Introduction to Sather for the C++ Literate

Satish R. Thatte

December 20, 1994

This does not purport to be a complete tutorial. Many interesting features of the language are not mentioned here. For instance, the only example of a user-defined *iter* is a very simple parameterless one, exception handling is not properly covered, features such as pre- and post-conditions, invariants, bound routines, and so on, are never mentioned. The purpose of this note is to help people familiar with C++ to understand the realization of basic object-oriented concepts in Sather, rather than to highlight the unusual aspects. Please send comments, corrections, and suggestions for improvement to `satish@sun.mcs.clarkson.edu`.

In the following discussion, a running example is developed as an illustration of Sather's features. The goal of the program is very simple: word counting. However, instead of attacking the problem directly, we produce some reusable abstract data types such as list and dictionary classes in the process of solving it. The code is interspersed with commentary on both the code and the language. The commentary should be treated like comments in a program-eliding it yields a working program.

For people accustomed to C++, the first thing to note is that in Sather interfaces and implementations are completely separated, and therefore subtyping and conformance of interfaces on the one hand, and (code) inheritance and polymorphism on the other, are entirely separate concepts. This makes for simplicity and clean designs except in a few cases where the "traditional" intertwining of the two concepts seems more useful (an example is given in the following).

All type names in Sather use capital letters only, and abstract types have names beginning with a `$`. The following abstract type `$PRINTABLE` defines a pure interface:

```
type $PRINTABLE is
     print;
end;
```

This is similar to an abstract class in C++ in that no implementation is given, or indeed, can be given, for the routine `print` mentioned in this type. However, we can use `$PRINTABLE` as a type in declaring attributes (data members to C++ fans) of concrete types, in declaring the types of parameters or results of routines, etcetera. Here is another abstract type `$ORD` of ordered objects.

```
type $ORD is
     is_leq(k : $ORD) : BOOL;
     is_geq(k : $ORD) : BOOL;
     is_eq (k : $ORD) : BOOL;
     is_neq(k : $ORD) : BOOL;
     is_gt(k : $ORD) : BOOL;
     is_lt(k : $ORD) : BOOL;
end;
```

Notice that type declarations for parameter and result types are in the Pascal rather than C style. In general, Sather follows the syntactic style of Eiffel, which is similar to Pascal. For instance, type expressions are used where appropriate rather than the sometimes mindboggling C practice of "prototype use" as a declaration form. A notable feature of the names of the routines in `$ORD` is that they are the standard (unsugared) names for the usual infix comparison operators `<=`, `/=`, etc. Like C++, Sather allows a class to define comparison, arithmetic, and assignment operators for itself.

It is obvious that all the other comparison operators in `$ORD` can be defined in terms of `<=` (`is_leq`) alone. In C++ we might leave `is_leq` as a "pure virtual" function and define the rest using it. We would then have a class that is partly abstract and partly concrete. This is not possible in Sather. Abstract types (interfaces) contain no implementation, and concrete types must implement every routine they contain. However, the implementation may merely raise an exception informing the client that a routine is in fact not implemented! Thus, we can define a class `ORD` which incorporates the common code for the dependent routines.

```
class ORD < $ORD is                        -- mainly for code reduction
     is_leq(other : $ORD) : BOOL is        -- this is left to subclasses
          raise "<= is not implemented in this incomplete class\n";
     end;
     is_geq(k : $ORD) : BOOL is            -- the rest are dependent defs
          return (k <= self); end;
     is_eq (k : $ORD) : BOOL is
          return (self <= k) and (k <= self); end;
     is_gt(k : $ORD) : BOOL is
          return (k <= self) and (self /= k); end;
     is_lt(k : $ORD) : BOOL is
          return (self <= k) and (self /= k); end;
     is_neq(k : $ORD) : BOOL is
          return ~(self=k); end;
end;
```

The first line of the declaration of `ORD` demonstrates the primary use of abstract types—to serve as interface constraints on concrete types. The declared subtype relationship denotes the constraint that the signature of `ORD` must conform to that of `$ORD` in the contravariant sense, similar to the relationship of the interface of a derived class with that of one of its abstract base classes in C++. This guarantees type-correct usability of `ORD` objects where `$ORD` objects are expected. Note the contrast with Eiffel (the Godfather of Sather) which uses covariant conformance and hence needs global checking to avoid unsoundness of the type system.

`ORD` is a reference class, and objects of this class are stored and used via references (rather like pointers in C except that there is no pointer arithmetic). This means that space for them must be allocated, they can be shared, and so on. However, they do not have to be deallocated because Sather has garbage collection built-in. Most classes are reference classes, but one does occasionally need the other kind of class—a value class. We will see an example of one later.

Subtyping constraints, such as `ORD < $ORD` above, are not limited to single supertypes. An abstract or concrete type may conform to multiple abstract interfaces. This is one kind of "multiple inheritance" in Sather. For instance,

```
type $ELEM < $PRINTABLE, $ORD is
end;
```

declares a new abstract type that is just a shorthand for the two previously defined abstract types. This would be needed if we want to later declare attributes and parameters that must be both printable and ordered. An attribute needs a single type!

In general, it is standard practice in Sather to separately declare an abstract interface for each primary concrete type one needs. We will need a list class so we begin by defining its interface:

```
type $LIST{DATA < $PRINTABLE} < $PRINTABLE is
     insert(x : DATA);
     trav! : DATA;            -- an iterator: see below
     find(k : DATA) : DATA;
end;
```

`$LIST` is actually a parameterized type—the data stored in a list is of some arbitrary type, and we use a type parameter `DATA` to represent that type. Since we have said that `$LIST{DATA}` is a subtype of `$PRINTABLE`, we do not need to include a `print` routine in the interface. Any concrete type that conforms to `$LIST{DATA}` must also conform to `$PRINTABLE`. If we want `$LIST{DATA}` objects to be printable, the `DATA` objects themselves must be printable. We express this expectation by constraining actual parameters of `$LIST` to be subtypes of `$PRINTABLE`. By using a type parameter for the data type (rather than an abstract supertype like `$PRINTABLE`), we are making it possible to construct concrete lists of this kind that are known to be homogeneous at compile-time.

Now we come to what I think is a wart. Suppose we need to use a class `LINK` in implementing a concrete type `LIST`, which conforms to the `$LIST{DATA}` interface. `$LIST{DATA}` does not mention `LINK` . Let us assume that no client of `LIST` will need to know that `LIST` is implemented this way. `LINK` should then be private to `LIST` and should not be exposed outside it. This is not possible in Sather because Sather does not (yet) allow nested class definitions. Thus the `LINK` class must be public.

```
class LINK{DATA} is
        attr next : SAME;
        attr data : DATA;
```

An interesting point is that the next `LINK`, instead of being a `LINK{DATA}` attribute, is declared to be of type `SAME` which corresponds to Eiffel's `like current`. This has no counterpart in C++. It is a way of specifying the "type of `self`" whatever `self` may be. In an object of class `LINK`, `SAME` is just `LINK`, but in a class that inherits from `LINK`, the inherited attribute `next` would be a reference to an object of the new class, as it should be, rather than to a `LINK` object. The use of `SAME` is allowed only in concrete types. Unlike Eiffel, it has no role in conformance testing.

Since (unlike `ORD`) we actually expect to create objects of type `LINK{DATA}`, we must provide a constructor routine for the type since there are no default constructors.

```
        create(a : DATA, n : SAME) : SAME is -- a constructor
          ret : SAME := new;                 -- new allocates storage
          ret.data := a; ret.next := n;
          return ret;
        end;
end; -- class LINK
```

A constructor in Sather is always called `create`, and one must always be provided for any concrete class one means to instantiate. In a reference class like `LINK`, the constructor is expected to both allocate space and initialize it. The `new` routine used in `create` returns a reference to a newly allocated object of type `SAME`—`new` is obviously meant primarily for use in `create` routines. The only way to create a new `LINK` object outside class `LINK` is to call a `create` routine from `LINK`. In this respect, `create` is like a cross between C++ `new` and a constructor. Like C++, one can have multiple constructors by defining multiple `create` routines distinguished from each other by parameter types. In general, Sather supports overloading of all kinds of names including class names (which are distinguished by different numbers of type parameters).

We are now ready to implement a generic list type. We shall use a circular list of `LINK`s as the representation.

```
class LIST{DATA} < $LIST{DATA} is
```

Note that this declaration implies that `DATA` in `LIST` can only be substituted by subtypes of `$PRINTABLE` because if we want to substitute `T` for `DATA`, `LIST{T}` must conform to `$LIST{T}`, but the latter type expression is ill-formed unless `T < $PRINTABLE`.

```
    private attr last : LINK{DATA};    -- last.next is head of LIST
```

This attribute is only visible inside class `LIST`, like a private member in C++.

```
    append(x : DATA) is      -- attach new data at the tail end
         if void(last) then
              last := #LINK{DATA}(x,void);
              last.next := last;
         else
              last.next := #LINK{DATA}(x, last.next);
              last := last.next;
         end;
    end;
```

The creation expression `#LINKDATA(x,void)` used in the routine above is equivalent to the C++ expression `new LINK<DATA>(x,void)`. The `#` syntax is a sugared form for a `create` call.

```
    insert(x : DATA) is -- the default behavior
         append(x);
    end;
    create : SAME is
         return new;
    end;
```

The attribute `last` in the new `LIST` object returned by `new` is automatically initialized to `void`, which is the null reference in this context (the value associated with `void` is context dependent).

The next feature in this class is a special kind of routine called an *iter* (short for iterator), which is used to control loops in Sather. Iters have no direct counterpart in C++[1]. An iter in Sather behaves like a kind of coroutine. Instead of a return, an iter normally `yield`s control (possibly along with a value) to its caller without terminating itself. Termination of an iter is caused by a `quit` statement which also causes termination of the loop in which the iter call occurred. The name of an iter always ends in a "`!`".

[1]C++ has normal looping constructs which can be combined with iterators that are defined via special iterator classes.

```
trav! : DATA is
     ptr : LINK{DATA};            -- a local variable
     if void(last) then ptr := last;
     else ptr := last.next;
     end;
     loop until! (void(ptr));      -- uses built-in iter
          tl ::= ptr;
          if ~SYS::ob_eq(ptr,last) then ptr := ptr.next;
          else ptr := void;
          end;
          yield tl.data;
     end;
     quit;
end;
```

Multiple instantiations of `trav!` may be active on the same `LIST` object simultaneously and they `yield` and `quit` entirely independently of each other. Since all the information about the current state of the iteration is maintained locally in the iter, rather than in the `LIST` object, the multiple iterations don't interact at all. This iter does not take any parameters, but iters in general can specify parameters just like routines.

Note the use of `SYS::ob_eq(ptr,last)` to perform a direct reference comparison between `ptr` and `last`. Using `ptr=last` would be quite different—it would invoke the routine `is_eq` in the `LINK` object to which `ptr` holds a reference, and of course, `LINK` does not define such a routine. This example also shows the use of a routine directly from the built-in class `SYS`, without an object to invoke it from. The syntax for such qualified names is identical to C++. An example of the use of the iter `trav!` is in the following routine:

```
find(k : DATA) : DATA is
     loop temp ::= self.trav!;
          if  temp = k then return temp; end;
     end;
     return void;
end;
```

There is no special position for the occurrence of an iter. The iter invocation occurs as an ordinary expression. The use of `::=` syntax in the assignment involving the iter is merely a short-form which allows us to declare a local variable `temp` without explicitly specifying its type. The type is inferred from the right-hand side of the assignment. This has nothing to do with the iter. Every time the loop body reaches the call `self.trav!`, the iter continues from the previous `yield` point, until finally it `quit`s instead of `yield`ing, which causes this loop to terminate as with a `break` statement in C and C++. The following `print` routine also uses the iter except that it omits the unnecessary use of `self` in invoking it.

```
print is
     #OUT + "\n\n";
     temp : DATA;
     loop temp := trav!;
           temp.print;
     end;
     #OUT + "\n\n";
end;
```

This routine also illustrates the Sather method for appending to standard output. It is quite similar to the C++ ostream idea except that a new object of the corresponding class `OUT` is created for every output statement. There are no stream constants correponding to `cout` or `cerr`. The reason for this is interesting. As in C++, I/O in Sather is provided by a standard library. However, the only global entities in Sather are classes. The library is therefore unable to export constants equivalent to `cout` and `cerr`. It must export distinct classes `OUT` and `ERR` instead.

```
end;      -- class LIST
```

`LIST` is obviously a very simple generic implementation subject to many variations. One normally thinks of inheritance as being the mechanism for code reuse with variation in object systems. Being an object-oriented language, Sather must obviously support inheritance, but it does so in a rather unusual, even startling way, by taking the idea that inheritance is for code reuse quite literally. Inheritance in Sather is merely an `include` statement which in theory simply copies the code of the included class into the including class. To avoid the code duplication that would result from a naive implementation of this notion, Sather does use dynamic dispatch at run-time[2]. However, there is no IS-A relationship between the included and including class, and no substitutability relation between them. In fact there is no relation between the two classes at all, except that one was generous enough to donate its code to the other. Substitutability and conformance applies only to abstract types. The class `SORTLIST` defined below illustrates this concept. Note that `SORTLIST` promises conformance to the `$LIST` interface, thus establishing an indirect relationship to `LIST`—conformance to the same interface.

```
class SORTLIST{DATA < $ORD} < $LIST{DATA} is
     include LIST{DATA} append -> private append;
```

`SORTLIST` needs an extra constraint on its type parameter to ensure that the data being stored can be sorted. The `append -> private append` part of the `include` clause makes the imported `append` routine private in `SORTLIST`. In general, this notation can be used to rename, or even remove features inherited via the `include` clause. Since substitutability is not expected, feature removal is no problem. Multiple inheritance in the code reuse sense simply entails multiple `include` clauses, with ambiguities being dealt with by renaming or removal of features. The only routine `SORTLIST` wants to replace is `insert`. This is accomplished by simply providing a new definition. It is not necessary to explicitly remove `insert` from `LIST` in the `include` clause.

```
     insert(x : DATA) is
          if void(last) then append(x);
          else
               temp ::= last;
               loop while!(temp.next.data <= x
                         and ~SYS::ob_eq(temp.next,last));
                   temp := temp.next;
               end;
               if (temp.next.data > x) then
                    temp.next := #LINK{DATA}(x,temp.next);
               else append(x);
               end;
          end;
     end;
```

[2] Of course, dynamic dispatch is also required to implement routine invocations on abstract attributes, and hence is not simply an implementation hack to avoid code bloat.

Note that `while!` in the loop in `insert` is simply a built-in iter, and as such, it is "stand-alone." It does not have to be invoked "on" any object in particular. Also, `while!` takes a boolean-valued expression as its parameter, and the parameter obviously needs to be reevaluated every time control returns to the iter. User-defined iters can also have parameters that need such reevaluation. The names of such parameters end in a "`!`" to indicate this.

```
end; -- class SORTLIST
```

We have so far defined two general utilities: the parameterized classes `LIST` and `SORTLIST`. We now use `SORTLIST` to implement a third abstraction, a general dictionary class called `VECTOR`. The reason to call it `VECTOR` is that we would like to use array syntax when using `VECTOR` objects. Thus, if `k` is a key object, `v` is a vector object, and `d` is an object of the class of values indexed by keys in `v`, then we would like to use "`v[k]`" to retrieve the value stored at `k` in `v`, and "`v[k] := d`" to reset it to `d`. This is possible in Sather, just as it is in C++.

The representation of `VECTOR` is a `SORTLIST` of `PAIR`s, where a `PAIR` object stores a key-value association in the dictionary. `PAIR` should really be a private class inside `VECTOR`. It is irrelevant to clients of `VECTOR` and pollutes the global environment. The class would also be simpler inside `VECTOR` since it would not have to be separately parameterized.

```
class PAIR{KEY < $ELEM, VALUE < $PRINTABLE}  < $ELEM is
  include ORD;                -- code reduction
  attr index : KEY;
  attr val : VALUE;
  is_leq(s : $ORD) : BOOL is
     typecase s
       when PAIR{KEY,VALUE} then return index <= s.index;
       else raise "Bad comparison: PAIR with non-PAIR";
       end;
  end;
```

Since `PAIR` must conform to the `$ORD` interface in order to be usable as `DATA` in a `SORTLIST`, its `is_leq` routine is forced to accept any `$ORD` object as a parameter. However, comparison between a `PAIR` and an arbitrary `$ORD` object obviously does not make sense. This is the kind of example that prompted Eiffel to use covariant conformance rules. Sather has a much simpler solution: run-time typechecking. A user can test the type of an object at run-time using a `typecase` statement with obvious semantics, as in the `is_leq` routine above.

The fact that `is_leq` has to be defined for `PAIR` brings out one of the costs of the complete separation of inheritance and conformance in Sather. In the C++ version of this program, `PAIR` could be local to `VECTOR`, and the `KEY` parameter of `VECTOR` could be used as a public base class of `PAIR`. `PAIR` would then simply inherit and use the comparison operations from `KEY`. This is natural since a `PAIR` can be thought of as an extended `KEY` which carries the baggage of some additional data. None of this is possible in Sather. The lack of nested class definitions is relatively easy to mend but the deeper problem is that an IS-A relationship between `PAIR` and `KEY` is excluded since both are concrete classes.

```
    print is
        index.print;
        #OUT + "  ";
        val.print;
        #OUT + "\n";
    end;
    create(k : KEY, v : VALUE) : SAME is
        ret ::= new;
        ret.index := k;
        ret.val := v;
        return ret;
    end;
end;      -- class PAIR
```

The abstract parameterized interface of vector-like classes is defined below as `$VECTOR` where `aget` and `aset` are unsugared names of the standard array element retrieval and update routines exemplified by "`v[k]`" and "`v[k] := d`" in the discussion above.

```
type $VECTOR{KEY < $ELEM, VALUE < $PRINTABLE} < $PRINTABLE is
     aget(k : KEY) : VALUE;
     aset(k : KEY, v : VALUE);
end;
```

The concrete class `VECTOR{KEY,VALUE}` below uses a `SORTLIST{PAIR{KEY,VALUE}}` object as its representation. We could have defined this as a private attribute, but instead we choose to use a private `include` clause which is similar to private inheritance in C++.

```
class VECTOR{KEY,VALUE} < $VECTOR{KEY,VALUE} is
     private include SORTLIST{PAIR{KEY,VALUE}} print->print;
```

The annotation `print->print` in the `include` clause serves to expose the `print` routine which would otherwise be private in `VECTOR`.

```
     private attr init : VALUE;
```

When a new `PAIR` object is created for a `KEY` seen for the first time, the `VALUE` attribute in it must be initialized to some sensible value. Sather generally has reasonable defaults for this[3], but `VECTOR` chooses to keep this under user control and expects the `create` routine below to be explicitly given a value for this purpose which is stored in the private attribute `init`. In C++, `init` would be a constant member which is initialized by the constructor, but Sather has no mechanism parallel to this. Constant attributes in Sather are shared, *i.e.*, they are like constant static members in C++.

```
     create(d : VALUE) : SAME is
          ret ::= new;
          ret.init := d;
          return ret;
     end;
```

[3]Expressed via the context dependent values of `void`.

The private lookup routine below finds a PAIR in the representation, given its KEY. This is the primary computation in both the aget and aset routines. To continue the discussion of the implications of the impossibility of an IS-A relationship between PAIR and KEY in Sather, the fact that comparisons for PAIR objects were defined separately in PAIR instead of being inherited from KEY makes the implementation of lookup somewhat clumsy. Since only a PAIR can be compared to another PAIR, the KEY object given to lookup must be wrapped in a PAIR object even though a PAIR with this KEY may already have been created. In the C++ version, find can be called directly with the KEY object k.

```
        private lookup(k : KEY) : PAIRKEY,VALUE is
             temp ::= #PAIR{KEY,VALUE}(k,init);
             p ::= find(temp);
             if void(p) then
                  p := temp;
                  insert(p);
             end;
             return p;
        end;
        aget(k : KEY) : VALUE is
             return lookup(k).val;
        end;
        aset(k : KEY, v : VALUE) is
             lookup(k).val := v;
        end;
   end;      -- class VECTOR
```

In our word-counting application of VECTORs, we need to use strings and integers as keys and values respectively. Both are built-in classes (STR and INT) in Sather, but they don't conform to the interfaces demanded by VECTOR for its parameters. We shall therefore wrap them in classes STRING and INTEGER below.

```
   class STRING < $ELEM is
        include ORD;
        attr rep : STR;
        print is
             #OUT + rep;
        end;
        is_leq(o : $ORD) : BOOL is
             typecase o
             when STRING then return rep <= o.rep;
             else raise "Bad argument for <= in STRING\n";
             end;
        end;
        create(x : STR) : SAME is
             ret ::= new;
             ret.rep := x;
             return ret;
        end;
   end;      -- class STRING
```

Class INTEGER below is our only example of a value class. Value classes generalize the notion of immutable values such as integers and booleans supported by most languages. These objects have no mutable state and therefore cannot be shared in any useful sense. Sather generally uses in place allocation for these objects (similar to non-pointer object variables in C++), except when they are assigned to abstract attributes. In most cases, therefore, an INTEGER object will be stored and used exactly like an INT object.

```
value class INTEGER < $PRINTABLE is
     attr rep : INT;
     print is
          #OUT + rep;
     end;
     plus(x : INT) : SAME is
          return #(rep+x);
     end;
```

As explained above, an INTEGER object is immutable—plus (the unsugared version of +) therefore creates a new INTEGER object and initializes it with the result of the addition before returning it. Since the type of the object to be created is omitted in the creation expression, the default type SAME is used.

```
     create(i : INT) : SAME is
          ret : SAME;
          return ret.rep(i);
     end;
```

Note the use of the unsugared version of assignment (assigning i to rep) in the return statement in the create routine above.

```
end;      - class INTEGER
```

The main program in sather is always a routine called main in a specially designated class, called MAIN by default. The name of the class for the main program can be specified with a command-line argument if it is different from MAIN. The main program below is a straightforward application of the VECTOR, STRING and INTEGER classes. The only noteworthy aspect is the use of the built-in standard-input class IN—the get_str and eof routines are invoked directly from the class not via any object. However, a new object of class IN could have been used with equivalent effect.

```
class MAIN is
     main is
          counts ::= #VECTOR{STRING,INTEGER}(#INTEGER(0));
          loop      buf ::= #STRING(IN::get_str);
               until!(IN::eof);
               #OUT + "Read in: \"";
               buf.print;
               #OUT + "\"\n";
               counts[buf] := counts[buf]+1;
          end;
          counts.print;
     end;
end;
```