March 23, 2023

Elara Language Specification

Alexander Wood

Contents

1 Introduction

Elara is a statically-typed multi-paradigm programming language targeting the JVM and based on the Hindley-Milner type system. It supports a succinct, Haskell-like syntax while preserving readability and ease of use.

Elara focuses on the purely functional paradigm, but also supports Object Oriented programming and imperative programming.

Elara's notable features include:

- Structural pattern matching with exhaustiveness checking
- Type classes for polymorphism
- Complete sound type inference with higher-kinded and higher-rank types

1.1 Code Examples

While all the following examples are syntactically correct, they may assume the existence of functions not provided in the examples in order to compile.

1.1.1 Hello World

```
let main = println "Hello World!"
```

1.1.2 Pattern Matching over Lists and Higher Order Functions

```
let map f list =
    match list with
        [] -> []
        (x :: xs) -> f x :: map xs f
let main =
    let list = [1, 2, 3, 4]
    let doubleNum i = i * 2
    println (map doubleNum list)
```

1.1.3 Custom Data Types

Elara has a very flexible type system which allows for many different forms of data types to be defined.

```
type Name = String // Simple type alias

type Animal = Cat | Dog // Simple Discriminated Union

type Person = { // Record Type
    name : Name,
    age : Int,
}

type RequestState = // Complex Discriminated Union with different constructor arities
      Connected String
    | Pending
    | Failed Error

type Option a = // Polymorphic (generic) data types
      Some a
    | None

type Fix f = Fix (f (Fix f)) // Recursive, higher-kinded data types

// Combination of multiple type features
```

```
type JSONElement =
      JSONString String
    | JSONNumber Int
    | JSONNull
    | JSONArray [JSONElement]
    | JSONObject [{ // record syntax can be used anonymously
        key : String,
        value : JSONElement
    }]
```

1.1.4 Pattern Matching on Data Types

```
type JSONElement =
      JSONString String
    | JSONNumber Int
    | JSONNull
    | JSONArray [JSONElement]
    | JSONObject [{
        key : String,
        value : JSONElement
    }]

let jsonToString elem = match elem with
    JSONNull -> "null"
    JSONString str -> str
    JSONNumber num -> toString num
    JSONArray arr -> "[" <> (String.join ", " (map jsonToString arr)) <> "]"
    JSONObject components ->
        let componentToString { key, value } =
            "\"" <> key <> "\" : " <> jsonToString value
        in "{" <> String.join ", " (map componentToString components) <> "}"
```

1.1.5 Type Classes

```
type String = [Char]

type class ToString a where
    toString : a -> String

instance ToString String where
    toString s = s

instance ToString Char where
    toString c = [c]

-- Impure function taking any s with an instance ToString s, returns Unit
def print : ToString s => s -> IO ()
let print s = println (toString s)

let main =
    print "Hello"
    print 'c'
    print 123 // Doesn't compile, no ToString instance for Int
```

1.1.6 Monad Comprehension / Notation

```
def sequenceActions : Monad m => [m a] -> m [a]
def sequenceActions list = match list with
    [] -> pure []
```

```
    (x:xs) ->
        let! x' = x
        let! xs' = sequenceActions xs
        pure (x' : xs')

def sequenceActions_ : Monad m => [m a] -> m ()
def sequenceActions_ list = match list with
    [] -> pure ()
    (x:xs) ->
        do! x
        sequenceActions_ xs
```

1.2 Syntax

1.2.1 Multi-line Environments

Some syntactic structures in Elara can create multi-line environments. Formally, this means that rather than a single expression, a semicolon-separated list of *statement*s, surrounded by braces, can be used where a multi-line environment is permitted.
Practically, this allows the imperative idea of "blocks" of code to be used, rather than having a binding be a single long expression. Note that this feature is merely syntax-sugar and does not change the purely-functional semantics of Elara. When in a multi-line environment, the syntax is extended to allow imperative statements:

- Standalone let bindings:
  `let x = 1;`
- Monadic let expressions:
  `let! x = action in x + 1;`
- Monadic let bindings:
  `let! x = action;`
- Monadic do statements:
  `do! action;`
- Monadic (applicative) return statements:
  `return! 1;`

1.2.2 Lightweight Syntax

Elara allows lightweight syntax, a feature heavily inspired by F#. This makes newlines and indentation significant, allowing the omission of many braces and other tokens. Its use is recommended in almost every case.

Note that the tokens that can be ignored when using lightweight syntax may still be written manually, making the use of lightweight syntax effectively optional. The only difference is that the lexer should insert them implicitly *if and only if they are missing* when using lightweight syntax.

1.2.2.1 Lightweight Syntax Rules by Example

The following describes the lightweight syntax rules in an informal, example-based manner.

1.2.2.1.1 Optional Semicolons

In normal syntax, semicolons are required to separate statements and must appear at the end of every declaration or statement. In lightweight mode, semicolons are optional and are inferred by the presence of a newline (`\n`) character.

Normal Syntax

```
let x = 1;
let y = 2;
let main = println (x + y);
```

Lightweight Syntax

```
    let x = 1
    let y = 2
    let main = println (x + y)
```

1.2.2.1.2 Optional Braces

In normal syntax, braces are required when beginning a multi-line environment (see Section 1.2.1) or in a few other cases (such as the body of a match expression)

In lightweight syntax, braces are optional and can be inferred by newlines and indentation.

Normal Syntax

```
let x = {
    1;
}
```

```
let y = \x -> {
    1;
}
```

```
let main = match x with {
    1 -> {
        println "it's 1";
    };
    _ -> {
        println "it's not 1";
    };
}
```

```
let test x = {
    if x then {
        1;
    } else {
        2;
    }
}
```

Lightweight Syntax

```
let x =
    1
```

```
let y = \x ->
    1
```

```
let main = match x with
    1 ->
        println "it's 1"
    _ ->
        println "it's not 1"
```

```
let test x =
    if x then
        1
    else
        2
```

1.2.2.2 Offside Rule

When using lightweight syntax, the indentation is flexible but not arbitrary. The "offside rule" is used to determine which columns code should be indented to. The offside rule marks specific columns in the source code as "offside" for a given syntactic construct. Code on proceeding lines must be indented to the same column to be considered part of the same syntactic construct. If the code is indented further and a new offside rule cannot be triggered, an error is raised. If the code is indented less, the offside rule is exited and the code is considered to be part of the parent syntactic construct

The following tokens trigger the offside rule at their respective columns:

- The first non-whitespace token after the `=` token of a `let` construct
- The first non-whitespace token after the `->` token of a lambda
- The first non-whitespace token after the `then` token of an `if` expression
- The first non-whitespace token after the `else` token of an `if` expression
- The first non-whitespace token after the `with` token of an `match` expression
- The first non-whitespace token after the `->` token of an `match` case
- The first non-whitespace token after a `{` token
- The start of a `let`, `if` or `module` token

1.2.2.2.1 The Offside Rule in Action

The following examples demonstrate the offside rule in action, noting when it is used incorrectly and errors should be raised.

```
module Main        | module keyword triggers the offside rule
let x = 1          | marks offside column
  let y = 2        | considered part of x, error!
let z = 3          | correctly indented
```

As the `module` keyword triggers the offside rule, all top-level declarations must be indented to the same column. This can produce some interesting, but correct results. All top-level declarations can be indented as long as it's consistent:

```
module Main      | module keyword triggers the offside rule
  let x = 1      | marks offside column
  let y = 2      | same column, fine
let z = 3        | bad! not considered part of the module, error!
```

```
let x =            | = triggers the offside rule
    let y = 1      | correctly indented
      let z = 2    | bad! indented too much
    y + z          | correctly indented
```

```
let main =                | = triggers the offside rule
  match [1, 2, 3] with    | with triggers the offside rule
    x :: [] ->  "one"     | marks offside column
   x :: xs -> "many"      | bad! not indented enough
     [] -> "empty list"   | bad! too indented
```

1.3 Code Structure

1.3.1 Packages

The basic unit of compilation in Elara is a *package*. A package is a collection of modules, which are typically compiled and distributed together. Packages are defined in a `elara.json` file located in the root directory of the package. This file provides metadata about the package such as name, author, version, etc.

1.3.1.1 `elara.json` structure

The `elara.json` file is a standard JSON file that must contain the following attributes at the top level:

- `name: string` - The package's name. This may only contain alphanumeric characters, and the `-` symbol. Conventionally, package names are written in `lower-snake-case`
- `version: string` - The package's version. This must be a valid semantic version string. Typically `1.0.0` is used for initial releases and should usually be the default value when generating `elara.json` files.

1.3.2 Modules

Inside packages, Elara code is organised into hierarchial *modules*. Modules are single files containing a (possibly empty) list of declarations which define namespaces for these declarations. Modules are named using the `module` keyword which must appear at the start of the file, and must be named as at least 1 `UpperCamelCase` section, separated by `.` characters. For example, the module names `Foo`, `Foo.Bar`, and `Foo.Bar.Baz` are all valid. Module names must be unique within a package, and must reflect the file structure of the package. For example, a module named `Foo.Bar` must be located at `src/Foo/Bar.elara`.

Importantly, modules are hierarchial with respect to importing. Given modules `Foo` and `Foo.Bar`, the module `Foo.Bar` is a "child" of `Foo`. This has 2 important implications:

- The module `Foo.Bar` can reference declarations in `Foo` without explicitly importing it
- Modules importing `Foo` will also import `Foo.Bar`, and any other child modules of `Foo`

The following code across multiple files demonstrates the above points:

src/Foo.elara

```
module Foo

def x : Int
let x = 1
```

src/Foo/Bar.elara

```
module Foo.Bar

def y : Int
let y = Foo.x + 1
```

src/Main.elara

```
module Main
import Foo

def main : IO ()
let main = print Foo.Bar.y
```

1.3.2.1 Imports

Importing is the action of bringing a module's declarations into scope. This is done using the `import` keyword, which must appear at the top of the file under the `module` declaration.

By default, imports are **qualified** and expose **everything**. This means that when referencing a member imported from another module, the module name must be prefixed, e.g. `Foo.Bar.y` rather than `y`

1.3.2.1.1 Qualification

Qualified imports can be made unqualified by using the `unqualified` keyword after the module name, e.g. `import Foo.Bar unqualified`.

Unqualified imports should be used sparingly as they can lead to name clashes and scope pollution. Note that even with an unqualified import, explicit qualification is still permitted.

1.3.2.1.2 Expositions

By default, all declarations in a module are exposed (brought into scope) when imported. If this is not desired, a subset of the declarations can be imported using the `exposing` keyword after the module name, e.g. `import Foo.Bar exposing (x, y)`.

This can be quite useful when combined with unqualified imports. For example, suppose a library provides a `HashMap` module whose members' names clash with the Prelude. We could write something like

```
import HashMap unqualified exposing (HashMap)
import HashMap

def testMap : HashMap String Int
let testMap = HashMap.singleton "foo" 1
```

to allow the use of the `HashMap` type name without qualification, but everything else must be qualified.

Modules may also control their exposed members in the module declaration with a very similar syntax: `module Foo exposing (x, y)`. This means that at most, the listed members can be imported. Any members not in the exposition list can be considered "private" to the module.

1.3.2.1.3 Aliasing

Finally, it can be convenient to rename a module when importing it. This can be done using the `as` keyword, e.g. `import Foo.Bar as Bar`. This allows us to refer to the module as `Bar` rather than `Foo.Bar` in the current file.

Going back to the previous `HashMap` example, we might rename the unqualified `HashMap` module to `Map` to avoid confusion:

```
import HashMap unqualified exposing (HashMap)
import HashMap as Map

def testMap : HashMap String Int
let testMap = Map.singleton "foo" 1
```