IMPLEMENTATION OF A CUSTOM LANGUAGE FOR DIGITAL IMAGE PROCESSING

Florin Georgian MARIN[1], Florin-Marian BÎRLEANU[2]
Faculty of Electronics, Communications and Computer Science,
University of Pitesti, Romania
[1]george.marin9520@gmail.com, [2]florin.birleanu@upit.ro

Keywords: educational software, image processing, domain specific languages, parsing

Abstract: *The paper presents the implementation of an interpreter for a custom programming language that enables the user to easily test basic image processing algorithms by writing short scripts. The interpreter is implemented as a web application using the PHP language for the backend and is useful as a lightweight didactic tool for teaching image processing.*

1. INTRODUCTION

Basic image processing skills are more and more important for computer engineers nowadays. While images are nothing more than two-dimensional signals [1], the visualization and interpretation of the effect of different filters on them is significantly different from the case of acoustic or other one-dimensional signals. From a didactic point of view, understanding the mathematics behind image filtering is somewhat less important than being able to visually understand the effects of each filtering operation. In this context, it is very useful to be able to play with these filters on different images and easily see the efects of varying each parameter. Of couse, scientific programming languages such as Matlab and Python offer this possibility through the use of specific toolboxes and modules [2, 3]. However, they require a significant amount of resources and are not very easy to learn. The language we propose here is extremely simple to learn (as it focuses only on a very limited set of basic image processing operations) and its interpreter is lightweight and portable, being implemented as a PHP based web application [4]. We present the design and implementation of our application in Section II and we show some examples of use and results in Section III. In the final section we present the conclusions, as well as some directions for future improvements.

2. DESIGN OF THE INTERPRETER

A. The source language

The source language for our interpreter should allow the user to easily load images from the local storage, extract the basic color components, compute the luminance, binarize, perform convolution based filtering, erosion, dilation, image composition from basic color components, and display of the results. It should also allow repetitions and basic math operations. All of these instructions are summarized in Table 1.

Table 1. The instructions of our custom language.

No.	**Instruction format**	**Operation**
1	*img* = incarca ”*image.jpg*”	Load image from file *image.jpg* into variable *img*
2	*img_y* = luminanta *img*	Compute luminance of the image *img*
3	*img_r* = extrage_componenta ”rosu” *img* *img_g*= extrage_componenta ”verde” *img* *img_b* = extrage_componenta ”albastru” *img*	Extract the basic color components from the image *img* into the images *img_r*, *img_g* and *img_b*
4	*img_bin* = binarizeaza *img_y127*	Binarize the image *img_y* using *127* as the threshold value

5	*img_blur* = blureaza *img*	Blur the image *img*
6	*img_plus = img+50*	Add the value *50* to each element in the image *img*
7	*img_rg0* = combina *img_rimg_g0*	Combine three grayscale images into a color image
8	*elem* = [1,1,1; 1,1,1; 1,1,1;] *img_e* = erodeaza *img_binelem*	Erode image using the structuring element *elem*
9	*elem* = [1,0,1; 1,1,1; 1,0,1;] *img_d* = dilateaza *img_binelem*	Dilate image using the structuring element *elem*
10	*filt* = [0,-0.25,0; -0.25,2,-0.25; 0,-0.25,0;] *img_f* = filtreaza *imgfilt*	Filter image by performing convolution with the kernel *filt*
11	afiseaza *img*	Display image *img*

12	repeta *3* {
	/* sequence of instructions */
	}
		Repeat the sequence of instructions *3* times

The grammar of the language is very simple, as a valid program could contain any sequence of instructions. All of the instructions have a fixed length, except for the repetition instruction (which can contain any sequence of instructions).

The Filter instruction is particularly interesting because it actually performs the convolution (see Figure 1) of the image with a kernel that is shifted over each pixel. Through this convolution operation different filters can be realized by only modifying the values from the kernel matrix. (For instance, a low-pass filtering could be done by choosing values such that the total sum of the elements is 1, while the low frequencies in the image are removed if the total sum of the elements is 0.)

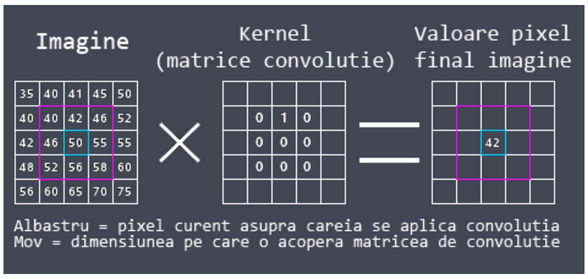

Imagine
Kernel (matrice convolutie)
Valoare pixel final imagine
35 40 41 45 50
40 40 42 46 52
42 46 50 55 55
48 52 56 58 60
56 60 65 70 75
0 1 0
0 0 0
0 0 0
42
Albastru = pixel curent asupra careia se aplica convolutia
Mov = dimensiunea pe care o acopera matricea de convolutie

Fig. 1. The operation of convolution of an image with a kernel. The current pixel is marked with blue and its neighbourhood (that is used in the computation) is marked with purple.

B. The interpreter

Figure 2 shows the graphical user interface of our interpreter. It contains the following elements: (1) a sidebar (on the left) for inserting templates for each of the available instructions, (2) an editable input text area where the instructions are inserted, (3) an output panel (on the right) for showing the displayed images, (4) a console (on the bottom) where text messages (or errors) from the interpreter are shown, (5) a fullscreen button on the bottom-right corner of the output panel, (6) a Run Code button (below the bottom-left corner of the input text area, (7) Verify Code, Save Code and Help buttons.

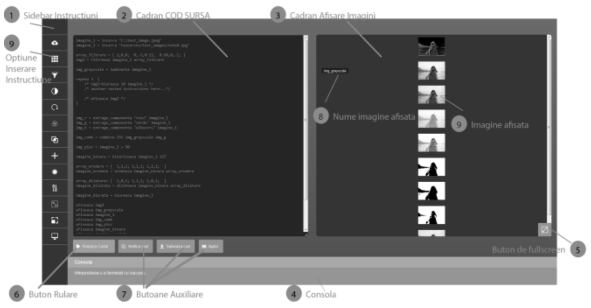
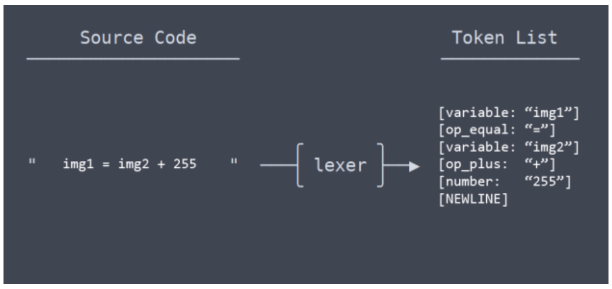


Fig. 2. The graphical user interface of the interpreter.

The main function of the application is split into three modules: Tokenizer, Parser, and Interpreter – that correspond to the three main steps in language interpretation [5, 6].

The **Tokenizer module** performs the lexical analysis of the input source code, *i.e.* it groups the individual characters (symbols) into tokens (words). It outputs a list of tokens (in the order of their appearance), each element in the list containing the type and value of the token.



```
" img1 = img2 + 255 "
```

```
[variable: "img1"]
[op_equal: "="]
[variable: "img2"]
[op_plus:  "+"]
[number:   "255"]
[NEWLINE]
```

Fig. 3. The operation of the Tokenizer module.

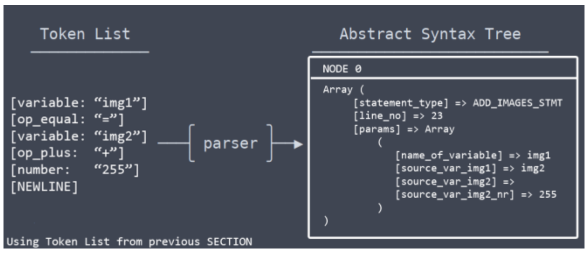
The lexical analysis step can be summarized by the following algorithm:

Algorithm 1: **Lexical analysis of the input source code**

```
1: Tokenizer.ADD("INCARCA_F", "/^incarca/");
2: Tokenizer. ADD("EQUALS", "/^=/");
3: Tokenizer. ADD("VARIABLE", " /^[a-zA-Z][a-zA-Z0-9_]*/");
4: Tokenizer. ADD("STRING", " /^("|')(\\?.)*?\1/ ");
...
5:  function TOKENIZE( source_code )
6: lengthOfPatterns ← TotalAtomiAdaugatiCuAdd()
7: tokens_array ← []
8: i ←0
9: for i = 0, i <= lengthOfPatterns do
10: if match_regex( currentAtom , source_code )then
11:        tokens_array ← currentAtom
12: souce_code ← remove( previouslyFoundAtomName , source_code)
13: return tokens_array
14:Tokenizer.TOKENIZE( source_code )
```

First, the tokenizer receives the regular expressions for all the token types it must recognize. Then, the input source code is scanned in order to find a match with one of the regular expressions. When such a match is found, the matching part of the input is consumed and a new token is added to the output list of tokens.

The **Parser module** performs the syntax analysis, *i.e.* it checks whether the order of appearance of the token types in the list returned by the Tokenizer module satisfies the grammar of the language. As we already said, the grammar of our language is very simple (as it can be noticed from the format of the instructions in Table 1). Hence, we can parse it using a top-down LL(k) predictive parser [6]. This parser will output an Abstract Syntax Tree (AST), as shown in Figure 4.



```
[variable: “img1”]
[op_equal: “=”]
[variable: “img2”]
[op_plus:  “+”]
[number:   “255”]
[NEWLINE]
```

```
NODE 0
Array (
     [statement_type] => ADD_IMAGES_STMT
     [line_no] => 23
     [params] => Array
         (
             [name_of_variable] => img1
             [source_var_img1] => img2
             [source_var_img2] =>
             [source_var_img2_nr] => 255
         )
)
```

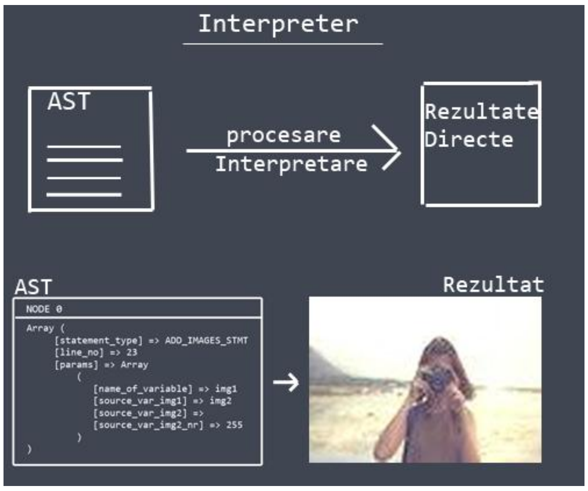
Fig. 4. The operation of the Parser module.

The syntax analysis step can be summarized by the following algorithm:

Algorithm 2: **Syntax analysis of the list of tokens**

```
1: function PARSE( tokens )
2: while  !endOfTokens()
3:   switch TokenUrmator()
4:     case 'VARIABLE'
5:         switch TokenUrmator(3) //look ahead 3 tokens
6:            case 'INCARCA_F'
7:              AST_ADD_INSTRUCTION_NODE(Incarca)
8:              .....
9:            case 'FILTER_F'
10:             AST_ADD_INSTRUCTION_NODE(Filtreaza)
```

The **Interpreter module** visits the AST generated by the parser and executes the instructions corresponding to each of the visited nodes, as sketched in Figure 5.



```
NODE 0
Array (
     [statement_type] => ADD_IMAGES_STMT
     [line_no] => 23
     [params] => Array
         (
             [name_of_variable] => img1
             [source_var_img1] => img2
             [source_var_img2] =>
             [source_var_img2_nr] => 255
         )
)
```

Fig. 5. The operation of the Interpreter module.

The interpretation step can be summarized by the following algorithm:

Algorithm 3: **Interpretation of the Abstract Syntax Tree**

```
1: function INTERPRET( AST ):
2: astNodes ← AST
3: for each astNodes as instruction do
4:   instruction_type = instruction['tip_instructiune']
5:   if instruction_type == "INCARCA_STMT"then
6:      INCARCA_IMAGINE();
7:   else if instruction_type == "FILTER_1"then
8:      PROCESARE_IMAGINE_1();
9:      .....
10:  else instruction_type == "FILTER_N"then
11:        PROCESARE_IMAGINE_N();
```

The instructions are interpreted sequentially, except for the sequence inside a Repeat instruction, which is interpreted again for the specified number of times.

3. RESULTS

In order to illustrate the operation of our interpreter, we present next some examples of using the resulted application.

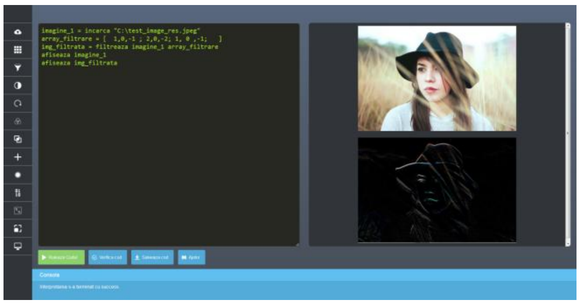
A. Edge detection

In this first example we show how we can use our custom scripting language to perform edge detection in an image. Although there is no instruction for edge detection in the proposed language, we can use the Filter instruction together with a proper kernel (corresponding to a high-pass filter), as shown in the following source code:

Script 1: **Edge detection**

```
img_1 = incarca ”C:\test_image_1.jpg”
filt = [ 1, 0, -1 ; 2, 0, -2; 1, 0, -1; ]
img_filt = filtreaza img_1 filt
afiseaza img_1
afiseaza img_filt
```

The result can be seen in Figure 6.



```
imagine_1 = incarca "C:\test_image_res.jpeg"
array_filtrare = [  1,0,-1 ; 2,0,-2; 1, 0 ,-1;   ]
img_filtrata = filtreaza imagine_1 array_filtrare
afiseaza imagine_1
afiseaza img_filtrata
```

Fig. 6. The result obtained after running the code for edge detection.

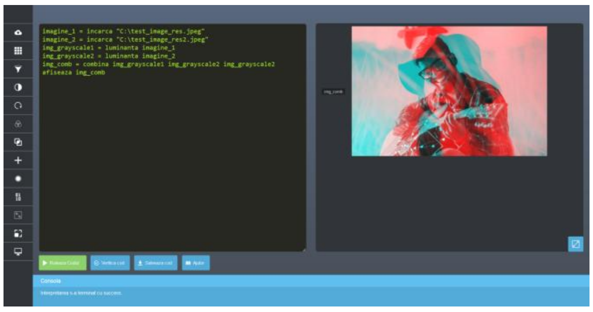
B. Image superposition

Next, we show how we can obtain a new image from the superposition of two images (of the same size). We can do this by using the Combine instruction, which takes three grayscale images as parameters and uses them as the red, green and blue color components in order to obtain the output image:

Script 2: **Image superposition**

```
img_1 = incarca ”C:\test_image_1.jpg”
img_2 = incarca ”C:\test_image_2.jpg”
y1 = luminanta img_1
y2 = luminanta img_2
img_comb = combina y1 y2 y2
afiseaza img_comb
```

The result of running this script in our interpreter can be seen in Figure 7.



```
imagine_1 = incarca "C:\test_image_res.jpeg"
imagine_2 = incarca "C:\test_image_res2.jpeg"
img_grayscale1 = luminanta imagine_1
img_grayscale2 = luminanta imagine_2
img_comb = combina img_grayscale1 img_grayscale2 img_grayscale2
afiseaza img_comb
```

Fig. 7. The result obtained after running the code for image superposition.

4. CONCLUSIONS

The paper presented the key steps in implementing an interpreter for a very simplified programming language that allows basic image processing operations on custom images. This dedicated language allows the user to combine the available operations in order to obtain more complex filters by writing short scripts and running them in our interpreter.

The toy language presented here can be easily extended to include new instructions (such as conversion to the hue-saturation-brightness color space or extraction of a specified area from an image).

5. REFERENCES

[1]. Russ, J., “*The Image Processing HandBook (Sixth Edition)*”, CRC Press, 2011.

[2]. Solomon, C., Breckon, T., “*Fundamentals of Digital Image Processing. A Practical Approach with Examples in Matlab*”, Wiley-Blackwell, 2011.

[3]. Dey, S., “*Hands-on Image Processing in Python*”, Packt, 2018.

[4]. Palala, J., Helmich, M., “*PHP 7 Programming Blueprints*”, Packt, 2016.

[5]. Spivak, R., “*Let’s Build a Simple Interpreter Blog*”, https://ruslanspivak.com/lsbasi-part1/, 2017.

[6]. Birleanu, F.-M., “*Limbaje formale, automate si compilatoare – Teorie si aplicatii*”, Editura Universitatii din Pitesti, 2016.