Programming in Idol: An Object Primer [1]

Clinton L. Jeffery

TR 90-10c

Abstract

Idol is an object-oriented extension and environment for the Icon programming language. This document describes Idol in two parts. The first part presents Idol's object-oriented programming concepts as an integral tool with which a programmer maps a good program design into a good implementation. As such, it serves as the "user's guide" for Idol's extensions to Icon. Idol's object-oriented programming facilities are viewed within the broader framework of structured programming and modular design in general. Idol's precise syntax and semantics are detailed in the second part, "An Icon-Derived Object Language", which serves as a reference manual.

January 25, 1990; Last revised August 4, 1992

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1] This work was supported in part by the National Science Foundation under Grant CCR-8713690.

Object-Oriented Programming After a Fashion

Object-oriented programming means different things to different people. In Idol, object-oriented programming centers around encapsulation, inheritance, and polymorphism. These key ideas are shared by most object-oriented languages as well as many languages that are not considered object-oriented. This part of the paper introduces these ideas and illustrates their use in actual code. Idol is relevant in this discussion because programming concepts are more than mental exercises; they are mathematical notations by which programmers share their knowledge.

Object-oriented programming can be done in Smalltalk, C++, or assembler language for that matter, but this does not mean these programming notations are equally desirable. Assembler languages are not portable. For most programmers, Smalltalk uses an alien notation; Smalltalk programs also share the flaw that they do not work well in environments such as UNIX and DOS that consist of interacting programs written in many languages. C++ has neither of these flaws, but the same low-level machine-oriented character that makes it efficient also makes C++ less than ideal as an algorithmic notation usable by nonexperts.

Idol owes most of its desirable traits to its foundation, the Icon programming language, developed at the University of Arizona [Gris90]. In fact, Idol presents objects simply as a tool to aid in the writing of Icon programs. Idol integrates a concise, robust notation for object-oriented programming into a language considerably more advanced than C or Pascal. Icon already uses a powerful notation for expressing a general class of algorithms. The purpose of Idol is to enhance that notation, not to get in the way.

Key Concepts

This section describes the general concepts that Idol supplies to authors of large Icon programs. The following section provides programming examples that employ these tools. The reader is encouraged to refer back to this section when clarification in the examples section is needed.

The single overriding reason for object-oriented programming is the large program. Simple programs can be easily written in any notation. Somewhere between the 1,000-line mark and the 10,000-line mark most programmers can no longer keep track of their entire program at once. By using a very high-level programming language, less lines of code are required; a programmer can write perhaps ten times as large a program and still be able to keep track of things. As programmers are required to write larger and larger programs, the benefit provided by very-high level languages does not keep up with program complexity. This obstacle has been labelled the “software crisis”, and object-oriented programming addresses this crisis. In short, the goals of object-oriented programming are to reduce the amount of coding required to write very large programs and to allow code to be understood independently of the context of the surrounding program. The techniques employed to achieve these goals are discussed below.

A second principal reason to consider object-oriented programming is that the paradigm maps very naturally onto certain problem domains, notably simulation. The first well-known object-oriented language, Simula67, certainly had this domain in mind. Experience with object-oriented techniques has led many practitioners to conclude that the concepts presented below are very general and widely applicable, but not all problems fit the object-oriented mold. Idol advocates use of objects as a guideline, not a rule.

Encapsulation

The primary concept advocated by object-oriented programming is the principle of encapsulation. Encapsulation is the isolation, in the source code that a programmer writes, of a data representation and the code that manipulates the data representation. In some sense, encapsulation is an assertion that no other routines in the program have "side-effects" with respect to the data structure in question. It is easier to reason about encapsulated data because all of the source code that could affect that data is immediately present with its definition.

Encapsulation does for data structures what the procedure does for algorithms: it draws a line of demarcation in the program text, the outside of which is (or can be, or ought to be) irrelevant to the inside. We call an encapsulated data structure an object. Just as a set of named variables called parameters comprise the only interface between a procedure and the code that uses it, a set of named procedures called methods comprise the only interface between an object and the code that uses it.

This textual definition of encapsulation as a property of program source code accounts for the fact that good programmers can write encapsulated data structures in any language. The problem is not capability, but verification. In order to verify encapsulation some object-oriented languages, like C++, define an elaborate mechanism by which a programmer can govern the visibility of each data structure. Idol instead stresses simplicity, while providing a compiler option that attempts to ease verification by preventing violations of encapsulation entirely.

Inheritance

In large programs, the same or nearly the same data structures are used over and over again for a myriad of different purposes. Similarly, variations on the same algorithms are employed by structure after structure. In order to minimize redundancy, techniques are needed to support *code sharing* for both data structures and algorithms. Code is shared by related data structures by a programming concept called inheritance.

The basic premise of inheritance is simple: when writing code for a new data structure that is similar to a structure that is already written, one specifies the new structure by giving the differences between it and the old structure, instead of copying and then modifying the old structure's code. Obviously there are times when the inheritance mechanism is not useful: if the two data structures are more different than they are similar, or if they are simple enough that inheritance would only confuse things, for example.

Inheritance addresses a variety of common programming problems found at different conceptual levels. The most obvious software engineering problem it solves might be termed *enhancement*. During the development of a program, its data structures may require extension via new state variables or new operations or both; inheritance is especially useful when both the original structure and the extension are used by the application. Inheritance also supports simplification, or the reduction of a data structure's state variables or operations. Simplification is analogous to argument culling after the fashion of the lambda calculus; it captures a logical relation between structures rather than a common situation in software development. In general, inheritance may be used in source code to describe any sort of relational hyponymy, or special-casing; in Idol the collection of all inheritance relations defines a directed (not necessarily acyclic) graph.

Polymorphism

From the perspective of the writer of related data structures, inheritance provides a convenient method for code sharing, but what about the code that *uses* objects? Since objects are encapsulated, that code is not dependent upon the internals of the object at all, and it makes no difference to the client code whether the object in questions belongs to the original class or the inheriting class.

In fact, we can make a stronger statement. Due to encapsulation, two different executions of some code that uses objects to implement a particular algorithm may operate on different objects that are not related by inheritance *at all*. Such code may effectively be shared by any objects that happen to implement the operations that the code invokes. This facility is called polymorphism, and such algorithms are called generic. This feature is found in non-object oriented languages; in object-oriented languages it is a natural extension of encapsulation.

Object Programming

The concepts introduced above are used in many programming languages in one form or another. The following text presents these concepts in the context of actual Idol code. This serves a dual purpose: it should clarify the object model adopted by Idol as well as provide an initial impression of these concepts' utility in coding. In order to motivate the constructs provided by Idol, our example begins by contrasting conventional Icon code with Idol code that implements the same behavior. The semantics of the Idol code given here is defined by the Idol reference manual, included later in this document in the section entitled, "An Icon-Derived Object Language".

Before Objects

In order to place Idol objects in their proper context, the first example is taken from from regular Icon. Suppose I am writing some text-processing application such as a text editor. Such applications need to be able to process Icon structures holding the contents of various text files. I might begin with a simple structure like the following:

```
record buffer(filename, text, index)
```

where filename is a string, text is a list of strings corresponding to lines in the file, and index is a marker for the current line at which the buffer is being processed. Icon record declarations are global; in principle, if the above declaration needs to be changed, the entire program must be rechecked. A devotee of structured programming would no doubt write Icon procedures to read the buffer in from a file, write it out to a file, examine, insert and delete individual lines, etc. These procedures, along with the record declaration given above, can be kept in a separate source file (buffer.icn) and understood independently of the program(s) in which they are used. Here is one such procedure:

```
# read a buffer in from a file
procedure read_buffer(b)
    f := open(b.filename) | fail
    b.text := [ ]
    b.position := 1
    every put(b.text, !f)
    close(f)
    return b
end
```

There is nothing wrong with this example; in fact its similarity to the object-oriented example that follows demonstrates that a good, modular design is the primary effect encouraged by object-oriented programming. Using a separate source file to contain a record type and those procedures that operate on the type allows an Icon programmer to maintain a voluntary encapsulation of that type.

After Objects

Here is part of the same buffer abstraction coded in Idol. A complete version of the source code is presented in Appendix B. This example lays the groundwork for some more substantial techniques to follow. In reading the code, you will encounter some new syntax. In general, the dollar character $ is a signal that something "object-oriented" is happening. In the example below, when you see the call self$erase() you should think of a buffer calling its erase() operation. Inside a method, self refers to an implicit buffer being operated on in the same way we used the parameter b to denote a buffer record in the previous section. Another important piece of new syntax is the $. object field access operator. Objects are just special records, and $.filename is equivalent to self.filename. $. is just a shorthand notation.

```
class buffer(public filename, text, index)
    # read a buffer in from a file
    method read()
        f := open($.filename) | fail
        self$erase()
        every put($.text, !f)
        close(f)
        return
    end
    # ...additional buffer operations
end
```

This first example is not complex enough to illustrate the full object-oriented style, but its a start. Note that when the Idol translator sees the syntax *object* $ *methodname* it already knows that a method invocation is taking place. If there are no parentheses, they are inserted automatically, so the above example could have used self$erase in place of self$erase(). This *parenthesis insertion* is analogous to Icon's *semicolon insertion* feature. We will use parenthesis insertion in examples from now on; it shortens the code.

Pertaining to the general concepts introduced above, we can make the following initial observations:

Polymorphism. A separate name space for each class's methods makes for shorter names. The same method name can be used in each class that implements a given operation. This notation is more concise than is possible with standard Icon procedures. More importantly it allows an algorithm to operate correctly upon objects of any class that implements the operations required by that algorithm.

Constructors. A section of code is executed automatically when the constructor is called, allowing initialization of fields to values other than &null. Of course, this could be simulated in Icon by writing a procedure that had the same effect; the value of the constructor is that it is automatic; the programmer is freed from the responsibility of remembering to call this code everywhere objects are created in the client program(s). This tighter coupling of memory allocation and its corresponding initialization removes one more source of program errors, especially on multiprogrammer projects.

These two observations share a common theme: the net effect is that each piece of data is made responsible for its own behavior in the system. Although this first example dealt with simple line-oriented text files, the same methodology applies to more abstract entities such as the components of a compiler's grammar[1].

Idol's code sharing facilities are illustrated if we extend the above example. Suppose the application is more than just a text editor— it includes word-associative databases such as a dictionary, bibliography, spell-checker, thesaurus, etc. These various databases can be represented internally using Icon tables. The table entries for the databases vary, but the databases all use string keyword lookup. As external data, the databases can be stored in text files, one entry per line, with the keyword at the beginning. The format of the rest of the line varies from database to database.

Although all these types of data are different, the code used to read the data files can be shared, as well as the initial construction of the tables. In fact, since we are storing our data one entry per line in text files, we can use the code already written for buffers to do the file i/o itself.

```
class buftable : buffer()
    method read()
        self$buffer.read
        tmp := table()
        every line := !$.text do
            line ?  tmp[tab(many(&letters))] := line | fail
        $.text := tmp
        return
    end
    method index(s)
        return $.text[s]
    end
end
```

[1] This example is taken from the Idol translator itself, which provides another extended example of polymorphism and inheritance.

This concise example shows how little must be written to achieve data structures with vastly different behavioral characteristics, by building on code that is already written. The superclass read() operation is one important step of the subclass read() operation; this technique is common enough to have a name: it is called *method combination* in the literature. It allows one to view the subclass as a *transformation* of the superclass. The buftable class is given in its entirety, but our code sharing example is not complete: what about the data structures required to support the databases themselves? They are all variants of the buftable class, and a set of possible implementations is given below. Note that the formats presented are designed to illustrate code sharing; clearly, an actual application might make different choices.

Bibliographies

Bibliographies might consist of a keyword followed by an uninterpreted string of information. This imposes no additional structure on the data beyond that imposed by the buftable class. An example keyword would be Jeffery90.

```
class bibliography : buftable()
end
```

Spell-checkers

The database for a spell-checker is presumably just a list of words, one per line; the minimal structure required by the buftable class given above. Some classes exist to introduce new terminology rather than define a new data structure. In this case we introduce a lookup operation that can fail, for use in tests. In addition, since many spell-checking systems allow user definable dictionaries in addition to their central database, we allow spellChecker objects to chain together for the purpose of looking up words.

```
class spellChecker : buftable(parentSpellChecker)
    method spell(s)
        return \ ($.text[s]) | (\ ($.parentSpellChecker))$spell(s)
    end
end
```

Dictionaries

Dictionaries are slightly more involved. Each entry might consist of a part of speech, an etymology, and an arbitrary string of uninterpreted text comprising a definition for that entry, separated by semicolons. Since each such entry is itself a structure, a sensible decomposition of the dictionary structure consists of two classes: one that manages the table and external file i/o, and one that handles the manipulation of dictionary entries, including their decoding and encoding as strings.

```
class dictionaryentry(word, pos, etymology, definition)
    method decode(s) # decode a dictionary entry into its components
        s ? {
            $.word := tab(upto(';'))
            move(1)
            $.pos := tab(upto(';'))
            move(1)
            $.etymology := tab(upto(';'))
            move(1)
            $.definition := tab(0)
            }
    end
    method encode() # encode a dictionary entry into a string
        return $.word || ";" || $.pos || ";" ||
            $.etymology || ";" || $.definition
    end
    initially
        if /$.pos then {
            # constructor was called with a single string argument
            self$decode($.word)
            }
    end

    class dictionary : buftable()
    method read()
        self$buffer.read
        tmp := table()
        every line := !$.text do
            line ? {
                tmp[tab(many(&letters))] := dictionaryentry(line) | fail
                }
        $.text := tmp
    end
    method write()
        f := open(b.filename, "w") | fail
        every write(f, (!$.text)$encode)
        close(f)
    end
end
```

Thesauri

Although an oversimplification, one might conceive of a thesauri as a list of entries, each of which consists of a comma-separated list of synonyms followed by a comma-separated list of antonyms, with a semicolon separating the two lists. Since the code for such a structure is nearly identical to that given for dictionaries above, we omit it here (but one might reasonably capture a generalization regarding entries organized as fields separated by semicolons).

Objects and Icon Programming Techniques

In examining any addition to a language as large as Icon, a significant question is how that addition relates to the rest of the language. In particular, how does object-oriented programming fit into the suite of advanced techniques used regularly by Icon programmers? Previous sections of this document expound objects as an organizational tool, analogous but more effective than the use of separate compilation to achieve program modularity. Object-oriented programming goes considerably beyond that viewpoint.

Whether viewed dynamically or statically, the primary effect achieved by object-oriented programming is the subdivision of program data in parallel with the code. Icon already provides a variety of tools that achieve related effects:

Local and Static Variables in Icon procedures are the simplest imaginable parallel association of data and code. We do not discuss them further, although they are by no means insignificant.

Records allow a simple form of user-defined types. They provide a useful abstraction, but keeping records associated with the right pieces of code is still the job of the programmer.

String Scanning creates scanning environments. These are very useful, but not very general: not all problems can be cast as string operations.

Co-expressions save a program state for later evaluation. This powerful facility has a sweeping range of uses, but unfortunately it is a relatively expensive mechanism that is frequently misused to achieve a simple effect.

Objects and classes, if they are successful, allow a significant generalization of the *techniques* developed around the above language mechanisms. Objects do not replace these language mechanisms, but in many cases presented below they provide an attractive alternative means of achieving similar effects.

Objects and Records

Objects are simply records whose field accesses are voluntarily limited to a certain set of procedures.

Objects and Scanning Environments

String scanning in Icon is another example of associating a piece of data with the code that operates on it. In an Icon scanning expression of the form `e1 ? e2`, the result of evaluating `e1` is used implicitly in `e2` via

a variety of scanning functions. In effect, the scanning operation defines a scope in which state variables &subject and &pos are redefined. [Walk86] proposes an extension to Icon allowing programmer-defined scanning environments. The extension involves a new record data type augmented by sections of code to be executed upon entry, resumption, and exit of the scanning environment. The Icon scanning operator was modified to take advantage of the new facility when its first argument was of the new environment data type.

While objects cannot emulate Icon string scanning syntactically, they generalize the concept of the programmer-defined scanning environment. Classes in the Idol standard library include a wide variety of scanning environments in addition to conventional strings. The variation is not limited to the type of data scanned; it also includes the form and function of the scanning operations. The form of scanning operations available are defined by the state variables they access; in the case of Icon's built-in string scanning, a single string and a single integer index into that string.

There is no reason that a scanning environment cannot maintain a more complex state, such as an input string, an output string, and a pair of indices and directions for each string. Rather than illustrate the use of objects to construct scanning environments with such an abstract model, a concrete example is presented below.

List Scanning

List scanning is a straightforward adaptation of string scanning to the list data type. It consists of a library class named ListScan that implements the basic scanning operations, and various user classes that include the scanning expressions. This format is required due to Idol's inability to redefine the semantics of the ? operator or to emulate its syntax in any reasonable way. The state maintained during a list scan consists of Subject and Pos, analogous to &subject and &pos, respectively.

ListScan defines analogies to the basic scanning functions of Icon, e.g. tab, upto, many, any, etc. These functions are used in methods of a ListScan client class that in turn defines itself as a subclass of ListScan. A client such as:

```
class PreNum : ListScan()
    method scan()
        mypos := $.Pos
        suspend self$tab(self$upto(numeric))
        $.Pos := mypos
    end
end
```

may be used in an expression such as

```
PreNum(["Tucson", "Pima", 15.0, [ ], "3"])$scan
```

producing the result ["Tucson", "Pima"]. The conventional Icon string scanning analogy would be: "abc123" ? tab(upto(&digits)), producing the result "abc". Note that ListScan methods frequently take list-element predicates as arguments where their string scanning counterparts take csets. In the above

example, the predicate numeric supplied to upto is an Icon function, but predicates may also be arbitrary user-defined procedures.

The part of the Idol library ListScan class required to understand the previous example is presented below. This code is representative of user-defined scanning classes allowing pattern matching over arbitrary data structures in Idol. Although user-defined scanning is more general than Icon's built-in scanning facilities, the scanning methods given below are *always* activated in the context of a specific environment. Icon string scanning functions can be supplied an explicit environment using additional arguments to the function.

```
class ListScan(Subject, Pos)
    method tab(i)
        if i < 0 then i := *$.Subject+1-i
        if i < 0 | i > *$.Subject+1 then fail
        origPos := $.Pos
        $.Pos := i
        suspend $.Subject[origPos:i]
        $.Pos := origPos
    end
    method upto(predicate)
        origPos := $.Pos
        every i := $.Pos to *($.Subject) do {
            if predicate($.Subject[i]) then suspend i
            }
        $.Pos := origPos
    end
initially
    /($.Subject) := [ ]
    /($.Pos) := 1
end
```

Objects and Co-expressions

Objects cannot come close to providing the power of co-expressions, but they do provide a more efficient means of achieving well-known computations such as parallel expression evaluation that have been promoted as uses for co-expressions. In particular, a co-expression is able to capture implicitly the state of a generator for later evaluation; the programmer is saved the trouble of explicitly coding what can be internally and automatically performed by Icon's expression mechanism. While objects cannot capture a generator state implicitly, the use of library objects mitigates the cost of explicitly encoding the computation to be performed, as an alternative to the use of co-expressions. The use of objects also is a significant alternative for implementations of Icon in which co-expressions are not available or memory is limited.

Parallel Evaluation

In [Gris87], co-expressions are used to obtain the results from several generators in parallel:

```
decimal := create(0 to 255)
hex := create(!"0123456789ABCDEF" || !"0123456789ABCDEF")
octal := create((0 to 3) || (0 to 7) || (0 to 7))
character := create(image(!&cset))
while write(right(@decimal, 3), " ", @hex, " ", @octal, " ", @character)
```

For the Idol programmer, one alternative to using co-expressions would be to link in the following code from the Idol standard library:

```
procedure sequence(bounds[ ])
    return Sequence(bounds)
end

class Sequence(bounds, indices)
    method max(i)
        elem := $.bounds[i]
        return (type(elem)== "integer", elem) | *elem-1
    end
    method elem(i)
        elem := $.bounds[i]
        return (type(elem)== "integer", $.indices[i]) |
                    elem[$.indices[i]+1]
    end
    method activate()
        top := *($.indices)
        if $.indices[1] > self$max(1) then fail
        s := ""
        every i := 1 to top do {
            s ||:= self$elem(i)
            }
        repeat {
            $.indices[top] +:= 1
            if top=1 | ($.indices[top] <= self$max(top)) then break
            $.indices[top] := 0
            top -:= 1
            }
        return s
    end
```

```
initially
    / ($.indices) := list(*$.bounds, 0)
end
```

On the one hand, the above library code is neither terse nor general compared with co-expressions. This class does, however, allow the parallel evaluation problem described previously to be coded as:

```
dec := sequence(255)
hex := sequence("0123456789ABCDEF", "0123456789ABCDEF")
octal := sequence(3, 7, 7)
character := sequence(string(&cset))
while write(right($@dec, 3), " ", $@hex, " ", $@octal, " ", image($@character))
```

$@ is the unary Idol meta-operator that invokes the activate() operation. Since the sequence class is already written and available, its use is an attractive alternative to co-expressions in many settings. For example, a general class of label generators (another use of co-expressions cited in [Gris87]) is defined by the following library class:

```
class labelgen : Sequence(prefix,postfix)
    method activate()
        return $.prefix||self$Sequence.activate||$.postfix
    end
initially
    /($.prefix) := ""
    /($.postfix) := ""
    /($.bounds) := [50000]
    self$Sequence.initially()
end
```

After creation of a label generator object (e.g. label := labelgen("L",":")), each resulting label is obtained via $@label. The sequence defined by this example is

```
L0:
L1:
...
L50000:
```

Conclusion

Idol presents object programming as a collection of tools to reduce the complexity of large Icon programs. These tools are encapsulation, inheritance, and polymorphism. Since a primary goal of Idol is to promote code sharing and reuse, a variety of specific programming problems have elegant solutions available in the Idol class library.

An Icon-Derived Object Language

This section serves as the language reference manual for Idol. Idol is a preprocessor for Icon that implements a means of associating a piece of data with the procedures that manipulate it. The primary benefits to the programmer are thus organizational. The Icon programmer may view Idol as providing an augmented record type in which field accesses are made not directly on the records' fields, but rather through a set of procedures associated with the type.

Classes

Since Idol implements ideas found commonly in object-oriented programming languages, its terminology is taken from that domain. The augmented record type is called a "class". The syntax of a class is:

```
class foo(field1, field2, field3, ...)
    # procedures to access
    # class foo objects

# code to initialize class foo objects
end
```

In order to emphasize the difference between ordinary Icon procedures and the procedures that manipulate class objects, these procedures are called "methods" (the term is again borrowed from the object-oriented community). Nevertheless, the syntax of a method is that of a procedure:

```
method bar(param1, param2, param3, ...)

    # Icon code that may access
    # fields of a class foo object
end
```

Since execution of a class method is always associated with a given object of that class, the method has access to an implicit variable called self that is a record containing fields whose names are those given in the class declaration. References to the self variable look just like normal record references; they use the dot (.) operator. In addition to methods, classes may also contain regular Icon procedure, global, and record declarations; such declarations have the standard semantics and exist in the global Icon name space.

Objects

Like records, instances of a class type are created with a constructor function whose name is that of the class. Instances of a class are called objects, and their fields may be initialized explicitly in the constructor in exactly the same way as for records. For example, after defining a class foo(x, y) one may write:

```
procedure main()

    f := foo(1, 2)
end
```

The fields of an object need not be initialized by the class constructor. For many objects it is more logical to initialize their fields to some standard value. In this case, the class declaration may include an "initially" section after its methods are defined and before its end. Initially sections are just special parameterless methods that are invoked automatically by the system.

This section begins with a line containing the word "initially" and then contains lines that are executed whenever an object of that class is constructed. These lines may reference and assign to the class fields as if they were normal record fields for the object being constructed. The "record" being constructed is named self; more on self later.

For example, suppose one wished to implement an enhanced table type that permitted sequential access to elements in the order they were inserted into the table. This can be implemented by a combination of a list and a table, both of which would initialized to the appropriate empty structure:

```
class taque(L, T) # pronounced 'taco'

    # methods to manipulate taques,
    # e.g. insert, index, foreach...

initially
    $.L := [ ]
    $.T := table()
end
```

And in such a case one can create objects without including arguments to the class constructor:

```
procedure main()

    mytaque := taque()
end
```

In the absence of an initially section, missing arguments to a constructor default to the null value. Together with an initially section, the class declaration looks rather like a procedure that constructs objects of that class. Note that one may write classes with some fields that are initialized explicitly by the constructor and other fields are initialized automatically in the initially section. In this case one must either declare the automatically initialized fields after those that are initialized in the constructor, or insert &null in the positions of the automatically initialized fields in the constructor.

Object Invocation

Once one has created an object with a class constructor, one manipulates the object by invoking methods defined by its class. Since objects are both procedures and data, object invocation is similar to both a procedure call and a record access. The dollar ($) operator invokes one of an object's methods. The syntax is *object* $ *method name* (*arguments*) where the parenthesis may be omitted if the argument list is empty.

If an object's class is known, object methods can also be called from Icon without difficulty using a normal procedure call. *object* $ *method name* (*arguments*) is equivalent to *class name_method name* (*object, arguments*) for objects of class *class name* or one of its subclasses.

Although object methods can be called using Icon procedure calls, the $ operator has certain distinct advantages. It handles inheritance automatically; in order to call an inherited method from Icon one must determine from which class the method is inherited. $ also handles invocation regardless of which class the object belongs to, allowing algorithms to be coded generically using polymorphic operations. Generic algorithms handle objects of any class that conforms to the set of methods used in the algorithm. Generic code is less likely to have to change if the program is later enhanced, such as by adding new subclasses that inherit from existing ones. In addition, if class names are long, the $ syntax is considerably shorter than writing out the class name for the invocation. Lastly, $ is looks like and is used similarly to the dot (.) operator used to access record fields. Using the taque example:

```
procedure main()
    mytaque := taque()
    mytaque$insert("greetings", "hello")
    mytaque$insert(123)
    every write(mytaque$foreach)
    if \ (mytaque$index("hello")) then write(", world")
end
```

Although objects are much like records, direct access to an object's fields using the usual dot (.) operator is not good practice outside of a method of the appropriate class, since it violates the principle of encapsulation. Although it is allowed by default, Idol includes a command-line option, -strict that checks for violations of this nature. In code generated with the -strict option, attempts to reference mystack.L in procedure main() result in a runtime error (invalid field name).

Within a class method on the other hand, record access to object fields is the norm. The implicit variable self allows access to the object's fields in the usual manner. The taque insert method is thus:

```
    method insert(x, key)
        /key := x
        put($.L, x)
        $.t[key] := x
    end
```

The self variable is both a record and an object. It allows field access just like a record, as well as method invocation like any other object. Thus class methods can use self to invoke other class methods without any special syntax.

Inheritance

In many cases, several classes of objects are very similar. In particular, many classes can be thought of simply as enhancements of some class that has already been defined. Enhancements might take the form of added fields, added methods, or both. In other cases a class is just a special case of another class. For example, if one had defined a class fraction(numerator, denominator), one might want to define a class inverses(denominator) whose behavior is identical to that of a fraction, but whose numerator is always 1.

Idol supports both of these ideas with the concept of inheritance. When the definition of a class is best expressed in terms of the definition of another class or classes, we call that class a subclass of the other classes. This corresponds to the logical relation of hyponymy. It means an object of the subclass can be manipulated just as if it were an object of one of its defining classes. In practical terms it means that similar objects can share the code that manipulates their fields. The syntax of a subclass is

```
class foo : superclass (fields...)
    # methods
# optional initially section
end
```

where *superclass* is the name of the class that foo inherits from. A subclass declaration is identical to a regular class, with the addition of one or more superclass names, separated by colons. The meaning of this declaration is the subject of the next section.

Inheritance semantics

There are times when a new class might best be described as a combination of two or more classes. Idol classes may have more than one superclass, separated by colons in the class declaration. This is called multiple inheritance. *Warning! Care should be taken employing multiple inheritance if the two parent classes have any fields or methods of the same name!*

Subclasses define a record type consisting of all the field names of the class itself and all its superclasses. The subclass has associated methods consisting of those in its own body, those in the first superclass that were not defined in the subclass, those in the second superclass not defined in the subclass or the first superclass, and so on. In ordinary single-inheritance, this addition of fields and methods follows a simple linear examination of each superclass, followed in turn by its parent superclass.

When a class has two or more superclasses, the search generalizes from a linear sequence to an arbitrary tree, dag, or graph traversal. In Idol, multiple inheritance adds fields and methods in an order defined by a depth-first traversal of the parent edges of the superclass graph. This is discussed in some more detail later on. For now, think of the second and following superclasses in the multiple inheritance case as only adding methods and fields if the single-inheritance case (following the first superclass and all its parents) has not already added a field or method of the same name.

Fields are initialized either by parameters to the constructor or by the class initially section. Initially sections are methods and are inherited in the normal way; the initially section that is used by a subclass is from the first class of the *class : superclass* list in which a method named initially is defined.

For example, to define a class of inverses in terms of a class fraction(numerator, denominator) one would write:

```
class inverse : fraction (denominator)
initially
    $.numerator := 1
end
```

Objects of class inverse can be manipulated using all the methods defined in class fraction; the code is actually shared by both classes at runtime.

Viewing inheritance as the addition of field names and methods of superclasses not already defined in the subclass is the opposite of the more traditional object-oriented view that a subclass starts with an instance of the superclass and augments or overrides portions of the definition with code in the subclass body. Idol's viewpoint adds quite a bit of leverage, such as the ability to define classes that are subclasses of each other. This feature is described further below.

Invoking Superclass Operations

When a subclass defines a method of the same name as a method defined in the superclass, invocations on subclass objects always result in the subclass' version of the method. This can be overridden by explicitly including the superclass name in the invocation:

```
object$superclass.method(parameters)
```

This facility allows the subclass method to do any additional work required for added fields before or after calling an appropriate superclass method to achieve inherited behavior. The result is frequently a *chain* of inherited method invocations.

Since initially sections are simply methods, they can invoke superclass operations including superclass initially sections. This allows a chain of initially sections to be specified to execute in either subclass-first or superclass-first order, or some mixture of the two.

Public Fields

As noted above, there is a strong correspondence between records and classes. Both define new types that extend Icon's built-in repertoire. For simple tasks, records are slightly faster as well as more convenient: the user can directly access a record's fields by name.

Classes, on the other hand, promote the re-use of code and reduce the complexity required to understand or maintain large, involved structures. They should be used especially when manipulating composite structures containing mixes of structures as elements, e.g. lists containing tables, sets, and lists in various positions.

Sometimes it is useful to access fields in an object directly, as with records. An example from the Idol translator itself is the *name* field associated with methods and classes—it is a string that is intended to be accessed outside the object. A method can always be implemented that returns (or assigns, for that matter) a field value, but this becomes tedious. Idol currently supports *read-only* access to fields via the public keyword. If public precedes a field name in a class declaration, Idol automatically generates a method of the same name that dereferences and returns the field. For example, the declaration

```
class sinner(pharisee, public publican)
```

generates code equivalent to the following class method in addition to any explicitly defined methods:

```
method publican()
    return $.publican
end
```

This feature, despite its utility, makes it possible to subvert object encapsulation: It returns a variable that can be assigned to. Idol's -strict command line option dereferences the field before returning (e.g. .$.publican), and generates runtime checks for structure types before returning the field, in order to prevent violations of encapsulation, since if the field has a structure value, Icon's pointer semantics allow elements of the structure to be modified from outside the class definition.

Superclass Cycles and Type Equivalence

In many situations, there are several ways to represent the same abstract type. Two-dimensional points might be represented by Cartesian coordinates x and y, or equivalently by radial coordinates expressed as degree d and radian r. If one were implementing classes corresponding to these types there is no reason why one of them should be considered a subclass of the other. The types are truly interchangeable and equivalent.

In Idol, expressing this equivalence is simple and direct. In defining classes Cartesian and Radian we may declare them to be superclasses of *each other*:

```
class Cartesian : Radian (x, y)
    # code that manipulates objects using cartesian coordinates
end

class Radian : Cartesian (d, r)
    # code that manipulates objects using radian coordinates
end
```

These superclass declarations make the two types equivalent names for the same type of object; after inheritance, instances of both classes will have fields x, y, d, and r, and support the same set of operations.

Equivalent types each have their own constructor given by their class name; although they export the same set of operations, the actual procedures invoked by the different instances may be different. For example, if both classes define an implementation of a method print, the method invoked by a given instance depends on which constructor was used when the object was created.

If a class inherits *any* methods from one of its equivalent classes, it is responsible for initializing the state of *all* the fields used by those methods in its own constructor, as well as maintaining the state of the inherited fields when its methods make state changes to its own fields. In the geometric example given above, in order for class Radian to use any methods inherited from class Cartesian, it must at least initialize x and y explicity in its constructor from calculations on its d and r parameters. In general, this added responsibility is minimized in those classes that treat an object's state as a value rather than a structure.

The utility of equivalent types expressed by superclass cycles remains to be seen. At the least, they provide a convenient way to write several alternative constructors for the same class of objects.

Miscellany

Unary Meta-operators

Idol supports shorthand notations for convenient object invocation. In particular, if a class defines methods named size, foreach, random, or activate, these methods can be invoked by a modified version of the usual Icon operator:

$*x is equivalent to x$size()
$?x is equivalent to x$random()
$!x is equivalent to x$foreach()
$@x is equivalent to x$activate()

Other operators may be added to this list. If x is an identifier, it may be used directly. If x is a more complex expression such as a function call, it must be parenthesized, e.g. $*(complex_expression()). This requirements are artifacts of the first implementation of Idol and are subject to change.

Another unary meta-operator is used only inside methods as a shorthand means of referring to an object's fields:

$.fieldname is equivalent to self.fieldname

The notation used to refer to an object's fields within a method body is again an artifact of the first implementation and is subject to change.

Nonunary Meta-operators

In addition to the unary meta-operators described above, Idol supports certain operators with more exotic capabilities. The expression x $$ y(*arguments*) denotes a list invocation of method y for object x and is analogous to Icon's list invocation operator (binary !). *Arguments* is some list that will be applied to the method as its actual parameter list. List invocation is particularly useful in handling methods that take a variable number of arguments and allows such methods to call each other. Idol list invocation is a direct application of Icon list invocation to object methods that could not be done otherwise without knowledge of Idol internals.

Another binary meta-operator is the object index operator given by $[, as in the expression x $[e]. This expression is an equivalent shorthand for x$index(e). Note that only the left brace is preceded by a dollar

sign. The expression in the braces is in actuality simply a comma separated list of arguments to the index method.

Constants

As a convenience to the programmer, Idol supports constant declarations for the builtin Icon types that are applicative— strings, integers, reals, and csets. Constant declarations are similar to global variable declarations with a predefined value:

```
const E_Tick := ".", E_Line := "_", E_Mask := '._'
```

Constant declarations are defined from their point of declaration to the end of the source file if they are defined globally, or to the end of the class definition if they are located within a class. Constants may not be declared within a procedure. Constants are equivalent to the textual replacement of the name by the value.

Include Files

Idol supports an #include directive as a convenience to the programmer. The include directive consists of a line beginning with the string "#include" followed by a filename that is optionally enclosed in quotation marks. When the include directive is encountered, Idol reads the contents of the named file as if it were part of the current file. Include files may be nested, but not recursive.

Since Idol and Icon do not have a compile-time type system, their need for sharing via file inclusion is significantly less than in conventional programming languages. Nevertheless, this is one of the more frequently requested features missing in Icon. Include files are primarily intended for the sharing of constants and global variable identifiers in separately translated modules.

Implementation Restrictions

The Idol translator is written in Idol and does not actually parse the language it purports to implement. In particular, the preprocessor is line-oriented and the initially keyword, and the class and method end keywords need to be on lines by themselves. Similarly, both the entire expression denoting the object being invoked and the method name must be on the same line for method invocations. If an object invocation includes an argument list, it must begin on the line of the invocation, since Idol inserts parentheses for invocations where they are omitted. This is comparable to Icon's semi-colon insertion; it is a convenience that may prove dangerous to the novice. Likewise, the $[index operator, its arguments, and its corresponding close brace must all be on the same line with the invoking object.

Class and method declarations are less restricted: the field/parameter list may be written over multiple lines if required, but the keyword is recognized only if it begins a line (only whitespace may precede it), and that line must include the class/method name, any superclasses, and the left parenthesis that opens the field/parameter list.

The Idol translator reserves certain names for internal use. In particular, _state and _methods are not legal class field names. Similarly, the name idol_object is reserved in the global name space, and may not be used as a global variable, procedure, or record name. Identifiers consisting of _*n*, where *n* is an integer are

reserved for Idol temporary variable names. Finally, for each class foo amongst the user's code, the names foo, foo_state, foo_methods, foo_oprec are reserved, as are the names foo_bar corresponding to each method bar in class foo. These details are artifacts of the current implementation and are subject to change.

Caveats

Subclass constructors can be confusing, especially when multiple inheritance brings in various fields from different superclasses. One significant problem for users of the subclass is that the parameters expected in the constructor may not be obvious if they are inherited from a superclass.

Problems with constructors can usually be solved by using two general techniques. One can guarantee constructor parameter order by naming fields explicitly in a subclass when initialization by constructor. Allowing initializations by parameter or by initially section can generally be done using the / operator in automatic field initializations unless the initialization should never be overridden.

While it is occasionally convenient to redeclare an inherited field in a subclass, accidentally doing so and then using that field to store an unrelated value would be disastrous. Although Idol offers no proper solution to this problem, the -strict option causes the generation of warning messages for each redefined field name noting the relevant sub- and superclasses.

Running Idol

Idol requires Version 8 of Icon. It runs best on UNIX systems. Idol has been ported to most but not all the various systems on which Icon runs. In particular, on versions of Icon that do not support the system() function, and on machines that do not have adequate memory available, Idol will not be able to invoke icont to complete its translation and linking. Since Idol is untested on several platforms, changes to the source code may be required in order to port it to a new system.

Since its initial inception, Idol has gone through several major revisions. This document describes Idol Version 9.0. Contact the author for current version information.

Getting a Copy

Idol is in the public domain. It is available on the Icon RBBS and by anonymous ftp from cs.arizona.edu. Idol is also distributed with the program library for Version 8 of Icon and is available by postal mail in this way. Interested parties may contact the author (cjeffery@cs.arizona.edu):

Clinton Jeffery
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Creating an Idol Executable

Idol is typically distributed in both Idol and Icon source forms. Creating an Idol executable requires a running version of Icon and a copy of idolboot.icn, the Icon source for Idol. A second Icon source file

contains the operating-system dependent portion of Idol; for example, unix.icn (see the Idol README file for the name of your system file if you are not on a UNIX system; you may have to write your own, but it is not difficult). Using icont, compile idolboot.icn and unix.icn into an executable file (named idolboot, or idolboot.icx). As a final step, rename this executable to idol (or idol.icx on some platforms).

Translating Idol Programs

The syntax for invoking idol is normally

```
idol file1[.iol] [files...]
```

(on some platforms this must read "iconx idol" where it says "idol" above). The Idol translator creates a separate Icon file for each class in the Idol source files you give it. On most platforms it calls icont automatically to create *ucode* (object code for the Icon virtual machine) for these files. If the first file on the command line has any Icon code in it (in addition to any class definitions it may contain), Idol attempts to link it to any classes it may need and create an executable.

The file extension defaults to .iol. Idol also accepts extensions .icn, .u1, and .cl (the latter is short for "class"). The first two refer to Icon source or already translated code for which Idol generates link statements in the main (initial) Idol source file. Idol treats arguments with the extension .cl as class names and generates link statements for that class and its superclasses. Class names are case-sensitive; Deque.cl is not the same class as deque.cl.

References

[Gris87] Griswold, R. E. Programming in Icon; Part II—Programming with Co-Expressions. Technical Report 87-6, Department of Computer Science, University of Arizona, June 1987.

[Gris90] Griswold, R. E. and Griswold, M. T. *The Icon Programming Language*, second edition. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Walk86] Walker, K. Dynamic Environments—A Generalization of Icon String Scanning. Technical Report 86-7, Department of Computer Science, University of Arizona, March 1986.

Appendix A: Idol Manual Page

NAME

idol - Icon-Derived Object Language

SYNOPSIS

idol [option ...] mainfile otherfiles... [-x arguments]

DESCRIPTION

Idol is an object-oriented preprocessor for Version 8+ Icon. It is a front-end for icont(1); typically one invokes idol on a source file (extension .iol) that is translated into an Icon source file (extension .icn) that is translated into a file suitable for interpretation by the Icon interpreter.

On systems with directories, Idol typically stores its generated class library code in a separate directory from the source code. If the environment variable IDOLENV is defined, Idol uses this directory for generated code. If no IDOLENV is defined, Idol creates a subdirectory named idolcode.env, and removes it after successful compilation if the creation occurred for a single source file.

Producing an executable is skipped when the first file on the list contains only classes and no Icon entities. Idol uses an Icon translator selected by the environment variable ICONT, if it is present.

The following options are recognized by Idol:

-c Suppress the linking phase
-t Suppress all translation by icont
-s Suppress removal of .icn files after translation by icont
-quiet Suppress most Idol-specific console messages
-strict Generate code that is paranoid about ensuring encapsulation
-version Print out the version of Idol and its date of creation

The second and following files on the command line may include extensions .icn, .u1, and .cl. The first two Idol treats as Icon source code that should be translated and linked into the resulting executable. Files with extension .cl are treated as

class names that are linked into the resulting executable.
If no extension is given, Idol attempts to find the desired
source file by appending .iol, .icn, .u1, or .cl in that order.

FILES

idol : the Idol translator itself
prog.iol : Idol source file
prog.icn : code generated for non-classes in prog.iol
idolcode.env/i_object.* : Icon code for the universal object type
idolcode.env/classname.icn : Icon files are generated for each class
idolcode.env/classname.u[12] : translated class files
idolcode.env/classname : class specification/interface

SEE ALSO

"Programming in Idol: An Object Primer"
(U of Arizona Dept of CS Technical Report #90-10)
serves as user's guide and reference manual for Idol

Appendix B: Source code for the buffer class

```
class buffer(public filename, text, index)
    # read a buffer in from a file
    method read()
        f := open($.filename) | fail
        self$erase()
        every put($.text, !f)
        close(f)
        return
    end
    # write a buffer out to a file
    method write()
        f := open($.filename, "w") | fail
        every write(f, !$.text)
        close(f)
    end

    # insert a line at the current index
    method insert(s)
        if $.index = 1 then {
            push($.text, s)
            }
        else if $.index > *$.text then {
            put($.text, s)
            }
        else {
            $.text := $.text[1:$.index] ||| [s] ||| $.text[$.index:0]
            }
        $.index +:= 1
        return
    end
    # delete a line at the current index
    method delete()
        if $.index > *$.text then fail
        rv := $.text[$.index]
        if $.index=1 then pull($.text)
        else if $.index = *$.text then pop($.text)
        else $.text := $.text[1:$.index]||$.text[$.index+1:0]
        return rv
    end
```

```
    # move the current index to an arbitrary line
    method goto(line)
        if (0 <= line) & (line <= $.index+1) then
            return $.index := line
    end
    # return the current line and advance the current index
    method forward()
        if $.index > *$.text then fail
        rv := $.text[$.index]
        $.index +:= 1
        return rv
    end
    method erase()
        $.text := [ ]
        $.index := 1
    end
    initially
        if  ($.filename) then {
            if not self$read() then self$erase()
            }
        else {
            $.filename := "*scratch*"
            self$erase()
            }
end
```