

## A New Fast and Safe Marking Algorithm\*

Toshiaki Kurokawa

Information Systems Lab., TOSHIBA R & D Center

Kawasaki, Japan

### Abstract:

A new marking algorithm for garbage collection is presented. The method is a variation of the usual simple stacking algorithm. In practice, the new algorithm requires both less stack space and less time. One modification is to stack a node only when both the sublists are unmarked. The other innovation is a "stacked-node-checking" method invoked after each stack-overflow. With this method, a number of unnecessary nodes are eliminated, the stack is compacted, and the marking process can resume using the generated space in the stack.

### Keywords and Phrases:

Marking algorithm, garbage collection, stack a node only when both sublists are unmarked, stacked node checking method, LISP

CR Categories: 4.19, 4.40, 4.49, 4.9

---

\*) This work is in part supported by PIPS project of Electro-technical Laboratory, Agency of Industry and Science, Ministry of International Trade and Industry, Japan.

Introduction:

Among existing marking algorithms, the simple stacking algorithm was known to be the fastest (see Knuth [1]). As shown in Alg. 1, it uses the stack space for storing the node elements whose cdr (right-link) is not yet attacked. Alg. 1 visits each node only once, which is the minimum number of visits. (A swapping algorithm, such as Schorr and Waite's [4] or Wegbreit's [5], visits each node twice.)

However, the number of stacking operations (push and pop) cannot be said to be minimum in Alg. 1.

The algorithm proposed here is a variation of this simple stacking algorithm and is faster, because a node is stacked only when both the sublists are unmarked. Thus, a number of stacking operations are eliminated.

The other problem on the marking algorithm is the amount of necessary working space. In this point, Wegbreit's algorithm is the best known. Generally speaking, stacking algorithms are worse than swapping algorithms, because a staking algorithm needs up to the total number of nodes in the worst case. If the stack space is less, stack overflow occurs.

Some escape methods have been proposed to handle this stack overflow. For the algorithm proposed here, the stacked-node-checking (SNC) method is employed, which deletes the unnecessary nodes from the stack. Some of the stacked nodes are unnecessary because the sublists are already marked, or one of its sublists is already marked

and the other one can be traced and marked. After SNC, the stack space is compacted by the elimination of these unnecessary stacked nodes, and the marking process is resumed using the generated space.

This SNC process is so simple that the total execution time, including many stack overflows, is faster than that of Schorr-Waite's swapping algorithm.

This fast marking algorithm was developed and implemented for LISP1.9 garbage collector [3].

#### Fast execution:

The problem for the tree marking (or the tree tracing) is, in short, a problem concerning the determination points (or branch points). The simple stacking algorithm (Alg. 1) uses the stack space to store the branch node. The swapping algorithm, on the other hand, uses the extra mark bit to indicate that this node is a branch point and its cdr link is not yet traced. In the swapping algorithm, the pointers are dynamically changed to keep track of the process history. To recover this destruction, each node must be visited twice. On the other hand, the simple stacking algorithm visits each node only once, which is the minimum number of visits. This difference in the amount of visiting makes the simple stacking algorithm faster than the swapping algorithms.

However, the number of stacking operations (push and pop), which occur once for each node in the simple stacking, is not the minimum. For example, examining Car Tree (Fig. 1), the marking can be easily accomplished without stacking, because there is eventually no decision

point in this tree.

The proposed algorithm (Alg. 2) uses two local variables, car-node and cdr-node, to check whether or not the node is really a branch node. The car-node and the cdr-node are both list elements and both unmarked. The push-stack operation occurs only when such a branch node is attacked.

On the contrary, the simple stacking algorithm (Alg. 1) pushes every encountered node.

This technique was hinted at from Wegbreit's bit-stacking algorithm [5]. There are two improvements. One involves checking the node to determine if it is a list-element or an atom-element. This algorithm checks whether it is an unmarked list element or not. The other improvement is that this algorithm uses stacking, while Wegbreit used pointer swapping. These improvements have made this fast algorithm both faster and require less space.

Table 1 shows the execution results for special cases. Table 2 shows the results for typical LISP programs. Remarkably that, the used stack space has reduced to less than half of that required by simple stacking.

#### Some extensions:

The introduction of two local variables (car-node and cdr-node) has enabled this stack space reduction. Is it possible to make more reduction if other local variables are added, such as caar-node, cdar-node, etc. ?

It is easily shown that stack depth reduction can be realized through introduction of other local variables. For example, using caar-node and cdar-node, the push stack operation can be postponed until such a time as all of the three nodes — caar-node, cdar-node and cdr-node — are known to be unmarked list elements. Algorithm 3 represents this modification for case 4 in Alg. 2. Stack reduction gain is 1, because the effect is realized just before the bottom of the tree.

Considering that only 10 or 11 stack spaces are used in typical LISP programs, another 22 local variables could delete the necessity for the stack space.

As compared to this explicit gain in stack reduction, the speed up gain will not be so clear. The reason is that the percentage of the stack operation time in the total marking process time has already become small, through the first introduction of car-node and cdr-node. Another reason is that the variable assignments cost some time proportional to the number of local variables, and it will lessen the gain realized from deleting stack operation time.

SNC method:

The weakpoint of a stacking algorithm is the theoretical need for large stack space. By the worst case analysis, a space as large as the total number of nodes is needed. The swapping algorithm, however, requires only the extra bit space which will cost (the-node-space/N) where there are N bits in a word (usually, there are 32 or 36).

In actual marking algorithm applications, this stacking algorithm weakness will not be realized because, usually, the situation is much better. Only about 30 words for stack space is sufficient.

In the above better case, the stacking algorithm can be said to be better than the swapping algorithm, even from the work storage viewpoint.

To solve the worst case problem, several methods were proposed to handle the stack overflow.

Schorr and Waite [4] proposed to use the swapping algorithm after the stack overflow. Kurokawa [2] proposed to move the tracing information from the stack to the tree itself, using extra mark bits. In this fast marking algorithm, Kurokawa's method cannot be applied because there is no node relation in the stack. Schorr and Waite's proposal is highly appropriate, but it is too expensive, because it means that two marking algorithms — stacking and swapping — co-exist at the same time and that both stack space and extra bit space are necessary.

The SNC method, proposed here, was invented for this fast marking algorithm. It uses no extra bits. First, each stacked node is checked to determine if it is necessary or unnecessary. Unnecessary nodes are then deleted from the stack. The space is made to resume the marking process.

Algorithm 4 shows the SNC method after the stack overflow. The first loop means stacked node checking. The function checks(shown

in Alg. 5) returns 0 when both the sublits are marked and unnecessary. It returns the node element (which may be different from the stacked element) when both of its sublits are unmarked. It should be noted that checks function is another kind of marking process and that the content of the stack will be changed.

The next loop deletes unnecessary cells and tries to make space for the marking process. If there is no space to be had, it gives up and advises of a fatal stack overflow error. If there is space to work in, the marking process resumes.

After the marking process successfully ends, the last loop begins to utilize the "checked" nodes, which are stacked at the first checking loop. The whole marking process ends itself after all the "checked nodes" are cleared from the stack.

The main SNC method characteristic is its speed. Even after a number of overflows, it is faster than the swapping algorithm, as shown in Table 3. If the overflow occurs only once or twice, the speed is faster than that of the simple stacking algorithm.

The SNC method defect is that it cannot ensure that no stack space is necessary. In a typical LISP program, where an 11 word stack is necessary without stack overflow, the SNC method can be successfully applied only after a 10 word stack is ensured. However, if stack space are used more, such as in the typical examples in Table 3, the SNC effect becomes very large.

Concluding remarks:

The fast marking algorithm was implemented for LISP1.9 [3] in the autumn of 1975. In LISP1.9, the marking process occupied about 3/4 of garbage collection time. The effects of this fast marking, both on speed and on stack space, are quite large. 20% speed-up gain is realized and only half of the stack space is necessary for the usual LISP programs. The old marking was by a simple stacking algorithm.


The SNC method, on the other hand, is not employed for typical LISP programs. In a sense, this is very pleasant for the fast marking algorithm, because it can be said to be better than others, even from the working space standpoint. The SNC method is a kind of "emergency exit" method for the fast marking algorithm (it can be applied to the simple stacking case, though). It would be better if the emergency exit were not used any more.

Reference

1. Knuth D. E., Fundamental Algorithms, The Art of Computer Programming 1, Addison-Wesley (1968)
2. Kurokawa T., New Marking Algorithms for Garbage Collection, Proc. of 2nd. USA-JAPAN Computer Conference (Aug. 1975).
3. Kurokawa T., LISP1.9 Programming System, Journal of Information Processing Society in Japan (~~in Press~~) vol 17, no. 11, 1056-1063
4. Schorr H. and Waite W. M., An efficient machine-independent procedure for garbage collection in various list structures, C.A.C.M. 10, pp. 501-506, (Aug. 1967) (1976)



5. Wegbreit B., A space-efficient list structure tracing algorithm,  
IEEE Trans. Comp. C-21, 9, pp. 1009-1010, (Sep. 1972)
6. Goldman N. M., Sentence Paraphrasing from a Conceptual Base,  
C. ACM, 18, 96-106, (Feb. 1975)

```
Procedure Simple_stack_mark (node);  
begin  
  if marked (node) then return ( )  
    else push (bottom-mark);  
loop {repeats until return ( ) is executed }  
  mark (node);  
  push (node); { Stack-overflow will occur here. }  
  node: = car[node];  
  if marked (node) then  
    loop { repeats popping the stack  
      until bottom or unmarked-  
      element is encountered }  
    node: = pop ( );  
    if node = bottom-mark  
      then return ( ) endif ;  
    node: = cdr[node];  
    if  marked (node)  
      then exit-loop ( ) endif ;  
    endloop  
  endif  
endloop  
end
```

Alg. 1 Simple\_stack\_marking algorithm

In this and following algorithms, marked (node) means that either the node is already marked according to the list element or it is an atom (i. e. non-list element).

Procedure Fastmark (node);

begin localvar: car-node, cdr-node;

if marked (node) then return ( )

else push (bottom-mark);

mark (node) endif;

loop { repeats until return ( ) is executed }

car-node: = car[node];

cdr-node: = cdr[node];

{ The following 4 cases exist for car-node and cdr-node. }

case 1: { Both car and cdr are already marked.

Pop stack will occur only for this case. }

if marked (car-node)  $\wedge$  marked (cdr-node)

then node: = pop ( );

if node = bottom-mark then return( ) endif

endif;

case 2: { Only car-node is already marked.

Marking will continue without push or pop. }

if marked (car-node)  $\wedge$   $\neg$  marked (cdr-node)

then mark (cdr-node);

node: = cdr-node

endif;

(Alg. 2 Fast marking algorithm) cont.

case 3: { Only cdr-node is already marked.

Marking will continue without push or pop. }

if  $\neg$  marked (car-node)  $\wedge$  marked (cdr-node)

then mark (car-node);

node: = car-node

endif ;

case 4: { Both car-node and cdr-node are unmarked.

Push stack occurs and the stack-overflow may be  
caused here. }

if  $\neg$  marked (car-node)  $\wedge$   $\neg$  marked (cdr-node)

then mark (car-node);

mark (cdr-node);

push (cdr-node);

node: = car-node

~~elseif~~

endif

endloop

end

Alg. 2 Fastmark algorithm

(Case 4 is modified in Alg. 3)

```
case 43: { Only cdar-node is marked. Loop again. }
  if  $\neg$  marked (caar-node)  $\wedge$  marked (cdar-node)
    then mark (caar-node);
        car-node: = caar-node
    endif ;
case 44: { Both are un-marked. Push-stack occurs. }
  if  $\neg$  marked (caar-node)  $\wedge$   $\neg$  marked (cdar-node)
    then mark (caar-node);
        mark (cdar-node);
        push (cdr-node);
        cdr-node: = cdar-node;
        car-node: = caar-node
    endif
  endloop
end
endif
```

Alg. 3 Modified Fastmark algorithm on the part of case 4.

case 4: { Expanded using caar-node and cdar-node.

Stack depth is decreased by 1. }

if  $\neg$  marked (car-node)  $\wedge$   $\neg$  marked (cdr-node) then

begin local vor : caar-node, cdar-node;

mark (car-node);

mark (cdr-node);

loop {repeat until exit-loop( ) is executed }

caar-node: = car[car-node];

cdar-node: = cdr[car-node];

case 41: { Both are marked. No push is necessary. }

if marked (caar-node)  $\wedge$  marked (cdar-node)

then node: = cdr-node;

exit-loop ( )

endif ;

case 42: { Only caar-node is marked. Loop again ! }

if marked (caar-node)  $\wedge$   $\neg$  marked (cdar-node)

then mark (cdar-node);

car-node: = cdar-node

endif ;

(Alg. 3 Modified Fastmark algorithm on the part of case 4.) cont.

Procedure Stacked-node-check (car-node, cdr-node);

{ This procedure is invoked when a stack-overflow occurs.

Thus stack-ptr = top-of-stack-space; you cannot push anymore. }

begin localvar: write-ptr, answer:

loop { repeats checking nodes until bottom-of-stack is encountered. }

node: = pop ( );

if node = bottom-mark then exit-loop ( ) endif ;

answer: = checks (node);

{ answer is either 0 (already marked) or tree-node }

write-to-stack (answer);

{ using stack as 1-dimensional array and replace the  
stacked node by 0 or node-element. }

{ The node-element means that both its car and cdr were  
un-marked. }

endloop;

{ The stacked elements may be changed. Some of them may  
be 0, which means that it can be deleted.

The stack-ptr is now at the bottom. }

write-ptr: = stack-ptr;

loop { deletes the 0 nodes until top-of-stack-space is encountered }

stack-ptr: = stack-ptr + 1 ;

if stack-ptr = top-of-stack-space

then exit-loop( ) endif ;

node: = read-stack ( ); {read-stack returns the stacked-node

which is at the stack-ptr position. }

```
if node = 0 then write-ptr: = write-ptr + 1 ;  
                                write-to-stack (node) { write-to-stack  
endif ;                                writes the node into the stack. }  
endloop ;  
{ After this deletion, stack-ptr = top-of-stack, and write-ptr =  
  top-of-necessary-nodes. }  
if write-ptr = top-of-stack  
    then error-of-fatal-stack-overflow ( ) endif ;  
{ Otherwise, there are some spaces for pushing cdr-node. }  
stack-ptr: = write-ptr;  
push (cdr-node); { Once overflow occurred due to this operation. }  
Fastmark (car-node); { Continue marking. }  
node: = pop ( ); { This node is actually old cdr-node. }  
Fastmark (node);  
loop { recovers the stacked nodes,until they exhaust. }  
    node: = pop ( );  
    if node = bottom-mark then return ( ) endif ;  
    { Otherwise, both car and cdr must be marked.  
      However, car[node] and cdr[node] are already  
      marked by the function checks [node]. }  
    car-node: = car[node] ;  
    cdr-node: = cdr[node] ;  
    push (cdr-node);
```



```
Fastmark (cdr-node);  
node: = pop ( );  
Fastmark (node)  
endloop  
end
```

Alg. 4. Stacked-Node-Checking method.

```
Function checks (node);  
begin localvar: car-node, cdr-node;  
  loop { repeats until either return ( $\emptyset$ ) or return (node) is executed. }  
    car-node: = car[node];  
    cdr-node: = car[node];  
    if marked (car-node)  $\wedge$  marked (cdr-node)  
      then return ( $\emptyset$ ) endif; { Unnecessary node ! }  
    if marked (car-node)  $\wedge$   $\neg$  marked (cdr-node)  
      then mark (cdr-node);  
          node: = cdr-node  
    endif;  
    if  $\neg$  marked (car-node)  $\wedge$  marked (cdr-node)  
      then mark (car-node)  
          node: = car-node  
    endif;  
    if marked (car-node)  $\wedge$  marked (cdr-node)  
      then mark (car-node);  
          mark (cdr-node);  
          return (node) { Cannot chase from this node.  
                          This node must exist in the stack. }  
    endif;  
  endloop  
end
```

Alg. 5 Function — check node

B-Tree	N = 8192		N = 16384	
	Time (msec)	Stack-Used (words)	Time (msec)	Stack-Used (words)
This algorithm	252	14	510	15
Simple-stacking	259	15	523	16
Pointer-swapping	714	0	1414	0

Car-Tree	N = 8192		N = 16384	
	Time (msec)	Stack-Used (words)	Time (msec)	Stack-Used (words)
This algorithm	289	0	574	0
Simple-stacking	349	8192	671	16384
Pointer-swapping	746	0	1547	0

Revised-Car-Tree	N = 8192		N = 16384	
	Time (msec)	Stack-Used (words)	Time (msec)	Stack-Used (words)
This algorithm	356	1	709	1
Simple-stacking	388	8192	773	16384
Pointer-swapping	835	0	1677	0

Pseudo-Car-Tree	N = 8192		N = 16384	
	Time (msec)	Stack-Used	Time (msec)	Stack-Used
This algorithm	362	2730	731	5461
Simple-stacking	398	8192	791	16384
Pointer-swapping	840	0	1670	0

Table 1. Comparison of several data on marking algorithms.

N means the total free list elements number.

	N = 30780	
	Time (msec)	Stack-Used (words)
This algorithm	803	11
Simple-stacking	974	26
Pointer-swapping	1896	0
Revised fast marking (caar-node and cdar-node Alg. 3)	799	10
Limited stack SNC-method used (overflow count = 6)	862	10

Table 2. Results for typical LISP program.

The program is BABEL, written by Goldman [6], which is one of the largest programs on LISP1.9.

N means the total free list elements number.

Pseudo-Car-Tree (N = 16384)	Stack-Area Allocated	Time (msec)	No. of Overflow
This algorithm	$\infty$ (5461)	731	0
	4000	769	1
	1000	790	5
	250	809	22
	50	849	113
Simple stacking	$\infty$ (16384)	791	0
Pointer-swapping	0	1670	0

Ladder (N = 16384)	Stack-Area Allocated	Time (msec)	No. of Overflow
This algorithm	$\infty$ (8192)	629	0
	2000	697	4
	500	707	16
	125	722	66
	25	878	356
	10	1201	1023
	3	5146	8190
Simple stacking	$\infty$ (8192)	651	0
Pointer-swapping	0	1542	0

Table 3. Marking time when stack-space is limited.

N means the total free list elements number.

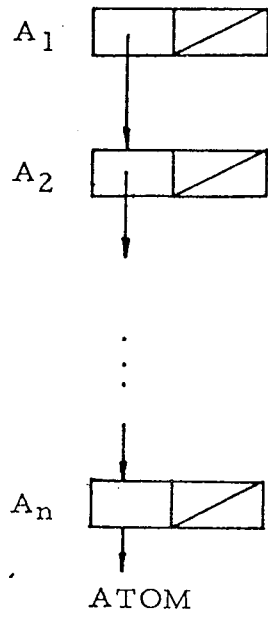


Fig. 1 Car-Tree

(Worst case for simple  
stacking algorithm.)

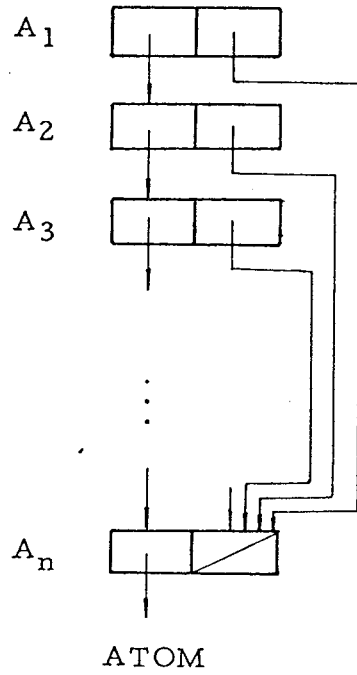


Fig. 2 Revised-Car-Tree  
(Worst case for Wegbreit's  
algorithm.)

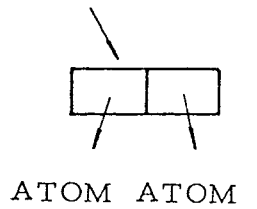
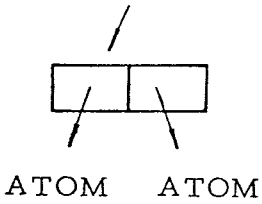
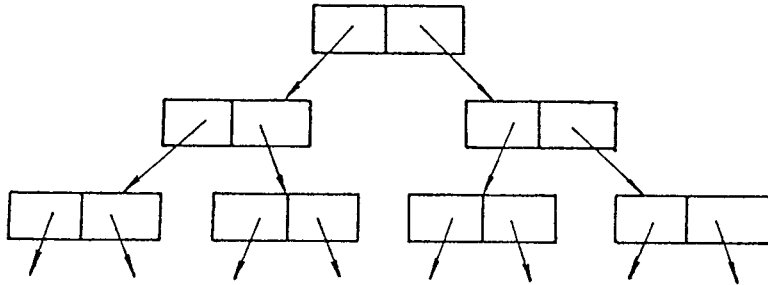


Fig. 3 B-Tree

(Difference is least between this algorithm and the simple stacking algorithm for this case.)



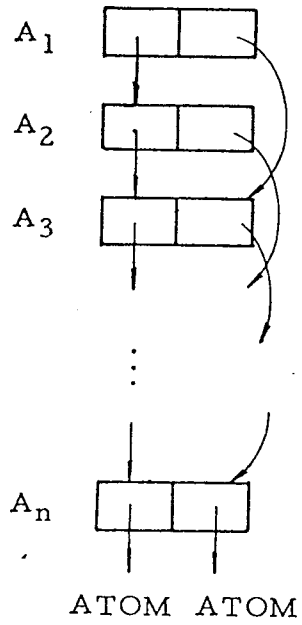


Fig. 4 Pseudo-Car-Tree

(This algorithm can work with only one  
one third or the stacking space,  
compared with the simple stacking  
algorithm.)

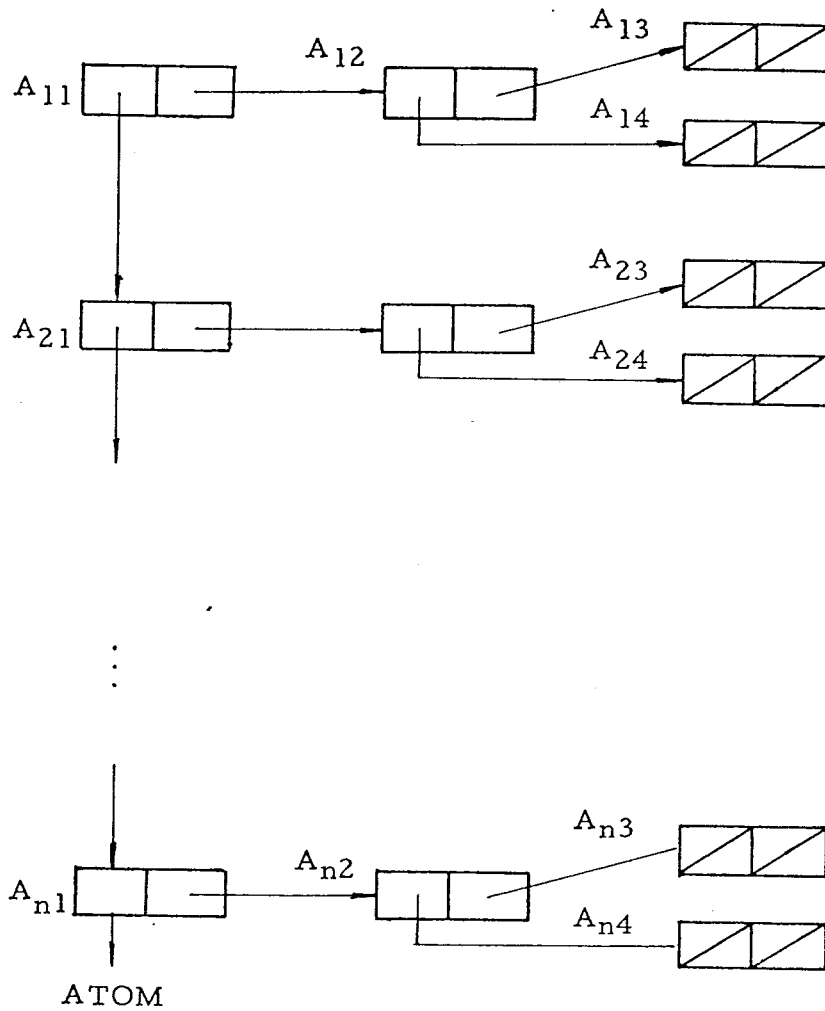


Fig. 5 Fork Case

(Worst case for this marking algorithm

If the stack space is less than  $n$ , the

marking process cannot be accomplished.)

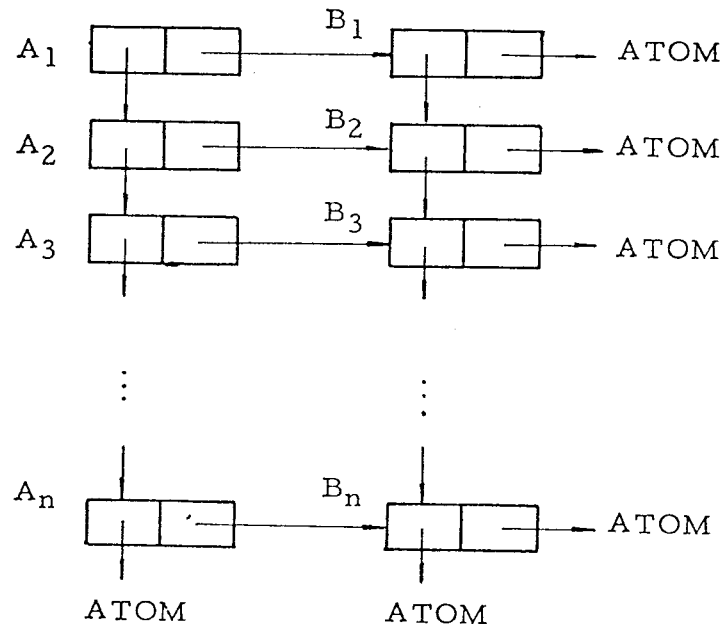


Fig. 6 Ladder Case

(This data needs  $n$ -words stack space for this marking algorithm. Using the SNC-method, however, the algorithm can work with only a 3-word stack space.)