t
Semantics *608*

We specify the semantics of Prograph by defining execution functions for executable elements, which are externals, methods, cases, and operations. Running a program corresponds to evaluating the execution function for some method of arity (0,0). Such a method is called a query.

If Y is an executable element with arity (m,k), an execution function of Y is a function fY that maps each m-tuple of values v to a pair (w,e), where e is one of next case, terminate, finish, continue, fail, or error, and w is either a k-tuple of values or the special result nil. We call w and e the result and message of fY(v) respectively. For example, if the element in question is an operation, the tuple v consists of the values that arrive at the terminals, the tuple w consists of the values that appear at the roots of the operation after it has executed, and e conveys information about any control conditions that occur as a result of execution. If executing the operation produces an error (for example, a primitive receiving inputs of an inappropriate type), the message is error and the result is nil.

In general, in any of the following definitions of executions of Prograph elements, if the element X and value tuple v do not satisfy any of the conditions for which fX(v) is defined, then fX(v) is defined to be (nil, error). Note that nil is not a Prograph data type and has only been introduced here to streamline the presentation of Prograph semantics.

It is important to note that because the operations of a case can generally be ordered in several ways, cases, and consequently methods and operations, do not have unique execution functions.

In the definition of the execution functions, we have sacrificed some formality for the sake of readability. In particular, certain execution functions are not functions in the mathematical sense, since their application causes side effects by altering the values of attributes or persistents. The description could, of course, be made strictly correct by adding an “environment” parameter to each function and an “environment” component to the function value for keeping track of attribute and persistent values.

The environment would be affected by those execution functions that alter such values. The price of such precision, however, would be more complex descriptions of both syntax and semantics, so we simply note these side effects in the function definitions. When an execution function is defined in terms of other functions that have side effects, the function being defined also entails the side effects.

External *609*

An external X has an execution function fX of the same arity. For any valid v, the message of fX(v) is continue, fail, or error if X is a primitive, and continue or error if X is a Mac Method. Primitives perform certain standard tasks such as addition, while Mac Methods call Macintosh toolbox routines. External execution functions are described in chapter 6, “Prograph Primitives,” and in Inside Macintosh.

Method *609*

Let M be a method with inarity m and a sequence (C[1],C[2]...,C[n]) of cases. Let fC[1],..., fC[n] be the execution functions for these cases, and v an m-tuple of values. We define fM, an execution function for M, by fM(v) = fC[k](v), where k is the smallest positive integer such that the message of fC[k](v) is not next case.
Note that if such k does not exist, then, according to the definition at the beginning of this section, fM(v) is (nil, error).

NOTE: This definition of execution function for a method captures the behavior discussed in other parts of this manual. The cases are executed in the order in which they occur until one of them produces a message other than next case. The message produced by the last executed case then becomes the message produced by execution of the method and can be used to control execution in the case containing the operation that called it. Exactly how the message is used depends on the calling operation: for example, if the operation is a multiplex, a terminate message stops iteration. The effects of messages are discussed further below.

Case "610"

If C is a case, an ordering (O[1],O[2],...,O[n]) of the operations of C is said to be executable if and only if it satisfies the following two conditions:

o Every operation in the ordering precedes (in the strict sense defined above) all those that follow it in the ordering.

o If O[i] is the first operation in the ordering with a control other than Continue on Success, then every operation that occurs in the ordering before O[i] precedes O[i] (in the sense defined above).

In the following, let C be a case with inarity m, (O[1],O[2],...,O[n]) an executable ordering of the operations of C, fO[1],...,fO[n] execution functions for these operations, and v an m-tuple of values.

Execution Sequence "610"

The messages produced by operation execution functions are used to set a flag in the case. This flag is set depending on the relative strengths of the execution messages produced by the operations. We define a partial ordering stronger than on execution messages as follows.

- finish stronger than continue
- terminate stronger than finish
- fail stronger than finish
- error stronger than finish
- next case stronger than finish

The execution sequence for C with v induced by the given ordering and operation execution functions is a sequence ((v1,w1,flag1), (v2,w2,flag2),..., (vk,wk,flagk)), where k is less than or equal to n, defined as follows.

First Operation (or Input) "611"

(v1,w1,flag1) = ((),v, continue)

Rest of the Sequence

Suppose all members of the sequence are defined up to (vi,wi,flagi). If i equals n, or flagi is terminate next case, fail, or error, then k=i and the sequence is finished. Otherwise, we define the next member (vi+1,wi+1,flagi+1) as follows.

o wi+1 is the result of fO[i+1](vi+1).

o
vi+1 is a pi+1-tuple of values where pi+1 is the inarity of O[i+1], and for each j from 1 to pi+1:
•

vi+1[j] = wx[r], if there is a datalink from the rth root of O[x] to the j-th terminal of O[i+1],

•
vi+1[j] = NONE otherwise.

o flagi+1 is the stronger of flagi and the message of fO[i+1](vi+1).

Note that because the execution sequence follows an executable ordering, for each i less than k, flagi is either finish or continue, so in the third part of the definition, since x < i, wx is a tuple of values. Also, since the first member of the execution sequence for a case is defined without reference to the input bar, it is not necessary for Input operations to have an execution function. Note as well that if a terminal of an operation has no datalink attached, then the corresponding input value is NONE.

_____

NOTE: An execution sequence captures the essential details of a case execution. Its order depends on two things. First, input data must be available before an operation can execute, and the order dictated by synchro links must be respected. Second, if some operation has a control that could cause execution of the case to stop, then such an operation should be executed as early as possible. When there is more than one operation with such a significant control, the programmer should use synchro links if the execution order of these operations is important.

Each member of an execution sequence contains the important information about execution of an operation: namely, the input data it receives, the result values it produces, and the control message it generates. If an operation produces the message finish, this message propagates through the remaining members of the sequence unless a subsequent operation produces a stronger message that ends the sequence completely.

Execution Function for a Case "612"

If C is a case with arity (m,p), we define an execution function for C with respect to a particular executable ordering and set of operation execution functions, as follows.
Let v be an m-tuple of values, and let ((v1,w1,flag1), (v2,w2,flag2),..., (vk,wk,flagk)) be the execution sequence for C induced by the chosen ordering and operation execution functions. Then:

o fC(v) = (w,flagk) if flagk is terminate, fail, next case, or error, where w is a p-tuple, every component of

which is the value UNDEFINED,

o fC(v) = (wk,flagk) if flagi=continue for each i from 1 to k-1,

o fC(v) = (wk,finish) otherwise.

Note that the second or third alternatives only take place if k=n, that is, if all the operations of the case have executed.

Operations in General [612]

Controls

If Control is a control and e is an execution message, we denote by Control(e) the execution message defined by the following table, where the column labeled “successful” applies to any of the execution messages finish, terminate, or continue.

e (execution message)
Control
successful
fail error

Next Case on Success
next case
continue*
error

Next Case on Failure
continue
next case
error

Continue on Success
continue
error
error

Continue on Failure
continue
continue*
error

Finish on Success
finish
continue*
error

Finish on Failure
continue
finish*
error

Terminate on Success
terminate
continue*
error

Terminate on Failure
continue
terminate
error

Fail on Success
fail
continue*
error

Fail on Failure
continue
fail
error

* Error if outarity ≠ 0

NOTE: In Prograph, the execution of operations can succeed or fail, providing information to be used for execution control. There is no direct analogy to this mechanism in Pascal, C, Smalltalk, or Lisp. Success and failure are used as execution controls in Prolog, but in Prograph their use is much more refined. First, in Prolog, no distinction is made between failure intentionally used as a control by the programmer and failure resulting from a programming error. In Prograph, however, if execution of an "uncontrolled" operation (that is, one with the default control Continue on Success) produces the message fail, this message is translated into error. Second, in Prolog, failure simply causes backtracking, whereas in Prograph controls provide several actions, triggered either on success or failure, and used to control conditional execution (as described in the execution of cases, above) or iteration (described below in execution of operations with mode Repeat).

The above table describes the way in which the control associated with an operation transforms messages passed back by the mechanism the operation calls, thereby propagating control information to be acted on by the case in which the operation occurs, as described above in the execution of cases.

Method Selection "614"

A method M is said to be applicable to a class C1 if and only if either M is a method of C1, or M is a method of some ancestor C2 of C1 and there is no method of the same kind as M (that is, Set, Get or Initialization) in a subclass of C2 that is not also a subclass of C1. If C1 is a subclass of C2, C1 has a method M1, and there is a method M2 applicable to C2, where M1 and M2 have the same name and are of the same kind, then M1 is said to overshadow M2.

Runtime Name Determination "614"

A string X is said to contract to an identifier Y if X has one of the forms Y, <space>Y, Y<space> or <space>Y<space>, where <space> is defined in the “Values” section in this appendix.

If the name of an operation O is an inject terminal, then for every value tuple v, if the component of v corresponding to the inject terminal is a string that contracts to an identifier N' that is not the name of a Mac Method, then we define fO(v) = fO'(v') where O' is the operation obtained by changing the name of O to N' and removing the inject terminal from the sequence of terminals, and v' is the tuple of values obtained by removing from v the value corresponding to the inject terminal.

NOTE: The identity of an operation with an inject terminal is determined at execution time from the value that arrives on the terminal. The value must be a string. This mechanism has no analogy in Pascal or C. In Lisp , however, a function can be a variable that becomes bound at runtime, and in Prolog, variables can similarly be used as predicates in goal literals.

Operations in Plain mode /3

In the following function definitions, the length of each value tuple used as an argument to a function is assumed to be equal to the inarity of the function.

In the rest of this section, we deal only with operations without inject terminals. In the following we define execution functions for various kinds of operations. The symbol O is used to denote an operation.

Simple Operations "614"

For every tuple of values v, if there exists an external or method X as defined below, and X has the same arity as O, then fO(v) = (w,Control(e)), where fX(v) = (w,e).

Universal Reference "615"

If Name is a nonempty universal reference, then:

o If Name is the name of an external, X is that external.

o Otherwise, X is the universal method named Name.

Context-determined Reference

If Name is a context-determined reference //Z, then:

o If Search= Super, then X is the method named Z applicable to the superclass of the method of the case of O.

o Otherwise, X is the method named Z applicable to the class of the method of the case of O.

Explicit Reference "615"

If Name is an explicit reference C/Z and there exists a class C, X is the method named Z applicable to class C.

Data-determined Reference

If Name is a data-determined reference /Z, then:

o If v[1] is an instance of a class C, X is the method named Z applicable to class C.

o Otherwise, if Z is the name of an external, X is that external,

o Otherwise, X is the universal method named Z.

NOTE: A Simple operation calls a primitive, Mac Method, or user-defined method, that is either in a class or is universal. An operation with a data-determined reference as its name corresponds to a message passed to some object in Smalltalk. In Prograph the message is the name of the operation, while the object is the instance input to the leftmost terminal. If the leftmost terminal does not receive an instance, however, a primitive, Mac Method, or universal method is called. An operation with a context-determined or explicit reference as its name calls a method applicable to a specific class, regardless of the type of the incoming data.

It is important to note that if there is a universal method with the same name as an external, an operation with this name calls the external rather than the method.
If the search origin of an operation is Super, the method called is the one applicable to the superclass of the class containing the operation. This is a standard mechanism in object-oriented languages for allowing the programmer to write a new method M1 in a class C that overshadows a method M2 previously applicable to class C, where M1 contains an operation that calls M2. This operation can be made Super so that it does not recursively call M1.

Get "616"

Let v be a 1-tuple, say (u), in the following.

Universal Reference

If N is a nonempty universal reference, then:

o If u is an instance with an attribute A named Name, then fO(v) = ((u,w),Control(continue)), where w is

the value of A.

o If u is a string that contracts to the name X of a class C with an attribute A named Name, then fO(v) = ((X,w),Control(continue)), where w is the value of A.

Data-determined Reference *616*

If Name is a data-determined reference /Z, then:

o If u is an instance of a class C, then:

•
If a Get method M named Z is applicable to C, then fO(v) = ((u,w),Control(e)), where fM(v) = (w,e).

•
Otherwise, if u has an attribute A named Z, or C has a class attribute A named Z, then fO(v) = ((u,w),Control(continue)), where w is the value of A.

o If u is a string that contracts to the name X of a class C, then
•
If a Get method M named Z is applicable to C, then fO(v) = (w,Control(e)), where fM(v) = (w,e).

•
Otherwise, if C has an attribute A named Z, then fO(v) = ((X,w),Control(continue)), where w is the value of A.

_____
NOTE: A Get is used for retrieving the value of an attribute. Given a string that names a class, it finds the default value of an attribute of that class, which can be a class or instance attribute. Given an instance, it returns the value of an instance attribute of that instance, or class attribute of the class to which the given instance belongs.

This behavior is modified if the name of the Get is a data-determined reference. It then either calls a Get method applicable to the class, or if no appropriate Get method exists, defaults to the behavior described above.

Set *617*

For every pair of values v:

Universal Reference

If Name is a nonempty universal reference, then:

o If v[1] is an instance with an attribute A named Name, then fO(v) = ((v[1]),Control(continue)), and the value of A is changed to v[2] (see note below).

o If v[1] is a string that contracts to the name X of a class C with an attribute A named Name, then fO(v) = ((X),Control(continue)), and the value of A is changed to v[2] (see note below).

Data-determined Reference

If Name is a data-determined reference /Z, then:

o If v[1] is an instance of a class C

•
If v[1] is an instance of a class C to which a Set method named Z is applicable, then fO(v) = (w,Control(e)), where fZ(v) = (w,e).

•
Otherwise, if either v has an attribute A named Z or C has a class attribute A named Z, then fO(v) = ((v[1]),Control(continue)), and the value of A is changed to v[2] (see note below).

o If v[1] is a string that contracts to the name X of a class C

•
If v[1] is a string that contracts to the name X of a class C to which a Set method named Z is applicable, then fO(v) = (w,Control(e)), where fZ(v) = (w,e).

•
Otherwise, if C has an attribute A named Z, then fO(v) = ((X),Control(continue)), and the value of A is changed to v[2] (see note below).

Note, however, that if A in the above is a system attribute, the message of fO(v) is error if A is not settable, or if v[2] is not an allowable value for A. For details, see chapter 5, “System Classes.”

_____
NOTE: A Set is used for assigning a value to an attribute. Given a string that names a class, it sets the default value of an attribute of that class, which can be a class or instance attribute. Given an instance, it sets the value of an instance attribute of that instance, or class attribute of the class to which the given instance belongs.This behavior is modified if the name of the Set is a data-determined reference. It then either calls a Set method applicable to the class, or if no appropriate Set method exists, defaults to the behavior described above.

Instance Generator "618"

If Name is the name of a class C, and v is any 1-tuple of values:

Without Initialization Method

If no Initialization method is applicable to class C, then:

o If v[1] = NONE, then fO(v) = ((w), Control(continue)) where w is an instance of the class C in which every attribute has the same value as the corresponding attribute of C.

o If v[1] is a list of pairs of the form ((x1,y1) ... (xn,yn)) where x1,...,xn are strings that contract to the names N1,...,Nn of attributes of C and y1,...,yn are values, then the Instance generator is semantically

equivalent to a Local operation in Repeat mode as illustrated by the following diagrams. The Instance generator on the left is semantically equivalent to the Local on the right with the same input, with Local method as shown below. This correspondence is trivially extended to accommodate an inject terminal.

ith Initialization Method *619*

If class C has an applicable Initialization method M, let O' be the operation obtained by replacing the control of O by Continue on Success, and let fO' be an execution function that would correspond to O' if the Initialization M was deleted from class C. We define fO(v) = (u,Control(e)), where fM(w) = (u,e) and fO'(v) = (w, continue).

NOTE: An Instance generator manufactures a new instance of a class, using the current default values for attributes. These default values can be overridden by giving a list of attribute names and values as input. If, in this input, a class attribute is named, the value of this class attribute is changed.

This behavior is modified if an Initialization method is applicable to the class. The instance is created as described above, and then the Initialization is called with the new instance as input.

Persistent Operation *619*

If w is any value and O is a Persistent with the name Name, then:

o If arity is (1,1), fO((w)) = ((w),Control(continue)), and value of O is changed to w.

o If arity is (0,1), fO(()) = ((x),Control(continue)), where x is the value of O.

o If arity is (1,0), fO((w)) = ((),Control(continue)), and value of O is changed to w.

o If arity is (0,0), fO(()) = ((),Control(continue)).

Local Operation "620"

Let M be a method such that the sequence of cases of M is the same as the name of O. fO(v) is defined as (w,Control(e)) where fM(v) = (w,e).

We remind the reader that, as defined in the “Syntax” section, above in this appendix, the name of a Local is a sequence of cases.

NOTE: A Local functions exactly like a call to a method, except that the Local method is directly associated (without using the name as an identifier) with the Local operation.

Evaluation "620"

If v is a tuple of values such that every component of v corresponding to a terminal of O that corresponds to a single letter operand of the expression Name is an <integer> or <real> (see the “Values” section above in this appendix), then fO(v) = (w,Control(continue)), where w is the result of evaluating the expression Name with single letter operands replaced by the corresponding components of v.

Evaluation of the expression follows the operator precedence and association inherent in the grammar given above defining the name of an Evaluation.

Output "620"

For every tuple v of values fO(v) = ((),Control(continue)).

NOTE: In normal circumstances, when the Control is continue on success or finish on success, execution of the Output completes execution of the case as well as of the method, and passes the values from its terminals to the roots of the calling operation.

Constant "620"

fO(()) = ((Name),continue).

NOTE: A Constant always succeeds and passes its name as a value to its root.

Match "621"

For every value w, fO((w)) = ((),Control(e)), where e is:

o
continue if w and Name are equal in the sense defined by the primitive = (see chapter 6, “Prograph Primitives”), and

o
fail otherwise.

Mac Constant "621"

fO(()) = ((w),continue), where w is the value of the Macintosh constant named Name.

Mac Match "621"

For every value w, fO((w)) = ((),Control(e)), where e is:

o
continue if the value of the Macintosh constant named Name is equal to w in the sense defined by the primitive = (see chapter 6, “Prograph Primitives”), and

o
fail otherwise.

Mac Global "621"

The execution function is analogous to that for a Persistent, except that if the inarity of O is 1 the message of fO((w)) is error if w is not an allowable value for the global (see Inside Macintosh).

Mac Get Field "621"

If Name is the name of a field of some Macintosh structure, and w is a <simple mac> or a Macintosh value, then fO((w)) = ((z,x),Control(continue)), where:

o
z=w if w is an indirect Macintosh value, and

o
z is a copy of w otherwise;

and x is the value represented by the sequence of bytes of the size corresponding to the field Name at the offset from the location of w specified for field Name. In particular, if Name is the name of a field of w, then x is the value of this field.

Mac Set Field "622"

If Name is the name of a field of some Macintosh structure, v is a pair such that v[1] is a <simple mac> or a Macintosh value, and v[2] is an allowable value for field Name, then fO(v) = ((w),Control(continue)), where:

o w=v[1] if v[1] is an indirect Macintosh value, and

o
w is a copy of v[1] otherwise.

The value of the sequence of bytes of the size corresponding to the field Name at the offset from the location of w specified for field Name is changed to v[2]. In particular, if Name is the name of a field of w, then the value of this field is changed to v[2].

Mac Get Address "622"

If w is a <simple mac> or a Macintosh value, and Name is the name of a field of some Macintosh structure, then fO((w)) = ((x),Control(continue)), where x is an address computed by adding to the location of w the offset specified for field Name.

Operations in Repeat Mode without True or False Roots

Let O be a Repeat operation with arity (m,k) and control Control. Let T and R be the terminals and roots respectively of O. Let v be an m-tuple of values such that for every i from 1 to m v[i] is a list if T[i] is a List terminal. Let O' be the operation obtained by changing the mode of O to Plain, and the control of O to the special control Void, defined by Void(e) = e for any message e.

Input and Output Sequences "622"

We define two sequences called the input sequence and output sequence for O with v, namely v1,v2, . . . and w1,w2, . . . respectively, where for each i, vi and wi are an m-tuple and k-tuple respectively. These sequences are defined as follows.

First Element of Input Sequence

For each j between 1 and m inclusive:

o If T[j] is Loop or Simple, v1[j] = v[j].

o If T[j] is a List, then v1[j] is the first component of v[j]. (Recall that v[j] is a list.)

Output Sequence and Rest of Input Sequence

Suppose all members of the input sequence are defined up to vi, then:

o wi is the result of fO'(vi).

o If the message fO'(vi) is:

•
finish, terminate, fail, or error, or

•
if there is a j such that T[j] is a List and the length of the list v[j] is i, then generation of both sequences stops. In this event we define the stopping index t to be i if the message was finish and i-1 if it was terminate or fail. We also define the final message to be fail if fO'(vi) is fail and continue otherwise.

o Otherwise for each j ranging from 1 to m:

• If T[j] is Simple, then vi[j] = v[j].

• If T[j]) is a Loop, then vi+1[j] = wi[r] where R(r) is the corresponding Loop root for some r between 1 and k.
•

If T[j] is a List, then vi[j] is the i-th component of v[j] (recall that v[j] is a list).

NOTE: These execution sequences chronicle the execution of a multiplex in the same way as an execution sequence for a case records the details of the execution of a case. The input and output sequences are values of terminals and roots that are actually produced internally by the interpreter, even though the user never sees them.

The value input to each terminal on each iteration depends on the kind of terminal. If it is Simple, the value is the initial value input to the terminal before iteration begins. If it is a Loop, the value for the first iteration is the initial value input to the terminal before iteration begins, and otherwise is the value output on the corresponding Loop root in the previous iteration. If it is a List, the initial value input to the terminal must be a list, and the members of this list are input to successive iterations. Note that a message to stop execution can be generated in two ways. It can be explicitly produced by an execution of the operation, or implicitly, when one of the lists input to a List terminal is exhausted.

Execution Function for Repeat "624"

If t is the stopping index and e is the final message, we define fO(v) = (w,Control(e)) where w is defined as follows.

For each r ranging from 1 to k:

Simple Roots

If R[r] is a Simple root, then:

o
w[r] = UNDEFINED if t = 0.

o
w[r] = wt[r] otherwise.

Loop Roots

If R[r] is a Loop root, then:

o
w[r] = v[j] if t = 0, where T[j] is the corresponding Loop terminal.

o

w[r] = wt[r] otherwise.

List Roots

If R[r] is a List root, then w[r] is obtained by removing all components equal to NONE from (w1[r] ... wt[r]).

_____

NOTE: The value produced by the multiplex on a List root is the list of all values for that root from the output sequence, with any occurrences of NONE deleted. The value on a Simple or Loop root is the value for that root in the member of the output sequence corresponding to the stopping index. Note that the stopping index can be 0. If this happens, the value of each List root is the empty list, the value of each Simple root is UNDEFINED, and the value of each Loop root is the initial value of the corresponding Loop terminal. Note that if the sequences do not terminate or terminate because the message error was produced, then according to the convention stated in the beginning of this section, fO(v) = (nil, error).

Prograph's multiplexes correspond to various iterative constructs in other languages, but provide far greater flexibility, since various stopping conditions can be combined and intermixed with computations that, in other languages, constitute the loop body. A multiplex with no List terminals or roots corresponds with while-do and do-until in Pascal and C, and to Lisp loop constructs. Prolog has no directly corresponding mechanism; however, loops of a restricted sort can be simulated by tail recursion. Multiplexes with List terminals and roots but no Loop terminals are analogous to the MAPCAR function of Lisp but have no equivalent in Pascal, C, Prolog, or Smalltalk.

Partition Operations

Every operation with True and False roots is semantically equivalent to a Local operation as illustrated by the following diagrams. The operation on the left, annotated as Partition, is equivalent to the Local on the right, the Local method of which is shown below.

his transformation can be trivially generalized to an operation with any number of Simple terminals or with an inject terminal.