

# The Sather Language and Libraries

Stephen M. Omohundro

International Computer Science Institute

Berkeley, California

- Language goals and description
- Library design
- Examples
- Using the Sather tools
- Future directions -- parallel version

# Addresses for Obtaining Sather

Sather mailing list (a couple of hundred participants):

- “sather-request@icsi.berkeley.edu” to join mailing list.
- “sather@icsi.berkeley.edu” to submit articles to mailing list.
- “sather-admin@icsi.berkeley.edu” to submit bug reports/suggestions.

Anonymous ftp sites for the obtaining the Sather distribution:

- “icsi-ftp.berkeley.edu” in the U.S.A.
- “gmdzi.gmd.de” in Europe.

Compressed tar file is 3 megabytes, installed system is 18 megabytes.

# 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.

# Sather Goals

- Satisfy practical needs for efficient, reusable code. Something in between Eiffel and C++.
- Parameterized classes.
- Object-oriented dispatch.
- Multiple-inheritance.
- Garbage collection.
- Compile into efficient portable C and link with existing C.
- Strong typing.
- Small set of keywords, simple and clean syntax.
- Library of efficient, well-written classes for many of the useful algorithms in computer science.
- Extensible to parallel environments.

# The Sather Syntax

```
class_list: | class | class_list ';' | class_list ';' class
class: CLASS IDENTIFIER opt_type_vars IS feature_list END
opt_type_vars: | '{' ident_list '}'
ident_list: IDENTIFIER | ident_list ',' IDENTIFIER
feature_list: feature | feature_list ';' | feature_list ';' feature
opt_private: | PRIVATE
feature: type_spec | opt_private var_dec | opt_private routine_dec
      | opt_private shared_attr_dec | opt_private const_attr_dec
type_spec: IDENTIFIER | '$' type_spec | IDENTIFIER '{' type_spec_list '}'
type_spec_list: type_spec | type_spec_list ',' type_spec
var_dec: ident_list ':' type_spec
shared_attr_dec: SHARED var_dec | SHARED var_dec ':' expr
var_dec_list: var_dec | var_dec_list ';' | var_dec_list ';' var_dec
routine_dec: IDENTIFIER IS statement_list END
      | IDENTIFIER '(' var_dec_list ')' IS statement_list END
      | single_var_dec IS statement_list END
      | IDENTIFIER '(' var_dec_list ')' ':' type_spec IS statement_list END
const_attr_dec: CONSTANT var_dec ':' expr
statement_list: | statement | statement_list ';' | statement_list ';' statement
statement: IDENTIFIER | local_dec | assignment | conditional | loop | switch
      | BREAK | RETURN | call | assert | debug
local_dec: var_dec | var_dec ':' expr
assignment: expr ':' expr
conditional: IF expr THEN statement_list elsif_part else_part END
elsif_part: | elsif_part ELSIF expr THEN statement_list
else_part: | ELSE statement_list
loop: UNTIL expr LOOP statement_list END | LOOP statement_list END
switch: SWITCH expr when_part else_part END
when_part: | when_part WHEN exp_list THEN statement_list
assert: ASSERT '(' IDENTIFIER ')' expr END
debug: DEBUG '(' IDENTIFIER ')' statement_list END
call: IDENTIFIER '(' exp_list ')' | cexpr '.' IDENTIFIER arg_vals
      | type_spec ':' IDENTIFIER arg_vals
arg_vals: | '(' exp_list ')'
exp_list: expr | exp_list ',' expr
expr: cexpr | nexpr
cexpr: IDENTIFIER | CHAR_CONST | INT_CONST | REAL_CONST | BOOL_CONST | STR_CONST
      | call | aref | '(' expr ')'
nexpr: NOT expr | expr '<' expr | expr '>' expr | expr '<=' expr | expr '>=' expr
      | expr '=' expr | expr '/=' expr | expr AND expr | expr OR expr | '-' expr
      | '+' expr | expr '+' expr | expr '-' expr | expr '*' expr | expr '/' expr
aref: cexpr '[' exp_list ']' | '[' exp_list ']'
```

# All Code is in Classes

- Two level name space.
- Every attribute and routine name is relative to the name of the class it is in.
- Encapsulation at the class level.
- Different classes can use the same name for similar features. (Conceptually nice in some cases, essential for dispatching.)
- Typical program might have several hundred classes, and each class twenty or thirty features. A two-level hierarchy feels about right for this level of complexity.
- Incorporating classes from different people: Just need to make sure the class names are different. Environment tools for changing all occurrences of a class name.

## Example: Stack Class

```
class STACK{T} is
  -- Stacks.

  s:ARRAY{T};      -- Holds the stack elements.
  size:INT;        -- The current insertion location.

  create:SELF_TYPE is
    -- An empty stack.
    res:=new; res.s:=ARRAY{T}::new(5);
  end; -- create

  pop:T is
    -- Return the top element and remove it. 'Void' if empty.
    if empty then res:=void
    else size:=size-1; res:=s[size]; s[size]:=void
    end;
  end; -- pop

  inline empty:BOOL is
    -- 'True' if stack is empty.
    res := (size=0)
  end; -- empty
```

# Stack continued

```
push(e:T) is
  -- Push 'e' onto the stack.
  if size>=s.asize then
    os:ARRAY{T}:=s; s:=os.extend(2*os.asize);
    os.clear;
  end; -- if
  s[size]:=e; size:=size+1;
end; -- push

top:T is
  -- The value of the top of the stack. 'Void' if empty.
  if empty then res:=void
  else res:=s[size-1]
  end; -- if
end; -- top

clear is
  -- Empty the stack.
  size:=0; s.clear;
end; -- clear

end; -- class STACK{T}
```

# Dispatch Example

```
class POLYGON is
  ...
  number_of_vertices:INT is end;
end; -- class POLYGON

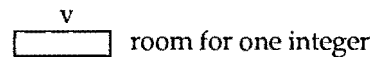
class TRIANGLE is
  POLYGON;
  ...
  number_of_vertices:INT is res:=3 end;
end; -- class TRIANGLE

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

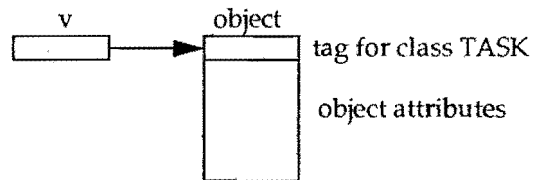
class FOO is
  s:STACK{$POLYGON};
  ...
  nv:INT:=s.pop.number_of_vertices;
  ...
end; -- class FOO
```

# Objects

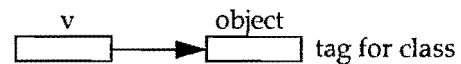
Basic types: (CHAR, BOOL, INT, REAL, DOUBLE):



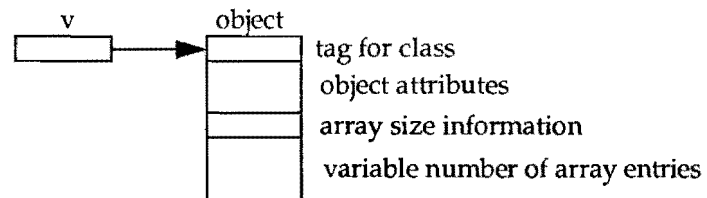
Typical objects:



Empty objects:



Array objects:



# Type declaration

- *Non-parameterized:*

```
a:INT; b:FOO;
```

- *Parameterized:*

```
a:ARRAY{INT}; b:STACK{QUEUE{STR}};
```

- *Type parameter, within the class definition of PCLASS{T} can say:*

```
a:T;
```

- *Dispatched:*

```
a:$FOO;
```

- *Special:*

```
a:SELF_TYPE, b:UNDEFINE, c:$OB, d:F_OB
```

# Type Conformance

A variable can hold an object only if the runtime type of the object *conforms* to the declared type of the variable.

t1 conforms to t2 if:

- *Non-parameterized types*: If t2 is FOO then t1 must also be FOO.
- *Dispatched non-parameterized types*: If t2 is \$FOO then t1 must be a subtype of FOO.
- *Parameterized types*: If t2 is FOO{A,\$B,C} then t1 must be FOO{A,D,C} where D is a subtype of B.
- *Dispatched parameterized types*: If t2 is \$FOO{A,\$B,C} then t1 must be BAR which inherits from something conformant to FOO{A,\$B,C}.
- *Basic types*: If t2 inherits from INT, so must t1.

## Children can UNDEFINE features

Only features which are actually used in a dispatched fashion need be defined in all descendents.

Shows the semantic importance of the "\$" specifications in addition to efficiency considerations.

Eg. Class POLYGON might define "add\_vertex". Descendent class "SQUARE" would "UNDEFINE" this routine. This allows:

```
p:POLYGON; p.add_vertex;
```

but disallows:

```
p:$POLYGON; p.add_vertex;
```

# Differences with Eiffel

- Eliminated many less central features and keywords.
- Simplified the syntax and inheritance rules.
- Introduced specification of dispatched types (\$FOO vs. FOO).
- Make efficient arrays a central part of the language:



- Introduced declaration of locals at point of use.
- Introduced shared variables and direct feature access.
- Made all changes to a variable be by assignment.
- Eiffel exception mechanism requires setjump/longjmp.
- Eiffel uses Dijkstra garbage collector, no register pointers.

## Eliminated Eiffel Keywords:

as

Create

div

expanded

feature

implies

invariant

like

name

old

redefine

require

unique

xor

BITS

deferred

do

export

Forget

inherit

^

local

Nochange

once

rename

rescue

variant

Clone

define

ensure

external

from

infix

language

mod

obsolete

prefix

repeat

retry

Void

# Preliminary Speed Comparison with Eiffel

1 million operations each

	SATHER:	EIFFEL:
array assignment:	1.260u 0.110s	4.670u 1.050s
array assignment dispatched:	1.950u 1.050s	4.670u 1.050s
routine call:	1.310u 0.080s	21.310u 75.540s
routine call dispatched:	2.340u 0.070s	21.310u 75.540s
feature assignment:	1.920u 0.090s	26.860u 75.110s
feature assignment dispatched:	2.940u 0.090s	26.860u 75.110s

## Differences with C++

- Strong typing.
- Garbage collection.
- Parameterized classes.
- Tagged objects allows deep copy and deep storage in files.
- Simpler and cleaner syntax.
- Automatic compilation management.
- Cleaner inheritance mechanism (no virtual function declarations).

# Class Definition

*Non-parameterized classes:*

```
class SIMPLE is
  <feature definitions>
end;
```

*Parameterized classes:*

```
class PARAM{T1,T2} is
  <feature definitions>
end;
```

# Class features

- *Constant attributes:*

```
constant pi:REAL:=3.141592;
```

- *Shared attributes:*

```
shared a:INT;
```

- *Object attributes:*

```
b:REAL;
```

```
c:INT;
```

```
d:ARRAY{DOUBLE};
```

- *Routines:*

```
test_rout(a,b:INT; c:FOO):REAL is
```

```
  <rout_body>
```

```
end;
```

- *Inheritance:* Just list class name.

## Other keywords

- The keyword `private` disallows external access.
- `a.self` is a pointer to `a`.
- `a.type` is `a`'s class tag.
- `a.new` is an uninitialized object of the same type as `a`.
- `a.copy` is a duplicate of `a`.
- `a.extend(size)` extends the size of an array object.
- `res` is a variable which holds the return result in routines.

# Statements

- *Local declarations:* `a:INT; b:FOO:=c;`

- *Assignments:* `a:=5; b:=c.new;`

- *Conditionals:*

```
if <boolean expression> then
  <statement list>
elsif <boolean expression> then
  <statement list>
else
  <statement list>
end;
```

- *Loops:*

```
until <boolean expression> loop
  <statement list>
end;
```

- *Break from within loops:* `break`

- *Return from routines:* `return`

# Statements Continued

- *Switch statement:*

```
switch <expression>
when <expression list> then
  <statement list>
when <expression list> then
  <statement list>
else
  <statement list>
end;
```

- *Routine calls:* a.rout(5,87); c.d.e.f; FOO::rout1(4);
- *Assertions:* assert (foo) a<200 and a>0 end;
- *Debugging statements:* debug (foo) <statement list> end;

# Expressions

- *Constants:* 1, true, void, 'a', '\000', 15, -15, 2.3, 2.4e-19, "a string"
- *Identifiers:* a, b, res, self, type, copy, new
- *Dotted expressions:* a.rout(5)
- *Class access:* CLASS::a, FOO::fun(x, y)
- *Boolean expressions:* a<b, c<=d, e/=f, a and b, a or b, c=void
- *Numerical expressions:* 1+2, 3\*5, 15.mod(7)
- *Array access:* a[2], b[3,4]
- *C routines:* C::ext\_rout(5)

# 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:
- $1+2+4+8+\dots+n < 2n$  copying operations.
- The *amortized* cost per insertion is 2.
- Only  $\log n$  chunks must be allocated and garbage collected.

# The Simplest Sather Program

```
class SIMPLEST is
  main is
    OUT::s("Hello world.") .nl;
  end;
end;
```

# Argument Modification vs. Object Return

- Often prefer the style of having routines modify an argument (self or one of the other ones) rather than creating a new object and returning it.
- Eg. In the VECTOR class there are both “plus(v:VECTOR):VECTOR” and “to\_sum\_with(v:VECTOR):VECTOR”. The first creates a new vector which is the sum of “self” and “v”, the second modifies “self” by adding in “v” and then returns it.
- The modification style doesn’t allocate new storage.
- The modification style can be applied to objects with (possibly unknown) pointers pointing to it.
- The modification style can be applied to descendent objects which it would not be possible for the routine to create (the descendent class may not have even been defined when the routine was written).

## Returning Self

- It is often very convenient to have a routine which modifies "self" also return it. This allows such modification operations to be chained and to be used in expressions.

- Eg. In the file class OUT:

```
OUT::s("Variable number ").i(num).s(" is ").r(val).nl;
```

- Eg. In the class VECTOR:

```
s:=v.to_uniform_random.to_sum_with(w).normalize.to_s;
```

- The expression becomes a series of modifications to an object.
- This style is also necessary for objects which may be resized as a result of a modification. Eg. in STR:

```
st:=st.r(a+b).s(" is ").i(fac).s(" times ").d(x).nl;
```

# Cursors

- Most collection classes have associated cursor classes for stepping through them. (eg. LIST{T}, OB\_HASH\_SET{T}, STR, etc.) with names formed by appending “\_CURSOR”. (eg. LIST\_CURSOR{T}, OB\_HASH\_SET\_CURSOR{T}, STR\_CURSOR, etc.).
- Each of these inherits from CURSOR{T} (a suggestion of Graham Dumpelton) which defines the basic operations: first:T, item:T, next:T, and is\_done:BOOL. This allows classes which combine all elements in a collection (eg. sum together a set of INT's, apply an operation to all objects in a collection, or filter out a specified subset).
- More than one cursor can point into a single collection. This is essential in many applications and is the reason one can't make the cursor a part of the collection object itself.
- String cursors are used to get more functionality than “scanf” in a more natural way to extract information from strings. Eventually a similar class will do regular expression search in strings. (Very useful as evidenced by widespread use in Emacs.)

# Test Classes

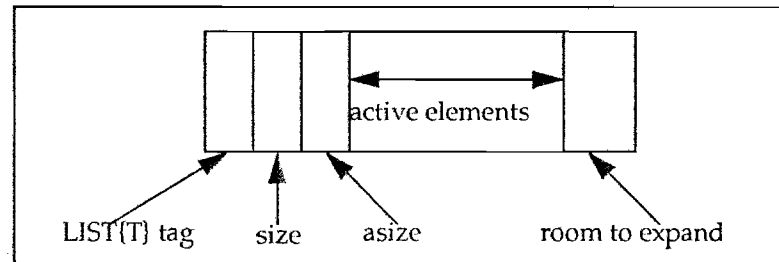
- Each class (or collection of related classes) has an associated test class whose name ends in “\_TEST” (eg. STR\_TEST, LIST\_TEST, etc.).
- These test classes all inherit from “TEST” (idea suggested by Bob Weiner) which defines:

```
class_name (nm:STR)
test (doc, does, should:STR)
unchecked_test (doc, does, should:STR)
finish
```

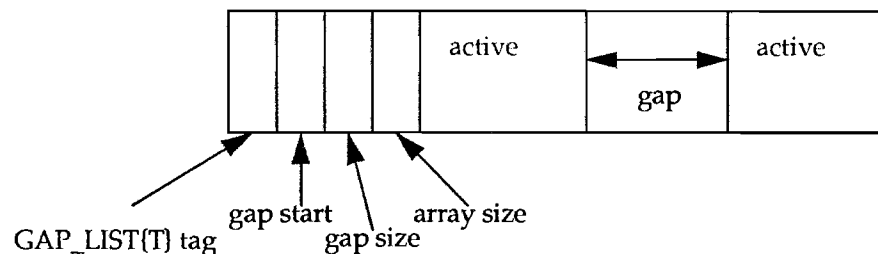
- Automatically numbers formats and keeps track of whether tests are passed or not.
- The test class should call every routine in the class being tested (not necessarily performing an extensive test, but at least checks that it compiles and runs).
- Critical for regression testing and testing changes to the compiler.

# LIST and GAP\_LIST

- The most used container class is `LIST{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 `GAP_LIST{T}` objects. These are arrays of objects of type `T` 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 forms 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.
- INT sets: LIST{INT}, GAP\_LIST{INT}, BIT\_VECTOR, INT\_HASH\_SET.
- INT\_HASH\_MAP{T} maps from non-negative integers to objects of type T.
- OB\_HASH\_SET and OB\_HASH\_MAP{T} hash on the pointer of ordinary objects. Essential for building clean structures.
- STR\_HASH\_SET, STR\_HASH\_MAP{T} very useful for efficient string manipulation tasks.
- GENERAL\_HASH{T} for implementing general hash tables with deletion using arbitrary hash function and equality test.

# Sorting Classes

- QUICKSORT{T} sorts LIST{T} in which T may be INT, REAL or DOUBLE.
- Classes may inherit from QUICKSORT{T} and redefine the “compare” routine to sort any other kind of object.
- This approach does not require the classes of the objects being sorted to be modified (eg. by requiring them to define “compare”) and so may be applied to already existing classes.
- Eg. STR\_SORT sorts LIST{STR} into alphabetical order by using quicksort.
- Usage:

```
l:LIST{STR};  
STR_SORT::sort(l)
```

## Real Life Task: Making a List of Referees

- Had to select three reviewers each for about one hundred papers.
- Two files required: A list of papers, each name followed by three reviewers:

Paper Number 1  
Albus Williams Zipser

Paper Number 2  
Bengle Nipsly Vesalius

- The second listing each reviewer in alphabetical order, with the list of papers he was to review in alphabetical order:

Albus  
Paper Number 1  
Paper Number 7  
Paper Number 89

Bengle  
Paper Number 2

## Referee Task: About 15 Minutes to write!

- Use LIST's, STR\_CURSOR, STR\_HASH\_MAP's, and STR\_SORT, all of which existed, were well tested, and easy to use.
- Code took about 15 minutes to write.
- Best part: Worked the first time!
- Simple example of the power of reusable encapsulated abstractions.
- General experience is that when you first start working in a domain, writing object oriented code takes longer. You have to think hard about how to cut things into classes.
- Each class takes longer to write because you include a whole abstract interface rather than just the things you need at the moment.
- But eventually you discover that most of what you need is already there! You get a great feeling of power as you simply glue together existing pieces. The resulting code is much easier to debug and to modify.

# Referee Task Code

```
class NIPS_CONVERT is

  main is
    -- Convert a NIPS paper list to a reviewer list.
    l:LIST{STR}:=l.create;-- A list of the lines in the input.
    until IN::check_eof loop l:=l.push(IN::get_s) end;

    sm:STR_HASH_MAP{LIST{STR}}:=sm.create;
      -- A list of papers for each reviewer.
    i:INT; until i+2>=l.size loop -- Loop over papers.
      sc:STR_CURSOR:=l[i+1].cursor;
      k:INT:=0; until k=3 loop -- Loop over 3 reviewers.
        rev:STR:=sc.get_word;
        if not sm.test(rev) then sm.insert(rev,LIST{STR}::create)end;
        sm.insert(rev, sm.get(rev).push(l[i]));
        -- Put paper on reviewer's list.
        k:=k+1
      end; -- loop
      i:=i+3
    end; -- loop
```

# Referee Task Code, Continued

```
smc:STR_HASH_MAP_CURSOR{LIST{STR}}:=sm.cursor;
  -- Make list of rev's.
revlist:LIST{STR}:=LIST{STR}::create;
until smc.is_done loop revlist:=revlist.push(smc.key);smc.next end;
STR_SORT::sort(revlist);-- Put reviewers in alphabetical order.

j:INT; until j=revlist.size loop -- Output info for each reviewer.
  OUT::s(revlist[j]).nl.nl;
  paps:LIST{STR}:=sm.get(revlist[j]);
  -- Papers for current reviewer.
  STR_SORT::sort(paps);-- Put in alphabetical order.
  k:INT:=0; until k=paps.size loop OUT::s(paps[k]); k:=k+1 end;
  OUT::nl.nl;
  j:=j+1
end; -- loop

end; -- main

end; -- class NIPS_CONVERT
```

# 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 `RANDOM` 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 `RANDOM_GEN` class which all generators inherit from. Which is used is determined by dispatch in `RANDOM`.
- Make classes like: `COMB_RANDOM_GEN{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 `VECTOR_MAP`.
- `COMPOSITION_VECTOR_MAP {M1, M2}` represents the composition of two maps.
- `PRODUCT_VECTOR_MAP {M1, M2}` represents the product of two maps.
- `SUBSET_VECTOR_MAP` maps a vector to a permutation of some of its components.
- `CONSTANT_VECTOR_MAP` 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.

## Running the Compiler cs

To compile the "hello world" program in source file hello.sa, first make a ".sather" file containing:

```
(source_files) $SATHER_HOME/etc/hello.sa
```

Type:

```
cs hello
```

Compiler responds:

```
Created directory -- /da/om/sather/test/hello.cs  
Current version is dated Fri May 31 04:09:43 1991  
Doing make.....:" (cd /da/om/sather/test/hello.cs...)
```

```
cc -c MAIN_.c  
cc -target sun4 -c hello_22.c  
cc *.o *.a -o hello
```

Run it with:

```
hello
```

# Compiling within Emacs

- First edit the “.sather” file.
- Type “M-x compile”.
- To “Compile command:” type “cs hello”.
- Errors are output to “\*compilation\*” buffer.
- Type “C-x ” and Emacs will move to the location in the source of the first error.
- When fixed, type “M-x compile” for a very fast compile, bug fix cycle.
- To test within Emacs use “M-x shell”.

# Emacs Editing Environment

- Many functions provided within GNU emacs editor for browsing and editing Sather source code.
- Intelligent indentation automatically indents code and comments. Useful for finding syntactic bugs during editing. Templates and abbreviations expand to common statement forms.
- Edit “.sather” file and type “M-x sather-tags”. This generates a tag table for all files in a system which is used to guide searches and browsing.
- “M-.” followed by a routine name will cycle through all routines of that name in the classes in the tag table.
- “M-x sather-documentation” will generate documentation for a routine or class.
- With arguments it can also generate documentation for all classes in a system in alphabetical order or on a per file basis.
- Extensive X-windows and mouse support as well.

## Using the Debugger sdb

- Compile with the “-sdb” option. Eg. “cs -sdb hello”.
- In Emacs type “M-x sdb” and give it the name of the program.
- “C-`<space>`” in a source file will set a breakpoint.
- “run” will execute until a breakpoint.
- An arrow will appear in the buffer with the source code showing the current line number.
- Single step with “step” or step across function calls with “next”.
- Shortest unique prefix may be used for any command.
- Many browsing options to look at variables, browse objects, change values, etc.

# 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.
- Initial version written in C, new version in Sather much cleaner. Sather version was hand compiled back into C to bootstrap.
- 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.

# Debugger and browser

- Written in Sather and linked with parts of GDB from the Free Software Foundation which has the machine dependent parts and has been ported to many machines.
- Maintain a stack of objects to browse and alter.
- Visual pointer into Sather source code in Emacs for single stepping. Set breakpoints and view variables using the mouse.
- Reverts to debugging C code when executing linked in C files.

# 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.
- Pilot implementation on Sequent Symmetry.