**Language Design Proposal: ScalellScript Inferred**

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

**Language Name:** ScalellScript Inferred

**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. Given the high-level target, this is primarily an exploration of typechecking. The syntax is based on S-expressions to simplify the parser, in exchange for making the typechecker more complex.

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

**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**: 10%. Uses S-expressions.
* **Typechecker:** 65%. Higher-order functions, generics, algebraic data types, exhaustivity checking on pattern matching, type inference.
* **Code Generator:** 15%. Mostly one-to-one with JS.

**Syntax:**

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

type ::= `Int` | `Unit` | `Boolean` | **Built-in types**

 **Higher-order function type; params first and return**

 **type last**

 `(` `=>` `(` type\* `)` type `)` |

 `(` `alg` algname type\* `)` | **Generic algebraic type**

 typevar **Type variable**

**Arithmetic and relational operations**

op ::= `+` | `-` | `\*` | `/` | `<` | `==`

param ::= `(` type var `)`

exp ::= var | i | **Variables and integers are expressions**

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

`true` | `false` | **Booleans**

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

 `(` op exp exp `)` | **Arithmetic operations**

 **Creates a higher-order function**

 `(` `=>` `(` param\* `)` exp `)` |

 `(` `callhof` exp exp\* `)` | **Calls a high-order function**

 **Calls a toplevel function**

`(` `call` fn exp\*)|

`(` `block` stmt\* exp `)` | **Blocks**

 **Execute a constructor**

`(` `cons`consname exp\* `)`|

`(` `match`exp case+ `)` **Pattern matching**

stmt ::= `(` `val` 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**

**Algebraic datatype definition**

algdef ::= `(` `algdef` algname `(` typevar\* `)` consdef+ )`)

consdef ::= `(` consname type\* `)` **Constructor definition**

**Function definition**

funcdef ::= `(` `def` fn `(` typevar\* `)` `(` param\* `)`

 type exp `)`

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

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

(algdef List (A)

 (cons A (alg List A))

 (nil))

(def map (A B) (((alg List A) list)

 ((=> (A) B) f))

 (alg List B)

 (match list

 (case (cons cons head tail)

 (cons cons (callhof f head) (call map tail f)))

 (case (cons nil) (cons nil))))

(def length (A) (((alg List A) list)) Int

 (match list

 (case (cons cons \_ tail)

 (+ 1 (call length tail)))

 (case (cons nil) 0)))

(call map (cons cons 1 (cons nil))

 (=> ((int x)) (+ x 1)))