MULTI - A LISP Based Multiprocessing System

Donald P. McKay and Stuart C. Shapiro

Department of Computer Science
SUNY/Buffalo
4226 Ridge Lea, Amherst, New York. 14226

Abstract

A package of LISP functions, collectively called MULTI, which extends LISP 1.5 to multiprogramming is presented. MULTI defines the notion of a process within a LISP implementation using function invocation as the only control primitive. A process is an executable entity consisting of a process template and a set of register values. The process template defines the operations the process carries out. Process environments are saved in what can be viewed as function call instances, i.e. LISP forms which have the name of a process template in functional position and the register values following it. The flexibility of this simple conceptualization of processes is demonstrated by several examples which use MULTI to implement recursion, backtracking, generators, agendas and AND/OR graph searching. The implementation of MULTI does not assume that the host LISP system provides any data or control environment saving mechanisms such as FUNARG or INTERLISP's spaghetti stack. Thus, MULTI is portable to other LISP systems.

This material is based on work supported in part by the National Science Foundation under Grant No MCS78-02274.

1. Introduction

The motivations of this paper are twofold. First, it describes a package of functions which enables a LISP program to define and use processes. Vaguely, a process is an instance of a function invocation (see Section 2). In general, the ability to manipulate processes yields various control structures such as recursion, backtracking, coroutines and generators. The package described here is transportable (with minor modification) to other LISP systems because it relies on function invocation as the only control structure primitive. In fact, we have implemented MULTI in LISP 1.6[24], UTLISP[10], and ALISP[16]. A second motivation is the desire to have a flexible control structure available for use in artificial intelligence programs. Here, our primary objective was to provide a facility for use in writing a deduction system for the SNePS semantic network processing system[26]. For example, the formula (P_1 v P_2 v ... v P_n), where the P_i are propositions and "v" is OR, could be proved by trying to show each of the P_i's in "parallel" and terminating the proofs of all the other disjuncts as soon as any one is proved. This "chronological" approach to disjunction is similar to the OR of Friedman and Wise[9] and the EITHER of Baker and Hewitt[2] which returns true as soon as any disjunct evaluates to true.

This paper presents a general overview of MULTI and gives several examples. The examples are intended to show the flexibility of the system and to demonstrate the utility of the notion of a process in programming. We are not proposing MULTI as a programming language but rather explaining in detail a system that we have been using for several years. Our current implementation of MULTI has been influenced by practicality, i.e. how to interface with ALISP [16] and how to write, debug and maintain programs which use MULTI. Debugging facilities available to a MULTI user in ALISP are detailed in a separate report [21].

2. General Description of MULTI

MULTI is a LISP based multiprocessing system designed for use as the control structure of a deduction system [29]. Strictly speaking, MULTI adds multiprogramming capabilities to LISP 1.5 [20] and Standard LISP[18]. MULTI consists of a simple evaluator, system primitives, a scheduler, and a debugging facility. The evaluator continually executes processes from a process queue until the queue becomes empty. System primitives include functions for creating processes, for scheduling processes to be executed, and for manipulating local variables or registers. The scheduler inserts a process into the process queue. Debugging facilities include a trace package and a break facility.

Conceptually, a MULTI process is similar to a "computational frame" [32] which consists of an "action", a "datum" or argument(s), "bindings", and a "continuation". The action specifies some task to be performed. The bindings define an environment in which local identifiers are given values. The continuation is a reference to another computational frame where processing is to continue upon completion of the action. Other sources of influence include the "activation record" concept of ALGOL [23] and the class concept of SIMULA [6]. An activation record generally contains local data, parameters, return point, temporary storage, and code to be executed. In SIMULA, classes are procedures which can have several instances active simultaneously. In a sense, classes are the obvious extension of ALGOL procedures. Instances of a particular class are called "objects". Once an object is created it remains in existence until all references to it have been severed. With the concept of classes, SIMULA includes control primitives which allow coroutine activity. MULTI treats processes like objects and the basic structure of a process is akin to a computational frame or activation record. Kupiers [17] states that SIMULA also influenced the design of so called "actor languages", ACTORS [12,13] and SMALLTALK [15].

A distinction is made between the definition of a MULTI process and a particular instance of a process. The term "process template" will be used to refer to the definition of a type of a process. A process template includes several local registers (variables) among which must be a NAME: and a CLINK: register. The NAME: refers to an action, a LISP function to be executed. The CLINK: specifies the continuation, another process, to which the process will return control. Also, the CLINK: serves as a communication link to another process. The remaining registers comprise the local environment. The term "process" refers to an instance of a process template with suitable values stored in its registers. A process is an executable entity; a process template is not executable. A similar distinction is made in SIMULA between objects and classes. In MULTI, each process is assigned a unique identifier whose NAME: identifies its process template. The identifier's LISP value is an ordered list of the values of its registers.

A process is not eligible for activation until it is placed in the process queue. Any process may create as many other subprocesses as it wishes, specifying for each subprocess any other process the parent process "knows" about as a continuation. A process remains in the system until no references to it exist, when it becomes eligible for LISP garbage collection.

The LISP function MULTIP implements the basic MULTI execution cycle which consists of selecting a process from the process queue and running it. Since the process queue is assumed to be ordered, MULTI always selects the first process on the queue. To execute the selected process, the process template is applied to the value of the process' identifier. MULTIP continues selecting and executing processes until the process queue is empty.

A MULTI process is a non-interruptable computation. This assumption makes a process similar to a procedure in that every execution of a process must run to completion and if a process is reactivated then it must be restarted from the beginning of its code. The reason for the completion assumption is to allow arbitrary LISP functions as process templates without writing a special process template interpreter or rewriting the LISP interpreter.

In general, a coroutine is a procedure which is suspendable in the midst of its execution. Thus, coroutines seem to violate the completion assumption. In MULTI, there are a number of ways to specify a coroutine with multiple parts. The simplest method is to create a new process which has as its NAME: the process template which implements the next part of the coroutine and to make the continuation of the current process the continuation of the new process. For example, Figure 1a shows a process P1, which is the current process and which is NAME:d A, and its CLINK:, P2. The arrows between processes in this diagram (and in subsequent diagrams) represent CLINK:s. Suppose P1 is a multipart coroutine and the NAME: of the next part is B. Suppose further that P1 creates a new process, P4, which P1 makes the current process. By making P4's CLINK: a B process, P3, P1's continuation can be scheduled when P4 terminates. The resulting control set is shown in Figure 1b. When the continuation of a coroutine has the same registers, an equivalent effect can be obtained by reNAME:ing the current process. This results in the control set shown in Figure 1c. In the remainder of the paper we use the term continuation to refer both to the value of a process' CLINK: and to the NAME: of the next process template in a coroutine. In the current example, P2 is the continuation of P1 and B is the continuation of A.

Finally, another implementation of coroutines is to include a state register (distinct from the NAME: register) which a process uses to select code to execute. Fikes [7] implemented coroutines for use in a modelling system in just this way. However, the implementation relied on FUNARG to save local environments, i.e. values of registers.

3. MULTI Primitives

Before discussing some example uses of MULTI we give a detailed presentation of the MULTI primitives. DP is a function which defines a process template. It is analogous to DE in most LISPs. Figure 2a gives an example definition of a MULTI analogue of LISP's PLUS. The effect of the call to DP is to define MPLUS as a process template with registers A1, A2, and ANS (besides NAME: and CLINK: which DP adds). The process template consists of a corresponding LISP function MPLUS shown in Figure 2b. Note that NAME: and CLINK: have been added to the argument list and that an

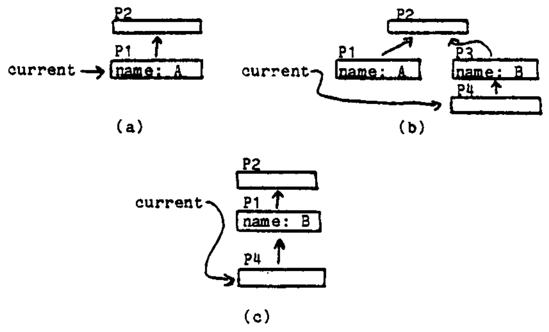

Figure 1
An implementation of coroutines using continuations.

extra form has been added to the function definition. The extra form in the LAMBDA expression version of the process template recovers the possibly changed register values when the process terminates (see below).

MULTI's evaluation loop is the LISP function MULTIP, a function of one argument, which is a list of process identifiers. This list is bound to MULTIP's process queue, EVNTS. Each execution cycle consists of removing the first process from EVNTS, making it the value of the variable CURNT:, and applying the process template to the value of the process identifier. Since a process template is a LISP function with register names as lambda variables, and the value of a process identifier is an ordered list of register values, the effect is that during execution, a process can use its register names to refer to their values, and CURNT: to refer to itself. MULTIP terminates when, at the beginning of a cycle, EVNTS is NIL. This can occur because no new processes are scheduled by the last ones executed, or because some process intentionally sets EVNTS to NIL. In the MPLUS example above, if "P1" is the identifier of an MPLUS process, (MULTIP (LIST 'P1)) would cause P1 to execute.

The functions REGFETCH and REGSTORE allow access to registers of a given process. REGFETCH, a function of two arguments, a process identifier and a register name, returns the current value of the specified register. REGSTORE takes a third parameter and changes the value of the specified register. In MPLUS, (SETQ ANS (PLUS A1 A2)) could be replaced by (REGSTORE CURNT: 'ANS (PLUS (REGFETCH CURNT: 'A1)(REGFETCH CURNT: 'A2))).

The function NEW creates a new instance of a process template. For the present example, (NEW 'MPLUS NIL 1 3 NIL) creates a new MPLUS process with a null CLINK:, addends 1 and 3, and a null ANS. NEW returns the unique identifier of the new process.

INITIATE invokes the scheduler to place its argument, which is a process identifier, on the process queue. It does this by calling the function SCHEDULE. SCHEDULE may be defined by the MULTI user to implement various scheduling regimens. Figure 3 shows the default SCHEDULE, which results in a first-in-first-out regimen. Section 4.2 presents an example which redefines SCHEDULE.

```
(DP  MPLUS  (A1 A2 ANS)
            (SETQ ANS (PLUS A1 A2)) )
```

(a)

```
(LAMBDA  (NAME: CLINK: A1 A2 ANS)
         (SETQ ANS (PLUS A1 A2))
         (SET CURNT: (LIST NAME: CLINK: A1 A2 ANS)) )
```

(b)

Figure 2
(a) Definition of process template; (b) resulting LAMBDA expression.

4. Examples

We now present a series of examples which demonstrate MULTI's ability to implement control structures of general utility.

4.1 Backtracking and Coroutines - The n queens problem

Our first example implements a solution to the n queens problem -- find a way to place n queens on an n by n chessboard so that none is attacking any other, i.e. so that no two are on the same row, column or diagonal. Our solution is roughly based on Wirth's [33]. The first two process templates, QUEENS and FAIL, implement a coroutine whose first part begins the solution and whose continuation is executed only if no solution exists. The two process templates are shown in Figure 4. (See [25] for definition of PRIN3.) Note that QUEENS specifies its continuation by reNAME:ing itself. Three other process templates implement a coroutine in three parts -- START places a queen on its row, STEP moves its queen to the next non-attacked square, REPORT is executed only when the problem has been solved and reports where its queen is placed. The register COLS records the columns already attacked by some queen. PDIAGS and MDIAGS record the attacked diagonals. These process templates appear in Figure 5. (See [16], [25] or [34] for the definition of REPEAT; it is similar to PROG in that its first argument is a list of local variables.)

Backtracking is implemented in the coroutine by expecting success and propagating failure. When a STEP process places a queen on a column, it prepares for success by making its continuation a REPORT process. When a STEP process fails, i.e. COL becomes 0, it propagates failure to its CLINK: by changing its CLINK:'s NAME: to STEP, which resumes looking for a new column on which to place its queen.

Evaluation of (MULTIP (LIST (NEW 'QUEENS NIL 8))) results in the output:

```
PUT  A  QUEEN  ON  ROW  1  AND  COLUMN  5
PUT  A  QUEEN  ON  ROW  2  AND  COLUMN  7
PUT  A  QUEEN  ON  ROW  3  AND  COLUMN  2
PUT  A  QUEEN  ON  ROW  4  AND  COLUMN  6
PUT  A  QUEEN  ON  ROW  5  AND  COLUMN  3
```

```
(DE SCHEDULE (PROCESS QUEUE)
   (COND
    ((NULL QUEUE) (LIST PROCESS))
    (T (CONS (CAR QUEUE)(SCHEDULE PROCESS (CDR QUEUE))) )))
```

Figure 3
Definition of default scheduler.

```
(DP  QUEENS  (N)
     (INITIATE (NEW 'START CURNT: N N (ADD1 N)  ;Create and schedule
                    NIL  NIL  NIL))   ;initial START process.
     (SETQ NAME: 'FAIL) )     ;Make continuation a FAIL process.


(DP  FAIL  (N)
       (PRIN3 "THE" *N "QUEEN PROBLEM IS IMPOSSIBLE" <>) )
```

Figure 4
The process templates QUEENS and FAIL.

```
PUT A QUEEN ON ROW 6 AND COLUMN 1
PUT A QUEEN ON ROW 7 AND COLUMN 4
PUT A QUEEN ON ROW 8 AND COLUMN 8.
```

Figure 6 shows a snapshot of the processes when a STEP process is working on row 3. The N register of the STEP and REPORT processes are not shown. The evaluation of (MULTIP (LIST (NEW 'QUEENS NIL 2))) produces the response:

```
THE 2 QUEENS PROBLEM IS IMPOSSIBLE.
```

4.2 Agendas - Sieve of Eratosthenes

A second example is an implementation of the Sieve of Eratosthenes algorithm for listing prime numbers. The basic algorithm is to declare 2 prime and then mark all multiples of 2 up to some maximum as not prime. The next step is to advance to the first number not marked and repeat the declaration and marking phases for this prime, and so on. The sieve is easy to implement in a language such as PASCAL [14] using an array as the primary data structure. However, such an implementation requires space proportional to the maximum number to be tested, i.e. the length of the array. Our implementation requires only one process per prime, so the space is proportional to the number of primes produced.

We use the MULTI process queue and a scheduler with ordered insertion to generate the primes between 2 and some maximum N. This implementation is based on an example program described in a SIMULA reference manual [30] which uses the predefined SIMULATION class. The basic notion is to use the simulation clock to represent the integers. Because there is no built in simulation class in MULTI, we effectively define one by changing the scheduler to use an ordered queue. The ordering relation is based on the TIME: register, which every process in this example is required to have.

```
(DP  START  (N ROW COL COLS PDIAGS MDIAGS)
 (COND ((ZEROP ROW)                   ;Done if ROW = 0
         (INITIATE CLINK:))           ;schedule REPORT process.
       (T (SETQ NAME: 'STEP) ;Otherwise make continuation STEP
            (INITIATE CURNT:))) )     ;and schedule it.


(DP  STEP  (N ROW COL COLS PDIAGS MDIAGS)
   (COND
    ((REPEAT NIL                     ;Search for free column.
            (SETQ COL (SUB1 COL)) ;Initially COL = N+1.
       UNTIL (ZEROP COL)
       WHILE (OR (MEMBER COL COLS)
                 (MEMBER (PLUS ROW COL) PDIAGS)
                 (MEMBER (DIFF ROW COL) MDIAGS)))
     (IF (EQ (REGFETCH CLINK: 'NAME:) 'REPORT)
         (REGSTORE CLINK: 'NAME: 'STEP))   ; Propagate failure.
     (INITIATE CLINK:))   ;If not found, force another solution.
    (T  (SETQ NAME: 'REPORT)        ;Prepare for success.
     (INITIATE                 ;If found, create new START process
      (NEW 'START
           CURNT:                  ;with
           N
           (SUB1 ROW)                     ;next ROW,
           (ADD1 N)                ;initial COL off board and
           (CONS COL COLS)          ;current COL and diagonals
           (CONS (PLUS ROW COL) PDIAGS)        ;reserved.
           (CONS (DIFF ROW COL) MDIAGS))))) )


(DP  REPORT  (N ROW COL COLS PDIAGS MDIAGS)
   (PRIN3 "PUT A QUEEN ON ROW" *ROW "AND COLUMN" *COL <>)
   (IF (EQ (REGFETCH CLINK: 'NAME:) 'REPORT) ;Report ROW and COLumn
       (INITIATE CLINK:)))       ;and propagate reporting.
```

Figure 5
START, STEP and REPORT process templates.

The function SCHEDULE, shown in Figure 7, maintains an ordered list of processes based on the value of the TIME: register. Note that this definition of SCHEDULE overrides the default scheduling function (see Section 3).

The top-level LISP function is PRIMES, shown in Figure 8. PRIMES lists all primes less than N. First, it prints that 2 is prime, initializes a PRIME process to start at TIME: 3 and schedules a termination process HALT to run at TIME: N. The three process templates, also in Figure 8, are PRIME, BLOCK and HALT. PRIME prints that its number, TIME:, is prime. It then creates a BLOCK process to prevent multiples of its TIME: being declared prime. The BLOCK process template creates a new PRIME process at TIME: + 2 if the TIME: of the next scheduled process is greater than the current TIME: + 2. The BLOCK process then reschedules itself to run at the next odd multiple of the TIME: of the PRIME process which created it. Essentially, the BLOCK processes are checking each odd number in the interval [3, N). The HALT process terminates the operation by emptying the process queue. The program can be made slightly more efficient if the scheduler advances a process' TIME: register so as to avoid scheduling more than one process at any given TIME:. Calling the function PRIMES with an argument of 1000 causes 160 PRIME and BLOCK processes to be created each of which uses approximately 6 words of storage. The ordered list of processes in the queue "at a TIME: of 11" is ((PRIME nil 11) (BLOCK nil 15 6) (BLOCK nil 15 10) (BLOCK nil 21 14) (HALT nil 1000)).

Others have described multilevel agendas [4,5], i.e. an ordered list of priority queues. To extend the notion of multilevel agendas, we

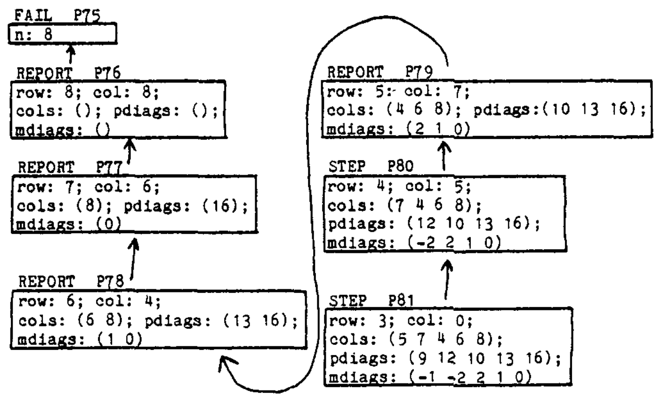

Figure 6
Snapshot of processes just after P81 changes P80's NAME:.

```
(DE SCHEDULE (PROCESS QUEUE)
   (COND
    ((NULL QUEUE) (LIST PROCESS))
    ((GREATERP (REGFETCH PROCESS 'TIME:)
               (REGFETCH (CAR QUEUE) 'TIME:))
     (CONS (CAR QUEUE) (SCHEDULE PROCESS (CDR QUEUE))))
    (T (CONS PROCESS QUEUE))))
```

Figure 7
Scheduler for Sieve of Erastosthenes example.

define a monitor to be a type of MULTI process which has as the value of one of its registers a process queue. When a monitor is the CURNT: process, its process queue is the MULTI process queue. A monitor can perform any action on its process queue. Thus, a monitor can detect that its process queue is empty. The monitor itself is also a repository for processes. For example, all of the processes of a monitor can be suspended by suspending the monitor. Also, a monitor can appear on other monitors' queues giving a system the ability to have a scheduling mechanism with no a priori fixed number of queues. In this way, monitors can be used to create arbitrarily nested queue structures -- directed graphs of queues or trees of queues, as well as multilevel agendas. Monitors are the topic of further research. We are currently focusing on applications to the SNePS [26,29] deduction system. In particular, monitors will be used to implement connectives which require deductions based on "lack of knowledge", to recognize completed subproofs, to suspend and resume processing without resorting to searches to find relevant processes and to provide a mechanism for resource limited processing.

4.3 Generators

This example demonstrates the use of MULTI to provide a "generator" facility. A generator is a function which produces results one at a time, suspending itself so that it can later resume execution where it left off. Such a package is available in INTERLISP [31] where its implementation depends on the spaghetti stack [3]. In LISP implementations which do not have a unified data structure for saving data and control environments, such a generator package is generally not available. This section presents several functions which use the notion of a MULTI process to extend such LISP systems.

There are several MULTI implementation alternatives for generators. The most primitive method involves writing MULTI process templates which behave as generators. A more sophisticated approach is to define generators as an abstract data type which allows nearly the same syntax as INTERLISP.

```
(DE  PRIMES (N)
   (PRIN3 2)  ;2 is prime.
   (MULTIP (LIST (NEW 'PRIME NIL 3 3)  ;Call MULTI evaluator
                                       ;with initial prime of 3
                 (NEW 'HALT NIL N)))
                               ;and HALT at TIME: N.
   (PRIN3 <> <>))

(DP PRIME (TIME:)
   (PRIN3 * TIME:) ;PRIME's TIME: is prime.
   (INITIATE (NEW 'BLOCK NIL TIME: (TIMES 2 TIME:)));Create and
                               ;schedule BLOCK process for
                               ;odd multiples of TIME:.
   )

(DP BLOCK (TIME: MP:)
   (IF (GREATERP (DIFF (REGFETCH (CAR EVNTS) 'TIME:) TIME:)
                 2) ;If next process' TIME: > current TIME:+2
       (INITIATE (NEW 'PRIME NIL (PLUS 2 TIME:))));then TIME:+2
                                                  ;is prime.
   (SETQ TIME: (PLUS TIME: MP:));In any case, BLOCK next odd
   (INITIATE CURNT:) ;multiple of corresponding prime.
   )

 (DP  HALT (TIME:)
   (SETQ EVNTS NIL) ) ;Make MULTI queue NIL to quit.
```

Figure 8
Process templates for Sieve of Erastosthenes.

First, consider what a process template which behaved as a generator would look like. It would take some input, apply some function to that input and produce a series of outputs, one per invocation. For example, the function LEAVES which returns the leaves of a tree when written in generator form, see Figure 9, "recursively" calls itself until a leaf is found. The leaf is then stored in the output register LEAF:. The answer produced by the generator can be accessed by using REGFETCH on the LEAF: register.

The function PRINT-LEAVES, see Figure 10, uses the LEAVES generator presented above to print the leaves of a tree. Figure 11 shows the structure built by the initial LEAVES process just after process L1 has found a leaf given the argument to PRINT-LEAVES is '((A B C) D E). The LEAVES processes are labelled for reference in order of creation. Note that this implementation requires a reference be kept to the original process' identifier so that the generator can later be resumed.

Three functions, GENERATOR, GENERATE and PRODUCE, are provided in INTERLISP which define generators. GENERATOR is a function which creates a control structure for a given generator function. (GENERATOR (LISTGEN '(A B C))) creates a spaghetti stack entry to treat the function LISTGEN as a generator. GENERATOR returns the unique internal identifier of the function instance which is the generator. No further action is taken. The function GENERATE takes as its argument a generator

```
(DP LEAVES (TREE: LEAF:)
   (REPEAT NIL
     UNTIL (ATOM TREE:)
           (IF (CDR TREE:)
               (SETQ CLINK: (NEW 'LEAVES
                                 CLINK:
                                 (CDR TREE:)
                                 'UNBOUND)))
           (SETQ TREE: (CAR TREE:)))
   (SETQ LEAF: TREE:) )
```

Figure 9
Simple form of LEAVES generator.

```
(DE PRINT-LEAVES (TREE)
  (IF TREE
   (REPEAT (G)
           (SETQ G (NEW 'LEAVES NIL TREE 'UNBOUND))
     BEGIN (MULTIP (LIST G))
           (PRINT (REGFETCH G 'LEAF:))
     WHILE (REGFETCH G 'CLINK:)
           (SETQ G (REGFETCH G 'CLINK:)))))
```

Figure 10
PRINT-LEAVES uses LEAVES as a generator.

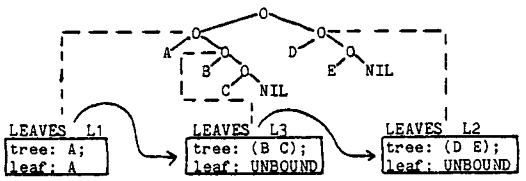

Figure 11
The LEAVES processes when L1 finds a leaf.

identifier and runs the specified generator until the first call of the function PRODUCE. PRODUCE takes a single argument which it returns as the value of the generator. For example, the INTERLISP generator version of a function which returns the top level elements of its argument one at a time could be defined as in Figure 12.

Our implementation defines generators as an abstract data type. The function DG defines a generator. In the process of defining a generator, the process template for the generator is stored under the %G% property of the generator name. Recursive calls to a generator are handled by defining a function associated with the name of the generator which creates a new generator process. For the LISTGEN example, the DG form in Figure 13a defines a process template which it stores under the %G% property of LISTGEN (see Figure 13b). DG also defines the LISP function shown in Figure 13c. (ALISP does not make the distinction between an atom's function value and LISP value. The placement of the process template on the property list is one way of avoiding this feature of ALISP.) The function GENERATOR creates a generator instance and returns a process queue of one element. The call (GENERATOR (LISTGEN '(A B C))) results in the MULTI process queue (P1) where P1 is a process with a NAME: of LISTGEN. GENERATE takes a LISP atom whose value is a process queue as input and runs the processes in the queue until the function PRODUCE is called. In this example, the MULTI execution loop is redefined in GENERATE. Thus, GENERATE takes the place of MULTIP. The reasons for this are to implement recursion by treating the MULTI process queue as a stack and to store the process template definition on the property list of the name of the process. The argument of PRODUCE is returned as the value of GENERATE and the generator variable, the argument of GENERATE, is updated to reflect the change in the process queue. For example, if the LISP atom G was assigned the result of the above call to GENERATOR then (GENERATE G) would return A with the generator variable G reassigned the new MULTI process queue.

This set of functions is minimally sufficient to define generators but one would want to add other functions to the data type. For example, a function EDITG for editing the definition of a generator without all the extra registers and forms would be useful.

```
(LISTGEN (L)
  (IF L THEN (PRODUCE (CAR L))
             (LISTGEN (CDR L))))
```

Figure 12
INTERLISP generator LISTGEN.

```
(DG LISTGEN (L)
   (IF L (PRODUCE (CAR L)) (LISTGEN (CDR L))) )
```

(a)

```
LISTGEN
PLIST
(%G%  (LAMBDA (NAME: CLINK: L)
       (IF L (PRODUCE (CAR L))
             (LISTGEN (CDR L)))
       (SET CURNT: (LIST NAME: CLINK: L)) ))
```

(b)

```
(LAMBDA %ARGS%
   (INITIATE (APPLY NEW (APPEND (LIST 'LISTGEN NIL) %ARGS%))))
```

(c)

Figure 13
(a)LISTGEN generator, (b)process template (c)and LISTGEN LISP function.

As a final example of generators, consider the "same fringe" problem -- test the equality of the fringes (leaves) of two trees. Almost all purported solutions avoid the naive LISP solution which flattens the two trees and uses the EQUAL function. Although there is some disagreement over what kinds of solutions are admissable [1,19], it is clear that the most popular approach is to make a left to right scan of the leaves of the trees terminating with failure on the first pair of leaves which disagree [1,8,11,19]. We discuss this problem not to justify generators or processes but to show a very simple solution using the MULTI generator abstract data type. Figure 14 displays the FRINGE generator and the LISP function SAME-FRINGE. Our FRINGE generator is similar in spirit to an example FRINGE generator of Hewitt's[12]. Note that a generator terminates when the generator variable (queue) becomes NIL (empty).

4.4 AND/OR Graph Processing

A simple application of AND/OR graph searching is a rewriting problem [22]. The problem is to take an initial data base and apply productions until a resultant data base consists only of terminal symbols. Productions are rewrite rules which replace an occurrence of their left hand side with the symbols of their right hand side. Some sample productions which are later used in an example are:

```
C -> D L
C -> B m
B -> m m
Z -> B B m
```

where "m" is terminal. Figure 15 shows a simple AND/OR graph. AND nodes are those nodes which have a bar across their outgoing arcs, e.g. the node labelled A in Figure 15 is an AND node. OR nodes are those nodes which are not AND nodes, e.g. B is an OR node. In the rewrite example, the AND nodes represent partial data bases and the OR nodes represent the application of a single production. What is interesting about this decomposition is that an OR node is satisfied when any one of its descendents is satisfied and an AND node is satisfied when all of its descendents are

```
(DG FRINGE (TREE)
   (COND
    ((ATOM TREE) NIL)
    ((ATOM (CAR TREE)) (PRODUCE (CAR TREE))
                       (FRINGE (CDR TREE)))
    (T (FRINGE (CAR TREE)) (FRINGE (CDR TREE)))) )

(DE SAME-FRINGE (TREE OTHERTREE)
  (REPEAT (G1 G2)
         (SETQ G1 (GENERATOR (FRINGE TREE))
               G2 (GENERATOR (FRINGE OTHERTREE)))
   BEGIN
   WHILE (EQ (GENERATE G1) (GENERATE G2))
   UNTIL (AND (NULL G1) (NULL G2))))
```

Figure 14
Generator implementation of SAME-FRINGE.

satisfied. Thus, the termination condition for an OR node is the same as the termination condition of the chronological disjunction operators mentioned in Section 1.

In addition, we need the notion of a data collector [27,28]. The purpose of a data collector is to avoid redundant computation. A data collector is a process which maintains a list of messages it has received and a set of bosses which are interested in sharing the results of the data collector's computation. Given that it is possible to identify that some data collector process is already working on a particular "problem", it is trivial to obtain the current set of results from the data collector and place another process in the list of bosses of the data collector. In the rewrite example, a "problem" is the application of a production and the productions are indexed by the symbol which is their left hand side. More sophisticated pattern matching is required for problem descriptions which are more complex [28,29].

The solution presented here uses three coroutines. The first coroutine has two parts - START and END (see Figure 16). START creates a MAPSTRING process with the same data base as passed to START and changes its NAME: to END. The END process template performs some action, here printing the resultant data base, and clears the MULTI process queue.

A second coroutine decomposes a data base so that productions can be applied to each part of the data base independently. The MAPSTRING process template (see Figure 17) tries to apply productions to each symbol in its DATABASE:. MAPSTRING could create a new APPLY-PRODUCTIONS process for each symbol in its DATABASE: but this approach propagates many redundant APPLY-PRODUCTIONS processes. Since the left hand side of a production is an atomic symbol and only one solution is required, one APPLY-PRODUCTIONS process is sufficient for any symbol. The function NEW-OLD (also in Figure 17) checks for the existence of an APPLY-PRODUCTIONS process for a given symbol. If such a process exists and a solution has already been discovered then the solution is immediately sent to the MAPSTRING process attempting to create the APPLY-PRODUCTIONS process. Also, the current process is added to the BOSSES: of the existing APPLY-PRODUCTIONS process and no new APPLY-PRODUCTIONS process is created. If no APPLY-PRODUCTIONS process exists for the symbol then a new one is created and indexed by its symbol. MAPSTRING-C (see Figure 17), the continuation of MAPSTRING, waits for the results of all production applications before sending its result to its CLINK:.

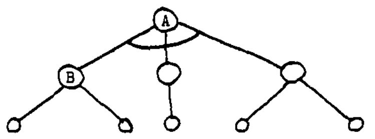

Figure 15
An example AND/OR graph.

```
(DP START (DATABASE: MESSAGE:)
   (INITIATE (NEW 'MAPSTRING CURNT: DATABASE: NIL))
   (SETQ NAME: 'END))

(DP END (DATABASE: MESSAGE:)
   (PRINT MESSAGE:)
   (SETQ EVNTS NIL))
```

Figure 16
START and END process templates.

Messages are always passed using the function SEND. SENDing a message to a process always results in the message being queued to the process' MESSAGE: register and the process being scheduled at the front of the process queue.

The process template APPLY-PRODUCTIONS (see Figure 18) is the first part of a coroutine which applies all productions whose left hand side is the same as its symbol. If an APPLY-PRODUCTIONS' symbol is terminal then no production is sought. If no productions have the symbol as a left hand side no further action will be taken. For every applicable production a new MAPSTRING process is created with its DATABASE: initialized to the right hand side of the production. APPLY-PRODUCTIONS-C (see Figure 18), the continuation of APPLY-PRODUCTIONS, passes any new MESSAGE: that it receives and keeps the list of messages it has received in its MESSAGE: register. The BOSSES: register is a list of all those processes which require the results of applying a production with a given left hand side. The APPLY-PRODUCTIONS and

```
(DP MAPSTRING (DATABASE: MESSAGE:)
   (MAPC DATABASE: (LAMBDA (C) ;For each symbol in data base
                   (NEW-OLD 'APPLY-PRODUCTIONS CURNT: C)))
                       ;create a process to try productions.
   (SETQ NAME: 'MAPSTRING-C)
   )

(DP MAPSTRING-C (DATABASE: MESSAGE:)
   (MAPC MESSAGE:
         (LAMBDA (C-S) ;For each element of message substitute
            (SETQ DATABASE: (SUBST (CDR C-S)
                                   (CAR C-S)
                                   DATABASE:))))
   (IF (NO-ATOM DATABASE:) ;Done when no toplevel atoms in
       (SEND (APPLY APPEND* DATABASE:) CLINK:));data base.
   (SETQ MESSAGE: NIL)
   )

(DE NEW-OLD (NAME CLINK SYMBOL)
   (COND
    ((GET SYMBOL NAME)       ;Get previous process id if any.
     (REGSTORE (GET SYMBOL NAME) ;Place CURNT: process into bosses
               'BOSSES:        ;of data collector.
               (CONS CLINK (REGFETCH (GET SYMBOL NAME) 'BOSSES:)))
     (SEND (CAR (REGFETCH (GET SYMBOL NAME) 'MESSAGE:)) CLINK))
      ;Send any result of data collector to CURNT: process.
    (T (INITIATE (PUTPROP SYMBOL  ;Otherwise, create and index
                          NAME  ;a new process.
                          (NEW NAME CLINK SYMBOL NIL (LIST CLINK) )))
    )))
```

Figure 17
MAPSTRING and MAPSTRING-C process templates.

```
(DP APPLY-PRODUCTIONS (SYMBOL: MESSAGE: BOSSES:)
   (COND
    ((TERMINALP SYMBOL:) ;If symbol is terminal then return it
        (SEND (LIST SYMBOL: SYMBOL:) BOSSES:));as producing itself.
    (T (MAPC (RIGHT-HAND-SIDES SYMBOL:)  ;Otherwise, for each
             (LAMBDA (RHS) ;production create a new MAPSTRING process.
                 (INITIATE (NEW 'MAPSTRING CURNT: RHS NIL))))) )
   (SETQ NAME: 'APPLY-PRODUCTIONS-C)  )


(DP APPLY-PRODUCTIONS-C (SYMBOL: MESSAGE: BOSSES:)
   (SEND (CONS SYMBOL: (CAR MESSAGE:)) BOSSES:))
```

Figure 18
APPLY-PRODUCTIONS and APPLY-PRODUCTIONS-C process templates.

APPLY-PRODUCTIONS-C process templates act as a data collector.

To demonstrate data collectors more concretely, consider the productions presented above. A snapshot of the processes created by the call (MULTIP (LIST (NEW 'START NIL '(C B Z) NIL))) after all productions have found the terminal symbol "m" appears as Figure 19. Note the arrows in this diagram represent the BOSSES: of a process which includes its CLINK:. In the figure only one APPLY-PRODUCTIONS process for "B" and "m" appear. The MAPSTRING-C process with a data base of (D L) will never send any messages.

If a production is directly or indirectly recusive, data collectors will produce a cycle in the graph of processes. Data will flow around the cycle until all solutions are produced. This is discussed in detail elsewhere [28].

5. Conclusion

MULTI is a package of LISP functions which define the notion of a process using function invocation as the only control structure primitive. Process environments are saved in what can be viewed as function call instances, i.e. LISP forms which have the name of a process template in functional position and the register values following it. We have demonstrated the flexibility of this simple conceptualization of processes by discussing several examples which use various control structures such as recursion, backtracking, generators, coroutines and AND/OR processing. Also, this implemetation does not assume that the host LISP system provides any control or data environment saving mechanisms such as INTERLISP's spaghetti stack or FUNARG. Thus, MULTI should be portable to other LISP implementations modulo some error conditions and error recovery functions. In fact, MULTI, as part of the SNePS deduction system [26], has moved from LISP 1.6 [24] to UTLISP [10] and then to ALISP [16]. The LISP source for MULTI and the other functions discussed here are presented in [21].

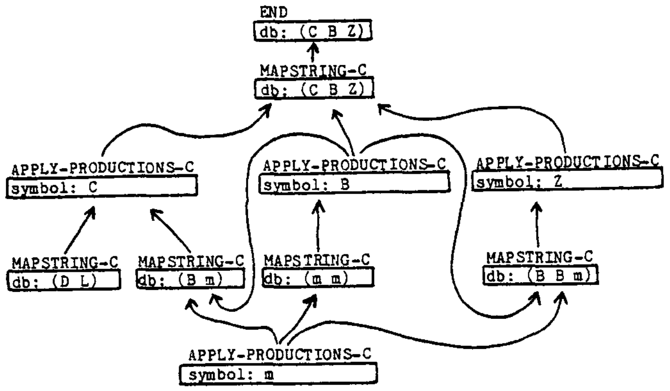

Figure 19
Snapshot of rewrite processes.

References

1. Anderson, B. The samefringe problem. SIGART Newsletter, No. 60, November, 1976, 4.
2. Baker, H.G. and Hewitt, C. The incremental garbage collection of processes. Proc. of the Symposium on Artificial Intelligence and Programming Languages. SIGART Newsletter, No. 64, August, 1977, 55-59.
3. Bobrow, D. G. and Wegbreit, B. A model and stack implementation of multiple environments. CACM 16(10), October, 1973, 591-603.
4. Bobrow, D.G. and Winograd, T. An overview of KRL, a knowledge representation language. Cognitive Science 1, 1 (January 1977), 3-46.
5. Bobrow, D.G. and Winograd, T. Experience with KRL-0, one cycle of a knowledge representation language. Proc. Fifth International Joint Conference on Artificial Intelligence, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, 1977, 213-222.
6. Dahl, O.-J. and Hoare, C. A. R. Hierarchical program structures. In Structured Programming (Dahl, O.-J.; Dijkstra, E.W.; Hoare C.A.R.), Academic Press, 1972, 175-200.
7. Fikes, R.E. Deductive retrieval mechanisms for state description models. Proc. Fourth International Joint Conference on Artificial Intelligence, MIT AI Laboratory, Cambridge, MA, 1975, 99-106.
8. Finin, T. and Rutter, P. Different fringe for different folk. SIGART Newsletter, No. 60, November, 1976, 4-5.
9. Friedman, D.P. and Wise, D.S. Applicative multiprogramming. Technical Report No. 72, Computer Science Dept., Indiana University, Bloomington, IN, 1977.
10. Greenwalt, E.M., Slocum, J. and Amsler, R.A. U. T. LISP Documentation. Computation Center, University of Texas at Austin, May, 1975.
11. Greussay, P. An iterative LISP solution to the samefringe problem. SIGART Newsletter, No. 59, August, 1976, 14.
12. Hewitt, C.E. Viewing control structures as patterns of passing messages. Artificial Intelligence Vol. 8(3), 1977, 323-364.
13. Hewitt, C.E. and Smith, B. Towards a programming apprentice. IEEE Transactions on Software Engineering SE-1(1), March, 1975, 26-45.
14. Jensen, K. and Wirth, N. PASCAL User Manual and Report. Lecture Notes in Computer Science Vol. 18, Goos, G. and Hartmanis, J. eds., Springer-Verlag, New York, 1974.
15. Kay, A. SMALLTALK, a communication medium for children of all ages, Xerox Palo Alto Research Center, Palo Alto, CA, 1974.
16. Konolige, K. ALISP User's Manual. University of Massachusetts Computing Center, Graduate Research Center, Amherst, MA, 1975.
17. Kupiers, B.J. A frame for frames: representing knowledge for recognition. In Representation and Understanding. Bobrow, D.G. and Collins, A.(eds.), Academic Press, 1975, 151-184.
18. Marti, J.B., et al. Standard LISP Report. SIGPLAN Notices 14(10), October, 1979 44-68. Also appeared as Technical Report No. 101, Computer Science Dept., University of Utah, Salt Lake City, UT, January, 1978.
19. McCarthy, J. Another samefringe. SIGART

Newsletter, No. 61, February, 1977, 4.

20. McCarthy, J., et al. LISP 1.5 Programmer's Manual. The M.I.T. Press. Massachusetts Institute of Technology, Cambridge, Massachussets, 1965.

21. McKay, D.P. and Shapiro, S.C. MULTI - A LISP base multiprocessing system. Technical Report, Department of Computer Science, SUNY/Buffalo, 1980.

22. Nilsson, N.J. Principles of Artificial Intelligence. Tioga Publishing Co., Palo Alto, California. 1980.

23. Pratt, T. W. Programming Languages: Design and Implementation. Prentice-Hall Englewood Cliffs, N.J.. 1975.

24. Quam, L.H. LISP 1.6. Stanford Artificial Intelligence Laboratory, Stanford University, September, 1969.

25. Shapiro, S.C. Techniques of Artificial Intelligence. D. Van Nostrand Company, New York, 1979.

26. Shapiro, S.C. The SNePS semantic network processing system. In Associative Networks: The Representation and Use of Knowledge by Computers, Findler, N.V., ed., Academic Press, New York. 1979, 179-203.

27. Shapiro, S.C. Compiling deduction rules from a semantic neetwork into a set of processes. Workshop on Automatic Deduction, MIT, Cambridge, MA. August 17-19, 1977.

28. Shapiro, S.C. and McKay, D.P. Inference with recursive rules. Proc. First Annual National Conference on Artificial Intelligence, AAAI, Stanford University, 1980.

29. Shapiro, S.C. and McKay, D.P. The representation and use of deduction rules in a semantic network, forthcoming.

30. SIMULA Version 1 Reference Manual. Control Data Corporation, publication number 60234800 Rev A, St. Paul, Minnesota, E-1.

31. Teitelman, W., et al. INTERLISP Reference Manual. Xerox Palo Alto Res. Ctr., Palo Alto, Calif., 1974.

32. Wand, M. The frame model of computation. Technical Report No. 20, Computer Science Dept., Indiana University, Bloomington, IN. December, 1974.

33. Wirth, N. Program development by stepwise refinement. CACM 14, 221-227.

34. Wise, D.S., Friedman, D.P., Shapiro, S.C. and Wand, M. Boolean-valued loops. BIT 15, 431-451.