Concata Programming Language Specification

Table of contents

Introduction

Concata is a general-purpose stack-based concatenative functional programming language. It's garbage-collected with dynamic typing and in that aspect comparable to scripting languages like Lua or Javascript, but also provides an optional compile-time *typechecker*. Programs are composed of *modules* which get imported during compilation. Concata source files use the `.conc` extension.

The syntax and design of Concata takes inspiration from languages like Forth, Joy and Elixir, but also Nim.

Lexical Structure

Characters

Concata source code is encoded using the UTF-8 text encoding. Characters are case-sensitive, which means upper-case and lower-case letters are distinct. The byte order mark (U+FEFF) should be ignored if it is the first Unicode codepoint in the source code and can be disallowed anywhere else. The null character (U+0000) must not be allowed as a part of the source code.

The characters in concata source code are separated into 3 major categories, *word characters*, *separators* and *whitespaces*.

Word Characters

Word characters are characters that make up *tokens* like *words*, but also *numbers* and *keywords*. They include all alphanumeric characters (lower-case and upper-case ASCII letters and digits), but also the symbols listed below:

```
~ ! @ $ % ^ & * _ - = + ; : \ , < . > / ?
```

Other non-ASCII characters should not be included.

Separators

Separators are symbols that make up tokens which can also separate other tokens. They include only the symbols listed below:

```
[ ] ( ) { } | " ' ` #
```

Whitespaces

Whitespaces include the ASCII whitespace characters: space (U+0020), horizontal tab (U+0009), carriage return (U+000D), newline (U+000A), vertical tab (U+000B) and form feed (U+000C). They make up *indent* tokens and separate other tokens.

Comments

Comments are chunks of text that are ignored by the compiler. They start with the hashtag character `#` and end at the end of the line. Comments can't start inside *character* or *string literals*. See example:

```
# Hello, i am a comment.
# I am another comment!
But this is not a comment anymore
```

Tokens

Tokens are the base component of Concata code. They are made up of sequences of characters. Theres 4 basic token types: *words*, *keywords*, *literals*, *punctuation* and *indent*. Tokens that would otherwise make a single token are separated by whitespaces or separator characters, which also form tokens on their own (Except the comment character `#`). Whitespaces that do not form the indent token are ignored.

Example of two words separated by a separator character:

```
Hello|World
```

Example of separating them using a whitespace:

```
Hello World
```

Without it, they merge into a single word token:

```
HelloWorld
```

This is the case for all tokens except the ones composed of separator characters. The example below shows 2 distinct tokens formed by 2 separator characters:

```
[]
```

Words

Words are referred to as identifiers or symbols in other languages. They name *variables*, *functions* and *typegroups*. A word is made up of any word characters, but it must not start with a dollar sign (`$`), an ampersand (`&`), a digit or a dash (`-`) followed by a digit. Examples of valid words:

```
MyValidWord
%valid_word%
-this-is-completely-fine-
*all.allowed+
+&$-12345_
```

Examples of invalid words:

```
$thisIsNotAWord
52
-123
&not-a-word
&
```

Some words cannot be used like regular words. These are called keywords.

Keywords

Keywords are reserved words that are assigned a special meaning in the language and cannot be used like regular words. See their list below:

`if`	`unless`	`cond`	`match`	`else`
`for`	`fun`	`inline`	`lambda`	`->`

`::`	`with`	`defer`	`typegroup`	`when`
`end`	`compfail`	`compwarn`	`depr`	`uses`
`requires`	`=>`	`true`	`false`	`nil`
`inf`	`-inf`	`nan`	`extract`	

Literals

Literals are constants representing a value of a certain type. Two exceptions are *environment variable literals* and *argument index literals*, which serve more like syntax sugar for certain operations.

Integer Literals

Integer literals represent constants of the `Int` and `Uint` integer types. They are made up of ASCII digits corresponding to a specific base. They are signed (`Int`) and negative if prefixed by a dash (`-`), otherwise they are unsigned (`Uint`) and positive. The dash must always be at the start of the literal.

Decimal integer literals are composed of decimal ASCII digits, which are:

```
0 1 2 3 4 5 6 7 8 9
```

They can also contain underscores (`_`), but they must not start with one. If the literal is prefixed by a dash, the dash must not be followed by an underscore.

Hexadecimal integer literals begin with a special sequence `0x`, but `0X` can also be allowed. The rest of the literal is composed of hexadecimal ASCII digits representing the value, which are:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

Like hexadecimal literals, octal integer literals begin with a special sequence `0o`, with `0O` possibly also allowed. The rest of the literal is composed of octal ASCII digits, which are:

```
0 1 2 3 4 5 6 7
```

And like the previous two, binary integer literals begin with a special sequence `0b` or possibly `0B`. The rest of the literal is composed of binary ASCII digits, which are `0` and `1`.

Examples of valid integer literals:

```
123
-52
052
1_000_000
0xFF
-0x99FA
0o72
0b1011
```

Floating-point Literals

Floating-point literals represent constants of the `Float` number type. They are composed mainly of decimal ASCII digits. Just like integer literals, they are negative if prefixed by a dash, otherwise positive, and the dash must not be followed by an underscore, which is otherwise allowed in the literal.

The difference from integer literals is that they must either contain a dot (`.`) which begins the fraction part, or an exponent character (`e`) which begins the exponent part. Both must be preceded by at least one decimal ASCII digit.

The exponent part is the power of ten by which the value part without the exponent is multiplied by. The exponent character is allowed to be followed by a plus (`+`) which makes no difference, or a dash which makes it a negative exponent. A floating-point literal is allowed to contain a fraction part followed by an exponent part, but not the other way around.

There are also 3 additional keywords to represent special floating-point values. Those are: `nan` (representing the IEEE floating-point NaN value), `inf` (representing the positive infinity value) and `-inf` (representing the negative infinity value).

Examples of valid floating-point literals:

```
2.5
-9.9
5e10
500e-2
5.5e+9
1_000.5e+1_000
nan
```

Boolean Literals

Boolean literals represent constants of the `Bool` type. There are 2 boolean literal keywords: `true` and `false`.

String Literals

String literals represent constants of the 'String' type. They start and end with a double-quote mark (`"`) and contain any amount of character representations.

A character representation is any printable ASCII or UTF-8 character except backslash (`\`). For string literals, double-quote marks are also not allowed, since they are used to mark the beginning and end of the string literal. To represent those characters and non-printable ASCII characters, *escape sequences* are used. Escape sequences begin with a backslash followed by a special character sequence. See the table of escape sequences below:

Escape sequence	Unicode codepoint
`\a`	U+0007
`\b`	U+0008
`\f`	U+000C
`\v`	U+000B
`\t`	U+0009
`\r`	U+000D
`\n`	U+000A
`\e`	U+001B
`\\`	U+005C (`\`)
`\"`	U+0022 (`"`)
`\'`	U+0027 (`'`)

An additional special escape sequence is `\x`, which must be followed by 2 hexadecimal ASCII digits. It represents an arbitrary byte value.

Examples of valid string literals:

```
"Hello, world!"
"First line\nSecond line"
"'single-quotes' are completely fine."
"Look, \"double-quotes\" inside a string!"
"The UTF-8 PI symbol: \xCF\x80"
```

Character Literals

Character literals represent constants of the `Char` type. They start and end with a single-quote mark (`'`) and must contain a single character representation.

Single-quote marks are not a valid character representation in character literals and must be escaped with an escape sequence. And unlike in string literals, double-quote marks are allowed as a character representation. Examples of valid character literals:

```
'a'
'\n'
'"'
'\''
```

Environment Variable Literals

Environment variable literals serve as syntax sugar for reading environment variables. They take the form of both string literals and words, prefixed by a dollar sign (`$`). Examples of valid environment variable literals:

```
$HOME
$"MY ENV VARIABLE"
```

Argument Index Literals

Argument index literals are used in *shortcut lambdas* to refer to lambda arguments by their index. They begin with an ampersand (`&`) followed by decimal ASCII digits. Examples of valid argument index literals:

```
&1
&52
```

Nil Literal

The nil literal is used to represent the `Nil` value type. It is represented by the `nil` keyword.

Punctuation

Punctuation tokens are made up of a single separator character. Every separator character except the hashtag, which starts a comment instead, represents a punctuation token.

Indent

Indent tokens are made up of zero or more whitespace characters. They always start at the start of the line and end at the first token in the line. If the line contains no other tokens (comments are not tokens), it contains no indent token as well.