CS381 Project Milestone

Teammate:

Mingming Su

Zhidong Zhang

Chia-Yu Tang

Yihong Liu

1. **Introduction**
    - **What is the name of your language?**

      Calculator Haskell Program

    - **What is the language's paradigm?**

      Imperative language view computation

2. **Design**
    - **What *features* does your language include? Be clear about how you satisfied the constraints of the feature menu.**
        i. **Basic data types and operations**

           We include some expression operators that appear in general calculators such as addition, subtraction, multiplication and division. To make it more special, we added optional expressions like less than, greater than, equal than, boolean negation and setup variable for users.

```
-- | Variables.
type Var = String

-- | Abstract syntax of expressions.

data Expr = Lit Int        -- literal integer
          | Add Expr Expr  -- integer addition
          | Sub Expr Expr  -- integer sub
          | Mul Expr Expr  -- integer multiply
          | Div Expr Expr  -- integer division
          | LTE Expr Expr  -- less than
          | Gt Expr Expr   -- great than
          | Equ Expr Expr  -- equal
          | Not Expr       -- boolean negation
          | Ref Var        -- variable reference
          | Fun Var Expr    -- anonymous function w/ one argument
          | App Expr Expr    -- function application
  deriving (Eq,Show)


-- | Abstract syntax of statements.

data Stmt = Set Var Expr
          | If Expr Stmt Stmt
          | While Expr Stmt
          | For Expr Expr Stmt
          | Block [Stmt]
  deriving (Eq,Show)

-- | Abstract syntax of types.

data Type = TInt
            | TBool
            | TF Var Expr
  deriving (Eq,Show)

-- | Abstract syntax of declarations.

type Decl = (Var,Type)

-- | Abstract syntax of programs.

data Prog = P [Decl] Stmt
  deriving (Eq,Show)
```

ii. **Conditionals**

If-Else statement – if it is true on the expression, run the first statement, otherwise, run the second statement.

iii. **Recursion/loops**:

While statement and For loop.

- For the while statement, the expression should be something that will return a boolean value such as less than, greater than, equal than.

  For example,

```
While (Equ (Ref "n") (Lit 100))
(Block [
 Set "sum" (Gt (Ref "n") (Lit 50))
 ])
```

- For the For Loop, to expressions should be integers. Execute the statement from Expr1 to Expr2.
  For example, from 0 to 2, increase 1 in “n” and add “n” to “sum”.

```
ex4 :: Prog
ex4 = P [("sum",TInt),("n",TInt)]
      (Block [
        Set "sum" (Lit 0),
        Set "n" (Lit 100),
        For (Lit 0) (Lit 3) (Block [
          Set "n" (Add (Ref "n") (Lit 1)),
          Set "sum" (Add (Ref "sum") (Ref "n"))
        ])
      ])
```

iv. **Variables/local names**: We use where statements to define local variables.

  The user can define their own variables in expression by “Ref” and giving an integer with “Lit”.

v. **Procedures/functions with arguments: In Imp.hs file**

  We are implementing a call-by-value interpreter. “Prime” is a function that can do multiplication and addition.

vi. **Static type system(2)**:

  We have a type checker to make sure that the type of value on the stack is correct and safe to be passed into the statement or argument. This can avoid type errors.

vii. **Strings and operations(1): incomplete**

  Define “++” concatenation and “reverse” statement and correspond semantics for them. Users are able to use “++” to combine to separate

strings, such as ++ “hello” “world” to “hello world”, or reverse “helloworld” to “ dlrowolleh”.

viii. **List/array data type and operations(2):**

The user can input a list of expressions or statements in our data type. And finally the output will be a list of variables and their values, which can be integer or bool.

- **What *level* does each feature fall under (core, sugar, or library), and how did you determine this?**
  i. **Basic data types and operations**

  Core, basis data is the feature which is crucial to the expressiveness and safety properties in our program.

  ii. **Conditionals**

  Core, if-else-then also is the kind of feature which can not instead.

  iii. **Recursion/loops**

  Syntactic sugar. If the function of for loop was removed, it could be rewritten in another way using different features.

  iv. **Variables/local names**

  Syntactic sugar. This function can be instead of another way.

  v. **Procedures/functions with arguments**

  Syntactic sugar., If we remove this feature,there are a lot of ways to replace it.

  vi. **Static type system(2)**:

  Core, type system is a checker. It should belong to the core feature.

  vii. **Strings and operations(1):**

  Syntactic sugar, we build this function by ourselves. So it can be replaced. It could be rewritten in another way using different features.

  viii. **List/array data type and operations(2):**

Library-level, The output type of ‘int’ and ‘bool’ both belong to standard libraries.

- **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.**

  We have a type checker to make sure that the values before passing are correct. This can avoid type errors happening in statements.

3. **Implementation**
   - **What *semantic domains* did you choose for your language? How did you decide on these?**

     Prog -> Maybe (Env Val)

     The Prog includes an initialized environment and statements. The environment will be declared by the user with a list of variables that need to be used and the corresponding type. We want the output to be printed with a list of variables and its values. Values can be bool or int. “Nothing” will be printed out if there is type error.

   - **Are there any unique/interesting aspects of your implementation you’d like to describe?**

     We spent some time figuring out how to use For loop in the Haskell language. Especially for how it can be implemented in semantics function. In the beginning, we used the “While” statement to run the For loop as a sugar, but it resulted in an infinite loop. We realize that this is actually a complicated way to achieve the goal. So, we decided just call the for loop recursively.