The **SETL2** Programming Language: Update On Current Developments

W. Kirk Snyder
Courant Institute of Mathematical Sciences
New York University
New York, NY 10012
September 7, 1990

Abstract

SETL2 is evolving rapidly as our work on it continues. This document describes many changes made since the original **SETL2** report ([Sny89]) was written.

The most significant change is the incorporation of features of object-oriented programming languages, including abstract data types, encapsulation, multiple inheritance, and operator overloading. The other changes we regard as temporary solutions to pressing problems. We have added several new I/O procedures and a link to lower level programming languages, but both of these areas are in need of considerably more work and are likely to change in the future.

New York University
Department of Computer Science
Courant Institute of Mathematical Sciences

Contents

1 Introduction

As our work on **SETL2** progresses, improvements in the language are being implemented at a rapid pace. Since the original **SETL2** report [Sny89] was written we have added a number of new I/O procedures, a temporary call-out facility, and most significantly support for object-oriented programming. This document describes those new facilities.

The goal in adopting features from object-oriented programming languages is to provide **SETL2** with abstract data types and the ability to overload operators. An important feature of SETL which is missing in **SETL2** is user-defined operators. We feel that the concept is good, but what one really wants is the ability to redefine **SETL2**'s built-in operators on user-defined types, i.e. more operator overloading. The concepts of object-oriented programming seem well-suited as a model for these features.

The I/O system in **SETL2** is very much like that in lower level programming languages, which is a shame. There seems to be no compelling reason why that need be the case. Most of the high-level dictions used on strings and maps can be easily extended to apply to files as well. Although this is our eventual goal, we haven't accomplished it yet. For the time being we have extended the I/O system somewhat, but not changed its nature significantly. What we have done is provide the low-level primitives which, combined with the object features, can be used to implement a superior I/O system in **SETL2** itself. Our next task in this area is to experiment with various new I/O systems using classes.

An often-requested feature is the ability to call procedures written in lower level languages. For various reasons this is not at the top of our priority list, although we do recognize its importance. As a stop-gap measure, we have implemented a temporary call-out facility, which we intend to replace as time permits. If the use of the provided features is encapsulated in a single package, it should be possible to use functions written in a lower level language now, without requiring drastic changes when we implement a more ambitious interface.

This is not a comprehensive description of **SETL2**, and as this is being written no such document exists. Plans are being made to produce such a reference but until that is available we suggest you refer to [SDDS86] for a general description on programming with sets, [Sny89] for a description of the core features of **SETL2**, and this document for those features not described in that report.

2 Operators

We have not added any new operators to **SETL2**, but the function of some of the relational operators has been enhanced. We have extended <, <=, >, and >= to apply to sets and maps in the obvious way:

s < ss	Yields true if s $\subset$ ss, false otherwise.
s <= ss	Yields true if s $\subseteq$ ss, false otherwise.
s > ss	Yields true if s $\supset$ ss, false otherwise.
s >= ss	Yields true if s $\supseteq$ ss, false otherwise.

This addition makes the subset and incs operators redundant. They have not been removed from the current implementation yet, but we consider them obsolete and expect to remove them eventually. For this reason we did not provide the ability to overload them (see 4.7), although we do provide the ability to overload the less than operator.

3 Statements

Statements are as described in the **SETL2** report, with one minor addition. The loop statements described there included for, while, and until loops, but nothing to support a *do forever* loop. It is obviously easy to code this as while true loop, but that is a bit uglier than we like. We now allow a loop header without a for, while, or until clause, so a *do forever* loop can be written with the following syntax:

Clearly there should be an explicit exit statement somewhere in the *statement-list*, or the loop will never terminate.

4 Objects and Classes

The most significant extension to **SETL2** is the incorporation of features found in object-oriented programming languages. The features provided include encapsulation, multiple inheritance, user-defined abstract data types, and operator overloading on user-defined types.

The principal goal of this project is to provide a mechanism for the **SETL2** programmer to create his own types and to define the normal **SETL2** operations (+, *, etc.) on those types. The concepts of object-oriented programming languages seem to be the best model for this feature. The overloading of methods provided in other languages is easily extended to include many of **SETL2**'s built-in operations.

Although we have provided multiple inheritance we do not provide the many mechanisms used to control exactly what is inherited and how it is named. We have provided an ‘‘all or nothing’’ inheritance, in which a subclass inherits *all* of the names in its superclasses without modification. There are rules for handling duplicate names, but these control what is hidden and what is visible, they do not allow selective inheritance or the ability to rename an inherited method.

We have gone to considerable effort to be consistent with the existing features of **SETL2**. We have preserved the concepts of value-semantics and passing parameters by copying even when it seemed more efficient to abandon these ideas. This is somewhat unusual among object-oriented programming languages. Pointer semantics are much more common.

4.1 Basic Concepts

Before getting into details on the **SETL2** object system, we will present some of the general concepts and terminology of object oriented programming. Much more detail on these topics can be found in [Mey88] or [GR89].

An *object* is a set of values and a set of operations which can be performed on that object. It will normally be used to represent something not easily represented by one of **SETL2**'s built-in types, such as a display item, file, heap, etc. The operations defined on an object are called *methods*, and are conceptually similar to procedures.

The fundamental difference between a method and a procedure is that a method specifies an operation to be performed, but *not* the specific procedure which should be used in performing the operation. To illustrate the difference between these ideas, suppose we have two objects, one a stack and the other a queue. Each of these objects has a method called top, which returns the top element. We have a variable, x, which can be either a stack or a queue. Then the expression x.top() will call either the method defined on stacks or the method defined on queues, depending on the value of x at the time the expression is evaluated. A method is therefore very much like an overloaded procedure.

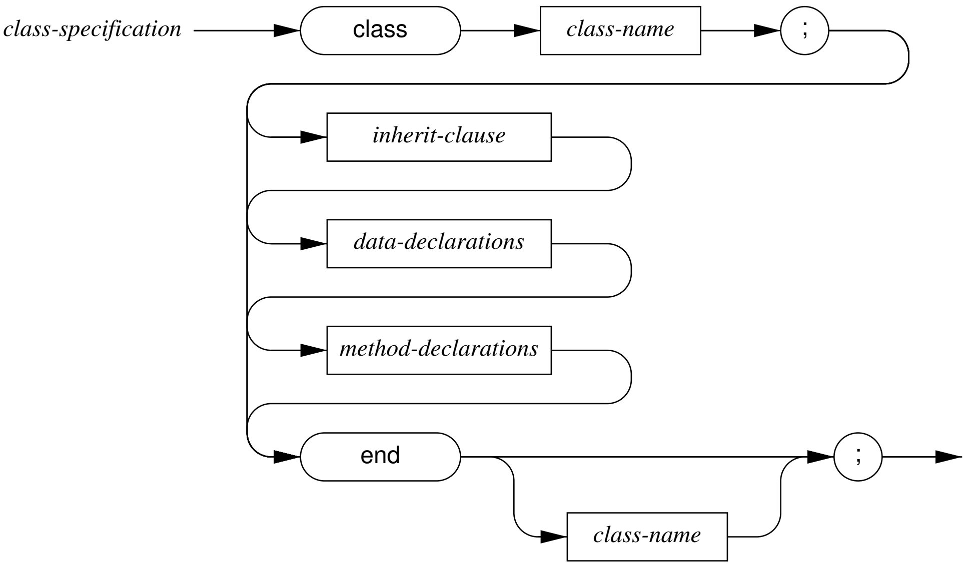
A method is invoked by passing it the containing object and any other operands, or arguments, required by the method. The process is much like a procedure call except the object determines the procedure called and becomes an implicit argument to the call. In the terminology of object-oriented programming this is called passing a *message* to the object.

A *class* describes the set of data elements and methods defined on a set of similar objects. The set of objects described by a class are called the *instances* of the class. Both data elements and methods defined on a class can be either public or private. We also allow public or private data elements common to all instances of a class, but this is somewhat unusual.

The data elements and methods of a class can be merged into another class by a process called *inheritance*. The source class in such an operation is called the *superclass* and the target is called the *subclass*. The subclass is able to use data elements or methods in the superclass as if they were declared internally. It is possible for the subclass to override methods in the superclass with its own, simply by locally defining a method with the same name.

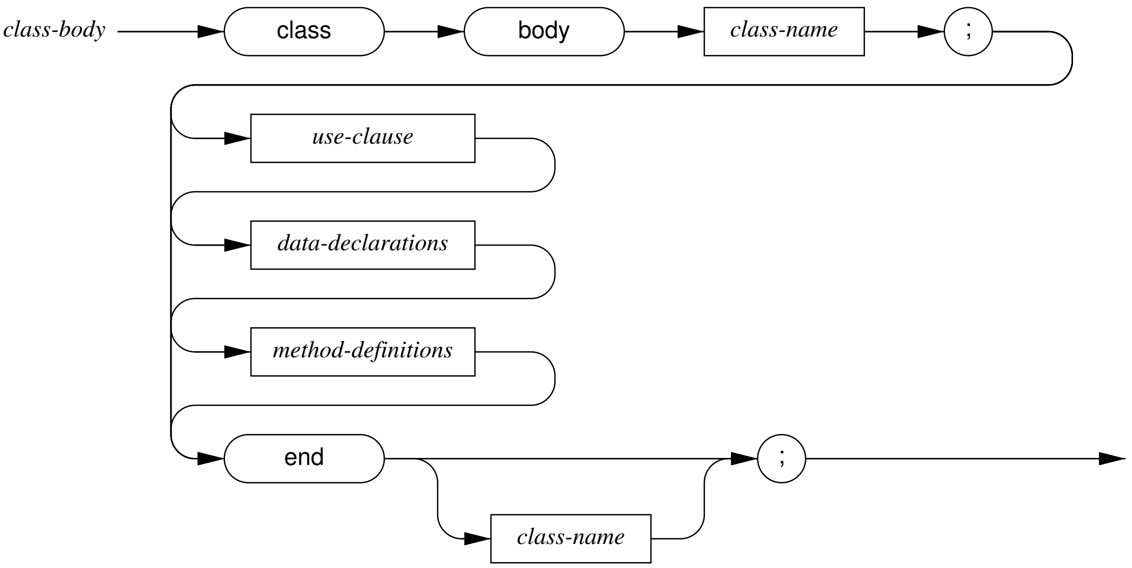
4.2 Overall Structure of a Class

The syntax of a class definition closely resembles a package definition. A class is treated as a compilation unit, at the same level as a package or program. It is not possible to embed a class within a package, program or other class. Like packages it consists of two parts, a class specification and a class body. It is not necessary for the specification and body to be in the same source file, but unlike packages there is no benefit in separating them. The class specification contains a list of superclasses and names of methods and data elements visible to units which use the class. The complete syntax of a class specification is:



Each of the components of a class specification will be elaborated in following sections.

A class body contains the bodies of methods declared in the class specification, along with methods and data elements visible only within the class. The syntax of a class body is:



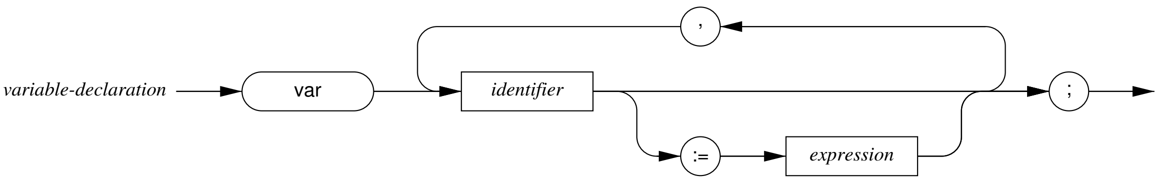
We have adopted syntax similar to the package specification / package body syntax for consistency, but there are a few differences we would like to point out. First, any procedure in a class specification is treated as a method and must be called as such. Second, the inherit clause is placed within a class specification, not a class body. Inheriting a superclass is quite different from using a package. The distinction will be explained in detail in 4.6.

If this is a little cryptic, please be patient. Each of these concepts will be explained in following sections, but before we get into details it is important to see how a class is declared at the outermost level.

4.3 Instance Variables And Class Variables

Data elements in classes are divided into four categories, based upon where they are stored and where they are visible. A data element can be stored either in an object, in which case there is a distinct element owned by each class instance, or it can be global to all instances of the class. We will refer to an element stored with an object as an *instance variable* and a variable global to all instances as a *class variable*.

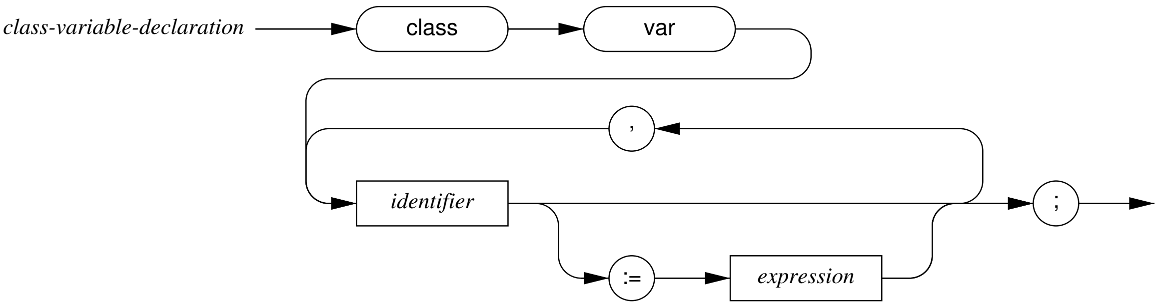
An instance variable is declared with the same syntax as a variable declaration:



Note that all instance variables *must* be explicitly declared. Instance variables in the *current instance* (see 4.4) may be referenced by name only and instance variables in other instances are referenced by *instance.name*.

The initialization clauses on a instance variable are assignments executed when an instance is created, but before any create method (see 4.5) is called.

Class variables are declared in a similar manner to instance variables, but we require an extra keyword:



The initialization clauses on class variables are executed when a class is loaded by the interpreter. This can happen at one of two times: If a class is explicitly used by a program, the class is loaded at the start of execution. Otherwise, the class might be loaded implicitly by the unbinstr procedure (see 6.2) or the getb procedure.

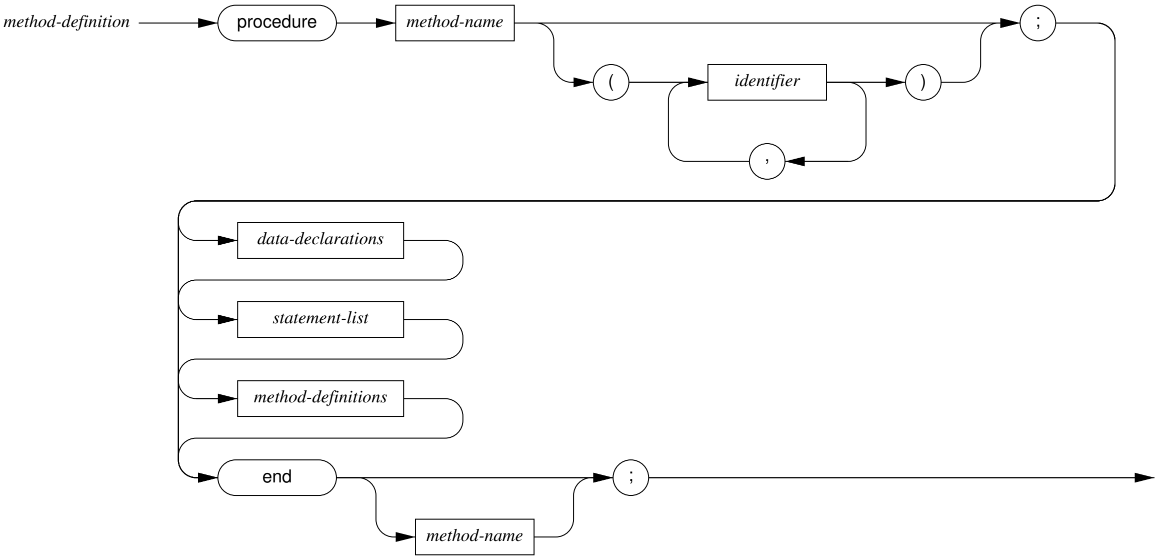
At this point we will start on an example, to illustrate the concepts presented so far. Suppose we are building a system for a corral, and we would like a class to store information about horses, donkeys, etc. We will start out with the following outline of a class. At this point we are only declaring instance and class variables, not methods.

```
class beast_of_burden;
    class var total_beasts;              -- count of instances of beast_of_burden
    var kind_of_beast;                   -- "horse", "donkey", etc.
end beast_of_burden;

class body beast_of_burden;
    class var beasts_in_use := { };      -- set of beasts currently assigned
    var assigned_to;                     -- who has a particular animal
end beast_of_burden;
```

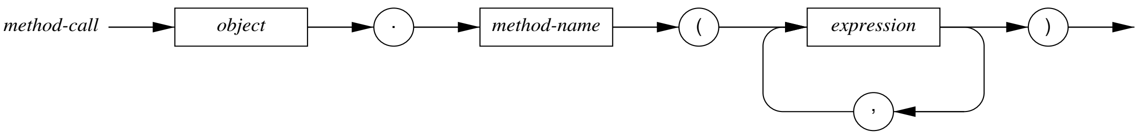
In this example the variable total_beasts is available to any unit using the class, and there will be only one such variable. Each instance of beast_of_burden will contain a variable kind_of_beast and any unit using the class will be able to use or change that variable. The variable beasts_in_use will be visible only within the class body and a single copy is shared by all instances. Each instance of beast_of_burden will contain a variable assigned_to but it will be visible only within the class body.

4.4 Methods

A method is similar to a procedure, except that it is owned by a particular instance of a class, called the *current instance*. The syntax of a method definition is identical to a procedure definition, except that read-write and write-only parameters are not allowed:



Methods are called with an expression like this:



Here *object* is an instance of a class in which *method-name* is a method. The method will be invoked and *object* will become the current instance. Any references to instance variables which are not preceeded by an explicit object will refer to variables in the current instance. A method may also refer to objects other than the current instance, using the expression *instance*.*instance-variable*.

Methods may be called within a class body without the instance prefix. In that case the current instance will remain current. There is also a new nullary operator, self, which will produce the value of the current instance.

Expanding on our corral example, suppose we now wish to provide a method to allow clients to check out beasts. With this method added our beast_of_burden class is as shown in figure 1. With that modified class we can assign a beast to a client with the following method call.

```
beast.check_out(client)
```

The current instance in a method call is analogous to a read-write parameter in a procedure. Internally, the **SETL2** interpreter stores objects as tuples, where the first element of the tuple is a key indicating the class of the object and the remainder of the tuple stores the values of instance variables. When a method is called, the instance variable values will be copied into the instance variables. When the method returns, those values will be copied back into the tuple. It is important to note that the current instance in a method call can change as a result of the call, but *not* until the method returns. This protocol is exactly the same as passing parameters by copying, not by reference.

```
class beast_of_burden;
    class var total_beasts;                   -- count of instances of beast_of_burden
    var kind_of_beast;                        -- "horse", "donkey", etc.
    procedure check_out(client);              -- assign beast to client
end beast_of_burden;

class body beast_of_burden;
    class var beasts_in_use := {};            -- set of beasts currently assigned
    var assigned_to;                          -- who has a particular animal

    procedure check_out(client);              -- assign beast to client
        assigned_to := client;
        beasts_in_use +:= self;
    end check_out;

end beast_of_burden;
```

Figure 1: Corral example with assignment method

4.4.1 Methods As First Class Objects

A method may be used as a first class object just as a procedure can, but only if there is an implicit instance variable included. Consider the method check_out in our corral example. The way to use its value is as follows:

```
result := beast.check_out;
```

The value of result is similar to a procedure. If used in a procedure context, i.e. y := result(x), then the method check_out will be invoked with beast as the current instance. It is a little like absorbing beast into the method.

This system was chosen for consistency with procedure values. When a procedure value is used, **SETL2** saves the environment of that procedure, or the current activations of all enclosing procedures. Within a class body the current instance is part of a method's environment as well. A method value is denoted by the method name alone within a class body, so we bind the current instance to the method and save the combination as a procedure. We do not allow methods to be used as first-class objects without an associated object.

4.5 Creating An Object

An object is created with a call to a class, so continuing with the corral example the statement to create a new beast of burden is:

```
new_beast := beast_of_burden();
```

As part of the creation process, each of the initialization clauses on the instance variable declarations are executed.

Although it isn't necessary, it is possible to provide a create method which accepts parameters and uses them to initialize the created instance. Such a method must appear in both the class specification and class body, since it must be visible outside the class body. Any parameters on the creation call will be passed to create. Thus if beast_of_burden contained the following method:

```
procedure create(a,b);
    kind_of_beast := a;
    assigned_to := b;
end create;
```

then the previous creation call might look like this:

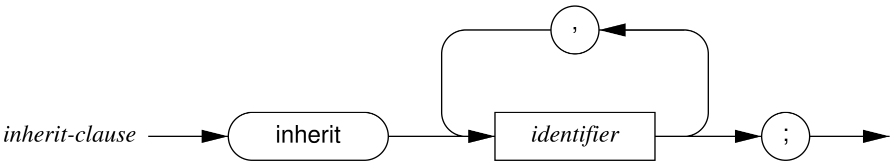
```
new_beast := beast_of_burden("horse","George");
```

The number of actual parameters on the creation call must agree with the formal parameters in create. Notice that create does not return anything. The create procedure does not actually create the new object. When create receives control, the object has been created by the interpreter and its instance variables have been set according to the initialization clauses on the instance variable declarations. The create procedure is invoked after this preliminary initialization, and the new object will be installed as the current instance. When create terminates that current instance will be returned. Any value returned by create will be ignored.

We would like to point out that create is *not* a reserved word. It is only special in that if a method named create is present in a class specification it will be called implicitly at the time an object is created.

4.6 Inheritance

The concept of inheritance among classes is related to the concept of using a package or class, but considerably stronger. The fundamental difference is this: When a class is used by a program, package, or another class the methods and instance variables are only available as components of objects of that class. When a class is inherited, its instance variables and methods become a part of the subclass, as if the superclass were textually inserted in the subclass. It is possible and reasonable for a class to both use and inherit the same class. This probably won't become clear without some examples, but we have to present the syntax first. The syntax of an inherit clause is:



The inherit clause is placed in a class specification, after the header and before any other declarations. Each of the *identifier*'s above must be classes available in the library. All variables and methods in the superclass will be brought into the current class, unless there are name conflicts. Here are the rules for resolving those:

- No variable names may be redefined.
- Method names follow similar rules to packages. A local definition overrides an inherited one. If there are conflicts among inherited names, all are hidden. Hidden names are accessible with the syntax: *superclass.method-name*. Hidden names are accessible only within class bodies, and then only for names in superclasses.

To illustrate the distinction between use and inherit, suppose we have a class animal in our library which looks like this:

```
class animal;
    var species;
    procedure create(kind);
end animal;
class body animal;
    var owner;
    procedure create(kind);
        species := kind;
    end create;
end animal;
```

Now a beast_of_burden is also an animal, so we might want to make use of some of the facilities in animal within beast_of_burden. What happens if we *use* the class animal? Then within beast_of_burden we can create animals and store and retrieve their species. We will *not* be able to access the owner instance variable because it is not public. Essentially, we can use an animal as a component of a beast of burden, but that's all.

Now suppose beast_of_burden *inherits* animal. In this case all of the instance variables in animal are available as part of a beast_of_burden. So if silver is a beast_of_burden we can get its species with silver.species. If beast_of_burden does not declare its own create method, then the one in animal will be called when a beast_of_burden is created. If it *does* contain a create method, then that method will be called. If we want to call the method in animal, it can be called with an expression like animal.create("horse").

The only difference between inheriting and textual inclusion is that redefined methods are hidden, not an error, and those hidden methods can be accessed by including the class with the name.

Another consequence of the differences in strength between inheriting and using a class, is the distance of the name transfer. Suppose a, b, and c are classes. If a uses b, and b uses c, then nothing in c is visible in a (unless of course a also uses c). That is not the case with inheritance. Remember it is similar to textual insertion *and is recursive*. If a inherits b and b inherits c then everything in c not blocked by a redefinition in a or b is visible in a. Even hidden methods in c are accessible with the expression c.*method-name*.

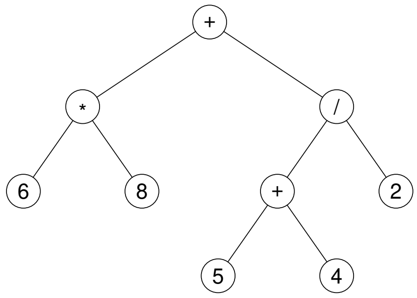
4.6.1 An Application Of Inheritance

At this point it seems worthwhile to illustrate inheritance with a somewhat more realistic example. Suppose we have a class in our library for ordered trees (an ordered tree is just a tree in which the order of child nodes is significant). A reasonable representation of ordered trees is a set of nodes, a distinguished root, and a map from nodes to children, where children are represented by tuples. That is, the first child of a node will be child(node)(1), the second child will be child(node)(2), etc. An outline of this class with a depth first search method might be as shown in figure 2.

Now suppose we are building a system which must manipulate expression trees. An expression tree is a kind of ordered tree, but each node represents an operator or a literal. The children of an operator node are its associated operands. To evaluate an expression, we can perform a depth first traversal of the tree, evaluating each node after we have evaluated its children. For example, the expression tree for (6 * 8) + ((5 + 4) / 2) would be as follows:

```
class ordered_tree;
    procedure dfs(p);
end ordered_tree;

class body ordered_tree;

    var nodes := {}, root, child := {};

    procedure dfs(p);                -- expect procedure to execute on each node

        recursive_dfs(root);

        procedure recursive_dfs(current);
            if current = om then
                return;
            end if;
            for i in [1 .. #(child(current))] loop
                recursive_dfs(child(current)(i));
            end loop;
            p(current);              -- visit the node
        end recursive_dfs;

    end dfs;

end ordered_tree
```

Figure 2: Ordered tree class



Expression tree for (6 * 8) + ((5 + 4) / 2)

Clearly, any operation defined on ordered trees might also be useful on expression trees, so this seems a good application for inheritance. We want to inherit ordered trees, but we have to add a couple of things. We would certainly want a label associated with each node, to hold either an operator or a literal. We would also like a method to return the value of an expression. A class for expression trees might look as shown in figure 3.

From this example we can get some notion of the situations in which inheritance is useful. Usually the subclass should be a specialization of the superclass. In our example above expression trees are clearly a specialization of ordered trees. Furthermore, it must be a large enough specialization that the underlying data structures of the superclass are still useful. For example, an expression tree is also a tree, but it is unlikely that a tree would be implemented in such a way that it would be a useful superclass unless we insist that it be ordered. It would probably be stored as a set of edges, or at best the children of a node would be a set rather than a tuple. Either of these representations are awkward for expression trees. A good indicator of applications of inheritance is to find situations in which variant records would be used in languages which

```
class expression_tree;
    inherit ordered_tree;
    procedure evaluate;
end expression_tree;

class body expression_tree;

    var label;

    procedure evaluate;

        return eval(root);

        procedure eval(current);
            case label(current)
                when "+" => return eval(child(current)(1)) + eval(child(current(2)));
                when "-" => return eval(child(current)(1)) - eval(child(current(2)));
                when "*" => return eval(child(current)(1)) * eval(child(current(2)));
                when "/" => return eval(child(current)(1)) / eval(child(current(2)));
                otherwise => return label(current);
            end case;
        end eval;

    end evaluate;

end expression_tree;
```

Figure 3: Expression tree class

have them.

4.7 SETL2 Operator Overloading

It is possible to overload **SETL2**'s built-in operators with methods defined on classes, and it is necessary if you want to use those operations on objects. Clearly the **SETL2** interpreter will not know how to add two objects unless an addition method is explicitly defined. To define an addition method for the class beast_of_burden, for example, the following could be placed in the class body:

```
procedure self + right;

    var result;

    result := beast_of_burden();
    if kind_of_beast = right.kind_of_beast then
        result.kind_of_beast := kind_of_beast;
    elseif {kind_of_beast, right.kind_of_beast} = {"horse","donkey"} then
        result.kind_of_beast := "mule";
    else
        abort("Unknown cross-breed: ", kind_of_beast, " and ", result.kind_of_beast);
    end if;
    return result;

end;
```

With this method defined two beasts can be mated with the normal **SETL2** addition operator, i.e.

mother_beast + father_beast.

There are a few things which are important to notice here. First, it is not necessary to declare this method in the class specification, only to define it in the class body. Since this is essentially a hook into the **SETL2** syntax it is always available. In fact, if an addition is attempted without this method being defined the interpreter will abort the program.

The second thing to notice is that it is the responsibility of the method to create a result instance and to explicitly return it. The current instance is not usually returned, although it could be. Note that you will not normally want to modify the current instance in one of these operations, although you can do so. If this is done you will just have to understand that the expression a + b might modify a as well as yielding a value.

Finally, the headers of operator methods are quite unusual. We include the operator itself in the header rather than an identifier. This has the advantage of being easy to remember, but the disadvantage that the method can not be explicitly invoked. The only way to call an operator method is with ordinary expression syntax. The lack of a name means that these methods do not have first class values and that once hidden they are completely inaccessable.

For a good example of the power of operator overloading, see the multiset example in Appendix A.

4.7.1 Binary Operator Methods

Most of **SETL2**'s binary operators have two associated methods. We always use the operator itself in the header, so they are fairly easy to remember. The headers have the following forms:

procedure self *binary-op* id_1 ;

procedure id_1 *binary-op* self ;

The reason for the two forms is that an object might appear either as the left or the right hand operand in an expression. Most **SETL2** operations require both operands to be of the same type, but there are exceptions. The * operator, for instance, will operate on integers and strings or integers and tuples, in which case the operands can appear in either order. We allow the two forms of binary operators to enable the same sort of thing here. The first form above is used if the left operand determines the method used, in which case the left operand will become the current instance. Otherwise the second method will be used and the right operand will become the current instance.

In deciding how to process a binary operation the **SETL2** interpreter gives precedence to the left operand. That is, it goes through the following steps before performing the operation:

1. If the left operand is an object and that object has a corresponding method then that method is used and the left operand will become the current instance.
2. If the left operand is not an object or it doesn't have a left operand method but the right operand is an object with a right operand method, then the right operand method is used and the right operand will become the current instance.
3. If neither operand has an appropriate method than the program is aborted.

Each of these methods should return a value, although that is not enforced by the system. If no value is returned, then Ω will be used, which is probably not what is desired. The method should create a new instance, build its values, and return it.

Here is a complete list of the binary operator methods.

+	-	*	/	**	mod
min	max	with	less	lessf	npow

4.7.2 Unary Operator Methods

All of **SETL2**'s unary operators also have associated methods. The headers for these methods have the following form:

procedure *unary-op* self ;

The particular method used will always be determined by the class of the value of the operand, which will become the current instance. Each of these methods should return a value, although that is not enforced by the system. If no value is returned, then Ω will be used, which is probably not what is desired. The method should create a new instance, build its values, and return it. Here is a complete list of the unary operator methods.

-	#	arb	domain	range	pow

4.7.3 Relational Operator Methods

Relational operators are somewhat of a problem, particularly the equality and inequality operators. For one thing, in order to reduce the number of branch instructions the **SETL2** code generator assumes that a < b iff b > a. Therefore, we allow a < method but do not allow a > method. For each operation we invoke the < method, but in the expression a < b, a will become the current instance and in a > b, b will become the current instance.

The difficulty with equality and inequality is that these primitive operations are used in determining set and map membership, as well as being explicitly used in **SETL2** programs. To further complicate matters, **SETL2** uses hash tables to implement sets, so we must absolutely guarantee that two equal values produce the same hash code. Since we can see no way to enforce this, we do not allow equality and inequality to be overridden. Two objects are considered equal if they are instances of the same class, and if all their corresponding instance variables are equal.

Because of these restrictions, only < and in have associated methods. The < method has two forms (left and right) just like other binary operators and follows the same rules. The in operator also has two forms but the precedence is reversed. We give precedence to the *right* operator in in expressions.

Each of these methods *must* return either true or false.

The < method will be called for any of the expressions: a < b, a <= b, a > b, or a >= b. The expression a <= b, means the < method returned true or the objects are equal in the sense described above.

4.7.4 Map, Tuple, And String Component Methods

SETL2 has four expressions used to refer to portions of maps, tuples or strings: f(x), f{x}, f(i..j) and f(i..). Each of these expressions can appear in both left and right hand side contexts. The ability to overload these syntactic constructs is particularly valuable, since it enables us to define our own aggregates organized any way we like, while retaining the ability to access components of those aggregates quite elegantly.

Each of these expressions has two associated methods, one for left hand contexts and one for right hand contexts. The syntax of the headers for each of these methods is as follows:

```
procedure self ( id1 ) ;

procedure self ( id1 ) := id2 ;

procedure self { id1 } ;

procedure self { id1 } := id2 ;

procedure self ( id1 .. id2 ) ;

procedure self ( id1 .. id2 ) := id3 ;

procedure self ( id1 .. ) ;

procedure self ( id1 .. ) := id2 ;
```

Each of the methods for right hand side contexts (those without an := symbol) should return a value. Otherwise Ω will be returned. The methods for left hand contexts should not return anything, but the rightmost argument should be used to modify the current instance.

4.7.5 Deletion Operations

SETL2 has three deletion operations, from, fromb, and frome, and methods for all of these may be defined on objects. The syntax of the method headers is:

```
procedure from self ;

procedure fromb self ;

procedure frome self ;
```

These must each return a value, or Ω will be used. Notice that there is no left operand in these headers, even though each is a binary operator. The left operand in a deletion operation is written but not read. Whatever value these methods return will be assigned to the left operand as well as the target operand.

4.7.6 Iteration Over An Object

SETL2 has several syntactic constructs which call for iteration over an aggregate. For instance, in the expression:

```
{x in S | x < 5}
```

the interpreter will iterate over S screening each element with the condition x < 5 and inserting into the result any value which satisfies that condition. Iterators are used in set and tuple forming expressions, for loops and quantified expressions. There are two general forms of iterators:

expression$_1$ in *expression*$_2$

expression$_1$ = *expression*$_2$ { *expression*$_3$ }

Note that the expression y = f(x) is equivalent to [x,y] in f, and so is included in the first form above.

We have two pairs of built-in methods, corresponding to these two syntactic constructs:

```
procedure iterator_start ;

procedure iterator_next ;

procedure set_iterator_start ;

procedure set_iterator_next ;
```

When the interpreter encounters code requiring an iteration over an object, it calls the iterator_start or set_iterator_start method depending on whether the iterator was of the first or second form above. Then it repeatedly calls iterator_next or set_iterator_next to get successive elements of the object.

The iterator_start and set_iterator_start methods need not return a value. They are only there to let the object initialize an iteration. The iterator_next and set_iterator_next methods should return the next element in the object *within a tuple* if such an element can be found, or Ω if there is no such element. The tuple enclosing the result value is used by the interpreter to determine if the iterator method was able to produce a value.

If the iterator expression is y = f(x), then the first pair of iterator methods will be used, but each return value must be a pair, so each return will look something like this:

```
return [[x,y]];
```

Notice the double brackets. The outer tuple indicates that a value was found, and the inner tuple is the pair of values required by this iteration form.

If the iterator expression is y = f{x} then the second pair of iterator methods will be used. The return values must be the same as for y = f(x) iterators.

None of the names of methods described in this section are reserved words. If not used as iterator methods, they can have any number of parameters and return anything you like. If they are to be used for iteration, they must conform to the rules above, or the program will be aborted.

4.7.7 Printing An Object

Objects are printed by first calling the built-in procedure str, then printing the string. The default value produced by str is useful mainly for debugging. It prints all the instance variables, but in an ugly format. This string can be overridden with a method having the name selfstr, declared with the following header:

```
procedure selfstr;
```

If this method is provided it will be called by str for objects of the relevant class. It can return any value, but ideally should return a printable string of the object.

4.8 Testing An Object's Type

The type of an object can be determined with the built-in type procedure. The value returned will be the name of the object's class as an upper case character string. **SETL2** is not case sensitive, and always keeps names as upper case.

4.9 Storing Objects In Files

Objects may be stored in either text or binary files, but may only be re-read from binary files. If an object is read from a binary file by a program which does not explicitly use the object's class, then the class will be loaded at the time the object is read. A similar load takes place if an object is converted from a string value with the unbinstr built-in procedure. After the class is implicitly loaded, all the methods corresponding to built-in operations will be available on objects of that class.

5 An Interface With C Functions

One extremely useful feature in very high level languages, previously lacking in **SETL2**, is the ability to call procedures in lower level languages. There are two compelling reasons why this is desirable: to make use of the vast amount of existing software written in other languages, and to code small portions of programs in a lower level language for efficiency. Unfortunately, providing such an interface raises many small technical problems, takes quite a bit of coding, and holds no research interest. For these reasons it is not a very high priority in spite of its importance.

As an interim solution to this problem, a crude call-out and call-back facility has been implemented. It is a kludge, but it does work and provides the ability to experiment with interfaces between **SETL2** and other languages before a more ambitious implementation is available. It requires re-linking the **SETL2** interpreter with the functions to be provided along with a substantial amount of glue code. There is also glue code required on the **SETL2** side. The following compromises were made in order to provide some level of call-out, with a minimum of effort:

1. The only language supported is C. Interfaces to other languages have to go through the same glue code, so have to go through C.

2. There is no name-binding between C and **SETL2**. Neither C nor **SETL2** have access to the other's identifiers, or even the ability to directly call the other's functions or procedures. **SETL2** is able to call a single pre-set C function, which may then re-route the call to the function desired based on the arguments passed from **SETL2**. There is no linking directly from **SETL2** to arbitrary C functions.

3. All parameters are passed as C character strings. This greatly increases the cost of call-out, but means we don't have to deal with the numerous type-conversion issues at this time.

4. There is the ability for C functions to call back into **SETL2**, but with similar restrictions.

So there is a crude, clumsy, callout capability which we expect to replace when time permits. We hope most readers are scared off by now. If so jump to section 6. For the intrepid, trudge on.

5.1 Call-Out

Call-out from **SETL2** to C is accomplished through two procedures. On the **SETL2** side, the user should call a new built-in procedure, callout. When callout is invoked, the interpreter will do some tranformation of the arguments, then call the C procedure `setl2_callout`, which must be provided by the user and linked with the **SETL2** interpreter. The procedure callout expects the following three arguments:

1. The first argument should be an integer service code. Remember that **SETL2** knows nothing of the C names, so will always call the same C function. The integer service code is convenient for the C function to use as a selector in a `switch` statement, to in turn call the function really desired.
2. The second argument should be a **SETL2** procedure, which will be called if the C function calls back into **SETL2**. The C functions can no more see the **SETL2** names than **SETL2** can see C names, so we must let the interpreter know how to handle callbacks. We'll go into much more detail on this below. If it is not necessary for the C functions to call **SETL2**, this argument can be anything, but Ω seems most appropriate.
3. A tuple of strings, comprising the data to be sent to the C function.

When callout is called, it will save the call-back handler (C will always call the same function, which will in turn call the call-back handler), convert the **SETL2** data into corresponding C forms, and call `setl2_callout`, which must have the following prototype:

```
char *setl2_callout(int service, int argc, char **argv);
```

The first argument is obviously the service code from **SETL2**. The second is the length of the tuple passed to callout, and the third is an array of pointers to the actual strings in that tuple. The function `setl2_callout` should convert those strings into C internal form and call some other C function based on the service code.

To illustrate, let's go through an example. Suppose we have a **SETL2** program which operates on matrices. We need a determinant function, and happen to have one available in C. We would like to have this available in **SETL2** since it seems to be a generally useful procedure. The first step is to decide how we would like to call the procedure in **SETL2**. This seems to be a reasonable setup:

```
result := determinant(m);
```

where m is a tuple of rows, and each row is a tuple of reals. We would expect result to be a real. The glue code on the **SETL2** side consists of converting the arguments into strings and passing the strings to callout, along with an integer code indicating the service we would like performed. We will assume the matrix is square. We want to pass the size of the matrix as the first element of the tuple followed by each cell, listed row by row. The glue code to perform this transformation might look like this:

```
procedure determinant(m);
   return unstr(callout(1,om,[str(#m)]+[str(x) : x in +/m]));
end determinant;
```

Now we move to the C side. The user must provide a function prepared to accept the arguments from the **SETL2** side. Here is a skeleton, assuming we have a function `c_determinant` which will calculate determinants.

```
char *setl2_callout(
   int service,               /* an integer service code          */
   unsigned argc,             /* length of argument vector        */
   char **argv)               /* argument vector                  */
{
static char return_string[100];
   switch (service) {
      case 1 :    /* this is our determinant service              */
      /*  You get the idea.  The first string in argv is the size */
      /*  of the matrix, the others are matrix data.  You have to */
      /*  take these strings and set up to call c_determinant().  */
      sprintf(return_string,"%f",c_determinant(/* args */));
      return return_string;
   }
}
```

There are a couple things to notice here. First, the service code is intended to be used as a selector in a `switch` statement, but you can do anything you want with it, including ignore it. It is also reasonable to use the first character string as the selector, so that you can get something readable when you print it, but then you would have to use `if..then..else`'s to pick out the C function you wish to call.

The second thing to notice is that setl2_callout *must* return a character pointer. That pointer can be `NULL`, but it can not be some random value. If it is you will get a segmentation error if you are lucky, and you will destroy random data in your program if you are unlucky.

5.2 Call-Back

Call-back is the ability of a C function to access services in **SETL2**. This is even uglier than call-out, so skip to section 6 if you don't really need this capability.

From the C side, you will call a function in the **SETL2** interpreter passing it character strings as arguments, and accepting a character string as return value. The function called is `setl2_callback`, and has the following prototype:

```
char *setl2_callback(char *firstarg,...);
```

The **SETL2** interpreter uses the ANSI C convention for functions accepting a variable number of arguments, so you will have to include `stdarg.h` in any C source files containing a call to `setl2_callback`. Any number of character pointers may be passed as arguments, but the final one *must* be a `NULL`. If you forget the `NULL`, `setl2_callback` will keep reading arguments until it happens to run into a `NULL`, or tries to access something which upsets the operating system. Again, if you're lucky you'll get a segmentation error, if not you'll destroy random data and continue.

The function `setl2_callback` in the **SETL2** interpreter will gather up all arguments except the first one into a tuple and pass them to the procedure passed as a call-back handler in callout. That procedure should have been declared with two arguments. We assume the first will be a service code, and the rest are data used to perform that service.

As an example, let's suppose there are some global variables in the **SETL2** side which we want to make available in C. We only want to allow C to reference them, not set them. We will provide a procedure get_value, which will return the value of those variables. The C statement to get the value of the variable user in the **SETL2** program is as follows:

```
void some_c_function();
{
char user_name[100];
   strcpy(user_name,setl2_callback("get_value","user",NULL));
}
```

The return value will be a pointer into the space of `setl2_callback`, so you have to do something with the return value before you call that function again. In this case we immediately copied the return value to local storage. Remember the `NULL` at the end of the argument list, without it you'll have big problems.

Now we move back to the **SETL2** side. We must call out to C passing a procedure which can handle call-backs. The call-back handler should look at the first argument to decide what service to provide, then use the remaining data to provide it. That should generally be done by calling some lower level provider. Here's an outline of a procedure which calls out to C and accepts call-backs.

```
procedure use_c_for_something;
    callout(some_service,callback_handler,some_args);
    return;

   procedure callback_handler(service,args);
      const callback_map := {["get_value",get_value]};
      return callback_map(service)(args);
   end callback_handler;

   procedure get_value(variable);
      [variable] := variable;            -- disassemble the tuple of strings
      case variable                      -- return one value
         when "user" =>                  -- should be more of these
            return user;
      end case;
   end get_value;

end use_c_for_something;
```

WARNING: This is just an outline. You'll have to fill in many details, particularly for error checking!

5.3 Avoiding String Conversions

If you are transfering large arrays of numeric data to and from a C function, the cost of converting that data to string form and back might become expensive. There are two new built-in procedures in **SETL2** which can help avoid that, binstr and unbinstr. These procedures convert **SETL2** values into a character string form and back, conceptually like str and unstr, but do not produce easily readable character strings. Their purpose is to provide a quick and reversible conversion, so unbinstr(binstr(x)) is always equal to x, but unstr(str(x)) is not necessarily x. **Caution:** The binary string of an atom or procedure can not be stored on disk and converted back in another program. These values are pointers, and only have a life of a single program execution.

The format of a binary string is dependent on the internal representation of **SETL2** values, and is therefore subject to change. We're only going to describe the format of integers, reals, and tuples here, but even that should not be taken as gospel.

Each binary string consists of a number of binary values concatenated. There are no alignment characters included, so even though we will use C structures to show the sequence of fields, you must understand that these are not really structures. If your C compiler must add extra characters for alignment, you'll have to pick apart these structures as character strings.

An integer consists of the following fields:

```
struct {
   int form_code;
   long int_value;
};
```

We assume type checking is done on the **SETL2** side, so the form code can be ignored. The integer value is a C `long`, but only the low-order half less one bit is used. If you pass integers longer than that you will get a long integer structure, which we prefer not to describe. This is only for short integers.

A real consists of the following fields:

```
struct {
   int form_code;
   double real_value;
};
```

Again, you can ignore the form code. The format of the `double` varies with machine implementation. We prefer IEEE long format, and that is how the system is compiled if IEEE long is available.

A tuple is a little more complex. The first thing you will find is a header, which has the following fields:

```
struct {
   int form_code;
   long tuple_length;
};
```

Here `tuple_length` is the number of components in the tuple. Following this header will be each of the components in sequence. They will be concatenated with no intervening spaces or nulls for alignment. This will be a little difficult to handle unless you verify on the **SETL2** side that the tuple is homogeneous.

5.4 Putting It All Together

Now that we have all the pieces, how do we assemble them? First notice that you should be thinking in terms of extensions to **SETL2**, rather than linking a function into **SETL2** for one specific program. It's a bit too much work if you aren't adding a generally useful feature. It's very important, though not required, to hide all this in a package rather than coding call-outs in more general **SETL2** programs. Remember that this is a *temporary* facility. We do intend to replace it with something easier to use and requiring fewer data conversions, so minimize the number of calls to callout appearing in your code. If you restrict it to a single package conversion will be easier when a better facility is available.

With this document, you should have received the **SETL2** interpreter in library form, along with a sample `Makefile` and skeletons of both a C call-out handler and a **SETL2** package for call-out. To assemble all this do the following:

1. Modify the `Makefile`, C source file, and **SETL2** source file to use the C procedures you want to provide in the **SETL2** interpreter.

2. Create your customized version of the **SETL2** interpreter by running make.

3. Compile the **SETL2** package.

4. Insert use callout_package in the beginning of any programs or packages which use the procedures in the **SETL2** package.

The **SETL2** interpreter contains many calls to library routines, so you'll have to have available the libraries we use. Here are the libraries (and possibly compilers) you will need:

Unix	Gnu C and `gnulib`.
VMS	VMS C.
Macintosh	MPW C 3.0.

6 New Built-In Procedures

We have added a number of new built-in procedures, primarily for input and output. There are also a few new string handling procedures, but these are really designed to support I/O as well.

6.1 Input-Output

The most drastic change in I/O so far is the addition of a *random* file type. Random files don't really fit nicely in **SETL2**, since there is no convenient fixed-length type. We implemented a kludgy random file system by assuming all values read or written will be character strings, and relying on the **SETL2** programmer to convert to internal types. We have provided procedures to do a default conversion, but these yield value strings of wildly varying length for similar types, so will be tricky to handle in random files.

A strong word of caution: We are not very satisfied with **SETL2** I/O, and are considering a radical redesign. The new procedures we have provided were chosen to support an I/O system based on map and string syntax, which will be implemented using objects.

Here are the new or changed procedures:

h := open(f,m) — The function of open has not changed, but there are two new file modes. Here is the complete list:

"text-in"	File will be opened for input in text, or formatted, mode. It may then be accessed with reada or geta.
"text-out"	File will be opened for output in text, or formatted, mode. It may then be accessed with printa.

"text-append"	File will be opened for output in text, or formatted, mode. The file is positioned at EOF. It may then be accessed with printa.
"binary-in"	File will be opened for input in binary mode. It may then be accessed with getb.
"binary-out"	File will be opened for output in binary mode. It may then be accessed with putb.
"random"	File will be opened for random access. All records read from or written to the file must be character strings. The file may be accessed with gets or puts.

gets(h,s,l,wr v) h must be a file handle created with a call to open. s and l must be integers. A string of length l starting from file position s will be read into the variable v.

puts(h,s,v) h must be a file handle created with a call to open. s must be an integer, and v must be a string. v will be written to the file starting at position s.

fsize(h) h must be a file handle created with a call to open, and the file must have been opened in random mode. This procedure returns the length of the file, in characters.

nprint(v1,v2...) Each of the values v1,v2... will be printed on the standard output device (usually the terminal). There will be no spaces or newlines between the values. The only difference between print and nprint is that nprint does not print a newline at the end.

nprinta(h,v1,v2...) h must be a file handle created with a call to open. Each of the values v1,v2... will be written to that file. There will be no spaces or newlines between the values.The only difference between printa and nprinta is that nprinta does not print a newline at the end.

6.2 String Handling

unstr(s) This procedure operates like reads, but has no write parameters. It converts a character string to internal form, following the same scanning procedure, and returns the internal value.

binstr(v) v can be any **SETL2** value. binstr converts v to a character string which is not readable, but can be converted back into its original form. Combined with unbinstr, this has the property that unbinstr(binstr(v)) = v, which is not true of str and unstr. It is valuable in writing data to random files or passing binary data in callout (see section 5).

unbinstr(s) — s must be a value created by binstr. unbinstr converts the string back into internal form.

Caution: Atoms and procedures may be converted to strings and back *only within a single program execution*. Their string representations may not be written to a file, read back in another execution, and converted to internal form. They are comparable to memory pointers in lower level languages.

6.3 System Access Procedures

callout(n,p,t) — Used to call functions written in other languages. See section 5 for a detailed explanation.

abort(s) — s must be a character string. abort calls the interpreter's abnormal end handler passing it s as an error message. The program is aborted and the message, current source location, and stack are printed. This is most useful in writing packages and classes meant to be used by other programmers or in program traps, since it is most appropriate in program error situations, not user error situations.

A Multisets: An Example Class

The ability to overload **SETL2**'s built-in operations on objects allows us to add new types to the language fairly easily. To illustrate this, we will construct a class for multisets, or *bags*. A bag is similar to a set, but we don't require all elements of a bag to be unique. We are not aware of any well-established mathematical properties of bags so we'll define those we need ourselves.

The first step in adding a new type is to decide which operations we want to support. In the case of bags, we would like to allow any of the set and map operations if that is possible. An ambitous set of operations is defined in figure 4. In addition, we want to be able to iterate over a bag and print it in some reasonable form.

Having established the operations we would like to allow, we now choose our data structures. At first glance, a good candidate seems to be a map from distinct elements to counts. The problem with this data structure is that the map-oriented operations (domain, range, lessf, and f{x}) will all be very slow, so let's consider alternatives.

Another option is to use the data structure just considered, but change the representation to some other format for maps as needed. The **SETL2** interpreter uses a scheme like this for maps. Essentially, we hope that the application program will use either map operations or set operations in long sequences, so we don't have to change representations very often. We have a problem with this data structure as well. Suppose we have an equality check on two bags, in which one happens to be in map format and the other is not. The equality test will yield false, even if the two bags contain the same elements, only the representation is different. Within the interpreter, the values would be converted to a common format before the equality test, but since we can't create an equality method, we can't do the same thing.

The solution is to use two maps, one for pairs and one for everything else. The map of pairs is organized so that the image of the first element of the pair is also a map, and that map is from the second element to the number of occurrences of the pair in the bag. We get the number of occurrences of [x,y] with the expression pairs{x}(y). The map of non-pairs is as described in our first try at a solution, namely it is a map from

#s	Yields the number of elements in s.
arb s	Yields an arbitrary element of s.
pow s	Yields the power set of s. The value will be a *set* of all bags containing only elements in s, and no more of any element than s has.
domain s	domain and range seem to be naturally sets, not bags. We can still use the notation, although it's not clear that it is a useful concept. For domain to work properly, we require each element of the bag to be a pair. We will then yield the *set* of all the left elements of those pairs.
range s	For range to work properly, we require each element of the bag to be a pair. We will then yield the *set* of all the right elements of those pairs.
s + ss	Analogous to set union. We create a bag containing all the elements of s and ss. Notice that we don't flush out duplicates here, so for example if s contains one "a" and ss contains two "a"'s then s + ss will contain three "a"'s.
s - ss	Analogous to set difference. We copy s into the result, then remove each element of ss.
s * ss	Analogous to set intersection. We create a bag, then for each distinct element of s we place the minimum of the number of occurrences of that element in s or ss in the result bag.
s mod ss	Defined as (s + ss) - (s * ss).
s npow i	Yields a subset of the power set, in which each bag has exactly i elements.
s with x	Yields a bag with all elements of s and {x}.
s less x	Yields a bag with all elements of s less {x}.
s lessf x	We require each element of s to be a pair. We remove all pairs with x as left element.
x from s	We select an arbitrary element from s, remove it, and assign the value to x.
x in s	Yields true if x is an element of s, false otherwise.
s = ss	Yields true if s and ss are the same, false otherwise.
s /= ss	Yields true if s and ss are different, false otherwise.
s < ss	Yields true if s $\subset$ ss, false otherwise. We define s $\subseteq$ ss when applied to bags to mean that for each distinct element e of s, there are at least as many occurrences of e in ss as there are in s.
s <= ss	Yields true if s $\subseteq$ ss, false otherwise.
s > ss	Yields true if s $\supset$ ss, false otherwise.
s >= ss	Yields true if s $\supseteq$ ss, false otherwise.
f(x)	This generally yields the image of x in f. It's not quite clear what this means for bags, since we allow duplicates, so let's make an unusual interpretation. Let f(x) yield the number of occurrences of x within f, when used on the right. When used on the left, we expect the right hand side to be an integer, and we will set the number of occurrences of x in f.
f{x}	Yields the image set of x in f. This makes a little more sense than the previous operation. For this expression to work at all, each element of f must be a pair. We look for all the pairs in f with x as left hand element and gather the right hand elements into a bag. If this expression is used on the left, we set the image set of x in f. Note that we must have a bag on the right in this case.

Figure 4: Operations on multisets

distinct elements to counts. We also keep the cardinality explicitly, since that would be fairly expensive to compute.

Having chosen a reasonable data structure, we're ready to start building the class. We're not going to show all the required methods here, since that would take too much space. We'll pick out those which are essential or illustrate an interesting point.

A.1 Class Specification

We don't have any methods referred to by name, all we're providing is **SETL2** operations. Therefore our class specification is almost empty. We do have to provide a creation function and make that globally visible.

```
class bag;
    procedure create(source);
end bag;
```

A.2 Class Body

All our data is private, to insure integrity. We start with declarations of instance variables.

```
class body bag;
    var pairs          := {},          -- pair values
        others         := {},          -- non-pair values
        cardinality    := 0;           -- cardinality
```

A.2.1 Create

We want to allow the programmer to provide an initial set of values on the creation call. We're just going to iterate over those values, so we don't really require a set, we just need something we can iterate over. It might be a set, a map, a tuple, a string, or some object with iteration methods defined.

```
procedure create(source);
    cardinality := #source;
    for x in source loop
        if is_tuple(x) and #x = 2 then
            [left,right] := x;
            pairs{left}(right) := (pairs{left}(right) ? 0) + 1;
        else
            others(x) := (others(x) ? 0) + 1;
        end if;
    end loop;
end create;
```

A.2.2 Number Of Elements

The cardinality operator is a snap. We don't want to have to count the number of elements, since that's expensive. Therefore we maintain the cardinality. This method just returns it.

```
procedure # self;
    return cardinality;
end;
```

A.2.3 Domain

The domain of a bag is the same as the domain of its pairs instance variable. Notice that we must check whether there are any non-pair values, since if there are the bag is not a valid map.

```
procedure domain self;
    if #others /= 0 then
        abort("May not find domain of non-map BAG:\n"+str(self));
    end if;
    return domain(pairs);
end;
```

A.2.4 Bag Union

Perhaps union isn't an appropriate name. What we really mean is the sum of two bags. Notice that we have only one version of this method, though in general there can be two for binary operators. We don't allow mixed mode addition on bags, so we don't need a method with self on the right. If an expression has a bag on the left we will be called. If not we should get a run-time error anyway.

The procedure we follow is pretty straightforward. We copy the left operand, then loop over the right operand adding all the elements to the result.

```
procedure self + right_bag;
    if type(right_bag) /= "BAG" then
        abort("Invalid operands for +:\nLeft => "+str(self)+"\nRight => "+str(right_bag));
    end if;
    result := self;
    result.cardinality +:= right_bag.cardinality;
    for right_map = right_bag.pairs{left}, count = right_map(right) loop
        result.pairs{left}(right) := (result.pairs{left}(right) ? 0) + count;
    end loop;
    for count = right_bag.others(left) loop
        result.others(left) := (result.others(left) ? 0) + count;
    end loop;
    return result;
end;
```

A.2.5 The npow Operator

The npow operator is interesting since its operands are ordinarily of different types, and it is commutative. We must have two forms therefore, one for each operand order. We will use a common procedure to do most of the work, to avoid code duplication.

```
procedure self npow right;

    if not is_integer(right) then
        abort("Invalid operands for NPOW\nLeft => "+str(self)+"\nRight => "+str(right));
    end if;

    return npower(right);

end;

procedure left npow self;

    if not is_integer(left) then
        abort("Invalid operands for NPOW\nLeft => "+str(left)+"\nRight => "+str(self));
    end if;

    return npower(left);

end;

procedure npower(i);

    power_array := [[0,x] : x in self];
    powerset := {};

    loop

        if +/{c : [c,-] in power_array} = i then
            powerset with := bag([e : [s,e] in power_array | s = 1]);
        end if;

        if not (exists n in [1 .. #power_array] | power_array(n)(1) = 0) then
            exit;
        end if;

        for i in [1 .. n - 1] loop
            power_array(i)(1) := 0;
        end loop;

        power_array(n)(1) := 1;

    end loop;

    return powerset;

end npower;
```

A.2.6 The from Operator

In most of the methods on bags we do not modify the current instance. The from method is an exception. Since from generally modifies its source set we also want to modify the source bag.

```
procedure from self;

    if cardinality = 0 then
        return om;
```

```
        end if;

        cardinality -:= 1;

        if #pairs > 0 then
            [left, [right, count]] from pairs;
            if count > 1 then
                pairs{left}(right) := count - 1;
            end if;
            return [left,right];

        elseif #others > 0 then
            [operand, count] from others;
            if count > 1 then
                others(operand) := count - 1;
            end if;
            return operand;
        end if;

    end;
```

A.2.7 The < Operator

The < operator performs a subset test. It's a crucial method, since it will be called for any of <, <=, >, or >=.

First we perform a quick cardinality test. If that fails we just return false. If it succeeds we have to perform a more expensive test, checking each element in the current instance.

```
    procedure self < right_bag;

        if type(right_bag) /= "BAG" then
            abort("Invalid operands for <:\nLeft => "+str(self)+"\nRight => "+str(right_bag));
        end if;

        if cardinality >= right_bag.cardinality then
            return false;
        end if;

        for right_map = pairs{left}, count = right_map(right) loop
            if count > right_bag.pairs{left}(right) ? 0 then
                return false;
            end if;
        end loop;

        for count = others(left) loop
            if count > right_bag.others(left) ? 0 then
                return false;
            end if;
        end loop;

        return true;

    end;
```

A.2.8 Image Set Assignment

The map and image set assignment methods are particularly useful. In this example, we set the image set of one domain value in a bag. All we have to do is verify our operands, remove the old image set, and install a new one.

```
procedure self{left} := value;

   if type(value) /= "BAG" then
      abort("Invalid value for f{x} assignment\nValue => "+str(value));
   end if;

   if #others /= 0 then
      abort("May not assign image set to non-map BAG:\n"+str(self));
   end if;

   for count = pairs{left}(right) loop                -- remove old image set
      cardinality -:= count;
   end loop;
   pairs{left} := {};

   for right in value loop                            -- and install a new one
      pairs{left}(right) := (pairs{left}(right) ? 0) + 1;
      cardinality +:= 1;
   end loop;

end;
```

A.2.9 Iterators

We want to provide both iterator forms for bags, since we have operations similar to maps and multi-valued maps. Notice that we always return a tuple unless the bag is empty.

One strange thing to notice here: in the process of iterating over a bag we destroy it. This certainly isn't necessary, but it *is* safe. Remember that we have preserved **SETL2**'s value semantics. This means that any other references to the bag are not affected by the iteration.

```
procedure iterator_start;

   null;

end iterator_start;

procedure iterator_next;

   if cardinality = 0 then
      return om;
   end if;

   cardinality -:= 1;

   if #pairs > 0 then
      [left, [right, count]] from pairs;
      if count > 1 then
         pairs{left}(right) := count - 1;
      end if;
      return [[left,right]];
   else
```

```
            [operand, count] from others;
            if count > 1 then
                others(operand) := count - 1;
            end if;
            return [operand];
        end if;

    end iterator_next;

    procedure set_iterator_start;

        if #others /= ∅ then
            abort("Attempt to iterate over non-map BAG:\n"+str(self));
        end if;

    end set_iterator_start;

    procedure set_iterator_next;

        if #pairs = ∅ then
            return om;
        end if;

        result := bag([]);
        [left, [right, count]] from pairs;
        result.others := {[right,count]};
        result.cardinality := count;
        return [[left,result]];

    end set_iterator_next;
```

A.2.10 Print Strings

The default print string will be particularly ugly for bags, since we have split our data into two variables based on a transparent distinction, and because we keep quite a bit of information the user isn't aware of. We'll create print strings similar to sets, but we'll use the delimiters {> and <}, to distinguish bags from sets.

```
    procedure selfstr;

        first_element := true;
        for x in self loop
            if is_string(x) then
                x := "\""+x+"\"";
            end if;
            if first_element then
                first_element := false;
                result := "{> "+str(x);
            else
                result +:= ", "+str(x);
            end if;
        end loop;
        if first_element then
            return "{> <}";
        else
            return result+" <}";
        end if;
```

```
end selfstr;
```

That's all the methods we care to show here. The rest are fairly straightforward to code. They are included in an example file `bags.stl` distributed with the system.

References

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison Wesley Publishing Company, New York, NY, 1989.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall International Ltd., London, U.K., 1988.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, NY, 1986.

[Sny89] W. Kirk Snyder. The setl2 programming language. Technical Report 490, Courant Institute of Mathematical Sciences, New York University, 1989.