**Language Design Proposal: ScalellScript Non-Inferred**

**Student Name(s):** Kyle Dewey

**Language Name:** ScalellScript Non-Inferred

**Target Language:** JavaScript

**Language Description:** has a Scala-like syntax ([**https://www.scala-lang.org/**](https://www.scala-lang.org/)), but with a feature set that somewhat resembles Haskell ([**https://www.haskell.org/**](https://www.haskell.org/)). Like Scala, it has mutable state and eager evaluation. Like Haskell, it has algebraic data types. Given the high-level target, this is primarily an exploration of typechecking. Compared to the Inferred version, this version emphasizes a much cleaner syntax, in exchange for removing type inference.

**Key Features:** Type variables / generics, algebraic data types, pattern matching with exhaustivity checking, mutable and immutable variables, higher-order functions, syntax NOT based on S-expressions.

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

**Suggested Scoring and Justification:**

* **Lexer**: 10%. Only support for reserved words, identifiers, and integers. No comments.
* **Parser**: 20%. Doesn't use S-expressions.
* **Typechecker:** 55%. Higher-order functions, generics, algebraic data types, exhaustivity checking on pattern matching. Typechecker needs to distinguish between call-like structures.
* **Code Generator:** 15%. Mostly one-to-one with JS.

**Concrete Syntax:**

id is an identifier

var is a variable

fn is a named function name

algname is an algebraic datatype name

consname is a constructor name

typevar is a type variable

i is an integer

comma\_type\_nonempty ::= type (`,` type)\*

comma\_type ::= [comma\_type\_nonempty]

type\_instantiation ::= [`<` comma\_type\_nonempty `>` ]

type ::=

 `Int` | `Unit` | `Boolean` | **Built-in types**

 typevar | **Type variable**

 algname type\_instantiation | **Generic algebraic type**

 **Functions and parenthesized types**

`(` comma\_type `)` (`=>` type)\*

param ::= var `:` type

comma\_param ::= [param (`,` param)\*]

comma\_exp ::= [exp (`,` exp)\*]

primary\_exp ::=

 id | **Could represent var, fn, or consname. Only the**

 **typechecker can disambiguate.**

 i | `true` | `false` | **Integers and booleans**

 `unit` | **Expression that creates a value of type Unit**

`(` exp `)` | **Parenthesized expression**

`println` `(` exp `)` | **Prints something to the console**

 `(` comma\_param `)` `=>` exp | **Creates a higher-order function**

 `{` stmt\* exp `}` | **Blocks**

`match`exp `{` case+ `}` **Pattern matching**

call\_exp ::= primary\_exp (type\_instantiation `(` comma\_exp `)`)\*

mult\_exp ::= call\_exp ((`\*` | `/`) call\_exp)\*

add\_exp ::= mult\_exp ((`+` | `-`) mult\_exp)\*

less\_than\_exp ::= add\_exp [`<` add\_exp]

equals\_exp ::= less\_than\_exp [`==` less\_than\_exp]

exp ::= equals\_exp

stmt ::= `val` param `=` exp `;` | **Immutable variable**

 **initialization**

`var` param `=` exp `;`| **Mutable variable**

 **initialization**

var `=` exp `;` **Mutable variable assignment**

case ::= `case` pattern `=>` exp

comma\_pattern\_nonempty ::= pattern (`,` pattern)\*

comma\_pattern ::= [comma\_pattern\_nonempty]

pattern ::= x | **Introduces a new variable**

 **`**\_` | **Matches everything**

 **Matches constructor**

consname `(` comma\_pattern `)`

comma\_typevar ::= typevar (`,` typevar)\*

**Constructor definition**

consdef ::= consname `(` comma\_type `)`

comma\_consdef ::= consdef (`,` consdef)\*

**Algebraic datatype definition**

algdef ::= `algdef` algname [`<` comma\_typevar `>`]

 `{` comma\_consdef`}`

**Function definition**

funcdef ::= `def` fn [`<` comma\_typevar `>`] `(` comma\_param `)`

 `:` type `=` exp `;`

program ::= algdef\* funcdef\* exp **Expression is the entry point**

**Example (functions on generic lists):**

algdef List<A> {

 Cons(A, List<A>),

 Nil()

}

def map<A, B>(list: List<A>, f: (A) => B): List<B> =

 match list {

 case Cons(head, tail) => Cons<B>(f(head), map<A, B>(tail, f))

 case Nil() => Nil<B>()

 };

def length<A>(list: List<A>): Int =

 match list {

 case Cons(\_, tail) => 1 + length<A>(tail)

 case Nil() => 0

 };

map<Int, Int>(Cons<Int>(1, Nil<Int>()), (x: Int) => x + 1)