Activities Template B/acvt

Purpose

To run the necessary rulebooks to carry out an activity.

---

§**1. The Activities Stack.** Activities are more like nested function calls than independent processes; they finish in reverse order of starting, and are placed on a stack. This needs only very limited size in practice: 20 might seem a bit low, but making it much higher simply means that oddball bugs in the user's code – where activities recursively cause themselves ad infinitum – will be caught less efficiently.

```
Constant MAX_NESTED_ACTIVITIES = 20;
Global activities_sp = 0;
Array activities_stack --> MAX_NESTED_ACTIVITIES;
Array activity_parameters_stack --> MAX_NESTED_ACTIVITIES;
```

§**2. Rule Debugging Inhibition.** The output from RULES or RULES ALL becomes totally illegible if it is applied even to the activities printing names of objects, so this is inhibited when any such activity is running. `FixInhibitFlag` is called each time the stack changes and ensures that `inhibit_flag` has exactly this meaning.

```
Global inhibit_flag = 0;
Global saved_debug_rules = 0;
[ FixInhibitFlag n act inhibit_rule_debugging;
    for (n=0:n<activities_sp:n++) {
        act = activities_stack-->n;
        if (act == PRINTING_THE_NAME_ACT or PRINTING_THE_PLURAL_NAME_ACT or
            PRINTING_ROOM_DESC_DETAILS_ACT or LISTING_CONTENTS_ACT or
            GROUPING_TOGETHER_ACT) inhibit_rule_debugging = true;
    }
    if ((inhibit_flag == false) && (inhibit_rule_debugging)) {
        saved_debug_rules = debug_rules;
        debug_rules = 0;
    }
    if ((inhibit_flag) && (inhibit_rule_debugging == false)) {
        debug_rules = saved_debug_rules;
    }
    inhibit_flag = inhibit_rule_debugging;
];
```

§**3. Testing Activities.** The following tests whether a given activity `A` is currently running whose parameter-object matches description `desc`, where as usual the description is represented by a routine testing membership, and where zero `desc` means that any parameter is valid. Alternatively, we can require a specific parameter value of `val`.

```
[ TestActivity A desc val i;
    for (i=0:i<activities_sp:i++)
        if (activities_stack-->i == A) {
            if (desc) {
                if ((desc)(activity_parameters_stack-->i)) rtrue;
            } else if (val) {
                if (val == activity_parameters_stack-->i) rtrue;
            } else rtrue;
        }
    rfalse;
];
```

§**4. Emptiness.** An activity is defined by its three rulebooks: it is empty if they are all empty.

```
[ ActivityEmpty A x;
    x = Activity_before_rulebooks-->A;
    if (((rulebooks_array-->x)-->0) ~= NULL) rfalse;
    x = Activity_for_rulebooks-->A;
    if (((rulebooks_array-->x)-->0) ~= NULL) rfalse;
    x = Activity_after_rulebooks-->A;
    if (((rulebooks_array-->x)-->0) ~= NULL) rfalse;
    rtrue;
];
[ RulebookEmpty rb;
    if (((rulebooks_array-->rb)-->0) ~= NULL) rfalse;
    rtrue;
];
```

§**5. Process Activity Rulebook.** This is really much like processing any rulebook, except that `self` is temporarily set to the parameter, and is preserved by the process.

```
[ ProcessActivityRulebook rulebook parameter  rv;
    @push self;
    if (parameter) self = parameter;
    rv = ProcessRulebook(rulebook, parameter, true);
    @pull self;
    if (rv) rtrue;
    rfalse;
];
```

§**6. Carrying Out Activities.** This is a three-stage process; most activities are run by calling the following simple routine, but some are run by calling the three subroutines independently.

```
[ CarryOutActivity A o rv;
    BeginActivity(A, o);
    rv = ForActivity(A, o);
    EndActivity(A, o);
    return rv;
];
```

§**7. Begin.** Note that when an activity based on the conjectural "future action" is being run – in a few parser-related cases, that is – the identity of this action is put temporarily into `action`, and the current value saved while this takes place. That allows rules in the activity rulebooks to have preambles based on the current action, and yet be tested against what is not yet the current action.

```
[ BeginActivity A o x;
    if (activities_sp == MAX_NESTED_ACTIVITIES) return RunTimeProblem(RTP_TOOMANYACTS);
    activity_parameters_stack-->activities_sp = o;
    activities_stack-->(activities_sp++) = A;
    FixInhibitFlag();
    MStack_CreateAVVars(A);
    if (Activity_atb_rulebooks->A) { x = action; action = action_to_be; }
    o = ProcessActivityRulebook(Activity_before_rulebooks-->A, o);
    if (Activity_atb_rulebooks->A) action = x;
    return o;
];
```

§**8. For.**

```
[ ForActivity A o x;
    if (Activity_atb_rulebooks->A) { x = action; action = action_to_be; }
    o = ProcessActivityRulebook(Activity_for_rulebooks-->A, o);
    if (Activity_atb_rulebooks->A) action = x;
    return o;
];
```

§**9. End.**

```
[ EndActivity A o rv x;
    if ((activities_sp > 0) && (activities_stack-->(activities_sp-1) == A)) {
        if (Activity_atb_rulebooks->A) { x = action; action = action_to_be; }
        rv = ProcessActivityRulebook(Activity_after_rulebooks-->A, o);
        if (Activity_atb_rulebooks->A) action = x;
        activities_sp--; FixInhibitFlag();
        MStack_DestroyAVVars(A);
        return rv;
    }
    return RunTimeProblem(RTP_CANTABANDON);
];
```

§**10. Abandon.** For (very) rare cases where an activity must be abandoned midway; such an activity must be being run by calling the three stages individually, and `EndActivity` must not have been called yet.

```
[ AbandonActivity A o;
    if ((activities_sp > 0) && (activities_stack-->(activities_sp-1) == A)) {
        activities_sp--; FixInhibitFlag();
        MStack_DestroyAVVars(A);
        return;
    }
    return RunTimeProblem(RTP_CANTEND);
];
```