Published: 05/03/67

Identification

Outward call argument management: arg_pull, arg_push
R. M. Graham, M. A. Padlipsky

Purpose

Arg_pull is a ring-0, slave procedure used only by the protection mechanism. Its function is the validation and copying of arguments for calls from an inner ring to an outer ring. The entire argument list (suitably modified) and all arguments must be copied into the stack of the called procedure, as data in the caller's ring are by definition inaccessible to outer ring procedures.

Arg_push is also a ring-0, slave procedure used only by the protection mechanism. Its function is the copying of return arguments back into the inner-ring areas where they are expected to be found on returns from an outer ring to an inner ring. That is, it is the converse of arg_pull.

Restrictions

Arg_pull and arg_push are predicated on the assumption that procedures making outward calls possess appropriately structured argument lists - specifically, "data descriptions" must be present. EPL/PL/I procedures may insure this by use of the "callback" option; see BP.0.02. Non-PL procedures must be coded so as to produce the equivalent of what PL ones do; see BD.1, BD.7.01, and Figure 1, below.

In the initial implementation, varying strings may not be passed as arguments on outward calls. All other data types mentioned in section BB.2 (System Interfaces) are acceptable.

Use

The Gatekeeper calls arg_pull as follows:

```
arg_pull (oldap, newsp, nextsp, ring, err_code);
```

with arguments declared

```
dcl (oldap, newsp, nextsp)ptr, (ring, err_code)
     fixed bin (17);
```

where oldap is equal to the argument pointer of the faulting procedure, newsp is equal to the stack pointer of the target procedure, ring is the ring number of the procedure for which the Gatekeeper is processing a call. Upon return from arg_pull, nextsp contains a pointer to a "newer" stack frame (which the Gatekeeper will place into newsp|18 using the terminology of Figure 2, BD.9.01), and err_code (if non-zero) contains a code indicating the type of error which occurred in attempting to "pull" the arguments.

The Gatekeeper calls arg_push as follows:

```
call arg_push (oldap, newap, ring, err_code);
```

with declarations

```
dcl (oldap, newap)ptr, (ring, err_code) fixed bin (17);
```

where oldap is the argument pointer for the procedure being returned to, newap is the argument pointer for the procedure being returned from, ring is the ring number of the procedure the Gatekeeper is processing a return from, and err_code is as above.

Method

1. Arg_pull

Figure 1 presents the format of a "callback"-type argument list. Figure 2 presents a block diagram of arg_pull. The logic is as follows: If there are no arguments (left half of first word of argument list equals zero), nextsp is set to point to newsp + 32, err_code is set to zero (indicating successful completion) and the routine returns. If there are no data descriptions (left half of second word of argument list is zero) and there exist arguments ("n" is not zero), an error condition exists and the routine returns, after setting err_code to 1. Next, check that none of the arguments is of illegal data type; if there is an illegal data type, set err_code to 2 and return. The final validity check which must be performed is the determination that each argument pointed to is indeed accessible to the routine whose argument list it appears in. (This step must be taken to prevent arg_pull from becoming an unwilling accomplice to an illegal act by exercising its reading privileges indiscriminately; it is not, of course, taken when arg_pull is operating in behalf of a ring-0 routine.) Call validate_arg (BD.9.03)

for the arguments and ring; if any argument is not accessible, set err_code to 3 and return. (To guard against possible alteration of the pointers on an interrupt, the validation is performed on arg_push's own copy of the argument list; the problem here is a consequence of the fact that segment-sharing allows for the possibility of some other user's altering the segment containing the argument list after validation - cf.BD.9.01). Otherwise, all of the argument list except the individual argument pointers can be copied directly into the new list, beginning at newsp+32.

The arguments themselves must be handled with some care. Scalars can be copied into the new stack in locations subsequent to the last data description with their corresponding argument pointer entries set to point to them. In the case of strings and one-dimensional arrays, the dope and data are copied without alteration into locations subsequent to p; new specifiers, to which the argument pointers are made to point, are created, taking into account the locations of the copies of the dope and data. (Specifiers and dope are discussed in section BP.2.02.) The final length of the area containing the argument list and the copied arguments is added to newsp+32 to determine the origin of the next available stack frame in the new stack; this value is returned to the Gatekeeper. Err_code is set to zero, indicating successful completion.

In the initial implementation, all arguments will be copied; that is, no attempt will be made to avoid dealing with arguments which may be accessible from the new ring without copying.

2. Arg_push

Figure 3 presents a block diagram of arg_push. The logic is as follows: If there are no arguments (left half of first word of argument list pointed to by oldap equals zero), set err_code to zero (indicating successful completion) and return. Otherwise, search the data descriptions associated with oldap, recording the number (i.e., position in argument list) and data type of any which are return arguments. If there are no return arguments, set err_code to zero and return. Next, call validate_arg (BD.9.03) for any return arguments found and ring, using copies of the argument pointers found in newap's list. Copying pointers and validating argument accessibility are done for the same reasons here as they are in arg_pull: possible alteration

in the pointers case, and possible fabrication in the accessibility case. (Note that ring is the ring number of the procedure being returned from.) If any return argument is not accessible from ring, set error_code to 1 and return. Otherwise, copy the data pointed to by the return argument pointers in the copy of the argument list pointed to by newap into the locations indicated by the corresponding argument pointers in the argument list pointed to by oldap. (Unlike the arg_pull case, arg_push need only copy data: for the data types permitted to be passed on inter-ring calls, dope and specifiers cannot have changed as a result of the call being returned from.) After copying the data, set error_code to zero and return.

Figure 1. Argument Lists

1a. In the calling procedure (general form):

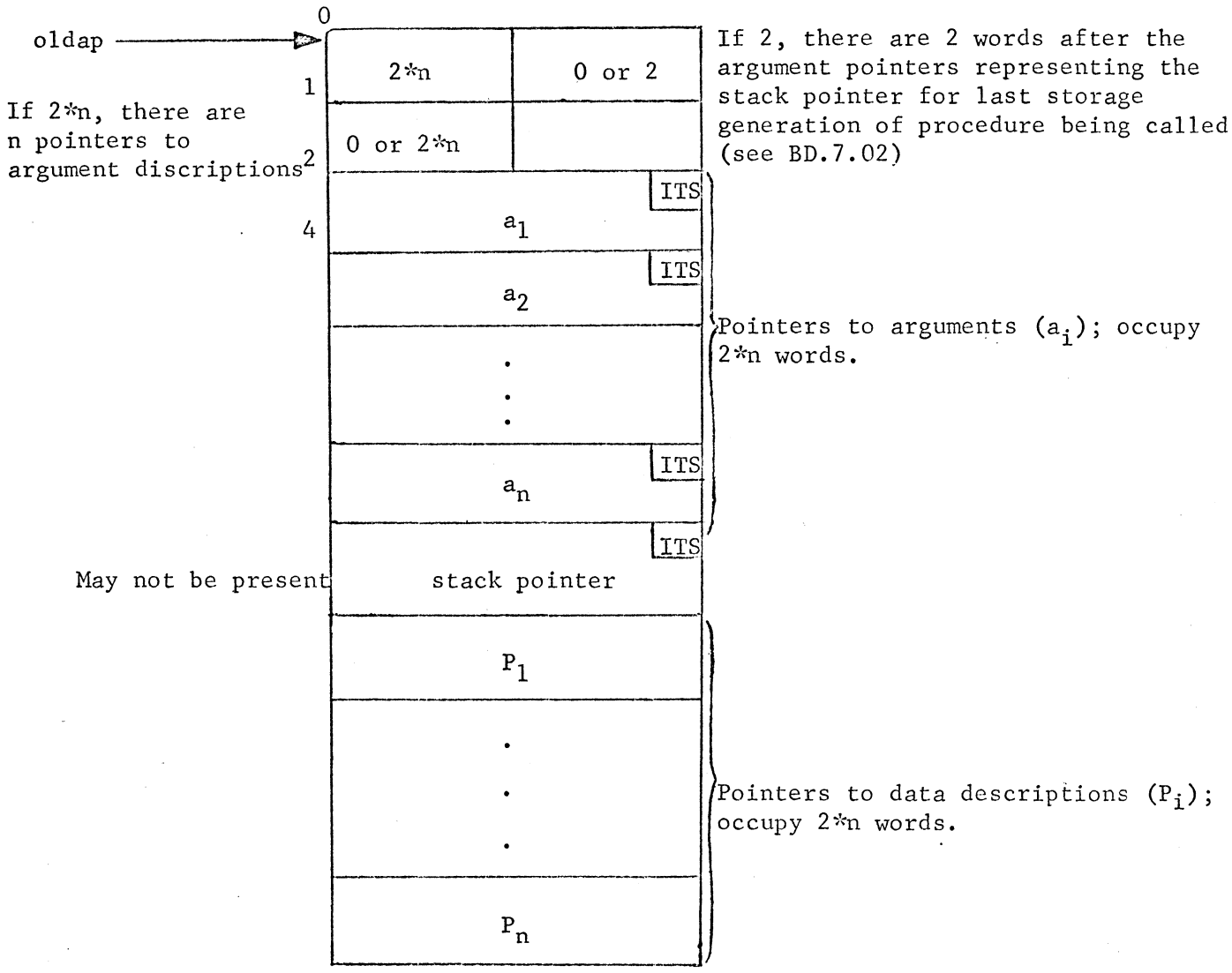
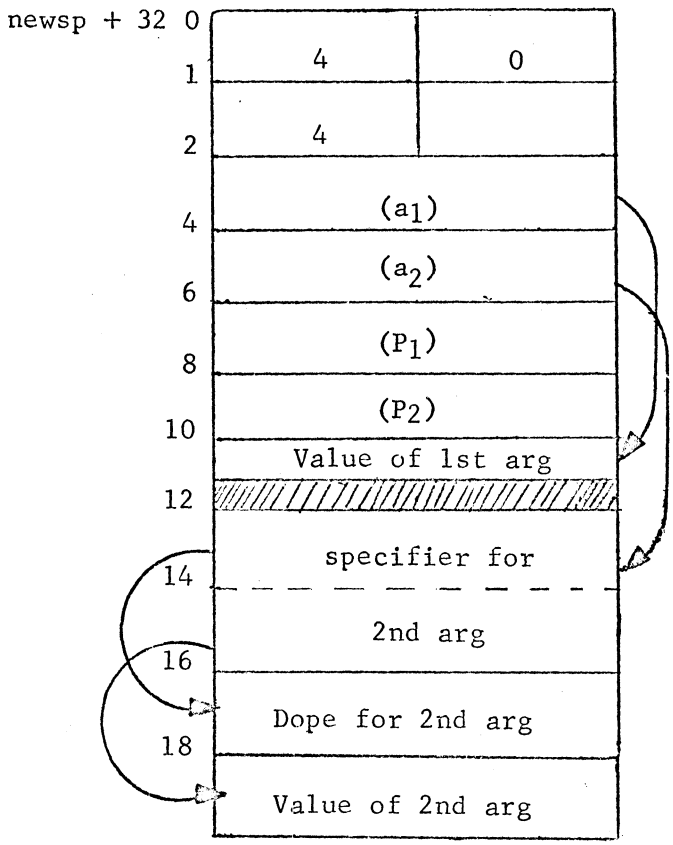


Figure 1, continued.

1b. Example of arg_pull-produced argument list:



Assume that the first argument (of 2) is a single precision scalar and the second a non-varying string.

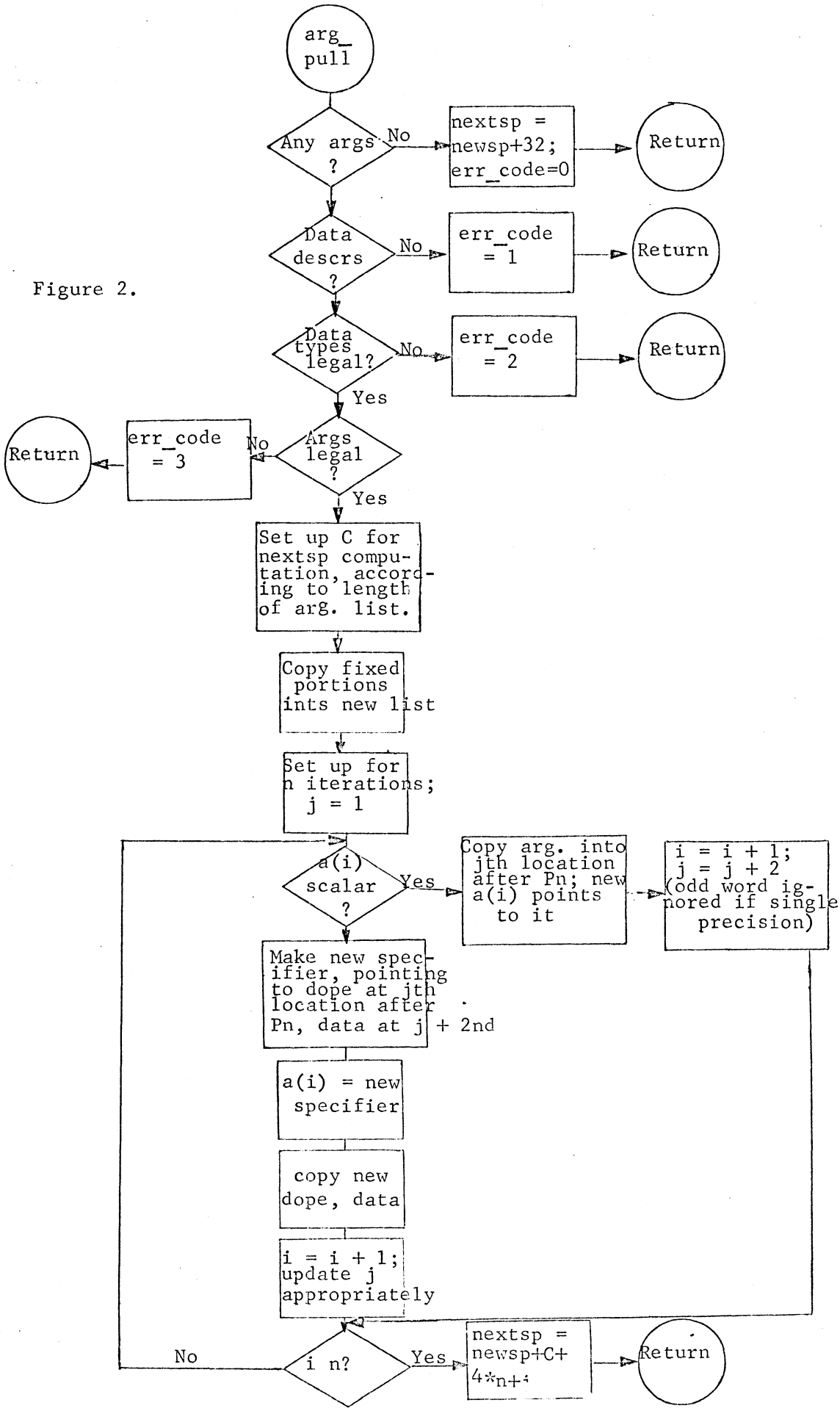

arg_ pull
Any args ?
No
nextsp = newsp+32; err_code=0
Return
Data descrs ?
No
err_code = 1
Return
Data types legal?
No
err_code = 2
Return
Yes
Args legal ?
No
err_code = 3
Return
Yes
Set up C for nextsp computation, according to length of arg. list.
Copy fixed portions ints new list
Set up for n iterations; j = 1
a(i) scalar ?
Yes
Copy arg. into jth location after Pn; new a(i) points to it
i = i + 1; j = j + 2 (odd word ignored if single precision)
Make new specifier, pointing to dope at jth location after Pn, data at j + 2nd
a(i) = new specifier
copy new dope, data
i = i + 1; update j appropriately
i n?
No
Yes
nextsp = newsp+C+ 4*n+4
Return

Figure 2.

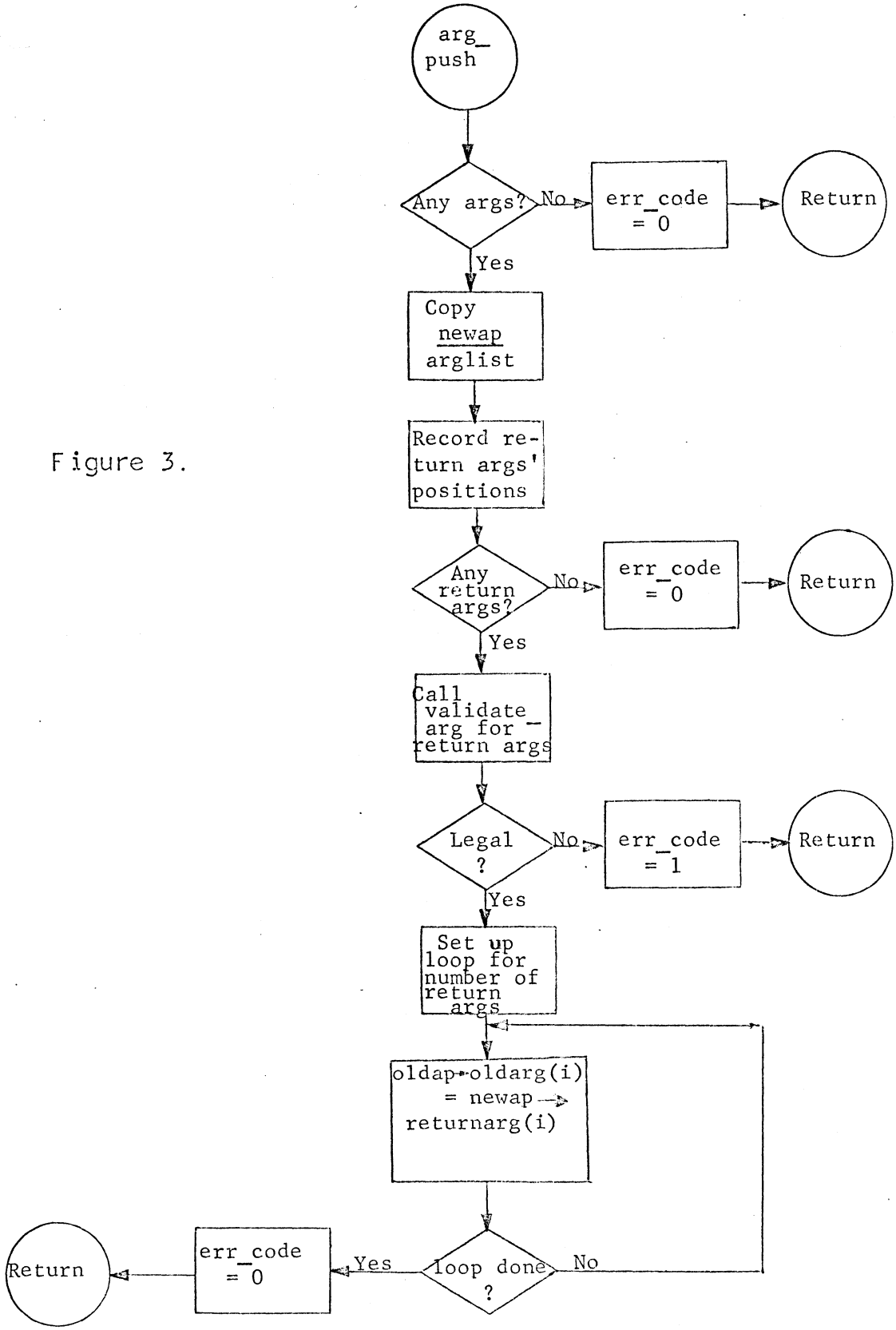

arg_ push
Any args?
No
err_code = 0
Return
Yes
Copy newap arglist
Record return args' positions
Any return args?
No
err_code = 0
Return
Yes
Call validate_arg for return args
Legal ?
No
err_code = 1
Return
Yes
Set up loop for number of return args
oldap->oldarg(i) = newap->returnarg(i)
loop done ?
Yes
No
err_code = 0
Return

Figure 3.