The Coral Language Specification

Version 0.1

Kateřina Nikola Lisová

April 11, 2014

Prague, Czech Republic

Contents

Lexical Syntax **5**

1.1 Identifiers *5*

1.2 Literals *7*

1.1.1 Integer Literals 7

Preface

Coral is a Ruby-like programming language which enhances advanced object-oriented programming with elements of functional programming. Every value is an object, in this sense it is a pure object-oriented language. Object blueprints are described by classes. Classes can be composed in multiple ways – classic inheritance, mixin composition, union and fusion types.

Coral is also a functional language in the sense that every function is also an object. Therefore, function definitions can be nested and higher-order functions are supported out-of-the-box. Coral also has a limited support for pattern matching, which can emulate the algebraic types used in other functional languages.

Coral has been developed from 2012 in a home environment out of pure enthusiasm for programming and out of a desire for a truly versatile language. This document is a work in progress and will stay that way forever. It acts as a reference for the language definition and some core library classes.

Chapter 1

Lexical Syntax

Coral programs are written using the Unicode character set; Unicode supplementary characters are supported as well. Coral programs are preferably encoded with the UTF-8 character encoding. While every Unicode character is supported, usage of Unicode escapes is encouraged, since fonts that IDEs might use may not support the full Unicode character set.

1.1 Identifiers

Syntax:

```
simple_id ::= lower [ id_rest ]
variable_id ::= simple_id | "_"
constant_id ::= upper [ id_rest ]
function_id ::= simple_id [ id_rest_ext ]
id_rest ::= { letter | digit | "_" }
id_rest_mid ::= id_rest [ ( "/" | "+" | "-" ) id_rest ]
id_rest_ext ::= id_rest [ id_rest_mid ] [ "?" | "!" ]
```

There are three kinds of identifiers.

First, *variable identifiers*, which are simply a lower-case letter followed by arbitrary sequence of letters (any-case), digits and underscores, or just one underscore (which has special meaning).

Second, *constant identifiers*, which are just like variable identifiers, but starting with an upper-case letter and never just an underscore.

And third, *function identifiers*, which are the most complicated ones. They can start as a variable identifier, then optionally followed by one of "`/`", "`+`" and "`-`", and then optionally ended with "`?`" or "`!`".

Coral programs are parsed greedily, so that a longest match rule applies. Letters from the syntax may be any Unicode letters, but English alphabet letters are recommended, along with English names.

1.2 Keywords

A set of identifiers is reserved for language features instead of for user identifiers. However, unlike in most other languages, keywords are not being recognized inside of paths, except for a few specific cases.

The following names are the reserved words.

```
alias
annotation
as
begin
bitfield
break
case
cast
catch
class
clone
constant
constructor
declare
def
destructor
do
else
elsif
end
ensure
enum
for
function
fusion
goto
if
implements
in
include
interface
is
let
loop
match
memoize
message
method
mixin
module
native
next
nil
no
of
opaque
operator
out
property
protocol
raise
range
record
redo
rescue
retry
return
self
skip
struct
super
template
test
then
this
throw
throws
transparent
type
undef
unless
until
union
use
var
void
yes
when
while
yield
```

Not every reserved word is a keyword in every context, this behavior will be further explained. For example, the **`bitfield`** reserved word is only recognized as a keyword inside an enumeration definition context, in a specific place. Every reserved word may be used as a function identifier, with a little work-around when used with an implicit receiver.

1.3 Literals

1.3.1 Integer Literals