Published: 09/30/66

Identification

Syntax Analyzer for the Debugging Language
parse
D. B. Wagner

Purpose

The procedure described here is used in conjunction with the procedures evaluate and setvalue (described in BY.6.04 and BY.6.05) to evaluate expressions for the debugging programs. The debugging expression language is described in BX.10.00 - it is essentially the PL/I expression language, with the addition of the operators "?" and "$", and with a data-type "address" added. Parse will be used by the debugging programs and will probably see some use elsewhere as well. It takes a symbolic expression and produces an operator-operand tree representing the expression. Then evaluate, a recursive routine, can work its way from the root down the tree to evaluate the expression.

Usage

The call is

```
call parse (expression, tree_pointer, eq_special, work_space);
```

The declarations associated with the arguments are:

```
dcl expression char(*) varying,
    tree_pointer ptr,
    eq_special bit(1),
    work_space area((*));
dcl 1 pole ctl(p),
    2 na fixed,      /* number of arguments */
    2 ln fixed,      /* length of name */
    2 type bit(1),
    2 name char (p → pole.ln);
    2 arguments (p → pole.na) ptr;
```

parse makes a tree each of whose nodes (hereafter called "poles" to distinguish them from the myriad other "nodes" mentioned in this manual) has the form of the structure

pole, above. These poles are allocated into the area work_space. Each pole.name is normally an identifier found in the expression: actually a variable or function name, the name of one of the infix operators such as "+" or "||", one of several special function names such as "[minus]" (substituted for the unary minus-sign), or perhaps a quoted string. The corresponding pole.type is "0"b for an "ordinary" variable or constant used in the expression, e.g. one that occurs directly after an operator or left parenthesis and not directly before a left parenthesis. Pole.type is "1"b for a function-name, dimensioned variable, or operator, and in this case the array pole.arguments contains pointers to poles for the arguments.

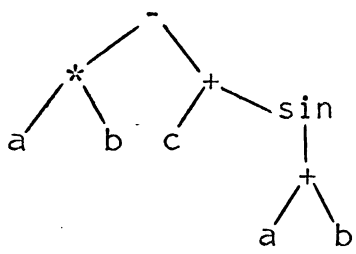
If the argument eq_special is "1"b, then an "=" operator at the outer level of parenthesis nesting will have the special operator ":=" substituted for it, with the proper priority (see table below). This "messing about" with operators is necessary because it was felt to be desirable to make the set request to probe look like a PL/I assignment statement, but the normal comparison "=" has the wrong priority for the assignment "=". See the example below for a detailed discussion of this problem.

Examples

For the expression

```
a*b-(c+sin(a+b))
```

parse produces a tree which can be diagrammed as follows:



To illustrate the problem of the "=" operator mentioned above, consider the PL/I expression

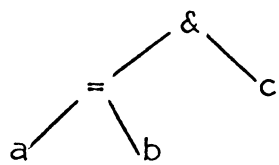
```
a = b&c
```

Seeing this expression out of context it is impossible to tell what it means. If it is a subexpression of an assignment statement, it means, "The logical conjunction of the truth value of the statement a = b and the value of the bit-string c." If on the other hand the above

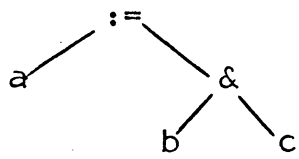
expression is the entirety of an assignment statement, it means, "Set the value of the bit-string a to the conjunction of the values of b and c."

This nasty assymmetry was solved in MAD by using ".E." as the comparison-equal operator, and in Algol by using ":=" as the assignment-equal operator. The PL/I language requires its compiler to be more clever and assign two different priorities to the one "=" operator depending upon context.

If parse is called to analyze the above expression with eq_special = "0"b, then the result is



while if eq_special = "1"b, then result is



Conventions

Operators recognized in expressions, in order of precedence, are:

?

.

¬

$

**

¬ + - (unary)

* /

+ -

= ¬= > < <= >=

&

|

||

:=

If an expression involves operations of the same priority, the binary operators "?" and "." take precedence on the right. Others, specifically "→", take precedence on the left.

(It will be remembered from BX.10.00 that the question mark is an operator used in the debugging language to indicate a restriction of table searching to symbols in a certain block, and that the dollar-sign is used as an operator which produces an address. The period and right-arrow are apparently not considered to be operators in PL/I, but this is by far the easiest way to handle them internally. These and the rest of the operators mentioned above have precisely the same effect as in PL/I.)

To distinguish them from their binary counterparts, the following unary operators are replaced by special function-names:

- [minus]

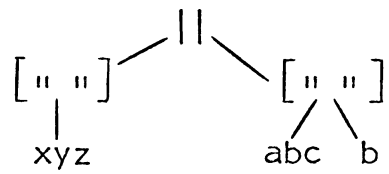
+ [plus]

Quotations are as in PL/I: double-quotes are used, and a doubled double-quote inside a quoted string stands for a single double-quote. An identifier may directly follow the terminating double-quote character indicating that some special interpretation is to be placed on the quoted string. In PL/I only "b" is used in this context: others may be defined in the debugging expression language.

Where a quotation occurs in an expression, the corresponding pole in the tree representing the expression is as follows: the dummy name [" "] (given here without quotation marks to avoid further confusion) is given with two arguments, the first an "ordinary" pole with pole.name equal to the quoted string, and the optional second argument the identifier (if any) following the quotation. For example the expression

```
"xyz"      ||  "abc" b
```

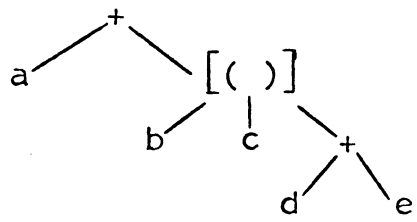
(which of course is not a correct expression but looks fine to parse) would be converted to a tree of the form



If a list of sub-expressions in parentheses separated by commas occurs in the expression, as in

```
a + (b,c,d+e)
```

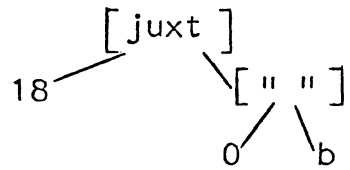
(this may not be allowed in the expression language but is easily handled by parse: presumably a would have to be an array for it to make sense), the dummy name [( )] is used as a function name indicating a list. The tree produced by parse for the above expression would have the form,



Parse uses the special function-name [juxt] to indicate that two subexpressions are juxtaposed without any operator in between. This is needed to allow handling of the strange syntax of the following PL/I expression:

(18) "0"b

which is shorthand for "000,000,000,000,000,000"b. This expression would be parsed as:



(Initial versions of evaluate may not recognize such curiosities as this one, but the debugging language, as an interaction language, desperately needs this kind of shorthand.)