Programming Language Fundamentals (PLaF) Notes

v0.105 – December 6, 2020

Eduardo Bonelli

Contents

Preface

Chapter 1

A Calculator

We introduce a toy language called ARITH. In ARITH programs are simple arithmetic expressions. The objective of this chapter is to provide a gentle introduction to various concepts we will be developing later in these notes. These concepts include the syntax of a language (concrete syntax, abstract syntax, parsing) and evaluation/execution of programs in a language (interpreter, specifying an interpreter, implementing an interpreter, handling errors).

1.1 Syntax

1.1.1 Concrete Syntax

The grammar below specifies the concrete syntax of ARITH. It determines what expressions are syntactically correct ARITH programs. Each line is called a **production**. Expressions enclosed in angle brackets are called **non-terminals**. The grammar below only has two non-terminals, $\langle\text{Expression}\rangle$ and $\langle\text{Number}\rangle$. Among all non-terminals one singles out the so called **start non-terminal**. In our case, the start non-terminal is $\langle\text{Expression}\rangle$. Symbols that appear to the right of "::=" and that are not non-terminals are called **terminals**. The grammar below has the following set of terminals: $\{-, /, (,)\}$. Note that we have not specified what terminals are generated by the $\langle\text{Number}\rangle$ non-terminal; we will assume these to be all integers.

$$
\begin{array}{lcl}
\langle\text{Expression}\rangle & ::= & \langle\text{Number}\rangle \\
\langle\text{Expression}\rangle & ::= & \langle\text{Expression}\rangle - \langle\text{Expression}\rangle \\
\langle\text{Expression}\rangle & ::= & \langle\text{Expression}\rangle / \langle\text{Expression}\rangle \\
\langle\text{Expression}\rangle & ::= & (\langle\text{Expression}\rangle)
\end{array}
$$

For the sake of simplicity, our language ARITH only supports two arithmetic operations, subtraction and division. We will later add others.

1.1.2 Abstract Syntax

Given a string of terminals `s`, a parser will produce an abstract syntax tree (AST) for `s` if it is a syntactically correct ARITH expression, otherwise it will fail with an error message. The AST is a value of type `expr`:

```
type expr =
  | Int of int
  | Sub of expr*expr
  | Div of expr*expr
```

ast.ml

1.2 Interpreter

An interpreter[1] is a process that, given an expression, produces the result of its evaluation. The implementation of interpreters in these notes will be developed in two steps, first we specify the interpreter and then we implement it proper.

- Specification of the interpreter. This includes a precise description of the possible results which a program can evaluate to, evaluation judgements that state what result a program evaluates to and a set of evaluation rules that define the meaning of evaluation judgements by describing the behavior of each construct in the language.
- Implementation of the interpreter. Using the evaluation rules of the specification as a guide, an implementation is presented. The time invested in producing the evaluation rules in the previous step, betters our understanding of the interpreter's behavior and hence diminishes the chances of introducing errors when it is implemented.

To illustrate this approach, we next specify an interpreter for ARITH and then implement it.

1.2.1 Specification

We begin by stating the possible results that can arise out of the evaluation of programs in ARITH. For now, we fix the set of **results** to be the integers $\mathbb{Z}$ since evaluation of ARITH programs produce integers:

$$\mathbb{R} := \mathbb{Z}$$

The ":=" symbol is used for definitional equality, meaning here that $\mathbb{R}$ is "defined to be" the set $\mathbb{Z}$. We continue with the specification of the interpreter for ARITH by introducing evaluation judgements. **Evaluation judgements** are expressions of the form:

$$\mathtt{e} \Downarrow n$$

where e is an expression in ARITH in abstract syntax and n is a result (*i.e.* n is an integer). It is read as, "expression e evaluates to the integer n". The meaning of $\mathtt{e} \Downarrow n$ is established via so called **evaluation rules**. The preliminary set of evaluation rules of ARITH are as follows:

$$\frac{}{\mathtt{Int(n)} \Downarrow n}\ \text{EInt}$$

$$\frac{\mathtt{e1} \Downarrow m \quad \mathtt{e2} \Downarrow n \quad p = m - n}{\mathtt{Sub(e1,e2)} \Downarrow p}\ \text{ESub} \qquad \frac{\mathtt{e1} \Downarrow m \quad \mathtt{e2} \Downarrow n \quad p = m/n}{\mathtt{Div(e1,e2)} \Downarrow p}\ \text{EDiv}$$

[1] We will use the words "interpreter" and "evaluator" interchangeably.

Judgements above the horizontal line in an evaluation rule are called **hypotheses** and the one below is called the **conclusion**. A rule that does not have hypotheses is called an **axiom rule** or just axiom. Hypotheses of an evaluation rule are read from left to right. In particular, evaluation of the arguments of all arithmetic operations proceeds from left to right. It could have been stated in the opposite order from right-to-left and, at this point in time, does not make much of a difference[2]. A **derivation tree** is a tree of evaluation judgements whose nodes are instances of evaluation rules and, moreover, whose leaves are instances of axioms. A judgement $\texttt{e} \Downarrow n$ is **derivable or holds** if there is a derivation tree with $\texttt{e} \Downarrow n$ as its root.

Example 1.2.1. *For example* `Sub(Int 3,Int 1)` $\Downarrow 2$ *is a derivable evaluation judgement:*

$$\dfrac{\dfrac{}{\texttt{Int(3)} \Downarrow 3}\,\text{EInt} \qquad \dfrac{}{\texttt{Int(1)} \Downarrow 1}\,\text{EInt} \qquad 2 = 3 - 1}{\texttt{Sub(Int 3,Int 1)} \Downarrow 2}\,\text{ESub}$$

The evaluation judgement `Sub(Div(Int 8,Int 2),Int 1)` $\Downarrow 3$ *is also derivable.*

We are not quite done with the task of specifying our interpreter since not all expressions in ARITH return numbers. For example, `Div(Int 2,Int 0)` does not evaluate to any number. Rather it should evaluate to an error. Thus, the above mentioned evaluation rules are incomplete since there are situations that are left unspecified. It is important for have a complete set of rules so that when we implement the interpreter there are no ambiguities. Moreover, we must revisit our notion of result since it should include an error as a possible outcome. The previously introduced evaluation judgements are thus revisited below. The final form that evaluation judgements take for ARITH are:

$$\texttt{e} \Downarrow r$$

where r denotes a result of the computation $r \in \mathbb{R}$. A result is either an integer or a special element $error$. In other words, we set $\mathbb{R} := \mathbb{Z} \cup \{error\}$. The subset of results that are integers are called expressed values: it is the name given to non-error results of evaluation.

The evaluation rules defining this new judgement, and therefore the evaluation rules for ARITH, are those presented in Figure 1.1, where $m, n, p \in \mathbb{Z}$. The first three rules are the ones already presented above. Rules ESubErr1, ESubErr2, EDivErr1, and EDivErr2 state how error propagation takes place. The last one introduces a new error, namely division by zero. Moving forward, and for the sake of brevity, we will not be specifying the error propagation rules when specifying the interpreter. We will only be presenting the error introduction rules.

We next address the implementation of an interpreter for ARITH. We will do so in two attempts, first a preliminary attempt and then a final one. The preliminary attempt has various drawbacks that we will point out along the way but has the virtue of serving as a convenient stepping stone towards the final one.

[2]But later in our development, when evaluation of expressions can cause certain effects (such as modifying mutable data structures), this difference will become relevant.

$$\frac{}{\texttt{Int(n)} \Downarrow n}\ \text{EInt}$$

$$\frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow n \quad p = m - n}{\texttt{Sub(e1,e2)} \Downarrow p}\ \text{ESub} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow n \quad p = m / n}{\texttt{Div(e1,e2)} \Downarrow p}\ \text{EDiv}$$

$$\frac{\texttt{e1} \Downarrow \mathit{error}}{\texttt{Sub(e1,e2)} \Downarrow \mathit{error}}\ \text{ESubErr1} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow \mathit{error}}{\texttt{Sub(e1,e2)} \Downarrow \mathit{error}}\ \text{ESubErr2}$$

$$\frac{\texttt{e1} \Downarrow \mathit{error}}{\texttt{Div(e1,e2)} \Downarrow \mathit{error}}\ \text{EDivErr1} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow \mathit{error}}{\texttt{Div(e1,e2)} \Downarrow \mathit{error}}\ \text{EDivErr2} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow 0}{\texttt{Div(e1,e2)} \Downarrow \mathit{error}}\ \text{EDivErr3}$$

Figure 1.1: Evaluation rules for ARITH

A summary of some important concepts we have covered are listed below. Make sure you lookup them up:

- Result
- Expressed Value
- Evaluation Judgement
- Evaluation Rules
- Derivation Tree
- Derivable Evaluation Judgements

1.2.2 Implementation

In order to use the evaluation rules as a guideline for our implementation we first need to model both components of evaluation judgements in OCaml, namely expression e and result r in $\texttt{e} \Downarrow r$. The former is already expressed in abstract syntax, which we encoded as the algebraic data type `expr` in OCaml. So that item has already been addressed. As for the latter, since it denotes either an integer or an error, we will model it in OCaml using the following type[3]:

```
type 'a result = Ok of 'a | Error of string
```

For example, the type `int result` may be read as a type that states that "the result of the evaluator is an integer or an error". Likewise, `bool result` may be read as a type that states that "the result of the evaluator is a boolean or an error". In summary, a result can either be a meaningful value of type `'a` prefixed with the constructor `Ok`, or else an error with an argument of type `string` prefixed with an `Error` constructor. For example, `Ok 3` has type `int result`.

[3] OCaml has a built-in type `type ('a,'b) result = Ok of 'a | Error of 'b`. We could have used this type but it is slightly more general than necessary since our errors will always be accompanied by a string argument rather than different types of arguments.

```
let rec eval_expr : expr -> int result =
  fun e ->
  match e with
  | Int(n) -> Ok n
  | Sub(e1,e2) ->
    (match eval_expr e1 with
     | Error s -> Error s
     | Ok m -> (match eval_expr e2 with
                | Error s -> Error s
                | Ok n -> Ok (m-n)))
  | Div(e1,e2) ->
    (match eval_expr e1 with
     | Error s -> Error s
     | Ok m -> (match eval_expr e2 with
                | Error s -> Error s
                | Ok n -> if n==0
                          then Error "Division by zero"
                          else Ok (m/n)))
```

interp.ml

Figure 1.2: Preliminary Interpeter for ARITH

In a type expression such as `int result`, we say `int` is a "type" and `int result` is a "type". But we refer to `result` as a **type constructor** since, given a type `'a`, it constructs a type `'a result`.

We can now proceed with an implementation of an interpreter for ARITH following the evaluation rules as close as possible. If we call our evaluator function `eval_expr`, its type can be expressed as follows, indicating that evaluation consumes an expression and returns either an integer or an error with a string description:

```
eval_expr : exp -> int result
```

The code is given in Figure 1.2. This figure is an example of a code listing. We occasionally add an indication to code listings as to where the snippet of code resides. For example, in this case it resides in file `interp.ml`. Notice that `eval_expr` is a recursive function over the structure of expressions in abstract syntax (*i.e.* values of type `expr`). In the first clause, `Int(n)`, it simply returns `Ok n`. Note that returning just `n` would be incorrect since out interpreter must return a value of type `int result`, not of type `int`. In the clause for `Sub(e1,e2)`, the `match` keyword forces evaluation of `e1` before `e2`, as indicated by the evaluation rules[4]. A similar comment applies to the other arithmetic operation. One notices, too, that a substantial amount of code checks for errors and then propagates them. Notice too that, in the `Div` case, in addition to propagating errors resulting from evaluating its arguments `e1` and `e2`, it generates a new one if the denominator is 0. This is in accordance with the evaluation rule EDivErr3. The only error modeled in ARITH is division by zero. An example of error propagation takes place in an ARITH expression such a `Add(Div(Int 1,Int 0),e)`, where `e` can be any expression. Here the expression `e` is never evaluated since evaluation of `Div(Int 1,Int 0)` produces `Error "Division by zero"`, hence `e` is ignored and `Error "Division by zero"` is immediately produced as the final result of evaluation of the entire expression.

[4] OCaml evaluates arguments from right to left.

```
let return : 'a -> 'a result =
  fun v -> Ok v

let error : string -> 'a result =
  fun s -> Error s

let (>>=) : 'a result -> ('a -> 'b result) -> 'b result =
  fun c f ->
  match c with
  | Error s -> Error s
  | Ok v -> f v
```
ds.ml

Figure 1.3: The Error Monad

Implementation: Final

Although certainly necessary, there is no interesting computational content in error propagation. It would be best to have it be handled behind the scenes, by appropriate error propagation helper functions.

We next introduce three helper functions for this purpose: `return`, `error` and `>>=` (pronounced "bind"). The code for these functions is given in Figure 1.3. The `return` function simply returns its argument inside an `Ok` constructor and may thus be seen as producing a non-error result. A similar comment applies to `error`: given a string it produces an error result by simply prefixing the string with the `Error` constructor. The infix operator `>>=` is called bind and is left associative[5]. It takes an expression `c` that returns a result (*i.e.* a non-error value or an error value), and if `c` returns an error, propagates it. Otherwise, if it evaluates to `Ok v`, for some `v`, then it passes `v` on to `f` by evaluating `f v`.

An alternative description of these helper functions is as follows. Let us dub expressions of type `int result`, **structured programs** (we could have taken the more general type `'a result` as our notion of structured programs, but the latter will suffice for our explanation). Structured programs may be seen as programs that, apart from producing an integer as end product, can manipulate additional structure such as error handling, state, non-determinism, etc. In our particular case, a structured program handles errors as additional structure. Under this light, we can describe the helper functions as follows:

- `return` may be seen as a function that creates a (trivial) structured program that returns an integer (*i.e.* non-error) result.
- `error` may be seen as a function that creates a (trivial) structured program that returns an error result.
- `>>=` may be seen as a function that models composition of a structured program `c` and a function `f` that, given an integer, produces a structured program. Allowing `f` to be a function that produces a structured program, rather than a structured program itself, allows `f` to use the result of evaluating the first argument.

[5] The precedence and associativity of user-defined infix/prefix operators may be consulted here: https://caml.inria.fr/pub/docs/manual-caml-light/node4.9.html

Let us rewrite our interpreter for our simple expression language using these helper functions.

```
let rec eval_expr : expr -> int result =
  fun e ->
  match e with
  | Int(n) -> return n
  | Sub(e1,e2) ->
    eval_expr e1 >>= (fun n1 ->
    eval_expr e2 >>= (fun n2 ->
    return (n1-n2)))
  | Div(e1,e2) ->
    eval_expr e1 >>= (fun n1 ->
    eval_expr e2 >>= (fun n2 ->
    if n2==0
    then error "Division by zero"
    else return (n1/n2)))
```

Consider the code for the `Sub(e1,e2)` case. Notice how if `eval_expr e1` produces an error result, say `Error "Division by zero"` because `e1` had a divison by zero, then `>>=` simply ignores its second argument, namely the expression `(fun n1 -> eval_expr e2 >>= (fun n2 -> return (n1+n2)))`, and returns the error result `Error "Division by zero"` immediately as the final result of the evaluation, thus effectively propagating the error.

In fact, we can further simplify this code by dropping superfluous parenthesis. This leads to our final evaluator for ARITH expressions.

```
let rec eval_expr : expr -> int result =
  fun e ->
  match e with
  | Int(n) -> return n
  | Sub(e1,e2) ->
    eval_expr e1 >>= fun n1 ->
    eval_expr e2 >>= fun n2 ->
    return (n1-n2)
  | Div(e1,e2) ->
    eval_expr e1 >>= fun n1 ->
    eval_expr e2 >>= fun n2 ->
    if n2==0
    then error "Division by zero"
    else return (n1/n2)
```

Some additional observations on the behavior of the error handling operations:

$$
\begin{array}{lcl}
\texttt{return e >>= f} & \simeq & \texttt{f e} \\
\texttt{m >>= return} & \simeq & \texttt{m} \\
\texttt{(m >>= f) >>= g} & \simeq & \texttt{m >>= (fun x -> f x >>= g)} \\
\texttt{error e >>= f} & \simeq & \texttt{error e}
\end{array}
$$

The symbol $\simeq$ above means that the left and right hand sides of these equations behave the same way.

The `result` type, together with the operations `return`, `error` and `>>=` is called an **Error Monad**. Monads are well-known in pure functional programming languages like Haskell, where they allow to handle side-effects behind the scenes without compromising equational reasoning (see the equations presented above). However, they are also important in non-pure functional languages, like OCaml, where they are a means to better structure one's code, as we have seen from our use of it here.

Chapter 2

Simple Functional Languages

2.1 LET

2.1.1 Concrete Syntax

$$
\begin{array}{lcl}
\langle \text{Expression} \rangle & ::= & \langle \text{Number} \rangle \\
\langle \text{Expression} \rangle & ::= & \langle \text{Identifier} \rangle \\
\langle \text{Expression} \rangle & ::= & \langle \text{Expression} \rangle \langle \text{BOp} \rangle \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & \texttt{zero?}(\langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{if}\ \langle \text{Expression} \rangle\ \texttt{then}\ \langle \text{Expression} \rangle\ \texttt{else}\ \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & \texttt{let}\ \langle \text{Identifier} \rangle = \langle \text{Expression} \rangle\ \texttt{in}\ \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & (\langle \text{Expression} \rangle) \\
 & & \\
\langle \text{BOp} \rangle & ::= & + \mid - \mid * \mid /
\end{array}
$$

2.1.2 Abstract Syntax

```
type expr =
  | Int of int
  | Var of string
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Let of string*expr*expr
```

2.1.3 Environments

Consider the LET expression `x+2`. We cannot determine the result of its evaluation because, in a sense, it is incomplete. Indeed, unless we are given the value assigned to the identifier `x`, we cannot determine whether evaluation of `x+2` should result in an error (if say, `x` held the value `true`[1]), or a number such as `4` (if, say, `x` held the value `2`). Therefore, evaluation of expressions in LET require an assignment of values to identifiers. These assignments are called environments. The interpreters developed in these notes are therefore known as environment-based interpreters as opposed to substitution-based interpreters. In the latter values are substituted directly into the expressions rather than recording, and then looking them up, in environments.

An **environment** is a partial function from the set of identifiers to the set of expressed values. **Expressed values**, denoted $\mathbb{EV}$, are the set of values that are not errors that we can get from evaluating expressions. In ARITH the only expressed values are the integers. In LET they are the integers and the booleans:

$$\mathbb{EV} := \mathbb{Z} \cup \mathbb{B}$$

where $\mathbb{B} := \{true, false\}$. If $\mathbb{ID}$ denotes the set of all identifiers, then we can define the set of all environments $\mathbb{ENV}$ as follows.

$$\mathbb{ENV} := \mathbb{ID} \rightharpoonup \mathbb{EV}$$

We use letters ρ and ρ' to denote environments. For example, $\rho = \{x := 1, y := 2, z := true\}$ is an environment that assigns 1 to x, 2 to y and $true$ to z. We write $\rho(id)$ for the value associated to the identifier id. For example, $\rho(x)$ is 1.

2.1.4 Interpreter

Specification

As you might recall from our presentation of ARITH, evaluation of an ARITH expression produces a result. A result could either be an integer or an error. We can still get an error from evaluating a LET expression since ARITH expressions are included in LET expressions. However, if there is no error, then in LET we can either get an integer or a boolean.The set of results for LET is thus:

$$\mathbb{R} := \mathbb{EV} \cup \{error\}$$

where $\mathbb{EV}$ was updated above during our discussion on environments. Evaluation judgements for LET include an environment:

$$\texttt{e}, \rho \Downarrow r$$

It should be read as follows, "evaluation of expression e under environment ρ, produces result r". The rules defining this judgement are presented in Figure 2.1. The last four rules handle error generation, the first eight handle standard (*i.e.* non-error) evaluation. The rules for error propagation are omitted. Rule EInt is the same as in ARITH, except that the judgement has an environment (which plays no role in this rule). Rule EVar performs lookup in the current environment. The related rule EVarErr models the error resulting from lookup failing to find the identifier in the environment. The rules for addition, subtraction and multiplication are similar as the one for division and omitted. The notation $\rho \oplus \{\texttt{id} := w\}$ used in ELet stands for the environment that maps expressed value w to identifier `id` and behaves as ρ on all other identifiers.

[1]We do not have booleans in ARITH but we will in LET.

$$\frac{}{\texttt{Int(n)}, \rho \Downarrow n}\text{EInt} \qquad \frac{\rho(\texttt{id}) = v}{\texttt{Var(id)}, \rho \Downarrow v}\text{EVar}$$

$$\frac{\texttt{e1} \Downarrow m \quad \texttt{e2}, \rho \Downarrow n \quad p = m/n}{\texttt{Div(e1,e2)}, \rho \Downarrow p}\text{EDiv}$$

$$\frac{\texttt{e}, \rho \Downarrow v \quad v = 0}{\texttt{IsZero(e)}, \rho \Downarrow \mathit{true}}\text{EIZTrue} \qquad \frac{\texttt{e}, \rho \Downarrow v \quad v \neq 0}{\texttt{IsZero(e)}, \rho \Downarrow \mathit{false}}\text{EIZFalse}$$

$$\frac{\texttt{e1}, \rho \Downarrow \mathit{true} \quad \texttt{e2}, \rho \Downarrow v}{\texttt{ITE(e1,e2,e3)}, \rho \Downarrow v}\text{EITETrue} \qquad \frac{\texttt{e1}, \rho \Downarrow \mathit{false} \quad \texttt{e3}, \rho \Downarrow v}{\texttt{ITE(e1,e2,e3)}, \rho \Downarrow v}\text{EITEFalse}$$

$$\frac{\texttt{e1}, \rho \Downarrow w \quad \texttt{e2}, \rho \oplus \{\texttt{id} := w\} \Downarrow v}{\texttt{Let(id,e1,e2)}, \rho \Downarrow v}\text{ELet}$$

$$\frac{\texttt{id} \notin \mathsf{dom}(\rho)}{\texttt{Var(id)}, \rho \Downarrow \mathit{error}}\text{EVarErr} \qquad \frac{\texttt{e1}, \rho \Downarrow m \quad \texttt{e2}, \rho \Downarrow 0}{\texttt{Div(e1,e2)}, \rho \Downarrow \mathit{error}}\text{EDivErr}$$

$$\frac{\texttt{e}, \rho \Downarrow v \quad v \notin \mathbb{Z}}{\texttt{IsZero(e)}, \rho \Downarrow \mathit{error}}\text{EIZErr} \qquad \frac{\texttt{e1}, \rho \Downarrow v \quad v \notin \mathbb{B}}{\texttt{ITE(e1,e2,e3)}, \rho \Downarrow \mathit{error}}\text{EITEErr}$$

Figure 2.1: Evaluation Semantics for LET

Implementation: Attempt I

Before implementing the evaluator we must first implement expressed values and environments. Expressed values can be naturally described using algebraic data types. Environments can be modeled in various ways in OCaml: as functions, as association lists, as hash tables, as algebraic data types, etc. Due to its simplicity we follow the last of these.

```
type exp_val =
  | NumVal of int
  | BoolVal of bool

type env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
```
ds.ml

Operations on environments are:

```
let empty_env : unit -> env =
  fun () -> EmptyEnv

let extend_env : env -> string -> exp_val -> env =
  fun env id v -> ExtendEnv(id,v,env)

let rec apply_env : string -> env -> exp_val result =
  fun id env ->
  match env with
  | EmptyEnv -> error (id^" not found!")
  | ExtendEnv(v,ev,tail) ->
    if id=v
    then return ev
    else apply_env id tail
```
ds.ml

Notice that `apply_env en id` has `exp_val result` as return type because it returns an error if `id` is not found in the environment `en`. However, if there is an expressed value associated to `id` in `en`, then that will be returned (wrapped with an `Ok` constructor).

Next we implement an interpreter for LET by following the evaluation rules of Figure 2.1 closely. Indeed, the evaluation rules shall serve as a specification for, and thus guide, our implementation. The code itself is given in Figure 2.2. The case for `Var(id)` simply invokes `apply_env` to look up the expressed value associated to the identifier `id` in the environment `env`. The case for `Div(e1,e2)` makes use of an auxiliary operation `int_of_numVal`, explained below.

```
let int_of_numVal : exp_val -> int result = function
  | NumVal n -> return n
  | _ -> error "Expected a number!"
```
ds.ml

Evaluation of `e1` in `Div(e1,e2)` could produce an expressed value other than a number (*i.e.* other than a `NumVal`). The function `int_of_numVal` checks to see whether its argument is a `NumVal` or not, returning a result that consists of an error, if it is not, or else the number itself (without the `NumVal` tag). This number is then bound to the variable `n1`. A similar description determines the value of `n2`. Finally, if `n2` is zero, an error is returned, otherwise the desired quotient is produced as a result.

```
let rec eval_expr : expr -> env -> exp_val result =
  fun e en ->
  match e with
  | Int(n) -> return (NumVal n)
  | Var(id) -> apply_env id en
  | Div(e1,e2) ->  (* Add, Sub and Mul are similar and omitted *)
    eval_expr e1 en >>=
    int_of_numVal >>= fun n1 ->
    eval_expr e2 en >>=
    int_of_numVal >>= fun n2 ->
    if n2==0
    then error "Division by zero"
    else return (NumVal (n1/n2))
  | IsZero(e) ->
    eval_expr e en >>=
    int_of_numVal >>= fun n ->
    return (BoolVal (n = 0))
  | ITE(e1,e2,e3) ->
    eval_expr e1 en >>=
    bool_of_boolVal >>= fun b ->
    if b
    then eval_expr e2 en
    else eval_expr e3 en
  | Let(id,def,body) ->
    eval_expr def en >>= fun ev ->
    eval_expr body (extend_env en id ev)
```
interp.ml

Figure 2.2: Preliminary Interpeter for LET

The code for the other binary operators, for the zero predicate and for the conditional are similar. The case for `ITE` uses a similar helper function `bool_of_boolVal`. The last case, `Let`, evaluates the definition and then extends the environment appropriately before evaluating the body. Notice how local scoping is implemented by adding a new entry into the environment.

The top level function for the interpreter is called `interp`. It parses the string argument, evaluates producing a function `c` that awaits an environment, and then feeds that function the empty environment `EmptyEnv`.

```
let interp (s:string) : exp_val result =
 let c = s |> parse |> eval_expr
 in c EmptyEnv
```
interp.ml

Here is an example run of our interpreter[2]:

```
utop # interp "
let x=2
in let y=3
in x+y";;
- : exp_val Ds.result = Ok (NumVal 5)

utop # interp "
```

[2]Negative numbers must be placed between parenthesis. For example, `interp "(-7)"` rather than `interp "-7"`.

```
let x=2
in let y=0
in x+(x/y)";;
- : exp_val Ds.result = Error "Division by zero"
```

utop

In ARITH there was only one possible error that could be generated and then propagated, namely the division by zero error. In LET there are four possible errors that can be generated and propagated: division by zero, identifier not found, expected a number and expected a boolean.

Weaving Environments

Our code for LET seems well-structured and robust enough to be extensible to additional language features. Even so, we can perhaps take a step further. Notice that the environment is explicitly threaded around the entire program. Indeed, consider the following excerpt from Figure 2.2 and notice how the environment (highlighted) is passed on to each occurrence of `eval_expr`.

```
   let rec eval_expr : expr -> env -> exp_val result =
2    fun e en ->
     match e with
4    | ...
     | ITE(e1,e2,e3) ->
6      eval_expr e1 en >>=
       bool_of_boolVal >>= fun b ->
8      if b
       then eval_expr e2 en
10     else eval_expr e3 en
```

The reason en is passed on in each case above, is that all expressions `e1`, `e2` and `e3` are evaluated under that same environment. This occurs with other language constructs too such as `Add(e1,e2)`, `Div(e1,e2)`, `Sub(e1,e2)` and `Mul(e1,e2)`, where both `e1` and `e2` are evaluated under the environment `en`. An alternative would be to have the environment be passed around "behind the scenes", in the same way that error propagation is handled behind the scenes. The resulting code would look something like this, where all references to the environment have been removed, including the one on line 2:

```
   let rec eval_expr : expr -> env -> exp_val result =
2    fun e ->
     match e with
4    | ...
     | ITE(e1,e2,e3) ->
6      eval_expr e1 >>=
       bool_of_boolVal >>= fun b ->
8      if b
       then eval_expr e2
10     else eval_expr e3
```

Listing 2.3: Naive removal of environment arguments

We would still need to provide an environment since `eval_expr` expects both expression and environment. That would be done by `interp`:

```
let interp (s:string) : exp_val result =
  let c = s |> parse |> eval_expr
  in c EmptyEnv
```

Listing 2.4: Naive removal of environment arguments

Unfortunately, the resulting code in Listing 2.3 doesn't type-check. Let us take a closer look at the bind operator used in line 6:

```
eval_expr e1 >>= ...
```

Recall from Figure 1.3 that the type of `>>=` is `'a result -> ('a -> 'b result) -> 'b result`. The expression `eval_expr e1` in line 6, which is the first argument of `>>=`, is therefore expected to have type `'a result` (where `'a` can be any type, in particular `exp_val`). However, since we removed the environment argument it instead has type `env -> exp_val result`. Indeed, given that we are not explicitly supplying the environment any longer, then `eval_expr e1` now produces a:

> function that waits for the environment and then produces a result.

This means that bind now has to be able to compose "functions that wait for environments and produce a result" rather than composing "results". In other words, we have to put forward a new proposal for the type of bind:

Right now	`(>>=) : 'a result -> ('a -> 'b result) -> 'b result`
New proposal	`(>>=) : (env -> 'a result) -> ('a -> (env -> 'b result)) -> (env -> 'b result).`

Lets give the type `env -> 'a result` a name, so that we can improve legibility. Consider the new `ea_result` type constructor, read "environment abstracted result", defined by simply abstracting the type of environments over the standard result type:

```
type 'a ea_result = env -> 'a result
```

Now we can apply this abbreviated type and recast our table above as:

Right now	`(>>=) : 'a result -> ('a -> 'b result) -> 'b result`
New proposal	`(>>=) : 'a ea_result -> ('a -> 'b ea_result) -> 'b ea_result.`

Of course, we'll need to update the code for bind (and the other helper functions). We will do so shortly. The new type for our interpreter is now:

```
eval_expr : expr -> exp_val ea_result
```

Updating the helper functions. We must now update the helper functions. The new code for them is in Figure 2.5. Function `return v` used to return `Ok v`. But notice now how it returns a function that waits for an environment `env` and only then returns `Ok v`. It may perhaps result odd that the environment seems not to be used for anything. However, other helper functions will make use of it (for example, `>>=`). Note also how `>>=` now passes the environment argument `env` first to `c` and then to `f v`, thus effectively threading the environment for us. You may safely ignore `>>+` for now, we'll explain it later. Also, we have a new operation `run` that given an environment abstracted result, will feed it the empty environment and thus perform the computation itself resulting in either an ok value or an error value. It is essentially the same as Listing 2.4 except that, since this function will be placed in the file `interp.ml`, it is best to avoid using the names of the constructors for environments.

```
type 'a result = Ok of 'a | Error of string

type 'a ea_result = env -> 'a result

let return : 'a -> 'a ea_result =
  fun v ->
  fun env -> Ok v

let error : string -> 'a ea_result =
  fun s ->
  fun env -> Error s

let (>>=) : 'a ea_result -> ('a -> 'b ea_result) -> 'b ea_result =
  fun c f ->
  fun env ->
  match c env with
  | Error err -> Error err
  | Ok v -> f v env

let (>>+) : env ea_result -> 'a ea_result -> 'a ea_result =
  fun c d ->
  fun env ->
  match c env with
  | Error err -> Error err
  | Ok newenv -> d newenv

let run : 'a ea_result -> 'a result =
  fun c -> c EmptyEnv
```

ds.ml

Figure 2.5: The Reader and Error Monad Combined

```
let interp (e:string) : exp_val result =
  let c = e |> parse |> eval_expr
  in run c
```
interp.ml

The variable `c` is used as mnemonic for "computation" (also referred to as a "structured program") the program that results from evaluating the abstract syntax tree of `e`. The computation is executed by passing it the empty environment.

Implementation: Final

We next revisit our evaluator for LET, this time making use of our new environment abstracted result type. The code is given in Figure 2.6. We briefly comment on some of the variants.

The code for the `Int(n)` variant, remains unaltered:

```
  | Int(n) -> return (NumVal n)
```

Note, however, that `return (NumVal n)` now returns a function that given an environment, ignores it and simply returns `Ok (NumVal n)`.

The `Var(id)` variant is slightly different, it is missing the environment:

```
  | Var(id) -> apply_env id
```

Now `apply_env` is applied only to the argument `id`, thus producing an expression (through partial application) that waits for the second argument, namely the environment. This environment will be supplied when we run the computation (using `run`).

The variants `Div(e1,e2)`, `IsZero(e)` and `ITE(e1,e2,e3)` are as in Figure 2.2 except that the environment argument has been dropped. Finally, consider `Let(id,def,body)`. Let us recall from Figure 2.2, the code we had for this variant:

```
  | Let(id,def,body) ->
    eval_expr def en >>= fun ev ->
    eval_expr body (extend_env en id ev)
```

We first evaluate `def` in the current environment `en` producing an expressed value `ev`. This expressed value is used to extend the current environment `ev`, before evaluating the body `body`. Dropping `en`, which is now threaded implicitly for us, results in:

```
  | Let(id,def,body) ->
    eval_expr def >>= fun ev ->
    eval_expr body (extend_env id ev)
```

There are two problems with this code. First we need to be able to produce the modified environment **as a result** so that we can pass it on when evaluating `body`. This is achieved by updating `extend_env`, and `empty_env` too although we will not be needing the latter for now, that produces the updated environment as a result (notice the `env` in `env ea_result`):

```
let extend_env : string -> exp_val -> env ea_result =
  fun id v ->
  fun env -> Ok (ExtendEnv(id,v,env))
```

With this new operation we can produce the following code which is almost correct; we still have to discuss what to put in place of `>>???`:

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | Int(n) -> return (NumVal n)
  | Var(id) -> apply_env id
  | Div(e1,e2) ->  (* Add, Sub and Mul are similar and omitted *)
    eval_expr e1 >>=
    int_of_numVal >>= fun n1 ->
    eval_expr e2 >>=
    int_of_numVal >>= fun n2 ->
    if n2==0
    then error "Division by zero"
    else return (NumVal (n1/n2))
  | IsZero(e) ->
    eval_expr e >>=
    int_of_numVal >>= fun n ->
    return (BoolVal (n = 0))
  | ITE(e1,e2,e3) ->
    eval_expr e1 >>=
    bool_of_boolVal >>= fun b ->
    if b
    then eval_expr e2
    else eval_expr e3
  | Let(id,def,body) ->
    eval_expr def >>=
    extend_env id >>+
    eval_expr body
  | _ -> error "Not implemented yet!"

let parse s =
  let lexbuf = Lexing.from_string s in
  let ast = Parser.prog Lexer.read lexbuf in
  ast

let interp (e:string) : exp_val result =
  let c = e |> parse |> eval_expr
  in run c
```

interp.ml

Figure 2.6: Evaluator for LET

```
| Let(id,def,body) ->
  eval_expr def >>= fun ev ->
  extend_env id ev >>???
  eval_expr body
```

which can be simplified to

```
| Let(id,def,body) ->
  eval_expr def >>=
  extend_env id >>???
  eval_expr body
```

This code evaluates `def` under the current environment threaded by bind, then feeds the resulting expressed value into `extend_env id` to produce an extended environment. But now we are faced with a second problem. It is this extended environment that should be fed into `eval_expr body` and **not** the current environment that is threaded by bind (the current environment presumably has no mapping for id). This suggests introducing the following environment update operation:

```
let (>>+) : env ea_result -> 'a ea_result -> 'a ea_result =
  fun c d ->
  fun env ->
  match c env with
  | Error err -> Error err
  | Ok newenv -> d newenv
```

An expression such as `c >>+ d` first evaluates `c env`, where `env` is the current environment, producing an environment `newenv` as a result, and then completely ignores the current environment `env` feeding that new environment as current environment for `d`.

With the help of environment update, we can now complete our code for `Let`:

```
| Let(id,def,body) ->
  eval_expr def >>=
  extend_env id >>+
  eval_expr body
```

You can think of `c1 (>>=) f` as a form of composition of computations, "given an environment `en`, pass it on to `c1` producing an expressed value `v`, then pass `v` and `en` on to `f`, and return its result as the overall result". While `c1 (>>+) c2` may be thought of as, "given an environment `en`, pass it on to `c1` producing an environment `newenv` (not an expressed value!) as a result which is passed on to computation `c2`, returning the latter's result as the result of the overall computation."

Note the absence of all references to `en` in Listing. 2.6. Indeed, the environment will be passed around when we execute `run c`. According to the definition of `run`, `run c` just applies `c` to the empty environment `EmptyEnv`.

Example 2.1.1. *We conclude this section with some examples of expressions whose type involve the `ea_result` type constructor:*

Expression	**Type**	**Informal Description**
return (NumVal 3)	*exp_val ea_result*	*Denotes a function that when given an environment, ignores it, and immediately returns Ok (NumVal 3).*
error "oops"	*'a ea_result*	*Denotes a function that when given an environment, ignores it, and immediately returns Error "oops".*
apply_env "x"	*exp_val ea_result*	*Denotes a function that when given an environment, inspects it to find the expressed value v associated to "x". If it finds it, it returns Ok v, otherwise it returns Error "x not found".*
extend_env "x" (NumVal 3)	*env ea_result*	*Denotes a function that when given an environment, extends it producing a new environment newenv, with the new key-value pair x:=NumVal 3, and returns Ok newenv.*

2.1.5 Inspecting the Environment

It is often convenient to be able to inspect the contents of the environment as a means of understanding how evaluation works or simply for debugging purposes. This section extends LET with a new expression `debug(e)` whose evaluation will print the contents of the current environment and ignore `e`. We first add a new production to the grammar defining the concrete syntax of LET:

$$\langle \text{Expression} \rangle \quad ::= \quad \texttt{debug}(\langle \text{Expression} \rangle)$$

We next add a new variant to the type `expr` defining the abstract syntax of LET:

```
type expr =
  ...
  | Debug of expr
```

The next step is to specify, and then implement, the extension to the interpreter for LET that handles the new construct. What should we choose as the value resulting from evaluating `Debug(e)`? In other words, what should we choose to replace the questions marks below with?

$$\frac{}{\texttt{Debug(e)}, \rho \Downarrow ???}\ \text{EDebug}$$

Since `Debug(e)` has to halt execution (and print the environment), we will have it return an error. This way, no matter where it is placed, the error will get propagated hence effectively halting all further execution. The evaluation rule EDebug becomes:

$$\frac{}{\texttt{Debug(e)}, \rho \Downarrow \mathit{error}}\ \text{EDebug}$$

Finally, the implementation of this evaluation rule is given below. It makes use of an auxiliary function `string_of_env`, defined in `ds.ml`, which traverses an environment and returns a string representation of it.

```
eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | Debug(e) ->
    string_of_env >>= fun str ->
    print_endline str;
    error "Debug called"
```

Note that there is a slight discrepancy between the specification of the evaluation rule describing how Debug(e) is evaluated (*i.e.* EDebug) and our implementation. Indeed, the latter prints two strings on the screen but the former does not mention any side-effects such as printing. The reason for this mismatch is that we have decided to keep the specification of our interpreters as simple as possible. In particular, we have decided not to model side-effects such as printing on the screen. Later we will show how to model other side-effecting operations when specifying interpreters. Notably, we will include an assignment operation in our language.

2.2 Exercises

Exercise 2.2.1 (◊). *Write an OCaml expression of each of the types below:*

1. `expr`
2. `env`
3. `exp_val`
4. `exp_val result`
5. `int result`
6. `env result`
7. `int ea_result`
8. `exp_val ea_result`
9. `env ea_result`

Exercise 2.2.2. *Consider the following code:*

```
open Ds

let c =
  empty_env () >>+
  extend_env "x" (NumVal 1) >>+
  extend_env "y" (BoolVal false) >>+
  string_of_env
```

where the helper function `string_of_env` *is defined as follows:*

```
let string_of_expval = function
  | NumVal n -> "NumVal " ^ string_of_int n
  | BoolVal b -> "BoolVal " ^ string_of_bool b

let rec string_of_env' ac = function
  | EmptyEnv -> ac
  | ExtendEnv(id,v,env) -> string_of_env' ((id^":="^string_of_expval v)::ac) env

let string_of_env : string ea_result =
  fun env ->
  match env with
  | EmptyEnv -> Ok ">>Environment:\nEmpty"
  | _ -> Ok (">>Environment:\n"^ String.concat ",\n" (string_of_env' [] env))
```
ds.ml

1. *Fill in the missing parenthesis in the definition of* `c`*. Recall that* `>>+` *associates to the left.*
2. *What is the type of* `c`*?*
3. *What happens when you load the code into utop and type* `c`*?*
4. *What happens when you load it into utop and type* `run c`*?*

Exercise 2.2.3. *Consider the following extension of LET with pairs. Its concrete syntax includes all the grammar productions of LET plus:*

$$\begin{array}{rcl} \langle\text{Expression}\rangle & ::= & \mathit{pair}(\langle\text{Expression}\rangle, \langle\text{Expression}\rangle) \\ & | & \mathit{fst}(\langle\text{Expression}\rangle) \\ & | & \mathit{snd}(\langle\text{Expression}\rangle) \end{array}$$

Examples of programs in this language are

1. `pair (2,3)`
2. `pair (pair(7,9),3)`
3. `pair(zero?(4),11-x)`
4. `snd(pair (pair(7,9),3))`

The abstract syntax includes the following additional variants:

```
type expr =
  ...
  | Pair of expr*expr
  | Fst of expr
  | Snd of expr
```

Specify the interpreter (i.e. its evaluation rules) and then implement it.

Exercise 2.2.4. *Consider another extension to LET with pairs. Its concrete syntax is:*

$$\begin{array}{rcl} \langle\text{Expression}\rangle & ::= & \mathit{pair}(\langle\text{Expression}\rangle, \langle\text{Expression}\rangle) \\ & | & \mathit{unpair}(\langle\text{Identifier}\rangle, \langle\text{Identifier}\rangle) = \langle\text{Expression}\rangle\ \mathit{in}\ \langle\text{Expression}\rangle \end{array}$$

Pairs are constructed in the same way as in Exercise 2.2.3. However, to eliminate pairs instead of `fst` and `snd` we now have `unpair`. The expression `unpair (x,y)=e1 in e2` evaluates `e1`, makes sure it is a pair with components v1 and v2 and then evaluates `e2` in the extended environment where x is bound to v1 and y to v2. Examples of programs in this extension are the first three examples in Exercise 2.2.3 and:

1. `unpair (x,y) = pair(3, pair(5 , 12)) in x` *is a program that evaluates to* `Ok (NumVal 3)`.
2. *The program* `let x = 34 in unpair (y,z)=pair(2,x) in z` *evaluates to* `Ok (NumVal 34)`.

The abstract syntax of this extension is:

```
type expr =
  ...
  | Pair of expr*expr
  | Unpair of string*string*expr*expr
```

ast.ml

Specify the interpreter (i.e. its evaluation rules) and then implement it.

Exercise 2.2.5. *Consider the extension of LET with tuples. Its concrete syntax is that of LET together with the following new productions:*

$$\begin{array}{lcl} \langle \text{Expression} \rangle & ::= & \texttt{<}\langle \text{Expression} \rangle^{*(,)}\texttt{>} \\ \langle \text{Expression} \rangle & ::= & \textit{untuple}\ \texttt{<}\langle \text{Identifier} \rangle^{*(,)}\texttt{>} = \langle \text{Expression} \rangle\ \textit{in}\ \langle \text{Expression} \rangle \end{array}$$

The $*(,)$ above the nonterminal indicates zero or more copies separated by commas. The angle brackets construct a tuple with the values of its arguments. An expression of the form `untuple <x1,...,xn>=e1 in e2` *first evaluates e1, makes sure it is a tuple of n values, say v1 to vn, and then evaluates e2 in the extended environment where each identifier* `xi` *is bound to vi. Examples of programs in this extension are:*

1. `<2,3,4>`
2. `<2,3,zero?(0)>`
3. `<<7,9>,3>`
4. `<zero?(4),11-x>`
5. `untuple <x,y,z> = <3, <5 , 12>,4> in x` *evaluates to* `Ok (NumVal 3)`.
6. `let x = 34 in untuple <y,z>=<2,x> in z` *evaluates to* `Ok (NumVal 34)`.

Specify the interpreter (i.e. its evaluation rules) and then implement it.

Exercise 2.2.6. *Consider the following extension of LET with records. Its concrete syntax is given adding the following new productions to that of LET:*

$$\begin{array}{lcl} \langle \text{Expression} \rangle & ::= & \{\ \langle \text{Identifier} \rangle = \langle \text{Expression} \rangle^{+(;)} \} \\ \langle \text{Expression} \rangle & ::= & \langle \text{Expression} \rangle.\langle \text{Identifier} \rangle \end{array}$$

Examples of programs in this extension are:

1. `{age=2; height=3}`
2. `let person = {age=2; height=3} in let student = {pers=person; cwid=10} in student`
3. `{age=2; height=3}.age`
4. `{age=2; height=3}.ages`
5. `{age=2; age=3}`

Assume that the expressed values of LET are extended so that now records of expressed values may be produced as a result of evaluating a program (see the examples above).

```
type exp_val =
  ...
  | RecVal of (string*exp_val) list
```

The `expr` *type encoding the AST is also extended:*

```
type expr =
  ...
  | Record of (string*expr) list
  | Proj of expr*string
```

Specify the interpreter (i.e. its evaluation rules) and then implement it. Some examples of the result of evaluation of this extension are:

1. `{age=2; height=3}`

   Should evaluate to `Ok (RecVal [("age", NumVal 2); ("height", NumVal 3)])`.

2. `let person = {age=2; height=3} in let student = {pers=person; cwid=10} in student`

   Should evaluate to `Ok (RecVal [("pers", RecVal [("age", NumVal 2); ("height", NumVal 3)]); ("cwid", NumVal 10)`

3. `{age=2; height=3}.age`

   Should evaluate to `Ok (NumVal 2)`.

4. `{age=2; height=3}.ages`

   Should evaluate to `Error "Field not found"`.

5. `{age=2; age=3}`

   Should evaluate to `Error "Record has duplicate fields"`.

2.3 PROC

2.3.1 Concrete Syntax

$$
\begin{array}{lcl}
\langle \text{Expression} \rangle & ::= & \langle \text{Number} \rangle \\
\langle \text{Expression} \rangle & ::= & \langle \text{Identifier} \rangle \\
\langle \text{Expression} \rangle & ::= & \langle \text{Expression} \rangle \langle \text{BOp} \rangle \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & \texttt{zero?}(\langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{if}\ \langle \text{Expression} \rangle\ \texttt{then}\ \langle \text{Expression} \rangle\ \texttt{else}\ \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & \texttt{let}\ \langle \text{Identifier} \rangle\ \texttt{=}\ \langle \text{Expression} \rangle\ \texttt{in}\ \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & (\langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{proc}(\langle \text{Identifier} \rangle)\{\langle \text{Expression} \rangle\} \\
\langle \text{Expression} \rangle & ::= & (\langle \text{Expression} \rangle \langle \text{Expression} \rangle) \\
\\
\langle \text{BOp} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{array}
$$

2.3.2 Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
```

2.3.3 Interpreter

Specification

Evaluation judgements for PROC are exactly the same as for LET except that now the value resulting from evaluation of non-error computations, namely the expressed values, may either be an integer, a boolean or a **closure**. A closure is a triple consisting of an identifier, an expression and an environment. All three sets, expressed values, closures and environments must be defined mutually recursively since they depend on each other:

$$
\begin{array}{lcl}
\mathbb{EV} & := & \mathbb{Z} \cup \mathbb{B} \cup \mathbb{CL} \\
\mathbb{CL} & := & \{(\texttt{id}, \texttt{e}, \rho) \mid e \in \mathbb{EXP}, id \in \mathbb{ID}, \rho \in \mathbb{ENV}\} \\
\mathbb{ENV} & := & \mathbb{ID} \rightharpoonup \mathbb{EV}
\end{array}
$$

The evaluation judgement for PROC reads:

$$\texttt{e}, \rho \Downarrow r$$

The evaluation rules include those of LET (see Figure 2.1) plus the additional rules given in Figure 2.7.

$$\frac{}{\texttt{Proc(id,e)}, \rho \Downarrow (\texttt{id}, \texttt{e}, \rho)}\ \text{EProc}$$

$$\frac{\texttt{e1}, \rho \Downarrow (\texttt{id}, \texttt{e}, \sigma) \quad \texttt{e2}, \rho \Downarrow w \quad \texttt{e}, \sigma \oplus \{\texttt{id} := w\} \Downarrow v}{\texttt{App(e1,e2)}, \rho \Downarrow v}\ \text{EApp}$$

$$\frac{\texttt{e1}, \rho \Downarrow v \quad v \notin \mathbb{CL}}{\texttt{App(e1,e2)}, \rho \Downarrow \mathit{error}}\ \text{EAppErr}$$

Figure 2.7: Evaluation rules for PROC

Implementation

To extend the interpreter for LET to PROC, we need to model closures and then extend `eval_expr`. Modeling closures as runtime values is easy since closures are simply triples consisting of an identifier, an expression and an environment:

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
and
  env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
```
ds.ml

Now, for `eval_expr`, we add code for two new variants in the definition of `eval_expr`, namely `Proc(id,e)` and `App(e1,e2)`. Let us first analyze the former. Our first attempt might look something like this:

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | Proc(id,e)  ->
    return (ProcVal(id,e, en))
```

Evaluation of `Proc(id,e)` should produce a closure that includes both of `id` and `e`. It must also include the current environment, denoted `en` above. However, the identifier `en` is not in scope. Indeed, the current environment is passed around in the background by the helper functions for `ea_result`. We introduce a new helper function that reads the current environment and returns it as a result (*i.e.* `Ok env`, where `env` is the environment).

```
let rec lookup_env : env ea_result =
  fun env -> Ok env
```
ds.ml

With this new function we can now implement the evaluator for `Proc(id,e)`:

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
```

```
  match e with
  | Proc(id,e)  ->
    lookup_env >>= fun en ->
    return (ProcVal(id,e,en))
```

interp.ml

Two alternative implementations for the `Proc(e1,e2)` case might be:

```
| Proc(id,e)  ->
  fun env -> return (ProcVal(id,e,env)) env
```

and

```
| Proc(id,e)  ->
  fun env -> Ok (ProcVal(id,e,env))
```

However these reveal the internal encoding of `exp_val ea_result`. The constructor `Ok` should not be used outside ds.ml. Likewise, the fact that `exp_val ea_result` is a function type should not be known outside ds.ml

We next consider the case for `App(e1,e2)`. Evaluation of an application requires that we first evaluate `e1` obtaining an expressed value `v1`, then `e2` obtaining `v2`, and then we have to apply `v1` to `v2`. This last step implies checking that `v1` is indeed a closure and then evaluating the corresponding body. It is achieved by means of the helper function `apply_proc`:

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | App(e1,e2)  ->
    eval_expr e1 >>= fun v1 ->
    eval_expr e2 >>= fun v2 ->
    apply_proc v1 v2
```

The function `apply_proc` below matches its first argument `f` with the pattern `ProcVal(id,body,env)` to see if it is a closure, returning an error if this is not the case. If it is, however, we must extend the environment `env` stored inside the closure, with a new binding for the parameter, and then evaluate the body. The expression `return env` returns the environment `env` as a result. We feed that environment into `extend_env id a` and then evaluate the body under the extended environment.

```
let rec apply_proc : exp_val -> exp_val -> exp_val ea_result =
  fun f a ->
  match f with
  | ProcVal(id,body,env) ->
    return env >>+
    extend_env id a >>+
    eval_expr body
  | _ -> error "apply_proc: Not a procVal"
```

Figure 2.8 summarizes the code described above for procedures and applications.

2.3.4 Dynamic Scoping

If we comment out the line below, then we implement dynamic scoping:

```
let rec apply_proc : exp_val -> exp_val -> exp_val ea_result =
  fun f a ->
  match f with
  | ProcVal(id,body,env) ->
    return env >>+
    extend_env id a >>+
    eval_expr body
  | _ -> error "apply_proc: Not a procVal"
and
  eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | Proc(id,e)  ->
    lookup_env >>= fun en ->
    return (ProcVal(id,e,en))
  | App(e1,e2)  ->
    eval_expr e1 >>= fun v1 ->
    eval_expr e2 >>= fun v2 ->
    apply_proc v1 v2
```

interp.ml

Figure 2.8: Interpreter for PROC

```
let rec apply_proc : exp_val -> exp_val -> exp_val ea_result =
  fun f a ->
  match f with
  | ProcVal(id,body,env) ->
    (* return env >>+ *)
    extend_env id a >>+
    eval_expr body
  | _ -> error "apply_proc: Not a procVal"
```

Indeed, in this case the environment that is extended by `extend_env id a` is the current environment and not the one that was saved in the closure. Here are some examples of executing programs in this variant of PROC:

```
interp "
let f = proc (x) { if zero?(x) then 1 else x*(f (x-1)) }
in (f 6) ";;
- : Ds.exp_val Ds.result = Ds.Ok (Ds.NumVal 720)

utop # interp "
let f = proc (x) { x+a }
in let a=2
in (f 2)";;
- : Ds.exp_val Ds.result = Ds.Ok (Ds.NumVal 4)

utop # interp "
let f= let a=2 in proc(x) { x+a}
in (f 2) ";;
- : Ds.exp_val Ds.result = Ds.Error "a not found!"
```

utop

2.4 Exercises

Exercise 2.4.1. *Write a derivation to show that* `let f = proc (x) { x-11 } in (f 77)` *is a valid program in PROC.*

Exercise 2.4.2. *Write down the parse tree for the expression* `let pred = proc(x) { x-1 } in (pred 5)`*.*

Exercise 2.4.3. *Write down the result of evaluating the expressions below. Depict the full details of the closure, including the environment. Use the tabular notation seen in class to depict the environment.*

- `proc (x) { x-11 }`
- `proc (x) { let y=2 in x }`
- `let a=1 in proc (x) { x }`
- `let a=1 in let b=2 in proc (x) { x }`
- `let f=(let b=2 in proc (x) { x }) in f`
- `proc (x) { proc (y) { x-y } }`

Exercise 2.4.4. *Depict the environment extant at the breakpoint (signalled with the* `debug` *expression):*

```
let a=1
in let b=2
in let c=proc (x) { x }
in debug((c b))
```

Exercise 2.4.5. *Depict the environment extant at the breakpoint:*

```
let a=1
in let b=2
in let c = proc (x) { debug(proc (y) { x-y } )}
in (c b)
```

Exercise 2.4.6. *Depict the environment extant at the breakpoint:*

```
let x=2
in let y=proc (d) { x }
in let z=proc(d) { x }
in debug(3)
```

Exercise 2.4.7 (◊)**.** *Use the "higher-order" trick of self-application to implement the mutually recursive definitions of* `even` *and* `odd` *in PROC:*

$$
\begin{array}{rcl}
even(0) & = & true \\
even(n) & = & odd(n-1) \\
\\
odd(0) & = & false \\
odd(n) & = & even(n-1)
\end{array}
$$

Exercise 2.4.8 (◊). *Use the “higher-order” trick of self-application to implement a function* `pbt` *that given a value* v *and a height* h *builds a perfect binary tree constructed out of pairs and that has* v *im the leaves and has height* v*. For example* `((pbt 2) 3)` *should produce*

```
PairVal
 (PairVal
   (PairVal (PairVal (NumVal 2, NumVal 2),
     PairVal (NumVal 2, NumVal 2)),
   PairVal (PairVal (NumVal 2, NumVal 2),
    PairVal (NumVal 2, NumVal 2))),
 PairVal
  (PairVal (PairVal (NumVal 2, NumVal 2),
    PairVal (NumVal 2, NumVal 2)),
  PairVal (PairVal (NumVal 2, NumVal 2),
   PairVal (NumVal 2, NumVal 2))))
```

Lists and Trees

Exercise 2.4.9.

2.5 REC

Our language unfortunately does not support recursion[3]. The following attempt at defining factorial and then applying it to compute factorial of 5 fails. The reason is that `f` is not visible in the body of the `proc`.

```
let f =
    proc (x) {
        if zero?(x)
        then 1
        else x*(f (x-1)) }
in (f 5)
```

In order to verify this, evaluate the following expression:

```
let f =
    proc (x) {
        debug(if zero?(x)
        then 1
        else x*(f (x-1))) }
in (f 5)
```

Note that the environment in the closure for `f` does not include a reference to `f` itself. The next language we shall look at, namely REC, includes a new programming abstraction that allows us to define recursive functions. In REC we will write:

```
letrec fact(x) =
    if zero?(x)
    then 1
    else x * (fact (x-1))
in (fact 5)
```

[3]See exercises **??** and **??** on the “higher-order” trick though.

2.5.1 Concrete Syntax

⟨Expression⟩ ::= ⟨Number⟩
⟨Expression⟩ ::= ⟨Identifier⟩
⟨Expression⟩ ::= ⟨Expression⟩⟨BOp⟩⟨Expression⟩
⟨Expression⟩ ::= `zero?`(⟨Expression⟩)
⟨Expression⟩ ::= `if` ⟨Expression⟩ `then` ⟨Expression⟩ `else` ⟨Expression⟩
⟨Expression⟩ ::= `let` ⟨Identifier⟩ = ⟨Expression⟩ `in` ⟨Expression⟩
⟨Expression⟩ ::= (⟨Expression⟩)
⟨Expression⟩ ::= `proc`(⟨Identifier⟩){⟨Expression⟩}
⟨Expression⟩ ::= (⟨Expression⟩⟨Expression⟩)
⟨Expression⟩ ::= `letrec` ⟨Identifier⟩(⟨Identifier⟩) = ⟨Expression⟩ `in` ⟨Expression⟩

⟨BOp⟩ ::= + | - | * | /

2.5.2 Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
  | Letrec of string*string*expr*expr
```

ast.ml

For example, the result of parsing the expression:

```
letrec fact(x) =
    if zero?(x)
    then 1
    else x * (fact (x-1))
in (fact 5)
```

is the AST:

```
Letrec ("fact", "x",
   ITE (IsZero (Var "x"), Int 1,
    Mul (Var "x", App (Var "fact", Sub (Var "x", Int 1)))),
   App (Var "fact", Int 5))
```

2.5.3 Interpreter

Recursive functions will be represented as special closures called recursion closures. Later we will look at another implementation involving circular environments. A **recursion closure** is

$$\frac{\texttt{e2}, \rho \oplus \{\texttt{id} := (\texttt{par}, \texttt{e1}, \rho)^r\} \Downarrow v}{\texttt{Letrec(id,par,e1,e2)}, \rho \Downarrow v} \text{ELetRec}$$

$$\frac{\rho(\texttt{id}) = (\texttt{par}, \texttt{e}, \sigma)^r}{\texttt{Var(id)}, \rho \Downarrow (\texttt{par}, \texttt{e}, \sigma \oplus \{\texttt{id} := (\texttt{par}, \texttt{e}, \sigma)^r\})} \text{EVarLetRec}$$

Figure 2.9: Evaluation rules for REC

a closure with a tag "r" to distinguish it from a standard closure, written $(\texttt{id}, \texttt{e}, \rho)^r$, where $e \in \mathbb{EXP}$, $id \in \mathbb{ID}$ and $\rho \in \mathbb{ENV}$. The set of all recursion closures is denoted $\mathbb{RCL}$:

$$\begin{array}{rcl} \mathbb{ENV} & := & \mathbb{ID} \rightharpoonup \mathbb{EV} \cup \mathbb{RCL} \\ \mathbb{EV} & := & \mathbb{Z} \cup \mathbb{B} \cup \mathbb{U} \cup \mathbb{CL} \\ \mathbb{CL} & := & \{(\texttt{id}, \texttt{e}, \rho) \mid e \in \mathbb{EXP}, id \in \mathbb{ID}, \rho \in \mathbb{ENV}\} \\ \mathbb{RCL} & := & \{(\texttt{id}, \texttt{e}, \rho)^r \mid e \in \mathbb{EXP}, id \in \mathbb{ID}, \rho \in \mathbb{ENV}\} \end{array}$$

Note that recursion closures are not expressed values. We cannot write a program that, when evaluated, returns a recursion closure. They are an auxiliary device for defining evaluation of recursive programs. More precisely, recursive function definitions will be stored as recursion closures. However, lookup of recursive functions will produce standard closures, the latter being computed on the fly.

Specification

The set of results are:

$$\mathbb{R} := \mathbb{EV} \cup \{error\}$$

Evaluation judgements for REC are the same as for PROC:

$$\texttt{e}, \rho \Downarrow r$$

where $r \in \mathbb{R}$. Evaluation rules for REC are those of PROC together with the ones in Figure 2.9. Two new evaluation rules are added to those of PROC to obtain REC. The rule ELetRec creates a recursion closure and adds it to the current environment ρ and then continues with the evaluation of e2. The rule EVarLetRec does lookup of identifiers that refer to previously declared recursive functions. Upon finding the corresponding recursion closure in the current environment, it creates a new closure and returns it. Note that the newly created closure includes an environment that has a reference to `f` itself.

Implementation

Recursion closures are implemented by adding a new constructor to the type `expr`, namely `ExtendEnvRec` below:

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
```

```
  | ProcVal of string*Ast.expr*env
and
  env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of string*string*Ast.expr*env
```
ds.ml

Note that the arguments of `ExtendEnvRec(id,par,body,env)` are four: the name of the recursive function being defined `id`, the name of the formal parameter `par`, the body of the recursive function `body`, and the rest of the environment `env`. If we consider the environment $\rho \oplus \{\mathtt{id} := (\mathtt{par}, \mathtt{e1}, \rho)^r\}$ in the rule ELetRec of Figure 2.9, it would seem we are missing an argument. Indeed, the operator "$\oplus$" in the evaluation rule is modeled by the `ExtendEnvRec` constructor in our implementation. However, there is no need to store ρ in our implementation since it is just the tail of the environment.

Next we need an operation similar to `extend_env` but that adds a new recursion closure to the environment:

```
let extend_env_rec : string -> string -> Ast.expr -> env ea_result =
  fun id par body ->
  fun env -> Ok (ExtendEnvRec(id,par,body,env))
```
ds.ml

In addition, we need to update the implementation of `apply_env` so that it deals with lookup of recursive functions, thus correctly implementing EVarLetRec. This involves adding a new clause (see code highlighted below):

```
let rec apply_env : string -> exp_val ea_result =
  fun id ->
  fun env ->
  match env with
  | EmptyEnv -> Error (id^" not found!")
  | ExtendEnv(v,ev,tail) ->
    if id=v
    then Ok ev
    else apply_env id tail
  | ExtendEnvRec(v,par,body,tail) ->
    if id=v
    then Ok (ProcVal (par,body,env))
    else apply_env id tail
```
ds.ml

Regarding the code for the interpreter itself, we need only add a new clause, namely the one for `Letrec(id,par,e1,e2)`:

```
  | Letrec(id,par,e1,e2) ->
    extend_env_rec id par e1 >>+
    eval_expr e2
```
interp.ml

Exercise 2.5.1. *Evaluate the following expressions in utop:*

1.
```
utop # interp "
let one=1
```

```
in letrec fact(x) =
             if zero?(x)
             then one
             else x * (fact (x-1))
in debug((fact 6))" ;;
```

2.
```
utop # interp "
let one=1
in letrec fact(x) =
       debug(if zero?(x)
             then one
             else x * (fact (x-1)))
in (fact 6)" ;;
```

3.
```
utop # interp "
let one=1
in letrec fact(x) =
             if zero?(x)
             then one
             else x * (fact (x-1))
in fact" ;;
```

Exercise 2.5.2 (◊). *Consider the following functions in OCaml:*

```
let rec add n m =
  match n with
  | 0 -> m
  | n' -> 1 + add (n'-1) m

let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | h::t -> h :: append t l2

let rec map l f =
  match l with
  | [] -> []
  | h::t -> (f h) :: map t f

let rec filter l p =
  match l with
  | [] -> []
  | h::t ->
    if p h
    then h :: filter t p
    else filter t p

let rec foldr l f a =
  match l with
  | [] -> a
  | h::t -> f h (foldr t f a)
```

Code them in the extension of REC of Exercise 2.4.9. For example, here is the code for `add`*:*

```
# interp "
```

```
letrec add(n) = proc (m) {
                    if zero?(n)
                    then m
                    else 1 + ((add (n-1)) m) }
in ((add 2) 3)";;
- : exp_val Rec.Ds.result = Ok (NumVal 5)
```
utop

Exercise 2.5.3 (◊). *Consider the following expression in REC*

```
let z = 0
in let prod = proc (x) { proc (y) { x*y }}
in letrec f(n) = if zero?(n) then 1 else ((prod n) (f (n-1)))
in (f 10)
```

A `debug` instruction was placed somewhere in the code and it produced the environments below. Where was it placed? Identify and signal (see instructions below) the location for each of the three items below. Note that there may be more than one solution for each item, it suffices to supply just one.

1.
```
>>Environment:
z:=NumVal 0,
prod:=ProcVal (x,Proc(y,Mul(Var x,Var y)),z:=NumVal 0),
f:=Rec(n,IfThenElse(Zero?(Var n),Int 1,
        App(App(Var prod,Var n),App(Var f,Sub(Var n,Int 1))))),
n:=NumVal 0
```

Draw a box around the argument of `debug`:

```
let z = 0
in let prod = proc (x) { proc (y) { x*y }}
in letrec f(n) = if zero?(n) then 1 else ((prod n) (f (n-1)))
in (f 10)
```

2.
```
>>Environment:
z:=NumVal 0
```

Draw a box around the argument of `debug`:

```
let z = 0
in let prod = proc (x) { proc (y) { x*y }}
in letrec f(n) = if zero?(n) then 1 else ((prod n) (f (n-1)))
in (f 10)
```

3.
```
>>Environment:
z:=NumVal 0,
x:=NumVal 10
```

Draw a box around the argument of `debug`:

```
let z = 0
in let prod = proc (x) { proc (y) { x*y }}
in letrec f(n) = if zero?(n) then 1 else ((prod n) (f (n-1)))
in (f 10)
```

Chapter 3

Imperative Programming

3.1 Mutable Data Structures in OCaml

3.1.1 A counter object

```
type counter =
  { inc: unit -> unit;
    dec: unit -> unit;
    read : unit -> int}
```

```
let c =
    let state = ref 0 in
    { inc = (fun () -> state := !state+1);
      dec = (fun () -> state := !state-1);
      read = (fun () -> !state) }
```

```
utop # c.read ();;
- : int = 0
utop # c.inc ();;
- : unit = ()
utop # c.read ();;
- : int = 1
utop # c.dec ();;
- : unit = ()
utop # c.read ();;
- : int = 0
```

utop

3.1.2 A stack object

```
type stack =
  { push : int -> unit;
    pop : unit -> int;
    top : unit -> int};;
```

```
let s = let state = ref []
        in { push = (fun i -> state:= i::!state);
             pop = (fun () -> let temp = List.hd !state
                              in state := List.tl !state; temp);
             top = (fun () -> List.hd !state)}
```

```
utop # s.push 1;;
- : unit = ()
utop # s.push 2;;
- : unit = ()
utop # s.pop ();;
- : int = 2
utop # s.top ();;
- : int = 1
```

utop

3.2 EXPLICIT-REFS

The following is an extension of REC.

3.2.1 Concrete Syntax

Examples of expressions in EXPLICIT-REFS:

```
newref(2)

let a=newref(2)
in a

let a=newref(2)
in deref(a)

let a=newref(2)
in setref(a,deref(a)+1)

let a=newref(2)
in begin
     setref(a,deref(a)+1);
     deref(a)
   end

let g =
     let counter = newref(0)
     in proc (d) {
         begin
          setref(counter, deref(counter)+1);
          deref(counter)
         end
       }
in (g 11) - (g 22)
```

⟨Expression⟩	::=	⟨Number⟩
⟨Expression⟩	::=	⟨Identifier⟩
⟨Expression⟩	::=	⟨Expression⟩⟨BOp⟩⟨Expression⟩
⟨Expression⟩	::=	`zero?`(⟨Expression⟩)
⟨Expression⟩	::=	`if` ⟨Expression⟩ `then` ⟨Expression⟩ `else` ⟨Expression⟩
⟨Expression⟩	::=	`let` ⟨Identifier⟩ = ⟨Expression⟩ `in` ⟨Expression⟩
⟨Expression⟩	::=	(⟨Expression⟩)
⟨Expression⟩	::=	`proc`(⟨Identifier⟩){⟨Expression⟩}
⟨Expression⟩	::=	(⟨Expression⟩⟨Expression⟩)
⟨Expression⟩	::=	`letrec`⟨Identifier⟩(⟨Identifier⟩)=⟨Expression⟩`in`⟨Expression⟩
⟨Expression⟩	::=	`newref`(⟨Expression⟩)
⟨Expression⟩	::=	`deref`(⟨Expression⟩)
⟨Expression⟩	::=	`setref`(⟨Expression⟩, ⟨Expression⟩)
⟨Expression⟩	::=	`begin` ⟨Expression⟩$^{*(;)}$ `end`
⟨BOp⟩	::=	+ | - | * | /

The notation $*(;)$ above the nonterminal ⟨Expression⟩ in the production for `begin`/`end` indicates zero or more expressions separated by semi-colons.

3.2.2 Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
  | Letrec of string*string*expr*expr
  | NewRef of expr
  | DeRef of expr
  | SetRef of expr*expr
  | BeginEnd of expr list
  | Debug of expr
```

3.2.3 Interpreter

Specification

We assume given a set of (symbolic) memory locations $\mathbb{L}$. We write ℓ, ℓ_i for memory locations. A heap or **store** is a partial function from memory locations to expressed values. The set of stores is denoted $\mathbb{S}$:

$$\mathbb{S} := \mathbb{L} \rightharpoonup \mathbb{EV}$$

$$\frac{\texttt{e}, \rho, \sigma \Downarrow v, \sigma' \quad \ell \notin \mathsf{dom}(\sigma)}{\texttt{NewRef(e)}, \rho, \sigma \Downarrow \ell, \sigma' \oplus \{\ell := v\}} \text{ENewRef}$$

$$\frac{\texttt{e}, \rho, \sigma \Downarrow v, \sigma' \quad v \in \mathbb{L} \quad v \in \mathsf{dom}(\sigma')}{\texttt{DeRef(e)}, \rho, \sigma \Downarrow \sigma'(v), \sigma'} \text{EDeRef}$$

$$\frac{\texttt{e}, \rho, \sigma \Downarrow v, \sigma' \quad v \notin \mathbb{L}}{\texttt{DeRef(e)}, \rho, \sigma \Downarrow \mathit{error}, \sigma'} \text{EDeRefErr1} \quad \frac{\texttt{e}, \rho, \sigma \Downarrow v, \sigma' \quad v \in \mathbb{L} \quad v \notin \mathsf{dom}(\sigma')}{\texttt{DeRef(e)}, \rho, \sigma \Downarrow \mathit{error}, \sigma'} \text{EDeRefErr2}$$

$$\frac{\texttt{e1}, \rho, \sigma \Downarrow v, \sigma' \quad v \in \mathbb{L} \quad \texttt{e2}, \rho, \sigma' \Downarrow w, \sigma''}{\texttt{SetRef(e1,e2)}, \rho, \sigma \Downarrow \mathit{unit}, \sigma'' \oplus \{v := w\}} \text{ESetRef} \quad \frac{\texttt{e1}, \rho, \sigma \Downarrow v, \sigma' \quad v \notin \mathbb{L}}{\texttt{SetRef(e1,e2)}, \rho, \sigma \Downarrow \mathit{error}, \sigma'} \text{ESetRefErr}$$

$$\frac{\texttt{n} > 0 \quad (\texttt{ei}, \rho, \sigma_i \Downarrow v_i, \sigma_{i+1})_{i \in 1..n}}{\texttt{BeginEnd([e1;...;en])}, \rho, \sigma_1 \Downarrow v_n, \sigma_{n+1}} \text{EBeginEndNE}$$

$$\frac{}{\texttt{BeginEnd([])}, \rho, \sigma \Downarrow \mathit{unit}, \sigma} \text{EBeginEndE}$$

Figure 3.1: Evaluation rules for EXPLICIT-REFS

where the set of expressed values includes locations:

$$\mathbb{EV} := \mathbb{Z} \cup \mathbb{B} \cup \mathbb{U} \cup \mathbb{CL} \cup \mathbb{L}$$

Also among expressed values we find $\mathbb{U} := \{\mathit{unit}\}$. This new value will be explained below, when we describe the evaluation rules for EXPLICIT-REFS.

Evaluation judgements in EXPLICIT-REFS take the following form, where `e` is an expression, ρ and environment, σ the initial store, r the result and σ' the final store

$$\texttt{e}, \rho, \sigma \Downarrow r, \sigma'$$

Note that the result of evaluating an expression now returns both a result and an updated store. The evaluation rules for EXPLICIT-REFS are given in Figure 3.1. The rule ESetRef and ESetRefErr describe the behavior of assignment. Notice that an assignment such as `SetRef(e1,e2)` is evaluated to cause an effect, namely update the contents of the location obtained from evaluating `e1` with the value obtained from evaluating `e2`. We do not expect to get any meaningful value back. However, all expressions have to denote a value. As a consequence, we use a new expressed value unit, as the expressed value returned by an assignment.

Implementing Stores

The implementation of the evaluator for EXPLICIT-REFS requires that we first implement stores. Since a store is a mutable data structure we will use OCaml arrays. The following interface file declares the types of the values in the public interface of the store. These values include a parametric type constructor `Store.t`, the type of the store itself and multiple functions.

```
open Ds
type 'a t

val empty_store : int -> 'a -> 'a t
val get_size : 'a t -> int
val new_ref : 'a t -> 'a -> int
val deref: 'a t -> int -> 'a ea_result
val set_ref : 'a t -> int -> 'a -> unit ea_result
val string_of_store : ('a -> string) -> 'a t -> string
```

store.mli

These operations are:

- `empty_store n v` returns a store of size `n` where each element is initialized to `v`
- `get_size s` returns the number of elements in the store.
- `new_ref s v` stores v in a fresh location and returns the location.
- `deref s l` returns the contents of location `l`, prefixed by `Some`, in the store `s`. This operation fails, returning `None`, if the location is out of bounds.
- `set_ref s l v` updates the contents of `l` in `s` with `v`. It fails, returning `None`, if the index is out of bounds.
- `string_of_store to_str s` returns a string representation of `s` resulting from applying `to_str` to each element.

Each of the above operations implemented in `store.ml`.

```
open Ds

type 'a t = { mutable data: 'a array; mutable size: int}
  (* data is declared mutable so the store may be resized *)

let empty_store : int -> 'a -> 'a t =
  fun i v -> { data=Array.make i v; size=0 }

let get_size : 'a t -> int =
  fun st -> st.size

let enlarge_store : 'a t -> 'a -> unit =
  fun st v ->
  let new_array = Array.make (st.size*2) v
  in Array.blit st.data 0 new_array 0 st.size;
  st.data<-new_array

let new_ref : 'a t -> 'a -> int =
  fun st v ->
  if Array.length (st.data)=st.size
  then enlarge_store st v
  else ();
  begin
    st.data.(st.size)<-v;
    st.size<-st.size+1;
    st.size-1
```

```
  end

let deref : 'a t -> int -> 'a ea_result =
  fun st l ->
  if l>=st.size
  then error "Index out of bounds"
  else return  (st.data.(l))

let set_ref :  'a t -> int -> 'a -> unit ea_result =
  fun st l v ->
  if l>=st.size
  then error "Index out of bounds"
  else return (st.data.(l)<-v)

let rec take n = function
  | [] -> []
  | x::xs when n>0 -> x::take (n-1) xs
  | _ -> []

let string_of_store' f st =
  let ss = List.mapi (fun i x -> (i,f x)) (take st.size (Array.to_list st.data))
  in
  if ss==[]
  then "Empty"
  else List.fold_left (fun curr (i,s) -> curr^string_of_int i^"->"^s^"\n")
        ""
        ss

let string_of_store f st =
  "Store:\n"^ string_of_store' f st
```

store.ml

Implementation

We could have followed the ideas we developed for environments and have them be threaded for us behind the scenes. This would lead to a similar extension of our current result type `ea_result` so that it also abstracts over the store. Consequently, we would avoid having to thread the store around. However, in order to keep things simple and since the concept of threading behind the scenes has already been introduced via environments, we choose to hold the store in a top-level or global variable `g_store`.

```
let g_store = Store.empty_store 20 (NumVal 0)
```

interp.ml

`g_store` denotes a store of size 20, whose values have arbitrarily been initialized to `NumVal 0`.

Next we consider the new expressed values, namely symbolic locations and unit. Locations will be denoted by an integer wrapped inside a `RefVal` constructor. For example, `RefVal 7` is a pointer to memory location 7.

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
  | UnitVal
```

```
  | RefVal of int
```
ds.ml

Next we move on to the interpreter, only addressing the new variants.

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | NewRef(e) ->
    eval_expr e >>= fun ev ->
    return (RefVal (Store.new_ref g_store ev))
  | DeRef(e) ->
    eval_expr e >>=
    int_of_refVal >>= fun l ->
    Store.deref g_store l
  | SetRef(e1,e2) ->
    eval_expr e1 >>=
    int_of_refVal >>= fun l ->
    eval_expr e2 >>= fun ev ->
    Store.set_ref g_store l ev >>= fun _ ->
    return UnitVal
  | BeginEnd([]) ->
    return UnitVal
  | BeginEnd(es) ->
    sequence (List.map eval_expr es) >>= fun l ->
    return (List.hd (List.rev l))
  | Debug(_e) ->
    string_of_env >>= fun str_env ->
    let str_store = Store.string_of_store string_of_expval g_store
    in (print_endline (str_env^"\n"^str_store);
    error "Debug called")
  | _ -> error ("Not implemented: "^string_of_expr e)
```
interp.ml

3.2.4 Extended Example: Encoding Objects

EXPLICIT-REFS with records

```
let c = let s = newref(0)
        in
         {
           inc = proc (d) { setref(s,deref(s)+d) };
           read = proc (x) { deref(s) };
           reset = proc (d) { setref(s,0) }
         }
in begin
     (c.inc 1);
     (c.inc 2);
     (c.read 0)
   end
```

```
letrec self(s) =
           { inc = proc (d) { setref(s,deref(s)+d) };
             read = proc (x) { deref(s) };
             reset = proc (d) {
                     let current = ((self s).read 0)
```

```
                     in ((self (s)).inc (-current))}
           }
in let new_counter =  proc(d) {
                        let s = newref(0)
                        in (self s)
                      }
in let c= (new_counter 0)
in begin
    (c.inc 1);
    (c.inc 2);
    (c.reset 0);
    (c.read 0)
  end
```

3.3 IMPLICIT-REFS

The following is an extension of REC.

3.3.1 Concrete Syntax

Examples of expressions in IMPLICIT-REFS

```
let x=2
in begin
     set x=3;
     x
end

let x=2
let y=x+1
in begin
     set y=y+1;
     y
end

let x=2
in let f = proc (n) {  begin set x=x+1; 1 end }
in let g = proc (n) { begin set x=x+1; 2 end }
in begin
     (f 0)+(g 0);
     x
   end
```

$$
\begin{array}{lcl}
\langle \text{Expression} \rangle & ::= & \langle \text{Number} \rangle \\
\langle \text{Expression} \rangle & ::= & \langle \text{Identifier} \rangle \\
\langle \text{Expression} \rangle & ::= & \langle \text{Expression} \rangle \langle \text{BOp} \rangle \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & \texttt{zero?}(\langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{if}\ \langle \text{Expression} \rangle\ \texttt{then}\ \langle \text{Expression} \rangle\ \texttt{else}\ \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & \texttt{let}\ \langle \text{Identifier} \rangle = \langle \text{Expression} \rangle\ \texttt{in}\ \langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & (\langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{proc}(\langle \text{Identifier} \rangle)\{\langle \text{Expression} \rangle\} \\
\langle \text{Expression} \rangle & ::= & (\langle \text{Expression} \rangle \langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{letrec}\langle \text{Identifier} \rangle(\langle \text{Identifier} \rangle)\texttt{=}\langle \text{Expression} \rangle\texttt{in}\langle \text{Expression} \rangle \\
\langle \text{Expression} \rangle & ::= & \texttt{set}(\langle \text{Expression} \rangle, \langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{begin}\ \langle \text{Expression} \rangle^{+(;)}\ \texttt{end} \\
\\
\langle \text{BOp} \rangle & ::= & + \mid - \mid * \mid /
\end{array}
$$

3.3.2 Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
  | Letrec of string*string*expr*expr
  | Set of string*expr
  | BeginEnd of expr list
  | Debug of expr
```

3.3.3 Interpreter

Specification

Since in IMPLICIT-REFS all identifiers are mutable, the environment will map all identifiers to locations in the store. Evaluation judgements in IMPLICIT-REFS take the following form, where `e` is an expression, ρ and environment, σ the initial store, r the result and σ' the final store

$$\texttt{e}, \rho, \sigma \Downarrow r, \sigma'$$

As mentioned, ρ maps identifiers to locations, it **no longer** maps them to expressed values. Hence identifier lookup now has to lookup the location first in the environment and then access the contents in the store. This is exactly what the rule EVar states:

$$\frac{\sigma(\rho(\texttt{id})) = v}{\texttt{Var(id)}, \rho, \sigma \Downarrow v, \sigma} \text{EVar}$$

$$\frac{\sigma(\rho(\texttt{id})) = v}{\texttt{Var(id)}, \rho, \sigma \Downarrow v, \sigma}\ \textsf{EVar} \qquad \frac{\rho(\texttt{id}) \notin \mathbb{L}\ \textsf{or} \rho(\texttt{id}) \notin \textsf{dom}(\sigma)}{\texttt{Var(id)}, \rho, \sigma \Downarrow \mathit{error}, \sigma}\ \textsf{EVarErr}$$

$$\frac{}{\texttt{Proc(id,e)}, \rho, \sigma \Downarrow (\texttt{id}, \texttt{e}, \rho), \sigma}\ \textsf{EProc}$$

$$\frac{\texttt{e1}, \rho, \sigma \Downarrow (\texttt{id}, \texttt{e}, \tau), \sigma_1 \quad \texttt{e2}, \rho, \sigma_1 \Downarrow w, \sigma_2 \quad \ell \notin \textsf{dom}(\sigma_1) \quad \texttt{e}, \tau \oplus \{\texttt{id} := \ell\}, \sigma_2 \oplus \{\ell := w\} \Downarrow v, \sigma_3}{\texttt{App(e1,e2)}, \rho, \sigma \Downarrow v, \sigma_3}\ \textsf{EApp}$$

$$\frac{\texttt{e}, \rho, \sigma \Downarrow v, \sigma'}{\texttt{Set(id,e)}, \rho, \sigma \Downarrow \mathit{unit}, \sigma' \oplus \{\rho(\texttt{id}) := v\}}\ \textsf{ESet} \qquad \frac{\rho(\texttt{id}) \notin \mathbb{L}\ \textsf{or}\ \rho(\texttt{id}) \notin \textsf{dom}(\sigma)}{\texttt{Set(id,e)}, \rho, \sigma \Downarrow \mathit{error}, \sigma}\ \textsf{ESetErr}$$

$$\frac{\texttt{n} > 0 \quad (\texttt{ei}, \rho, \sigma_i \Downarrow v_i, \sigma_{i+1})_{i \in 1..n}}{\texttt{BeginEnd([e1;...;en])}, \rho, \sigma_1 \Downarrow v_n, \sigma_{n+1}}\ \textsf{EBeginEndNE}$$

$$\frac{}{\texttt{BeginEnd([])}, \rho, \sigma \Downarrow \mathit{unit}, \sigma}\ \textsf{EBeginEndE}$$

Figure 3.2: Evaluation rules for IMPLICIT-REFS

Indeed, $\rho(\texttt{id})$ denotes a location whose contents is looked up in the store σ. If $\rho(\texttt{id})$ is not a valid location, then an error is returned, as described by rule EVarErr. The full set of evaluation rules are given in Figure 3.2.

Implementation

We address the implementation of the evaluator. For now we ignore `Letrec` and then take it up later. Instead we focus on the `App(e1,e2)` case, which needs some minor updating, and also on the new variants.

Regarding the `App(e1,e2)` case, we need to slightly modify the `apply_proc` function. We briefly recall the code for `apply_proc` as implemented in the PROC (Figure 2.8):

```
let rec apply_proc : exp_val -> exp_val -> exp_val ea_result =
  fun f a ->
  match f with
  | ProcVal(id,body,env) ->
    return env >>+
    extend_env id a >>+
    eval_expr body
  | _ -> error "apply_proc: Not a procVal"
```

Note that evaluation of the body requires extending the environment with a new key-value pair, namely (`id`,`a`), where `a` is the expressed value supplied as argument. Environments no longer map identifiers to expressed values, but to locations. So we first need to allocate `a` in the store in a fresh location `l` and then extend the environment with the key-value pair (`id`,`l`). The updated code for `apply_proc` is given below.

```
let rec apply_proc ev1 ev2 =
  match ev1 with
  | ProcVal(id,body,en) ->
    return en >>+
    extend_env id (RefVal (Store.new_ref g_store ev2)) >>+
    eval_expr body
  | _ -> error "apply_proc: Not a procVal"
```

We now address the new cases (and the ones we need to modify) for the interpreter:

```
  | Var(id) ->
    apply_env id >>=
    int_of_refVal >>=
    Store.deref g_store
  ...
  | Let(v,def,body) ->
    eval_expr def >>= fun ev ->
    let l = Store.new_ref g_store ev
    in extend_env v (RefVal l) >>+
    eval_expr body
  | Set(id,e) ->
    eval_expr e >>= fun ev ->
    apply_env id >>=
    int_of_refVal >>= fun l ->
    Store.set_ref g_store l ev >>= fun _ ->
    return UnitVal
  | BeginEnd([]) ->
    return UnitVal
  | BeginEnd(es) ->
    sequence (List.map eval_expr es) >>= fun vs ->
    return (List.hd (List.rev vs))
 | Debug(_e) ->
    string_of_env >>= fun str_env ->
    let str_store = Store.string_of_store string_of_expval g_store
    in (print_endline (str_env^"\n"^str_store);
    error "Debug called")
  | _ -> error ("Not implemented: "^string_of_expr e)
```

`letrec` Revisited

Our implementation of `letrec` in REC consisted in adding a specific entry in the environment to signal the declaration of a recursive function. Then, upon lookup, a closure was created on the fly. This is the code from REC. The highlighted excerpt `Ok (ProcVal (par,body,env))` indicates that a closure is being created.

```
type env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of string*string*Ast.expr*env

let rec apply_env : string -> exp_val ea_result =
  fun id env ->
  match env with
  | EmptyEnv -> Error (id^" not found!")
  | ExtendEnv(v,ev,tail) ->
```

```
    if id=v
    then Ok ev
    else apply_env id tail
  | ExtendEnvRec(v,par,body,tail) ->
    if id=v
    then Ok (ProcVal (par,body,env))
    else apply_env id tail
```
ds.ml

We could follow the same approach in IMPLICIT-REFS, but that would require `apply_env` to access the store so that it could allocate space for the closure created on the fly. This is not very neat: why would looking up a value in the environment involve using the store? An alternative approach is simply to allow circular environments. That is, an environment `env` that has an entry which is a reference into the store that contains a closure whose environment is `env` itself.

So we first remove the special entry in environments for `letrec` declarations since they will no longer be needed:

```
type env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of string*string*Ast.expr*env

let rec apply_env : string -> exp_val ea_result =
  fun id env ->
  match env with
  | EmptyEnv -> Error (id^" not found!")
  | ExtendEnv(v,ev,tail) ->
    if id=v
    then Ok ev
    else apply_env id tail
  | ExtendEnvRec(v,par,body,tail) ->
    if id=v
    then Ok (ProcVal (par,body,env))
    else apply_env id tail
```
ds.ml

We now use "back-patching" to code the circular environment:

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | Letrec(id,par,e,target) ->
    let l = Store.new_ref g_store UnitVal in
    extend_env id (RefVal l) >>+
    (lookup_env >>= fun env ->
     Store.set_ref g_store l (ProcVal(par,e,env)) >>= fun _ ->
      eval_expr target
    )
```
interp.ml

Parenthesis right after `>>+` are necessary since `>>=` and `>>+` are left-associative. Remove them, execute the resulting interpreter on an example expression and explain what goes wrong.

3.4 Parameter Passing Methods

We consider several parameter passing methods in IMPLICIT-REFS.

3.4.1 Call-by-Value

3.4.2 Call-by-Reference

```
let x=2
in let f = proc (z) { set z = z+1 }
in begin
     (f x);
     x
   end
```

Returns `Ok (NumVal 2)` in IMPLICIT-REFS.

Modifying the Interpreter

```
let rec value_of_operand : expr -> exp_val ea_result = fun e ->
  match e with
  | Var(id) -> apply_env id
  | _ -> eval_expr e >>= fun ev ->
    return (RefVal (Store.new_ref g_store ev))
and
  apply_proc ev1 ev2 =
  ...
and
  eval_expr : expr -> exp_val ea_result = fun e ->
  match e with
  ...
  | App(e1,e2)  ->
    eval_expr e1 >>= fun v1 ->
    eval_expr e2 >>= fun v2
    value_of_operand e2 >>= fun v2 ->
    apply_proc v1 v2
```

interp.ml

```
let x=2
in let f = proc (z) { set z = z+1 }
in begin
     (f x);
     x
   end
```

Now returns `Ok (NumVal 3)`.

```
let x=2
in let y=1
in let f = proc (u) { proc (v) {
        let temp = u
        in begin
            set u = v;
            set v = temp
```

```
            end }}
in begin
     ((f x) y);
     x
  end
```

Returns `Ok (NumVal 1)`.

3.4.3 Call-by-Name

3.4.4 Call-by-Need

3.5 Exercises

Exercise 3.5.1. *Consider the following extension of LET with records (Exercise 2.2.6). It has the same syntax except that one can declare a field to be mutable by using* `<=` *instead of* `=`*. For example, the* `ssn` *field is immutable but the* `age` *field is mutable;* `age` *is then updated to 31:*

```
let p = {ssn = 10; age <= 30}
in begin
     p.age <= 31;
     p.age
   end
```

Evaluating this expression should produce `Ok (NumVal 31)`*. This other expression should produce* `Ok (RecordVal [("ssn", (false, NumVal 10)); ("age", (true, RefVal 1))])`*:*

```
let p = {ssn = 10; age <= 30}
in begin
     p.age <= 31;
     p
   end
```

Updating an immutable field should not be allowed. For example, the following expression should report an error `Error "Field not mutable"`*:*

```
let p = { ssn = 10; age = 20}
in begin
    p.age <= 21;
    p.age
   end
```

The abstract syntax requires modifying the `Record` *constructor and adding a new one for field update:*

```
type expr =
  ...
  | Record of (string*(bool*expr)) list
  | SetField of expr*string*expr
```

For example,

```
# parse "
let p = {ssn = 10; age <= 30}
in begin
```

```
      p.age <= 31;
      p
    end";;
- : expr =
Let ("p", Record [("ssn", (false, Int 10)); ("age", (true, Int 30))],
 BeginEnd [SetField (Var "p", "age", Int 31); Var "p"])
```

utop

Here `false` indicates the field is immutable and `true` that it is mutable. You are asked to implement the interpreter extension. The `RecordVal` constructor has been updated for you.

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
  | PairVal of exp_val*exp_val
  | TupleVal of exp_val list
  | RecordVal of (string*(bool*exp_val)) list
```

ds.ml

As for `eval_expr`, the case for `Record` has already been updated for you. You are asked to update `Proj` and complete `SetField`:

```
let rec  eval_expr : expr -> exp_val ea_result = fun e ->
  match e with
  | Record(fs) ->
    sequence (List.map process_field fs) >>= fun evs ->
    return (RecordVal (addIds fs evs))
  | Proj(e,id) ->
     error "update"
  | SetField(e1,id,e2) ->
     error "implement"
and
 process_field (_id,(is_mutable,e))  =
  eval_expr e >>= fun ev ->
  if is_mutable
  then return (RefVal (Store.new_ref g_store ev))
  else return ev
```

Exercise 3.5.2. *Depict the environment and store that is extant at the breakpoint.*

```
let a = 2
in let b =3
in begin
    set a = b;
    debug(a)
   end
```

Exercise 3.5.3. *Depict the environment and store that is extant at the breakpoint.*

```
let a = 2
in let b = a
in begin
    set b = 3;
    debug(a)
   end
```

Exercise 3.5.4. *Depict the environment and store that is extant at the breakpoint.*

```
let a = 2
in let b = proc(x) {
                begin
                 set a = x;
                 debug(a)
                end
            }
in (b 3)
```

Exercise 3.5.5. *Depict the environment and store that is extant at the breakpoint.*

```
let a = 2
in let b = proc(x) {
       begin
        set a = x;
        a
       end
   }
in (b 3) + debug((b 4))
```

Chapter 4

Types

This chapter extends the REC language to support type-checking.

4.1 CHECKED

4.1.1 Concrete Syntax

⟨Expression⟩	::=	⟨Number⟩
⟨Expression⟩	::=	⟨Identifier⟩
⟨Expression⟩	::=	⟨Expression⟩⟨BOp⟩⟨Expression⟩
⟨Expression⟩	::=	`zero?`(⟨Expression⟩)
⟨Expression⟩	::=	`if` ⟨Expression⟩ `then` ⟨Expression⟩ `else` ⟨Expression⟩
⟨Expression⟩	::=	`let` ⟨Identifier⟩ = ⟨Expression⟩ `in` ⟨Expression⟩
⟨Expression⟩	::=	(⟨Expression⟩)
⟨Expression⟩	::=	`proc`(⟨Identifier⟩ : ⟨Type⟩){⟨Expression⟩}
⟨Expression⟩	::=	(⟨Expression⟩⟨Expression⟩)
⟨Expression⟩	::=	`letrec`⟨Identifier⟩(⟨Identifier⟩ : ⟨Type⟩) : ⟨Type⟩=⟨Expression⟩`in`⟨Expression⟩
⟨BOp⟩	::=	+ | - | * | /
⟨Type⟩	::=	`int`
⟨Type⟩	::=	`bool`
⟨Type⟩	::=	⟨Type⟩->⟨Type⟩
⟨Type⟩	::=	(⟨Type⟩)

4.1.2 Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Sub of expr*expr
  | Let of string*expr*expr
```

```
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*texpr*expr
  | App of expr*expr
  | Letrec of string*string*texpr*texpr*expr*expr
and
 texpr =
  | IntType
  | BoolType
  | FuncType of texpr*texpr
```

4.1.3 Type-Checker

We specify the behavior of our type-checker before implementing it, much like we do with interpreters. For this task we use **type systems**. A type system is an inductive set that helps identify a subset of the expressions that are considered to be well-typed. The elements of the inductive set are called **type judgements**. Which type judgements belong to the set and which don't is determined by a sert of type rules. A type judgement is an expression of the form

$$\Gamma \vdash \texttt{e} : t$$

where Γ is a type environment, `e` is an expression in CHECKED, and t is a type expression. These components together with the type rules are introduced below.

Specification

As mentioned, a type judgement consists of a of type environment, an expression in CHECKED and a type. Types are defined as follows:

$$t \quad ::= \quad \texttt{int} \,|\, \texttt{bool} \,|\, t \to t$$

A **type context** is a partial function that assigns a type to an identifier. Type contexts are required for typing expressions that contain free variables. For example, an expression such as `x+2` will require that we have the type of `x` at our disposal in order to determine whether `x+2` is typable at all. If the type of `x` were `bool`, then it is not typable; but if the type of `x` is `int`, then it is. Type contexts are defined as follows:

$$\Gamma \quad ::= \quad \epsilon \,|\, \Gamma, id : t$$

We use ϵ to denote the empty type environment. Also, $\Gamma, id : t$ assigns type t to identifier id and behaves as Γ for identifiers different from id. We assume that Γ does not have repeated entries for the same identifier. An example of a type contexts is $\epsilon, x : \texttt{int}, y : \texttt{bool}$. We abbreviate it as $x : \texttt{int}, y : \texttt{bool}$.

The type rules are given in Figure 4.1.

Towards and Implementation

Our type checker will behave very much like our interpreter, except that instead of manipulating runtime values such as integers and booleans, it manipulates types like `int` and `bool`. One might say that a type checker is a symbolic evaluator, where our symbolic values are the types. This analogy allows us to apply the ideas we have developed on well-structuring an evaluator to our type checker. Thus one might be tempted to state the type of our type-checker as

$$\frac{}{\Gamma \vdash \texttt{n} : \texttt{int}}\ \text{TConst} \qquad \frac{\Gamma(\texttt{x}) = t}{\Gamma \vdash \texttt{x} : t}\ \text{TVar} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{int}}{\Gamma \vdash \texttt{zero?(e)} : \texttt{bool}}\ \text{TZero}$$

$$\frac{\Gamma \vdash \texttt{e1} : \texttt{int} \qquad \Gamma \vdash \texttt{e2} : \texttt{int}}{\Gamma \vdash \texttt{e1-e2} : \texttt{int}}\ \text{TDiff}$$

$$\frac{\Gamma \vdash \texttt{e1} : bool \quad \Gamma \vdash \texttt{e2} : t \quad \Gamma \vdash \texttt{e3} : t}{\Gamma \vdash \texttt{if e1 then e2 else e3} : t}\ \text{TIf}$$

$$\frac{\Gamma \vdash \texttt{e1} : t1 \quad \Gamma, \texttt{id} : t1 \vdash \texttt{e2} : t2}{\Gamma \vdash \texttt{let id=e1 in e2} : t2}\ \text{TLet}$$

$$\frac{\Gamma \vdash \texttt{rator} : t1 \to t2 \quad \Gamma \vdash \texttt{rand} : t1}{\Gamma \vdash \texttt{(rator rand)} : t2}\ \text{TApp}$$

$$\frac{\Gamma, \texttt{id} : t1 \vdash \texttt{e} : t2}{\Gamma \vdash \texttt{proc (id:t1) \{e\}} : t1 \to t2}\ \text{TProc}$$

Figure 4.1: Type Rules for CHECKED

```
type_of_expr : expr -> texpr ea_result
```

reflecting that, given an expression, it returns a function that given a type environment returns either a type or an error. Note, however, that `ea_result` abstracts environments, and not type environments:

```
type 'a ea_result = env -> 'a result
```

We could create a new type constructor, let us call it `tea_result`, where `env` is replaced with `tenv`:

```
type 'a tea_result = tenv -> 'a result
```

But we would also have to duplicate all of `return`, `error`, `>>=`, `lookup_env`, etc. to support this new type and end up having two copies of all these operations (one supporting `ea_result` and one supporting `tea_result`) with the exact same code. Since the only difference between `ea_result` and `tea_result` is the kind of environment they abstract over, we choose to define a more general type constructor `a_result` ("a" for "abstracted") and have both of these be instances of them:

```
type ('a,'b) a_result = 'b -> 'a result
```

Notice that, contrary to `ea_result` and `tea_result`, the type constructor `a_result` is parameterized over two types,

1. the type `a` representing the result of the computation, and
2. the type `b` representing that over which the function type is being abstracted over.

```
type 'a result = Ok of 'a | Error of string

type ('a,'b) a_result = 'b -> 'a result

let return : 'a -> ('a,'b) a_result =
  fun v ->
  fun env -> Ok v

let error : string -> ('a,'b) a_result =
  fun s ->
  fun env -> Error s

let (>>=) : ('a,'c) a_result -> ('a -> ('b,'c) a_result) -> ('b,'c) a_result =
  fun c f ->
  fun env ->
  match c env with
  | Error err -> Error err
  | Ok v -> f v env

let (>>+) : ('b,'b) a_result -> ('a,'b) a_result -> ('a,'b) a_result =
  fun c d ->
  fun env ->
  match c env with
  | Error err -> Error err
  | Ok newenv -> d newenv
```
reM.ml

Figure 4.2: The `a_result` type

The type `('a,'b) a_result`, together with its supporting operations are declared in Figure 4.2. Notice that the code for the supporting operations, namely `return`, `error`, `>>=` and `>>+` is exactly the same as before. The only difference is their type. That being said, we still call the formal parameter of type `'b` in these operations, `env`.

With the newly declared type constructor `a_result` in place, we can now redefine `ea_result` and `tea_result` as instances of it. Indeed, `ea_result` is simply defined as:

```
type 'a ea_result = ('a,env) a_result
```

and `tea_result` is defined as:

```
type 'a tea_result = ('a,tenv) a_result
```

Implementation

We next address the implementation of the type-checker for CHECKED. The code is given in Figure 4.3.

```
utop # chk "
let add = proc (x:int) { proc (y:int) { x+y }} in (add 1)";;
- : texpr ReM.result = Ok (FuncType (IntType, IntType))
```
utop

```
let rec type_of_expr : expr -> texpr tea_result =
  fun e ->
  match e with
  | Int _n -> return IntType
  | Var id -> apply_tenv id
  | IsZero(e) ->
    type_of_expr e >>= fun t ->
    if t=IntType
    then return BoolType
    else error "isZero: expected argument of type int"
  | Add(e1,e2) | Sub(e1,e2) | Mul(e1,e2)| Div(e1,e2) ->
    type_of_expr e1 >>= fun t1 ->
    type_of_expr e2 >>= fun t2 ->
    if (t1=IntType && t2=IntType)
    then return IntType
    else error "arith: arguments must be ints"
  | ITE(e1,e2,e3) ->
    type_of_expr e1 >>= fun t1 ->
    type_of_expr e2 >>= fun t2 ->
    type_of_expr e3 >>= fun t3 ->
    if (t1=BoolType && t2=t3)
    then return t2
    else error "ITE: condition not bool/types of then-else do not match"
  | Let(id,e,body) ->
    type_of_expr e >>= fun t ->
    extend_tenv id t >>+
    type_of_expr body
  | Proc(var,t1,e) ->
    extend_tenv var t1 >>+
    type_of_expr e >>= fun t2 ->
    return (FuncType(t1,t2))
  | App(e1,e2) ->
    type_of_expr e1 >>=
    pair_of_funcType "app: " >>= fun (t1,t2) ->
    type_of_expr e2 >>= fun t3 ->
    if t1=t3
    then return t2
    else error "app: type of argument incorrect"
  | Debug(_e) ->
    string_of_tenv >>= fun str ->
    print_endline str;
    error "Debug: reached breakpoint"
  | _ -> error "type_of_expr: implement"
```

checker.ml

Figure 4.3: Type checker for CHECKED

4.1.4 Adding Letrec

The typing rule for `letrec` is as follows:

$$\dfrac{\begin{array}{l}\Gamma, \texttt{param} : tVar, \texttt{id} : tPar \to \texttt{tRes} \vdash \texttt{body} : tRes \\ \Gamma, \texttt{id} : tPar \to tRes \vdash \texttt{target} : t\end{array}}{\Gamma \vdash \texttt{letrec id (param:tPar):tRes = body in target} : t}\ \text{TRec}$$

```
let rec type_of_expr : expr -> texpr tea_result = function
  | Letrec(id,param,tPar,tRes,body,target) ->
    extend_tenv id (FuncType(tPar,tRes)) >>+
    (extend_tenv param tPar >>+
    type_of_expr body >>= fun t ->
    if t=tRes
    then type_of_expr target
    else error "LetRec: Type of rec. function does not match declaration")
```

4.1.5 Exercises

Exercise 4.1.1. *Provide typing derivations for the following expressions:*

1. `if zero?(8) then 1 else 2`
2. `if zero?(8) then zero?(0) else zero?(1)`
3. `proc (x:int) { x-2 }`
4. `proc (x:int) { proc (y:bool) { if y then x else x-1 } }`
5. `let x=3 in let y = 4 in x-y`
6. `let two? = proc(x:int) { if zero?(x-2) then 0 else 1 } in (two? 3)`

Exercise 4.1.2. *Recall that an expression `e` is typable, if there exists a type environment Γ and a type expression `t` such that the typing judgement $\Gamma \vdash \texttt{e} : t$ is derivable. Argue that the expression `x x` (a variable applied to itself) is not typable.*

Exercise 4.1.3. *Give a typable term of each of the following types, justifying your result by showing a type derivation for that term.*

1. `bool->int`
2. `(bool -> int) -> int`
3. `bool -> (bool -> bool)`
4. `(s -> t) -> (s -> t)`, *for any types `s` and `t`.*

Exercise 4.1.4. *Show that the following term is typable:*

```
letrec int double (x:int) = if zero?(x)
                            then 0
                            else (double (x-1)) + 2
in double
```

Exercise 4.1.5. *What is the result of evaluating the following expressions in CHECKED?*

```
> (check "
letrec int double (x:int) = if zero?(x)
                               then 0
                               else (double (x-1)) + 2
in (double 5)")
```

```
> (check "
letrec int double (x:int) = if zero?(x)
                               then 0
                               else (double (x-1)) + 2
in double")
```

```
> (check "
letrec bool double (x:int) = if zero?(x)
                               then 0
                               else -((double -(x,1)), -2)
in double")
```

```
> (check "
letrec bool double (x:int) = if zero?(x)
                               then 0
                               else 1
in double")
```

```
> (check "
letrec int double (x:bool) = if zero?(x)
                               then 0
                               else 1
in double")
```

Exercise 4.1.6. *Consider the extension of Exercise 2.2.4 where pairs are added to our language. In order to extend type-checking to pairs we first add pair types to the concrete syntax of types:*

<*Type*>	::=	`int`
<*Type*>	::=	`bool`
<*Type*>	::=	<*Type*> `->` <*Type*>
<*Type*>	::=	`<`<*Type*> `*` <*Type*>`>`
<*Type*>	::=	`(` <*Type*>`)`

Recall from Exercise 2.2.4 that expressions are extended with a `pair(e1,e2)` *construct to build new pairs and an* `unpair (x,y)=e1 in e2` *construct that given an expression* `e1` *that evaluates to a pair, binds* `x` *and* `y` *to the first and second component of the pair, respectively, in* `e2`*. Here are some examples of expressions in the extended language:*

```
pair(3,4)

pair(pair(3,4),5)

pair(zero?(0),3)

```

```
pair(proc (x:int) { x-2 },4)

proc (z:<int*int>) { unpair (x,y)=z in x }

proc (z:<int*bool>) { unpair (x,y)=z in pair(y,x) }
```

You are asked to give typing rules for each of the two new constructs.

Chapter 5

Modules

5.1 Syntax

SIMPLE-MODULES is an extension to the EXPLICIT-REFS language. A program in SIMPLE-MODULES consists of a list of module declarations together with an expression (the "main" expression). Here is an example that consists of one module declaration, the module called `m1`, and a main expression consisting of a let expression. A module has an interface and a body.

```
module m1
  interface
   [a : int
    b : int
    c : int]
  body
   [a = 33
    x = a-1   (* =32 *)
    b = a-x   (* = 1 *)
    c = x-b] (* =31 *)
let a = 10
in ((from m1 take a) - (from m1 take b))-a
```

5.1.1 Concrete Syntax

⟨Program⟩	::=	{⟨ModuleDefn⟩}* ⟨Expression⟩
⟨ModuleDefn⟩	::=	`module` ⟨Identifier⟩ `interface` ⟨Iface⟩ `body` ⟨ModuleBody⟩
⟨Iface⟩	::=	[{⟨Decl⟩}*]
⟨Decl⟩	::=	⟨Identifier⟩ : ⟨Type⟩
⟨ModuleBody⟩	::=	[{⟨Defn⟩}*]
⟨Defn⟩	::=	⟨Identifier⟩ = ⟨Expression⟩
⟨Expression⟩	::=	$...asbefore...$
⟨Expression⟩	::=	`from`⟨Identifier⟩`take`⟨Identifier⟩

5.1.2 Abstract Syntax

```
type expr =
  ...
  | QualVar of string*string
and
  texpr =
  | IntType
  | BoolType
  | UnitType
  | FuncType of texpr*texpr
  | RefType of texpr
and
  vdecl = string*texpr
and
  vdef = string*expr

type interface = ASimpleInterface of vdecl list
type module_body = AModBody of vdef list
type module_decl = AModDecl of string*interface*module_body
type prog = AProg of (module_decl list)*expr
```

5.2 Interpreter

5.2.1 Specification

$$\frac{\texttt{mdecls}, \epsilon \Downarrow \rho \quad \texttt{e}, \rho \Downarrow v}{\texttt{AProg(mdecls,e)} \Downarrow v}\ \text{EProg}$$

$$\frac{}{\epsilon, \rho \Downarrow \rho}\ \text{EMDeclsEmpty}$$

$$\frac{\texttt{body}, \rho \Downarrow \sigma \quad \texttt{ms}, \rho \oplus \{\texttt{id} := \sigma\} \Downarrow \tau}{\texttt{AModDecl(id,iface,body)}\ \texttt{ms}, \rho \Downarrow \tau}\ \text{EMDeclsCons}$$

$$\frac{}{\epsilon, \rho \Downarrow \rho}\ \text{EBValsEmpty}$$

$$\frac{\texttt{e}, \rho \Downarrow v \quad \texttt{vs}, \rho \oplus \{\texttt{id} := v\} \Downarrow \tau}{\texttt{(id,e)}\ \texttt{vs}, \rho \Downarrow \tau}\ \text{EBValsCons}$$

$$\frac{\rho(\texttt{mid}) = \sigma \quad \sigma(\texttt{vid}) = v}{\texttt{QualVar(mid,vid)}, \rho \Downarrow v}\ \text{EQualVar}$$

5.2.2 Implementation

We first extend environments to support bindings for modules:

```
type env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of string*string*Ast.expr*env
  | ExtendEnvMod of string*env*env
```

ds.ml

Evaluation of programs consists in first evaluating all module definitions producing an environment as a result, and then evaluating the main expression using this environment. The former is achieved with the function `eval_module_definitions : module_decl list -> env ea_result`:

```
let eval_prog (AProg(ms,e)) : exp_val ea_result =
   eval_module_definitions ms >>+
   eval_expr e
```

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | Int(n) ->
    return (NumVal n)
  | Var(id) ->
    apply_env id
  | QualVar(module_id,var_id) ->
    apply_env_qual module_id var_id
```

```
  ...
and
  eval_module_definition : module_body -> env ea_result =
  fun (AModBody vdefs) ->
  lookup_env >>= fun glo_env ->
  (List.fold_left (fun loc_env (var,decl)  ->
       loc_env >>+
       (append_env_rev glo_env >>+
        eval_expr decl >>=
        extend_env var))
      (empty_env ())
      vdefs)
and
  eval_module_definitions : module_decl list -> env ea_result =
  fun ms ->
  List.fold_left
    (fun curr_en (AModDecl(mname,minterface,mbody)) ->
       curr_en >>+
       (eval_module_definition mbody >>=
        extend_env_mod mname))
    lookup_env
    ms
 and
   eval_prog (AProg(ms,e)) : exp_val ea_result =
   eval_module_definitions ms >>+
   eval_expr e
```

interp.ml

```
utop # interp "
 module m1 interface
[u : int] body
[u = 44]
module m2 interface
[v : int] body
[v = (from m1 take u)-11]
let a=zero?(0)
in debug(0)";;
Environment:
(a,BoolVal true)
Module m2[(v,NumVal 33)]
Module m1[(u,NumVal 44)]
Store:
Empty
- : Ds.exp_val ReM.result = ReM.Ok Ds.UnitVal
```

utop

5.3 Type-Checking

5.3.1 Specification

Judgements for typing programs	$\vdash$ `AProg(ms,e)` $: t$
Judgements for typing expressions	$\Delta;\Gamma \vdash$ `e` $: t$
Judgements for typing list of module declarations	$\Delta_1 \vdash$ `ms` $: \Delta_2$

$$\frac{\epsilon \vdash \texttt{ms} :: \Delta \quad \Delta;\epsilon \vdash \texttt{e} :: t}{\vdash \texttt{AProg(ms,e)} :: t}\ \text{TProg}$$

$$\frac{\texttt{m} \in \mathsf{dom}(\Delta) \quad x \in \mathsf{dom}(\Delta(\texttt{m})) \quad \Delta(\texttt{m},\texttt{x}) = \texttt{t}}{\Delta;\Gamma \vdash \texttt{from m take x} :: t}\ \text{TFromTake}$$

$$\frac{}{\Delta \vdash \epsilon :: \epsilon}\ \text{TModE}$$

$$\frac{\begin{array}{c}[x_i]_{i\in I} \lhd [y_j]_{j\in J} \\ (\Delta_1;[y_1 := s_1]\ldots[y_{j-1} := s_{j-1}]\Gamma \vdash e_j :: s_j)_{j\in J} \\ (t_i = s_{f(i)})_{i\in I} \\ \texttt{m}[x_i : t_i]_{i\in I}\Delta_1 \vdash \texttt{ms} :: \Delta_2\end{array}}{\Delta_1 \vdash \texttt{m}[x_i : t_i]_{i\in I}[y_j = e_j]_{j\in J}\texttt{ms} :: \texttt{m}[x_i : t_i]_{i\in I}\Delta_2}\ \text{TModNE}$$

Figure 5.1: Typing rules for SIMPLE-MODULES

Γ is the standard type environment from before Δ is a module type environment and is required for typing the expression `from m take x`

A **module type** is an expression of the form $\texttt{m}[u_1 : t_1, \ldots, u_n : t_n]$. A **module type environment** is a sequence of module types.

$$\Delta \quad ::= \quad \epsilon \,|\, \texttt{m}[u_1 : t_1, \ldots, u_n : t_n]\ \Delta$$

We use letters Δ to denote module type environments. The empty module type environment is written ϵ. If $\texttt{m}[u_1 : t_1, \ldots, u_n : t_n] \in \Delta$, then we $\texttt{m} \in \mathsf{dom}(\Delta)$. Moreover, in that case, have $u_i \in \mathsf{dom}(\Delta(\texttt{m}))$, for $i \in 1..n$, and also $\Delta(\texttt{m}, u_i) = t_i$.

There is just one typing rule for typing programs, namely TProg. There is one new typing rule for expressions, namely TFromTake, it allows to type qualified variables. There are two typing rules for lists of module definitions: one for when the list is empty (TModE) and one for when it is not (TModNE). Regarding the latter,

- Δ_2 is the type of the list of modules `ms`
- $\texttt{m}[x_i : t_i]_{i\in I}\Delta_1$ is the type of the list of modules that `ms` can use
- $[x_i]_{i\in I} \lhd [y_j]_{j\in J}$ means that the list of variables $[x_i]_{i\in I}$ is a sublist of the list of variables $[y_j]_{j\in J}$. This relation determines an injective, order preserving function $f : I \to J$

5.3.2 Implementation

```
type tenv =
  | EmptyTEnv
  | ExtendTEnv of string*texpr*tenv
  | ExtendTEnvMod of string*tenv*tenv
```
dst.ml

```
let rec
  type_of_prog (AProg (ms,e)) =
  type_of_modules ms >>+
  type_of_expr e
and
  type_of_modules : module_decl list -> tenv tea_result =
  fun mdecls ->
  List.fold_left
    (fun curr_tenv (AModDecl(mname,ASimpleInterface(expected_iface),mbody)) ->
      curr_tenv >>+
      (type_of_module_body mbody >>= fun i_body ->
      if (is_subtype i_body expected_iface)
      then
        extend_tenv_mod mname (var_decls_to_tenv expected_iface)
      else
        error("Subtype failure: "^mname))
    )
    lookup_tenv
    mdecls
and
  type_of_module_body : module_body -> tenv tea_result =
  fun (AModBody vdefs) ->
  lookup_tenv >>= fun glo_tenv ->
  (List.fold_left (fun loc_tenv (var,decl)  ->
       loc_tenv >>+
       (append_tenv_rev glo_tenv >>+
        type_of_expr decl >>=
        extend_tenv var))
      (empty_tenv ())
      vdefs) >>= fun tmbody ->
  return (reverse_tenv tmbody)
and
  type_of_expr : expr -> texpr tea_result =
  fun e ->
  match e with
  | Int n -> return IntType
  | Var id -> apply_tenv id
  | QualVar(module_id,var_id) ->
    apply_tenv_qual module_id var_id
  ...
```

5.4 Further Reading

Module inclusion Private types First-class modules

Appendix A

Supporting Files

A.1 File Structure

Typical file structure for an interpreter (in this example, ARITH).

```
.
|____arith.opam
|____test
| |____dune
| |____test.ml
|____dune-project
|_____build
|____.ocamlinit
|____src
| |____ds.ml
| |____ast.ml
| |____.merlin
| |____lexer.mll
| |____dune
| |____interp.ml
| |____parser.mly
```

The source files are in the `src` directory and the unit tests are in the `test` directory.

`ast.ml`	Abstract Syntax
`ds.ml`	Supporting data structures including expressed values, environments and results
`interp.ml`	Interpreter
`lexer.mll`	Lexer generator
`parser.mly`	Parser generator
`test.ml`	Unit tests
`.ocamlinit`	Loaded by utop upon execution; opens some modules

We use the dune build system for OCaml. You can find documentation on dune at `https://readthedocs.org/projects/dune/downloads/pdf/latest/`. Some common dune commands:

- Builds the project and then runs utop

```
$ dune utop
```

- Builds the project

```
$ dune build
```

- Clean the current project (erasing `_build` directory)

```
$ dune clean
```

- Run tests (building if necessary)

```
$ dune runtest
```

Appendix B

Solution to Selected Exercises

Section 2.1

Answer B.0.1 (Exercise 2.2.1). *Sample expressions of each of the following types are:*

1. `expr`. *An example is:* `Int 2`.
2. `env`. *Examples are:* `EmptyEnv` *and* `ExtendEnv("x",NumVal 2,EmptyEnv)`
3. `exp_val`. *Examples are:* `NumVal 3` *and* `BoolVal true`.
4. `exp_val result`. *Examples are:* `Ok (NumVal 3)` *and* `Ok (BoolVal true)` *and* `Error "oops"`.
5. `int result`. *Examples are:* `Ok 1` *and* `Error "oops"`.
6. `env result`. *Examples are:* `Ok EmptyEnv` *and* `Ok (ExtendEnv("x",NumVal 2,EmptyEnv))`.
7. `int ea_result`. *Examples are:* `return 2` *and* `error "oops"`.
8. `exp_val ea_result`. *Examples are:* `return (NumVal 2)` *and* `return (BoolVal true)` *and* `error "oops"`. *Also,* `apply_env "x"`.
9. `env ea_result`. *Examples are:* `return (EmptyEnv)` *and* `error "oops"`. *Also,* `extend_env "x" (NumVal 7)`.

Section 2.3

Answer B.0.2 (Exercise 2.4.7).

```
let even = proc (e) { proc (o) { proc (x) {
    if zero?(x)
    then zero?(0)
    else (((o e) o) (x-1)) }}}
in
let odd = proc(e) { proc (o) { proc (x) {
    if zero?(x)
    then zero?(1)
    else (((e e) o) (x-1)) }}}
in (((even even) odd) 4)
```

Answer B.0.3 (Exercise 2.4.8)**.**

```
let f =
  proc (g) {
    proc (leaf) {
      proc (depth) {
        if zero?(depth) then (leaf,leaf)
        else ((((g g) leaf) (depth-1)),(((g g) leaf) (depth-1))) }}}
in let pbt = proc (leaf) {  proc (height) { (((f f) leaf) height) }}
in ((pbt 2) 3)
```

Section 2.5

Answer B.0.4 (Exercise 2.5.2)**.**

```
# interp "
let l1 = cons(1,cons(2,cons(3,emptylist)))
in let l2 = cons(4,cons(5,emptylist))
in letrec append(l1) = proc (l2) {
                          if empty?(l1)
                          then l2
                          else cons(hd(l1),((append tl(l1)) l2))
                        }
in ((append l1) l2)
";;
- : exp_val Rec.Ds.result =
Ok (ListVal [NumVal 1; NumVal 2; NumVal 3; NumVal 4; NumVal 5])


# interp "
let l = cons(1,cons(2,cons(3,emptylist)))
in let succ = proc (x) { x+1 }
in letrec map(l) = proc (f) {
                      if empty?(l)
                      then emptylist
                      else cons((f hd(l)),((map tl(l)) f))
                    }
in ((map l) succ)
";;
- : exp_val Rec.Ds.result = Ok (ListVal [NumVal 2; NumVal 3; NumVal
4])


# interp "
let l = cons(1,cons(2,cons(1,emptylist)))
in let is_one = proc (x) { zero?(x-1) }
in letrec filter(l) = proc (p) {
                        if empty?(l)
                        then emptylist
                        else (if (p hd(l))
                              then cons(hd(l),((filter tl(l)) p))
                              else ((filter tl(l)) p))
                      }
in ((filter l) is_one)";;
- : exp_val Rec.Ds.result = Ok (ListVal [NumVal 1; NumVal 1])
```

utop

Answer B.0.5 (Exercise 2.5.3). *1.* `debug` *Must be placed in the* `then` *case:*

```
let z = 0
in let prod = proc (x) { proc (y) { x*y }}
in letrec f(n) = if zero?(n) then debug(1) else ((prod n) (f (n-1)))
in (f 10)
```

2. Two possible solutions are:

```
let z = 0
in debug(let prod = proc (x) { proc (y) { x*y }}
in letrec f(n) = if zero?(n) then 1 else ((prod n) (f (n-1)))
in (f 10))
```

or:

```
let z = 0
in let prod = debug(proc (x) { proc (y) { x*y }})
in letrec f(n) = if zero?(n) then 1 else ((prod n) (f (n-1)))
in (f 10)
```

3. `debug` *must be placed in the body of* `proc (x)`*:*

```
let z = 0
in let prod = proc (x) { debug(proc (y) { x*y })}
in letrec f(n) = if zero?(n) then 1 else ((prod n) (f (n-1)))
in (f 10)
```