The Sather Language

STEPHEN M. OMOHUNDRO

International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, California 94704
Phone: 415-643-9153
Internet: om@icsi.berkeley.edu
June 3, 1991

Abstract. This report describes the object-oriented programming language Sather. Sather is a simplified, optimized variant of Eiffel (Sather tower is a landmark in Berkeley). It is meant to be a working language, focussed on the practical needs of writing efficient, reusable code. It is designed for individual researchers or small groups and eliminates features which are pedantic, of purely theoretical interest, or necessary only for large or naive programming teams. Sather has clean and simple syntax, parameterized classes, object-oriented dispatch, multiple inheritance, strong typing, and garbage collection. It generates efficient C code which is easily integrated with existing code.

1. Table of Contents

2. Introduction

The two primary goals of Sather are efficiency and reusability. The Sather compiler generates efficient C code and easily links with existing C code. This allows it to have C's portablility and efficiency without adopting its complexities and dangers (some have suggested that C is really a portable assembly language).

The last few decades have shown the critical importance of reusability in the development of large software systems. Because effort is amortized over many projects, reusable code is more likely to be thoroughly debugged and to be written with more care and concern for efficiency. Reusable components allow the construction of systems of far greater complexity with given resources. The traditional libraries in languages like C allow new code to call old code, but do not easily allow old code to call new code. Sather provides two related mechanisms for old code to call new code. Parameterized classes allow the compiler to optimize such calls while object-oriented dispatch is a run-time mechanism which gives more flexibility at the expense of some efficiency. The key to both of these mechanisms is encapsulation of functionality into classes.

Sather attempts to support these two mechanisms as efficiently and simply as possible. The other aspects of the design were influenced by the envisioned usage. It is primarily aimed at small groups of competent researchers working on applications rather than systems programming. It does not attempt to be completely typesafe but does try to catch most common errors during compilation. It tries to be very compatible with user C code and C libraries and to not sacrifice the efficiency of C. It efficiently supports numerical computations. The first implementation assumes a 32 bit Unix machine with sufficient memory. Characters are assumed to be one byte and integers and pointers are assumed to be four bytes. Much of the basic system is defined in classes which are identical to user classes, giving the user flexibility to choose methods and approaches.

2.1. A Sather Example

To give the reader a sense for Sather at the outset, this section presents a simple example which uses both parameterized classes and object-oriented dispatch. The (highly concocted) application is a system for maintaining a collection of tasks on a stack. The program consists of five classes. "STACK{T}" is a parameterized class which implements stacks and might be found in a pre-existing library. In use, the type parameter T is instantiated with the type of the objects that will be put on the stack. The class "TASK" defines the objects which represent simple tasks that are characterized only by their name. The class "CLIENT_TASK" defines a refined set of objects which not only have a name but also a client number and an allowed-time limit. The class "TASK_MANAGER" specializes the "STACK" class for the job of holding task objects and instantiates its type parameter. Both "TASK" objects and "CLIENT_TASK" objects may be put on the stack, so object-oriented dispatch is used to decide which code is run on an object which is popped from the stack. Finally, "TASK_TRIAL" is a class which defines an example run of the program.

```
-- File: example.sa
-- Author: Stephen M. Omohundro
-- Created: Sun Feb 25 23:46:42 1990
-- Simple demonstration of Sather mechanisms, maintains a stack of tasks.

-----------------------------------------------------------------------

class STACK{T} is
   -- General purpose stack of objects of type T.

   arr:ARRAY{T};                -- Array holding stack elements.
   ssize:INT;                   -- Number of elements in stack = insertion loc.
   constant initial_size:INT:=5; -- Start size of the stack array.

   create:SELF_TYPE is          -- 'SELF_TYPE' used so it works when inherited.
      -- A new stack.
      res := new;               -- 'res' is returned at routine completion.
      res.arr := ARRAY{T}::new(initial_size);
   end;

   push(e:T) is
      -- Insert the element 'e'.
      if ssize >= arr.asize then  -- Resize if stack area is full.
         arr := arr.extend(2 * arr.asize) -- This copies over the old elements.
      end;
      arr[ssize] := e;          -- Put new element at the top of the stack.
      ssize := ssize + 1;       -- Increase the stack size.
   end;

   empty:BOOL is
      -- True if stack is empty.
      res := (ssize=0);
   end;

   pop:T is
      -- Pops off the first element or 'void' if empty.
      if empty then return end;
      ssize := ssize - 1
      res := arr[ssize];
      arr[ssize] := void;
   end;

end; -- class STACK
```

```
-------------------------------------------------------------------

class TASK is
   -- Simple task, parent of more specialized tasks.
   name:STR;                        -- The name of the task.

   create(nm:STR):TASK is
      res := new;
      res.name := nm;
   end;

   print is
      -- Print the task name on stdout.
      OUT::s("Task: ").s(name).nl;
   end;

end; -- class TASK

-------------------------------------------------------------------

class CLIENT_TASK is
   -- Tasks that will be billed to a client.

   TASK;                          -- Inherits the properties of simple tasks.
   client:INT;                    -- The client number.
   allowed_time:REAL;              -- The time allowed in hours.

   create(nm:STR; cl:INT; tm:REAL):CLIENT_TASK is
      -- A new 'CLIENT_TASK'.
      res := new;
      res.name := nm;
      res.client := cl;
      res.allowed_time := tm;
   end;

   print is
      -- Print the task parameters on 'stdout'.
      OUT::s("Task: ").s(name).s(", Client: ").i(client);
      OUT::s(", Allowed time: ").r(allowed_time).s(" hours").nl
   end;

end; -- class CLIENT_TASK

-------------------------------------------------------------------

class TASK_MANAGER is
   -- A stack of tasks.

   STACK{$TASK};                  -- Inherits create, push, pop, empty.
      -- The $ indicates that the actual type may be a subtype of TASK.

   pop_and_print is
      -- Print the next item on the stack.
      pop.print;                  -- Will call the proper print routine
                                  -- by dynamic dispatch on the type.
   end;

end; -- class TASK_MANAGER

-------------------------------------------------------------------

class TASK_TRIAL is
```

```
   -- Try out a stack of tasks.

   main is
      -- Do a trial run, creating, pushing and printing some tasks.
      t:TASK; ct:CLIENT_TASK;
      tm:TASK_MANAGER:=TASK_MANAGER::create;
      tm.push(t.create("File report 1"));
      tm.push(ct.create("Edit Malloy document", 17, 2.5));
      tm.pop_and_print;
      tm.push(t.create("Prepare transparencies"));
      tm.pop_and_print;
      tm.pop_and_print;
   end;

end; -- class TASK_TRIAL

-------------------------------------------------------------------------
```

The example is contained in the file "example.sa". While this example defines five classes in a single source file, they would more typically be separated into two or more separate files. Section 10 describes the compilation process which is initiated by the shell command "cs task_trial" . The compiler produces a directory "task_trial.cs" which contains the generated C files and the binary file "task_trial" which is the executable program. The routine named "main" in the class "TASK_TRIAL" will be executed when the program is run. On execution, the program generates the output:

```
Task: Edit Malloy document Client: 17 Allowed time: 2.5 hours
Task: Prepare transparencies
Task: File report 1
```

2.2. *Terminology*

The example shows the structures common to all Sather programs. All code is partitioned into units called *classes*. The entities defined in a class are called its *features*. There are four kinds of feature: *routines, object attributes, shared attributes,* and *constant attributes*. The collection of object attributes in a class defines the structure of the *objects* which belong to that class. Internally, a typical object consists of a chunk of memory and an inital tag which identifies its class. A program may create many objects which belong to the same class. In contrast, only one instance of any shared or constant attribute is ever created. These are statically allocated variables that are directly accessible by all objects in a class. The values of constant attributes are not modifiable.

The features of *non-parameterized classes* are completely specified by the class definition. The only *parameterized class* in the example is "STACK{T}". Such classes leave certain *type parameters* to be specified when the class is used. In the example, "TASK_MANAGER" inherits "STACK{$TASK}". This instantiates the type parameter "T" in "STACK{T}" to have the value "$TASK". Notice that the author of "STACK{T}" need not have known about the eventual instantiation of "T" by "$TASK". The compiler can generate efficient code appropriate for use with a newly written class from an unchanged version of previously written code.

In cases where a variable must be able to hold objects of more than one class, feature access must be deferred to run-time. The compiler generates code which uses an object's tag to dispatch dynamically to the proper feature. In Sather this *dynamic dispatch* is constrained by the type system via the notion of *inheritance*. The inheritance relation gives rise to the notion of the *descendent classes* of a class. The type specifier for dynamically typed variables consists of a dollar sign followed by a static type specifier. The dollar sign indicates that references to these variables are slightly more expensive than other references. In the example "$TASK" refers to the descendents of the class "TASK". The rule is that the dynamic run-time class associated with an object which is held in a variable must be a descendent of the class which follows the dollar sign in its type specification. As detailed in section 4, only features which are defined in the parent class and have the same form in all descendents may be applied to such an object. Their definition will be determined at runtime by the object's actual type.

3. Classes

All code in Sather is organized into *classes*. Class names must consist of only upper case letters and underscores. Each class definition must be contained in a single source code file (with the exception of the C class), but multiple classes may be defined in the same file by separating their definitions by semicolons. Class definitions have the form:

```
class <class descriptor> is
    <feature list>
end
```

All class names are globally visible, but feature names belong to a separate namespace and do not conflict with class names. The body of a class definition consists of a semicolon separated list of feature specifications.

3.1. Parameterized and Non-parameterized Classes

Classes are either *parameterized* or *non-parameterized*. The <class descriptor> in non-parameterized classes is just the name of the class (eg. "FOO"). In parameterized classes, the <class descriptor> consists of the name of the class followed by a list of comma separated type variables in curly brackets (eg. "PCLASS{A,B,C}"). When a parameterized class is used, each of its parameters must be instantiated with classes (eg. "PCLASS{INT, REAL, BOOL}"). Parameter names are local to the parameterized class in which they are defined, but are in the same name space as class names and shadow them.

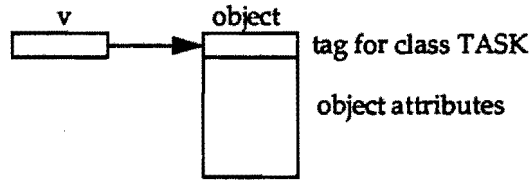
3.2. Inheritance

A class "A" inherits the features of another class "B" when the name "B" appears in A's feature list. We say that B is a parent class and that A is a child. The semantics is exactly as if the features of B were textually copied into A at the point the name appears. Any refer-

ences that appear in the inherited feature definitions will be resolved in the new namespace. Later features in a class with the same name as earlier ones override the earlier definitions. This means that the order in which classes are inherited can have an effect on the meaning. A class may inherit from many other classes. This is called *multiple inheritance.* The inheritance relations relating classes define a directed graph whose vertices are the classes. This graph may not contain cycles. For example, if A inherits from B, then B must not inherit from A. The inheritance graph is therefore a directed acyclic graph (DAG).

3.3. Ordinary Object Layout

In the most common situation, the memory for a variable "v" holding an object of type "TASK" will look like the following diagram. The variable v actually holds a pointer to the memory allocated for the object. The object itself consists of an integer tag which specifies the class "TASK" and enough memory to hold the various object attributes specified in that class. There is no distinction between the objects of parameterized and non-parameterized classes.



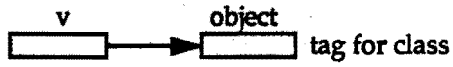
3.4. Void

All variables are automatically initialized. Variables which hold pointers are initialized to the special value "void". This value means that no object is currently held. Variables may be explicitly assigned the void value (eg. "a:=void") and equality with void may be tested (eg. "a=void"). There are certain restrictions on the operations which may be performed on variables whose value is void which are described later.

3.5. Stateless Classes

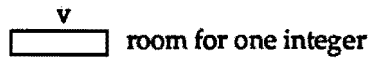
Classes serve both the functions of declaring the structure of objects and encapsulating code into well-defined modules. It is often convenient to define classes which don't have any object attributes. They may be used to declare a collection of constants, shareds or routines which are useful in more than one place. We will call such clases *stateless classes.* In many cases they are defined only to be inherited by other classes. When a variable is declared to be an explicit stateless class, it is often convenient to keep its value as "void" because no space will be declared to hold anything anyway. In cases in which we would like to choose routines or shared or constant attributes dynamically at runtime, we need to ac-

tually create objects corresponding to stateless classes. Since they have no object attributes, internally such objects consist of only the tag which is used for dispatching.



3.6. Basic Classes

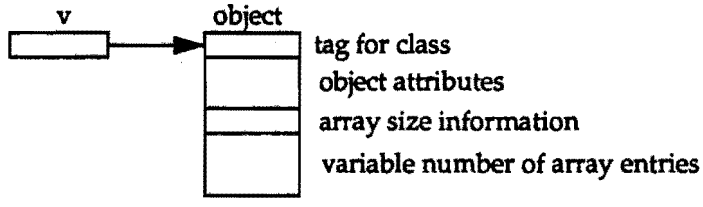
The basic classes form the construction material for all other classes. They include the built in classes "BOOL", "CHAR", "INT", "REAL", and "DOUBLE" and any class which inherits from one of these. Entities which are declared to hold basic types reserve space for the data they hold rather than a pointer to another object. For example if "v" is declared to hold an integer, then the memory allocated for it looks like the following diagram. .



To prevent ambiguity during dispatching, basic classes may only inherit from other basic classes derived from the same built in class. If "A" inherits from "INT" and "B", then "B" must also inherit from "INT". Basic classes may not inherit from stateless classes, array classes (see below) or classes with object attributes. Dispatching on basic classes is not permitted (eg. "$INT" is not a legal declaration). Variables of basic type are initialized as follows: BOOL's to False, CHAR's to \0, INT's to 0, REAL's and DOUBLE's to 0.0.

3.7. Array Classes

A class may inherit from at most one of the builtin classes "ARRAY{<type-spec>}", "ARRAY2{<type-spec>}", "ARRAY3{<type-spec>}" or "ARRAY4{<type-spec>}". Such classes are called *array classes* and their objects include space for 1, 2, 3, or 4 dimensional arrays, respectively. The space allocated to such objects is determined at the time of creation and includes a variable sized array portion after the object attributes. The memory layout is shown in the diagram.



Array classes are usually fairly low level classes which are defined in the system libraries. Most user classes will not directly inherit from array classes but rather will have object attributes which point to array class objects (eg. "a:ARRAY{INT}").

4. Type Declaration

There are six kinds of variables in Sather: the shared, constant, and object attributes of classes, and the arguments, local variables, and return values of routines. The type of each of these variables is declared when it is introduced. Most variable declarations consist of the variable name followed by a colon and a type specification.

4.1. The Four Kinds of Type Specification

The four kinds of type specification are:

1. *Non-parameterized:* The name of a non-parameterized class: eg. "TASK".
2. *Parameterized:* The name of a parameterized class with its type variables specified: eg. PCLASS{INT, REAL, TASK}. The type variables may be instantiated by any of the specifiers given here, eg. A{B{INT}} or A{$TASK}.
3. *Type Parameter:* A type variable in parameterized classes. In the class "STACK{T}", "T" is a valid type specification.
4. *Dispatched:* A dollar sign, followed by a type specification. This is a type which will be dynamically dispatched at runtime, eg. "$TASK". This specification indicates that the runtime class will be a subclass of the given class. Only features defined in the given class may be applied. Basic types may not be dispatched.

Some example variable declarations include: "a:INT", "a:TASK", "a:$TASK", "a:PCLASS{INT,REAL,FOO}", and in class "STACK{T}" the specification "a:T".

4.2. SELF_TYPE, UNDEFINE, $OB, F_OB

There are four additional declarations which are used in special circumstances. The declaration "SELF_TYPE" may be used to refer to the type of the class in which the declaration appears. This is useful in classes which will be inherited by other classes. The declaration refers to the final class in which it occurs rather than that of the parent. A typical example might be a class "LINK" which is used in implementing linked lists. The basic class must define a pointer to the next entry in the list. A descendent of this class will typically add extra information to be stored with these links. We want the links in the descendent class to point to other objects whose type is the descendent class, not the class "LINK". By declaring the pointer to be of type "SELF_TYPE" rather than "LINK" we obtain the proper behavior.

In class feature lists the specification "UNDEFINE" may be used to eliminate features. This is especially useful when a class inherits from another class but should not retain all of its features. Elimination of a parent's object or shared attributes can save space when they are not needed. If "a:INT" is defined in a parent class, then a descendent may want to have the line "a:UNDEFINE" in its feature list. As with other feature definitions, this will be overridden by any later definitions in the list with the same name.

The declaration $OB may be used to declare a variable which may hold an arbitrary object of non-basic type. The feature "copy" may be applied to such a variable to get a copy of the object it points to. The feature "type" may be applied to retrieve the integer type label. It is also permitted to apply certain other restricted features to "$OB" variables but this is most common in system code. If the call "o.foo" appears when o is declared "o:$OB", then it is required that "foo" be a routine with no arguments and no return value in any class in which it is defined. The runtime type of "o" will be checked and if a function with this name is defined for it, it will be exectuted, otherwise the call is ignored. This is used in the system libraries, for example, to cause hash tables which hash on object pointer values to be rehashed when deep copied or restored from a disk file. "$OB" variables may also be tested for equality with other variables or with "void". A common use for such declarations is in general container classes such as lists. When an element is extracted it is assigned to a variable of the appropriate type (perhaps by switching on the type) and more specific operations are then applied to the new variable.

The type "F_OB" is used to refer to "foreign pointers". These might be used, for example, to hold references to C structures. Such pointers are never followed by Sather and are treated essentially as integers which disallow arithmetic operations.

4.3. Type Conformance

The basic type rule in Sather is that a variable may hold an object only if the class of the object *conforms* to the variable's type specifier. This section defines the conformance relation in terms of the inheritance graph. That t1 conforms to t2 is meant to capture the notion that code generated under the assumption that an object's type is t2 will still be type correct on objects of type t1. Some properties we would like conformance to satisfy are that a type conforms to itself, conformance is transitive (ie. if t1 conforms to t2 and t2 conforms to t3 then t1 conforms to t3), if a type specifier t2 appears in the inheritance list for t1, then both t1 and $t1 conform to $t2 and if t1 and t2 are different instantiations of the same parameterized class (eg. FOO{A} and FOO{$B}) and each parameter instantiation in t1 conforms to the corresponding one in t2, then t1 conforms to t2. As an example to see why this last form of conformance is important, consider tree nodes which hold geometric shapes. We would like to be able to hold nodes of type TREE{SQUARE} in variables of type TREE{$POLYGON}. Finally, all descendents of a basic type conform to one another. There is a nice theorem (see "The Sather Implementation") that gives a simple set of rules which lead to the simplest conformance partial order which satisfies these requirements.

Type specifications which begin with a "$", are called "dispatched" types. In Sather, a type specification t1 conforms to a type specification t2 when:

1. If t2 is a non-dispatched non-parameterized type (eg. FOO), then t1 is the same type (eg. FOO).
2. If t2 is a dispatched non-parameterized type (eg. $FOO), then t1 is a subtype of the non-parameterized type (eg. BAR where BAR is a descendent of FOO).

3. If t2 is a non-dispatched parameterized type (eg. FOO{A,$B,C}), then t1 is also parameterized, has the same number of type variables, has the same base class as t2, and has type parameter instantiations which conform to those of t2's (eg. FOO{A, D, C} where D is a subtype of B).
4. If t2 is a dispatched parameterized type (eg. $FOO{A,$B,C}), then t1 must either be or inherit from a class which would conform with the underlying type (eg. BAR if it is a descendent of FOO{A,D,C} where D conforms to $B).
5. If t2 is a basic type, then t1 is a descendent of the same ancestor. Eg. all descendents of INT conform to one another.

4.4. Dynamic Dispatch

If features are inherited for the purpose of dynamic dispatch, then several aspects of the form of entities in the parent class must be preserved in the descendents. If this were not the case, the compiler could not infer the information it needs about dispatched calls. For example, if the variable "a" is declared as "a:$PARENT" and we wish to make the dispatched call "a.f", then the definition of "f" in the descendents of "PARENT" must agree in several respects with its definition in "PARENT". If "f" is respectively an object attribute, a shared attribute, a constant attribute, or a routine in "PARENT", then it must also be in all of the descendents of "PARENT". If "f" is an attribute, then its type in the descendents must conform to its type in "PARENT". If "f" is a routine, it must have the same number of arguments and return value and their types must conform with those in the parent class. If dispatched array access is used, then the descendent classes must have the same number of array dimensions as the parent.

There is no constraint on any feature which is never used in a dispatched fashion. A common example is the function "create". This is often defined in a class to make new instances, initializing their attribute slots with its arguments. Different classes will typically need different numbers of arguments, yet it is convenient to use the same name "create" in each class. This causes no problems if one never applies "create" to a variable with a dispatched type.

Routine calls on objects of dynamic type must be routed through a dispatch table and so are not as efficient as calls on objects whose type is statically determined. A local caching scheme is used so that on average only about one extra comparison and two pointer indirections are needed for dynamic dispatch.

Sather does not attempt to catch all possible type errors during compilation. Typical rules for guaranteeing absolute type safety are either too complex or too confining. An opportunity for type checking arises in assignment statements and in specifying routine arguments. The basic type rule is that the runtime class of an object must conform to the type specifier of a variable which holds it. The difficulty in checking this at compile time arises when the type specifier for the right hand side of an assignment is itself dispatched. In this case the runtime class of the object may legally be any of a number of possibilities. Some of these possibilities may conform to the left hand side while others do not. To guarantee

type-safety, we could disallow any assignments in which the type specifier for right hand side admitted objects which would not be allowed on the left hand side. Unfortunately, such a constraint is very confining and tends to lead to convoluted code. (eg: A library class defines lists of polygons, you have created a list of squares, you pop off a square and would like to assign it to a variable of type square. Because the list was defined to hold polygons, the compiler would not allow the assignment). The Sather approach is to generate a compiler error only if there is no possible value for the right hand side which conforms to the left hand side. For non-dispatched type specifiers this convention yields strict type checking. Runtime type checking of assignments of dynamically typed variables to statically typed ones may be enabled by compiling with the "-chk" flag.

5. *Attributes*

The feature lists of classes consist of five kinds of specification which are separated by semi-colons. We have already discussed the specification of class inheritance. This section discusses the three kinds of class attribute: object, shared, and constant and the next section describes routines. Attribute, shared attribute, and constant names are local to and visible throughout a class. External access may be prevented by the *private* declaration.

5.1. *Object Attributes*

Object attributes define the slots in objects. The declaration appears in a class's feature list and consists of the attribute's name, a colon, and one of the type specifications discussed in the last section, eg:"a:TASK". Several variables may be declared to be of the same type by separating them by commas, eg: "a,b,c:TASK". If an object attribute is specified to be a basic class, such as BOOL, CHAR, INT, REAL, or DOUBLE, then sufficient aligned space will be reserved in the runtime objects to hold the appropriate quantities. Any other declaration reserves space for a pointer.

5.2. *Shared Attributes*

Shared attributes are shared by all objects in a given class. Their declarations are preceded by the "shared" keyword:

```
shared a:INT;
```

and they may be initialized:

```
shared a:INT := 3;
```

The initialization code is run before any other code. If the initialization of a variable requires features from another class, that class must be properly initialized before the feature is called. It is therefore an error to have circular dependences in the initialization calls (i.e. if an initializer for class A refers to an entity in class B, then the initializers in B may not

refer to entities in A). A shared attribute in another class may be accessed using the form "CLASS::a" and assigned to using "CLASS::a := 5;". We refer to this as "class access" to a shared variable, as opposed to "object access" to it.

5.3. Constant Attributes

Named constant attributes are defined by a similar syntax:

```
constant a:INT := 3;
```

All constants must be initialized. It is an error to try to modify the value of a constant attribute. The values of constant INT's and CHAR's whose initializer doesn't involve function calls are computed by the compiler and are especially valuable in conjunction with switch statements. As with shared variables, circular dependencies between the initializers are not allowed.

5.4. Private

Any feature definition may be preceded with the keyword "private". Only routines defined within the class itself may access such features. Private features will also be private in any classes which inherit them. This declaration is useful for guaranteeing that certain relationships within a class will be preserved. A common example is to define an attribute as private and to provide a routine which returns its value. This keeps other classes from changing the value, while still allowing them to read it.

5.5. The "type" Attribute

All classes have a predefined constant attribute named "type". Its value belongs to CLASS_TYPE, a descendent of INT, and equals the tag on objects used to identify a class. It is useful in constructs such as "if a.type = FOO::type then ...".

5.6. Creating Objects

A special function, "new", is defined in each class to actually allocate space for a new object. It is an error to override the definition of this routine. In array classes it has an integer argument for each dimension which specifies the size (eg. "ARRAY{INT}::new(15)" or ARRAY2{INT}::new(7,4)"). All attributes are initialized with the default values described above. It is common to define a function "create" which calls "new" and further initializes the attributes. The typical usage is "a:=FOO::new". Sather is garbarge collected and there is no need to explicitly deallocate objects.

5.7. Copying Objects

Another predefined function in each class is "copy". It takes no arguments and produces an exact duplicate of an object. The library class "SYS" also defines "deep_copy" which copies the entire structure reachable from an object by following object pointers. It deals properly with general graph structures. Array classes support the operation "extend". It takes a number of arguments which is equal to the number of array dimensions of the object it is applied to. It creates a new object whose array sizes are specified by the arguments to extend. The old entries are copied into the appropriate locations and any extra entries are initialized with the standard values (eg. "a.extend(15)" will return an extended version of the array "a" which has 15 elements).

6. Routines

A routine specification has the form:

```
routine_name(a:INT; b,c:REAL):BOOL is
    <statement list>
end
```

As with attributes, routine specifications may also be preceded by the keyword "private" to allow only internal access.

If a routine has arguments, they are specified by following the routine name by a parentheses bounded list of semicolon separated argument specifications. A return value is specified by a colon followed by a type specifier for the return type. Each of the possible combinations is legal:

1. "foo is ..." (no args, no result)
2. "foo(a:INT) is ..." (args, no result)
3. "foo:REAL is ..." (no args, result)
4. "foo(a:INT):REAL is ..." (args and result)

The body of a routine is a semi-colon separated list of statements. The possible kinds of statements are local declarations, assignments, conditionals, loops, switches, function calls, break, and return statements. These are described in the following section. Routine argument and local variable names are only visible within the routine and may shadow the names of class attributes.

6.1. Res

A variable named "res" is automatically declared in routines with a return value. When the routine exits, either at its end or due to a "return" statement, the current value of res is returned. Like all other variables, "res" is properly initialized.

6.2. Self

Sather routines have an implicit first argument which holds the object on which the routine was called. This object may be refered to in the body of a routine using the name "self". It is an error to attempt to assign to self.

7. Statements

7.1. Local Variable Declarations

Local variables may be declared anywhere a statement is allowed. The form of a initialized declaration is "a:FOO := 5". If no initialization is provided (eg. "a:FOO") variables are initialized to the default initial values listed above. Multiple variables may be declared together as in "a,b,c:INT" (they all receive the same initialization if one is provided). The scope of the declaration is local to the statement list in which it appears and any explicit assignment occurs each time the statement is encountered during execution.

7.2. Assignments

The symbol ":=" is used to denote assignment. The left hand side of an expression must be an attribute or shared attribute name (a:=5), a class accessed shared attribute (FOO::a:=7), an attribute or shared attribute relative to an object (b.a:=7), or an array references (a[5]:=6, FOO::a[4]:=7, b.a[3]:=8). The right hand side is an arbitrary expression. If there is no legal value for the right hand side which conforms to the type specifier for the left hand side, the compiler will generate an error.

7.3. Conditionals

Conditionals have the form:

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

where the "elsif" and "else" parts are optional and there may be an arbitrary number of "elsif" parts.

7.4. Loops

Loops have the form:

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

or just

```
loop
    <statement list>
end
```

The latter form is only useful if it includes a "break" or "return" statement.

7.5. Break and Return Statements

The statement consisting of only the keyword "break" is used to leave the innermost loop lexically containing it. It may only be used inside loops. The statement consisting of only the keyword "return" causes immediate return from the routine. The return value (if there is one) will be the value of "res".

7.6. Switch Statements

The switch statement has the form:

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

There may be an arbitrary number of "when" clauses and the "else" clause is optional. If the type of the switch expression is a base type, all expressions in the "when" lists must be of the same type. If the value of the switch expression appears in any of the "when" expression lists, then the statement list following it is executed, otherwise the statements following the optional "else" clause are executed. If a value appears in more than one "when" expression list, only one of the statement lists will be executed but which one is not specified. This construct is most efficient when the expression type is INT or CHAR and the "when" expression lists are all constants. In this case the compiler will usually produce a jump table. In other cases the generated code will be equivalent to a series of "elsif" clauses.

7.7. Routine Calls

Routine calls are usually made relative to an object and are denoted by a period. If "a" is of class "FOO", then "a.fun(arg1,arg2)" will call the routine named "fun" in class "FOO" with the specified arguments. As with shared and constant attributes, routines may also be specified directly by "class access". The class name is followed by two colons and the routine name as in "FOO::fun(arg1,arg2)". In this case any references to "self" will have the value "void".

It is legal to access the routines, constants, and shared attributes of variables with value "void" as long as their type is specified at compile time. This is commonly used with the "new" routine. It is also syntactically legal to access routines from the result of expressions which are either parenthesized (eg. (a+b).mod(c)), array references (eg. a[7,8].fun), calls (eg. a(9).b(7)), or identifiers (eg. foo.fun).

Dotted routine access may also be applied directly to constants, though the use of parentheses will sometimes make the call clearer (eg. "a string".len, 10.mod(3), 2.3.to_i). One potential source of errors is the fact that unary minus has a lower precedence than dotted access. Thus "-10.mod(3)" means "-(10.mod(3))". One must use explicit parentheses to get the interpretation "(-10).mod(3)".

7.8. Assert and Debug Statements

There are two mechanisms for including conditionally compiled code:

```
assert(table_overflow) a<200 and a>0 end;
```

and

```
debug(print_slot_name)
    a.print;
    b.print
end;
```

The first consists of the keyword "assert", a key in parentheses, a boolean expression, and the keyword "end". If the keyword is specified in the compiler control file, then the test is performed at runtime and a message is printed if it fails. The second mechanism consists of the keyword "debug", a compiler key, a statement list and the keyword "end". If the key is specified in the compiler control file, then the statement list will be compiled.

A list of keywords to be enabled is specified in the dot-sather file after the indicator "(debug_keys)". To enable the assert and debug expressions for a keyword in only a specific file, this list may include entries of the form: "CLASS::key_to_be_enabled". This enables an author to use the same name for similar checks in different classes and to only enable those in a class which is currently being debugged.

There are also builtin checks that may be enabled with the "-chk" compilation option. This enables runtime type checking of assignment and routine arguments, runtime checking of

array indices, and checks for dispatching from "void" variables. A standard convention is to use the debug key "pre" for simple tests that make sure the arguments to a routine satisfy the preconditions of that routine. When first using a class or when problems are occuring, compiling with the "pre" key will ensure that these preconditions are satisfied.

8. Expressions

8.1. Constants

The simplest expressions are constants. The boolean constants are "true" and "false". "void" may be thought of as a constant object. Character, integer, real, and string constants obey the same conventions as in C. Some examples include: 'a', '\000', 15, -15, 2.3, -2.4, 2.4e-19, and "this is a string". Hexadecimal integers are indicated by a leading zero followed by "x" or "X" and then digits or leters in the range "a" through "f" (eg. 0x37f or 0X9A7B). The one exception is that floating point constants using the "e" notation must have explicit digits after the decimal point. A floating point constant like "1.e3" would be confused with the application of the "e3" feature to the integer constant "1". "1.0e3" must be used instead.

8.2. Identifiers

Inside routines, the names of local variables, arguments, the object attributes of self and the special variables "res" and "self" are all expressions. In addition the routines "extend" (for array classes) and "copy" may be called. In expressions appearing in both initializers and routines, the names of constant and shared attributes, routines, and the special names "type" and "new" are legitimate.

8.3. Dotted Expressions

Access to the features of an object is obtained with the dotted notation: "a.foo". Dotted access may be applied to identifiers, constants, dotted expressions, array references, and parenthesized expressions.

8.4. Class Access

The features of a class may be accessed directly through class access. This is signified with two colons: "CLASS::a" and "CLASS::fun(x,y)".

8.5. Boolean Expressions

The boolean-valued numerical comparison operators are: "=", "/=", ">", "<", ">=", and "<=". The predefined boolean operations are "and", "or", "not" and parentheses may be used to specify precedence. Unlike the C versions of "and" and "or", Sather always evalu-

ates all arguments of operator expressions. "=" is also used for testing the equality of pointers. An important case is "a=void". Parentheses may be used to modify the usual precedence relations. The class BOOL defines a number of other operations using ordinary syntax (eg. "xor", "nand", etc.).

8.6. Numerical Expressions

The built in binary operators "*","/","+","-" may be applied to INT's, REAL's and DOUBLE's. The usual precedence relations apply, as described in the section on syntax, and they may be modified with parentheses. As in C, integer divide gives the integer part of the true quotient. The basic classes CHAR, INT, REAL, or DOUBLE define a number of other operations using ordinary syntax (eg. a.mod(b), f.to_i, a.bit_xor(b)). Automatic conversion between REAL and DOUBLE is provided. Variables of either type may be assigned to the other and used as argments which are declared to be of either type.

8.7. Array Access

Array classes automatically have the attributes "asize", "asize1", "asize2", "asize3", "asize4" defined, depending on their dimension. If "a" is an object with a two-dimensional array class then access to its elements is written "a[2,3]", assignment by "a[2,3]:= 5" and similarly for the other dimensions. Regardless of the dimension of the array, references of the form "a[4]" access it as if it were one dimensional and "a.asize" gives the one-dimensional size of the array. Access to array elements in "self" takes the form "[4]".

9. Interfacing with C Code

9.1. Accessing C from Sather

Sather code may be linked with standard C code so that Sather functions may call C functions and vice versa. Access to C functions within Sather is provided by a special class named "C". This class may contain only shared attribute specifications and routine specifications. Unlike other classes, the "C" class declarations may be distributed over serveral files as long as the definitions don't conflict. The name of each shared attribute and routine corresponds to a C external variable or function. The type specification given is used by Sather to properly link the calls to these functions. Within a Sather class, access to these functions is written in the standard form: "C::foo". User routines with a variable number of arguments are not supported in Sather and such C functions (eg. printf) must be shielded by C code with a fixed number of arguments.

The return value will be assumed to be of the type specified in the "C" class. Sather BOOL's are passed as chars with value zero for false, non-zero to true. INT, REAL, and DOUBLE are passed as C ints, floats, and doubles. Objects are passed as pointers to the object. The

header file "all.h" provides the appropriate "typedefs" so that "bool", "real", and "ptr" may be used in external C declarations. It is often convenient to declare the C functions needed for a Sather class in the same file as that class, so that they are guaranteed to be present when the class is compiled. Certain low level C operations such as type casts are actually implemented as macros which expand into the generated C files. In every case a function version is provided as well, however, since system tools like the interpreter will sometimes need them.

9.2. Accessing Sather from C

A separate mechanism is provided to access Sather features from C. In the compiler control file, sections headed with the keyword "(c_name)" allow one to give a C name to a Sather routine, shared attribute, or constant attribute. These specifications have the form "(c_name) sat_fun1 FOO::fun1". In the generated code, the given name will be used to refer to the specified attribute. External C routines may access the attribute under the given name. The external routine is responsible for using an "extern" delcaration which is consistent with the Sather declaration.

10. Compilation

The Sather compiler is invoked by the command "cs" (for "compile sather") followed by a class name. To find the files it needs, the compiler first looks in the current directory and then in the user's home directory for a file named ".sather". This file contains the information the system needs for compilation. It consists of a series of specifications among which white-space and Sather style comments may be interspersed. Keywords in this file are surrounded by parentheses and are followed by space separated lists. The keywords are:

- (source_files): Followed by a list of files in which Sather source code appears.
- (c_files): Followed by a list of C object files which will be linked with the Sather generated code.
- (c_makefiles): A list of makefile which are exectued before the Sather compilation begins. These are typically used to ensure that any external C object files are up to date.
- (debug_keys): Followed by a list of debug and assert keys that will be enabled during the compilation of the C code.
- (cc_flags): C compiler flags enabled during compilation (eg. -O, -g).
- (c_name): Followed by a name and a feature specification of the form "CLASS::feature". The name will be used in the generated files to refer to the specified feature.
- (include): Followed by a list of files in the .sather format. The effect is as if their contents were textually included.
- (sather_home): Followed by the directory which contains system files, by default it uses "/usr/local/src/sather".
- (c_compiler): By default, uses "cc".

All sections are optional and the order of their appearance is arbitrary. For most keys, multiple sections with the same key behave as if their lists had been appended. There must be a separate (c_name) key for each name defined. The "(include)" facility makes it convenient to share system wide information.

A new directory is created in the current directory with a name of the form class_name.cs. This directory will contain the generated C files, a makefile, and object files. There is one C file per class with a name consisting of the first 3 letters of the class followed by its index (eg. "CLA123.c"). There will be several other files containing the runtime system, garbage collector, and header files. An executable with the name of the class will also be produced in the directory in which compilation was initiated.

10.1. Main

There must be a routine named "main" in the class the compiler is called on. After initialization, execution will begin with a call to this function. In this initial call to "main" any references to "self" will have the value "void". If "INIT" is the class, then the behavior is exactly as if "INIT::main" were called. If an object of the class on which the compiler is called is desired, it must be explicitly created within "main".

The routine "main" in the class the compiler is called on may either have no arguments or a single argument of type "ARRAY{STR}". If no argument is specified then any command line arguments the user might provide will be ignored. If an argument is provided then it will be filled in with an array of the strings making up the command line. The "asize" of the array will be one larger than the number of command line arguments (similar to "argc" in C). The zeroeth entry of the array will be the string used to invoke execution of the program. The successive entries will be the command line arguments.

The simplest Sather program is then:

```
class SIMPLEST is
   main is
      OUT::s("Hello world.\n")
   end
end
```

To get numerical parameters from the command line, it is convenient to use the functions "to_i", etc. which are defined in the string class "STR". For example, a program which expects two integers and a real might look like:

```
class TST is
   main(args:ARRAY{STR}) is
      i1:INT:=args[1].to_i;
      i2:INT:=args[2].to_i;
      r:REAL:=args[3].to_r;
      ...
   end; -- main
end; -- class TST
```

If "main" is declared to return an INT, then the value of "res" at program termination will become the return status of the program (eg. the $status variable in the UNIX csh will hold the value).

11. The Lexical Structure of Source Files

The names of Sather source files all have the extension ".sa" (as in "example.sa"). To facilitate linking with C code, Sather is case sensitive. Class names must consist entirely of upper case letters and underscores. For readability, it is recommended that other names not be of this form. The system libraries use the style of lower case names separated by underscores (eg. "this_is_a_name"). Anything on a line following "--" is a comment. Sequences of spaces, tabs, newlines and ASCII VT, BS, CR, FF are considered to be whitespace. Comments and whitespace may appear anywhere between syntactic constructs and are ignored by the compiler.

All entity names may be of arbitrary length. They must consist of only letters, numbers, and underscores and must begin with a letter. To avoid conflicts with internally generated names, it is recommended that no user names end with an underscore. The keywords: and, assert, break, class, constant, debug, else, elsif, end, if, is, loop, not, or, private, return, shared, switch, then, until, and when have syntactic significance and may not be used for any other purpose. All other tokens beginning with a letter are syntactically considered to be identifiers. The single character tokens: +, -, *, /, =, <, >, (,), [,], comma, ;, :, $, ., {, }, and the two-character tokens /=, <=, >=, :=, and :: also have syntactic significance. The characters: ^, !, @, #, %, &, ~, ', ?, \, and | are illegal outside of string and character constants.

Integer constants consist of a series of digits. Hexadecimal integers begin with "0x" or "0X" and consist of digits and characters in the ranges "a-f" or "A-F". Floating point constants have the form "12.345E-87". When the "e" notation is not used, floating point constants without digits after the decimal (eg. "1.") are also allowed. String constants begin with a double quote and continue until the next unescaped double quote. As in C, a backslash inside a string escapes the following character: \" inserts a double quote, \' inserts a single quote, \\ inserts a backslash, \ followed by a newline causes the newline to be ignored, \n inserts a newline, and \t inserts a tab. Character constants begin and end with a single quote and use the same escape conventions.

12. Syntax

There are a number of reserved words in the language. The keywords have a syntactic significance and consist of:

```
and, assert, break, class, constant, debug, else, elsif, end, if, is, loop, not, or,
private, return, shared, switch, then, until, when
```

Other reserved words are the names of automatically defined routines and variables:

```
asize, asize1, asize2, asize3, asize4, copy, extend, false, new, res, self, true,
type, void
```

Finaly there are a number of predefined type specifiers with special importance:

```
ARRAY, ARRAY2, ARRAY3, ARRAY4, BOOL, C, CHAR, DOUBLE, ERR, FILE, IN, INT, OB, OUT,
REAL, SELF_TYPE, STR, STR_SCAN, SYS, UNDEFINE, UNIX
```

The Sather compiler creates C names for all Sather constructs. To avoid conflict with these created names, no C names or Sather names should end with an underscore.

The precedence relations between the binary operators from weakest to strongest are (in YACC notation):

```
%left OR   %left  AND   %left '=' '/='    %nonassoc '<=' '<' '>' '>='   %left '+' '-'
%left '*' '/'   %right NOT UNARY_MINUS   %left '.'
```

The syntax of Sather in a notation close to YACC follows. The text of each source file should parse into a class_list.

```
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 ']'
```

13. *Differences between Sather and Eiffel*

Sather eliminates many Eiffel features which are not essential for practical code development. It also adds several features to increase efficiency, and modifies others to make them more standard or systematic.

Certain Eiffel reserved words were renamed in Sather: "BOOLEAN" became "BOOL", "check" became "assert", "CHARACTER" became "CHAR", "Current" became "self", "INTEGER" became "INT", "inspect" became "switch", "Result" became "res", "STRING" became "STR". Other reserved words were eliminated: "as", "BITS", "Clone" (use "copy"), "Create" (use "new"), "deferred", "define", "div" (use version in INT), "do", "ensure" (use "assert"), "expanded", "export" "external" (use the "C" class), "feature", "Forget" (use assignment of "void"), "from", "implies" (use version in BOOL), "inherit" (use classes in feature list), "infix", "invariant" (use "assert"), "is", "language", "like" (use "SELF_TYPE" for most common case), "local" (use local declaration statement), "mod" (use version in INT), "name", "Nochange", "obsolete", "old", "once" (use shared variables), "prefix", "redefine", "rename", "repeat", "require" (use "assert"), "rescue", "retry", "unique", "variant" (use "assert"), "Void" (use equality test with "void"), "xor" (use feature in BOOL), "^" (use "pow" in MATH).

The most fundamental change is the addition of the ability to explicitly specify the types of entities. This puts more burden on the programmer but can have a dramatic effect on efficiency. In Eiffel, Create and Clone are different from every other function in that they change the value of a variable they are applied to. In Sather the only way to change the value of a variable is to assign to it. Sather adds the "type" feature which is necessary to do the old-style switch based dispatch (which is still an important operation on occasion). The usual usage of Eiffel arrays requires a double indirection to get to the array, in Sather the objects themselves may be extended with an array part (this is a common technique in efficient C code, though it looks kludgy in C). Two dimensional arrays are accessed by pointer indirection rather than by multiplication (this is especially important on RISC machines which do integer multiplication in software).

Sather provides library classes to support exception handling rather than building it into the language. Sather's garbage collector doesn't have any overhead when it is not running, whereas the Eiffel collector extracts a cost on any pointer variable assignment. Eiffel's collector also must keep a stack of pointers to pointers on the stack. This keeps these variables from being put in registers, which can severely affect performance on RISC machines. Eiffel's ability to replace functions by variables in descendents has been eliminated. This allows most attribute access to be done without the overhead of function calls. More than

one class may be defined in a Sather file and the names are not subject to the restrictions of Unix file names. The Sather inheritance mechanism is less powerful than Eiffel's but it is easier to understand the semantics.

Eiffel tries to ensure that type errors are impossible if a program makes it through the compiler. Sather has no such goal, though it does try to catch all common errors. Sather adds shared variables to classes. Using these, Eiffel's once functions may be easily implemented and shared variables are often much more convenient. Direct class access to the features of another class are provided with the "::" syntax (as in C++). This facility is subject to abuse but there are many situations in which it can greatly simplify the structure of a system. Sather's interface to C is cleaner and more efficient than Eiffel's.

Sather does not provide expanded classes or overloading of infix operators because the increased complexity does not seem worth the gain. Sather allows local variables to be declared at the point of use, as in C++, rather than in a separate local declaration section. All of the different assertion clauses (ensure, require, check, invariant, variant) are replaced by "assert" in Sather. In Sather, the debug and assert clauses are named and all clauses with a given name may be turned on or off independently. Type checking on dispatched assignments is only checked at runtime if the corresponding compiler key is enabled. Sather's syntax is simpler than Eiffel's, particularly in the structure of class and routine definitions, though several new features were added. Sather makes all class features available from outside the class by default, using the "private" mechanism to mark features which shouldn't be exported. The marking is located at the feature specification avoiding the necessity of maintaining a separate list as in Eiffel. Eiffel takes the view that features should be private by default and only externally available when explicitly listed in an extern list.

14. Acknowledgements

Sather clearly owes its major debt to Eiffel and has been heavily influenced by the description in "Eiffel: The Language" by Bertrand Meyer. It has also adopted ideas from a variety of other languages including Smalltalk, CLU, Common Lisp, Modula 3, Pascal, SAIL, Objective C, C, and C++. The tools from the Free Software Foundation, the public domain BYACC by Corbett and the garbage collector by Hans-J. Boehm and Alan J. Demers were used in its implementation. Improvements were suggested by Subutai Ahmad, Joachim Beer, Jeff Bilmes, Peter Blicher, Henry Cejtin, Richard Durbin, Jerry Feldman, Carl Feynman, Chu-Cheow Lim, Franco Mazzanti, Heinz Schmidt, Carlo Sequin and Bob Weiner. The Sather compiler was written in Sather by Chu-Cheow Lim and extended by Jeff Bilmes. The Sather debugger was written by Jeff Bilmes and the Sather emacs programming environment was written by Heinz Schmidt.