Design Document

Haskell Calculator

for

CS 381

### **by**

### **Drake Ryan Evans**

### **Inez Hernandez Fuerte**

### **Lea Kathryn Messenger**

### **Zijing Huang**

###### **Oregon State University**

### **Introduction**

Scoop++, a simple Haskell calculator. It will include all the standard operations addition, subtraction, multiplication, division, and powers of n. Scoop++ will have some type systems in that the language will prevent users from using negative or zero values when using the powers of n or division functions respectively. After some considerations the programming paradigm that Scoop++ will be designed in is the functional paradigm. The stack-based paradigm would have also been a good option, but the functional paradigm also provided similar benefits and a bit more flexibility.

### **Design**

For Scoop++ the features will include the addition of two integer values and the sum of those two integer values will produce an output of an integer value. To the screen, the user will see a string dictating what they inputted and what the result is. A similar process will occur for subtraction and multiplication. The division and power of n features will require a few more specifications which are detailed further in the document.

*Features:*

Because our language intends to replicate the functions of a simple calculator it will have the add, subtract, multiply, and divide functions. These functions will satisfy the basic data types and operations requirement by using integers for the operations and boolean values to handle some of the error checking. We can represent integers in both literal values and operations through the nature of the tasks. Boolean literals and operations will be addressed when doing most of our error checking. This is discussed in more detail under the *Safety Properties* section of this document.

Considering the conditionals requirement for the project, Scoop++ will have to do some error checking when doing arithmetic operations. Meaning that we will rely heavily on conditionals like if-then-else statements to ensure that a user is not attempting to do division by zero or inputting operations that could lead to mathematical errors. Because users do not need to concern themselves with the backend of a language our calculator will have to handle errors effectively so that the user does not get exposed to a series of unhelpful Haskell error messages.

When it comes to recursion and loops, Scoop++ will take advantage of explicit looping constructs when doing arithmetic operations on the powers of x. When it comes to thinking about the implementation of this function, the most straightforward implementation would be a type of while loop. Specifically, our language will breakdown into y \* y while x, this is the most intuitive style solution. We will also have to consider error states that can occur when it comes to doing powers of n which are discussed in the *Safety Properties* section of this document.

When using the Scoop++ language, users will be working with immutable objects that should not be modified. When engaging with Scoop++ the language has set features and expected inputs and outputs, for that reason Scoop++ does not have a need for mutable objects. Introducing mutable objects into our language would only serve to reduce our libraries’ error prevention and usability, adding unnecessary complexity. Lastly, we will be doing our functions with arguments through call by need as it takes the best aspects and allows for the most flexibility.

*Additional Features:*

Scoop++ will give the user some visibility of the status of the function by returning the input values and the output values in a string format, ‘The result of x \_ y is z’. Scoop++ will simply output the string and integer values to the screen, it does not do any implicit conversion between string and integer types.

The static type system works the best with our Scoop++ program to ensure that the user does not try to input an argument of the wrong type and that there are no errors caused by a user diving or power of n in the negative values.

*Feature Levels:*

Given the nature of the Scoop++ language, the feature levels for each individual feature in our language will be the same. Thinking about the different levels of each feature provided by the Scoop++ language, the best fit would be the library level. Syntactic sugar was the first to be ruled out. The main reason for this is that syntactic sugar is syntax created within a language to make a complex language easier to read and understand. Considering the Scoop++ language, the complexity is minimal, there would be no need to introduce syntactic sugar for this particular language. Core could describe our language to some extent, but core implies a much more interconnected system. Library best first out language since our features are those that users can easily engage with. There is the minimal backend that the user needs to know about and it simply helps the user complete a task.

*Safety Properties:*

As stated in the *Conditionals* section, some arithmetic operations can lead to issues. Thinking about the features of Scoop++ we can create this language to be type-safe since most errors are easily recognizable and happen through the user interface. Thus, Scoop++ does static type checking as previously stated to ensure that a user does not input integer values that would lead to unexpected outputs. Mainly because of the nature of the errors we would expect a user to create, these can all be caught beforehand. For Scoop++, division by zero and entering a number of negative power can cause unexpected errors. The addition, subtraction, and multiplication features should not encounter any errors when working with integer values. The only other time that an error would occur is if the user inputs something other than an integer value, which can be caught using the default all else case for inputs or when doing implicit conversion between string types and integer values if we wanted to store the final output to a variable, but in the case of Scoop++ that will not be an issue.

For the division by zero error, we know that it is a possibility and can easily account for that by doing some simple error checking. We will be using a conditional statement that returns a boolean value as that is the most intuitive implementation while still maintaining the project requirements. If the user inputs a denominator then Scoop++ will recognize that invalid input and prevent a crash and or unpredictable evaluations. Other than that, the division operator should not return any other errors.

Now, powers of n arithmetic operations can have a few unexpected behaviors. Scoop++ will restrict values to be positive integers. This will prevent our calculator from going into negative values. Restrictions will be enforced in a similar manner as that of the division operation, through the use of conditional boolean statements.

### **Implementation**

The semantic domains we chose for our language includes...