Essentials of Programming Languages

Daniel P Friedman Mitchell Wand

1 Inductive Sets of Data

Here we introduce the basic tools needed for interpreters, checkers, etc. Recursion is at the core of these techniques.

1.1 Recursively Specified Data

Here we look at formal techniques for specifying sets of values.

1.1.1 Inductive Specification

D1.1.1 (a subset of naturals) $n \in S$ iff $n = 0$ or $n - 3 \in S$.

Thus all and only multiples of 3 are in S.

```
(define in-S?
    (lambda (n)
        (if (zero? n) #t
            (if (>= (- n 3) 0)
                (in-S? (- n 3))
                #f))))
```

Alternatively,
D1.1.2 S is the smallest set contained in N s.t. $0 \in S$ and $n \in S \implies n+3 \in S$.

Another version is by using rules of inference.

Exercise 1.1
–

Exercise 1.2
1. $\{(i, 1 + 7i) : i \in N\}$
2. $\{(i, 2^i) : i \in N\}$
3. $\{(i, F_i, F_{i+1}) : i \in N\}$
4. $\{(i, 2i + 1, i^2) : i \in N\}$

Exercise 1.3
$T = \{0, 1, 3, 4, 6, 7, \dots\}$

1.1.2 Defining Sets Using Grammars

Exercise 1.4
–

D1.1.6 (s-list, s-exp)
S-list → ({S-exp}*)
S-exp → Symbol | S-list

Context-senstivice constraints are usually added by other methods to context-sensitive grammars.

1.1.3 Induction

Exercise 1.5
–

1.2 Deriving Recursive Programs

1.2.1 `list-length`

1.2.2 `nth-element`

Exercise 1.6
–

Exercise 1.7
–

1.2.3 `remove-first`

Exercise 1.8
List → `()`

Exercise 1.9
–

1.2.4 `occurs-free?`

Exercise 1.10
–

1.2.5 `subst`

Exercise 1.11
–

Exercise 1.12
–

Exercise 1.13
–

1.3 Auxiliary Procedures and Context Arguments

An argument that tells us where we are in a recursion (that may not decrease at calls) is called a context argument or inherited attribute.

Exercise 1.14

–

1.4 Exercises

Exercise 1.15

```
(define duple
    (lambda (n x)
        (conde
            ((zero? n) '())
            (else (cons x (duple (- n 1) x))))))
```

Exercise 1.16

```
(define invert
    (lambda (lst)
        (conde
            ((null? lst) '())
            (else (cons (flip (car x)) (cdr x))))))

(define flip
    (lambda (pair)
        (cons (cadr pair) (cons (car pair) '()))))
```

2 Data Abstraction

2.1 Specifying Data via Interfaces

Data abstraction divides data into an interface and implementation. A data type that employs this is an ADT. Client code that only uses the interface is representation-independent.

For example, for natural numbers in Scheme, we have an interface of four procedures:

```
(zero)
(is-zero? n)
(successor n)
(predecessor n)
```

Now we can write client programs using these functions:

```
(define plus
    (lambda (x y)
```

```
        (if (is-zero? x) y
            (successor (plus (predecessor x) y)))))
```

The implementation of the numbers can follow a unary representation (list of `#t`s), the Scheme number representation or a "bignum" representation (a list of digits in base N). None of these, however, implement data abstraction – they are transparent representations.

Exercise 2.1

```
(define zero (quote ()))

(define is-zero? (lambda (x) (null? x)))

(define successor (lambda (x)
    (cond
        ((is-zero? x)
            '(1))
        ((eq? (car x) (- N 1))
            (cons 0 (successor (cdr x))))
        (else
            (cons (+ (car x) 1) (cdr x))))))

(define predecessor (lambda (x)
    (cond
        ((equal? x '(1))
            '())
        ((eq? (car x) 0)
            (cons (- N 1) (predecessor (cdr x))))
        (else
            (cons (- (car x) 1) (cdr x))))))

(define add (lambda (x y)
    (cond
        ((is-zero? x)
            y)
        (else
            (successor (add (predecessor x) y))))))

(define mult (lambda (x y)
    (cond
        ((is-zero? (predecessor x))
            y)
        (else
            (add (mult (predecessor x) y) y)))))

(define fact (lambda (x)
```

```
    (cond
        ((is-zero? x)
            (successor zero))
        (else
            (mult x (fact (predecessor x)))))))
```

Exercise 2.2

–

Exercise 2.3

```
(define zero '(diff (one) (one)))

(define is-zero? (lambda (x)
    (cond
        ((equal? x '(one))
            #f)
        (else
            (equal? (cadr x) (caddr x))))))

(define successor (lambda (x)
    `(diff ,x (diff (diff (one) (one)) (one)))))

(define predecessor (lambda (x)
    `(diff ,x (one))))

(define diff-tree-plus (lambda (x y)
    `(diff ,x (diff ,zero ,y))))
```

2.2 Representation Strategies for Data Types

We will use the environment data type to illustrate representations.

2.2.1 The Environment Interface

The interface has three procedures:

```
(empty-env)
(apply-env f var)
(extend-env var v f)
```

Exercise 2.4

–

2.2.2 Data Structure Representation

We note that every environment can be built by starting with the empty environment and using `extend-env` some number of times. Thus we can use a grammar that imitates this process.

Exercise 2.5
What are arrows?

Exercise 2.6
–

Exercise 2.7-2.10
–

2.2.3 Procedural Representation

Since the interface has only one observer, we can represent environments as procedure. To convert a datatype to its procedural representation, first define a constructor procedure for each lambda expression that yields values of the type, and then an `apply-` procedure for places where a value of the type is applied.

The representation can be defunctionalised in languages that do not allow higher-order procedures.

Exercise 2.12
How?

2.3 Interfaces for Recursive Data Types

In general, we need to include

- one constructor for each kind of data,
- one predicate for each kind of data, and
- one extractor for each component of each kind.

Exercise 2.15

```
(define number->sequence (lambda (n)
    `(,n () ())))

(define current-element (lambda (seq)
    (car seq)))

(define move-to-left (lambda (seq)
    (cons (caadr seq)
    (cons (cdadr seq)
    (cons (cons (car seq) (caddr seq))
          '()))))))

(define move-to-right (lambda (seq)
    (cons (caaddr seq)
    (cons (cons (car seq) (cadr seq))
    (cons (cdaddr seq)
          '()))))))
```

```
(define insert-to-left (lambda (n seq)
    (cons (car seq)
    (cons (cons n (cadr seq))
    (cons (caddr seq)
          '()))))))

(define insert-to-right (lambda (n seq)
    (cons (car seq)
    (cons (cadr seq)
    (cons (cons n (caddr seq))
          '()))))))

(define at-left-end? (lambda (seq)
    (null? (cadr seq))))

(define at-right-end? (lambda (seq)
    (null? (caddr seq))))
```

2.4 A Tool for Defining Recursive Data Types

We can automatically define datatypes using this syntax:

```
(define-datatype type-name type-predicate-name
    (variant-name1
        (field-name1 predicate1)
        (field-name2 predicate2))
    (variant-name2
        (field-name predicate)))
```

Datatypes defined in this way can be mutually recursive as well.

In functions that use the datatype, we can use the `cases` syntax to manipulate different kinds:

```
(cases type-name expression
    (variant-name1 (field-name1 field-name2)
        consequent1)
    (variant-name2 (field-name)
        consequent2)
    (else
        default))
```

The form `define-datatype` is an example of a DSL.

2.5 Abstract Syntax and Its Representation

The `define-datatype` form gives us a way to define an internal representation, called the abstract syntax, which only stores the data fields and no extra symbols that may be part of the external representation (or concrete syntax).

The task of deriving the abstract syntax tree from a string of characters is called parsing. This task is easy once the input is parsed into lists and symbols; for example, in the case of lambda calculus,

```
(define parse-expression (lambda (datum)
    (cond
        ((symbol? datum) (var-exp datum))
        ((pair? datum)
            (if (eqv? (car datum) 'lambda)
                (lambda-exp
                    (cadr datum)
                    (parse-expression (caddr datum)))
                (app-exp
                    (parse-expression (car datum))
                    (parse-expression (cadr datum)))))
        (else <error-message>))))
```

Converting from an AST to a concrete representation is usually straightforward:

```
(define unparse-lc-exp (lambda (exp)
    (cases lc-exp exp
        (var-exp (var) var)
        (lambda-exp (bound-var body)
            (list 'lambda
                  (list bound-var)
                  (unparse-lc-exp body)))
        (app-exp (rator rand)
            (list (unparse-lc-exp rator)
                  (unparse-lc-exp rand))))))
```

3 Expressions

We will write specifications of small languages that illustrate binding and scoping of variables. A context argument called the *environment* keeps track of the meaning of each variable.

3.1 Specification and Implementation Strategy

The form of the specification will be

```
(value-of exp rho) = val
```

The program text in the languages will first be converted to an AST, which will be passed through the interpreter (written in the *implementation* or *defining language*). Alternatively, a compiler may convert the AST to a *target language* (usually a machine language).
The target language may also be a simpler special-purpose language (called a *byte code*), which is further interpreter by a *virtual machine*.

A compiler usually has an *analyser* (which deduces information about the program) and a *translator* (which uses this information to translate the program).

Converting programs to ASTs starts with scanning (splitting the input into tokens) and parsing (finding the structure of the program).

Parser generators are used to generate scanners and parsers for a given specification of grammar.

3.2 LET: A Simple Language

3.2.1 Specifying the Syntax

The syntax of this simple language is follows

```
Program := Expression
    a-program (exp1)

Expression := Number
    const-exp (num)
           := -(Expression , Expression)
    diff-exp (exp1 exp2)
           := zero? (Expression)
    zero?-exp (exp1)
           := if Expression then Expression else Expression
    if-exp (exp1 exp2 exp3)
           := Identifier
    var-exp (var)
           := let Identifier = Expression in Expression
    let-exp (var exp1 body)
```

For example,

```
(scan&parse "-(55, -(x,11))")
#(struct:a-program
   #(struct:diff-exp
      #(struct:const-exp 55)
      #(struct:diff-exp
         #(struct:var-exp x)
         #(struct:const-exp 11))))
```

3.2.2 Specification of Values

There are two sets of values associated with a language: the *expressed values* (the possible values of expressions) and the *denoted values* (the values bound to variables).
In this language, we will have
ExpVal = Int + Bool
DenVal = Int + Bool

The interface for the datatype of expressed values will consist of

$$\begin{aligned} \text{num-val} &: \text{Int} \to \text{ExpVal}, \\ \text{bool-val} &: \text{Bool} \to \text{ExpVal}, \\ \text{expval->num} &: \text{ExpVal} \to \text{Int}, \\ \text{expval->bool} &: \text{ExpVal} \to \text{Bool} \end{aligned}$$

3.2.3 Environments

The values associated with each variable are stored in an environment. We will denote `(extend-env var val rho)` by `[var = val]rho`. For example,

```
[x = 3]
 [y = 7]
  [u = 5]rho
```

means

```
(extend-env 'x 3
  (extend-env 'y 7
    (extend-env 'u 5 rho)))
```

3.2.4 Specifying the Behaviour of Expressions

We will have six constructors and one observers as part of the interface for expressions; ExpVal denotes the set of expressed values.

$$\begin{aligned} \text{const-exp} &: \text{Int} \to \text{Exp} \\ \text{zero?-exp} &: \text{Exp} \to \text{Exp} \\ \text{if-exp} &: \text{Exp} \times \text{Exp} \times \text{Exp} \to \text{Exp} \\ \text{diff-exp} &: \text{Exp} \times \text{Exp} \to \text{Exp} \\ \text{var-exp} &: \text{Var} \to \text{Exp} \\ \text{let-exp} &: \text{Var} \times \text{Exp} \times \text{Exp} \to \text{Exp} \end{aligned}$$

and

$$\text{value-of} : \text{Exp} \times \text{Env} \to \text{ExpVal}$$

The specification, therefore, will be

```
(value-of (const-exp n) rho) = (num-val n)
(value-of (var-exp var) rho) = (apply-env rho var)
(value-of (diff-exp exp1 exp2) rho)
    = (num-val (-
                  (expval->num (value-of exp1 rho))
                  (expval->num (value-of exp2 rho))))
```

3.2.5 Specifying the Behavious of Programs

In order to find the value of the program, we need to specify the values of the free variables, which needs an initial environment. We will use `[i = 1, v = 5, x = 10]`. Thus,

```
(value-of-program exp) = (value-of exp [i=1,v=5,x=10])
```

3.2.6 Specifying Conditionals

There is one constructor (**`zero?`**) and one observer (**`if`**) for booleans.

If `(value-of exp1 rho) = val1`, then

```
(value-of (zero?-exp exp1) rho)
  = (bool-val #t)       ; if (expval->num val1) = 0
  = (bool-val #f)       ; if (expval->num val1) != 0
```

Similarly,

```
(value-of (if-exp exp1 exp2 exp3) rho)
  = (value-of exp2 rho)         ; if (expval->bool val1) = #t
  = (value-of exp3 rho)         ; if (expval->bool val1) = #f
```

Alternatively,

```
(value-of (if-exp exp1 exp2 exp3) rho)
  = (if (expval->bool (value-of exp1 rho))
    (value-of exp2 rho)
    (value-of exp3 rho))
```

3.2.7 Specifying `let`

Again, if `(value-of exp1 rho) = val1`, then

```
(value-of (let-exp var exp1 body) rho)
  = (value-of body [var = val1]rho)
```

3.2.8 Implementing the Specification of LET

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))
```

```
(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var identifier?))
  (let-exp
    (var identifier?)
    (exp1 expression?)
    (body expression?)))

(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?)))

(define expval->num
  (lambda (val)
    (cases expval val
      (num-val (num) num)
      (else (report-expval-extractor-error 'num val)))))

(define expval->bool
  (lambda (val)
    (cases expval val
      (bool-val (bool) bool)
      (else (report-expval-extractor-error 'bool val)))))

(define run
  (lambda (string)
    (value-of-program (scan&parse string))))

(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (var-exp (var) (apply-env env var))
      (diff-exp (exp1 exp2)
        (let ((val1 (value-of exp1 env))
              (val2 (value-of exp2 env)))
          (let ((num1 (expval->num val1))
                (num2 (expval->num val2)))
            (num-val (- num1 num2)))))
       (zero?-exp (exp1)
         (let ((val1 (value-of exp1 env)))
           (let ((num1 (expval->num val1)))
             (if (zero? num1)
               (bool-val #t)
               (bool-val #f)))))
       (if-exp (exp1 exp2 exp3)
         (let ((val1 (value-of exp1 env)))
           (if (expval->bool val1)
             (value-of exp2 env)
             (value-of exp3 env))))
       (let-exp (var exp1 body)
         (let ((val1 (value-of exp1 env)))
           (value-of body
             (extend-env var val1 env)))))))
```

PROC: A Language with Procedures

In this language, we allow new procedures to be created. Thus
ExpVal = Int + Bool + Proc
DenVal = Int + Bool + Proc

Proc is considered an abstract datatype. The syntax for procedure creation and calling is given by

```
Expression := proc (Identifier) Expression
        proc-exp (var body)
           := (Expression Expression)
        call-exp (rator rand)
```

For example,

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

(proc (f) (f (f 77)) proc (x) -(x,11))
```

The Proc datatype has a constructor `procedure` and an observer `apply-procedure`. A value representing a procedure must depend on the environment in which it is evaluated; therefore we have

```
(value-of (proc-exp var body) rho)
 = (proc-val (procedure var body rho))
```

and

```
(value-of (call-exp rator rand) rho)
 = (let ((proc (expval->proc (value-of rator rho)))
         (arg (value-of rand rho)))
     (apply-procedure proc arg))
```

The invocation of `apply-procedure` should satisfy

```
(apply-procedure (procedure var body rho) val)
 = (value-of body [var=val]rho)
```

3.3.1 An Example

3.3.2 Representing Procedures

We have the following specification for `procedure`:

```
(apply-procedure (procedure var body rho) val)
 = (value-of body (extend-env var val rho))
```

Now, the implementation can be

```
(define-datatype proc proc?
  (procedure
    (var identifier?)
    (body expression?)
    (saved-env environment?)))

(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body (extend-env var val saved-env))))))
```

We also need to add

```
(proc-val
  (proc proc?))
```

to `expval`'s definition, and

```
(proc-exp (var body)
  (proc-val (procedure var body env)))
(call-exp (rator rand)
```

```
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of rand env)))
    (apply-procedure proc arg)))
```

to `value-of`'s.

3.4 LETREC: A Language with Recursive Procedures

The production for recursive declarations is

```
Expression := letrec Identifier (Identifier) = Expression in Expression
    letrec-exp (p-name b-var p-body letrec-body)
```

which has the desired behaviour

```
(value-of letrec-exp proc-name bound-var proc-body letrec-body rho)
 = (value-of letrec-body (extend-env-rec proc-name bound-var proc-body rho))
```

To find out the specification of `extend-env-rec`, let `rho1` be its return value in the above expression. Then

```
(apply-env rho1 proc-name)
 = (proc-val (procedure bound-var proc-body rho1))
```

because we need a function with bound variable `bound-var` and body `proc-body`, in an environment where `proc-name` is bound to this procedure (which is `rho1` itself); or

```
(apply-env rho1 var)
 = (apply-env rho var)
```

if `var` is not the same as `proc-name`‘.

3.5 Scoping and Binding of Variables

Variables can occur as *references* (uses of the variable) or *declarations* (introducing the variable). The reference is *bound by* the declaration with which it is associated, as in

```
(lambda (x) (+ x 3))
```

where the first `x` is the declaration that binds the reference.

Rules that determine the declaration to which each reference refers are called *scoping* rules. This can be determined without executing the program, and is therefore a *static property*.

In Scheme, we search outwards to find the correct declaration. For example, in

```
(let ((x 3)
      (y 4))
  (+ (let ((x (+y 5)))
```

```
      (* x y))
    x))
```

The first reference to `x` refers to `(+ y 5)`, and the second refers to `3`. This is an example of a *lexical scoping* rule.
This allows us to create a hole in the scope by redeclaring a variable, which *shadows* the outer declaration.

The association between a variable and its value is called a *binding*. The *extent* of a binding is the time during which it is maintained; here, we have used *semi-infinite* extent (they are maintained indefinitely once bound).

3.6 Eliminating Variable Names

The number of contours crossed while searching for the variable binding is called its *lexical* or *static depth*. We could get rid of variable names, using their depth instead; for example,

```
(lambda (x)
  ((lambda (a) (x a))
      x))
```

is replaced with

```
(nameless-lambda
  ((nameless-lambda (#1 #0))
      #0))
```

These numbers are called *lexical addresses* or *de Bruijn indices*.

3.7 Implementing Lexical Addresing

We will write `translation-of-program` that takes a program and replaces all variables with their depths. For example,

```
let x = 37
in proc (y)
    let z = -(y,x)
    in -(x,y)
```

is translated to

```
#(struct:a-program
  #(struct:nameless-let-exp
    #(struct:const-exp 37)
    #(struct:nameless-proc-exp
      #(struct:nameless-let-exp
        #(struct:diff-exp
          #(struct:nameless-var-exp 0)
          #(struct:nameless-var-exp 1))
```

```
        #(struct:diff-exp
          #(struct:nameless-var-exp 2)
          #(struct:nameless-var-exp 1))))))
```

3.7.1 The Translator

The `translation-of` procedure will take 2 arguments; the program, and a static environment (which is a simple list of variables giving the scopes of each one). Entering a new scope means adding a new variable to the list.

```
(define empty-senv
  (lambda ()
    '()))

(define extend-senv
  (lambda (var senv)
    (cons var senv)))

(define apply-senv
  (lambda (senv var)
    (cond
      ((null? senv)
         (report-unbound-var var))
      ((eqv? var (car senv)) 0)
      (else
        (+ 1 (apply-senv (cdr senv) var))))))
```

The changes that `translation-of` makes are

- Every `var-exp` is replaced by `nameless-var-exp` with the right lexical address.
- Every `let-exp` is replaced by `nameless-let-exp`, with the RHS translated in the same scope, and the body translated in the old scope extended by one binding.
- Every `proc-exp` is replaced by `nameless-proc-exp` with the body translated in the new scope.

`translation-of-program` runs `translation-of` in a suitable static environment.

3.7.2 The Nameless Interpreter

We will use nameless environments, with the following interface:

$$\begin{aligned}
\text{nameless-environment?} &: \text{SchemeVal} \to \text{Bool} \\
\text{empty-nameless-env} &: () \to \text{Nameless-Env} \\
\text{extend-nameless-env} &: \text{ExpVal} \times \text{Nameless-Env} \to \text{Nameless-Env} \\
\text{apply-nameless-env} &: \text{Nameless-Env} \times \text{Lexaddr} \to \text{DenVal}
\end{aligned}$$

The procedures are defined similar to those for static environments; lookup is carried out by simple indexing.

4 State

4.1 Computational Effects

Effects produced by computations are different from values in that they are *global*. There are many types of effects, but we will be concerned with variable assignment.

Memory is modelled as a map from locations to *storable values*, called the *store* (the storable values are typically the same as the expressed values).

A data structure representing a location is called a *reference* (the location/reference distinction is analogous to the URL/file distinction). References are sometimes called *lvalues*, and expressed values *rvalues*.

We have two designs for a language with a store, which we call *explicit* and *implicit* references.

4.2 EXP-REFS: A Languages with Explicit References

Here,

$$\begin{aligned}\text{ExpVal} &= \text{Int} + \text{Bool} + \text{Proc} + \text{Ref(ExpVal)}\\ \text{DenVal} &= \text{ExpVal}\end{aligned}$$

where Ref(ExpVal) is the set of references to locations that contain expressed values.

We add three new operators: `newref` (which allocates and returns a reference to a new location), `deref` (which dereferences a reference), and `setref` (which changes the contents of the location represented by a reference).

Communication via a shared variable is convenient as it allows for hiding (intermediate procedures needn't know about a shared value if the call is not direct). Assignment can also create a hidden state through the use of private variables.

4.2.1 Store-Passing Specifications

In a store-passing specification, the store is passed as an explicit argument to `valueof` and is returned from it as a result. Thus, for example,

```
(valueof (Const n) rho sigma) = (n, sigma)
```

In the case of difference-expressions, the operands are evaluated in order, passing the store from each evaluation to the other, followed by the subtraction which

returns the last state. In this way we can write the specifications of all the expression types.

We also have a new keyword `begin`:

```
Expression := begin Expression {; Expression}* end
```

4.2.2 Specifying Operations on Explicit References

We have three new operations:

```
Expression := newref ( Expression )
           := deref ( Expression )
           := setref (Expression , Expression)
```

Note that `setref` is executed *for effect*, rather than value; it returns an arbitrary value.

3.2.4 Implementation

They use a global Scheme variable (`:vomiting_face:`), but we'll pass it around like normal people (store-passing). The store is implemented as a list of values, and references as indices into the list.

4.3 IMP-REFS: A Language with Implicity References

EXP-REFS gives a clear account of allocation, dereferencing and mutation, as these are all explicit. Most PLs, however, abstract these away from the programmer's code.

Here, every variable denotes a reference; denoted values are references to locations that contain expressed values. Thus references are no longer expressed values, and exist only as bindings.

$$\begin{aligned} \text{ExpVal} &= \text{Int} + \text{Bool} + \text{Proc} \\ \text{DenVal} &= \text{Ref}(\text{ExpVal}) \end{aligned}$$

Locations are created with each binding operation, *i.e.*, procedure call, `let` or `letrec`.

When a variable appears in an expression, we look it up in the environment to find the location it refers to, and then look up the store to find the value. We use `set` to change the contents of a location, which is expressed as an `Assign` statement in the AST.

4.3.1 Specification

Whenever a variable appears as an expression, it has to be dereferenced by looking it up in the environment and the store. Assignment also occurs similarly to `setref`.

For `let` and `apply-procedure`, we need to create new locations and modify the store.

4.3.2 The Implementation

For variable expressions and assignments, we continue to use `deref` and `setref`.

For creating references, four cases must be considered: the initial environment, `let`, a procedure a call and a `letrec`.
For the initial environment, the allocation is explicit.
For `let`, we allocate a new location containing the value (using `newref`) and bind the variable to reference it.
For procedure calls, too, we call `newref`.
For `letrec`, we modify `extendenvrec` to return a reference to the location containing the closure.