Chapter 13
ACTION SEMANTICS

The formal methods discussed in earlier chapters, particularly denotational semantics and structural operational semantics, have been used extensively to provide accurate and unambiguous definitions of programming languages. Unlike informal definitions written in English or some other natural language, formal definitional techniques can serve as a basis for proving properties of programming languages and the correctness of programs. Although most programmers rely on informal specifications of languages, these definitions are often vague, incomplete, and even erroneous. English does not lend itself to precise and unambiguous definitions.

In spite of the arguments for relying on formal specifications of programming languages, programmers generally avoid them when learning, trying to understand, or even implementing a programming language. They find formal definitions notationally dense, cryptic, and unlike the way they view the behavior of programming languages. Furthermore, formal specifications are difficult to create accurately, to modify, and to extend. Formal definitions of large programming languages are overwhelming to both the language designer and the language user, and therefore remain mostly unread.

Programmers understand programming languages in terms of basic concepts such as control flow, bindings, modifications of storage, and parameter passing. Formal specifications often obscure these notions to the point that the reader must invest considerable time to determine whether a language follows static or dynamic scoping and how parameters are actually passed. Sometimes the most fundamental concepts of the programming language are the hardest to understand in a formal definition.

Action semantics, which attempts to answer these criticisms of formal methods for language specification, has been developed over the past few years by Peter Mosses with the collaboration of David Watt. The goal of their efforts has been to produce formal semantic specifications that directly reflect the ordinary computational concepts of programming languages and that are easy to read and understand. In this chapter we present an introduction to the methods of action semantics by specifying three languages: the calculator language from Chapter 9, Wren, and Pelican.

13.1 CONCEPTS AND EXAMPLES

Action semantics has evolved out of the tradition of denotational semantics, where syntactic entities (abstract syntax trees) are mapped compositionally by semantic functions into semantic entities that act as the denotations of the syntactic objects. The chief difference between the two methods of formal specification lies in the nature of the semantic entities. The semantic functions of denotational semantics map syntactic phrases into primitive mathematical values, structured objects, and such higher-order functions as are found in the lambda calculus where functions can be applied to other functions. In contrast, action semantics uses three kinds of first-order entities as denotations: **actions** , **data**, and **yielders** . "First-order" means that actions cannot be applied to other actions.

- The semantic entities known as **actions** incorporate the performance of computational behavior, using values passed to them to generate new values that reflect changes in the state of the computation. Actions are the engines that process data and yielders.
- The **data** entities consist of mathematical values, such as integers, Boolean values, and abstract cells representing memory locations, that embody particles of information. Data are classified into sorts so that the kinds of information processed by actions are well specified in a language definition. Sorts of data are defined by algebraic specifications in the manner discussed in Chapter 12.
- **Yielders** encompass unevaluated pieces of data whose values depend on the current information incorporating the state of the computation. Yielders are entities that, depending on the current storage and environment, can be evaluated to yield data.

We begin our discussion of action semantics by considering the meaning of several simple language constructs from Pelican (see section 9.5), first viewing denotational definitions and then introducing enough action notation to describe the constructs in action semantics. Figure 13.1 displays the semantic equations for a denotational specification of constant and variable declarations and identifier evaluation.

Denotational semantics expresses the details of a semantic equation functionally, so we see many parameters being passed to, and values returned from, the semantic functions explicitly. In contrast, each action in action semantics entails particular modes of control and data flow implicitly. Much of the information processed by an action is manipulated automatically when the action is performed.

```
elaborate [[const I = E]] env sto = (extendEnv(env,I,evaluate E env sto), sto)

elaborate [[var I : T]] env sto = (extendEnv(env,I,var(loc)), sto1)
            where (sto1, loc) = allocate sto

evaluate [[I]] env sto =
            if dval = int(n) or dval = bool(p)
                    then dval
                    else if dval = var(loc)
                              then if applySto(sto,loc) = undefined
                                            then error
                                            else applySto(sto,loc)
            where dval = applyEnv(env,I)
```

Figure 13.1: Denotational Semantics for Part of Pelican

In action semantics, the meaning of a programming language is defined by mapping program phrases to actions. The performance of these actions models the execution of the program phrases. To define these few constructs from Pelican, we need to describe several primitive actions, two operations that yield data, and two composite actions. Primitive actions can store data in storage cells, bind identifiers to data, compute values, test Boolean values, and so on. The following primitive actions include the ones needed to define the fragment of Pelican plus a few others as examples:

complete	Terminate normally the action being performed.
fail	Abort the action being performed.
give _	Give the value obtained by evaluating a yielder.
allocate a cell	Allocate a memory location.
store _ in _	Store a value in a memory location.
bind _ to _	Bind an identifier to data produced by a yielder.

These examples illustrate a syntactic convention wherein parameters to operations are indicated by underscores. Operations in action semantics can be prefix, infix, or outfix. Outfix operators have only internal place holders such as in “sum(_,_)”. The last two examples above are considered prefix since they end with a place holder. Infix operators begin and end with argument places—for example, “_ or _”. The operations are evaluated with prefix having the highest precedence and outfix the lowest. Prefix operators are executed from right to left, and infix from left to right.

Other operations—the yielders in action semantics—give values that depend on the current information, such as the current storage and the current bindings:

the _ stored in _	Yield the value of a given type stored in a memory location.

the _ bound to _	Yield the object of a certain type bound to an identifier.
the given _	Yield the value of the specified type given to the action.

Action combinators are binary operations that combine existing actions, using infix notation, to control the order in which subactions are performed as well as the data flow to and from the subactions. Action combinators are used to define sequential, selective, iterative, and block structuring control flow as well as to manage the flow of information between actions. The following two combinators model sequential control and nondeterministic choice, respectively:

_ then _

Perform the first action; when it completes, perform the second action taking the data given by the first action.

_ or _

Perform either one of the two actions, choosing one arbitrarily; if it fails, perform the other action using the original state.

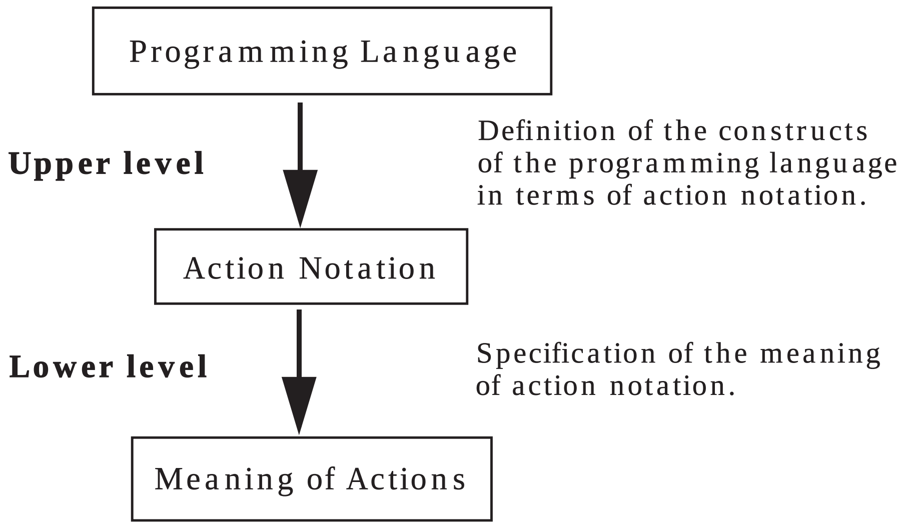
With these operations, we specify the two declarations and identifier evaluation from Pelican in Figure 13.2.

```
elaborate ⟦var I : T⟧ =
                  allocate a cell
              then
                  bind I to the given Cell

elaborate ⟦const I = E⟧ =
                  evaluate E
               then
                  bind I to the given Value

evaluate ⟦I⟧ =
                  give the Value stored in the Cell bound to I
              or
                  give the Value bound to I
```

Figure 13.2: Action Semantics for Part of Pelican

These examples convey the basic idea of action specifications. Since prefix operations are evaluated from right to left, we may omit the parentheses in “bind I to (the given Cell)” and “give (the Value stored in (the Cell bound to I))”. Observe that one of the actions in the last semantic equation must fail, thereby producing either the constant binding or the variable binding to the identifier. In the sequel we describe these primitive actions, yielders, and action combinators in more detail.

A specification of a programming language using action semantics naturally breaks into the two parts shown in the diagram below.


Programming Language
Upper level
Definition of the constructs of the programming language in terms of action notation.
Action Notation
Lower level
Specification of the meaning of action notation.
Meaning of Actions

The description of action semantics in the book by Peter Mosses [Mosses92] specifies the meaning of action notation (the lower level, which is also known as microsemantics) formally using algebraic axioms to present the notation and structural operational semantics to give the semantics of action performance. Here we describe action notation using examples, short English definitions, and diagrams, concentrating our efforts in the upper level, also known as macrosemantics, where semantics is bestowed on a programming language in terms of action notation.

Data and Sorts

The data manipulated by a programming language need to be specified in a semantic definition of the language. These data are static, mathematical objects that include entities such as cells, tuples, and maps—as well as the expected sets of integers and Boolean values. These entities are required to describe the behavior of programs in the language.

In action semantics, data are classified into sorts, which are sets of mathematical objects equipped with assorted operations on those objects. These sorts are defined by algebraic specifications. The languages presented in this chapter require the sorts TruthValue and Integer, which can be specified in a way similar to the modules in Chapter 12. In the spirit of action semantics, we define the sorts TruthValue and Integer following the syntax for algebraic specifications found in [Mosses92]. We omit the equations in the specifications and refer the reader to Chapter 12 for examples.

```
module TruthValues
    exports
        sorts TruthValue
        operations
            true    : TruthValue
            false   : TruthValue
            not _   : TruthValue → TruthValue
            both ( _ , _ )   : TruthValue, TruthValue → TruthValue
            either ( _ , _ ): TruthValue, TruthValue → TruthValue
            _ is _  : TruthValue, TruthValue → TruthValue     -- the equality relation
    end exports
    equations
        :
end TruthValues

module Integers
    imports TruthValues
    exports
        sorts Integer
        operations
            0   : Integer
            1   : Integer
            10  : Integer
            successor _          : Integer → Integer
            predecessor _        : Integer → Integer
            sum ( _ , _ )        : Integer,  Integer → Integer
            difference ( _ , _ ) : Integer,  Integer → Integer
            product ( _ , _ )    : Integer,  Integer → Integer
            integer-quotient ( _ , _ ) : Integer,  Integer → Integer
            _ is _ : Integer,  Integer → TruthValue          -- the equality relation
            _ is less than _     : Integer,  Integer → TruthValue
            _ is greater than _ : Integer,  Integer → TruthValue
    end exports
    equations
        :
end Integers
```

Sort operations allow sorts to be compared and combined to form new sorts. These operations correspond to normal set operations.

Definition : Let S_1 and S_2 be two sorts.

a) The **join** or union of S_1 and S_2 is denoted by $S_1 \mid S_2$.

b) The **meet** or intersection of S_1 and S_2 is denoted by S_1 & S_2.

c) The notation $S_1 \leq S_2$ means that S_1 is a **subsort** of S_2. ▮

The sorts used in an action semantics specification form a lattice according to the partial order ≤. Every sort automatically includes a special element, called nothing, representing the absence of information in much the same way as ⊥ was used in domain theory. We use the sort Datum to include all the values manipulated by actions and refer to tuples each of whose components is a Datum as Data. Every Datum can be viewed as a member of Data (Datum ≤ Data), since a singleton tuple is identified with the individual in the tuple. Using this notation, we can make a few assertions about sorts of data:

- The expressible values in Wren constitute the sort (Integer | TruthValue).
- (Integer | TruthValue) ≤ Datum.
- (Integer & TruthValue) = nothing.

The special value nothing plays a particularly important role in action specifications, representing the result of any operation or action that terminates abnormally. Every sort automatically contains the value nothing, which represents the empty sort. Most actions and operations specify the sort of values that will be used and produced by their performance. Whenever the wrong kind of value appears, the result will be nothing. As with any semantic methodology, programs are expected to be syntactically correct—adhering to both the context-free syntax (BNF) and the context-sensitive syntax (context constraints dealing with type checking)—before they are submitted to semantic analysis. In spite of this, action semantics follows a strict type discipline in specifying the meaning of language constructs. This careful delineation of the types of objects manipulated by actions adds to the information conveyed by the semantic descriptions. Performing an action corresponding to a language construct that violates type constraints results in failure. An operation (yielder) that fails for any reason produces the value nothing, and an action that contains such an operation simply fails.

Although we can describe the sort of actions, actions themselves do not form a subsort of Datum, since actions, which work on data, cannot manipulate actions. Later we will see that actions can, however, be encapsulated into data, called abstractions, that can be "enacted", thereby causing the performance of the actions. This mechanism enables action semantics to model subprogram declaration and invocation.

Action semantics classifies data according to how far they tend to be propagated during action performance.

- **Transient** Data or tuples of data given as the immediate results of action performance are called transients. These values model the data given by expressions. They must be used immediately or be lost.
- **Scoped** These data consist of bindings of tokens (identifiers) to data as in environments. They are accessible (visible) throughout the performance of an action and its subactions, although they may be hidden temporarily by the creation of inner scopes.
- **Stable** Stable data model memory as values stored in cells (or locations) defined in a language's specification. Changes in storage made during action performance are enduring, so that stable data may be altered only by explicit actions.

When we describe actions themselves later, we will see that actions are also classified by the fundamental kind of data that they modify. This classification gives rise to the so-called **facets** of action semantics that are determined by the kind of information principally involved in an action's performance.

Yielders

During the performance of an action, certain **current information** is maintained implicitly, including:

- The transients given to and given by actions
- The bindings received by and produced by actions
- The current state of the storage

Terms that evaluate to produce data, depending on the current information, are called **yielders**. The yielders in an action semantics specification select information for processing by actions from transients, bindings, and storage, verifying its type consistency. Below we describe four yielders that play an important role in language specification.

the given _ : Data → Yielder

Yield the transient data given to an action, provided it agrees with the sort specified as Data.

the given _ # _ : Datum, PositiveInteger → Yielder

Yield the n^{th} item in the tuple of transient data given to an action, provided it agrees with the sort specified as Datum, where n is the second argument.

the _ bound to _ : Data, Token → Yielder

Yield the object bound to an identifier denoted by the Token in the current bindings, after verifying that its type is the sort specified as Data.

the _ stored in _ : Data, Yielder → Yielder

Yield the value of sort Data stored in the memory location denoted by the cell yielded by the second argument.

Token denotes a subsort of Yielder that gives identifiers. PositiveInteger is a subsort of Integer. These yielders are evaluated during action performance to produce values (Data) to be used by and given by actions.

Actions

Actions are dynamic, computational entities that model the operational behavior of programming languages. When performed, actions accept the data passed to them in the form of the current information—namely, the given transients, the received bindings, and the current state of storage—to give new transients, produce new bindings, and/or update the state of the storage. If no intermediate result is to be passed on to the next action, the transient is simply the empty tuple. Similarly, the empty binding, with every identifier unbound, is passed to the next action if the action produces no bindings.

Depending on the principal type of information processed, actions are classified into several different facets, including:

- **Functional Facet** : actions that process transient information
- **Imperative Facet** : actions that process stable information
- **Declarative Facet** : actions that process scoped information
- **Basic Facet** : actions that principally specify flow of control
- **Reflective Facet** : actions that handle abstractions (subprograms)
- **Hybrid Action Notation** : actions that deal with recursive bindings

An action performance may **complete** (terminate normally), **fail** (terminate abnormally), or **diverge** (not terminate at all).

The Functional Facet

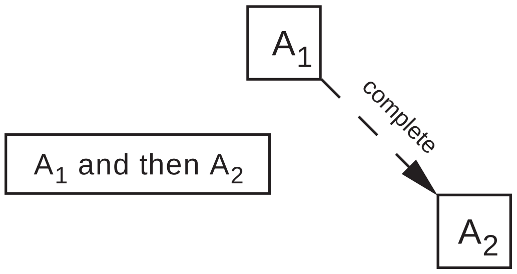
Actions and yielders classified in the functional facet primarily manipulate the transients given to and given by actions. First we consider composite actions in the functional and basic facets, a few of the so-called action combinators. These combinators may also process scoped information, but we defer the discussion of bindings until a later section in this chapter.

Action combinators have the signature

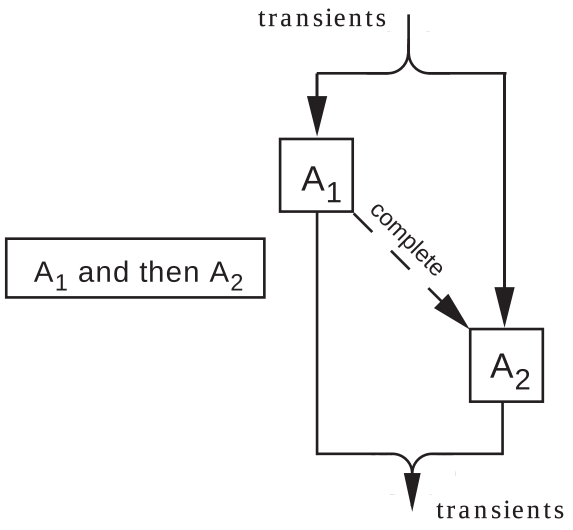
$$\text{combinator} : \text{Action}, \text{Action} \to \text{Action}$$

and are normally written using infix notation. At this point we are concerned only with control flow and the processing of transients.

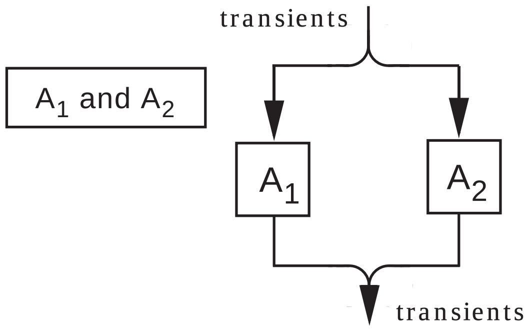
The basic combinator "A_1 and then A_2" performs the first action and then performs the second. We illustrate the control flow between the two actions by a dashed line in a diagram that indicates that the first action must terminate normally (complete) before the second action can be performed.



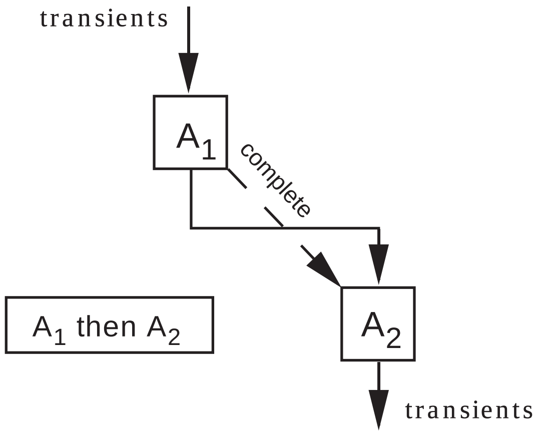
Both actions can use the transients passed to the combined action. The transients given by each action are concatenated and given by the combined action. We depict the concatenation of transients (tuples) by joining the data flow lines. The transient from the first action precedes that from the second in the concatenation, which is ordered from left to right. Adding the processing of the transients to "A_1 and then A_2" gives the following diagram:



The basic action combinator "A_1 and A_2" allows interleaving of the performance of the two actions. The diagram below shows no control dependency between the two actions, suggesting that they can be performed collaterally. Both actions use the transients passed to the combined action. The transients given by each action are concatenated and given by the combined action.



The functional action combinator "A_1 then A_2" performs the first action using the transients given to the combined action and then performs the second action using the transients given by the first action. The transients given by the combined action are those given by the second action. The dashed line shows the control dependency.



For each of these action combinators, if one of the actions gives the value nothing, the result of the composite action is nothing. We say these combinators are **strict** in the value nothing.

The sample language in the next section—namely, the calculator language from section 9.2—uses a primitive functional action give : Yielder → Action, which was mentioned earlier; "give Y" where Y is a yielder (a term that evaluates to a data value) gives the value computed from Y as its transient.

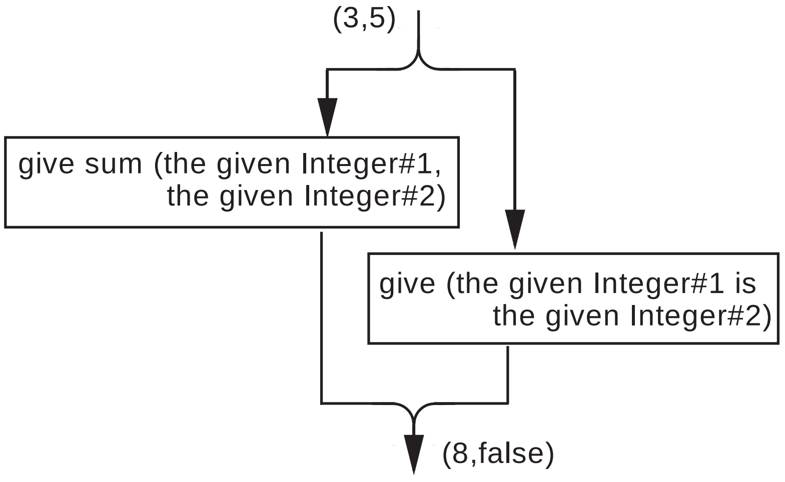
The yielder "the given S" where S is a sort of data, retrieves and type checks the datum in the given transient. The yielder takes a parameter that is a sort to document the type of the datum in the transient.

The yielder "the given S#n" retrieves and type checks the nth datum in the given transient tuple.

For example, the composite action

```
        give sum (the given Integer#1, the given Integer#2)
    and
        give (the given Integer#1 is the given Integer#2)
```

provided with the tuple (3,5) as a transient, gives the tuple (8,false) as its result. The operation is serves as equality for integers.


(3,5)
give sum (the given Integer#1, the given Integer#2)
give (the given Integer#1 is the given Integer#2)
(8,false)

The tuple (3,3) given as a transient will result in the tuple (6,true) as the transient given by this composite action.

The Imperative Facet

Actions and yielders in the imperative facet deal with storage, allocating memory locations, updating the contents of locations, and fetching values from memory. All actions work on a common store consisting of an unlimited number of **cells**, which are data of the sort Cell. Initially all cells are considered unused. When an object of sort Cell is allocated, it changes from being unused to containing a special value called undefined. The values that may be stored in memory belong to a sort called Storable, corresponding to the storable values in denotational semantics. Thus when specifying an imperative programming language, we need to specify the sort Storable. Any action may alter the state of a cell, and such a modification remains in effect until some other action modifies the cell again by storing a different value in it or by deallocating the cell. Therefore we think of the data stored in cells as **stable** information.

Cells form a sort of data that can be left unspecified. This abstract data type requires only that cells are distinguishable and that we have an unlimited number of them, although only a finite number will be in use at any one time. We can view the current storage as a finite mapping from cells to the sort (Storable | undefined).

Two primitive imperative actions allocate and update storage cells.

allocate a cell

Find an unused cell, storing the value undefined in it, and give the (object of sort) Cell as the transient of the action.

The actual allocation algorithm is not important, as long as it always yields an unused cell when performed. In [Mosses92] the allocate operation is a hybrid action defined in terms of primitive actions from the imperative and functional facets. We treat it as a primitive action to simplify the discussion. The precedence rules for action semantics allow us to use multiword operation names without any confusion. The expression “allocate a cell” represents a nullary operation. A primitive action defines the modification of a memory cell.

store Y_1 in Y_2

Update the cell yielded by Y_2 to contain the Storable yielded by Y_1.

The imperative facet has no special action combinators, but any action has the potential of altering storage. In the combination “A_1 and then A_2”, any changes to storage by A_1 precede those made by A_2. In contrast, if both A_1 and A_2 in “A_1 and A_2” modify memory, the results are unpredictable because of the possible interleaving.

The yielder “the S stored in Y” gives the datum currently stored in the cell yielded by evaluating Y provided that the value given is a datum of sort S. Otherwise, the yielder gives nothing.

Suppose that two locations, denoted by $cell_1$ and $cell_2$, have been allocated and currently contain the value undefined. Also assume that the next cell to be allocated is $cell_3$. Figure 13.3 shows snapshots of the current storage as a composite action is performed.

The first subaction to the combinator then gives $cell_3$ as a transient to the second subaction that stores a value there. Observe how indenting determines the grouping of the actions.

Initial storage:

$cell_1$	undefined
$cell_2$	undefined

```
            store 3 in cell_1
    and then
            store 5 in cell_2
    and then
                    allocate a cell
            then
                    store sum (the Integer stored in cell_1,
                               the Integer stored in cell_2)
                         in the given Cell
```

After store 3 in $cell_1$:

$cell_1$	3
$cell_2$	undefined

After store 5 in $cell_2$:

$cell_1$	3
$cell_2$	5

After store sum … in the given Cell:

$cell_1$	3
$cell_2$	5
$cell_3$	8

Figure 13.3: Current Storage While Performing an Action

Exercises

1. Assuming the value 5 as the given transient, diagram the following composite actions:

a)
```
            give -7
        and
            give the given Integer
    then
        give product (the given Integer#1, the given Integer#2)
```

b)
```
        give difference (0,the given Integer)
    then
            give successor(the given Integer)
        and
            give predecessor(the given Integer)
    then
        give sum (the given Integer#1, the given Integer#2)
```

2. Actions from the functional facet can be viewed as describing a simple language of expressions that define functions that take a value (the tran-

sient) and give a result (the new transient). Describe the functions defined by the following actions:

a)
```
     give successor (the given Integer)
   then
     give product (the given Integer, the given Integer)
```

b)
```
     give product (2, the given Integer)
   then
     give successor (the given Integer)
```

c)
```
     give successor (the given Integer)
   and then
     give product (the given Integer, the given Integer)
```

d)
```
           give predecessor (the given Integer)
       and
           give successor (the given Integer)
   then
       give product (the given Integer#1, the given Integer#2)
```

3. Suppose that the following action is given a cell containing the integer 5 as a transient. What (possible) numbers can be stored in the cell after performing this action?

```
   store 0 in the given Cell
and
       store successor (the Integer stored in the given Cell) in the given Cell
   and then
       store sum (the Integer stored in the given Cell, 10) in the given Cell
```

4. Suppose that the current storage contains only two defined cells: { $\text{cell}_1 \mapsto 6$, $\text{cell}_2 \mapsto -2$ }. Describe the current storage after performing the following action:

```
           give Integer stored in cell₁
       and
           give 10
   then
       store product (the given Integer#1, the given Integer#2) in the given cell₁
   then
       give successor (the Integer stored in cell₂)
   then
       store difference (the given Integer, the Integer stored in cell₁) in  cell₂
```

Assuming that cell_1 corresponds to the variable x and cell_2 to y, what assignment command(s) are modeled by the performance of this action?

13.2 ACTION SEMANTICS OF A CALCULATOR

We use the calculator from section 9.2 as the first example of a complete specification by means of action semantics. Here the definition of the calculator semantics is somewhat simplified by using the imperative facet to provide a storage location for the calculator memory. A module describes the necessary imperative features needed for the specification.

```
module Imperative
    imports Integers, Mappings
    exports
        sorts     Storable = Integer,
                  Storage = Mapping [Cell to (Storable | undefined)],
                  Cell ≤ Datum
        operations
                  cell₁  : Cell
                  allocate a cell : Action
                  store _ in _ : Yielder, Yielder → Action
                  the _ stored in _ : Storable, Yielder → Yielder
                          :
    end exports
    equations
        :
end Imperative
```

The module Imperative imports the module Mappings that specifies objects to model finite functions, instantiating an object of sort Mapping using the notation “Mapping [domain to codomain]”. We use slightly different (from Chapter 12) but equivalent notation–namely, Storage = Mapping [Cell to (Storable | undefined)]— to instantiate the parameters to the Mappings module and to rename (really give synonyms for) identifiers from the imported module. Using Mappings, Imperative can specify an empty map, a mechanism for adding a new ordered pair to the map, a way to change the image of a domain element, and notation for applying the map to a domain item. Here we only name the operations, actions, data, and yielders, used to manipulate storage in action notation.

For reference we repeat the abstract syntax of the calculator language in Figure 13.4, slightly modified for the action semantic specification. It is a common practice to fit the definition of the abstract syntax to the method used for the semantic specification. For example, we used different definitions of abstract syntax in Chapter 8 and Chapter 9. However, since the concrete syntax of the calculator language is unchanged from Chapter 9, we are specifying the semantics of the same language.

Abstract Syntactic Domains

P : Program	E : Expression	D : Digit
S : ExprSequence	N : Numeral	

Abstract Production Rules

Program ::= ExprSequence

ExprSequence ::= Expression | Expression ExprSequence

Expression ::= Numeral | $\mathbf{M}^{\mathbf{R}}$ | **Clear** | Expression **+** Expression
 | Expression **–** Expression | Expression **x** Expression
 | Expression $\mathbf{M}^{+}$ | Expression **=** | Expression $^{+}$**/-**

Numeral ::= Digit | Numeral Digit

Digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Figure 13.4: Abstract Syntax for the Calculator Language

Semantic Functions

As with denotational semantics, meaning is ascribed to the calculator language via semantic functions, mostly mapping syntactic domains to actions. Because of the expressiveness of action semantics using the imperative facet, we need fewer semantic functions.

meaning _	:	Program → Action	
perform _	:	ExprSequence → Action	
evaluate _	:	Expression → Action	
the value of _	:	Numeral → Integer	-- uses only nonnegative integers
digit value _	:	Digit → Integer	

The action that serves as the meaning of a program gives a value that can be taken as the semantics of the program in the calculator language. This value, corresponding to an expressible value in denotational semantics, is the integer shown in the display as a result of executing the program. We describe its sort, a subsort of Datum, by the definition

Value = Integer -- expressible values.

The sort Action includes many operational behaviors but conveys no specifics about the nature of individual actions. To make specifications more precise, entities of sort Action can be qualified by describing the **outcome**, the sort of information produced by an action, and the **income**, the sort of information used by an action. A subsort of Action can be defined by writing

Action [outcome] [income].

We omit the details that describe possible income and outcome entities formally, but the terminology itself suggests the intent of the qualifications. The semantic functions for the calculator are more accurately specified by the following signatures:

meaning _ : Program → Action [completing | giving a Value | storing]
[using current storage]

perform _ : ExprSequence → Action [completing | giving a Value | storing]
[using current storage]

evaluate _ : Expression → Action [completing | giving a Value | storing]
[using current storage]

Note that the symbol |, which denotes the join or union of sorts, is an associative operation.

Semantic Equations

The semantic function evaluate does the bulk of the work in specifying the calculator language. The value given by performing the action resulting from evaluate, and from the functions meaning and perform, is the value shown in the display of the calculator. Although the display value can be considered as the semantics of a phrase in the calculator language, meaning is more fully specified by describing the activities performed to obtain this value. The actions of action semantics are designed to model these activities.

The evaluate function takes the nine different forms of expressions as its actual parameter. For each we describe the intended operational behavior, and then give the semantic equation that describes this behavior as an action. Observe how closely the definition in action notation parallels the informal description.

- **Numeral**

  To evaluate a numeral, simply display its integer value on the display.

  evaluate N = give the value of N

  The value given as a transient by the action give is the displayed integer.

- **Memory Recall**

  Display the value stored in the single memory location that we assume has been allocated initially and named cell_1. The module Imperative asserts the existence of a constant cell, cell_1, to serve this purpose.

  evaluate $\mathbf{M^R}$ = give Integer stored in cell_1

  Again the transient given by the action is the displayed value.

- **Clear**

The clear operation resets the memory location to zero and displays zero.

```
        evaluate Clear =
                                store 0 in cell$_1$
                         and
                                give 0
```

Since we have no reason to perform one of these subactions before the other, the composite action uses the and combinator. If interference were possible between the two activities, we could use and then to establish order. In the case of **Clear**, choosing an order of performance over-specifies the behavior.

- **Addition of Two Expressions**

This binary operation gives the sum of the integers that result from the two expressions. The left expression must be evaluated first since it may involve a side effect by storing a value in the calculator memory.

```
        evaluate [[E1 + E2]] =
                                  evaluate E1
                          and then
                                  evaluate E2
                     then
                          give sum (the given Integer#1, the given Integer#2)
```

The first combinator forms a tuple (a pair) consisting of the values of the two expressions, which are evaluated from left to right. That tuple is given to the sum operation, which adds the two components. Action semantics uses indenting to describe the evaluation order of action operations. Parentheses are also allowed for this purpose, but generally the indenting convention is easier to follow. For comparison, consider the semantic equation for addition written using parentheses.

```
      evaluate [[E1 + E2]] =  (evaluate E1 and then evaluate E2)  then
                              (give sum (the given Integer#1, the given Integer#2))
```

Since evaluate is a prefix operation, it takes precedence over the and then combinator. Actually none of these parentheses are necessary, but composite actions are easier to read if they employ some grouping mechanism.

- **Difference of Two Expressions**
- **Product of Two Expressions**

Subtraction and multiplication are handled in the same way as addition, but difference and product are used to implement the operations.

- **Add to Memory**

  Display the value of the current expression and add it to the calculator memory.

```
        evaluate ⟦E M+⟧ =
                        evaluate E
                  then
                              store sum (the Integer stored in cell1, the given Integer) in cell1
                        and
                              give the given Integer
```

  The second subaction to then must propagate the transient from the first subaction so that it can be given by the composite action. The primitive action "store _ in _" yields no transient, which is represented by an empty tuple. The action "give the given Integer" propagates the integer from the evaluation of E. Action semantics views a single datum as identical to a singleton tuple containing the same value. Concatenating the empty tuple and the singleton tuple produces the value of E as the transient of the and combinator. Without the subaction "give the given Integer", the value from E will be lost. It must be propagated as the resulting transient from the entire action.

  Action semantics has a primitive action regive that abbreviates "give the given Data". We use this abbreviation in some of the remaining examples, although using regive reduces the information since the sort of the Data is not specified.

- **Equal**

  The equal key terminates an evaluation, displaying the value from the current expression.

```
        evaluate ⟦E =⟧ = evaluate E
```

- **Change Sign**

  The +/- key flips the sign of the integer produced by the latest expression evaluation.

```
        evaluate ⟦E +/-⟧ =
                                  evaluate E
                              then
                                  give difference (0, the given Integer)
```

The meaning function initializes the calculator by storing zero in the memory location and then evaluates the expression sequence. The storing operation gives an empty transient. The semantic function perform evaluates the expressions in the sequence one at a time, ignoring the given transients. The semantic functions value of and digit value have essentially the same behavior as the corresponding functions in denotational semantics. All of the semantic equations for the action semantics of the calculator language are collected in Figure 13.5.

```
meaning P =
            store 0 in cell$_1$
        and then
            perform P

perform ⟦E S⟧ =
            evaluate E
        then
            perform S

perform E = evaluate E

evaluate N = give the value of N

evaluate **M^R** = give Integer stored in cell$_1$

evaluate **Clear** =
            store 0 in cell$_1$
        and
            give 0

evaluate ⟦E$_1$ + E$_2$⟧ =
                evaluate E$_1$
            and then
                evaluate E$_2$
        then
            give sum (the given Integer#1, the given Integer#2)

evaluate ⟦E$_1$ – E$_2$⟧ =
                evaluate E$_1$
            and then
                evaluate E$_2$
        then
            give difference (the given Integer#1, the given Integer#2)

evaluate ⟦E$_1$ x E$_2$⟧ =
                evaluate E$_1$
            and then
                evaluate E$_2$
        then
            give product (the given Integer#1, the given Integer#2)
```

Figure 13.5: Semantic Equations for the Calculator Language (Part 1)

```
evaluate ⟦E M+⟧ =
          evaluate E
       then
               store sum (the Integer stored in cell1, the given Integer) in cell1
          and
               regive               -- give the given Data

evaluate ⟦E =⟧ = evaluate E

evaluate ⟦E +/-⟧ =
          evaluate E
       then
          give difference (0, the given Integer)

the value of ⟦N D⟧ = sum (product (10,the value of N), the value of D)

the value of D = digit value D

digit value 0 = 0
     :
digit value 9 = 9
```

Figure 13.5: Semantic Equations for the Calculator Language (Part 2)

The use of the combinator “and then” in the definition of meaning and perform is only used to sequence the control flow since the transients are ignored between the subactions.

Action semantics uses the emphatic brackets “⟦” and “⟧” slightly differently than denotational semantics. Semantic functions are applied to abstract syntax trees. In action semantics the notation “⟦E_1 **+** E_2⟧” denotes the abstract syntax tree composed of a root and three subtrees, E_1, **+**, and E_2. Since E_1 is already an abstract syntax tree, we have no need for another set of brackets in the expression “evaluate E_1”. We omit the brackets in each semantic equation that gives meaning to an abstract syntax tree that consists of a single subtree (a single object) as in “evaluate **Clear**”.

A Sample Calculation

As an example, consider the calculator program elaborated in Figure 9.7:

12 + 5 $^{+}$/- = x 2 M^{+} 123 M^{+} M^{R} $^{+}$/- – 25 = + M ^{R} =

This sequence of calculator keystrokes parses into three expressions, so that the overall structure of the action semantics evaluation has the form

meaning ⟦**12 + 5 $^{+}$/- = x 2 M^{+} 123 M^{+} M^{R} $^{+}$/- – 25 = + M ^{R} =**⟧

```
=        store 0 in cell₁
    and then
         perform ⟦12 + 5 +/- = x 2 M+ 123 M+ MR +/- – 25 = + M R =⟧

=        store 0 in cell₁
    and then
         evaluate ⟦12 + 5 +/- = x 2 M+⟧
    then
         evaluate ⟦123 M+⟧
    then
         evaluate ⟦MR +/- – 25 = + M R =⟧
```

The first expression begins with an empty transient and with cell_1 containing the value 0. We show the transient (as a tuple) given by each of the subactions as well as the value stored in cell_1.

```
                                                                  Transient   cell₁
evaluate ⟦12 + 5 +/- = x 2 M+⟧ =                                     ( )         0

            give the value of 12                                    (12)         0
        and then
              give the value of 5                                    (5)         0
            then
              give difference (0, the given Integer)                (-5)         0
      then                                                        (12,-5)        0
        give sum (the given Integer#1, the given Integer#2)          (7)         0
    and then
        give the value of 2                                          (2)         0
  then                                                              (7,2)        0
    give product (the given Integer#1, the given Integer#2)         (14)         0
  then
      store sum (the Integer stored in cell₁, the given Integer) in cell₁   ( )   14
    and
      regive                                                        (14)        14
```

This action gives the value 14, which is also the value in cell_1. The second expression starts with 14 in memory, ignoring the given transient, and results in the following action:

```
evaluate ⟦123 M+⟧ =
    give the value of 123                                          (123)        14
  then
      store sum(the Integer stored in cell₁,the given Integer) in cell₁   ( )   137
    and
      regive                                                       (123)       137
```

This action gives the value 123 and leaves the value 137 in $cell_1$. The third expression completes the evaluation, starting with 137 in memory, as follows:

```
evaluate ⟦MR +/- – 25 = + M R =⟧ =
            give Integer stored in cell1                         (137)        137
          then
            give difference (0, the given Integer)               (-137)       137
        and then
          give the value of 25                                    (25)        137
      then                                                     (-137,25)      137
        give difference (the given Integer#1, the given Integer#2) (-162)     137
    and then
      give Integer stored in cell1                               (137)        137
  then                                                        (-162,137)      137
    give sum (the given Integer#1, the given Integer#2)          (-25)        137
```

This final action gives the value -25, leaving the value 137 in the calculator's memory.

Exercises

1. Evaluate the semantics of these combinations of keystrokes using the action semantics definition in this section:

   a) **8 $^+$/- + 5 x 3** =

   b) **7 x 2 M$^+$ M$^+$ M$^+$ – 15 + M ^{R}** =

   c) **10 – 5 $^+$/- M$^+$ 6 x M^R M$^+$** =

   Consult the concrete syntax of the calculator language in section 9.2 when parsing these programs. For instance, the program in part a is grouped in the manner as shown by the parentheses below:

   ((((**8 $^+$/-**) + **5**) **x 3**) =)

2. Add to the calculator a key **sqr** that computes the square of the value in the display. Alter the semantics to model the action of this key. Its syntax should be similar to that of the **$^+$/-** key.

3. Prove that for any expression E, meaning ⟦E = **M$^+$**⟧ = meaning ⟦E **M$^+$** =⟧.

4. Some calculators treat "**=**" differently than the calculator in this section, repeating the most recent operation, so that "**2 + 5 = =** " leaves 12 on the display and "**2 + 5 = = =** " leaves 17. Consider the changes that must be made in the action semantics to model this alternative interpretation.

13.3 THE DECLARATIVE FACET AND WREN

Actions and yielders in the declarative facet deal primarily with scoped information in the form of bindings between identifiers, represented as tokens, and various semantic entities such as constants, variables, and procedures. In this section we illustrate several fundamental concepts from the declarative facet, along with a couple of actions dealing with control flow from the basic facet, in specifying the programming language Wren. Since Wren has such a simple structure with respect to declarations—namely, a single global scope—only a few features of the declarative facet are introduced. More complicated actions from the declarative facet are discussed in section 13.4, where we provide an action specification of Pelican.

One aspect of defining a programming language involves specifying what kinds of values can be bound to identifiers, the so-called denotable values in denotational semantics. In action semantics the subsort of Datum that consists of entities that can be bound to identifiers is known as the sort Bindable. Wren allows binding identifiers only to simple variables, which are modeled as cells in action semantics. The algebraic specification in a module called Declarative suggests the salient aspects of the entities in the declarative facet.

```
module Declarative
    imports Imperative, Mappings
    exports
        sorts   Token
                Variable = Cell,
                Bindable = Variable,
                Bindings = Mapping [Token to (Bindable | unbound)]
        operations
                empty bindings : Bindings
                bind _ to _ : Token, Yielder → Action
                the _ bound to _ : Data, Token  → Yielder
                produce _ : Yielder → Action
                        :
    end exports
    equations
        :
end Declarative
```

The term “empty bindings” denotes bindings with every identifier unbound. Action semantics establishes a binding using the primitive declarative action “bind T to Y”, which produces a singleton binding mapping that we represent informally by [T |→B] where B is the datum of sort Bindable yielded by Y.

A declarative yielder finds the value associated with an identifier in the current bindings. The term “the S bound to T” evaluates to the entity bound to the Token T provided it agrees with the sort S; otherwise the yielder gives nothing. The action “produce Y” creates the bindings consisting of the map yielded by Y. It corresponds to the action “give Y” in the functional facet.

Before considering composite actions from the declarative facet, we observe that the action combinators defined earlier process bindings as well as transients. Although the action combinators introduced as part of the functional and basic facets do not concentrate on processing bindings, they receive bindings as part of the current information and produce possibly new bindings as a result of their subactions. The bindings of two actions can combine in two fundamental ways:

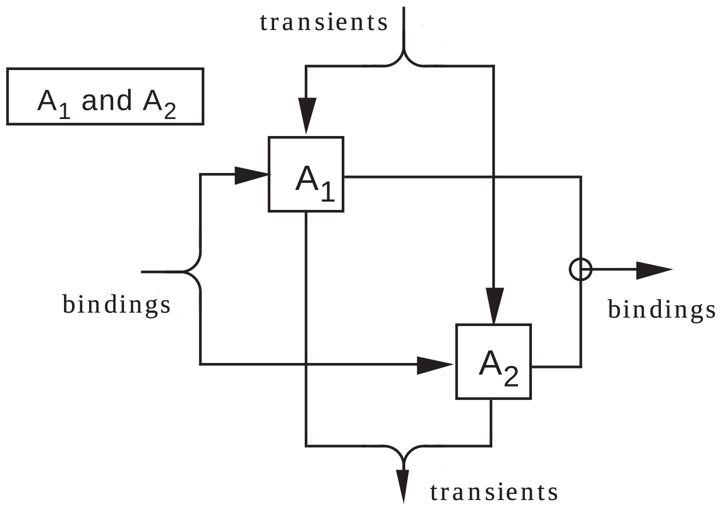
merge(bindings_1,bindings_2):

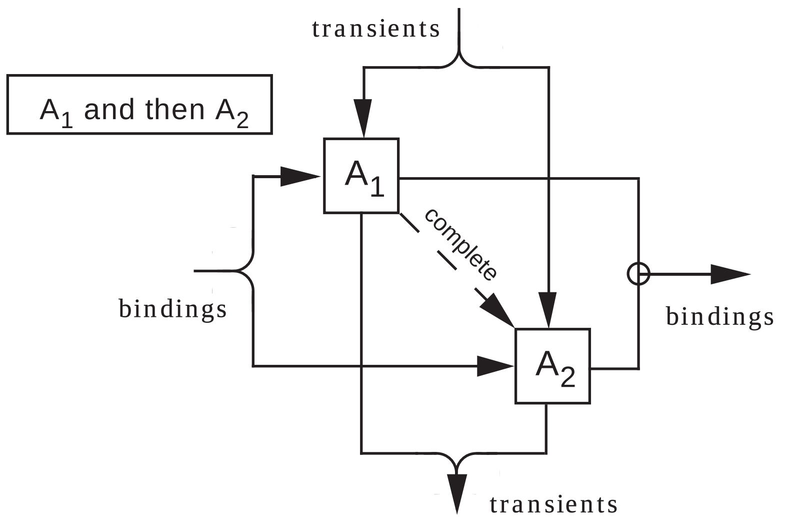
Merging the sets of bindings means to form their (disjoint) union with the understanding that if any identifier has bindings in both sets, the operation fails, producing nothing.

overlay(bindings_1,bindings_2):

The bindings are combined in such a way that the associations in bindings_1 take precedence over those in bindings_2.

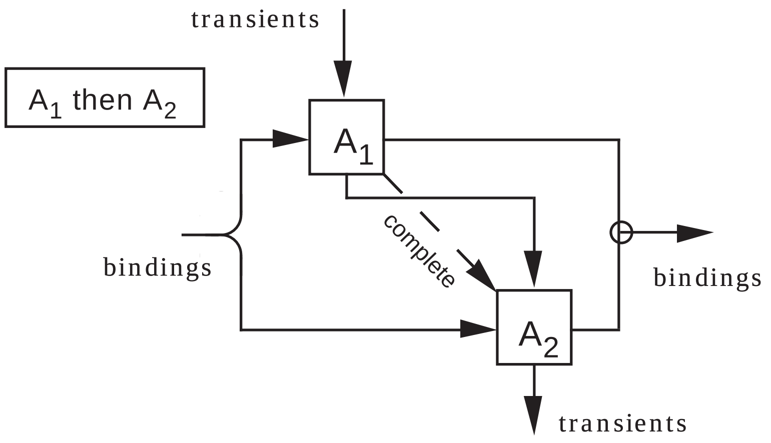
In the following diagrams, scoped information flows from left to right whereas transients still flow from top to bottom. We depict the merging of bindings by having the lines for scoped information connected by a small circle suggesting a disjoint union. Later when action combinators use the overlay operation, the lines show a break indicating which set of bindings takes precedence.



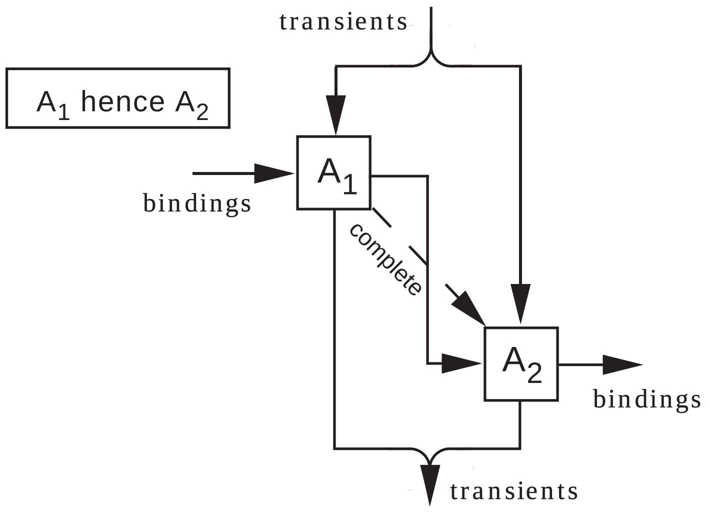


For both of the combinators and and and then, each action receives the bindings for the composite action, and the bindings produced by the subactions are merged. The only difference between these two action combinators is that and then enforces an ordering in the performance of the two subactions.

The action combinator then has the same declarative behavior as the combinator and then.



The only primarily declarative action combinator required in the Wren specification is the composite action hence. This combinator sequences the bindings with the first subaction receiving the original bindings, the second subaction receiving the bindings produced by the first, and the bindings produced by the combined action being those produced by the second subaction. The combinator hence processes transients in the same way as the combinator and then.



The Programming Language Wren

We now turn to describing an action specification of Wren (see section 1.3 or 9.3 for the syntax of Wren). We omit that part of action semantics used to describe input and output, so the **read** and **write** commands from Wren are ignored in this chapter. Input and output require the communicative facet, a topic beyond the scope of our presentation. In the action semantics description of Wren, we specify the declarative information of the language despite the simplicity of its scope rules. The kinds of information processed by Wren can be specified as the three sorts:

```
sorts  Value = Integer | TruthValue,      -- expressible values
       Storable = Integer | TruthValue,   -- storable values
       Bindable = Variable                -- denotable values (Variable = Cell)
```

Four new semantic functions provide meaning to the phrases of Wren. The signatures below include the outcome and income to help describe the behavior of the resulting actions.

```
run _ : Program → Action  [completing | diverging | storing]
                          [using current storage]
elaborate _ : Declaration → Action  [completing | binding | storing]
                                    [using current bindings | current storage]
execute _ : Command → Action [completing | diverging | storing]
                             [using current bindings | current storage]
```

evaluate _ : Expression $\rightarrow$ Action [completing | giving a Value]
[using current bindings | current storage]

For each syntactic construct, we give a brief informal description of its semantics and then provide its definition in action semantics.

- **Program**

  First elaborate the declarations, which involve only variables, and then execute the body of the program using the resulting bindings. The program identifier is ignored, serving as documentation only.

  run ⟦**program** I **is** D **begin** C **end**⟧ = elaborate D hence execute C

- **Variable Declaration**

  Allocate a cell from storage and then bind the identifier to that cell. The definition handles declarations of a single variable only. Multiple variable declarations can be treated as a sequence of declarations.

```
elaborate ⟦var I : T⟧ =
            allocate a cell
        then
            bind I to the given Cell
```

- **Empty Declaration**

  Produce no bindings. "⟦ ⟧" denotes an empty tree.

  elaborate ⟦ ⟧ = produce empty bindings

- **Sequence of Declarations**

  Elaborate the first declaration and then elaborate the second using the bindings from the first and producing the combined bindings.

  elaborate ⟦D_1 D_2⟧ = elaborate D_1 and then elaborate D_2

  The bindings in D_1 should be visible to the second declaration, although in Wren D_2 has no way to refer to an identifier in D_1. For this reason, the "and then" combinator suffices to specify declaration sequencing in Wren. With and then each subaction constructs bindings independently, and the two sets of bindings are merged. In a program that satisfies the context constraints for Wren, no conflict can arise between the declarations in D_1 and D_2 when they merge. The combinator then could be used as well since the transients play no role in these declarations.

- **Sequence of Commands**

  Execute the first command and then execute the second.

  execute ⟦C_1 **;** C_2⟧ = execute C_1 and then execute C_2

- **Skip**

  Do nothing.

  ```
  execute skip = complete
  ```

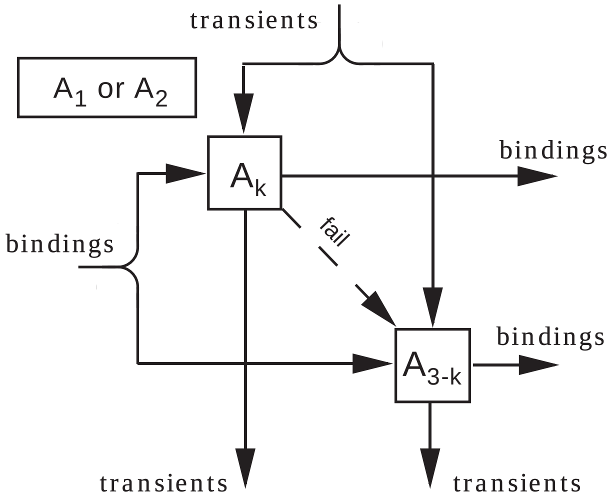
- **Assignment**

  Find the cell bound to the identifier and evaluate the expression. Then store the value of the expression in that cell.

  ```
  execute ⟦I := E⟧ =
                        give the Cell bound to I and evaluate E
                    then
                        store the given Value#2 in the given Cell#1
  ```

  The parameters to the and combinator are presented without indentation. The "bound to" yielder and the give action take precedence because prefix operations are always performed before infix ones. Parentheses can be used to alter precedence or to enhance clarity.

To describe the decision process in **if** and **while** commands, we need an action combinator that belongs to the basic facet and a primitive action from the functional facet. The action combinator or models nondeterministic choice. "A_1 or A_2" arbitrarily chooses one of the subactions and performs it with the given transients and the received bindings. If the chosen action fails, the other subaction is performed with the original transients and bindings. The effect of or is shown in the diagram below with k = 1 or k = 2, but which one is not specified by action semantics.



Although most action combinators are strict relative to failure (if one of the subactions fails, the composite action also fails), "A_1 or A_2" can complete (succeed) even though one of its subactions fails. However, if the chosen action fails after making a change to storage, the change is irrevocable, the other action is ignored, and the whole action fails.

The primitive functional action "check Y", where Y is a yielder that gives a TruthValue, completes if Y yields true and fails if it yields false. The action gives empty transients and produces empty bindings. The action check acts as a guard, which when combined with the composite action or enables a specification to carry out decision making.

- **If Commands**

  The **if** commands evaluate the Boolean expression that serves as the test, and then they perform the **then** command or the **else** command depending on the test. If the **else** part is missing, the command does nothing when the condition is false.

```
execute [[if E then C1 else C2]] =
          evaluate E
     then
               check (the given TruthValue is true) and then execute C1
          or
               check (the given TruthValue is false) and then execute C2

execute [[if E then C]] =
          evaluate E
     then
               check (the given TruthValue is true) and then execute C
          or
               check (the given TruthValue is false) and then complete
```

  The operation is acts as equality for the sort TruthValue. Observe that for each of the **if** commands only one of the conditions supplied to the action check can be true. The phrase "and then complete" may be omitted from the second definition. It simply provides symmetry to the or combinator. Also, the first check test can read

  check (the given TruthValue) and then execute C.

To complete the specification of commands in Wren, we need two more actions, unfolding _ and unfold, from the basic facet to define the **while** command. These actions serve only to determine the flow of control during the performance of subactions.

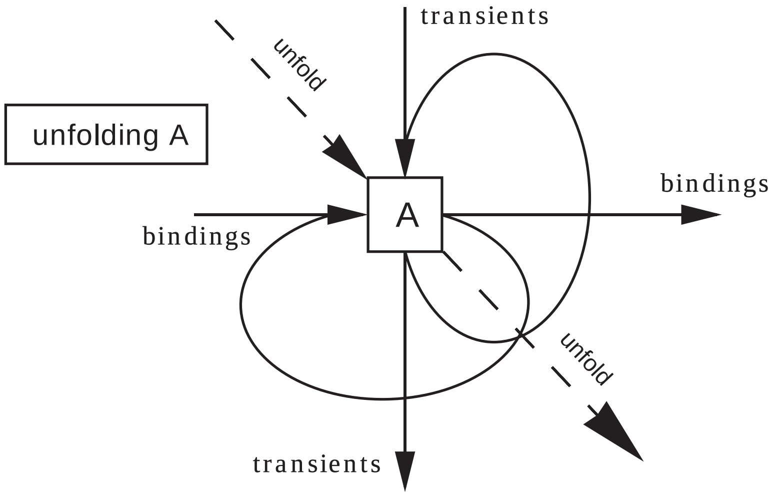
unfolding _

  The composite action unfolding : Action $\rightarrow$ Action performs its argument action, but whenever the dummy action unfold is encountered, the argument action is performed again in place of unfold.

unfold

The primitive action unfold is a dummy action, standing for the argument action of the innermost enclosing unfolding.

The diagram below suggests the behavior of the action unfolding A. Whenever the action A performs unfold, it is restarted with the transients and bindings that are given to unfold. Eventually we expect A to complete producing the final transients and bindings.



The actions unfolding and unfold are used to describe indefinite iteration—in this case, the **while** command in Wren. Inside a performance of unfolding, an invocation of unfold has the effect of restarting the original action.

- **While Command**

  The Boolean expression is evaluated first. If its value is true, the body of the loop is executed and then the **while** command is started again when the execution of the loop body completes; otherwise, the command terminates.

```
execute ⟦while E do C⟧ =
        unfolding
               evaluate E
            then
                  check (the given TruthValue is true)
                      and then  execute C
                          and then unfold
               or
                  check (the given TruthValue is false) and then complete
```

We conclude the specification of Wren by giving the semantic equations for evaluate, the function that defines the meaning of expressions.

- **Variable Name**

  Give the value stored in the memory location bound to the variable.

  ```
  evaluate I = give Value stored in the Cell bound to I
  ```

  The precedence rules of action semantics assume that this action is interpreted as “give (the Value stored in (the Cell bound to I))”.

- **Literal**

  Give the value of the literal.

  ```
  evaluate N = give the value of N

  evaluate true = give true

  evaluate false = give false
  ```

- **Arithmetic on Two Expressions**

  Evaluate the two expressions and give the sum of their values.

  ```
  evaluate ⟦E1 + E2⟧ =
                      evaluate E1
                  and
                      evaluate E2
              then
                  give sum (the given Integer#1, the given Integer#2)
  ```

  Since Wren allows no side effects in expressions, we have no need to specify an order of evaluation of the components in a binary expression. Subtraction, multiplication, and division are handled in a similar manner. If the integer-quotient operation is given zero as a divisor, the operation gives nothing, and that causes the action to fail.

  ```
  evaluate ⟦E1 / E2⟧ =
                      evaluate E1
                  and
                      evaluate E2
              then
                  give integer-quotient (the given Integer#1, the given Integer#2)
  ```

- **Unary Minus**

  Evaluate the expression and give the negation of the resulting value.

  ```
  evaluate ⟦- E⟧ =
                  evaluate E
              then
                  give difference (0, the given Integer)
  ```

- **Relational Expressions**

  Evaluate the two expressions and give the result of applying the appropriate relation operation to the two values.

```
evaluate ⟦E₁ < E₂⟧ =
                evaluate E₁
            and
                evaluate E₂
        then
            give (the given Integer#1 is less than the given Integer#2)
```

- **Binary Boolean Operations**

  Evaluate the two expressions and give the result of applying the appropriate Boolean operation to the two values.

```
evaluate ⟦E₁ and E₂⟧ =
                evaluate E₁
            and
                evaluate E₂
        then
            give both (the given TruthValue#1, the given TruthValue#2)
```

- **Boolean Not**

  Evaluate the expression and give the logical negation of the given value.

```
evaluate ⟦not(E)⟧ =
            evaluate E
        then
            give not (the given TruthValue)
```

Exercises

1. Add these language constructs to Wren and define them using action semantics.

   a) repeat-until commands

      Command ::= ... | **repeat** Command **until** Expression

   b) conditional expressions

      Expression ::= ... | **if** Expression **then** Expression **else** Expression

   c) expressions with side effects

      Expression ::= ... | **begin** Command **return** Expression **end**

2. Provide a definition of conditional (short-circuit) **and** and **or** in action semantics. Use the syntactic forms “E_1 **and then** E_2” and “E_1 **or else** E_2” for these expressions.

3. Extend Wren to allow constant declarations and explain how the action specification needs to be modified.

4. Give an action specification of the vending machine in exercise 8 of section 9.3.

13.4 THE REFLECTIVE FACET AND PELICAN

The major changes when we move from Wren to Pelican (see section 9.5) have to do with declarations: Identifiers can now also be bound to constant values and to procedures. Therefore the sort Bindable includes two more possibilities.

```
sorts Bindable = Variable | Value | Procedure          -- denotable values
```

In action semantics procedure objects are modeled as abstractions, which are yielders that encapsulate actions. We defer specifying procedures until later in this section. Now we consider the scope rules of Pelican, which are more complicated than those in Wren, requiring several additional declarative actions.

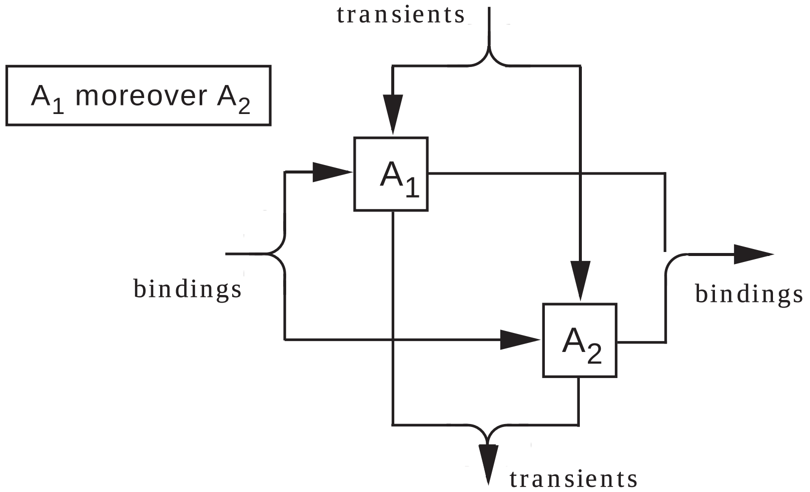
rebind

This primitive declarative action reproduces all of the received bindings. The action rebind propagates bindings in a manner analogous to the way regive propagates transients. The effect of rebind is to extend the scope of the current bindings.

_ moreover _

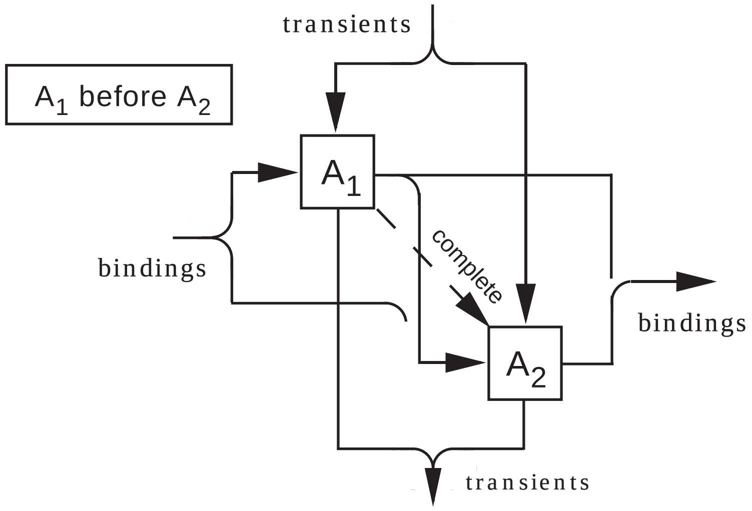
As with the combinator and, moreover allows the performance of the two actions to be interleaved. Both actions use the transients and bindings passed to the combined action. The bindings produced by the combined action are the bindings produced by the first action overlaid by those produced by the second. Transients are handled as with the and combinator.

The diagram below shows the blending of the bindings using the overlay operation by means of a broken line. The bindings that follow the solid line take precedence.



_ before _

The declarative action combinator before performs the first action using the transients and the bindings passed to the combined action, and then performs the second action using the transients given to the combined action and the bindings received by the combined action overlaid by those produced by the first action. The combined action produces the bindings produced by the first action overlaid with those produced by the second. The transients given by the combined action are those given by the first action concatenated with those given by the second.



Pelican allows several more kinds of bindings than Wren. We give the three sorts that specify the kinds of information processed by Pelican, noting that only Bindable is different from the specification for Wren.

```
sorts  Value = Integer | TruthValue,              -- expressible values
       Storable = Integer | TruthValue,           -- storable values
       Bindable = Variable | Value | Procedure    -- denotable values
```

The semantic functions for Pelican have the same signatures as in the specification of Wren, but we need to add several semantics equations for the additional language constructs in Pelican. We postpone describing procedures for now and concentrate on constant declarations and the **declare** command.

- **Constant Declaration**

  Evaluate the expression and then bind its value to the identifier.

```
elaborate [[const I = E]] =
            evaluate E
        then
            bind I to the given Value
```

- **Sequence of Declarations**

  Elaborate the declarations sequentially. Since the scope rules for Pelican are more complicated, allowing nested scopes, we use the composite action before to combine the bindings from the two declarations so that D_1 overlays the enclosing environment and D_2 overlays D_1.

  elaborate [[D_1 D_2]] = elaborate D_1 before elaborate D_2

  The “and then” combinator no longer suffices for declaration sequencing. Pelican requires that each declaration has access to the identifiers that are defined earlier in the same block as well as those in any enclosing block, as illustrated by the declaration sequence below:

```
const  max = 50;
       max1 = max+1;
```

  Pelican allows “dynamic expressions” in constant definitions. Using before ensures that identifiers elaborated in D_1 are visible when D_2 is elaborated. Pelican does not require that D_2 overlay D_1, since declarations in a sequence must have distinct identifiers. They may just as well be merged, but no problems arise when before performs an overlay at two points in the processing of bindings. Now that we have the combinator before, it can also be used in place of and then in defining declaration sequencing in Wren.

- **Variable Name or Constant Identifier**

  An identifier can be bound to a constant value or to a variable. Evaluating an identifier gives the constant or the value assigned to the variable.

```
evaluate ⟦I⟧ =
                give the Value stored in the Cell bound to I
            or
                give the Value bound to I
```

Only one of the subactions to the or combinator succeeds, so that the action gives the appropriate value denoted by the identifier I.

- **Anonymous Block (declare)**

  Elaborate the declarations in the block, producing bindings that overlay the bindings received from the enclosing block, and execute the body of the block with the resulting bindings. The bindings created by the local declaration are lost after the block is executed.

```
execute ⟦declare D begin C end⟧ =
                rebind moreover elaborate D
            hence
                execute C
```

  The action rebind propagates the bindings given to it. Therefore the action “rebind moreover elaborate D” overlays the received bindings (from the enclosing block) with the local bindings from D to provide the environment in which C will execute.

As an illustration of this mechanism for handling the declarations in Pelican, consider the following program.

```
program scope is
    const c = 5;
    var n : integer;
begin
    declare
            const m = c+8;          -- D1
            const n = 2*m;          -- D2
    begin
      :                             -- C
    end;
  :
end
```

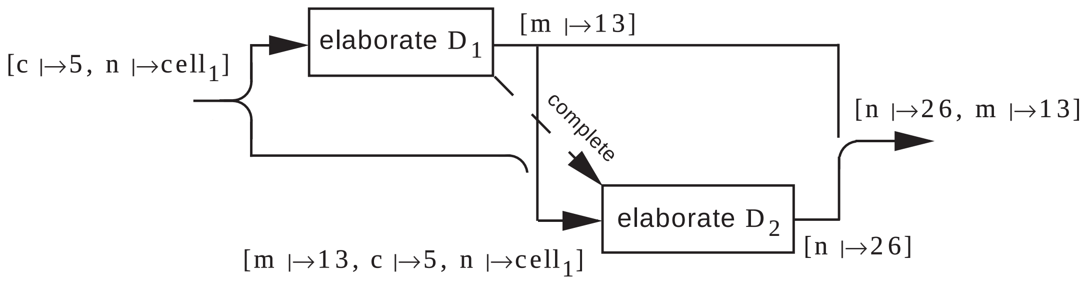
Assuming that the first cell allocated is cell_1, the action that elaborates the first two declarations produces the bindings [c |→5, n |→cell_1], which are received by the body of the program and therefore by the **declare** command. The following action models the execution of the declare command.

```
execute ⟦declare D1; D2; begin C end⟧ =
            rebind moreover elaborate ⟦D1 D2⟧
        hence
            execute C
```

Working from the inside, we first elaborate the declarations

elaborate ⟦D_1 D_2⟧ = elaborate D_1 before elaborate D_2.

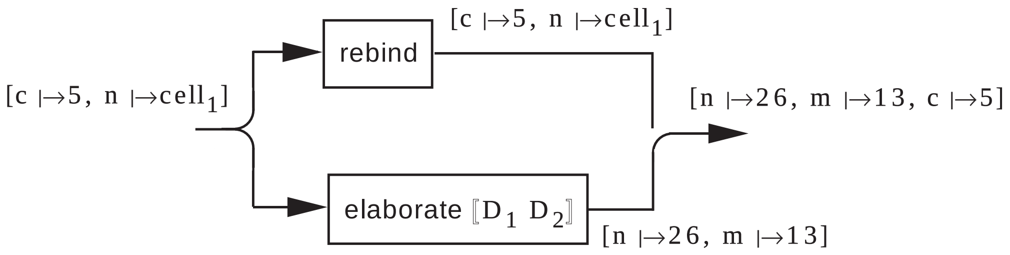
The diagram below, with the empty transients omitted, illustrates the activities carried out by the before combinator.



This action, elaborate ⟦D_1 D_2⟧, serves as the second subaction in

rebind moreover elaborate ⟦D_1 D_2⟧,

which is depicted in the next diagram.



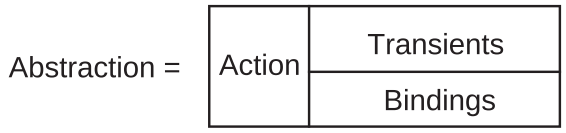
Therefore the body of the anonymous block will execute in an environment containing three bindings, [n |→26, m |→13, c |→5].

The Reflective Facet and Procedures

The reflective facet addresses those actions and yielders that allow the description of subprogram declaration and invocation. The activity of a procedure in Pelican can be modeled by the performance of an action. Recall that actions themselves are not data but can be incorporated in data called abstractions. Objects that can be bound to identifiers in Pelican include procedures, which are modeled as abstractions.

sorts Procedure = Abstraction
Bindable = Variable | Value | Procedure

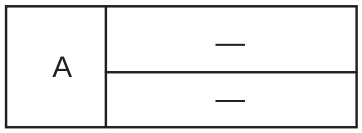
View an abstraction datum as an entity with three components, the action itself and the transients and bindings, if any, that will be given to the action when it is performed.



As with subprograms in a programming language, we concern ourselves with two aspects: the creation of a procedural object by means of a declaration and the invocation of the object that sets it into action. When a Pelican procedure declaration is elaborated, the code of the procedure modeled as an action is incorporated into an abstraction using an operation that acts as a yielder.

abstraction of _ : Action $\rightarrow$ Yielder

The yielder "abstraction of A" encapsulates the action A into an abstraction together with no transients and no bindings.



If we want the action inside an abstraction to be performed with certain transients and bindings, they must be supplied after the abstraction is constructed. The current bindings are inserted into an abstraction using an operation on yielders.

closure of _ : Yielder $\rightarrow$ Yielder

The yielder "closure of Y" incorporates the bindings received by the enclosing action into the abstraction given by Y. Attaching the declaration-time bindings, those bindings in effect when the subprogram is declared, ensures that the resulting action performs the defined procedure in its static environment, thereby producing static scoping for resolving references to nonlocal identifiers. Assuming that StaticBindings denotes the current bindings in effect when the declaration is elaborated, the term "closure of abstraction of A" yields the object shown below. In this example, bindings are inserted into an abstraction at abstraction-time.

A	—
	StaticBindings

Once bindings are incorporated into an abstraction, no further changes can be made to the bindings. A later performance of “closure of _” will have no effect. Dynamic scoping ensues if bindings are attached at enaction-time—that is, when a procedure is called and the action in its abstraction is to be performed. We define the execution of a procedure using a reflective action enact that takes as its parameter a yielder that gives an abstraction.

enact _ : Yielder → Action

> The action “enact Y” activates the action encapsulated in the abstraction yielded by Y, using the transients and bindings that are included in the abstraction. If no transients or bindings have been incorporated into the abstraction, the enclosed action is given empty transients or empty bindings at enaction-time.

Procedures Without Parameters

We now have enough action notation to specify parameterless procedures in Pelican, handling both their declaration and call, but first we repeat that procedures are represented by the subsort of Datum known as Abstraction in the action specification.

sorts Procedure = Abstraction

- **Procedure Declaration (no parameter)**

  Bind the identifier of the declaration to a procedure object that incorporates the body of the procedure, so that it will be executed in the declaration-time environment.

```
elaborate ⟦procedure I is D begin C end⟧ =
                bind I to
                   closure of
                      abstraction of
                            rebind moreover elaborate D
                         hence
                            execute C
```

  The abstraction bound to I incorporates the current (static) bindings and empty transients. Executing the body of the procedure resembles the execution of a **declare** command (see the semantic equation for **declare**).

- **Procedure Call (no parameter)**

  Execute the procedure object bound to the identifier.

  execute ⟦I⟧ = enact the Procedure bound to I

  Recall that the procedure object, an abstraction, brings along its static environment. The action corresponding to a parameterless procedure expects no transients, and the abstraction bound to I has empty transients.

Procedures With A Parameter

We need a mechanism that allows an actual parameter to be passed to the procedure. Another operation on yielders constructs an unevaluated term—a yielder—that provides a way for the current transient to be incorporated into the abstraction.

application of _ to _ : Yielder, Yielder → Yielder

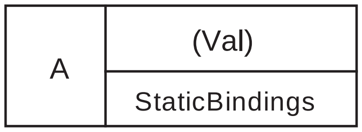
The yielder “application of Y_1 to Y_2” attaches the argument value yielded by Y_2 as the transient that will be given to the action encapsulated in the abstraction yielded by Y_1 when that action is enacted. As with bindings, a further supply of transients to an abstraction is ignored. The argument, a value, is inserted into the abstraction when the procedure is called.

- **Procedure Call (one parameter)**

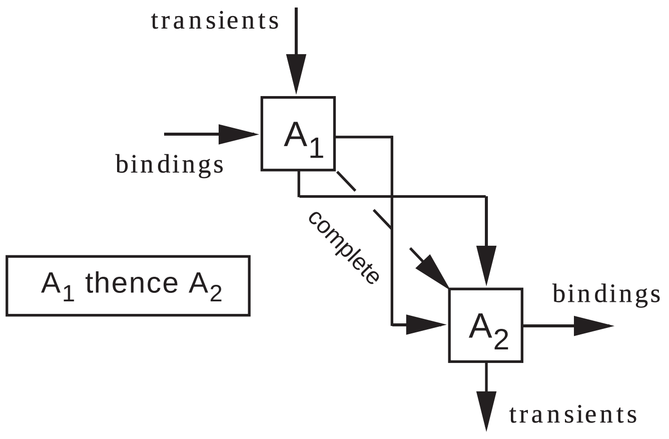
  Evaluate the actual parameter, an expression, and then execute the procedure bound to the identifier with the value of the expression.

```
execute ⟦I (E)⟧ =
            evaluate E
        then
            enact application of (the Procedure bound to I) to the given Value
```

  Assuming that Abs, the abstraction bound to I, incorporates the action A and the bindings StaticBindings, and that Val is the value of the expression E, “application of Abs to the given Value” creates the abstraction that will be enacted. The actual parameter(s) to a procedure provide the only transient information that is relevant at enaction-time.



To specify the declaration of procedures with one parameter, we need another action combinator thence that combines the behavior of then for transients and hence for bindings. Therefore both transients and bindings flow sequentially through the two actions.



The action encapsulated in an abstraction formed by a declaration of a procedure with a parameter expects a value, the actual parameter, to be given to it as a transient. This value is stored in a new memory location allocated by the action. The command that constitutes the body of the procedure is executed in an environment that consists of the original static environment, inserted into the abstraction using "closure of", overlaid by the binding of the formal parameter to the allocated variable, and then overlaid by the local declarations.

- **Procedur e Declaration (one parameter)**

  Bind the procedure identifier in the declaration to a procedure object that incorporates the body of the procedure, so that when it is called, it will be executed in the declaration-time environment and will allocate a local variable for the actual parameter passed to the procedure.

```
elaborate ⟦procedure I₁ (I₂) is D begin C end⟧ =
        bind I₁ to
            closure of
                abstraction of
                        allocate a cell and give the given Value and rebind
                    thence
                            rebind
                        moreover
                                bind I₂ to the given Cell#1
                            and
                                store the given Value#2 in the given Cell#1
                    hence
                        rebind moreover elaborate D
                    hence
                        execute C
```

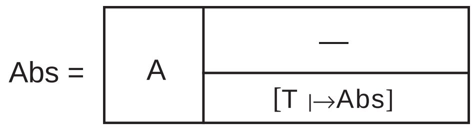
The three uses of rebind ensure that the bindings at each stage of the specification are extensions of the bindings at the previous stage. The first argument to thence passes a tuple consisting of a Cell and a Value (Integer or TruthValue) as transients to the second argument. The action combinators thence and hence are associative, so we have no need of indentation in the expression “A_1 thence A_2 hence A_3 hence A_4”.

Recursive Definitions

The specifications of procedure declarations shown above do not allow recursive calls of the procedures, since the identifiers (procedure names) being declared are not included in the bindings associated with the abstractions created by the declarations. The details of the hybrid actions that implement recursive bindings are beyond the scope of our discussion of action semantics. We can, however, describe a hybrid action for establishing recursive bindings that is defined in terms of more primitive actions.

recursively bind _ to _ : Token, Bindable $\rightarrow$ Action

The action “recursively bind T to abstraction of A” produces the binding of T, an identifier, to an abstraction Abs so that the bindings attached to the action A incorporated in Abs include the binding being produced.



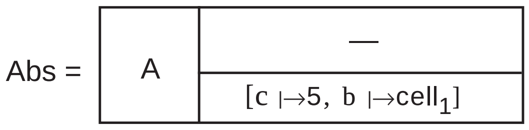
Therefore the action “recursively bind _ to _” permits the construction of a circular binding.

```
elaborate ⟦procedure I is D begin C end⟧ =
            recursively bind I to
                closure of
                    abstraction of
                            rebind moreover elaborate D
                        hence
                            execute C
```

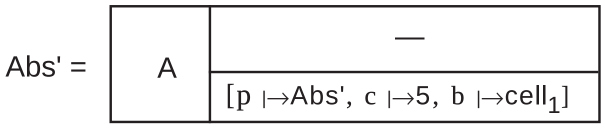
To illustrate the effects of a recursive declaration, consider the bindings created by a Pelican program.

```
program example is
    const c = 5;
    var b : boolean ;
    procedure p is … begin … end;
begin … end
```

Let A denote the action corresponding to the body of the procedure. The action “closure of abstraction of A” creates the abstraction Abs shown below, which does not allow a recursive call of the procedure.



The action “bind p to closure of abstraction of A” produces the binding [p |→Abs]. Any reference to the procedure identifier p inside the procedure is an illegal reference, yielding nothing. In contrast, the action “recursively bind p to closure of abstraction of A” changes the abstraction Abs into a new abstraction Abs' whose attached bindings include the association of the procedure abstraction with p. Now a recursive call is permitted.



The recursive action produces the binding [p |→Abs'], which when overlaid on the previous (enclosing) bindings, produces the bindings [p |→Abs', c |→5, b |→cell$_1$] to be received by the procedure p and the body of the program.

Figure 13.6 collects the definitions for an action semantic specification of Pelican. Observe how many of the definitions are identical to those of Wren.

Translating to Action Notation

Action notation can be viewed as a metalanguage for the semantic specification of programming languages. The semantic equations in Figure 13.6 define a translator from Pelican programs into action notation, which can act as an intermediate language in an interpreter or a compiler. By providing an interpreter of action notation, we can obtain a prototype implementation of any programming language with a specification in action semantics. A translator of action notation into a machine language produces a compiler of the language.

The metalanguage of action semantics can also be used to verify semantic equivalence between language phrases. Although an action specification can be read at an informal level, it is a formal definition. Furthermore, action notation can be manipulated algebraically using properties such as associativity, commutativity, and identity laws to prove the equivalence of certain action expressions. Two language phrases are semantically equivalent if their translations into action notation are equivalent. Discovering the algebraic properties of action notation is an area of ongoing research. See the further readings for more on this topic.

```
run _ : Program → Action  [completing | diverging | storing]
                          [using current storage]

    run ⟦program I is D begin C end⟧ = elaborate D hence execute C


elaborate _ : Declaration → Action [completing | binding | storing]
                                   [using current bindings | current storage]

    elaborate ⟦ ⟧ = produce empty bindings

    elaborate ⟦D1 D2⟧ = elaborate D1 before elaborate D2

    elaborate ⟦var I : T⟧ =
                    allocate a cell
                then
                    bind I to the given Cell

    elaborate ⟦const I = E⟧ =
                    evaluate E
                then
                    bind I to the given Value

    elaborate ⟦procedure I is D begin C end⟧ =
            recursively bind I to
                closure of
                    abstraction of
                            rebind moreover elaborate D
                        hence
                            execute C

    elaborate ⟦procedure I1 (I2) is D begin C end⟧ =
            recursively bind I1 to
                closure of
                    abstraction of
                            allocate a cell and give the given Value and rebind
                        thence
                                rebind
                            moreover
                                    bind I2 to the given Cell#1
                                and
                                    store the given Value#2 in the given Cell#1
                        hence
                            rebind moreover elaborate D
                        hence
                            execute C
```

Figure 13.6: Semantic Equations for Pelican (Part 1)

```
execute _ : Command → Action  [completing | diverging | storing]
                              [using current bindings | current storage]

    execute ⟦C1 ; C2⟧ = execute C1 and then execute C2

    execute ⟦declare D begin C end⟧ =
                rebind moreover elaborate D
            hence
                execute C

    execute skip = complete

    execute ⟦I := E⟧ =
                give the Cell bound to I and evaluate E
            then
                store the given Value#2 in the given Cell#1

    execute ⟦if E then C⟧ =
                evaluate E
            then
                    check (the given TruthValue is true) and then execute C
                or
                    check (the given TruthValue is false) and then complete

    execute ⟦if E then C1 else C2⟧ =
                evaluate E
            then
                    check (the given TruthValue is true) and then execute C1
                or
                    check (the given TruthValue is false) and then execute C2

    execute ⟦while E do C⟧ =
            unfolding
                    evaluate E
                then
                        check (the given TruthValue is true) and then
                            execute C and then unfold
                    or
                        check (the given TruthValue is false) and then complete

    execute I = enact the Procedure bound to I

    execute ⟦I (E)⟧ =
                evaluate E
            then
                enact application of (the Procedure bound to I) to the given Value
```

Figure 13.6: Semantic Equations for Pelican (Part 2)

```
    evaluate _ : Expression → Action  [completing | giving a Value]
                                      [using current bindings | current storage]

        evaluate I =
                    give the Value stored in the Cell bound to I
                or
                    give the Value bound to I

        evaluate N = give the value of N

        evaluate true = give true

        evaluate false = give false

        evaluate ⟦E1 + E2⟧ =
                    evaluate E1 and evaluate E2
                then
                    give sum (the given Integer#1, the given Integer#2)
          :           :

        evaluate ⟦– E⟧ =
                    evaluate E
                then
                    give difference (0, the given Integer)

        evaluate ⟦E1 >= E2⟧ =
                    evaluate E1 and evaluate E2
                then
                    give not (the given Integer#1 is less than the given Integer#2)
          :           :

        evaluate ⟦E1 or E2⟧ =
                    evaluate E1 and evaluate E2
                then
                    give either (the given TruthValue#1, the given TruthValue#2)
          :           :

        evaluate ⟦not(E)⟧ =
                    evaluate E
                then
                    give not (the given TruthValue)
```

Figure 13.6: Semantic Equations for Pelican (Part 3)

We conclude this section by translating a Pelican program into its equivalent action notation. This task is aided by the property of compositionality: Each phrase is defined solely in terms of the meaning of its immediate subphrases. Furthermore, any phrase may be substituted for a semantically equivalent phrase without changing the meaning of the program.

We illustrate a translation of the following Pelican program annotated as shown below:

```
program action is
     const max = 50;             -- D1
     var sum : integer;          -- D2
     var switch : boolean;       -- D3
     var n : integer;            -- D4
     procedure change is         -- D5
          begin
               n := n+3;
               switch := not(switch)
          end;
begin
     sum := 0;                        -- C1
     n := 1;                          -- C2
     switch := true;                  -- C3
     while n<=max do                  -- C4
          if switch then sum := sum+n end if;
          change
     end while
end
```

The overall structure of the translation takes the form

run ⟦**program** I **is** D_1 D_2 D_3 D_4 D_5 **begin** C_1; C_2; C_3; C_4 **end**⟧

= elaborate ⟦D_1 D_2 D_3 D_4 D_5⟧ hence execute ⟦C_1; C_2; C_3; C_4⟧

= elaborate D_1 before elaborate D_2 before elaborate D_3
before elaborate D_4 before elaborate D_5
hence
execute C_1 and then execute C_2 and then execute C_3 and then execute C_4

The elaboration uses the property that the combinators and then and before are both associate. We proceed by elaborating the four declarations in the program.

elaborate D_1 = give the value of 50 then bind max to the given Value

elaborate D_2 = allocate a cell then bind sum to the given Cell

elaborate D_3 = allocate a cell then bind switch to the given Cell

elaborate D_4 = allocate a cell then bind n to the given Cell

```
elaborate D5 =
   recursively bind change to closure of(abstraction of(
            rebind
         moreover
            produce empty bindings
      hence
                  give the Cell bound to n
               and
                        give the Value stored in Cell bound to n
                     or
                        give the Value bound to n
                  and
                     give the value of 3
                  then
                     give sum(the given Integer#1,the given Integer#2)
            then
               store the given Value#2 in the given Cell#1
         and then
                  give the Cell bound to switch
               and
                     give the Value stored in Cell bound to switch
                  or
                     give the Value bound to switch
               then
                  give not(the given Truthvalue)
            then
               store the given Value#2 in the given Cell#1))
```

The translation of the Pelican program is completed by expanding the four commands.

```
execute C1 =   give the Cell bound to sum and give the value of 0
            then
               store the given Value#2 in the given Cell#1

execute C2 =   give the Cell bound to n and give the value of 1
            then
               store the given Value#2 in the given Cell#1

execute C3=    give the Cell bound to switch and give true
            then
               store the given Value#2 in the given Cell#1
```

```
execute $C_4$ =
    unfolding
                give the Value stored in Cell bound to n
             or
                give the Value bound to n
          and
                give the Value stored in Cell bound to max
             or
                give the Value bound to max
        then
          give not(the given Integer#1 is greater than the given Integer#2)
     then
             check (the given Truthvalue is true)
          and then
                      give the Value stored in Cell bound to switch
                   or
                      give the Value bound to switch
                then
                      check the given Truthvalue is true
                   and then
                               give the Cell bound to sum
                             and
                                   give the Value stored in Cell bound to sum
                                  or
                                   give the Value bound to sum
                               and
                                   give the Value stored in Cell bound to n
                                 or
                                   give the Value bound to n
                           then
                             give sum(the given Integer#1,the given Integer#2)
                        then
                           store the given Value#2 in the given Cell#1
                 or
                      check the given Truthvalue is false
                   and then
                      complete
          and then
             enact the Procedure bound to change
          and then unfold
        or
             check the given Truthvalue is false and then complete
```

Exercises

1. Suppose that the current bindings contain two pairs: [x |→cell$_1$, y |→2]. Consider two actions:

   A$_1$ = bind y to 15

   A$_2$ = bind x to successor(the Integer bound to y)

   What are the (possible) current bindings after performing the following composite actions?

   a) A$_1$ and then A$_2$

   b) A$_1$ hence A$_2$

   c) A$_1$ and A$_2$

   d) A$_1$ moreover A$_2$

   e) A$_1$ before A$_2$

2. Extend Pelican to include a definite iteration command using the syntax

   for I := E$_1$ **to** E$_2$ **do** C **end for**

   and assuming iteration over integer values only. Following the semantics of the **for** command in Pascal and Ada, provide an action specification of this command. Observe the difference in how Pascal and Ada treat the loop variable I:

   a) Pascal: Assume I has been declared in the block containing the **for** command.

   b) Ada: The **for** command implicitly declares I to have the subrange E$_1$..E$_2$ and to have scope extending through the body of the command only.

3. Modify Pelican so that parameters are passed by

   a) reference

   b) value-result

4. Modify Pelican so that it uses dynamic scoping to resolve nonlocal variable references.

5. Suppose that Pelican is extended to include functions of one parameter, passed by value. The abstract syntax now has productions of the form

   Declaration ::= … | **function** Identifier$_1$ (Identifier$_2$) **is** Declaration **begin** Command **return** Expression **end**

   and

   Expressions ::= … | Identifier (Expression).

Make all the necessary changes in the action definition of Pelican to incorporate this new language construct.

6. A unit for a binary operation @ : A,A → A is an element u of A such that for all a∈A, a@u = u@a = a. Using the primitive actions complete, fail, regive, and rebind, identify units for the following action combinators:

   and then, and, then, or, hence, moreover, before, and thence.

7. Which of the binary combinators in exercise 6 are associative, commutative, and/or idempotent?

8. Translate the following Pelican programs into action notation:

a)
```
program facwhile is
      var n : integer ;
      var f : integer ;
begin
         n := 8; f := 1;
         while n>1 do
               f := f*n; n := n–1
         end while
end
```

b)
```
program facproc is
      const num = 8;
      var n : integer ;
      var f : integer ;
      procedure fac(n : integer) is
         procedure mul(m : integer) is
            begin f := f*m end;
         begin
            if n=0 then f := 1 else fac(n–1); mul(n) end if
         end;
begin n := num; fac(n) end
```

13.5 LABORATORY: TRANSLATING INTO ACTION NOTATION

Prolog serves well as an implementation language for a translator from Pelican to action notation. The compositional definitions of the meaning of Pelican given in Figure 13.6 convert to Prolog clauses directly. The resulting actions can be represented as Prolog structures by writing actions, yielders, and auxilliary operations with prefix syntax. First we show a sample execution of the translator. The output has been edited (indented) to make the scope of the actions easier to determine.

```
>>> Translating Pelican into Action Semantics  <<<
Enter name of source file: **small.pelican**
    program small is
       const c = 34;
       var n : integer;
      begin
        n := c+21
      end
Translated Action:
hence(
  before(
    then(give(valueof(34)),bind(c,given(Value))),
    before(then(allocateacell,bind(n,given(Cell))),
           produce(emptybindings))),
  andthen(
    then(
      and(give(boundto(Cell,n)),
          then(and(or(give(storedin(Value,boundto(Cell,c))),
                      give(boundto(Value,c))),
                   give(valueof(21))),
               give(sum(given(Integer,1),given(Integer,2))))),
      storein(given(Value,2),given(Cell,1))),
    complete))
yes
```

Since this translation is purely a static operation, we need not be concerned with stores and environments—these are handled when action notation is interpreted or compiled. At the top level a predicate run translates a program. Observe that we have dispensed with the syntactic category of blocks to match the specification in Figure 13.6, thereby giving another example of tailoring the abstract syntax to the specification method. Several small changes will be needed in the parser to reflect this alteration in the abstract syntax.

```
run(prog(Decs,Cmds),hence(ElaborateD,ExecuteC)) :-
                                    elaborate(Decs,ElaborateD),
                                    execute(Cmds,ExecuteC).
```

The Prolog predicate that implements the translation of programs builds Prolog structures that represent the equivalent action using calls to the predicates elaborate and execute to construct pieces of the structure. Two clauses deal with sequences of the declarations.

```
elaborate([ ],produce(emptybindings)).
elaborate([Dec|Decs],before(ElaborateDec,ElaborateDecs)) :-
                                                elaborate(Dec,ElaborateDec),
                                                elaborate(Decs,ElaborateDecs).
```

Individual declarations are translated by Prolog clauses that match the action definitions in Figure 13.6 in their logical structure.

```
elaborate(var(T,var(Ide)),then(allocateacell,bind(Ide,given('Cell')))).

elaborate(con(Ide,E),then(EvaluateE,bind(Ide,given('Value')))) :-
                                                 evaluate(E,EvaluateE).

elaborate(proc(Ide,param(Formal),Decs,Cmds),
recursivelybind(Ide,
   closureof(abstractionof(
     hence( hence(
              thence(and(allocateacell,and(give(given('Value')),rebind)),
                          moreover( rebind,
                               and( bindto(Formal,given('Cell',1)),
                                    storein(given('Value',2),given('Cell',1))))),
              moreover(rebind,ElaborateD)),
          ExecuteC))))) :-  elaborate(Decs,ElaborateD),
                            execute(Cmds,ExecuteC).
```

We leave the clause for procedures with no parameters as an exercise. Commands are translated by the predicate execute. We provide several examples and leave the remaining clauses as exercises.

```
execute([Cmd|Cmds],andthen(ExecuteCmd,ExecuteCmds)) :-
                                                execute(Cmd,ExecuteCmd),
                                                execute(Cmds,ExecuteCmds).

execute([ ],complete).

execute(declare(Decs,Cmds),hence(moreover(rebind,ElaborateD),ExecuteC)) :-
                                                elaborate(Decs,ElaborateD),
                                                execute(Cmds,ExecuteC).

execute(skip,complete).

execute(assign(Ide,Exp),then(and(give(boundto('Cell',Ide)),EvaluateE),
                                    storein(given('Value',2),given('Cell',1)))) :-
                                                evaluate(Exp,EvaluateE).
```

```
execute(if(Test,Then),
     then(EvaluateE,or( andthen(check(is(given('Truthvalue'),true)),ExecuteC),
                        andthen(check(is(given('Truthvalue'),false)),complete)))) :-
                                          evaluate(Test,EvaluateE),
                                          execute(Then,ExecuteC).

execute(while(Test,Body),unfolding(
     then(EvaluateE,or( andthen(check(is(given('Truthvalue'),true)),
                                          andthen(ExecuteC,unfold)),
                        andthen(check(is(given('Truthvalue'),false)),complete))))) :-
                                          evaluate(Test,EvaluateE),
                                          execute(Body,ExecuteC).

execute(call(Ide,E),
    then(EvaluateE,enact(application(boundto('Procedure',Ide),given('Value'))))) :-
                                          evaluate(E,EvaluateE).
```

Expressions are translated by the Prolog predicate evaluate. Again we show several of the clauses, leaving the rest as exercises. Observe how closely the Prolog clauses agree with the action specifications.

```
evaluate(ide(Ide),or( give(storedin('Value',boundto('Cell',Ide))),
                      give(boundto('Value',Ide)))).

evaluate(num(N),give(valueof(N))).

evaluate(minus(E),then(EvaluateE,give(difference(0,given('Integer'))))) :-
                                          evaluate(E,EvaluateE).

evaluate(plus(E1,E2), then(and(EvaluateE1,EvaluateE2),
                           give(sum(given('Integer',1),given('Integer',2))))) :-
                                            evaluate(E1,EvaluateE1),
                                            evaluate(E2,EvaluateE2).

evaluate(neq(E1,E2), then(and(EvaluateE1,EvaluateE2),
                          give(not(is(given('Integer',1),given('Integer',2)))))) :-
                                            evaluate(E1,EvaluateE1),
                                            evaluate(E2,EvaluateE2).

evaluate(and(E1,E2), then(and(EvaluateE1,EvaluateE2),
                    give(both(given('Truthvalue',1),given('Truthvalue',2))))) :-
                                            evaluate(E1,EvaluateE1),
                                            evaluate(E2,EvaluateE2).
```

This action notation translator is just the first step in building a prototype implementation of Pelican. To complete the task, we need to construct an interpreter for actions. Although this code can be written in Prolog, the number of parameters may make the clauses cumbersome. For example, a predicate for interpreting an action combinator (a binary operation) will require six parameters for incoming transients, bindings, and store and three parameters for the resulting information. A language that allows us to maintain the store imperatively may produce more readable code. See the further readings at the end of this chapter for an alternative approach.

Exercises

1. Complete the implementation in Prolog of the action notation translator by writing the missing clauses.

2. Add these language constructs to Pelican and extend the translator by defining clauses that construct the appropriate action notation.

   a) repeat-until commands

      Command ::= ... | **repeat** Command **until** Expression

   b) conditional expressions

      Expression ::= ... | **if** Expression **then** Expression **else** Expression

   c) expressions with side effects

      Expression ::= ... | **begin** Command **return** Expression **end**

   d) definite iteration commands

      Command ::= ...
      | **for** Identifier := Expression **to** Expression **do**
      Command **end for**

3. Write a Prolog predicate that prints the resulting action following the indenting conventions of action semantics. Use the example at the end of section 13.4 as a model.

4. Write an interpreter for action notation in Prolog or some other programming language to produce a prototype system for this subset of Pelican (no input and output).

13.6 FURTHER READING

The standard reference for action semantics is the book by Peter Mosses [Mosses92]. He uses a subset of Ada to illustrate the full power of action semantics, including the communicative facet, which is beyond the scope of our presentation. Mosses also gives a formal specification of action notation (the lower level) using structural operational semantics. This book contains an extensive description of the literature that covers the development of action semantics over the past ten years. Note that action notation has evolved over this time frame from a more symbolic notation to a more English-like presentation. Mosses uses a slightly different framework for the algebraic specification of data, the so-called unified algebras [Mosses89].

A shorter introduction to action semantics can be found in a technical report [Mosses91]. These works contain extensive bibliographies that point to the earlier literature on action semantics. When consulting the earlier papers, note that the notation of action semantics has evolved considerably during its development.

David Watt [Watt91] has a lengthy description of action semantics with many examples, culminating in a complete action specification of Triangle, his example imperative programming language. Watt is also involved in a project, called ACTRESS, using action semantics to construct compilers [Brown92].

We mentioned in section 9.5 that Prolog may not be the best language in which to write an action interpreter. Functional programming provides a better paradigm for manipulating actions. Watt suggests implementing action semantics in ML [Watt91]. A full description of using ML to develop semantic prototypes of programming languages can be found in [Ruei93]. In this report a programming language is translated into ML functions that represent the actions and yielders. These ML functions are executed directly to provide a prototype interpreter for the language Triangle.