Published: 10/7/66

Identification

Expression-evaluator for Interactive Debugging Programs

evaluate

D. B. Wagner, M. A. Padlipsky

Purpose

The two procedures parse and evaluate constitute the expression-evaluating machinery used by the interactive debugging aids. Parse is described in BY.6.03; briefly, it takes a symbolic expression as a character-string and returns a pointer to an operator-operand tree representing the expression.

Evaluate is given such a tree and evaluates the expression, looking up symbols in the Combined Symbol Table (see BY.6.02). It returns two pointers: one points to data representing the value of the expression, and the other points to an indication of the interpretation to be placed upon this data.

Usage

The call is

```
call evaluate (tree_pointer, data_pointer, node_pointer,
               work_space);
```

The declaration associated with the arguments is:

```
dcl (tree_pointer, data_pointer, node_pointer)
    ptr, work_space area ((*));
```

Tree_pointer is a pointer to the root of an operator-operand tree produced by parse. Evaluate starts at the root and works its way by recursive calls to itself to the end-points of the tree. It places into data_pointer a pointer to data representing the value of the expression, and places into node_pointer a pointer to a "node" similar to the "nodes" in the Combined Symbol Table (see BY.6.02). This node and its associated "information block" contain all information relevant to the interpretation of the value of the expression.

Node_pointer may in some cases point to a node which is actually in the Combined Symbol Table, but in general it points to a dummy created in work_space by evaluate. This node contains a pointer to the special translator-name "[internal]", and the format of the "information block" associated with the node is an extension of the PL/I symbol table format described in BD.1.02.

Work_space is an area into which successively-called generations of evaluate may place information for the caller; this tactic is necessary to untangle problems arising from the freeing of automatic storage in recursive routines. Ultimately, the calculated value of the expression will be placed into the work_space of the original caller by the final recursion of evaluate. See implementation discussion, below, for details.

Conventions

The final version of evaluate will be able to handle any expression of the debugging expression language, that is any PL/I expression with any data-types allowed in PL/I, and also any expression containing the special operators "?" and "$" or having the special data-type "address". See BX.10.00 for details.

The initial implementation of evaluate allows all the PL/I operators except "." and "→". It recognizes the "$" operator, the substr and unspec functions from PL/I, and the c and cr ("contents" and "contents of register") functions of the debugging language. It returns only the following data-types:

```
fixed binary (17)
float binary (27)
char
bit
"address"
```

Note that the PL/I symbol table is not recognized by the initial implementation of evaluate, so that the only data-type which may be possessed by a symbol involved in an expression is "address". However constants in the expression (e.g. 2.3 or "0"b) may have the fixed, float, character, and bit data-types according to the form of the constant, and the built-in functions recognized by evaluate may have values with any of these data-types.

Implementation

General

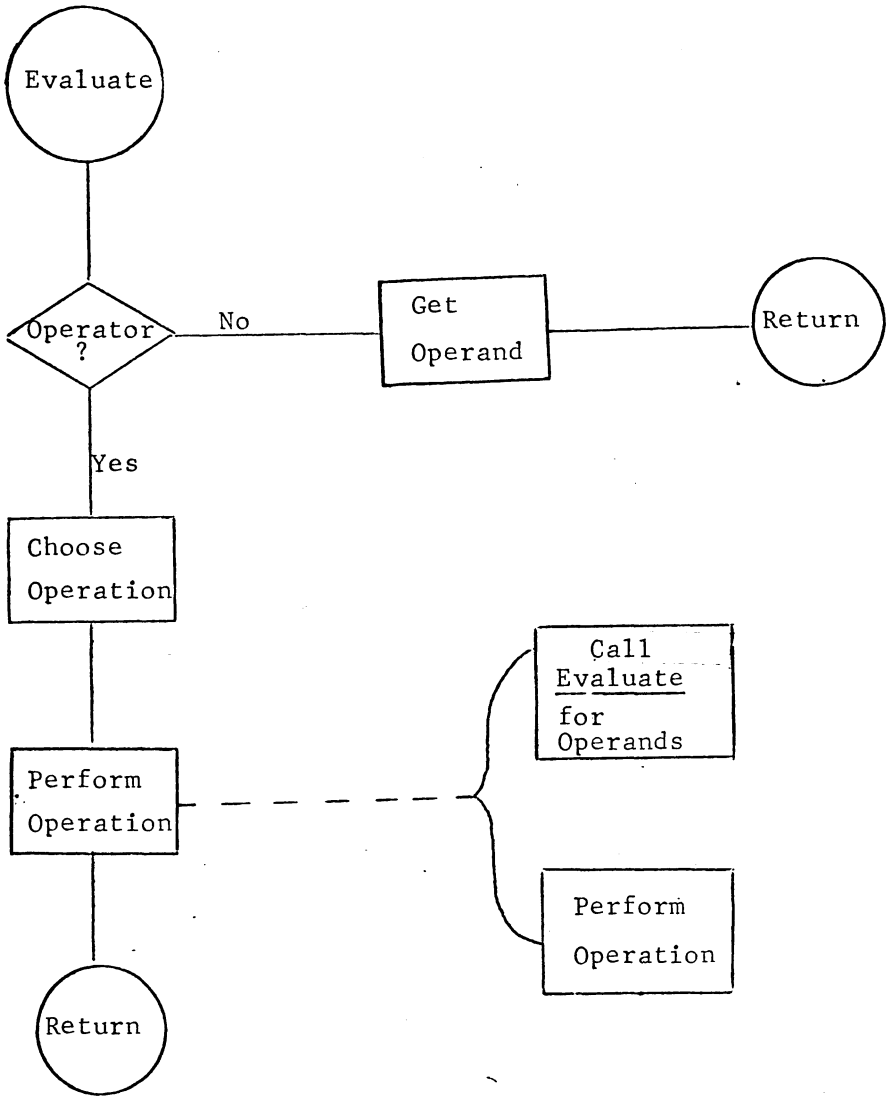
The overall structure of evaluate is quite straightforward. An input argument named "tree_pointer" points to a structure called "pole", generated by the parse routine (BY.6.03). If the item "pole.type" pointed to is "0"b, the structure at hand deals with an operand and evaluate immediately calls arg, a routine which will get the operand, allocate it into storage accessible to the caller, and return pointers to the operand and to descriptive information. Otherwise, the structure at hand deals with an operator and evaluate calls the appropriate operator routine, according to the value of the item "pole.name". (E.g., if pole.name equals "+", add is called.) The operator routines perform their particular operations on operands which reside in "pole"-structures which are pointed to by members of an array called "pole.arguments" in the structure at hand. Of course, the pointed-to structures may themselves represent operators (see, for an example, BY.6.03). Each operator routine, then, begins by calling evaluate for each "pole"-structure pointed to by a "pole.arguments", and the recursion will eventually (or immediately) lead to the value(s) on which the particular operation will be performed. An answer allocated into the "work_space" furnished as an input argument, a "data_pointer" to the answer, and a "node_pointer" to the description of the answer will then be returned by the operator routine.

Operator Routines

There are two difficulties which must be coped with in the operator routines:

1. With four possible data types (fixed, floating, bit, and character) and legal mixing of types for most operators, the handling of such mixing requires rather space-consuming treatment in EPL. That is, for example, the "less than" (lth) routine must contain three EPL statements for each of the sixteen possible permutations of operand types, generating large amounts of machine-language code. It seems likely that machine-language coding of the operator routines could effect considerable savings both of space and of execution time; nevertheless the initial implementation will be in PL/I for clarity and maximum flexibility.

BLOCK DIAGRAM



Surprisingly enough, this is all there is to it- through the wonders of recursion.

2. The highly recursive nature of evaluate makes the issue of generations of storage a tricky one. When an operator routine calls evaluate in order to "recurse" toward an operand, it is not sufficient merely to return a pointer to the operand (from the recursively-called generation(s) of evaluate), for the storage space the operand occupies will have automatically been freed if it was "internal" to the called ("descendant") routine. However, each descendant generation can place its (intermediate or final) result into storage belonging to the routine which called it ("dynamic ancestor", perhaps). Such storage (the "work_space" argument of the calling sequence, as above) will be automatically freed only on return from the calling ("ancestor") routine, and hence represents a solution to the vanishing operands problem. (See also the example below which presents a schematic version of add.)

Input and output arguments and declarations for the several operator routines are identical to those of evaluate.

E.g.,

```
call add (tree_pointer,data_pointer,node_pointer,work_space);
```

In addition, each operator routine must declare working space of its own for recursively-called "descendant" routines to place their (intermediate) results into. E.g., schematically,

```
add: proc (tree_pointer,data_pointer,node_pointer,work_space);

...

dcl (p1,p2,np1, np2) ptr;

dcl space area ((1024));

...

call evaluate (tree_pointer->pole.arguments (1),p1,np1,space);

...

allocate answer in (work_space) set (data_pointer);

data_pointer->answer = p1->arg_1 + 2->arg_2

/* actual coding takes data types of arguments into account */...
```

The operands generated by the recursion will be allocated into "space", then, with p1 and p2 as pointers to them; the result of their addition, on the other hand, is allocated "back" into "work_space" (belonging to the generation of evaluate which invoked add, or to the routine which originally invoked evaluate), with "data_pointer" pointing to it. "Space" will be automatically freed when add returns to the generation of evaluate which called it; this is safe--and desirable--as it contains only the intermediate values, arg_1 and arg_2. (Responsibility for the size and ultimate disposition of the original "work_space" is, of course, left to the calling program.) The size of "space" is at present arbitrary, and is, of course, subject to revision in the light of future experience.

The operator routines fall into five categories:

a) Arithmetic: addition(add), subtraction (sub), multiplication (mul), division (div), exponentiation (exp--which will not be implemented in EPL), unary negation (neg), and unary addition (uad);

b) Comparison: equal (equ), not equal (neq), less than (lth), less than or equal to (leq), greater than (gth), and greater than or equal to (geq);

c) Bit: "and"-ing (and), "or"-ing (or), "not"-ing (not), and concatenating (cat);

d) "Built-in": sin, cos, and the like (not yet fully enumerated, but in principle adaptable from EPL); and

e) "Debugging language special": "?" (que) and "$" (dol) (see BX.10.00).

Members of the first three categories are very similar on a per-category basis; that is, for example, add is just like sub except for the signs and the three letters "a,d,d" vs. the three letters "s,u,b".

Other necessary routines on the assumption that machine-language coding is permitted, are data-type conversion (fltfix, fixflt, fixchr, etc.); these will be adapted from EPL code.

The operator routines, of course, are only sensibly callable from evaluate.

Operand Routine

"Arg"'s function is to call "search_tables" (see BY.6.02) appropriately and then to return the desired operand. Arg will subsequently allocate the operand into the "work_space" it was given as an input argument, with the output argument "data_pointer" pointing to it, and return.

It is also likely that arg is only sensibly callable from evaluate.