CSE 120/130

Introduction to

Programming Languages and Techniques

Fall 2000

Handout 5

Evaluation

A question about evaluation

Consider the functions

```
let f x = x + 3
let g y = y + 3
```

Is there any difference between `f` and `g`? They differ only in the name of the argument variable. It seems clear that the particular name given to an argument does not make a difference in the "meaning" of the function.

Question: Are there cases in which the particular name used for an argument actually does make a difference in the meaning of a function?

Answer: Yes, if there are free variables in a function definition and we use a different evaluation scheme than the one used in OCAML. Consider:

```
let x = 3
let foo y  = x
let bar x = foo 10
bar 5
```

In OCAML, this evaluates to `3`, since `foo` gets the value of `x` from the environment in which `foo` is *defined*. This is called STATIC binding, and we've talked about it a lot, although not under that name.

An alternative to STATIC binding is DYNAMIC binding, in which the free variables in a function get their values from the environment in which the function is *evaluated*, not where it is defined. If OCAML has dynamic binding (remember, it doesn't), then

```
let x = 3
let foo y  = x
let bar x = foo 10
bar 5
```

evaluates to `5`. When `bar 5` is evaluated, `x` is set to 5. Then, when `foo 10` is evaluated, it returns the value of `x` in the evaluation environment, namely `5`.

An answer to the question about argument names: Suppose that instead of

```
let x = 3
let foo y  = x
let bar x = foo 10
bar 5
```

we change the definition of `bar` to use `z` instead of `x`.

```
let x = 3
let foo y  = x
let bar z = foo 10
bar 5
```

Under static binding, this evaluates to `3`, just as before. But with dynamic binding, it would now also evaluate to `3`, whereas before it evaluated to `5` under dynamic binding. So here is a case in which the name used for an argument can affect the evaluation of an expression.

THIS IS NOT GOOD - It makes it very hard to determine whether programs will do what we want them to do, not an easy task to begin with.

Static and dynamic binding - summary

dynamic binding - a function is evaluated in the environment in which it is evaluated.

static binding - a function is evaluated in the environment in which it is defined.

Crucial point: For static binding, it is not sufficient to identify value of a function definition with just the arguments and body of the function. The value instead consists of the arguments and body of the function, *together with environment in which the function was defined.* This text-environment pair is called a CLOSURE.

(Most languages use static, not dynamic, binding.)

A more careful look at evaluation

We are now going to take a more precise look at what happens when expressions are evaluated in ML, not just for functions as just discussed, but for all expression in ML.

To get a more precise picture of how this works, we'll proceed in two steps:

- define the *syntax* of a simplified fragment of ML, omitting most features of the full language (e.g., most base types and built-in functions, lists, tuples, datatypes, pattern matching, exceptions, top-level definitions, etc.) in order to emphasize the most fundamental ones;
- define an *evaluation semantics* for this language by giving a collection of rules specifying how to evaluate each form of expression in a given *environment*.

Core ML

An *expression* in Core ML can have any of the following forms:

- an *integer constant* `0`, `1`, `2`, etc.
- a *boolean constant* `true` or `false`
- a *variable* `x`
- an *arithmetic expression* e_1`+`e_2, e_1`-`e_2, or e_1`*`e_2, where e_1 and e_2 are expressions
- a *boolean expression* e_1`<`e_2, where e_1 and e_2 are expressions
- a *conditional expression* `if` e_1 `then` e_2 `else` e_3, where e_1, e_2, and e_3 are expressions
- a *value definition expression* `let x =` e_1 `in` e, where `x` is a variable and e_1 and e are expressions
- a *function definition expression* `let rec f x =` e_1 `in` e, where `f` and `x` are variables and e_1 and e are expressions (all function definitions in Core ML are recursive and take a single parameter)
- a *function application* `f(`e`)`, where `f` is a variable and e is an expression

Evaluation

Recall (from Handout 3) that expressions are always evaluated in some *environment* that assigns values to their variables.

If we start ML and make the top-level definition `let foo = 3;;`, we obtain the environment

`foo` $\rightarrow$ *3*
`...`

where . . . represents the "built in" environment provided by ML.

Evaluating constants and variables

We can now give a straightforward rule for evaluating each form of expression in Core ML.

- To evaluate a constant number like `56` in an environment `E`, we simply yield `56` as the result (ignoring `E`).
- Similarly, to evaluate a boolean constant `true` or `false` in an environment `E`, we yield `true` or `false` as the result.
- To evaluate a variable `x` in an environment `E`, we look up the (most recent) binding of `x` in `E` and return the associated value.

Evaluating arithmetic expressions

To evaluate an arithmetic expression e_1+e_2 in environment E, we do the following:

1. evaluate e_1 in E to obtain a value v_1
2. evaluate e_2 in E to obtain a value v_2
3. return $v_1 + v_2$ (assuming both v_1 and v_2 are integers; otherwise raise an error)

Example

To evaluate the expression x+y in the environment

x	$\rightarrow$	7
y	$\rightarrow$	6
x	$\rightarrow$	5
...		

we calculate as follows:

1. To evaluate x+y, we must first evaluate x and y
   (a) To evaluate x, we look it up in the environment, yielding 7
   (b) To evaluate y, we look it up in the environment, yielding 6
2. We now add 7 and 6 to produce 13, which is the result of evaluating the whole expression x+y

Expressions of the forms e_1-e_2 and e_1*e_2 are treated similarly.

Evaluating boolean expressions

To evaluate a boolean expression e_1<e_2 in environment E, we do the following:

1. evaluate e_1 in E to obtain a value v_1
2. evaluate e_2 in E to obtain a value v_2
3. if $v_1 < v_2$, then return `true`; otherwise return `false`

Evaluating conditional expressions

To evaluate a conditional expression "`if` e_1 `then` e_2 `else` e_3" in environment E, we do the following

1. evaluate e_1 in E to obtain a value v_1
2. if v_1 is the boolean `true`, then evaluate evaluate e_2 in E and return the result
3. otherwise (i.e., if v_1 is the boolean `false`), evaluate e_3 in E and return the result

Evaluating value definitions

To evaluate the `let` expression

$$\texttt{let x = } e_1 \texttt{ in } e$$

in environment E, we do the following

1. evaluate e_1 in E to obtain a value v_1
2. augment E with the binding $x \rightarrow v$ to obtain an environment E_1
3. evaluate e in E_1

Example of evaluating `let`

Suppose the current environment E looks like

`foo`	$\rightarrow$	*3*
...		

and we want to evaluate

```
let x = 4*foo in
x*x + x
```

We first evaluate `4*foo` in E to get 12. Then we create a new environment E_1 by extending E with the binding $x \rightarrow 12$:

`x`	$\rightarrow$	12
`foo`	$\rightarrow$	3
...		

Now we evaluate `x*x + x` in E_1.

(Note that E_1 is *only* used to evaluate `x*x + x`.)

Evaluating functions

Recall that a definition of a function has the form:

```
let rec f x = e1 in
e
```

The variable `x` is the *parameter* to the function and e_1 is an expression called the *body* of the function, which will usually contain occurrences of the parameter `x` and the function `f`. Also, the expression e will usually make use of `f`. For example:

```
let rec f x = x * x in
f(3) + f(4)
```

We need to explain

- how the *definition* of `f` is evaluated so that we can make use of it in e, and
- how *applications* of `f` (such as `f(3)` and `f(4)`) are evaluated.

Evaluating function definitions

To evaluate `let rec f x =` e_1 `in` e in an environment E, we

1. extend E with a binding that associates `f` with its parameter `x` and its body e_1:

   E_1 = | `f` $\rightarrow$ $\langle$`x`, $e_1\rangle$ |
   |---|
   | ... |

   (We shall refer to this environment—the environment containing the binding of `f` and everything below it—as the *definition environment* of `f`.)

2. Evaluate e in the environment E_1

Evaluating function applications

We now have to describe how to evaluate an application of `f` such as `f(3+y)` in an environment `E`. (Note that `E` may not be just the definition environment of `f`: other `let` bindings may have added more entries since `f` was defined.)

Here are the rules for evaluating an application $\texttt{f}(e)$ in an environment `E`.

1. Evaluate e in `E` to get a value v.
2. Look up `f` in `E` to find its binding $\langle \texttt{x}, e_1 \rangle$.
3. Augment the definition environment of `f` with $\texttt{x} \rightarrow v$ to get an environment $\texttt{E}_1$
4. Evaluate e_1 in $\texttt{E}_1$ and return the result

Note, in step 3, that we augment just the definition environment of `f`, not the whole environment `E`, to form $\texttt{E}_1$. This is crucial!

Function application – a simple example

Suppose we want to evaluate:

```
let rec f x = x + x in
f(2+7) + f(5)
```

We first extend the current environment to get the definition environment of `f`:

f	$\rightarrow$	$\langle$x,x+x$\rangle$
...		

Next we need to evaluate `f(2+7)` in this environment. We evaluate `2+7` to get `9`, extend the environment with $\texttt{x} \rightarrow 9$,

x	$\rightarrow$	9
f	$\rightarrow$	$\langle$x,x+x$\rangle$
...		

and finish by evaluating `x+x`—the body of `f`—in this environment, yielding 18.

Function evaluation – a more interesting example

Consider the following nested `let`-expression in Core ML:

```
let y = 0 in
let rec f x = x + y in
let y = 1 in
f(3)
```

Note that `y` is bound, used to define `f`, and then a new binding for y is created before `f` is called. When we evaluate the body of `f`, the *earlier* binding of `y` (to `0`) should be visible.

Starting at the top we make three extensions to our initial environment to get:

y	$\rightarrow$	1
f	$\rightarrow$	⟨x, x+y⟩
y	$\rightarrow$	0
...		

We now evaluate the application `f(3)`. We first evaluate the argument in this environment. Not much to do here – the result is `3`.

Now we extend the definition environment of `f` by binding the parameter of `f` to `3`.

x	$\rightarrow$	3
f	$\rightarrow$	⟨x, x+y⟩
y	$\rightarrow$	0
...		

In this environment, we evaluate the body of `f`, `x+y`, yielding `3`.

One more variation

As a more subtle example, consider the evaluation of:

```
let y = 0 in
let rec f x = x + y in
let y = 1 in
f(3+y)
```

We construct the same extended environment as before:

y	$\rightarrow$	1
f	$\rightarrow$	$\langle$x, x+y$\rangle$
y	$\rightarrow$	0
...		

Next we evaluate the argument of the application `f(3+y)` to get 4.

We then proceed as before, binding the parameter of `f` to 4 and evaluating the body of `f` in the definition environment of `f` augmented with this binding. That is, we evaluate `x + y` in the environment

x	$\rightarrow$	4
f	$\rightarrow$	$\langle$x, x+y$\rangle$
y	$\rightarrow$	0
...		

to get 4.

Implementing Evaluation

We have now given a careful description, in English, of how any Core ML program can be evaluated to produce a result. Our task for the remainder of this handout will be to turn this description into an OCaml *program* for evaluating Core ML expressions.

This exercise is interesting for several reasons:

- It gives us a *completely precise* definition of what Core ML programs "mean"
- It sharpens our intuitions about how programs evaluate (in Core ML, full OCaml, or any other language)
- It introduces several key ideas needed for implementing a full-scale interpreter or compiler (for any programming language)

Arithmetic/boolean expressions

To get a feeling for how this is going to work, let's begin with an even simpler subset of Core ML containing just the numbers, booleans, and arithmetic and relational expressions. For the moment, then, an *expression* can have any of the following forms:

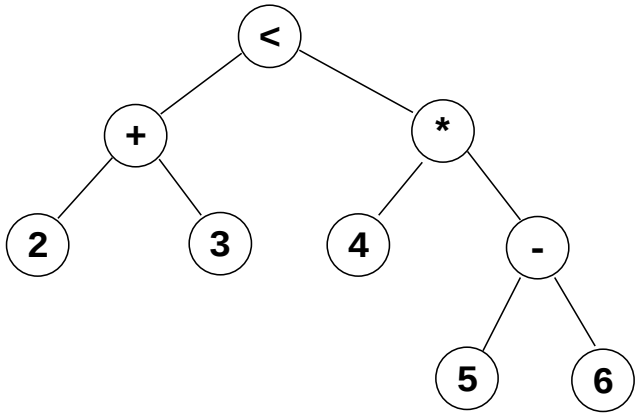
- an *integer constant* `0`, `1`, `2`, etc.
- a *boolean constant* `true` or `false`
- a *binary operation* e_1`+`e_2, e_1`-`e_2, e_1`*`e_2, or e_1`<`e_2, where e_1 and e_2 are expressions

(Note that we've combined the "arithmetic expressions" and "boolean expressions" from the earlier definition into one category of "binary operations.")

An expression like

```
(2 + 3) < (4 * (5 - 6))
```

can be thought of as a tree, where the leaves are constants and the interior nodes are arithmetic/boolean operators:



This "tree-structured" view of expressions leads directly to a datatype definition:

```
# type exp =
      IntConst of int
    | BoolConst of bool
    | OpExp of exp * string * exp;;
```

For example, the expression `(2 + 3) < (4 * (5 - 6))` is represented as follows:

```
# let test1 =
    OpExp(OpExp(IntConst(2), "+", IntConst(3)),
          "<",
          OpExp(IntConst(4),
                "*",
                OpExp(IntConst(5), "-", IntConst(6))));;
```

What expression is represented like this?

```
OpExp(IntConst(1),
      "+",
      OpExp(OpExp(OpExp(IntConst(2),"-",IntConst(3)),
                  "*",
                  IntConst(4)),
            "+",
            IntConst(5)))
```


To evaluate expressions, we also need a datatype for representing the possible results of evaluation. For the present language of arithmetic/boolean expressions, this datatype is very simple: the result of evaluation is either a number or a boolean:

```
# type value =
      IntVal of int
    | BoolVal of bool;;
```

The *evaluation function* `eval` is now straightforward:

```
# let rec eval e =
    match e with
      IntConst(i) -> IntVal(i)
    | BoolConst(b) -> BoolVal(b)
    | OpExp(e1,opname,e2) ->
        let v1 = eval e1 in
        let v2 = eval e2 in
        (match (opname,v1,v2) with
           ("+",IntVal(i1),IntVal(i2)) -> IntVal(i1+i2)
         | ("-",IntVal(i1),IntVal(i2)) -> IntVal(i1-i2)
         | ("*",IntVal(i1),IntVal(i2)) -> IntVal(i1*i2)
         | ("<",IntVal(i1),IntVal(i2)) -> BoolVal(i1<i2));;

# eval (OpExp(IntConst(77),"+",IntConst(88)));;
- : value = IntVal 165
```



```
# eval (IntConst(77));;
- : value = IntVal 77

# eval test1;;
- : value = BoolVal false
```

But:

```
# eval (OpExp(IntConst(77), "<", BoolConst(true)));;
Uncaught exception: Match_failure("", 200, 446)
```

Instead of allowing the nonexhaustive match of in `eval` to fail at runtime, it's better to define our own exception and raise it explicitly:

```
# exception EvaluationError;;

# let rec eval e =
    match e with
      IntConst(i) -> IntVal(i)
    | BoolConst(b) -> BoolVal(b)
    | OpExp(e1,opname,e2) ->
        let v1 = eval e1 in
        let v2 = eval e2 in
        (match (opname,v1,v2) with
           ("+",IntVal(i1),IntVal(i2)) -> IntVal(i1+i2)
         | ("-",IntVal(i1),IntVal(i2)) -> IntVal(i1-i2)
         | ("*",IntVal(i1),IntVal(i2)) -> IntVal(i1*i2)
         | ("<",IntVal(i1),IntVal(i2)) -> BoolVal(i1<i2)
         | _ -> raise EvaluationError);;
```


Aside: parsing

Obviously, typing long inputs like

```
# let test1 =
    OpExp(OpExp(IntConst(2), "+", IntConst(3)),
          "<",
          OpExp(IntConst(4),
                "*",
                OpExp(IntConst(5), "-", IntConst(6))));;
```

would not be particularly convenient if we had a lot of expressions we needed to evaluate.

In practice, the evaluator would be used in combination with a *parser* whose job it is to take ordinary strings and turn them into expressions.

```
val parse : string -> exp = <fun>
```

The `parse` and `eval` functions can be used together to express evaluation in a much more pleasant way:

```
# let evalString(s) = eval(parse(s));;

# evalString("(2+3) < (4*(5-6))");;
- : value = BoolVal false
```

Parsing is a rich and elegant topic with deep mathematical foundations. We do not have time to do it justice in this course, but you'll see it in detail if you take a course on compilers later on.

Expressions with variables

The expressions of Core ML also include *variables*. We can add these to our datatype of expressions like this:

```
# type exp =
      IntConst of int
    | BoolConst of bool
    | OpExp of exp * string * exp
    | Var of string;;
```

For example, the expression `x+2` is represented as:

```
# OpExp(Var("x"), "+", IntConst(2));;
```

As we saw in the first few slides of this handout, when we consider expressions with variables, we need to evaluate with respect to an *environment* that provides values for the variables that an expression may contain.

[This would be a huge amount easier to explain if we had defined `environment` as a `list` of bindings.]

```
# type environment =
      EmptyEnv
    | ValBinding of string * value * environment;;
```

For example, the environment `env_xy` mapping `x` to 4 and `y` to 7...

x	$\rightarrow$	4
y	$\rightarrow$	7

...is represented like this:

```
# let env_y  = ValBinding("y", IntVal(7), EmptyEnv);;
# let env_xy = ValBinding("x", IntVal(4), env_y);;
```


A simple helper function allows us to look up a variable in an environment:

```
# exception UnboundVariable;;

# let rec lookup env x =
    match env with
      EmptyEnv -> raise UnboundVariable
    | ValBinding (y,v,rest) ->
        if x=y then v
        else lookup rest x;;
```

The evaluator is now extended to deal with variables as follows:

```
# let rec eval env e =
    match e with
      IntConst(i) -> IntVal(i)
    | BoolConst(b) -> BoolVal(b)
    | OpExp(e1,opname,e2) ->
        let v1 = eval env e1 in
        let v2 = eval env e2 in
        (match (opname,v1,v2) with
           ("+",IntVal(i1),IntVal(i2)) -> IntVal(i1+i2)
         | ("-",IntVal(i1),IntVal(i2)) -> IntVal(i1-i2)
         | ("*",IntVal(i1),IntVal(i2)) -> IntVal(i1*i2)
         | ("<",IntVal(i1),IntVal(i2)) -> BoolVal(i1<i2)
         | _ -> raise EvaluationError)
    | Var(x) -> lookup env x;;

# eval env_xy (OpExp(Var("x"), "+", IntConst(2)));;
- : value = IntVal 6
```


Adding `let` definitions

Having introduced environments, we have everything we need to deal with Core ML's `let`-definition form.

We extend our datatype of expressions like this:

```
# type exp =
      IntConst of int
    | BoolConst of bool
    | OpExp of exp * string * exp
    | Var of string
    | LetValExp of string * exp * exp;;
```

For example,

```
# let a_let_exp =
    LetValExp("x",
              IntConst(99),
              OpExp(Var("x"), "+", IntConst(2)));;
```

represents the expression `let x = 99 in x + 2`.

The extended evaluator is:

```
# let rec eval env e =
    match e with
      IntConst(i) -> IntVal(i)
    | BoolConst(b) -> BoolVal(b)
    | OpExp(e1,opname,e2) -> (* as before... *)
    | Var(x) -> lookup env x
    | LetValExp(x,e1,e2) ->
        let v1 = eval env e1 in
        let newenv = ValBinding (x, v1, env) in
        eval newenv e2;;
```

For example:

```
# eval env_xy a_let_exp;;
- : value = IntVal 101
```


Adding conditionals

The `if...then...else...` expressions of Core ML are also easy to add:

```
# type exp =
      ...
    | IfExp of exp * exp * exp;;

# let rec eval env e =
    match e with
      ...
    | IfExp(e1,e2,e3) ->
       (match eval env e1 with
          BoolVal(true) -> eval env e2
        | BoolVal(false) -> eval env e3
        | _  -> raise EvaluationError);;
```

Adding functions

Our final job is dealing with function definitions (`let rec f a = ... in ...`) and applications (`f(e)`).

The extension to our datatype of expressions is easy. Here is the complete datatype definition:

```
# type exp =
      IntConst of int
    | BoolConst of bool
    | OpExp of exp * string * exp
    | Var of string
    | LetValExp of string * exp * exp
    | IfExp of exp * exp * exp
    | LetRecFunExp of string * string * exp * exp
    | AppExp of string * exp;;
```


The factorial function in Core ML:

```
let rec f a = if a < 1 then 1 else a * (f(a-1)) in
f(5)
```

Represented using the datatype `exp`:

```
# let factprog =
    LetRecFunExp("f", "a",
              IfExp(OpExp(Var("a"), "<", IntConst(1)),
                    IntConst(1),
                    OpExp(Var("a"),
                          "*",
                           AppExp("f",
                                  OpExp(Var("a"), "-", IntConst(1))))),
              AppExp("f", IntConst(5)));;
```

Environments with function definitions

We saw on slide 18 that function definitions need to be stored in the environment in a special way:

f	$\rightarrow$	$\langle a, e_1 \rangle$
x	$\rightarrow$	4
y	$\rightarrow$	7

We therefore extend the datatype of environments with a new clause for function bindings:

```
# type environment =
      EmptyEnv
    | ValBinding of string * value * environment
    | RecFunBinding of string * string * exp * environment;;
```


A function definition `let rec f(a) = e1 in e2` is evaluated by adding a function binding for `f` to the environment and evaluating the body `e2` in this extended environment:

```
# let rec eval env e =
    match e with
      ...
    | LetValExp(x,e1,e2) ->
        let v1 = eval env e1 in
        let newenv = ValBinding (x, v1, env) in
        eval newenv e2
    | ...
    | LetRecFunExp(f,a,e1,e2) ->
        let newenv = RecFunBinding (f, a, e1, env) in
        eval newenv e2;;
```

Note the similarities and differences between the `LetValExp` and `LetRecFunExp` cases.

Application

Recall from slide 19 the rule for evaluating an application `f(e)` in an environment `env` (rephrased slightly here):

1. Evaluate `e` in `env` to get a value `v`.
2. Look up `f` in `env` to find
   (a) the name `a` of its bound variable
   (b) its body `e1`
   (c) its definition environement `defenv`
3. Augment `f`'s definition environment `defenv` with `a`→`v` to get an environment `newenv`
4. Evaluate `e1` in `newenv` and return the result

Note, in the third step, that we augment just the *definition environment* of `f`, not the original environment `env`, to form `newenv`.

Function Values

So, when we look up a variable that is bound to a function in the environment, we need to return three things: `a`, `e1`, and `defenv`:

```
# let rec lookup env x =
    match env with
      EmptyEnv -> raise UnboundVariable
    | ValBinding (y,v,rest) ->
        if x=y then v
        else lookup rest x
    | RecFunBinding(f,a,e1,rest) ->
        if x=f then ???
        else lookup rest x;;
```

The expression that we want to fill in for `???` should have type `value` (since that is what the other clause of `lookup` returns), and it should include `a`, `e1`, and `env` (since `env` is the definition environment of `f`).

We can accomplish this by extending the datatype of values with a new clause carrying just this information:

```
# type value =
      IntVal of int
    | BoolVal of bool
    | FunVal of string * exp * environment
```

A `FunVal` is a *value* representing (all the information we need to invoke) a *function*. Here is the completed `lookup` function:

```
# let rec lookup env x =
    match env with
      EmptyEnv -> raise UnboundVariable
    | ValBinding (y,v,rest) ->
        if x=y then v
        else lookup rest x
    | RecFunBinding(f,a,e1,rest) ->
        if x=f then FunVal(a,e1,env)
        else lookup rest x;;
```


Note, in passing, that the datatypes of values and environments are now *mutually recursive*: environments bind variables to values, while a `FunVal` includes a definition environment.

```
# type value =
      IntVal of int
    | BoolVal of bool
    | FunVal of string * exp * environment

  and environment =
      EmptyEnv
    | ValBinding of string * value * environment
    | RecFunBinding of string * string * exp * environment;;
```

The keyword `and` introducing the type `environment` tells OCaml to treat the two definitions as mutually recursive.

So, to evaluate an application `f(e)`, we

1. evaluate `e`, yielding a value `v`
2. look up `f` in the current environment, yielding a function value `FunVal(a,e1,defenv)`
3. extend `defenv` with the binding `a→v`, yielding a new environment `newenv`
4. evaluate the function body `e1` in `newenv`

Translating this into OCaml, we obtain:

```
# let rec eval env e =
    match e with
       ...
    | LetRecFunExp(f,a,e1,e2) ->
        let newenv = RecFunBinding (f, a, e1, env) in
        eval newenv e2
    | AppExp(f,e) =
        let v = eval env e in
        (match lookup env f with
           FunVal(x,e1,defenv) ->
             let newenv = ValBinding (x, v, defenv) in
             eval newenv e1
         | _ -> raise EvaluationError);;
```

For example:

```
# let funprog =
    LetRecFunExp("f", "x",
                 OpExp(Var("x"), "+", Var("x")),
                 AppExp("f", IntConst(5)));;

# eval EmptyEnv funprog;;
- : value = IntVal 10
```



```
# let factprog =
    LetRecFunExp("f", "x",
              IfExp(OpExp(Var("x"), "<", IntConst(1)),
                    IntConst(1),
                    OpExp(Var("x"),
                          "*",
                           AppExp("f",
                                  OpExp(Var("x"), "-", IntConst(1))))),
              AppExp("f", IntConst(5)));;

# eval EmptyEnv factprog;;
- : value = IntVal 120
```

Function parameters

Functions like `mapl` and `filter` take functions as their parameters. Here is a simpler (and sillier) example: a function `thrice` that takes a function `g` as argument, applies it three times to the constant `5`, and returns the result. Let's see how it evaluates.

```
let rec thrice g = g(g(g(5))) in
let rec add3 a = a+3 in
thrice(add3)
```


This program is represented like this:

```
# let triceprog =
    LetRecFunExp("thrice", "g",
              AppExp("g",
                     AppExp("g",
                            AppExp("g", IntConst(5)))),
              LetRecFunExp("add3", "a",
                           OpExp(Var("a"), "+", IntConst(3)),
                           AppExp("thrice", Var("add3"))));;

# eval EmptyEnv triceprog;;
- : value = IntVal 14
```

Note that, during evaluation, the `FunVal` corresponding to `add3` is built when it is first looked up (i.e., when `Var("add3")` is evaluated), not when it is later applied (i.e., when `g` is looked up inside of the body of `f`).