Programming Language Fundamentals (PLaF) Notes

v0.116 – April 13, 2024

Eduardo Bonelli

Contents

Preface

These course notes provide supporting material for CS496 and CS510. The most recent version of these notes...

They draw from ideas in [FW08] but have been evolving over the semesters.

Chapter 1

Inductive Definitions

1.1 Judgements

A **judgement** is a statement about one or more objects. For example:

$$n \ \mathsf{nat}$$

is a judgement that states that the object n is a natural number. We call the object n the **subject** of the statement. This statement is understood as a predicate: it may or may not be the case that the judgement n nat holds. Whether a judgement holds or not is determined by a set of **derivation rules**. For example, the derivation rules for the judgement n nat are:

$$\frac{}{zero \ \mathsf{nat}}\ \mathsf{NatZ} \qquad \frac{n \ \mathsf{nat}}{succ(n) \ \mathsf{nat}}\ \mathsf{NatS} \tag{1.1}$$

The derivation rule NatZ states that the judgement $zero$ nat holds. The derivation rule NatS states that the judgement $succ(n)$ nat holds, if the judgement n nat holds. The judgement below the horizontal line in a derivation rule is its <u>conclusion</u>; the judgements above the horizontal line are its <u>hypothesis</u>. A derivation rule always has a single conclusion but may have zero or more hypothesis. A derivation rule that has no hypothesis is often called an <u>axiom</u>. In our example, NatZ is an axiom.

A judgement is **derivable** if there is a derivation of it. A derivation is a tree whose nodes are instances of derivation rules and whose leaves are instances of axioms. The following is a sample derivation of the judgement $succ(succ(zero))$ nat:

$$\frac{\dfrac{\dfrac{}{zero \ \mathsf{nat}}\ \mathsf{NatZ}}{succ(zero) \ \mathsf{nat}}\ \mathsf{NatS}}{succ(succ(zero)) \ \mathsf{nat}}\ \mathsf{NatS}$$

Another example of a judgement is e expr, which is defined together with the judgement n int. The former may be seen to describe a set of well-formed arithmetic expressions. An expression that is missing an operand (*e.g.* "3-"), or one that involves symbols not denoting integers (*e.g.* "1a+2"), or one that has unbalanced parenthesis (*e.g.* "(1+2"), are examples of expressions that are considered not to be well-formed.

$$\frac{n \in \mathbb{Z}}{n \text{ int}}\,\mathsf{Int} \quad \frac{n \text{ int}}{n \text{ expr}}\,\mathsf{ExpInt}$$
$$\frac{e1 \text{ expr} \quad e2 \text{ expr}}{e1 - e2 \text{ expr}}\,\mathsf{ExpSub} \quad \frac{e1 \text{ expr} \quad e2 \text{ expr}}{e1/e2 \text{ expr}}\,\mathsf{ExpDiv} \tag{1.2}$$
$$\frac{e \text{ expr}}{(e) \text{ expr}}\,\mathsf{ExpPar}$$

One may show, for example, that the judgement $((4/0)-4)$ expr holds by providing a derivation of it using the above derivation rules.

Judgements that are used to define well-formed expressions have an alternative, more specialized presentation called grammars. We will introduce the grammar based presentation of Example 1.2 in Chapter 2.

1.2 Inductive Sets

The set of derivation rules for one or more judgements constitutes an **inductive definition**. An inductive definition specifies an **inductive set**. The rules in Example 1.1 and Example 1.2 are examples of inductive definitions. The inductive set that they specify are sets of judgements, namely those that are derivable, as described in further detail below. Alternatively, the inductive set that they specify may be characterized as the smallest set $\mathcal{A}$ of judgments such that:

1. For every axiom:

   $$\frac{}{J}$$

   we have $J \in \mathcal{A}$.

2. For every rule:

   $$\frac{J_1 \dots J_n}{J}$$

   we have $J \in \mathcal{A}$, if $\{J_1, \dots, J_n\} \subseteq \mathcal{A}$.

For example, the inductive set $\mathcal{A}$ defined by the inductive definition of Example 1.1 is

$$\{zero \text{ nat}, succ(zero) \text{ nat}, succ(succ(zero)) \text{ nat}, \dots\}$$

Iterated inductive definitions are those that are built on top of others. They determine a set of **iterated inductive sets**. For example, consider the following inductive definition consisting of a set of rules deriving judgements of the form l listNat representing lists of natural numbers. It is built on top of Example 1.1 in the sense that it assumes the former to have been given.

$$\frac{}{nil\ \mathsf{listNat}}\ \mathsf{ListNatN} \qquad \frac{n\ \mathsf{nat} \quad l\ \mathsf{listNat}}{cons(n,l)\ \mathsf{listNat}}\ \mathsf{ListNatC} \tag{1.3}$$

In fact, Example 1.2, presented above is also an example of an iterated inductive set.

Simultaneous inductive definitions are those in which two or more judgements are defined at once. The simplest example is the simultaneous definition of the judgements n even and n odd.

$$\frac{}{zero\ \mathsf{even}}\ \mathsf{EvenZ} \qquad \frac{n\ \mathsf{odd}}{succ(n)\ \mathsf{even}}\ \mathsf{EvenS} \qquad \frac{n\ \mathsf{even}}{succ(n)\ \mathsf{odd}}\ \mathsf{OddS} \tag{1.4}$$

We will introduce various judgements in these notes, including the following:

$\mathsf{e}, \rho \Downarrow r$	Evaluation judgement
$\Gamma \vdash \mathsf{e} : t$	Typing judgement

1.3 Notes

A undergraduate level text with many examples is [FS91]. A thorough approach is [Acz77].

Chapter 2

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 and how its programs are executed.

2.1 Syntax

This section presents the syntax of ARITH. The syntax determines what sequences of symbols which make up our code counts as a syntactically correct program. The syntax is typically presented in the form of a grammar and referred to as the concrete syntax. There are many details in the concrete syntax that are irrelevant for executing a program. For example, an expression such as `4 / 2` might denote a program that divides 4 by 2. But one may also use an expression such as `4 div 2` to denote the same program. Or perhaps the expression `div 4 2`. Which of these two is considered syntactically correct is determined by the concrete syntax. In order to execute the program, all we need to know is that 4 is being divided by 2, regardless of how the language requires you to write the division operator itself. The abstract syntax of a language is the underlying representation of a syntactically correct program, once we abstract away any inessential, concrete details. It typically takes the form of a tree and is referred to as an Abstract Syntax Tree. A program called a parser, receives a sequence of symbols and determines whether it conforms to the rules of the concrete syntax. If it doesn't it reports a "syntax error"; if it does, it produces an abstract syntax tree. We next address these topics in further detail for ARITH.

2.1.1 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, ⟨Expression⟩ and ⟨Number⟩. Among all non-terminals one singles out the so called **start non-terminal**. In our case, the start non-terminal is ⟨Expression⟩. 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 ⟨Number⟩ non-terminal. Such non-terminals are known as tokens and are specified outside the grammar, typically by means of regular expressions; the sequence of symbols matching the regular expressions are known as lexemes. In our particular example, the sequence of symbols identified as ⟨Number⟩ shall be either a sequence of digits (*e.g.* `123`) or a sequence of digits prefixed with a "-" and surrounded by parenthesis (*e.g.* `(-123)`). For the sake of simplicity, our language ARITH only supports two arithmetic operations, subtraction and division. We will later add others.

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

> An equivalent presentation of this grammar but based on judgements was given in Example 1.2. Concrete syntax for programming languages are traditionally presented in terms of grammars, not judgements.

Examples of syntactically correct expressions are `3-4`, `((4/0)-4)`, `3-4-1`, and `(-4)/2`. That each of these sequence of terminals is a syntactically correct expression, may be justified by providing a derivation of the sequence of terminals from the non-terminal ⟨Expression⟩. For example, a derivation of `3-4` is:

$$
\begin{array}{lcl}
\langle\text{Expression}\rangle & \Rightarrow & \langle\text{Expression}\rangle\texttt{-}\langle\text{Expression}\rangle \\
& \Rightarrow & \langle\text{Number}\rangle\texttt{-}\langle\text{Expression}\rangle \\
& \Rightarrow & 3\texttt{-}\langle\text{Expression}\rangle \\
& \Rightarrow & 3\texttt{-}\langle\text{Number}\rangle \\
& \Rightarrow & 3\texttt{-}4
\end{array}
$$

Each step of the derivation results from unfolding a production of the grammar given above. Examples of syntactically incorrect expressions are `3--4`, `4 div 2`, and `3-()`.

2.1.2 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 `prog`[1]. A value of type `prog` is an expression of the form `AProg(cs,e)` where `cs` is a list of class declarations and `e` is an expression of type `expr`. As discussed below, `cs` will be empty for now. Indeed, we shall only be interested in `e` for the moment:

```
type
  prog = AProg of (cdecl list)*expr
and
  expr =
  | Int of int
  | Sub of expr*expr
  | Div of expr*expr
```

[1] The types `prog` and `expr` are examples of variant types in OCaml.

parser_plaf/lib/ast.ml

This figure is an example of a code listing. It occasionally includes an indication as to where the snippet of code resides. For example, in this case it resides in file parser_plaf/lib/ast.ml.

The `parse` function parses a string. If the string obeys the rules of the concrete syntax, it produces an associated abstract syntax tree; otherwise it reports an error. For example,

```
# parse "1+2*3";;
- : prog = AProg ([], Add (Int 1, Mul (Int 2, Int 3)))
```

utop

Notice that `parse` has type `string->prog` rather than `string->expr`. As mentioned above, a program is an expression of the form `AProg(cs,e)`, where `cs` is a list of class declarations and `e` is an expression of type `expr`. We shall ignore class declarations for now and discuss them in detail in Chapter 6. In the meantime, our programs will always have the form `AProg([],e)` and we shall only be interested in evaluating the expression `e`.

2.2 Interpreter

An interpreter[2] 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 consists in first providing a precise description of the possible results which a program can evaluate to. Then introducing evaluation judgements that state what result a program evaluates to. Finally, a set of evaluation rules is introduced 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.

2.2.1 Specification

We begin by stating the possible results that can arise out of the evaluation of programs in ARITH. The set of **results** is the integers $\mathbb{Z}$ or a special element $error$. This reflects that evaluation of ARITH programs produce integers or result in an error (from division by zero):

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

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

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

$$\frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow n \quad p = m - n}{\texttt{Sub(e1,e2)} \Downarrow p}\ \textsf{ESub} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow n \quad n \neq 0 \quad p = m/n}{\texttt{Div(e1,e2)} \Downarrow p}\ \textsf{EDiv}$$

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

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

Figure 2.1: Evaluation rules for ARITH

The ":=" symbol is used for definitional equality, meaning here that $\mathbb{R}$ is "defined to be" the set $\mathbb{Z}$ of integers or the special symbol error. The subset of results that are integers are called expressed values: it is the name given to non-error results of evaluation. We continue with the specification of the interpreter for ARITH by introducing evaluation judgements in the sense introduced in Chapter 1. **Evaluation judgements** are expressions of the form:

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

where e is an expression in ARITH in abstract syntax and r is a result. Evaluation judgements are read as, "expression e evaluates to the result r". The meaning of $\texttt{e} \Downarrow r$ is established via so called **evaluation rules**. The set of evaluation rules of ARITH are as presented in Figure 2.1, where $m, n, p \in \mathbb{Z}$.

Recall from Chapter 1 that 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[3].

The first three rules determine the behavior of expressions where no errors occur during evaluation. 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. As discussed in Chapter 1, some judgements are derivable and some are not.

Example 2.2.1. *The evaluation judgements* $\texttt{Sub(Div(Int 8,Int 0),Int 1)} \Downarrow \mathit{error}$ *and* $\texttt{Sub(Div(Int 4,Int 2),Int 1)} \Downarrow 2$ *are derivable. A derivation for the latter is:*

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

$$\dfrac{\dfrac{}{\texttt{Int 4} \Downarrow 4}\,\text{EInt} \quad \dfrac{}{\texttt{Int 2} \Downarrow 2}\,\text{EInt}}{\texttt{Div(Int 4,Int 2)} \Downarrow 2}\,\text{EDiv} \quad \dfrac{}{\texttt{Int(1)} \Downarrow 1}\,\text{EInt} \quad 1 = 2 - 1}{\texttt{Sub(Div(Int 4,Int 2),Int 1)} \Downarrow 1}\,\text{ESub}$$

An example of an evaluation judgement that is not derivable is `Sub(Int 3,Int 1)` $\Downarrow 1$.

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.

> A summary of some important concepts we have covered are listed below. Make sure you look them up above:
>
> - Result
> - Expressed Value
> - Evaluation Judgement
> - Evaluation Rules
> - Derivation Tree
> - Derivable Evaluation Judgements

2.2.2 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 e $\Downarrow r$. The former is already expressed in abstract syntax, which we encoded as the variant 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[4]:

```
type 'a result = Ok of 'a | Error of string
```
ds.ml

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`.

> 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`.

[4]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 2.2: Preliminary Interpeter for ARITH

2.2.2.1 Preliminary

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 2.2. The `eval_expr` function is defined by recursion 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[5]. A similar comment applies to the other arithmetic operation. One notices 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.

[5] 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 2.3: The Error Monad

2.2.2.2 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 2.3. 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[6]. Consider the expression `c >>= f`; its behavior may be described as follows:

1. evaluate the argument `c` to produce a result (*i.e.* a non-error value or an error value); if `c` returns an error, propagate it and conclude.
2. otherwise, if `c` returns `Ok v`, for some expressed value `v`, then pass `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.

[6] 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

- `error` may be seen as a function that creates a (trivial) structured program that returns an error result.
- `(>>=)` may be seen as a means of composing structured programs. In `c >>= f`, structured program `c` is composed with structured program `f v`, where `v` is the non-error result of `c`. If `c` produces an error, then evaluation of `f` is skipped.

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)))
```

interp.ml

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.

2.3 Exercises

Exercise 2.3.1. *For each of the following sequence of terminals, write a derivation that shows that it belongs to the grammar generated by the nonterminal* ⟨Expression⟩*:*

1. *(3-4)*
2. *((4/0)-4)*
3. *3-4-1*

Exercise 2.3.2. *What is the difference between a result and an expressed value?*

Exercise 2.3.3. *Consider the extension of ARITH with a new expression that returns the absolute value of the value of its argument. The concrete syntax of ARITH is extended with the production:*

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

and the abstract syntax with a new constructor:

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

parser_plaf/lib/ast.ml

1. *Do the set of results in this extended language need to be modified? Think about whether new errors are introduced by the* `abs` *construct or whether new kinds of expressed values (other than integers) are possible.*
2. *Do evaluation judgements need to be modified? Think about whether evaluation judgements still have the form* $e \Downarrow r$*, for* e *an expression in the extended language and* r *a result.*
3. *Add the two evaluation rules for the new language construct* `abs(e)`*. You may assume that* abs *is the name of the mathematical function that returns the absolute value of an integer.*

4. *Extend the interpreter* `eval_expr` *to handle this case.*

Exercise 2.3.4. *Consider the extension of ARITH with a new expression that returns the minimum number between the values of its two arguments. The concrete syntax of ARITH is extended with the production:*

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

and the abstract syntax with a new constructor:

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

parser_plaf/lib/ast.ml

The set of expressed values and the evaluation judgements remain unaltered. The new evaluation rules describing its behavior are defined as expected.

Extend the interpreter (`src/arith/lib/interp.ml`*) to support this new language extension. For example, evaluation of* `interp "3-min(1,2)"` *should return* `Ok 2`*.*

Chapter 3

Simple Functional Languages

3.1 LET

3.1.1 Concrete Syntax

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

Note that we have not specified what terminals are generated by the ⟨Identifier⟩ non-terminal; we will assume these to be sequences of symbols 'a'-'z', 'A'-'Z', '0'-'9', or '_' that start with a letter.

3.1.2 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
```

3.1.3 Environments

Consider the LET expression `x+2`. Variables such as `x` are referred to as identifiers. We cannot determine the result of evaluating this expression 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[2], 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.

3.1.4 Interpreter

3.1.4.1 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:

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

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 3.1. 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

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

[2] The elements of $\mathbb{ID}$ are assumed to be the same as those generated by the non-terminal ⟨Identifier⟩.

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

$$\frac{\texttt{e1}, \rho \Downarrow m \quad \texttt{e2}, \rho \Downarrow n \quad n \neq 0 \quad p = m/n}{\texttt{Div(e1,e2)}, \rho \Downarrow p}\ \textsf{EDiv}$$

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

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

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

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

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

Figure 3.1: Evaluation Semantics for LET (error propagation rules omitted)

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.

3.1.4.2 Implementation: Attempt I

Before implementing the evaluator we must first implement expressed values and environments. Expressed values can be naturally described using variant types. Environments can be modeled in various ways in OCaml: as functions, as association lists, as hash tables, as variant 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 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)
  | _ -> failwith "Not implemented yet!"
```

interp.ml

Figure 3.2: Preliminary Interpeter for LET

```
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 3.1 closely. Indeed, the evaluation rules shall serve as a specification for, and thus guide, our implementation. The code itself is given in Figure 3.2. 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 =
  fun ev ->
  match ev with
  | 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.

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 default case, in the last line of Figure 3.2, handles language constructs that the parser supports but will only be explained later. This case simply reports that they are not implemented yet. Notice how the OCaml runtime Failure exception is used to report an error, rather than the `error` operation reserved for object-level (*i.e.* those arising from errors resulting from the execution of the LET) errors.

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[3]:

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

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

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

> 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.

3.1.4.3 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 3.2 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 <u>same</u> 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 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
```

Listing 3.3: 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 =
2   let c = s |> parse |> eval_expr
    in c EmptyEnv
```

Listing 3.4: Naive removal of environment arguments

Unfortunately, the resulting code in Listing 3.3 doesn't type-check. Let us take a closer look at the bind operator used in line 6:

$$\texttt{eval\_expr e1 >>= ...} \tag{3.1}$$

Recall from Figure 2.3 that the type of `(>>=)` is

```
(>>=) : 'a result -> ('a -> 'b result) -> 'b result
```

The expression `eval_expr e1` in (3.1) 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, `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:

Currently	`(>>=) : '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 of the type expression above. Consider the following 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 type synonym and recast our table above as:

Currently	`(>>=) : '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 sho shortly. Applying the type synonym again, this time to the type of `eval_expr`, the new type for our interpreter is now:

Currently	`eval_expr : expr -> env -> exp_val result`
New proposal	`eval_expr : expr -> exp_val ea_result`

Updating the helper functions. Since the type for the helper functions such as bind has changed, we must now update their code. The new code for them is in Figure 3.5. 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 3.4 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 3.5: 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.

3.1.4.4 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 3.6. 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 similar, 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 3.2 except that the environment argument has been dropped. Finally, consider `Let(id,def,body)`. Let us recall from Figure 3.2, 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 the environment arguments, which are now threaded implicitly for us, results in:

```
  | Let(id,def,body) ->
    eval_expr def en >>= 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 resulting from adding the new key value-pair id:=ev into environment en **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))
```

```
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
  | _ -> failwith "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 3.6: Evaluator for LET

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(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. 3.6. 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 3.1.1. *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`.

3.1.5 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, ignoring `e` and halting evaluation with an error message. 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.

3.1.6 Exercises

Exercise 3.1.2 (◊). *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 3.1.3. *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. *Knowing that* `(>>+)` *associates to the left, fill in all the implicit parenthesis in the definition of c.*
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 3.1.4. *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
```

1. *Specify the interpreter (i.e. its set of results and the new evaluation rules). You may assume that you have a product operation* $\times$ *that computes the product of two sets.*
2. *Extend the implementation of* `eval_expr` *to handle the new language constructs. Are there any new errors?*

Exercise 3.1.5. *Consider another extension to LET with pairs. Its concrete syntax is:*

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

Pairs are constructed in the same way as in Exercise 3.1.4. 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 3.1.4 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
```

`parser_plaf/lib/ast.ml`

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

Exercise 3.1.6. *Consider the extension of LET with tuples. Its concrete syntax is that of LET together with the following new productions:*

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

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

$$\langle\text{Expression}\rangle \quad ::= \quad \{\{\langle\text{Identifier}\rangle = \langle\text{Expression}\rangle\}^{+(;)}\}$$
$$\langle\text{Expression}\rangle \quad ::= \quad \langle\text{Expression}\rangle.\langle\text{Identifier}\rangle$$

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 =
  ...
  | RecordVal of (string*(bool*exp_val)) list
```

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

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

Thus a record node in the AST holds a list of pairs composed of the name of field and the expression assigned to that field. The boolean may be ignored for now; it shall always be `false`. Its use will be justified later, in Exercise 4.5.2.

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}`

   Evaluates to:
   `Ok (RecordVal [("age", (false,NumVal 2)); ("height", (false,NumVal 3))]).`

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

   Evaluates to:
   `Ok (RecordVal [("pers", (false, RecordVal [("age", (false,NumVal 2)); ("height", (false,NumVal 3))]); ("cwid", (false,NumVal 10)))]`

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

   Evaluates to:
   `Ok (NumVal 2).`

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

   Evaluates to:
   `Error "Field not found".`

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

   Evaluates to:

   `Error "Record has duplicate fields".`

3.2 PROC

This section adds first-class functions to LET.

3.2.1 Concrete Syntax

Some examples of expressions in PROC are listed below:

```
let f = proc (x) { x-11 }
in (f (f 77))

(proc (f) { (f (f 77)) } proc (x) { x-11 })

let x = 2
in let f = proc (z) { z-x }
in (f 1)

let x = 2
in let f = proc (z) { z-x }
in let x = 1
in let g = proc (z) { z-x }
in (f 1) - (g 1)
```

The concrete syntax for PROC consists in adding two new productions to the grammar of the concrete syntax for LET:

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

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
```

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

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

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

Figure 3.7: Additional Evaluation rules for PROC (error propagation rules omitted)

```
| Div of expr*expr
| Let of string*expr*expr
| IsZero of expr
| ITE of expr*expr*expr
| Proc of string*texpr option*expr
| App of expr*expr
```

Note that the `Proc` constructor has three arguments. The first is the formal parameter and the last is the body of the function. The second parameter is an optional type annotation. It will play a role when we study type-checking. For now, values of type `expr` constructed using `Proc` will always have the form `Proc(id,None,e)`. For example, parsing the expression `let f=proc(x) x+1 in (f 3)` will produce the AST:

```
AProg([],
      Let ("f", Proc ("x", None, Add (Var "x", Int 1)), App (Var "f", Int 3)))
```

3.2.3 Interpreter

3.2.3.1 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 3.1) plus the additional rules given in Figure 3.7.

3.2.3.2 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*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 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(id,e)` 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))
```

This last one is perhaps the least recommendable since the constructor `Ok` should best not be used outside `ds.ml`.

We next consider the case for `App(e1,e2)`. Evaluation of an application requires that we first evaluate `e1` and make sure it is a closure. If that is the case, then `clos` will be bound to a triple containing its three components (parameter, body and environment). We then evaluate `e2` and feed it to the helper function `apply_clos clos` (explained below), which does the rest of the job:

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | App(e1,e2)  ->
    eval_expr e1 >>=
    clos_of_procVal >>= fun clos ->
    eval_expr e2 >>=
    apply_clos clos
```

The function `clos_of_procVal` is similar to `int_of_numVal` from Listing 3.2. It's code is:

```
let clos_of_procVal : exp_val -> (string*expr*env) ea_result =
  fun ev ->
  match ev with
  | ProcVal(id,body,en) -> return (id,body,en)
  | _ -> error "Expected a closure!"
```

ds.ml

The function `apply_clos` sets `en` to be the new current environment and then extends it with the assignment of `id` to `ev`. Under this extended environment, it proceeds with the evaluation of the body of the closure, namely `e`.

```
let rec apply_clos : string*expr*env -> exp_val -> exp_val ea_result =
  fun (id,e,en) ev ->
  return en >>+
  extend_env id ev >>+
  eval_expr e
```

In passing we mention that the tuple type constructor has higher precedence than the function type constructor. Consequently, there is no need to place the type expression `string*expr*env` between parenthesis.

Listing 3.8 summarizes the code described above for procedures and applications.

3.2.4 Dynamic Scoping

If we remove the line below, then we implement dynamic scoping:

```
let rec apply_clos : string*expr*env -> exp_val -> exp_val ea_result =
  fun (id,e,en) ev ->
  return en >>+
  (extend_env id ev >>+
  eval_expr e)
```

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:

```
let rec apply_clos : string*expr*env -> exp_val ->
  exp_val ea_result =
  fun (id,e,en) ev ->
  return en >>+
  extend_env id ev >>+
  eval_expr e
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 >>=
    clos_of_procVal >>= fun clos ->
    eval_expr e2 >>=
    apply_clos clos
```
interp.ml

Figure 3.8: Interpreter for 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)

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

# 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

3.2.5 Exercises

Exercise 3.2.1. *Write a grammar derivation to show that* `let f = proc (x) { x-11 } in (f 77)` *is a valid program in PROC.*

Exercise 3.2.2. *Write down the parse tree for the expression* `let pred = proc(x) { x-1 } in (pred 5)`*.*

Exercise 3.2.3. *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 3.2.4. *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 3.2.5. *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 3.2.6. *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 3.2.7. *The result of evaluating the following expression is* `Ok (NumVal 4)`*. Verify this.*

```
let a = 3
in let p = proc (z) { z+a }
in let f = proc (x) { (p 1) }
in let a = 6
in (f 2)
```

Modify the interpreter so that it uses dynamic scoping rather than static scoping. Then evaluate the above expression again. What value does it return?

Exercise 3.2.8 (◊)**.** *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 3.2.9 (◊)**.** *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))))
```

Exercise 3.2.10. *Consider the following extension of PROC supporting lists. The concrete syntax is obtained from that of PROC with the following additional grammar productions:*

Lists

$$\begin{array}{lcl}
\langle \text{Expression} \rangle & ::= & \texttt{emptylist()} \\
\langle \text{Expression} \rangle & ::= & \texttt{cons}(\langle \text{Expression} \rangle, \langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{hd}(\langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{tl}(\langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{empty?}(\langle \text{Expression} \rangle) \\
\langle \text{Expression} \rangle & ::= & \texttt{mklist}(\langle \text{Expression} \rangle^{*(,)})
\end{array}$$

The abstract syntax is as follows:

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Abs of expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*texpr option*expr
  | App of expr*expr
  | EmptyList of texpr option
  | Cons of expr*expr
  | Hd of expr
  | Tl of expr
  | IsEmpty of expr
  | List of expr list
  | Debug of expr
```

Exercise 3.2.11.

Trees

3.3 REC

Our language unfortunately does not support recursion[4]. 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`.

[4]See exercises 3.2.8 and 3.2.9 on the "higher-order" trick though.

```
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)
```

REC also supports mutually recursive function declarations such as:

```
let true = zero?(0)
in let false = zero?(1)
in letrec
      even(x) = if zero?(x) then true else (odd (x-1))
      odd(x) = if zero?(x) then false else (even (x-1))
in (odd 99)
```

3.3.1 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⟩ ::= `+` | `-` | `*` | `/`

Note that the curly braces in the last grammar production for ⟨Expression⟩ are not terminals, they simply indicate that the sequence of terminals and non-terminals '⟨Identifier⟩(⟨Identifier⟩) = ⟨Expression⟩' may occur once or more.

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*texpr option*expr
  | App of expr*expr
  | Letrec of rdecs*expr
and
  rdecs = (string*string*texpr option*texpr option*expr) list
```

parser_plaf/lib/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:

```
AProg ([],
 Letrec
  ([("fact", "x", None, None,
     ITE (IsZero (Var "x"), Int 1,
      Mul (Var "x", App (Var "fact", Sub (Var "x", Int 1)))))],
  App (Var "fact", Int 5)))
```

The arguments `None` may be ignored for now. They indicate there are no typing annotations; we shall consider typing annotations in Chapter 5. Also, the syntax for `letrec` supports the definition of mutually recursive functions (hence the reason for `list` in the type definition `rdecs`), however for now we will only be using examples where a single recursive function is declared.

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

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

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

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

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

Figure 3.9: Additional evaluation rules for REC

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.

3.3.3.1 Specification

The set of results is the same as in PROC:

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

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 3.9. 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.

3.3.3.2 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*expr*env
and
  env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of string*string*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 $\rho \oplus \{\texttt{id} :=$

$(\mathtt{par},\mathtt{e1},\rho)^r\}$ in the rule ELetRec of Figure 3.9, 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 -> 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 3.3.1. *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 3.3.2 (◊). *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 3.2.10. 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 3.3.3 (◊). *Consider the following expression in REC*

```
let z = 0
in let p = proc (x) { proc (y) { x*y }}
in letrec f(n) = if zero?(n) then 1 else ((p 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 four items below. Note that there may be more than one solution for each item (it suffices to supply just one) or none at all.*

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 p,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 p = proc (x) { proc (y) { x*y }}
in letrec f(n) = if zero?(n) then 1 else ((p n) (f (n-1)))
in (f 10)
```

2.

```
>>Environment:
z:=NumVal 0,
p:=ProcVal (x,Proc(y,Mul(Var x,Var y)),z:=NumVal 0),
f:=Rec(n,IfThenElse(Zero?(Var n),Int 1,
       App(App(Var p,Var n),App(Var f,Sub(Var n,Int 1))))),
n:=NumVal 9
```

Draw a box around the argument of `debug`:

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

3.

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

Draw a box around the argument of `debug`:

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

4.

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

Draw a box around the argument of `debug`:

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

Chapter 4

Imperative Programming

4.1 Mutable Data Structures in OCaml

This section discusses some OCaml language features that allow data to be updated in-place. In-place means that the data, which is stored in some memory location, is updated at that same location. This is in contrast to functional update, which involves updating by first making a fresh copy of the original data item and then performing the update. We introduce three well-known data types that support in-place update in OCaml: references, arrays and mutable record fields.

4.1.1 References

References...

One use of references is in simulating the behavior of objects. By an 'object' we mean an abstraction of a state, a set of operations that can access and modify the state and which are the only means of doing so, and the ability to refer to the state and operations within the object itself (typically through special variables such as `this` or `self`).

4.1.1.1 An impure or stateful function

Mathematical functions are relations in which each element of a domain set is assigned a unique element in the codomain set. Consider the following function in OCaml:

```
let f = let state = ref 0
        in fun () ->
           begin
             state:=!state+1;
             !state
           end
```

Every time we apply it to the same argument, we get a different result:

```
# f ();;
- : int = 1
# f ();;
- : int = 2
```

```
# f ();;
- : int = 2
```

utop

We call this function impure or stateful in order to distinguish it from the pure mathematical functions mentioned above. It is stateful since its result does not only rely on the argument but on an additional (hidden, internal) state, namely the value held inside the pointer `state`.

4.1.1.2 A counter object

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

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

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

utop

A counter object with models self reference using recursion. Note how `dec` is implemented by calling the `inc` method.

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

4.1.1.3 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)}
```

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

utop

4.2 EXPLICIT-REFS

The following is an extension of REC.

4.2.1 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.

4.2.2 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 rdecs*expr
  | NewRef of expr
  | DeRef of expr
  | SetRef of expr*expr
  | BeginEnd of expr list
  | Debug of expr
```

4.2.3 Interpreter

4.2.3.1 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 $\mathbb{S}$:

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

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

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

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

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

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

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

Figure 4.1: Evaluation rules for EXPLICIT-REFS (error propagation rules omitted)

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 4.1. 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.

4.2.3.2 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 -> string_of_int i^"->"^f x) @@ take st.size @@ Array.to_list st.data
  in
  String.concat ",\n" ss

let string_of_store f st =
  match st.size with
  | 0 -> ">>Store:\nEmpty"
  | _ -> ">>Store:\n"^ string_of_store' f st
```

store.ml

In OCaml, every `.ml` file is wrapped into a module. Modules package together related definitions and help provide consistent namespaces. For example, `store.ml` will be wrapped in a module called `Store`. Modules can provide not just functions but also types and submodules, among others. By default, all definitions provided in a module are accessible. Through interface files one may restrict the definitions that are accessible. For example, the `store.mli` file above, lists the definitions that are to be made accessible within the module `Store`.

4.2.3.3 Implementation

We could now follow the ideas we developed for environments and have stores 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. Also, the updated store would have to be returned. Thus the return type `result` would have be updated to return a pair consisting of the updated store and the result itself[1]. 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`.

[1] This handling of the store in the background, including its auxiliary data types, is know as a state monad. Thus we would end up with a combination of error, reader, and state monads. Combining monads can be done through monad transformers.

```
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*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) ->
    eval_exprs es >>= fun evs ->
    return (List.hd (List.rev evs))
  | 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")
  | _ -> failwith ("Not implemented: "^string_of_expr e)
and
  eval_exprs =
  fun es ->
  match es with
   | [] -> return []
   | h::t ->
     eval_expr h >>= fun ev ->
     eval_exprs t >>= fun evs ->
     return (ev::evs)
```

interp.ml

4.2.4 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(init) {
                        let s = newref(init)
                        in (self s)
                      }
in let c= (new_counter 0)
in begin
    (c.inc 1);
    (c.inc 2);
    (c.reset 0);
    (c.read 0)
  end
```

4.3 IMPLICIT-REFS

The following is an extension of REC.

4.3.1 Concrete Syntax

Examples of expressions in IMPLICIT-REFS

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

let x=2
in 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
```

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

4.3.2 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 rdecs*expr
  | Set of string*expr
  | BeginEnd of expr list
  | Debug of expr
and
  rdecs = (string*string*texpr option*texpr option*expr) list
```

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

$$\frac{\mathtt{e1}, \rho, \sigma_1 \Downarrow w, \sigma_2 \qquad \ell \notin \mathsf{dom}(\sigma_2) \quad \mathtt{e2}, \rho \oplus \{\mathtt{id} := \ell\}, \sigma_2 \oplus \{\ell := w\} \Downarrow v, \sigma_3}{\mathtt{Let(id,e1,e2)}, \rho, \sigma_1 \Downarrow v, \sigma_3}\ \mathsf{ELet}$$

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

$$\frac{\mathtt{e}, \rho, \sigma \Downarrow v, \sigma'}{\mathtt{Set(id,e)}, \rho, \sigma \Downarrow \mathit{unit}, \sigma' \oplus \{\rho(\mathtt{id}) := v\}}\ \mathsf{ESet}$$

$$\frac{\rho(\mathtt{id}) \notin \mathbb{L} \text{ or } \rho(\mathtt{id}) \notin \mathsf{dom}(\sigma)}{\mathtt{Set(id,e)}, \rho, \sigma \Downarrow \mathit{error}, \sigma}\ \mathsf{ESetErr}$$

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

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

Figure 4.2: Evaluation rules for IMPLICIT-REFS (error propagation rules are omitted)

4.3.3 Interpreter

4.3.3.1 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

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

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:

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

Indeed, $\rho(\mathtt{id})$ denotes a location whose contents is looked up in the store σ. If $\rho(\mathtt{id})$ is not a valid location, then an error is returned, as described by rule EVarErr. The full set of evaluation rules are those of REC (adapted to the new format of the evaluation judgments) plus the ones given in Figure 4.2.

4.3.3.2 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_clos` function. We briefly recall the code for `apply_clos` as implemented in the PROC (Figure 3.8):

```
let rec apply_clos : string*expr*env -> exp_val -> exp_val ea_result =
  fun (id,e,en) ev ->
  return en >>+
  (extend_env id ev >>+
  eval_expr e)
```

Note that evaluation of the body requires extending the environment with a new key-value pair, namely (`id`,`ev`), where `ev` is the expressed value supplied as argument. Environments no longer map identifiers to expressed values, but to locations. So we first need to allocate `ev` 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_clos` is given below.

```
let rec apply_clos : string*expr*env -> exp_val -> exp_val ea_result =
  fun (id,e,en) ev ->
  return en >>+
  (extend_env id (RefVal (Store.new_ref g_store ev)) >>+
  eval_expr e)
```

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

```
  | Var(id) ->
    apply_env id >>=
    int_of_refVal >>=   (* make sure id is mapped to a location *)
    Store.deref g_store (* if so, dereference it *)
  ...
  | Let(v,def,body) ->
    eval_expr def >>= fun ev ->       (* evaluate definition *)
    let l = Store.new_ref g_store ev (* allocate it in the store *)
    in extend_env v (RefVal l) >>+    (* extend env with new key-value pair *)
    eval_expr body                    (* eval body in extended env *)
  | Set(id,e) ->
    eval_expr e >>= fun ev ->   (* eval RHS *)
    apply_env id >>=
    int_of_refVal >>= fun l -> (* make sure id is mapped to location *)
    Store.set_ref g_store l ev >>= fun _ -> (* update the store *)
    return UnitVal
  | BeginEnd([]) ->
    return UnitVal
  | BeginEnd(es) ->
    sequence (List.map eval_expr es) >>= fun vs ->
    return (List.hd (List.rev vs))
```

4.3.4 `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*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 there is a better way, which avoids having to create closures on the fly. The idea is to allow circular environments. That is, an environment `env` that has an entry to a location on the store that holds a closure whose environment has a reference to this same location.

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*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 ->
```

$$\frac{\begin{array}{c} \texttt{e1},\rho,\sigma \Downarrow (\texttt{id},\texttt{e},\tau),\sigma_1 \\ \text{e2 not identifier} \\ \texttt{e2},\rho,\sigma_1 \Downarrow w,\sigma_2 \\ \ell\notin \mathsf{dom}(\sigma_2) \\ \texttt{e},\tau\oplus\{\texttt{id}:=\ell\},\sigma_2\oplus\{\ell:=w\}\Downarrow v,\sigma_3\end{array}}{\texttt{App(e1,e2)},\rho,\sigma\Downarrow v,\sigma_3}\ \text{EApp1}$$

$$\frac{\texttt{e1},\rho,\sigma \Downarrow (\texttt{id},\texttt{e},\tau),\sigma_1 \quad \texttt{e},\tau\oplus\{\texttt{id}:=\rho(id2)\},\sigma_2\Downarrow v,\sigma_3}{\texttt{App(e1,id2)},\rho,\sigma\Downarrow v,\sigma_3}\ \text{EApp2}$$

Figure 4.3: Evaluation rules for Call-by-Reference

```
     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.

4.4 Parameter Passing Methods

We consider several parameter passing methods in IMPLICIT-REFS.

4.4.1 Call-by-Value

This method consists in first evaluating the argument, before passing on its value to the function. This is the parameter passing method we have implemented in PROC and all the languages that extend it.

4.4.2 Call-by-Reference

If the argument to a function is a variable, then we provide a copy of its reference to the function. Otherwise, the argument is processed just like in call-by-value. For example, evaluation of

```
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. However, using call-by-reference, it will return `Ok (NumVal 3)`. It is helpful to place a breakpoint inside the body of the `f`, evaluate the resulting expression and examine the environment and store.

4.4.2.1 Implementation

The evaluation rules for call-by-reference are those for IMPLICIT-REFS of Fig. 4.2, but where EApp is replaced by the two rules EApp1 and EApp2 in Fig. 4.3. Note that EApp1 is just EApp. The new rule is EApp2, which applies when the argument of an application is an identifier. In this case, no allocation takes place on the store.

```
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_clos =
  ...
and
  eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | App(e1,e2)  ->
    eval_expr e1 >>=
    clos_of_procVal >>= fun clos ->
    eval_expr e2 >>=
    value_of_operand e2 >>=
    apply_clos clos
```

interp.ml

The following example shows how one may swap the contents of two variables:

```
let a = 2
in let b = 1
in let f = proc (x) { proc (y) {
        let temp = x
        in begin
            set x = y;
            set y = temp
           end }}
in begin
    ((f a) b);
    a
  end
```

Returns `Ok (NumVal 1)`.

4.4.3 Call-by-Name

Consider the following expression. What is the result of its evaluation?

```
letrec infinite_loop (x) = (infinite_loop  (x+1))
in let f = proc (y) { 11 }
in (f (infinite_loop 0))
```

The parameter `z` is not used. However, the argument to `f`, namely `(infinite_loop 0)`, is evaluated all the same. The call-by-name parameter passing method consists in freezing

$$\frac{\sigma(\rho(\texttt{id})) = v \quad v \text{ not a thunk}}{\texttt{Var(id)}, \rho, \sigma \Downarrow v, \sigma} \text{EVarValue} \qquad \frac{\sigma(\rho(\texttt{id})) = (\texttt{e}, \rho')^t \quad \texttt{e}, \rho', \sigma \Downarrow v, \sigma'}{\texttt{Var(id)}, \rho, \sigma \Downarrow v, \sigma'} \text{EVarThunk}$$

$$\frac{\texttt{e1}, \rho, \sigma \Downarrow (\texttt{id}, \texttt{e}, \tau), \sigma_1 \quad \ell \notin \mathsf{dom}(\sigma_2) \quad \texttt{e}, \tau \oplus \{\texttt{id} := \ell\}, \sigma_2 \oplus \{\ell := (\texttt{e2}, \rho)^t\} \Downarrow v, \sigma_3}{\texttt{App(e1,e2)}, \rho, \sigma \Downarrow v, \sigma_3} \text{EAppFreeze}$$

Figure 4.4: Evaluation rules for IMPLICIT-REFS with call-by-name parameter passing

the evaluation of arguments until they are actually needed. This is achieved as follows. In an application `App(e1,e2)`, if `e2` is an identifier, then call-by-name proceeds just like call-by-reference: it passes a copy of the address of the identifier, after looking it up in the environment, as an argument to the parameter. Regardless of whether the value of `e2` will be used, copying an address is a constant time operation, hence not costly. However, if `e2` is an expression different from an identifier, then its evaluation does not take place. Rather, `e2` together with the current environment are stored for later evaluation. The pair consisting of `e2` and the current environment is called a thunk. A thunk is just a closure without the formal parameter. We next implement call-by-name, the parameter passing method that implements this idea.

4.4.3.1 Implementation

The evaluation rules for call-by-name are those for IMPLICIT-REFS of Fig. 4.2, but where EVar and EApp are replaced by the rules EVarValue, EVarThink and EAppFreeze in Fig. 4.5. Note that EAppThunk freezes the evaluation of the argument `e2` by placing it inside a thunk, together with the current environment: $(\texttt{e2}, \rho)^t$. Moreover, EVarValue deals with variable lookup when the location for `id` has been allocated an expressed value. EVarThunk deals with the case when the location for `id` has been allocated a thunk.

We will use the code for call-by-reference as a starting point. We begin by adding thunks to the set of expressed values:

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

Note that we use `Thunk` rather than `ThunkVal` as the name of the constructor, to emphasize that a thunk is not a run-time value that may be returned as the result of a evaluation. Next we update `value_of_operand`. If the argument or operand is a variable, our interpreter behaves just like in call-by-reference. However, if it is not a variable, we create a thunk:

```
let rec value_of_operand =
  fun op ->
  match op with
  | Var id -> apply_env id
  | _ ->
    lookup_env  >>= fun en ->
    return (RefVal (Store.new_ref g_store (Thunk(op, en))))
```

$$\frac{\sigma(\rho(\texttt{id})) = v \quad v \text{ not a thunk}}{\texttt{Var(id)}, \rho, \sigma \Downarrow v, \sigma}\ \text{EVarValue} \qquad \frac{\sigma(\rho(\texttt{id})) = (\texttt{e}, \rho')^t \quad \texttt{e}, \rho', \sigma \Downarrow v, \sigma'}{\texttt{Var(id)}, \rho, \sigma \Downarrow v, \sigma' \oplus \{\rho(\texttt{id}) := v\}}\ \text{EVarThunk}$$

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

Figure 4.5: Evaluation rules for IMPLICIT-REFS with call-by-need parameter passing

interp.ml

Finally, we have to consider what happens when we lookup the contents of the store through the reference assigned to a variable in the environment and it consists of a thunk `Thunk(e,en)`. We "thaw" the thunk by evaluating `e` under the environment `en`:

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | Int(n) -> return (NumVal n)
  | Var(id) ->
    apply_env id >>=
    int_of_refVal >>=
    Store.deref g_store >>= fun ev ->
    (match ev with
     | Thunk(e,en) -> return en >>+ eval_expr e
     | _ -> return ev)
  ...
```

interp.ml

Evaluate the example from the beginning of this section in CBN. Then do the same but this time inserting a breakpoint in the body of `f`:

```
letrec infinite_loop (x) = (infinite_loop  (x+1))
in let f = proc (y) { debug(11) }
in (f (infinite_loop 0))
```

4.4.4 Call-by-Need

One drawback of call-by-name is that a thunk is "thawed" every time it is needed. Consider the following example:

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

Here the factorial of 5 is computed four times, one for each occurrence of `y` in `g`. A more reasonable approach is to do this once, the first time, and then store the result for further uses. This optimization technique is called <u>memoization</u>. Call-by-need consists in applying this optimization technique to call-by-name.

4.4.4.1 Implementation

All we need to do is update the implementation for the variable constructor in `eval_expr`:

```
let rec  eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  | Int(n) -> return @@ NumVal n
  | Var(id) ->
    apply_env id >>=
    int_of_refVal >>= fun l ->
    Store.deref g_store l >>= fun ev ->
    (match ev with
     | Thunk(e,en) ->
       return en >>+
       eval_expr e >>= fun ev ->
       Store.set_ref g_store l ev >>= fun _ ->
       return ev
    | _ -> return ev)
  ...
```

interp.ml

The factorial example above now involves computing factorial of 5 just once.

Call-by-need and call-by-name may return different results for effectful computation. Evaluate the following expression in CBN and CBNeed:

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

4.5 Exercises

Exercise 4.5.1. *Depict the environment and store at the breakpoint for the following EXPLICIT-REFS programs:*

1.
```
let a = 3
in let b = newref(if zero?(a) then 1 else 2)
in debug(a)
```

2.
```
let a = newref(newref(2+1))
in debug(a)
```

3.
```
let a = (let s = newref(0)
         in proc(i) {
               begin
                  setref(s,deref(s)+i);
                  deref(s)
```

```
                    end
            })
in debug(a)
```

Exercise 4.5.2. *Consider the following extension of LET with records (Exercise 3.1.7). 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 was introduced in Exercise 3.1.7. We recall it below and add a new constructor for field update:

```
type expr =
  ...
  | Record of (string*(bool*expr)) list
  | Proj of expr*string
  | 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*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) ->
     failwith "update"
  | SetField(e1,id,e2) ->
     failwith "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 4.5.3. *Depict the environment and store extant at the breakpoint in the following IMPLICIT-REFS expressions.*

1.
```
let a = 2
in let b = 3
in begin
    set a = b;
    debug(a)
   end
```

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

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

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

Exercise 4.5.4. *Depict the enviroment and store at the breakpoint first assuming call-by-reference as parameter passing method and then call-by-value.*

1.
```
let a=1
in let b=2
in let f = proc (x) { debug(if zero?(x) then 1 else 2)}
in (f a)
```

2.
```
let a=1
in let b=2
in let f = proc (x) { debug(if zero?(x) then 1 else 2)}
in (f (a+1))
```

3.
```
let a=1
in let b=2
in let f = proc (x) { proc (d) { debug(set x=x+d) }}
in ((f a) (b+1))
```

Exercise 4.5.5. *Depict the enviroment and store at the breakpoint first assuming call-by-name as parameter passing method and then call-by-need.*

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

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

Chapter 5

Types

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

5.1 CHECKED

5.1.1 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⟩`)`

5.1.2 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 option*expr
  | App of expr*expr
  | Letrec of rdecs*expr
and
  rdecs = (string*string*texpr option*texpr option*expr) list
and
 texpr =
  | IntType
  | BoolType
  | FuncType of texpr*texpr
```

5.1.3 Type-Checker

We specify the behavior of our type-checker, before implementing it, by introducing a **type system**. Using the type system we will then implement a **type-checker**.

5.1.3.1 Specification

A type system is an inductive set that helps identify a subset of the expressions that are considered to be well-typed or typable. The elements of the inductive set are called **typing judgements**. Which typing judgements belong to the inductive set and which don't is determined by a set of typing rules. A typing judgement is an expression of the form

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

where Γ is a type environment, `e` is an expression in CHECKED, and t is a type expression. Types are defined as follows:

$$t \quad ::= \quad \mathtt{int} \,|\, \mathtt{bool} \,|\, t \rightarrow t$$

A **type environment** is a partial function that assigns a type to an identifier. Type environments 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 environments 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 environment is $\epsilon, x : \mathtt{int}, y : \mathtt{bool}$. We abbreviate it as $\{x : \mathtt{int}, y : \mathtt{bool}\}$.

The typing rules are given in Figure 5.1. The typing rules for addition, multiplication and division are omitted; they are similar to TSub. An expression e is typable if there exists a typing environment Γ and a type t such that the typing judgement $\Gamma \vdash \mathsf{e} : t$ is derivable using the typing rules in Figure 5.1. Otherwise, e is said to be untypable.

Example 5.1.1. *Consider the typing judgement* $\epsilon \vdash$ *`letrec f(x:int):int = e in (f 5)`* : `int`*, where* e *stands for* *`if zero?(x) then 1 else x*(f (x-1))`. *A typing derivation for it follows.*

$$\frac{}{\Gamma \vdash \texttt{n} : \texttt{int}}\ \textsf{TInt} \qquad \frac{\Gamma(\texttt{x}) = t}{\Gamma \vdash \texttt{x} : t}\ \textsf{TVar} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{int}}{\Gamma \vdash \texttt{zero?(e)} : \texttt{bool}}\ \textsf{TIsZero}$$

$$\frac{\Gamma \vdash \texttt{e1} : \texttt{int} \qquad \Gamma \vdash \texttt{e2} : \texttt{int}}{\Gamma \vdash \texttt{e1-e2} : \texttt{int}}\ \textsf{TSub}$$

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

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

$$\frac{\Gamma \vdash \texttt{e1} : t1 \rightarrow t2 \quad \Gamma \vdash \texttt{e2} : t1}{\Gamma \vdash \texttt{(e1 e2)} : t2}\ \textsf{TApp}$$

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

$$\frac{\begin{array}{l}\Gamma, \texttt{id2} : tPar, \texttt{id1} : tPar \rightarrow tRes \vdash \texttt{e1} : tRes \\ \Gamma, \texttt{id1} : tPar \rightarrow tRes \vdash \texttt{e2} : t\end{array}}{\Gamma \vdash \texttt{letrec id1(id2:tPar):tRes = e1 in e2} : t}\ \textsf{TLetrec}$$

Figure 5.1: Typing Rules for CHECKED

$$\frac{\dfrac{\pi}{\{f:\mathtt{int}\to\mathtt{int},x:\mathtt{int}\}\vdash e:\mathtt{int}}\,\text{TITE} \qquad \dfrac{\dfrac{\{f:\mathtt{int}\to\mathtt{int}\}(f)=\mathtt{int}\to\mathtt{int}}{\{f:\mathtt{int}\to\mathtt{int}\}\vdash f:\mathtt{int}\to\mathtt{int}}\,\text{TVar} \quad \dfrac{}{\{f:\mathtt{int}\to\mathtt{int}\}\vdash 5:\mathtt{int}}\,\text{TInt}}{\{f:\mathtt{int}\to\mathtt{int}\}\vdash \mathtt{(f\ 5)}:\mathtt{int}}\,\text{TApp}}{\epsilon\vdash \mathtt{letrec\ f(x:int):int\ =\ e\ in\ (f\ 5)}:\mathtt{int}}\,\text{TLetrec}$$

The typing derivation π is given below; Γ is a shorthand for $\{f:\mathtt{int}\to\mathtt{int},x:\mathtt{int}\}$:

$$\frac{\dfrac{\dfrac{\Gamma(x)=\mathtt{int}}{\Gamma\vdash x:\mathtt{int}}\,\text{TVar}}{\Gamma\vdash \mathtt{zero?(x)}:\mathtt{bool}}\,\text{TIsZero} \qquad \dfrac{}{\Gamma\vdash 1:\mathtt{int}} \qquad \dfrac{\dfrac{\Gamma(x)=\mathtt{int}}{\Gamma\vdash x:\mathtt{int}}\,\text{TVar} \quad \dfrac{\dfrac{\Gamma(f)=\mathtt{int}\to\mathtt{int}}{\Gamma\vdash f:\mathtt{int}\to\mathtt{int}}\,\text{TVar} \quad \dfrac{\dfrac{\Gamma(x)=\mathtt{int}}{\Gamma\vdash x:\mathtt{int}}\,\text{TVar} \quad \dfrac{}{\Gamma\vdash 1:\mathtt{int}}\,\text{TInt}}{\Gamma\vdash \mathtt{x-1}:\mathtt{int}}\,\text{TSub}}{\Gamma\vdash \mathtt{(f\ \ (x-1))}:\mathtt{int}}\,\text{TApp}}{\Gamma\vdash \mathtt{x*(f\ \ (x-1))}:\mathtt{int}}\,\text{TMul}}{\Gamma\vdash e:\mathtt{int}}\,\text{TITE}$$

5.1.3.2 Towards an 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

```
chk_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 5.2: The `a_result` type

The type `('a,'b) a_result`, together with its supporting operations are declared in Figure 5.2. 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
```

5.1.3.3 Implementation

We next address the implementation of the type-checker for CHECKED. The code is given in Figure 5.3. In the `Letrec` case, note that since >>+ is left associative, the mapping `param` := `tPar` is added to the typing environment only for type-checking `body`.

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

utop

```
let rec chk_expr : expr -> texpr tea_result =
  fun e ->
  match e with
  | Int _n -> return IntType
  | Var id -> apply_tenv id
  | IsZero(e) ->
    chk_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) ->
    chk_expr e1 >>= fun t1 ->
    chk_expr e2 >>= fun t2 ->
    if (t1=IntType && t2=IntType)
    then return IntType
    else error "arith: arguments must be ints"
  | ITE(e1,e2,e3) ->
    chk_expr e1 >>= fun t1 ->
    chk_expr e2 >>= fun t2 ->
    chk_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) ->
    chk_expr e >>= fun t ->
    extend_tenv id t >>+
    chk_expr body
  | Proc(var,t1,e) ->
    extend_tenv var t1 >>+
    chk_expr e >>= fun t2 ->
    return (FuncType(t1,t2))
  | App(e1,e2) ->
    chk_expr e1 >>=
    pair_of_funcType "app: " >>= fun (t1,t2) ->
    chk_expr e2 >>= fun t3 ->
    if t1=t3
    then return t2
    else error "app: type of argument incorrect"
  | Letrec([(id,param,tPar,tRes,body)],target) ->
    extend_tenv id (FuncType(tPar,tRes)) >>+
    (extend_tenv param tPar >>+
    chk_expr body >>= fun t ->
    if t=tRes
    then chk_expr target
    else error "LetRec: Type of rec. function does not match declaration")
  | Debug(_e) ->
    string_of_tenv >>= fun str ->
    print_endline str;
    error "Debug: reached breakpoint"
  | _ -> failwith "chk_expr: implement"
and
  chk_prog (AProg(_,e)) =
  chk_expr e
```

checker.ml

Figure 5.3: Type checker for CHECKED

5.1.4 Exercises

Exercise 5.1.2 (◊). *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 5.1.3. *Recall that an expression `e` is typable, if there exists a type environment Γ and a type expression `t` such that the typing judgement $\Gamma \vdash e : t$ is derivable. Argue that the expression `x x` (a variable applied to itself) is not typable.*

Exercise 5.1.4. *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 5.1.5. *Show that the following term is typable:*

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

Exercise 5.1.6. *What is the result of evaluating the following expressions in CHECKED?*

```
utop # chk "
letrec double (x:int):int = if zero?(x)
                              then 0
                              else (double (x-1)) + 2
in (double 5)";;
```
utop

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

```
utop # chk "
letrec double (x:int):bool = if zero?(x)
                              then 0
                              else (double (x-1)) + 2
in double";;
```

utop

```
utop # chk "
letrec double (x:int):bool = if zero?(x)
                              then 0
                              else 1
in double";;
```

utop

Exercise 5.1.7. *Consider the extension of Exercise 3.1.5 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 3.1.5 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.

Exercise 5.1.8. *Consider the following the extension of CHECKED with records, as introduced in Exercise 3.1.7. The concrete syntax for the new type constructor for records is given by the second to last production below:*

<Type>	::=	`int`
<Type>	::=	`bool`
<Type>	::=	<Type> `->` <Type>
<Type>	::=	$\{\{\langle \text{Identifier} \rangle$: <Type>$\}^{+(;)}\}$
<Type>	::=	(<Type>)

The abstract syntax is as follows:

```
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 rdecs*expr
  | Record of (string*expr) list
  | Proj of expr*string
and
  rdecs = (string*string*texpr option*texpr option*expr) list
and
 texpr =
  | IntType
  | BoolType
  | FuncType of texpr*texpr
  | RecordType of (string*texpr) list
```

For example,

1. {`age=2; height=3`} *should have type* {`age:int; height:int`}.
2. {`age=2; present=zero?(0)`} *should have type* {`age:int; present:bool`}.
3. {`inc = proc(x:int) {x+1}; dec = proc(x:int) {x-1}`} *should have type* {`inc:int->int; dec:int->int`}.
4. {`inc = proc(x:int) {x+1}; dec = proc(x:int) {x-1}`}`.inc` *should have type* `int->int`.
5. {} *should produce a type error since empty records are not allowed.*
6. {`age=2; height=3`}`.weight` *should produce a type error since there is no field named* `weight`.

The additional typing rules are:

$$\frac{\Gamma \vdash \texttt{e1} : t1 \quad \ldots \quad \Gamma \vdash \texttt{en} : tn \quad n > 0 \quad li, i \in 1..n, \textit{distinct}}{\Gamma \vdash \{\ \texttt{l1=e1;}\ \ \texttt{...;}\ \ \texttt{ln=en}\} : \{l1 : t1; \ldots; ln : tn\}}\ \text{TRec}$$

$$\frac{\Gamma \vdash \texttt{e} : \{l1 : t1; \ldots; ln : tn\} \quad l = li,\ \textit{for some } i \in 1..n}{\Gamma \vdash \texttt{e.l} : ti}\ \text{TProj}$$

Extend the type checker `chk_expr` *to deal with the two new constructs.*

Chapter 6

Simple Object-Oriented Language

6.1 Concrete Syntax

The concrete syntax for SOOL is presented below. Before doing so, we exhibit an example.

```
(* class declarations *)

(* counterc *)
class counterc extends object {
  field c
  method initialize() { set c=7 }
  method add(i) { set c=c+i }
  method bump() { send self add(1) }
  method read() { c }
}

(* reset counter *)
class resetc extends counterc {
  field v
  method reset() { set c=v }
  method setReset(i) { set v=i }
}

(* backup counter *)
class bkpcc extends resetc {
  field b
  method initialize() {
    begin
      super initialize();
      set b=12
    end
  }
  method add(i) {
    begin
      send self backup();
      super add(i)
    end
  }
  method backup() { set b=c }
  method restore() { set c=b}
```

```
}

(* main expression *)
let o = new bkpcc  ()
in begin
     send o add(10);
     send o bump();
     send o restore();
     send o read()
   end
```

Listing 6.1: Example program in SOOL

Next we'll introduce the concrete syntax, in stages.

$$\langle \text{Program} \rangle \quad ::= \quad \langle \text{ClassDecl} \rangle^{*} \langle \text{Expression} \rangle$$

A SOOL program is a (possibly empty) list of class declarations followed by a main expression.

$$\begin{array}{lll}
\langle \text{Expression} \rangle & ::= & \texttt{new}\ \langle \text{Identifier} \rangle (\langle \text{Expression} \rangle^{*(,)}) \\
\langle \text{Expression} \rangle & ::= & \texttt{self} \\
\langle \text{Expression} \rangle & ::= & \texttt{send}\ \langle \text{Expression} \rangle \langle \text{Identifier} \rangle (\langle \text{Expression} \rangle^{*(,)}) \\
\langle \text{Expression} \rangle & ::= & \texttt{super}\ \langle \text{Identifier} \rangle (\langle \text{Expression} \rangle^{*(,)})
\end{array}$$

An expression can be any expression in IMPLICIT-REFS together with four new object-oriented specific expressions (the first four listed above) and some additional ones that support lists. Only the new productions that are added to the grammar for IMPLICIT-REFS are depicted above.

$$\langle \text{ClassDecl} \rangle \quad ::= \quad \langle \text{Identifier} \rangle\ \texttt{extends}\ \langle \text{Identifier} \rangle \{ \langle \text{FieldDecl} \rangle^{*} \langle \text{MethodDecl} \rangle^{*} \}$$

A class declaration consists of the name of the class being defined, the name of the superclass, a list of field declarations and a list of method declarations.

$$\begin{array}{lll}
\langle \text{FieldDecl} \rangle & ::= & \texttt{field}\ \langle \text{Identifier} \rangle \\
\langle \text{MethodDecl} \rangle & ::= & \texttt{method}\ \langle \text{Identifier} \rangle (\langle \text{Identifier} \rangle^{*(,)}) \{ \langle \text{Expression} \rangle \}
\end{array}$$

A field declaration is just the keyword `field` followed by an identifier, the name of the field. A method declaration consists of the keyword `method`, an identifier representing the name of the method, a list of formal parameters between parenthesis, and the body.

6.2 Abstract Syntax

```
type
  prog = AProg of (cdecl list)*expr
and
  expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Abs of expr
  | Let of string*expr*expr
```

```
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*texpr option*expr
  | App of expr*expr
  | Letrec of rdecs*expr
  | Set of string*expr
  | BeginEnd of expr list
  | Self
  | Send of expr*string*expr list
  | Super of string*expr list
  | NewObject of string*expr list
  | Debug of expr
and
  cdecl = Class of string*string*string option*(string*texpr option) list*mdecl list
and
  mdecl = Method of string*texpr option*(string*texpr option) list*expr
```

We omit the type declaration for `texpr`; this will be given later.

You can parse expressions that reside in a file by using `parsef`:

```
# parsef "ex3.sool";;
- : prog =
AProg
 ([Class ("counterc", "object", None, [("c", None)],
    [Method ("initialize", None, [], Set ("c", Int 7));
     Method ("add", None, [("i", None)],
      Set ("c", Add (Var "c", Var "i")));
     Method ("bump", None, [], Send (Self, "add", [Int 1]));
     Method ("read", None, [], Var "c")]);
   Class ("resetc", "counterc", None, [("v", None)],
    [Method ("reset", None, [], Set ("c", Var "v"));
     Method ("setReset", None, [("i", None)],
      Set ("v", Var "i"))]);
   Class ("bkpcc", "resetc", None, [("b", None)],
    [Method ("add", None, [("i", None)],
      BeginEnd
       [Send (Self, "backup", []);
        Super ("add", [Var "i"])]);
     Method ("backup", None, [], Set ("b", Var "c"));
     Method ("restore", None, [], Set ("c", Var "b"))])],
 Let ("o", NewObject ("bkpcc", []),
  BeginEnd
   [Send (Var "o", "add", [Int 10]);
    Send (Var "o", "bump", []);
    Send (Var "o", "restore", []);
    Send (Var "o", "read", [])]))
```

utop

6.3 Interpreter

6.3.1 Specification

We start this section by introducing some auxiliary notation. We write $\overline{\bullet}_n$ to a denote a sequence of n items. For example, $\overline{v}_n$ denotes a sequence of n values $v_1, \ldots, v_n$. Similarly, $\overline{id}_n$ denotes a sequence of n identifiers $id_1, \ldots, id_n$. We will write $\overline{v}$ and $\overline{id}$ when then we do not wish to emphasize the length of the sequence. Evaluation judgements take the form:

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

where e, ρ, σ, r and σ' are as in IMPLICIT-REFS. There are two differences, however. First, expressed values now include objects.

$$\begin{array}{rcl} \mathbb{EV} & := & \mathbb{Z} \cup \mathbb{B} \cup \mathbb{U} \cup \mathbb{CL} \cup \mathbb{L} \cup \mathbb{O} \\ \mathbb{O} & := & \{\mathcal{O}(id, \rho) \,|\, id \in \mathbb{ID}, \rho \in \mathbb{ENV}\} \end{array}$$

A object is denoted $\mathcal{O}(id, \rho)$ and consists of an identifier id representing the name of class of the object, and an environment ρ mapping locations to all the fields of the object. The second difference is the presence of a **class environment** κ, a partial function that maps class names to their declarations. For example, in the case of Listing 6.1, we have:

$$\begin{array}{rcl} \kappa(\text{“}\texttt{bkpcc}\text{”}) & = & \mathcal{C}(\text{“}\texttt{resetc}\text{”}, [\text{“}b\text{”}; \text{“}v\text{”}; \text{“}c\text{”}], \\ & & [\text{“}\texttt{initialize}\text{”} := \mathcal{M}([], \texttt{BeginEnd [Super ("initialize", []); Set ("b", Int 12)]}, \texttt{bl} \\ & & \text{“}\texttt{add}\text{”} := \mathcal{M}([\text{“}i\text{”}], \texttt{BeginEnd [Send (Self, "backup", []); Super ("add", [Var "i"])} \\ & & \text{“}\texttt{backup}\text{”} := \mathcal{M}([], \texttt{Set ("b", Var "c")}, \texttt{bkupcc}); \\ & & \text{“}\texttt{restore}\text{”} := \mathcal{M}([], \texttt{Set ("c", Var "b")}, \texttt{bkupcc}); \\ & & \text{“}\texttt{reset}\text{”} := \mathcal{M}([], \texttt{Set ("c", Var "v")}, \texttt{resetc}); \\ & & \text{“}\texttt{setReset}\text{”} := \mathcal{M}([\text{“}i\text{”}], \texttt{Set ("v", Var "i")}, \texttt{resetc}); \\ & & \text{“}\texttt{initialize}\text{”} := \mathcal{M}([], \texttt{Set ("c", Int 7)}, \texttt{counterc}); \\ & & \text{“}\texttt{add}\text{”} := \mathcal{M}([\text{“}i\text{”}], \texttt{Set ("c", Add (Var "c", Var "i"))}, \texttt{counterc}); \\ & & \text{“}\texttt{bump}\text{”} := \mathcal{M}([], \texttt{Send (Self, "add", [Int 1])}, \texttt{counterc}); \\ & & \text{“}\texttt{read}\text{”} := \mathcal{M}([], \texttt{Var "c"}, \texttt{counterc})] \\ & &) \end{array}$$

A **class declaration** is a tuple:

$$\mathcal{C}(id, \overline{fid}, \overline{me})$$

where id is an identifier representing the name of the superclass, $\overline{fid}$ is a sequence of identifiers representing the fields, and $\overline{me}$ is a sequence of method declarations. We often write fields$(\mathcal{C}(id, \overline{fid}, \overline{me}))$ to denote $\overline{fid}$. We use κ for the set of all classes in our program. It maps a class identifier to a class declaration. Moreover, we assume that classes have a unique name.

A **method declaration** is a tuple:

$$\mathcal{M}(\overline{id}, \texttt{e}, cid)$$

It includes a sequence of identifiers representing the formal parameters $\overline{id}$, the body of the method e and the name of the class hosting the method cid. Note that we assume that a class declaration includes all the methods defined in that class together with the ones it inherits. This will simplify our presentation of the evaluation rules. In the example above, starting from the bottom of the list, the first four methods listed are inherited from `counter`, the next one is inherited from `resetc` and the last four methods (the topmost ones) are the ones declared in the class `bkpcc` itself.

We write $\texttt{es}, \rho, \sigma_0, \kappa \Downarrow^* \overline{v}_m, \sigma_m$ for the sequence of evaluation judgements $\texttt{es}, \rho, \sigma_0, \kappa \Downarrow v_1, \sigma_1, \ldots, \texttt{es}, \rho, \sigma_{m-1}, \kappa \Downarrow v_m, \sigma_m$. Note that this sequence may be empty. In that case $\texttt{es}, \rho, \sigma_0, \kappa \Downarrow^* \overline{v}_m, \sigma_m$ just stands for the empty sequence ϵ.

We next discuss the evaluation rules for each of the new constructs. The hypothesis of the upcoming evaluation rules will include a number at the end of line for easy reference when describing them.

6.3.1.1 Self

We shall always ensure that the current environment ρ maps the reserved identifiers self and super to locations containing an object and an identifier, respectively. The evaluation rule is thus:

$$\frac{}{\texttt{Self()}, \rho, \sigma, \kappa \Downarrow \sigma(\rho(\mathit{self})), \sigma}\ \textsf{ESelf}$$

Note that since SOOL is an extension of IMPLICIT-REFS, the enviroment maps variables to locations, whose contents then have to be retreived in the store. This explains $\sigma(\rho(\mathit{self}))$ above.

6.3.1.2 New.

For presentation purposes, we consider three cases. The first is when the class being instantiated does not exist. The evaluation rules reads as follows:

$$\frac{\mathit{id} \notin \mathsf{dom}(\kappa)}{\texttt{New(id,[])}, \rho, \sigma_0, \kappa \Downarrow \mathit{error}, \sigma_0}\ \textsf{ENewErr1}$$

The second case is when the class exists but there is no initialize method to be evaluated upon creation of the object. The evaluation rule reads as follows:

$$\frac{\begin{array}{rl} \kappa(\mathit{id}) = \mathcal{C}(\mathit{sid}, \overline{\mathit{fid}}, \overline{\mathit{me}}) & (1) \\ \mathsf{newenv}(\overline{\mathit{fid}}, \sigma_0) = (\mu, \sigma_1) & (2) \\ \texttt{initialize} \notin \mathsf{dom}(\overline{\mathit{me}}) & (3) \end{array}}{\texttt{New(id,[])}, \rho, \sigma_0, \kappa \Downarrow \mathcal{O}(\mathit{id}, \mu), \sigma_1}\ \textsf{ENewNoInit}$$

We first lookup the details of the class declaration for class `id`, as indicated by item (1) above. We then allocate a fresh memory location in the store for all the fields in scope for class `id`, as indicated by item (2) above. All locations are initialized to the default value 0. This is all achieved through the helper function newenv defined below:

$$\mathsf{newenv}(\overline{\mathit{id}}_n, \sigma) := (\mu, \sigma') \text{ where } \mu = \{\overline{\mathit{id}}_n := \overline{\ell}_n\} \text{ for } \overline{\ell}_n \not\subseteq \mathsf{dom}(\sigma), \text{ and } \sigma' = \sigma \oplus \{\overline{\ell}_n := 0\}$$

We then check that the `initialize` method indeed has not be declared and that, therefore, there is no initialization code to evaluate (indicated in item (3) above). Finally, we return the newly created object, namely $\mathcal{O}(\mathit{id}, \mu)$, and the updated store σ_1.

The third case is when the class exists and there is initialization code to be evaluated, just after an object is created. The process of evaluation of `New(id,es)` starts off very similarly to ENewNoInit, but then it deviates by invoking `initialize`. The details are as follows:

$$\frac{\begin{array}{cr} \texttt{es}, \rho, \sigma_0, \kappa \Downarrow^* \overline{v}_m, \sigma_m & (1) \\ \kappa(id) = \mathcal{C}(sid, \overline{fid}_n, \overline{me}) & (2) \\ \mathsf{newenv}(\overline{fid}_n, \sigma_m) = (\mu, \sigma_{m+1}) & (3) \\ \overline{me}(\texttt{initialize}) = \mathcal{M}(id_1, \overline{pid}_m, \texttt{e}) & (4) \\ \nu = \mathsf{slice}(\mu, \mathsf{fields}(\kappa(id_1))) \oplus \{\overline{pid}_m := \overline{\ell}_m, \texttt{self} := \ell_{m+1}, \texttt{super} := \ell_{m+2}\} \quad \{\overline{\ell}_{m+2}\} \nsubseteq \mathsf{dom}(\mu) & (5) \\ \sigma_{m+2} = \sigma_{m+1} \oplus \{\overline{\ell}_m := \overline{v}_m, \ell_{m+1} := \mathcal{O}(id, \mu), \ell_{m+2} := \mathsf{super}(\kappa(id_1))\} & (6) \\ \texttt{e}, \nu, \sigma_{m+2}, \kappa \Downarrow w, \sigma_{m+3} & (7) \end{array}}{\texttt{New(id,es)}, \rho, \sigma_0, \kappa \Downarrow \mathcal{O}(id, \mu), \sigma_{m+3}} \text{ ENew}$$

1. We begin with evaluation of all the arguments es that will be passed on to `initialize`. Note that it is possible that the list of arguments is empty. This is indicated as item (1) in the evaluation rule ENew.

2. We then lookup the details of the class `id`; this is item (2).

3. Next we allocate a default value of 0 for all the fields visible to the class `id` and create an environment μ. We use the helper function newenv, just like in the case of the evaluation rule ENewNoInit.

4. Next we lookup the method `initialize` among the methods visible to class `id`. The lookup will return the name of the class that hosts the method id_1, the list of its formal parameters $\overline{pid}$ and the body `e`.

5. Slicing. Consider Listing 6.1 but where the main expression is: `new resetc()`. Evaluation should create a new object value instance of the class `resetc`. Upon creation and before initialization, this object should take the form $\mathcal{O}(\texttt{resetc}, \{v := \ell_1, c := \ell_2\})$, where the store has the form $\{\ell_1 := 0, \ell_2 := 0\}$. The method `initialize` is not defined in class `resetc` but rather is inherited from `counter`. Only field `c` is in scope; in particular field `v` is not visible to it. This is the reason behind slicing. In particular $\mathsf{slice}(\{v := \ell_1, c := \ell_2\}, [\text{"}c\text{"}]) = \{c := \ell_1\}$.

   In addition to slicing, mappings for the formal parameters and the built-in `super` and `self` are included in the environment ν.

6. Evaluation of initialization code. In the last step, the initialization code is executed using ν, from the previous step, as the environment.

Evaluation returns the newly created object $\mathcal{O}(id, \mu)$ and the updated store σ_{m+3}.

6.3.1.3 Send

We split the presentation into two cases, the one where the method invoked does not exist and the one where it does. Let us first consider the case where, in an expression `Send(e,mid,es)`, the method `mid` does not exist.

$$\frac{\begin{array}{cr} \texttt{e1}, \rho, \sigma_0, \kappa \Downarrow \mathcal{O}(id, \mu), \sigma_1 & (1) \\ \kappa(id) = \mathcal{C}(sid, \overline{fid}, \overline{me}) & (2) \\ \texttt{mid} \notin \mathsf{dom}(\overline{me}) & (3) \end{array}}{\texttt{Send(e1,mid,es)}, \rho, \sigma_0, \kappa \Downarrow \mathit{error}, \sigma_1} \text{ ESendErr1}$$

$$
\frac{\begin{array}{c}
\texttt{e1}, \rho, \sigma_0, \kappa \Downarrow \mathcal{O}(id, \mu), \sigma_1 \\
\kappa(id) = \mathcal{C}(sid, \overline{fid}_n, \overline{me}) \\
\texttt{es}, \rho, \sigma_1, \kappa \Downarrow^* \overline{v}_m, \sigma_{m+1} \\
\overline{me}(\texttt{mid}) = \mathcal{M}(id_1, \overline{pid}_m, \texttt{e2}) \\
\nu = \mathsf{slice}(\mu, \mathsf{fields}(\kappa(id_1))) \oplus \{\overline{pid}_m := \overline{\ell_{nm}}, \texttt{self} := \ell_{m+m+1}, \texttt{super} := \ell_{m+m+2}\} \quad \{\overline{\ell_{nm}}, \ell_{n+m+1}, \ell_{n+m+2}\} \nsubseteq \mathsf{dom}(\mu) \\
\sigma_{m+2} = \sigma_{m+1} \oplus \{\ell_{n+1} := v_1, \ldots, \ell_{n+m} := v_m, \ell_{n+m+1} := \mathcal{O}(id, \mu), \ell_{n+m+2} := \mathsf{super}(\kappa(id_1))\} \\
\texttt{e2}, \nu, \sigma_{m+2}, \kappa \Downarrow w, \sigma_{m+3}
\end{array}}{\texttt{Send(e1,mid,es)}, \rho, \sigma_0, \kappa \Downarrow w, \sigma_{m+3}}
$$

$$
\frac{\begin{array}{c}
\texttt{es}, \rho, \sigma_1, \kappa \Downarrow^* \overline{v}_m, \sigma_{m+1} \\
\kappa(\sigma(\rho(super))) = \mathcal{C}(sid, \overline{fid}_n, \overline{me}) \\
\overline{me}(\texttt{mid}) = \mathcal{M}(id_1, \overline{pid}_m, \texttt{e2}) \\
\nu = \mathsf{slice}(\mu, \mathsf{fields}(\kappa(id_1))) \oplus \{\overline{pid}_m := \overline{\ell_{nm}}, \texttt{self} := \ell_{m+m+1}, \texttt{super} := \ell_{m+m+2}\} \quad \{\overline{\ell_{nm}}, \ell_{n+m+1}, \ell_{n+m+2}\} \nsubseteq \mathsf{dom}(\mu) \\
\sigma_{m+2} = \sigma_{m+1} \oplus \{\ell_{n+1} := v_1, \ldots, \ell_{n+m} := v_m, \ell_{n+m+1} := \sigma(\rho(self)), \ell_{n+m+2} := \mathsf{super}(\kappa(id_1))\} \\
\texttt{e2}, \nu, \sigma_{m+2}, \kappa \Downarrow w, \sigma_{m+3}
\end{array}}{\texttt{Super(mid,es)}, \rho, \sigma_0, \kappa \Downarrow w, \sigma_{m+3}}
$$

First e is evaluated to produce an object $\mathcal{O}(id, \mu)$, as indicated by (1) above. Then the class declaration for class id is looked up in κ. Finally, the method `mid` is verified not to exist in that class declaration and an error is returned as result.

We next consider the evaluation rule ESend, where we assume the method `mid` to exist. It starts off with the same first two steps as in ESendErr1. After that, evaluation of all the arguments es takes place, producing values $\overline{v}_m$, as indicated in (3). Next we perform method dispatch and locate the expression to evaluate for method `mid`, namely `e2`. The method declaration also includes the name of the super class of the class that hosts `e2` and the list of formal parameters $\overline{pid}_m$. The final step in the process is to evaluate `e2`; this is indicate in line (6) of the evaluation rule. However, first we must set up the environment. This, in turn, will require allocating new values in the heap.

6.3.1.4 Super

Let us first consider the case where, in an expression `Super(mid,es)`, the method `mid` does not exist. This is modeled by the evaluation rule SuperErr1.

$$
\frac{\begin{array}{c}
\kappa(\sigma(\rho(super))) = \mathcal{C}(sid, \overline{fid}, \overline{me}) \\
\texttt{mid} \notin \mathsf{dom}(\overline{me})
\end{array}}{\texttt{Super(mid,es)}, \rho, \sigma_0, \kappa \Downarrow error, \sigma_0} \text{ESuperErr1}
$$

First we lookup the class declaration for the superclass of the class that hosts the call to super itself. We then verify that `mid` is not declared in that class and return an error as the result of evaluation.

We next consider the case where method dispatch for `mid` is successful. The evaluation rule in question is ESuper.

6.3.2 Implementation

Evaluation of a program `AProg(cs,e)` is done in two steps, both performed by the function `eval_prog`:

```
let rec
  eval_expr : expr -> exp_val ea_result =
    fun e ->
    ...
and
  eval_prog : prog -> exp_val ea_result =
    fun (AProg(cs,e)) ->
    initialize_class_env cs;    (* Step 1 *)
    eval_expr e                 (* Step 2 *)
```

Step 1 consists in processing the class declarations in `cs`, producing a class environment (referred to as κ in our evaluation judgements above). The aim is to have ready access not just to the fields and methods declared in a class, but also to all those it inherits. Class environments are implemented as a list of pairs:

```
type class_env = (string*class_decl) list
```

Each entry in this list consists of a pair whose first component is the name of the class and the second one is a class declaration. A class declaration is a tuple of type `string*string list*method_env` consisting of the name of the class, the list of the fields **visible** from that class and the list of methods **visible** from that class.

```
type method_decl = string list*Ast.expr*string*string list
type method_env = (string*method_decl) list
type class_decl = string*string list*method_env
```

The resulting class environment is placed in the global variable `g_class_env` of type `class_env ref` for future use. Thus `g_class_env` is a reference to an association list, that is, a list of pairs.

For the example, for Listing 6.1 the contents of `g_class_env` may be inspected as follows[1].

```
# interpf "ex3.sool";;
- : exp_val Sool.ReM.result = Sool.ReM.Ok (NumVal 17)
utop # !g_class_env;;
- : class_env =
[("bkpcc",
  ("resetc", ["b"; "v"; "c"],
   [("add",
     (["i"], BeginEnd [Send (Self, "backup", []); Super ("add", [Var "i"])],
      "resetc", ["b"; "v"; "c"]));
    ("backup",
     ([], Set ("b", Var "c"), "resetc", ["b"; "v"; "c"]));
    ("restore",
     ([], Set ("c", Var "b"), "resetc", ["b"; "v"; "c"]));
    ("reset",
     ([], Set ("c", Var "v"), "counterc", ["v"; "c"]));
    ("setReset",
     (["i"], Set ("v", Var "i"), "counterc", ["v"; "c"]));
    ("initialize",
```

[1] It is possible that the output is truncated by utop. The directive in utop `#print_length 2000;;` changes this to allow printing up to 2000 items.

```
      ([], Set ("c", Int 7), "object", ["c"]));
     ("add",
      (["i"], Set ("c", Add (Var "c", Var "i")), "object",["c"]));
     ("bump",
      ([], Send (Self, "add", [Int 1]), "object", ["c"]));
     ("read",
      ([], Var "c", "object", ["c"]))]));
  ("resetc",
   ("counterc", ["v"; "c"],
    [("reset",
      ([], Set ("c", Var "v"), "counterc", ["v"; "c"]));
     ("setReset",
      (["i"], Set ("v", Var "i"), "counterc", ["v"; "c"]));
     ("initialize",
      ([], Set ("c", Int 7), "object", ["c"]));
     ("add",
      (["i"], Set ("c", Add (Var "c", Var "i")), "object",["c"]));
     ("bump",
      ([], Send (Self, "add", [Int 1]), "object", ["c"]));
     ("read",
      ([], Var "c", "object", ["c"]))]));
  ("counterc",
   ("object", ["c"],
    [("initialize",
      ([], Set ("c", Int 7), "object", ["c"]));
     ("add",
      (["i"], Set ("c", Add (Var "c", Var "i")), "object",["c"]));
     ("bump",
      ([], Send (Self, "add", [Int 1]), "object", ["c"]));
     ("read",
      ([], Var "c", "object", ["c"]))]));
  ("object", ("", [], []))]
```

utop

In particular, notice that the first entry in the list is of the form

```
[("bkpcc", ("resetc", ["b"; "v"; "c"],...))]
```

Here:

- `bkpcc` is the name of the class
- `resetc` is the name of the superclass of `bkpcc`
- `["b"; "v"; "x"]` is the list of all the fields that are visible to `bkpcc`, including the ones that are inherited.
- The ellipses ... is a list of all the methods that are visible to `bkpcc`, including the ones that are inherited.

Step 2 consists in evaluating the main expression. This process consults `g_class_env` whenever it requires information from the class hierarchy. Evaluation takes place via the function `eval_expr`. We next describe each case in turn. The full code is provided in Listing 6.2.

Self. First we implement the evaluation rule for `Self`, described in Section 6.3.1.1. We simply need to lookup the contents of the location associated to the identifier `_self`. The underscore is to avoid having users declare an identifier with the same name.

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | Self ->
    eval_expr (Var "_self")
```

interp.ml

New. We implement the evaluation rules for `New`, described in Section 6.3.1.2.

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | NewObject(c_name,es) ->
    eval_exprs es >>= fun args ->
    (match List.assoc_opt c_name !g_class_env with
     | None -> error ("NewObject: lookup_class: class "^c_name^" not found")
     | Some (_super,fields,methods) ->
       new_env fields >>= fun env ->
       let self = ObjectVal(c_name,env)
       in (match List.assoc_opt "initialize" methods with
           | None -> return self
           | Some m -> apply_method "initialize" self args m >>= fun _ ->
             return self))
and
  eval_exprs : expr list -> exp_val list ea_result =
  fun es ->
  match es with
  | [] -> return []
  | h::t ->
    eval_expr h >>= fun ev ->
    eval_exprs t >>= fun evs ->
    return (ev::evs)
```

interp.ml

Send. We implement the evaluation rules for `Send`, described in Section 6.3.1.3.

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | Send(e,m_name,es) ->
    eval_expr e >>= fun self ->
    obj_of_objectVal self >>= fun (c_name,_) ->
    eval_exprs es >>= fun args ->
    (match lookup_method c_name m_name !g_class_env with
     | None -> error "Method not found"
     | Some m -> apply_method m_name self args m)
```

interp.ml

Super. We implement the evaluation rules for Super, described in Section 6.3.1.4.

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | Super(m_name,es) ->
    eval_exprs es >>= fun args ->
    eval_expr (Var "_super") >>=
    string_of_stringVal >>= fun c_name ->
    eval_expr (Var "_self") >>= fun self ->
    (match lookup_method c_name m_name !g_class_env with
     | None -> error "Method not found"
     | Some m -> apply_method m_name self args m)
```

interp.ml

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | NewObject(c_name,es) ->
    eval_exprs es >>= fun args ->
    (match List.assoc_opt c_name !g_class_env with
     | None -> error ("NewObject: lookup_class: class "^c_name^" not found")
     | Some (_super,fields,methods) ->
       new_env fields >>= fun env ->
       let self = ObjectVal(c_name,env)
       in (match List.assoc_opt "initialize" methods with
           | None -> return self
           | Some m -> apply_method "initialize" self args m >>= fun _ ->
             return self))
  | Send(e,m_name,es) ->
    eval_expr e >>= fun self ->
    obj_of_objectVal self >>= fun (c_name,_) ->
    eval_exprs es >>= fun args ->
    (match lookup_method c_name m_name !g_class_env with
     | None -> error "Method not found"
     | Some m -> apply_method m_name self args m)
  | Self ->
    eval_expr (Var "_self")
  | Super(m_name,es) ->
    eval_exprs es >>= fun args ->
    eval_expr (Var "_super") >>=
    string_of_stringVal >>= fun c_name ->
    eval_expr (Var "_self") >>= fun self ->
    (match lookup_method c_name m_name !g_class_env with
     | None -> error "Method not found"
     | Some m -> apply_method m_name self args m)
     ...
and
  eval_exprs : expr list -> exp_val list ea_result =
  fun es ->
  match es with
  | [] -> return []
  | h::t ->
    eval_expr h >>= fun ev ->
    eval_exprs t >>= fun evs ->
    return (ev::evs)
```

```
and
  eval_prog : prog -> exp_val ea_result =
  fun (AProg(cs, e)) ->
  initialize_class_env cs;    (* Step 1 *)
  eval_expr e                 (* Step 2 *)
```
interp.ml

6.4 Exercises

Exercise 6.4.1. *Consider the counter example of Listing 6.1, but where the class* `bkupcc` *and the main expression are modified as follows:*

```
(* backup counter *)
class bkpcc extends resetc {
  field b
  method initialize() {
    begin
      super initialize();
      set b=12
    end
  }
  method add(i) {
    begin
      send self backup();
      super add(i)
    end
  }
  method add(i,j) {
    begin
      send self backup();
      super add(i);
      super add(j)
    end
  }
  method backup() { set b=c }
  method restore() { set c=b}
 }

(* main expression *)
let o = new bkpcc  ()
in begin
     send o restore();
     send o add(2,3);
     send o read()
   end
```

Evaluation will produce the error:

```
# interpf "ex4.sool";;
- : exp_val Sool.ReM.result =
Sool.ReM.Error "add: args and params have different lengths"
```
utop

In this exercise you must add support for overloading through <u>*name mangling*</u>*. Consider the following name mangling function*

```
let name_mangle n es =
  n^"_"^string_of_int (List.length es)
```

Add it to the source code (`interp.ml`) together with appropriate calls to it, so that the interpreter for SOOL supports overloading.

Chapter 7

Modules

7.1 Concrete 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
```

A program in SIMPLE-MODULES consists of a possible empty sequence of module declarations followed by an expression:

$$\langle \text{Program} \rangle \quad ::= \quad \{\langle \text{ModuleDefn} \rangle\}^* \langle \text{Expression} \rangle$$

Expressions are the those of REC but with an extra production that we refer to as a <u>qualified variable reference</u>:

$$\langle \text{Expression} \rangle \quad ::= \quad \texttt{from}\ \langle \text{Identifier} \rangle\ \texttt{take}\ \langle \text{Identifier} \rangle$$

The concrete syntax of modules is given by the following grammar:

$$
\begin{array}{lcl}
\langle \text{ModuleDefn} \rangle & ::= & \texttt{module}\ \langle \text{Identifier} \rangle\ \texttt{interface}\ \langle \text{Iface} \rangle\ \texttt{body}\ \langle \text{ModuleBody} \rangle \\
\\
\langle \text{Iface} \rangle & ::= & [\{\langle \text{Decl} \rangle\}^*] \\
\langle \text{Decl} \rangle & ::= & \langle \text{Identifier} \rangle : \langle \text{Type} \rangle \\
\\
\langle \text{ModuleBody} \rangle & ::= & [\{\langle \text{Defn} \rangle\}^*] \\
\langle \text{Defn} \rangle & ::= & \langle \text{Identifier} \rangle = \langle \text{Expression} \rangle
\end{array}
$$

7.2 Abstract Syntax

The abstract syntax is presented below.

```
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
```

7.3 Interpreter

7.3.1 Specification

The set of results is the same as that for EXPLICIT-REFS. In particular, the set of expressed values consists of integers, booleans, unit, closures and locations:

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

There are three kinds of evaluation judgements in SIMPLE-MODULES, one for programs, one for expressions, and a third auxiliary one used to evaluate module definitions. The evaluation judgement for programs is:

$$\texttt{AProg(mdecls,e)}, \rho, \sigma \Downarrow r, \sigma'$$

A program `AProg(mdecls,e)` consists of a sequence of module declarations `mdecl` and a main expression `e`. Also, ρ is the initial environment and σ the initial store. The result of the evaluation is `r` and the updated store is σ'. Evaluation judgements for expressions are similar to those of EXPLICIT-REFS:

$$\frac{\texttt{mdecls}, \rho, \sigma \Downarrow \rho', \sigma' \quad \texttt{e}, \rho', \sigma' \Downarrow r, \sigma''}{\texttt{AProg(mdecls,e)}, \rho, \sigma \Downarrow r, \sigma''} \text{EProg}$$

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

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

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

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

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

Figure 7.1: Evaluation Semantics for SIMPLE-MODULES (error propagation rules omitted)

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

where `e` is an expression, ρ an environment, σ the initial store, r the result and σ' the final store. The difference is that the environment will also allow for mappings between module identifiers and their bodies. The body of a module will be implemented as an environment too. The third evaluation judgement is:

$$\texttt{mdecls}, \rho, \sigma \Downarrow \rho', \sigma'$$

Here `mdecls` is a sequence of module declarations, ρ is an initial environment and σ is an initial store. Evaluation of `mdecls` will produce an environment ρ' which associates to each module `mid` in `mdecls` an environment ρ_{mid}. Evaluation also produces an updated store σ'.

7.3.2 Implementation

We first extend environments to support bindings for modules:

```
type env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of string*string*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 helper function `eval_module_definitions : module_decl list -> env ea_result`. We'll describe this function shortly.

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

Evaluation of expressions is just like in EXPLICIT-REFS, except that we must deal with the new case, namely that of a qualified variable `QualVar(module_id,var_id)`.

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | QualVar(module_id,var_id) ->
    apply_env_qual module_id var_id
    ...
```

The helper function `apply_env_qual` inspects the environment looking for a module name `module_id` and then an identifier `var_id` declared within that module:

```
let rec apply_env_qual : string -> string -> exp_val ea_result =
  fun mid id ->
  fun env ->
  match env with
  | EmptyEnv -> Error "Key not found"
  | ExtendEnv(key,value,env) -> apply_env_qual mid id env
  | ExtendEnvRec(key,param,body,env) -> apply_env_qual mid id env
  | ExtendEnvMod(moduleName,bindings,env) ->
    if mid=moduleName
    then apply_env id bindings
    else apply_env_qual mid  id env
```

Finally, we turn to the above mentioned `eval_module_definitions` helper function. Given a list of module declarations `ms`, it evaluates them one by one using `eval_module_definition` and then returning a value of type `env ea_result` holding the resulting environment.

```
let rec eval_expr : expr -> exp_val ea_result =
   ...
and
  eval_module_definition : module_body -> env ea_result =
  fun (AModBody vdefs) ->
  lookup_env >>= fun glo_env -> (* holds all previously declared modules *)
  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

7.4 Type-Checking

7.4.1 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` : Δ_2

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

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

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

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,

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

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

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

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

Figure 7.2: Typing rules for SIMPLE-MODULES

- 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 $f : I \to J$

7.4.2 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 >>+
  chk_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 >>+
        chk_expr decl >>=
        extend_tenv var))
      (empty_tenv ())
      vdefs) >>= fun tmbody ->
  return (reverse_tenv tmbody)
and
  chk_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
  ...
```

7.5 Further Reading

Module inclusion Private types First-class modules

Appendix A

Supporting Files

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

A.1 File Structure

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

```
.
|____dune-project
|____lib
| |____.merlin
| |____.ocamlinit
| |____ds.ml
| |____dune
| |____interp.ml
|____test
| |____dune
| |____test.ml
```

A Dune project is provided for each language we implement. Each Dune project has a dune-project file. Moreover, each folder has a Dune configuration file called dune. The source files are in the lib directory and the unit tests are in the test directory.

.merlin	Tells Merlin where to locate sources
.ocamlinit	Loaded by utop upon execution; opens some modules
ds.ml	Supporting data structures including expressed values, environments and results
dune	Dune build file
dune_project	Dune project configuration file
interp.ml	Interpreter
test.ml	Unit tests

File structure for the parser.

```
.
|____dune-project
|____lib
| |____ast.ml
| |____grammar.mly
| |____lexer.mly
| |____dune
| |____parser.ml
| |____parserMessages.messages
|____test
| |____dune
| |____test.ml
```

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

`ast.ml`	Abstract Syntax
`dune`	Dune build file
`lexer.mll`	Lexer generator
`grammar.mly`	Grammar for parser generator
`parserMessages.messages`	Error messages for parser (under construction)
`parser.ml`	Parser
`test.ml`	Unit tests

Some common dune commands:

- Build the project and then run utop

```
$ dune utop
```

- Build the project

```
$ dune build
```

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

```
$ dune clean
```

- Run tests (building if necessary)

```
$ dune runtest
```

- Generate documentation (install odoc with opam first):

```
$ dune build @doc
```

The generated html files are in:

```
$ open _build/default/_doc/_html/index.html
```

A.2 Running the Interpreters

First you must build and install the parser.

```
# cd PLaF/src/parser_plaf
# dune build
Read 280 sample input sentences and 280 error messages.
Read 280 sample input sentences and 280 error messages.
Read 280 sample input sentences and 280 error messages.
# dune install
```

bash

Building and running the LET interpreter (similar instructions should be followed for the other languages):

```
# cd ../let/lib
# dune utop
```

bash

This builds and runs utop. You should be able to parse a simple expression by typing:

```
utop # parse "3+3";;
- : prog = AProg ([], Add (Int 3, Int 3))
```

utop

You may also evaluate an expression as follows:

```
# interp "let x=2 in x+x";;
- : exp_val Let.Ds.result = Ok (NumVal 4)
```

utop

Appendix B

Solution to Selected Exercises

Section 3.1

Answer B.0.1 (Exercise 3.1.2)**.** *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 3.2

Answer B.0.2 (Exercise 3.2.8)**.**

```
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 3.2.9).

```
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 3.3

Answer B.0.4 (Exercise 3.3.2).

```
# 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 3.3.3)**.**

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)
```

Answer B.0.6 (Exercise 5.1.2)**.** *Provide typing derivations for the following expressions:*

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

   Let Γ *stand for the typing environment* $\{x : \mathtt{int}\}$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x :: \mathtt{int}}\,\text{TVar} \quad \dfrac{}{\Gamma \vdash 2 :: \mathtt{int}}\,\text{TInt}}{\Gamma \vdash \mathtt{x\text{-}2} :: \mathtt{int}}\,\text{TSub}}{\Gamma \vdash \mathtt{zero?(x\text{-}2)} :: \mathtt{bool}}\,\text{TZero} \quad \dfrac{}{\Gamma \vdash 0 :: \mathtt{int}}\,\text{TInt} \quad \dfrac{}{\Gamma \vdash 1 :: \mathtt{int}}\,\text{TInt}}{\Gamma \vdash \mathtt{if\ zero?(x\text{-}2)\ then\ 0\ else\ 1} :: \mathtt{int}}\,\text{TITE}}{\epsilon \vdash \mathtt{proc(x{:}int)}\ \{\mathtt{if\ zero?(x\text{-}2)\ then\ 0\ else\ 1}\} :: \mathtt{int} \to \mathtt{int}}\,\text{TProc} \quad \dfrac{\dfrac{}{\{\mathtt{two?} : \mathtt{int} \to \mathtt{int}\} \vdash \mathtt{two?} :: \mathtt{int} \to \mathtt{int}}\,\text{TVar} \quad \dfrac{}{\{\mathtt{two?} : \mathtt{int} \to \mathtt{int}\} \vdash 3 :: \mathtt{int}}\,\text{TInt}}{\{\mathtt{two?} : \mathtt{int} \to \mathtt{int}\} \vdash (\mathtt{two?\ \ 3}) :: \mathtt{int}}\,\text{TApp}}{\epsilon \vdash \mathtt{let\ two?{=}proc(x{:}int)\ \{\ if\ zero?(x\text{-}2)\ then\ 0\ else\ 1\ \}\ in\ (two?\ \ 3)} :: \mathtt{int}}\,\text{TLet}
$$

Bibliography

[Acz77] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, HANDBOOK OF MATHEMATICAL LOGIC, volume 90 of Studies in Logic and the Foundations of Mathematics, pages 739–782. Elsevier, 1977.

[FS91] P.A. Fejer and D. Simovici. Mathematical Foundations of Computer Science: Sets, Relations, and Induction. Mathematical Foundations of Computer Science. Springer New York, 1991.

[FW08] Daniel P. Friedman and Mitchell Wand. Essentials of Programming Languages. The MIT Press, 3rd edition, 2008.