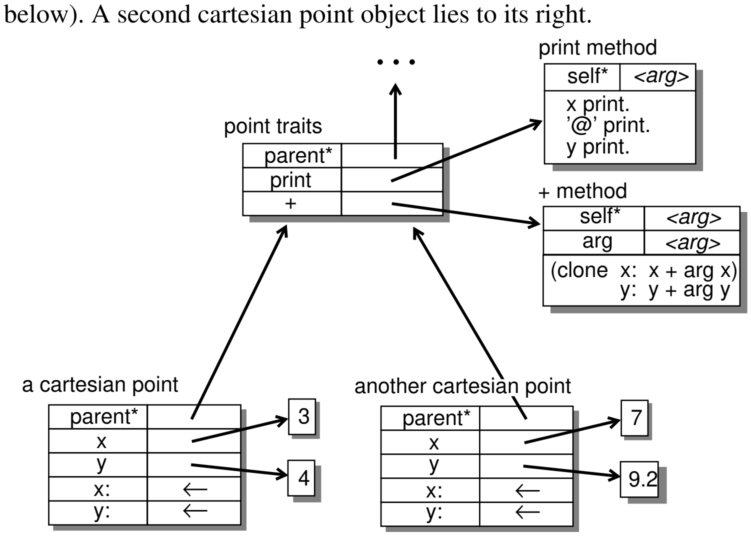
Chapter 4 The SELF Language

SELF is a dynamically-typed prototype-based object-oriented language with multiple, dynamic inheritance, originally designed by David Ungar and Randy Smith at Xerox PARC in 1986 [US87, HCC+91, UCCH91, CUCH91] as a successor to the Smalltalk-80 programming language. Like Smalltalk, SELF is intended for exploratory programming environments in which rapid program development and modification are primary goals. Hence SELF is dynamically-typed, affording greater flexibility and ease of development and modification, at the cost of reduced reliability and, given existing implementation technology, reduced run-time performance. Additionally, SELF includes the features described in Chapter 2 as desirable in an object-oriented language: abstract data types, a pure object-oriented model with dynamic binding on all messages (including all variable accesses), closures for user-defined control structures and exceptions, robust primitives, and support for generic arithmetic. Those readers familiar with SELF may choose to skim this chapter.

4.1 Basic Object Model

A SELF object consists of a set of *named slots*, each of which contains a reference to some other object. Some slots may be designated as *parent slots*. Objects may also have SELF source code associated with them, in which case the object is a *method*. To make a new object in SELF, an existing object (called the *prototype*) is simply *cloned* (shallow-copied) to produce a new object with the same name/value pairs as the prototype.

For example, the following picture portrays several SELF objects. The bottom-left object represents a cartesian point "instance" containing 5 slots: a parent slot named **`parent`** (identified as a parent slot by the asterisk next to the slot's name) containing a reference to the point traits object, two slots named **`x`** and **`y`** containing references to integer objects, and two slots named **`x:`** and **`y:`** that contain references to the assignment primitive method (notated using the ← symbol and described below). A second cartesian point object lies to its right.



The top-left object labeled **`point traits`** is inherited by all cartesian point objects. It also contains a parent slot named **`parent`** containing a reference to another object not shown in this diagram, a slot named **`print`** and **+** each containing a reference to a method object.

Method objects differ from other objects only in that they have attached SELF code in addition to slots. Each method object has a parent slot named **`self`** that is an argument slot; its contents in filled in with the receiver of the message when the method is invoked, as described below. The **+** method has an additional argument slot named **`arg`** that is filled in with the right-hand argument to the **+** message when the method is invoked. The two integer objects also have their own slots, but for conciseness we omit them from this diagram.

Two other kinds of objects appear in SELF: object arrays and byte arrays. Arrays are just like normal data objects, except that they additionally contain a variable number of array elements indexed by number instead of name. As their names suggest, object arrays contain elements that are arbitrary objects, while byte arrays contain only integer objects in the range 0 to 255, but in a more compact form suitable for interacting with external character- or byte-stream based systems. Primitive operations support fetching and storing elements of arrays as well as determining the size of an array and cloning a new array of a particular size.

4.1.1 Object Syntax

A programmer may describe a SELF object in textual form by listing the object's slots and its code inside parentheses. The slots are listed between vertical bars at the beginning of the object, with the code following afterwards; either of these components of an object may be omitted. A slot declaration begins with the slot's name, then an asterisk if the slot is a parent slot,* then either a left-arrow or an equal sign depending on whether or not, respectively, an assignment slot is desired, and then an expression which is evaluated to determine the slots contents. An assignable slot initialized to **nil** may be declared concisely by omitting the left-arrow and the initializer expression. Slots are separated by periods.

For example, the cartesian point object above could be defined as follows (comments are between double quotes):

```
( |
  parent* = traits point. "evaluates to the point traits object"
  x <- 3. "left-arrow creates corresponding assignment slot"
  y <- 4.
| )
```

This example illustrates the use of **=** to define a single data slot and **<–** to define a data slot/assignment slot pair; the name of the assignment slot is computed by appending a colon to the name of the data slot.

The point traits object could be defined as follows:

```
( |
  parent* = ... "code to evaluate to the parent of point traits"
  print = ( x print. '@' print. y print ).
  + = ( | :arg | (clone x: x + arg x) y: y + arg y ).
| )
```

The **print** and **+** method objects are defined directly as contents of slots. Method objects look just like other object declarations, except that they specify code in addition to any slots. Argument slots are prefixed with colons and may not be initialized. SELF defines a syntactic sugar for argument slots that allows them to be written in as part of the slot name; the **+** slot declaration could also have been written as follows:

```
+ arg = ( (clone x: x + arg x) y: y + arg y ).
```

SELF includes a few other forms for object literals, including integer and floating point literal expressions that evaluate to the corresponding integer and floating point objects and string literals delimited by single quotes.

4.2 Message Evaluation

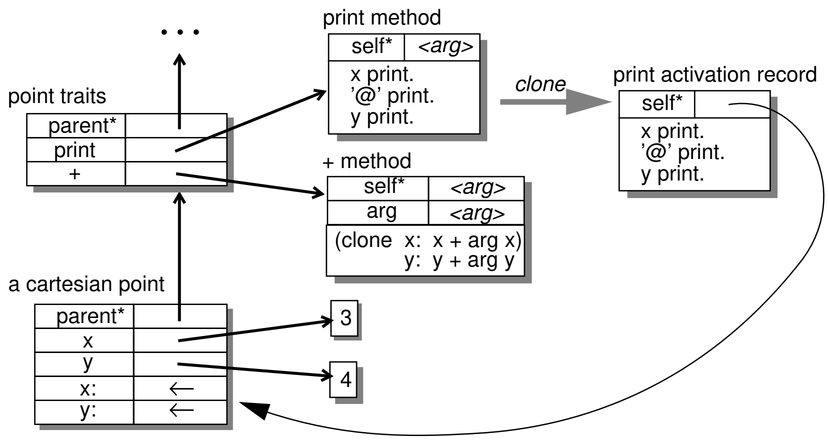
When a message is sent to an object (called the *receiver* of the message), the receiver object is scanned for a slot with the same name as the message. If a matching slot is not found, then the contents of the object's parent slots are searched recursively, using SELF's multiple inheritance rules to disambiguate any duplicate matching slots. For example, if the **x** message were sent to the cartesian point object pictured above, the system would search the cartesian point for a slot whose name is **x**, locating the slot referring to the 3 object. If instead the **print** message were sent to the cartesian point object, the system would first scan the cartesian point object for a slot named **print**, unsuccessfully. The system would then search each object stored in a parent slot of the cartesian point, which in this example would be the point traits object, and the system would find the matching **print** slot in this parent object.

Once a matching slot is found, the object referred to by the slot is *evaluated* and the result is returned as the result of the message send. An object without code evaluates to itself, and so the slot holding it acts like a variable. For example, when sending the **x** message to the cartesian point, the system locates the **x** slot in the point, extracts its contents (the

* The current version of SELF supports prioritized parents with differing numbers of asterisks for different parent priorities. Further details may be found in [CUCH91].

3 integer object), evaluates it (in this case just returning 3 again, since the 3 object contains no code and hence evaluates to itself), and returns the result (3) as the result of the original **`x`** message.

An object with code (a method) is treated as a prototype activation record. When evaluated, the system clones the method object, fills in the clone's **`self`** slot with the receiver of the message, fills in the clone's argument slots with the arguments of the message (if any), and executes its code. For example, if the **`print`** message were sent to the cartesian point, the system would locate the **`print`** slot in the point traits object, extract the **`print`** method object referenced by the slot, and evaluate the method object. Evaluating the method would involve cloning the method object to create a fresh activation record, filling in the contents of the **`self`** slot of the new activation record with the receiver cartesian point object, and then executing the messages specified by the code associated with the **`print`** method. The result of the last message in the **`print`** method would be returned as the result of the **`print`** message.



SELF supports assignments to data slots by associating an *assignment slot* with each assignable data slot. The assignment slot contains the *assignment primitive* method object, which takes one argument. When the assignment primitive is evaluated as the result of a message send, it stores its argument into the associated data slot. A data slot with no corresponding assignment slot is called a *constant* or *read-only slot* (as opposed to an *assignable data slot*), since a running program cannot change its value. For example, most parent slots are constant slots. However, SELF's object model allows a parent slot to be assignable just like any other slot, simply by defining its corresponding assignment slot. Such an assignable parent slot permits an object's inheritance to change on-the-fly at run-time, for instance as a result of a change in the object's state. We call such run-time changes in an object's inheritance *dynamic inheritance*, and we have found this facility to be of practical value in our SELF programming. Further information on the uses of dynamic inheritance may be found in [UCCH91].

4.2.1 Message Syntax

SELF message syntax is much like Smalltalk-80 message syntax. Both languages define three classes of message, distinguished syntactically:

- *Unary messages*. A unary message takes no arguments other than the receiver. Syntactically, a unary message name is written after its receiver expression (in postfix form), and is distinguished from other forms of message name by being an sequence of letters or digits that begins with a lower-case letter and does not end with a colon. Thus **`x`**, **`print`**, and **`isFirstQuadrant`** are all valid unary message names. Unary messages have highest precedence, and associate from left to right.
- *Binary messages*. A binary message takes a receiver and one argument, with the binary message name separating the two. A binary message is easily distinguished as any sequence of punctuation characters (excluding a few reserved sequences). Thus **`>`**, **`&&`**, **`=`**, and **`&^$#^`** are legal binary message names. Binary messages have medium precedence. No associativity is defined for binaries (programmers must explicitly add parenthesis to disambiguate sequences of binary messages), except that two binary messages left-associate if they are the same binary message. Therefore expressions like **`3 + 4 + 5`** are legal, with **`3 + 4`** being evaluated first, while expressions like **`3 + 4 * 5`** are illegal and must be explicit parenthesized. Arguments are always evaluated from left to right.

- *Keyword messages*. A keyword message takes a receiver and one or more arguments. Keyword message names are unusual in that the message name is written interspersed between the arguments to the message. Each piece of a keyword message name is a sequence of letters and digits, beginning with a letter, and ending with a colon (unlike unary messages which do not end with a colon). To aid in limiting the number of parentheses required for parsing, the first keyword piece must begin with a lower-case letter, while all subsequent keyword pieces must begin with an upper-case letter. The receiver is written before the keyword message, with an argument after each colon at the end of the keyword message pieces. The name of the message is the concatenation of the various name pieces. Therefore, **`x:`**, **`ifTrue:`**, and **`ifTrue:False:`** are all legal keyword message names, the first two taking one argument (and a receiver) and the third taking two arguments, while **`ifTrue:ifFalse:`** is not.

  Keyword messages have lowest precedence and associate from right to left. For example, the message **`x ifTrue: 5 False: 6`** sends the **`ifTrue:False:`** message to the result of the **`x`** message with 5 and 6 as arguments, while the message **`x ifTrue: 5 ifFalse: 6`** first sends the **`x`** message, then the **`ifFalse:`** message to 5 with 6 as an argument, and then the **`ifTrue:`** message to the result of the **`x`** message with the result of the **`ifFalse:`** message as an argument, as if the original message were parenthesized as **`x ifTrue: (5 ifFalse: 6)`**.

The code part of a method is simply a sequence of period-separated messages.

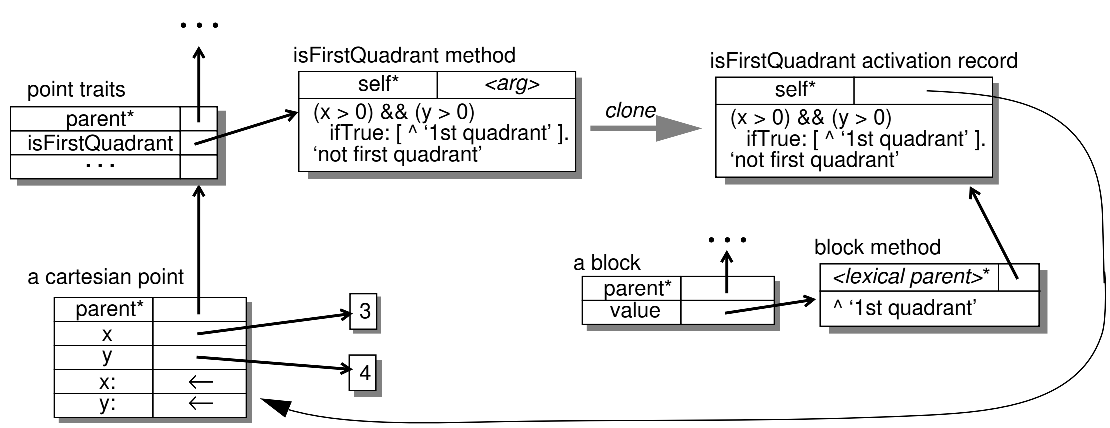
4.3 Blocks

SELF allows programmers to build their own control structures using *blocks*, SELF's version of closures. A block in SELF is an object with a slot named **`value`** that contains a special kind of method. When invoked (by sending **`value`** to the block object), this special block method runs as a child of its lexically-enclosing activation record (the activation record that was executing when the block object was created). A block method does not include a **`self`** parent slot, but instead has an anonymous parent slot that refers to the lexically-enclosing activation record object; the value of **`self`** is inherited from the enclosing method activation. These differences from "normal" methods enable blocks and block methods to act like lexically-scoped closures; SELF uses normal inheritance to implement lexical scoping.

Syntactically, blocks are identical to other method definitions, except that they are enclosed in square brackets instead of parentheses. In particular, variables local to a block activation record are declared as normal data slots in the slot list of the block literal.

For example, suppose an **`isFirstQuadrant`** method were added to the point traits object. This method tests whether both the **`x`** and **`y`** components of the receiver point are positive and if so returns the string literal **`'1st quadrant'`**. Otherwise the string literal **`'not first quadrant'`** is returned.

The following diagram shows the state of the system after invoking the **`isFirstQuadrant`** method and creating a new activation record.



The block object corresponds to the block literal enclosed in square brackets in the **`isFirstQuadrant`** method. The block's **`value`** slot refers to a block method object with an anonymous lexical parent slot, which refers to the block's lexically-enclosing activation record object.

A block method may terminate with a *non-local return* by prefixing the result expression with a ^ symbol (reminiscent of an up-arrow), causing the result to be returned not to the caller of the block method (the sender of **`value`**) but to the caller of the lexically-enclosing normal (non-block) method. Non-local returns thus have much the same effect as a **`return`** statement in C. For example, when executing the non-local return in the **`isFirstQuadrant`** example, the block would return not to the sender of **`value`** somewhere inside the **`ifTrue:`** user-defined control structure but instead to the caller of the lexically-enclosing method, in this case returning the **`'1st quadrant'`** string object to the sender of **`isFirstQuadrant`**.

4.4 Implicit Self Sends

Local variables and arguments are accessed in SELF using *implicit self* sends. These sends have **`self`** as the receiver of the message but begin the search for a matching slot with the current activation record rather than **`self`**. This search will follow the lexical chain of activation records (following the anonymous parent slots of nested block methods). Since arguments and local variables are simply normal slots in method prototype objects and their cloned activation records, implicit self message sends can support argument and local variable accesses using the same mechanisms used to access data slots and methods in "normal" objects. Since **`self`** is a parent slot of the outermost method activation record, implicit self sends can also be used to access slots in the receiver or its ancestors.

Implicit self sends are so termed because the **`self`** receiver is elided from the message send syntax; a message without an explicit receiver is implicitly a send to **`self`**. For example, the point **+** method contains the fragment **`x + arg x`**. This code first sends the **`x`** message to **`self`** implicitly. The lookup starts with the current activation record, and since the activation record does not contain an **`x`** slot, the system will scan the contents of the activation record's parent slots. The activation record's **`self`** slot is the only parent slot, and so the system will search the contents of the **`self`** slot, the receiver cartesian point, for an **`x`** slot. This search will be successful, and the system will evaluate the contents of the **`x`** slot to compute the result of the **`x`** message.

The example code fragment will next send the **`arg`** message to **`self`** implicitly. Again the lookup will begin with the current activation record, but this time the system will find a matching **`arg`** slot in the activation record. The contents of the **`arg`** local slot are accordingly evaluated, returning the argument to the original **+** message send.

Implicit self messages allow the SELF syntax for local slot accesses and slot accesses in the receiver to have the same concise syntactic expression as local, instance, and global variable accesses in Smalltalk, but with the more powerful semantics of full message sends. In particular, code which looks like it is accessing an instance variable, and which originally did access an instance variable, can be reused in situations in which the message actually invokes a method. This possibility enables the SELF system to solve such thorny reuse problems as the **`polygon`** and **`rectangle`** example from section 2.2.3 and the **`cartesian`** and **`polar`** point example to be described in section 4.6.

4.5 Primitives

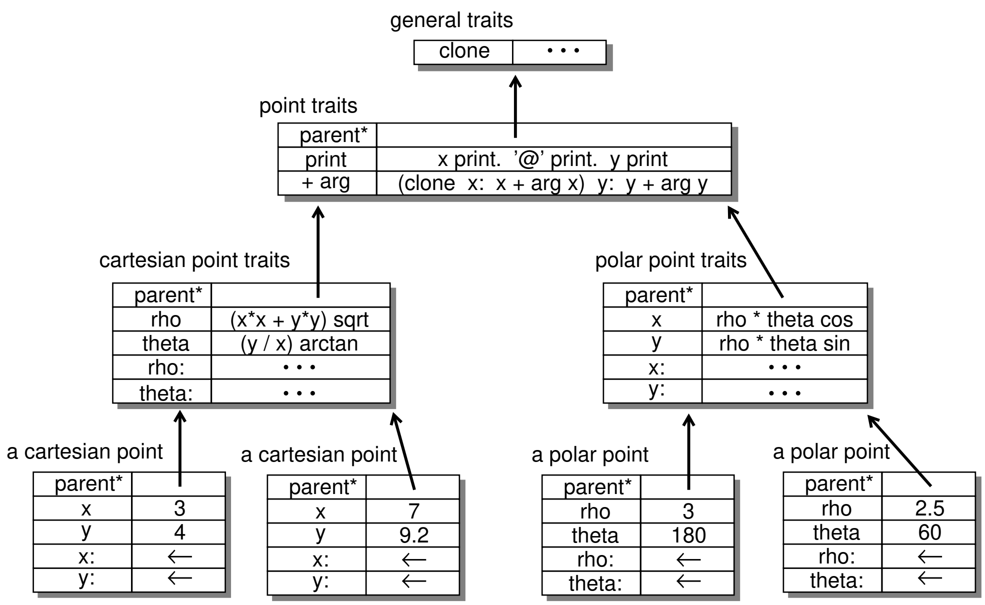
Much of the real work of a SELF program is performed by primitive operations provided by the virtual machine and implemented below the level of the language. Integer arithmetic, array accessing, and input/output are all provided via primitives to the SELF programmer. Primitive operations are invoked with the same syntax used to send a message, except that the message name begins with an underscore ("**_**"). For instance, **`_IntAdd:`** invokes the standard integer addition primitive. Every call of a primitive operation may optionally pass in a block to be invoked if the primitive fails by appending **`IfFail:`** to the message name and passing in the block as an additional argument. If invoked, the block is passed an error string identifying the nature of the failure (such overflow, divide by zero, or incorrect argument type). For example, **`3 _IntAdd: 'abc' IfFail: [ | :code | ...]`** passes a failure block in addition to the arguments to be added; this block will be invoked with the **`'badTypeError'`** object by the primitive since the arguments to the primitive are not both fixed-precision integers.

Loops are implemented in SELF via the **`_Restart`** primitive. A call to **`_Restart`** transfers control back to the beginning of the scope containing the **`_Restart`** call, creating a loop. The programmer uses a non-local return to break out of such a loop. Programmers can combine **`_Restart`**, non-local returns, and closures to build arbitrary user-defined looping control structures.

SELF uses **`_Restart`** to implement loops explicitly. Other languages such as Scheme instead perform *tail-recursion elimination* to automatically transform recursion into iteration, without introducing an extra language construct explicitly for iteration. Unfortunately, tail-recursion elimination, and more generally *tail-call elimination*, violates the user's execution and debugging model by eliminating activation records that the user expects to see. The Scheme language definition specifies that all tail-recursive calls must be transformed into iterations, which effectively introduces a special language mechanism for looping. In SELF, such looping code is explicit and easy to recognize, since only **`_Restart`** creates a loop. In Scheme, on the other hand, any procedure call that happens to be tail recursive will be transformed into an iterative loop, whether or not the programmer desired or expected it, and identifying when a procedure call is tail-recursive can be tricky.

4.6 An Example: Cartesian and Polar Points

The figure below presents an example collection of SELF objects. The bottom objects are two-dimensional point objects, the left ones represented using cartesian coordinates and the right ones using polar coordinates. The cartesian point traits object is the immediate parent object shared by all cartesian point objects, and defines four methods for interpreting cartesian points in terms of polar coordinates; the polar point traits object does the reverse for polar point objects. The point traits object is the shared ancestor of all point objects, defining general methods for printing and adding points, regardless of coordinate system. The point traits object inherits in turn from the topmost object in the diagram, which defines even more general behavior, such as how to copy objects.



Sending the **`x`** message to the leftmost cartesian point object finds the **`x`** slot immediately. The contents of the slot is the integer **`3`**, which evaluates to itself (it has no associated code), producing **`3`** as the result of the **`x`** message. Sending **`x`** to the rightmost polar point object, however, does not find a matching **`x`** slot immediately. Consequently, the object's parent is searched, finding the **`x`** slot defined in the polar point traits object. That **`x`** slot contains a method that computes a polar point's **`x`** coordinate from its **`rho`** and **`theta`** coordinates. The method gets cloned and executed, producing the floating point result **`1.25`**.

If the **`print`** message were sent to a point object, the **`print`** slot defined in the point traits object would be found. The method contained in the slot prints out the point object in cartesian coordinates. If the point were represented using cartesian coordinates, the **`x`** and **`y`** messages (implicitly sent to **`self`**) would access the corresponding data slots of the cartesian point object. But the **`print`** method works fine even for points represented using polar coordinates: the **`x`** and **`y`** messages would find the conversion methods defined in the polar point traits object to compute the correct **`x`** and **`y`** values.

This example illustrates conventional SELF programming practice. Most SELF code is structured into hierarchies of *traits objects*, abstract objects used to hold behavior to be inherited and refined by child objects. These traits objects play a role similar to one of the roles of classes in class-based languages. Concrete objects inherit from the traits objects, filling in any missing implementation, such as assignable data slots holding object-specific state information. The initial concrete objects are used as prototypical instances of the abstract data type and are cloned to create new instances.

The example also illustrates some of the challenges facing the SELF implementation. The frequency of message sends is very high; in this **`print`** example, nearly every source token corresponds to a message send. Even instance variables are accessed using message sends. Some other challenges facing the implementation that are not illustrated by this short example include user-defined control structures and generic arithmetic support.