The Sather Language and Libraries

Stephen M. Omohundro

International Computer Science Institute

Berkeley, California

- Language goals and features
- Objects, types, classes
- The interpreter
- Iters and bound routines

Features from Other Languages

- ***Efficiency:*** C, C++, Fortran
- ***Elegance, Encapsulation, Safety:*** Eiffel, Clu
- ***Interactive Environment, Higher-order functions:*** Common Lisp, Scheme, Smalltalk

Sather 1.0 Features

- Statically-checked strong typing
- Object-oriented dispatch
- Parameterized classes
- Multiple-inheritance
- Separate subtyping and implementation inheritance
- Garbage collection
- Iteration abstraction, like structured co-routines
- Exception handling
- Higher-order routines and iters
- Assertions, pre-conditions, post-conditions, class invariants
- Interactive interpreted environment
- Efficiently compiles into and links with C
- Literal form for arbitrary data structures

Reusability

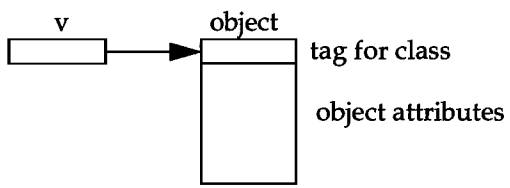
- Traditional language libraries allow *new code* to call *old code* but do not make it easy for *old code* to call *new code*.

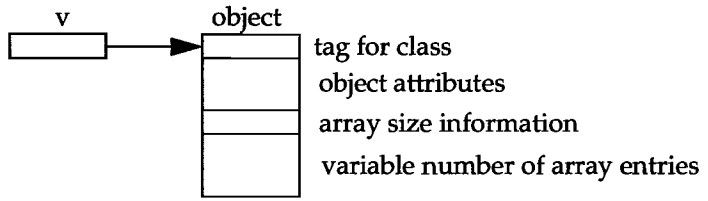
- ***Parameterized classes:*** Compile-time mechanism. Old classes may have type parameters which may be instantiated to new types.

- ***Object-oriented dispatch:*** Run-time mechanism. Old code makes calls by dispatching on the object type. New object types cause new code to be called.

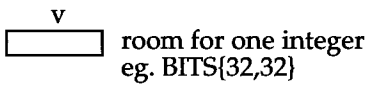
Objects

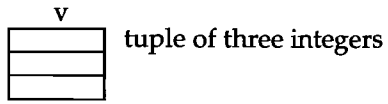
- ***Reference objects:*** passed by reference, tagged, may have dynamically sized array portion





- ***Value objects:*** passed by value, fixed but arbitrary size and alignment, passed on the stack, BITS, tuples



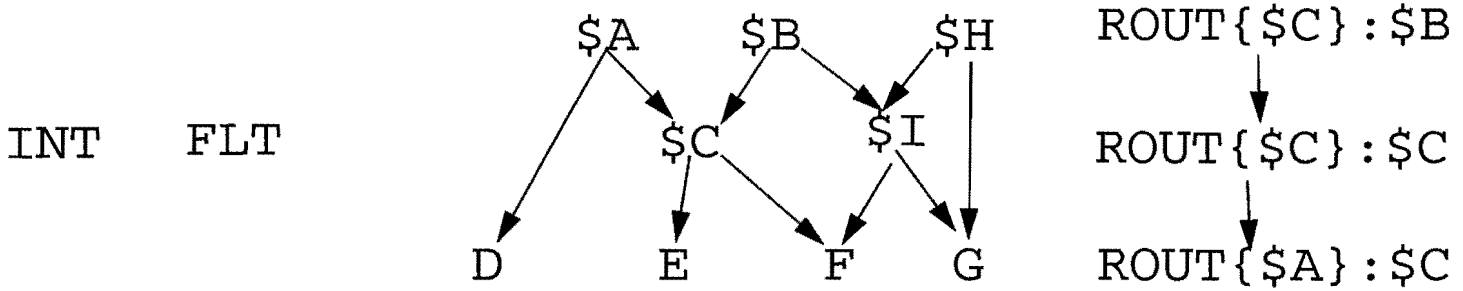


- ***Bound objects:*** object-oriented version of closures, bind together a routine or iter with some of its arguments

Statically-Checked Strong Typing

Every object and every variable has a ***type***.

The ***type graph*** is a DAG defining ***conformance***.



Fundamental typing rule: *An object's type must conform to the declared type of a variable that holds it.*

- ***Abstract types:*** Sets of reference objects
- ***Reference types:*** A single type of reference object
- ***Value types:*** A single type of value object
- ***Bound types:*** Sets of contravariantly conforming bound objects.
- ***External types:*** Interaces to other languages.

Classes

- ***Reference classes:*** Define the features of reference objects.

```
class A is ... end
```

- ***Abstract classes:*** Define a common interface to a set of reference object types.

```
abstract class $B is ... end
```

- ***Value classes:*** Define the features of value objects.

```
value class C is ... end
```

- ***External classes:*** Define the interface to object code from another language.

```
external class D is ... end
```

Parameterized Classes

```
class STACK{T} is
   attr arr:ARRAY{T};
   push(T) is ... end;
   pop:T is ... end;
   ...
end;



class FOO is
   bar is
      s:STACK{INT}; s.push(15) end;

   baz is
      s:STACK{STR}; s.push("This is a string.") end
end;
```

Object-Oriented Dispatch

```
abstract class $POLYGON is
   ...
   abstract 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 FOO is
   s:STACK{$POLYGON};
   ...
      n::=s.pop.number_of_vertices;
   ...
end;
```

The Interpreter

```
unix_prompt>si
>5+7
12

>1923838.is_prime
false

>1923838.type
INT

>doc INT::is_prime
is_prime:BOOL -- True if self is a prime number.

>routs INT
abs band bcount beqv binary_str bit bnand bnor bnot boole bor bxor char
cube digit_char evenly_divides extended_gcd factorial flt fltd flte
from_int from_str gcd hex_str high_bits highest_bit int is_bet ...

>doc INT::is_*
is_bet is_eq is_eq is_eq is_eq is_even is_geq is_gt is_leq is_lt is_neg
is_neq is_non_neg is_non_pos is_odd is_pos is_prime
is_relatively_prime_to is_ugeq is_ugt is_uleq is_ult is_zero

>classes INT*
INT INTFIX INTINF
```

Example: Vectors

```
>v::=#VEC(1.0, 2.0, 3.0)

>v
#VEC(1.0, 2.0, 3.0)

>w::=#VEC(4.0, 5.0, 6.0)

>v.dot(w)
32.0

>v+w
#VEC(5.0, 7.0, 9.0)

>v.square_length
14.0
```

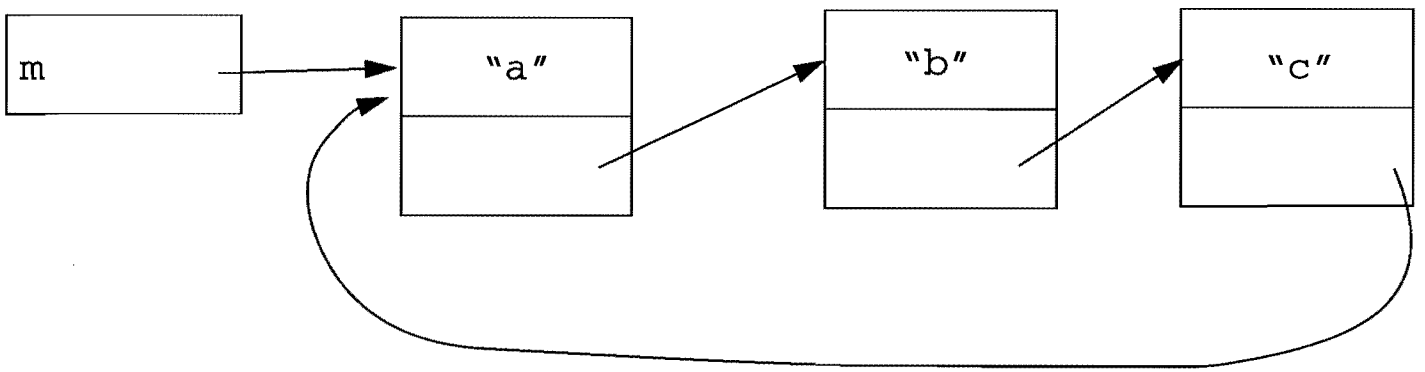
Describing Circular Data Structures

```
>class LINK is
>> attr name:STR;
>> attr link:LINK end

>#LINK("single",)
#LINK("single",void)

>m::=#LINK("a",#LINK("b",#LINK("c",#1)))

>m.link.link.link=m
true
```



Iteration Abstraction

- ***Iteration*** is central to computation.
- ***Initialize, increment, test*** loop variables
- Prone to ***fencepost*** errors
- Often relies on ***implementation details*** of a class
- Should be ***encapsulated*** as part of a class's interface
- Existing constructs: *CLU iters, cursors, riders, streams, series, generators, coroutines, blocks, closures,* and *lambda expressions.*

Sather Iters

- Sather iters are like routines, but they may ***yield*** in addition to returning. From the INT class:

```
iter times:INT is
   -- Yield successive integers from 0 upto self-1.
   loop if res<self then yield end; res:=res+1 end end end;
```

- Iters may only be called in ***loops***. When an iter yields, the loop continues, but the iter retains its state. The next call starts after the yield.
- The loop is broken when the iter returns.
- The statement:

```
loop #OUT & 15.times & " " end
```

causes the output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Array iters

```
class ARRAY{T} is
   ...
   iter elts:T is
      -- Yield the successive elements of self.
      loop res:=[asize.times]; yield end end;

   iter set_elts(T) is
      -- Set successive elements of self to arg.
      loop [asize.times]:=arg; yield end end;
end;
```

Examples:

```
loop set_elts(7) end;              -- Set all elements to 7.

loop set_elts(2*elts) end;         -- Double all elements.

loop set_elts(a.elts) end;         -- Make self a copy of a.

loop x:=sum(a.elts*b.elts) end;    -- Dot product of two arrays.
```

Sieve of Eratosthenes

```
iter sieve(INT):BOOL is
   -- Sieve out successive primes.
   d::=arg; res:=true;
   loop yield;
      if d.divides(arg) then res:=false
      else res:=sieve(arg) end end end;

iter primes:INT is
   -- Yield successive primes.
   res:=2; loop if sieve(res) then yield end; res:=res+1 end end;
```

Iters for other data structures

- ***Tree*** classes have iters to yield the nodes in *pre-order, in-order,* and *post-order.*
- ***Graph*** classes have iters to yield the vertices in *depth-first* and *breadth-first* orders.
- ***Container classes*** such as *hash tables, bit-vectors, skip-lists, stacks, queues, etc.* provide iters to yield and insert elements.
- Acts as a *lingua-franca* for passing sets of items around.
- Iters are essentially ***structured co-routines***.

Bound Routines and Iters

- The Sather analog of ***function pointers*** and ***closures***.
- No nested contexts, uses only ***encapsulated interfaces***.
- Binds together a routine or iter and values for some of its arguments.
- May ***dispatch*** at the point of call.
- Example:

```
>br::=#ROUT(2.plus(_));              -- A bound routine which adds 2.

>br.call(5)
7

>br.call(12)
14
```

Bound routine example: compose

```
compose_helper(r1,r2:ROUT{INT}:INT, INT):INT is
   -- 'r1' applied to 'r2' applied to arg.
   res:=r1.call(r2.call(arg)) end;

compose(r1,r2:ROUT{INT}:INT):ROUT{INT}:INT is
   -- The composition of 'r1' and 'r2'.
   res:=#ROUT(compose_helper(r1,r2,_)) end;
```

Example of composing *square root* and *absolute value:*

```
>r::=compose(#ROUT(_:INT.sqrt), #ROUT(_:INT.abs))

>r.call(-9)
3
```

The Sather Library

- Currently several hundred classes.
- Eventually want efficient classes in every area of computer science. The library is generally available by anonymous ftp. Unrestrictive license encourages sharing software and crediting authors without prohibiting use in proprietary projects.
- Current directories: *base, connectionist, data_structure, geometry, grammar, graphics, image, numerical, statistics, user_interface.*

Amortized Doubling

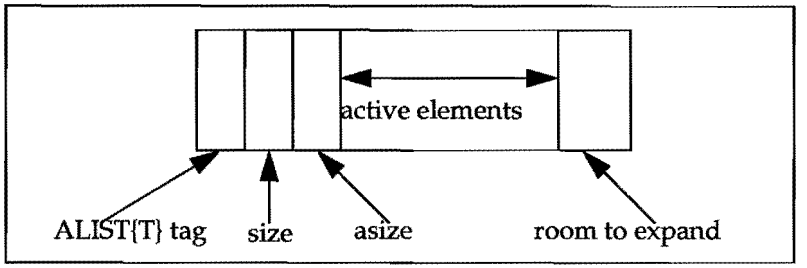
- Strings, stacks, hash tables, queues, etc. all need incremental space allocation .
- We initially allocate a fixed-sized array.
- Each time it fills up, we double the size.
- If we make n insertions we end up with:
- $1+2+4+8+...+n<2n$ copying operations.
- The amortized cost per insertion is 2.
- Only log n chunks must be allocated and garbage collected.

The Simplest Sather Program

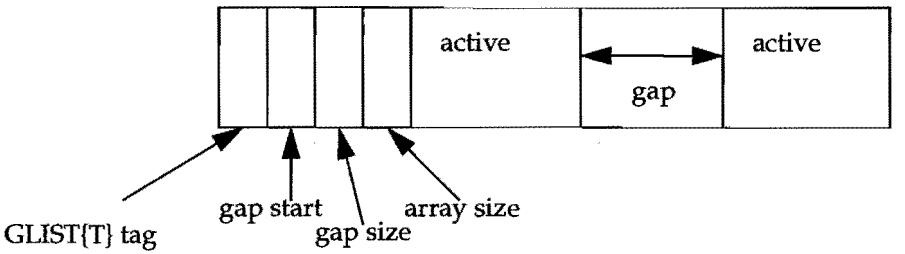
```
class HELLO is
   main is #OUT & "Hello world.\n" end
end
```

ALIST{T} and GLIST{T}

- The most used container class is ALIST{T}. This implements a stack of objects of type T which grows by amortized doubling and which allows array indexed access. More efficient than linked lists.



- If arbitrary insertion and deletion are needed use GLIST{T} objects with a movable gap. On insertion or deletion the gap is moved to the location in question. When the gap is filled, space is reallocated by amortized doubling.



Maps and Sets via Hash Tables

- Two basic abstractions are sets of elements of some type and maps from one set to another.
- Most useful form are built with hash tables. Made very fast using very inexpensive hash function, amortized doubling, load factor of a half, insertion algorithm that supports deletion. Efficiently support set operations such as union, intersection, symmetric difference, etc.

- **HSET{T}** represents sets of objects of type T.
- **HMAP{S,T}** represents maps from type S to type T.

Graphs

- Usual graph representation would make the vertices be objects and the edge list an attribute. All vertices would inherit from a generic graph vertex class.
- Operations like "transitive closure" are functions which take one graph and produce another. The vertices of these two graphs are the same.
- In the standard representation, vertices can only be a member of one graph for inheritance to work.
- Instead: A graph is a hash table which hashes vertex objects to edge lists. Each vertex can be in as many graphs as desired and graph operations can take one graph as input and produce another as output.

- Same idea is used in "union-find" class for maintaining sets of sets which support efficient set union and element find operations.

Random number generation

- Want class `RND` to produce random numbers from a variety of different distributions, eg. normal, binomial, gamma, Poisson, geometric, etc.
- Want the standard generator to be very fast.
- Want to rerun code changing generators to ensure there is no difference.
- Want to combine two generators to form better generators.
- Make `RNDGEN` class which all generators inherit from. Which is used is determined by dispatch in `RND`.
- Make classes like: `COMB_RNDGEN{G1,G2}` whose objects point to two other generators and combine them in such a way as to make better random variables.

Mappings between Vector Spaces

- Connectionist nets, linear fits, non-linear regression all represent mappings from one vector space to another. Make them all inherit from `VECMAP`.
- COMPOSITION_VECMAP{M1,M2}represents the composition of two maps.
- PRODUCT_VECMAP{M1,M2}represents the product of two maps.
- SUBSET_VECMAP maps a vector to a permutation of some of its components.
- CONSTANT_VECMAP represents a constant vector.
- These classes may be combined like tinkertoys to build up arbitrarily complex maps, each of which still obeys the defining interface.

Compiler

- Written in Sather (with some C code, eg. the parser uses YACC).
- Fairly large program: About 27000 lines of Sather code in 183 classes.
- Makes 15 passes, yet typically much faster than the C compilation. Works entirely in memory (let VM deal with the disk).
- Dot-sather file, lexes and parses source, check for duplicate classes and deal with C class, resolve parameterized classes, resolve inheritance and UNDEFINE, resolve array classes, evaluate compiler constants, compute attribute offsets, build ancestor DAG, output C, output hash tables, generate makefile and compile the C.
- Object oriented structure very natural since everything is built from code trees which use dynamic dispatch extensively.
- Use of pmake to do the C compiles in a multi-workstation environment. Ports to several machines, take about an afternoon.

Parallel Sather

- Encapsulation and reusability are even more important in parallel code than in serial.
- Uses library classes that are machine dependent, but user classes that are not.
- Adds constructs for:
  1) Executing and blocking separate threads. Indicating which processor execution should occur on.
  2) Locking collections of variables and thread synchronization.
  3) Safe: read-only access to shared data.
- Implementations on Sequent Symmetry, CM-5, Sparc.