Theory of Calculation Planning work Spring Semester 2017

TECHNICAL UNIVERSITY OF CRETE

School of Electrical and Computer Engineering

PLN 402

THE BUDGET REVIEW

Planning work

Verbal and Editorial Analysis

of the Programming Language FC

Instructor
Michael G. Lagoudakis

Help
George Anestis

Spring Semester 2017

Last updated: 2017-03-16

Theory of Calculation Planning work Spring Semester 2017

1 Introduction

The planning of the course **"PLI 402 - Theory of Computation"** aims at
a deeper understanding of the use and application of theoretical tools, such as regular expressions and
contextual grammars, the language compilation problem
programming. In particular, the work concerns the design and implementation of the initial stages
a compiler (compiler) for the imaginary **FC** programming language **(F** ictional **C),** which
is described in detail below.

In particular, a **source-to-source compiler** (trans-compiler or transpiler) will be created,
a type of compiler that takes the source code of a program into
a programming language and generates the equivalent source code in another language
programming. In our case the source code will be written in the fantastic
programming language **FC** and the generated code will be in programming language **C.**

To implement the work, you will use the **flex** and **bison** tools available
as free software and **C** language .

The thesis includes two parts:

- Implementation of a **verbal analyzer** for the **FC** language using **flex**
- Implementing a **syntax analyzer** for the **FC** language using a **bison**
  - Converting **FC** code to **C** code using **bison** actions

Remarks

- Work will be done **individually** . Blind plagiarism, even from older ones
  work, can be easily detected and leads to zeroing.
- Computers can be used to compute the work
  Center and personal computers. Flex and bison tools are available in any
  Linux distribution.

- The delivery of the work will be done **electronically** through the course website at courses . The archive archive (.zip, .tar, .rar) should contain all the necessary records according to the job specifications.
- Work must be delivered within the deadline. Delayed jobs are not accepted. Non-delivery of the job automatically leads to failure in the lesson.
- The evaluation of the work will include an **examination of the good functioning** of the deliverable program in accordance with the specifications as well as an **oral examination** . The examination will be done in the Polytechnic School, in days and hours to be announced.
- Reminder: the degree of work should be at least **50/100** .

Page 3

2 The programming language FC

The **FC** language **(F** ictional **C)** based on the language **C.** Due to the similarities of **FC** with **C** , the description below highlights where the two languages differ. In cases of possible ambiguity, you can refer to the description of C. The description of the language below probably contains

and elements which do not fit into verbal or editorial analysis. It is your responsibility to identify these items and ignore them when developing your analyzer.

Each program in **FC** language is a set of *verbal units,* which are arranged based on *editorial rules* , as described below.

2.1 Teaching units

The word units of the **FC** language are divided into the following categories:

• The **keywords** *(keywords),* which are the following:

static	**boolean**	**integer**	**char**	**real**
true	**false**	**string**	**void**	**while**
to	**break**	**continue**	**return**	**begin**
if	**else**	**for**	**end**	**or**
not	**and**	**mod**		

The keywords are case-sensitive, meaning you can not use uppercase for them.

• **IDs** *(identifiers)* are used for variable names, function and classes and consist of a lowercase or capital letter of the Latin alphabet, followed from a series of zero or more pedestals or capital letters, decimal digits system or underscore characters. Identifiers must not match with the keywords mentioned above.

Examples: x y1 angle my_value Distance_02

• The **integer (positive) constants** *(integer positive constants),* consisting of one or more digits of the decimal system without unnecessary zeros at the beginning.

Examples: 0 42 1284200 3 100001

• The **actual (positive) constants** *(real positive constants),* which comprise an integral part, a fractional part and an optional exponential part. The whole part consists of one or more digits (the decimal system) without unnecessary zeros at the beginning. The fractional part consists of the character of the decimal point (.) followed by one or more decimal digits. Finally, the exponential part consists of the lower case or the capital letter E, an optional + or - sign and one or more digits without unnecessary zeros at the beginning.

Examples: 42.0 4.2e1 0.420E + 2 1234.12345678e-123

• **Logical constants** *(boolean constants),* which are the **true** words-values and **false.**

• **Fixed characters** *(constant characters),* consisting of a single character in quotation marks. This character may be any common character or sequence escape sequence. Common characters are all printable characters except

Page 4

simple and double quotes and character backslash. The escape sequences start with the \ backslash character and are described in the table below.

Escape character	Description
\ **n**	line feed
\ **t**	tab character (tab)
\ **r**	return character at the beginning of the line
\\	character \ (backslash)
\ **'**	character **'** (simple introductory)
\ **"**	character **"** (double introductory)

Examples: **'** a **'** **'** M **'** **'** 1 **'** **'** ~ **'** **'** @ **'** **'** \ n **'** **'** \ **' '**

• **Constant** *strings* , consisting of a sequence of common ones characters or escape sequences within double quotes. Fixed strings are not can extend to more than one line of the input file.

Examples: " abc " " Route 66 " " Hello world! \ N "

" Item: \ t \ " Laser Printer \ " \ nPrice: \ t $ 142 \ n "

• The **operators** *(operators),* which are the following:

numerical operators:	+	-	*	/	**mod**
relational operators:	=	> =	<=	>	<	! =
reasonable operators:	**and or**		**not &&**	||	!
award operator:	: =				
Significant operators:	+	-			

Operator	**Description**
+ - * / mod	Add, Remove, Multiply, Divide, Balance
=> = <=	Equal (=), Higher or equal (≥), Smaller or equal (≤)
> <! =	Larger than (>), Less than (<), Miscellaneous from (≠)
and &&	Reasonable coupling
or ||	Logical disjunction
not!	Logical refusal

• **Separators** , which are the following:

begin end; (), []

In addition to the verbal units mentioned above, an **FC** program may also contain and items that are ignored (that is, they are recognized, but no analysis is made):

• **Blank characters** *(white space),* ie sequences consisting of blank spaces (space), tabs, line feeds, or return characters
at the beginning of the line (carriage return).

Page 5

• **Comments** *(comments),* which begin with the character sequence / * and terminate with the first occurrence of the character string * / . Comments may not be nested. Inside, any character is allowed to appear.

• **Line** *comments,* starting with the // string and extending until the end of the current line.

2.2 Constitutional rules

The editorial rules of the **FC** language define the correct syntax of its verbal units.

2.2.1 Programs

An **FC** program can be located within a **.fc** extension **file** and composed of the following building blocks (detailed description is given below):

• Statements of variables

• Function statements

These statements must appear exactly in the above order and the following apply as well:

1. An **FC** program may have **zero or more** variable statements.
2. A **FC** program must have **one or more** function statements. More in particular, there should at least be a definition of the function with **integer** header **main ()** from where the program starts.

2.2.2 Data types

FC language supports the following main types of data:

● **integer** : integers

- **boolean** : reasonable prices
- **character** : characters
- **real** : real numbers
- **string** : strings (up to 128 characters in size)

2.2.3 Variables

Variable statements are made by typing the type followed by one or more variable names (separated by separator,) and ending with the separator?

<type> <identifier1>, <identifier2>, ..., <identifierk>;

The type of each variable can be one of the key ones. Examples of statements are:

integer i; real a, b, c; string phrase, word;

The **FC** also supports data tables, that is declaration multidimensional tables of a basic type by displaying the size of the table in each dimension next to the corresponding identifier:

<type> <identifier> [n] [m] ... [k];

N, m, ..., k indicate the size of the table in each dimension and should be positive integer constants. Examples of table statements:

real spik [45]; integer sp [23] [8], spk [45]; character spak [2] [2];

Combinations of simple variables and tables of the same type in the same statement are allowed. Be allowed also the initialisation of simple variables (not tables) in the statement with fixed and optional

use of the effector operator. For example:

real x: = - 100.50, matrix [40] [100], y_24: = + 30e-4;

Table position pointers can be integer (positive) constants or positive value expressions. Below are examples of tabs identifying:

matrix [1] [5] y [k] [(k + 2) * n]

In addition to the statement of variables, the term **static is also** supported . For example:

static integer I: = 25, k [40] [5];

2.2.4 Functions

Each function is a structural unit consisting of the heading followed by the her body. The heading specifies the name of the function, its standard parameters within brackets and type of result (one of the main types, not tables). The brackets are mandatory even if a function has no standard parameters. Also, if the function does not return a value, then the formula is defined as **void** . Each typical parameter is characterized by its name and type (only basic types, not tables) are allowed.

The body of a function is bounded by the **begin** and **end** separators and consists of variable statements and a sequence of commands (see below) in this order of occurrence. The body of a function can not contain statements of other functions. Here is an example function with headline and body:

```
integer foo (integer k, real bound)
begin
        integer p, i: = 0, z: = 0;
        p: = 34 * k;
        for (i: = 1; i <= k; i ++)
                if (z <bound)
                        z: = p * i;
        return z;
end
```

The **FC** gives some predefined library functions for input and output data, the which are at the developer's disposal for use without a declaration. Below are given the their headings:

string readString ()	**void writeString (string s)**
integer readInteger ()	**void writeInteger (integer i)**
real read ()	**void writeReal (real r)**

2.2.5 Expressions

Expressions are probably the most important part of a programming language. The Basic expression forms are constants, variables of any type, and calls functions. Complex expression forms are obtained by using operators and parentheses.

Page 7

The operators of **FC** are distinguished in operators with an argument and operators with two arguments. From the First, some are written before the prefix and postfix, while the second are always written between the sentences (infix). The evaluation of the operators' arguments with two arguments are made from left to right. The table below sets the priority and the suitability of **FC** operators . The operators that appear higher in the top are preceded table. Those who are in the same cell have the same priority. Note that they can parentheses are used in an expression to indicate the desired priority.

Operators	Description	Arguments	Position Accessibility

not!	Logical refusal operator	1	prefix, right
+ -	Significant operator	1	prefix, right
*** / mod**	Operators with factors	2	infix, left
+ -	Operators on terms	2	infix, left
=! = <> <=> =	Relational operators	2	infix, left
and &&	Reasonable coupling	2	infix, left
or ||	Logical disjunction	2	infix, left

Here are examples of correct expressions:

```
-a                                  // opposite to variable a
a + b * (b / a)                     // arithmetic expression
4 + 50.0 * x / 2.45                 // arithmetic expression
(a + 1) mod cube (b + 3)            // Balance operator, function call
(a & lt; b) and (c = d)             // logical relational operators
a + (c1 = d)                        // numerical relational operators
a + b [1] [k] [(k + 1) * 2]         // arithmetic expression with a table
```

2.2.6 Commands

The commands supported by the **FC** language are the following (all commands except complex, are considered simple and also every simple command of the **FC** language terminates with the special separator symbol, with the exception of if, for, while, where this requirement is transferred to internal instructions, as detailed below):

1. The *empty command* (?) That does nothing, as its name implies.
2. The *composite command* consists of a (possibly vacant) sequence of simple commands separated with each other **?** and is delimited by the **begin** and **end** separators.
3. The *assignment order* **v: = e;** , where v is a variable and is an expression.
4. The **if (e) s** *command* $_1$ **else s** $_2$. The else section is optional. E is one expression, while s $_1$ and s $_2$ are simple or complex commands.

Page 8

5. The *repeat command* **for (s** $_1$ **; e; s** $_2$ **) s** where s $_1$, s $_2$ are simple commands (exceptionally, without a separator? at the end) performed before the start and each iteration respectively, e is an optional expression that is checked / calculated before each iteration and s is a simple or complex command that runs on each iteration.
6. *Loop commands* **while (e) s** and **do s while (e);** . E is an expression and s a simple or complex command.
7. The **break** *stop command* **?** which causes immediate exit from the innermost loop.
8. **Does the** *continuation command* **continue?** causing the cessation of the current repeat and the start the next iteration of the loop in which it is located.
9. *Return order* **return e?** which terminates (possibly, prematurely) the execution of a function and returns a value given by the expression e.
10. The *command to call* a function or method **f (e** $_1$ **, ..., e** $_n$ **);** , where f is its name function / method and e $_1$, ..., e $_n$ are expressions corresponding to the declared ones arguments.

3 Matching language FC to C language

The ultimate goal of this work is to produce the source-to-source compiler produced as input the source code to **FC** and produce the source code equivalent programs in **C** language . In this context, the rules of our editorial grammar

analyst (in fact, the corresponding actions of rules that will be triggered each time) will they must also take care to correctly match the **FC** language code to an appropriate - equivalent language code **C** taking into account the following.

3.1 Aligning types and constants

The **FC** types are mapped to the **C** types based on the following table:

Type of FC	Corresponding type of C
integer	**int**
boolean	**int**
char	**char**
real	**double**
T [n1] ... [nk]	**T [n1] ... [nk]**

Based on the above table, the **FC** constants are mapped to **C** constants. For example, the boolean constants of **FC** , true and false, are matched to integer values 0 and 1 respectively.

3.2 Anti- alignment of structural units

Each **FC** module includes declarations of variables (optional) and functions. OR the main building block of a program in **FC** corresponds to a **.c** file that it includes some initial statements necessary in **C** , statements of (global) variables (optional) and functions, including the **int main () function** of **C** as its corresponding function of the original **integer main () function** of **FC** .

Theory of Calculation Planning work Spring Semester 2017

Assigning unit testing between the two languages will generally be as follows:

- a **FC** foo type T of **FC** corresponds to a variable with the same name and with it matched type of **C**
- a func function of **FC** corresponds to a function of the same name and matching parameter types and return value of **C**
- Simple and complex program commands are mapped in the obvious way
- Library function calls can be implemented as follows:

Calling FC	**Implementation function in C**
string readString ()	**gets ()**
integer readInteger ()	**atoi (gets ()**
real read ()	**atof (gets ()**
void writeString (string s)	**puts (s)**
void writeInteger (integer i) printf ("% d", j)	
void writeReal (real r)	**printf ("% g", r)**

4 Detailed job description

4.1 Tools

To successfully complete the job you need to know good programming in **C** , **flex** and **bison** . The flex and bison tools have been developed within the GNU program and you can find them on all nodes of the web that have GNU software (e.g. **www.gnu.org**). More information, manuals and links for these two tools will find on the course website.

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 στις διανομές Linux Ubuntu και Mint, αλλά είναι πιθανόν να υπάρχουν κάποιες μικροδιαφορές σε άλλες διανομές.

4.2 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 **FC** 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 ASSIGNMENT:       :=
token IDENTIFIER:       k
token OP_PLUS:          +
token CONST_INTEGER:    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 ASSIGNMENT: :=
token IDENTIFIER: k

Unrecognized token ^ in line 56: i := k ^ 2;

όπου 56 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 στην 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 στην 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.fc, για είσοδο example.fc.

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

flex mylexer.l

gcc -o mylexer lex.yy.c -lfl

./mylexer < example.fc

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

- **2η φάση: Συντακτική ανάλυση**

Το τελικό προϊόν αυτής της φάσης θα είναι ένας συντακτικός αναλυτής και μεταφραστής της **FC** σε **C** , δηλαδή ένα πρόγραμμα το οποίο θα παίρνει ως είσοδο ένα αρχείο με κάποιο πρόγραμμα της γλώσσας **FC** και θα αναγνωρίζει αν αυτό το πρόγραμμα ακολουθεί τους συντακτικούς κανόνες της **FC** . Στην έξοδο θα παράγει το ισοδύναμο του προγράμματος που αναγνώρισε πλέον σε γλώσσα **C** , εφόσον το πρόγραμμα που δόθηκε είναι συντακτικά σωστό ή διαφορετικά θα εμφανίζεται ο αριθμός γραμμής όπου διαγνώσθηκε το

Page 11

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 56 : i := k + 2 * ;
Syntax error in line 56: i := k + 2 * ;(expression expected)

όπου 56 είναι ο αριθμός της γραμμής μέσα στο αρχείο εισόδου όπου βρίσκεται η συγκεκριμένη εντολή συμπεριλαμβανομένων των γραμμών σχολίων.

Για να φτιάξετε έναν συντακτικό αναλυτή 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:

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

b. 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.

c. 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. When κάνετε compile to 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.fc γράφοντας:

./mycompiler < test.fc

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 τη διαδικασία. Είναι καλή ιδέα να φτιάξετε ένα script για όλα τα παραπάνω.

Page 12

4.3 Παραδοτέα

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

- mylexer.l: Το αρχείο flex.
- myanalyzer.y: Το αρχείο bison.
- mycompiler: Το εκτελέσιμο αρχείο του αναλυτή σας.
- report.pdf: Συνοπτική (δακτυλογραφημένη, PDF μορφή) αναφορά που θα περιγράφει τις σχεδιαστικές σας επιλογές για τις κανονικές εκφράσεις και την γραμματική που επινοήσατε και θα εξηγεί τις τεχνικές που χρησιμοποιήσατε και τις ιδιαιτερότητες της εργασίας σας. Επίσης, θα πρέπει να αιτιολογεί τις όποιες συμβάσεις ή παραδοχές κάνατε και να αναφέρει τα στοιχεία της γλώσσας που θεωρήσατε ότι δεν εμπίπτουν στα πλαίσια της λεκτικής ή συντακτικής ανάλυσης.
- correct1.fc,correct2.fc: Δύο σωστά προγράμματα/παραδείγματα της **FC** .
- correct1.c,correct2.c: Τα ισοδύναμα προγράμματα των δύο παραπάνω σε γλώσσα **C** .
- wrong1.fc,wrong2.fc: Δύο λανθασμένα προγράμματα/παραδείγματα της **FC** .

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

4.4 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 σωστού αναλυτή* . Ο βαθμός σας θα εξαρτηθεί 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* **FC** . Θα ελεγχθούν σίγουρα αυτά του Παραρτήματος και άλλα άγνωστα σ' εσάς παραδείγματα. Η καλή εκτέλεση τουλάχιστον

των γνωστών παραδειγμάτων θεωρείται αυτονόητη.

- *Έλεγχος αναλυτή στα δικά σας παραδείγματα προγραμμάτων* **FC** . Τέτοιοι έλεγχοι θα βοηθήσουν σε περίπτωση που θέλετε να αναδείξετε κάτι από τη δουλειά σας.
- *Ερωτήσεις σχετικά με την υλοποίηση και τα παραδοτέα κείμενα.* Θα πρέπει να είστε σε θέση να εξηγήσετε θέματα σχεδιασμού, επιλογών και τρόπων υλοποίησης.

5 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 τηρείτε ως μηχανικοί για να μπορέσετε στο μέλλον να εργάζεσθε συλλογικά σε μεγάλες ομάδες εργασίας, όπου η συνέπεια είναι το κλειδί για την συνοχή και την επιτυχία του κάθε έργου.

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

Καλή επιτυχία!

ΠΑΡΑΡΤΗΜΑ

6 Παραδείγματα προγραμμάτων της FC

6.1 Hello World!

```
/* My first FC program - myprog.fc*/

integer main()
begin
        writeString("Hello World!\n");
        return 0;
end
```

Ζητούμενο αποτέλεσμα λεκτικής – συντακτικής ανάλυσης:

Token KEYWORD_INTEGER: integer

Token IDENTIFIER: main

Token PARENTHESIS_LEFT: (

Token PARENTHESIS_RIGHT:)

Token KEYWORD_BEGIN: begin

Token IDENTIFIER: writeString

Token PARENTHESIS_LEFT: (

Token CONST_STRING: "Hello World!\n"

Token PARENTHESIS_RIGHT:)

Token SEMICOLON: ;

Token KEYWORD_RETURN: return

Token CONST_INTEGER: 0

Token SEMICOLON: ;

Token KEYWORD_END: end

Your program is syntactically correct!

Page 14

Θεωρία Υπολογισμού Εργασία Προγραμματισμού Εαρινό Εξάμηνο 2017

Το παραπάνω πρόγραμμα **myprog.fc** θα μπορούσε **ενδεικτικά** να μεταφραστεί σε **C** ως εξής:

```
#include <stdio.h>
/* FC library */
void writeString(char *str) { printf("%s",str); }

int main()
{{
        writeString ("Hello World!\n");
        return 0;
}
```

Το παραπάνω πρόγραμμα **myprog.c** μπορεί να μεταγλωττισθεί από τον compiler της **C** με την εντολή

gcc -Wall myprog.c

και στη συνέχεια να εκτελεσθεί.

Page 15

Θεωρία Υπολογισμού Εργασία Προγραμματισμού Εαρινό Εξάμηνο 2017

6.2 Δομικές μονάδες FC

Παράδειγμα για την κατανόηση της σύνταξης δομικών μονάδων στη γλώσσα **FC** .

```
// A useless piece of FC code - useless.fc

integer i, k;

integer cube(integer i)
begin
        return i * i * i;
end

void add(integer n, integer k)
begin
        integer j;
        j := n + cube(k);
        writeInteger(j);
```

```
end

/* Here you can see some useless lines.
** Just for testing the multi-line comments ...
*/
integer main()
begin
        k := readInteger();
        i := readInteger();
        add(k,i); //Here you can see some dummy comments!
        return 0;
end
```

Το παραπάνω πρόγραμμα **useless.fc** θα μπορούσε **ενδεικτικά** να μεταφραστεί σε **C** ως εξής:

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

int i,k;
int cube(int i) {
        int result;
        result = i*i*i;
        return result;
}

void add(int n, int k)
{{
        int j;

        j = (int)(100-n) + cube(k);
        writeInteger(j);
}

int main()
{{
        k = readInteger();
        i = readInteger();
        add(k,i);
```

```
        return 0;
}
```

Το παραπάνω πρόγραμμα **useless.c** μπορεί να μεταγλωττισθεί από τον compiler της **C** με την εντολή

gcc -Wall useless.c

Page 17

6.3 Πρώτοι αριθμοί

Το παρακάτω πρόγραμμα **FC** υπολογίζει τους πρώτους αριθμούς μεταξύ **1** και **n** , για το **n** που δίνεται.

```
boolean prime(integer n)
begin
        integer i; boolean isPrime, result;
        if (n < 0)
                result := prime(-n);
        else if (n < 2)
                result := false;
        else if (n = 2)
```

```
                result := true;
        eLse if (n mod 2 = 0)
                result := false;
        else
                begin
                        i := 3;
                        isPrime := true;
                        while ( isPrime && i <= n / 2 )
                        begin
                                isPrime := (n mod i != 0);
                                i := i+2;
                        end
                        result := isPrime;
                end
        return result;
end
integer main( )
begin
        integer limit, number, counter:=0;
        limit := readInteger();
        for (number:=1; number<=3; number:=number + 1)
                if (limit >= number)
                        begin
                                counter := counter + 1;
                                writeInteger(number);
                        end
        number := 6;
        while (number <= limit)
                begin
                        if (prime(number-1))
                                begin
                                        counter := counter + 1;
                                        writeInteger(number-1);
                                end
                        if ((number != limit) && prime(number+1))
                                begin
                                        counter := counter + 1;
                                        writeInteger(number+1);
                                end
                        number := number + 6;
                end
        writeInteger(counter);
        return 0;
```

```
end
```


6.4 Παράδειγμα με συντακτικό λάθος

```
/* My wrong FC program */

integer main()
begin
        writeString("Hello World!\n";
        return 0;
end
```

Ζητούμενο αποτέλεσμα λεκτικής – συντακτικής ανάλυσης:

Token KEYWORD_INTEGER: integer

Token IDENTIFIER: main

Token PARENTHESIS_LEFT: (

Token PARENTHESIS_RIGHT:)

Token KEYWORD_BEGIN: begin

Token IDENTIFIER: writeString

Token PARENTHESIS_LEFT: (

Token CONST_STRING: "Hello World!\n"

Token SEMICOLON: ;

Syntax error in line 5: writeString("Hello World!\n";

or

Syntax error in line 5: writeString("Hello World!\n";
(Missing parenthesis)