Farango Documentation

Release 0.1.0

FLIHABI

July 07, 2015

Contents

DISCLAIMER: This document is a work in progress. It's content may change at any given moment and should not be built upon.

CHAPTER 1

Language goals

- Be distributed out of the box. Concurrency and distribution should be easy to do.
  - As a result, the language would benefit from immutable data structures
  - Functional languages are best fit candidates for these kind of tasks
- Be fast. If the benefits of distributed calculations are outdone by poor optimisation, this would be useless.
- Be safe and high-level. The programmer should work on a *theoric machine*, and as such, the language should abstract away the inner work.

1.1 Lexical Structure

This chapter defines the lexical structure of the Farango programming language.

1.1.1 Characters

Programs are written using the Unicode 6.2 character set.

Line breaks

A line break is defined as follows:

```
line-break = ? ASCII LF character ?
           | ? ASCII CR character ? [? ASCII LF character ?] ;
```

An input character is defined as any unicode code point that is not a line break.

Whitespaces

A whitespace is defined as follows:

```
whitespace = line-break
           | " "                      (* space *)
           | ? ASCII HT character ? (* horizontal tab *) ;
```

Comments

Comments come in two forms: an inline form, and a block form.

- inline comments (`// text`) mark all the characters from `//` until a line breaks to be ignored by the compiler.
- block comments (`/* text */`) mark all the characters from `/*` until the matching `*/` to be ignored by the compiler. Block comments may nest.

1.1.2 Tokens

Tokens are the terminal symbols of the syntactic grammar, that are neither whitespace nor comments.

A token is either an identifier, a keyword, a literal, or a symbol:

```
token = identifier | keyword | literal | symbol ;
```

1.1.3 Identifiers

An indentifier is an unbounded sequence of word characters, the first of which must not be a digit.

```
identifier-start = ? any non-digit letter ([:alpha:]) ? ;
identifier-part = ? any alphanumeric character ([:alnum:]) ? ;

identifier = ? (identifier-start [{identifier-part}]) but not a keyword or a boolean-literal ?
```

It is a compile time error for an identifier to have the same spelling than a keyword or a boolean literal.

Two identifiers are the same if they contain the same contents; i.e. each of their characters are two-by-two equal.

1.1.4 Literals

A literal is the source code notation for representing the value of a builtin type.

Integer literals

An integer literal is the representation of an integer value of base 10 or 16.

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

hex-digit = digit | "a" | "b" | "c" | "d" | "e" | "f"
                  | "A" | "B" | "C" | "D" | "E" | "F" ;

integer-literal = decimal-literal
                | hexadecimal-literal ;

decimal-literal = {digit} ;

hexadecimal-literal = "0x" {hex-digit} ;
```

The largest integer literal is 9223372036854775808 (2^63) – this number may only appear as the operand of the unary minus operator (-). It is a compile time error if 9223372036854775808 (2^63) appear in a context other than the operand of the unary minus or if the value of an integer literal is greater than 9223372036854775808 (2^63). It is a compile time error if an hexadecimal integer literal does not fit in 64 bits.

Floating-point literals

A floating-point literal possess a whole-number part, a period, a fraction part, and an exponent:

```
number = {digit} ;

sign = "+" | "-" ;

exponent = "e" [sign] number ;

float-num = number "." [number]
          | "." number ;

float-literal = float-num [exponent] ;
```

Floating-point literals are converted to the IEEE 754 64-bit double-precision binary format.

The largest positive finite floating-point literal is 1.7976931348623157e308. The smallest positive finite non-zero floating-point literal is 4.9e-324.

It is a compile time error if the literal is too large that its IEEE 754 conversion becomes infinity, or if the literal is too small that its IEEE 754 conversion becomes 0.

Boolean literals

A boolean literal is either `true` or `false`:

```
boolean-literal = "true" | "false" ;
```

String literals

A string literal is a sequence of at least one character enclosed by double quotes:

```
character = ? all characters except for " and \ ? | escape-sequence ;

string-literal = '"' {character} '"';
```

A string literal may contain escape sequences for invisible characters that bear special meaning:

```
escape-sequence = "\b" (* backspace *)
                | "\t" (* horizontal tab *)
                | "\n" (* line feed *)
                | "\r" (* carriage return *)
                | '\"' (* double quote *)
                | "\'" (* single quote *)
                | "\\" (* backslash *)
                ;
```

It is a compile time error for a line terminator to appear after the opening double quote and before the closing matching double quote.

1.1.5 Symbols

Symbols in the grammatical context of Farango are either separators or operators:

```
symbol = separator | operator ;
```

Separators

The following tokens are used as separators:

```
separator = "(" | ")" | "{" | "}" | "[" | "]" | ";" | "," | "." ;
```

Operators

An operator is a sequence of one or more operator characters:

```
operator-char = "!" | "#" | "$" | "%" | "&" | "*" | "+" | "-" | "/"
              | ":" | "<" | "=" | ">" | "?" | "@" | "^" | "|" | "~" ;

operator = {operator-char}
```

1.2 Types

```
type = primitive-type | structure-type | union-type | empty-type ;

type-prefix = "type" identifier [generic-list] ;

generic-list-r = identifier | identifier "," generic-list-r ;
generic-list = "(" generic-list-r ")" ;
```

1.2.1 Empty types

An empty type is an abstraction on the concept of "nothing". A variable with an empty type always hold a value representing "nothing".

Value of empty types are not compatible between each other.

```
empty-type = type-prefix ;
```

1.2.2 Primitive types

The following types are known as primitive types:

- int (64-bit integer)
- float (64-bit double precision floating point number)
- bool

```
primitive-type = "int" | "float" | "bool" ;
```

1.2.3 Structure types

Structure types are records that contain one or more fields. Each field is denoted by an identifier that is unique to the record, and a type.

```
structure-type = type-prefix "=" "{" field-list "}" ;

field-list = field
           | field ";" field-list ;

field = identifier ":" type-identifier ;
```

1.2.4 Union Types

Union types denote an abstract type representing the logical union of all the specified types – a value with a type that is contained within an union can fit in a variable of this union type.

```
union-type = type-prefix "=" type-list ;

type-list = type-identifier
          | type-identifier "|" type-list ;
```

1.3 Expressions

Everything in Farango has a value – there are no statements, only expressions.

1.3.1 Binary Operators

Binary operators are operators taking two parameters. Invoking an operator can be done with two possible syntaxes:

- Operator-like: `<expr> <op> <expr>`
- Function-like: `(<op>)(<expr>, <expr>)`

The language shall natively provide the following operators:

Operator	Description
*	Multiplication
/	Division
%	Modulo
+	Addition / Union
-	Subtraction
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Equal or greater than
<=	Equal or less than
<=>	Compare
&&	Logical AND
||	Logical OR
>>	Bitwise right shift
<<	Bitwise left shift
^	Bitwise XOR
|	Bitwise OR
&	Bitwise AND

1.3.2 Unary operators

Unary operators, unlike binary operators, only take one parameter.

The language shall natively provide the following operators:

Operator	Description
!	Logical not
~	Bitwise not
-	Minus
+	Plus

1.3.3 Operator precedence

Operators in an expression have evaluation priority: this is called precedence. An operator takes precedence over an other operator if it is evaluated before the other. As an example, * takes precedence over +, because `a + b * c` can be expanded to `a + (b * c)`, and not `(a + b) * c`.

Below is a table of operators sorted from high precedence (top) to low precedence (bottom):

Operator	Precedence
Unary	`+expr -expr ~ !`
User-defined	
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >>`
Relational	`< > <= >= <=>`
Equality	`== !=`
Bitwise AND	`&`
Bitwise XOR	`^`
Bitwise OR	`|`
Logical AND	`&&`
Logical OR	`||`
Assignment	`= += -= *= /= %= &= ^= |= <<= >>=`

1.3.4 User defined operators

User may define or overload operators by declaring a function with the operator symbol enclosed in parenthesis as identifier:

```
fun (<>)(lhs, rhs) = {
    lhs != rhs
}
```

Here we declare the binary operator <> as an alias of `!=`. Alternatively, one could implement the operator as:

```
var (<>) = (!=)
```

There are no requirements on the purity of user-defined operators, but programmers should aspire to make their operators pure.

There are also no requirements on operators laws, with some exceptions on default operator overloads:

- `+`, `*`, `^`, `|`, `&`, == and `!=` shall be associative and commutative.
- All comparison operators shall be transitive.

1.4 Callables

A callable is an object that, given a set of inputs known as *parameters* produce one output known as the *return value* through a sequence of computations.

1.4.1 Declaration

A callable can be declared with the following syntax:

```
function = "fun" identifier "(" parameter-list ")" "=" expression ;

parameter-list = parameter
               | parameter "," parameter-list ;

parameter = identifier | identifier ":" type-identifier ;
```

which yields:

```
fun <identifier>(<param0>, <param1>, [...], <paramN>) = <expression>
```

where `<identifier>` is the name which the callable shall be refered to in the module, `<param0>` through `<paramN>` are the identifiers of each parameter, and `<expression>` is the expression that shall be evaluated when the callable is invoked.

1.4.2 Invokation

Callable invokation is done by specifying its identifier, then the list of its parameters enclosed in parenthesis:

```
<identifier>(<param0>, <param1>, [...], <paramN>)
```

1.4.3 Purity

A callable is called *pure* when it has no side-effects, and when given a set of parameters, two invokations does produce the same return value.

A common example of pure callables are the arithmetic operators.

1.4.4 Functions

Functions are the default (and most used) kind of callables. It has an internal context that is the parameters and local variables, and is destroyed when a return value is produced.

1.4.5 Coroutines

Coroutines are the second kind of callables, and can be inferred from the usage of the `yield` control flow keyword. They are much like functions, except that the internal context of the function is not discarded when a return value is produced. Instead, the context is saved, and the coroutine shall resume when it is invoked again later.

1.4.6 Tasks

Tasks are the final kind of callables, and they express a way to make asynchronous tasks units that can be passed on different execution environments. Tasks can be inferred from the usage of the `offer` control flow keyword, and much like functions, their execution context is destroyed after a return value has been offered.

1.4.7 Control Flow

In addition to the standard control flow statements, the following statements are provided to change the control flow in any callable:

- `return <expr>`: returns the value of the given expression in a function. Execution of the current function stops.
- `yield <expr>`: yields the value of the given expression in a coroutine. Execution resumes after this point when the coroutine is called again.
- `offer <expr>`: offers the value of the given expression in a task to the underlying task manager. Execution of the current function stops.