Homework 5
COSE212, Fall 2017

Hakjoo Oh

Due: 12/9, 24:00

Academic Integrity / Assignment Policy

- *All assignments must be your own work.*
- Discussion with fellow students is encouraged including how to approach the problem. However, your code must be your own.
  - Discussion must be limited to general discussion and must not involve details of how to write code.
  - You must write your code by yourself and must not look at someone else's code (including ones on the web).
  - Do not allow other students to copy your code.
  - Do not post your code on the public web.
- Violating above rules gets you 0 points for the entire HW score.

Problem 1 Consider the language:

```
type exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | READ
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
```

Define the function

```
expand : exp -> exp
```

that transforms an expression into a semantically-equivalent expression where every let-bound variable in the original expression gets replaced by its definition. Examples and caveat:

- Evaluating

  ```
  expand (LET ("x", CONST 1, VAR "x"))
  ```

  produces `CONST 1`.

- Evaluating

  ```
  expand (
    LET ("f", PROC ("x", VAR "x"),
     IF (CALL (VAR "f", ISZERO (CONST 0)),
      CALL (VAR "f", CONST 11),
      CALL (VAR "f", CONST 22))))
  ```

  produces

  ```
  IF (CALL (PROC ("x", VAR "x"), ISZERO (CONST 0)),
   CALL (PROC ("x", VAR "x"), CONST 11),
   CALL (PROC ("x", VAR "x"), CONST 22))
  ```

- Unused definitions should not go away. For example, Evaluating

  ```
  expand (LET ("x", ADD (CONST 1, ISZERO (CONST 0)), CONST 2))
  ```

  should return `LET ("x", ADD (CONST 1, ISZERO (CONST 0)), CONST 2)`, not `CONST 2`.

Try it yourself As discussed in class, the function `expand` can be used for implementing the let-polymorphic type system. The type checker `typeof : exp -> typ` in Homework 4 does not support polymorphism and would not accept the program:

```
# typeof(
  LET ("f", PROC ("x", VAR "x"),
   IF (CALL (VAR "f", ISZERO (CONST 0)),
    CALL (VAR "f", CONST 11),
    CALL (VAR "f", CONST 22))));;

= Equations =
t2 = (t6 -> t7)
t7 = t6
```

```
(t5 -> bool) = t2
t5 = bool
int = int
(t4 -> t1) = t2
t4 = int
(t3 -> t1) = t2
t3 = int

The program does not have type. Rejected.
```

With expand, however, the same type checking algorithm will succeed:

```
# typeof(
    expand(
      LET ("f", PROC ("x", VAR "x"),
       IF (CALL (VAR "f", ISZERO (CONST 0)),
        CALL (VAR "f", CONST 11),
        CALL (VAR "f", CONST 22)))));;

= Equations =
(t8 -> bool) = (t9 -> t10)
t10 = t9
t8 = bool
int = int
(t5 -> t1) = (t6 -> t7)
t7 = t6
t5 = int
(t2 -> t1) = (t3 -> t4)
t4 = t3
t2 = int

= Substitution =
t3 |-> int
t4 |-> int
t2 |-> int
t6 |-> int
t7 |-> int
t1 |-> int
t5 |-> int
t9 |-> bool
t10 |-> bool
t8 |-> bool

Type of the given program: int
```

Problem 2 Consider the language of lambda calculus:

```
type lambda = V of var
            | P of var * lambda
            | C of lambda * lambda
and var = string
```

A program in lambda calculus is a variable, a procedure abstraction, or a call.

Write the function

```
check : lambda -> bool
```

that checks if a given program is well-formed. A program is said to be *well-formed* if and only if the program does not contain free variables; i.e., every variable name is bound by some procedure that encompasses the variable. For example, well-formed programs are:

- `P ("a", V "a")`
- `P ("a", P ("a", V "a"))`
- `P ("a", P ("b", C (V "a", V "b")))`
- `P ("a", C (V "a", P ("b", V "a")))`

Ill-formed ones are:

- `P ("a", V "b")`
- `P ("a", C (V "a", P ("b", V "c")))`
- `P ("a", P ("b", C (V "a", V "c")))`