



FunctionCraft Programming Language Documentation

© 2024 University of Tehran

Shaghayegh Tavassoli

Mohammad Ali Zamani

Ali Ataollahi, Ali Darabi, Mohammad Javad Pesaraklo, Mohammad Reza Nemati,
Mohammad Sourì

Table of Contents

1 - Introduction	2
2 - General Structure	2
2 - 1 - General rules of syntax	2
2 - 2 - Comments	3
2 - 3 - Rules for naming functions and arguments	3
3 - Define Functions and Lambda Functions	4
4 - Function Call	5
5 - Data Types	5
6 - Parameters	5
7 - Operators	6
7 - 1 - Arithmetic operators	6
7 - 2 - Comparison operators	6
7 - 3 - logical operators	7
7 - 4 - Precedence of operators	8
7 - 5 - Append operator	9
7 - 6 - Assignment operator	9
8 - Decision Making Statements	10
9 - Scope Rules	11
10 - Default Functions and Fields	12
11 - Loop Structure	12
12 - Filter Structure	13
13 - Pattern Matching Structure	13
14 - Examples	14

1 – Introduction

This documentation methodically guides you through the key phases of this novel functional language developed at the University of Tehran. It initially establishes the foundational elements by outlining general syntax rules, code structure conventions, and defining functions, lambda expressions, and parameter handling. The documentation then delves into core features like supported data types and a comprehensive repertoire of operators. It provides an in-depth exploration of control flow constructs including decision statements, loops, list filtering, and pattern matching, loop clarifying scope rules. Finally, it explains the default functions, how to output things, work with strings, and run the main program. There are examples throughout to help you understand [FunctionCraft](#) fully.

2 – General Structure

Files in this language have `.fl` extensions. Each file is made up from two following parts:

- Declaration of a number of functions
- Declaration of the main function which is the program's entry point

In the following code you can see an example of the main structure of this language:



```
1  def get_size(i)
2      return len(g()[i]);
3  end
4
5  def g()
6      return [[1, 2]];
7  end
8
9  def main()
10     puts(get_size(0));
11 end
12
```

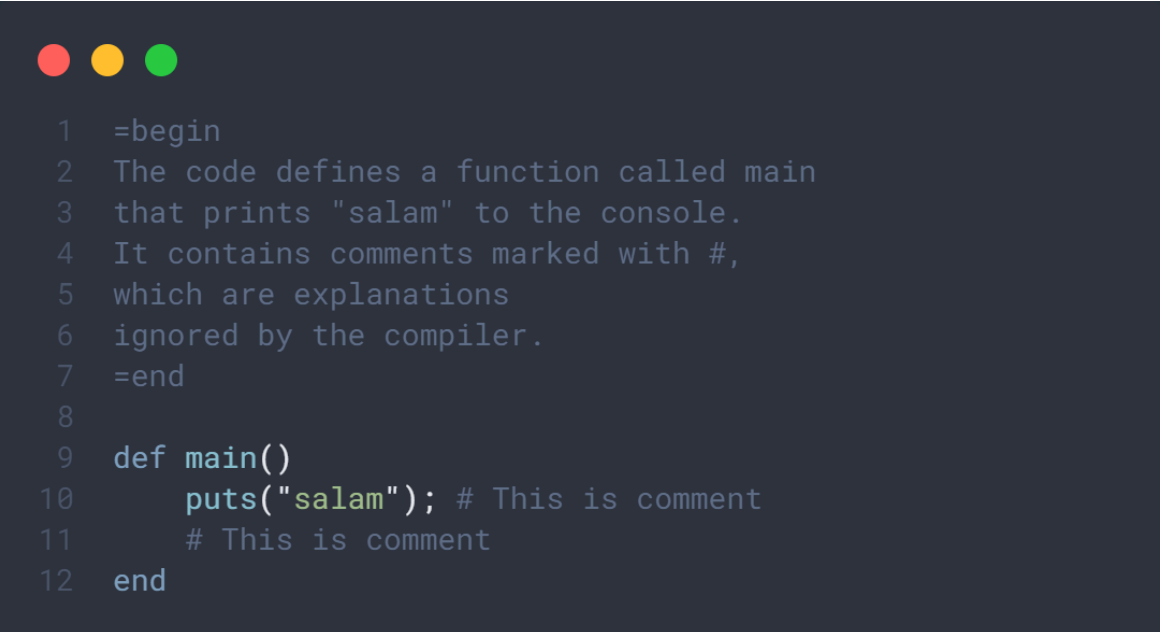
2 - 1 - General rules of syntax

Some of the general rules are in the following:

- Each statement should end with a semicolon.
- Empty lines have no effect on the program output or execution.
- The last statement to be written is always the main function declaration.
- After the *return* statement in each scope, no extra statement can be added.
- The order of arrangement of functions is without importance and in every function we can call any other function even if the called function is declared at a later point in the code.

2 - 2 - Comments

- Single-line comments are marked with a *#*. Every character between *#* and end of the line is considered a comment.
- Multi-line comments start with *=begin* and end with *=end*. Every character between *=begin* and *=end* of the line is considered a comment.



```
1  =begin
2  The code defines a function called main
3  that prints "salam" to the console.
4  It contains comments marked with #,
5  which are explanations
6  ignored by the compiler.
7  =end
8
9  def main()
10     puts("salam"); # This is comment
11     # This is comment
12 end
```

2 - 3 - Rules for naming functions and arguments

Every variable and function name should comply with the following rules:

- The string must consist of characters A...Z and a...z and numbers and underscores.
- It should not start with numbers.
- You can not use keywords of the language as variable names or function names.
- This table presents all keywords of the [FunctionCraft](#) language:

puts	push	len	return
if	else	elseif	main
return	true	false	chomp
chop	is	not	def
pattern	match	loop	do
method	method		

3 – Define Functions and Lambda Functions

- All function declarations start with the **def** keyword followed by the function name and parentheses that can contain parameters. The parameters are comma-separated. The function definition ends with the **end** keyword.
- The main function never has any parameters.
- The language has a special type of function called lambda functions. These functions are defined without naming them. They can only be called within that function where they are defined. The syntax for defining a lambda function is:

-> (param1, param2, ...) { function_body }

For example: **-> (a) {return a;}** This defines an lambda function that takes one parameter **a** and simply returns **a** in its body.

- lambda functions can be called after being defined.

```
1  -> (arg1, arg2) { return arg1 + arg2; } (1, 2);
```

This calls the above lambda function with arguments 1 and 2.

- A function can return a pointer to a lambda function.

```
1  def foo()  
2      return -> (arg1, arg2) {return arg1 + arg2;};  
3  end
```

- Functions can return without a value.

4 – Function Call


- All functions are called with parentheses. And arguments are separated by commas.
- Lists are passed by reference, while other types are passed by value.

5 – Data Types

- The language supports the following data types: integer (*int*), floating-point number (*float*), string (*string*), boolean value (*bool*), list (*list*), and function pointer (*fptr*).
- Strings are only enclosed in double quotes, like **"String"** (not single quotes 'String').
- All elements of a list have the same data type.
- Two-dimensional and multi-dimensional lists are also supported. Access to the elements is provided by integer indices.
- In order to create function pointers, *method* function is used. The syntax is *method(:function_name)*.
- Calling function pointers is just like calling a regular function.

6 – Parameters

Default values can be set for function parameters. In function calls, passing an argument with default value is optional. In this example, b and c have default values.



```
1 def f2 (a, [b = 10, c = 20])
2     return a + b + c;
3 end
```

7 - Operators

In this language, operators are divided into Arithmetic, Comparison, Logical, Precedence, Append, and Assignment operators.

7 - 1 - Arithmetic operators

These operators only work on numbers. The list of these operators is provided in the table below. In the examples used, consider A equals 100 and B equals 10.

Operator	Associativity	Explanation	Example
+	Left	Addition	$A + B = 110$
-	Left	Subtraction	$A - B = 90$
*	Left	Multiplication	$A * B = 1000$
/	Left	Division	$A / B = 10$
-	Right	Unary Negation	$-A = -100$

7 - 2 - Comparison operators

These operators are responsible for comparison, so their result must be either **true** or **false**. Note that the operands for the > and < operators should be of the int type, and the output of these operators is indeed a bool. Also, for operators like **is** and **is not**, the operands must be of the same type and necessarily of primitive types; otherwise, a compile error should be thrown.

The list of comparison operators is provided in the table below. In the examples used, consider the value of A to be 100 and the value of B to be 10.

Operator	Associativity	Explanation	Example
----------	---------------	-------------	---------

is	Left	Equality	(A is B) = false
is not	Left	Inequality	(A is not B) = true
<	Left	Less than	A < B = false
>	Left	Greater than	A > B = true

7 - 3 - logical operators

In this language, logical operators can be applied to the bool type. These operators are mentioned in the table below. In the examples, the values of A and B are true and false, respectively, for bool.

Operator	Associativity	Explanation	Example
&&	Left	Logical AND	(A && B) = false
	Left	Logical OR	(A B) = true
!	Right	Logical NOT	(!A) = false

7 - 4 - Append operator

This operator is used to add an element to a list. Note that the type of the element to be added must match the type of the other elements in the list.

In fact, this operator both updates and returns the list. They can be used consecutively, starting from the leftmost operation. This operator can also be used to concatenate strings.



```
1 list << a << b << c;
2 puts("Hello" << " " << "World!");
```

7 - 5 - Assignment operator

The assignment operators in this language are described in the table below. It's important to note that the types of parameters on both sides of the assignment operators must be the same; otherwise, an error will occur.

Operator	Associativity	Explanation	Example
=	Right	Sets the value of the left operand equal to the value of the right operand.	A = B
+=	Right	Increases the value of the left operand by the variable on the right side and replaces it with the new value.	A += B
-=	Right	Decreases the value of the left operand by the variable on the right side and replaces it with the new value.	A -= B
=*	Right	Multiplies the value of the left operand by the value of the right operand and replaces it with the new value.	A *= B
=/	Right	Divides the value of the left operand by the value of the right operand and replaces it with the new value.	A /= B
%=	Right	Calculates the remainder of the left operand divided by the right operand and replaces the value of the left operand with the new value.	A %= B

Function Pointer:

A function pointer can be assigned to a variable, which can then be used to call the function by adding parentheses and arguments after the variable name.



```
1 compare_ptr = method(:compare);  
2 puts(compare_ptr(a, b));
```

7 - 6 - Precedence of operators

Precedence	Category	Operators	Associativity
1	Parenthesis	()	Left to Right
2	Accessing list elements	[]	Left to Right
3	Prefix unary operator	!, -, ++, --	Right to Left
4	Multiplication and Division	*, /	Left to Right
5	Addition and Subtraction	+, -	Left to Right
6	Comparison	<, >	Left to Right
7	Equality Comparison	==, !=	Left to Right
8	Append	<<	Left to Right

8 - Decision Making Statements

- The conditions in if and while statements are not necessarily of boolean type.
- If and while must have parentheses. Logical operators come between the parentheses. In example:



```
1  if (a == 3) && (b == 1)
2      puts(a + b);
3      if (a == 3) && ((b == 1) || (c == 5))
4          puts(a + b + c);
5      end
6  end
```

This means the condition can have one or more sets of parentheses with logical operators between them (*&&* or *||*) .

In example: in C we implement an if statement like: `if(e1 && (e2 || e3))` . but instead, in [FunctionCraft](#) we implement it like: `if (e1) && ((e2) || (e3))`.

- The keyword "end" follows consecutive *if*, *elseif* and *else* blocks.



```
1  if (a)
2      ...
3  elseif (b)
4      ...
5  elseif (c)
6      ...
7  else
8      ...
9  end
```

- We can have any combination of if with elseif and else. However, there can be at most one else block.
- We do not have one-line if statements, and *end* must always be present. Inside if, else, and elseif blocks.

9 – Scope Rules

New scopes are generally created by the following:

- Arguments and lines of code inside a method
- Code inside decision statements

Some rules regarding Scope:

- Variables defined inside a function are not accessible outside that scope and are only accessible within the inner scopes.
- It is not possible to define arguments with the same name within a single scope.
- The return statement can optionally have parentheses around the value being returned.

10 – Default Functions and Fields

- puts: Prints variables and texts to the screen. Boolean values are printed as 1 and 0.
- push: Appends an element to a list or a character to a string.
- len: Returns the size of a list or string.
- main: This function is the entry point of any [FunctionCraft](#) program
- chop: Removes the last character in the string and returns the remaining string.
- chomp: Removes all newline characters in the string (\n) and returns the remaining string.

11 - Loop Structure

- In this language, we have the **loop do** structure which has no condition, and its scope ends with the **end** keyword.

```
1 loop do
2   puts("start");
3   ...
4   puts("end");
5 end
```

- We have the **for ... in ...** structure, which is used on a list or range. The range uses this syntax (**start..end**) where start and end are numbers. The **break**, **break if** and **next**, **next if** structures can be used within **loop do** and **for** loops:

```
1 list = [1, 2, 3, 4, 5, 6, 7, 8, 9];
2 for num in list
3   next if (num / 2 == 1);
4   puts(num);
5 end
6
7 start = 0;
8 finish = 9;
9 for num in (start..finish)
10  next if (num / 2 == 1);
11  puts(num);
12 end
```

12 - Filter Structure

- In this structure, a list on the right-hand side specifies the intersection of multiple lists. On the left-hand side, an arithmetic operation or function is performed on each element: [elements | range, condition1, condition2, ...]

- There can be any number of conditions (even none). For example:

```

1  def main()
2      listTimes3 = [x * 3 | x -> (1..10), x * 3 < 20];
3  end
4

```

(این مورد نوشته نشود) تابعی که در سمت چپ می نویسیم باید تایپ چک شود که تایپ یکسان داشته باشند.

(ایده) به شکل فیلتر در haskell نوشته بشه

13 – Pattern Matching Structure


- It's a structure similar to a switch case. It takes one or more inputs and like a function, it returns an output based on the input value.
- After each '|' character, a condition statement such as if comes, then after the '=' character, a value is returned.
- In each condition we observe the indentation (\t or 4 space).

```

1  pattern fib(n)
2      | (n == 0) = 1
3      | (n == 1) = 1
4      | (n > 2) = fib (n-1) + fib (n-2)
5      ;
6
7  def main()
8      fib_5 = fib.match(5);
9  end
10

```

- When we want to call the pattern and apply it to the inputs, we use this syntax:
pattern_name.match(arg1, arg2,...)



```
1 fib_5 = fib.match(5);
```

14 - Examples

lorem ipsum