POLYTECHNIC OF CRETE

School of Electrical and Computer Engineering

PLE 402

COMPUTATION

THEORY

Programming Task Verbal and Syntactic Analysis of the Kappa Programming Language

Teacher

Michael C. Lagoudakis

Laboratory

George Anestis

Nektarios Mumoutzis

Spring Semester 2023

Last update: 6/4/2023

1 Introduction

The programming assignment of the course **"PLE 402 - Computation Theory"** aims at a deeper understanding of the use and application of theoretical tools, such as regular expressions and context-free grammars, to the problem of compilation of programming languages. Specifically, the work concerns the design and implementation of the initial stages of a compiler for the imaginary programming language **Kappa**, which is described in detail below.

More specifically, a **source-to-source compiler** (trans-compiler or transpiler) will be created, i.e. a type of compiler that takes as input the source code of a program in one programming language and produces the equivalent source code in another programming language. In our case, the input source code will be written in the fictional programming language **Kappa** and the generated code will be in the well-known programming language **C.**

To implement the project you will use the **flex** and **bison** tools, which are available as free software, and the **C** programming language.

The work includes two sections:

- Implementation of **a word parser** for the **Kappa** language using **flex**
- Implementation of **a syntactic analyzer** for the **Kappa** language using **bison**
  - Converting **Kappa** code to **C** code using **bison** scripts

Comments

The work will be carried out **individually**. Blind copying (plagiarism), even from previous work, can be easily detected and will result in a mark of zero.

Computers of the Computer Centre through the VDI service or personal computers can be used for the preparation of the project. The flex and bison tools are available on any Linux distribution. You can also very easily install an Ubuntu Linux terminal on Windows via the Microsoft Store (look for the Ubuntu application) and then install the flex and bison tools.

The assignment will be handed **in electronically** through the course website in eClass. The delivered archive (`.zip` or `.rar` or `.tar`) file should contain all necessary files according to the specifications of the assignment.

The work must be delivered **within** the deadline. Late work will not be accepted. Failure to hand in the assignment will automatically result in failure of the course.

The evaluation of the work will include **an examination of** the **proper functioning of** the delivered program, according to the specifications, as well as an **oral examination** in which you will have to explain each part of the code you have delivered and answer the relevant questions. The examination will take place on dates and at times to be announced.

Reminder: the grade of the assignment must be at least **40/100** to be considered adequate. Therefore, it is not sufficient to hand in only a part of it.

2 The Kappa programming language

The description of the **Kappa** language below follows the general format of a programming language description. It probably also contains elements which are not part of the verbal or syntactic analysis. It is your responsibility to recognize these elements and ignore them when developing your parser. Each **Kappa** language program is a set of *verbal units* arranged according to *syntactic rules*, as described below.

2.1 Verbal units

The lexical units are the vocabulary of the **Kappa** programming language and are divided into the following categories:

- *Identifiers* are used for variable and function names and consist of an uppercase or lowercase letter of the Roman alphabet followed by a series of zero or more uppercase or lowercase letters of the Roman alphabet, decimal digits or underscore characters. Identifiers must not coincide with keywords

  Examples of identifiers: x y1 `angle` `myValue` `Distance_02`

- The *keywords*, which are reserved words and cannot be used as identifiers or be used to infer, which are the following:

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

  Keywords are case-sensitive, i.e. they cannot be written in uppercase or a combination of uppercase and lowercase letters, except as given above. Otherwise, they are considered identifiers.

- *Integer* **constants**, which consist of one or more digits of the decimal system with no odd zeros at the beginning.

  Examples of integer constants: `0` `42` `1233503` `10001`

- *The floating-point const**ants***, which consist of an integer part, a fractional part and an optional exponential part. The integer part consists of one or more digits (of the decimal system) with no odd zeros at the beginning. The fractional part consists of the sub-division character (**`.`**) followed by one or more digits of the decimal system. Finally, the exponential part shall consist of the lower case or upper case letter e or E, an optional sign**+** or **−** and one or more digits of the decimal system without leading zeros.

  Examples of real constants: `42.4` `4.2e1` `0.420E+2` `42000.0e-3`

- The ***logical constants*** *(boolean constants)*, which are the words-values **`True`** and **`False`**.
- ***Constant*** *strings*, consisting of a sequence of common or escape characters within double quotation marks. Common characters are the period, comma, space, a-z, A-Z, 0-9 and the symbols -, +, *, /, :, _, $, %, !, #, @, &, ~,, (,). Escape characters start with \ (backslash) and are only those described in the following table.

Escape Character	Description
`\n`	line feed character (line feed)
`\t`	tab character (tab)
`\r`	return character at the beginning of the line

\\	character \ (backslash)
\"	character " (double quote)

A fixed string cannot span more than one line of the input file. It must be fully contained in a single line.

The following are examples of valid strings:

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

- The *operators*, which are the following:

  numerical operators: `+` `-` `*` `/` `%` `**`

  relational operators: `==` `!=` `<` `<=` `>` `>=`

  logical operators: sign `and` `or` `not`

  operators: `+` `-`

  assignment operator: `=` `+=` `-=` `*=` `/=` `%=`

- The *delimiters*, which are the following:

  `;` `(` `)` `,` `[` `]` `:` `.`

In addition to the verbal units mentioned above, a **Kappa** program may also contain elements that are ignored (i.e., recognized but not analyzed):

*White space **characters***, i.e. sequences consisting of *white* space, tab characters, line feed characters or carriage return characters.

Line comments, which start with the characters `--` and extend to the end of the current line.

2.2 Syntactical rules

The syntactic rules of the **Kappa** language define the correct syntax of its word units.

i) Programmes

A **Kappa** program can be found inside a file with the extension `.ka` and consists of the following components listed in that order and **terminated** with the character `;`.

Declarations of complex types	(optional)
Statements of constants	(optional)
Variable declarations	(optional)
Function definitions	(optional)
Main structural unit	(mandatory)

The main building block is the `main` function, which takes no arguments and is assumed to return no value. This function is the starting point for program execution and is of the form:

```
def main():
        body of the main
enddef;
```

A simple example of a valid `.ka` file is the following:

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

ii) Data types

The Kappa language supports the following data types:

integer	integers
scalar	real numbers
str	string of characters
boolean	reasonable values
comp	complex formula, which is formed by using other formulas, as well functions (detailed description in section vi) below)
[arrayLength]:type	arrayLength array size type with elements of type type, where type is one of the above data types and arrayLength should be an integer constant with a positive value. Example: sequenceOf10Ints[10]: int;
[]:type	array type, as above, where arrayLength may be omitted, for example when declaring the type of a function parameter.

iii) Variables

Variable declarations consist of one or more variable identifiers, separated by a comma, with the separator : at the end, followed by a data type. The following is an example of variable declarations, where a complex type is also used, which is defined before the variable declaration:

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

i, j: scalar;
s1, s2: str;
n1: integer;
x, y: scalar;
s, ss: str;
test: boolean;
grades[5]: scalar;
directions[4]: str;
w, z: coordinates;
```

iv) Fixed

Constants are declared using the const keyword and must necessarily be given an initial value using the assignment operator= . Example:

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

v) Settings

A function is a building , which consists of the following componentslisted in this order:

```
def function name ( parameter declarations ) -> return type : local
      variable and constant declarations                    (optional)
      commands                                              (required)
      return expression                                     (optional)
enddef
```

The declaration of a function starts with the keyword **def**, followed by the name of the function, followed by its parameters in parentheses. Optionally, if the function returns a value, it is followed by the characters **->** and the type of the returned result, **which cannot be a table.** Then there is the character **:** and the body of the function, terminated by the keyword **enddef.**

The body of a function contains a sequence of local variable declarations, constant declarations, and statements. The body may contain a **return** statement if the function returns a value. Parentheses in parameters are mandatory, even if a function has no parameters. The return type may be omitted if the function does not need to return a specific value. A function may contain a **return** statement without returning a value when its return type is not declared. The following are examples of valid function definitions:

```
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;
```

The **Kappa** language supports a set of predefined functions, which are available to the programmer for use anywhere in the program. Below, their headings are given:

```
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) Declarations of complex types (comp)

The **Kappa** language provides the ability to declare complex types using the reserved word **comp**. The general form of declaring a composite type is as follows:

```
comp composite type name:
  variable declarations        (members - variable names must begin with the # character)
  functions                    (methods)
endcomp;
```

Example of a complex type declaration:

```
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; #x=#x+dx;
      #y=#y+dx; enddef;
endcomp;
```

Functions of a complex type are called *methods*, are considered *part of* the type, and have access to the member variables of the type.

Once a complex type is defined, it can then be used like all types. For example:

```
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) Expressions

Expressions are probably the most important part of a programming language. The basic forms of expressions are constants, variables of any type, and function calls. Advanced forms of expressions are obtained by using operators and parentheses.

Kappa operators are divided into single-argument operators and two-argument operators. Of the former, definite ones are written before the argument (prefix) and definite ones after (postfix), while the latter are always written between the definitions (infix).

The evaluation of the definitions of operators with two arguments is done from left to right. The following table defines the precedence and attractiveness of **Kappa**'s operators. The operators are listed in **descending** order of precedence. All operators in the same row have the same precedence. Note that parentheses can be used in an expression to indicate the desired precedence.

Operators	Description	Convenience

`.`
`()`
`[]`
	access to complex type membership
	function or method call
	access to a table element
		from left to right

`**`	elevation in power	from right to left
`+ -`	sign operators (unitary)	from right to left
`* / %`	multiplication, division, remainder of division	from left to right
`+ -`	addition, subtraction	from left to right
`< <=` `> >=`	relative inequality operators	from left to right
`== !=`	equality, inequality	from left to right
`not`	logical negation operator (unitary)	from right to left
`and`	logical coupling	from left to right
`or`	logical decoupling	from left to right

`=`
`+= -= *=`
`/= %= :=`
	assignment operators	from right to left

Here are examples of correct expressions:

```
-a                      -- opposite of variable a a +
b * (b / a)             -- numerical expression
4+ 50.0*x / 2.45        -- numerical expression
(a+1) % cube(b+3)       -- numeric expression with function call
```

```
(a <= b) and (d <= c) -- operators logical with relational
(c+a) != (2*d)          -- arithmetic operators with
relational a + b[(k+1)*2]   -- arithmetic expression with
table
```

viii) Commandments

The statements supported by the **Kappa** language are the following:

- The *assignment* command **v= expr;** , where **v** is a variable,= is the assignment operator, and **expr** is an expression.
- The *control command* **if (expr): stmts$_1$else: stmts$_2$endif;** The **else** part is optional. The **expr** is an expression, while **stmts$_1$**and **stmts$_2$**each correspond zero or more commands terminated by the character **;.**
- The *iteration command* **for integer_variable in [start:stop:step]: stmts endfor;** where **start, stop, step**, are numeric expressions corresponding to the initial value, final value and step of the integer variable **integer_variable**. The **step** is optional. When absent, the step of the integer variable is **+1. stmts** corresponds to zero or more commands terminated by the character **;.** and executed in each iteration.
- The *simple compact array over integer values command* **new_array := [expr for elm:size] : new_type;** The **expr** is an expression containing **elm**, **elm** is an **integer** type variable that will run through the integer values from **0** to **size-1**, and **size** is a positive integer. The command creates a new **array new_array** with elements of type **new_type** by applying the expression **expr** to each value of **elm** and storing the result in the appropriate location in the **new_array**.
- The *compact array command using another array* **new_array := [expr for elm: type in array of size] : new_type;** The **expr** is an expression containing **elm**, **elm** is a variable of type**, array** is an array variable, and **size** is the size of the array. The command creates a new **array new_array** with elements of type **new_type** by applying the **expr** expression to each **elm** element of the **array**, storing the result in the appropriate location in the **new_array**.
- The *loop command* **while (expr): stmts endwhile; expr** is an expression and **stmts** corresponds to zero or more instructions terminated with the character **;.** and are executed in each iteration of the loop.
- *The command* **;** which causes the immediate exit from the innermost loop.
- The *continue* **;** *command* that causes the current iteration to stop and next iteration of the loop it is in to start.
- The **;** or **return expr;** that terminates (possibly, prematurely) the execution of the function it is in and returns, where **expr** is an (optional) expression.
- The *instruction to call* a function **f(expr$_{(1)}$,...,expr$_n$) ;** where **f** is the name of the function and **expr$_{(1)}$,...,expr$_n$**are expressions corresponding to the declared function arguments.
- The blank command **;**

2.3 Mapping from Kappa to C99

C99 is the revision of the **C** language standard made 1999. In this revision several useful extensions were added to the somewhat old **C89**. See the corresponding Wikipedia article for more

Details. As **C99** is a rich language, it is particularly easy to map **Kappa** programs to **C99** programs. The details of this mapping will be described below.

2.3.1 Mapping of formulas and constants

The types of **Kappa** are mapped to the types of **C99** based on the table below:

Type of Kappa	Corresponding type of C99
`scalar`	`double`
`integer`	`int`
`str`	`char*`
`boolean`	`int`
`array[n]:T`	`TC array[n]`
`[]:T`	`TC*`
`def func(a1:T1, ... ak:Tk) -> type`	`type (*) func(TC1 a1, ... TCk ak)`

where `T, T1, ..., Tk` is a type of **Kappa** and `TC, TC1, ..., TCk` is the corresponding type of **C**.

On the basis of the above table, **Kappa** constants are also assigned to **C99** constants. For example, the boolean constants of **Kappa**, `True` and `False`, are mapped to the integer values `1` and `0` respectively.

Especially for declaring complex types using the reserved word `comp`, the mapping to **C** code will be as follows. First, we reformulate the generic form of the composite type declaration in **Kappa**:

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

in the corresponding **C** format:

```
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) {
     fbody function code mapping
}

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

Additionally, when a variable of a complex type is declared:

```
var_1: type_name;
```

the mapping to **C** code will be as follows:

```
type_name myvar1= ctor_type_name;
```

2.3.2 Mapping of compact table commands

The **Kappa** language supports two forms of compact table commands. One form runs a set of integer numbers from zero to a maximum value and uses these values to create a new table. The other form traverses the elements of an existing table and creates a new one. We describe these two formats below.

Compact table command over integer values

The *compact array over integer values command* **new_array := [expr for elm:size] : new_type;** creates a new array **new_array** with elements of type **new_type** by applying expression **expr** over every positive integer in the interval from **0** to element **size-1**.

So, the command:

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

is mapped to the following code in C:

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

For example the **Kappa** code:

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

is converted to C in the following code:

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

Compact table command using another table

For the *compact array command using another array of the form* **new_array := [expr for elm: type in array of size] : new_type;** the mapping to **C** code will be as described below. The **expr** is an expression which contains **elm**. **elm** is a variable of type type**,array** is an array variable and **size** is the size of the **array**. The command creates a **new array new_array** with elements of type **new_type** by applying the **expr** expression over each **elm** element of the **array** table (i.e., the elements from position 0 to position **size-1**).

So, the command:

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

is mapped to the following code in 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; -- where in the expression appears elm
                                -- is replaced by array[array_i]
```

For example the **Kappa** code: **const**

```
N= 100: integer; a[100]:
integer;
def main():
      for i in [0:N-1]:
            a[i]= i;
      endfor;
      half := [ x / 2 for x: integer in a of 100] :
scalar; enddef;
```

is converted to C in the following code:

```
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 Matching of building blocks

A **Kappa** program optionally includes declarations of constants, variables, complex types, functions and obligatory main code, i.e. the special function **main** and corresponds to **.c** file that includes, statements of constants, global variables, structures, functions and the original **main()** routine.

The mapping is as follows:

A **Kappa** variable **foo** with type **T**

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

corresponds to a variable with the same name and with the assigned type **TC**

```
TC foo, bar;
```

A **Kappa** function corresponds to a **C99** function with the same name and assigned parameter types.

Function of Kappa	Function of C99
`def foo(x1:T1, x2:T2,..., xn:Tn) -> type`	`type foo(TC1 x1, TC2 x2,..., TCn xn)`

Program commands are mapped in an obvious way. Library calls

could be implemented as follows:

Call Kappa	Implementation function in C99
`readStr() -> str`	Use the implementation given in the **`thlib.h`** file
`readInteger() -> integer`	
`readScalar() -> scalar`	
`writeStr(s:string)`	
`writeInteger(n:integer)`	
`writeScalar(n:scalar)`	
`write(fmt: str, ...)`	

Kappa's predefined functions are treated like all other functions. When converting **Kappa** source code to **C**, be sure to include (**`#include`**) in the generated **C** code the **`thlib.h`** file given to you that contains the **C** implementation of **Kappa**'s predefined functions.

3 Detailed job description

3.1 The tools

To successfully complete the assignment, you need to be proficient in **C**, **flex** and **bison** programming. **The flex** and **bison** tools have been developed under the GNU project and can be found on all Internet hubs that have GNU software (e.g. www.gnu.org). More information, manuals and links to these two tools can be found on the course website.

In the Linux operating system (any distribution) these tools are usually built-in. If not, the corresponding packages can be installed very easily. The usage instructions given below for the two tools have been tested on the Linux Ubuntu distribution. There may be minor differences in other distributions.

3.2 Approach to work

For your convenience in understanding the tools you will use, and the these tools work together, it is suggested that you implement the project in two phases.

Phase 1: Verbal Analysis

The final product of this phase will be a Lexical Analyzer, i.e. a program which will take as input a file with a **Kappa** language program and will recognize the tokens in this file. Its output will be a list of the tokens it read and their characterization. For example, for input:

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

the output of your project should be

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

```
token SEMICOLON: ;
```

In the case of an unrecognisable verbal unit, an appropriate error message should be printed on the screen and the verbal analysis should be terminated. For example, for the incorrect input:

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

the output of your project should be

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

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

where 46 is the number of the line within the input file where the specific command is found, including comment lines.

To build a Lexical Analyzer you will use the flex tool and the gcc compiler. Give `man flex` at the command line to see the flex manual or refer to the PDF file found in eClass. Files with flex code have a `.l` extension. To compile and run your code follow the instructions
given below.

1. Write the flex code in a file with a `.l` extension, e.g. `mylexer.l`.
2. Compile, by typing `flex mylexer.l` at the command line.
3. Give `ls` to see the file `lex.yy.c` generated by flex.
4. Create the executable with `gcc -o mylexer lex.yy.c -lfl`
5. If you have no errors in `mylexer.l`, the `mylexer` executable is generated.
6. Run with `./mylexer< example.ka`, for `example.ka` login.

Every time you change `mylexer.l` you have to do the whole process:

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

Therefore, it is a good idea to make a script or makefile to do all of the above automatically.

Phase 2: Editorial Analysis and Translation

The final product of this phase will be a **Kappa** Syntactic Analyzer and Translator in **C**, i.e. a program which will take as input a file containing a **Kappa** language program and will recognize whether this program follows the **Kappa** syntax rules. In the output it will produce the program it recognized, in **C**, if the program given is syntactically correct, otherwise it will display the line number where the first error was recognized, the content of the line with the error and *optionally* an informative message. For example, for the incorrect input

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

your program should terminate with one of the following error messages

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

where 46 is the number of the line within the input file where the specific command is found, including comment lines.

To build a syntax analyzer and compiler you will use the bison tool and the gcc compiler. Give `man bison` to see the bison manual. Files with bison code have a `.y` extension. To compile and run your code follow the instructions given below.

1. We assume that you already have the word parser in `mylexer.l`.
2. Write the bison code in a file with a `.y` extension, e.g. `myanalyzer.y`.
3. To join the flex with the bison you must do the following:

   Put the files `mylexer.l` and `myanalyzer.y` in the same directory.

   Remove the `main` function from the flex file and create a `main` in the bison file. To start, all the new `main` needs to do is call the bison macro `yyparse()` once. `yyparse()` repeatedly runs `yylex()` and tries to map each token returned by the Lexical Analyzer to the grammar you have written in the Syntactic Analyzer. It returns 0 for a successful termination and 1 for a termination with a syntactic error.

   Remove the defines you made for the tokens in the flex or other `.h` file. These will now be declared in the bison file one on each line with the `%token` command. When you compile `myanalyzer.y`a file called `myanalyzer.tab.h` is automatically created. You should include this file in the `mylexer.l` file so that the lexical analyzer will understand the same tokens as the syntactic analyzer.
4. Compile your code with the following commands:

   ```
   bison -d -v -r all myanalyzer.y
   flex mylexer.l
   gcc -o mycompiler lex.yy.c myanalyzer.tab.c cgen.c -lfl
   ```
5. Call the `mycompiler` executable for input `test.ka`:

   ```
   ./mycompiler< test.ka
   ```

Attention! You must first to do compile the `myanalyzer.y` and then the mylexer.l, because the `mylexer.l`
`myanalyzer.tab.h` is included in `mylexer.l`.

The text file `myanalyzer.output` produced by flag `-r all` will help you identify possible problems with conflicts like shift/reduce and reduce/reduce.

Every time you change `mylexer.l` and `myanalyzer.y` you have to do the whole process. It's a good idea to make a script or makefile for all of the above.

3.3 Deliverables

The deliverable for the coursework will contain the following files (from phase 2):

`mylexer.l`: To flex file.

`myanalyzer.y`: To file bison.

`correct1.ka, correct2.ka`: Two correct **Kappa** programs/examples.

`correct1.c, correct2.c`: The **C** equivalent of the above two. It is your responsibility to

show off your work through representative programs.

3.4 Examination

The following will be checked during the examination of your work:

Compilation of the delivered programs and creation of the executable analyzer. Unsuccessful compilation means that you have delivered a rough draft, as its operation cannot be seen. It goes without saying that you should be able to compile your code yourself!

Successful creation of the analyser. Your grade will be significantly affected by the number of shift-reduce and reduce-reduce conflicts that occur during the creation of your analyzer.

*Analyzer check on correct and incorrect examples of **Kappa** programs*. The ones in the Appendix will certainly be checked, but also other examples unknown to you. Good performance of at least the known examples is taken for granted. The result of the analysis on the set of examples will have a significant impact on your grade.

*Control analyzer in your own examples of **Kappa** programs*. Such checks will help in case you want to highlight something from your work.

Questions on implementation. You should be able to explain design issues, options and implementation methods, as well as each section of code you have delivered and answer the relevant questions.

4 Epilogue

In conclusion, we would like to stress that it is important to follow the instructions closely and deliver results according to the specifications set. This is something you must adhere to as engineers, so that in the future you can work collectively in large work teams, where consistency is key to the coherence and success of each project.

Clarifications will be provided during the semester where necessary. For questions, please contact the course lab instructors. General questions are best discussed in the course discussion area in eClass for your colleagues to see.

Good luck!

ANNEX

5 Examples of Kappa programs

5.1 Hello World!

```
-- My first Kappa program. File: myprog.ka
const message = "Hello world!\n" :str;
def main():
  writeStr(message);
enddef;
```

The desired result of verbal - syntactic analysis:

```
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 RIGHT_PARENTHESIS:      )
Token SEMICOLON:              ;
Token KEYWORD_ENDDEF:         enddef
Token SEMICOLON:              ;


Your program is syntactically correct!
```

5.2 Settings of Kappa

Example for understanding function syntax in the **Kappa** language.

```
-- File: useless.ka
-- A piece of Kappa code for demonstration purposes

const N= -100: integer;

a, b: integer;

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

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

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

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

The above program could be mapped to **C** as follows:

```c
#include "kappalib.h"
/* program */
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);
   writeInteger(j);
   return j;
}
int main() {
   a= readInteger(); readInteger();
   b= readInteger();
   add(a, b);
}
```

Then, the above **C** program can be translated by the compiler into executable with the command

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

5.3 Raw numbers

The following example program in **Kappa** is a program that calculates the prime numbers between **1** and **n**, where **n** is given by the user.

```
-- File: prime.ka
limit, num, counter: integer;

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

  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;
  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 Example with a syntax error

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

The desired result of verbal - syntactic analysis:

```
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)
```