Action Semantics

Formal Specification of Programming Languages

Advantages:

- Unambiguous definitions
- Basis for proving properties of programs and languages
- Mechanical generation of language processors

Disadvantages:

- Notationally dense
- Often cryptic
- Unlike the way programmers view languages
- Difficult to create and modify accurately

Formal Syntax

BNF — In common use

Formal Semantics

Denotational semantics
Structural operational semantics
Axiomatic semantics
Algebraic semantics

Used only by specialists in prog. languages.

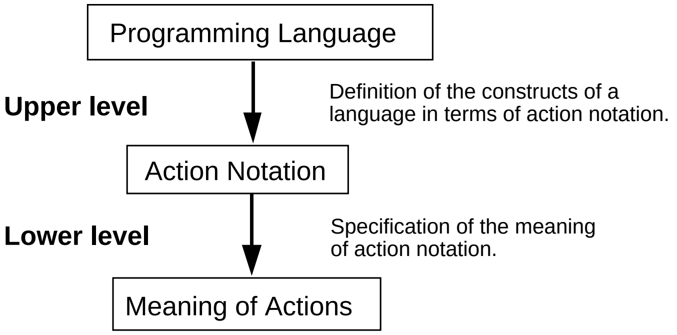
Action Semantics

- Developed by Peter Mosses and David Watt
- Based on ordinary computation concepts
- English-like notation (readable)
- Completely formal, but can be understood informally
- Reflects the ordinary computational concepts of programming languages

---

Specifying a Programming Language

An action specification breaks into two parts:



Programming Language

Upper level — Definition of the constructs of a language in terms of action notation.

Action Notation

Lower level — Specification of the meaning of action notation.

Meaning of Actions

Meaning of a language is defined by mapping program phrases to actions whose performance describes the execution of the program phrases.

Introduction to Action Semantics

Three kinds of first-order entities:

- **Data**: Basic mathematical values
- **Yielders**: Expressions that evaluate to data using current information
- **Actions:** Dynamic, computational entities that model operational behavior

Data and Sorts

Data manipulated by a programming language

- integers
- booleans
- maps
- cells
- tuples

Classification of Data

Data classified according to how far it tends to be propagated during action performance.

Transient

Tuples of data given as the immediate results of action performance. Use them or lose them.

Scoped

Data consisting of bindings of tokens (identifiers) to data as in environments.

Stable

Stable data model memory as values stored in cells (locations); may be altered by explicit actions only.

Actions are also classified this way.

Data Specification

```
module TruthValues
  exports
    sort TruthValue
    operations
      true   : TruthValue
      false : TruthValue
      not _ : TruthValue → TruthValue
      both(_,_) : TruthValue,TruthValue → TruthValue
      either(_,_)  : TruthValue,TruthValue → TruthValue
      _ is _      : TruthValue,TruthValue → TruthValue
  end exports
  equations
  …
end TruthValues
module Integers
  imports TruthValues
  exports
    sort 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
      _ is less than _ : Integer,Integer → TruthValue
      _ is greater than _ : Integer, Integer →
                                              TruthValue
  end exports
  equations
  …
end Integers
```

Sort operations (a lattice)

Join (union) of two sorts S_1 and S_2: S_1 | S_2.

Meet (intersection) of sorts S_1 and S_2: S_1 & S_2.

Bottom element: nothing

Yielders

Current information (maintained implicitly)

- the given transients,
- the received bindings, and
- the current state of the storage.

Yielders are terms that evaluate to data dependent on the current information.

the given S : Data → Yielder

Yield the transient data given to an action, provided it agrees with the sort S.

the given S # n : Datum, PosInteger → Yielder

Yield the nth item in tuple of transient data given to action, provided it agrees with sort S.

the _ bound to _ : Data, Token → Yielder

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

the _ stored in _ : Data, Yielder → Yielder

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

Precedence

Highest:	Prefix	(right-to-left)
	Infix	(left-to-right)
Lowest:	Outfix	

Actions

- When performed, actions accept the data passed to them as the current information

  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.

An action performance may

complete (terminate normally),
fail (terminate abnormally), or
diverge (not terminate at all).

Facets of Action Semantics

Actions are classified into facets, depending on the main type of information processed.

- **Functional Facet**: actions that process transient information
- **Imperative Facet**: actions that affect memory
- **Declarative Facet**: actions that process scoped information
- **Basic Facet**: actions that principally specify flow of control

---

Functional and Basic Facets

Primitive functional action

give Y	Give value obtained by evaluating the yielder Y

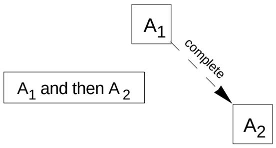
Action combinators are used to define control flow as well as to manage the movement of information between actions.

combinator : Action, Action → Action

A_1 and then A_2

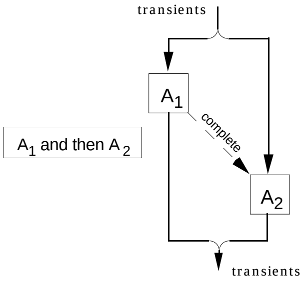
Perform the first action and then perform the second.

Dashed line shows control flow



Flow lines from the top to the bottom of the diagram show the behavior of the transients.

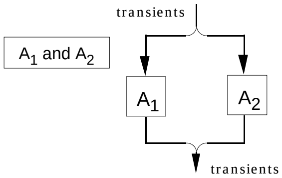
Concatenation: Join the data flow lines



A_1 and A_2

Allows the performance of the two actions to be interleaved.

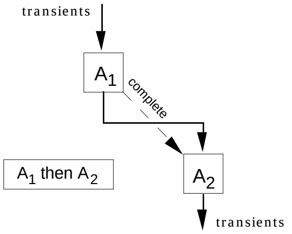
No control dependency in diagram, so actions can be performed collaterally.



A_1 then A_2

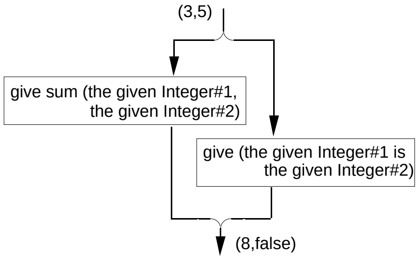
Perform the first action using the transients given to the combined action and then perform the second action using the transients given by the first action.

The transients given by the combined action are the transients given by the second action.



Example

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



Imperative Facet

Imperative facet deals with storage

- allocating memory locations
- updating the contents of locations
- fetching values from memory
- deallocating memory locations

Any action may alter the state of a cell, and such a modification remains in effect until some other action modifies the cell again.

Current storage is a finite mapping from cells to the sort (Storable | undefined).

Imperative yielder

the S stored in Y : Data, Yielder → Yielder

Yield the value of sort S stored in the cell yielded by Y.

Primitive Imperative Actions

allocate a cell

Find an unused cell, storing undefined in it, and give cell as the transient of action.

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.

Suppose that one location, denoted by cell_1, has been allocated and currently contains the value *undefined*.
Also assume that the next cell to be allocated will be cell_2.

Initial storage:

cell_1	undefined

store 77 in cell_1

cell_1	77

and then

allocate a cell

cell_1	77
cell_2	undefined

then

store 15 in the given Cell

cell_1	77
cell_2	15

and then

store product (the Integer stored in cell_1,
the Integer stored in cell_2)
in cell1

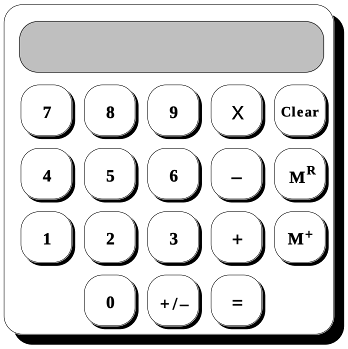
cell_1	1155
cell_2	15

Module for Imperative Features

```
module Imperative
  imports Integers, Maps
  exports
    sort Storable = Integer
    sort Storage =
              map [Cell to (Storable | undefined)]
    sort Cell
    operations
      cell1 : Cell
      allocate a cell : Action
      store _ in _ : Yielder, Yielder → Action
      the _ stored in _ : Storable, Yielder → Yielder
  end exports
  equations
  …
end Imperative
```

This module is defined to support the calculator specification that comes next.

Action Semantics of a Calculator



A Three-function Calculator

A “program” on this calculator consists of a sequence of keystrokes generally alternating between operands and operators.

6 + 33 x 2 = produces the value **78**.

Outlaw unusual combinations such as:

5 + + 6 = and **88 x +/- 11 + $\mathbf{M^R}$ $\mathbf{M^R}$**

Concrete Syntax

<program> ::= <expression sequence>

<expression sequence> ::= <expression>
| <expression> <expression sequence>

<expression> ::= <term>
| <expression> <operator> <term>
| <expression> <answer>
| <expression> <answer> **+/-**

<term> ::= <numeral> | **$\mathbf{M^R}$**
| **Clear** | <term> **+/-**

<operator> ::= **+** | **–** | **x**

<answer> ::= **$\mathbf{M^+}$** | **=**

<numeral> ::= <digit> | <numeral> <digit>

<digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Abstract Syntax

Abstract Syntactic Domains

P : Program E: Expression D: Digit

S : ExprSequence N : Numeral

Abstract Production Rules

Program ::= ExprSequence

ExprSequence ::= Expression
| Expression ExprSequence

Expression ::= Numeral | **M^R** | **Clear**
| Expression **+** Expression
| Expression **–** Expression
| Expression **x** Expression
| Expression **M^+** | Expression **=**
| Expression **+/-**

Numeral ::= Digit | Numeral Digit

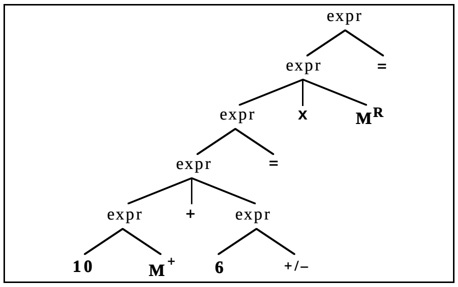
Digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Example

Following the concrete syntax for the calculator language, given the sequence of keystrokes,

10 M^+ + 6 +/- = x M^R =

a parser will construct the abstract syntax tree shown below.



An Abstract Syntax Tree

Semantic Functions

Meaning is ascribed to the calculator language via semantic functions, mostly mapping syntactic domains to actions.

meaning _ : Program → Action

perform _ : ExprSequence → Action

evaluate _ : Expression → Action

value of _ : Numeral → Integer

digit value _ : Digit → Integer

Action [outcome] [income].

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

perform_ : ExprSeq →
Action [completing | giving a Value | storing]
[using current storage]

evaluate_: Expr →
Action [completing | giving a Value | storing]
[using current storage]

Semantic Equations

1. **Numeral**

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

evaluate N = give value of N

The value given as a transient by the action give is the displayed integer. Prefix operations are evaluated from right to left, so omit the parentheses from “give (value of N)”.

2. **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 **M^R** =
give the Integer stored in cell$_1$

3. **Clear**

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

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

If interference were possible between the two activities, we could use the combinator “and then” to establish order.

4. **Addition of Two Expressions**

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

```
evaluate ⟦$E_1$ + $E_2$⟧ =
            evaluate $E_1$
        and then
            evaluate $E_2$
    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.

5. **Difference of Two Expressions**

6. **Product of Two Expressions**

These operations are handled in the same way as addition.

7. **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 cell$_1$,
                      the given Integer) in cell$_1$
        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 and combinator forms a tuple, in this case a singleton tuple, which action semantics does not distinguish from a single datum.

The primitive action “store _ in _” yields no transient, which is represented by an empty tuple.

Without the subaction “give the given Integer”, the value from E will be lost.

8. **Equal**

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

```
evaluate ⟦E =⟧ = evaluate E
```

9. **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 semantic function “meaning” initializes the calculator by storing zero in the memory location cell$_1$ and then evaluates the expression sequence.

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

“perform” evaluates the expressions in the sequence one at a time, ignoring the given transients.

```
perform ⟦E S⟧ =
        evaluate E
    and then
        perform S

perform E = evaluate E
```

“value of” and “digit value” define the meaning of integers.

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

value of D = digit value D

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

A Sample Calculation

Consider the following calculator program:

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

The first expression begins with an empty transient and with $cell_1$ containing the value 0. We show the transient 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 value of 12                                         (12)      0
        and then
            give 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 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
    give the given Integer                                         (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 value of 123                                  (123)      14
  then
      store sum (the Integer stored in cell1,
                       the given Integer) in cell1      ( )       137
    and
      give the given Integer                           (123)      137
```

This action gives the value 123 and leaves 137 in $cell_1$.

The third expression completes the evaluation, starting with 137 in memory, as follows:

```
evaluate ⟦MR +/- – 25 = + MR =⟧ =
            give the Integer stored in cell1            (137)      137
          then
            give difference (0, the given Integer)      (-137)     137
        and then
          give value of 25                              (25)       137
      then                                              (-137,25)  137
        give difference (the given Integer#1,
                         the given Integer#2)           (-162)     137
    and then
      give the 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 137 in $cell_1$.

Wren and Pelican

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

- **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

- **Unary Minus**

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

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

  Value given by evaluating expression is given as a transient to the difference operation, whose value is given as the result of the action.

- **Arithmetic on Two Expressions**

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

```
evaluate ⟦$E_1$ + $E_2$⟧ =
        evaluate $E_1$
    and
        evaluate $E_2$
then
    give sum(the given Integer#1,
                    the given Integer#2)
```

  Wren has no side effects in expressions

- **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
```

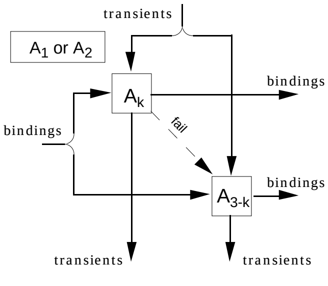
Basic Facet Actions

Deal primarily with control flow.

A_1 or A_2

Arbitrarily choose one of the subactions and perform it with given transients and received bindings.

If the chosen action fails, perform the other subaction with original transients and bindings.



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.

Acts as a guard when combined with the composite action “or”.

- **If Commands**

  Evaluate a Boolean expression and then perform **then** command or **else** command depending on the test. If the **else** part is missing, do nothing.

```
execute ⟦if E then $C_1$ else $C_2$⟧ =
      evaluate E
   then
         check (the given TruthValue is true)
               and then execute $C_1$
      or
         check (the given TruthValue is false)
               and then execute $C_2$
```

```
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
```

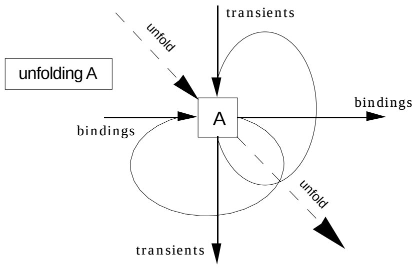
Actions needed to define the **while** command:

unfolding A

Perform the action A, but whenever the dummy action “unfold” is encountered, the action A is performed again in place of it.

unfold

A dummy action, standing for the argument action of the innermost enclosing “unfolding”.



- **While Command**

Evaluate the Boolean expression; if its value is true, execute the body of the loop and then start the **while** command again when the execution of 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
```

Declarative Facet

Deals primarily with scoped information in the form of bindings between identifiers and semantic entities such as constants, variables, and procedures.

```
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
```

Primitive declarative action

bind T to Y

produces a singleton binding mapping that we represent informally by [T |→ B] .

Declarative yielder

the S bound to T: Data, Token → Yielder

evaluates to the entity bound to the Token T provided it agrees with the sort S.

All action combinators process bindings as well as transients.

Combining bindings

merge (bindings$_1$, bindings$_2$):

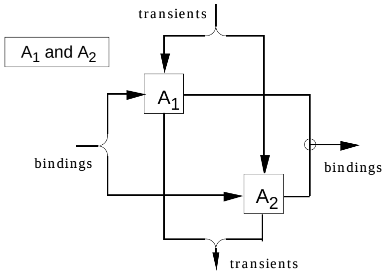
Form the (disjoint) union of the bindings so that if any identifier has bindings in both sets, the operation fails, producing nothing.

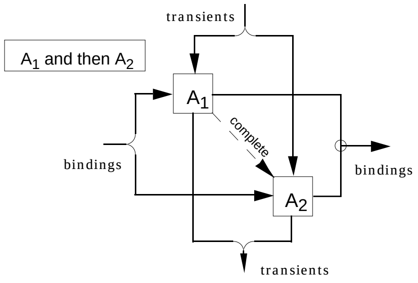
Shown by having the lines for scoped information connected by a small circle overlay (bindings$_1$, bindings$_2$):

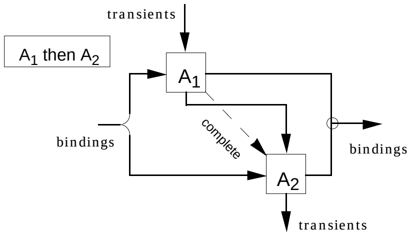
Combine bindings so that the associations in bindings$_1$ take precedence over those in bindings$_2$.

Lines show a break suggesting which set of bindings takes precedence.

In diagrams, scoped information flows from left to right whereas transients flow from top to bottom.







Declarative Facet Actions

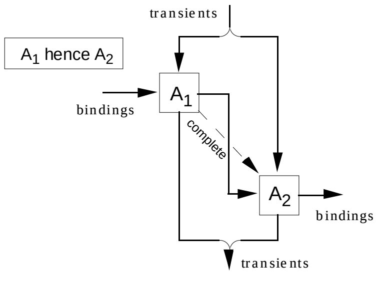
Manipulate environments.

rebind

This primitive declarative action reproduces all of the received bindings.

A$_1$ hence A$_2$

This combinator sequences the bindings and concatenaties the transients.



- **Program**

  Elaborate the declarations and execute the body of the program with the resulting bindings.

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

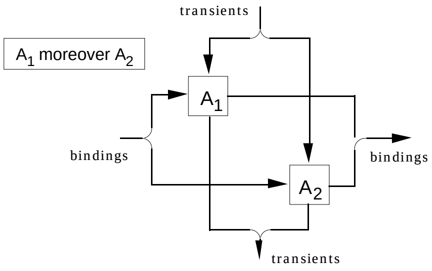
A$_1$ moreover A$_2$

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 concatenated.



- **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
```

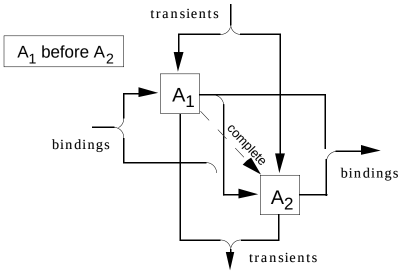
  "rebind moreover elaborate D" overlays the received bindings with the local bindings from D to provide the environment for C.

A_1 before A_2

Perform first action using transients and bindings passed to the combined action, and then perform second action using transients given to the combined action and the bindings received by the combined action overlaid by those produced by first action.

Produces the bindings produced by first action overlaid with those produced by second.

Transients are concatenated.



- **Sequence of Declarations**

  Elaborate the declarations sequentially.

```
elaborate ⟦D₁ ; D₂⟧ =
        elaborate D₁ before elaborate D₂
```

  "before" combines the bindings from the two declarations so that D_1 overlays the enclosing environment and D_2 overlays D_1, producing the combined bindings.

  Each declaration has access to the identifiers that were defined earlier in the same block as well as those in any enclosing block.

- **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
```

- **Variable Declaration**

  Allocate a cell from storage and then bind the identifier to that cell.

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

- **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
```

Visualizing Action Semantics

```
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
```

Assume that the first cell allocated is cell$_1$.

The action that elaborates the first two declarations produces the bindings

$[c \mapsto 5, n \mapsto \text{cell}_1]$.

These bindings are received by the body of the program and by the **declare** command.

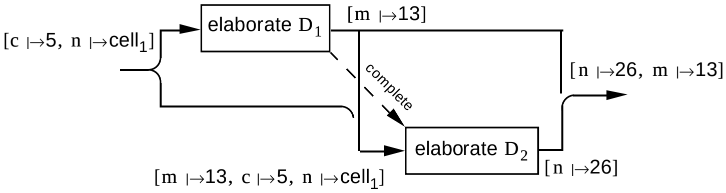
Following action models the **declare** command:

execute $\llbracket$**declare** D_1 ; D_2 **begin** C **end**$\rrbracket$ =
rebind moreover elaborate $\llbracket D_1 ; D_2 \rrbracket$
hence
execute C

First elaborate the declarations

elaborate $\llbracket D_1 ; D_2 \rrbracket$ =
elaborate D_1 before elaborate D_2.

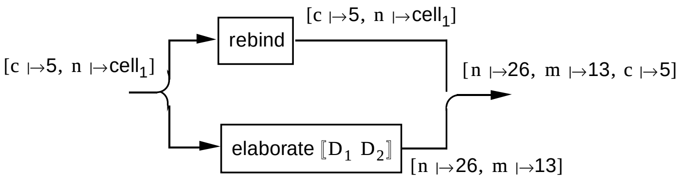
Following diagram with empty transients omitted illustrates the activities carried out by the before combinator.



The action elaborate $\llbracket D_1 ; D_2 \rrbracket$ serves as the second subaction in

rebind moreover elaborate $\llbracket D_1 ; D_2 \rrbracket$,

which is depicted in the next diagram.



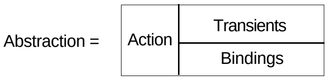
The body of the anonymous block will execute in an environment containing three bindings, $[n \mapsto 26, m \mapsto 13, c \mapsto 5]$.

Reflective Facet and Procedures

- Subprogram declaration and invocation.
- Activity of a procedure modeled by the performance of an action.

sorts Procedure = Abstraction
Bindable = Variable | Value | Procedure

Abstraction datum is 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.



Creating an Abstraction

abstraction of _ : Action → Yielder

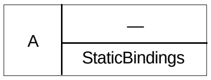
The yielder “abstraction of A” encapsulates the action A into an abstraction together with no transients and no bindings.

Transients and bindings can be supplied after the abstraction is constructed.

The current bindings are inserted into an abstraction using an operation on yielders.

closure of _ : Yielder → Yielder

Attaching the declaration-time bindings provides static scoping for resolving references to nonlocal variables.



A later performance of “closure of _” will have no effect.

Dynamic scoping ensues if bindings are attached at enaction-time (when the action in the abstraction is performed).

Execute of a procedure using a reflective action that takes as its parameter a yielder giving 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.

- **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
```

- **Procedure Call (no parameter)**

  Execute the procedure object bound to the identifier.

  execute I = enact the Procedure bound to I

  The procedure object, an abstraction, brings along its static environment.

An operation on yielders constructs an unevaluated term that incorporates the current transient 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.

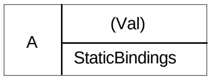
As with bindings, a second supply of transients is ignored.

- **Procedure Call (one parameter)**

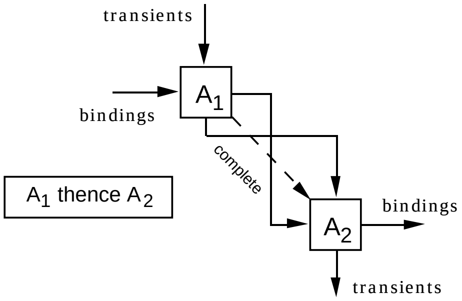
  Evaluate the actual parameter, an expression, and then execute the procedure bound to the identifier with the value of the expression.

```
execute ⟦I (E)⟧ =
    evaluate E
  then
    enact application of (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 combinator thence joins the behavior of then for transients and hence for bindings.



Declaration of Procedure (One Parameter)

The action encapsulated in an abstraction 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 overlaid by the binding of the formal parameter to a local variable, and then overlaid by local declarations.

- **Procedure Declaration (one parameter)**

  Bind 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 declaration-time environment and will allocate a local variable for the actual parameter passed to procedure.

```
elaborate ⟦procedure $I_1$ ($I_2$) is D begin C end⟧ =
  bind $I_1$ to
    closure of
      abstraction of
          allocate a cell
              and give the given Value
                   and rebind
        thence
            rebind
          moreover
              bind $I_2$ to the given Cell#1
            and
              store the given Value#2 in
                                    the given Cell#1
        hence
          rebind moreover elaborate D
        hence
          execute C
```

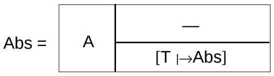
Recursive Definitions

The specifications of procedure declarations above do not allow recursive calls of procedures, since identifiers being declared are not included in bindings in abstractions created by declarations.

A hybrid action for establishing recursive bindings that is defined in terms of more primitive actions.

recursively bind _ to _ : Token, Bindable → Action

“recursively bind T to abstraction of A” produces the binding of T 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.

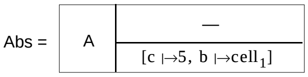
```
elaborate ⟦procedure I is D begin C end⟧ =
          recursively bind I to
             closure of
                abstraction of
                      rebind moreover elaborate D
                   hence
                      execute C
```

Example

```
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.

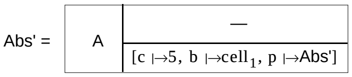
Action “closure of abstraction of A” creates the abstraction Abs, which does not allow a recursive call of procedure.



Action “bind p to closure of abstraction of A” produces the binding [p|→Abs].

Any reference to procedure identifier p inside the procedure is an illegal reference, yielding nothing.

Action “recursively bind p to closure of abstraction of A” changes abstraction Abs into a new abstraction Abs' whose attached bindings include the association of procedure abstraction with p.



The recursive action produces the binding [p|→Abs'], which when overlaid on the previous bindings, produces the bindings [c|→5,

---

b|→cell$_1$, p|→Abs'] to be received by the procedure p and the body of the program.

Translating to Action Notation

Action notation can be viewed as a metalanguage for the semantic specification of programming languages.

Semantic equations define a translator from Pelican programs into action notation.

Consider interpreting or compiling action notation.

The metalanguage of action semantics can also be used to verify semantic equivalence between language phrases.

Translate a Pelican program into its equivalent action notation.

Task is aided by the property of compositionality:
Each phrase is define solely in terms of the meaning of its immediate subphrases.

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

The overall structure of the translation takes the form

```
run ⟦program I is D1 D2 D3 D4 D5
         begin C1; C2; C3; C4 end⟧

   = elaborate ⟦D1 D2 D3 D4 D4⟧
         hence execute ⟦C1; C2; C3; C4⟧

   =  elaborate D1
         before elaborate D2
            before elaborate D3
               before elaborate D4
                  before elaborate D5
   hence
      execute C1 and then execute C2
         and then execute C3
         and then execute C4
         and then execute C5
```

The combinators and then and before are associative.

The five declarations:

```
elaborate ⟦const max = 50⟧ =
      give value of 50
   then
      bind max to the given Value


elaborate ⟦var acc : integer⟧ =
      allocate a cell
   then
      bind acc to the given Cell


elaborate ⟦var switch : boolean⟧ =
       allocate a cell
   then
      bind switch to the given Cell


elaborate ⟦var n : integer⟧ =
      allocate a cell
   then
      bind acc to the given Cell
```

```
elaborate ⟦procedure change is
            begin n := n+3;
              switch := not(switch) end⟧ =
   recursively bind change to closure of abstraction of
         rebind
       moreover
         produce empty bindings
     hence
             give Cell bound to n
           and
                 give Value stored in Cell bound to n
               or
                 give Value bound to n
             and
               give value of 3
           then
             give sum(the given Int#1,the given Int#2)
         then
           store the given Value#2 in the given Cell#1
       and then
             give Cell bound to switch
           and
                 give Value stored in
                                 Cell bound to switch
               or
                 give Value bound to switch
             then
               give not(the given Truthvalue)
         then
           store the given Value#2 in the given Cell#1
```

The four commands:

```
execute ⟦acc := 0⟧ =
          give Cell bound to acc
       and
          give value of 0
    then
       store given Value#2 in the given Cell#1


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


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

```
execute ⟦while n<=max do
    if switch then acc := acc+n end if;
                        change end while⟧ =
   unfolding
          give Value stored in Cell bound to n
        or
          give Value bound to n
      and
          give Value stored in Cell bound to max
        or
          give Value bound to max
     then
      give not(greaterthan(the given Integer#1,
                              the given Integer#2))
   then
        check given Truthvalue is true
      and then
              give Value stored in Cell bound to switch
            or
              give Value bound to switch
          then
              check given Truthvalue is true
             and then
                  give Cell bound to acc
                 and
                      give Value stored in
                                Cell bound to acc
                     or
                      give Value bound to acc
                    and
```

```
                      give Value stored in Cell bound to
                     or
                      give Value bound to n
                  then
                   give sum(the given Int#1,
                                the given Int#2)
               then
                 store the given Value#2 in
                                the given Cell#1
           or
              check given Truthvalue is false
             and then
              complete
         and then
           enact Procedure bound to change
       and then unfold
     or
        check given Truthvalue is false
                  and then complete
```

LABORATORY

Translator from Pelican to action notation using Prolog.

The compositional definitions of the meaning of Pelican convert to Prolog clauses directly.

The resulting action can be represented as a Prolog structure by viewing actions, yielders, and auxilliary opertions with prefix syntax.

```
>>> Translating Pelican into Action Semantics  <<<
Enter name of source file: small.wren
     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(Value))),
         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
```

Translation is a static operation

Need not be concerned with stores and environments—these are handled when action notation is interpreted or compiled.

We have dispensed with the syntactic category of blocks to match the specification in Figure 13.6.

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

Build Prolog structures that represent the resulting action using calls to the predicates elaborate and execute to construct pieces of the structure.

Declarations

```
elaborate([],produce(emptybindings)).

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

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

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).
```

Commands

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

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

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

Expressions

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

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

evaluate(minus(E),
        then(EvaluteE,
                give(sum(given('Integer',1),
                            given('Integer',2))))) :-
            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).
```