CS 381 Final Project Design

Faaiq Waqar, Jonathan Alexander, Julian Fortune

Introduction:

Name of Our Language:

Barely Functional-C

Language Paradigm:

We decided to use an **Imperative** paradigm for our programming language. This paradigm aligns with our goal of evoking the same thinking that we've used when programming in other classic imperative languages, like C, but our language is designed to be easier to reason about (i.e., no side effects from functions, no pointers) and explicit (i.e., static types, explicit casting).

Language Motivation:

Our language is designed to be friendly for people new to programming, but useful for advanced programmers.

Design:

Features of Our Language:

Basic data types and operations: There will be Integer, Boolean, and Floating point data types. Integer and Floats each individually support the 4 basic math arithmetic operations (*, /, +, -). The Boolean will have `and`, `or`, and `not` operators. There will be explicit casting between all three types.

Conditionals: There will be If statements. There will not be any other branching features, because if statements reinforce the key concepts of making decisions, and else if/else/switch statements are less precise and less explicit.

Recursion/loops: There will be looping structures in the program: *While* loops, *for* loops, and *for each* loops (iterate through the items in a list).

Variables/local names: Our language will allow binding and referencing variables. Scoping is based on functions.

Procedures/functions with arguments: There will be function structures in the program that take in parameters fed by basis of execution in the function program itself, which also support return values. The functions will be bound to names, and can be referenced by name.

List/array data type and operations : There will be lists based on the built in data types (Integer, Boolean, Float). The lists will support indexing, length/count, append, insert, and remove.

Static type system : Programmers writing code in our programming language will be unable to `compile` a program with type errors, because a static type system will be called to check the correctness of the program. This system will return error indicators, ideally in a specific Expression as defined in the syntax. This will give users an idea of the correctness of their programs. The function is called 'compile' and takes in a 'prog' or string of commands

Feature Levels and Determination:

Basic data types and operations: All data types are core. We need these to be core to form the bedrock of the language. Casting between the types will be a core feature. Most operators will be core (e.g. equal), because they are at the atomic level, but some will be syntactic sugar (e.g. greater than or equal to), because some operators can be composed from others. The `and` and `not` will be core, but the `or` will be syntactic sugar (using DeMorgan's law).

Conditionals: The If statements will be core features, because they manipulate the control flow in a unique way.

Recursion/loops: The loops and recursion will be core features. We will implement *While* and Recursion in core, since these cannot be abstracted further. For loops will use the *While* framework to be established as syntactic sugar, as they operate similarly with

the addition of the iterator included. *For each* will build upon the *for* loop, and be included in syntactic sugar.

Variables/local names: Variable names and their scope can't be sugar coated—these must be implemented in the core of the language.

Procedures/functions with arguments: Function declaration and calling will be a core feature, because they do not operate using similar logic to other core features, and have parameter passing which is a unique feature. Functions invocation is limited to the inclosing function scope in which they are declared (or program scope in the global case).

List/array data type and operations : The list will be implemented in Core, using the Haskell lists data type. The indexing lookups will be core as well. The rest of the list features (remove, insert, and pop) will be at the library level, because we can use indexing and *for each* loops.

Static type system : Core, this system is built on the basis of the language itself, and the verification that it operates correctly according to its own syntactic domain, and thus will be entirely core as we have no other interface to evaluate this.

Safety Properties of the Language:

We are currently working on using a static type system via. a compiler to recognize where errors may be in specific expressions. Take an example of the data type arithmetic that we perform in our language. With the definition of the variables used, our syntactic domain could technically allow you to add a boolean value to an integer value without the safety feature that is the compiler. The compiler made up specific parsers for different data types allows us to recognize where program errors may be, and stop us from using the program incorrectly.

Some other errors that we work on recognizing are errors in parameter passing, this is recognizing the names of variables that are placed in the string array in the function parameters and matching them with program expressions to evaluate the use of variables. This will also apply to our conditional based processes such as loops and if statements.

Implementation:

Language Semantic Domains & Decisions:

The **semantic domain** of the language is from *Store* -> *Store*, where a *Store* is (1) a mapping from variable names to values, (2) a mapping of function names to function data, (3) a stack of commands to execute.

Interesting Implementation Facts:

All programs written in our language have a return, and the default is False.