

**Exercise 1.** Partial recursive functions

Show that every **while** program over  $\mathbb{N}$  is equivalent to a **while** program with at most 1 *while* loop.

**Exercise 2.** Primitive recursive functions

Show that a function is primitive recursive iff it is computed by a Turing Machine with a primitive recursive time bound.

For this problem, assume a Turing Machine computes a function  $\Sigma^* \rightarrow \{0, 1\}$ , where the argument is the input string, and the result is 1 iff the Turing machine accepts.

A Turing Machine is time-bounded by a primitive recursive function  $f : \mathbb{N} \rightarrow \mathbb{N}$  iff for any input string  $x$ , the Turing Machine accepts or rejects within  $f(|x|)$  moves.

**Exercise 3.** Laboratory

For this lab, we will build an evaluator for the  $\lambda$ -calculus with numbers and conditionals. Here is the syntax of the language:

$e ::= i$	(numbers)
$x$	(variables)
$e_1 e_2$	(application)
$\lambda x.e$	(functions)
$e_1 \text{ binop } e_2$	(binary arithmetic)
<b>if</b> $e_1$ <b>then</b> $e_2$ <b>else</b> $e_3$	(conditional)
<b>let</b> $v = e_1$ <b>in</b> $e_2$	(let definition)
<b>letrec</b> $f v = e_1$ <b>in</b> $e_2$	(recursive function definition)

$\text{binop} ::= + \mid - \mid * \mid / \mid = \mid < \mid > \mid \geq \mid \neq$

Evaluation is defined using *evaluation rules*. If  $e \downarrow v$  we say the program  $e$  *evaluates to* the value  $v$ , where a value is a number or a function. The exact specification of evaluation is defined through evaluation rules. If we were using substitution (like we did in class), the evaluation semantics would look something like this.

$$\begin{array}{c}
 \frac{}{i \downarrow i} \text{ number} \qquad \frac{e_1 \downarrow i_1 \quad e_2 \downarrow i_2}{e_1 \text{ binop } e_2 \downarrow (i_1 \text{ binop } i_2)} \text{ binop} \\
 \\
 \frac{e_1 \downarrow \lambda x.e_3 \quad e_2 \downarrow v_2 \quad e_3[e_2/x] \downarrow v_3}{e_1 e_2 \downarrow v_3} \text{ app} \qquad \frac{}{\lambda x.e \downarrow \lambda x.e} \text{ fun} \\
 \\
 \frac{e_1 \not\downarrow 0 \quad e_2 \downarrow v_2}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \downarrow v_2} \text{ if-true} \\
 \\
 \frac{e_1 \downarrow 0 \quad e_3 \downarrow e_3 \downarrow v_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \downarrow v_4} \text{ if-false}
 \end{array}$$

The problem with this semantics is that it is really inefficient because the usual definition of substitution  $e_1[e_2/v]$  requires copying the entire expression  $e_1$ .

An alternative definition that is much more widely used is to define a variable *environment* that keeps track of values for variables. When a variable is evaluated, the value is retrieved from the environment. We'll use the notation  $\sigma$  for an environment. The environment supports two operations: we get the value of a variable  $v$  with the operation  $\sigma[v]$ ; and we can set the value of a variable  $v$  to value  $x$  with the operation  $\sigma[v := x]$ .

Next, we redefine the evaluation rules to use the environment. If  $e \sigma \downarrow x$  we say expression  $e$  evaluates to the value  $x$  in environment  $\sigma$ .

$$\frac{}{i \sigma \downarrow i} \text{ number} \qquad \frac{e_1 \sigma \downarrow i_1 \quad e_2 \sigma \downarrow i_2}{(e_1 \text{ binop } e_2) \sigma \downarrow (i_1 \text{ binop } i_2)} \text{ binop}$$

$$\frac{e_1 \sigma \downarrow \lambda x.e_3 \quad e_2 \sigma \downarrow v_2 \quad e_3 \sigma[x := v_2] \downarrow v_3}{(e_1 e_2) \sigma \downarrow v_3} \text{ app} \qquad \frac{}{(\lambda x.e) \sigma \downarrow (\lambda x.e) \sigma} \text{ fun}$$

$$\frac{e_1 \sigma \neq 0 \quad e_2 \sigma \downarrow v_2}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \sigma \downarrow v_2} \text{ if-true}$$

$$\frac{e_1 \sigma \downarrow 0 \quad e_3 \sigma \downarrow v_3}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \sigma \downarrow v_2} \text{ if-true}$$

$$\frac{e_1 \sigma \downarrow v_1 \quad e_2 \sigma[x := v_1] \downarrow v_2}{(\text{let } x = e_1 \text{ in } e_2) \sigma \downarrow v_2} \text{ let}$$

$$\frac{e_2 \sigma[f := \mu f.\lambda x.e_1]}{(\text{letrec } f \ x = e_1 \text{ in } e_2) \sigma \downarrow v_2} \text{ letrec}$$

Here is an English description of the rules.

**number** Numbers evaluate to themselves.

**binop** To evaluate an expression like  $e_1 + e_2$ , evaluate  $e_1$  to get a number  $i_1$ , then evaluate  $e_2$  to get a number  $i_2$ , then add  $i_1$  and  $i_2$ . It is an error if  $e_1$  and  $e_2$  do not evaluate to numbers.

For the relations like  $e_1 < e_2$ , the expression evaluates to 1 if the relation is true, and 0 otherwise.

**app** To evaluate a function application  $e_1 e_2$ : First evaluate  $e_1$ . It should evaluate to some function  $\lambda x.e_3$ ; if not, it is an error. Next, evaluate  $e_2$  to get a value  $v_2$ . Finally, substitute  $v_2$  for the variable  $x$  in  $e_3$ , and evaluate it to get the result.

**fun** Functions evaluate to themselves.

**if** To evaluate a conditional like **if**  $e_1$  **then**  $e_2$  **else**  $e_3$ : First, evaluate  $e_1$ . It should evaluate to a number  $i_1$ . If  $i_1 = 0$ , then evaluate  $e_3$ . Otherwise, evaluate  $e_2$ .

**let** To evaluate a let definition like **let**  $x = e_1$  **in**  $e_2$ , evaluate the expression  $e_1$  to get a value  $v_1$ . Add the binding  $x := v_1$  to the environment, then evaluate  $e_2$ .

**letrec** Ok, this is the hard one, and will give you the most trouble in this lab. The  $\mu f.e$  operator is called a *fixpoint* operator, and it has the semantics

$$\mu f.e \rightarrow_{\beta} e[\mu f.e/f].$$

That is, the semantics is the same as  $Y(\lambda f.e)$  (remember the definition of the Y-combinator from class).

However, we will not use this semantics directly. We'll explain more when we describe how to evaluate the recursive definition.

# 1 Getting started

We are not going to consider parsing in this problem. Instead, we define the  $\lambda$  expressions as an ML type. The template for this code is in the file `eval.ml`. We'll assume that variables are represented as strings. We have the type `binop` that represents binary arithmetic; and the type `exp` that represents  $\lambda$ -expressions.

```
type binop =
  AddOp      (* Integer + *)
| SubOp      (* Integer - *)
| MulOp      (* Integer * *)
| DivOp      (* Integer / *)
| EqOp       (* Integer = *)
| NeqOp      (* Integer != *)
| LtOp       (* Integer < *)
| LeOp       (* Integer <= *)
| GtOp       (* Integer > *)
| GeOp       (* Integer >= *)

type exp =
  Int of int
| Var of string
| Apply of exp * exp
| Binop of binop * exp * exp
| Lambda of string * exp
| If of exp * exp * exp
| LetVar of string * exp * exp
| LetFun of string * string * exp * exp
```

The `LetVar` represents a normal **let** definition, and `LetRec` represents a recursive function definition.

## 1.1 Program values

The *values* are the programs that evaluate to themselves. We have only two values in this language: integers and functions. We use the following type definition.

```
type value =
  ValInt of int
| ValLambda of (value -> value)
```

Note that `ValLambda` represents that value of a function as a ML function.

## 1.2 Variable environment

The variable environment is defined as the type `venv`, which is defined using the `Map` module in the usual way. We have three basic operations:

<code>venv_empty : venv</code>	The empty state
<code>venv_add : venv -&gt; string -&gt; value -&gt; venv</code>	Add a variable binding
<code>venv_find : venv -&gt; string -&gt; value</code>	Lookup the value of a variable

The `venv_add venv v x` function corresponds to  $\sigma[v := x]$ , and `venv_find venv v` corresponds to  $\sigma[v]$ . Note that the `venv_find` function aborts with an error message if the variable is not bound.

### 1.3 Evaluation

The **main** objective in this lab is to build an `eval : venv -> exp -> value` function that evaluates an expression and produces a result. The `venv` represents the variable environment  $\sigma$ , and the expression argument is the term to be evaluated. Of course, evaluation of expressions in the  $\lambda$ -calculus may not terminate, so our function may not terminate either.

The following text described some pseudo-code that sketches the outline of the `eval` function.

```
let rec eval venv e =
  match e with
  | Int i ->
    ValInt i
  | Var v ->
    venv_find venv v
  | Apply (e1, e2) ->
    1. evaluate e1 to get a function ValLambda f
    2. evaluate e2 to get a value x
    3. return f(x)
  | Binop (e1, AddOp, e2) ->
    1. evaluate e1 to a number i1
    2. evaluate e2 to a number i2
    3. return ValInt (i1 + i2)
  | IfThenElse (e1, e2, e3) ->
    1. evaluate e1 to a number i1
    2. if i1 <> 0 then evaluate e2
       otherwise evaluate e3
  | Lambda (v, e) ->
    1. Define a function that takes a value x and
       a. Adds [v := x] to the current environment
       b. Evaluates e
    ValLambda (fun x -> eval (venv_add venv v x) e)
  | LetVar (v, e1, e2) ->
    1. Evaluate e1 to get a value v1
    2. Add [v := v1] to the venv
    3. Evaluate e2 in the new venv
  | LetFun (f, v, e1, e2) ->
    See discussion below
```

Note the evaluation of `Lambda (v, e)`. The meaning of this expression is that it is supposed to be a function that takes a value for  $v$  and then returns the value of expression  $e$ . We represent this as a value of the form `ValLambda f`, where  $f$  is a function that takes a value  $x$ , and returns the evaluated value of expression  $e$ . Of course, another way to write this is:

```
let f x =
  let venv = venv_add venv v x in
  eval venv e1
in
ValLambda f
```

The evaluation of `LetFun (f, v, e1, e2)` is similar, except that the function  $f$  is defined *within its own body*. That is, the `LetFun` is a recursive definition, while `Lambda` is not.

How do we make the function available in its own body? Essentially, we want to do that same as we did for ValLambda, but we also adds the function to the environment.

```
let rec g x =
  let venv = venv_add venv v x in
  let venv = (* Define g as the value for f in venv *) in
    eval venv e1
in
  ValLambda g
```

## 1.4 What to turn in

For this lab, you have to implement the eval function in the file eval.ml.

There are other parts in the file of course, including type definitions, the venv definition, printing functions, and functions to get the arguments from the command line.

At the bottom of the file, there are two functions to compute factorial and Fibonacci numbers.

You should turn in the following:

- Your completed eval.ml file.
- A README file describing what you did, and any problems you had.
- Output of runs for the following arguments: 0, 1, 10, 32.

You can use the -debug\_eval option to help debug your code. Here is an example:

```
<jyh:kenai 1463>./eval -debug_eval 0
Evaluating let rec fact i = if (i < 0) then 1 else 1 in fact(0)
Evaluating fact(0)
Evaluating fact
Evaluating 0
Evaluating if (i < 0) then 1 else 1
Evaluating (i < 0)
Evaluating i
Evaluating 0
Evaluating 1
fact(0) = 1
Evaluating let rec fib i = if (i <= 1) then i else i in fib(0)
Evaluating fib(0)
Evaluating fib
Evaluating 0
Evaluating if (i <= 1) then i else i
Evaluating (i <= 1)
Evaluating i
Evaluating 1
Evaluating i
fib(0) = 0
```