Rulebooks Template B/rbt

Purpose

To work through the rules in a rulebook until a decision is made.

§**1. Rule Change Stack.** The rulebook is a fundamental data structure in Inform 7: it's the basic way in which code is organised. The metaphor is that a rulebook is a loose-leaf ring-binder rather than a bound volume: so we can not only tell NI how to bind the rulebooks at compile time, we can also rearrange the pages during use at run-time.

We keep track of such run-time changes not by actually changing the rulebook arrays but by using the "rule change stack". This is a tally of any temporary abolition or modification of rules: thus, before using any rule from a rulebook we need to check that there is no note of its abolition. As the rule change stack gets larger, rule processing gets slower: this is why it's always more efficient to change rulebooks at compile time than at run-time, if there's a choice between the two.

The rule change stack contains 3-word (6 byte) records, a usage code and two operands (normally both rules).

```
Constant RULECHANGE_STACK_SIZE = 501;
Global rulechange_sp = 0;
Array rulechange_stack --> RULECHANGE_STACK_SIZE;

[ PushRuleChange usage rule1 rule2;
    if (rulechange_sp >= RULECHANGE_STACK_SIZE) return RunTimeProblem(RTP_RULESTACK);
    if ((rulechange_stack-->rulechange_sp == RS_SUCCEEDS or RS_FAILS) &&
        (KOVIsBlockValue(rulechange_stack-->(rulechange_sp+1))))
        BlkValueDestroy(rulechange_stack-->(rulechange_sp+2));
    if ((usage == RS_SUCCEEDS or RS_FAILS) && (KOVIsBlockValue(rule1)))
        rule2 = BlkValueCopy(BlkValueCreate(rule1), rule2);
    rulechange_stack-->rulechange_sp++ = usage;
    rulechange_stack-->rulechange_sp++ = rule1;
    rulechange_stack-->rulechange_sp++ = rule2;
];
```

§**2. Usage Codes.** The first word of each record indicates a usage type, of which one value is special and indicates the start of a frame. A new frame is opened on the stack each time a new “follow” occurs; recall that this processes a rulebook after consulting procedural rules, which may in turn modify the behaviour of other rules. These frame divisions, therefore, mark local scopes for modifications: anything from the top of the stack down to the topmost frame marker is currently in force.

There are then seven kinds of frame marking different ways in which rules have been modified; and three further values which indicate possible outcomes. In the following, we refer to the record as containing word 1, one of the usage codes below, word 2 and word 3.

(0) `RS_FRAME` marks the lowest point on the stack of a frame. Words 2 and 3 are not used.

(1) `RS_DONOTRUN` marks the rule or rulebook in word 1 as to be ignored.
(2) `RS_RUN` reinstates the rule or rulebook in word 1. This is useful only to reverse the effect of an `RS_DONOTRUN` frame.
(3) `RS_MOVEBEFORE` moves the rule/rulebook in word 1 to immediately before the rule/rulebook in word 2. If the latter is never invoked, nor is the former.
(4) `RS_MOVEAFTER` moves the rule/rulebook in word 1 to immediately after the rule/rulebook in word 2. If the latter is never invoked, nor is the former.
(5) `RS_DONOTUSE` marks the rule/rulebook in word 1 as to be invoked in the normal way, but then have its result (if any) ignored.
(6) `RS_USE` reverses the effect of an earlier `RS_DONOTUSE` frame.
(7) `RS_SUBSTITUTE` moves the rule/rulebook in word 1 so that it is invoked instead of the rule/rulebook in word 2. If the latter is never invoked, nor is the former.

(8) `RS_SUCCEEDS` occurs only exactly above the top of the stack: it is an ephemeral frame wiped out as soon as the stack is used again. It indicates that the most recent rule or rulebook processed ended in success. Word 2 is a flag: `true` means that a value was returned, `false` that it wasn’t. If this is `true` then word 3 contains the value.
(9) `RS_FAILS` is similar, but for a failure. Note that failures can also return values.
(10) `RS_NEITHER` is similar except that it cannot return any value, so that words 2 and 3 are meaningless.

```
Constant RS_FRAME       = -1;

Constant RS_DONOTRUN = 1;
Constant RS_RUN = 2;
Constant RS_MOVEBEFORE = 3;
Constant RS_MOVEAFTER = 4;
Constant RS_DONOTUSE = 5;
Constant RS_USE = 6;
Constant RS_SUBSTITUTE = 7;

Constant RS_SUCCEEDS = 8;
Constant RS_FAILS = 9;
Constant RS_NEITHER = 10;
```

§**3. Following.** At the I6 level, there are two ways to invoke a rulebook: we can "follow" it, or simply "process" it. The former is a grander and slightly slower method which contains the latter. The I7 language talks about "considering" and "abiding by" rules and rulebooks, but these aren't handled at the I6 level: they are both achieved by "process" and the difference between the two is a matter of what is done with the result. (See the Standard Rules for the definitions.)

To "follow" a rulebook, we start a new frame, process the procedural rules, then process the rulebook, then clear the frame back off the stack. (To avoid circularity, the procedural rulebook is the only one which is exempted from the procedural rules.)

```
Global rule_frames = 0; ! Number of frames currently in force
Constant MAX_SIMULTANEOUS_FRAMES = 20;

[ FollowRulebook rulebook parameter no_paragraph_skips  rv;
    @push self;
    if ((Protect_I7_Arrays-->0 ~= 16339) || (Protect_I7_Arrays-->1 ~= 12345)) {
        print "^^*** Fatal programming error: I7 arrays corrupted ***^^";
        @quit;
    }
    if (parameter) self = parameter;
    if (rulebook ~= PROCEDURAL_RB) BeginFollowRulebook();
    rv = ProcessRulebook(rulebook, parameter, no_paragraph_skips);
    if (rulebook ~= PROCEDURAL_RB) EndFollowRulebook();
    @pull self;
    if (rv) rtrue;
    rfalse;
];

[ BeginFollowRulebook;
    PushRuleChange(RS_FRAME, RS_FRAME, RS_FRAME);
    rule_frames++;
    if (rule_frames == MAX_SIMULTANEOUS_FRAMES) {
        RunTimeProblem(RTP_TOOMANYRULEBOOKS);
        rule_frames = -1; ! For recovery: this terminates rulebook processing
        return;
    }
    ProcessRulebook(PROCEDURAL_RB, 0, true);
];

[ EndFollowRulebook r x y;
    if (rulechange_stack-->rulechange_sp == RS_SUCCEEDS) r = 1;
    else if (rulechange_stack-->rulechange_sp == RS_FAILS) r = 0;
    else r = -1;
    if (r ~= -1) {
        x = rulechange_stack-->(rulechange_sp+1);
        y = rulechange_stack-->(rulechange_sp+2);
    }
    rule_frames--;
    while (rulechange_sp > 0) {
        rulechange_sp = rulechange_sp - 3;
        if (rulechange_stack-->rulechange_sp == RS_FRAME) break;
    }
    if (rulechange_sp == 0) rule_frames = 0;
    if (r == 1) rulechange_stack-->rulechange_sp = RS_SUCCEEDS;
    else if (r == 0) rulechange_stack-->rulechange_sp = RS_FAILS;
    if (r ~= -1) {
        rulechange_stack-->(rulechange_sp+1) = x;
        rulechange_stack-->(rulechange_sp+2) = y;
    }
];
```

§4. Processing. The routine `ProcessRulebook` is arguably the most important in the whole of I7. It does something essentially simple but has deceptively complicated implications. To complicate matters, it reuses its variables to keep the virtual machine stack usage to an absolute minimum – here we use 10 locals per call to `ProcessRulebook`, which is the fewest I can comfortably manage. In the early days of I7, stack usage became a serious issue since some forms of the Frotz Z-machine interpreter provided only 4K of stack by default. ("Only" 4K. In the mid-1980s, one of the obstacles facing IF authors at Infocom was the need to get the stack usage down to fewer than 600 bytes in order that the story file could be run on the smaller home computers of the day.)

`ProcessRulebook` takes three arguments, of which only the first is compulsory:

(a) The `rulebook` is an I7 value of kind "rule", which means it can be either the ID number of a rulebook – from 0 up to $N-1$, where N is the number of rulebooks compiled by NI, typically about 600 – or else the address of a routine representing an individual rule.
(b) The `parameter` supplied to the rulebook. Much as arguments can be supplied to a function in a conventional language's function call, so a parameter can be supplied whenever a rulebook is invoked.
(c) The value `bits` is initially a flag: if not supplied, this is `false`; if explicitly set `true`, then the rulebook is run with paragraph breaking suppressed. This is the process by which paragraph division points are placed between rules, so that if two rules both print text then a paragraph break appears between. While that is appropriate for rulebooks attached to actions or for "every turn" rules, it is disastrous for rulebooks attached to activities such as "printing the name of something". Once the routine is running, however, `bits` becomes a bitmap containing five flags, made up from the `RS_*_BIT` values defined below.

`ProcessRulebook` returns `R` if rule `R` in the rulebook (or rule) chose to "succeed" or "fail", and `false` if it made no choice. (To repeat: if the rule explicitly fails, then `ProcessRulebook` returns `true`. It's easy to write plausible-looking code which goes wrong because it assumes that the return value is success vs. failure.) The outcome of `ProcessRulebook` is lodged just above the top of this stack: thus the most recent rule or rulebook succeeded or failed if –

```
    (rulechange_stack-->rulechange_sp == RS_SUCCEEDS)
    (rulechange_stack-->rulechange_sp == RS_FAILS)
```

and otherwise there was no decision. If there was indeed a decision, then the second word of this record is a flag: `true` means that a value was returned, in which case the third word is that value, and `false` means that no value was returned, so that the third word is meaningless.

```
Constant RS_ACTIVE_BIT = 1;
Constant RS_MOVED_BIT = 2;
Constant RS_USERESULT_BIT = 4;
Constant RS_ACTIVITY = 8;
Constant RS_NOSKIPS = 16;
Constant RS_AFFECTED_BIT = 32;

Global process_rulebook_count; ! Depth of processing recursion
Global debugging_rules = false; ! Are we tracing rule invocations?

[ ProcessRulebook rulebook parameter bits rv
    x frame_base substituted_rule usage original_deadflag rbaddress ra acf gc ga;
    if (bits) bits = RS_ACTIVITY + RS_NOSKIPS;
    if (say__pc & PARA_NORULEBOOKBREAKS) bits = bits | RS_NOSKIPS;
    if (rule_frames<0) rfalse;
    if (parameter) parameter_object = parameter;
    for (x = rulechange_sp-3: x>=0: x = x - 3) {
        usage = rulechange_stack-->x;
        if (usage == RS_FRAME) { x=x+3; break; }
        if (rulechange_stack-->(x+1) == rulebook) {
            bits = bits | (RS_AFFECTED_BIT);
            if (usage == RS_MOVEBEFORE or RS_MOVEAFTER)
                bits = bits | (RS_MOVED_BIT);
        }
        if (rulechange_stack-->(x+2) == rulebook) {
            bits = bits | (RS_AFFECTED_BIT);
        }
    } if (x<0) x=0; frame_base = x;
    if ((bits & RS_MOVED_BIT) && (rv == false)) { rfalse; }
    ! rv was a call parameter: it's no longer needed and is now reused
    bits = bits | (RS_ACTIVE_BIT + RS_USERESULT_BIT);
    substituted_rule = rulebook; rv = 0;
    if (bits & RS_AFFECTED_BIT)
        for (: x<rulechange_sp: x = x + 3) {
            usage = rulechange_stack-->x;
            if (rulechange_stack-->(x+1) == rulebook) {
                if (usage == RS_DONOTRUN) bits = bits & (~RS_ACTIVE_BIT);
                if (usage == RS_RUN) bits = bits | (RS_ACTIVE_BIT);
                if (usage == RS_DONOTUSE) bits = bits & (~RS_USERESULT_BIT);
                if (usage == RS_USE) bits = bits | (RS_USERESULT_BIT);
                if (usage == RS_SUBSTITUTE)
                    substituted_rule = rulechange_stack-->(x+2);
            }
            if ((usage == RS_MOVEBEFORE) && (rulechange_stack-->(x+2) == rulebook)) {
                rv = ProcessRulebook(rulechange_stack-->(x+1),
                    parameter, (bits & RS_ACTIVITY ~= 0), true);
                if (rv) return rv;
            }
        }
    if ((bits & RS_ACTIVE_BIT) == 0) rfalse;
    ! We now reuse usage to keep the stack frame slimmer
    usage = debugging_rules;
    #ifndef MEMORY_ECONOMY;
    if (debugging_rules) DebugRulebooks(substituted_rule, parameter);
    #endif;
    ! (A routine defined in the I7 code generator)
    process_rulebook_count = process_rulebook_count + debugging_rules;
    if ((substituted_rule >= 0) && (substituted_rule < NUMBER_RULEBOOKS_CREATED)) {
        rbaddress = rulebooks_array-->substituted_rule;
        ra = rbaddress-->0; x = 0; original_deadflag = deadflag;
        if (ra ~= NULL) {
            acf = (bits & RS_ACTIVITY ~= 0);
            if (substituted_rule ~= ACTION_PROCESSING_RB) MStack_CreateRBVars(substituted_rule);
            if (ra == (-2)) {
                for (x=1: original_deadflag == deadflag: x++) {
                    ra = rbaddress-->x;
                    if (ra == NULL) break;
                    if (gc == 0) {
                        ga = ra; x++; gc = rbaddress-->x;
                        if ((gc<1) || (gc>31)) { gc = 1; x--; }
                        x++; ra = rbaddress-->x;
                    }
                    gc--;
                    if (ga ~= (-2) or action) continue;
                    if ((rv = (ProcessRulebook(ra, parameter, acf)))
                        && (bits & RS_USERESULT_BIT)) jump NonNullResult;
                }
            } else {
                for (: original_deadflag == deadflag: x++) {
                    ra = rbaddress-->x;
                    if (ra == NULL) break;
                    if ((rv = (ProcessRulebook(ra, parameter, acf)))
                        && (bits & RS_USERESULT_BIT)) jump NonNullResult;
                }
            }
            rv = 0;
            .NonNullResult;
            if (substituted_rule ~= ACTION_PROCESSING_RB) MStack_DestroyRBVars(substituted_rule);
        }
    } else {
        if ((say__p) && (bits & RS_NOSKIPS == 0)) DivideParagraphPoint();
        rv = indirect(substituted_rule);
        if (rv == 2) rv = reason_the_action_failed;
        else if (rv) rv = substituted_rule;
    }
    if (rv && (bits & RS_USERESULT_BIT)) {
        process_rulebook_count = process_rulebook_count - debugging_rules;
        if (process_rulebook_count < 0) process_rulebook_count = 0;
        #ifndef MEMORY_ECONOMY;
        if (debugging_rules) {
            spaces(2*process_rulebook_count);
            if (rulechange_stack-->rulechange_sp == RS_SUCCEEDS)
                print "[stopped: success]^";
            if (rulechange_stack-->rulechange_sp == RS_FAILS)
                print "[stopped: fail]^";
        }
        #endif;
        debugging_rules = usage;
        return rv;
    }
    if (bits & RS_AFFECTED_BIT)
        for (x=rulechange_sp-3: x>=frame_base: x = x-3) {
            if ((rulechange_stack-->x == RS_MOVEAFTER) &&
                (rulechange_stack-->(x+2) == rulebook)) {
                rv = ProcessRulebook(rulechange_stack-->(x+1),
                    parameter, (bits & RS_ACTIVITY ~= 0), true);
                if (rv) {
                    process_rulebook_count--;
                    debugging_rules = usage;
                    return rv;
                }
            }
        }
    process_rulebook_count = process_rulebook_count - debugging_rules;
    rulechange_stack-->rulechange_sp = 0;
    debugging_rules = usage;
    rfalse;
];
```

§5. Specifying Outcomes. The following provide ways for rules to succeed, fail or decline to do either. `SetRulebookOutcome` is a little different: it changes the outcome state of the most recent rule completed, not the current one. (It's used only when saving and restoring this in the actions machinery: rules should not call it.)

```
[ ActRulebookSucceeds rule_id;
    if (rule_id) reason_the_action_failed = rule_id;
    RulebookSucceeds();
];
[ ActRulebookFails rule_id;
    if (rule_id) reason_the_action_failed = rule_id;
    RulebookFails();
];
[ RulebookSucceeds weak_kind value;
    PushRuleChange(RS_SUCCEEDS, weak_kind, value);
    rulechange_sp = rulechange_sp - 3;
];
[ RulebookFails weak_kind value;
    PushRuleChange(RS_FAILS, weak_kind, value);
    rulechange_sp = rulechange_sp - 3;
];
[ RuleHasNoOutcome;
    PushRuleChange(RS_NEITHER, 0, 0);
    rulechange_sp = rulechange_sp - 3;
];
[ SetRulebookOutcome a;
    rulechange_stack-->rulechange_sp = a;
];
```

§6. Discovering Outcomes. And here is how to tell what the results were.

```
[ RulebookOutcome a;
    a = rulechange_stack-->rulechange_sp;
    if ((a == RS_FAILS) || (a == RS_SUCCEEDS)) return a;
    return RS_NEITHER;
];
[ RulebookFailed;
    if (rulechange_stack-->rulechange_sp == RS_FAILS) rtrue; rfalse;
];
[ RulebookSucceeded;
    if (rulechange_stack-->rulechange_sp == RS_SUCCEEDS) rtrue; rfalse;
];
[ ResultOfRule RB V F K a;
    if (RB) ProcessRulebook(RB, V, F);
    a = rulechange_stack-->rulechange_sp;
    if ((a == RS_FAILS) || (a == RS_SUCCEEDS)) {
        a = rulechange_stack-->(rulechange_sp + 1);
        if (a) return rulechange_stack-->(rulechange_sp + 2);
    }
    if (K) return DefaultValueOfKOV(K);
    return 0;
];
```

§7. Procedural Rule Changes. The following routines provide a sort of rule-changing API, and correspond closely to the I7 phrases documented in *Writing with Inform*, so they won't be discussed in any detail here.

```
Global DITS_said = false;
[ SuppressRule rule;
    if (rule == TURN_SEQUENCE_RB) {
        if (DITS_said == false) RunTimeProblem(RTP_DONTIGNORETURNSEQUENCE);
        DITS_said = true;
    } else PushRuleChange(RS_DONOTRUN, rule, 0);
];
[ ReinstateRule rule; PushRuleChange(RS_RUN, rule, 0); ];
[ DonotuseRule rule; PushRuleChange(RS_DONOTUSE, rule, 0); ];
[ UseRule rule; PushRuleChange(RS_USE, rule, 0); ];
[ SubstituteRule rule1 rule2; PushRuleChange(RS_SUBSTITUTE, rule2, rule1); ];
[ MoveRuleBefore rule1 rule2; PushRuleChange(RS_MOVEBEFORE, rule1, rule2); ];
[ MoveRuleAfter rule1 rule2; PushRuleChange(RS_MOVEAFTER, rule1, rule2); ];
```

§8. Printing Rule Names. This is the I6 printing rule used for a value of kind "rule", which as noted above can either be rulebook ID numbers in the range 0 to $N - 1$ or are addresses of individual rules.

Names of rules and rulebooks take up a fair amount of space, and one of the main memory economies enforced by the "Use memory economy" option is to omit the necessary arrays. (It's not the text which is the problem so much as the table of addresses pointing to that text, which has to live in precious readable memory on the Z-machine.)

```
#IFNDEF MEMORY_ECONOMY;
{-array:Code::Phrases::RulebookNames}
#ENDIF; ! MEMORY_ECONOMY

[ RulePrintingRule R p1;
#ifndef MEMORY_ECONOMY;
    if ((R>=0) && (R<NUMBER_RULEBOOKS_CREATED)) {
        print (string) (RulebookNames-->R);
    } else {
{-call:Code::Phrases::compile_rule_printing_switch}
        print "(nameless rule at address ", R, ")";
    }
#ifnot;
    if ((R>=0) && (R<NUMBER_RULEBOOKS_CREATED)) {
        print "(rulebook ", R, ")";
    } else {
        print "(rule at address ", R, ")";
    }
#endif;
];
```

§9. Debugging. Two modest routines to print out the names of rules and rulebooks when they occur, in so far as memory economy allows this.

```
[ DebugRulebooks subs parameter i;
    spaces(2*process_rulebook_count);
    print "[", (RulePrintingRule) subs;
    if (parameter) print " / on O", parameter;
    print "]^";
];

[ DB_Rule R N blocked;
    if (R==0) return;
    print "[Rule ~", (RulePrintingRule) R, "~ ";
    #ifdef NUMBERED_RULES; print "(", N, ") "; #endif;
    if (blocked == false) "applies.]";
    "does not apply.]";
];
```