OrderOfPlay Template B/ordt

Purpose

The sequence of events in play: the Main routine which runs the startup rulebook, the turn sequence rulebook and the shutdown rulebook; and most of the I6 definitions of primitive rules in those rulebooks.

---

§**1. Main.** This is where every I6 story file begins execution: it can end either by returning, or by a `quit` statement or equivalent opcode. (In I7 this does indeed happen when the quitting the game action is carried out, or when QUIT is typed as a reply to the final question; it's only if the user has altered the shutdown rulebook that we might ever actually return from `Main`.) The return value from `Main` is not meaningful.

The `EarlyInTurnSequence` flag is used to enforce the requirement that the "parse command rule" and "generate action rule" do nothing unless the turn sequence rulebook is being followed directly from `Main`, an anomaly explained in the Standard Rules.

```
Global EarlyInTurnSequence;

[ Main;
    #ifdef TARGET_ZCODE; max_z_object = #largest_object - 255; #endif;
    ProcessRulebook(STARTUP_RB);
    #ifdef DEBUG; InternalTestCases(); #endif;
    while (true) {
        while (deadflag == false) {
            EarlyInTurnSequence = true;
            action = ##Wait; meta = false; noun = nothing; second = nothing;
            actor = player;
            FollowRulebook(TURN_SEQUENCE_RB);
        }
        if (FollowRulebook(SHUTDOWN_RB) == false) return;
    }
];
```

§**2. Virtual Machine Startup Rule.** Note that we consider the three rulebooks for the "starting the virtual machine" activity, but do not formally carry it out, because that might invoke procedural rules: this early in the run, before the screen can accept text, for instance, procedural rules would be risky. We then delegate to the appropriate VM-specific section of code for the real work. The printing of three blank lines at the start of play is traditional: on early Z-machine interpreters such as InfoTaskForce and Zip it was a necessity because of the way they buffered output. On modern windowed ones it still helps to space the opening text better.

```
[ VIRTUAL_MACHINE_STARTUP_R;
    ProcessRulebook(Activity_before_rulebooks-->STARTING_VIRTUAL_MACHINE_ACT);
    ProcessRulebook(Activity_for_rulebooks-->STARTING_VIRTUAL_MACHINE_ACT);
    ProcessRulebook(Activity_after_rulebooks-->STARTING_VIRTUAL_MACHINE_ACT);
    VM_Initialise();
    print "^^^";
    rfalse;
];
```

§**3. Initial Situation.** The array `InitialSituation` is compiled by NI and contains:

(0) The object number for the player, which is usually `selfobj`.
(1) The object in or on which the player begins, if he does. (This will always be an enterable container or supporter, or `nothing`.)
(2) The room in which the player begins, which is usually the first room created in the source text.
(3) The initial time of day, which is usually 9 AM.

The start object and start room are meaningful only if the player's object is compiled outside of the object tree (as can happen if the source text reads, say, "Mrs Bridges is a woman. The player is Mrs Bridges."): in other circumstances they are often correct, but this must not be relied on.

```
Constant PLAYER_OBJECT_INIS = 0;
Constant START_OBJECT_INIS = 1;
Constant START_ROOM_INIS = 2;
Constant START_TIME_INIS = 3;
Constant DONE_INIS = 4;
{-array:Plugins::Player::InitialSituation}
```

§**4. Initialise Memory Rule.** This rule amalgamates some minimal initialisations which all need to happen before we can risk using some of the more exotic I7 kinds:

(a) The language definition might call for initialisation, although the default language of play (English) does not.
(b) A handful of variables are filled in. `I7_LOOKMODE` is a constant created by the use options "use full-length room descriptions" or "use abbreviated room descriptions", but otherwise not existing. It is particularly important that `player` have the correct value, as the process of initialising the memory heap uses the player as the presumed actor when creating memory representations of literal stored actions where no actor was specified; this is why `player` is initialised here and not in the "position player in model world rule" below. The other interesting point here is that we explicitly set `location` and `real_location` to `nothing`, which is certainly incorrect, even though we know better. We do this so that the "update chronological records rule" cannot see where the player is: see the Standard Rules for an explanation of why this is, albeit perhaps dubiously, a good thing.
(c) We start the machinery needed to check that property accesses are valid during play.
(d) And we initialise the memory allocation heap, and expand the literal constants, as hinted above: these are called "block constants" since they occupy blocks of memory.

The not_yet_in_play flag, which is cleared when the first command is about to be read from the keyboard, suppresses the standard status line text: thus, if there is some long text to read before the player finds out where he is, the surprise will not be spoiled.

```
[ INITIALISE_MEMORY_R;
    #ifdef TARGET_GLULX; VM_PreInitialise(); #Endif;
    #Ifdef LanguageInitialise; LanguageInitialise(); #Endif;

    not_yet_in_play = true;
    #ifdef I7_LOOKMODE; lookmode = I7_LOOKMODE; #endif;
    player = InitialSituation-->PLAYER_OBJECT_INIS;
    the_time = InitialSituation-->START_TIME_INIS;
    real_location = nothing;
    location = nothing;

    CreatePropertyOffsets();
    HeapInitialise(); ! Create a completely unused memory allocation heap
    InitialHeapAllocation(); ! Allocate empty blocks for variables, properties, and such
    CreateBlockConstants(); ! Allocate and fill in blocks for constant values
    DistributeBlockConstants(); ! Ensure these exist in multiple independent copies when needed
    CreateDynamicRelations(); ! Create relation structures on the heap

    rfalse;
];
```

§**5. Seed Random Number Generator Rule.** Unless a seed is provided by NI, and it won't be for released story files, the VM's interpreter is supposed to start up with a good seed in its random number generator: something usually derived from, say, the milliseconds part of the current time of day, which is unlikely to repeat or show any pattern in real-world use. However, early Z-machine interpreters often did this quite badly, starting with poor seed values which meant that the first few random numbers always had something in common (being fairly small in their range, for instance). To obviate this we extract and throw away 100 random numbers to get the generator going, shaking out more obvious early patterns, but that cannot really help much if the VM interpreter's RNG is badly written. "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" (von Neumann).

```
[ SEED_RANDOM_NUMBER_GENERATOR_R i;
    if ({-value:rng_seed_at_start_of_play}) VM_Seed_RNG({-value:rng_seed_at_start_of_play});
    for (i=1: i<=100: i++) random(i);
    rfalse;
];
```

§**6. Position Player In Model World Rule.** This seems as good a place as any to write down the invariant we attempt to maintain for the player's position variables:

(1) The `player` variable is the object through which the player plays, which is always a person: its value is always set by starting from `selfobj` and then making a sequence of 0 or more `ChangePlayer(new_value)` calls. (This enables us to make sure it has the correct property values for printed name and so forth.)

(2) The `location` is always either the current room, a valid I7 room, or `thedark`, which is not a valid I7 object but is distinguishable both from all I7 objects and from `nothing`. The `real_location` is always the current room, which is always a valid I7 room. `location` equals `thedark` if and only if the player does not have light to see by; the routine `SilentlyConsiderLight` updates this without printing any messages to announce the fall or lifting of darkness (hence "silently").

(3) The `player` object is always in the subtree of `real_location`, and is always in a chain $O_1 < O_2 < ... < O_n$ where O_1 is the player, O_n is `real_location`, $n \geq 2$ and $O_2, ..., O_{n-1}$ are all either enterable containers, enterable supporters, or component parts of such. The routine `LocationOf`, applied to the player object, always agrees with `real_location`.

(4) "Floating" objects, such as backdrops and two-sided doors, are in theory present in more than one room at once. In practice they can only be in a single position in the I6 object tree at any one time. The rule is that if they are theoretically present in the `real_location`, then they are actually present in the subtree of `real_location`. The routine `MoveFloatingObjects` updates this, and must be called whenever the player moves from one room to another.

(5) Any objects carried by the player have the I6 `moved` attribute set. The `SACK_OBJECT` variable is always set to the object of kind "player's holdall" which the player has most recently been carrying, or had as a component part of himself. The "note object acquisitions" rule updates this.

These invariants are usually all false before the following rule is executed; they are all true once it has completed. In addition, because the global action variables usually hold details of the action most recently carried out, we initialise these as if the most recent action had been the player waiting. (Nobody ought to use these variables at this point, but in case they do use them by accident in a "when play begins" rule, we want Inform to behave predictably and without type-unsafe values entering code.)

```
[ POSITION_PLAYER_IN_MODEL_R player_to_be;
    player = selfobj;
    player_to_be = InitialSituation-->PLAYER_OBJECT_INIS;
    location = LocationOf(player_to_be);
    if (location == 0) {
        location = InitialSituation-->START_ROOM_INIS;
        if (InitialSituation-->START_OBJECT_INIS)
            move player_to_be to InitialSituation-->START_OBJECT_INIS;
        else move player_to_be to location;
    }
    if (player_to_be ~= player) { remove selfobj; ChangePlayer(player_to_be); }
    real_location = location; SilentlyConsiderLight();
    NOTE_OBJECT_ACQUISITIONS_R(); MoveFloatingObjects();
    actor = player; act_requester = nothing; actors_location = real_location; action = ##Wait;
    InitialSituation-->DONE_INIS = true;
    rfalse;
];
```

§**7. Parse Command Rule.** This section contains only two primitive rules from the turn sequence rulebook, the matched pair of the "parse command rule" and the "generate action rule"; the others are found in the sections on Light and Time.

We use almost identically the same parser as that in the I6 library, since it is a well-proven and understood algorithm. The I6 parser returns some of its results in a supplied array (here `parser_results`, though the I6 library used to call this `inputobjs`), but others are in global variables:

(1) The `parser_results` array holds four words, used as indexed by the constants below.
   (a) The action can be a valid I6 action number, or an I6 "fake action", a concept not used overtly in I7. Most valid I6 actions correspond exactly to I7 actions, but in principle it is possible to define (say) extra debugging commands entirely at the I6 level.
   (b) The count `NO_INPS_PRES` is always 0, 1 or 2, and then that many of the next two words are meaningful.
   (c) Each of the "inp" values is either 0, meaning "put the multiple object list here"; or 1, meaning "not an object but a value"; or a valid I6 object. (We use the scoping rules to ensure that any I6 object visible to the parser is also a valid I7 object, so – unlike with actions – we need not distinguish between the two.)

(2) The global variable `actor` is set to the person asked to carry out the command, or is the same as `player` if nobody was mentioned. Thus it will be the object for Floyd in the command FLOYD, GET PERMIT, but will be just `player` in the command EAST.

(3) The global variables `special_number1` and, if necessary, `special_number2` hold values corresponding to the first and second of the "inps" to be returned as 1. Thus, if one of the "inps" is a value and the other is an object, then `special_number1` is that value; only if both are values rather than objects will `special_number2` be used. There is no indication of the kind of these values: I6 is typeless.

(4) At most one of the "inps" is permitted to be 1, referring to a multiple object list. (And a multiple value list is forbidden.) If this happens, the list of objects is stored in an I6 table array (i.e., with the 0th word being the number of subsequent words) called `multiple_object`, and the parser will have set the `toomany_flag` if an overflow occurred – that is, if the list was truncated because it originally called for more than 63 objects.

(5) The global variable `meta` is set if the action is one marked as such in the I6 grammar. A confusion in the design of I6 is that being out of world, as we would say in I7 terms, is associated not with an action as such but with the command verb triggering it. (This in practice caused no trouble since we never used, say, the word SAVE for both saving the game and saving, I don't know, box top coupons.) The state of `meta` returned by the I6 parser does not quite correspond to I7's "out of world" concept, so we will alter it in a few cases.

Some of these conventions are a little odd-looking now: why not simply have a larger results array, rather than this pile of occasionally used variables? The reasons are purely historical: the I6 parser developed gradually over about a decade.

```
Constant ACTION_PRES = 0;
Constant NO_INPS_PRES = 1;
Constant INP1_PRES = 2;
Constant INP2_PRES = 3; ! Parser.i6t code assumes this is INP1_PRES + 1

[ PARSE_COMMAND_R;
    if (EarlyInTurnSequence == false) rfalse; ! Prevent use outside top level
    not_yet_in_play = false;
    Parser__parse();
    TreatParserResults();
    rfalse;
];
```

§**8. Treat Parser Results.** We don't quite use the results exactly as they are returned by the parser: we make modifications in a few special cases.

(1) `##MistakeAction` is a valid I6 action, but not an I7 one. It is used to implement "Understand ... as a mistake", which provides a short-cut way for I7 source text to specify responses to mistaken guesses at the syntax expected for commands. It can therefore result from a whole variety of different commands, some of which might be flagged `meta`, others not. We forcibly set the `meta` flag: a mistake in guessing the command always happens out of world.

(2) A command in the form PERSON, TELL ME ABOUT SOMETHING is altered to the action resulting from ASK PERSON ABOUT SOMETHING, so that `##Tell` is converted to `##Ask` in these cases.

```
[ TreatParserResults;
    if (parser_results-->ACTION_PRES == ##MistakeAction) meta = true;
    if (parser_results-->ACTION_PRES == ##Tell &&
        parser_results-->INP1_PRES == player && actor ~= player) {
        parser_results-->ACTION_PRES = ##Ask;
        parser_results-->INP1_PRES = actor; actor = player;
    }
];
```

§**9. Generate Action Rule.** For what are, again, historical reasons to do with the development of I6, the current action is recorded in a slate of global variables:

(1) `actor` is as above; `action` is the I6 action number or fake action number, though in I7 usage no fake actions should ever reach this point.

(2) `act_requester` is the person requesting that the actor should perform the action, or `nothing` if the action is the actor's own choice. In the command FLOYD, MOP FLOOR, the `act_requester` is the player and the actor is Floyd; but when an action arises from a try phrase in I7, such as "try Floyd mopping the floor", `act_requester` is `nothing` because it is Floyd's own decision to do this. (The computer, of course, represents the will-power of all characters other than the player.)

(3) `inp1` and `inp2` are global variables whose contents mean the same as those of `parser_results-->INP1_PRES` and `parser_results-->INP2_PRES`. (This is not duplication, because actions also arise from "try" rather than the parser.)

(4) The variable `multiflag` is set during the processing of a multiple object list, and clear otherwise. (It is used for instance by the Standard Rules to give more concise reports of some successful actions.) Note that it remains set during any knock-on actions caused by actions in the multiple object list: the following rule is the only place where `multiflag` is set or cleared.

(5) `noun` and `second` are global variables which are equal to `inp1` and `inp2` when the latter hold valid object numbers, and are equal to `nothing` otherwise. (This is not duplication either, because it provides us with type-safe access to objects: there is no KOV which can safely represent `inp1` and `inp2`, but `noun` and `second` are valid for the I7 kind of value "object".)

In the following rule, we create this set of variables for the action or multiple action(s) suggested by the parser: each action is sent on to `BeginAction` for processing. Once done, we reset the above variables in what might seem an odd way: we allow straightforward actions by the player to remain in the variables, but convert requests to other people to the neutral "waiting" action carried out by the player (which is the zero value for actions). Now, in a better world, we would always erase the action like this, because an action once completed ought to be forgotten. The value of `noun` ought to be visible only during the action's processing.

But in practice many I7 users write "every turn" rules which are predicated on what the turn's main action was: say, "Every turn when going: ..." The every turn stage is not until later in the turn sequence, so such rules can only work if we keep the main parser-generated action of the turn in the action variables when we finish up here: so that's what we do. (Note that `BeginAction` preserves the values of the action variables,

storing copies on the stack, so whatever may have happened during processing, we finish this routine with the same action variable values that we set at the beginning.)

Finally, note that an out of world action stops the turn sequence early, at the end of action generation: this is what prevents the time of day advancing, every turn rules from firing, and so forth – see the Standard Rules.

```
[ GENERATE_ACTION_R i j k l;
    if (EarlyInTurnSequence == false) rfalse; ! Prevent use outside top level
    EarlyInTurnSequence = false;
    action = parser_results-->ACTION_PRES;
    act_requester = nothing; if (actor ~= player) act_requester = player;
    inp1 = 0; inp2 = 0; multiflag = false;
    if (parser_results-->NO_INPS_PRES >= 1) {
        inp1 = parser_results-->INP1_PRES; if (inp1 == 0) multiflag = true;
    }
    if (parser_results-->NO_INPS_PRES >= 2) {
        inp2 = parser_results-->INP2_PRES; if (inp2 == 0) multiflag = true;
    }
    if (inp1 == 1) {
        noun = nothing; ! noun = special_number1;
    } else noun = inp1;
    if (inp2 == 1) {
        second = nothing;
        ! if (inp1 == 1) second = special_number2; else second = special_number1;
    } else second = inp2;
    if (multiflag) {
        if (multiple_object-->0 == 0) { L__M(##Miscellany, 2); return; }
        if (toomany_flag) { toomany_flag = false; L__M(##Miscellany, 1); }
        GenerateMultipleActions();
        multiflag = false;
    } else BeginAction(action, noun, second);
    if ((actor ~= player) || (act_requester)) action = ##Wait;
    actor = player; act_requester = 0;
    if (meta) { RulebookSucceeds(); rtrue; }
    rfalse;
];
```

§10. Generate Multiple Actions. So this routine is used to issue the individual actions necessary when a multiple object list has been supplied as either the noun or second noun part of an action generated by the parser. Note that we stop processing the list in the event of the game ending, or of the `location` variable changing its value, which can happen either through movement of the player, or through passage from darkness to light or vice versa.

We use `RunParagraphOn` to omit skipped lines as paragraph breaks between the results from any item in the list: this is both more condensed on screen in ordinary lists, and might allow the user to play tricks such as gathering up reports from a list and delivering them later in some processed way.

```
[ GenerateMultipleActions initial_location k item;
    initial_location = location;
    for (k=1: k<=multiple_object-->0: k++) {
        item = multiple_object-->k;
        RunParagraphOn();
        if (inp1 == 0) { inp1 = item; BeginAction(action, item, second, item); inp1 = 0; }
        else { inp2 = item; BeginAction(action, noun, item, item); inp2 = 0; }
        if (deadflag) return;
        if (location ~= initial_location) { L__M(##Miscellany, 51); return; }
    }
];
```

§11. Timed Events Rule. A timed event is a rule stored in the `TimedEventsTable`, an I6 table array: zero entries in this table are ignored, and the sequence is significant only if more than one event goes off at the same moment, in which case earlier entries go off first. Each rule in the table has a corresponding timer value in `TimedEventTimesTable`. If this is negative, it represents a number of turns to go before the event happens – or properly speaking, the number of times the timed events rule is invoked. Otherwise the timer value must be a valid time of day at which the event happens (note that valid times are all non-negative integers). We allow a bracket of 30 minutes after the event time proper; this is designed to cope with situations in which the user sets some timed events, then advances the clock by hand (or uses a long step time, say in which each turn equates to 20 minutes).

Because an event is struck out of the table just before it is fired, it will not continue to go off the rest of the half-hour. Moreover, because the striking out happens *before* rather than after the rule fires, a rule can re-time itself to go off again later, somewhat like the snooze feature on an alarm clock, without the risk of it going off again immediately in the same use of the timed events rule: there is guaranteed to be a blank slot in the timer array at or before the current position because we have just blanked one.

```
[ TIMED_EVENTS_R i event_timer fire rule;
    for (i=1: i<=(TimedEventsTable-->0): i++)
        if ((rule=TimedEventsTable-->i) ~= 0) {
            event_timer = TimedEventTimesTable-->i; fire = false;
            if (event_timer<0) {
                (TimedEventTimesTable-->i)++;
                if (TimedEventTimesTable-->i == 0) fire = true;
            } else {
                if ((the_time >= event_timer) && (the_time < event_timer+30)) fire = true;
            }
            if (fire) {
                TimedEventsTable-->i = 0;
                ProcessRulebook(rule);
            }
        }
    rfalse;
];
```

§**12. Setting Timed Events.** This is the corresponding routine which adds events to the timer tables, and is used to define phrases like "the cuckoo clock explodes in 7 turns from now" or "the cuckoo clock explodes at 4 PM". Here the `rule` would be "cuckoo clock explodes", and the `event_time` would either be 4 PM with `absolute_time` set, or simply 7 with `absolute_time` clear.

Note that the same event can occur only once in the timer tables: a new setting for its firing overwrites an old one. (This ensures that the table does not slowly balloon in size if the user has not been careful to ensure that events always fire.)

```
[ SetTimedEvent rule event_time absolute_time i b;
    for (i=1: i<=(TimedEventsTable-->0): i++) {
        if (rule == TimedEventsTable-->i) { b=i; break; }
        if ((b==0) && (TimedEventsTable-->i == 0)) b=i;
    }
    if (b==0) return RunTimeProblem(RTP_TOOMANYEVENTS);
    TimedEventsTable-->b = rule;
    if (absolute_time) TimedEventTimesTable-->b = event_time;
    else TimedEventTimesTable-->b = -event_time;
];
```

§**13. Setting Time Of Day.** This is the old I6 library routine `SetTime`, which is no longer used in I7 at present; but might be, some day.

```
Global time_step;
[ SetTime t s;
    the_time = t; time_rate = s; time_step = 0;
    if (s < 0) time_step = 0-s;
];
```

§**14. Advance Time Rule.** This rule advances the two measures of the passing of time: the number of `turns` of play, and `the_time` of day.

```
[ ADVANCE_TIME_R;
    turns++;
    if (the_time ~= NULL) {
        if (time_rate >= 0) the_time = the_time+time_rate;
        else {
            time_step--;
            if (time_step == 0) {
                the_time++;
                time_step = -time_rate;
            }
        }
        the_time = the_time % TWENTY_FOUR_HOURS;
    }
    rfalse;
];
```

§**15. Note Object Acquisitions Rule.** See the Standard Rules for comment on this.

```
[ NOTE_OBJECT_ACQUISITIONS_R obj;
    objectloop (obj in player) give obj moved;
    objectloop (obj has concealed)
        if (IndirectlyContains(player, obj)) give obj ~concealed;
    #Ifdef RUCKSACK_CLASS;
    objectloop (obj in player)
        if (obj ofclass RUCKSACK_CLASS)
            SACK_OBJECT = obj;
    objectloop (obj ofclass RUCKSACK_CLASS && obj provides component_parent
        && obj.component_parent == player)
        SACK_OBJECT = obj;
    #Endif;
    rfalse;
];
```

§**16. Resurrect Player If Asked Rule.** If a rule in the "when play ends" rulebook set `resurrect_please`, by executing the "resume the game" phrase, then this is where we notice that: making the shutdown rulebook succeed then tells `Main` to fall back into the turn sequence.

```
[ RESURRECT_PLAYER_IF_ASKED_R;
    if (resurrect_please) {
        RulebookSucceeds(); resurrect_please = false;
        deadflag = 0; story_complete = false; rtrue;
    }
    rfalse;
];
```

§**17. Ask The Final Question Rule.** And so we come to the bittersweet end: we ask the final question endlessly, until the player gives a reply which takes drastic enough action to destroy the current execution context in the VM, for instance by typing QUIT, RESTART, UNDO or RESTORE. The question and answer are all managed by the activity, which is defined in I7 source text in the Standard Rules.

```
[ ASK_FINAL_QUESTION_R;
    print "^";
    while (true) {
        CarryOutActivity(DEALING_WITH_FINAL_QUESTION_ACT);
        DivideParagraphPoint();
    }
];
```

§**18. Read The Final Answer Rule.** This erases the current command, so is a technique we couldn't use during actual play, but here commands are but a distant memory. So we can use the same buffers for the final question as for game commands.

```
[ READ_FINAL_ANSWER_R;
    DrawStatusLine();
    KeyboardPrimitive(buffer, parse);
    players_command = 100 + WordCount();
    num_words = WordCount();
    wn = 1;
    rfalse;
];
```

§**19. Immediately Restart VM Rule.** Now for four rules acting on typical responses to the final question.

```
[ IMMEDIATELY_RESTART_VM_R; @restart; ];
```

§**20. Immediately Restore Saved Game Rule.** It is almost certainly unnecessary to set `actor` to `player` here, but we do so just in case, because `RESTORE_THE_GAME_R` is protected against doing anything when it thinks it might have been called erroneously through a command like "DAPHNE, RESTORE". (Out of world actions should never be carried out that way, but again, it's a precaution.)

```
[ IMMEDIATELY_RESTORE_SAVED_R; actor = player; RESTORE_THE_GAME_R(); ];
```

§**21. Immediately Quit Rule.**

```
[ IMMEDIATELY_QUIT_R; @quit; ];
```

§**22. Immediately Undo Rule.** An UNDO is disallowed when `turns` is 1, because there is nothing to revert to: but suppose the player died or won as a result of the very first command? Then the game will be over with `turns` equal to 1, but UNDO disallowed, even though there is a saved state to revert to, captured just before that command was typed. To prevent this, we increment `turns` to include the one only partially completed, but only if a command was actually typed. (If the player died as a result of a monstrously unfair rule applied before the very first command had been typed, UNDO is indeed impossible, and `turns` is left at 1.)

```
[ IMMEDIATELY_UNDO_R;
    if (not_yet_in_play == false) turns++;
    Perform_Undo();
    if (not_yet_in_play == false) turns--;
];
```

§**23. Print Obituary Headline Rule.** Finally, definitions of three primitive rules for the "printing the player's obituary" activity.

```
[ PRINT_OBITUARY_HEADLINE_R;
    print "^^    ";
    VM_Style(ALERT_VMSTY);
    print "***";
    if (deadflag == 1) L__M(##Miscellany, 3);
    if (deadflag == 2) L__M(##Miscellany, 4);
    if (deadflag == 3) L__M(##Miscellany, 75);
    if (deadflag ~= 0 or 1 or 2 or 3)  {
        print " ";
        if (deadflag ofclass Routine) (deadflag)();
        if (deadflag ofclass String) print (string) deadflag;
        print " ";
    }
    print "***";
    VM_Style(NORMAL_VMSTY);
    print "^^"; #Ifndef NO_SCORE; print "^"; #Endif;
    rfalse;
];
```

§**24. Print Final Score Rule.**

```
[ PRINT_FINAL_SCORE_R;
    #Ifndef NO_SCORING; ANNOUNCE_SCORE_R(); #Endif;
    rfalse;
];
```

§**25. Display Final Status Line Rule.**

```
[ DISPLAY_FINAL_STATUS_LINE_R;
    sline1 = score; sline2 = turns;
    rfalse;
];
```