An Implementation of Generators in C*

Timothy A. Budd

TR 81-5

ABSTRACT

This report describes a new language, *Cg*, that extends the programming language *C* to include *generators*, and facilitates backtracking and goal directed evaluation. The first section introduces the idea of generators and their uses in backtracking and goal directed evaluation. This is followed by examples showing how the solution of several classic programming problems can be simplified using these concepts. A more complicated example, a program for matching regular expressions, is then described. The paper concludes with a description of the current implementation of Cg.

August 21, 1981

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MSC79-03890.

An Implementation of Generators in C

In its most basic form, a *generator* is any expression that can be repeatedly activated to provide a succession of different values. This notion of generators is found in a number of languages, notably Alphard [9], CLU [8], Icon [3] and MLISP [1]. For example, in Alphard and CLU generators can be used to iterate over the elements of a programmer-defined data structure, but are only accessible in the context of a particular type of **for** statement. In contrast, more sophisticated uses of generators involve notions of backtracking and goal directed evaluation [7]. These features are found in the language *Icon* [3], a descendent of SNOBOL4 and SL5 [4]. In Icon any expression can be a generator or drive other generators.

This paper describes a new language, *Cg*, that extends the programming language *C* [6] to include generators. As in Alphard and CLU, but unlike Icon, the activation of generators in Cg is restricted to two specific language constructs, the **every** statement and the **drive** statement. However the domain of objects that generators can manipulate and produce is not restricted to any particular datatype.

This paper is divided into four sections. The first section introduces the idea of generators and their use in backtracking. The second section presents several examples of the use of generators in solving several classic programming problems. The third section examines in detail a solution using generators to a problem that is difficult to solve using conventional methods. Finally the paper concludes with a description of an implementation of Cg on the PDP-11/70.

1. Generators and Backtracking

The task of enumeration is a commonly occurring process in many computer programs. For example, the Algol **for** or FORTRAN **DO** statements can be regarded as enumerating a sequence of values in an arithmetic progression. Other data objects, such as nodes in a linked list, leaves in a binary tree, or characters from a sequential file, are also frequently enumerated. Syntactically, the **for** loop in the language C goes a long way towards capturing the enumerative nature of these other cases. However there are still situations, such as enumerating leaves in a binary tree, that are difficult to describe using a **for** statement.

What is basic to these operations is the notion of *generation*. That is, the user has a data object (binary tree, sequential file) or set (arithmetic progression) and wants to generate, one by one, elements from the data object or set. The generated objects need not be of any fixed type, nor does the set need to be finite; one can imagine, for example, generating the set of primes, or the set of Fibonocci numbers.

Consider, for example, binary trees composed of nodes containing a left and right pointer (set to zero in leaf nodes) and an integer value field. Suppose a user wants to perform two different operations on such trees: The first operation adds one to each value field in each node in the tree, the second operation computes the sum of the value fields.

```
addtree(node)
  struct tree *node;
{
  if (tree->left != 0)
    addtree(tree->left);
  tree->value += 1;
  if (tree->right != 0)
    addtree(tree->right);
}
```

Figure 1: A conventional tree traversal algorithm

There are two conventional methods for solving this problem. The first involves writing a separate function for each operation, for example the postorder tree traversal algorithm shown in Figure 1. The disadvantage with this solution is the redundant code that must be written to perform the tree traversal for each new operation. In order to avoid this, a second conventional solution would pass the name of a procedure which performs the operation as an argument to the tree traversal function, as shown in Figure 2. The disadvantage here is that each operation requires the creation of a new subfunction, and each subfunction so defined must have the same number and type of parameters.

```
ptree(node, process)
  struct tree *node;
  int process();
{
  if (node->left != 0)
    pnode(node->left, process);
  process(node);
  if (node->right != 0)
    pnode(node->right, process);
}
```

Figure 2: A tree traversal algorithm using function parameters

Neither of these solutions correspond closely to the user's abstract view of the operation. In the abstract, the first process can be described as "for every node in the tree, add one to the value field". Thus the generation of a new element is abstractly a subordinate task to the task at hand, which is just the opposite from the situation in the two conventional solutions given above.

A closer approximation to the abstract description is provided by co-routines [2]. Here the main program and the subprogram producing leaf nodes each run independently. The main program is a *consumer*, processing values (nodes) produced by the second program, which is a *producer*.

Although the co-routine model corresponds more closely to the abstract description of the problem, and indeed any generator can be thought of as a specialized type of co-routine, co-routines are both more powerful and more complex than the situation requires. Using co-routines, the user must design an initialization protocol in order to ensure the procedures synchronize correctly. Furthermore, in order to accommodate recursive procedure calls and/or multiple copies of procedures, co-routines are usually implemented using dynamically allocated, rather than stacked, activation records. It is not unusual for this to increase by a factor of 3 or more the run-time overhead of a procedure call [10].

By concentrating on a less general control structure than the co-routine, generators can hide from the user most of the details of initilization and synchronization. Furthermore it is possible to produce a very efficient run time implementation (Section 4).

Before describing generators in detail it is necessary to consider the several problems that generators are designed to avoid. The first is *initialization*, meaning the actions taken on the first function invocation (and perhaps the second, and so on), are different from the actions taken on subsequent invocations. For example a generator producing the list of Fibonocci numbers produces the value 1 the first two times it is invoked, and thereafter it returns the sum of the last two numbers generated. Hence there must be some mechanism by which the generator can distinguish which invocation is taking place.

The second problem is *memory*, since the generator must have some ability to remember information from preceding invocations, in order to produce new values.

A third problem is *termination*; finding a way to terminate the enumeration loop when all the elements have been produced.

All of these problems can be solved using conventional means, such as using global or static variables. However the user is required to explicitly design and construct code to handle these problems, rather than letting the language hide unnecessary implementation details. The advantage of generators is one of conceptual simplicity.

Instead of producing a value and then terminating, a generator procedure produces a value and then goes into a state of "suspended animation." If a subsequent value is requested the function is "reawakened" and processing continues in the procedure just as if the suspension had never taken place. That is, local variables and the values of parameters retain whatever values they had at the time of suspension, and execution continues from the point immediately following the statement that caused the suspension.

In order to differentiate it from the conventional process of terminating via a **return** statement, the statement

```
suspend(value);
```

indicates the process of producing a value and going into "suspended animation". Any procedure that suspends, rather than returning, is called a *generator*, since it may generate a succession of values. A generator for producing Fibonocci numbers is shown in Figure 3.

```
int fib() {
   int last1, last, sum;

   last1 = 0;
   last2 = 1;
   suspend(1);
   while (1) {
      sum = last1 + last2;
      suspend(sum);
      last1 = last2;
      last2 = sum;
      }
}
```

Figure 3: A generator for Fibonocci numbers

Notice how both the problems of initialization and memory have been solved. On subsequent calls, execution continues from the point of last suspension. Thus the program can suspend following some initialization code, and subsequently suspend in a different way in the normal case. The variables **last1**, **last2** and **sum** are all local, and no procedure other than **fib** can gain access to them.

The routine **fib** produces a potentially infinite sequence of values. That is, as long as the user continues to ask for values it continues to produce new Fibonocci numbers. Many generators, however, such as the generator which produces leaves from a binary tree, produce only a finite number of values. In cases where a successor value is undefined, a request for the next value is said to cause the generator to *fail*. A generator fails by executing the statement

```
fail;
```

Failure is a terminal state; once a generator has failed it cannot be reinvoked for a subsequent value. A generator that fails is said to have *exhausted* its values. Figure 4 illustrates a generator that fails after producing the list of prime numbers less than ten.

Up to now the syntactic context in which generators can be invoked has not been discussed. One solution is to simply let successive calls on a generator produce successive values. The system function **getchar()**, which produces the next character from the standard input file, can be thought of as such a generator. In general, however, such a solution is unsatisfactory for several reasons. The first reason is that without a detailed examination of the code it is impossible to tell where the first invocation of a generator occurs. Thus in executing a procedure the results may depend on the (potentially unknown) previous calling history of all generators referenced in the program. This problem could be avoided with some mechanism for saying "initialize generator x," In fact in this case such a mechanism would be necessary if generators were to be

```
int p10()
{
  suspend(2);
  for (i = 3; i <= 7; i = i + 2)
    suspend(i);
  fail;
}
```

Figure 4: A generator for the primes less than 10

allowed to produce any more than their initial sequence of values (i.e., to produce the list of primes a number of times, or in different places). However the elimination of such an initialization protocol was precisely one of the motivations given for the use of generators.

The solution to this problem in Cg is to have special statements which mark the syntactic scope of the generation process. The first such statement is the **every** statement.[1] Like the looping statements (**while**, **for**) in C, the **every** statement is followed by the statement it acts upon. Following successful execution of this statement, the most recently suspended generator is reinvoked and control continues from that point. Thus the following statement would print the list of primes produced by the prime number generator **p10** (Figure 4).

```
every {
  i = p10();
  printf(" %d ",i);
  }
```

```
struct tree *nodein(x)
struct tree *x;
{
  struct tree *y;

  if (x != 0) {
    every
      suspend(nodein(x -> left));
    suspend(x);
    every
      suspend(nodein(x -> right));
  }
  fail;
}
```

Figure 5: A generator to return the leaves of a tree

Figure 5 shows a recursive generator for enumerating the leaves of a binary tree. The following statement would add one to every node in the binary tree pointed to by the variable **top**.

```
every {
 node = nodein(top);
 node->value += 1;
 }
```

1 Note that the **every** statement in Cg is not identical to the **every** statement in Icon. In particular the Cg **every** statement cannot be followed by a **do** clause

Note that every is *not* a looping construct, although it is in may respects similar to one. When the inner statement is completed, the last suspended generator is reinvoked, and control proceeds from *that point.* Thus the statement

```
every {
  sum = 0;
  sum += nodein(top)->value;
  }
```

produces the sum of the value fields. The final failure of the generator **nodein** causes the inner statement, and the **every** statement which contains it, to be immediately terminated.

If two or more generators are referenced within the scope of the same **every** statement, the failure of one generator causes the reinvocation of the most recently suspended generator. When control again reaches the generator that failed, it is reinitialized. Thus all possible combinations of generated values can potentially be produced.

```
fil4(x)
   int x;
{
   if (x >= 4)
      suspend;
   fail;
}
```

Figure 6: A generator used as a filter

As a simple example, consider the generator shown in Figure 6. Here a generator is being used as a *filter* to eliminate unwanted values, in this case values less than four. Note the use of the suspend statement without a value, used to merely indicate success.

Using the filter **fil4** in conjunction with the generator **p10**, the statement

```
every {
 i = p10();
 fil4(i);
 printf(" %d ",i);
}
```

causes the values 5 and 7 to be printed. The failure of the generator fil4 when presented with the values 2 and 3 causes the generator p10 to be reinvoked for new values. This process of backtracking, of searching for values that satisfy some criterion (or goal, hence the term *goal directed evaluation*), is one of the most powerful applications of generators and is extensively used the examples.

In using backtracking to solve a problem it is frequently the case that instead of examining all possible combinations of generator values the user merely want to find the first set of values that satisfy some criterion. For this reason there is a second control structure, called the *drive* statement, that reinvokes generators only until one completely successful execution of the inner statement has been accomplished. For example the statement

```
drive {
 i = p10();
 fil4(i);
 printf(" %d ",i);
}
```

prints only the value 5.

Control passes from the drive statement in one of two ways. Either values are generated that permit a complete execution of the inner statement to take place, or all generators become exhausted before such values can be found. In many cases the solution to a problem may require knowing which of these two cases has occurred. In order to allow for this, a drive statement can be followed by two optional clauses, a *then* clause and/or an *else* clause.

In the statement

```
drive
  statement1;
then
  statement2;
else
  statement3;
```

statement$_2$ is executed only if the driven **statement**$_1$ completed normally. If all generators in **statement**$_1$ became exhausted without the statement being able to complete, **statement**$_3$ will be executed. Neither statements 2 or 3 can activate generators (except, of course, within the context of nested **every** or **drive** statements).

There are problems where it is neither appropriate to find all combinations of generator values that satisfy some criterion, nor to find only the first. For these problems the goal is to enumerate values until some condition is specified. An example problem of this sort is to produce all Fibonocci numbers up to and including the first number greater than 50. Note that a filter (as we used in the prime number problem in Section 1) cannot be used in this case, since the Fibonocci generator does not terminate. One way to solve this problem is to write a special routine, as was done for the prime number example. A better solution is to use the drive statement with an **until** clause, as in the following example:

```
drive {
  i = fib();
  printf(" %d ",i);
  }
until (i > 50);
```

The **until** clause indicates that the inner statement is to be repeatedly reinvoked until the condition becomes true. Thus an **every** statement can written as "**drive** .. **until** (0);".

As in looping constructs, it is not always convenient to test for termination at the beginning or end of the inner statement. This problem is handled in a manner similar to that used in looping constructs. The statement

```
break;
```

causes the innermost **drive** or **every** statement to be immediately terminated, and execution continues with the next statement not in the scope of the **drive** or **every** statement ("breaking out" of the loop). The statement

```
continue;
```

acts like a generator failure, in that it causes the most recently suspended generator to be reinvoked. For example the following statement can be used to produce the list of Fibonocci numbers less than 50.

```
every {
  i = fib();
  if (i > 50)
    break;
  printf(" %d ",i);
}
```

Using **break** and **continue**, the **every** statement can be described in terms of the **drive** statement, and vice versa. These equivalences are shown in Figure 7.

```
drive                                   every {
  statement;                              statement;
                                          break;
                                        }

every                                   drive {
  statement;                              statement;
                                          continue;
                                        }
```

Figure 7: Drive and Every equivalences

2. Recursion, Data structures

A generalization of the prime number generator in Section 1 is a program for generating all the primes less than some value n. The body of such a program might look something like the following

```
int prime(n)
  int n;
{
  int i;
  suspend(2);
  for (i = 3; i <= n; i = i + 2)
    if (i is a prime)
      suspend(i);
  fail;
}
```

Here the only difficulty is to fill in the "if i is a prime" conditional. A number is defined to be prime if it is not divisible by any smaller prime number. This definition points clearly to a recursive solution to the problem, since the primality test is defined in terms of earlier prime numbers. Note also that it is only necessary to check primes smaller than sqrt(i), since primes larger than sqrt(i) cannot possibly divide i. A complete recursive solution is shown in Figure 8.

```
int prime(n)
  int n;
{
  int i, j;

  suspend(i);
  for (i = 3; i <= n; i = i + 2)
    drive {
      j = prime(i-1);
      if (i % j == 0)
        break;
      }
    until (j * j > i);
    then suspend(i);
  fail;
}
```

Figure 8: A generator for prime numbers

A classic problem used to illustrate backtracking methods is the eight queens problem [11]. The problem involves finding the ways that eight queens can be placed on a chess board so that no queen can attack any other.

The solution given here is based on the Icon solution given in [3]. A generator q(c) attempts to place queen number c. If it is successful at finding a free location, it suspends. If it is unable to find a free location, it fails, causing the most recently suspended queen to search for a new location.

Three arrays keep track of the free rows, upward facing diagonals, and downward facing diagonals. Free squares are indicated by zero values, while squares which are occupied are indicated by the value one.

A program to solve the eight queens problem is shown in Figure 9. Note that the program finds all solutions to the eight queens problem. Replacing the **every** statement in the main routine with a **drive** statement causes the program to halt after producing the first solution.

```
int up[15] = 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
int down[15] = 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
int rows[8] = 0,0,0,0,0,0,0,0;

main(){
  every printf("%d %d %d %d %d %d %d\n",
        q(1),q(2),q(3),q(4),q(5),q(6),q(7),q(8));
}

int q(c)
  int c;
{
  int r;

  for(r = 1; r <= 8; r++)
    if (rows[r-1] == 0 && up[r-c+7] == 0 && down[r+c-2] == 0) {
      rows[r-1] = up[r-c+7] = down[r+c-2] = 1;
      suspend(r);
      rows[r-1] = up[r-c+7] = down[r+c-2] = 0;
      }
  fail;
}
```

Figure 9: A solution to the eight queens problem

3. An Application of Generators to the Problem of Pattern Matching

In Chapter 5 of their book *Software Tools* [5], Kernighan and Plauger present a simple pattern matching module. Their code implements literal matches, character classes (simple one character alternation), beginning-of-line and end-of-line matching, arbitrary character matching, and closures of any single character pattern. The omission of arbitrary alternation and closures (which would allow the pattern matcher to recognize any regular expression) seems curious, until one attempts to write such a regular expression pattern matcher based on the model in [5]. As such an attempt quickly illustrates, the problems of alternations and closures can be very subtle. Consider, for example, the pattern (a*|aba*)* matching the text aabaaba.

Conventional solutions which do implement complete regular expression matching tend to do so by simulating the non-deterministic finite state automata recognizer, and tend to be larger and more complex than the *software tools* program. In contrast, the generator solution described here is much closer in style and spirit to the solution in [5].

There are six basic patterns. These patterns are translated into an internal character encoding, represented here using the symbolic character constants **CHAR, BOL, EOL, ANY, CCL, and CLOSURE**. Each basic pattern, encoding, and meaning is given by the following chart.

Symbol	Encoding	Meaning
c	**CHAR** *c*	Matches the literal character *c*
%	**BOL**	Matches the beginning of line
$	**EOL**	Matches the end of line
?	**ANY**	Matches any single character
[*charlist*]	**CCL** *number charlist*	Matches any single character from the character list
[~*charlist*]	**NCCL** *number charlist*	Matches any single character not in the character list

In addition, any pattern except % and $ can be followed by a * to represent zero or more repetitions of the indicated pattern. The largest string matching the pattern is tested first, and if that fails successively smaller matches are attempted.

Closures are represented internally by the symbolic constant **CLOSURE**, followed by number of characters to the end of closure character (**EOC**), followed by the pattern to be replicated, followed by the symbolic constant **EOC**. Alternation is represented internally by the constant **BA**, the number of characters to the end of the alternatives, the number of characters to the next alternative, the first alternative pattern, the list of alternatives, and finally the constant **EA**. The list of alternatives is composed of the constant **AL**, followed by the number of characters to the next alternative, followed by the alternative pattern, repeated as often as necessary.

The details of how the external representation gets translated into the internal form are covered in [5], and are not presented here.

The primary routine involved in this portion of the pattern matcher is **amatch**, which takes a text line, a position in the line, a pattern, and a position in that pattern, and returns either zero, if the pattern fails to match the given position, or the position of the next character following the pattern match. In the generator solution, **amatch** simply drives the generator **rmatch**, attempting to find a valid match. If **rmatch** terminates unsuccessfully, **amatch** returns zero. The code for **amatch** is as follows (**EOS** is a symbolic constant used to represent the end of character strings).

```
int amatch(lin, i, pat, j)
  char lin[], pat[];
  int i, j;
  { int k;

  drive {
   k = rmatch(lin, i, pat, j, EOS);
   return(k);
   }
  return(0);
}
```

At the heart of the pattern matcher is a conventional routine that implements simple pattern element matching. The routine **omatch** takes the pattern starting at pat[j] and sets the variable **bump** to the number of characters matched. For successful matches, this is either zero (for **BOL** and **EOL**) or 1 (for all other patterns). For unsuccessful matches, **bump** is set to -1. **omatch** returns either zero (if the match is unsuccessful) or the updated position in the text pattern. The program **omatch** is shown in Figure 10.

```
int omatch(lin, i, pat, j)
  char lin[], pat[];
  int i, j;
  { int bump;
    char c;

  bump = -1;
  c = lin[i];
  if (c == 0)
    return (0);
  switch (pat[j]) {
    case CHAR:  if (c == pat[j+1])
                  bump = 1;
               break;
    case BOL:   if (i == 1)
                  bump = 0;
               break;
    case EOL:   if (c == '0)
                  bump = 0;
               break;
    case ANY:   if (c != '0)
                  bump = 1;
               break;
    case CCL:   if (locate(c, pat, j + 1))
                  bump = 1;
               break;
    case NCCL:  if (c != '0 & locate(c, pat, j + 1) == 0)
                  bump = 1;
               break;
    default:
      error("in omatch: can't happen");
    }
  if (bump < 0)
    return (0);
  return (i + bump);
}
```

Figure 10: Matching pattern elements

Without closures and alternatives, a simple version of **rmatch** can then be written as follows:

```
int rmatch(lin, i, pat, j, delim)
  char lin[], pat[], delim;
  int i, j;
  { int k, l, m, ap;

  for ( ; pat[j] != EOS && pat[j] != delim; j = j + patsize(pat,j))
    if ((i = omatch(lin, i, pat, j)) == 0)
      fail;
  suspend (i);
  fail;
}
```

Now consider the problem of alternation, say for example the pattern "(a|ab)c". There are two separate problems: matching the current alternative ("a" or "ab") and matching the remainder of the pattern ("c").

Expressed generally, the solution is to try to match the current alternative; if that is successful try to match the remainder, if both matches are successful update the text marker, otherwise if either match fails try the next alternative. If no alternatives remain the match fails.

The only complication to this approach involves nested alternations, for example "((a|ab)|b)". Instead of simply matching the current alternative and the continuation, they must be driven through all possible outcomes, and only if no successful matches can be found is the next alternative attempted. Thus the code for alternation is as follows:

```
if (pat[j] == BA) { /* beginning of alternatives */
    k = j + pat[j+1]; /* pointer to continuation */
    ap = j+1;
    while (1) {
      every {
        l = rmatch(lin, i, pat, ap+2, AL);
        m = rmatch(lin, l, pat, k, delim);
        suspend(m);
        }
      if (pat[ap+1] == 0)
        break;
      ap = ap + pat[ap+1];
      }
    fail;
    }
```

Now consider the problem of closures. Again there are two subpatterns; the pattern to be replicated and the remainder pattern. The simplest way to manage this is with a recursive routine **cmatch**

```
int cmatch() {
  try to match closure pattern
  if successful
    recursively call cmatch
    suspending the result
  try to match remainder
  }
```

Again, in order to handle nested closures it is necessary that the recursive calls to **rmatch** be driven through all possibilities. The final routine **cmatch** is shown in Figure 11. The final routine **rmatch** is Figure 12.

```
int cmatch(lin, i, pat, cls, cns, delim)
   char lin[], pat[], delim;
   int i, cls, cns;
   {  int k;

   every {
      k = rmatch(lin, i, pat, cls, EOC);
      if (k > i)
         suspend( cmatch(lin, k, pat, cls, cns, delim) );
      }
   every
      suspend( rmatch(lin, i, pat, cns, delim) );
   fail;
}
```

Figure 11: The routine **cmatch**

```
int rmatch(lin, i, pat, j, delim)
  char lin[], pat[], delim;
  int i, j;
  { int k, l, m, ap;

  for ( ; pat[j] && pat[j] != delim; j = j + patsize(pat,j))
    if (pat[j] == CLOSURE) {
      k = j + pat[j+1];  /* end of closure */
      every {
        l = cmatch(lin, i, pat, j+2, k, delim);
        suspend(l);
        }
      fail;
      }
    else if (pat[j] == BA) {  /* beginning of alternatives */
      k = j + pat[j+1];  /* pointer to continuation */
      ap = j+1;
      while (1) {
         every {
           l = rmatch(lin, i, pat, ap+2, AL);
           m = rmatch(lin, l, pat, k, delim);
           suspend(m);
           }
         if (pat[ap+1] == 0)
           break;
         ap = ap + pat[ap+1];
         }
      fail;
      }
    else if ((i = omatch(lin, i, pat, j)) == 0)
      fail;
  suspend (i);
  fail;
}
```

Figure 12: The routine **rmatch**

4. Implementation

Cg is implemented as a preprocessor to the C compiler [6]. The preprocessor was written using YACC, a compiler-complier running under the UNIX[2] timesharing system. In the resulting C programs, registers 2 and 3 are reserved for the use of the generator runtime primitives. With the exception of declarations for these registers, a few built-in generators, and the translation of **drive** and **every** statements, the source program is left unchanged.

There are five primitive routines, written in assembly code, which implement the generation process. In order to provide a context for understanding the interaction of these primitives, note that an **every** statement is translated into the code shown in Figure 13, and a **drive** statement into the code shown in Figure 14. Parts of the **drive** code sequence may be omitted out if there are no optional clauses. Within the context of an **every** or **drive** statement, **break**s are translated into the sequence "**unmark(); break;**".

```
do if (!mark())
        statement;
while (drive(),0);
```

Figure 13: The code produced for an **every** statement

```
do
        if(!mark()){
                statement;
                if (! (untilcondition)
                        continue;
                unmark();
                thenstatement;
                break;
        }
        else {
                drive();
                elsestatement;
                break;
        }
while (drive(),0);
```

Figure 14: The code produced for a **drive** statement

The primitive operations are as follows:

Mark	mark the current stack position
Suspend	suspend execution
Fail	indicate generator failure
Drive	revive the most recently suspended generator
Unmark	remove generator information from the stack

Detailed descriptions of each of these primitive operations follow.

2. UNIX is a Trademark of Bell Laboratories

mark

This routine marks the stack position at the beginning of a driven statement. There are two pointers which are used to save/restore information: the *expression frame pointer* and the *generator frame pointer*. For efficiency, both are kept in registers.

The mark routine saves the previous expression frame and generator frame pointers, in addition to the return program counter for the call on mark. This block is called the *expression frame*. The *expression frame pointer* is set to the address of this block, and the *generator frame pointer* is set to zero. The resulting stack is as shown in Figure 15.

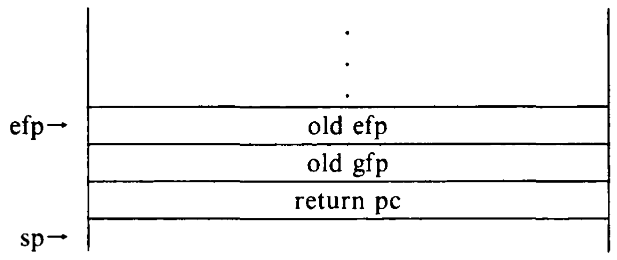

Figure 15: The stack after calling mark

suspend

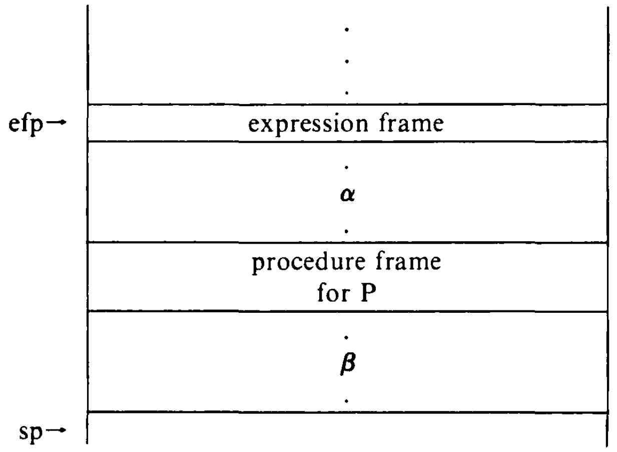

Figure 16: The stack before calling suspend

When suspend is called the current stack looks something like Figure 16. Here α contains any temporaries placed on the stack prior to calling the generator P. P creates a procedure frame, containing the arguments to P, return program counter, registers, and automatic storage. β contains any temporaries that P may have created.

The suspend routine first saves the registers, including the generator and expression frame pointer registers. This block is called the *generator frame* and the generator frame pointer is set to its address. Next the area α is *duplicated* on the stack. Finally the registers saved by P are restored, and the suspended value is returned to the procedure which called P. To the calling routine, the stack now looks *exactly* as if P had returned normally. The information between the expression frame and generator frame pointers is effectively hidden from any access by the calling program.

In the case of multiple generators, the stack at the time of the suspend operation looks something like Figure 18. The actions taken by suspend are the same as in the previous case, with the exception that only the area between the last generator frame pointer and the current procedure frame (γ) is duplicated.

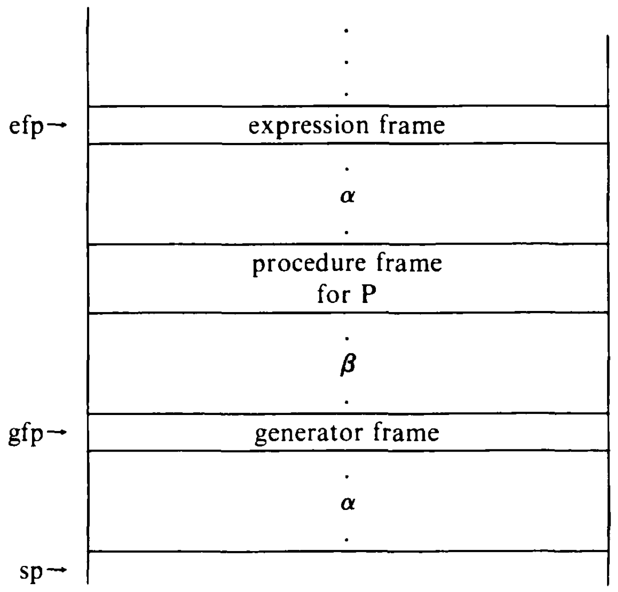

Figure 17: The stack after calling suspend

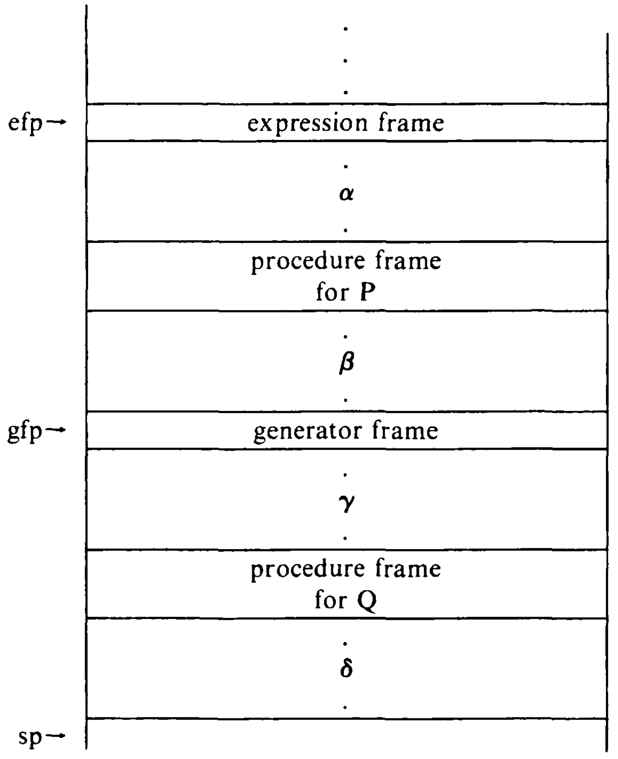

Figure 18: The situation with multiple generators

fail

Fail basically simulates a nonzero value being returned from the most recent call on mark. The return program counter for this purpose is taken from the current expression frame. Note that mark in this case transfers to the routine drive. Branching to mark is necessary since the program counter for the current call on drive is not known to the generator routines, and the correct call on drive must be invoked in case there are no pending generators (see the description below).

drive

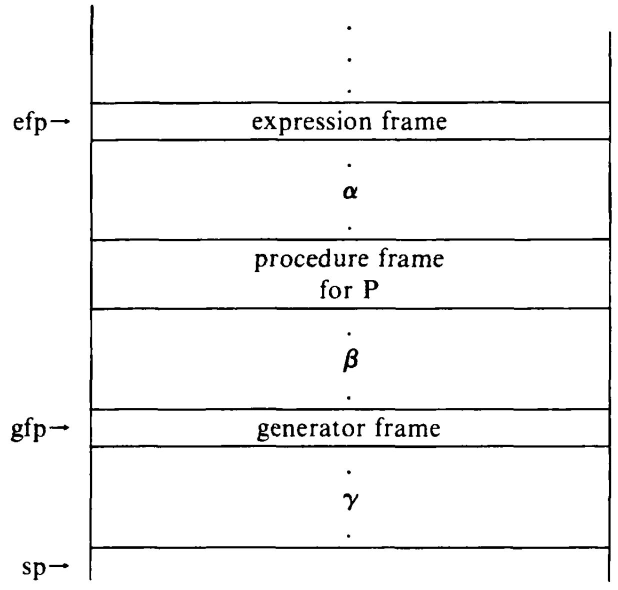

Figure 19: The stack just before calling drive

If there are no dormant generators, the drive routine simply calls unmark, and returns. If there are dormant generators the stack is as shown in Figure 19. Here α and β are as in Figure 16, and γ is the temporary area for P. The stack is popped down to the generator frame pointer (note that γ cannot contain any currently relevant information) and the registers saved in the generator frame are restored. The generator frame is popped, and execution passed back to the statement following the suspend statement in P. The registers and stack are now just as if control had never passed from P. The stack at the end of the drive operation is shown in Figure 20.

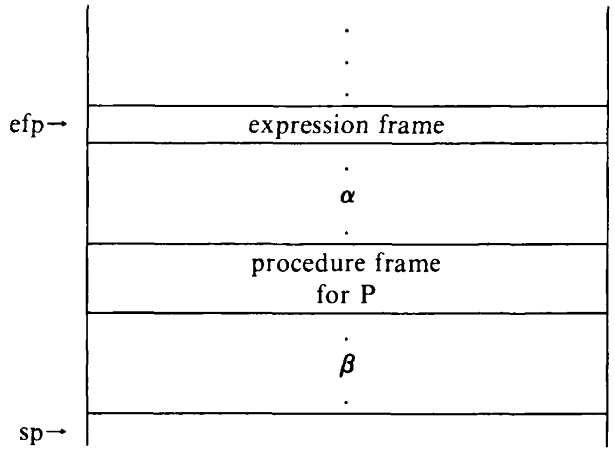

Figure 20: The stack just after calling drive

unmark

The unmark routine pops the stack down to the last expression frame, and restores the previous values the the expression frame and generator frame pointers. Unmark returns normally to the caller.

5. Time and Space Requirements

In analyzing the overhead involved in using generators there are two factors to consider, *time* and *space*. The overhead involved in time is not excessive, since each basic run-time routine is only an handful of instructions. The greatest cost, and the only loop, in the run-time system is in copying the part of the stack saved in the suspend operation.

The amount of stack space used depends on several variables. If the generator suspended is several levels below the statement which is driving the generator, it is possible for the duplicated area to contain several procedure frames and hence to be quite large. In the more usual situation the suspended procedure is only one level below the driving statement, and the area which is saved contains only those temporary values used in calling the generator, the generator procedure frame, and those temporary variables used in computing the suspended value. It is more common for the saved area to be on the order of a few dozen bytes.

6. Conclusions

The notion of generators is not unique to Cg, having been adapted from the language *Icon*. What is novel about Cg is that fact that generators have been adapted for use in a conventional programming language. The examples cited in this paper demonstrate that generators can be beneficial in the solution of many programming problems. However, up to now the use of generators has been restricted to a handful of languages and they have not received widespread recognition. Programmers tend to have favorite languages, and to write almost exclusively in those languages, often bending the problem to fit the language. By extending a language that has already acquired a widespread user community, Cg allows programmers to add a new capability to their repertoire of tools, without the need to learn a totally new computer language.

Acknowledgments

The language Icon is principally due to Ralph Griswold, and he is responsible for introducing me to the idea of generators. Cary Coutant provided invaluable aid in describing the implementation of generators in Icon on the PDP-11/70, and in writing the assembly language run time routines for Cg. The binary tree example was suggested by Steve Wampler. This work has also benefited from discussion on the nature of generators with the Icon group at the University of Arizona, consisting of Ralph Griswold, Cary Coutant, Steve Wampler, Dave Hanson and Tim Korb. Ralph Griswold provided extremely detailed and helpful comments on an earlier draft of this report.

References

[1] D. C. Smith, and H. K. Enea, *Backtracking in MLISP2*, Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Page 677-685.

[2] M. E. Conway, Design of a separable transition-diagram compiler, *Communications of the ACM* 6(7), July 1963, 396-408.

[3] C. A. Coutant, R. E. Griswold, and S. B. Wampler, *Reference Manual for the Icon Programming Language; Version 4*, University of Arizona Technical Report 81-4, 1981.

[4] D. R. Hanson and R. E. Griswold, The SL5 procedure mechanism, *Communications of the ACM* 21, May 1978, 392-400.

[5] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

[6] B. W. Kernighan and D. M. Richie, *The C Programming Language*, Prentice-Hall, 1978.

[7] J. T. Korb, *The Design and Implementation of a Goal-Directed Programming Language*, PhD thesis, University of Arizona, 1979. Also University of Arizona Technical Report TR 79-11.

[8] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, Abstraction mechanisms in CLU, *Communications of the ACM* 20(8), August 1977, pages 564-576.

[9] M. Shaw, W. A. Wulf and R. L. London, Abstraction and verification in Alphard: defining and specifying iteration and generators, *Communications of the ACM*, 20(8), 553-564, August 1977.

[10] K. Thompson, D. M. Ritchie, cr (VII), coroutine scheme. In *Unix Programmer's Manual*, sixth edition, May 1975.

[11] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.