -- αριθμητική έκφραση με κλήση συνάρτησης
(a <= b) and (d <= c)   -- τελεστές λογικοί με σχεσιακούς
(c+a) != (2*d)          -- τελεστές αριθμητικοί με σχεσιακούς
a + b[(k+1)*2]          -- αριθμητική έκφραση με πίνακα
```

viii) Εντολές

Οι εντολές (statements) που υποστηρίζει η γλώσσα **Lambda** είναι οι ακόλουθες:

- Η εντολή *ανάθεσης* **`v = expr;`** , όπου **`v`** μία μεταβλητή, **`=`** ο τελεστής ανάθεσης και **`expr`** μια έκφραση.
- Η *εντολή ελέγχου* **`if (expr): stmts`**$_1$ **`else: stmts`**$_2$ **`endif;`** Το τμήμα **`else`** είναι προαιρετικό. Το **`expr`** είναι μια έκφραση, ενώ τα **`stmts`**$_1$ και **`stmts`**$_2$ αντιστοιχούν το καθένα σε μηδέν ή περισσότερες εντολές που τερματίζονται με το χαρακτήρα **`;`**
- Η *εντολή επανάληψης* **`for integer_variable in [start:stop:step]: stmts endfor;`** όπου τα **`start, stop, step`**, είναι αριθμητικές εκφράσεις που αντιστοιχούν στην αρχική τιμή, την τελική τιμή και το βήμα μεταβολής της ακέραιας μεταβλητής **`integer_variable`**. Το **`step`** είναι προαιρετικό. Όταν απουσιάζει, το βήμα μεταβολής της ακέραιας μεταβλητής είναι **`+1`**. Το **`stmts`** αντιστοιχεί σε μηδέν ή περισσότερες εντολές που τερματίζονται με τον χαρακτήρα **`;`** και εκτελούνται σε κάθε επανάληψη.
- Η *απλή εντολή συμπαγούς πίνακα επί ακεραίων τιμών* **`new_array := [expr for elm:size] : new_type;`** Το **`expr`** είναι μία έκφραση η οποία περιέχει το **`elm`**, το **`elm`** είναι μια μεταβλητή τύπου **`integer`**, που θα διατρέξει τις ακέραιες τιμές από το **`0`** έως το **`size-1`** και το **`size`** είναι ένας θετικός ακέραιος. Η εντολή δημιουργεί ένα νέο πίνακα **`new_array`** με στοιχεία τύπου **`new_type`**, εφαρμόζοντας την έκφραση **`expr`** πάνω σε κάθε τιμή του **`elm`** και αποθηκεύοντας το αποτέλεσμα στην ανάλογη θέση του πίνακα **`new_array`**.
- Η *εντολή συμπαγούς πίνακα με χρήση άλλου πίνακα* **`new_array := [expr for elm: type in array of size] : new_type;`** Το **`expr`** είναι μία έκφραση, η οποία περιέχει το **`elm`**, το **`elm`** είναι μια μεταβλητή τύπου **`type`**, το **`array`** είναι μια μεταβλητή πίνακα και το **`size`** είναι το μέγεθος του πίνακα. Η εντολή δημιουργεί ένα νέο πίνακα **`new_array`** με στοιχεία τύπου **`new_type`**, εφαρμόζοντας την έκφραση **`expr`** πάνω σε κάθε στοιχείο **`elm`** του πίνακα **`array`**, αποθηκεύοντας το αποτέλεσμα στην ανάλογη θέση του πίνακα **`new_array`**.

- Η *εντολή βρόχου* **`while (expr): stmts endwhile;`** Το **`expr`** είναι μία έκφραση και το **`stmts`** αντιστοιχεί σε μηδέν ή περισσότερες εντολές που τερματίζονται με το χαρακτήρα `;` και εκτελούνται σε κάθε επανάληψη του βρόχου.
- *Η εντολή διακοπής* **`break;`** που προκαλεί την άμεση έξοδο από τον πιο εσωτερικό βρόχο.
- Η *εντολή συνέχισης* **`continue;`** που προκαλεί τη διακοπή της τρέχουσας επανάληψης και την έναρξη της επόμενης επανάληψης του βρόχου μέσα στον οποίο βρίσκεται.
- Η *εντολή επιστροφής* **`return;`** ή **`return expr;`** που τερματίζει (πιθανά, πρόωρα) την εκτέλεση της συνάρτησης μέσα στην οποία βρίσκεται και επιστρέφει, όπου **`expr`** είναι μια (προαιρετική) έκφραση.
- Η *εντολή κλήσης* μιας συνάρτησης **`f(`**$\texttt{expr}_1$**`,...,`**$\texttt{expr}_n$**`);`** όπου **`f`** είναι το όνομα της συνάρτησης και $\texttt{expr}_1$**`,...,`**$\texttt{expr}_n$ είναι εκφράσεις που αντιστοιχούν στα δηλωθέντα ορίσματα της συνάρτησης.
- Η κενή εντολή `;`

2.3 Αντιστοίχιση από την Lambda στη C99

Η C99 είναι η αναθεώρηση του standard της γλώσσας **C** που έγινε το 1999. Στην αναθεώρηση αυτή προστέθηκαν διάφορες χρήσιμες επεκτάσεις στην κάπως παλιά **C89**. Δείτε το αντίστοιχο άρθρο της Wikipedia για παραπάνω λεπτομέρειες. Καθώς η **C99** είναι μια πλούσια γλώσσα, είναι ιδιαίτερα εύκολο να αντιστοιχίσει κανείς προγράμματα της **Lambda** σε προγράμματα της **C99**. Τις λεπτομέρειες της απεικόνισης αυτής θα περιγράψουμε στη συνέχεια.

2.3.1 Αντιστοίχιση τύπων και σταθερών

Οι τύποι της **Lambda** αντιστοιχίζονται με τους τύπους της **C99** με βάση τον παρακάτω πίνακα:

Τύπος της Lambda	Αντιστοιχημένος τύπος της C99
`scalar`	`double`
`integer`	`int`
`str`	`char*`
`bool`	`int`
`array[n]:T`	`TC array[n]`
`[]:T`	`TC*`
`def func(a1:T1, ... ak:Tk) -> type`	`type (*) func(TC1 a1, ... TCk ak)`

όπου **`T`**, **`T1`**, …, **`Tk`** είναι κάποιος τύπος της **Lambda** και **`TC`**, **`TC1`**, …, **`TCk`** ο αντίστοιχος τύπος της **C**.

Στη βάση του παραπάνω πίνακα αντιστοιχίζονται και οι σταθερές της **Lambda** σε σταθερές της **C99**. Για παράδειγμα, οι boolean σταθερές της **Lambda**, `True` και `False`, αντιστοιχίζονται στις ακέραιες τιμές `1` και `0` αντίστοιχα.

Ειδικά για τη δήλωση σύνθετων τύπων με τη χρήση της δεσμευμένης λέξης `comp`, η αντιστοίχιση σε κώδικα **C** θα γίνει ως εξής. Καταρχήν, αναδιατυπώνουμε τη γενική μορφή της δήλωσης σύνθετου τύπου στην **Lambda**:

```
comp type_name:
      #var_name1, #var_name2: type1;
       ...
      def func(a1:T1, …, ak:Tk) -> type: fbody endef;
```

```
  ...
endcomp;
```

στην αντίστοιχη μορφή σε **C**:

```
typedef struct type_name {
       type1 var_name1, var_name2;
       ...
       type (*func) (struct type_name *self, TC1 a1, …, TCk ak);
  ...
} type_name ;

type func (struct type_name *self, TC1 a1, …, TCk ak) {
     αντιστοίχιση κώδικα συνάρτησης fbody
}

type_name ctor_type_name = { .func = func , ... } ;
```

Επιπροσθέτως όταν δηλώνεται μία μεταβλητή ενός σύνθετου τύπου:

```
var_1: type_name;
```

η αντιστοίχιση σε κώδικα **C** θα έχει ως εξής:

```
type_name myvar1 = ctor_type_name;
```

2.3.2 Αντιστοίχιση εντολών συμπαγούς πίνακα

Η γλώσσα **Lambda** υποστηρίζει δύο μορφές εντολών συμπαγούς πίνακα. Η μία μορφή διατρέχει ένα σύνολο ακεραίων αριθμών από το μηδέν έως μια μέγιστη τιμή και χρησιμοποιεί αυτές τις τιμές για να δημιουργήσει έναν καινούργιο πίνακα. Η άλλη μορφή διατρέχει τα στοιχεία ενός υπάρχοντος πίνακα και δημιουργεί έναν καινούργιο. Στη συνέχεια περιγράφουμε αυτές τις δύο μορφές.

Εντολή συμπαγούς πίνακα επί ακεραίων τιμών

Η *εντολή συμπαγούς πίνακα επί ακεραίων τιμών* `new_array := [expr for elm:size] : new_type;` δημιουργεί ένα νέο πίνακα `new_array` με στοιχεία τύπου `new_type` εφαρμόζοντας την έκφραση `expr` πάνω σε κάθε θετικό ακέραιο στο διάστημα από `0` έως στοιχείο `size-1`.

Έτσι, η εντολή:

```
new_array := [expr for elm:size] : new_type;
```

αντιστοιχίζεται στον εξής κώδικα σε **C**:

```
new_type* new_array = (new_type*)malloc(size * sizeof(new_type));
for (int elm = 0; elm < size; ++elm)
      new_array[elm] = expr;
```

Για παράδειγμα ο κώδικας **Lambda**:

```
def main():
      a := [i for i:100]:integer;
enddef;
```

μετατρέπεται στον ακόλουθο κώδικα σε C:

```
int main() {
      int* a=(int*)malloc(100*sizeof(int));
      for(int i=0; i < 100; ++i)  {
```

```
            a[i] = i;
      }
}
```

Εντολή συμπαγούς πίνακα με χρήση άλλου πίνακα

Για την *εντολή συμπαγούς πίνακα με χρήση άλλου πίνακα της μορφής* **new_array := [expr for elm: type in array of size] : new_type;** η αντιστοίχιση σε κώδικα **C** θα γίνει όπως περιγράφεται παρακάτω. Το **expr** είναι μία έκφραση η οποία περιέχει το **elm**. Το **elm** είναι μια μεταβλητή τύπου **type,** το **array** είναι μια μεταβλητή πίνακα και το **size** είναι το μέγεθος του πίνακα **array**. Η εντολή δημιουργεί ένα νέο πίνακα **new_array** με στοιχεία τύπου **new_type** εφαρμόζοντας την έκφραση **expr** πάνω σε κάθε στοιχείο **elm** του πίνακα **array** (δηλαδή τα στοιχεία από τη θέση 0 έως τη θέση **size-1**).

Έτσι, η εντολή:

```
new_array := [expr for elm: type in array of size] : new_type;
```

αντιστοιχίζεται στον εξής κώδικα σε C:

```
new_type* new_array = (new_type*)malloc(size * sizeof(new_type));
for (int array_i = 0; array_i < array_size; ++array_i)
      new_array[array_i] = expr;  -- όπου μέσα στη έκφραση εμφανίζεται το elm
                                  -- αντικαθίσταται από το array[array_i]
```

Για παράδειγμα ο κώδικας **Lambda**:

```
const N = 100: integer;
a[100]: integer;
def main():
      for i in [0:N]:
            a[i] = i;
      endfor;
      half := [ x / 2 for x: integer in a of 100] : scalar;
enddef;
```

μετατρέπεται στον ακόλουθο κώδικα σε C:

```
const int N = 100;
int a[100];
int main() {
      for (int i =0; i<N; i++) {
            a[i] = i;
      }
      double* half=(double*)malloc(100*sizeof(double));
      for (int a_i=0; a_i < 100; ++a_i) {
            half[a_i] = a[a_i] / 2;
      }
}
```

2.3.3 Αντιστοίχιση δομικών μονάδων

Ένα πρόγραμμα της **Lambda** περιλαμβάνει προαιρετικά δηλώσεις σταθερών, μεταβλητών, σύνθετων τύπων, συναρτήσεων και υποχρεωτικά το τμήμα του κυρίως κώδικα, δηλαδή την ειδική συνάρτηση `main` και αντιστοιχεί σε ένα αρχείο `.c` που περιλαμβάνει, δηλώσεις σταθερών, καθολικών μεταβλητών, δομών, συναρτήσεων και την αρχική ρουτίνα `main()`.

Η αντιστοίχιση είναι ως εξής:

Μια **Lambda** μεταβλητή `foo` με τύπο `T`

```
foo, bar: T;
```

αντιστοιχεί σε μεταβλητή με ίδιο όνομα και με τον αντιστοιχισμένο τύπο `TC`

```
TC foo, bar;
```

Μια συνάρτηση της **Lambda** αντιστοιχεί σε συνάρτηση της **C99** με ίδιο όνομα και τους αντιστοιχισμένους τύπους παραμέτρων.

Συνάρτηση της Lambda	Συνάρτηση της C99
`def foo(x1:T1, x2:T2,..., xn:Tn) -> type`	`type foo(TC1 x1, TC2 x2,..., TCn xn)`

Οι εντολές προγράμματος αντιστοιχούνται με προφανή τρόπο.

Οι κλήσεις βιβλιοθήκης θα μπορούσαν να υλοποιηθούν ως εξής:

Κλήση Lambda	Συνάρτηση υλοποίησης σε C99
`readStr() -> str`	Χρησιμοποιείστε την υλοποίηση που σας δίνεται στο αρχείο `lambdalib.h`
`readInteger() -> integer`	
`readScalar() -> scalar`	
`writeStr(s:string)`	
`writeInteger(n:integer)`	
`writeScalar(n:scalar)`	
`write(fmt: str, ...)`	

Οι προκαθορισμένες συναρτήσεις της **Lambda** αντιμετωπίζονται όπως όλες οι άλλες συναρτήσεις. Φροντίστε κατά τη μετατροπή του πηγαίου κώδικα της **Lambda** σε **C** να συμπεριλάβετε (`#include`) στον παραγόμενο **C** κώδικα το αρχείο `lambdalib.h` που σας δίνεται και περιέχει την υλοποίηση των προκαθορισμένων συναρτήσεων της **Lambda** σε **C**.

3 Αναλυτική περιγραφή εργασίας

3.1 Τα εργαλεία

Για να ολοκληρώσετε επιτυχώς την εργασία, χρειάζεται να γνωρίζετε καλά προγραμματισμό σε **C**, **flex** και **bison**. Τα εργαλεία **flex** και **bison** έχουν αναπτυχθεί στο πλαίσιο του προγράμματος GNU και μπορείτε να τα βρείτε σε όλους τους κόμβους του διαδικτύου που διαθέτουν λογισμικό GNU (π.χ. `www.gnu.org`). Περισσότερες πληροφορίες, εγχειρίδια και συνδέσμους για τα δύο αυτά εργαλεία θα βρείτε στην ιστοσελίδα του μαθήματος.

Στο λειτουργικό σύστημα Linux (οποιαδήποτε διανομή) τα εργαλεία αυτά είναι συνήθως ενσωματωμένα. Αν δεν είναι, μπορούν να εγκατασταθούν τα αντίστοιχα πακέτα πολύ εύκολα. Οι οδηγίες χρήσης που δίνονται παρακάτω για τα δύο εργαλεία έχουν δοκιμαστεί στη διανομή Linux Ubuntu. Είναι πιθανόν να υπάρχουν μικροδιαφορές σε άλλες διανομές.

3.2 Προσέγγιση της εργασίας

Για τη δική σας διευκόλυνση στην κατανόηση των εργαλείων που θα χρησιμοποιήσετε, καθώς και του τρόπου με τον οποίο τα εργαλεία αυτά συνεργάζονται, προτείνεται η υλοποίηση της εργασίας σε δύο φάσεις.

1η φάση: Λεκτική Ανάλυση

Το τελικό προϊόν αυτής της φάσης θα είναι ένας Λεκτικός Αναλυτής, δηλαδή ένα πρόγραμμα το οποίο θα παίρνει ως είσοδο ένα αρχείο με κάποιο πρόγραμμα της γλώσσας **Lambda** και θα αναγνωρίζει τις λεκτικές μονάδες (tokens) στο αρχείο αυτό. Η έξοδός του θα είναι μία λίστα από τα tokens που διάβασε και ο χαρακτηρισμός τους. Για παράδειγμα, για είσοδο:

```
i = k + 2;
```

η έξοδος του προγράμματός σας θα πρέπει να είναι

```
token IDENTIFIER: i
token ASSIGN_OP:  =
token IDENTIFIER: k
token PLUS_OP:    +
token CONST_INT:  2
token SEMICOLON:  ;
```

Σε περίπτωση μη αναγνωρίσιμης λεκτικής μονάδας, θα πρέπει να τυπώνεται κάποιο κατάλληλο μήνυμα λάθους στην οθόνη και να τερματίζεται η λεκτική ανάλυση. Για παράδειγμα, για τη λανθασμένη είσοδο:

```
i = k ^ 2;
```

η έξοδος του προγράμματός σας θα πρέπει να είναι

```
token IDENTIFIER: i
token ASSIGN_OP:  =
token IDENTIFIER: k

Unrecognized token ^ in line 46: i = k ^ 2;
```

όπου 46 είναι ο αριθμός της γραμμής μέσα στο αρχείο εισόδου, όπου βρίσκεται η συγκεκριμένη εντολή, συμπεριλαμβανομένων των γραμμών σχολίων.

Για να φτιάξετε έναν Λεκτικό Αναλυτή θα χρησιμοποιήσετε το εργαλείο flex και τον compiler gcc. Δώστε `man flex` στη γραμμή εντολών για να δείτε το manual του flex ή ανατρέξτε στο PDF αρχείο που βρίσκεται στο eClass. Τα αρχεία με κώδικα flex έχουν προέκταση `.l`. Για να μεταγλωττίσετε και να τρέξετε τον κώδικά σας ακολουθήστε τις οδηγίες που δίνονται παρακάτω.

1. Γράψτε τον κώδικα flex σε ένα αρχείο με προέκταση `.l`, π.χ. `mylexer.l`.
2. Μεταγλωττίστε, γράφοντας `flex mylexer.l` στη γραμμή εντολών.
3. Δώστε `ls` για να δείτε το αρχείο `lex.yy.c` που παράγεται από τον flex.
4. Δημιουργήστε το εκτελέσιμο με `gcc -o mylexer lex.yy.c -lfl`
5. Αν δεν έχετε λάθη στο `mylexer.l`, παράγεται το εκτελέσιμο `mylexer`.
6. Εκτελέστε με `./mylexer < example.la`, για είσοδο `example.la`.

Κάθε φορά που αλλάζετε το `mylexer.l` θα πρέπει να κάνετε όλη τη διαδικασία:

```
flex mylexer.l
gcc -o mylexer lex.yy.c -lfl
./mylexer < example.la
```

Επομένως, είναι καλή ιδέα να φτιάξετε ένα script ή ένα makefile για να κάνει όλα τα παραπάνω αυτόματα.

2η φάση: Συντακτική Ανάλυση και Μετάφραση

Το τελικό προϊόν αυτής της φάσης θα είναι ένας Συντακτικός Αναλυτής και Μεταφραστής της **Lambda** σε **C**, δηλαδή ένα πρόγραμμα το οποίο θα παίρνει ως είσοδο ένα αρχείο με κάποιο πρόγραμμα της γλώσσας **Lambda** και θα αναγνωρίζει αν αυτό το πρόγραμμα ακολουθεί τους συντακτικούς κανόνες της **Lambda**. Στην έξοδο θα παράγει το πρόγραμμα που αναγνώρισε, στη γλώσσα **C**, εφόσον το πρόγραμμα που δόθηκε είναι συντακτικά σωστό, διαφορετικά θα εμφανίζεται ο αριθμός γραμμής, όπου διαγνώσθηκε το πρώτο λάθος, το περιεχόμενο της γραμμής με το λάθος και *προαιρετικά* ένα κατατοπιστικό μήνυμα διάγνωσης. Για παράδειγμα, για τη λανθασμένη είσοδο

```
...
i = k + 2 * ;
...
```

το πρόγραμμά σας θα πρέπει να τερματίζει με ένα από τα παρακάτω μηνύματα λάθους

```
Syntax error in line 46: i = k + 2 * ;
Syntax error in line 46: i = k + 2 * ;    (expression expected)
```

όπου 46 είναι ο αριθμός της γραμμής μέσα στο αρχείο εισόδου, όπου βρίσκεται η συγκεκριμένη εντολή, συμπεριλαμβανομένων των γραμμών σχολίων.

Για να φτιάξετε ένα συντακτικό αναλυτή και μεταφραστή θα χρησιμοποιήσετε το εργαλείο bison και τον compiler gcc. Δώστε `man bison` για να δείτε το manual του bison. Τα αρχεία με κώδικα bison έχουν προέκταση `.y`. Για να μεταγλωττίσετε και να τρέξετε τον κώδικά σας ακολουθήστε τις οδηγίες που δίνονται παρακάτω.

1. Υποθέτουμε ότι έχετε ήδη έτοιμο το λεκτικό αναλυτή στο `mylexer.l`.
2. Γράψτε τον κώδικα bison σε αρχείο με προέκταση `.y`, π.χ. `myanalyzer.y`.
3. Για να ενώσετε το flex με το bison πρέπει να κάνετε τα εξής:

   Βάλτε τα αρχεία `mylexer.l` και `myanalyzer.y` στο ίδιο directory.

   Βγάλτε τη συνάρτηση `main` από το flex αρχείο και φτιάξτε μια `main` στο bison αρχείο. Για αρχή το μόνο που χρειάζεται να κάνει η καινούρια `main` είναι να καλεί μια φορά την μακροεντολή του bison `yyparse()`. Η `yyparse()` τρέχει επανειλημμένα την `yylex()` και προσπαθεί να αντιστοιχίσει κάθε token που επιστρέφει ο Λεκτικός Αναλυτής στη γραμματική που έχετε γράψει στον Συντακτικό Αναλυτή. Επιστρέφει 0 για επιτυχή τερματισμό και 1 για τερματισμό με συντακτικό σφάλμα.

   Αφαιρέστε τα defines που είχατε κάνει για τα tokens στο flex ή σε κάποιο άλλο `.h` αρχείο. Αυτά θα δηλωθούν τώρα στο bison αρχείο ένα σε κάθε γραμμή με την εντολή `%token`. Όταν κάνετε compile το `myanalyzer.y` δημιουργείται αυτόματα και ένα αρχείο με όνομα `myanalyzer.tab.h`. Το αρχείο αυτό θα πρέπει να το κάνετε include στο αρχείο `mylexer.l` και έτσι ο λεξικός αναλυτής θα καταλαβαίνει τα ίδια tokens με τον συντακτικό αναλυτή.
4. Μεταγλωττίστε τον κώδικά σας με τις παρακάτω εντολές:

   ```
   bison -d –v –r all myanalyzer.y
   flex mylexer.l
   gcc -o mycompiler lex.yy.c myanalyzer.tab.c cgen.c -lfl
   ```

5. Καλέστε τον εκτελέσιμο `mycompiler` για είσοδο `test.la`:

   `./mycompiler < test.la`

Προσοχή! Πρέπει πρώτα να κάνετε compile το `myanalyzer.y` και μετά το `mylexer.l`, γιατί το `myanalyzer.tab.h` γίνεται include στο `mylexer.l`.

Το αρχείο κειμένου `myanalyzer.output` που παράγεται με το flag `-r all` θα σας βοηθήσει να εντοπίσετε πιθανά προβλήματα με conflicts τύπου shift/reduce και reduce/reduce.

Κάθε φορά που αλλάζετε το `mylexer.l` και `myanalyzer.y` θα πρέπει να κάνετε όλη τη διαδικασία. Είναι καλή ιδέα να φτιάξετε ένα script ή ένα makefile για όλα τα παραπάνω.

3.3 Παραδοτέα

Το παραδοτέο για την εργασία του μαθήματος θα περιέχει τα παρακάτω αρχεία (από τη 2η φάση):

`mylexer.l`: Το αρχείο flex.

`myanalyzer.y`: Το αρχείο bison.

`correct1.la, correct2.la`: Δύο σωστά προγράμματα/παραδείγματα της **Lambda**.

`correct1.c, correct2.c`: Τα ισοδύναμα προγράμματα των δύο παραπάνω σε γλώσσα **C**.

Είναι δική σας ευθύνη να αναδείξετε τη δουλειά σας μέσα από αντιπροσωπευτικά προγράμματα.

3.4 Εξέταση

Κατά την εξέταση της εργασίας σας θα ελεγχθούν τα εξής:

Μεταγλώττιση των παραδοτέων προγραμμάτων και δημιουργία του εκτελέσιμου αναλυτή. Ανεπιτυχής μεταγλώττιση σημαίνει ότι παραδώσατε πρόχειρη εργασία, καθώς δεν μπορεί να φανεί η λειτουργία της. Είναι αυτονόητο ότι θα πρέπει να μπορείτε να κάνετε compile τον κώδικά σας μόνοι σας!

Επιτυχής δημιουργία του αναλυτή. Ο βαθμός σας θα επηρεαστεί σημαντικά από τον αριθμό των shift-reduce και reduce-reduce conflicts που εμφανίζονται κατά τη δημιουργία του αναλυτή σας.

*Έλεγχος αναλυτή σε σωστά και λανθασμένα παραδείγματα προγραμμάτων **Lambda***. Θα ελεγχθούν σίγουρα αυτά του Παραρτήματος, αλλά και άλλα άγνωστα σ' εσάς παραδείγματα. Η καλή εκτέλεση τουλάχιστον των γνωστών παραδειγμάτων θεωρείται αυτονόητη. Το αποτέλεσμα της ανάλυσης στο σύνολο των παραδειγμάτων θα επηρεάσει σημαντικά τον βαθμό σας. **Τα παραδείγματα ελέγχονται ως προς την ορθή εκτέλεσή τους, δηλαδή τα παραγόμενα .c αρχεία κώδικα C από το συντακτικό αναλυτή, μεταγλωτίζονται με το μεταγλωττιστή της C (gcc) και στη συνέχεια εκτελούνται τα παραγόμενα εκτελέσιμα αρχεία.**

*Έλεγχος αναλυτή στα δικά σας παραδείγματα προγραμμάτων **Lambda***. Τέτοιοι έλεγχοι θα βοηθήσουν σε περίπτωση που θέλετε να αναδείξετε κάτι από τη δουλειά σας.

Ερωτήσεις σχετικά με την υλοποίηση. Θα πρέπει να είστε σε θέση να εξηγήσετε θέματα σχεδιασμού, επιλογών και τρόπων υλοποίησης, καθώς και κάθε τμήμα του κώδικα που έχετε παραδώσει και να απαντήσετε στις σχετικές ερωτήσεις.

4 Επίλογος

Κλείνοντας, θα θέλαμε να τονίσουμε ότι είναι σημαντικό να ακολουθείτε πιστά τις οδηγίες και να παραδώσετε αποτελέσματα, σύμφωνα με τις προδιαγραφές που έχουν τεθεί. Αυτό είναι κάτι που πρέπει να τηρείτε ως μηχανικοί, για να μπορέσετε στο μέλλον να εργάζεσθε συλλογικά σε μεγάλες ομάδες εργασίας, όπου η συνέπεια είναι το κλειδί για τη συνοχή και την επιτυχία του κάθε έργου.

Στη διάρκεια του εξαμήνου θα δοθούν διευκρινίσεις, όπου χρειάζεται. Για ερωτήσεις, μπορείτε να απευθύνεστε στους υπεύθυνους εργαστηρίου του μαθήματος. Γενικές απορίες καλό είναι να συζητώνται στο χώρο συζητήσεων του μαθήματος στο eClass για να τις βλέπουν και οι συνάδελφοί σας.

Καλή επιτυχία!

ΠΑΡΑΡΤΗΜΑ

5 Παραδείγματα προγραμμάτων της Lambda

5.1 Hello World!

```
-- My first Lambda program. File: myprog.la
@defmacro message "Hello world!\n"

def main():
  writeStr(message);
enddef;
```

Ζητούμενο αποτέλεσμα λεκτικής – συντακτικής ανάλυσης:

```
Token KEYWORD_DEF:          def
Token KEYWORD_BEGIN:        main
Token LEFT_PARENTHESIS:     (
Token RIGHT_PARENTHESIS:    )
Token COLON:                :
Token IDENTIFIER:           writeStr
Token LEFT_PARENTHESIS:     (
Token CONST_STRING:         "Hello World!\n"
Token RIGHT_PARENTHESIS:    )
Token SEMICOLON:            ;
Token KEYWORD_ENDDEF:       enddef
Token SEMICOLON:            ;


Your program is syntactically correct!
```

5.2 Συναρτήσεις της Lambda

Παράδειγμα για την κατανόηση της σύνταξης συναρτήσεων στη γλώσσα **Lambda**.

```
-- File: useless.la
-- A piece of Lambda code for demonstration purposes

@defmacro N -100

a, b: integer;

def cube(i: integer)-> integer:
   return i*i*i;
enddef;

def add(n: integer, k: integer)-> integer:
   j: integer;

   j = (N-n) + cube(k);
   writeInteger(j);
   return j;
enddef;

def main():
   a = readInteger();
   b = readInteger();
   add(a, b); -- Here you can see some dummy comments!
enddef;
```

Το παραπάνω πρόγραμμα θα μπορούσε **ενδεικτικά** να αντιστοιχηθεί σε **C** ως εξής:

```c
#include "lambdalib.h"
/* program */
int a, b;
int cube(int i) { return i * i * i; }
int add(int n, int k) {
   int j;
   j = (-100 - n) + cube(k);
   writeInteger(j);
   return j;
}
int main() {
   a = readInteger();
   b = readInteger();
   add(a, b);
}
```

Στη συνέχεια, το παραπάνω **C** πρόγραμμα μπορεί να μεταφραστεί από τον compiler σε εκτελέσιμο με την εντολή

```
gcc -std=c99 -Wall  myprog.c
```

5.3 Πρώτοι αριθμοί

Το παρακάτω παράδειγμα προγράμματος στη γλώσσα **Lambda** είναι ένα πρόγραμμα που υπολογίζει τους πρώτους αριθμούς μεταξύ **1** και **n**, όπου το **n** δίνεται από το χρήστη.

```
-- File: prime.la

limit, num, counter: integer;

def prime(n: integer)-> bool:
   i: integer;
   result, isPrime: bool;

  if (n < 0):
    result = prime(-n);
  else:
    if (n < 2):
      result = False;
    else:
      if (n == 2):
        result = True;
      else:
        if (n % 2 == 0):
          result = False;
        else:
          i = 3;
          isPrime = True;
          while (isPrime and (i < n / 2) ):
            isPrime = n % i != 0;
            i = i + 2;
          endwhile;
          result = isPrime;
        endif;
      endif;
    endif;
  endif;

  return result;
enddef;

def main():
  limit = readInteger();
  -- 2 is prime
  writeInteger(2);
  writeStr(" ");

  counter = 1; -- count the prime numbers found
  -- check only odd numbers for primality
  for num in [3: limit+1: 2]:
    if (prime(num)):
      counter = counter + 1;
      writeInteger(num);
      writeStr(" ");
    endif;
  endfor;

   writeStr("\nThe total number of primes found is:");
   writeInteger(counter);
   writeStr("\n");
enddef;
```

5.4 Παράδειγμα με συντακτικό λάθος

```
1 -- My first Lambda program
2 const message = "Hello world!\n": str;
3
4 def main():
5     writeStr(message
6 enddef;
```

Ζητούμενο αποτέλεσμα λεκτικής – συντακτικής ανάλυσης:

```
Token KEYWORD_CONST:            const
Token IDENTIFIER:               message
Token ASSIGN_OP:                =
Token CONST_STRING:             "Hello World!\n"
Token COLON:                    :
Token KEYWORD_STR:              str
Token SEMICOLON:                ;
Token KEYWORD_DEF:              def
Token KEYWORD_BEGIN:            main
Token LEFT_PARENTHESIS:         (
Token RIGHT_PARENTHESIS:        )
Token COLON:                    :
Token IDENTIFIER:               writeStr
Token LEFT_PARENTHESIS:         (
Token IDENTIFIER:               message
Token KEYWORD_ENDDEF:           enddef


Syntax error in line 6:         enddef;
```

ή

```
Syntax error in line 6:         enddef;     (Missing parenthesis)
```