ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

ΠΛΗ 402

ΘΕΩΡΙΑ ΥΠΟΛΟΓΙΣΜΟΥ

Εργασία Προγραμματισμού
Λεκτική και Συντακτική Ανάλυση
της Γλώσσας Προγραμματισμού TeaC

Διδάσκων

Μιχαήλ Γ. Λαγουδάκης

Εργαστήριο

Γιώργος Ανέστης

Παράδοση: 15 Μαΐου 2019

Εαρινό Εξάμηνο 2019

Τελευταία ενημέρωση: 2019-03-13

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 **TeaC** (**Tea**ching **C**ompilers), 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 **TeaC** 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 **C**.

Η εργασία περιλαμβάνει δύο τμήματα:

- Υλοποίηση **λεκτικού αναλυτή** για τη γλώσσα **TeaC** με χρήση **flex**
- Υλοποίηση **συντακτικού αναλυτή** για τη γλώσσα **TeaC** με χρήση **bison**
  - Μετατροπή του κώδικα της **TeaC** 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 και προσωπικοί υπολογιστές. Τα εργαλεία flex και bison είναι διαθέσιμα σε οποιαδήποτε διανομή Linux.
- Η παράδοση της εργασίας θα γίνει **ηλεκτρονικά** μέσα από την ιστοσελίδα του μαθήματος στο courses. Το παραδοτέο αρχείο τύπου archive (`.zip` 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 στην Πολυτεχνειούπολη, σε ημέρες και ώρες που θα ανακοινωθούν.
- Το τμήμα της εργασίας που αφορά στη Λεκτική Ανάλυση αντιστοιχεί στο 30% του συνολικού βαθμού της εργασίας και το υπόλοιπο 70% αντιστοιχεί στη Συντακτική Ανάλυση.
- Υπενθύμιση: ο βαθμός της εργασίας θα πρέπει να είναι τουλάχιστον **40/100**. Συνεπώς, δεν αρκεί να παραδώσετε μόνο το τμήμα της Λεκτικής Ανάλυσης.

2 Η γλώσσα προγραμματισμού TeaC

Η περιγραφή της γλώσσας **TeaC** 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 **TeaC** είναι ένα σύνολο από *λεκτικές μονάδες,* οι οποίες είναι διατεταγμένες βάσει *συντακτικών κανόνων*, όπως περιγράφονται παρακάτω.

2.1 Λεκτικές μονάδες

Οι λεκτικές μονάδες της γλώσσας **TeaC** χωρίζονται στις παρακάτω κατηγορίες:

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

`int`	`real`	`bool`	`string`	`true`
`false`	`if`	`else`	`fi`	`while`
`loop`	`pool`	`const`	`let`	`break`
`return`	`not`	`and`	`or`	`start`

Οι λέξεις κλειδιά είναι case-sensitive, δηλαδή δεν μπορείτε να χρησιμοποιήσετε κεφαλαία γράμματα γι' αυτές.

- 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, ψηφίων του δεκαδικού συστήματος ή χαρακτήρων υπογράμμισης (underscore). Τα αναγνωριστικά δεν πρέπει να συμπίπτουν με τις λέξεις κλειδιά .

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

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

  Παραδείγματα: `0` `42` `1284200` `3` `100001`

- Οι ***πραγματικές (θετικές) σταθερές*** *(real positive 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 το χαρακτήρα της υποδιαστολής (**.**) ακολουθούμενο από ένα ή περισσότερα ψηφία του δεκαδικού συστήματος. Τέλος, το εκθετικό μέρος αποτελείται από το πεζό ή κεφαλαίο γράμμα E, ένα προαιρετικό πρόσημο **+** ή **−** και ένα ή περισσότερα ψηφία του δεκαδικού συστήματος χωρίς περιττά μηδενικά στην αρχή.

  Παραδείγματα: `42.0` `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 είναι όλοι οι εκτυπώσιμοι χαρακτήρες πλην των απλών και διπλών εισαγωγικών και του χαρακτήρα **** (backslash). Οι χαρακτήρες διαφυγής ξεκινούν με το **** (backslash) και περιγράφονται στον παρακάτω 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. 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: **`<-`**

Τελεστής	Περιγραφή
`=`	ίσο με (=)
`<`	μικρότερο από (<)
`<=`	μικρότερο ή ίσο με (≤)
`!=`	διάφορο από (≠)
`and`	λογική σύζευξη
`or`	λογική διάζευξη
`not`	λογική άρνηση

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

  `;` **`(`** **`)`** **`,`** **`[`** **`]`** **`<-`** **`:`**

Εκτός από τις λεκτικές μονάδες που προαναφέρθηκαν, ένα πρόγραμμα **TeaC** 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*** *(comments)*, τα οποία ξεκινούν με την ακολουθία χαρακτήρων **`(*`** και τερματίζονται με την πρώτη μετέπειτα εμφάνιση της ακολουθίας χαρακτήρων **`*)`**. Τα σχόλια δεν επιτρέπεται να είναι φωλιασμένα. Στο εσωτερικό τους επιτρέπεται η εμφάνιση οποιουδήποτε χαρακτήρα.
- ***Σχόλια γραμμής*** *(line comments),* το οποία ξεκινούν με την ακολουθία χαρακτήρων **`--`** και εκτείνονται ως το τέλος της τρέχουσας γραμμής.

2.2 Συντακτικοί κανόνες

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

i) Προγράμματα

Ενα πρόγραμμα **TeaC** μπορεί να βρίσκεται μέσα σε ένα αρχείο με κατάληξη **`.tc`** και αποτελείται από τα παρακάτω συστατικά τα οποία παρατίθενται μ' αυτή τη σειρά και διαχωρίζονται μεταξύ τους με το διαχωριστικό **`;`** :

- Δηλώσεις μεταβλητών (προαιρετικά)
- Ορισμοί συναρτήσεων (προαιρετικά)
- Την κύρια δομική μονάδα η οποία είναι η συνάρτηση **`start`** η οποία δεν λαμβάνει ορίσματα και θεωρείται ότι επιστρέφει έναν **`int`**. Αυτή η συνάρτηση είναι το σημείο εκκίνησης για την εκτέλεση του προγράμματος και είναι της μορφής:

```
const start <- () : int => {

}
```

Ενα απλό παράδειγμα έγκυρου αρχείου **.tc** είναι το παρακάτω:

```
const start <- (): int => {
      let x: int;
      x <- 1 + 2 + 3 +4;
      writeInt(x);
      return 0;
}
```

ii) Τύποι δεδομένων

Η γλώσσα **TeaC** υποστηρίζει τέσσερις βασικούς τύπους δεδομένων (data types):

- **int** : ακέραιοι αριθμοί
- **real** : πραγματικοί αριθμοί
- **bool** : λογικές τιμές
- **string** : χαρακτήρες

Η **TeaC** υποστηρίζει επίσης μονοδιάστατους πίνακες αποτελούμενοι από **n** στοιχεία τύπου **type** της μορφής:

- **identifier[n] : type** , όπου το **n** θα πρέπει να είναι ακέραια σταθερά με θετική τιμή και το **type** έγκυρος βασικός τύπος. Παράδειγμα: **listOfNums[10] : int**

iii) Μεταβλητές

Οι δηλώσεις μεταβλητών ξεκινούν με τη λέξη κλειδί **let**. Ακολουθούν ένα ή περισσότερα αναγνωριστικά μεταβλητών χωρισμένα με κόμμα και, τέλος, ο διαχωριστής **:** ακολουθούμενος από ένα τύπο δεδομένων. Πολλαπλές συνεχόμενες δηλώσεις μεταβλητών διαχωρίζονται μεταξύ τους με το διαχωριστικό **,** και εντάσσονται υποχρεωτικά στο ίδιο **let**. Οι μεταβλητές μπορούν να αρχικοποιούνται με κάποια τιμή κατά τη δήλωσή τους με χρήση του τελεστή ανάθεσης **<-**. Ακολουθεί ένα παράδειγμα δηλώσεων μεταβλητών:

```
let i, j <- 10 : int;
let x <- 3.5, y: real;
let s [80]: string;
```

iv) Σταθερές

Οι σταθερές δηλώνονται όπως και οι μεταβλητές χρησιμοποιώντας τη λέξη κλειδί **const** αντί της **let** με τη διαφορά ότι θα πρέπει υποχρεωτικά να τους δίνεται αρχική τιμή.

v) Συναρτήσεις

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

- **const** *όνομα συνάρτησης* **<-** (*δηλώσεις παραμέτρων*) **:** *τύπος επιστροφής* **=>** **{**
  Δηλώσεις τοπικών μεταβλητών (προαιρετικά)
  Εντολές
  return έκφραση (προαιρετικά)
  }

Αναφέρεται κατ' αρχήν το όνομα της συνάρτηση το οποίο δηλώνεται ως σταθερά (**const**), ακολουθεί το σύμβολο της ανάθεσης (assignment) **<-**, οι παράμετροί της μέσα σε παρενθέσεις και ο τύπος του επιστρεφόμενου αποτελέσματος. Οι παρενθέσεις είναι υποχρεωτικές ακόμη κι αν μια συνάρτηση δεν έχει παραμέτρους. Αν η

συνάρτηση δεν περιέχει εντολή **return** ή η εντολή **return** δεν επιστρέφει κάποια τιμή τότε θεωρείται ότι η τιμή επιστροφής είναι ***undefined***. Ακολουθούν παραδείγματα ορισμού έγκυρων συναρτήσεων:

```
const f1 <- (x: real): real { return x * x; }
const f2 <- (s[]: string) : int { return 100; }
const f3 <- (x: real): [] real { }
```

Μια συνάρτηση μπορεί να περιέχει δηλώσεις τοπικών μεταβλητών (προαιρετικά) και τις εντολές της.

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

```
const readString <- (): [] string;
const readInt <- (): int;
const readReal <- (): real;
const writeString <- (s[]: string ) : int;
const writeInt <- (j: int) : int;
const writeReal <- (j: real) : real;
```

vi) 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 **TeaC** 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 δυο ορίσματα γίνεται από αριστερά προς τα δεξιά. Στον παρακάτω πίνακα ορίζεται η προτεραιότητα και η προσεταιριστικότητα των τελεστών της **TeaC**. Προηγούνται οι τελεστές που εμφανίζονται πιο ψηλά στον πίνακα. Οσοι 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	Ορίσματα	Θέση Προσεταιριστικότητα
`not`	Τελεστής λογικής άρνησης	1	prefix, δεξιά
`+ -`	Τελεστές προσήμου	1	prefix, δεξιά
`* / %`	Τελεστές με παράγοντες	2	infix, αριστερή
`+ -`	Τελεστές με όρους	2	infix, αριστερή
`= != < <=`	Σχεσιακοί τελεστές	2	infix, αριστερή
`and`	Λογική σύζευξη	2	infix, αριστερή
`or`	Λογική διάζευξη	2	infix, αριστερή

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)       -- τελεστής αύξησης, κλήση συνάρτησης
(a <= b) and (d <= c)   -- τελεστές λογικοί με σχεσιακούς
a + (c != d)            -- τελεστές αριθμητικοί με σχεσιακούς
a + b[(k+1)*2]          -- αριθμητική έκφραση με πίνακα
```

vii) Εντολές

Οι εντολές που υποστηρίζει η γλώσσα **TeaC** είναι οι ακόλουθες:

- Η *εντολή ανάθεσης* **`v <- expr`**, όπου **`v`** είναι μία μεταβλητή και **`expr`** μια έκφραση.
- Η *εντολή ελέγχου* **`if expr then stmt`**$_1$ **`else stmt`**$_2$ **`fi.`** Το τμήμα του **`else`** είναι προαιρετικό. Το **`expr`** είναι μια έκφραση, ενώ τα **`stmt`**$_1$ και **`stmt`**$_2$ είναι μία η περισσότερες εντολές (statements).
- Οι *εντολές βρόχου* **`while expr loop stmt pool`**. Το **`expr`** είναι μια έκφραση και το **`stmt`** μια ή περισσότερες εντολές.
- Η *εντολή επιστροφής* **`return`**, που τερματίζει (πιθανά, πρόωρα) την εκτέλεση της συνάρτησης στην οποία βρίσκεται και επιστρέφει.
- Η *εντολή κλήσης* μιας διαδικασίας ή συνάρτησης **`f(expr`**$_1$**`,...,expr`**$_n$**`)`**, όπου **`f`** είναι το όνομα της συνάρτησης και **`expr`**$_1$**`,...,expr`**$_n$ είναι εκφράσεις που αντιστοιχούν στα δηλωθέντα ορίσματα.

2.3 Αντιστοίχιση από την TeaC στη C99

Η C99 είναι η αναθεώρηση του standard της γλώσσας **C** που έγινε το 1999. Στην αναθεώρηση αυτή προστέθηκαν διάφορες χρησιμότατες 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 **TeaC** σε προγράμματα της **C99**. Τις λεπτομέρειες της απεικόνισης αυτής θα περιγράψουμε στη συνέχεια.

2.3.1 Αντιστοίχιση τύπων και σταθερών

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

Τύπος της TeaC	Αντιστοιχημένος τύπος της C99
`int`	`int`
`bool`	`int`
`real`	`double`
`array[n]: T`	`T array[n]`
`[] T`	`T*`
`const func <-(a1: T1, ... ak: Tk) : type`	`type (*) func(T1 a1, ... Tk ak)`

όπου **`T`**, **`T1`**, …, **`Tk`** είναι κάποιος τύπος της **TeaC**.

Στη βάση του παραπάνω πίνακα αντιστοιχίζονται και οι σταθερές της **TeaC** σε σταθερές της **C99**. Για παράδειγμα, οι boolean σταθερές της **TeaC**, **`true`** και **`false`**, αντιστοιχίζονται σε ακέραιες τιμές.

2.3.2 Αντιστοίχιση δομικών μονάδων

Ενα πρόγραμμα της **TeaC** περιλαμβάνει προαιρετικά δηλώσεις μεταβλητών, συναρτήσεων και υποχρεωτικά το τμήμα του κυρίως κώδικα, δηλαδή την ειδική συνάρτηση **`start`**, και αντιστοιχεί σε ένα αρχείο **`.c`** που περιλαμβάνει, δηλώσεις global μεταβλητών, συναρτήσεων και την αρχική ρουτίνα **`main()`**.

Η αντιστοίχιση είναι ως εξής:

- Μια **TeaC** μεταβλητή foo με τύπο T αντιστοιχεί σε μεταβλητή με ίδιο όνομα και με τον αντιστοιχημένο τύπο

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

  αντιστοιχεί σε κάτι σαν

  ```
  T foo, bar;
  ```

- Μια συνάρτηση της **TeaC** αντιστοιχεί σε συνάρτηση της **C99** με ίδιο όνομα και τους αντιστοιχημένους τύπους παραμέτρων.

Συνάρτηση της TeaC	Συνάρτηση της C99
`const foo <-(x1, x2: T1,..., xn: Tn): type`	`type foo(T1 x1, T1 x2,...,Tn xn)`

- Οι εντολές προγράμματος αντιστοιχούνται με προφανή τρόπο.
- Οι κλήσεις βιβλιοθήκης θα μπορούσαν να υλοποιηθούν ως εξής:

Κλήση TeaC	Συνάρτηση υλοποίησης σε C99
`readString(): string`	`fgets()`
`readInt(): int`	`atoi(readString())`
`readReal(): real`	`atof(readString())`
`writeString(s[]: string)`	`printf("%s", s)`
`writeInt(i: int)`	`printf("%d", i)`
`writeReal(r: real)`	`printf("%g", r)`

Οι προκαθορισμένες συναρτήσεις της **TeaC** αντιμετωπίζονται όπως όλες οι άλλες συναρτήσεις. Φροντίστε κατά τη μετατροπή του πηγαίου κώδικα της **TeaC** σε **C** να συμπεριλάβετε (`#include`) στον παραγόμενο **C** κώδικα το αρχείο `teaclib.h` που σας δίνεται και περιέχει την υλοποίηση των προκαθορισμένων συναρτήσεων της **TeaC** 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 στα πλαίσια 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 και Mint, αλλά 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 **TeaC** 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 ένα 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 courses. 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.tc`, για είσοδο `example.tc`.

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.tc
```

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 **TeaC** σε **C**, δηλαδή ένα πρόγραμμα το οποίο θα παίρνει ως είσοδο ένα αρχείο με κάποιο πρόγραμμα της γλώσσας **TeaC** και θα αναγνωρίζει αν αυτό το πρόγραμμα ακολουθεί τους συντακτικούς κανόνες της **TeaC**. Στην έξοδο θα παράγει το πρόγραμμα που αναγνώρισε, στη γλώσσα **C**, εφόσον το πρόγραμμα που δόθηκε είναι συντακτικά σωστό ή 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 στο 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`. Οταν 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 flex θα καταλαβαίνει τα ίδια tokens με τον bison.
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 -lfl
   ```

5. Καλέστε τον εκτελέσιμο `mycompiler` για είσοδο `test.tc` γράφοντας:

   ```
   ./mycompiler < test.tc
   ```

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 shift/reduce και reduce/reduce.

Κάθε φορά που αλλάζετε το `mylexer.l` και `myanalyzer.y` θα πρέπει να κάνετε όλη την 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.
- `mycompiler`: Το εκτελέσιμο αρχείο του αναλυτή σας.
- `correct1.tc, correct2.tc`: Δύο σωστά προγράμματα/παραδείγματα της **TeaC**.
- `correct1.c, correct2.c`: Τα ισοδύναμα προγράμματα των δύο παραπάνω σε γλώσσα **C**.

- `wrong1.tc`,`wrong2.tc`: Δύο λανθασμένα προγράμματα/παραδείγματα της **TeaC**.

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 η εργασία σας δεν μπορεί να εξετασθεί περαιτέρω.
- *Επιτυχής δημιουργία του αναλυτή*. Ο βαθμός σας θα εξαρτηθεί από τον αριθμό των shift-reduce και reduce-reduce conflicts που εμφανίζονται κατά τη δημιουργία του αναλυτή σας.
- *Ελεγχος αναλυτή σε σωστά και λανθασμένα παραδείγματα προγραμμάτων **TeaC***. Θα ελεγχθούν σίγουρα αυτά του Παραρτήματος, αλλά και άλλα άγνωστα σ' εσάς παραδείγματα. Η καλή εκτέλεση τουλάχιστον των γνωστών παραδειγμάτων θεωρείται αυτονόητη.
- *Ελεγχος αναλυτή στα δικά σας παραδείγματα προγραμμάτων **TeaC***. 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 δώσει και να απαντήσετε στις σχετικές ερωτήσεις. Επίσης θα πρέπει να μπορείτε να κάνετε compile τον κώδικά σας.

4 Επίλογος

Κλείνοντας θα θέλαμε να τονίσουμε ότι είναι σημαντικό να ακολουθείτε πιστά τις οδηγίες και να παραδίδετε αποτελέσματα σύμφωνα με τις προδιαγραφές που έχουν τεθεί. Αυτό είναι κάτι που πρέπει να τηρείται 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 στον διδάσκοντα και στους υπεύθυνους εργαστηρίου του μαθήματος. Γενικές απορίες καλό είναι να συζητώνται στο χώρο συζητήσεων του μαθήματος για να τις βλέπουν και οι συνάδελφοί σας.

Καλή επιτυχία!

ΠΑΡΑΡΤΗΜΑ

5 Παραδείγματα προγραμμάτων της TeaC

5.1 Hello World!

```
(* My first TeaC program. File: myprog.tc*)

const message <- "Hello world!\n":  string;

const start <- (): int => {
 writeString(message);
}
```

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_STRING:           string
Token SEMICOLON:                ;
Token KEYWORD_CONST:            const
Token KEYWORD_START:            start
Token ASSIGN_OP:                <-
Token LEFT_PARENTHESIS:         (
Token RIGHT_PARENTHESIS:        )
Token COLON:                    :
Token KEYWORD_INT:              int
Token ARROW_OP                  =>
Token LEFT_CURLY_BRACKET:       {
Token IDENTIFIER:               writeString
Token LEFT_PARENTHESIS:         (
Token IDENTIFIER:               message
Token RIGHT_PARENTHESIS:        )
Token SEMICOLON:                ;
Token RIGHT_CURLY_BRACKET:      }
Your program is syntactically correct!
```

5.2 Συναρτήσεις της TeaC

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

```
-- File: useless.tc
-- A piece of TeaC code for demonstration purposes

const N <- 100: int;

let a, b: int;

const cube <- (i: int): int => {
  return i*i*i;
}

const add <- (n: int, k: int): int => {
  let j: int;

  j <- (N-n) + cube(k);
  writeInt(j);
  return j;
}

(* Here you can see some useless lines.
 * Just for testing the multi-line comments ...
 *)
const start <- (): int => {
  a <- readInt();
  b <- readInt();
  add(a, b); -- Here you can see some dummy comments!

  return 0;
}
```

Το παραπάνω πρόγραμμα θα μπορούσε να ενδεικτικά να μεταφραστεί ως εξής:

```
#include <stdio.h>
/* TeaC library */
int readInt() { int ret; scanf("%d", &ret); return ret; }
void writeInt(int n) { printf("%d",n); }

const int N = 100;

int a,b;

int cube(int i) {
   return i*i*i;
 }
 int add(int n, int k) {
   int j;

    j = (N-n) + cube(k);
    writeInt(j);
     return j;
 }

 int main() {
   a = readInt();
   b = readInt();
   add(a, b);

   return 0;
 }
```

Μπορεί να μεταφραστεί από τον compiler με την εντολή

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

5.3 Πρώτοι αριθμοί

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

```
-- File: prime.tc
let limit, number, counter: int;


const prime <- (n: int): bool => {

  let i: int;
  let result, isPrime: bool;


  if n < 0 then
    result <- prime(-n);
  else if n < 2 then
    result <- false;
  else if n = 2 then
    result <- true;
  else if n % 2 = 0 then
    result <- false;
  else
    i <- 3;
    isPrime := true;
    while isPrime and (i < n / 2) loop
      isPrime <- n % i != 0;
      i <- i + 2;
    pool;
    result <- isPrime;
  fi;

  return result;
};

const start <- ():int => {

  limit <- readInt();
  counter <- 0;
  number <- 2;

  while number <= limit loop
    if prime(number) then
      counter <- counter + 1;
      writeInt(number);
      writeString(" ");
    fi;
    number <- number + 1;
  pool;

  writeString("\n");
  writeInt(counter);

  return 0;
}
```

5.4 Παράδειγμα με συντακτικό λάθος

```
(* My first TeaC program *)

1 const message <- "Hello world!\n":  string;
2
3 const start <- (): int => {
4       writeString(message
5 }
```

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_STRING:           string
Token SEMICOLON:                ;
Token KEYWORD_CONST:            const
Token KEYWORD_START:            start
Token ASSIGN_OP:                <-
Token LEFT_PARENTHESIS:         (
Token RIGHT_PARENTHESIS:        )
Token COLON:                    :
Token KEYWORD_INT:              int
Token ARROW_OP                  =>
Token LEFT_CURLY_BRACKET:       {
Token IDENTIFIER:               writeString
Token LEFT_PARENTHESIS:         (
Token IDENTIFIER:               message
Token RIGHT_CURLY_BRACKET:      }

Syntax error in line 4:       writeString(message
```

ή

```
Syntax error in line 4:       writeString(message
(Missing parenthesis)
```