Design Document

Blake Le, Cameron Kocher, and Robert Yelas

1. **Introduction**

    - What is the name of your language?
    - What is the language's paradigm?
    - (Optional) Is there anything especially unique/interesting about your language that you want to highlight up front?

Our language is called **does this sound bad**, also known as **dtsb**. It uses an imperative paradigm, and is intended to be simple and easy to use, with uncomplicated syntax. We built it using the language defined in imp.hs as a foundation, and added extra features including strings, tuples, lists, and functions.

2. **Design**

    - What *features* does your language include? Be clear about how you satisfied the constraints of the feature menu.

Our language will include the following features:

Strings and operations

- We satisfy the constraints for the strings and operations feature by implementing a way to create strings as well as allow for concatenation of strings. The contents of the string can consist of any characters surrounded by " ". Some examples of valid strings are "{2149r*", "Hello", "123", "False". Declaration of a string is done by x = string "hello123". Concatenation of a string is done by utilizing the plus function. Plus takes two strings as arguments and provides a new string with the two strings concatenated to each other. An example of usage: x = plus "string1" "string2".

Tuples and operations

- We satisfy the constraints for the tuples and operations feature by implementing a way to create tuples of elements consisting of bools, ints, or tuples. This is satisfied through the tuple_gen function. An example of a tuple would be (2,(true,"hello")). The first and second elements of a tuple may be accessed by the tuple_first function and the tuple_second function.

List/array data type and operations

- We satisfy the constraints for the list/array data type as well as the operations by allowing for creation and list comprehension of several data types including, ints, bools, and strings. We satisfy the requirements by allowing for list creation, indexing in the list, list concatenation, as well as looping over a list to determine if an element is inside the list.
- What *level* does each feature fall under (core, sugar, or library), and how did you determine this?
    - What are the *safety properties* of your language? If you implemented a static type system, describe it here. Otherwise, describe what kinds of errors can occur in your language and how you handle them.

When completed, our language will include the basic building blocks of a programming language: standard data types, declarable and mutable variables, if-else conditionals, while and for loops, and functions that are able to accept arguments. We'll also implement strings, tuples, and lists, as well as built-in methods that operate on these data types. Of the aforementioned planned features, we have already implemented the following: standard data types, if-else conditionals, tuples, basic list operations, strings, and string manipulation functions.

Core Features: data types, variables, if-else conditionals, while loops, functions, arrays

We chose these properties as our core features because we believe that these are the most basic features that a programming language should have. Data types and variables are the building blocks, and loops, functions, arrays, and conditionals all seem to be present in almost every language. If we removed data types or variables, then the rest of the language could not be used and would lose all functionality

Syntactic Sugar: tuples

Tuples are syntactic sugar because they are not as necessary as data types or variables in programming languages. They are nice to have, but not explicitly necessary.

Library: strings, string manipulation functions

Strings and string manipulation are in the library level because they add more functionality to our language.

We have not yet implemented any safety properties and are relying solely on Haskell's error-handling for now. We currently have no plans to implement a static type system to allow for error checking prior to running.

3. **Implementation**

- What *semantic domains* did you choose for your language? How did you decide on these?
- Are there any unique/interesting aspects of your implementation you'd like to describe?

For our semantic domains, we decided that Env Val -> Env Val would suffice. We decided this because we needed our semantic domain to capture the scope of all of the data types and functionality we use in our programming language. Statements in our programming language are within the Env Val -> Env Val spectrum. By doing so we are able to capture our usage of Vals. A unique aspect of our implementation is our goal to keep the language simple with less barriers between the programmer and the language.