TECHNICAL UNIVERSITY OF CRETE

School of Electrical and Computer Engineering

PLI 402

THEORY OF COMPUTATION

Programming Assignment
Verbal and Syntactic Analysis
of the MiniScript Programming Language

Teacher

Michael G. Lagoudakis

George

Anestis Laboratory

Spring Semester 2020

Last update: 2020-03-27

1 Introduction

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

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

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

The work includes two parts:

- **Parser** implementation for the **MiniScript** language using **flex**
- Implemented **a parser** for the **MiniScript** language using **bison**
    - ÿ Convert **MiniScript** code to **C** code using **bison** actions

Observations

- The assignment will be prepared **individually.** Plagiarism, even from earlier work, can be established very easily and leads to nullification.
- Computers of the IT Center can be used to prepare the work and personal computers. The flex and bison tools are available on any Linux distribution.
- The assignment will be submitted **electronically** through the course website at courses. The deliverable archive file **(.zip** or .tar) should contain all necessary files according to the specifications of the work.
- The assignment must be submitted by the deadline. Late assignments will not be accepted. Non handing in the assignment automatically results in a failure in the course.
- The evaluation of the work will include **an examination of the proper functioning** of the deliverable program, according to the specifications, as well as **an oral examination** in which you will have to explain each piece of code you have submitted and answer the related questions. The exam will take place on days and times to be announced.
- The part of the work related to Verbal Analysis corresponds to 30% of its total grade work and the remaining 70% corresponds to the Editorial Analysis.
- Reminder: the grade of the assignment should be at least **40/100.** Therefore, it is not enough to submit only the Verbal Analysis section.

2 The MiniScript programming language

The description of the **MiniScript** language below follows the general format of a language description programming. It probably also contains elements that do not fit into the verbal or syntactic analysis. It is your responsibility to recognize these elements and ignore them when developing your parser. Each program in **MiniScript** language is a set of *verbal units,* which are arranged based on *syntax rules,* as described below.

2.1 Verbal units

Verbal units of the **MiniScript** language are divided into the following categories:

- Keywords , *which* are reserved and cannot be used as identifiers or to be redefined, which are the following:

number	**boolean**	**string const**	**void**	**true**
false	**var**		**if**	**else**
for	**while**	**function**	**break**	**continue**
not	**and**	**or**	**return**	**null**

Keywords are case-sensitive, meaning you cannot capitalize them.

- The ***identifiers*** *(identifiers)* used for variable and function names and consist of a lowercase or uppercase letter of the Latin alphabet, followed by a series of zero or more lowercase or uppercase letters, decimal digits, or characters underscore. Identifiers must not match keywords.

  Examples: x y1 angle myValue Distance_02

- The ***numerical constants*** *(number constants),* consisting of an integer part, a fractional part and an optional exponent. The integer part consists of one or more digits (of decimal system) without extra leading zeros. The fractional part consists of the character of the decimal point **(.)** followed by one or more decimal digits. Finally, the exponent consists of the lowercase or uppercase letter E, an optional **+** or **-** sign , and an or more digits of the decimal system without extra leading zeros.

  Examples: 42 42.4 4.2e1 0.420E+2 42000.0e-3

- The ***logical constants*** *(boolean constants),* which are the word-values **true** and **false.**

- ***Constant strings ,*** *consisting* of a sequence of common characters or escape characters within double or single quotes. Common characters are all printable characters except single and double quotes and the **** (backslash) character. The escape characters begin with **** (backslash) and are described in the table below.

Escape character	Description
\n	line feed character
\t	tab character
\r	return character at the beginning of the line
****	**** character (backslash)
\\"	character **"** (double quote)

A constant string cannot span more than one line of the input file.
Here are examples of valid strings:

"M" "\n" '\"' "abc" 'def' "Route 66"

'Hello world!\n' "Item:\t\"Laser Printer\"\nPrice:\t$142\n"

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

arithmetic operators:	**+**	**-**	*****	**/**	**%**	******
relational operators:	**==**	**!=**	**<**	**<=**		
logical operators:	**and**	**or**	**not**			
case operators:	**+**	**-**				
assignment operator:	**=**					

Operator	Description
+, -, *, /	addition, subtraction, multiplication, division
%	division remainder
******	Exposure to force, e.g. **2**3, a**2,** etc.
==	equal to (=)
<	less than (<)
<=	less than or equal to (ÿ)
!=	different from (ÿ)
and	logical coupling
or	logical decoupling
not	reasonable denial

• The ***delimiters ,*** which are the following:

; ( ) , [ ] { } :

In addition to the lexical units mentioned above, a **MiniScript** program can also contain and elements that are ignored (i.e. recognized, but no analysis is made):

• White ***space ,*** i.e. sequences consisting of spaces, tab characters, line feed characters or carriage return characters of the line (carriage return).

• ***Comments ,*** which start with the character sequence **/*** and end with the first subsequent occurrence of the character sequence ***/.** Comments may not be nested. In the inside them any character is allowed to appear.

• ***Line comments*** *(line comments),* which start with the // character sequence and extend to end of current line.

2.2 Syntax rules

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

i) Programs

A **MiniScript** program can be inside a file with the extension .ms and consists of the following components which are listed in this order and **separated** from each other by the separator symbol ; :

- Constant declarations (optional)
- Variable declarations (optional)
- Function definitions (optional)

• The main structural unit which is the **start function,** which takes no arguments and is assumed not to return any value (return type **void).** This function is the starting point for program execution and is of the form:

```
function start() : void { body of start

}
```

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

```
function start() : void {
        var x: number;
        x = 1 + 2 + 3 + 4?
        writeNumber(x); return 0;

}
```

i) Data types

The **MiniScript** language supports four basic data types:

number: numbers (integer and real)

string: characters

boolean: logical values

void: no value, used as the return type of functions that do not return a value

MiniScript also supports one-dimensional arrays consisting of **n** elements of type **type of** the form:

• **identifier[n] : type ,** where array size **n** should be a positive integer constant and **type** a valid base type. Example: **listOfNums[10] : int**

iii) Variables

Variable declarations begin with the keyword **var.** This is followed by one or more variable identifiers separated by commas, and finally the : separator , followed by a data type. Multiple contiguous variable declarations are separated from each other by the delimiter , and must be included in the same **var** and base type. Variables can be initialized to a value when declared using the = assignment operator. Here is an example of variable declarations:

```
var i, j = 10.5: number; var s1, s2: string;
var n1: number; var x = 3.5, y:
number; var s = "hello",
ss: string; var test = false: boolean;
```

iv) Fixed

Constants are declared like variables, using the **const** keyword instead of **var,** except that they must be initialized using the assignment operator. Example:

```
const pi = 3.14: number;
```

v) Functions

The function (function) is a structural unit, which consists of the following components, which listed in this order:

- **function** *function name(parameter declarations): return type* **{**

  local variable and constant declarations (optional)

  commands

  return *expression* (optional)

  }

A function declaration begins with the **function** keyword followed by the function name, followed by its parameters in parentheses, the : separator followed by its type return result and the function body inside brackets **{ and }.** The brackets are mandatory, even if a function has no parameters. The return type can be and array using the symbols **[]** before the formula. The return type can be omitted if the function does not return any specific value. If the function does not contain a **return** statement or the **return** statement does not returns some value, then the return value is assumed to be ***void.*** Here are definition examples valid functions:

function f1(b: number, e: number) : number { return b ** function f2(s[]: string) : huh? }
number { return 100; }
function f3(x[]: number) : [] number { x[2] = 2 * x[4]; return x?}

A function in its body contains declarations of (optionally) local variables, constants and its commands.

MiniScript supports a set **of** predefined functions, which are at its disposal developer for use anywhere within the program. Below are their headings:

function readString(): string function
readNumber(): number
function writeString(s: string)
function writeNumber(n: number)

i) Expressions

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

MiniScript operators are divided into one-argument operators and two-argument operators. Of the first, some are written before the argument (prefix) and some after (postfix), while the latter are always written between of arguments (infix). Arguments of two-argument operators are evaluated from left to right right. The following table defines the precedence and associativity of the **MiniScript operators.** The operators that appear higher in the table come first. The operators on the same line have the same priority. Note that parentheses can be used in an expression to denote the desired priority.

Performers	Description	Arguments	Position, Cooperativeness
not	Logical negation operator	1	prefix, right
+ -	Operators of the sign	1	prefix, right
******	Increase in strength	2	infix, right
*** / %**	Operators with factors	2	infix, left

+ -	Verbs with conditionals	2	infix, left
== != **< <=**	Relational operators	2	infix, left
and	Logical coupling	2	infix, left
or	Logical disconnection	2	infix, left

Here are examples of correct expressions:

-a -- opposite of variable a
a + b * (b / a) 4 + -- numeric expression
50.0*x / 2.45 (a+1) % -- numeric expression
cube(b+3) (a <= b) and -- numeric expression with function call
(d <= c) -- logical and relational operators
(c+a) != (2*d) -- arithmetic with relational operators
a + b[(k+1)*2] -- numeric expression with matrix

vii) Commands

The commands (statements) supported by the **MiniScript** language are the following (all commands, except the complex, are considered simple):

- *The compound instruction,* consisting of a **(non-** empty) sequence of simple instructions delimited by the separators **{ and }.**
- The *assignment statement* **v = expr,** where **v** is a variable and **expr** is an expression.
- The *control statement* **if (expr) stmt1 else stmt2.** The **else** section is optional. expr is an expression, while **stmt1** and **stmt2** are simple or complex commands.
- The *iteration statement* **for (stmt1 ; expr ; stmt2) stmt,** where **stmt1, stmt2** are simple assignment statements executed before start and at each iteration respectively, **expr** is optional expression checked/evaluated before each iteration and **stmt** is simple or compound command that is executed on each iteration.
- The *loop command* **while (expr) stmt.** expr is an expression and **stmt is** a simple or complex command.
- The **break** *instruction* that causes immediate exit from the innermost loop.
- The **continue** *command* which causes the current iteration to stop and start next iteration of the loop it is in.
- The *return command* **return** or **return expr,** which terminates (possibly prematurely) its execution function it is in and returns, where **expr** is an (optional) expression.
- The *command to call* a function **f(expr1,...,exprn),** where **f** is the name of the function and **expr1,...,exprn** are expressions corresponding to the declared arguments of the function.

Each command (simple or complex) of the **MiniScript** language is terminated with the delimiter ? at the point where appears, regardless of whether other commands follow or not. Excludes the if , **for, while** and two assignment statements in the parentheses of the **for statement.**

2.3 Mapping from MiniScript to C99

C99 is the revision of the **C** language standard made in 1999. In this revision were added various useful extensions to the somewhat old **C89.** See the corresponding Wikipedia article for more details. As **C99** is a rich language, it is particularly easy to assign programs of **MiniScript** in **C99 programs.** The details of this visualization will be described below.

2.3.1 Mapping types and constants

MiniScript types are mapped to **C99** types based on the following table:

Type of MiniScrip	Corresponding type of C99
number	**double**
boolean	**int**
string	**char***
void	**void**
array[n]: T	**T array[n]**
[]: T	**T***
function func(a1: T1, ... ak: Tk) : type type (*) func(T1 a1, ...	**Thank you)**

where **T, T1, ...,** **Tk** is some type of **MiniScript.**

Based on the table above, the **MiniScript** constants are also mapped to C99 constants . For for example, the **MiniScript** boolean constants **true** and **false** map to integer values.

2.3.2 Matching building blocks

A **MiniScript** program optionally includes declarations of variables, functions, and mandatorily part of the main code, namely the special **start** function and corresponds to a .c file which includes, declarations of global variables, functions and the original **main() routine.**

The mapping is as follows:

- A **MiniScript** variable **foo** with type **T**

  var foo, bar: T;

  corresponds to a variable with the same name and type assigned

  T foo, bar?

- A **MiniScript** function corresponds to a **C99** function of the same name and their counterparts parameter types.

MiniScript function	Function of C99
function foo(x1: T1,, x2: T2,..., xn: Tn): type	**type foo(T1 x1, T2 x2,...,Tn xn)**

- Program commands are assigned in an obvious way.
- Library calls could be implemented as follows:

Call MiniScript	Implementation function in C99
readString(): string	Use the implementation given to you in the **mslib.h** file
readNumber(): number	

writeString(s: string)	
writeNumber(n: number)	

MiniScript 's predefined functions are treated like any other function. When converting **MiniScript** source code to **C,** be sure to include **(#include)** in the generated **C** code the given **mslib.h** file , which contains the C implementation of MiniScript's predefined **functions .**

3 Detailed job description

3.1 The tools

To complete the job successfully you need to know well programming in C, **flex** and **bison.** The **flex** and **bison** tools are developed as part of the GNU program and can be found on all GNU software hubs on the web (eg **www.gnu.org).** More information, manuals and links for 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, but there may be slight differences in other distributions.

3.2 Approach to work

For your convenience in understanding the tools you will use and how these tools work together, it is suggested that the work be carried out in two phases.

- **1st phase: Verbal Analysis**

The final product of this phase will be a Verbal Analyzer, i.e. a program that will take as input a file with a program of the **MiniScript** language and will recognize the verbal units (tokens) in this file. Its output will be a list of the tokens it read and their designation. For example, to enter:

i = k + 2;

the output of your program should be

token IDENTIFIER: i

token ASSIGN_OP : = token
IDENTIFIER : k

token PLUS_OP: token **+**

CONST_INT: 2 token
SEMICOLON: ;

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

i = k ^ 2?

the output of your program should be

token IDENTIFIER: i

token ASSIGN_OP : = token
IDENTIFIER : k

Unrecognized token ^ in line 46: i = k ^ 2?

where 46 is the line number within the input file where the particular command is located including comment lines.

To build a Parser you will use the flex tool and the gcc compiler. Enter man flex at the command line to view the flex manual or refer to the PDF file found in courses. 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 .l extension, e.g. mylexer.l.
2. Compile by typing flex mylexer.l on the command line.
3. Issue ls to see the lex.yy.c file produced by flex.
4. Build the executable with gcc -o mylexer lex.yy.c –lfl
5. If you have no errors in mylexer.l, the mylexer executable is produced.
6. Run with ./mylexer < example.ms, to input example.ms.

Every time you change mylexer.l you should do the whole process:

flex mylexer.l

gcc -o mylexer lex.yy.c -lfl

./mylexer < example.ms

So it's a good idea to make a script or makefile to do all of the above automatically.

• 2nd phase: Editorial Analysis and Translation

The final product of this phase will be a **MiniScript** to C Parser and Translator , i.e. a program that will take as input a file with a **MiniScript** language program and recognize whether this program follows **MiniScript syntax rules.** It will output the program it identified, in C, if the given program is syntactically correct, otherwise it will display the line number where the first error was diagnosed, the contents of the line with the error, and *optionally* an informative diagnostic message. For example, for the wrong entry

```
...
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 line number within the input file where the particular command is located including comment lines.

To build a parser and translator you will use the bison tool and the gcc compiler. Enter man bison to view 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 you already have the parser ready in mylexer.l.
2. Write the bison code to a file with a .y extension, e.g. myanalyzer.y.
3. To join flex with bison you need to do the following:
    - Put the files mylexer.l and myanalyzer.y in the same directory.
    - Remove the main function from the flex file and make a main in the bison file. To begin with, all the new main needs to do is call bison's yyparse() macro once. yyparse() repeatedly runs yylex() and tries to match each

token that the Parser returns to the grammar you've written in the Parser. Returns 0 for successful termination and 1 for termination with a syntax error.

- Remove the defines you made for the tokens in flex or some other .h file. These will now be declared in the bison file one per line with the %token command. When you compile myanalyzer.y, a file named myanalyzer.tab.h is automatically created. You should include this file in the mylexer.l file and thus flex will understand the same tokens as bison.

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 -lfl

5. Call the mycompiler executable to input test.ms by writing:

./mycompiler < test.ms

Caution! You must first compile myanalyzer.y and then mylexer.l because myanalyzer.tab.h is included in mylexer.l.

The myanalyzer.output text file produced with the –r all flag will help you identify potential shift/reduce and reduce/reduce problems.

Every time you change mylexer.l and myanalyzer.y you will have to go through the whole process. It is a good idea to make a script or a makefile for all of the above.

3.3 Deliverables

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

- mylexer.l: The flex file.
- myanalyzer.y: The bison file.
- mycompiler: Your parser executable.
- correct1.ms, correct2.ms: Two correct **MiniScript programs/examples.**
- correct1.c, correct2.c: The equivalent programs of the two above in C language.

It is your responsibility to showcase your work through representative programs.

3.4 Examination

When reviewing your work, the following will be checked:

- *Compile the deliverables and create the executable parser.* A failed compilation means that you have submitted rough work, as it cannot be seen to work.
- *Successful creation of the parser.* Your grade will depend on the number of shift-reduce and reduce-reduce conflicts that occur when building your parser.
- *Parser check on correct and incorrect* ***MiniScript program examples.*** Those in the Appendix will certainly be checked, as well as other examples unknown to you. Good execution of at least the known examples is taken for granted.
- *Parser control in your own* ***MiniScript example programs.*** Such checks will help to case you want to show off something from your work.

- *Implementation questions.* You should be able to explain design issues, options and implementations as well as each part of the code you have provided and answer to the relevant questions. Also, you should be able to compile your code yourself.

4 Epilogue

In closing we would like to emphasize that it is important to follow the instructions closely and deliver results according to the specifications that have been set. This is something you must adhere to as engineers so that in the future you can work collaboratively in large workgroups, where consistency is key for the coherence and success of each project.

During the semester, clarifications will be given where necessary. For questions you can contact teacher and to the course laboratory managers. General questions should be discussed at course discussion area for your colleagues to see.

Good luck!

ANNEX

5 Examples of MiniScript programs

5.1 Hello World!

```
/* My first MiniScript program. File: myprog.ms */
const message = "Hello world!\n" : string;
function start(): void {
    writeString(message);
}
```

Desired result of verbal and syntactic analysis:

KEYWORD_CONST token:	const
Token IDENTIFIER:	message
Token ASSIGN_OP:	=
Token CONST_STRING:	"Hello World!\n"
Token COLON:	:
Token KEYWORD_STRING:	string
SEMICOLON tokens:	;
KEYWORD_FUNC token:	function
KEYWORD_START token:	start
Token LEFT_PARENTHESIS:	(
Token RIGHT_PARENTHESIS:	)
Token COLON:	:
Token KEYWORD_VOID:	void
Token LEFT_CURLY_BRACKET:	{
Token IDENTIFIER:	writeString
Token LEFT_PARENTHESIS:	(
Token IDENTIFIER:	message
Token RIGHT_PARENTHESIS:	)
SEMICOLON tokens:	;
Token RIGHT_CURLY_BRACKET:	}

Your program is syntactically correct!

5.2 MiniScript Functions

Example to understand the syntax of functions in the **MiniScript language.**

```
// File: useless.ms

// A piece of MiniScript code for demonstration purposes


const N = -100 : number;


var a, b: number;


function cube(i: number): number {
    return i*i*i;
}


function add(n: number, k: number): number {
    var j: number;


    j = (Nn) + cube(k);
    writeNumber(j);
    return j;
}


/* Here you can see some useless lines.
  * Just for testing the multi-line comments ...
  */
function start(): void {
    a = readNumber();
    b = readNumber();
    add(a, b); // Here you can see some dummy comments!
}
```

The above program could be indicatively translated as follows:

```
#include <stdio.h>
/* MiniScript Library */
double readNumber() { double ret; scanf("%lf", &ret); return ret; }
void writeNumber(double n) { printf("%0.3lf",n); }

const double N = -100;

float a, b;

double cube(double i) {
    return i*i*i;
}

double add(double n, double k) {
    double j?

    j = (Nn) + cube(k);
    writeNumber(j);
    return j;
}

void main() {
    a = readNumber();
    b = readNumber();
    add(a, b);
}
```

It can be translated by the compiler with the command

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

Machine Translated by Google

5.3 Prime numbers

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

```
// File: prime.ms
var limit, num, counter: number;

function prime(n: number): boolean { var i: number; var 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;

           };
           result = isPrime;
      };

      return result;
}

function start(): void {

    limit = readNumber(); counter = 0;
    num = 2;


    while (num <= limit) { if (prime(num))
        { counter = counter + 1;
            writeNumber(num); writeString(" "); };
            num = num + 1;



    };

    writeString("\n");
    writeNumber(counter);
}
```

5.4 Example with a syntax error

```
1 /* My first MiniScript program */
2 const message = "Hello world!\n": string;
3
4 function start(): void {
5         writeString(message
6 }
```

Desired result of verbal and syntactic analysis:

KEYWORD_CONST token:	const
Token IDENTIFIER:	message
Token ASSIGN_OP:	=
Token CONST_STRING:	"Hello World!\n"
Token COLON:	:
Token KEYWORD_STRING:	string
SEMICOLON tokens:	;
KEYWORD_FUNC token:	function
KEYWORD_START token:	start
Token LEFT_PARENTHESIS:	(
Token RIGHT_PARENTHESIS:	)
Token COLON:	:
Token KEYWORD_VOID:	void
Token LEFT_CURLY_BRACKET:	{
Token IDENTIFIER:	writeString
Token LEFT_PARENTHESIS:	(
Token IDENTIFIER:	message
Token RIGHT_CURLY_BRACKET:	}

Syntax error in line 5: writeString(message

the

Syntax error in line 5: writeString(message
(Missing parenthesis)