

# Test 2

## The take-home part

CS3490: Programming Languages

Due: Monday November 21, 10AM

### The Language PCF

In this assignment, you will implement a variant of the simply typed lambda calculus known as (signed) PCF. This language has the following definition.

#### Syntax

The *types* are generated from a single base type  $\mathbb{Z}$ , representing the type of integers (both positive and negative) using the function type constructor:

$$\mathbb{T} ::= \mathbb{Z} \mid \mathbb{T} \rightarrow \mathbb{T}$$

The *terms* are generated from variables and integer constants using application, abstraction, addition, and `lfZero` — an if-then-else construct based on testing whether a given number equals zero.

Finally, there is also a Y-combinator that enables recursive definitions.

$$\Lambda ::= \mathbb{V} \mid \Lambda\Lambda \mid \lambda\mathbb{V}\Lambda \mid \underline{\mathbb{Z}} \mid \Lambda + \Lambda \mid \text{lfZero}(\Lambda, \Lambda, \Lambda) \mid \mathbb{Y}$$

#### Reduction rules

If  $n$  is an integer, we write  $\underline{n}$  for the constant representing  $n$  inside the set of terms  $\Lambda$ .

The rules governing computation of this language are given as follows:

$$\begin{aligned} (\lambda x.s)t &\longrightarrow s[t/x] \\ \underline{m} + \underline{n} &\longrightarrow \underline{m+n} \\ \text{lfZero}(r, s, t) &\longrightarrow \begin{cases} s & \text{if } r = \underline{n}, \text{ where } n = 0 \\ t & \text{if } r = \underline{n}, \text{ where } n \neq 0 \end{cases} \\ \mathbb{Y}t &\longrightarrow t(\mathbb{Y}t) \end{aligned}$$

This results in a Turing-complete language that can encode arbitrary algorithms.

## Examples

1. Let  $\text{sub}_1 = \lambda n. n + \underline{-1}$ .

Then  $\text{sub}_1 \underline{1} \rightarrow \underline{1} + \underline{-1} \rightarrow \underline{1 - 1} = \underline{0}$ .

Similarly,  $\text{sub}_1 \underline{2} \rightarrow^* \underline{1}$  and  $\text{sub}_1 \underline{100} \rightarrow^* \underline{99}$ .

2. Let  $\text{mul} = Y(\lambda X. \lambda mn. \text{IfZero}(m, \underline{0}, n + X(\text{sub}_1 m)n))$ .

Then

$$\begin{aligned} \text{mul} &= Y(\lambda X. \lambda mn. \text{IfZero}(m, \underline{0}, n + X(\text{sub}_1 m)n)) \\ &= (\lambda X. \lambda mn. \text{IfZero}(m, \underline{0}, n + X(\text{sub}_1 m)n)) (Y(\lambda X. \lambda mn. \text{IfZero}(m, \underline{0}, n + X(\text{sub}_1 m)n))) \\ &= (\lambda X. \lambda mn. \text{IfZero}(m, \underline{0}, n + X(\text{sub}_1 m)n))(\text{mul}) \\ &= \lambda mn. \text{IfZero}(m, \underline{0}, n + \text{mul}(\text{sub}_1 m)n) \end{aligned}$$

In particular,

$$\begin{aligned} \text{mul } \underline{2} \underline{3} &= (\lambda mn. \text{IfZero}(m, \underline{0}, n + \text{mul}(\text{sub}_1 m)n)) \underline{2} \underline{3} \\ &= \text{IfZero}(\underline{2}, \underline{0}, \underline{3} + \text{mul}(\text{sub}_1 \underline{2}) \underline{3}) \\ &= \underline{3} + \text{mul}(\text{sub}_1 \underline{2}) \underline{3} \\ &= \underline{3} + \text{mul } \underline{1} \underline{3} \\ &= \underline{3} + (\lambda mn. \text{IfZero}(m, \underline{0}, n + \text{mul}(\text{sub}_1 m)n)) \underline{1} \underline{3} \\ &= \underline{3} + \text{IfZero}(\underline{1}, \underline{0}, \underline{3} + \text{mul}(\text{sub}_1 \underline{1}) \underline{3}) \\ &= \underline{3} + \underline{3} + \text{mul}(\text{sub}_1 \underline{1}) \underline{3} \\ &= \underline{6} + \text{mul } \underline{0} \underline{3} \\ &= \underline{6} + (\lambda mn. \text{IfZero}(m, \underline{0}, n + \text{mul}(\text{sub}_1 m)n)) \underline{0} \underline{3} \\ &= \underline{6} + \text{IfZero}(\underline{0}, \underline{0}, \underline{3} + \text{mul}(\text{sub}_1 \underline{0}) \underline{3}) \\ &= \underline{6} + \underline{0} \\ &= \underline{6} \end{aligned}$$

3. Let  $\text{fact} = Y(\lambda F. \lambda x. \text{IfZero}(x, \underline{1}, \text{mul } x (\text{fact}(\text{sub}_1 x))))$ . Then

$$\begin{aligned} \text{fact } \underline{n} &= Y(\lambda F. \lambda x. \text{IfZero}(x, \underline{1}, \text{mul } x (F(\text{sub}_1 x)))) \underline{n} \\ &= (\lambda F. \lambda x. \text{IfZero}(x, \underline{1}, \text{mul } x (F(\text{sub}_1 x)))) (\text{fact}) \underline{n} \\ &= \text{IfZero}(\underline{n}, \underline{1}, \text{mul } \underline{n} (\text{fact}(\text{sub}_1 \underline{n}))) \end{aligned}$$

In particular,

$$\begin{aligned} \text{fact } \underline{1} &= \text{IfZero}(\underline{1}, \underline{1}, \text{mul } \underline{1} (\text{fact}(\text{sub}_1 \underline{1}))) = \text{mul } \underline{1} (\text{fact } \underline{0}) = \text{mul } \underline{1} \underline{1} = \underline{1} \\ \text{fact } \underline{3} &= \text{IfZero}(\underline{3}, \underline{1}, \text{mul } \underline{3} (\text{fact}(\text{sub}_1 \underline{3}))) \\ &= \text{mul } \underline{3} (\text{fact}(\text{sub}_1 \underline{3})) = \text{mul } \underline{3} (\text{fact } \underline{2}) \\ &= \text{mul } \underline{3} (\text{IfZero}(\underline{2}, \underline{1}, \text{mul } \underline{2} (\text{fact}(\text{sub}_1 \underline{2})))) \\ &= \text{mul } \underline{3} (\text{mul } \underline{2} (\text{fact } \underline{1})) \\ &= \text{mul } \underline{3} (\text{mul } \underline{2} \underline{1}) = \text{mul } \underline{3} \underline{2} = \underline{6} \end{aligned}$$

## Haskell representation

The representation of the grammar of types is self-evident:

```
--      T ::= Z | T -> T
data Types = Ints | Fun Types Types
```

The representation of the grammar of terms will make use of *nested type encoding*, which uses a type parameter to keep track of the free variables of the term.

```
--      /\ ::= \/ | /\ /\ | \ \/ /\ | C Int | /\ + /\ | IfZero(/\, /\, /\) | Y
data Terms a = Var a | App (Terms a) (Terms a) | Abs (Terms (Maybe a)) | Const Integer
              | Add (Terms a) (Terms a) | IfZero (Terms a) (Terms a) (Terms a) | Y
              deriving (Show, Eq)
```

The idea behind this definition is that, an expression `t` of type `Terms a` represents a  $\lambda$ -term which has variables labelled by elements of type `a`.

Using a lambda abstraction introduces one more variable that can be used in the *body* of the function. This possibility is encoded by updating the type argument to `Maybe a`. Now there is a new variable — represented by `Nothing`, which can be used along with any other variable `x` of type `a` — represented by `Just x`.

### Example

*Make sure you understand this example before proceeding.*

Let  $M = x(\lambda z.xzy)$ .

$M$  has two free variables:  $x$  and  $y$ . They can be modeled as elements of type `a = Bool`.

$M$  also has a bound variable  $z$  created by lambda. It can be modeled as `Nothing :: Maybe Bool`.

The term itself can now be represented as follows. Note how the second occurrence of  $x$  shifts into a `Just`, because it is below one more abstraction.

```
m :: Terms Bool           -- x(\z.xzy). Here x is True, y is False.
m = App (Var True) (Abs (App (App (Var (Just True))      -- x
                              (Var Nothing))             -- z
                              (Var (Just False))))       -- y
```

## 1. Type constructor instances

### 1.1. Question (10 pts)

Define helper functions needed to declare `Monad` instance for the `Terms` constructor.

```
unitTerms :: a -> Terms a
liftTerms :: (a -> Terms b) -> Maybe a -> Terms (Maybe b)
bindTerms :: (a -> Terms b) -> Terms a -> Terms b
```

(*Hint.* In the abstraction case for `bindTerms`, you should use `liftTerms` to resolve a type mismatch involving `Maybe`.)

## 1.2. Question (10 pts)

Define `Functor`, `Applicative`, and `Monad` instances for `Terms`.

## 2. Parsing and Lexical Analysis

### 2.1. Question (10 pts)

Define the following lexical grammar

```
data Token = VSym String | CSym Integer | AddOp | IfZeroOp | YComb
           | LPar | RPar | Dot | Comma | Backslash
           | Err String | PT (Terms String)
           deriving (Eq, Show)
```

Write a function `lexer :: String -> [Token]` which takes a textual representation of a term and generates a list of tokens. The lexical rules are as follows:

- A *variable* is any string which *starts with a lowercase letter* and is followed by *zero or more alphanumeric characters*.

Examples of variables: `x y xy x1 xY x1z5 xXzZ`

- A *constant* is either a string which consists of one or more digits, OR it is the minus sign `'-'` followed by such a string. The latter form encodes *negative* integers.
- The addition operator is represented by plus: `+`
- `IfZero`-operator is represented on the input level by the string `IfZero`, and the `Y`-combinator is represented by `Y`. (Note that both begin with an upper-case letter.)
- The punctuation tokens are self-explanatory.
- `Err` and `PT` are auxiliary tokens to be used internally by the parser for error flagging and keeping track of parsed subterms.

### 2.2. Question (10 pts)

Write a function `parser :: [Token] -> Either (Terms String) String` that takes a list of tokens and attempts to produce a term out of them.

If no parse is possible, it should output `Right e` with `e :: String` being the error message identifying the error.

Otherwise, it should output `Left t`, where `t :: Terms String`.

(*Hint.* Write a shift-reduce helper function following the examples done in class.)

The rules for parsing should follow the standard conventions for writing lambda terms on paper.

- Variables and constants appear on their own with no additional signs.

- Application is written by juxtaposition. Two terms  $M$  and  $N$  appearing next to each other denote the application term  $MN$ .
- Abstraction is written in the form  $\backslash x.M$ : backslash, a variable, a dot, and a term.
- Addition is written in the standard infix notation.
- `lfZero` is written as a function of three arguments: `lfZero(L, M, N)`. The function name `lfZero`, the parentheses, and the commas are all mandatory.

*Another hint.* To parse lambda abstractions, you should use the function

```
capture :: String -> Terms String -> Terms (Maybe String)
capture x s = s >>= (\y -> if x==y then Var Nothing else Var (Just y))
```

With this function, you can parse a lambda expression presented as described above, by promoting its body, which should be a parsed expression of type `Terms String`, into a term of type `Terms (Maybe String)`, and then using the `Abs` constructor.<sup>1</sup>

### 3. Reduction and Evaluation

Let  $\rightarrow_0 \subseteq \Lambda \times \Lambda$  be the relation of *root* reduction, obtained as the union of the reduction rules given on page 1.

That is,  $s \rightarrow_0 t$  if and only if  $s \rightarrow t$  using one of the rules given there.

Let  $\Rightarrow$  be the relation of *parallel reduction*, defined as the closure of  $\rightarrow_0$  under the congruence rules:

$\frac{x \in \mathbb{V}}{x \Rightarrow x}$	$\frac{n \in \mathbb{Z}}{n \Rightarrow n}$	$\frac{s \rightarrow_0 t}{s \Rightarrow t}$	$\frac{r_0 \Rightarrow r_1 \quad s_0 \Rightarrow s_1 \quad t_0 \Rightarrow t_1}{\text{lfZero}(r_0, s_0, t_0) \Rightarrow \text{lfZero}(r_1, s_1, t_1)}$
$\frac{r_0 \Rightarrow r_1}{\lambda x.r_0 \Rightarrow \lambda x.r_1}$	$\frac{s_0 \Rightarrow s_1 \quad t_0 \Rightarrow t_1}{s_0 t_0 \Rightarrow s_1 t_1}$	$\frac{s_0 \Rightarrow s_1 \quad t_0 \Rightarrow t_1}{s_0 + t_0 \Rightarrow s_1 + t_1}$	$\frac{}{Y \Rightarrow Y}$

Finally, let  $\Rightarrow^*$  be the reflexive-transitive closure of  $\Rightarrow$ :

$\frac{}{s \Rightarrow^* s}$	$\frac{r \Rightarrow s \quad s \Rightarrow^* t}{r \Rightarrow^* t}$
------------------------------	---

---

<sup>1</sup>

## Examples

Let  $I = \lambda x.x$  and  $K = \lambda x.\lambda y.x$ . Then

- $I\underline{5} \rightarrow_0 \underline{5}$ ,  $\underline{2} + \underline{5} \rightarrow_0 \underline{7}$ ,  $YI \rightarrow_0 I(YI)$ .

Each of these steps can be performed by applying one of the reduction rules given on page 1 at the very top of the term.

- $K(I\underline{5}) \not\rightarrow_0 K\underline{5}$ ,  $\lambda z.z + (\underline{2} + \underline{5}) \not\rightarrow_0 \lambda z.z + \underline{7}$ .

Neither of these steps can be performed at the very top of the term. However,

- $K(I\underline{5}) \Rightarrow K\underline{5}$ ,  $\lambda z.z + (\underline{2} + \underline{5}) \Rightarrow \lambda z.z + \underline{7}$ .

Both of them can be performed once congruence rules are added.

- $\text{IfZero}(I\underline{5}, \underline{2} + \underline{5}, YI) \Rightarrow \text{IfZero}(\underline{5}, \underline{7}, I(YI))$ .

Also,  $\text{IfZero}(\underline{5}, \underline{7}, I(YI)) \Rightarrow I(YI)$ . And  $I(YI) \Rightarrow YI$ .

Parallel subterms can be contracted simultaneously by  $\Rightarrow$  if they do not overlap.

- $\text{IfZero}(I\underline{5}, \underline{2} + \underline{5}, YI) \not\Rightarrow YI$ .

$\Rightarrow$  is limited to a single parallel step only.

- $\text{IfZero}(I\underline{5}, \underline{2} + \underline{5}, YI) \Rightarrow^* YI$ .

$\Rightarrow^*$  allows chaining together any number (including 0) of  $\Rightarrow$ -steps.

### 3.1. Question (10 pts)

Implement the relation  $\Rightarrow$  in the form of a function `predstep :: Terms a -> Terms a`.

If `m :: Terms a` is an internal representation of the term  $M$ , and  $M \Rightarrow N$ , then `predstep m` should produce an internal representation of the term  $N$ .

*Hint.* You should use a variation of the similar function that we implemented in the lecture on 11.14 — see the relevant code files. However, the rules followed by your function `predstep` should instead be those that are given above in the definition of  $\Rightarrow$ . Your function should basically implement that definition, rule-by-rule.

You should use the `bind` for terms to implement substitution:

```
subst :: Terms (Maybe a) -> Terms a -> Terms a
subst s t = s >>= maybe t Var
```

This code assumes you have correctly solved Question 3.

The basic behavior of the `predstep` function should be:

- If a given term can be reduced at the top level, do it and return the result.
- If a given term cannot be reduced anymore (constant or variable), return it.
- If a term is neither reducible at the top level, nor is a leaf of the syntax tree, then recursively reduce its immediate subterms.

This code assumes you have correctly solved Question 3.

### 3.2. Question (10 pts)

Implement the relation  $\Rightarrow^*$  in the form of a function

`preds :: Eq a => Terms a -> Terms a`

The function should apply `predstep` while it is making progress. Once the output produced by `predstep` is the same as the input, this means that the normal form has been reached, and the resulting term can be returned.

## 4. Typing and Computation

Recall that the set of types of PCF is generated by the grammar

$$\mathbb{T} ::= \mathbb{Z} \mid \mathbb{T} \rightarrow \mathbb{T}$$

The typing rules specify the types allowed for each term constructor. They are given in the figure below.

$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \text{Var}$	$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x. r : A \rightarrow B} \text{Abs}$	$\frac{n \in \{0, 1, -1, 2, -2, \dots\}}{\Gamma \vdash \underline{n} : \mathbb{Z}} \text{Num}$
$\frac{\Gamma \vdash s : A \rightarrow B \quad \Gamma \vdash t : A}{\Gamma \vdash st : B} \text{App}$		$\frac{\Gamma \vdash s : \mathbb{Z} \quad \Gamma \vdash t : \mathbb{Z}}{\Gamma \vdash s + t : \mathbb{Z}} \text{Add}$
$\frac{\Gamma \vdash r : \mathbb{Z} \quad \Gamma \vdash s : A \quad \Gamma \vdash t : A}{\Gamma \vdash \text{IfZero}(r, s, t) : A} \text{IfZero}$		$\frac{\Gamma \vdash t : A \rightarrow A}{\Gamma \vdash \mathbf{Y}t : A} \text{Y-Comb}$

### 4.1. Question (10 pts)

Infer a valid type  $A$  for the PCF term  $t$  given below by computing the complete derivation tree of the typing judgment  $\vdash t : A$  using the typing rules above:

$$(\lambda x \lambda f. f(fx)) \underline{3} (\lambda y. y + y)$$

Show the full derivation tree in your answer.

### 4.2. Question (10 pts)

Compute the normal form of the above term using the reduction rules of the language, until a value is reached and no more simplifications are possible.

### 4.3. Question (10 pts)

Let  $\text{mul} = \lambda f. \lambda m. \lambda n. \text{IfZero}(m, \underline{0}, n + f(m + \underline{-1})n)$  be the term from Example 2 on p.2.

Infer a valid type  $A$  for this term and show that it's correct by giving an exact derivation of  $\vdash \text{mul} : A$  using the typing rules above.

#### 4.4. Question (10 pts)

Construct a term `iter` of type  $\mathbb{Z} \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$  which has the following behavior:

$$\begin{aligned}\text{iter } \underline{0} \ f \ x &= x \\ \text{iter } \underline{n+1} \ f \ x &= f(\text{iter } \underline{n} \ f \ x)\end{aligned}$$

That is, `iter` is a higher-order function which takes as input a number  $n$ , a function  $f$ , and an input  $x$ , and applies the function  $f$   $n$  times to the input  $x$ .

*Hint.* First implement such a function in Haskell, then translate it into PCF one language construct at a time. This process will involve:

- Implementing pattern matching using `lfZero`.
- Implementing function definition using  $\lambda$ .
- Implementing recursion using the Y-combinator.

### Testing and debugging

You are welcome to make use of the following functions while debugging your program.

```
-- a pretty-printer for Terms String
showTerm :: Terms String -> String
showTerm = st 0
  where -- st d (Var "o") = "v0"
        st d (Var x) = let (h,v) = span (== '|') x
                        l = length h + 1
                        in if l <= d then v ++ show (d - l) else drop l x
        st d (Const n) = show n
        st d (Y) = "Y"
        st d (Add s t) = "(" ++ st d s ++ " + " ++ st d t ++ ")"
        st d (IfZero r s t) = "IfZero(" ++ st d r ++ "," ++ st d s ++ "," ++ st d t ++ ")"
        st d (Abs r) = '\\' : 'v' : show d ++ "." ++ st (d+1) (maybe "v" ('|':) <$> r)
        st d (App s t@(App _ _)) = st d s ++ "(" ++ st d t ++ ")"
        st d (App s t@(Abs _)) = st d s ++ "(" ++ st d t ++ ")"
        st d (App s t) = st d s ++ st d t

printTerm :: Terms String -> IO ()
printTerm = putStrLn . showTerm

readTerm :: String -> Terms String
readTerm s = case parser (lexer s) of
  Just t -> t
  Nothing -> error $ "No parse: " ++ s

eval :: String -> Terms String
eval = preds . readTerm

($$) :: Terms String -> Integer -> Terms String
t $$ n = App t (Const n)
```

```

-- subtract 1
sub1 = "\\n.(n+(-1))"
-- multiply two inputs
mul = "Y(\\f.\\m.\\n.IfZero(m,0,n+(f(m+-1)n)))"
-- factorial
fac = "Y(\\f.\\x.IfZero(x,1," ++ '(' : mul ++ ")x(f(x+-1))))"

mult = readTerm mul -- parse of mul
fact = readTerm fac -- parse of fac

threeTimesTwenty = (printTerm . preds) (mult $$ 3 $$ 20)
fiveFactorial     = (printTerm . preds) (fact $$ 5)

main :: IO ()
main = do
  putStrLn "Enter a PCF term:"
  s <- getLine
  let repl t = do
    putStrLn "Enter a command"
    i <- getLine
    case i of
      "lex"   -> putStrLn (show (lexer s)) >> repl t
      -- "parse" -> putStrLn (showTerm (readTerm s)) >> repl t
      "red"   -> putStrLn (showTerm t') >> repl t' where t' = predstep t
      "norm"  -> putStrLn (showTerm t') >> repl t where t' = preds t
      "show"  -> putStrLn (showTerm t) >> repl t
      "quit"  -> return ()
      "new"   -> main
  repl (readTerm s)

```

```

*Main> threeTimesTwenty
60
*Main> fiveFactorial
120
*Main> lexer mul
[YComb,LPar,Backslash,VSym "f",Dot,Backslash,VSym "m",Dot,Backslash,VSym "n",Dot,IfZeroOp,LPar,VSym "m",Comma,
  CSym 0,Comma,VSym "n",AddOp,LPar,VSym "f",LPar,VSym "m",AddOp,CSym (-1),RPar,VSym "n",RPar,RPar,RPar]
*Main> parser it
Just (App Y (Abs (Abs (Abs (IfZero (Var (Just Nothing)) (Const 0) (Add (Var Nothing) (App (App (Var (Just (Just
  Nothing))) (Add (Var (Just Nothing)) (Const (-1)))) (Var Nothing)))))))
*Main> predstep (mult $$ 2 $$ 3)
App (App (App (Abs (Abs (Abs (IfZero (Var (Just Nothing)) (Const 0) (Add (Var Nothing) (App (App (Var (Just (
  Just Nothing))) (Add (Var (Just Nothing)) (Const (-1)))) (Var Nothing))))))) (App Y (Abs (Abs (Abs (IfZero
  (Var (Just Nothing)) (Const 0) (Add (Var Nothing) (App (App (Var (Just (Just Nothing))) (Add (Var (Just
  Nothing)) (Const (-1)))) (Var Nothing))))))) (Const 2)) (Const 3)
*Main> predstep it
App (App (Abs (Abs (IfZero (Var (Just Nothing)) (Const 0) (Add (Var Nothing) (App (App (App Y (Abs (Abs (Abs (
  IfZero (Var (Just Nothing)) (Const 0) (Add (Var Nothing) (App (App (Var (Just (Just Nothing))) (Add (Var (
  Just Nothing)) (Const (-1)))) (Var Nothing))))))) (Add (Var (Just Nothing)) (Const (-1)))) (Var Nothing)))
  ))) (Const 2)) (Const 3)
*Main> predstep it
App (Abs (IfZero (Const 2) (Const 0) (Add (Var Nothing) (App (App (App Y (Abs (Abs (Abs (IfZero (Var (Just
  Nothing)) (Const 0) (Add (Var Nothing) (App (App (Var (Just (Just Nothing))) (Add (Var (Just Nothing)) (
  Const (-1)))) (Var Nothing))))))) (Add (Const 2) (Const (-1)))) (Var Nothing)))))) (Const 3)
*Main> predstep it

```

