June 5, 2023

The Cara Language Specification

Martin Janiczek
https://cara-lang.com

Contents

1 Introduction

Cara[1] is a general-purpose programming language aiming to combine the functional style of ML-family languages (Elm, OCaml, Haskell) with the familiar mainstream ALGOL-style syntax (Kotlin, Rust).

More specifically, Cara aims to inhabit a sweet spot in the programming language design space: pleasant to write, read and maintain while staying safe and dependable.

This is achieved by building on functional features (algebraic data types, purity, automatic full type inference) while keeping a façade of imperative procedural syntax.

Outside the scope of this specification are the reference interpreter and compiler[2], the latter of which aims to transpile Cara programs to HVM[3] programs, which can in turn be compiled to native binaries with *automatic parallelism*.

This specification document describes the detailed features, syntax, and semantics of the Cara language.

[1] https://cara-lang.com
[2] https://github.com/cara-lang/compiler
[3] https://github.com/HigherOrderCO/HVM

2 Example programs

```
quickSort(List[Int]): List[Int]
quickSort([]) = []
quickSort([x,...xs]) = {
  (lt, gt) = List.partition(#(x >= _), xs)
  quickSort(lt) ++ x ++ quickSort(gt)
}

[5,1,3,2,4]
  |> quickSort
  |> IO.println! // -> [1,2,3,4,5]
```

```
#!/usr/bin/env cara
dst = IO.ask!("Enter destination filename: ")
dstHandle = FS.open!(dst, FS.Write)

timestampFmt = "hh:mm:ss.fff"
1..10 |> IO.forEach!(\i -> IO {
  time = Time.now!()
  dstHandle |> FS.write!("[${Time.format(timestampFmt, time)}] ${i=}\n")
})
```

```
fizzbuzz(n) =
  if n % 15 == 0 then "FizzBuzz"
  else if n % 3 == 0 then "Fizz"
  else if n % 5 == 0 then "Buzz"
  else "$n"

1..20
  |> List.map(fizzbuzz)
  |> String.join(", ")
  |> IO.println!
```

```
type Maybe[a] =
  | Nothing
  | Just(a)

traverse(fn: a -> Maybe[b], list: List[a]): Maybe[List[b]]
traverse(fn,list) = go(list,[])
  where
    go([],bs) = Just(List.reverse(bs))
    go([a,...as],bs) =
      case fn(a) of
        Nothing -> Nothing
        Just(b) -> go(as,b++bs)

xs = [1,2,3,4,5]
ys = [6,7,8,9,10]
f = \n -> if n == 3 then Nothing else Just(n)
IO.println!(xs |> traverse(f)) // -> Nothing
IO.println!(ys |> traverse(f)) // -> Just([6,7,8,9,10])
```

3 In this document

Syntax

```
# This is how Ungrammar-like syntax is stylized.
# Each syntax section usually begins with this block.
# It's not meant to be precise, particularly not around whitespace and EOLs.
#
# For more on Ungrammar see:
# https://rust-analyzer.github.io/blog/2020/10/24/introducing-ungrammar.html
```

TODO: This is how TODOs are stylized.

Hopefully there are none by the time you read it!

```
// This is how Cara source code is stylized.
x = 1
IO.println!("Hello world!")
```

This is how examples are stylized.

```
IO.println!("They can contain code too!")
```

This is how links are stylized: Section 3.

This is how `inline code` is stylized.

4 Syntax

At the very top, a Cara program consists of a series of declarations:

Syntax

```
program = shebangComment? declaration*
```

The program is implicitly in an `IO` effect block (Section 4.4.14), with the exception that declarations are allowed (effect blocks normally only permit statements):

```
// myScript.cara
name = IO.ask!("What's your name? ")
x = 1 + 2 + 3
IO.println!("Hello $name! Sum of first three natural numbers is $x.")
```

4.1 Declarations

Declaration can be one of the following:

Syntax

```
declaration = sumTypeDecl
            | typeAliasDecl
            | moduleDecl
            | extendModuleDecl
            | statementDecl
            | functionDecl
            | unaryOpDecl
            | binaryOpDecl
            | valueAnnotationDecl
            | functionAnnotationDecl
```

4.1.1 Sum type

Syntax

```
# type User = Anonymous | LoggedIn(token: String, name: String)
# type Maybe[a] = Nothing | Just(a)
sumTypeDecl = sumModifier? 'type' upperName
              ('[' typeVar (',' typeVar)* ']')?
              '=' constructorList

sumModifier = 'private' | 'opaque'

# Foo | Bar
# \n | Anonymous | LoggedIn(String)
constructorList = (EOL+ '|')?
                  constructor ('|' constructor)*

# Foo
# Foo(Int)
# Foo(Int, String)
# Foo(x: Int, y: Int)
constructor = upperName
              ('(' constructorArg (',' constructorArg)* ')')?

# Int
# x: Int
constructorArg = (constructorArgName ':')? type

# x
# userName
constructorArgName = lowerName
```

A core feature of Cara are *sum types*, sometimes also called *tagged unions.*[4]

These allow representing a choice: eg. an user can be an anonymous user *OR* a logged in user *OR* a logged in administrator.

Contrast this with *product types*, which are ubiquitous in programming and allow holding multiple pieces of data at the same time: an user has a name *AND* an address.

Here is an example of defining a new sum type in Cara:

```
type User =
  | Anonymous
  | LoggedIn
  | LoggedInAdmin

currentUser = LoggedInAdmin // A value can only be one of these at any given time!
```

This is only the simplest variant of sum types, but they can also contain data:

[4] https://en.wikipedia.org/wiki/Tagged_union

```
type User =
  | Anonymous
  | LoggedIn({ token: String, permissions: List[Permission] })
  | LoggedInAdmin({ token: String })

currentUser = LoggedInAdmin({ token: "xoxb-..." })
```

Each constructor can also hold multiple pieces of data without resorting to a record or tuple:

```
type User =
  | Anonymous
  | LoggedIn(String, List[Permission]) // holding String AND List[Permission]
  | LoggedInAdmin(String)

currentUser = LoggedIn("xoxb-...", [])
```

Each defined constructor also creates a way to create values of this type:

```
Anonymous     // : User
LoggedIn      // (String, List[Permission]): User
LoggedInAdmin // (String): User

LoggedIn("xoxb-...", [])  // : User
LoggedInAdmin("xoxb-...") // : User
```

The constructor arguments can also be labeled:

```
type User =
  | Anonymous
  | LoggedIn(token: String, permissions: List[Permission])
  | LoggedInAdmin(token: String)

currentUser = LoggedIn("xoxb-...", [])
```

The labels are enforced when two values of the same type are next to each other:

```
type Expr =
  | EInt(Int)
  | EPlus(x: Expr, y: Expr)
```

If a constructor holds no arguments, it is not a function and doesn't need the function call parentheses when used. Thus, continuing the examples above:

```
user1 = Anonymous   // ✔
user2 = Anonymous() // ✘
```

Data can be extracted from the sum types via pattern matching:

```
token =
  case currentUser of
    Anonymous            -> Nothing
```

```
    LoggedIn(token, _)   -> Just(token)
    LoggedInAdmin(token) -> Just(token)
```

4.1.1.1 Alternative shorthand syntax

There is an alternative syntax for sum type declaration that puts the first type on the same line as the `=` and doesn't use the leading `|` character:

```
type NullableInt = MyNull | MyInt(Int)
```

```
type NullableInt = MyNull
                 | MyInt(Int)
```

Note that putting the `=` on a new line is not allowed:

```
type NullableInt
  = MyNull // ✘
  | MyInt(Int)
```

4.1.1.2 Generic sum types

Sum types can be parameterized over a type they will hold:

```
type List[a] =
  | Empty
  | Cons(a, List[a])
```

```
type Result[err,ok] =
  | Err(err)
  | Ok(ok)
```

```
x: List[Result[String,Int]] = [Err("boo"),Ok(42)]

type alias MyResult[ok] = Result[Int,ok]
y: MyResult[a] = Err(123)
```

4.1.1.3 TODO Private sum type

4.1.1.4 TODO Opaque sum type

4.1.2 Type alias

Syntax

```
# type alias Foo = Int
# private type alias Bar[a,b] = Result[a,b]
typeAliasDecl = 'private'? 'type' 'alias' upperName
                ('[' typeVar (',' typeVar)* ']')?
                '=' type
```

Type alias gives a new name to an existing type:

```
type alias XY = (Int, Int)
type alias C = { real: Float, imag: Float }
type alias MyResult[a] = Result[String,a]

position: XY = (1,2)
```

This is *structural*, not *nominal*, and so you can use values of the original type where values of the alias are expected and vice versa:

```
velocity: (Int,Int) = (0,-5)

step1((px,py): XY, (vx,vy): XY): XY =
  (px+vx, py+vy)

step2((px,py): (Int,Int), (vx,vy): (Int,Int)): (Int,Int) =
  (px+vx, py+vy)

step1((0,1), velocity) // -> (0,-4)
step2((0,1), velocity) // -> (0,-4)
```

4.1.2.1 TODO Private type alias

4.1.3 Module

Syntax

```
# module Foo { ... }
# private module Foo { ... }
moduleDecl = 'private'? 'module' upperName
             '{' declaration* '}'
```

A module serves as a container or *namespace* for a set of declarations:

```
module Maybe {

  type Maybe[a] = Nothing | Just(a)

  map(fn: a -> b, Nothing) = Nothing
  map(fn: a -> b, Just(a)) = Just(fn(a))

}
// outside the module, its public contents can be used via the namespace:
Maybe.Nothing // -> Nothing
Maybe.Just(1) |> Maybe.map(#(_ + 1)) // -> Just(2)
```

Modules can be nested:

```
module Foo {
  module Bar {
    x = 1
  }
}

Foo.Bar.x // -> 1
```

There can also be multiple modules next to each other; there is no one-to-one correspondence between modules and files.

TODO: Talk about opening a module (importing its members into the current namespace).

4.1.3.1 TODO Private module

4.1.4 Extend module

Syntax

```
# extend module Foo { ... }
# extend module Foo.Bar { ... }
extendModuleDecl = 'extend' 'module' upperIdentifier
                   '{' declaration* '}'
```

Defining a module second time or redefining existing definitions is disallowed, but you can add new definitions with the `extend module` declaration:

```
module Foo {
  x = 1
}

extend module Foo {
  y = x + 1
  module Bar {
    z = 0
  }
}

Foo.y // -> 2
Foo.Bar.z // -> 0
```

The `extend module` declaration can't access private definitions.

4.1.5 Statement

Syntax

```
# x = 1
# x = foo!(123)
# foo!(123)
statementDecl = statement
```

Statements like `x = 1` and `IO.println!(123)` are declarations as well. See Section 4.2 for more on Statements.

4.1.6 Function

Syntax

```
# f() = 1
# f(a,b) = a + b
# f(a,b): Int = a + b
# f(a: Int, b: Int) = a + b
# f(a: Int, b: Int): Int = a + b
# private f(a,b) = a + b
functionDecl = 'private'? lowerName
               '(' (functionArg (',' functionArg)*)? ')'
               (':' type)?
               '=' expr
```

Functions are a syntax sugar over let statements (Section 4.2.1) and case expressions (Section 4.4.23) that allows a concise equation style.

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)
```

The above is equivalent to the following case expression:

```
fib(n) =
  case n of
    0 -> 0
    1 -> 1
    _ -> fib(n-1) + fib(n-2)
```

which is itself equivalent to the following let statement:

```
fib = \n ->
  case n of
    0 -> 0
    1 -> 1
    _ -> fib(n-1) + fib(n-2)
```

TODO: Mention difference between `fn(Int,Bool): String` (function of two arguments) and `fn((Int,Bool)): String` (function of one tuple argument). Do we want to make them equivalent and interchangeable?

4.1.6.1 TODO Private function

4.1.7 Unary operator overloading

Syntax

```
# `!`(x: Int) = x + 1
# private `!`(x: Int) = x + 1
unaryOpDecl = 'private'? '`' unaryOp '`'
              '(' functionArg ')'
              (':' type)?
              '=' expr

unaryOp = '-' | '!' | '~' | '..'
```

Overloading unary operators is done via a function declaration where the function name is the operator literal surrounded by backticks:

```
type alias C = (Float, Float)
`!`((r,i): C): C = (r,-i) // complex conjugate

!(1,2) // -> (1,-2)
```

4.1.8 Binary operator overloading

Syntax

```
# `+`(x: (), y: ()) = 42
binaryOpDecl = 'private'? '`' binaryOp '`'
               '(' functionArg ',' functionArg ')'
               (':' type)?
               '=' expr

binaryOp = '+' | '-' | '*' | '/' | '%' | '**'
         | '|' | '^' | '&'
         | '||' | '&&'
         | '++'
         | '..' | '...'
         | '<=' | '<' | '==' | '!=' | '>' | '>='
         | '<<' | '>>' | '>>>'
```

Overloading binary operators is done via a function declaration where the function name is the operator literal surrounded by backticks:

```
type alias XY = (Int, Int)
`*`(k: Int, (x,y): XY): XY = (k*x, k*y)

5 * (1,2) // -> (5,10)
```

4.1.9 Value type annotation

Syntax

```
# x: Int
valueAnnotationDecl = lowerName ':' type
```

Values can be given type annotations outside their definition:

```
x : (Int, Bool)
x = (123, True)
```

This is equivalent to defining both on the same line:

```
x: (Int, Bool) = (123, True)
```

4.1.10 Function type annotation

Syntax

```
# x(Int, Bool): Int
# add(a: Int, b: Int): Int
functionAnnotationDecl = lowerName
                         '(' (functionTypeArg (',' functionTypeArg)*)? ')'
                         ':' type
```

Functions can be given type annotations outside their definition:

```
lengthEquals(Int, String): Bool
lengthEquals(length, str) =
  String.length(str) == length
```

This is equivalent to defining both on the same line:

```
lengthEquals(length: Int, str: String): Bool =
  String.length(str) == length
```

The return type and the argument types in the signatures are mandatory; labels (`x: Int`) are optional.

Note it's disallowed to have two arguments of the same type next to each other without labelling them:

```
setDiff(Set[a], Set[a]): Set[a]                  // ✘
setDiff(orig: Set[a], toRemove: Set[a]): Set[a]  // ✔
```

The labels in the type signatures only serve the programmer, they can differ from the signatures in the declarations themselves.

4.2 Statements

Syntax

```
statement = letStatement
          | letBangStatement
          | bangStatement
```

Statements exist inside block expressions (Section 4.4.13) and wherever declarations are admitted (Section 4.1.5), eg. in the top-level and in modules:

```
IO.println!("In top-level")

module Foo {
  IO.println!("In a module")
}

x1 = IO {
  IO.println!("In a block")
}

x2 = {
  y = 1
  y + 1 // can't use bangs in non-effect blocks
}
```

Statements can roughly be translated into monadic expressions:

Statement	Example	Translation (Haskell)
Let	`x = 1`	`pure 1 >>= \x -> ...`
Let-bang	`x = bang!`	`bang  >>= \x -> ...`
Bang	`bang!`	`bang  >>= \_ -> ...`

Thus a list of statements with a return expression at the end can be reduced into a single expression:

```
isTitleNonEmpty(doc) = Maybe {
  head = doc.head!
  title = head.title!
  !title.isEmpty()
}
```

is equivalent to:

```
isTitleNonEmpty(doc) =
  doc.head |> Maybe.andThen (\head ->
    head.title |> Maybe.andThen (\title ->
      Maybe.pure(!title.isEmpty())
    )
  )
```

4.2.1 Let

Syntax

```
# x = 123
# private x = 123
# (r,i) = complexNumber
# Foo(a) = myFoo
letStatement = 'private'? pattern
               (':' type)?
               '=' expr
```

A let statement executes an expression and binds its value according to the pattern on the left-hand side.

```
n = 10
string = List.repeat(n, "Hello world!")
           |> String.join(" ")
```

It is possible to use pattern-matching on the left-hand side:

```
(x,y) = position
```

It is disallowed to use the `_` wildcard pattern in let statements though (as any expression on the right-hand side is guaranteed to be effect-less if used outside a bang, and calculating it and throwing it away would not make sense).

TODO: What about debug logging? ie. the Elm pattern `let _ = Debug.log "x" (foo 123) in ...`

4.2.2 Let-bang

A let-bang statement executes a bang, along with any effects it might have, and binds its resulting value according to the left-hand side pattern.

Syntax

```
# x: Int = foo!
# private (x,y) = Foo.bar!(123)
letBangStatement = 'private'? pattern
                   (':' type)?
                   '=' bang
```

This is handy as it effectively unwraps a monadic value, similarly to Haskell's `do` notation[5], Gleam's `use` [6] and Roc's backpassing[7], saving you from nested callback lambdas.

```
destination = IO.ask!("Enter destination filename: ") // "foo.txt"
destinationHandle = FS.open!(destination, FS.Write)   // <handle>
time = Time.now!                                      // 2023-06-04 ...
// ... presumably write something to the destination
```

TODO: Should it be `Time.now!` or `Time.now!()`?

TODO: Is it possible to relax the requirements and have bangs anywhere on the right side, even multiple? Seems like it should be possible to automate this. Do we want it, or would it lead to a more confusing code?

The catch is that behind the scenes this *is equivalent* to the nested callback lambdas, and your final value can't escape the monadic context (`IO` in the above example). For more information see Section 4.4.14 on effect blocks: the top-level scope is implicitly in an IO effect block.

Note the difference between a let and let-bang:

```
input: Maybe[Int] = Just(10)
x = Maybe {
  a = input! // Int,        10
  b = input  // Maybe[Int], Just(10)
}
```

[5] https://en.wikibooks.org/wiki/Haskell/do_notation
[6] https://gleam.run/book/tour/use.html
[7] https://github.com/roc-lang/roc/blob/master/roc-for-elm-programmers.md#backpassing

4.2.3 Bang

A bang statement executes a bang, along with any effects it might have, and throws away its result.

Syntax

```
# foo!
# Foo.bar!(123)
bangStatement = bang
```

A typical example of a bang is the humble `IO.println`:

```
IO.println!(123)
```

Because it returns `IO[()]` (and thus the best a let-bang could hope to unwrap from it is `()`), there is no point in using a let-bang for it and instead a bang statement is typically used.

4.3 Bangs

There are two kinds of bangs:

Syntax

```
bang = valueBang
     | callBang
```

4.3.1 Value bang

Syntax

```
# foo!
# Bar.foo!
valueBang = expr '!'
```

A value bang allows unwrapping a monadic value:

```
maybeInt = Just(123)
maybeInt! // 123, if in effect block
```

4.3.2 Call bang

Syntax

```
# foo!(123)
# Bar.foo!("abc","def")
callBang = expr '!'
           '(' expr (',' expr)* ')'
```

A call bang is equivalent to a let statement and a value bang:

```
// Original
List.head!([1,2,3]) // Int, if in Maybe context

// Equivalent to:
x = List.head([1,2,3]) // Maybe[Int]
x!                     // Int, if in Maybe context
```

Note that since pipelines parse into function calls, it's also possible to have bangs looking like `x |> IO.println!` and `x |> Foo.bar!(1,2,3)`.

4.4 Expressions

Syntax

```
expr = intExpr
     | floatExpr
     | charExpr
     | stringExpr
     | multilineStringExpr
     | boolExpr
     | unitExpr
     | tupleExpr
     | listExpr
     | recordExpr
     | recordAccessExpr
     | recordGetterExpr
     | blockExpr
     | effectBlockExpr
     | constructorExpr
     | idExpr
     | rootIdExpr
     | callExpr
     | pipelineExpr
     | lambdaExpr
     | holeExpr
     | ifExpr
     | caseExpr
     | opExpr
```

4.4.1 TODO Integers

TODO: Mention `_` allowed in integers (`123_456_789`).

4.4.2 TODO Floats

4.4.3 TODO Characters

4.4.4 TODO Strings

TODO: Note that string interpolation is present in this type of string!

4.4.5 TODO Multi-line strings

TODO: Note that string interpolation is present in this type of string!

4.4.6 Booleans

Syntax

```
# True
# False
boolExpr = 'True' | 'False'
```

Booleans `True` and `False` represent truth values, useful for holding yes/no and on/off information. Closely connected to predicates (functions `pred(a): Bool`):

```
isEmpty(s: String): Bool =
  String.length(s) == 0

isEmpty("hello") // -> False
isEmpty("")      // -> True
```

Used in if-expressions (Section 4.4.22):

```
if isEmpty(myString) then
  doSomething()
else
  doSomethingElse()
```

Note that it's often better to create your own sum type of two values than to hold the data as a boolean:

```
type ReportLevel = Short | Verbose
level = Verbose

// rather than:
isLevelVerbose = True
```

4.4.7 Unit

Syntax

```
# ()
unitExpr = '(' ')'
```

An unit expression is the only value of the unit type (Section 4.6.1). As such it has no space to convey varying information and is often used to mean "nothing". Such as in `IO.println` returning an `IO[()]`: there's no information other than that the call succeeded.

```
echo: IO[()] = IO {
  input = IO.ask!("")
  IO.println!(input)
  () // I'm done!
}
```

Note that the above function is equivalent to one that omits the `()`, since blocks without the return value return `()` implicitly:

```
echo: IO[()] = IO {
  input = IO.ask!("")
  IO.println!(input) // <- we're NOT returning the return value of this
  // the return value is missing, `()` is used automatically
}
```

4.4.8 Tuples

Syntax

```
# (1,True,foo)
tupleExpr = '(' expr ',' expr (',' expr)* ')'
```

Tuples are *heterogenous* collections of data (meaning, the elements do not have to be of the same type):

```
myTuple = (1, "Hello", ['a','b','c'])
```

They are an example of a *product type* (contrast with *sum types*, Section 4.1.1): eg. a tuple `("Martin", 29)` holds the user name *AND* their age *at the same time*. For another example of product types see records (Section 4.4.10).

Tuples start from length 2, since:

- tuples of length 0 are just an unit expression (Section 4.4.7).
- tuples of length 1 are just the expression they contain.

TODO: Mention how Cara deals with leading/trailing commas in tuples.

TODO: Mention the implicit getters for tuples: `.el1 .. .elN`, `.first .. .tenth`. Mention them being one-indexed.

4.4.9 Lists

Syntax

```
# []
# [1]
# [1,2]
listExpr = '[' (expr (',' expr)*)? ']'
```

Lists are *homogenous* collections of data (meaning, the elements have to be of the same type):

```
list0: List[a]    = []
list1: List[Int]  = [1]
list2: List[Bool] = [True, False]

badList = [1,True] // ✘ This won't compile
```

TODO: Mention how Cara deals with leading/trailing commas in lists.

4.4.10 Records

Syntax

```
# {a:1, b:True}
# {...a, x:123, y}
recordExpr = '{' (recordExprContent (',' recordExprContent)*)? '}'

recordExprContent = recordExprField
                  | recordExprPun
                  | recordExprSpread
```

Records are collections of data, each with an associated unique *label*:

```
user = {name: "Martin", age: 29}
```

The order of fields doesn't matter: the following records are equivalent:

```
user1 = {name: "Martin", age: 29}
user2 = {age: 29, name: "Martin"}
user1 == user2 // -> True
```

TODO: Mention default comparison instance to be one with fields sorted ASC. Can we get to a world where most everything is comparable? (ie. sum types as well)

It is an error to give a record two fields of the same name:

```
{name: "Martin", name: "Janiczek"} // ✘ This won't compile
```

TODO: Mention how Cara deals with leading/trailing commas in records.

There are three types of record content:

4.4.10.1 Record field

Syntax

```
# field: 123
recordExprField = lowerName ':' expr
```

Fields are a combination of a field name and its corresponding value:

```
{x: 123, y: True} // record with two fields
```

4.4.10.2 Record pun

Syntax

```
# field
recordExprPun = lowerName
```

Puns only consist of the field name. The value is taken from the current scope:

```
x = 1
record = {x} // -> {x: 1}
```

It is an error to use a pun for a field name that cannot be found in the current scope:

```
record = {y} // ✘ This won't compile
```

4.4.10.3 Record spread

Syntax

```
# ...record
recordExprSpread = '...' lowerIdentifier
```

Spread adds fields from an existing record to the record being created.

```
velocity = {vx: 1, vy: 2}
ball = {x: 123, y: -5, ...velocity}
```

TODO: Add an end-to-end test for record creation via spread.

TODO: Mention what happens when there's a collision. Is spread somehow special in that collisions are allowed (what field wins?), or are they disallowed?

TODO: Add an end-to-end test for what happens when fields collide when using spread.

4.4.11 Record access

Syntax

```
# myRecord.field
# createPerson(123).age
recordAccessExpr = expr '.' lowerName
```

Record access is one of ways to extract a field value from a record.

```
myRecord = {name: "Martin", age: 29}
myRecord.name // -> "Martin"
myRecord.age  // -> 29
```

The expression on the left doesn't need to be a record *literal*:

```
createPerson(age: Int): {name: String, age: Int}
createPerson(age) =
  { name: "Unknown soldier"
  , age
  }

createPerson(35).age // -> 35
```

Record access is equivalent to calling a record getter (Section 4.4.12):

```
myRecord = {name: "Martin", age: 29}
myRecord.name // -> "Martin"
// is equivalent to:
myRecord |> .name
.name(myRecord)
```

4.4.12 Record getters

Syntax

```
# .recordField
recordGetterExpr = '.' lowerName
```

Record getters are a syntax shorthand for a function that extracts a field value from a record:

```
.name
// is equivalent to
(\{name} -> name)
(\record -> record.name)
```

```
people =
  [
    {name: "Martin", age: 29},
    {name: "Dumbledore", age: 115},
  ]

ages = people |> List.map(.age) // -> [29, 115]
```

4.4.13 Blocks

Syntax

```
# {
#   x = 1     // let
#   y = 1 + x // let
#   (x,y)     // expr
# }
blockExpr = '{'
            (EOL+ letStatement)+
            EOL+ expr EOL+
            '}'
```

Blocks allow the use of `let` statements (Section 4.2.1) inside expressions:

```
greeting = {
  n = 3
  string = "Hello world!"
  List.repeat(n,string)
    |> String.join(" ")
}
// -> "Hello world! Hello world! Hello world!"
```

(The consequence of that is that bangs are disallowed inside blocks. For blocks with bangs see effect blocks: Section 4.4.14.)

The last item inside a block (which must be an expression, not a `let` statement) will be returned.

TODO: Mention what happens to the bindings from inside the block expression (they get discarded after exiting the scope.

4.4.14 Effect blocks

Syntax

```
# Maybe {
#   head = doc.head!               // let-bang
#   title = head.title!            // let-bang
#   cap = String.capitalize(title) // let
#   cap != ""                      // expr
# }

# IO {
#   input = IO.ask!("") // let-bang
#   IO.println!(input)  // bang
# }
effectBlockExpr = upperIdentifier
                  '{'
                  (EOL+ statement)+
                  (EOL+ expr)?
                  EOL+
                  '}'
```

Effect blocks are like normal blocks (Section 4.4.13), but allow using bangs and let-bangs inside, as well as not specifying the return value.

When the return value is not specified, `()` is used implicitly:

```
echo: IO[()] = IO {
  input = IO.ask!("")
  IO.println!(input)
  // return expr is missing; returning ()
}
```

For more info on how statements inside an effect block translate back to a single expression, see the section on Statements: Section 4.2.

The identifier before the `{...}` specifies which monad's `pure` and `andThen` functions will be used when translating the block to an expr.

This also means that depending on how the underlying monad behaves, the latter lines might not happen (eg. if encountering a Nothing in a Maybe context):

```
convertThenNegate(string: String): Maybe[Int] = Maybe {
  int = String.toInt!(string)
  -int
}

try1 = convertThenNegate("123")   // -> Just(-123)
try2 = convertThenNegate("hello") // -> Nothing, doesn't get past the first line
```

TODO: Mention what happens to the bindings from inside the block expression (they get discarded after exiting the scope.

4.4.15 Sum type constructors

Syntax

```
# Bar
# Foo.Bar(1,2,3)
constructorExpr = upperIdentifier
                  ('(' expr (',' expr)* ')')?
```

When you define a sum type (Section 4.1.1):

- a value will be created for every constructor *without* arguments.
- a function will be created for every constructor *with* arguments.

```
type Outcome =
  | GotNothing       // 0 args, creates value    GotNothing: Outcome
  | GotError(Error)  // 1 arg,  creates function GotError(Error): Outcome
  | GotInts(Int,Int) // 2 args, creates function GotInts(Int,Int): Outcome
```

These functions work as advertised:

```
outcome: Outcome = GotInts(1,2)
```

You can extract data from the values and/or dispatch on them with `case..of` pattern matching (Section 4.5.3):

```
outcomeInt: Int =
  case outcome of
    GotNothing   -> -1
    GotError(_)  -> -2
    GotInts(a,b) -> a + b
```

TODO: Do we want some kind of `fn(x) = case x of` shorthand like OCaml and Haskell has?

4.4.16 Identifiers

Syntax

```
# foo
# Foo.bar
idExpr = lowerIdentifier
```

Identifiers allow access to values previously bound via let or let-bang statements (Section 4.2):

```
x = 123
x + 50 // -> evaluates to `123 + 50`
```

TODO: Perhaps talk about allowed characters in an identifier/name or reference some section talking about them.

TODO: Talk about the scoping rules / how we try to match the identifier in ancestors if it can't be found in the current scope.

4.4.17 Root identifiers

Syntax

```
# ::foo
# ::Foo.bar
rootIdExpr = '::' lowerIdentifier
```

Root identifiers allow access to values previously bound via let or let-bang statements (Section 4.2); they differ from identifiers (Section 4.4.16) by searching in the root scope only.

```
x = 1
module Foo {
  x = 2
  module Bar {
    x = 3
    IO.println!(x)   // 3
    IO.println!(::x) // 1
  }
}
```

TODO: Is this even needed? Is there a structure of modules in which the `id-expr` syntax is not able to access a binding in some kind of sibling?

TODO: Do we need a relative access as well? A way to get into a parent scope (`../`)? If yes, and if root identifiers are still useful, should they have the `/` syntax instead?

4.4.18 Function calls

Syntax

```
# x()
# x(1)
# x(1,2)

callExpr = expr '('
           (expr (',' expr)*)?
           ')'
```

Function calls allow running a function (replacing the call expression with the body of the function with its arguments bound to the values from the call expression):

```
myFn(x) = (x, x + 1, x + 2)

myFn(100)
// -------->
{
  x = 100
  (x, x + 1, x + 2)
}
// -------->
(100, 100 + 1, 100 + 2)
// -------->
(100, 101, 102)
```

This extends to any pattern used in function arguments:

```
myFn({a,b}, Foo(c)) = a + b + c

myFn({a: 1, b: 2, c: 3, d: 4}, Foo(5))
// -------->
{
  {a,b} = {a: 1, b: 2, c: 3, d: 4}
  Foo(c) = Foo(5)
  a + b + c
}
// -------->
1 + 2 + 5
// -------->
8
```

4.4.19 Pipelines

Syntax

```
# a |> b
# a |> fn(1,2)
pipelineExpr =  expr '|>' expr
```

Pipelines are a syntax sugar for function calls (Section 4.4.18): the left argument is piped into the *last* argument of the function on the right.

```
sub(x,y) = x - y
1 |> sub(5)
// equivalent to:
sub(5,1) // -> 4
```

```
addThree(x,y,z) = x + y + z
1 |> addThree(10,20)
// equivalent to:
addThree(10,20,1) // -> 31
```

The expression on the right doesn't need to be a function call, it can be anything that evaluates to a function (`myRecord.someFunction` , `myFunction` , `\x -> x + 1` etc.):

```
negate(x) = -x
10 |> negate
// equivalent to:
negate(10) // -> -10
```

Note that there is no currying or partial application in Elm: the arity needs to agree; all of the following would throw an error:

```
addThree(x,y,z) = x + y + z
1 |> addThree           // ✘
1 |> addThree(10)       // ✘
1 |> addThree(10,20,30) // ✘
```

4.4.20 TODO Anonymous functions (lambdas)

4.4.21 TODO Hole shorthand

4.4.22 TODO If expressions

4.4.23 TODO Case expressions

4.4.24 Operators

Syntax

```
opExpr = unaryOpExpr
       | binaryOpExpr
```

Operators are 1-argument functions (*unary*) or 2-argument functions (*binary*) that are called with special syntax rules instead of the classic `foo(1,2)` function call.

```
a = -5    // unary: numeric negation
b = 5..   // unary: infinite range
c = 1 + 2 // binary: plus
```

There is a hardcoded number of binary and unary operators in Cara. Users are free to assign them specific meaning for various type combinations, but it is disallowed to define a *new* operator literal.

4.4.24.1 Unary operators

Syntax

```
unaryOpExpr = expr '..'
            | ('-' | '!' | '~') expr
```

- Numeric negation: `-e`
- Boolean negation: `!e`
- Binary negation: `~e`
- Infinite range: `e..`

For overloading unary operators see Section 4.1.7.

4.4.24.2 Binary operators

Syntax

```
binaryOpExpr = expr binaryOp expr
```

All binary operators take a left and right expression as an argument (`e + e`).

In the following table, lower precedence value means lower priority when disambiguating expressions.

Thus `1 + 2 * 3` parses as `1 + (2 * 3)`, since `+` has precedence 12 and `*` has precedence 13.

Left-associative operators (eg. `&&`) of the same precedence do get grouped left-first:

`1 + 2 - 3` turns into `(1 + 2) - 3`.

Right-associative operators (eg. `**`) of the same precedence instead do get grouped right-first:

`2 ** 3 ** 4` turns into `2 ** (3 ** 4)`.

Precedence	Operator	Literal	Note
1	Boolean AND	`&&`	**TODO:** What about short-circuiting?
2	Boolean OR	`||`	**TODO:** What about short-circuiting?
3	Append	`++`	
4	Pipeline	`|>`	This is not an user-definable operator. Parses into a function call instead.
5	Inclusive range	`..`	
5	Exclusive range	`...`	
6	Binary OR	`|`	
7	Binary XOR	`^`	
8	Binary AND	`&`	
9	Equal	`==`	
9	Not equal	`!=`	
10	Less than or equal	`<=`	
10	Less than	`<`	
10	Greater than	`>`	
10	Greater than or equal	`>=`	
11	Left shift	`<<`	
11	Right shift	`>>`	
11	Unsigned right shift	`>>>`	
12	Plus	`+`	
12	Minus	`-`	
13	Times	`*`	
13	Division	`/`	
13	Modulo	`%`	
14	Power	`**`	Right-associative.
15	Left parenthesis	`(`	This is not an user-definable operator. Right-associative.
16	Record getter	`.`	This is not an user-definable operator.

For overloading unary operators see Section 4.1.8.

4.5 TODO Patterns

4.5.1 TODO Unit

4.5.2 TODO Variable

TODO: Make sure we use `lowerName` here, not `lowerIdentifier`.

4.5.3 TODO Constructor

4.5.4 TODO Int

4.5.5 TODO Float

4.5.6 TODO Tuple

4.5.7 TODO List

4.5.8 TODO Wildcard

4.5.9 TODO Spread

4.5.10 TODO Record spread

4.5.11 TODO Record fields

4.6 TODO Types

4.6.1 TODO Unit

4.6.2 TODO Variable

4.6.3 TODO Named type

4.6.4 TODO Type application

4.6.5 TODO Function

4.6.6 TODO Tuple

4.6.7 TODO Record

4.7 Comments

Comments are annotations in the source code that are ignored by the compiler or the interpreter. Cara has three types of comments:

Syntax

```
comment = lineComment
        | blockComment
        | shebangComment
```

4.7.1 Line comments

Syntax

```
lineComment = '//' .* EOL
```

Line comments start with `//` and anything to the right of the two slashes is ignored. A newline ends the line comment.

```
// This is a line comment
IO.println!(1) // Another example
```

4.7.2 Block comments

Syntax

```
blockComment = '/*'
               ((! '*/') | blockComment)*
               '*/'
```

Block comments start with `/*` and end with `*/`. Thus, they are handy for multi-line text.

```
/* A block comment
   spanning multiple lines
*/
```

These can happen anywhere in the source code:[8]

```
myFunction(1, /* Some description */ 2)
```

Block comments can be nested, which is ocassionally useful when temporarily commenting code:

```
/*
  /* Bar */
  foo(1,2)
*/
```

Unbalanced block comments will raise an error:

```
/*
  /* This will fail. ✘
  foo(1,2)
*/
```

[8] TODO: I'm pretty sure there is some exception to this but I'm not aware of it yet.

4.7.3 Shebangs

Syntax

```
shebangComment = '#!' .* EOL
```

Shebangs[9] are a special type of line comment that starts with `#!` and has special meaning on UNIX systems: it tells the system how to run the script when executed:

```
#!/usr/bin/env cara
IO.println!("This is a script you can run via ./myScript.cara")
```

```
$ ./myScript.cara
This is a script you can run via ./myScript.cara
```

Without the shebang line the system wouldn't know how to run the script:

```
$ ./myScript.cara
./myScript.cara: line 1: syntax error near unexpected token `"This is a script you
can run via ./myScript.cara"'
./myScript.cara: line 1: `IO.println!("This is a script you can run via ./
myScript.cara")'
```

(Note you also need to make the script executable beforehand, eg. via `chmod +x ./myScript.cara`.)

Shebang can only be present as the very first bytes of the script:

```
// Welcome in my program
#!/home/martin/.bin/cara
// This will trigger an error
```

```
  #!/home/martin/.bin/cara
// This will trigger an error as well
```

[9] https://en.wikipedia.org/wiki/Shebang_(Unix)

5 TODO Semantics

5.1 TODO Effects

5.2 TODO Type system

6 TODO sections

6.1 Record update

6.2 Standard library

6.2.1 Maybe

6.2.2 Result

6.3 GADTs

6.4 Tests

6.5 Function overloading

6.6 Type classes

6.7 Imports, exports and modules

6.8 Type annotations

6.9 Type aliases

6.10 Sum types

6.11 Pattern matching

6.12 Pattern guards

6.13 Where