Elara Language Specification

Alexander Wood

May 31, 2022

Contents

1 Introduction

The Elara programming language is a multi-paradigm, statically typed, general purpose programming language with type inference. Elara follows a functional-first design - while all major paradigms are supported, the functional paradigm is encouraged, and the language is designed around use in a functional way. This includes features like structural pattern matching, higher-order functions with compiler-enforced purity, and type classes.

Elara is platform agnostic, but its primary target is the JVM, and again, language features are designed around interoperability with the JVM

For 'first-class' code (compiled / source code written in Elara), Elara provides a fully-fledged Hindley-Milner type system with type inference. For code written in other languages, full interoperability is provided, but a more basic type inference system is used.

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 Simple Input and Output

```
let main =
    println "Input a message"
    let! message = readLine
    println "Your message reversed is: "
    println (reverse message)
```

1.1.3 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)
```

1.1.4 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 # Union type

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

type NetworkState = # Discriminated Union with Data constructors
      Connected
    | Pending
    | Failed Error

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

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
    }]
```

1.1.5 Pattern Matching on Data Types

```
type JSONElement = # Combination of multiple type features
      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) ++ "}"
```

1.1.6 Type Classes

```
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 => ()
# 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
```

2 Grammar

This section defines the grammars used for parsing code in the Elara language.

2.1 Notation

These notational structures are used to describe different grammar elements:

[pattern] Optional
pattern+ One or more repetitions
*pattern** Zero or more repetitions
pattern1 — *pattern2* Choice
(pattern) Grouping
`Text` Literal text

2.2 Lexical Program Structure