Chapter 7

Languages with Contexts

Virtually all languages rely on some notion of context. The context in which a phrase is used influences its meaning. In a programming language, contexts are responsible for attributing meanings to identifiers. There are several possible notions of a programming language context; let's consider two of them. The first, a simplistic view, is that the store establishes the context for a phrase. The view works with the languages studied in Chapter 5, but it does suggest that the context within the block:

```
begin
    integer X; integer Y;
    Y:=0;
    X:=Y;
    Y:=1;
    X:=Y+1
end
```

is constantly changing. This is counterintuitive; surely the declarations of the identifiers X and Y establish the context of the block, and the commands within the block operate within that context. A second example:

```
begin integer X;
    X:=0;
    begin real X;
        X:=1.5
    end;
    X:=X+1
end
```

shows that the meaning of an identifier isn't just its storable value. In this example, there are two distinct definitions of X. The outer X denotes an *integer* object, while the inner X is a *real* object. These objects are different; X happens to be the name used for both of them. Any potential ambiguities in using X are handled by the scope rules of ALGOL60. The ''objects'' mentioned are in actuality computer storage locations, and the primary meaning of an ALGOL60 identifier is the location bound to it. The version of context we choose to use is the set of identifier, storage location pairs that are accessible at a textual position. Each position in the program resides within a unique context, and the context can be determined without running the program.

In denotational semantics, the context of a phrase is modelled by a value called an *environment.* Environments possess several distinctive properties:

1. As mentioned, an environment establishes a context for a syntactic phrase, resolving any ambiguities concerning the meaning of identifiers.
2. There are as many environment values as there are distinct contexts in a program.

Multiple environments may be maintained during program evaluation.

3. An environment is (usually) a static object. A phrase uses the same environment each time it is evaluated with the store.

An environment argument wasn't needed for the languages in Chapter 5, because the programs in the languages used exactly one environment. The single environment was ''pasted onto'' the store, giving a map from identifiers to storable values. In this chapter, that simple model is split apart into two separate components, the environment and the store.

The primary real-life example of an environment is a complier's symbol table. A compiler uses a symbol table to translate a source program into compiled code. The symbol table contains an entry for each identifier in the program, listing the identifier's data type, its mode of usage (variable, constant, parameter, . . .), and its relative location in the run-time computer store. Since a block-structured language allows multiple uses of the same identifier, the symbol table is responsible for resolving naming conflicts. The schemes for implementation are many: one is to keep a different symbol table for each block of the program (the portions in common between blocks may be shared); another is to build the table as a single stack, which is incremented and decremented upon respective entry and exit for a block. Symbol tables may be entirely compile-time objects, as in ALGOL68 and standard Pascal, or they can be run-time objects, as in SNOBOL4, or they can be used in both phases, as in ALGOL60.

Those portions of a semantics definition that use an environment to resolve context questions are sometimes called the *static semantics.* The term traditionally describes compile-time actions such as type-checking, scope resolution, and storage calculation. Static semantics may be contrasted with the ''real'' production of meaning, which takes the name *dynamic semantics.* Code generation and execution comprise the implementation-oriented version of dynamic semantics. In general, the separation of static from dynamic semantics is rarely clear cut, and we will not attempt formal definitions here.

Environments are used as arguments by the valuation functions. The meaning of a command is now determined by the function:

$$\mathbf{C}\text{: Command} \to \mathit{Environment} \to \mathit{Store} \to \mathit{Store}_{\perp}$$

The meaning of a command as a $\mathit{Store} \to \mathit{Store}_{\perp}$ function is determined once an environment establishes the context for the command. An environment belongs to the domain:

$$\mathit{Environment} = \mathit{Identifier} \to \mathit{Denotable\text{-}value}$$

The *Denotable-value* domain contains all the values that identifiers may represent. This domain varies widely from language to language and its structure largely determines the character of the language.

In this chapter, we study language features whose semantics are understood in terms of environments. These features include declarations, block structure, scoping mechanisms, recursive bindings, and compound data structures. The concepts are presented within the framework of two languages: an imperative block-structured language and an applicative language.

7.1 A BLOCK-STRUCTURED LANGUAGE

The basic principles and uses of environments are seen in the semantics of a simple block-structured language. The language is similar to the one defined in Figure 5.2, but includes declarations and blocks. The new semantic domains are listed in Figure 7.1.

The more realistic store requires a primitive domain of storage locations, and the locations domain is listed first. The operations are the same as in the algebra in Example 3.4 of Chapter 3: *first-locn* is a constant, marking the first usable location in a store; *next-locn* maps a location to its immediate successor in a store; *equal-locn* checks for equality of two values; and *lessthan-locn* compares two locations and returns a truth value based on the locations' relative values.

The collection of values that identifiers may represent is listed next. Of the three components of the *Denotable-value* domain, the *Location* domain holds the denotations of variable identifiers, the *Nat* domain holds the meanings of constant identifiers, and the *Errvalue* domain holds the meaning for undeclared identifiers.

For this language, an environment is a pair. The first component is the function that maps identifiers to their denotable values. The second component is a location value, which marks the extent of the store reserved for declared variables. In this example, the environment takes the responsibility for assigning locations to variables. This is done by the *reserve-locn* operation, which returns the next usable location. Although it is not made clear by the algebra, the structure of the language will cause the locations to be used in a stack-like fashion. The *emptyenv* must be given the location marking the beginning of usable space in the store so that it can build the initial environment.

The store is a map from storage locations to storable values, and the operations are the obvious ones. Errors during evaluation are possible, so the store will be labeled with the status of the evaluation. The *check* operation uses the tags to determine if evaluation should continue.

Figure 7.2 defines the block-structured language and its semantics.

Since the *Denotable-value* domain contains both natural numbers and locations, denotable value errors may occur in a program; for example, an identifier with a number denotation might be used where an identifier with a location denotation is required. An identifier with an erroneous denotable value always induces an error. Expressible value errors occur when an expressible value is inappropriately used.

The **P** valuation function requires a store and a location value, the latter marking the beginning of the store's free space. The **K** function establishes the context for a block. The **D** function augments an environment. The composition of declarations parallels the composition of commands. A constant identifier declaration causes an environment update, where the identifier is mapped to its numeral value in the environment. The denotation of a variable declaration is more involved: a new location is reserved for the variable. This location, l_1, plus the current environment, e_1, are used to create the environment in which the variable $[\![\mathrm{I}]\!]$ binds to in*Location*(l_1).

Of the equations for the **C** function, the one for composition deserves the most study. First, consider the *check* operation. If command $\mathbf{C}[\![\mathrm{C}_1]\!]e$ maps a store into an erroneous post-store, then *check* traps the error and prevents $\mathbf{C}[\![\mathrm{C}_2]\!]e$ from altering the store. This would be implemented as a branch around the code for $[\![\mathrm{C}_2]\!]$. The environment is also put to good use:

Figure 7.1

V. Storage locations
Domain $l \in Location$
Operations

$first\text{-}locn : Location$

$next\text{-}locn : Location \to Location$

$equal\text{-}locn : Location \to Location \to Tr$

$lessthan\text{-}locn : Location \to Location \to Tr$

VI. Denotable values
Domain $d \in Denotable\text{-}value = Location + Nat + Errvalue$
where $Errvalue = Unit$

VII. Environment: a map to denotable values and the maximum store location
Domain $e \in Environment = (Id \to Denotable\text{-}value) \times Location$
Operations

$emptyenv : Location \to Environment$
$emptyenv = \lambda l.\ ((\lambda i.\ \text{in}Errvalue\ ()),\ l)$

$accessenv : Id \to Environment \to Denotable\text{-}value$
$accessenv = \lambda i.\lambda(map,\ l).\ map(i)$

$updateenv : Id \to Denotable\text{-}value \to Environment \to Environment$
$updateenv = \lambda i.\lambda d.\lambda(map,\ l).\ ([\ i \mapsto d\ ]map,\ l)$

$reserve\text{-}locn : Environment \to (Location \times Environment)$
$reserve\text{-}locn = \lambda(map,\ l).\ (l,\ (map,\ next\text{-}locn(l)))$

VIII. Storable values
Domain $v \in Storable\text{-}value = Nat$

IX. Stores
Domain $s \in Store = Location \to Storable\text{-}value$
Operations

$access : Location \to Store \to Storable\text{-}value$
$access = \lambda l.\lambda s.\ s(l)$

$update : Location \to Storable\text{-}value \to Store \to Store$
$update = \lambda l.\lambda v.\lambda s.\ [\ l \mapsto v\ ]s$

X. Run-time store, labeled with status of computation
Domain $p \in Poststore = OK + Err$
where $OK = Err = Store$

Figure 7.1 (continued)

Operations

return: *Store* → *Poststore*
return = λ*s*. in*OK*(*s*)

signalerr: *Store* → *Poststore*
signalerr = λ*s*. in*Err*(*s*)

check: (*Store* → *Poststore*$_\perp$) → (*Poststore*$_\perp$ → *Poststore*$_\perp$)
check f = $\underline{\lambda}$*p*. cases *p* of
is*OK*(*s*) → (*f s*)
[] is*Err*(*s*) → *p* end

Figure 7.2

Abstract syntax:

P∈ Program
K∈ Block
D∈ Declaration
C∈ Command
E∈ Expression
B∈ Boolean-expr
I∈ Identifier
N∈ Numeral

P ::= K.

K ::= **begin** D; C **end**

D ::= D_1;D_2 | **const** I=N | **var** I

C ::= C_1;C_2 | I:=E | **while** B **do** C | K

E ::= E_1+E_2 | I | N

Semantic algebras:

I.-III. Natural numbers, truth values, identifiers
(defined in Figure 5.1)

IV. Expressible values
Domain $x \in \mathit{Expressible\text{-}value} = \mathit{Nat} + \mathit{Errvalue}$
where $\mathit{Errvalue} = \mathit{Unit}$

V.-X. (defined in Figure 7.1)

Valuation functions:

P: Program$\to \mathit{Location} \to \mathit{Store} \to \mathit{Poststore}_\perp$
$\mathbf{P}[\![\text{K.}]\!] = \lambda l.\ \mathbf{K}[\![\text{K}]\!](\mathit{emptyenv}\ l)$

K: Block$\to \mathit{Environment} \to \mathit{Store} \to \mathit{Poststore}_\perp$
$\mathbf{K}[\![\textbf{begin}\ \text{D;C}\ \textbf{end}]\!] = \lambda e.\ \mathbf{C}[\![\text{C}]\!](\mathbf{D}[\![\text{D}]\!]e)$

D: Declaration$\to \mathit{Environment} \to \mathit{Environment}$
$\mathbf{D}[\![\text{D}_1;\text{D}_2]\!] = \mathbf{D}[\![\text{D}_2]\!] \circ \mathbf{D}[\![\text{D}_1]\!]$
$\mathbf{D}[\![\textbf{const}\ \text{I=N}]\!] = \mathit{updateenv}[\![\text{I}]\!]\ \text{in}\mathit{Nat}(\mathbf{N}[\![\text{N}]\!])$
$\mathbf{D}[\![\textbf{var}\ \text{I}]\!] = \lambda e.\text{let}\ (l', e') = (\mathit{reserve\text{-}locn}\ e)\ \text{in}\ (\mathit{updateenv}[\![\text{I}]\!]\ \text{in}\mathit{Location}(l')\ e')$

C: Command $\to \mathit{Environment} \to \mathit{Store} \to \mathit{Poststore}_\perp$
$\mathbf{C}[\![\text{C}_1;\text{C}_2]\!] = \lambda e.\ (\mathit{check}(\mathbf{C}[\![\text{C}_2]\!]e)) \circ (\mathbf{C}[\![\text{C}_1]\!]e)$

Figure 7.2 (continued)

$\mathbf{C}[\![\text{I:=E}]\!] = \lambda e.\lambda s.$ cases $(\mathit{accessenv}[\![\text{I}]\!]\ e)$ of
  is$\mathit{Location}(l) \rightarrow$ (cases $(\mathbf{E}[\![\text{E}]\!] e\ s)$ of
    is$\mathit{Nat}(n) \rightarrow (\mathit{return}(\mathit{update}\ l\ n\ s))$
    [] is$\mathit{Errvalue}() \rightarrow (\mathit{signalerr}\ s)$ end)
  [] is$\mathit{Nat}(n) \rightarrow (\mathit{signalerr}\ s)$
  [] is$\mathit{Errvalue}() \rightarrow (\mathit{signalerr}\ s)$ end

$\mathbf{C}[\![\textbf{while}\ \text{B}\ \textbf{do}\ \text{C}]\!] = \lambda e.\ \mathit{fix}(\lambda f.\lambda s.$ cases $(\mathbf{B}[\![\text{B}]\!] e\ s)$ of
  is$\mathit{Tr}(t) \rightarrow (t \rightarrow (\mathit{check}\ f) \circ (\mathbf{C}[\![\text{C}]\!] e)\ []\ \mathit{return})(s)$
  [] is$\mathit{Errvalue}() \rightarrow (\mathit{signalerr}\ s)$ end)

$\mathbf{C}[\![\text{K}]\!] = \mathbf{K}[\![\text{K}]\!]$

E: Expression $\rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow \mathit{Expressible\text{-}value}$

$\mathbf{E}[\![\text{E}_1\text{+E}_2]\!] = \lambda e.\lambda s.$ cases $(\mathbf{E}[\![\text{E}_1]\!] e\ s)$ of
  is$\mathit{Nat}(n_1) \rightarrow$ (cases $(\mathbf{E}[\![\text{E}_2]\!] e\ s)$ of
    is$\mathit{Nat}(n_2) \rightarrow$ in$\mathit{Nat}(n_1\ \mathit{plus}\ n_2)$
    [] is$\mathit{Errvalue}() \rightarrow$ in$\mathit{Errvalue}()$ end)
  [] is$\mathit{Errvalue}() \rightarrow$ in$\mathit{Errvalue}()$ end

$\mathbf{E}[\![\text{I}]\!] = \lambda e.\lambda s.$ cases $(\mathit{accessenv}[\![\text{I}]\!]\ e)$ of
  is$\mathit{Location}(l) \rightarrow$ in$\mathit{Nat}(\mathit{access}\ l\ s)$
  [] is$\mathit{Nat}(n) \rightarrow$ in$\mathit{Nat}(n)$
  [] is$\mathit{Errvalue}() \rightarrow$ in$\mathit{Errvalue}()$ end

$\mathbf{E}[\![\text{N}]\!] = \lambda e.\lambda s.$ in$\mathit{Nat}(\mathbf{N}[\![\text{N}]\!])$

B: Boolean-expr $\rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow (\mathit{Tr} + \mathit{Errvalue})$ (omitted)

N: Numeral $\rightarrow \mathit{Nat}$ (omitted)

---

commands $[\![\text{C}_1]\!]$ and $[\![\text{C}_2]\!]$ are both evaluated in the context represented by *e*. This point is important, for $[\![\text{C}_1]\!]$ could be a block with local declarations. It would need its own local environment to process its commands. However, $\mathbf{C}[\![\text{C}_2]\!]$ retains its own copy of *e*, so the environments created within $\mathbf{C}[\![\text{C}_1]\!]$ do not affect $\mathbf{C}[\![\text{C}_2]\!]$'s version. (Of course, whatever alterations $\mathbf{C}[\![\text{C}_1]\!]e$ makes upon the store are passed to $\mathbf{C}[\![\text{C}_2]\!]e$.) This language feature is called *static scoping*. The context for a phrase in a statically scoped language is determined solely by the textual position of the phrase. One of the benefits of static scoping is that any identifier declared within a block may be referenced only by the commands within that block. This makes the management of storage locations straightforward. So-called *dynamically scoped* languages, whose contexts are not totally determined by textual position, will be studied in a later section.

The meaning of an expression supplied with an environment and store is an expressible value. Error checking also occurs at the expression level, as an undeclared identifier might be used in an expression.

A key feature of this language, as well as other strongly typed languages, is that environment processing can proceed independently of store processing. That is, the denotation **P**⟦P⟧ can be simplified without the values of the initial base location *l* and the initial store *s* (see Section 5.1.1 of chapter 5). The result is a smaller function expression that contains no occurrences of environment arguments nor of the *cases* expressions that check denotable and expressible value tags. The simplifications correspond to the declaration and type-checking actions that occur in a compiler.

The following example demonstrates this point. Since the value of the run-time store is not known, we do no simplification on any operations from the algebras *Store* and *Poststore.* Expressions using these operations are ''frozen''— their evaluation is delayed until run-time. The static semantics analysis of a program is presented in Figure 7.3.

The diagrammatic form:

$$(\cdots E \cdots)$$

$$E'$$

represents the simplification of expression E to E' and its replacement in the larger expression, giving $(\cdots E' \cdots)$. Notice how the environment arguments distribute throughout the commands and their subparts without interfering with the frozen *check*(λ*s.* ⋯) expression forms. All *Denotable-value* and *Expressible-value* labels are consumed by the *cases* expressions. As simplification proceeds, environment arguments disappear, variables map to their location values, and constants reveal their natural number values. The final expression can be read more easily if reverse function composition is used: let $f \,!\, g$ stand for $g \circ f$. Then the result is:

```
λl. (λs. return(update l (one plus two) s))
    ! (check(fix(λf.λs.
        ((access l s) equals zero →
            (λs. return(update(next-locn l) (access l s) s)) ! (checkf)
            [] return
        ) s))
    ! (check(λs. return(update l one s))))
```

The expression rightly resembles a series of machine code instructions, parameterized on a store's base address *l.* In Chapter 10, we study the mapping of expressions such as this one into object code for a real machine.

Figure 7.3

Let D_0 = **const** A=1
D_1 = **var** X
$C_0 = C_1;C_2;C_3$
C_1 = X:=A+2
C_2 = **begin var** A;C_4 **end**
C_3 = X:=A
C_4 = **while** X=0 **do** A:=X

P⟦**begin** $D_0;D_1;C_0$ **end**⟧

$\lambda l.$ **K**⟦**begin** $D_0;D_1:C_0$ **end**⟧(*emptyenv l*)

let this be e_0

C⟦C_0⟧(**D**⟦$D_0;D_1$⟧e_0)

D⟦D_1⟧(**D**⟦**const** A=1⟧e_0)

(*updateenv*⟦A⟧ in*Nat*(*one*) e_0)

let this be e_1= (*map*,*l*)

D⟦**var** X⟧e_1

let (l_1,e_1)= (*reserve-locn* e_1) in · · ·

(*l*, (*map*, (*next-locn l*)))

e_2

(*updateenv*⟦X⟧ in*Location*(*l*) e_2)

e_3

(*check*(**C**⟦$C_2;C_3$⟧e_3)) ∘ (**C**⟦X:=A+2⟧e_3)

$\lambda s.$ cases (*accessenv*⟦X⟧ e_3) of · · · end

in*Location*(*l*)

Figure 7.3 (continued)

cases (**E**⟦A+2⟧e_3 *s*) of · · · end

in*Nat*(*one plus two*)

return(*update l* (*one plus two*) *s*)

(*check*(**C**⟦C_3⟧e_3)) ∘ (**C**⟦**begin var** A; C_4 **end**⟧e_3)

C⟦C_4⟧(**D**⟦**var** A⟧e_3)

(*updateenv*⟦A⟧ in*Location* (*next-locn l*) e_4)

e_5

C⟦**while** X=0 **do**A:=X⟧e_5

fix(λ*f*. λ*s*. cases (**B**⟦X=0⟧e_5 *s*) of · · · end

in*Tr*((*access l s*) *equals zero*)

((*access l s*) *equals zero*→
(*checkf*) ∘ (**C**⟦A:=X⟧e_5) [] *return*) *s*

λ*s*. *return*(*update*(*next-locn l*) (*access l s*) *s*)

C⟦X:=A⟧e_3

λ*s*. *return*(*update l one s*)

7.1.1 Stack-Managed Storage

The store of a block-structured language is used in a stack-like fashion— locations are bound to identifiers sequentially using *nextlocn*, and a location bound to an identifier in a local block is freed for re-use when the block is exited. The re-use of locations happens automatically due to the equation for **C**⟦C_1;C_2⟧. Any locations bound to identifiers in ⟦C_1⟧ are reserved by the environment built from *e* for **C**⟦C_1⟧, but **C**⟦C_2⟧ re-uses the original *e* (and its original location marker), effectively deallocating the locations.

Stack-based storage is a significant characteristic of block-structured programming languages, and the *Store* algebra deserves to possess mechanisms for stack-based allocation and deallocation. Let's move the storage calculation mechanism over to the store algebra.

Figure 7.4 presents one of many possible results.

The new store domain uses the (*Location* → *Storable-value*) component as the data space of the stack, and the *Location* component indicates the amount of storage in use— it is the ''stack top marker.'' Operations *access* and *update* verify that any reference to a storage location is a valid one, occurring at an active location beneath the stack top. The purposes of *mark-locn* and *allocate-locn* should be clear; the latter is the run-time version of the environment's *reserve-locn* operation. The *deallocate-locns* operation releases stack storage from the stack top to the value indicated by its argument.

Freed from storage management, the environment domain takes the form *Environment* = *Id* → *Denotable-value*. The operations are adjusted accordingly, and the operation *reserve-locn* is dropped. (This is left as an easy exercise.) If the environment leaves the task of storage calculation to the store operations, then processing of declarations requires the store as well as the environment. The functionality of the valuation function for declarations becomes:

D: Declaration → *Environment* → *Store* → (*Environment* × *Poststore*)

$\mathbf{D}[\![\textbf{var I}]\!] = \lambda e.\lambda s.$ let $(l, p) = (\textit{allocate-locn}\ s)$
in $((\textit{updateenv}[\![\text{I}]\!]\ \text{in}\textit{Location}(l)\ e),\ p)$

$\mathbf{D}[\![\text{D}_1;\text{D}_2]\!] = \lambda e.\lambda s.$ let $(e', p) = (\mathbf{D}[\![\text{D}_1]\!] e\ s)$ in $(\textit{check}\ \mathbf{D}[\![\text{D}_2]\!] e')(p)$

where *check*: (*Store* → (*Environment* × *Poststore*)) → (*Poststore* → (*Environment*

Figure 7.4

IX'. Stack-based store

Domain *Store* = (*Location* → *Storable-value*) × *Location*

Operations

access: *Location* → *Store* → (*Storable-value* + *Errvalue*)
access = $\lambda l.\lambda$(*map, top*). *l lessthan-locn top* → in*Storable-value*(*map l*)
[] in*Errvalue*()

update: *Location* → *Storable-value* → *Store* → *Poststore*
update = $\lambda l.\lambda v.\lambda$(*map, top*). *l lessthan-locn top* → in*OK*($[l \mapsto v]$*map, top*)
[] in*Err*(*map, top*)

mark-locn: *Store* → *Location*
mark-locn = λ(*map, top*). *top*

allocate-locn: *Store* → *Location* × *Poststore*
allocate-locn = λ(*map, top*). (*top,* in*OK*(*map, next-locn*(*top*)))

deallocate-locns: *Location* → *Store* → *Poststore*
deallocate-locns = $\lambda l.\lambda$(*map, top*). (*l lessthan-locn top*)
or (*l equal-locn top*) → in*OK*(*map, l*) [] in*Err*(*map, top*)

× *Poststore*)) behaves like its namesake in Figure 7.1. This version of declaration processing makes the environment into a run-time object, for the binding of location values to identifiers cannot be completed without the run-time store. Contrast this with the arrangement in Figure 7.2, where location binding is computed by the environment operation *reserve-locn*, which produced a result relative to an arbitrary base address. A solution for freeing the environment from dependence upon *allocate-locn* is to provide it information about storage management strategies, so that the necessary address calculations can be performed independently of the value of the run-time store. This is left as an exercise.

The **K** function manages the storage for the block:

K⟦**begin** D;C **end**⟧ = λ*e*.λ*s*. let *l* = *mark-locn s* in
let (*e*′, *p*) = **D**⟦D⟧*e s* in
let *p*′ = (*check*(**C**⟦C⟧*e*′))(*p*)
in (*check*(*deallocate-locns l*))(*p*′)

The *deallocate-locns* operation frees storage down to the level held by the store prior to block entry, which is (*mark-locn s*).

7.1.2 The Meanings of Identifiers

The notion of context can be even more subtle than we first imagined. Consider the Pascal assignment statement X:=X+1. The meaning of X on the right-hand side of the assignment is decidedly different from X's meaning on the left-hand side. Specifically, the ''left-hand side value'' is a location value, while the ''right-hand side value'' is the storable value associated with that location. Apparently the context problem for identifiers is found even at the primitive command level.

One way out of this problem would be to introduce two environment arguments for the semantic function for commands: a left-hand side one and a right-hand side one. This arrangement is hardly natural; commands are the ''sentences'' of a program, and sentences normally operate in a single context. Another option is to say that any variable identifier actually denotes a pair of values: a location value and a storable value. The respective values are called the identifier's *L-value* and *R-value.* The L-value for a variable is kept in the environment, and the R-value is kept in the store. We introduce a valuation function **I**: *Id* → *Environment* → *Store* → (*Location* × *Storable-value*). In practice, the **I** function is split into two semantic functions **L**: *Id* → *Environment* → *Location* and **R**: *Id* → *Environment* → *Store* → *Storable-value* such that:

L⟦I⟧ = *accessenv*⟦I⟧
R⟦I⟧ = *access* ∘ *accessenv*⟦I⟧.

We restate the semantic equations using variables as:

C⟦I:=E⟧ = λ*e*.λ*s*. *return*(*update*(**L**⟦I⟧*e*) (**E**⟦E⟧*e s*) *s*)
E⟦I⟧= **R**⟦I⟧

The definitions are a bit simplistic because they assume that all identifiers are variables. Constant identifiers can be integrated into the scheme — a declaration such as ⟦**const** A=N⟧ suggests **L**⟦A⟧*e*= in*Errvalue*(). (What should (**R**⟦A⟧*e s*) be?)

Yet another view to take is that the R-value of a variable identifier is a function of its L-value. The ''true meaning'' of a variable is its L-value, and a ''coercion'' occurs when a variable is used on the right-hand side of an assignment. This coercion is called *dereferencing.* We formalize this view as:

J: Id → *Environment* → *Denotable-value*
J⟦I⟧= λ*e*. (*accessenv*⟦I⟧ *e*)

C⟦I:=E⟧ = λ*e*.λ*s*. *return*(*update*(**J**⟦I⟧*e*) (**E**⟦E⟧*e s*) *s*)
E⟦I⟧ = λ*e*.λ*s*. *dereference* (**J**⟦I⟧*e*) *s*
where *dereference*: *Location* → *Store* → *Storable-value*
dereference = *access*

An identifier's meaning is just its denotable value. Those identifiers with locations as their meanings (the variables) are dereferenced when an expressible value is needed. This is the view that was taken in the previous sections.

The implicit use of dereferencing is so common in general purpose programming languages that we take it for granted, despite the somewhat unorthodox appearance of commands such as X=X+1 in FORTRAN. Systems-oriented programming languages such as BCPL, Bliss, and C use an explicit dereferencing operator. For example, in BCPL expressible values include locations, and the appropriate semantic equations are:

E⟦I⟧ = λ*e*.λ*s*. in*Location*(**J**⟦I⟧*e*)
E⟦@E⟧ = λ*e*.λ*s*. cases (**E**⟦E⟧*e s*) of
is*Location*(*l*)→ (*dereference l s*)
[] ··· end

The @ symbol is the dereferencing operator. The meaning of X:=X+1 in BCPL is decidedly different from that of X:=@X+1.

7.2 AN APPLICATIVE LANGUAGE

The next example language using environments is an *applicative language.* An applicative language contains no variables. All identifiers are constants and can be given attributes but once, at their point of definition. Without variables, mechanisms such as assignment are superfluous and are dropped. Arithmetic is an applicative language. Another example is the minimal subset of LISP known as ''pure LISP.'' The function notation that we use to define denotational definitions can also be termed an applicative language. Since an applicative language has no variables, its semantics can be specified without a *Store* domain. The environment holds the attributes associated with the identifiers.

The language that we study is defined in Figure 7.5. It is similar to pure LISP. A

Figure 7.5

Abstract syntax:

E∈ Expression
A∈ Atomic-symbol
I∈ Identifier

E ::= LET I = E_1 IN E_2 | LAMBDA (I) E | E_1 E_2 |
E_1 CONS E_2 | HEAD E | TAIL E | NIL | I | A | (E)

Semantic algebras:

I. Atomic answer values
Domain $a \in Atom$
Operations
(omitted)

II. Identifiers
Domain $i \in Id =$ Identifier
Operations
(usual)

III. Denotable values, functions, and lists
Domains $d \in Denotable\text{-}value = (Function + List + Atom + Error)_\perp$
$f \in Function = Denotable\text{-}value \to Denotable\text{-}value$
$t \in List = Denotable\text{-}value^*$
$Error = Unit$

IV. Expressible values
Domain $x \in Expressible\text{-}value = Denotable\text{-}value$

V. Environments
Domain $e \in Environment = Id \to Denotable\text{-}value$
Operations

$accessenv : Id \to Environment \to Denotable\text{-}value$
$accessenv = \lambda i.\lambda e.\ e\,(i)$

$updateenv : Id \to Denotable\text{-}value \to Environment \to Environment$
$updateenv = \lambda i.\lambda d.\lambda e.\ [\,i \mapsto d\,]e$

Valuation functions:

E: Expression $\to$ *Environment* $\to$ *Expressible-value*
$\mathbf{E}[\![\text{LET I=}E_1\text{ IN }E_2]\!] = \lambda e.\ \mathbf{E}[\![E_2]\!](updateenv[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!]e)\ e)$
$\mathbf{E}[\![\text{LAMBDA (I) E}]\!] = \lambda e.\ \text{in}Function(\lambda d.\ \mathbf{E}[\![\text{E}]\!](updateenv[\![\text{I}]\!]\ d\ e))$

Figure 7.5 (continued)

$\mathbf{E}[\![\mathrm{E}_1\ \mathrm{E}_2]\!] = \lambda e.$ let $x = (\mathbf{E}[\![\mathrm{E}_1]\!]e)$ in cases x of
is*Function*(f) → $f(\mathbf{E}[\![\mathrm{E}_2]\!]e)$
[] is*List*(t) → in*Error*()
[] is*Atom*(a) → in*Error*() [] is*Error*() → in*Error*() end

$\mathbf{E}[\![\mathrm{E}_1$ CONS $\mathrm{E}_2]\!] = \lambda e.$ let $x = (\mathbf{E}[\![\mathrm{E}_2]\!]e)$ in cases x of
is*Function*(f) → in*Error*()
[] is*List*(t) → in*List*($\mathbf{E}[\![\mathrm{E}_1]\!]e$ *cons* t)
[] is*Atom*(a) → in*Error*() [] is*Error*() → in*Error*() end

$\mathbf{E}[\![$HEAD E$]\!] = \lambda e.$ let $x = (\mathbf{E}[\![\mathrm{E}]\!]e)$ in cases x of
is*Function*(f) → in*Error*()
[] is*List*(t) → (*null* t → in*Error*() [] (*hd* t))
[] is*Atom*(a) → in*Error*() [] is*Error*() → in*Error*() end

$\mathbf{E}[\![$TAIL E$]\!] = \lambda e.$ let $x = (\mathbf{E}[\![\mathrm{E}]\!]e)$ in cases x of
is*Function*(f) → in*Error*()
[] is*List*(t) → (*null* t → in*Error*() [] in*List*(*tl* t))
[] is*Atom*(a) → in*Error*() [] is*Error*() → in*Error*() end

$\mathbf{E}[\![$NIL$]\!] = \lambda e.$ in*List*(*nil*)

$\mathbf{E}[\![\mathrm{I}]\!] =$ *accessenv*$[\![\mathrm{I}]\!]$

$\mathbf{E}[\![\mathrm{A}]\!] = \lambda e.$ in*Atom*($\mathbf{A}[\![\mathrm{A}]\!]$)

$\mathbf{E}[\![(\mathrm{E})]\!] = \mathbf{E}[\![\mathrm{E}]\!]$

A: Atomic-symbol → *Atom* (omitted)

program in the language is just an expression. An expression can be a LET definition; a LAMBDA form (representing a function routine with parameter I); a function application; a list expression using CONS, HEAD, TAIL, or NIL; an identifier; or an atomic symbol. We learn much about the language by examining its semantic domains. *Atom* is a primitive answer domain and its internal structure will not be considered. The language also contains a domain of functions, which map denotable values to denotable values; a denotable value can be a function, a list, or an atom. For the first time, we encounter a semantic domain defined in terms of itself. By substitution, we see that:

$$\begin{aligned} \textit{Denotable-value} = ((\textit{Denotable-value} \rightarrow \textit{Denotable-value}) + \textit{Denotable-value}^* \\ + \textit{Atom} + \textit{Error})_\perp \end{aligned}$$

A solution to this mathematical puzzle exists, but to examine it now would distract us from the study of environments, so the discussion is saved for Chapter 11. Perhaps you sensed that this

semantic problem would arise when you read the abstract syntax. The syntax allows LAMBDA forms to be arguments to other LAMBDA forms; a LAMBDA form can even receive itself as an argument! It is only natural that the semantic domains have the ability to mimic this self-applicative behavior, so recursive domain definitions result.

E determines the meaning of an expression with the aid of an environment. The meaning of an expression is a denotable value. An atom, list, or even a function can be a legal ''answer.'' The LET expression provides a definition mechanism for augmenting the environment and resembles the declaration construct in Figure 7.2. Again, static scoping is used. Functions are created by the LAMBDA construction. A function body is evaluated in the context that is active at the point of function definition, augmented by the binding of an actual parameter to the binding identifier. This definition is also statically scoped.

7.2.1 Scoping Rules

The applicative language uses static scoping; that is, the context of a phrase is determined by its physical position in the program. Consider this sample program (let a_0 and a_1 be sample atomic symbols):

```
LET F = a0 IN
  LET F = LAMBDA (Z) F CONS Z  IN
    LET Z = a1 IN
      F(Z CONS NIL)
```

The occurrence of the first F in the body of the function bound to the second F refers to the atom a_0— the function is not recursive. The meaning of the entire expression is the same as (LAMBDA (Z) a_0 CONS Z) (a_1 CONS NIL)'s, which equals (a_0 CONS (a_1 CONS NIL))'s. Figure 7.6 contains the derivation.

An alternative to static scoping is *dynamic scoping*, where the context of a phrase is determined by the place(s) in the program where the phrase's value is required. The most general form of dynamic scoping is macro definition and invocation. A definition LET I=E binds identifier I to the *text* E; E is not assigned a context until its value is needed. When I's value is required, the context where I appears is used to acquire the text that is bound to it. I is replaced by the text, and the text is evaluated in the existing context. Here is a small example (the => denotes an evaluation step):

```
  LET X = a0 IN
    LET Y = X CONS NIL IN
      LET X = X CONS Y IN  Y

=> (X is bound to a0)
    LET Y = X CONS NIL IN
      LET X = X CONS Y IN  Y
```

Figure 7.6

Let E_0 = LET F = a_0 IN E_1
E_1 = LET F = LAMBDA (Z) F CONS Z IN E_2
E_2 = LET Z = a_1 IN F(Z CONS NIL)

$\mathbf{E}[\![E_0]\!]e_0$

$\mathbf{E}[\![E_1]\!](\mathit{updateenv}[\![F]\!]\ (\mathbf{E}[\![a_0]\!]e_0)\ e_0)$

e_1

$\mathbf{E}[\![E_2]\!](\mathit{updateenv}[\![F]\!]\ (\mathbf{E}[\![\text{LAMBDA (Z) F CONS Z}]\!]e_1)\ e_1)$

e_2

$\mathbf{E}[\![\text{F(Z CONS NIL)}]\!](\mathit{updateenv}[\![Z]\!]\ (\mathbf{E}[\![a_1]\!]e_2)\ e_2)$

e_3

let x = $(\mathbf{E}[\![F]\!]e_3)$ in cases x of $\cdots$ end

$\mathbf{E}[\![\text{LAMBDA (Z) F CONS Z}]\!]e_1$
= in*Function*($\lambda d.\ \mathbf{E}[\![\text{F CONS Z}]\!](\mathit{updateenv}[\![Z]\!]\ d\ e_1)$)

$(\lambda d.\ \cdots)(\mathbf{E}[\![\text{Z CONS NIL}]\!]e_3)$

$\mathbf{E}[\![\text{F CONS Z}]\!](\mathit{updateenv}[\![Z]\!]\ (\mathbf{E}[\![\text{Z CONS NIL}]\!]e_3)\ e_1)$

e_4

let x = $(\mathbf{E}[\![Z]\!]e_4)$ in cases x of $\cdots$ end

$(\mathit{accessenv}[\![Z]\!]\ e_4)$ = $\mathbf{E}[\![\text{Z CONS NIL}]\!]e_3$ = in*List*(in*Atom*(a_1) *cons nil*)

in*List*(($\mathbf{E}[\![F]\!]e_4$) *cons* (in*Atom*(a_1) *cons nil*))

$(\mathit{accessenv}[\![F]\!]\ e_4) = (e_4[\![F]\!]) = (e_1[\![F]\!])$
= $(\mathbf{E}[\![a_0]\!]e_0)$ = in*Atom*(a_0)

in*List*(in*Atom*(a_0) *cons* (in*Atom*(a_1) *cons nil*))

=> (X is bound to a_0)
(Y is bound to X CONS NIL)
LET X = X CONS Y IN Y

=> (X is bound to a_0)
(Y is bound to X CONS NIL)
(X is bound to X CONS Y)
Y

=> (X is bound to a_0)
(Y is bound to X CONS NIL)
(X is bound to X CONS Y)
X CONS NIL

=> (X is bound to a_0)
(Y is bound to X CONS NIL)
(X is bound to X CONS Y)
(X CONS Y) CONS NIL

=> (X is bound to a_0)
(Y is bound to X CONS NIL)
(X is bound to X CONS Y)
(X CONS (X CONS NIL)) CONS NIL

=> ...

and the evaluation unfolds forever.

This form of dynamic scoping can lead to evaluations that are counterintuitive. The version of dynamic scoping found in LISP limits dynamic scoping just to LAMBDA forms. The semantics of ⟦LAMBDA (I) E⟧ shows that the construct is evaluated within the context of its application to an argument (and not within the context of its definition).

We use the new domain:

$$Function = Environment \to Denotable\text{-}value \to Denotable\text{-}value$$

and the equations:

$$\mathbf{E}[\![\text{LAMBDA (I) E}]\!] = \lambda e.\ \text{in}Function(\lambda e_1.\lambda d.\ \mathbf{E}[\![\text{E}]\!](updateenv[\![\text{I}]\!]\ d\ e_1))$$

$$\mathbf{E}[\![\text{E}_1\ \text{E}_2]\!] = \lambda e.\ \text{let } x = (\mathbf{E}[\![\text{E}_1]\!]e) \text{ in cases } x \text{ of}$$
$$\text{is}Function(f) \to (f\,e\,(\mathbf{E}[\![\text{E}_2]\!]e))$$
$$[]\ \text{is}List(t) \to \text{in}Error()$$
$$[]\ \text{is}Atom(a) \to \text{in}Error()$$
$$[]\ \text{is}Error() \to \text{in}Error()\ \text{ end}$$

The example in Figure 7.6 is redone using dynamic scoping in Figure 7.7.

The differences between the two forms of scoping become apparent from the point where **E**⟦F(Z CONS NIL)⟧ is evaluated with environment e_3. The body bound to ⟦F⟧ evaluates with environment e_3 and not e_1. A reference to ⟦F⟧ in the body stands for the function bound to the second ⟦F⟧ and not the atom bound to the first.

Since the context in which a phrase is evaluated is not associated with the phrase's textual position in a program, dynamically scoped programs can be difficult to understand. The inclusion of dynamic scoping in LISP is partly an historical accident. Newer applicative languages, such as ML, HOPE, and Scheme, use static scoping.

7.2.2 Self-Application

The typeless character of the applicative language allows us to create programs that have unusual evaluations. In particular, a LAMBDA expression can accept itself as an argument. Here is an example: LET X= LAMBDA (X) (X X) IN (X X). This program does nothing more than apply the LAMBDA form bound to X to itself. For the semantics of Figure 7.5 and a hypothetical environment e_0:

$$\begin{aligned}
&\mathbf{E}[\![\text{LET X = LAMBDA (X) (X X) IN (X X)}]\!]e_0\\
&= \mathbf{E}[\![\text{(X X)}]\!]e_1, \text{ where } e_1 = (\mathit{updateenv}[\![\text{X}]\!]\ (\mathbf{E}[\![\text{LAMBDA (X) (X X)}]\!]e_0)\ e_0)\\
&= \mathbf{E}[\![\text{LAMBDA (X) (X X)}]\!]e_0\ (\mathbf{E}[\![\text{X}]\!]e_1)\\
&= (\lambda d.\ \mathbf{E}[\![\text{(X X)}]\!](\mathit{updateenv}[\![\text{X}]\!]\ d\ e_0))(\mathbf{E}[\![\text{X}]\!]e_1)\\
&= \mathbf{E}[\![\text{(X X)}]\!](\mathit{updateenv}[\![\text{X}]\!]\ (\mathbf{E}[\![\text{LAMBDA (X) (X X)}]\!]e_0)\ e_0)\\
&= \mathbf{E}[\![\text{(X X)}]\!]e_1
\end{aligned}$$

The simplification led to an expression that is identical to the one that we had four lines earlier. Further simplification leads back to the same expression over and over.

A couple of lessons are learned from this example. First, simplification on semantic expressions is not guaranteed to lead to a constant that is ''the answer'' or ''true meaning'' of the original program. The above program has *some* meaning in the *Denotable-value* domain, but the meaning is unclear. The example points out once more that we are using a *notation* for representing meanings and the notation has shortcomings. These shortcomings are not peculiar to this particular notation but exist in some inherent way in all such notations for representing functions. The study of these limitations belongs to computability theory.

The second important point is that a circular derivation was produced without a recursive definition. The operational properties of a recursive definition $f = \alpha(f)$ are simulated by defining a function $h(g) = \alpha(g(g))$ and letting $f = h(h)$. A good example of this trick is a version of the factorial function:

$$\begin{aligned}
&f(p) = \lambda x.\ \mathit{if}\ x{=}0\ \mathit{then}\ 1\ \mathit{else}\ x * ((p\,(p))(x-1))\\
&\mathit{fac} = f(f)
\end{aligned}$$

The recursiveness in the applicative language stems from the recursive nature of the *Denotable-value* domain. Its elements are in some fundamental way ''recursive'' or

Figure 7.7

$\mathbf{E}[\![\mathrm{E}_0]\!]e_0$

$\cdots$

$(\mathbf{E}[\![\mathrm{F(Z\ CONS\ NIL)}]\!]e_3)$

where $e_3 = (\mathit{updateenv}[\![\mathrm{Z}]\!]\ (\mathbf{E}[\![\mathrm{a}_1]\!]e_2)\ e_2)$
$e_2 = (\mathit{updateenv}[\![\mathrm{F}]\!]\ (\mathbf{E}[\![\mathrm{LAMBDA\ (Z)\ F\ CONS\ Z}]\!]e_1)\ e_1)$
$e_1 = (\mathit{updateenv}[\![\mathrm{F}]\!]\ (\mathbf{E}[\![\mathrm{a}_0]\!]e_0)\ e_0)$.

let $x = (\mathbf{E}[\![\mathrm{F}]\!]e_3)$ in cases x of $\cdots$ end

$\mathit{accessenv}[\![\mathrm{F}]\!]\ e_3$

$\mathbf{E}[\![\mathrm{LAMBDA\ (Z)\ F\ CONS\ Z}]\!]e_1$

$\mathrm{in}\mathit{Function}(\lambda e_\iota.\lambda d.\ \mathbf{E}[\![\mathrm{F\ CONS\ Z}]\!]\ (\mathit{updateenv}[\![\mathrm{Z}]\!]\ d\ e_\iota))$

$(\lambda e_\iota.\lambda d.\ \cdots)\ e_3\ (\mathbf{E}[\![\mathrm{Z\ CONS\ NIL}]\!]e_3)$

$\mathbf{E}[\![\mathrm{F\ CONS\ Z}]\!](\mathit{updateenv}[\![\mathrm{Z}]\!]\ (\mathbf{E}[\![\mathrm{Z\ CONS\ NIL}]\!]e_3)\ e_3)$

e_4

let $x = (\mathbf{E}[\![\mathrm{Z}]\!]e_4)$ in cases x of $\cdots$ end

$\mathit{accessenv}[\![\mathrm{Z}]\!]\ e_4$
$= \mathbf{E}[\![\mathrm{Z\ CONS\ NIL}]\!]e_3$
$\cdots$
$= \mathrm{in}\mathit{List}(\mathrm{in}\mathit{Atom}(\mathrm{a}_1)\ \mathit{cons}\ \mathit{nil})$

$\mathrm{in}\mathit{List}(\mathbf{E}[\![\mathrm{F}]\!]e_4\ \mathit{cons}\ (\mathrm{in}\mathit{Atom}(\mathrm{a}_1)\ \mathit{cons}\ \mathit{nil}))$

$\mathit{accessenv}[\![\mathrm{F}]\!]\ e_4$
$= \mathbf{E}[\![\mathrm{LAMBDA\ (Z)\ F\ CONS\ Z}]\!]e_1$
$= \mathrm{in}\mathit{Function}(\lambda e_\iota.\lambda d.\ \cdots)$

$\mathrm{in}\mathit{List}(\mathrm{in}\mathit{Function}(\lambda e_\iota.\lambda d.\ \cdots)\ \mathit{cons}\ \mathrm{in}\mathit{Atom}(\mathrm{a}_1)\ \mathit{cons}\ \mathit{nil})$

''infinite.'' The nature of an element in a recursively defined semantic domain is more difficult to understand than the recursive specification of an element in a nonrecursive domain. Chapter 11 examines this question in detail.

Finally, it is conceivable that the meaning of the example program is $\perp$— the simplification certainly suggests that no semantic information is contained in the program. In fact, this is the case, but to *prove* it so is nontrivial. Any such proof needs knowledge of the method used to build the domain that satisfies the recursive domain specification.

7.2.3 Recursive Definitions

Now we add a mechanism for defining recursive LAMBDA forms and give it a simple semantics with the *fix* operation. We add two new clauses to the abstract syntax for expressions:

E ::= . . . | LETREC I = E_1 IN E_2 | IFNULL E_1 THEN E_2 ELSE E_3

The LETREC clause differs from the LET clause because all occurrences of identifier I in E_1 refer to the I being declared. The IF clause is an expression conditional for lists, which will be used to define useful recursive functions on lists.

First, the semantics of the conditional construct is:

$$\begin{aligned}&\mathbf{E}[\![\text{IFNULL } E_1 \text{ THEN } E_2 \text{ ELSE } E_3]\!] = \lambda e.\ \text{let } x = (\mathbf{E}[\![E_1]\!]e) \text{ in cases } x \text{ of}\\ &\quad \text{is}\mathit{Function}(f) \rightarrow \text{in}\mathit{Error}()\\ &\quad [] \text{ is}\mathit{List}(t) \rightarrow ((\mathit{null}\ t) \rightarrow (\mathbf{E}[\![E_2]\!]e)\ [] \ (\mathbf{E}[\![E_3]\!]e))\\ &\quad [] \text{ is}\mathit{Atom}(a) \rightarrow \text{in}\mathit{Error}()\\ &\quad [] \text{ is}\mathit{Error}() \rightarrow \text{in}\mathit{Error}()\ \text{ end}\end{aligned}$$

The semantics of the LETREC expression requires a recursively defined *environment*:

$$\begin{aligned}&\mathbf{E}[\![\text{LETREC I} = E_1 \text{ IN } E_2]\!] = \lambda e.\ \mathbf{E}[\![E_2]\!]e'\\ &\quad \text{where } e' = \mathit{updateenv}[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!]e')\ e\end{aligned}$$

$\mathbf{E}[\![E_2]\!]$ requires an environment that maps $[\![\text{I}]\!]$ to $\mathbf{E}[\![E_1]\!]e'$ for some e'. But to support recursive invocations, e' must contain the mapping of $[\![\text{I}]\!]$ to $\mathbf{E}[\![E_1]\!]e'$ as well. Hence e' is defined in terms of itself. This situation is formally resolved with least fixed point semantics. We write:

$$\mathbf{E}[\![\text{LETREC I} = E_1 \text{ IN } E_2]\!] = \lambda e.\ \mathbf{E}[\![E_2]\!](\mathit{fix}\ (\lambda e'.\ \mathit{updateenv}[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!]e')\ e))$$

The functional $G = (\lambda e'.\ \mathit{updateenv}[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!]e')\ e)$: *Environment* $\rightarrow$ *Environment* generates the family of subfunctions approximating the recursive environment. This family is:

$$\begin{aligned}&G^0 = \lambda i.\ \perp\\ &G^1 = \mathit{updateenv}[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!](G^0))\ e\\ &\quad = \mathit{updateenv}[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!]\ (\lambda i.\ \perp))\ e\\ &G^2 = \mathit{updateenv}[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!](G^1))\ e\\ &\quad = \mathit{updateenv}[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!]\ (\mathit{updateenv}[\![\text{I}]\!]\ (\mathbf{E}[\![E_1]\!](\lambda i.\ \perp))\ e))\ e\end{aligned}$$

. . .

G^{i+1} = *updateenv* (**E**[[E_1]](G^i)) *e*

Each subenvironment G^{i+1} produces a better-defined meaning for [[I]] than its predecessor G^i and acts like *e* otherwise. A subenvironment G^{i+1} is able to handle *i* recursive references to [[I]] in [[E_1]] before becoming exhausted and producing $\perp$. The limit of the chain of subenvironments is an environment that can handle an unlimited number of recursive references. Rest assured that you don't need to remember all these details to define and use recursive environments. We give the details to demonstrate that fixed point theory has intuitive and practical applications.

Now consider this example:

LETREC F = LAMBDA (X) IFNULL X THEN NIL ELSE a_0 CONS F(TAIL X)
IN F(a_1 CONS a_2 CONS NIL)

Function F transforms a list argument into a list of the same length containing only a_0 atoms. The value of the above expression is the same as (a_0 CONS a_0 CONS NIL). Figure 7.8 shows the simplification. References to [[F]] in E_1 and E_2 are resolved by the recursive environment.

Now that we have seen several examples, comments about the LET construct are in order. The purpose of LET is similar to that of the constant declaration construct in Figure 7.2. In fact, **E**[[LET I = E_1 IN E_2]] equals **E**[[[E_1/I] E_2]], where [[[E_1/I] E_2]] denotes the physical substitution of expression [[E_1]] for all free occurrences of [[I]] in [[E_2]], with renaming of the identifiers in [[E_2]] as needed. (The proof of this claim is left as an exercise.) As an example:

LET X = a_0 IN
LET Y = X CONS NIL IN
(HEAD Y) CONS X CONS NIL

rewrites to the simpler program:

LET Y = a_0 CONS NIL IN
(HEAD Y) CONS a_0 CONS NIL

which rewrites to:

(HEAD (a_0 CONS NIL)) CONS a_0 CONS NIL

which rewrites to:

a_0 CONS a_0 CONS NIL

These rewriting steps preserve the semantics of the original program. The rewritings are a form of computing, just like the computing done on an arithmetic expression.

The LETREC construct also possesses a substitution principle: for [[LETREC I = E_1 IN E_2]], all free occurrences of [[I]] in [[E_2]] are replaced by [[E_1]], *and* to complete the substitution, any free occurrences of [[I]] in the resulting expression are also replaced (until they are completely eliminated). Of course, the number of substitutions is unbounded: LETREC I= α(I) IN β(I) writes to $\beta(\alpha(I))$, then to $\beta(\alpha(\alpha(I)))$, then to $\beta(\alpha(\alpha(\alpha(I))))$ $\cdots$, and so on. This is expected, since the environment that models the recursion uses *fix* to generate a similar chain of semantic values. Complete substitution isn't feasible for producing answers in a finite amount of time, so the substitutions must be perfomed more cautiously: occurrences of

Figure 7.8

Let E_0 = LAMBDA (X) E_1
E_1 = IFNULL X THEN NIL ELSE a_0 CONS F(TAIL X)
E_2 = F(a_1 CONS a_2 CONS NIL).

$\mathbf{E}[\![\text{LETREC F} = E_0 \text{ IN } E_2]\!]e_0$

$\mathbf{E}[\![E_2]\!]e_1$

where $e_1 = fix\, G$
$G = (fix(\lambda e_1.update[\![\text{F}]\!]\ (\mathbf{E}[\![E_0]\!]e_1)\ e_0))$

$\mathbf{E}[\![\text{F}(a_1 \text{ CONS } a_2 \text{ CONS NIL})]\!]e_1$

let $x = (\mathbf{E}[\![\text{F}]\!](fix\, G))$ in cases x of $\cdots$ end

$accessenv[\![\text{F}]\!]\ (fix\, G)$
$= (fix\, G)[\![\text{F}]\!]$
$= G(fix\, G)[\![\text{F}]\!]$
$= (updateenv[\![\text{F}]\!]\ (\mathbf{E}[\![E_0]\!](fix\, G))\ e_0)\ [\![\text{F}]\!]$
$= \mathbf{E}[\![\text{LAMBDA (X) } E_1]\!](fix\, G)$
$= \text{in}Function(\lambda d.\ \mathbf{E}[\![E_1]\!](updateenv[\![\text{X}]\!]\ d\ e_1))$

$\mathbf{E}[\![E_1]\!](updateenv[\![\text{X}]\!]\ (\mathbf{E}[\![a_1 \text{ CONS } a_2 \text{ CONS NIL}]\!]e_1)\ e_1)$

e_2

$\mathbf{E}[\![\text{IFNULL X THEN NIL ELSE } a_0 \text{ CONS F(TAIL X)}]\!]e_2$

let $x = (\mathbf{E}[\![\text{X}]\!]e_2)$ in cases x of $\cdots$ end

$\mathbf{E}[\![a_1 \text{ CONS } a_2 \text{ CONS NIL}]\!]e_1$

$\text{in}List(\text{in}Atom(a_1)\ cons\ \cdots)$

$null(\text{in}Atom(a_1)\ cons\ \cdots) \rightarrow \mathbf{E}[\![\text{NIL}]\!]e_2\ [\!]\ \mathbf{E}[\![a_0 \text{ CONS F(TAIL X)}]\!]e_2$

$\mathbf{E}[\![a_0 \text{ CONS F(TAIL X)}]\!]e_2$

Figure 7.8 (continued)

let $x = (\mathbf{E}[\![\text{F(TAIL X)}]\!]e_2)$ in cases x of $\cdots$ end

let $x = (\mathbf{E}[\![\text{F}]\!]e_2)$ in cases x of $\cdots$ end

$$\begin{aligned} &\mathit{accessenv}[\![\text{F}]\!]\ e_2 = e_1[\![\text{F}]\!] = (\mathit{fix}\,G)[\![\text{F}]\!] \\ &\cdots \\ &= G(\mathit{fix}\,G)[\![\text{F}]\!] = \text{in}\mathit{Function}(\lambda d.\ \mathbf{E}[\![\text{E}_1]\!] \cdots) \end{aligned}$$

$\mathbf{E}[\![\text{E}_1]\!](\mathit{updateenv}[\![\text{X}]\!]\ (\mathbf{E}[\![\text{TAIL X}]\!]e_2)\ e_1)$

$\text{in}\mathit{List}(\text{in}\mathit{Atom}(a_0)\ \mathit{cons}\ \mathit{nil})$

$\text{in}\mathit{List}(\mathbf{E}[\![\text{a}_0]\!]e_2\ \mathit{cons}\ (\text{in}\mathit{Atom}(\text{a}_0)\ \mathit{cons}\ \mathit{nil}))$

$\text{in}\mathit{List}(\text{in}\mathit{Atom}(a_0)\ \mathit{cons}\ \text{in}\mathit{Atom}(a_0)\ \mathit{cons}\ \mathit{nil})$

$[\![\text{I}]\!]$ in $[\![\text{E}_2]\!]$ are replaced by $[\![\text{E}_1]\!]$ only when absolutely needed to complete the simplification of $[\![\text{E}_2]\!]$. This strategy matches the conventional approach for evaluating calls of recursively defined functions.

These examples suggest that computation upon applicative programs is just substitution. Since the environment is tied to the semantics of substitution, it is directly involved in the execution and is a run-time structure in an implementation of the applicative language. The pre-execution analysis seen in Section 7.1 will not eliminate occurrences of environment arguments in the denotations of applicative programs. Like the *Store* algebra of Figure 7.1, these expressions are ''frozen'' until run-time.

7.3 COMPOUND DATA STRUCTURES

Both imperative and applicative languages use compound data structures— values that can be structurally decomposed into other values. The applicative language used lists, which are compound structures built with CONS and NIL and decomposed with HEAD and TAIL. Another favorite compound structure for applicative languages is the tuple, which is built with a tupling constructor and decomposed with indexing operations. The semantics of these objects is straightforward: finite lists of A elements belong to the A^* domain, and tuples of A, B, C, . . . elements are members of the product space $A \times B \times C \times \cdots$.

The problems with modelling compound structures increase with imperative languages, as variable forms of the objects exist, and an object's subcomponents can be altered with assignment. For this reason, we devote this section to studying several versions of array

variables.

What is an array? We say that it is a collection of homogeneous objects indexed by a set of scalar values. By *homogeneous*, we mean that all of the components have the same structure. This is not absolutely necessary; languages such as SNOBOL4 allow array elements' structures to differ. But homogeneity makes the allocation of storage and type-checking easier to perform. Consequently, compiler-oriented languages insist on homogeneous arrays so that these tasks can be performed by the compiler. The disassembly operation on arrays is indexing. The indexing operation takes an array and a value from the index set as arguments. The index set is *scalar,* that is, a primitive domain with relational and arithmetic-like operations, so that arithmetic-like expressions represent index values for the indexing operation. Normally, the index set is restricted by lower and upper bounds.

The first version of array that we study is a linear vector of values. Let some primitive domain *Index* be the index set; assume that it has associated relational operations *lessthan, greaterthan,* and *equals.* The array domain is:

$$1DArray = (Index \rightarrow Location) \times Lower\text{-}bound \times Upper\text{-}bound$$
$$\text{where } Lower\text{-}bound = Upper\text{-}bound = Index$$

The first component of an array maps indexes to the locations that contain the storable values associated with the array. The second and third components are the lower and upper bounds allowed on indexes to the array. You are left with the exercise of defining the indexing operation.

The situation becomes more interesting when multidimensional arrays are admitted. Languages such as ALGOL60 allow arrays to contain other arrays as components. For example, a three-dimensional array is a vector whose components are two-dimensional arrays. The hierarchy of multidimensional arrays is defined as an infinite sum. For simplicity, assume the index set for each dimension of indexing is the same domain *Index.* We define:

$$1DArray = (Index \rightarrow Location) \times Index \times Index$$

and for each $n \geq 1$:

$$(n+1)DArray = (Index \rightarrow nDArray) \times Index \times Index$$

so that the domain of multidimensional arrays is:

$$a \in MDArray = \sum_{m=1}^{\infty} mDArray$$
$$= ((Index \rightarrow Location) \times Index \times Index)$$
$$+ ((Index \rightarrow ((Index \rightarrow Location) \times Index \times Index)) \times Index \times Index)$$
$$+ ((Index \rightarrow ((Index \rightarrow ((Index \rightarrow Location) \times Index \times Index)) \times Index$$
$$\times Index)) \times Index \times Index)$$
$$+ \cdots$$

The definition says that a one-dimensional array maps indexes to locations, a two-dimensional array maps indexes to one-dimensional arrays, and so on. Any $a \in MDArray$ has the form in $kDArray\,(map, lower, upper)$ for some $k \geq 1$, saying that a is a k-dimensional array.

The indexing operation for multidimensional arrays is:

$$\textit{access-array} : \textit{Index} \to \textit{MDArray} \to (\textit{Location} + \textit{MDArray} + \textit{Errvalue})$$
$$\textit{access-array} = \lambda i.\lambda r.\ \text{cases } r \text{ of}$$
$$\quad \text{is}\textit{1DArray}(a) \to \textit{index}_1\ a\ i$$
$$\quad []\ \text{is}\textit{2DArray}(a) \to \textit{index}_2\ a\ i$$
$$\quad \cdots$$
$$\quad []\ \text{is}\textit{kDArray}(a) \to \textit{index}_k\ a\ i$$
$$\quad \cdots \text{ end}$$

where, for all $m \geq 1$, $\textit{index}_m$ abbreviates the expression:

$$\lambda(\textit{map}, \textit{lower}, \textit{upper}).\lambda i.\ (i\ \textit{lessthan}\ \textit{lower})$$
$$\quad \textit{or}\ (i\ \textit{greaterthan}\ \textit{upper}) \to\ \text{in}\textit{Errvalue}()\ []\ \textit{mInject}(\textit{map}\,(i))$$

where *mInject* abbreviates the expressions:

$$\textit{1Inject} = \lambda l.\ \text{in}\textit{Location}(l)$$
$$\cdots$$
$$\textit{(n+1)Inject} = \lambda a.\ \text{in}\textit{MDArray}(\ \text{in}\textit{nDArray}(a))$$

The *access-array* operation is represented by an infinite function expression. But, by using the pair representation of disjoint union elements, the operation is convertible to a finite, computable format. The operation performs a one-level indexing upon an array *a*, returning another array if *a* has more than one dimension. We define no *update-array* operation; the store-based operation *update* is used in combination with the *access-array* operation to complete an assignment to a location in an array.

Unfortunately, the straightforward model just seen is clumsy to use in practice, as realistic programming languages allow arrays to be built from a variety of components, such as numbers, record structures, sets, and so on. A nested array could be an array of records containing arrays. Here is a Pascal-like syntax for such a system of data type declarations:

T∈ Type-structure
S∈ Subscript
T ::= **nat** | **bool** | **array** [N_1..N_2] **of** T | **record** D **end**
D ::= D_1;D_2 | **var** I:T
C ::= . . . | I[S]:=E | . . .
E ::= . . . | I[S] | . . .
S ::= E | E,S

We provide a semantics for this type system. First, we expand the *Denotable-value* domain to read:

$$\textit{Denotable-value} = (\textit{Natlocn} + \textit{Boollocn} + \textit{Array} + \textit{Record} + \textit{Errvalue})_\perp$$
$$\quad \text{where } l \in \textit{Natlocn} = \textit{Boollocn} = \textit{Location}$$
$$\quad\quad a \in \textit{Array} = (\textit{Nat} \to \textit{Denotable-value}) \times \textit{Nat} \times \textit{Nat}$$
$$\quad\quad r \in \textit{Record} = \textit{Environment} = \textit{Id} \to \textit{Denotable-value}$$

Each component in the domain corresponds to a type structure. The recursiveness in the syntax definition motivates the recursiveness of the semantic domain.

The valuation function for type structures maps a type structure expression to storage allocation actions. The *Store* algebra of Figure 7.4 is used with the *Poststore* algebra of Figure 7.1 in the equations that follow.

T: Type-structure $\rightarrow$ *Store* $\rightarrow$ (*Denotable-value* $\times$ *Poststore*)

$\mathbf{T}[\![\textbf{nat}]\!] = \lambda s.$ let $(l,p) = (\textit{allocate-locn}\ s)$ in (in*Natlocn*(l), p)

$\mathbf{T}[\![\textbf{bool}]\!] = \lambda s.$ let $(l,p) = (\textit{allocate-locn}\ s)$ in (in*Boollocn*(l), p)

$\mathbf{T}[\![\textbf{array}\ [\mathrm{N}_1..\mathrm{N}_2]\ \textbf{of}\ \mathrm{T}]\!] = \lambda s.$ let $n_1 = \mathbf{N}[\![\mathrm{N}_1]\!]$ in let $n_2 = \mathbf{N}[\![\mathrm{N}_2]\!]$
in n_1 *greaterthan* $n_2 \rightarrow$ (in*Errvalue*(), (*signalerr* s))
[] *get-storage* n_1 (*empty-array* n_1 n_2) s

where

get-storage : *Nat* $\rightarrow$ *Array* $\rightarrow$ *Store* $\rightarrow$ (*Denotable-value* $\times$ *Poststore*)
get-storage $= \lambda n.\lambda a.\lambda s.$ n *greaterthan* $n_2 \rightarrow$ (in*Array*(a), *return* s)
[] let $(d, p) = \mathbf{T}[\![\mathrm{T}]\!]s$
in (*check*(*get-storage* (*n plus one*) (*augment-array n d a*)))(p)

and

augment-array : *Nat* $\rightarrow$ *Denotable-value* $\rightarrow$ *Array* $\rightarrow$ *Array*
augment-array $= \lambda n.\lambda d.\lambda(map, lower, upper).\ ([n \mapsto d]map, lower, upper)$
empty-array : *Nat* $\rightarrow$ *Nat* $\rightarrow$ *Array*
empty-array $= \lambda n_1.\lambda n_2.((\lambda n.$ in*Errvalue*$()), n_1, n_2)$

$\mathbf{T}[\![\textbf{record}\ \mathrm{D}\ \textbf{end}]\!] = \lambda s.$ let $(e, p) = (\mathbf{D}[\![\mathrm{D}]\!]\ \textit{emptyenv}\ s)$ in (in*Record*(e), p)

The heart of the strategy for creating an array value is *get-storage*, which iterates from the lower bound of the array to the upper bound, allocating the proper amount of storage for a component at each iteration. The component is inserted into the array by the *augment-array* operation.

A declaration activates the storage allocation strategy specificed by its type structure:

D: Declaration $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ (*Environment* $\times$ *Poststore*)
$\mathbf{D}[\![\mathrm{D}_1;\mathrm{D}_2]\!] = \lambda e.\lambda s.$ let $(e', p) = (\mathbf{D}[\![\mathrm{D}_1]\!]e\ s)$ in $(\textit{check}(\mathbf{D}[\![\mathrm{D}_2]\!]e'))(p)$
$\mathbf{D}[\![\textbf{var}\ \mathrm{I}{:}\mathrm{T}]\!] = \lambda e.\lambda s.$ let $(d, p) = \mathbf{T}[\![\mathrm{T}]\!]s$ in $((\textit{updateenv}[\![\mathrm{I}]\!]\ d\ e), p)$

Now assume that the operation *access-array* : *Nat* $\rightarrow$ *Array* $\rightarrow$ *Denotable-value* has been defined. (This is left as an easy exercise.) Array indexing is defined in the semantic equations for subscripts:

S: Subscript $\rightarrow$ *Array* $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Denotable-value*
$\mathbf{S}[\![\mathrm{E}]\!] = \lambda a.\lambda e.\lambda s.$ cases $(\mathbf{E}[\![\mathrm{E}]\!]e s)$ of
. . .

```
                [] is Nat(n) → access-array n a
                ··· end
S[[E, S]] = λa.λe.λs. cases (E[[E]]e s) of
                ···
                [] is Nat(n) → (cases (access-array n a) of
                                    ···
                                    [] is Array(a') → S[[S]]a' e s
                                    ··· end)
                ··· end
```

A version of first order array assignment is:

```
C[[I[S]:=E]] = λe.λs. cases (accessenv[[I]] e) of
                ···
                [] is Array(a) → (cases (S[[S]]a e s) of
                                    ···
                                    [] is Natlocn(l) → (cases (E[[E]]e s) of
                                          ···
                                          [] is Nat(n) → return (update l in Nat(n) s)
                                          ··· end)
                                    ··· end)
                ··· end
```

As usual, a large amount of type-checking is required to complete the assignment, and an extra check ensures that the assignment is first order; that is, the left-hand side [[I[S]]] denotes a location and not an array.

The final variant of array assignment we examine is the most general. The array is heterogeneous (its components can be elements of different structures), its dimensions and index ranges can change during execution, and by using a recursive definition, it can possess itself as an element. Variants on this style of array are found in late binding languages such as APL, SNOBOL4, and TEMPO, where pre-execution analysis of arrays yields little.

The domain of heterogeneous arrays is the domain *Array* just defined in the previous example, but the operations upon the domain are relaxed to allow more freedom. Since an array is a denotable value, the usual methods for accessing and updating a heterogeneous array are generalized to methods for handling all kinds of denotable values. A first order denotable value is just a degenerate array. The *access-value* operation fetches a component of a denotable value. It receives as its first argument a list of indexes that indicates the path taken to find the component.

```
access-value : Nat* → Denotable-value → Denotable-value
access-value = λnlist.λd.
    null nlist → d
    [] (cases d of
```

```
    isNatlocn(l)→ inErrvalue()
    ...
    [] isArray(map, lower, upper)→
          let n = hd nlist in
          (n lessthan lower) or (n greaterthan upper) → inErrvalue()
                [] (access-value (tl nlist) (map n))
    ... end)
```

The operation searches through the structure of its denotable value argument until the component is found. An empty index list signifies that the search has ended. A nonempty list means that the search can continue if the value is an array. If so, the array is indexed at position (*hd nlist*) and the search continues on the indexed component.

The updating operation follows a similar strategy, but care must be taken to preserve the outer structure of an array while the search continues within its subparts. The argument *new-value* is inserted into the array *current-value* at index *nlist:*

```
update-value : Nat* → Denotable-value → Denotable-value → Denotable-value
update-value = λnlist.λnew-value λcurrent-value .
    null nlist → new-value
    [] (cases current-value of
        isNatlocn(l)→ inErrvalue()
        ...
        [] isArray(map, lower, upper)→
            let n = hd nlist in
            let new-lower = (n lessthan lower → n [] lower) in
            let new-upper = (n greaterthan upper → n [] upper)
            in augment-array n (update-value (tl nlist) new-value (map n))
                              (map, new-lower, new-upper)
        ...
        [] isErrvalue() → augment-array n (update-value (tl nlist) new-value
                              inErrvalue()) (empty-array n n)
      ... end)
where
augment-array : Nat → Denotable-value → Array → Denotable-value
augment-array = λn.λd.λ(map, lower, upper).inArray([i ↦ d]map, lower, upper)
```

If an index list causes a search deeper into an array structure than what exists, the ''is*Errvalue*() → ...'' clause creates another dimension to accommodate the index list. Thus an array can grow extra dimensions. If an index from the index list falls outside of an array's bounds, the ''is*Array*(...) → ...'' clause expands the arrays's bounds to accommodate the index. Thus an array can change its bounds. The outer structure of a searched array is preserved by the *augment-array* operation, which inserts the altered component back into the

structure of the indexed array. Note that *update-value* builds a new denotable value; it does not alter the store. A separate dereferencing step is necessary to cause conventional assignment.

The exercises continue the treatment of this and other kinds of array.

SUGGESTED READINGS

Semantics of block structure: Henhapl & Jones 1982; Landin 1965; Meyer 1983; Mosses 1974; Oles 1985; Reynolds 1981; Strachey 1968

Semantics of applicative languages: Abelson & Sussman 1985; Gordon 1973, 1975; Muchnick & Pleban 1982; Reynolds 1970; Steele & Sussman 1978

Semantics of compound data structures: Abelson & Sussman 1985; Andrews & Henhapl 1982; Gordon 1979; Jones & Muchnick 1978; Tennent 1977

EXERCISES

1. a. Let the domain *Location* be *Nat.* (Thus, *first-locn* = *zero*, *next-locn* = (λ*l*. *l plus one*), etc.) Using the strategy of *not* simplifying away occurrences of *access, update, check,* or *return,* simplify **P**⟦**begin var** A; A:=2; **begin var** B; B:=A+1 **end end**.⟧ as far as possible.
   b. Let the result of part a be called *Object-code.* Do *one* step of simplification to the expression *Object-code*(*zero*).
   c. Let the result of part b be called *Loaded-object-code.* Simplify the expression *Loaded-object-code*(*newstore*) to a post-store value.

2. Extend the language in Figure 7.2 to include declarations of variables of Boolean type; that is:

   D ::= ··· | **bool var** I

   and expressions of boolean type:

   E ::= ··· | **true** | ¬ E

   Adjust the semantic algebras and the semantic equations to accommodate the extensions.

3. Augment the language of Figure 7.2 to include procedures:

   D ::= ··· | **proc** I=C

   and procedure invocations:

   C ::= ··· | **call** I

   Now augment the *Denotable-value* domain with the summand $Proc = Store \to Poststore_\perp$ to accommodate procedures.

   a. If the semantics of procedure definition is written **D**⟦**proc** I=C⟧ = λ*e*. (*updateenv* ⟦I⟧ in*Proc*(**C**⟦C⟧*e*) *e*), write the semantic equation for **C**⟦**call** I⟧.

What kind of scoping is used?

b. Say that the domain *Proc* is changed to be $Proc = Environment \rightarrow Store \rightarrow Post\text{-}store_{\perp}$. Write the semantic equations for procedure definition and invocation. What kind of scoping is used?

4. a. For the semantics of Figure 7.2, show that there exist identifiers I and J and a command C such that **B**[[**begin var** I; **var** J; C **end**]] ≠ **B**[[**begin var** J; **var** I; C **end**]].
   b. Revise the language's semantics so that for all identifiers I and J and command C, the above inequality becomes an equality. Using structural induction, prove this.
   c. Does the semantics you defined in part b support the equality **B**[[**begin var** I; I:=I **end**]] = *return*?

5. a. Define the semantics of the ALGOL60 **for**-loop.
   b. Define the semantics of the Pascal **for**-loop. (Recall that the loop index may not be altered within the loop's body.)

6. It is well known that the environment object in Figure 7.2 can be implemented as a single global stack. Where is the stack concept found in the semantic equations?

7. The semantics in Figure 7.2 is somewhat simple minded in that the block [[**begin var** A; **const** A=0; C **end**]] has a nonerroneous denotation.
   a. What is [[A]]'s denotation in [[C]]?
   b. Adjust the semantics of declarations so that redeclaration of identifiers in a block produces an error denotation for the block.

8. Use structural induction to prove that the semantics in Figure 7.2 is constructed so that if any command in a program maps a store to an erroneous post-store, then the denotation of the entire program is exactly that erroneous post-store.

9. a. Add to the language of Figure 7.2 the declaration [[**var** I:=E]]. What problems arise in integrating the new construct into the existing valuation function **D** for declarations?
   b. Attempt to handle the problems noted in part a by using a new declaration valuation function:

$$\mathbf{D} : \text{Declaration} \rightarrow Environment \rightarrow (Store \rightarrow Poststore)$$
$$\rightarrow (Environment \times (Store \rightarrow Poststore))$$
$$\mathbf{B}[\![\textbf{begin}\ \text{D};\ \text{C}\ \textbf{end}]\!] = \lambda e.\ \text{let}\ (e', c) = \mathbf{D}[\![\text{D}]\!]e\ return\ \text{in}\ ((check\,\mathbf{C}[\![\text{C}]\!]e') \circ c)$$

Write the semantic equations for $\mathbf{D}[\![\text{D}_1;\text{D}_2]\!]$ and **D**[[**var** I:=E]] in the new format.

10. Make the needed adjustments so that the stack-based store model of Figure 7.4 can be used with the semantics of Figure 7.2.

11. A design deficiency of the language in Figure 7.2 is its delayed reporting of errors. For example, a denotable value error occurs in the assignment [[B:=A+1]] when [[A]] is not

previously declared. The error is only reported when the run-time store is mapped to an erroneous post-store. The error reporting need not be delayed until run-time: consider the valuation function **C** : Command → *Environment* → *Compiled-code*, where *Compiled-code* = (*Store* → *Poststore*) + *Error-message*. Rewrite the semantics of Figure 7.2 using the new form of **C** so that an expressible value or denotable value error in a command leads to an *Error-message* denotation.

12. Extend the language of Figure 7.2 with pointers. In particular, set

    Denotable-value = *Natlocn* + *Ptrlocn* + *Nat* + *Errvalue*
    where *Natlocn* = *Location* (locations that hold numbers)
    Ptrlocn = *Location* (locations that hold pointers)
    Errvalue = *Unit*.

    Augment the syntax of the language and give the semantics of pointer declaration, dereferencing, assignment, and dynamic storage allocation. How does the integration of pointers into the language change the stack-based storage model?

13. Augment the file editor language of Figure 5.4 with environments by introducing the notion of *window* into the editor. A user of the file editor can move from one window to another and be able to manipulate more than one file concurrently during a session.

14. Give an example of a programming language whose notion of R-value for an identifier is not a function of the identifier's L-value.

15. Using the semantic definition of Figure 7.5, determine the denotations of the following expressions:

    a. ⟦LET N = a_0 IN LET N = N CONS NIL IN TAIL N⟧
    b. ⟦LET G = LAMBDA (X) X IN LET G = LAMBDA (Y) (G Y) IN (G a_0)⟧
    c. ⟦LET F = LAMBDA(X) (X X) IN LET G = (F F) IN a_0⟧

    Redo parts a through c using the LISP-style dynamic scoping semantics of Section 7.2.1.

16. Using structural induction, prove the following claim for the semantics of Figure 7.5: for all I∈ Identifier, E_1, E_2∈ Expression, **E**⟦LET I = E_1 IN E_2⟧ = **E**⟦[E_1/I]E_2⟧.

17. a. Using the semantics of the LETREC construct in Section 7.2.3, determine the denotations of the following examples:

        i. ⟦LETREC APPEND = LAMBDA (L1) LAMBDA (L2)
           IFNULL L1 THEN L2 ELSE (HEAD L1) CONS (APPEND
           (TAIL L1) L2) IN APPEND (a_0 CONS NIL) (a_1 CONS NIL)⟧
        ii. ⟦LETREC L = a_0 CONS L IN HEAD L⟧
        iii. ⟦LETREC L = HEAD L IN L⟧

    b. Reformulate the semantics of the language so that a defined denotation for part ii above is produced. Does the new semantics practice ''lazy evaluation''?

18. In LISP, a denotable value may be CONSed to any other denotable value (not just a list), producing a *dotted pair.* For example, ⟦A CONS (LAMBDA (I) I)⟧ is a dotted pair. Reformulate the semantics in Figure 7.5 to allow dotted pairs. Redo the denotations of the programs in exercises 15 and 17.

19. Formulate a semantics for the applicative language of Section 7.2 that uses macro substitution-style dynamic scoping.

20. Define the appropriate construction and destruction constructs for the record structure defined in Section 7.3. Note that the denotation of a record is a ''little environment.'' Why does this make a block construct such as the Pascal **with** statement especially appropriate? Define the semantics of a **with**-like block statement.

21. a. Integrate the domain of one-dimensional arrays *1DArray* into the language of Figure 7.2. Define the corresponding assembly and disassembly operations and show the denotations of several example programs using the arrays.
    b. Repeat part a with the domain of multidimensional arrays *MDArray.*
    c. Repeat part a with the Pascal-like type system and domain of arrays *Array.*

22. After completing Exercise 21, revise your answers to handle arrays whose bounds are set by expressions calculated at runtime, e.g., for part c above use: T ::= ··· | **array** [E_1..E_2] **of** T

23. After completing Exercise 21, adjust the semantics of the assignment statement ⟦I:=E⟧ so that:

    a. If ⟦I⟧ is an array denotable value and (**E**⟦E⟧*e s*) is an expressible value from the same domain as the array's components, then a copy of (**E**⟦E⟧*e s*) is assigned to each of the array's components;
    b. If ⟦I⟧ is an array denotable value and (**E**⟦E⟧*e s*) is an array expressible value of ''equivalent type,'' then the right-hand side value is bound to the left-hand side value.

24. Rewrite the valuation function for type structures so that it uses the *Environment* and *Store* algebras of Figure 7.1; that is, **T**: Type-structure → *Environment* → (*Denotable-value* × *Environment*), and the valuation function for declarations reverts to the **D**: Declaration → *Environment* → *Environment* of Figure 7.2. Which of the two versions of **T** and **D** more closely describes type processing in a Pascal compiler? In a Pascal interpreter? Which version of **T** do you prefer?

25. Consider the domain of one-dimensional arrays; the denotations of the arrays might be placed in the environment (that is, an array is a denotable value) or the store (an array is a storable value). Show the domain algebras and semantic equations for both treatments of one-dimensional arrays. Comment on the advantages and disadvantages of each treatment with respect to understandability and implementability.

26. In FORTRAN, an array is treated as a linear allocation of storage locations; the lower

bound on an array is always *one*. Define the domain of one dimensional FORTRAN arrays to be $Location \times Nat$ (that is, the location of the first element in the array and the upper bound of the array). Show the corresponding operations for allocating storage for an array, indexing, and updating an array.

27. Strachey claimed that the essential characteristics of a language are delineated by its *Denotable-value*, *Expressible-value*, and *Storable-value* domains.

    a. Give examples of programming languages such that:

        i. Every expressible value is storable but is not necessarily denotable; every storable value is expressible but is not necessarily denotable; a denotable value is not necessarily expressible or storable.
        ii. Every denotable value is expressible and storable; every storable value is expressible and denotable; an expressible value is not necessarily denotable or storable.
        iii. Every denotable value is expressible and storable; every expressible value is denotable and storable; every storable value is denotable and expressible.

    b. Repeat part a with $Expressible\text{-}value = (Nat + Tr + Location + Expressible\text{-}value^*)_\perp$; with $Expressible\text{-}value = ((Id \rightarrow Expressible\text{-}value) + Nat)_\perp$.
    c. Pick your favorite general purpose programming language and list its denotable, expressible, and storable value domains. What limitations of the language become immediately obvious from the domains' definitions? What limitations are *not* obvious?

28. Language design is often a process of consolidation, that is, the integration of desirable features from other languages into a new language. Here is a simple example. Say that you wish to integrate the notion of imperative updating, embodied in the language in Figure 7.2, with the notion of value-returning construct, found in the language of Figure 7.5. That is, you desire an imperative language in which every syntactic construct has an associated expressible value (see ALGOL68 or full LISP).

    a. Design such a language and give its denotational semantics. Does the language's syntax look more like the language in Figure 7.2 or 7.5?
    b. Repeat part a so that the new language appears more like the other figure (7.5 or 7.2) than the language in part a did. Comment on how the characteristics of the two languages were influenced by your views of the languages' syntax definitions.

    Attempt this exercise once again by:

    c. Integrating an imperative-style data structure, the array, with an expression-based, applicative notation like that of Figure 7.5.
    d. Integrating an applicative-style list structure with a command-based, imperative notation like that of Figure 7.2.

29. The function notation used for denotational semantics definitions has its limitations, and one of them is its inability to simply express the Pascal-style hierarchy of data-type. If you were asked to define an imperative programming language with simple and

compound data-types, and you knew nothing of the ALGOL/Pascal tradition, what kinds of data-types would you be led to develop if you used denotational semantics as a design tool? What pragmatic advantages and disadvantages do you see in this approach?