Actions Template B/actt

Purpose

To try actions by people in the model world, processing the necessary rulebooks.

§**1. Summary.** To review: an action is an impulse to do something by a person in the model world. Commands such as DROP POTATO are converted into actions ("dropping the Idaho potato"); sometimes they succeed, sometimes they fail. While they run, the fairly complicated details are stored in a suite of I6 global variables such as `actor`, `noun`, `inp1`, and so on (see "OrderOfPlay.i6t" for details); the running of an action is mainly a matter of processing many rulebooks, chief among them they "action processing rules".

In general, actions can come from five different sources:

(i) As a result of parsing the player's command: there are actually two ways this can happen, one if the command calls for a single action, and another if it calls for a whole run of them (like TAKE ALL). See the rules in "OrderOfPlay.i6t".
(ii) From an I7 "try" phrase, in which case `TryAction` is called.
(iii) From an I6 angle-bracket-notation such as `<Wait>`, though this is a syntax which is deprecated now, and is never normally used in I7. The I6 compiler converts such a syntax into a call to the `R_Process` below.
(iv) Through conversion of an existing action. For instance, "removing the cup from the table" is converted in the Standard Rules to "taking the cup". This is done via a routine called `GVS_Convert`.
(v) When a request is successful, the "carry out requested actions" rule turns the original action – a request by the player, such as is produced by CHOPIN, PLAY POLONAISE – into an action by the person asked, such as "Chopin playing the Polonaise".

Certain exceptional cases can arise for other reasons:

(vi) Implicit taking actions are generated by the "carrying requirements rule" when the actor tries something which requires him to be holding an item which he can see, but is not currently holding.
(vii) A partial but intentionally incomplete form of the "looking" action is generated when describing the new location at the end of a "going" action.

In every case except (vii), the action is carried out by `BeginAction`, the single routine which unifies all of these approaches. Except the last one.

This segment of the template is divided into two: first, the I6 code needed for (i) to (vii), the alternative ways for actions to begin; and secondly the common machinery into which all actions eventually pass.

§2. Action Data. This is perhaps a good place to document the `ActionData` array, a convenient table of metadata about the actions. Since this is compiled by NI, the following structure can't be modified here without making matching changes in NI. `ActionData` is an I6 `table` containing a series of fixed-length records, one on each action.

The routine `FindAction` locates the record in this table for a given action number, returning its word offset within the table: the argument −1 means "the current action".

```
Constant AD_ACTION = 0; ! The I6 action number (0 to 4095)
Constant AD_REQUIREMENTS = 1; ! Such as requiring light; a bitmap, see below
Constant AD_NOUN_KOV = 2; ! Kind of value of the first noun
Constant AD_SECOND_KOV = 3; ! Kind of value of the second noun
Constant AD_VARIABLES_CREATOR = 4; ! Routine to initialise variables owned
Constant AD_VARIABLES_ID = 5; ! Frame ID for variables owned by action
Constant AD_RECORD_SIZE = 6;
[ FindAction fa t;
    if (fa == -1) fa = action;
    t = 1;
    while (t <= ActionData-->0) {
        if (fa == ActionData-->t) return t;
        t = t + AD_RECORD_SIZE;
    }
    rfalse;
];
[ ActionNumberIndexed i;
    if ((i>=0) && (i < AD_RECORDS)) return ActionData-->(i*AD_RECORD_SIZE + AD_ACTION + 1);
    return 0;
];
```

§3. Requirements Bitmap. As noted above, the `AD_REQUIREMENTS` field is a bitmap of flags for various possible action requirements:

```
Constant TOUCH_NOUN_ABIT   = $$00000001;
Constant TOUCH_SECOND_ABIT = $$00000010;
Constant LIGHT_ABIT        = $$00000100;
Constant NEED_NOUN_ABIT    = $$00001000;
Constant NEED_SECOND_ABIT  = $$00010000;
Constant OUT_OF_WORLD_ABIT = $$00100000;
Constant CARRY_NOUN_ABIT   = $$01000000;
Constant CARRY_SECOND_ABIT = $$10000000;
[ NeedToCarryNoun;       return TestActionMask(CARRY_NOUN_ABIT); ];
[ NeedToCarrySecondNoun; return TestActionMask(CARRY_SECOND_ABIT); ];
[ NeedToTouchNoun;       return TestActionMask(TOUCH_NOUN_ABIT); ];
[ NeedToTouchSecondNoun; return TestActionMask(TOUCH_SECOND_ABIT); ];
[ NeedLightForAction;    return TestActionMask(LIGHT_ABIT); ];
[ TestActionMask match mask at;
    at = FindAction(-1);
    if (at == 0) rfalse;
    mask = ActionData-->(at+AD_REQUIREMENTS);
    if (mask & match) rtrue;
    rfalse;
];
```

§**4. Try Action.** This is method (ii) in the summary above.

```
[ TryAction req by ac n s stora smeta tbits saved_command text_of_command;
    if (stora) return STORED_ACTION_TY_New(ac, n, s, by, req, stora);
    tbits = req & (16+32);
    req = req & 1;
    @push actor; @push act_requester; @push inp1; @push inp2;
    @push parsed_number; smeta = meta;
    actor = by; if (req) act_requester = player; else act_requester = 0;

    by = FindAction(ac);
    if (by) {
        if (ActionData-->(by+AD_NOUN_KOV) == OBJECT_TY) inp1 = n;
        else { inp1 = 1; parsed_number = n; }
        if (ActionData-->(by+AD_SECOND_KOV) == OBJECT_TY) inp2 = s;
        else { inp2 = 1; parsed_number = s; }
        if (((ActionData-->(by+AD_NOUN_KOV) == UNDERSTANDING_TY) ||
            (ActionData-->(by+AD_SECOND_KOV) == UNDERSTANDING_TY)) && (tbits)) {
            saved_command = INDEXED_TEXT_TY_Create();
            INDEXED_TEXT_TY_Cast(players_command, SNIPPET_TY, saved_command);
            text_of_command = INDEXED_TEXT_TY_Create();
            INDEXED_TEXT_TY_Cast(parsed_number, TEXT_TY, text_of_command);
            SetPlayersCommand(text_of_command);
            if (tbits == 16) {
                n = players_command; inp1 = 1; parsed_number = players_command;
            } else {
                s = players_command; inp2 = 1; parsed_number = players_command;
            }
            BlkFree(text_of_command);
            @push consult_from; @push consult_words;
            consult_from = 1; consult_words = parsed_number - 100;
        }
    }

    BeginAction(ac, n, s, 0, true);

    if (saved_command) {
        @pull consult_words; @pull consult_from;
        SetPlayersCommand(saved_command);
        BlkFree(saved_command);
    }

    meta = smeta; @pull parsed_number;
    @pull inp2; @pull inp1; @pull act_requester; @pull actor;
    TrackActions(true, smeta);
];
```

§5. I6 Angle Brackets. This is method (iii) in the summary above. The routine here has slightly odd conventions and a curious name which would take too long to explain: neither can be changed without amending the veneer code within the I6 compiler.

```
[ R_Process a i j;
    @push inp1; @push inp2;
    inp1 = i; inp2 = j; BeginAction(a, i, j);
    @pull inp2; @pull inp1;
];
```

§6. Conversion. This is method (iv) in the summary above.

```
Global converted_action_outcome = -1;
[ GVS_Convert ac n s;
    converted_action_outcome = BeginAction(ac, n, s);
    rtrue;
];
[ ConvertToGoingWithPush i oldrm newrm infl;
    i=noun;
    if (IndirectlyContains(noun, actor) == false) { move i to actor; infl = true; }
    move_pushing = i;
    oldrm = LocationOf(noun);
    BeginAction(##Go, second);
    newrm = LocationOf(actor);
    move_pushing = nothing; move i to newrm;
    if (newrm ~= oldrm) {
        if (IndirectlyContains(i, player)) TryAction(0, player, ##Look, 0, 0);
        RulebookSucceeds();
    } else RulebookFails();
];
```

§7. Implicit Take. This is method (vi) in the summary above.

```
[ ImplicitTake obj ks;
    if (actor == player) L__M(##Miscellany, 69, obj);
    else L__M(##Miscellany, 68, obj);
    ClearParagraphing();
    @push keep_silent; keep_silent = true;
    if (act_requester) TryAction(true, actor, ##Take, obj, nothing);
    else TryAction(false, actor, ##Take, obj, nothing);
    @pull keep_silent;
    if (obj in actor) rtrue;
    rfalse;
];
```

§8. Look After Going. This is method (vii) in the summary above.

Fundamentally, room descriptions arise through looking actions, but they are also printed after successful going actions, with a special form of paragraph break (see "Printing.i6t" for an explanation of this). Room descriptions through looking are always given in full, unless we have SUPERBRIEF mode set.

```
[ LookAfterGoing;
    GoingLookBreak();
    AbbreviatedRoomDescription();
];
```

§9. Abbreviated Room Description. This is used when we want a room description with the same abbreviation conventions as after a going action, and we don't quite want a looking action fully to take place. We nevertheless want to be sure that the action variables for looking exist, and in particular, we want to set the "room-describing action" variable to the action which was prevailing when the room description was called for. We also set "abbreviated form allowed" to "true": when the ordinary looking action is running, this is "false".

The actual description occurs during `LookSub`, which is the specific action processing stage for the "looking" action: thus, we use the check, carry out, after and report rules as if we were "looking", but are unaffected by before or instead rules.

Uniquely, this pseudo-action does not use `BeginAction`: it works only through the specific action processing rules, not the main action-processing ones, though that is not easy to see from the code below because it is hidden in the call to `LookSub`. The `-Sub` suffix is an I6 usage identifying this as the routine to go along with the action `##Look`, and so it is, but it looks nothing like the `LookSub` of the old I6 library. NI compiles `-Sub` routines like so:

```
    [ LookSub; return GenericVerbSub(153,154,155); ];
```

(with whatever rulebook numbers are appropriate). `GenericVerbSub` then runs through the specific action processing stage.

```
[ AbbreviatedRoomDescription  prior_action pos frame_id;
    prior_action = action;
    action = ##Look;
    pos = FindAction(##Look);
    if ((pos) && (ActionData-->(pos+AD_VARIABLES_CREATOR))) {
        frame_id = ActionData-->(pos+AD_VARIABLES_ID);
        Mstack_Create_Frame(ActionData-->(pos+AD_VARIABLES_CREATOR), frame_id);
        ProcessRulebook(SETTING_ACTION_VARIABLES_RB);
        (MStack-->MstVO(frame_id, 0)) = prior_action; ! "room-describing action"
        (MStack-->MstVO(frame_id, 1)) = true; ! "abbreviated form allowed"
    }
    LookSub(); ! The I6 verb routine for "looking"
    if (frame_id) Mstack_Destroy_Frame(ActionData-->(pos+AD_VARIABLES_CREATOR), frame_id);
    action = prior_action;
];
```

§**10. Begin Action.** We now begin the second half of the segment: the machinery which handles all actions.

The significance of 4096 here is that this is how I6 distinguishes genuine actions – numbered upwards in order of creation – from what I6 calls "fake actions" – numbered upwards from 4096. Fake actions are hardly used at all in I7, and certainly shouldn't get here, but it's possible nonetheless using I6 angled-brackets, so... In other respects all we do is to save details of whatever current action is happening onto the stack, and then call `ActionPrimitive`.

```
[ BeginAction a n s moi notrack  rv;
    ChronologyPoint();
    @push action; @push noun; @push second; @push self; @push multiple_object_item;
    action = a; noun = n; second = s; self = noun; multiple_object_item = moi;
    if (action < 4096) rv = ActionPrimitive();
    @pull multiple_object_item; @pull self; @pull second; @pull noun; @pull action;
    if (notrack == false) TrackActions(true, meta);
    return rv;
];
```

§**11. Action Primitive.** This is somewhat different from the I6 library counterpart which gives it its name, but the idea is the same. It has no arguments at all: everything it needs to know is now stored in global variables. The routine looks long, but really contains little: it's all just book-keeping, printing debugging information if ACTIONS is in force, etc., with all of the actual work delegated to the action processing rulebook.

We use a rather sneaky device to handle out-of-world actions, those for which the `meta` flag is set: we make it look to the system as if the "action processing rulebook" is being followed, so that all its variables are created and placed in scope, but at the crucial moment we descend to the specific action processing rules directly instead of processing the main rulebook. This is what short-circuits out of world actions and protects them from before and instead rules: see the Standard Rules for more discussion of this.

```
[ ActionPrimitive  rv p1 p2 p3 p4 p5 frame_id;
    MStack_CreateRBVars(ACTION_PROCESSING_RB);
    if ((keep_silent == false) && (multiflag == false)) DivideParagraphPoint();
    reason_the_action_failed = 0;
    frame_id = -1;
    p1 = FindAction(action);
    if ((p1) && (ActionData-->(p1+AD_VARIABLES_CREATOR))) {
        frame_id = ActionData-->(p1+AD_VARIABLES_ID);
        Mstack_Create_Frame(ActionData-->(p1+AD_VARIABLES_CREATOR), frame_id);
    }
    if (ActionVariablesNotTypeSafe()) {
        if (frame_id ~= -1)
            Mstack_Destroy_Frame(ActionData-->(p1+AD_VARIABLES_CREATOR), frame_id);
        MStack_DestroyRBVars(ACTION_PROCESSING_RB);
        return;
    }
    ProcessRulebook(SETTING_ACTION_VARIABLES_RB);
    #IFDEF DEBUG;
    if ((trace_actions) && (FindAction(-1))) {
        print "["; p1=actor; p2=act_requester; p3=action; p4=noun; p5=second;
        DB_Action(p1,p2,p3,p4,p5);
        print "]^"; ClearParagraphing();
    }
    ++debug_rule_nesting;
    #ENDIF;
    TrackActions(false, meta);
    BeginFollowRulebook();
    if ((meta) && (actor ~= player)) { L__M(##Miscellany, 74, actor); rv = RS_FAILS; }
    else if (meta) { DESCEND_TO_SPECIFIC_ACTION_R(); rv = RulebookOutcome(); }
    else { ProcessRulebook(ACTION_PROCESSING_RB); rv = RulebookOutcome(); }
    #IFDEF DEBUG;
    --debug_rule_nesting;
    if ((trace_actions) && (FindAction(-1))) {
        print "["; DB_Action(p1,p2,p3,p4,p5); print " - ";
        switch (rv) {
            RS_SUCCEEDS: print "succeeded";
            RS_FAILS: print "failed";
                #IFNDEF MEMORY_ECONOMY;
                if (reason_the_action_failed)
                    print " the ",
                        (RulePrintingRule) reason_the_action_failed;
                #ENDIF;
            default: print "ended without result";
        }
        print "]^"; say__p = 1;
        SetRulebookOutcome(rv); ! In case disturbed by printing activities
    }
    #ENDIF;
    if (rv == RS_SUCCEEDS) UpdateActionBitmap();
    EndFollowRulebook();
    if (frame_id ~= -1) {
        p1 = FindAction(action);
        Mstack_Destroy_Frame(ActionData-->(p1+AD_VARIABLES_CREATOR), frame_id);
    }
    MStack_DestroyRBVars(ACTION_PROCESSING_RB);
    if ((keep_silent == false) && (multiflag == false)) DivideParagraphPoint();
    if (rv == RS_SUCCEEDS) rtrue;
    rfalse;
];
```

§**12. Type Safety.** Some basic action requirements have to be met before we can go any further: if they aren't, then it isn't type-safe even to run the action processing rulebook.

(i) For an out of world action, we set the `meta` flag. Otherwise:
(ii) If either the noun or second noun is a topic, then this is an action arising from parsing (such actions do not arise through the "try" phrase, unless by stored actions in which case this has all happened before and doesn't need to be done again) – the parser places details of which words make up the topic in the I6 global variables `consult_words` and `consult_from`. We convert them to a valid I7 snippet value.
(iii) If either the first or second noun is supposed to be an object but seems here to be a value, or vice versa, we stop with a parser error. (This should be fairly difficult to provoke: NI's type-checking will make it difficult to arrange without I6 subterfuges.)
(iv) If either the first or second noun is supposed to be an object and required to exist, yet is missing, we use the "supplying a missing noun" or "supplying a missing second noun" activities to fill the void.

We return `true` if type safety is violated, `false` if all is well.

```
[ ActionVariablesNotTypeSafe mask noun_kova second_kova at;
    at = FindAction(-1); if (at == 0) rfalse; ! For any I6-defined actions

    noun_kova = ActionData-->(at+AD_NOUN_KOV);
    second_kova = ActionData-->(at+AD_SECOND_KOV);

    !print "at = ", at, " nst = ", noun_kova, "^";
    !print "consult_from = ", consult_from, " consult_words = ", consult_from, "^";
    !print "inp1 = ", inp1, " noun = ", noun, "^";
    !print "inp2 = ", inp2, " second = ", second, "^";
    !print "sst = ", second_kova, "^";

    if (noun_kova == SNIPPET_TY or UNDERSTANDING_TY) {
        if (inp1 ~= 1) { inp2 = inp1; second = noun; }
        parsed_number = 100*consult_from + consult_words;
        inp1 = 1; noun = nothing; ! noun = parsed_number;
    }
    if (second_kova == SNIPPET_TY or UNDERSTANDING_TY) {
        parsed_number = 100*consult_from + consult_words;
        inp2 = 1; second = nothing; ! second = parsed_number;
    }

    mask = ActionData-->(at+AD_REQUIREMENTS);
    if (mask & OUT_OF_WORLD_ABIT) { meta = 1; rfalse; }

    if (inp1 == 1) {
        if (noun_kova == OBJECT_TY) {
            return L__M(##Miscellany, 61); }
    } else {
        if (noun_kova ~= OBJECT_TY) {
            return L__M(##Miscellany, 62); }
        if ((mask & NEED_NOUN_ABIT) && (noun == nothing)) {
            @push act_requester; act_requester = nothing;
            CarryOutActivity(SUPPLYING_A_MISSING_NOUN_ACT);
            @pull act_requester;
            if (noun == nothing) {
                if (say__p) rtrue;
                return L__M(##Miscellany, 59);
            }
        }
        if (((mask & NEED_NOUN_ABIT) == 0) && (noun ~= nothing)) {
            return L__M(##Miscellany, 60); }
    }
    if (inp2 == 1) {
        if (second_kova == OBJECT_TY) {
            return L__M(##Miscellany, 63); }
    } else {
        if (second_kova ~= OBJECT_TY) {
            return L__M(##Miscellany, 64); }
        if ((mask & NEED_SECOND_ABIT) && (second == nothing)) {
            @push act_requester; act_requester = nothing;
            CarryOutActivity(SUPPLYING_A_MISSING_SECOND_ACT);
            @pull act_requester;
            if (second == nothing) {
                if (say__p) rtrue;
                return L__M(##Miscellany, 65);
            }
        }
        if (((mask & NEED_SECOND_ABIT) == 0) && (second ~= nothing)) {
            return L__M(##Miscellany, 66); }
    }
    rfalse;
];
```

§**13. Basic Visibility Rule.** This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

Note that this rule only blocks the player from acting in darkness: this is because light is only reckoned from the player's perspective in any case, so that it would be unfair to apply the rule to any other person.

```
[ BASIC_VISIBILITY_R;
    if (act_requester) rfalse;
    if ((NeedLightForAction()) &&
        (actor == player) &&
        (ProcessRulebook(VISIBLE_RB)) &&
        (RulebookSucceeded())) {
        BeginActivity(REFUSAL_TO_ACT_IN_DARK_ACT);
        if (ForActivity(REFUSAL_TO_ACT_IN_DARK_ACT)==false) L__M(##Miscellany, 17);
        EndActivity(REFUSAL_TO_ACT_IN_DARK_ACT);
        reason_the_action_failed = BASIC_VISIBILITY_R;
        RulebookFails();
        rtrue;
    }
    rfalse;
];
```

§14. Basic Accessibility Rule. This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

```
[ BASIC_ACCESSIBILITY_R mask at;
    if (act_requester) rfalse;
    at = FindAction(-1);
    if (at == 0) rfalse;
    mask = ActionData-->(at+AD_REQUIREMENTS);
    if ((mask & TOUCH_NOUN_ABIT) && noun && (inp1 ~= 1)) {
        if (noun ofclass K3_direction) {
            RulebookFails();
            reason_the_action_failed = BASIC_ACCESSIBILITY_R;
            if (actor~=player) rtrue;
            return L__M(##Miscellany, 67);
        }
        if (ObjectIsUntouchable(noun, (actor~=player), FALSE, actor)) {
            RulebookFails();
            reason_the_action_failed = BASIC_ACCESSIBILITY_R;
            rtrue;
        }
    }
    if ((mask & TOUCH_SECOND_ABIT) && second && (inp2 ~= 1)) {
        if (second ofclass K3_direction) {
            RulebookFails();
            reason_the_action_failed = BASIC_ACCESSIBILITY_R;
            if (actor~=player) rtrue;
            return L__M(##Miscellany, 67);
        }
        if (ObjectIsUntouchable(second, (actor~=player), FALSE, actor)) {
            RulebookFails();
            reason_the_action_failed = BASIC_ACCESSIBILITY_R;
            rtrue;
        }
    }
    rfalse;
];
```

§**15. Carrying Requirements Rule.** This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

```
[ CARRYING_REQUIREMENTS_R mask at;
    at = FindAction(-1);
    if (at == 0) rfalse;
    mask = ActionData-->(at+AD_REQUIREMENTS);
    if ((mask & TOUCH_NOUN_ABIT) && noun && (inp1 ~= 1)) {
        if ((mask & CARRY_NOUN_ABIT) && (noun notin actor)) {
            BeginActivity(IMPLICITLY_TAKING_ACT, noun);
            if (ForActivity(IMPLICITLY_TAKING_ACT, noun)==false)
                ImplicitTake(noun);
            EndActivity(IMPLICITLY_TAKING_ACT, noun);
            !if (act_requester) rfalse;
            if (noun notin actor) {
                RulebookFails();
                reason_the_action_failed = CARRYING_REQUIREMENTS_R;
                rtrue;
            }
        }
    }
    if ((mask & TOUCH_SECOND_ABIT) && second && (inp2 ~= 1)) {
        if ((mask & CARRY_SECOND_ABIT) && (second notin actor)) {
            BeginActivity(IMPLICITLY_TAKING_ACT, second);
            if (ForActivity(IMPLICITLY_TAKING_ACT, second)==false)
                ImplicitTake(second);
            EndActivity(IMPLICITLY_TAKING_ACT, second);
            !if (act_requester) rfalse;
            if (second notin actor) {
                RulebookFails();
                reason_the_action_failed = CARRYING_REQUIREMENTS_R;
                rtrue;
            }
        }
    }
    rfalse;
];
```

§**16. Requested Actions Require Persuasion Rule.** This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

```
[ REQUESTED_ACTIONS_REQUIRE_R rv;
    if ((actor ~= player) && (act_requester)) {
        @push say__p;
        say__p = 0;
        rv = ProcessRulebook(PERSUADE_RB);
        if (RulebookSucceeded() == false) {
            if ((deadflag == false) && (say__p == FALSE)) L__M(##Miscellany, 72, actor);
            ActRulebookFails(rv); rtrue;
        }
        @pull say__p;
    }
    rfalse;
];
```

§**17. Carry Out Requested Actions Rule.** This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

```
[ CARRY_OUT_REQUESTED_ACTIONS_R rv;
    if ((actor ~= player) && (act_requester)) {
        @push act_requester; act_requester = nothing;
        rv = BeginAction(action, noun, second);
        if (((meta) || (rv == false)) && (deadflag == false)) {
            if (ProcessRulebook(UNSUCCESSFUL_ATTEMPT_RB) == false) L__M(##Miscellany, 58);
        }
        @pull act_requester;
        ActRulebookSucceeds();
        rtrue;
    }
    rfalse;
];
```

§**18. Generic Verb Subroutine.** In I6, actions are carried out by routines with names like `TakeSub`, consisting of `-Sub` tacked on to the action name `Take`. `Sub` stands for "subroutine": this is all a convention going back to Inform 1, which was in 1993 practically an assembler. In the I6 code generated by I7, every `-Sub` routine corresponding to an I7 action consists only of a call to `GenericVerbSub` which specifies the three rulebooks it owns: its check, carry out and report rulebooks.

```
Array Details_of_Specific_Action-->5;

[ GenericVerbSub ch co re vis rv;
    @push converted_action_outcome;
    converted_action_outcome = -1;

    Details_of_Specific_Action-->0 = true;
    if (meta) Details_of_Specific_Action-->0 = false;
    Details_of_Specific_Action-->1 = keep_silent;
    Details_of_Specific_Action-->2 = ch; ! Check rules for the action
    Details_of_Specific_Action-->3 = co; ! Carry out rules for the action
    Details_of_Specific_Action-->4 = re; ! Report rules for the action

    ProcessRulebook(SPECIFIC_ACTION_PROCESSING_RB, 0, true);
    if ((RulebookFailed()) && (converted_action_outcome == 1)) ActRulebookSucceeds();
    @pull converted_action_outcome;
    rtrue;
];
```

§**19. Work Out Details Of Specific Action Rule.** This is one of the I6 primitive rules in the specific action processing rulebook, and it's basically a trick to allow information known to the `GenericVerbSub` routine to be passed down as rulebook variables for the specific action-processing rules – in effect allowing us to pass not one but five parameters to the rulebook: the out-of-world and silence flags, plus the three specific rulebooks needed to process the action.

```
[ WORK_OUT_DETAILS_OF_SPECIFIC_R;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 0) = Details_of_Specific_Action-->0;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 1) = Details_of_Specific_Action-->1;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 2) = Details_of_Specific_Action-->2;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 3) = Details_of_Specific_Action-->3;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 4) = Details_of_Specific_Action-->4;
    rfalse;
];
```

§**20. Actions Bitmap.** This is a fairly large bitmap recording which actions have succeeded thus far on which nouns. It was to some extent an early attempt at implementing a past-tense system; I'm not at all sure it was successful, since it is hindered by certain restrictions – it only records action/noun combinations, for instance, and the notion of "success" is a vexed one for actions anyway. There is a clearly defined meaning, but it doesn't always correspond to what the user might expect, which is unfortunate.

```
[ TestActionBitmap obj act i j k bitmap;
    if (obj == nothing) bitmap = ActionHappened;
    else {
        if (~~(obj provides action_bitmap)) rfalse;
        bitmap = obj.&action_bitmap;
    }
    if (act == -1) return (((bitmap->0) & 1) ~= 0);
    for (i=0, k=2; i<ActionCount; i++) {
        if (act == ActionCoding-->i) {
            return (((bitmap->j) & k) ~= 0);
        }
        k = k*2; if (k == 256) { k = 1; j++; }
    }
    rfalse;
];
[ UpdateActionBitmap;
    SetActionBitmap(noun, action);
    if (action == ##Go) SetActionBitmap(location, ##Enter);
];
[ SetActionBitmap obj act i j k bitmap;
    for (i=0, k=2; i<ActionCount; i++) {
        if (act == ActionCoding-->i) {
            if (obj provides action_bitmap) {
                bitmap = obj.&action_bitmap;
                bitmap->0 = (bitmap->0) | 1;
                bitmap->j = (bitmap->j) | k;
            }
            ActionHappened->0 = (ActionHappened->0) | 1;
            ActionHappened->j = (ActionHappened->j) | k;
        }
        k = k*2; if (k == 256) { k = 1; j++; }
    }
];
```

§21. Printing Actions. This is really for debugging purposes, but also provides us with a way to print a stored action, for instance, or to print an action name value. (For instance, printing an action name might result in "taking"; printing a whole action might produce "Henry taking the grapefruit".)

```
[ SayActionName act; DB_Action(0, 0, act, 0, 0, 2); ];
[ DA_Name n; if (n ofclass K3_direction) print (name) n; else print (the) n; ];
[ DA_Topic x a b c d i cf cw;
    cw = x%100; cf = x/100;
    print "~";
    for (a=cf:d<cw:d++,a++) {
        wn = a; b = WordAddress(a); c = WordLength(a);
        for (i=b:i<b+c:i++) {
            print (char) 0->i;
        }
        if (d<cw-1) print " ";
    }
    print "~";
];
[ DA_Number n; print n; ];
[ DA_TruthState n; if (n==0) print "false"; else print "true"; ];
[ DB_Action ac acr act n s for_say t at l j v c clc;
    if ((for_say == 0) && (debug_rule_nesting > 0))
        print "(", debug_rule_nesting, ") ";
    if ((ac ~= player) && (for_say ~= 2)) {
        if (acr) print "asking ", (the) ac, " to try ";
        else print (the) ac, " ";
    }
    DB_Action_Details(act, n, s, for_say);
    if ((keep_silent) && (for_say == 0)) print " - silently";
];
```