Language Design Proposal: LavaScript

Student Name(s): Nver Khachoyan
Language Name: LavaScript

Target Language: JavaScript

Language Description: Object-oriented programming. The goal is to learn how compiler design works, and understand quirks of the JavaScript language more deeply.

Key Features: Objects + methods with class-based inheritance, subtyping, checking if a variable is initialized before use, checking if void is used as a value, checking that a function returning non-void always returns, non-S-expression-based syntax.

Planned Restrictions: No optimizations.

Suggested Scoring and Justification:

- **Lexer**: 10%. Only support for reserved words, identifiers, and integers. No comments.
- **Parser**: 20%. Does not use S-expressions.
- **Typechecker:** 40%. Handles subtyping and method overloading, checking if a variable is initialized before use, checking if void is used as a value, checking that a function returning non-void always returns.
- **Code Generator:** 30%. Needs to work with JavaScript's prototype-based inheritance, which isn't quite one-to-one, but still pretty close.

Concrete Syntax:

`var` is a variable
`classname` is the name of a class
`methodname` is the name of a method
`str` is a string
`i` is an integer

```
type ::= `Int` | `Boolean` | `Void` | Built-in types
         classname class type; includes Object and String

comma_exp ::= [exp (`,` exp)*]

primary_exp ::=
  var | str | i | Variables, strings, and integers are
                  expressions
```

```
  `(` exp `)` | Parenthesized expressions
  `this` | Refers to my instance
  `true` | `false` | Booleans
  `println` `(` exp `)` | Prints something to the terminal
  `new` classname `(` comma_exp `)` Creates a new object

call_exp ::= primary_exp (`.` methodname `(` comma_exp `)`)*

mult_exp ::= call_exp ((`*` | `/`) call_exp)*

add_exp ::= mult_exp ((`+` | `-`) mult_exp)*

exp ::= add_exp

vardec ::= type var

stmt ::= exp `;` | Expression statements
         vardec `;` | Variable declaration
         var `=` exp `;` | Assignment
         `while` `(` exp `)` stmt | while loops
         `break` `;` | break
         `return` [exp] `;` | return, possibly void
         if with optional else
         `if` `(` exp `)` stmt [`else` stmt] |
         `{` stmt* `}` Block

comma_vardec ::= [vardec (`,` vardec)*]

methoddef ::= `method` methodname `(` comma_vardec `)` type
              `{` stmt* `}`

constructor ::= `init` `(` comma_vardec `)` `{`
                [`super` `(` comma_exp `)` `;` ]
                stmt*
                `}`
classdef ::= `class` classname [`extends` classname] `{`
             (vardec `;`)*
             constructor
             methoddef*
             `}`

program ::= classdef* stmt+  stmt+ is the entry point
```

Example (animals with a speak method):

```
class Animal {
```

```
  init() {}
  method speak() Void { return println(0); }
}
class Cat extends Animal {
  init() { super(); }
  method speak() Void { return println(1); }
}
class Dog extends Animal {
  init() { super(); }
  method speak() Void { return println(2); }
}

Animal cat;
Animal dog;
cat = new Cat();
dog = new Dog();
cat.speak();
dog.speak();
```