Published: 11/26/69

Identification

LSM Utility Procedure
lsut
Edwin W. Meyer, Jr.

Purpose

lsut comprises a number of entries intended for use in creating, manipulating, and interrogating the LSM-format list structure described in MSPM section BY.22.01.

Functions

Detailed calling sequences for the entries provided in lsut are presented in MSPM Section BS.22.09. An enumeration of the types of functions available and their logical calling sequences is presented below. Note that the actual calling sequences and entry names are somewhat different.

```
node = make_fix (fixed_number);
```

make_fix creates a single element fixed array data block containing `fixed_number' and returns the `node' address of the block.

```
fixed_number = get_fix (node);
```

get_fix returns the value of the zeroth element of the fixed array block at `node'.

```
node = make_bit (bit_string);

node = make_char (character_string);
```

make_bit/make_char creates a bit/character string data block containing the supplied string and returns the `node' of the block.

```
bit_string = get_bit (node);

character_string = get_char(node);
```

get_bit/get_char returns the value of the bit/character string data block at `node'.

```
array_node = make_array (node_0, node_1, ..., node_n);
```

make_array creates a node array data block of n + 1 elements containing the supplied nodes and returns the `array_node' of the created block.

```
node = get_array (array_node, array_index);
```

get_array returns the `array_index'th node of the node array data block at `array_node'. (The lower bound of `array_index' is zero.)

```
call set_array (array_node, array_index, node);
```

set_array inserts `node' into the `array_index'th element of the node array data block at `array_node'.

```
char_node = concat (char_node_a, char_node_b, ...,
                    char_node_n);
```

concat creates a character string block containing the concatenation of the values of the supplied character string block nodes and returns the `char_node' of the created block.

```
truth_value = equal (node_a, node_b);
```

equal compares the data block nodes `node_a' and `node_b' for equivalency. Nodes are equivalent if they are of the same data type: fixed, bit, or character, and contain identical data. Node array type nodes are equivalent if they are of the same length and contain equivalent nodes in each of the array elements. Null nodes or indirect type nodes or hash list type of the same length are equivalent by definition.

```
truth_value = greater (char_node_a, char_node_b);
```

greater compares the character string data blocks `char_node_a' and `char_node_b'. `truth_value' is returned as T if `char_node_a' is greater in the ascii collating sequence than `char_node_b'; otherwise F is returned.

parsed_node = parse (character_string);

This function parses `character_string' into character string tokens delimited by one or more blank/new line characters. Leading and trailing blank/new line characters are ignored. Character string blocks containing these tokens are created and their nodes are inserted into the elements of a created node array in the order in which the tokens were found in the left-to-right scan of `character_string'. The array is of the minimum size sufficient to contain all the tokens -- there are no unused elements. For the double quote token, a null node is inserted into the proper element of the node array, rather than there being a character string block created for it. The node of the created node array is returned is `parsed_node'.

call print (node_a, node_b, ..., node_n);

This function prints the character string equivalents of the supplied data blocks on-line without any seperation. Only fixed, bit string, or character string data blocks are printed; other types are ignored. Only the zeroth element of a fixed array block is printed.

call print_columns (fixed_width, node_a, node_b, ..., node_n);

This function is similar to the `print' function except that the values of the data blocks are printed in columns of `fixed_width' characters. Short strings have added blanks to fill out the column, and long strings are truncated to fit the column.

call print_data_block (node);

A complete description of any type data block is printed on-line, including its type and length and character string equivalents of all items.

Examples

The following examples are designed to illustrate the usage of these functions in manipulating list structure data blocks. Note that the PL/I programs illustrated below are incomplete, lacking declarations.

The first example involves a set of three-element node array data blocks which are currently unlinked. It is desired to link them in a threaded list of four-element array blocks. The zeroth element of each block is to contain the node of the next block, and the remaining elements are to contain the three nodes of the original block. This threaded list is terminated by a null (0) node in the zeroth element of a block. The function outlined below takes a three-element array and places the four-element equivalent at the head of a supplied threaded list of such blocks.

```
new_list_top = convert_insert (old_block, list_top);

     convert_insert: proc (old_block, list_top,
                           new_list_top);

     new_list_top = make array (list_top, get_array
     (old_block, 0), get_array (old_block, 1), get_array
     (old_block, 2));

     end convert_insert;
```

The second example involves the threaded list described above; it is desired to delete a certain block specified by node from the list. The following recursive function does that.

```
new_list_top = delete_block (list_top, del_block);

     delete_block: proc (list_top, del_block,
                         new_list_top);

     if list_top = 0 /*null*/ then new_list_top = 0;

     else if equal (list_top, del_block) then
     new_list_top = get_array (list_top, 0);

     else do;

     call set_array (list_top, 0, delete_block
     (get_array (list_top, 0) del_block));
     new_list_top = list_top;
     end;

     end delete_block;
```

The third example presumes that element 1 of the previously described 4-element block is a character string key by which the threaded list is alphabetically ordered. The following recursive function inserts a new block into the proper position in the threaded list.

```
new_list_top = insert_block (list_top, block);

     insert_block: proc (list_top, block,
                         new_list_top);

     if list_top = 0 then do;  /*insert new block at
                                 end of list*/
     new_list_top = block;
     call set_array (block, 0, 0);  /*insert null thread*/
     end;

     else if greater (get_array (list_top, 1), get_array
     (block, 1)) then do;  /*insert it here*/
     call set_array (block, 0, list_top);
     new_list_top = block;
     end;

     else do;
     call set_array (list_top, 0, insert_block (get_array
     (list_top, 0), block));
     new_list_top = list_top;
     end;

     end insert_block;
```