3) Create a new unattached frame.

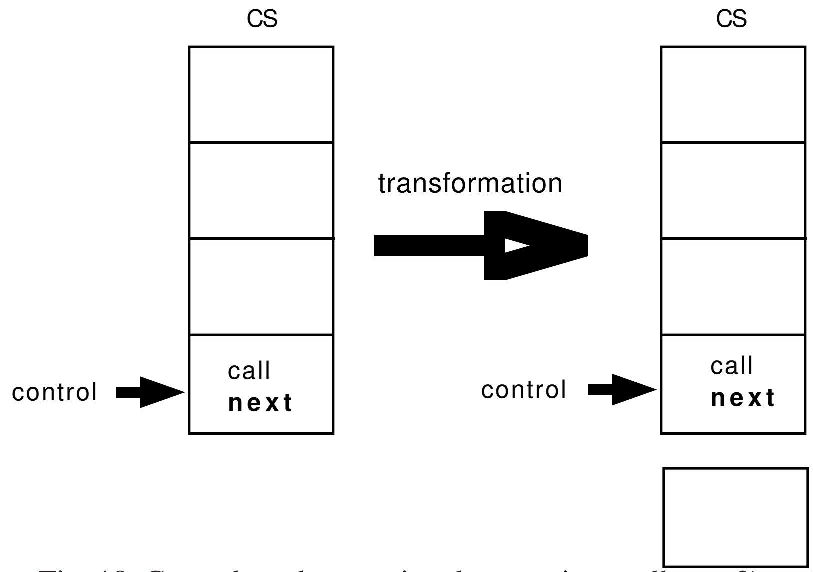


Fig. 18. Control stack operational semantics - call step 3).

4) Transfer control to the newly created unattached frame.

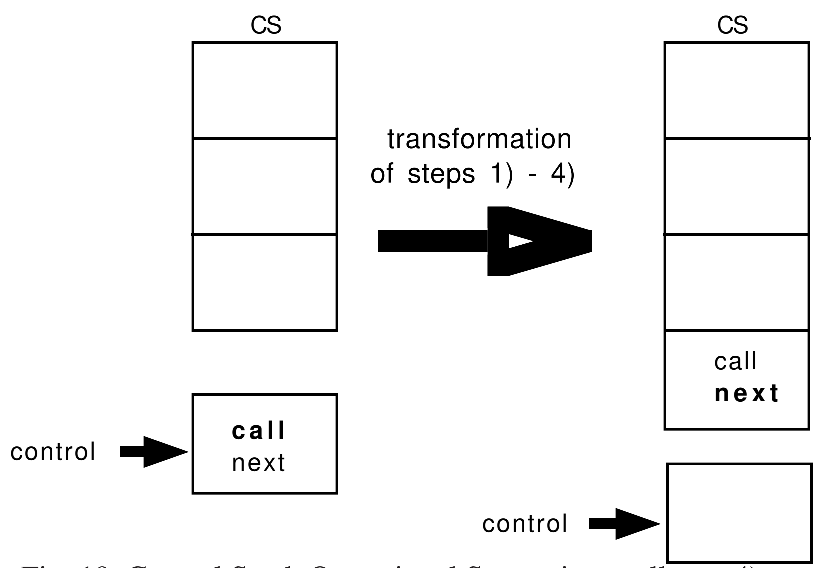


Fig. 19. Control Stack Operational Semantics - call step 4).

Return

1) Place return value in calling function. Through out all computation there are numerous values that are stored temporarily including return values, they are not stored as locals but more as registers. The return value exists in this form on both sides of the return, both in the function returned from and returned to. How this value is dealt with is highly implementation dependent. In the below diagram we include the return value as part of the position indicator, partially resembling how a stack machine might store the return value.

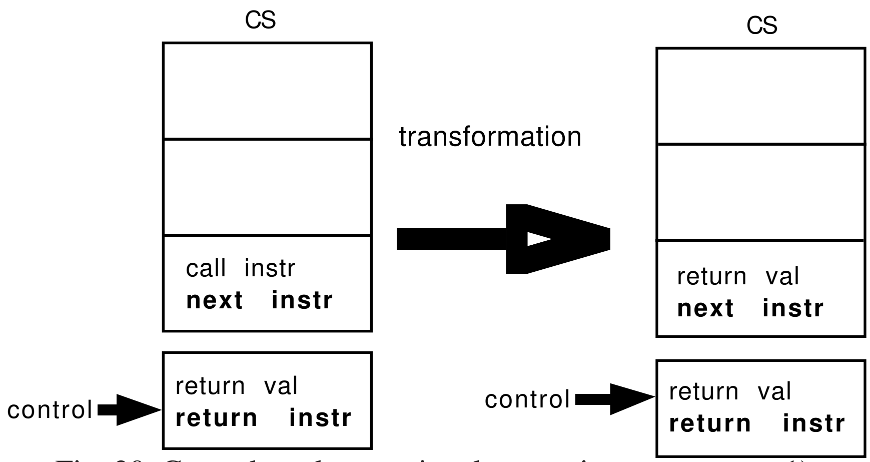


Fig. 20. Control stack operational semantics - return step 1).

2) Remove unattached frame. Turn the top frame of the control stack into an unattached frame and place control on it.

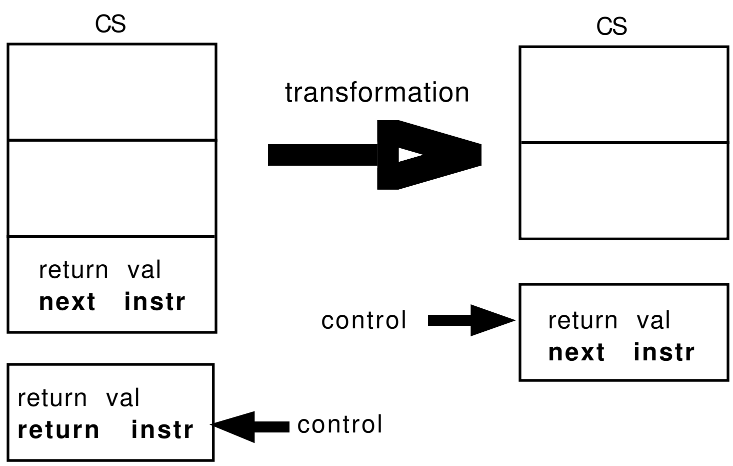


Fig. 21. Control stack operational semantics - return step 2).

2.3 TRACE

Let us use the function q-ord defined in figure 3 where we define a function that traverses a tree with two children for every nonleaf node, printing out each element. There are a number of markers placed in a revision of Figure 3 shown in Figure 22. These are not actual pieces of code, they are markers that we will be using in a control flow trace. There are two markers for each defq call (we use the term queue function to refer to a structure defined by defq). One marker each for each assembly level instruction. Pos1 and pos3 are call1's, and pos2 and pos4 are call2's.

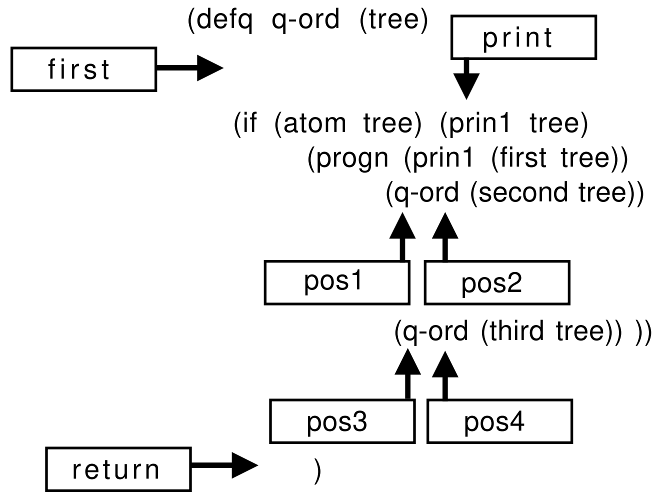
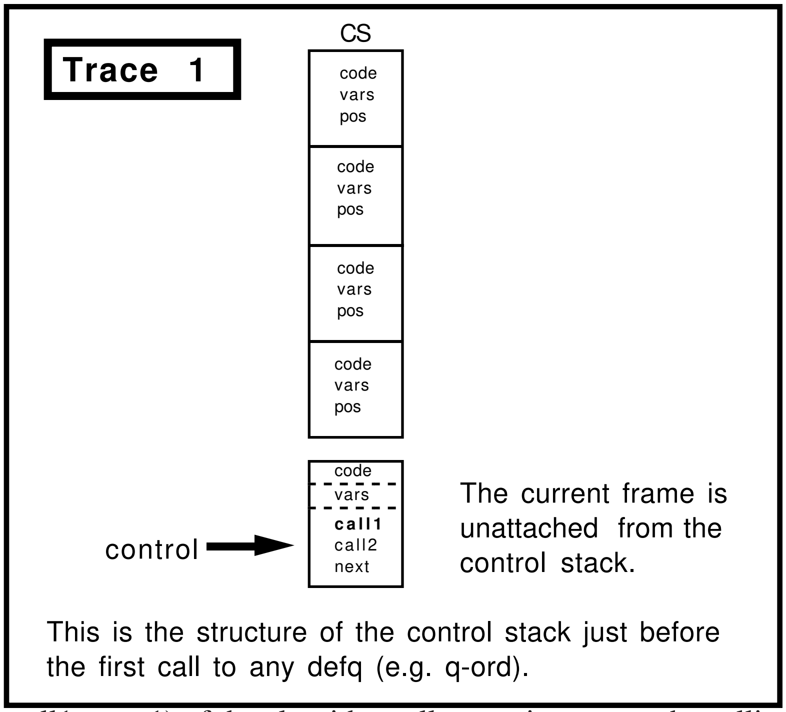


Fig. 22. A queue function for traversing a tree breadth-first.

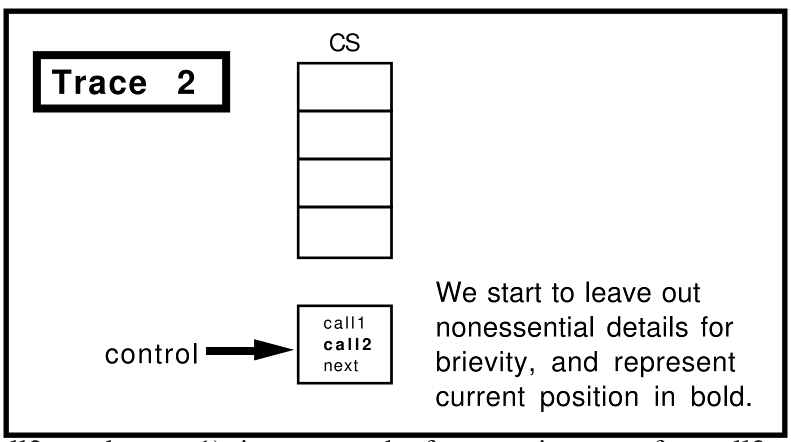
In this example we are assuming that "fun2" is called from some unspecified function. The control stack has five frames thus the function calling fun2 has been called itself as have been the other frames.

Each of the steps of the above algorithm will be described in more detail as we go through a trace. The following eleven trace diagrams trace the q-ord defq call of Figure 3, which does a breadth-first traversal of the tree of Figure 2.

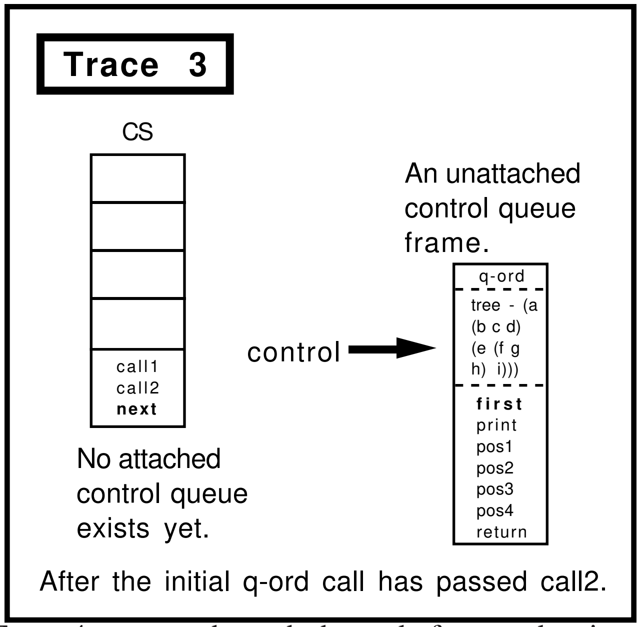
Trace 1 includes a typical diagram for a control stack. Note that the current frame is not on the stack, the frame that is currently active is usually stored in current variables of the interpreter, this will not always be the case with the control queue. This is the first call of q-ord deep in a programs control flow. Note that in the diagram we are nested in four unidentified function calls.



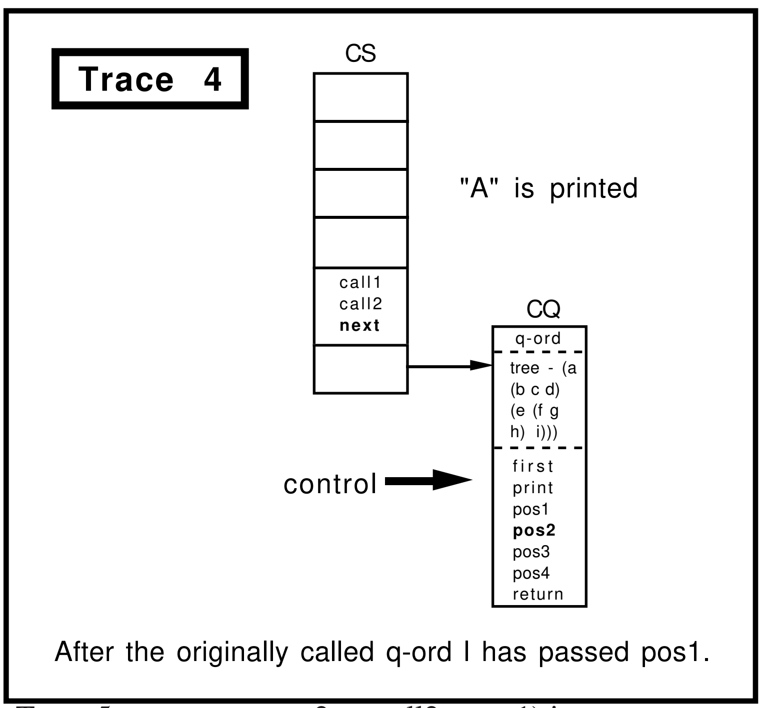
We execute call1, step 1) of the algorithm tells us to increment the calling frame pointer to call2; step 3) control returns to the stack frame. This is our status in Trace 2.



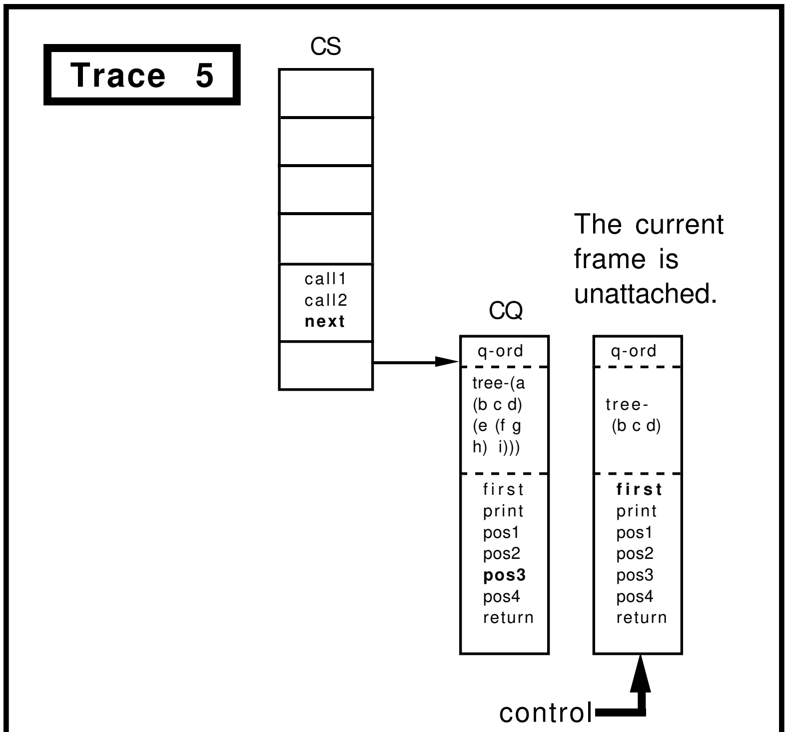
Executing call2 we do step 1) increment the frame pointer to after call2; step 2) add the stack frame to the control stack; step 3) create a control queue frame for the fun2 call on the initial bitree and transfer control to this new control queue frame reaching Trace 3.



Before reaching Trace 4 we step through the code for q-ord, print out the top node "A" and advance to pos1 which is a call1, so we step 1) increment the frame pointer to pos2; step 2) attach the current frame on the back of the control queue; and step 3) transfer control to the front of the control queue.

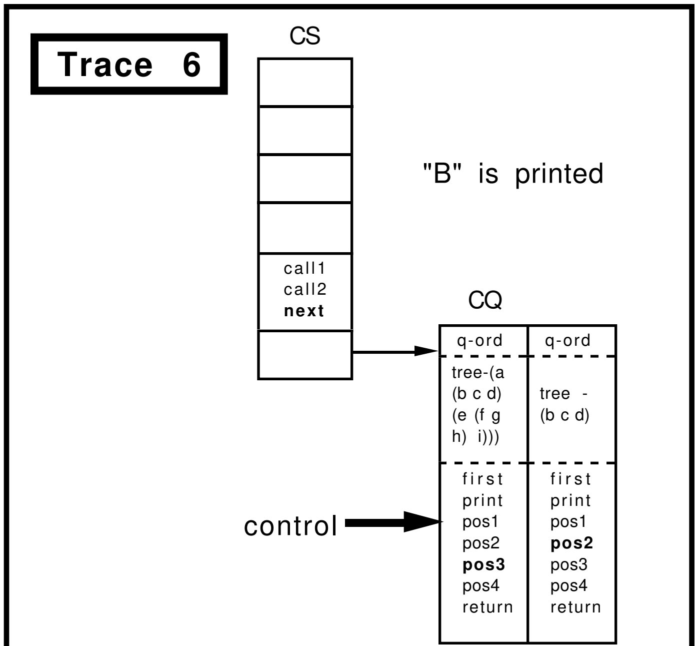


Advancing to Trace 5 we execute pos2 - a call2: step 1) increment to pos3; step 3) create a control queue frame for the left of the initial tree and set control to the new frame.



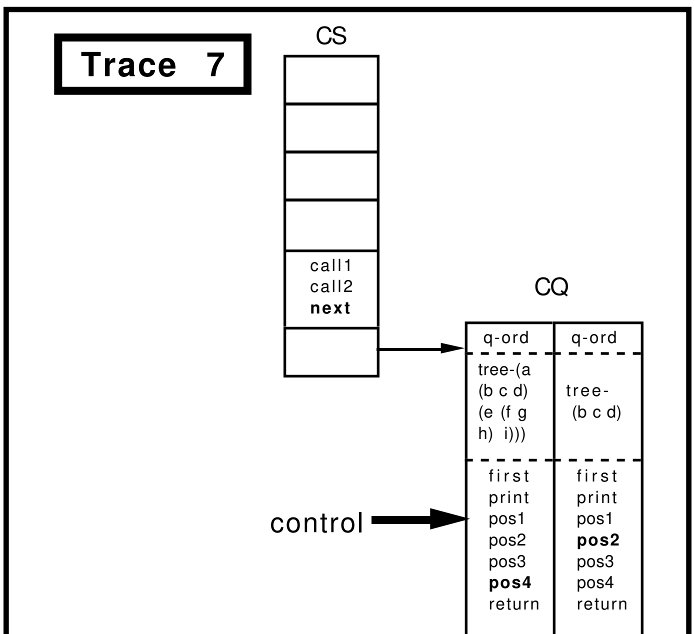
Position pos2 is passed and a queue call is made on the left branch.

To reach Trace 6 we advance the position pointer printing "B" and reach pos1 of the q-rod call on the left branch, there we execute pos 1 which is a call1: step 1) increment to pos2; step 2) place the left tree control queue frame on the back of the control queue; and step 3) move control to the front of the queue.



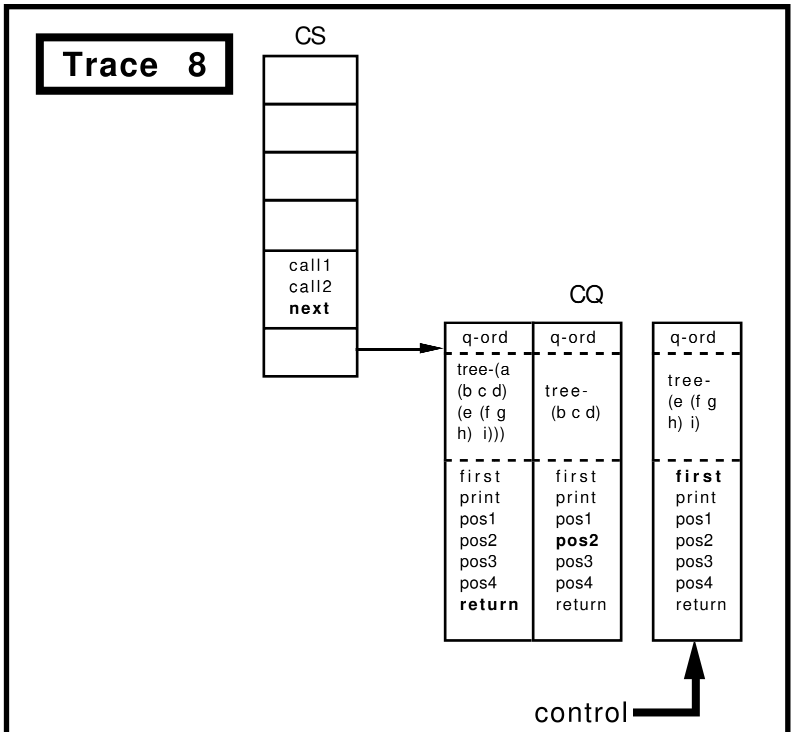
The left branch passes pos1 and control is passed to the front of the control queue.

We reach Trace 7 by executing pos3 of the first control queue frame, pos3 is a call1: step 1) advance to pos4; step 3) set control to the front of the control queue.



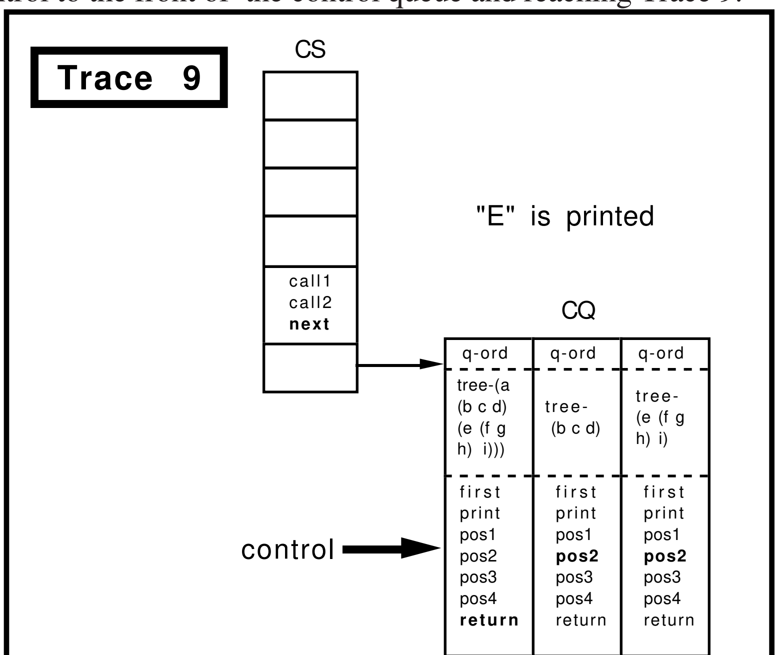
The original q-ord call passes pos3 and control is passed to the front of the control queue.

We then execute pos4 which is a call2: step 1) advance to return; step 3) create a control queue frame for the q-ord call on the right branch and set control to this new frame. This is what is diagramed in Trace 8.



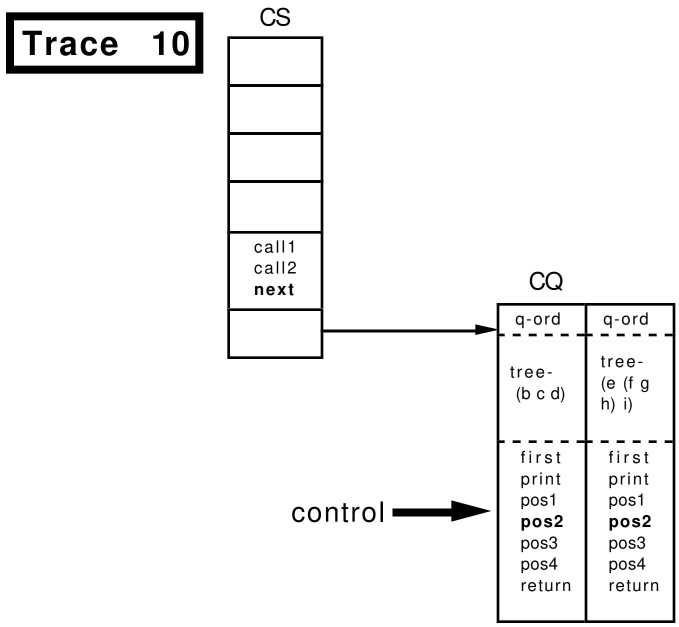
Pos4 is passed and q-ord call on the right branch is made.

We advance through first and print "E", then on to pos1 of the new control queue frame which is a call1: step1) advance to pos2; step 2) attach the frame to the control queue step 3) transfer control to the front of the control queue and reaching Trace 9.



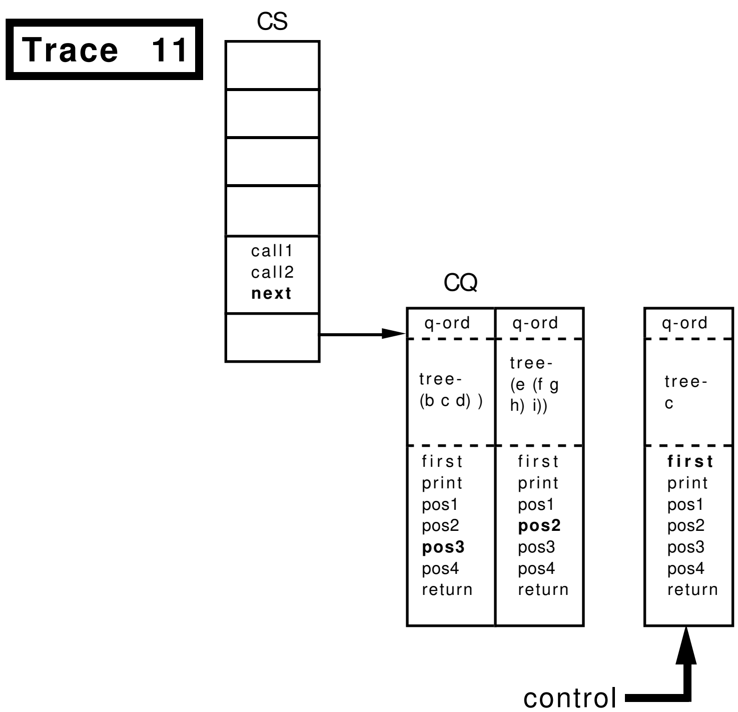
The right branch passes pos1 and control is passed to the front.

Executing return step 1) remove the first queue frame and pass control to the new front of the control queue we reach the state indicated by Trace 10.



The initial q-ord frame passes return and is removed.

Pos2 is a call2 so: step 1) the frame pointer is incremented to pos3; step 3) we create a new queue frame for the left left branch as is described in Trace 11.



Pos2 is passed and q-ord call on the left left branch is made.

The trace will continue in a similar fashion and it would be redundant to show it here. The control queue mechanism described by the above algorithm and trace behaves like the usage of a queue in doing a breadth-first traversal. This can be seen by looking at the flow of control. For each queue-function call its node will be printed out before its children will be called. When the children are called they will print out their nodes and halt in the middle of calls to the grandchildren. The node printing of the children will be done in the order they are called and will be done before any grandchildren calls are completed. Each child will go on to call their grandchildren in order. Likewise all great grandchildren and other ancestors will be traversed in order.

2.4 Comparison with Defined Function

In this subsection we first demonstrate that control queues can be used for a variety of different breadth-first traversal problems. We also compare it with a defined normal function as a general breadth-first traversal tool.

Our next example demonstrates that normal function calls and defq function calls can be intermixed. This function is equivalent to the function of Figures 3 and 22, but is defined differently. This function has an auxiliary regularly defined function called "branch" which handles the case of a nonleaf node:

```
qlisp> (defq qorder (tree) (if (endp tree) (prin1 tree)
    (branch tree)))
QORDER

qlisp> (defun branch (tree)
        (prin1 (first tree))
        (qorder (second tree))
        (qorder (third tree)))
BRANCH

qlisp> (qorder (quote (a (b c d) (e (f g h) i))))
ABECDFIGHNIL
```

We next show that the control queue mechanism can be used to do depth-first traversal on different types of trees which use different internal formats. This is essential if control queues are to be considered a general mechanism. It is also important from the point of view of comparison. If control queues or a defined function are to compete as general mechanisms they must both be able to handle breadth-first traversal on any type of structure given as input. We start by showing that control queues can handle different types of trees. Our first example handles trinary instead of binary trees. Terminal trees also have a different format than before they are lists with only one element, the leaf element. We include numerous auxiliary functions to further demonstrate control queues integration with normal functions.

```
qlisp> (defq fun2 (tritree)
  (printnode tritree)
  (unless (leaf tritree)
        (fun2 (left tritree))
        (fun2 (middle tritree))
        (fun2 (right tritree))))
FUN2

qlisp> (defun printnode (tree) (prin1 (first tree)))
```

```
PRINTNODE
qlisp> (defun leaf (tree) (= (length tree) 1))
LEAF
qlisp> (defun left (tree) (second tree))
LEFT
qlisp> (defun middle (tree) (third tree))
MIDDLE
qlisp> (defun right (tree) (fourth tree))
RIGHT

qlisp> (fun2 (quote (a (b (c) (d) (e)) (f (g) (h) (i)) (j (k) (l) (m)) )))
ABFJCDEGHIKLMNIL
```

We now show a control queue that can handle a tree with any number of branches. We return to the format of our initial examples.

```
qlisp> (defq traversetree (tree)
   (if (symbolp tree) (prin1 tree)
      (progn (prin1 (first tree))
           (recurseonchildren (rest tree)))))
TRAVERSETREE

qlisp> (defun recurseonchildren (children)
   (unless (null children)
         (traversetree (first children))
         (recurseonchildren (rest children))))
RECURSEONCHILDREN

qlisp> (traversetree (quote (a (b c d e f) (g h i j))))
ABGCDEFHIJNIL
```

What if a different structure is to be traversed. What if we need to traverse a graph instead of a tree. In the next example a directional graph is represented by a list of nodes paired with a list of all of the nodes each node is connected to. We use a global variable "visited" to keep track of all of the nodes visited so far.

```
qlisp> (setf graph (quote ((1 ( 2 3))
                 (2 (1 4 5))
                 (3 (1 6 7))
                 (4 (2 8))
                 (5 (2 8))
                 (6 (3 8))
                 (7 (3 8))
                 (8 (4 5 6 7)))) )
((1 (2 3)) (2 (1 4 5)) (3 (1 6 7)) (4 (2 8)) (5 (2 8)) (6 (3 8)) (7 (3 8)) (8 (4 5 6 7)))
qlisp> (setf visited nil)
NIL
qlisp> (defq graphtraverse (node)
  (unless (member node visited)
         (print node)
         (setf visited (cons node visited))
         (traversechildren (second (assoc node graph))) ))
GRAPHTRAVERSE
qlisp> (defun traversechildren (children)
```

```
   (unless (null children)
             (graphtraverse (first children))
             (traversechildren (rest children)) ))
TRAVERSECHILDREN
qlisp> (graphtraverse 1)
1
2
3
4
5
6
7
8
NIL
```

In order for a general defined function to handle different formats of input structures, it must have some way of handling this problem, it can not guess or presume the inputs it will receive. To deal with this problem the function must have auxiliary functions as arguments that allow it to pull apart the input structure and operate on it. We introduce the defined function bfs which does this. There must be five additional auxiliary functions: one to test if there are children of the current node; one to perform a task on a nonleaf node; one to access a nonleaf node so the task can be performed; one to test if a node is a leaf or not; and one to perform the task on a leaf node.

```
(defun bfs (tree &key
          (children #'rest)
          (performtask #'prin1)
          (accessnode #'first)
          (leaf #'symbolp)
          (performleaf #'prin1))
   (setf queue ())
   (bfs-aux tree children performtask accessnode leaf performleaf))

(defun bfs-aux (tree children performtask accessnode leaf performleaf)
   (cond ((funcall leaf tree) (funcall performleaf tree))
       (t (funcall performtask (funcall accessnode tree))
            (setf queue (append queue (funcall children tree))) ))
   (unless (null queue) (bfs-aux (pop queue)
                             children
                             performtask
                             accessnode
                             leaf
                             performleaf)))
```

Using bfs we can now mimic the tasks we accomplished with control queues. First we handle a tree of any size using the standard format.

```
> (bfs '(a (b c d e) (g h i)))
ABGCDEHI
NIL
```

Next we look at trinary trees and a different representation of trees.

```
> (defun leaf (tree) (= (length tree) 1))
```

```
LEAF
> (defun printnode (tree) (prin1 (first tree)))
PRINTNODE
> (bfs (quote (a (b (c) (d) (e)) (f (g) (h) (i)) (j (k) (l) (m)) ))
                              :leaf #'leaf
                              :performleaf #'printnode)
ABFJCDEGHIKLM
NIL
```

Finally we handle the graph problem.

```
(setf graph '((1 ( 2 3))
        (2 (1 4 5))
        (3 (1 6 7))
        (4 (2 8))
        (5 (2 8))
        (6 (3 8))
        (7 (3 8))
        (8 (4 5 6 7)) ))

(setf visited nil)

(bfs 1 :accessnode
         #'(lambda (node) node)
         :performtask
         #'(lambda (node)
                  (setf visited (cons node visited))
                  (print node))
         :children
         #'(lambda (node) (second (assoc node graph)))
         :leaf
         #'(lambda (node) (member node visited))
         :performleaf
         #'(lambda (node)) )
```

None of these has given us a problem. However each of them is nonintuitive. The programmer using bfs has to have a special knowledge of its internal workings. The programmer has to have so much knowledge of its internal workings that he might as well as have written his own function to do the breadth-first traversal. With control queues there is no special knowledge that has to be known besides the underlying operational semantics. Likewise the operational semantics of normal function calling has to be known when programming. Operational semantics of normal function calling are usually kept in the back of the mind unless unusual cases come up, the same is true with control queues. All that must be done is to picture what will happen with normal function calling then realize that a breadth-first instead of depth-first traversal will occur if a defq is used instead of a defun.

We could ask why have we not had to create a defined function to do general depth-first traversal. We could easily define a function dfs to do the task. However it is unnecessary since normal function calling implicitly implements this form of traversal. It is an accident that the assumed operational semantics for function calling capture this. There is no built in necessity that the operational semantics that are attached to procedure-function calling syntax must capture depth-first traversal. We are providing control queues so that breadth-first traversal can be implemented implicitly in procedure-function calling using different operational semantics. This avoids the necessity of having a bfs.

Our next to last example takes several arguments instead of one. Previously we assumed that there would only be one structure to be traversed. What if more values need to be passed to the function than the structures that need to be traversed. We also pass additional arguments in this example. We pass two structures that must be traversed breadth-first simultaneously along with an additional argument that will be used in the traversal.

```
qlisp> (defq bfswithvars (tree1 tree2 var3)
  (if (symbolp tree1)
     (progn (prin1 tree1) (prin1 tree2) (prin1 var3) (terpri))
     (progn (prin1 (first tree1))
          (prin1 (first tree2))
          (prin1 var3)
          (terpri)
          (bfsrecursechild (rest tree1) (rest tree2) var3)
     )))
BFSWITHVARS

qlisp> (defun bfsrecursechild (children1 children2 var3)
  (unless (null children1)
        (bfswithvars (first children1) (first children2) var3)
        (bfsrecursechild (rest children1) (rest children2) var3)))
BFSRECURSECHILD

qlisp> (bfswithvars (quote (a (b c (d e f) g) h i (j k l) m))
         (quote (z (y x (w v u) t) s r (q p o) n))
         (quote (hi there)))AZ(HI THERE)
BY(HI THERE)
HS(HI THERE)
IR(HI THERE)
JQ(HI THERE)
MN(HI THERE)
CX(HI THERE)
DW(HI THERE)
GT(HI THERE)
KP(HI THERE)
LO(HI THERE)
EV(HI THERE)
FU(HI THERE)
NIL
```

Of course we can modify bfs so that it handles this problem also. But the special knowledge that must be known by a programmer using this function becomes greater. A programmer has to know how the five auxiliary functions will be integrated. In particular s/he must know that the first argument that they provide must be several arguments bound together into a list, as well as knowing that each auxiliary function must know how to pull that first argument apart into several different arguments. Only in this way can bfs handle several different input values.

```
(defun bfs (tree &key
        (children #'(lambda (list) (list (rest (first list)))))
        (performtask #'(lambda (node) (prin1 node)))
        (accessnode #'(lambda (list) (first (first list))))
        (leaf #'(lambda (list) (symbolp (first list))))
        (performleaf #'(lambda (list) (prin1 (first list)))) )
```

```
  (setf queue ())
  (bfs-aux tree children performtask accessnode leaf performleaf))

(defun bfs-aux (tree children performtask accessnode leaf performleaf)
  (cond ((funcall leaf tree) (funcall performleaf tree))
        (t (funcall performtask (funcall accessnode tree))
        (setf queue (append queue (funcall children tree))) ))
  (unless (null queue) (bfs-aux (pop queue)
                                children
                                performtask
                                accessnode
                                  leaf
                                performleaf)) )
```

The call to this new bfs is definitely nonintuitive:

```
(bfs    (list (quote (a (b c (d e f) g) h i (j k l) m))
         (quote (z (y x (w v u) t) s r (q p o) n))
         (quote (hi there)))
  :accessnode
  #'(lambda (list)
          (list (first (first list))
                (first (second list))
                (third list)))
  :performtask
  #'(lambda (list)
          (prin1 (first list))
          (prin1 (second list))
          (prin1 (third list)))
  :performleaf
  #'(lambda (list)
          (prin1 (first list))
          (prin1 (second list))
          (prin1 (third list)))
  :children
  #'(lambda (list)
         (mapcar #'(lambda (A B) (list A B (third list)))
         (rest (first list)) (rest (second list))) ))
```

In our final example we ask if control queues can be nested within each other? The answer is yes, some of the finer points of the implementation of this are shown in Section 7. What if the control queues have the same name, can they still be nested? Then the first answer to this is no, but with an additional tool called grab it becomes yes. For if a control second queue call is done on the name of a first control queue call, then they are really the same control queue. In the implementation of control queues, control queues identify the frames they will place themselves next to by name only. Control stacks are forced into a specific location - the top of the stack, whereas control queue frames will be located next to frames of the same name.

Using grab the answer to the second question becomes yes. "For what is in a name, a rose by any other name is still a rose." The whole motivation behind using a nested function is to avoid rewriting the procedure-function a second time. We can avoid this.

In this next example we will be traversing a tree depth first. For each node of the tree traversed an action will be performed. The action performed will likewise be the traversal of a tree breadth-first. We will be using the same function to traverse both the outside tree and the trees inside of each node. We will do this by attaching a new name to the same function. We start by creating a breadth-first traversal function called treeintree. We then copy that function attaching a new name treeit2. We use the control tools grab and code to do the recursive call. For if we put a specific name in the recursive call, then both functions would have the same recursive call and would be the same control queue. The tools grab and code are described in detail in Section 3. The way these two tools are used here they return the function that they are called from and act exactly as a recursive call would. For more information consult Section 3.

```
qlisp> (defq treeintree (tree performtask)
   (if (symbolp tree)
        (funcall performtask tree)
        (progn (funcall performtask (first tree))
               (recurseonchildren (rest tree)
                                  performtask
                                  (code (grab 0))))))   ; recursively calls the function that they are
                                                         ; contained in, either treeintree or treeit2
TREEINTREE

qlisp> (defun recurseonchildren (children performtask cqf)
    (unless (null children)
           (funcall cqf (first children) performtask)
           (recurseonchildren (rest children) performtask cqf) ))
RECURSEONCHILDREN

qlisp> (defun gettree (treesymbol)
    (eval treesymbol))
GETTREE

qlisp> (setf treeit2 (function treeintree))
(QUEUE (TREE PERFORMTASK) (IF (SYMBOLP TREE) (FUNCALL
PERFORMTASK TREE) (PROGN (FUNCALL PERFORMTASK (FIRST
TREE)) (RECURSEONCHILDREN (REST TREE) PERFORMTASK (CODE
(GRAB 0))))))
```

The subtree contained within each node will be attached to a global variable of the same name as the node.

```
qlisp> (setf A (quote (aa (ab ac ad) (ae af) (ag ah (ai aj) ak) (al am an)) ))
(AA (AB AC AD) (AE AF) (AG AH (AI AJ) AK) (AL AM AN))
qlisp> (setf B (quote (ba (bb bc (bd (be bf bg)) bh) bi (bj (bk bl))) ))
(BA (BB BC (BD (BE BF BG)) BH) BI (BJ (BK BL)))
qlisp> (setf C (quote (ca (cb (cc (cd ce cf cg) ch ci) (cj ck)) (cl (cm cn co))) ))
(CA (CB (CC (CD CE CF CG) CH CI) (CJ CK)) (CL (CM CN CO)))
qlisp> (setf D (quote (da db dc dd (de df dg) (dh di dj) dk dl) ))
(DA DB DC DD (DE DF DG) (DH DI DJ) DK DL)
qlisp> (setf E (quote (ea eb (ec ed ee) ef (eg eh ei) ej ek) ))
(EA EB (EC ED EE) EF (EG EH EI) EJ EK)
qlisp> (setf F (quote (fa (fb fc fd) (fe ff (fg fh fi) fj) fk) ))
(FA (FB FC FD) (FE FF (FG FH FI) FJ) FK)
qlisp> (setf G (quote (ga gb gc (gd ge gf) gh gi gj) ))
(GA GB GC (GD GE GF) GH GI GJ)
qlisp> (setf H (quote (ha (hb (hc hd) he hf) hg hh hi) ))
```

```
(HA (HB (HC HD) HE HF) HG HH HI)
qlisp> (setf I (quote (ia (ib ic id ie (if ig) ih) ij ik il) ))
(IA (IB IC ID IE (IF IG) IH) IJ IK IL)
qlisp> (setf J (quote (ja jb jc jd (je jf jg) jh ji) ))
(JA JB JC JD (JE JF JG) JH JI)
qlisp>  (setf tree (quote (A (B (C D)) E F (G H (I J))) ))
(A (B (C D)) E F (G H (I J)))
```

The call of the outer control queue takes a second argument which indicates the task that must be accomplished for each node. The task includes printing the name of the node plus the inner nested call that must be done on each node. The inner nested call itself must also take two arguments, the first being the tree it is to traverse (gettree tree), and the second the action it must take - a prin1.

```
qlisp> (treeintree tree
         (function (lambda (tree)
                      (print tree)
                      (treeit2 (gettree tree)
                                (function prin1)
                      ))))
A AAABAEAGALACADAFAHAIAKAMANAJ
B BABBBIBJBCBDBHBKBEBLBFBG
E EAEBECEFEGEJEKEDEEEHEI
F FAFBFEFKFCFDFFFGFJFHFI
G GAGBGCGDGHGIGJGEGF
C CACBCLCCCJCMCDCHCICKCNCOCECFCG
H HAHBHGHHHIHCHEHFHD
I IAIBIJIKILICIDIEIFIHIG
D DADBDCDDDEDHDKDLDFDGDIDJ
J JAJBJCJDJEJHJIJFJG
NIL
```

The defined bfs function must be further refined in order to handle this new example. The big change being and auxiliary argument "queue" which will explicitly keep track of the "frames" that have been called or need to be called. In essence the difference between bfs and control queues is that bfs explicitly keeps track of control flow while control queues handle control flow implicitly.

```
(defun bfs (tree &key
         (children #'(lambda (list) (mapcar #'(lambda (A) (list A))
                                 (rest (first list)) )))
         (performtask #'(lambda (node) (prin1 node)))
         (accessnode #'(lambda (list) (first (first list))))
         (leaf #'(lambda (list) (symbolp (first list))))
         (performleaf #'(lambda (list) (prin1 (first list))))
    &aux queue)
    (declare (special queue))

(defun bfs-aux (tree children performtask accessnode leaf performleaf)
    (declare (special queue))
    (cond ((funcall leaf tree) (funcall performleaf tree))
         (t (funcall performtask (funcall accessnode tree))
               (setf queue (append queue (funcall children tree))) ))
    (unless (null queue) (bfs-aux (pop queue)
```

```
                              children
                              performtask
                              accessnode
                              leaf
                              performleaf)) )
  (bfs-aux tree children performtask accessnode leaf performleaf))
```

It appears that each time a more difficult breadth-first traversal problem comes up this function has to be revised. There is no guarantee that even this final version will suffice. Whereas control queues and its associated control tools have be able to handle all of the examples.

With the function call we are luckier this time, the call is simpler, however we still need to have a great deal of special knowledge of how bfs works internally to get it right.

```
(bfs (list tree) :performtask
   #'(lambda (node) (print node)
                    (bfs (list (gettree node)))
                )
   :performleaf
   #'(lambda (list) (print (first list))
                     (bfs (list (gettree (first list))))
          ))
```

2.5 Converttoq

The converttoq command will change a control stack into a control queue. This command will break the control stack into control queue frames by searching for all the frames in the control stack with the appropriate queue function name.

The converttoq instruction does more than just change the values in the control structure it actually changes the structure itself and the interpreter algorithm that manipulates that structure. Demonstrating a tremendous amount of reflective power. The algorithm for control stacks is no longer used. The algorithm for control queues takes over. This causes a switch from a depth-first traversal to a breadth-first traversal. Yet the same control flow information that is contained in the original control structure is used. Thus a meaningful translation from one control structure to another is realized.

What might such a tool be used for? Let's say we wanted to go through a list from front to back twice in a row. Depth-first traversal will start out by traversing a list from front to back, but on returning it will traverse the list from back to front. If we switch to control queues in midstream then we can realize a second traversal from front to back. The following function traverses a list twice from front to back.

```
qlisp>(defun print12 (list)
  (if (null list)
     (converttoq print12)
     (progn (print (first (first list)))
          (print12 (rest list))
          (print (second (first list))))))
PRINT12
qlisp> (print12 (quote ((a 1) (b 2) (c 3))))
A
B
C
1
```