CS 381 Project Milestone
Michael Andrews 932-774-260
Daniel Green 932-637-313
Nicholas Matsumoto 932-709-078
Quan Nguyen 933-953-856

Introduction

Our language, current titled “Unẗiẗled Group Projecẗ” (abbreviated “T̈T̈T̈” and pronounced “French”,) is a pass-by-value imperative language.

The basic logical unit of our language is the Expression: we do not distinguish between top-level definitions, statements and expressions. Rather, everything is composed of Expressions that can be composed arbitrarily, and which will always evaluate to some end result. Expressions may also, naturally, alter state. We approximate top level definitions by allowing the user to bind a function literal to a variable name at any time. (Via, of course, an Expression.)

Most, but not all, Expressions take additional Expressions as arguments: for example, Add takes two additional Expressions. When an Expression has Expression arguments, the arguments are evaluated (recursively if necessary) until some final value is obtained. Evaluation is always performed left-to-right, and changes to the program state are passed along.

To give an idea of what this means, there is an expression which simply evaluates to a literal value, `Val` and another expression which simply dereferences a variable name and evaluates to the literal (if any) bound to it, `Var`. It is perfectly valid syntax to simply use one of these as a statement unto itself, and indeed, that is how functions (and programs, which are functions themselves) return a value: the function itself evaluates to the value evaluated to by its final expression.

Consider the following pseudocode, in which `x` and `y` are initially undefined:

```
y = (x = 3) + x++
```

In our implemented semantics, this would be written as:

```
Assign "y" (Add (Assign "x" (Var (I 3)) (increment "x"))
```

In T̈T̈T̈, this is perfectly valid. Assign both binds a literal to a name and evaluates to the bound value; increment (which is syntactic sugar) both modifies the bound value associated with the name and evaluates to the result. The overall expression would therefore evaluate to 7, and afterwards the name `y` would bound to the value 7, while `x` would be bound to 4.

By taking this approach, nearly everything can be composed arbitrarily. Note that this flexibility does come at a cost, though: because nearly everything in the grammar is composed of the same Haskell type, it is extremely easy to pass an invalid argument to almost any expression: Add expects, ultimately, to get either two integers or two strings: nothing stops the user from writing a program that attempts to add two functions, though this does of course result in an error.

There is also the question of scope. 'State' in our language is a collection (a hashmap, in fact) of names bound to literal values. Since functions are first-class values in our language, the state is the repository of all bound functions. A function has in its definition a list of names (parameters;) when a function is called, a list of expressions is provided (arguments,) and these are bound pairwise to the parameter names in the function definition to form a new state, which is passed into the function call. Variables which are themselves functions are also passed in, but variables of any other type are stripped out. When a function returns, any functions which were bound to variables inside are passed back to the call site's state, but all other variables are discarded. In this way, function variables are global, but all other kinds of variable are local.

This approach makes our scoping system dynamic from a technical standpoint, inasmuch as there is no closure system. From a practical standpoint, however, because we do not allow global variables other than functions (and function bindings themselves are immutable,) this makes very little difference: the only free variables visible from inside a function's scope are immutable. The approach is dynamic, but in practice, scope is so restricted as to make the distinction relatively unimportant.

We have tried to implement as core features only things which we feel are needed to allow the language to be sufficiently expressive, and which we have been unable to find a way to implement as sugar on existing features. (While loops are the exception.)

Design

Basic Data Types

Our basic data types are Integers, Strings, Functions, Lists, and Booleans. Booleans are implemented as syntactic sugar, but all others are Core features. Our language also has a concept of Nil and Error values, but these exist only as evaluated values; they do not exist as literals.

Conditionals

If-Else-Then is implemented as a Core feature.

Loops

Both recursion and While loops are available, and are implemented as core features. We plan to add for-loops as sugar. While loops represent our only major break from orthogonality; while we recognize that they're not strictly necessary given that we have recursion, we want to have them.

Variables

Mutable variables exist. Function variables are global, all other variables are local. Any basic data type which can be expressed as a literal (Integer, String, List, Function) can be bound to a name. This is core.

Procedures/Functions with Arguments

Functions exist, can be defined within a program, and can take arguments. A function's scope consists initially of whatever arguments it was given and any function variables bound in the calling scope. This is core, exept that 'defining' a function is syntactic sugar overlaid on the core variable assignment expression.

Strings (1 pt)

Strings are a basic data type. Operations on them are limited, but they can be concatenated, multiplied by an integer (which self-concatenates the requested number of times) or divided by an integer (which truncates the string in the proportion requested.) This is core.

Lists (2 pts)

Lists are available. They can be indexed into, prepended to, appended to, and arbitrary indices overwritten. It is possible to loop over an index, and we intend to add a map function as a library tool.

List are, of course, intrinsically invertible; anything you put into a list you can extract again at will.

Lists, and their basic operations, are implemented as core.

First-Class Functions (2 pts)

Functions are a basic data type; they can be passed into functions, stored in lists, and bound to variables. Trying to do arithmetic on a function will generate an error, however.

Additional Note

Our language has a number of features not listed here, like boolean operations for And, Or, and Not, equality evaluation, etc. We also plan to continue implementing additional features as sugar and library features going forward, so the above is not a comprehensive feature list, just what the milestone guidelines ask us to list.

Safety

T̈T̈T̈ is a dynamically-typed language. Static typing is infeasible due to our List operations, and the design decisions we made regarding how we wanted to handle certain operations. Specifically, the issue is Nil. In our milestone feedback, the question of why Nil exists was raised, and the answer is simply: so that an out-of-bounds read from a list can return something which is not an error. There is no tombstone value we could return in that context: the idea of an empty list was floated, but because our lists are not type-specific, an empty list is in fact a valid possible return from any list.

The existence of Nil, and the general unpredictability of List operations, make a static type system infeasible. For example, it is not enough to know that the Index operation was called, and that the operands are an integer and a list: without knowing both the size of the list, its contents, and the *exact* integer, we cannot know what the return will be. This makes the static typing approaches covered in the class materials impractical as long as the language is run in an interpreted fashion.

Next, we come to the question of error handling. We have revamped it since the milestone to use a more idiomatic style. We have a recursively-defined `Error` type, which has the following definition:

```
data Error = E ErrType [Error]
```

The `ErrType` datatype has a constructor for every category of error we have defined: by using a recursive data type for Errors themselves, we can thereby create, in Haskell, a representation of an Error (and any sub-errors that contributed to it) that captures the nature of the problem. And, of course, we can print it for the user when the program is run.

We now halt execution on the first line that contains an error: due to composability, that may result in any number of nested errors, which can be inspected in Haskell when using `eval` directly, and which are printed as indentation-nested strings for clarity when using `run`.

Implementation

Our language has two semantic domains, one for Expressions and one for Programs. The semantic domain for Expressions is:

```
Context -> Expression -> (Context, Result)
```

Context, in this context is a strict hashmap that contains the state; the collection of Name/Value bindings known to the current Expression. (Due to scoping, this may not be everything known to every part of the program.)

Result has the following definition:

```
data Result = Valid Value | Error | Nil
```

(See the Safety section for a brief discussion of Nil; the short version is that exists to support our desired behaviors for certain list operations.)

This domain was chosen because our program has a mutable state; each individual expression must necessarily receive a copy of that state, and return a (possibly modified) state for the next expression to use.

The semantic domain for programs is:

```
Context -> Value -> IO ()
```

This is akin to the domain for statement evaluation, except that we cannot take an expression (because no program is yet running to evaluate it) so we must take a raw function literal. The second argument (the Value) is the function literal to be run; if that seems odd, recall that functions are a basic data type in our language. (Attempting to run a non-function literal, is, of course, an error.) A context is passed in, containing the library, or whatever other definitions the user wishes to pass in. This forms the initial state for the program. When the program is finished, we take the `(Context, Result)` returned by evaluation of its final statement and print only the user-relevant portions of the `Result`.