The Generator Paradigm in Smalltalk

Tim Budd[1]
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The method of *generators* is a powerful programming technique for many problems. Unfortunately, it is not widely known or employed. This paper describes the generator paradigm, illustrating the use of the technique by several programs in the language Smalltalk.

Introduction

In computer science, as in many other areas of human endeavor, people frequently attempt to solve problems by relating them to a paradigm, that is an example or model. Students, for example, when presented with a novel problem, will often try to formulate it in a manner similar to problems they have already been shown or solved for themselves. This is only natural, and is one reason why it is important for students in computer science to be exposed to a wide variety (not necessarily a wide number) of computer languages. Just as with natural languages, the computer language in which a problem is solved colors in a forceful way the type of solution employed. The wider the variety of languages a student is exposed to, the larger the number of paradigms they will have seen, and the broader will be their outlook on a new problem.

This paper describes one such programming paradigm, that of *generators*. The concept of generators occurs in many computer languages, but unfortunately is not a natural technique in the languages most commonly encountered by the student (Pascal, C, Fortran). Thus, while it is a valuable problem solving method for many classes of problems, it is not widely known or employed.

The generator paradigm is most useful in situations where there may not be a single, unambiguous, correct answer to a given question. This occurs frequently in such fields as pattern matching, data base systems, artificial intelligence as well as others. To illustrate, consider the pattern matching question

1 This paper was written while the author was on leave from the University of Arizona. The author's present address is: Department of Computer Science, Oregon State University, Corvallis, Oregon, 97331 USA.

"At what location does the letter 'i' occur in the word 'Mississippi'?" A single answer, such as '2', is correct but unsatisfactory. For example, this question might be a result of a larger problem, such as "at what location does the string 'ip' occur in the word 'Mississippi'?" Thus it is not sufficient to consider only the first answer to a question, but one must produce *all* answers to the question. One possibility is to return a *set* of answers. This works if the set of solutions is small, but becomes unwieldy if the set of answers is large or infinite. The generator paradigm is an attractive alternative.

For the purposes of this paper we will say a *generator* is any object (procedure, module, abstract datatype, class instance, or whatever) that can respond with a *sequence* of answers to a given query, one at a time. A generator for our example problem when asked for the position of 'i' in the word 'Mississippi' would first respond '2'. When asked again it would respond '5', then '8', then '11', and finally 'no more answers'.

Generators can be written in any language that supports co-routines, such as Simula [1], or SL5 [7]. They are used, although not always in ways that are obvious to the programmer, in Prolog [4]. They appear as a fundamental programming technique in the language Icon [6]. They can even be written, with the aid of a little run-time support, in C [2]. In this paper we will discuss the use of generators in Smalltalk [5], and in particular the examples will use the dialect of Smalltalk called Little Smalltalk [3].

SMALLTALK AND OBJECT ORIENTED PROGRAMMING

There is not sufficient space to present more than an elementary introduction to the Smalltalk language, however a few concepts are central to the discussion and must be advanced. A more complete description of the language can be found in [3,5].

The traditional model describing the behavior of a computer executing a program can be characterized as the process-state, or 'pigeon-hole' model. In this view the computer is a data manager, following some pattern of instructions, wandering through memory pulling values out of various slots (memory addresses), transforming them in some manner, and pushing the results back into other slots. By examining the values in the slots one can determine the state of the machine, or the results produced by the computation. While this may be a more or less accurate picture of what is physically taking place in a computer, it does little to help us understand how to solve problems using the computer and is certainly not the way most people (pigeons and postmen excepted) go about solving problems.

Let us examine a real world situation and then ask how we might make the solution of problems on a computer more closely model the methods people use in everyday life. Suppose I wish to send flowers to my grandmother for her birthday. She, however, lives in a city many miles away. It is a task easy enough to do; I merely go to a local florist, describe the nature and number of flowers I desire, and I can be assured that they will be automatically delivered. If I stopped to investigate how this gets accomplished I would probably discover that my florist sends a message describing my order to another florist in my grandmother's city, who then takes care of the actual delivery. I might inquire further to find out how the florist in my grandmother's city obtains the flowers, finding perhaps that they are obtained in bulk in the morning from a flower wholesaler. If I persist, I might even be able to follow the chain all the way back to the farmer who grows the flowers, and discover what requests were made by each member of the chain in order to solicit the desired outcome from the next.

The important point, however, is that I do not *need* to, indeed most of the time do not *want* to, know how my simple directive 'send flowers to my grandmother' is going to be carried out. In real life we call this process 'delegation of authority'. In computer science it is called 'abstraction' or 'information hiding'. At the heart, these terms amount to the same thing. There is a resource (a florist, a file server) that I wish to use. In order to communicate, I must know the commands the resource will respond to (send flowers to my grandmother, return a copy of the file named 'chapter1'). The steps the resource must take in order to respond to my request are in all likelihood much more complex than I realize, but in any case there is no reason for me to know the details of how my directive is implemented, as long as the response (the delivery of the flowers, receiving a copy of my file) is well defined and predictable.

The *object-oriented* model of problem solving views the computer in much the same fashion as just described. Indeed many people who have no training in computer science and no idea how a computer works find the object-oriented model of problem solving quite natural. Surprisingly, however, many people who have a traditional background in computer programming initially think there is something strange about the object-oriented view. The notion that '7' is an object, and '+' a request for an addition, may at first seem strange. But soon, the uniformity, power, and flexibility the object-message metaphor brings to problem solving makes this interpretation seem natural.

As we have been suggesting, the Smalltalk universe is inhabited by *objects*. If we invert the metaphor, using it to describe my flower example, I am an object and the flower shop (or the florist in it) is another object. Actions are instigated by sending requests (or *messages*) between objects. I transmitted the request 'send flowers to my grandmother' to the florist-object. The reaction of the *receiver* for the message is to execute some sequence of actions, or *method*, to satisfy my request. It may be the case that the receiver can immediately satisfy my request. On the other hand it will often be the case that in order to meet my needs, the receiver is required to transmit other messages to yet more objects (the message my florist sends to the florist in my grandmothers city, or a command to a disk drive). In addition, there is an explicit *response* (a receipt, for example, or a result code) returned directly back to me. DAN INGALLS describes the Smalltalk philosophy [8]:

> 'Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires.'

Such anthropomorphic viewpoints are common among Smalltalk programmers. In subsequent sections we will see how the Smalltalk language embodies this object-oriented view of programming. By describing the solution of several problems in Smalltalk, we hope to show how the object-oriented model aids in the creation of software systems, and assists in the solution of problems using the computer.

Smalltalk syntax

In this section we present a brief overview of Smalltalk syntax, just enough to make the examples presented later in the paper understandable. Once more, the reader interested in further information should consult the references.

An *object* can be a literal object, such as a number (2, for example), or a named object, such as an identifier (*x*, for example). An assignment arrow is used to associate a name with an object. The statement

```
x ← 2
```

makes the identifier *x* temporarily represent the same object as the literal object 2. This assignment may be later overwritten by other assignments to the same identifier.

Action is initiated by sending messages to objects. A message can simply be a command, with no arguments. For example, the following statement:

```
x squared
```

illustrates the message *squared* being sent to the object *x*. In response, the object *x* will return a new object. The particular nature of the response is always defined by the category (in Smalltalk terms, the *Class*) of the recipient for the message. If *x* is a number, the response to the message *squared* will be the object representing the value of the number multiplied by itself. Thus, the following example will make the identifier *y* represent the object 4.

```
y ← x squared
```

Messages can also take arguments. The arithmetic operations, for example, are interpreted as messages to the left side, having the right side as argument. Thus the expression:

```
x + 3
```

shows the message '+' being passed to the object *x*, accompanied by argument 3.

A third form of message is permitted to take an arbitrary number of arguments. This form of message is written as a sequence of *keywords*, that is names followed by colons, separating the receiver and arguments. For example:

```
x between : 2 and : 4
```

shows the message *between:and:* being passed to the object *x*, accompanied by two arguments. Messages can be composed, with messages having no arguments taking precedence over binary (arithmetic style) messages, and binary messages taking precedence over keyword messages. Parenthesis can be used to provide alternative groupings.

Some messages have side effects, in addition to returning a value. The message *print*, for example, will display a value on an output device. Thus:

> (*x squared* + 3) *print*

will cause the value 7 to be displayed.

A novel feature of Smalltalk is the ability to easily encapsulate actions for performance at a later time. This is accomplished using a *block*, which is written as a pair of square braces surrounding a list of Smalltalk statements. Because a block is an object, it can be assigned to an identifier or used as an argument in an expression, like other objects. For example:

> *z* ← [*x print. x* ← *x* + 1]

assigns a block to the identifier *z*. Note that a period is used to separate the statements within the block. These statements are not immediately executed; instead, they are executed when the message *value* is passed to the block. If at some later time the statement:

> *z value*

is executed, the value 2 (the current binding of the object *x*) will be displayed and *x* will be updated. The block continues to exist, and the actions can be repeated merely by sending the *value* message to the block again, as often as necessary. Note that the binding for the identifier *x* derives from the surrounding context of the definition of the block, and not from the context in which the message *value* is used.

Blocks are used to implement a number of control structures in Smalltalk. For example the conditional execution statement is constructed as a message passed to an object of type boolean using a block as an argument.

> (*x* < 3) *ifTrue:* [*x* ← *x* + 1]

If the boolean (the recipient of the *ifTrue:* message) is true, the block is executed; otherwise not. Similarly the while loop is constructed using a block for both the recipient and the argument.

> [*x* < 10] *whileTrue:* [*x print. x*← *x* + 1]

Blocks can also be written so as to take parameter values, and thus in many ways act like statically scoped in-line procedures. For example, the following block:

> *w* ← [*:a* | *a squared print*]

defines w to be a block taking one argument. The keyword message *value:* is used to invoke such a block. For example:

> *w value:* 6

will result in the value 36 being displayed.

The sequence of actions to be executed in response to a message is described by a *method*, which corresponds in some ways to a procedure in a conventional language. For example, the following method describes the message *squared* :

squared

$\uparrow$ *self* * *self*

Within a method, the identifier *self* refers to the receiver of the message. The up arrow ($\uparrow$) precedes the value to be returned in response to the message. A method for a message that takes arguments must provide identifier names for the arguments:

between: *lower* **and:** *upper*

$\uparrow$ (*self* >= *lower*) & (*self* <= *upper*)

One final bit of syntax is useful in situations that might otherwise require the introduction of temporary variables. A *cascade* is formed from an expression, and one or more continuations of messages (messages without a receiver) separated by semicolons. An example might be:

$(x+2)$; *squared*

The result of a cascade is the result of the expression to the left of the first semicolon. This value is also used as the receiver for messages to the right of the semicolon, and whatever response they produce is discarded. In almost all situations where a cascade is used the expression to the left is creating a new object, and the message on the right is initializing it in some fashion. The cascade is used for the side effect whatever message on the right side may have, not for the response it will return.

GENERATORS

A *generator* is any object that represents a collection of other objects and that responds to the following two messages:

first	The response should be an element of the collection, or the special value **nil** if there are no elements in the collection.
next	The response should be another element in the collection, or **nil** if there are no more elements in the collection.

We do not require that the collections be in any specific order, only that all elements will eventually be produced if a sufficient number of *next* messages are received and no element will be produced more than once.

A simple generator is one used to produce values in arithmetic sequence. The message *to:*, used in conjunction with numbers, produces such a generator.

For example:

```
          x ← 3 to: 9
          x first
     3
          x next
     4
```

The following method describes a useful message for dealing with generators, the message *do:*.

```
     do: aBlock          | item |

          item ← self first.
          [ item notNil ] whileTrue:
               [ aBlock value: item .  item ← self next ].
          ↑ nil
```

We will explain several features of this method. There is a temporary variable named *item* used in the method; this variable is declared by placing it in a list surrounded by vertical bars following the pattern describing the names of the method and of the arguments. This particular method appears as part of the description of all generators. The special identifier *self* refers to the recipient of the do: message. Since this must be able to respond to *first* and *next*, it must be a generator. The body of the method is a simple loop. Before entering the loop the temporary variable *item* is assigned the result of passing the message *first* to the recipient. While *item* is not **nil** the value is used as an argument in a message *value:*, passed to the variable *aBlock*. The variable *item* is then updated by requesting the next value from the sequence.

The argument used with this message must be a one argument block. The block is executed on each element of the collection. For example:

```
          (0 to: 5) do: [ :x | x squared print ]
     0
     1
     4
     9
     16
     25
```

Subsequent sections will illustrate the utility of the generator concept.

A Simple example

An example will illustrate how generators can be written in Smalltalk. Consider the problem of producing prime numbers. By definition, a prime number is a value having only two distinct divisors, itself and 1. A generator for prime numbers will produce the first prime value (namely 2) when offered the message *first*, and successive prime numbers in response to each *next* message. If a number n divides a number m, then the prime factors of n must also divide m. Thus to tell if a number m is prime we need not test all values less than m, only those values that are prime. Therefore a simple generator for primes can be constructed by merely retaining the previously generated primes in a **List**, a data structure that will maintain elements in their order of insertion. Each time a value is requested, an object representing the last prime produced is incremented and tested until a value having no factors is found. The new value is then appended to the list and returned. Keeping the primes in order allows the loop to terminate as soon as a value larger than $\sqrt{n}$ is encountered, where n is the value to be tested.

The methods for each type of object in the Smalltalk language are gathered together to form what is known as a *class*. For example all integers are elements of class **Integer**, all arrays of class **Array** and so on. The response of an object when presented with any given message is determined by the methods associated with the class of that object. The following is a class description for a prime number generator. Each instance of this class will, in response to the messages *first* and *next*, return a stream of prime numbers. The variables *prevPrimes* and *lastPrime*, listed following the class name, are local variables for the class. Each instance of the class maintains its own copies of these variables, and they can be used only in the methods for the class.

New instances of a class are created using the message *new*. In this example, a new instance of the class **List** is created and stored in the identifier *prevprimes*. A List is a simple data structure that maintains elements in the order that they are inserted (using the message *add:*). Like most data structures in Smalltalk, a List is also a generator, and thus responds to the message *do:*

```
Class Primes
| prevPrimes lastPrime |
[
        first
                prevPrimes ← List new.
                prevPrimes add: (lastPrime ← 2).
                ↑ lastPrime
|
        next
                [ lastPrime ← lastPrime + 1.
                self testNumber: lastPrime ] whileFalse.
                prevPrimes add: lastPrime.
                ↑ lastPrime
|
        testNumber: n
                prevPrimes do: [:x|
                        (x squared > n) ifTrue: [ ↑ true ].
                        (n \\ x = 0) ifTrue: [ ↑ false ] ]
]
```

The message *testNumber:* is used to determine whether a proposed number is prime. It accomplishes this by performing a modular division (\\) of the number with previous primes. If the remainder after division is zero, a previous prime divides the number and it cannot be prime. If no number less than the square root of the proposed number divides the number, then it must be prime.

An obvious problem with this prime number generator is that it requires an ever increasing amount of storage to maintain the list of previous primes. If one were constructing a long list of prime values, the size of this storage could easily become a problem. A recursive version is possible which trades longer computation time for reduced storage. This is analogous to a recursive procedure in programming languages such as Pascal. The following program does not maintain the list of previous primes, but instead regenerates the list each time a new number is to be tested. The expression '*Primes new*' produces a new instance of the prime generator each time the message *testNumber:* is received.

```
Class Primes
| lastPrime |
[
        first
                ↑ ( lastPrime ← 2 )
|
        next
                [ lastPrime ← lastPrime + 1.
                self testNumber: lastPrime ] whileFalse.
                ↑ lastPrime
|
        testNumber: n
                (Primes new) do: [:x |
                        (x squared > n) ifTrue: [ ↑ true ].
                        (n \\ x = 0) ifTrue: [ ↑ false ] ]
]
```

FILTERS

An entirely different program, solving the same task as the prime number generators described in the last section, will illustrate another programming technique that is frequently useful in conjunction with generators, which is the notion of *filters*. A filter is a generator that filters, or modifies, the values of another underlying generator. The class **FactorFilter** (below) exemplifies some of the essential features of a filter. Instances of **FactorFilter** are initialized by giving them a generator and a specific nonnegative value, using the message *remove:from:*. In response to *next* (the message *first* is in this case replaced by the initialization protocol *remove:from:*), values from the underlying generator are requested, and returned, with the exception of values for which the given number is a factor, which are repressed. Thus the sequence returned by an instance of **FactorFilter** is exactly the same as that given by the underlying generator, with the exception that values for which the given number is a factor are filtered out. (The symbol ~= is the Smalltalk message representing 'not equals').

```
Class FactorFilter
| myFactor generator |
[
        remove: factorValue from: generatorValue
                myFactor ← factorValue.
                generator ← generatorValue
|
        next            | possible |
                [ (possible ← generator next) notNil ]
                        whileTrue:
                                [ (possible \\ myFactor ~= 0)
                                        ifTrue: [ ↑ possible ] ].
                ↑ nil
]
```

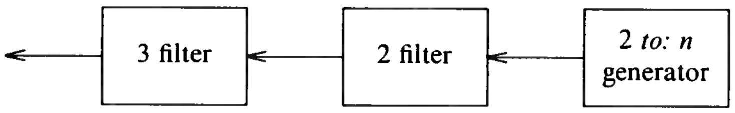
Using **FactorFilter**, a simple generator for prime numbers can be constructed. An instance of **Interval** (the generator that merely returns numbers in arithmetic progression) is first constructed generating all numbers from 2 to some fixed limit. As each value is removed, a filter is inserted in front of the generator to insure that all subsequent multiples of the value will be eliminated. A new value is then requested from the updated generator.

```
Class Primes
| primeGenerator lastFactor |
[
        first
                primeGenerator ← 2 to: 100 .
                lastFactor ← primeGenerator first .
                ↑ lastFactor
|
        next
                primeGenerator ← FactorFilter new;
                        remove: lastFactor from: primeGenerator
                lastFactor ← primeGenerator next .
                ↑ lastFactor
]
```

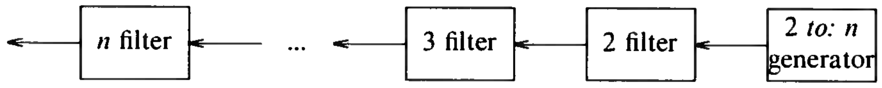
Pictorially, the underlying generator constructed by the first occurrence of the message *next* can be viewed as follows:

$$\begin{tikzcd} {} & \boxed{\text{2 filter}} \arrow[l] & \boxed{\text{2 \textit{to: n} generator}} \arrow[l] \end{tikzcd}$$

When asked for the next prime, the generator is modified by adding a filter, this time for the last prime value returned, the number 3.



The program continues, each time a new prime is requested a filter is constructed to remove all factors of the previous prime. In this fashion, all the primes are eventually generated.



Of course, like the first two programs in the last section, the storage required for the chain of filters is proportional to the number of primes generated so far. Despite this, actual timings on running programs show that the filter program is the fastest of the three prime number generating programs described here. However, we should note that these programs do not represent the fastest algorithms known for producing prime numbers, but are merely intended as instructive examples of classes and generators in Smalltalk.

Goal directed evaluation

A useful programming technique when used in conjunction with generators is *goal directed evaluation*. Using this technique, a generator is repeatedly queried for values until some condition is satisfied. In a certain sense the notion of filters we have just described represents a simple form of goal directed evaluation. The goal of instances of **FactorFilter**, for example, is to find a value from the underlying generator for which the given number is not a factor. In the more general case of goal directed evaluation the condition frequently involves the outcome of several generators acting together. An example will illustrate this method.

Consider the problem of placing eight queens on a chess board in such a way that no queen can attack any other queen (illustrated below). In this section we will describe how such a problem can be formulated and solved using generators, filters, and goal directed evaluation.

We first observe that in any solution, no two queens can occupy the same column, and that no column can be empty. We can therefore assign a specific column to each queen at the start, and reduce the problem to finding a correct row assignment for each of the eight queens.

In general terms, our approach will be to place queens from left to right (the order in which we assign numbers to columns). An *acceptable solution for columns 1 through n* is one in which no queen in columns 1 through *n* can attack any other queen in those columns. Once we have found an acceptable solution for columns 1 through 8 we are finished. Before that, however, we can formulate the problem of finding an acceptable solution for columns 1

	1	2	3	4	5	6	7	8
1	Q							
2					Q			
3								Q
4						Q		
5			Q					
6							Q	
7		Q						
8				Q				

A solution to the eight queens problem

through *n* recursively, as follows:

1. If $n > 1$, find an acceptable solution for columns 1 through $n-1$. If there is none, return nil, there is no acceptable solution. Otherwise place the queen for column *n* in row 1. Go to step 2.
2. Test to see if any queen in columns 1 through $n-1$ can attack the queen in column *n*. If not, then an acceptable solution has been found. If some other queen can attack, then go to step 3.
3. If the queen for column *n* is in row 8, then go to step 4, otherwise advance the queen by one row and go back to step 2.
4. Find the next acceptable solution for columns 1 through $n-1$. If there is none, return nil, otherwise place the queen for column *n* in row 1 and go to step 2.

Of course, all positions are acceptable in column 1. Responding to *first* corresponds to starting in step 1, whereas responding to *next* corresponds to starting in step 3. We represent each queen by a separate object, an instance of class **Queen**. Each queen maintains its own position in a pair of variables, and also a pointer to the immediate neighbor on the left. A skeleton for the class **Queen** can be given as follows:

```
Class Queen
| row column neighbor |
[
        setColumn: aNumber neighbor: aQueen
                column ← aNumber.
                neighbor ← aQueen
|
        ...
]
```

Using this skeleton, our eight queens can be initialized as follows:

```
lastQueen ← nil
(1 to: 8) do: [:i | lastQueen ← Queen new ;
                setColumn: i neighbor: lastQueen ]
```

Following the execution of this code the variable *lastQueen* points to the last (rightmost) queen.

We have already described our algorithm in terms of finding the first acceptable position and finding the next acceptable position. It is therefore easy to apply our generator paradigm (using the messages *first* and *next*) to this situation. Step 1, for example, corresponds to the following method

```
first
        (neighbor notNil)
                ifTrue: [ neighbor first ].
        row ← 1.
        ↑ self testPosition
```

Rather than falling directly into step 2, as we did in the informal description of the algorithm, an explicit message (*testPosition*) is used to perform steps 2, 3 and 4. Thus one can read *self testPosition* as being the equivalent of 'go to step 2' in the informal description. Before describing the method for this message, we describe the method used to find the *next* acceptable position, which is a combination of steps 3 and 4 in our description.

```
next
        (row = 8)
                ifTrue: [ ((neighbor isNil) or: [ neighbor next isNil ])
                                ifTrue: [ ↑ nil ].
                          row ← 0 ].
        row ← row + 1.
        ↑ self testPosition
```

A coding trick is used here; the zero assigned to the identifier row is immediately incremented, resulting in the queen being placed into row 1. Once more the '*self testPosition*' message can be interpreted as 'go to step 2'.

All that remains is to test a position to see if any queen to the left can attack. As we have already noted, any position is acceptable to the leftmost queen. Suppose a queen, call her Q, is not the leftmost queen. We pass a message to the neighbor of Q asking if she can attack the position of the queen Q. If the neighbor queen can attack, she will return true, otherwise she will pass the message on to her neighbor, and so on until the leftmost queen is reached. If the leftmost queen cannot attack, she will return false. Notice the recursive use of the message *next* to find the next acceptable position, in case an attack is possible. This corresponds to the directive 'go to step 3' found in step 2 of our informal description.

```
testPosition
        (neighbor isNil) ifTrue: [ ↑ self ].
        (neighbor checkRow: row column: column)
                ifTrue: [ ↑ self next ]
                ifFalse: [ ↑ self ]
```

We have reduced the problem to the much simplier one of each queen taking a pair of coordinates for a queen positioned to the right, and responding whether she or any queen to the left can attack that position. Since we know the queen to the right is in a different column from the queen to the left, she can only be attacked if she is in the same row or if the differences in the columns is equal to the differences in the rows (i.e., a diagonal).

```
checkRow: testRow column: testColumn    | columnDifference |
        columnDifference ← testColumn − column.
        (((row = testRow) or:
          [ row + columnDifference = testRow ] ) or:
          [ row − columnDifference = testRow ])
                ifTrue: [ ↑ true ].
        (neighbor notNil)
                ifTrue: [ ↑ neighbor checkRow: testRow
                                        column: testColumn ]
                ifFalse: [ ↑ false ]
```

A final method is useful for producing the answer in a visual form:

```
printBoard
        (neighbor notNil)
                ifTrue: [ neighbor printBoard ].
        ('column ', column, ' row ', row) print
```

Putting all the methods for class **Queen** together, we could type the following example script:

```
lastQueen ← nil.
(1 to: 8) do: [:i | lastQueen ← Queen new;
                        setColumn: i neighbor: lastQueen ]
```

```
        lastQueen first
        lastQueen printBoard
column 1 row 1
column 2 row 5
column 3 row 8
column 4 row 6
column 5 row 3
column 6 row 7
column 7 row 2
column 8 row 4

        lastQueen next
        lastQueen printBoard
column 1 row 1
column 2 row 6
column 3 row 8
column 4 row 3
column 5 row 7
column 6 row 4
column 7 row 2
column 8 row 5
```

SUMMARY

We have, unfortunately, been able to present only the briefest glimpse of two topics in the paper; namely generators and the language Smalltalk. Readers interested in the first topic would do well to read the contrasting presentation of generators using the language Icon [6]. Readers interested in further information on Smalltalk can find the definitive description in the book by GOLDBERG and ROBSON [5]. The material in this paper is condensed from a fuller exposition in Chapter 8 of [3].

REFERENCES

1. G.M. BIRTWISTLE, O.J. DHAL, B. MYHRHAUG, K. NYGAARD. (1973). *Simula Begin,* Studentlitteratur, Lund, Sweden.
2. T.A. BUDD. (1982). An implementation of generators in C. *J. Computer Lang. 7, 2,* 69-88.
3. T.A. BUDD. *A Little Smalltalk,* Addison-Wesley (to be published in 1986).
4. W.F. CLOCKSIN, C.S. MELLISH. (1981). *Programming in Prolog,* Springer-Verlag, New York.
5. A. GOLDBERG, D. ROBSON, (1984). *Smalltalk-80: The language and Its Implentation,* Addison-Wesley.
6. R.E. GRISWOLD, M.T. GRISWOLD, (1983). *The Icon Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, NJ.

7. D.R. HANSON, R.E. GRISWOLD (1978). The SL5 procedure mechanism. *Comm. ACM 21,* 392-400.
8. D.H. INGALLS (1981). Design principles behind smalltalk. *BYTE 6, 8,* 286-302.