Appendix IV

Syntax and Semantics *593*

t Definitions and Notation
t Syntax
t Semantics - See next section

In this appendix, the syntax and semantics of Prograph are presented, as far as is practical, in an abstract form, free from representation issues. In a particular implementation, the entities that constitute a Prograph program are represented by symbols or pictures. We suggest that while following the description of syntax below, the reader consult chapter 8, “Pictopedia,” to match syntactic elements with their appearance in the Macintosh implementation.

As is the case with any programming language, some low-level details depend on the implementation environment, so at a certain point our description becomes more concrete, to deal with these aspects of the Macintosh implementation. Generally, we refer the reader to other chapters of this manual as appropriate.

Although the Prograph language is extremely powerful, the simple pictorial interface that makes this power readily accessible also disguises the complexity of the underlying concepts. Natural languages such as English are not designed for describing such things. Their punctuation and syntax cannot stretch to expressing complicated conditions, and the nomenclature they provide, such as the articles “the” and “a,” is not up to the task of unambiguously identifying all the items necessary to an explanation. So in order to precisely describe these concepts and to ensure that the structure and function of Prograph programs are clearly and unambiguously defined, we need compact notation and terminology that cannot be misunderstood. The material in this chapter may therefore appear terse and mathematical; however, it is all defined here, so no prior knowledge of any specific mathematical concepts is required. Time invested in appreciating these definitions will be well repaid later with a clear understanding of what Prograph is and what it does.

Wherever possible, the concepts described are related to corresponding concepts in other languages or are accompanied by informal explanatory comments.

t
Definitions and Notation *594*

Certain entities in a program are referred to as elements: namely classes, attributes, methods, cases, operations, roots, terminals and persistents.

The word “tuple” is used to mean an ordered set of (not necessarily distinct) items, its standard meaning in mathematics. It can be preceded by an integer indicating the size: for example, (), (dog), (old, new), (1, 2, 3) are respectively, a 0-tuple, 1-tuple, 2-tuple, and 3-tuple. The words “pair” and “triple” are used as synonyms for “2-tuple” and “3-tuple,” respectively. If x is either a tuple or a Prograph list, |x| denotes the size of x, and if i is an integer ranging from 1 to |x|, x[i] denotes the member of x in position i. The word “sequence” is used as a synonym for “tuple.”

Prograph programs and certain elements consist of (possibly nested) sets of elements. No element is a

member of more than one such set, so we can, without ambiguity, use phrases of the form “the X of a Y,” where Y is a member of some set that is a constituent of an element X: for instance “the method of a case.”

All elements except roots, terminals, and operations Input and Output have names. A set of named elements is said to be unambiguous if and only if no two elements in the set have the same nonempty string as their names.

Certain elements have two associated integers, the inarity and outarity. If m and k are the inarity and outarity of an element, respectively, then the pair (m,k) is called the arity of the element.

An identifier is any string of characters that neither begins nor ends with <space>, where <space> is as defined in the “Values” section below in this appendix.

A reference is on of the following: *594*

o an explicit reference, which is a string of the form <class>/<method>, where <class> is a nonempty identifier not containing the character /, and <method> is a nonempty identifier

o a context-determined reference, which is a string of the form //<method>, where <method> is a nonempty identifier

o a data-determined reference, which is a string of the form /<method>, where <method> is a nonempty identifier

o a universal reference, which is any identifier not included in the above three definitions

t
Syntax *595*

Values

A value is either a simple value or a compound value.

A simple value is a string of characters defined by the following grammar in Backus-Naur Form, where nonterminal symbols defining classes of strings are enclosed in < and >, and | separates alternative productions. For some nonterminal symbols we give a descriptive definition if a formal one would be obscure or tedious.

<simple value> : : =
<delimited value> | <default string>

<string> : : =
<quoted string> | <atomic string> | <default string>

<default string> : : =
any string which is not a <delimited value>

<delimited value> : : =
<quoted string> | <atomic string> | <integer> | <real> |
<boolean> | <null> | <undefined> | <none> | <list> | <simple mac>

<atomic string> : : =
any <atom> other than FALSE, TRUE, NULL, UNDEFINED, or
NONE

<boolean> : : =
FALSE | TRUE

<null> : : =
NULL

<undefined> : : =
UNDEFINED

<none> : : =
NONE

<list> : : =
(<value*>)

<value*> : : =
<delimited value> <space><value*> | <delimited value> | <empty>

<simple mac> : : =
<point> | <rectangle> | <RGB>

<point> : : =
<integer> <space> <integer>

<rectangle> : : =
<integer> <space> <integer> <space> <integer> <space> <integer> "596"

<RGB> : : =
<integer> <space> <integer> <space> <integer>

<space> : : =
any nonempty string of characters whose ASCII values are less than or
equal to that of blank

<quoted string> : : =
"<no double>"<quoted string> | "<no double>"

<no double> : : =
any string of characters, except "

<atom> : : =
<letter atom> | <special atom>

<letter atom> : : =
<letter> <alphanumeric> | <letter>

<special atom> : : =

any nonempty string of characters having ASCII values greater
than that of blank except those included in <alphanumeric>

<integer> : : =
<sign> <unsigned integer> "596"

<unsigned integer> : : =
<decimal integer> | <based integer> | <ascii integer>

<decimal integer> : : =
<digit> <spaced digits>

<spaced digits> : : =
<digit> <spaced digits> | _<spaced digits> | <empty>

<based integer> : : =
<digit+>#<alphanumeric>

<alphanumeric> : : =
<letter> <alphanumeric> | <digit> <alphanumeric> | <letter> |
<digit>

<letter> : : =
a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I |
J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | _

<ascii integer> : : =
'<no single>' | '<no single><no single>' | '<no single><no
single><no single><no single>'

<no single> : : =
any string of characters having ASCII values greater than that of
blank, except ' | ' '

<real> : : =
<sign> <mantissa> <exponent>

<mantissa> : : =
<digit+> | <digit+>.<digit*> | <digit*>.<digit+>

<exponent> : : =
e<sign> <digit+> | E<sign> <digit+> | <empty> "596"

<sign> : : =
+ | - | <empty> "597"

<digit*> : : =
<digit+> | <empty>

<digit+> : : =
<digit> <digit*>

<digit> : : =

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<empty> : : =
the empty string

A compound value is a compound list, a direct Macintosh value, an indirect Macintosh value, or an instance of a class.

A compound list is a tuple of values, at least one of which is compound.

Macintosh values are directly or indirectly referenced Macintosh records. These are specific to the implementation environment and are not discussed here. See Inside Macintosh for details.

An instance D of a class C is a set of attributes in one-to-one correspondence with the set of instance attributes of C, such that corresponding attributes from D and C have the same name (see the definition of “class” below).

The primitive data types in Prograph are the following: boolean, integer, list, none, null, real, string, and undefined (see appendix I,“ Prograph Data Types”). *597*

Program *597*

A Prograph program consists of:

o an unambiguous set of externals, partitioned into a set of primitives and a set of Mac Methods

o an unambiguous set of persistents

o a class hierarchy, consisting of an unambiguous set C of classes together with a set I of pairs of classes in C with the property that if (c1,c2) and (c3,c2) are both in I, then c1= c3

An element (c1,c2) of I is called an inheritance link from c1 to c2; c1 is called the superclass of c2; and c2 is called a subclass of c1. If c1 and c2 are classes, c1 is called an ancestor of c2 if and only if either c1 is the superclass of c2 , or c1 is an ancestor of some class c3 that is the superclass of c2 . If c1 is an ancestor of c2, then we say that c2 is a descendent of c1, and c2 inherits from c1.

o an unambiguous set of methods, called the universal methods of the program

External *598*

An external consists of an identifier called the name of the external, and two nonnegative integers called the inarity and outarity respectively.

NOTE: Externals provide certain basic functions, and correspond to operators and built in procedures and functions of languages such as C or Pascal. For example, the primitive + corresponds with the C or Pascal arithmetic operator +. Mac Methods provide access to the Macintosh toolbox, like most language implementations on the Macintosh.

Persistent *598*

A persistent consists of an identifier and a value, respectively called the name and value of the persistent.

NOTE: A Prograph persistent plays the same role as global variables in such languages as C and Pascal. It provides a convenient “pigeonhole” to keep values for later use. The difference, however, is that the value of a persistent is retained
between executions and is saved with the program.

Class *598*

A class consists of:

o an identifier, called the name of the class, which cannot be the name of any primitive data type or Macintosh type, as listed in the Info menu

o an unambiguous set of attributes, partitioned into two sets, class attributes and instance attributes

•
An attribute y of class Y is inherited if and only if it has the same name as some attribute x of the superclass of Y: y is also said to inherit from x. An attribute that is not inherited is said to be locally defined.

•
An attribute y of class Y descends from an attribute x of class X if and only if X is an ancestor of Y, x and y have the same name, and x is not inherited.

•
If an attribute y inherits from an attribute x, then y is a class attribute if and only if x is a class attribute.

•
If Y is a subclass of X, then for every attribute x of X there is an attribute y of Y that inherits from x.

o four unambiguous sets of methods called, respectively, the Simple, Set, Get, and Initialization methods of the class

•
There is at most one Initialization method, which has name <<>> and arity (1,1).

•
The arity of each Set method is (2,1).

•
The arity of each Get method is (1,2).

NOTE: Class hierarchies in Prograph are exactly comparable to class hierarchies in any other object-oriented language with single inheritance. A hierarchy of classes forms a “forest of trees,” in which

methods and attributes are inherited from the roots of the trees to the leaves.

Attribute "599"

An attribute consists of an identifier and a value, respectively called the name and value of the attribute.

Method "599"

A method consists of an identifier, called the name of the method, and a sequence of cases all with the same arity. A method has an associated arity, which is the same as the arity of each of its cases.

NOTE: Prograph methods are analogous to Smalltalk methods, Pascal procedures, functions in Lisp or C, and predicates in Prolog.

Case "599"

A case consists of:

o A set of operations containing exactly one Input and exactly one Output, called the input bar and output bar respectively.

The inarity and outarity of the case are, respectively, the outarity of the input bar and the inarity of the output bar.

No operation precedes itself, where “precedes” is defined below.

o A set of datalinks, each of which is a pair of the form (r,t) where r and t are, respectively, a root and a terminal of some operations of the case. A datalink (r,t) is said to be from r to t. No terminal occurs in more than one datalink.

o A set of synchro links, each of which is a pair of operations of the case. A synchro link (o1,o2) is said to be from o1 to o2. Neither the input bar nor the output bar can occur in a synchro link.

If o1 and o2 are distinct operations , o1 precedes an operation o2 if and only if :

•
o1 is the input bar, or

•
o2 is the output bar, or

•

there is a synchro link from o1 to o2, or

•
there is a datalink from a root of o1 to a terminal of o2, or

•
o1 precedes some operation o3, which precedes o2.

NOTE: A case is loosely analogous to the if-then construct in familiar structured languages such as Pascal. A sequence of cases in Prograph is also analogous to a COND in Lisp. In a case, however, the computations associated with testing conditions are intermingled with those that compute values, and do not necessarily have to return TRUE or FALSE, unlike in if-thens or CONDs. The sequence of cases defining a method is strictly analogous to the sequence of clauses that define a predicate in Prolog.

A case consists of a network of operations connected by datalinks; these distribute data from the roots of operations that produce it to the terminals of operations that use it.
The order in which the operations are executed must therefore be such that no attempt is made to execute an operation before its input data are available. The execution order of operations in a case is further constrained by synchro links. The programmer can use synchros to ensure that two operations are executed in a particular order, if this order is important, as it may be if side effects are involved, such as graphics or assigning values to attributes.

The input and output bars provide communication between the case and the outside world. The input bar copies the values from the terminals of the calling operation onto its own roots, and from there they are distributed to the terminals of operations in the case. The output bar copies the values on its terminals to the roots of the calling operation.

Operation

General "601"

An operation is a set of components, which always includes the following:

o Mode, the mode, which is either Plain or Repeat. An operation that has mode Repeat is also called a multiplex.

o Control, a control, which consists of:

•
a trigger, which is either Success or Failure

•
an action, which is one of Next Case, Terminate, Finish, Continue, or Fail

We frequently refer to a control with action A and trigger T by the phrase “A on T,” for example, “Next Case on Failure.”

Note that the control “Continue on Success” is the default and has no pictorial representation in the implementation of Prograph.

o Terminals, the sequence of terminals. The length of this sequence is called the inarity of the operation.

Every terminal has a mode, which is Simple, List, or Loop.

One terminal can be the name of the operation; if so, it is called an inject terminal.

o Roots, the sequence of roots. The length of this sequence is called the outarity of the operation.

Every root has a mode, which is Simple, List, Loop, True, or False.

Every operation, with the exception of Input and Output, also includes the following:

o Name, the name. This can be one of the terminals in T.

All the above are called standard components. Simple operations also include the following:

o Search, the search origin, which is either Normal or Super.

Throughout the rest of this chapter, whenever the words Mode, Control, Terminals, Roots, Name, or Search are used in reference to an operation, they denote an appropriate component of this operation as defined above.

NOTE: The terminals and roots of an operation signify input values and output results. They correspond approximately to the parameters of functions and procedures in C and Pascal, and to the parameters of goal literals in Prolog. In contrast to parameters in these languages, however, the inputs and outputs of an operation in Prograph are clearly distinguished. There is a closer analogy in Lisp: the parameters of a Lisp function evaluation correspond to terminals of an operation, while the single output of a function evaluation corresponds with a single root.

Every operation has an associated control. The default control is Continue on Success, which has no pictorial representation in the implementation of Prograph. The control interprets any condition that arises during the execution of the operation to determine how such a condition should affect the progress of execution. The closest analogy to this in other languages is the use of failure in Prolog: we can consider that every goal literal in Prolog has the associated control Backtrack on Failure.

The name of an operation depends on the category to which the operation belongs. It can be a value, an expression, a terminal, or a string that refers to another Prograph element such as a method, attribute or persistent. In the case of a terminal, the identity of the operation is determined at execution time from the string that arrives at this terminal.

Every operation satisfies the following conditions:

o The number of Loop roots equals the number of Loop terminals. The sequential order of roots and terminals therefore induces a one-to-one correspondence between Loop terminals and Loop roots. We say that a Loop terminal and Loop root correspond if they are matched by this mapping.

o If there is a True or False root, then Roots consists of exactly two roots, the first having mode True and the second False, and Terminals contains exactly one List terminal, which cannot also be an inject terminal.

o If there is a root or terminal of mode List or Loop, then the mode of the operation is Repeat.

NOTE: Matching Loop roots and terminals represent values that are iterated when an operation is executed as a multiplex. They correspond to loop variables in structured languages such as C and Pascal.

An operation is either a Simple operation, Get, Set, Instance generator, Persistent operation, Local, Evaluation, Input, Output, Constant, Match, or Environment. Environment operations are specific to the implementation environment. In the Macintosh implementation, an Environment operation is one of Mac Constant, Mac Match, Mac Global, Mac Address, Mac Get Field, or Mac Set Field.

Simple[603]

A Simple consists of the standard components together with Search where:

o Name is an identifier or an inject terminal.

o Search, which is Super only if Name is the empty string or a context-determined reference.

NOTE: Simple operations are equivalent to calls to procedures and functions in languages like C and Pascal, to goal literals in Prolog, and to messages in Smalltalk.

Get[603]

A Get consists of the standard components where:

o Name is a universal reference, a data-determined reference, or an inject terminal.

o Arity is (1,2).

o Each root is either Simple or List.

NOTE: A Get corresponds exactly to retrieving the value of the field of a record or structure in C, Pascal, or Smalltalk. There are no precisely analogous actions in Lisp or Prolog, although retrieving the value of a property in Lisp or binding a variable to a subterm of some Prolog structure are similar.

Set "603"

A Set consists of the standard components where:

o Name is a universal reference, a data-determined reference, or an inject terminal.

o Arity is (2,1).

o Each root is either Simple or List.

NOTE: A Set corresponds exactly to assigning a value to the field of a record or structure in C, Pascal, or Smalltalk. Setting the value of a property in Lisp is similar. There is no analogy in Prolog.

Instance Generator "604"

An Instance generator consists of the standard components where:

o Name is either an identifier or an inject terminal.

o Arity is (1,1).

o Each root is either Simple or List.

NOTE: An Instance generator is analogous to a call to the function “new” in Pascal and Smalltalk, and NewHandle or NewPtr in C (in a Macintosh environment). Lisp and Prolog have no analogous constructs.

Persistent "604"

A Persistent operation consists of the standard components where:

o Name is an identifier or an inject terminal.

o Inarity is at most 1.

o Outarity is at most 1.

o The root is either Simple or List, and is List only if there is a List terminal.

o Mode is Repeat only if there is a List terminal.

NOTE: A Persistent operation with input arity 1 corresponds to assignment to a global variable in C, Pascal, and Smalltalk, and to SETQ in Lisp. A persistent operation with outarity 1 corresponds to retrieving a value from a variable. Prolog has no equivalent.

Local "604"

A Local consists of the standard components where:

o Name is a sequence of cases, each having the same arity as the operation. This sequence of cases is also called the Local method corresponding to the Local operation.

NOTE: In Prograph a nested conditional structure could be handled by defining a method that would be called only from one operation to implement the inner conditional. Using a Local, however, makes this proliferation of named methods unnecessary by attaching the cases of the nested conditional directly to an operation. In Pascal, C, and Smalltalk, nested conditionals are implemented by textually nesting if-then compound-statement constructs. This textual nesting could be eliminated by turning the nested conditional into a procedure, which would be referenced only once.
In Prolog, nested conditionals are implemented by compound goals consisting of alternative goal literal sequences separated by ;.

The Name of a local as defined above should not be confused with its textual label, which serves purely as an informative comment (see chapter 8, “Pictopedia”).

Evaluation "605"

An Evaluation consists of the standard components where:

o Name is either the empty string or an expression defined by the grammar below, using the operations @ (exponentiation), +, - (binary and unary), *, /, // (integer division), % (remainder from integer division), & (bitwise AND), | (bitwise OR), ^ (bitwise exclusive OR), ~ (bitwise complement), << (left shift), >> (right shift). In this grammar, to avoid confusion with the symbol | which indicates alternative productions, we have used ! for the bitwise OR operator. The symbol | is actually used for this operator in the names of Evaluations. Likewise we have used « in place of << and » in place of >>.

<expression> : : =
<!expression>

<!expression> : : =
<!expression> ! <^strong> | <^weak> | <^strong>

<^weak> : : =
<^weak> ^ <&strong> | <&weak>

<^strong> : : =
<^strong> ^ <&strong> | <&strong>

<&weak> : : =
<&weak> & <«strong> | <«weak>

<&strong> : : =
<&strong> & <«strong> | <«strong>

<«weak> : : =
<«weak> <shiftop> <+strong> | <+weak>

<«strong> : : =
<«strong> <shiftop> <+strong> | <+strong>

<+weak> : : =
<+weak> <addop> <*strong> | <*weak>

<+strong> : : =
<+strong> <addop> <*strong> | <*strong>

<*weak> : : =
<*weak> <multop> <@strong> | <@weak>

<*strong> : : =
<*strong> <multop> <@strong> | <@strong>

<@weak> : : =
<weak operand> @ <@strong> | <weak operand>

<@strong> : : =
<strong operand> @ <@strong> | <strong operand>

<strong operand> : : =
<variable> | <real> | <integer> | (<expression>)

<weak operand> : : =
<unary> <operand>

<shiftop> : : =
« | »

<addop> : : =
+ | -

<multop> : : =
* | / | // | %

<unary> : : =
+ | - | ~

o If Name is not the empty string, inarity is equal to the largest index in the alphabet of the single letter operands occurring in Name. (Upper-case and lower-case versions of a letter are not distinguished.)

o Outarity is 1.

o The root is Simple, List, or Loop, and is List only if there is a List terminal.

o Mode is Repeat only if there is a List terminal.

o If there is a Loop terminal, there must also be a List terminal.

NOTE: An Evaluation is analogous to an expression in C and uses the same operators.

Input "606"

An Input consists of the standard components, except for Name, where:

o Mode is Plain.

o Control is Continue on Success.

o Inarity is 0.

o Every root is Simple.

Output "607"

An Output consists of the standard components except for Name, where:

o Mode is Plain.

o Outarity is 0.

o Every terminal is Simple.

Constant "607"

A Constant consists of the standard components where:

o Name is a <simple value>.

o Mode is Plain.

o Control is Continue on Success.

o Arity is (0,1).

o The root is Simple.

NOTE: A Constant is analogous to a constant in C and Pascal, to lists and constants in Lisp, and to ground terms in Prolog (terms containing no variables).

Match "607"

A Match consists of the standard components where:

o Name is a <simple value>.

o Arity is (1,0).

o Mode is Repeat only if the terminal is List.

o The root is Simple or List.

NOTE: A Match is comparable in most other languages to a test for equality of a datum with a constant (or list in Lisp). It corresponds more closely, however, to unification with a ground term in Prolog.

Mac Constant, Mac Match, Mac Global "608"

A Mac Constant (Mac Match, Mac Global) is syntactically equivalent to a Constant (Match, Persistent operation), except that the name must be the name of a Macintosh constant (constant, global). See Inside Macintosh for more information.

Mac Get Field, Mac Set Field "608"

A Mac Get Field (Mac Set Field) is syntactically equivalent to a Set (Get), except that N must be the name of a field of a Macintosh record (see Inside Macintosh), no root can be a List unless there is a List terminal, and there is an optional terminal for indexing into fields that are arrays.

Mac Address "608"

A Mac Address is syntactically equivalent to a Mac Get Field except that the outarity is 1.