# Falak Language Spec

Ariel Ortiz – ariel.ortiz@tec.mx – Version 1.0.0, August 25, 2021.

## Table of Contents



## 1. Introduction

This document contains the complete technical specification of the *Falak* programming language.

> *Falak is a giant serpent from Arabian mythology. It appears in the "One Thousand and One Nights". It resides below Bahamut, the giant fish, which carries along with a bull and an angel, the rest of the universe including six hells, the earths and the heavens. Falak itself resides in the seventh hell below everything else and it is said to be so powerful that only its fear of the greater power of Allah prevents it from swallowing all the creation above.*
>
> — *Wikipedia (https://en.wikipedia.org/wiki/Falak_(Arabian_legend))*

This language specification was developed for the 2021 autumn semester TC3048 *Compiler Design* course at the Tecnológico de Monterrey, Campus Estado de Mexico.

## 2. Lexicon

In the following sections, a letter is any character from the English alphabet from `A` to `Z` (both lowercase and uppercase). A digit is any character from `0` to `9`.

## 2.1. Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Spaces, tabulators, newlines, and comments (collectively, "white space") are used as delimiters between tokens, but are otherwise ignored.

If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

## 2.2. Comments

Comments can be either single or multi-line. Single line comments start with a hash symbol ( `#` ) and conclude at the end of the line. Multi-line comments start with a less than symbol followed by a hash symbol ( `<#` ) and end with a hash symbol followed by a greater that symbol ( `#>` ). Comments cannot be placed inside string literals. Multi-line comments cannot nest.

## 2.3. Identifiers

An identifier is composed of a letter and a sequence of zero or more letters, digits and the underscore character ( `_` ). Uppercase and lowercase letters are considered different. Identifiers can be of any length.

An identifier token appears as an ‹*id*› terminal symbol in the language grammar.

## 2.4. Keywords

The following twelve identifiers are reserved for use as keywords with special meanings and may not be used for any other purpose:

| | | |
|---|---|---|
| `break` | `elseif` | `return` |
| `dec` | `false` | `true` |
| `do` | `if` | `var` |
| `else` | `inc` | `while` |

## 2.5. Literals

At a lexical level there are four kinds of literals: booleans, integers, characters, and strings.

### 2.5.1. Booleans

A boolean literal is either the keyword `true` or the keyword `false` . These are equal to 1 and 0, respectively.

A boolean literal token appears as a ‹*lit-bool*› terminal symbol in the language grammar.

### 2.5.2. Integers

An integer literal is a sequence of one or more digits from 0 to 9. It can optionally start with a minus ( `-` ) sign. Only decimal (base 10) numbers are supported. Valid range: -2,147,483,648 to 2,147,483,647 ($-2^{31}$ to $2^{31} - 1$).

An integer literal token appears as a ‹*lit-int*› terminal symbol in the language grammar.

### 2.5.3. Characters

A character literal is a Unicode character enclosed in single quotes, as in `'x'`. The compiler translates the specified character into its corresponding Unicode integer code point.

Character literals cannot contain the quote character ( `'` ) or a newline character; in order to represent them, and certain other characters, the following escape sequences may be used:

| Name | Escape Sequence | Code Point |
|---|---|---|
| Newline | \n | 10 |
| Carriage Return | \r | 13 |
| Tab | \t | 9 |
| Backslash | \\ | 92 |
| Single Quote | \' | 39 |
| Double Quote | \" | 34 |
| Unicode Character | \u*hhhhhh* | *hhhhhh* |

The escape sequence \u*hhhhhh* consists of the backslash, followed by the lower case letter “u”, followed by six hexadecimal digits (digits “0” to “9” and the upper or lower case letters “a” to “f”), which are taken to specify the code point in base 16 of the desired character.

A character literal token appears as a ‹*lit-char*› terminal symbol in the language grammar.

### 2.5.4. Strings

A string literal is a sequence of zero or more Unicode characters delimited by double quotes, for example: `"this is a string"`. String literals cannot contain newline or double-quote characters. These and other special characters can be represented using the same escape sequences available for character literals.

A string literal is stored in memory as an array (accessible through a 32-bit handle (https://en.wikipedia.org/wiki/Handle_(computing))) containing zero or more int32 values. Each value is the code point of the character in the corresponding position of the given string. In other words, a string is stored using the UTF-32 (https://en.wikipedia.org/wiki/UTF-32) encoding.

A string literal token appears as a ‹*lit-str*› terminal symbol in the language grammar.

## 3. Syntax

The following BNF context free grammar defines the syntax of the Falak programming language. The **red** elements represent explicit terminal symbols (tokens).

‹*program*› → ‹*def-list*›

‹*def-list*› → ‹*def-list*› ‹*def*›

‹*def-list*› → ε

‹def› → ‹var-def›

‹def› → ‹fun-def›

‹var-def› → **var** ‹var-list› **;**

‹var-list› → ‹id-list›

‹id-list› → ‹id› ‹id-list-cont›

‹id-list-cont› → **,** ‹id› ‹id-list-cont›

‹id-list-cont› → ε

‹fun-def› → ‹id› **(** ‹param-list› **) {** ‹var-def-list› ‹stmt-list› **}**

‹param-list› → ‹id-list›

‹param-list› → ε

‹var-def-list› → ‹var-def-list› ‹var-def›

‹var-def-list› → ε

‹stmt-list› → ‹stmt-list› ‹stmt›

‹stmt-list› → ε

‹stmt› → ‹stmt-assign›

‹stmt› → ‹stmt-incr›

‹stmt› → ‹stmt-decr›

‹stmt› → ‹stmt-fun-call›

‹stmt› → ‹stmt-if›

‹stmt› → ‹stmt-while›

‹stmt› → ‹stmt-do-while›

‹stmt› → ‹stmt-break›

‹stmt› → ‹stmt-return›

‹stmt› → ‹stmt-empty›

‹stmt-assign› → ‹id› = ‹expr› **;**

‹stmt-incr› → **inc** ‹id› **;**

‹stmt-decr› → **dec** ‹id› **;**

‹stmt-fun-call› → ‹fun-call› **;**

‹fun-call› → ‹id› **(** ‹expr-list› **)**

‹*expr-list*› → ‹*expr*› ‹*expr-list-cont*›

‹*expr-list*› → ε

‹*expr-list-cont*› → **,** ‹*expr*› ‹*expr-list-cont*›

‹*expr-list-cont*› → ε

‹*stmt-if*› → **if (** ‹*expr*› **) {** ‹*stmt-list*› **}** ‹*else-if-list*› ‹*else*›

‹*else-if-list*› → ‹*else-if-list*› **elseif (** ‹*expr*› **) {** ‹*stmt-list*› **}**

‹*else-if-list*› → ε

‹*else*› → **else {** ‹*stmt-list*› **}**

‹*else*› → ε

‹*stmt-while*› → **while (** ‹*expr*› **) {** ‹*stmt-list*› **}**

‹*stmt-do-while*› → **do {** ‹*stmt-list*› **} while (** ‹*expr*› **) ;**

‹*stmt-break*› → **break ;**

‹*stmt-return*› → **return** ‹*expr*› **;**

‹*stmt-empty*› → **;**

‹*expr*› → ‹*expr-or*›

‹*expr-or*› → ‹*expr-or*› ‹*op-or*› ‹*expr-and*›

‹*op-or*› → ||

‹*op-or*› → ^

‹*expr-or*› → ‹*expr-and*›

‹*expr-and*› → ‹*expr-and*› **&&** ‹*expr-comp*›

‹*expr-and*› → ‹*expr-comp*›

‹*expr-comp*› → ‹*expr-comp*› ‹*op-comp*› ‹*expr-rel*›

‹*expr-comp*› → ‹*expr-rel*›

‹*op-comp*› → ==

‹*op-comp*› → **!**=

‹*expr-rel*› → ‹*expr-rel*› ‹*op-rel*› ‹*expr-add*›

‹*expr-rel*› → ‹*expr-add*›

‹*op-rel*› → <

‹*op-rel*› → <=

‹op-rel› → >

‹op-rel› → >=

‹expr-add› → ‹expr-add› ‹op-add› ‹expr-mul›

‹expr-add› → ‹expr-mul›

‹op-add› → +

‹op-add› → -

‹expr-mul› → ‹expr-mul› ‹op-mul› ‹expr-unary›

‹expr-mul› → ‹expr-unary›

‹op-mul› → *

‹op-mul› → /

‹op-mul› → %

‹expr-unary› → ‹op-unary› ‹expr-unary›

‹expr-unary› → ‹expr-primary›

‹op-unary› → +

‹op-unary› → -

‹op-unary› → !

‹expr-primary› → ‹id›

‹expr-primary› → ‹fun-call›

‹expr-primary› → ‹array›

‹expr-primary› → ‹lit›

‹expr-primary› → ( ‹expr› )

‹array› → [ ‹expr-list› ]

‹lit› → ‹lit-bool›

‹lit› → ‹lit-int›

‹lit› → ‹lit-char›

‹lit› → ‹lit-str›

# 4. Semantics

## 4.1. Compile Time Validations

1. The language only supports a 32-bit signed two's complement integer (int32) data type. This is the data type for every variable, parameter and function return value. This means that a Falak compiler doesn't need to verify type consistency.
2. Every program starts its execution in a function called `main`. It is an error if the program does not contain a function with this name.
3. Any variable defined outside a function is a global variable. The scope of a global variable is the body of all the functions in the program, even those defined before the variable itself.
4. Function names and global variables exist in different namespaces. This means that you can have a global variable with the same name as a function and vice versa.
5. It's an error to define two global variables with the same name.
6. It's an error to define two functions with the same name.
7. A function definition is visible from the body of all the functions in a program, even from itself. Thus, functions can call themselves recursively directly or indirectly.
8. In every function call the number of arguments must match the number of parameters contained in the corresponding function definition.
9. The following names are part of the initial namespace for functions and constitute Falak's API (the number after the slash symbol (/) is the arity (https://en.wikipedia.org/wiki/Arity) of the given function):
   - `printi` /1
   - `printc` /1
   - `prints` /1
   - `println` /0
   - `readi` /0
   - `reads` /0
   - `new` /1
   - `size` /1
   - `add` /2
   - `get` /2
   - `set` /3
10. Each function has its own independent namespace for its local names. This means that parameter and local variable names have to be unique inside the body of each function. It's valid to have a parameter or local variable name with the same name as a global variable. In that case the local name shadows (https://en.wikipedia.org/wiki/Variable_shadowing) the global variable.
11. It's an error to refer to a variable, parameter or function not in scope in the current namespace.
12. The `break` statement can only be used inside the body of a `while` or `do-while` statements.
13. Values of integer literals should be between `-2147483648` and `2147483647` ($-2^{31}$ and $2^{31} - 1$, respectively).

## 4.2. Runtime Behavior

1. A function returns zero by default, except if it executes an explicit `return` statement with some other value.
2. The value returned by the `main` function must be the exit code returned by the program to the operating system.
3. For the conditional and loop statements (`if`, `while`, and `do-while`) the number 0 is the only value considered *false*, everything else is considered *true*.
4. The assignment, function call, conditional, loops, break, and return statements behave like their C# counterparts. The increment statement (`inc`) adds one to the provided variable, while the decrement statement (`dec`) subtracts one from it.
5. The Falak syntax supports string and array literals. Both of these are represented in memory as arrays accessible through API managed 32-bit handles (https://en.wikipedia.org/wiki/Handle_(computing)).
6. The following are the supported operators. Precedence and associativity are established in the language grammar.

*Table 1. Arithmetic operators*

| Operator | Syntax | Description |
|---|---|---|
| Unary minus | `-x` | Produces *x* negated (with its sign changed). |
| Unary plus | `+x` | Produces *x*. |
| Multiplication | `x*y` | Produces *x* times *y*. |
| Division | `x/y` | Produces *x* divided by *y* truncating the result towards zero. An exception is thrown if *y* is zero. |
| Remainder | `x%y` | Produces the remainder of dividing *x* by *y*. An exception is thrown if *y* is zero. |
| Addition | `x+y` | Produces *x* plus *y*. |
| Subtraction | `x-y` | Produces *x* minus *y*. |

*Table 2. Logical operators*

| Operator | Syntax | Description |
|---|---|---|
| Not | `!x` | Evaluates *x* and produces 1 if its result is equal to 0. Otherwise produces 0. |
| And | `x&&y` | Produces 1 if both *x* and *y* evaluate to non-zero values. Otherwise, produces 0. The operation is short circuited, this means that if *x* evaluates to 0 then *y* is not evaluated. |
| Or | `x\|\|y` | Produces 1 if either *x* or *y* evaluate to a non-zero value. Otherwise, produces 0. The operation is short circuited, this means that if *x* evaluates to a non-zero value then *y* is not evaluated. |
| Xor | `x^y` | Produces 1 only if *x* evaluates to a zero value and *y* evaluates to a non-zero value, or vice versa. Otherwise, produces 0. |

*Table 3. Comparison and relational operators*

| Operator | Syntax | Description |
|---|---|---|
| Equal to | $x == y$ | Produces 1 if $x$ is equal to $y$, otherwise produces 0. |
| Not equal to | $x != y$ | Produces 1 if $x$ is not equal to $y$, otherwise produces 0. |
| Greater than | $x > y$ | Produces 1 if $x$ is greater than $y$, otherwise produces 0. |
| Less than | $x < y$ | Produces 1 if $x$ is less than $y$, otherwise produces 0. |
| Greater than or equal to | $x >= y$ | Produces 1 if $x$ is greater than or equal to $y$, otherwise produces 0. |
| Less than or equal to | $x <= y$ | Produces 1 if $x$ is less than or equal to $y$, otherwise produces 0. |

*Table 4. Function call*

| Operator | Syntax | Description |
|---|---|---|
| Function call | $f(arg_1, arg_2, ..., arg_n)$ | Invoke $f$ with the given arguments and return its result. All arguments are fully evaluated before the call. |

## 5. API

This section documents all the functions from the Falak application programming interface (API).

*Table 5. Input/Output Operations*

| Signature | Description |
|---|---|
| `printi(i)` | Prints `i` to stdout as a decimal integer. Does not print a new line at the end. Returns 0. |
| `printc(c)` | Prints a character to stdout, where `c` is its Unicode code point. Does not print a new line at the end. Returns 0. |
| `prints(s)` | Prints `s` to stdout as a string. `s` must be a handle to an array containing zero or more Unicode code points. Does not print a new line at the end. Returns 0. |
| `println()` | Prints a newline character to stdout. Returns 0. |
| `readi()` | Reads from stdin an optionally signed decimal integer and returns its value. Does not return until a valid integer has been read. |
| `reads()` | Reads from stdin a string (until the end of line) and returns a handle to a newly created array containing the Unicode code points of all the characters read. |

*Table 6. Array List Operations*

| Signature | Description |
|---|---|
| `new(n)` | Creates a new array object with `n` elements and returns its handle. All the elements of the array are set to zero. Throws an exception if `n` is less than zero. |

| Signature | Description |
|---|---|
| `size(h)` | Returns the size (number of elements) of the array referenced by handle `h`. Throws an exception if `h` is not a valid handle. |
| `add(h, x)` | Adds `x` at the end of the array referenced by handle `h`. Returns 0. Throws an exception if `h` is not a valid handle. |
| `get(h, i)` | Returns the value at index `i` from the array referenced by handle `h`. Throws an exception if `i` is out of bounds or if `h` is not a valid handle. |
| `set(h, i, x)` | Sets to `x` the element at index `i` of the array referenced by handle `h`. Returns 0. Throws an exception if `i` is out of bounds or if `h` is not a valid handle. |

## 6. Code Examples

| Source File | Description |
|---|---|
| 001_hello.falak | The classical "hello, world" program. |
| 002_binary.falak | Converts decimal numbers into binary. |
| 003_palindrome.falak | Determines if a string is a palindrome. |
| 004_factorial.falak | Computes factorials using iteration and recursion. |
| 005_arrays.falak | Implementation of typical array operations. |
| 006_next_day.falak | Given the date of a certain day, determines the date of the day after. |
| 007_literals.falak | Verifies that the implementation of literal values meet the specified requirements. |
| 008_vars.falak | Example using global and local variables. |
| 009_operators.falak | Verifies that the implementation of all the operators meet the specified requirements. |
| 010_breaks.falak | Verifies that the implementation of the `break` statement meets the specified requirements. |

Version 1.0.0
Last updated 2021-08-25 17:41:05 UTC