Purpose

To allocate space on the memory stack for frames of variables to be used by rulebooks, activities and actions.

§**1. The Memory Stack.**   The M-Stack, or memory stack, is a sequence of frames, piled upwards. If we had an accessible stack in memory, we could use that, but neither the Z-machine nor Glulx has such a stack, alas, alas, alas. The following is not a very good solution, but it just about works.

```
Constant MAX_MSTACK_FRAME = 2 + {-value:max_frame_size_needed};
Constant MSTACK_CAPACITY = 20;
Constant MSTACK_SIZE = MSTACK_CAPACITY*MAX_MSTACK_FRAME;

Array MStack --> MSTACK_SIZE;
Global MStack_Top = 0; ! Topmost word currently used
```

§**2. Create Frame.**   A frame is created by calling the following function with two arguments: `creator`, a function which initialises a block of variables, and an ID number identifying the owner.

The `creator` function is called with the address at which to initialise the variables as its first argument, and the value 1 as the second argument. (The idea is that the same function can be used later to deallocate the variables, and then the second argument will be -1.) The `creator` function returns the extent of the block of memory it has used, in words. Thus is required to be strictly less than `MAX_MSTACK_FRAME` minus 1.

```
[ Mstack_Create_Frame creator id extent;
    if (creator == 0) rfalse;
    extent = creator.call(MStack_Top+2, 1);
    if (extent == 0) rfalse;
    if (MStack_Top + MAX_MSTACK_FRAME >= MSTACK_SIZE + 2) {
        RunTimeProblem(RTP_MSTACKMEMORY, MSTACK_SIZE);
        Mstack_Backtrace();
        rfalse;
    }
    MStack_Top++;
    MStack-->MStack_Top = id;
    MStack_Top++;
    MStack_Top = MStack_Top + extent;
    MStack-->MStack_Top = -(extent+2);
    rtrue;
];
```

§**3. Destroy Frame.** As sketched above, the same creator function and ID number are passed to the following routine to destroy the frame again. It takes the stack down to the level of the most recently created frame with this ID number: note that each action, for instance, has its own ID number for this purpose, but can be taking place several times in a nested fashion – one taking action might have caused another taking action which caused a third, for instance, so that there are three incomplete taking actions at once. In that case, there will be three independent sets of taking action variables on the M-stack, all with the same ID number. We remove the topmost one: the implication of that is that frames must always be destroyed in reverse order of creation.

In practice, I7 uses frames such that the frame sought should always be the topmost one in any case, and so that frames are always explicitly destroyed, not wiped by being undercut when an earlier-created frame is destroyed.

```
[ Mstack_Destroy_Frame creator id pos;
    pos = Mstack_Seek_Frame(id);
    if (pos == 0) rfalse; ! Not found: do nothing
    MStack_Top = pos - 2; ! Clear mstack down to just below this frame
    if (creator) creator.call(pos, -1);
    rtrue;
];
```

§**4. Seek Frame.** We return the position on the M-stack of the most recently created frame with the given ID number (see above), or 0 if no such frame exists; the size is stored in the global variable `MStack_Frame_Extent`. (Because word 0 on the stack is used as a sentinel – all frames are placed above it – no frame can actually begin at word 0 on the stack, so 0 is safe to use as an exception.)

```
Global MStack_Frame_Extent = 0;

[ Mstack_Seek_Frame id pos;
    pos = MStack_Top;
    while ((pos > 0) && (MStack-->pos ~= 0)) {
        MStack_Frame_Extent = MStack-->pos;
        pos = pos + MStack_Frame_Extent;
        MStack_Frame_Extent = (-2) - MStack_Frame_Extent;
        if (MStack-->(pos+1) == id) return pos+2;
    }
    MStack_Frame_Extent = 0;
    return 0; ! Not found
];
```

§**5. Backtrace.**   Purely for debugging purposes, and giving feedback if the stack runs out of memory:

```
[ Mstack_Backtrace pos k;
    print "Mstack backtrace: size ", MStack_Top+1, " words^";
    pos = MStack_Top;
    while (MStack-->pos ~= 0) {
        MStack_Frame_Extent = MStack-->pos;
        pos = pos + MStack_Frame_Extent;
        MStack_Frame_Extent = (-2) - MStack_Frame_Extent;
        print "Block at ", pos+2,
            " owner ID ", MStack-->(pos+1), " size ", MStack_Frame_Extent, "^";
        for (k=0: k<MStack_Frame_Extent: k++) print MStack-->(pos+2+k), " ";
        print "^";
    }
];
```

§**6. Access to Variables.**   An M-stack variable is identified by a combination of ID number and offset: for instance ID 20007, offset 1, is the variable "room gone to" belonging to the going action. The following routine converts that into an address on the M-stack, in the topmost block with the given ID number (since "room gone to", for instance, always means its value in the most current going action of those now under way). Typechecking in the compiler should mean that it is impossible to produce either error message below: NI will only compile valid uses of MstVO ("M-stack variable offset") where the seek succeeds and the offset is within range.

```
[ MstVO id off pos;
    pos = Mstack_Seek_Frame(id);
    if (pos == 0) {
        print "Variable unavailable for this action, activity or rulebook: ",
            "internal ID number ",
            id, "/", off, "^";
        rfalse;
    }
    if ((off<0) || (off >= MStack_Frame_Extent)) {
        print "Variable stack offset wrong: ", id, "/", off, " at ", pos, "^";
        rfalse;
    }
    return pos+off;
];
```

§**7. Access to Nonexistent Variables.**   A long-standing point where I7 is not as strict in type-checking as it might be occurs when checking rule preambles like "Before going to a dead end...". Such a preamble must be checked whatever the current action is – in many cases, it will not be a going action at all; which means that "room gone to", a value implied by the "to" clause, will not exist. If the type-checking were stricter, it would be a nuisance for authors, and instead we relax a little by accessing such variables using a more forgiving routine. Here, if a variable does not exist, we return 0 to mean that it can be read at M-stack position 0: this is the sentinel word, which is not part of any frame, and which contains 0. Thus the variable reads as if it is 0, the default for the kind of value "object", which is the KOV for action variables such as "room gone to".

The routine may only be used where the variable is being read, and never where it is to be written, of course: that would corrupt the sentinel.

```
[ MstVON id off pos;
    pos = Mstack_Seek_Frame(id);
    if (pos == 0) {
        return 0; ! word position 0 on the M-stack
    }
    if ((off<0) || (off >= MStack_Frame_Extent)) {
        print "Variable stack offset wrong: ", id, "/", off, " at ", pos, "^";
        rfalse;
    }
    return pos+off;
];
```

§**8. Rulebook Variables.**   Each rulebook has a slate of variables, usually empty, with ID number the same as the rulebook's own ID number. (Rulebook IDs number upwards from 0 in order of creation in the source text.) The associated creator functions, usually null, are stored in an array if there is no problem about memory usage, but with a switch statement if `MEMORY_ECONOMY` is in force; this costs a very small amount of time, but saves 1K of readable memory.

```
{-array:Code::Rulebooks::rulebook_var_creators}

[ MStack_CreateRBVars rb cr;
{-call:Code::Rulebooks::rulebook_var_creators_lookup}
    if (cr == 0) return;
    Mstack_Create_Frame(cr, rb);
];

[ MStack_DestroyRBVars rb cr;
{-call:Code::Rulebooks::rulebook_var_creators_lookup}
    if (cr == 0) return;
    Mstack_Destroy_Frame(cr, rb);
];
```

§**9. Activity Variables.**   Exactly the same goes for activity variables except that here the ID number is $10000 + N$, where N is the allocation ID of the activity. (This would fail if there were more than 10,000 rulebooks, but this is very difficult to see happening.)

```
{-array:Code::Activities::activity_var_creators}

[ MStack_CreateAVVars av cr;
    cr = activity_var_creators-->av;
    if (cr == 0) return;
    Mstack_Create_Frame(cr, av + 10000);
];

[ MStack_DestroyAVVars av cr;
    cr = activity_var_creators-->av;
    if (cr == 0) return;
    Mstack_Destroy_Frame(cr, av + 10000);
];
```