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.

Key Features: Typeclasses, type variables / generics, algebraic data types, pattern matching with exhaustivity checking, tuples, mutable and immutable variables, higher-order functions.

Planned Restrictions: there is no type inference, hindering practical usage. There are no optimizations.

Suggested Scoring and Justification:
- **Lexer**: 2%. Only support for reserved words, identifiers, and integers. No comments.
- **Parser**: 5%. Uses S-expressions.
- **Typechecker:** 40%. Typeclasses, higher-order functions, tuples, generics, algebraic data types, exhaustivity checking on pattern matching.
- **Code Generator:** 33%. Will **not** use higher-order functions in the translation, and will instead compile these down to objects in JavaScript which behave like closures. Typeclasses will pass around a JavaScript object that has these sorts of functions on it.

Syntax:

`var` is a variable
`algname` is an algebraic datatype name
`consname` is a constructor name
`traitname` is a 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 trait definitions like Rust, referring
to the type the typeclass is implemented on
         `(` `=>` `(` type* `)` type `)` | Higher-order function
type, params first and return type last
         `(` `tuple` type type+ `)` | Tuples
         `(` `alg` algname type* `)` | Generic algebraic type
         typevar Type variable
op ::= `+` | `-` | `*` | `/` Arithmetic operations

param ::= `(` type var `)`
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
        `(` op exp exp `)` | Arithmetic operations
        `(` `=>` `(` param* `)` exp `)` | Creates a higher-order
function
        `(` `call` exp exp* `)` | Calls a high-order function
        `(` `call` fn `(` type* `)` exp*) | Calls a toplevel
function, with given generic type parameters
        `(` `mcall` exp fn `(` type* `)` exp* `)` | Calls a
function defined in a typeclass, with given generic type
parameters
        `(` `block` stmt* exp `)` | Blocks
        `(` `tuple` exp exp+ `)` | Creates a tuple
        `(` `cons` consname `(` type* `)` exp* `)` | Creates a
user-defined type, with given generic type parameters
        `(` `match` exp case `)` Pattern matching
stmt ::= `(` `val` type var exp `)` | Immutable variable
initialization
         `(` `var` type var exp `)` | Mutable variable
initialization
         `(` `=` var exp `)` Mutable variable assignment
case ::= `(` `case` pattern exp `)`
pattern ::= x | Introduces a new variable
            `_` | Matches everything
            `(` `cons` consname pattern* `)` | Matches
constructor
            `(` `tuple` pattern pattern+ `)`  Matches tuples
tintro ::= typevar | `(` `extends` typevar traitname )`)
     Introduces a type variable, possibly with a constraint that
it implements a typeclass
algdef ::= `(` `algdef` algname `(` tintro* `)` consdef+ )`)
```

```
  Algebraic datatype definition
consdef ::= `(` `cons` consname type* `)` Constructor definition
funcdef ::= `(` `def` fn `(` tintro* `)` `(` param* `)`
            type exp `)` Function definition
trait ::= `(` `trait` traitname funcdef* `)` Trait (typeclass)
definition
toplevel ::= algdef | funcdef | trait Toplevel definitions
program ::= toplevel* exp Expression is the entry point
```