Alex Molotkov

Ashley Vieira

Connor Wilson

Zeyad Shureih

Design Document

Introduction

The name of our language is Crypt0. We chose this name because we aim for our language to have password-locked executables and source code encryption. Crypt0 is an imperative language that was heavily inspired by C. The language is imperative because it includes top-level definitions to define variables/procedures, statements which make up the function definitions, and expressions to evaluate values.

Design

Core Features:

The core feature level of the language represents the integral parts of our language that we describe in our abstract syntax. We currently have four basic data types implemented: Doubles, Ints, Strings, and Bools. Alongside this, we have implemented the declaration and referencing of variables. We have also implemented core expressions that operate on variables of compatible types: Addition, Subtraction, Multiplication, and Division. We have also defined If statements on Boolean values.

Syntactic Sugar:

Both while and for loops are syntactic sugar, with both being specialized versions of functions. We plan to use explicit looping constructs. We will be using functions with pass by value for reusing repeated code and altering the state. Functions are syntactic sugar, as they are not necessary to run a program in the language, unlike in C where a main function is a requirement. For convenience, we have defined s0 as the empty state. The additional features we will be including in our language are string operations (such as concatenation) and a static-type system.

Safety Properties

The static-type system is the main safety property of our language. The program will check for type errors before it is executed. This is to ensure that there are no type errors caused by applying an operation, function, or procedure to an argument of the wrong type. We also have various error checking cases scattered around the language to check for undefined behavior that goes beyond the type system (functions with no return type, etc).

Implementation

Our semantic domain is State to State, with a state defined as all declared variables. In other words, a program in our language takes in a number of variables (sometimes none), runs operations on those variables, and then returns all globally declared variables. Functionally, our state is actually implemented using Haskell Maps. The state Map takes a name and a variable, storing them as a key-value relationship.

On compilation of our programs from concrete to abstract syntax, the abstract syntax will be scrambled and encrypted - requiring a password to run. The encryption will work in a manner much similar to a one time pad, where a key defines how offset each character in the source code will be. In order to avoid a simple decryption as a result of similar beginnings to program, and the fact that Haskell does not play well with pseudo random number generators, we are applying a trigonometric function based off of position in the program's source code to further encrypt the program. When executing the program, the key must be passed along with the executable otherwise the source code will not be decrypted.