Elara Language Specification

Alexander Wood

December 2022

1 Introduction

Elara is a statically-typed multi-paradigm programming language 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 features include:

- Structural pattern matching
- A first-class effects system
- Type classes for polymorphism
- Complete sound type inference

Elara primarily targets the JVM but may also target other platforms in the future.

1.1 Code Examples

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

1.1.1 Hello World

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

1.1.2 Pattern Matching on Lists

```
let map f list =
    match list
        [] -> []
        (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 an extremely flexible type system allowing many different types of custom data types

```
type Name = String # Type alias
type Animal = Cat | Dog # Simple Discriminated Union
type Person = { # Record Type
    name : Name,
    age : Int,
}
type NetworkState = # Complex Discriminated Union with
      Connected
    | Pending
    | Failed Error

type Option a = # Generic data types
      Some a
    | None

type JSONElement = # Combination of multiple type features
      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
    JSONNull -> "null"
    JSONString str -> str
    JSONNumber num -> toString num
    JSONArray arr -> "[" ++ (join ", " (map jsonToString arr)) ++ "]"
    JSONObject components ->
        let componentToString { key, value } =
            "\"" ++ key ++ "\" : " ++ jsonToString value
        in "{" ++ 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]

def print : ToString s :> s -> () \ IO
# Impure function taking any s | ToString s, returns Unit
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 Polymorphic Effects

```
def map : (a -> b \ ef) -> [a] -> [b] \ ef
let map f list = match list with
    [] -> []
    (x:xs) -> let! y = f x in y : (map f xs)
```

1.1.7 Monad 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
```

2 Syntax

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;`

The following syntactic structures can create multiline environments. The locations where these environments are allowed are marked with the `{ ... }` symbols:

- A declaration / let binding:
  `let x = { ... }`
- The body of a let expression:
  `let x = { ... } in { ... }`
- A lambda:
  `\x -> { ... }`
- The body of either branch of an if expression:
  `if x then { ... } else { ... }`

- The body of a branch of a match expression:
  `match x with { case1 -> { ... } }`
  Note that though the body of the match expression uses curly braces to mark the start and end, it does **not** create a multi-line environment.

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.

2.2.1 Lightweight Syntax Rules by Example

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

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)
```

Optional Braces

In normal syntax, braces are required when beginning a multi-line environment (see §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 -> {
```

Lightweight Syntax

```
let x =
    1

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

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

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

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

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.

As the `module` keyword triggers the offside rule, all top-level declarations must be indented to the same column.

```
module Main       | the 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
```

This can produce some interesting, but correct results. All top-level declarations can be indented as long as it's consistent:

```
module Main     | the 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
```