CS 421 (Summer 2017) 4th Hour Project

An Interpreter for Dynamic Delimited Continuations

Eric Kutschera (erick2@illinois.edu)

July 30, 2017

Overview

Motivation

This project is an implementation of an interpreter based on *"A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations"*[1]. That paper derives various properties about the constructs **control** and **prompt**, but this project is based only on the abstract machine presented in Figure 2 of that paper.

The abstract machine describes the evaluation of the untyped lambda calculus (variables: x, abstraction: $\lambda x.e$, application: $e_0 e_1$) extended with **control** ($\mathcal{F}x.e$) and **prompt** ($\#e$). The new constructs allow for the current continuation to be captured and used as a variable. The exact behavior is best understood by looking at the abstract machine or the test cases of this project, but the core idea is that **control** captures any function application operations that are in progress up until the nearest enclosing **prompt**. The captured operations can be applied as a variable any number of times inside the **control** expression.

In order to make testing and demonstrating the behavior of the interpreter easier, integer constants, arithmetic operators $(+, -, *, /)$, and comparison operators $(<, >, <=, >=, ==, /=)$are included in the language.

Goals

- Understand how to evaluate the untyped λ-calculus
- Work with non-trivial λ-calculus programs
- Understand an implementation of dynamic delimited continuations
- Understand what dynamic delimited continuations are good for

[1]Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. 2015. A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations. ACM Trans. Program. Lang. Syst. 38, 1, Article 2 (October 2015), 25 pages. DOI: http://dx.doi.org.proxy2.library.illinois.edu/10.1145/2794078

Broad Accomplishments

- Translated a programming language abstract machine definition into Haskell
- Extended the λ-calculus with integer arithmetic and comparison operators
- Implemented β-reduction and α-renaming for the core λ-calculus
- Utilized the Church encoding for booleans and λ-calculus recursion to create complex test cases

Implementation

Major Capabilities of Code

The code is able to evaluate abstract syntax trees for the λ-calculus extended with **control** ($\mathcal{F}x.e$) and **prompt** ($\#e$) as well as integer arithmetic and comparison operators. This can result in an integer in the simplest case, a closure if the AST evaluates to a λ-abstraction, or a continuation captured by a $\mathcal{F}$ expression.

Components of Code

- Definitions of the data types used to represent the AST
- A pretty-printer for the AST
- Definitions of the data types used to represent the objects manipulated by the abstract machine (This includes the types of possible output values)
- Functions for lifting Haskell operations on integers so that they work on the AST
- Functions which implement α-renaming and β-reduction
- The evaluator for the AST based on the abstract machine presented in the motivating paper

Status of project

All features outlined in the proposal given for this project have been implemented. In addition, integer comparison operators have been implemented which were not mentioned in the proposal.

- What works well
  - Simplification of λ-calculus expressions
  - Evaluation of the extended λ-calculus according to the rules of the abstract machine

  - Evaluation of integer operations

- What works partially
  - α-renaming works correctly for variables, abstractions, applications, and integer operations. α-renaming is not attempted for expressions which contain **control** ($\mathcal{F}x.e$) or **prompt** ($\#e$). No example scenarios have been thought through, and at this point the lack of α-renaming may be either a feature or a bug.

- Potential improvements
  - A parser and REPL could be added to improve usability
  - **control** ($\mathcal{F}$) cannot interrupt integer operators in the current implementation. This is because integer operations are evaluated directly without creating continuations to be processed by the abstract machine. This could be modified to allow for capturing. In the current version this limitation can be overcome by specifying the order of integer operations manually with λ-abstractions.

Tests

The test runner is defined in `test/Spec.hs` and it executes various groupings of individual test cases. A description of each grouping follows:

- evalBasic: Evaluation of minimal expressions
- lambdaApplication: Simplification of λ-calculus expressions. This is based on the examples from the PDF used in lecture.
- freeVars: A unit test for identifying the set of free variables. This is needed for α-renaming.
- alphaRenaming: Simplification of λ-calculus expressions which require α-renaming. This is based on the examples from the PDF used in lecture.
- integerArithmetic: Evaluation of integer arithmetic operations for cases when operands can and cannot be evaluated
- integerFunctions: Evaluation of integer functions implemented as abstractions which are applied to integer expressions
- booleanFunctions: Evaluation of expressions which apply the Church encoding for booleans. The booleans are the results of integer comparison operations.
- factorialCalls: The factorial function is implemented as an AST and applied to various integer expressions. This is intended to be a comprehensive test of everything except $\mathcal{F}$ and $\#$.

- basicPrompt: Evaluation of expressions which have # but not $\mathcal{F}$
- basicControl: Evaluation of expressions which have $\mathcal{F}$ but not #
- basicControlAndPrompt: Evaluation of relatively simple expressions which involve $\mathcal{F}$ and which behave differently when # is added in certain places
- controlAndPrompt: Evaluation of complex expressions involving control and prompt including:
  - An expression which returns a closure which has a built-up continuation trail with multiple captured continuations in the environment
  - An expression with nested $\mathcal{F}$ expressions delimited by #
  - Expressions which use a definition of factorial implemented with $\mathcal{F}$ and #

Listing

The full source code for `src/Main.hs`, `test/Spec.hs`, and `test/Tests.hs` is presented below. Other project files are omitted since they serve only to build and execute the files listed here.

src/Main.hs

```
module Main where

import Data.Set (Set)
import qualified Data.Set as Set

import Text.Printf (printf)

data Expr = VarExp String
          | LambdaExp String Expr
          | AppExp Expr Expr
          | IntExp Int
          | IntOpExp IntOp Expr Expr
          | PromptExp Expr
          | ControlExp String Expr
          | ExnExp String
  deriving (Show, Eq)

data IntOp = Plus | Minus | Times | Divide
           | Less | Greater | LessEq | GreaterEq | Equal | NotEq
  deriving Eq

instance Show IntOp where
  show Plus = "+"
  show Minus = "-"
```

```haskell
  show Times = "*"
  show Divide = "/"
  show Less = "<"
  show Greater = ">"
  show LessEq = "<="
  show GreaterEq = ">="
  show Equal = "=="
  show NotEq = "/="

ppExpr :: Expr -> String
ppExpr (VarExp s) = s
ppExpr (LambdaExp p b) = printf "(\\%s.%s)" p $ ppExpr b
ppExpr (AppExp a b) = printf "(%s)(%s)" (ppExpr a) (ppExpr b)
ppExpr (IntExp i) = show i
ppExpr (IntOpExp op lExp rExp) =
  printf "(%s %s %s)" (ppExpr lExp) (show op) (ppExpr rExp)
ppExpr (PromptExp e) = printf "#(%s)" (ppExpr e)
ppExpr (ControlExp p b) = printf "F%s.%s" p (ppExpr b)
ppExpr exn@(ExnExp _) = show exn

data Val = CapturedCont Continuation ContTrail
         | Closure String Expr Env
         | IntVal Int
         | ExnVal String
  deriving (Show, Eq)

-- An Environment maps variable names to values
newtype Env = Env [(String, Val)]
  deriving (Show, Eq)

emptyEnv :: Env
emptyEnv = Env []

data Continuation = End
                  | Arg Expr Env Continuation
                  | Fun Val Continuation
  deriving (Show, Eq)

-- ContTrail holds the continuations which have been queued as a
-- result of the application of a captured continuation. Control
-- allows continuations to be used as variables and applied before the
-- "current" continuation. This can happen many times so the
-- "then-current" continuations are stored in a list.
newtype ContTrail = ContTrail [Continuation]
  deriving (Show, Eq)

emptyKTrail :: ContTrail
emptyKTrail = ContTrail []

-- MetaConts holds (Continuation, ContTrail) pairs which have been
```

```haskell
-- delimited by Prompt expressions. Prompt moves the current
-- Continuation and ContTrail into the MetaConts. This can happen many
-- times resulting in a list of "contexts"
newtype MetaConts = MetaConts [(Continuation, ContTrail)]
  deriving (Show, Eq)

emptyMetaKs :: MetaConts
emptyMetaKs = MetaConts []

-- Perform binary operations over ints
performBinIntOp :: IntOp -> Int -> Int -> Expr
performBinIntOp Plus = performArithOp (+)
performBinIntOp Minus = performArithOp (-)
performBinIntOp Times = performArithOp (*)
performBinIntOp Divide = performArithOp div
performBinIntOp Less = performCompOp (<)
performBinIntOp Greater = performCompOp (>)
performBinIntOp LessEq = performCompOp (<=)
performBinIntOp GreaterEq = performCompOp (>=)
performBinIntOp Equal = performCompOp (==)
performBinIntOp NotEq = performCompOp (/=)

performArithOp :: (Int -> Int -> Int) -> Int -> Int -> Expr
performArithOp f a b = IntExp $ f a b

-- Conditionals are implemented using the Church encoding for booleans
performCompOp :: (Int -> Int -> Bool) -> Int -> Int -> Expr
performCompOp f a b
  | f a b = churchTrue
  | otherwise = churchFalse
  where churchTrue = LambdaExp "x" (LambdaExp "y" (VarExp "x"))
        churchFalse = LambdaExp "x" (LambdaExp "y" (VarExp "y"))

-- Determine the set of free variables in an Expr
freeVariablesExp :: Expr -> Set String
freeVariablesExp (VarExp s) = Set.singleton s
freeVariablesExp (LambdaExp p b) = Set.delete p $ freeVariablesExp b
freeVariablesExp (AppExp e1 e2) =
  Set.unions [freeVariablesExp e1, freeVariablesExp e2]
freeVariablesExp (IntExp _) = Set.empty
freeVariablesExp (IntOpExp _ lExp rExp) =
  Set.unions [freeVariablesExp lExp, freeVariablesExp rExp]
freeVariablesExp (PromptExp e) = freeVariablesExp e
freeVariablesExp (ControlExp p b) = Set.delete p $ freeVariablesExp b
freeVariablesExp (ExnExp _) = Set.empty

-- Add prime suffixes to a variable name until it does not occur in the
-- provided set of variable names
makeUnique :: String -> Set String -> String
makeUnique initial others
```

```haskell
  | initial `Set.member` others = makeUnique (initial ++ "'") others
  | otherwise = initial

-- If the expression being substituted contains a free variable that
-- gets a new binding from a lambda then change that lambda binding to
-- a fresh variable that does not occur among the free variables in
-- the lambda body.
alphaRename :: String -> Expr -> Expr -> Expr
alphaRename name e1 e2 =
  case e2 of
    (LambdaExp p b) ->
      let (newP, newB) = maybeReplace p b
      in LambdaExp newP newB
    (ControlExp p b) ->
      let (newP, newB) = maybeReplace p b
      in ControlExp newP newB
    _ -> ExnExp $ printf
         "a LambdaExp or ControlExp is required: alphaRename(%s, %s, %s)"
         name (show e1) (show e2)
  where maybeReplace p b =
          let freeExp = freeVariablesExp e1
              freeBody = freeVariablesExp b
              fresh = makeUnique p $ Set.insert p freeBody
              withFresh = replaceVarWithExpr p (VarExp fresh) b
          in if p `Set.member` freeExp
             then (fresh, replaceVarWithExpr name e1 withFresh)
             else (p, replaceVarWithExpr name e1 b)

-- Substitute an expression for all occurrences of the variable
-- name. This represents replacing the parameter name of a lambda
-- abstraction with the supplied argument.
replaceVarWithExpr :: String -> Expr -> Expr -> Expr
replaceVarWithExpr name expr ve@(VarExp v)
  | name == v = expr
  | otherwise = ve
replaceVarWithExpr name expr le@(LambdaExp p _)
  | name == p = le
  | otherwise = alphaRename name expr le
replaceVarWithExpr name expr (AppExp e1 e2) = AppExp r1 r2
  where r1 = replaceVarWithExpr name expr e1
        r2 = replaceVarWithExpr name expr e2
replaceVarWithExpr _ _ ie@(IntExp _) = ie
replaceVarWithExpr name expr (IntOpExp op lExp rExp) = IntOpExp op lR rR
  where lR = replaceVarWithExpr name expr lExp
        rR = replaceVarWithExpr name expr rExp
replaceVarWithExpr name expr (PromptExp e) =
  PromptExp $ replaceVarWithExpr name expr e
replaceVarWithExpr name expr ce@(ControlExp p _)
  | name == p = ce
  | otherwise = alphaRename name expr ce
```

```haskell
replaceVarWithExpr _ _ exn@(ExnExp _) = exn

-- Evaluate an expression in a fresh evaluator
evaluate :: Expr -> Val
evaluate e = evalSimplified e emptyEnv End emptyKTrail emptyMetaKs

-- Simplify the expression before evaluating
evalSimplified :: Expr -> Env -> Continuation -> ContTrail -> MetaConts
↪  -> Val
evalSimplified = eval . simplify

-- Attempt to simplify an expression by evaluating sub-expressions.
-- This is helpful since the regular evaluator will not evaluate the
-- bodies of un-applied lambdas and those bodies can be reduced.
simplify :: Expr -> Expr
simplify e = if firstPass == secondPass then firstPass else keepGoing
  where firstPass = simplifyOnePass e
        secondPass = simplifyOnePass firstPass
        keepGoing = simplify secondPass

-- Go down the expression tree once, reducing if possible. Do not
-- simplify applications which have control or prompt since those
-- expressions depend on the structure of the expression.
simplifyOnePass :: Expr -> Expr
simplifyOnePass ve@(VarExp _) = ve
simplifyOnePass (LambdaExp param body) = LambdaExp param $
↪  simplifyOnePass body
simplifyOnePass ae@(AppExp _ _)
  | hasControlOrPrompt ae = ae
  where hasControlOrPrompt (VarExp _) = False
        hasControlOrPrompt (LambdaExp _ e) = hasControlOrPrompt e
        hasControlOrPrompt (AppExp a b) = any hasControlOrPrompt [a, b]
        hasControlOrPrompt (IntExp _) = False
        hasControlOrPrompt (IntOpExp _ a b) = any hasControlOrPrompt [a,
         ↪  b]
        hasControlOrPrompt (PromptExp _) = True
        hasControlOrPrompt (ControlExp _ _) = True
        hasControlOrPrompt (ExnExp _) = False
simplifyOnePass (AppExp lExp rExp) =
  case simpL of
    LambdaExp p b -> replaceVarWithExpr p simpR b
    _ -> AppExp simpL simpR
  where simpL = simplifyOnePass lExp
        simpR = simplifyOnePass rExp
simplifyOnePass ie@(IntExp _) = ie
simplifyOnePass (IntOpExp op lExp rExp) =
  case (simpL, simpR) of
    (IntExp l, IntExp r) -> performBinIntOp op l r
    _ -> IntOpExp op simpL simpR
  where simpL = simplifyOnePass lExp
```

```
        simpR = simplifyOnePass rExp
simplifyOnePass (PromptExp e) = PromptExp $ simplifyOnePass e
simplifyOnePass (ControlExp param body) = ControlExp param $
 ↪  simplifyOnePass body
simplifyOnePass ee@(ExnExp _) = ee

-- Evaluate the expression as far as possible given the state of the
-- evaluator. These functions are intended to mirror the operations of
-- the adjusted call-by-value abstract machine shown in Figure 2 of
-- the paper.
-- eval         maps to eval
-- applyCont    maps to cont1
-- applyTrail   maps to trail1
-- applyMetaCont maps to cont2
eval :: Expr -> Env -> Continuation -> ContTrail -> MetaConts -> Val
eval (VarExp v) (Env env) k kTrail metaKs = case lookup v env of
  Just x -> applyCont k x kTrail metaKs
  Nothing -> ExnVal $ "var lookup failed: " ++ v
eval (LambdaExp p b) env k kTrail metaKs =
  applyCont k (Closure p b env) kTrail metaKs
eval (AppExp e1 e2) env k kTrail metaKs =
  eval e1 env (Arg e2 env k) kTrail metaKs
eval (IntExp i) _ k kTrail metaKs = applyCont k (IntVal i) kTrail metaKs
eval (IntOpExp op lExp rExp) env k kTrail metaKs =
  case (lVal, rVal) of
    (IntVal lInt, IntVal rInt) ->
      eval (performBinIntOp op lInt rInt) env k kTrail metaKs
    _ -> ExnVal $ "IntOpExp with non IntVal operand: " ++ (show (op,
    ↪  lVal, rVal))
  where lVal = eval lExp env End emptyKTrail emptyMetaKs
        rVal = eval rExp env End emptyKTrail emptyMetaKs
eval (PromptExp e) env k kTrail (MetaConts metaKs) =
  eval e env End emptyKTrail (MetaConts ((k, kTrail) : metaKs))
eval (ControlExp p b) (Env env) k kTrail metaKs =
  eval b newEnv End emptyKTrail metaKs
  where newEnv = Env $ (p, CapturedCont k kTrail) : env
eval (ExnExp exn) _ _ _ _ = ExnVal exn

-- Apply the current continuation when evaluation  has reached a value
applyCont :: Continuation -> Val -> ContTrail -> MetaConts -> Val
applyCont End v kTrail metaKs = applyTrail kTrail v metaKs
applyCont (Arg argExp env k) v kTrail metaKs =
  eval argExp env (Fun v k) kTrail metaKs
applyCont (Fun (CapturedCont cK (ContTrail cT)) k) v (ContTrail t) metaKs
 ↪  =
  applyCont cK v newTrail metaKs
  where newTrail = ContTrail $ cT ++ (k : t)
applyCont (Fun (Closure paramName body (Env closureEnv)) k) v kTrail
 ↪  metaKs =
  let updatedEnv = Env $ (paramName, v) : closureEnv
```

```haskell
  in eval body updatedEnv k kTrail metaKs
applyCont (Fun (IntVal i) _) _ _ _ =
  ExnVal $ "Cannot apply an Int: " ++ (show i) ++ " to an argument"
applyCont (Fun exn@(ExnVal _) _) _ _ _ = exn

-- Apply the correct continuation from the trail of continuations when
-- the current context is completed
applyTrail :: ContTrail -> Val -> MetaConts -> Val
applyTrail (ContTrail []) v metaKs = applyMetaCont metaKs v
applyTrail (ContTrail (c:cs)) v metaKs = applyCont c v (ContTrail cs)
 ↪  metaKs

-- Apply the correct meta-context when both the current continuation
-- is completed and the trail of then-current continuations is empty
applyMetaCont :: MetaConts -> Val -> Val
applyMetaCont (MetaConts []) v = v
applyMetaCont (MetaConts ((c, ct):mcs)) v = applyCont c v ct (MetaConts
 ↪  mcs)

main :: IO ()
main = putStrLn "Main not implemented"
```

test/Spec.hs

```haskell
module Spec where

import Tests (allTests)

main :: IO ()
main = do putStrLn ""
          putStrLn "Running Tests"
          putStrLn "============="
          mapM_ (putStrLn . showTR) allTests
          putStrLn ""

type TestResult = (String, [Bool])

showTR :: TestResult -> String
showTR (name, results) = name ++ ": " ++ correct ++ "/" ++ total ++ extra
  where correct = show $ length $ filter id results
        total = show $ length results
        extra = if total /= correct then ": " ++ (show results) else ""
```

test/Tests.hs

```haskell
module Tests where

import Main (
              Expr(..)
```

```
            , IntOp(..)
            , Val(..)
            , Env(..)
            , Continuation(..)
            , ContTrail(..)
            , evaluate
            , freeVariablesExp
            , emptyEnv
            , emptyKTrail
            , ppExpr
            )

import Data.Set (Set)
import qualified Data.Set as Set

allTests :: [(String, [Bool])]
allTests = [
             ("evalBasic", evalBasic)
           , ("lambdaApplication", lambdaApplication)
           , ("freeVars", freeVars)
           , ("alphaRenaming", alphaRenaming)
           , ("integerArithmetic", integerArithmetic)
           , ("integerFunctions", integerFunctions)
           , ("booleanFunctions", booleanFunctions)
           , ("factorialCalls", factorialCalls)
           , ("basicPrompt", basicPrompt)
           , ("basicControl", basicControl)
           , ("basicControlAndPrompt", basicControlAndPrompt)
           , ("controlAndPrompt", controlAndPrompt)
           ]

evalBasic :: [Bool]
evalBasic =
  [
    -- x
    -- == Exception var lookup failed: x
    (evaluate (VarExp ("x")))
     == (ExnVal "var lookup failed: x")
    -- \x.x
    -- == \x.x
  , (evaluate (LambdaExp "x" (VarExp "x")))
     == (Closure "x" (VarExp "x") emptyEnv)
    -- Exception s
    -- == Exception s
  , (evaluate (ExnExp "exception"))
     == (ExnVal "exception")
  ]

-- From lambda-calculus.pdf from lecture 2017-06-15
lambdaApplication :: [Bool]
```

```haskell
lambdaApplication =
  [
    -- \y.((\x.x)y)
    -- == \y.y
    (evaluate (LambdaExp "y" (AppExp (LambdaExp "x" (VarExp "x")) (VarExp "y"))))
     == (Closure "y" (VarExp "y") emptyEnv)
    -- \x.\y.((\z.x)y)
    -- ==  \x.\y.x
  , (evaluate (LambdaExp "x" (LambdaExp "y" (AppExp (LambdaExp "z" (VarExp "x")) (VarExp "y")))))
     == (Closure "x" (LambdaExp "y" (VarExp "x")) emptyEnv)
    -- \a.\b.\y.((\z.azbz)y)
    -- == \a.\b.\y.(ayby)
  , (evaluate (LambdaExp "a" (LambdaExp "b" (LambdaExp "y" (AppExp (LambdaExp "z" (AppExp (AppExp (AppExp (VarExp "a") (VarExp "z")) (VarExp "b")) (VarExp "z"))) (VarExp "y"))))))
     == (Closure "a" (LambdaExp "b" (LambdaExp "y" (AppExp (AppExp (AppExp (VarExp "a") (VarExp "y")) (VarExp "b")) (VarExp "y")))) emptyEnv)
    -- \y.((\x.(\z.z)x)y)
    -- == \y.y
  , (evaluate (LambdaExp "y" (AppExp (LambdaExp "x" (AppExp (LambdaExp "z" (VarExp "z")) (VarExp "x"))) (VarExp "y"))))
     == (Closure "y" (VarExp "y") emptyEnv)
    -- \a.\b.\y.((\x.x(\z.ax)(\x.bx))y)
    -- == \a.\b.\y.(y(\z.ay)(\x.bx))
  , (evaluate (LambdaExp "a" (LambdaExp "b" (LambdaExp "y" (AppExp (LambdaExp "x" (AppExp (AppExp (VarExp "x") (LambdaExp "z" (AppExp (VarExp "a") (VarExp "x")))) (LambdaExp "x" (AppExp (VarExp "b") (VarExp "x"))))) (VarExp "y"))))))
     == (Closure "a" (LambdaExp "b" (LambdaExp "y" (AppExp (AppExp (VarExp "y") (LambdaExp "z" (AppExp (VarExp "a") (VarExp "y")))) (LambdaExp "x" (AppExp (VarExp "b") (VarExp "x")))))) emptyEnv)
    -- \b.\y.((\x.(\z.zx)(\x.bx))y)
    -- == \b.\y.(by)
  , (evaluate (LambdaExp "b" (LambdaExp "y" (AppExp (LambdaExp "x" (AppExp (LambdaExp "z" (AppExp (VarExp "z") (VarExp "x"))) (LambdaExp "x" (AppExp (VarExp "b") (VarExp "x"))))) (VarExp "y")))))
     == (Closure "b" (LambdaExp "y" (AppExp (VarExp "b") (VarExp "y"))) emptyEnv)
    -- \y.((\x.xx)y)
    -- == \y.(yy)
  , (evaluate (LambdaExp "y" (AppExp (LambdaExp "x" (AppExp (VarExp "x") (VarExp "x"))) (VarExp "y"))))
     == (Closure "y" (AppExp (VarExp "y") (VarExp ("y"))) emptyEnv)
    -- \a.\y.((\x.axxa)y)
    -- == \a.\y.(ayya)
```

```
  , (evaluate (LambdaExp "a" (LambdaExp "y" (AppExp (LambdaExp "x"
  ↪  (AppExp (AppExp (AppExp (VarExp "a") (VarExp "x")) (VarExp "x"))
  ↪  (VarExp "a"))) (VarExp "y")))))
      == (Closure "a" (LambdaExp "y" (AppExp (AppExp (AppExp (VarExp "a")
       ↪  (VarExp "y")) (VarExp "y")) (VarExp "a"))) emptyEnv)
    -- \q.\y.((\x.(\z.zx)q)y)
    -- == \q.\y.(qy)
  , (evaluate (LambdaExp "q" (LambdaExp "y" (AppExp (LambdaExp "x"
  ↪  (AppExp (LambdaExp "z" (AppExp (VarExp "z") (VarExp "x"))) (VarExp
  ↪  "q"))) (VarExp "y")))))
      == (Closure "q" (LambdaExp "y" (AppExp (VarExp "q") (VarExp "y")))
       ↪  emptyEnv)
    -- \b.\y.((\x.x((\z.zx)(\x.bx)))y)
    -- == \b.\y.(y(by))
  , (evaluate (LambdaExp "b" (LambdaExp "y" (AppExp (LambdaExp "x"
  ↪  (AppExp (VarExp "x") (AppExp (LambdaExp "z" (AppExp (VarExp "z")
  ↪  (VarExp "x"))) (LambdaExp "x" (AppExp (VarExp "b") (VarExp
  ↪  "x")))))) (VarExp "y")))))
      == (Closure "b" (LambdaExp "y" (AppExp (VarExp "y") (AppExp (VarExp
       ↪  "b") (VarExp "y")))) emptyEnv)
    -- (\a.a)(\b.b)(\c.cc)(\d.d)
    -- == \d.d
  , (evaluate (AppExp (AppExp (AppExp (LambdaExp "a" (VarExp "a"))
  ↪  (LambdaExp "b" (VarExp "b"))) ((LambdaExp "c" (AppExp (VarExp "c")
  ↪  (VarExp "c"))))) (LambdaExp "d" (VarExp "d"))))
      == (Closure "d" (VarExp "d") emptyEnv)
  ]

freeVars :: [Bool]
freeVars =
  [
    -- (\x.(\y.yx))y
    -- == {y}
    (freeVariablesExp (AppExp (LambdaExp "x" (LambdaExp "y" (AppExp
     ↪  (VarExp "y") (VarExp "x")))) (VarExp "y")))
      == Set.singleton "y"
    -- (\f.(\x.fx))(\y.(\x.y))
    -- == {}
  , (freeVariablesExp (AppExp (LambdaExp "f" (LambdaExp "x" (AppExp
  ↪  (VarExp "f") (VarExp "x")))) (LambdaExp "y" (LambdaExp "x" (VarExp
  ↪  "y")))))
      == Set.empty
    -- (\a.b)(\b.a)
    -- {a, b}
  , (freeVariablesExp (AppExp (LambdaExp "a" (VarExp "b")) (LambdaExp "b"
  ↪  (VarExp "a"))))
      == (Set.fromList ["a", "b"])
  ]

-- From lambda-calculus.pdf from lecture 2017-06-15
```

```haskell
alphaRenaming :: [Bool]
alphaRenaming =
  [
    -- \y.((\x.(\y.yx))y)
    -- == \y.(\y'.y'y)
    (evaluate (LambdaExp "y" (AppExp (LambdaExp "x" (LambdaExp "y"
     ↪  (AppExp (VarExp "y") (VarExp "x")))) (VarExp "y"))))
     == (Closure "y" (LambdaExp "y'" (AppExp (VarExp "y'") (VarExp "y")))
      ↪  emptyEnv)
    -- (\x.(\y.\x.y)x)
    -- == (\x.(\x'.x))
  , (evaluate (LambdaExp "x" (AppExp (LambdaExp "y" (LambdaExp "x"
  ↪  (VarExp "y"))) (VarExp "x"))))
     == (Closure "x" (LambdaExp "x'" (VarExp "x")) emptyEnv)
    -- (\f.(\x.fx))(\y.(\x.y))
    -- == (\x.(\x'.x))
  , (evaluate (AppExp (LambdaExp "f" (LambdaExp "x" (AppExp (VarExp "f")
  ↪  (VarExp "x")))) (LambdaExp "y" (LambdaExp "x" (VarExp "y")))))
     == (Closure "x" (LambdaExp "x'" (VarExp "x")) emptyEnv)
  ]

integerArithmetic :: [Bool]
integerArithmetic =
  [
    -- 123
    -- == 123
    (evaluate (IntExp 123))
     == (IntVal 123)
    -- 3 + 2
    -- == 5
  , (evaluate (IntOpExp Plus (IntExp 3) (IntExp 2)))
     == (IntVal 5)
    -- ((5 * 4) / 3) - (2 + 1)
    -- == 3
  , (evaluate (IntOpExp Minus (IntOpExp Divide (IntOpExp Times (IntExp 5)
  ↪  (IntExp 4)) (IntExp 3)) (IntOpExp Plus (IntExp 2) (IntExp 1))))
     == (IntVal 3)
    -- \x.(x + 2)
    -- == \x.(x + 2)
  , (evaluate (LambdaExp "x" (IntOpExp Plus (VarExp "x") (IntExp 2))))
     == (Closure "x" (IntOpExp Plus (VarExp "x") (IntExp 2)) emptyEnv)
    -- \a.\b.(((5 * a) / b) - (2 + 1))
    -- == \a.\b.(((5 * a) / b) - 3)
  , (evaluate (LambdaExp "a" (LambdaExp "b" (IntOpExp Minus (IntOpExp
  ↪  Divide (IntOpExp Times (IntExp 5) (VarExp "a")) (VarExp "b"))
  ↪  (IntOpExp Plus (IntExp 2) (IntExp 1))))))
     == (Closure "a" (LambdaExp "b" (IntOpExp Minus (IntOpExp Divide
      ↪  (IntOpExp Times (IntExp 5) (VarExp "a")) (VarExp "b")) (IntExp
      ↪  3))) emptyEnv)
  ]
```

```
integerFunctions :: [Bool]
integerFunctions =
  [
    -- def inc(x): x + 1; => (\x.x+1); inc(inc 0)
    -- == 2
    (evaluate (AppExp (LambdaExp "x" (IntOpExp Plus (VarExp "x") (IntExp
    ↪  1))) (AppExp (LambdaExp "x" (IntOpExp Plus (VarExp "x") (IntExp
    ↪  1))) (IntExp 0))))
     == (IntVal 2)
    -- def square(x): x*x; => (\x.x*x); square 11
    -- == 121
  , (evaluate (AppExp (LambdaExp "x" (IntOpExp Times (VarExp "x") (VarExp
  ↪  "x"))) (IntExp 11)))
      == (IntVal 121)
  ]

booleanFunctions :: [Bool]
booleanFunctions =
  [
    -- (1 == 1) 1 0
    -- == 1
    (evaluate (AppExp (AppExp (IntOpExp Equal (IntExp 1) (IntExp 1))
    ↪  (IntExp 1)) (IntExp 0)))
     == (IntVal 1)
    -- (1 == 2) 1 0
    -- == 0
  , (evaluate (AppExp (AppExp (IntOpExp Equal (IntExp 1) (IntExp 2))
  ↪  (IntExp 1)) (IntExp 0)))
     == (IntVal 0)
    -- (2 <= 3) 1 0
    -- == 1
  , (evaluate (AppExp (AppExp (IntOpExp LessEq (IntExp 2) (IntExp 3))
  ↪  (IntExp 1)) (IntExp 0)))
     == (IntVal 1)
    -- (3 <= 3) 1 0
    -- == 1
  , (evaluate (AppExp (AppExp (IntOpExp LessEq (IntExp 3) (IntExp 3))
  ↪  (IntExp 1)) (IntExp 0)))
     == (IntVal 1)
    -- (4 <= 3) 1 0
    -- == 0
  , (evaluate (AppExp (AppExp (IntOpExp LessEq (IntExp 4) (IntExp 3))
  ↪  (IntExp 1)) (IntExp 0)))
     == (IntVal 0)
  ]

-- Factorial is intended to be a comprehensive test of the lambda
↪  calculus
-- with integers. It includes recursion and conditionals
```

```haskell
factorialCalls :: [Bool]
factorialCalls = map (\(x, r) -> (evaluate (AppExp factorial x)) == r)
  [
    (IntExp 0, IntVal 1)        -- fact(0) = 1
  , (IntExp 1, IntVal 1)        -- fact(1) = 1
  , (IntExp 2, IntVal 2)        -- fact(2) = 2
  , (IntExp 3, IntVal 6)        -- fact(3) = 6
  , (IntExp 4, IntVal 24)       -- fact(4) = 24
  , (IntExp 5, IntVal 120)      -- fact(5) = 120
  , (IntExp 10, IntVal 3628800) -- fact(10) = 3628800
  ]
factorial :: Expr
{-
def factorial(x):
  if x <= 1:
    return 1
  return x * factorial(x - 1)
-}
factorial = AppExp factorialCore factorialCore

factorialCore :: Expr
factorialCore =
  LambdaExp "f"
  (LambdaExp "x" (lazyIfLam (IntOpExp LessEq (VarExp "x") (IntExp 1))
                  (IntExp 1)
                  (IntOpExp Times (VarExp "x")
                   (AppExp
                    (AppExp (VarExp "f") (VarExp "f"))
                    (IntOpExp Minus (VarExp "x") (IntExp 1))))))
  where lazyIfLam cond tBranch fBranch =
          let closureT = LambdaExp "" tBranch
              closureF = LambdaExp "" fBranch
              chosenBranch = AppExp (AppExp cond closureT) (closureF)
          in AppExp chosenBranch (IntExp 0)

basicPrompt :: [Bool]
basicPrompt =
  [
    -- #0
    -- == 0
    (evaluate (PromptExp (IntExp 0)))
     == (IntVal 0)
    -- ##7
    -- == 7
  , (evaluate (PromptExp (PromptExp (IntExp 7))))
     == (IntVal 7)
    -- (#123) / (#7)
    -- == 17
  , (evaluate (IntOpExp Divide (PromptExp (IntExp 123)) (PromptExp
  ↪  (IntExp 7))))
```

```
    == (IntVal 17)
   -- #\x.1
   -- == \x.1
  , (evaluate (PromptExp (LambdaExp "x" (IntExp 1))))
    == (Closure "x" (IntExp 1) emptyEnv)
   -- #((\x.#x * 4)8)
   -- == 32
  , (evaluate (PromptExp (AppExp (LambdaExp "x" (IntOpExp Times
  ↪  (PromptExp (VarExp "x")) (IntExp 4))) (IntExp 8))))
    == (IntVal 32)
  ]

basicControl :: [Bool]
basicControl =
  [
   -- Fx.1
   -- == 1
   (evaluate (ControlExp "x" (IntExp 1)))
    == (IntVal 1)
   -- (\x.0)(Fx.1)
   -- == 1
  , (evaluate (AppExp (LambdaExp "x" (IntExp 0)) (ControlExp "x" (IntExp
  ↪  1))))
    == (IntVal 1)
   -- (Fx.1)(\x.0)
   -- == 1
  , (evaluate (AppExp (ControlExp "x" (IntExp 1)) (LambdaExp "x" (IntExp
  ↪  0))))
    == (IntVal 1)
   -- (\x.0)(Fx.(x)1)
   -- == 0
  , (evaluate (AppExp (LambdaExp "x" (IntExp 0)) (ControlExp "x" (AppExp
  ↪  (VarExp "x") (IntExp 1)))))
    == (IntVal 0)
   -- (Fx.x(\y.y))(\x.0)
   -- == \x.0
  , (evaluate (AppExp (ControlExp "x" (AppExp (VarExp "x") (LambdaExp "y"
  ↪  (VarExp "y")))) (LambdaExp "x" (IntExp 0))))
    == (Closure "x" (IntExp 0) emptyEnv)
  ]

basicControlAndPrompt :: [Bool]
basicControlAndPrompt =
  [
   -- (\x.0)((\y.1)(Fz.2))
   -- == 2
   (evaluate (AppExp (LambdaExp "x" (IntExp 0)) (AppExp (LambdaExp "y"
   ↪  (IntExp 1)) (ControlExp "z" (IntExp 2)))))
    == (IntVal 2)
   -- (\x.0)#((\y.1)(Fz.2))
```

```
    -- == 0
  , (evaluate (AppExp (LambdaExp "x" (IntExp 0)) (PromptExp (AppExp
  ↪  (LambdaExp "y" (IntExp 1)) (ControlExp "z" (IntExp 2))))))
     == (IntVal 0)
    -- ((Fy.(\z.1))(\x.0))(2)
    -- == \z.1 env:{"y":=(Arg(\x.0, env:{}, Arg(2, env:{}, End)),
     ↪  kTrail:[])}
  , (evaluate (AppExp (AppExp (ControlExp "y" (LambdaExp "z" (IntExp 1)))
  ↪  (LambdaExp "x" (IntExp 0))) (IntExp 2)))
     == (Closure "z" (IntExp 1) (Env [("y", CapturedCont (Arg (LambdaExp
      ↪  "x" (IntExp 0)) emptyEnv (Arg (IntExp 2) emptyEnv End))
      ↪  emptyKTrail)]))
    -- (#((Fy.(\z.1))(\x.0)))(2)
    -- == 1
  , (evaluate (AppExp (PromptExp (AppExp (ControlExp "y" (LambdaExp "z"
  ↪  (IntExp 1))) (LambdaExp "x" (IntExp 0)))) (IntExp 2)))
     == (IntVal 1)
  ]

-- (\y.#((\f.\x.(factorial body))(Fk.((k)k)y)))
controlFactorial :: Expr
controlFactorial = LambdaExp "y" (PromptExp (AppExp factorialCore
↪  (ControlExp "k" (AppExp (AppExp (VarExp "k") (VarExp "k")) (VarExp
↪  "y")))))

controlAndPrompt :: [Bool]
controlAndPrompt =
  [
    -- (\a.a + 1)(Fb.(\c.3 * c)(b 7))
    -- == 24
    (evaluate (AppExp (LambdaExp "a" (IntOpExp Plus (VarExp "a") (IntExp
     ↪  1))) (ControlExp "b" (AppExp (LambdaExp "c" (IntOpExp Times
     ↪  (IntExp 3) (VarExp "c"))) (AppExp (VarExp "b") (IntExp 7))))))
     == (IntVal 24)
    -- (\a.(Fd.(\e.0)))(Fb.(\c.3 * c)(b 7))
    -- == \e.0 env:{"a":=7, "d":=(End, kTrail:[Fun(\c.3 * c
     ↪  env:{b:=(Fun(\a.(Fd.(\e.0)) env:{}, End), kTrail:[])}, End)])}
  , (evaluate (AppExp (LambdaExp "a" (ControlExp "d" (LambdaExp "e"
  ↪  (IntExp 0)))) (ControlExp "b" (AppExp (LambdaExp "c" (IntOpExp
  ↪  Times (IntExp 3) (VarExp "c"))) (AppExp (VarExp "b") (IntExp
  ↪  7))))))
     == (Closure "e" (IntExp 0) (Env [("d", CapturedCont End (ContTrail
      ↪  [Fun (Closure "c" (IntOpExp Times (IntExp 3) (VarExp "c")) (Env
      ↪  [("b",CapturedCont (Fun (Closure "a" (ControlExp "d" (LambdaExp
      ↪  "e" (IntExp 0))) (Env [])) End) (ContTrail [])]])) End])),
      ↪  ("a",IntVal 7)]))
    -- (\a.#((\b.b*2)(Fc.(c(c a)))))(Fd.(\e.e*3)(d 5))
    -- 60
```

```
  , (evaluate (AppExp (LambdaExp "a" (PromptExp (AppExp (LambdaExp "b"
  ↪  (IntOpExp Times (VarExp "b") (IntExp 2))) (ControlExp "c" (AppExp
  ↪  (VarExp "c") (AppExp (VarExp "c") (VarExp "a"))))))) (ControlExp
  ↪  "d" (AppExp (LambdaExp "e" (IntOpExp Times (VarExp "e") (IntExp
  ↪  3))) (AppExp (VarExp "d") (IntExp 5))))))
     == (IntVal 60)
    -- (\a.(\b.b*2)(Fc.(c(c a))))(Fd.(\e.e*3)(d 5))
    -- 180
  , (evaluate (AppExp (LambdaExp "a" (AppExp (LambdaExp "b" (IntOpExp
  ↪  Times (VarExp "b") (IntExp 2))) (ControlExp "c" (AppExp (VarExp
  ↪  "c") (AppExp (VarExp "c") (VarExp "a")))))) (ControlExp "d"
  ↪  (AppExp (LambdaExp "e" (IntOpExp Times (VarExp "e") (IntExp 3)))
  ↪  (AppExp (VarExp "d") (IntExp 5))))))
     == (IntVal 180)
    -- fact(5)
    -- 120
  , (evaluate (AppExp controlFactorial (IntExp 5)))
     == (IntVal 120)
    -- fact(fact(3))
    -- 720
  , (evaluate (AppExp controlFactorial (AppExp controlFactorial (IntExp
  ↪  3))))
     == (IntVal 720)
  ]

-- The following expressions for the factorial function are intended
-- to illustrate the ability of control and prompt to write more
-- concise functions.
factorialString :: String
factorialString = ppExpr factorial
-- ((\f.(\x.((((x <= 1))((\.1)))((\.(x * ((f)(f))((x -
↪  1))))))(0))))((\f.(\x.((((x <= 1))((\.1)))((\.(x * ((f)(f))((x -
↪  1))))))(0))))

controlFactorialString :: String
controlFactorialString = ppExpr controlFactorial
-- (\y.((\f.(\x.((((x <= 1))((\.1)))((\.(x * ((f)(f))((x -
↪  1))))))(0))))(Fk.((k)(k))(y)))
```