Dr. Dobb's SOURCEBOOK

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

Tomorrow's Programming Languages Today!

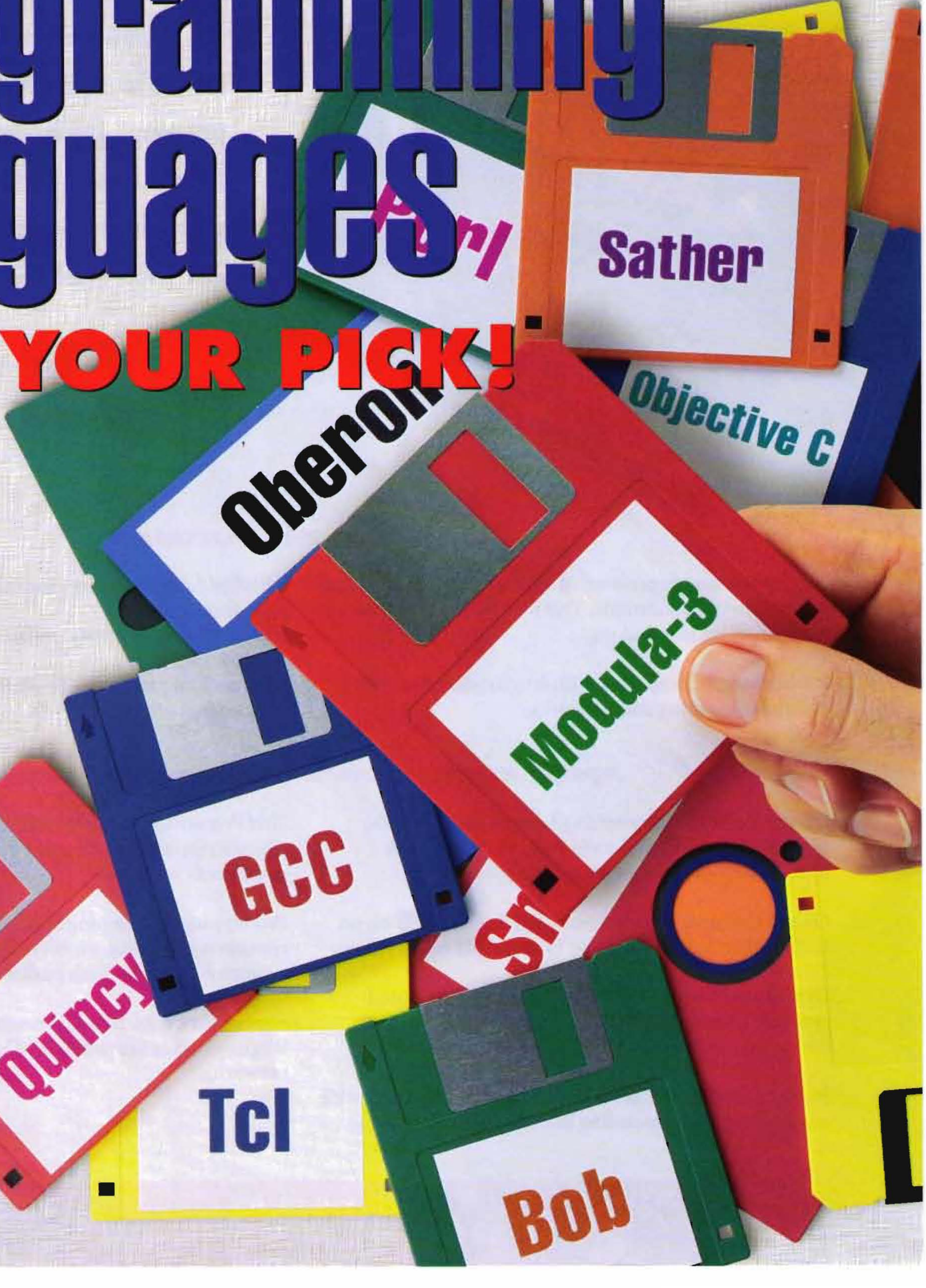
SPECIAL ISSUE!

#221 Winter 1994

Alternative Programming Languages

TAKE YOUR PICK!

- Parasol
- Perl
- Sather
- Modula-3
- Oberon
- Bob
- Tcl



l Stevens' uincy C nterpreter

.95 U.S. ($5.95 Canada)

/iller Freeman Publication

Raise Your Programming

The **Alternative Programming Languages CD-ROM** from *Dr. Dobb's Journal* is your resource for the latest in cutting-edge programming environments. Gain access to the most innovative, solution-providing languages and tools available today.

Dr.Dobb's
CD-ROM
LIBRARY

DELVE INTO THE FUTURE

Examine the next generation of languages. Write the next generation of applications. This CD-ROM is the tool you need to get up to speed on…

- Distributed Computing with languages that support parallel computing and networks.
- Embeddable Languages for raising the power of your applications.
- Increased Productivity utilizing cleaner, more efficient code.
- Future Languages that break the chains of today's languages.

APPLY TOMORROW'S LANGUAGES TODAY

The **Alternative Programming Languages CD-ROM** provides languages and tools you can put to work immediately on your development projects:

Object-Oriented Development. Tired of C++? Look no further than Sather, Dylan or Bob and get re-energized.

Distributed Computing. When building distributed systems, languages like Parasol and M0 make your life easier.

Multiple Platforms. Check out Perl and Python, plus a wide range of compilers, including the GNU compiler suite.

Text Processing and Document Formatting. Lout and Ghostscript let you edit and produce complex documents with ease.

Prototyping. Developing applications for distributed systems is one thing, prototyping connected distributed systems is another. Glish makes it all happen.

Neural Networks. Neudl, a neural net description language, takes the pain out of building complex neural networks.

ONLY $44.95
60-DAY MONEY-BACK GUARANTEE!

CONTENTS

Winter 1994
Volume 19, Issue 12
Language Sourcebook

FEATURES

PROGRAMMER'S SERVICES

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (California residents add sales tax) to *Dr. Dobb's Journal*, 411 Borel Ave., San Mateo, CA 94402, call 415-655-4321 x5701, or use your credit card to order by fax, 415-358-9749. Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ) or via anonymous FTP from site ftp.mv.com (192.80.84.1) in the /pub/ddj directory.

The Sather Programming Language

Efficient, interactive, and object oriented

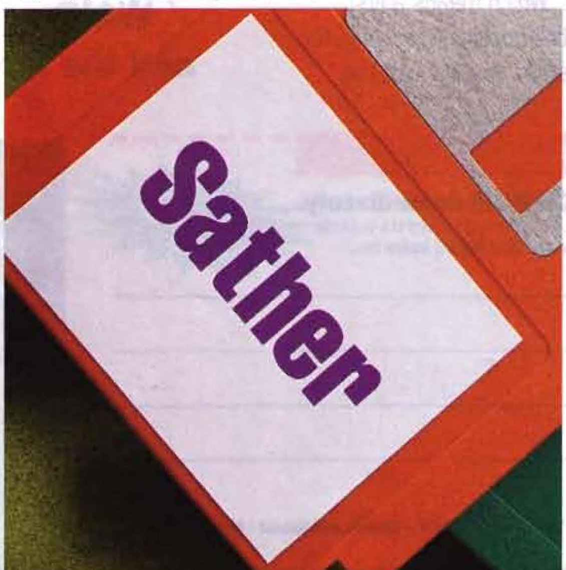
Stephen M. Omohundro

Sather is an object-oriented language which aims to be simple, efficient, interactive, safe, and nonproprietary. One way of placing it in the "space of languages" is to say that it aims to be as efficient as C, C++, or Fortran, as elegant and safe as Eiffel or CLU; and as supportive of interactive programming and higher-order functions as Common Lisp, Scheme, or Smalltalk.

Sather has parameterized classes, object-oriented dispatch, statically checked strong typing, separate implementation and type inheritance, multiple inheritance, garbage collection, iteration abstraction, higher-order routines and iters, exception handling, constructors for arbitrary data structures and assertions, preconditions, postconditions, and class invariants. This article describes a few of these features. The development environment integrates an interpreter, a debugger, and a compiler. Sather programs can be compiled into portable C code and can efficiently link with C object files. Sather has a very unrestrictive license which allows its use in proprietary projects but encourages contribution to the public library.

Stephen does research on learning and computer vision, as well as developing Sather at the International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704. He wrote the three-dimensional graphics for Mathematica and was a codesigner of Star-Lisp for the Connection Machine. He can be contacted at om@icsi.berkeley.edu.

The original 0.2 version of the Sather compiler and tools was made available in June 1991. This article describes version 1.0. By the time you read this, the combined 1.0 compiler/interpreter/debugger should be available on *ftp.icsi.berkeley.edu,* and the newsgroup *comp.lang.sather* should be activated for discussion.



Code Reuse

The primary benefit object-oriented languages promise is code reuse. Sather programs consist of collections of modules called "classes" which encapsulate well-defined abstractions. If the abstractions are chosen carefully, they can be used over and over in a variety of different situations.

An obvious benefit of reuse is that less new code needs to be written. Equally important is the fact that reusable code is usually better written, more reliable, and easier to debug because programmers are willing to put more care and thought into writing and debugging code which will be used in many projects. In a good object-oriented environment, programming should feel like plugging together prefabricated components. Most bugs occur in the 10 percent or so of newly written code, not in the 90 percent of well-tested library classes. This usually leads to simpler debugging and greater reliability.

Why don't traditional subroutine libraries give the same benefits? Subroutine libraries make it easy for newly written code to make calls on existing code but not for existing code to make calls on new code. Consider a visualization package that displays data on a certain kind of display by calling display-interface routines. Later, the decision is made that the package should work with a new kind of display. In traditional languages, there's no simple way to

(continued from page 18)
get the previously written visualization routines to make calls on the new display interface. This problem is especially severe if the choice of display interface must be made at run time.

Sather provides two primary ways for existing code to call newly written code. "Parameterized classes" allow the binding to be made at compile time, and "object-oriented dispatch" allows the choice to be made at run time. I'll demonstrate these two mechanisms using simple classes for stacks and polygons.

Parameterized Classes

Listing One (page 22) shows a class which implements a stack abstraction. We want stacks of characters, strings, polygons, and so on, but we don't want to write new versions for each type of element. STACK{T} is a parameterized class in which the parameter *T* specifies the stack-element type. When the class is used, the type parameter is specified.

For example, the class FOO in Listing One defines a routine which uses both a stack of characters and a stack of strings. The type specifier STACK{CHAR} causes the compiler to generate code with the type parameter *T* replaced by CHAR. The specifier STACK{STR} similarly causes code to be generated based on *STR*. Since character objects are usually eight bits and strings are represented by pointers, the two kinds of stack will have different layouts in memory. The same Sather source code is reused to generate different object code for the two types. We may define a new type (such as triple-length integers) and immediately use stacks of elements of that type without modifying the STACK class. Using parameterized classes adds no extra run-time cost, but the choice of type parameter values must be made at compile time.

Object-Oriented Dispatch

Listing Two (page 22) shows an example of object-oriented dispatch. The class $POLYGON is an "abstract" class, which means it represents a set of possible object types called its "descendants" (in this case, TRIANGLE and SQUARE). Abstract classes define abstract interfaces which must be implemented by all their descendants. Listing Two only shows the single routine *number_of_vertices:INT*, which returns the number of vertices of a polygon. TRIANGLE's implementation returns the value 3, and SQUARE's returns 4.

Routines in the interface of an abstract type may be called on variables declared by that type. The actual code that's called, however, is determined at run time by the type of the object which is held by the variable. The class FOO2 defines a routine with a local variable of type STACK{$POLYGON}. Both TRIANGLE and SQUARE objects can be pushed onto stacks of this type. The call *s.pop* might return either a triangle or a square. The call *s.pop.number_of_vertices* calls either the *number_of_vertices* routine defined by TRIANGLE and returns 3, or the *number_of_vertices* routine defined by SQUARE and returns 4. The choice is made according to the run-time type of the popped object. The names of abstract types begin with a $ (dollar sign) to help distinguish them (calls on abstract types are slightly more expensive than nondispatched calls).

The Sather type system is a major factor in the computational efficiency, clarity, and safety of Sather programs

Strong Typing

The Sather type system is a major factor in the computational efficiency, clarity, and safety of Sather programs. It also has a big effect on the "feel" of Sather programming. Many object-oriented languages have either weak typing or none at all. Sather, however, is "strongly typed," meaning that every Sather object and variable has a specified type and that there are precise rules defining the types of object that each variable can hold. Sather is able to statically check programs for type correctness — if a piece of Sather code is accepted by the interpreter or compiler, it's impossible for it to assign an object of an incorrect type to a variable.

Statically checked, strong typing helps the Sather compiler generate efficient code because it has more information. Sather avoids many of the run-time tag-checking operations done by less strongly typed languages.

Statically checked, strongly typed languages help programmers to produce programs that are more likely to be correct. For example, a common mistake in C is to confuse the C assignment operator = with the C equality test ==. Because the C conditional statement *if(...)* doesn't distinguish between Boolean and other types, a C compiler is just as happy to accept *if(a=b)* as *if(a==b)*. In Sather, the conditional statement will only accept Boolean values, rendering impossible this kind of mistake.

Languages like Beta are also strongly typed, but not statically checkable. Consequently, some type checking must be done at run time. While this is preferable to no type checking at all, it reduces the safety of programs. For instance, there may be a typing problem in obscure code that isn't exercised by test routines. Errors not caught by the compiler can make it into final releases.

Sather distinguishes "abstract types," which represent more than one type of object, from other types, which do not. This has consequences for both the conceptual structure and the efficiency of programs. An example which has been widely discussed is the problem of the *add_vertex* routine for polygons. This is a routine which makes sense for generic polygons but does not make sense for triangles, squares, and so on. In languages which do not separate abstract types from particular implementations, you must either make all descendants implement routines that don't make sense for them, or leave out functionality in parent classes.

The Sather solution to this is based on abstract types. The Sather libraries include the abstract class $POLYGON, which defines the abstract interface that all polygons must provide. It also includes the descendant class POLYGON, which implements generic polygons. The *add_vertex* routine is defined in POLYGON but is not defined in $POLYGON. TRIANGLE and SQUARE, therefore, do not need to define it.

Run-time dispatching is only done for calls on variables declared by abstract types. The Sather compiler is, itself, a large program written in Sather which uses a lot of dispatching. The performance consequences of abstract types were studied by comparing a version of the compiler, in which all calls were dispatched, to the standard version (Lim and Stolcke, 1991). The use of explicit typing causes one-tenth the number of dispatches and an 11.3 percent reduction in execution time.

Separate Implementation and Type Inheritance

In most object-oriented languages, inheritance defines the subtype relation and causes the descendant to use an implementation provided by the ancestor. These are quite different notions; confusing them often causes semantic problems. For example, one reason why Eiffel's type system is difficult to check is that it mandates "covariant" conformance for routine argument types (Meyer, 1992). This means a routine in a descendant must have argument types which are subtypes of the corresponding argument types in the ancestor. Because of this choice, the compiler can't ensure argument expressions conform to the argument type of the called routine at compile time. In Sather, inheritance from abstract classes defines subtyping while inheritance from other classes is used solely for implementation inheritance. This allows Sather to use the statically type-safe contravariant rule for routine argument conformance.

Multiple Inheritance

In Smalltalk and Objective-C, each class only inherits from a single class. In Sather, classes can inherit from an arbitrary number of classes, a property called "multiple inheritance." This is important because it commonly occurs in modeling physical types. For example, there might be types representing "means of transportation" and "major expenditures." The type representing "automobiles" should be a descendant of both of these. In Smalltalk or Objective-C, which only support single inheritance, you'd be forced to make all "means of transportation" be "major expenditures" or vice versa.

Garbage Collection

Languages derived from C are usually not "garbage collected," making you responsible for explicitly creating and destroying objects. Unfortunately, these memory-management issues often cut across natural abstraction boundaries. The objects in a class usually don't know when they are no longer referenced, and the classes which use those objects shouldn't have to deal with low-level memory-allocation issues.

Memory management done by the programmer is the source of two common bugs. If an object is freed while still being referenced, a later access may find the memory in an inconsistent state. These so-called "dangling pointers" are difficult to track down because they often cause code errors far removed from the offending statement.

Memory leaks caused when an object is not freed even though there are no references to it, are also hard to find. Programs with this bug use more and more memory until they crash. Sather uses a garbage collector which tracks down unused objects and reclaims the space automatically. To further enhance performance, the Sather libraries generate far less garbage than is typical in languages such as Smalltalk or Lisp.

Sather combines flexibility with high efficiency

Interactive, Interpreted Programming

Sather combines the flexibility of an interactive, interpreted environment with very high-efficiency compiled code. During development, the well-tested library classes are typically run compiled, while the new experimental code is run interpreted. The interpreter also allows immediate access to all the built-in algorithms and data structures for experimentation. Listing Three (page 22) is an example of an interactive Sather session.

Iteration Abstraction

Most code is involved with some form of iteration. In loop constructs of traditional languages, iteration variables must be explicitly initialized, incremented, and tested. This code is notoriously tricky and is subject to "fencepost errors." Traditional iteration constructs require the internal implementation details of data structures like hash tables to be exposed when iterating over their elements.

Sather allows you to cleanly encapsulate iteration using constructs called "iters" (Murer, Omohundro, and Szyperski, 1993) that are like routines, except their names end in an exclamation point (!), their bodies may contain *yield* and *quit* statements, and they may only be called within loops. The Sather loop construct is simply: *loop...end.* When an iter yields, it returns control to the loop. When it is called in the next iteration of the loop, execution begins at the statement following the *yield.* When an iter quits, it terminates the loop in which it appears. All classes define the iters *until!(BOOL)*, *while!(BOOL)*, and *break!* to implement more traditional looping constructs. The integer class defines a variety of useful iters, including *upto!(INT):INT*, *downto!(INT):INT*, and *step!(num,step:INT):INT.* Listing Four (page 22) shows how *upto!* is used to output digits from 1 to 9.

Container classes, such as arrays or hash tables, define an iter *elts!:T* to yield the contained elements and an iter called *set_elts!(T)* to insert new elements. Listing Four shows how to set the elements of an array to successive integers and then how to double them. Notice that this loop doesn't have to explicitly test indexes against the size of the array.

The tree classes have iters to yield their elements according to the "pre," "post," and "in" orderings. The graph classes have iters to yield the vertices according to depth-first and breadth-first search orderings.

The Implementation

The first version of the Sather compiler was written in Sather by Chu-Cheow Lim and has been operational for several years. It compiles into C code and has been ported to a wide variety of machines. It is a fairly large program with about 30,000 lines of code in 183 classes (this compiles into about 70,000 lines of C code).

Lim and Stolcke extensively studied the performance of the compiler on both MIPS and SPARC architectures. Because the compiler uses C as an intermediate language, the quality of the executable code depends on the match of the C-code templates used by the Sather compiler to the optimizations employed by the C compiler. Compiled Sather code runs within 10 percent of the performance of handwritten C code on the MIPS machine and is essentially as fast as handwritten C code on the SPARC architectures. On a series of benchmark tests (towers of Hanoi, 8 queens, and the like) Sather performed slightly better than C++ and several times better than Eiffel. The new compiler performs extensive automatic inlining and so provides more opportunities for optimization than typical handwritten C code.

The Libraries

The Sather libraries currently contain several hundred classes, and new ones are continually being written. Eventually, we hope to have efficient, well-written classes in every area of computer science. The libraries are covered by an unrestrictive license which encourages the sharing of software and crediting of authors, without prohibiting use in proprietary and commercial projects. Currently there are classes for basic data structures, numerical algorithms, geometric algorithms, graphics, grammar manipulation, image processing, statistics, user interfaces, and connectionist simulations.

pSather

Sather is also being extended to support parallel programming. An initial version of the language, "pSather" (Murer, Feldman, and Lim, 1993), runs on the Sequent Symmetry and the Thinking Machines CM-5. pSather adds constructs for programming on a distributed-memory, shared-address machine model. It includes support for control parallelism (thread creation, synchronization), an SPMD form of data parallelism, and mechanisms to manipulate execution control and data in a nonuniform access machine. The issues which make object-oriented programming important in a serial setting are even more important in parallel programming. Efficient parallel algorithms are often quite complex and should be encapsulated in well-written library classes. Different parallel architectures often require the use of different algorithms for optimal efficiency. The object-oriented approach allows the optimal version of an algorithm to be selected according to the machine it is actually running on. It is often the case that parallel code development is done on simulators running on serial machines. A powerful object-oriented approach is to write both simulator and machine versions of the fundamental classes in such a way that a user's code remains unchanged when moving between them.

Conclusion

I've described some of the fundamental design issues underlying Sather 1.0. The language is quite young, but we are excited by its prospects. The user community is growing, and new class development has become an international, cooperative effort. We invite you join in its development!

Acknowledgments

Sather has adopted ideas from a number of other languages. Its primary debt is to Eiffel, designed by Bertrand Meyer, but it has also been influenced by C, C++, CLOS, CLU, Common Lisp, Dylan, ML, Modula-3, Oberon, Objective C, Pascal, SAIL, Self, and Smalltalk. Many people have contributed to the development and design of Sather. The contributions of Jeff Bilmes, Ari Huttunen, Jerry Feldman, Chu-Cheow Lim, Stephan Murer, Heinz Schmidt, David Stoutamire, and Clemens Szyperski were particularly relevant to the issues discussed in this article.

References

ICSI Technical reports are available via anonymous ftp from ftp.icsi.berkeley.edu.

Lim, Chu-Cheow and Andreas Stolcke. "Sather Language Design and Performance Evaluation." *Technical Report TR-91-034*. International Computer Science Institute, Berkeley, CA, May 1991.

Meyer, Bertrand. *Eiffel: The Language*. New York, NY: Prentice Hall, 1992.

Murer, Stephan, Stephen Omohundro, and Clemens Szyperski. "Sather Iters: Object-oriented Iteration Abstraction." *ACM Letters on Programming Languages and Systems* (submitted), 1993.

Murer, Stephan, Jerome Feldman, and Chu-Cheow Lim. "pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation." *Technical Report TR-93-028*. International Computer Science Institute, Berkeley, CA, June 1993.

Omohundro, Stephen. "Sather Provides Non-proprietary Access to Object-oriented Programming." *Computers in Physics*. 6(5):444-449, 1992.

Omohundro, Stephen and Chu-Cheow Lim. "The Sather Language and Libraries." *Technical Report TR-92-017*. International Computer Science Institute, Berkeley, CA, 1991.

Schmidt, Heinz and Stephen Omohundro. "CLos, Eiffel, and Sather: A Comparison," in *Object Oriented Programming: The CLOS Perspective*, edited by Andreas Paepcke. Boston, MA: MIT Press, 1993.

DDJ

To vote for your favorite article, circle inquiry no. 3.

Listing One *(Text begins on page 18.)*

```
class STACK(T) is
   -- Stacks of elements of type T.
   attr s:ARR(T);         -- An array containing the elements.
   attr size:INT;         -- The current insertion location.

   is_empty:BOOL is
      -- True if the stack is empty.
      res := (s=void or size=Ø) end;

   pop:T is
      -- Return the top element and remove it. Void if empty.
      if is_empty then res:=void
      else size:=size-1; res:=s[size]; s[size]:=void end end;

   push(T) is
      -- Push arg onto the stack.
      if s=void then s:=#ARR(T)(asize:=5)
      elsif size=s.asize then double_size end;
      s[size]:=arg; size:=size+1 end;

   private double_size is
      -- Double the size of `s'.
      ns::=#ARR(T)(asize:=2*s.asize); ns.copy_from(s); s:=ns end;

   clear is
      -- Empty the stack.
      size:=Ø; s.clear end

end; -- class STACK(T)

class FOO is
   bar is
      s1:STACK(CHAR); s1.push('a');
      s2:STACK(STR);  s2.push("This is a string.") end;
end;
```

Listing Two

```
abstract class $POLYGON is
   ...
   number_of_vertices:INT;
end;

class TRIANGLE is
   inherit $POLYGON;
   ...
   number_of_vertices:INT is res:=3 end;
end;

class SQUARE is
   inherit $POLYGON;
   ...
   number_of_vertices:INT is res:=4 end;
end;

class FOO2 is
   bar2 is
      s:STACK($POLYGON);
      ...
      n:=s.pop.number_of_vertices;
      ...
   end;
end;
```

Listing Three

```
>5+7
12

>4Ø.intinf.factorial
815915283247897734345611269596115894272ØØØØØØØØØ

>#OUT + "Hello world!"
Hello world!

>v::=#VEC(1.Ø,2.Ø,3.Ø); w::=#VEC(1.Ø,2.Ø,3.Ø);
>v+w
#VEC(2.Ø, 4.Ø, 6.Ø)

>v.dot(w)
14.Ø

>#ARRAY(STR)("grape", "cherry", "apple", "plum", "orange").sort
#ARRAY(STR)("apple","cherry","grape","orange","plum")
```

Listing Four

```
>loop #OUT+1.upto!(9) end
123456789

>a::=#ARRAY(INT)(asize:=1Ø)
>loop a.set_elts!(1.upto!(1Ø)) end
>a
#ARRAY(INT)(1,2,3,4,5,6,7,8,9,1Ø)

>loop a.set_elts!(2*a.elts!) end
>a
#ARRAY(INT)(2,4,6,8,1Ø,12,14,16,18,2Ø)
```

End Listings