Language Design Proposal: ScalellScript

Student Name(s): Kyle Dewey
Language Name: ScalellScript

Compiler Implementation Language and Reasoning: Scala. I'm familiar with the language already, and it provides pattern matching.

Target Language: JavaScript

Language Description: has a Scala-like syntax (**https://www.scala-lang.org/**), but with a feature set that somewhat resembles Haskell (**https://www.haskell.org/**). Like Scala, it has mutable state and eager evaluation. Like Haskell, it has algebraic data types and typeclasses. The syntax used for typeclasses is based on Rust (**https://www.rust-lang.org/**). Given the high-level target, this is primarily an exploration of typechecking.

Planned Restrictions: there is no type inference, hindering practical usage. Moreover, tuples are required all over the place, which is very inconvenient. This is intentional to make the language itself simpler, at the cost of making its use more obnoxious. There are no optimizations.

Syntax:

`var` is a variable
`uname` is a user-defined type name
`cn` us a user-defined constructor name
`traitname` is a user-defined trait (typeclass) name
`typevar` is a type variable
`str` is a string
`i` is an integer

```
type ::= String | Int | Unit | Built-in types
         Self | used in a trait definition to refer to the type
                an implementation is defined for
         type => type | Higher-order function type
         (type+) | Tuple type.  Must contain at least two types
         uname[type*] | Generic user defined type.  [] required
         typevar Type variables
op ::= + | - | * | / Arithmetic operations
exp ::= var | str | i | Variables, strings, and integers are
                        expressions
        unit | Expression that creates a value of type Unit
        self | Expression that refers to the data that a trait
               implementation is for
```

```
        println(exp) | Prints something to the console
        exp op exp | Arithmetic operations
        (x: type) => e | Creates a higher-order function
        exp(exp) | Calls a higher-order function
        fn(exp)  | Calls a toplevel function
        exp.fn(exp) | Calls a function defined in a typeclass
        { stmt* exp } | Block (statements and an expression)
        (exp+) | Creates a tuple.  Must contain at least two
                 expressions
        cn[type*] | Creates a user-defined type, with given
                    generic type parameters
        e match { case* } Pattern matching
stmt ::= val x: type = exp | Immutable variable initialization
         var x: type = exp | Mutable variable initialization
         x = eep  Mutable variable assignment
case ::= pattern => exp
pattern ::= x | Introduces a new variable
            _ | Matches everything
            cn(pattern) | Matches constructor
            (pattern*)  Matches tuples
tintro ::= typevar | typevar : traitname Introduces a type
                                         variable, possibly with
                                         a constraint that it
                                         implements a typeclass
tdef ::= data un[tintro*] = cdef+ Algebraic datatype definition
cdef ::= cn(type) Constructor definition
fdef ::= def fn[tintro*](x: type): type = exp Function
                                              definition
trait ::= trait traitname { fdef* } Trait (typeclass) definition
toplevel ::= tdef | fdef | trait Toplevel definitions
program ::= toplevel* exp Expression is the entry point
```

Computation Abstraction Non-Trivial Feature: Typeclasses. Rust's OOP-like syntax is used.

Non-Trivial Feature #2: Type variables / generics.

Non-Trivial Feature #3: Full pattern matching.

Work Planned for Custom Component: Typeclass implementation. Until the custom component deadline, typeclasses will not be supported.