Appendices

§A1 Library attributes

`absent` A 'floating object' (one with a `found_in` property, which can appear in many different rooms) which is `absent` will in future no longer appear in the game. Note that you cannot make a floating object disappear merely by giving it `absent`, but must explicitly `remove` it as well.

`animate` "Is alive (human or animal)." Can be spoken to in "richard, hello" style; matches the `creature` token in grammar; picks up "him" or "her" (according to gender) rather than "it", likewise "his"; an object the player is changed into becomes `animate`; some messages read "on whom", etc., instead of "on which"; can't be taken; its subobjects "belong to" it rather than "are part of" it; messages don't assume it can be "touched" or "squeezed" as an ordinary object can; the actions `Attack`, `ThrowAt` are diverted to `life` rather than rejected as being 'futile violence'.

`clothing` "Can be worn."

`concealed` "Concealed from view but present." The player object has this; an object which was the player until `ChangePlayer` happened loses this property; a `concealed door` can't be entered; does not appear in room descriptions.

`container` Affects scope and light; object lists recurse through it if `open` (or `transparent`); may be described as closed, open, locked, empty; a possession will give it a `LetGo` action if the player tries to remove it, or a `Receive` if something is put in; things can be taken or removed from it, or inserted into it, but only if it is `open`; likewise for "transfer" and "empty"; room descriptions describe using `when_open` or `when_closed` if given; if there is no defined `description`, an `Examine` causes the contents to be searched (i.e. written out) rather than a message "You see nothing special about..."; `Search` only reveals the contents of `containers`, otherwise saying "You find nothing". *Note:* an object cannot be both a `container` and a `supporter`.

`door` "Is a door or bridge." Room descriptions describe using `when_open` or `when_closed` if given; and an `Enter` action becomes a `Go` action. If a `Go` has to go through this object, then: if `concealed`, the player "can't go that way"; if not `open`, then the player is told either that this cannot be ascended or descended (if the player tried "up" or "down"), or that it is in the way (otherwise); but if neither, then its `door_to` property is

consulted to see where it leads; finally, if this is zero, then it is said to "lead nowhere" and otherwise the player actually moves to the location.

`edible` "Can be eaten" (and thus removed from game).

`enterable` Affects scope and light; only an `enterable` on the floor can be entered. If an `enterable` is also a `container` then it can only be entered or exited if it is `open`.

`female` This object has a feminine name. In games written in English, this makes her a female person, though in other languages it might be inanimate. The parser uses this information when considering pronouns like "her". (In English, anything `animate` is assumed to be male unless `female` or `neuter` is set.)

`general` A general-purpose attribute, defined by the library but never looked at or altered by it. Available for designers to use if they choose to do so.

`light` "Is giving off light." (See §19.) Also: the parser understands "lit", "lighted", "unlit" using this; inventories will say "(providing light)" of it, and so will room descriptions if the current `location` is ordinarily dark; it will never be automatically put away into the player's `SACK_OBJECT`, as it might plausibly be inflammable or the main light source.

`lockable` Can be locked or unlocked by a player holding its key object, which is given by the property `with_key`; if a `container` and also `locked`, may be called "locked" in inventories.

`locked` Can't be opened. If a `container` and also `lockable`, may be called "locked" in inventories.

`male` This object has a masculine name. In games written in English, this makes him a male person, though in other languages it might be inanimate. The parser uses this information when considering pronouns like "him". (In English, anything `animate` is assumed to be male unless `female` or `neuter` is set.)

`moved` "Has been or is being held by the player." Objects (immediately) owned by the player after `Initialise` has run are given it; at the end of each turn, if an item is newly held by the player and is `scored`, it is given `moved` and `OBJECT_SCORE` points are awarded; an object's `initial` message only appears in room descriptions if it is `unmoved`.

`neuter` This object's name is neither masculine nor feminine. (In English, anything without `animate` is assumed neuter, because only people and higher animals have gender. Anything `animate` is assumed male unless `female` or `neuter` is set. A robot, for instance, might be an `animate` object worth making `neuter`.)

`on` "Switched on." A `switchable` object with `on` is described by `with_on` in room descriptions; it will be called "switched on" by `Examine`.

`open` "Open door or container." Affects scope and light; lists (such as inventories) recurse through an `open container`; if a `container`, called

	"open" by some descriptions; things can be taken or removed from an `open container`; similarly inserted, transferred or emptied. A `container` can only be entered or exited if it is both `enterable` and `open`. An `open door` can be entered. Described by `when_open` in room descriptions.
`openable`	Can be opened or closed, unless `locked`.
`pluralname`	This single object's name is in the plural. For instance, an object called "seedless grapes" should have `pluralname` set. The library will then use the pronoun "them" and the indefinite article "some" automatically.
`proper`	Its short name is a proper noun, and never preceded by "the" or "The". The player's object must have this (so something changed into will be given it).
`scenery`	Not listed by the library in room descriptions; "not portable" to be taken; "you are unable to" pull, push, or turn it.
`scored`	The player gets `OBJECT_SCORE` points for picking it up for the first time; or, if a room, `ROOM_SCORE` points for visiting it for the first time.
`static`	"Fixed in place" if player tries to take, remove, pull, push or turn.
`supporter`	"Things can be put on top of it." Affects scope and light; object lists recurse through it; a possession will give it a `LetGo` action if the player tries to remove it, or a `Receive` if something is put in; things can be taken or removed from it, or put on it; likewise for transfers; a player inside it is said to be "on" rather than "in" it; room descriptions list its contents in separate paragraphs if it is itself listed. *Note:* an object cannot be both a `container` and a `supporter`.
`switchable`	Can be switched on or off; listed as such by `Examine`; described using `when_on` or `when_off` in room descriptions.
`talkable`	Player can talk to this object in "thing, do this" style. This is useful for microphones and the like, when `animate` is inappropriate.
`transparent`	"Contents are visible." Affects scope and light; a `transparent` container is treated as if it were `open` for printing of contents.
`visited`	"Has been or is being visited by the player." Given to a room immediately after a `Look` first happens there: if this room is `scored` then `ROOM_SCORE` points are awarded. Affects whether room descriptions are abbreviated or not.
`workflag`	Temporary flag used by Inform internals, also available to outside routines; can be used to select items for some lists printed by `WriteListFrom`.
`worn`	"Item of clothing being worn." Should only be an object being immediately carried by player. Affects inventories; doesn't count towards the limit of `MAX_CARRIED`; won't be automatically put away into the `SACK_OBJECT`; a `Drop` action will cause a `Disrobe` action first; so will `PutOn` or `Insert`.

Note. The only library attributes which it's useful to apply to locations are `light`, `scored` and `visited`.

§A2 Library properties

The following table lists every library-defined property. The banner headings give the name, what type of value makes sense and the default value (if other than 0). The symbol ⊕ means "this property is additive" so that inherited values from class definitions pile up into a list, rather than wipe each other out. Recall that `false` is the value 0 and `true` the value 1.

`n_to, s_to, e_to, w_to, ...` *Room, object or routine*

For rooms These twelve properties (there are also `ne_to`, `nw_to`, `se_to`, `sw_to`, `in_to`, `out_to`, `u_to` and `d_to`) are the map connections for the room. A value of 0 means "can't go this way". Otherwise, the value should either be a room or a `door` object: thus, `e_to` might be set to `crystal_bridge` if the direction "east" means "over the crystal bridge".
Routine returns The room or object the map connects to; or 0 for "can't go this way"; or 1 for "can't go this way; stop and print nothing further".
Warning Do not confuse the direction properties `n_to` and so on with the twelve direction objects, `n_obj` et al.

`add_to_scope` *List of objects or routine*

For objects When this object is in scope (unless it was itself only placed in scope by `PlaceInScope`) so are all those listed, or all those nominated by the routine. A routine given here should call `PlaceInScope(obj)` to put `obj` in scope.
No return value

`after` *Routine* `NULL` ⊕

Receives actions after they have happened, but before the player has been told of them.
For rooms All actions taking place in this room.
For objects All actions for which this object is `noun` (the first object specified in the command); and all fake actions for it.
Routine returns `false` to continue (and tell the player what has happened), `true` to stop here (printing nothing).
The `Search` action is a slightly special case. Here, `after` is called when it is clear that it would be sensible to look inside the object (e.g., it's an open container in a light room) but before the contents are described.

`article` *String or routine* `"a"`

For objects Indefinite article for object or routine to print one.
No return value

`articles` *Array of strings*

For objects: If given, these are the articles used with the object's name. (Provided for non-English languages where irregular nouns may have unusual vowel-contraction rules with articles: e.g., with French non-mute 'H'.)

`before` *Routine* `NULL` ⊕

Receives advance warning of actions (or fake actions) about to happen.
For rooms All actions taking place in this room.
For objects All actions for which this object is `noun` (the first object specified in the command); and all fake actions, such as `Receive` and `LetGo` if this object is the container or supporter concerned.
Routine returns `false` to continue with the action, `true` to stop here (printing nothing).
First special case: A vehicle object receives the `Go` action if the player is trying to drive around in it. In this case:
Routine returns 0 to disallow as usual; 1 to allow as usual, moving vehicle and player; 2 to disallow but do (and print) nothing; 3 to allow but do (and print) nothing. If you want to move the vehicle in your own code, return 3, not 2: otherwise the old location may be restored by subsequent workings.
Second special case: in a `PushDir` action, the `before` routine must call `AllowPushDir()` and then return `true` in order to allow the attempt (to push an object from one room to another) to succeed.

`cant_go` *String or routine* `"You can't go that way."`

For rooms Message, or routine to print one, when a player tries to go in an impossible direction from this room.
No return value

`capacity` *Number or routine* `100`

For objects Number of objects a `container` or `supporter` can hold.
For the player-object Number of things the player can carry (when the player is this object); the default player object (`selfobj`) has `capacity` initially set to the constant `MAX_CARRIED`.

`daemon` *Routine*

This routine is run each turn, once it has been activated by a call to `StartDaemon`, and until stopped by a call to `StopDaemon`.

`describe` *Routine* `NULL` ⊕

For objects Called when the object is to be described in a room description, before any paragraph break (i.e., skipped line) has been printed. A sometimes useful trick is to print nothing in this routine and return `true`, which makes an object 'invisible'.
For rooms Called before a room's long ("look") description is printed.
Routine returns `false` to describe in the usual way, `true` to stop printing here.

`description` *String or routine*

For objects The `Examine` message, or a routine to print one out.
For rooms The long ("look") description, or a routine to print one out.
No return value

`door_dir` *Direction property or routine*

For compass objects When the player tries to go in this direction, e.g., by typing the name of this object, then the map connection tried is the value of this direction property for the current room. For example, the `n_obj` "north" object normally has `door_dir` set to `n_to`.
For objects The direction that this `door` object goes via (for instance, a bridge might run east, in which case this would be set to `e_to`).
Routine returns The direction property to try.

`door_to` *Room or routine*

For objects The place this door object leads to. A value of 0 means "leads nowhere".
Routine returns The room. Again, 0 (or `false`) means "leads nowhere". Further, 1 (or `true`) means "stop the movement action immediately and print nothing further".

`each_turn` *String or routine* `NULL` ⊕

String to print, or routine to run, at the end of each turn in which the object is in scope (after all timers and daemons for that turn have been run).
No return value

`found_in` *List of rooms or routine*

This object will be found in all of the listed rooms, or if the routine says so, unless it has the attribute `absent`. If an object in the list is not a room, it means "present in the same room as this object".
Routine returns `true` to be present, otherwise `false`. The routine can look at the current `location` in order to decide.
Warning This property is only looked at when the player changes rooms.

`grammar` *Routine*

For `animate` *or* `talkable` *objects* This is called when the parser has worked out that the object in question is being spoken to, and has decided the `verb_word` and `verb_wordnum` (the position of the verb word in the word stream) but hasn't yet tried any grammar. The routine can, if it wishes, parse past some words (provided it moves `verb_wordnum` on by the number of words it wants to eat up).
Routine returns `false` to carry on as usual; `true` to indicate that the routine has parsed the entire command itself, and set up `action`, `noun` and `second` to the appropriate order; or a dictionary value for a verb, such as `'take'`, to indicate "parse the command from this verb's grammar instead"; or minus such a value, e.g. `-'take'`, to indicate "parse from this verb and then parse the usual grammar as well".

`initial` *String or routine*

For objects The description of an object not yet picked up, used when a room is described; or a routine to print one out.
For rooms Printed or run when the room is arrived in, either by ordinary movement or by `PlayerTo`.
Warning If the object is a `door`, or a `container`, or is `switchable`, then use one of the `when_` properties rather than `initial`.
No return value

`inside_description` *String or routine*

For objects Printed as part or all of a room description when the player is inside the given object, which must be `enterable`.

`invent` *Routine*

This routine is for changing an object's inventory listing. If provided, it's called twice, first with the variable `inventory_stage` set to 1, second with it set to 2. At stage 1, you have an entirely free hand to print a different inventory listing.
Routine returns Stage 1: `false` to continue; `true` to stop here, printing nothing further about the object or its contents.
At stage 2, the object's indefinite article and short name have already been printed, but messages like " (providing light)" haven't. This is an opportunity to add something like " (almost empty)".
Routine returns Stage 2: `false` to continue; `true` to stop here, printing nothing further about the object or its contents.

`life` *Routine* `NULL` ⊕

This routine holds rules about `animate` objects, behaving much like `before` and `after` but only handling the person-to-person events:

```
Attack Kiss WakeOther ThrowAt Give Show Ask Tell Answer Order
```

See §17, §18 and the properties `orders` and `grammar`.
Routine returns `true` to stop and print nothing, `false` to resume as usual (for example, printing "Miss Gatsby has better things to do.").

`list_together` *Number, string or routine*

For objects Objects with the same `list_together` value are grouped together in object lists (such as inventories, or the miscellany at the end of a room description). If a string such as `"fish"` is given, then such a group will be headed with text such as `"five fish"`.

A routine, if given, is called at two stages in the process (once with the variable `inventory_stage` set to 1, once with it set to 2). These stages occur before and after the group is printed; thus, a preamble or postscript can be printed. Also, such a routine may change the variable `c_style` (which holds the current list style). On entry, the variable `parser_one` holds the first object in the group, and `parser_two` the current depth of recursion in the list. Applying `x=NextEntry(x,parser_two);` moves `x` on from `parser_one` to the next item in the group. Another helpful variable is `listing_together`, set up to the first object of a group being listed (or to 0 whenever no group is being listed).

Routine returns Stage 1: `false` to continue, `true` not to print the group's list at all.

Routine returns Stage 2: No return value.

`name` *List of dictionary words* ⊕

For objects A list of dictionary words referring to this object.

Warning The `parse_name` property of an object may take precedence over this, if present.

For rooms A list of words which the room understands but which refer to things which "do not need to be referred to in this game"; these are only looked at if all other attempts to understand the player's command have failed.

Warning Uniquely in Inform syntax, these dictionary words are given in double quotes `"thus"`, whereas in all other circumstances they would be `'thus'`. This means they can safely be only one letter long without ambiguity.

`number` *Any value*

A general purpose property left free: conventionally holding a number like "number of turns' battery power left". (Now unnecessary, `number` is a feature left over from earlier versions of Inform where it was less easy to make new properties.)

For compass objects Note that the standard compass objects defined by the library all provide a `number` property, in case this might be useful to the designer.

`orders` *Routine*

For `animate` *or* `talkable` *objects* This carries out the player's orders (or doesn't, as it sees fit): it looks at `actor`, `action`, `noun` and `second` to do so. Unless this object is the current player, `actor` is irrelevant (it is always the player) and the object is the person being ordered about.

If the player typed an incomprehensible command, like "robot, og sthou", then the action is `NotUnderstood` and the variable `etype` holds the parser's error number.

If this object is the current player then `actor` is the person being ordered about. `actor` can either be this object -- in which case an action is being processed, because the player has typed an ordinary command -- or can be some other object, in which case the player has typed an order. See §18 for how to write `orders` routines in these cases.

Routine returns `true` to stop and print nothing further; `false` to continue. (Unless the object is the current player, the `life` routine's `Order` section gets an opportunity to meddle next; after that, Inform gives up.)

`parse_name` *Routine*

For objects To parse an object's name (this overrides the `name` but is also used in determining if two objects are describably identical). This routine should try to match as many words as possible in sequence, reading them one at a time by calling `NextWord()`. (It can leave the "word marker" variable `wn` anywhere it likes).
Routine returns 0 if the text didn't make any sense at all, −1 to make the parser resume its usual course (looking at the `name`), or the number of words in a row which successfully matched.
In addition to this, if the text matched seems to be in the plural (for instance, a blind mouse object reading `blind mice`), the routine can set the variable `parser_action` to the value `##PluralFound`. The parser will then match with all of the different objects understood, rather than ask a player which of them is meant.
A `parse_name` routine may also (voluntarily) assist the parser by telling it whether or not two objects which share the same `parse_name` routine are identical. (They may share the same routine if they both inherit it from a class.) If, when it is called, the variable `parser_action` is set to `##TheSame` then this is the reason. It can then decide whether or not the objects `parser_one` and `parser_two` are indistinguishable.
Routine returns −1 if the objects are indistinguishable, −2 if not.

`plural` *String or routine*

For objects The plural name of an object (when in the presence of others like it), or routine to print one; for instance, a wax candle might have `plural` set to `"wax candles"`.
No return value

`react_after` *Routine*

For objects Acts like an `after` rule, but detects any actions in the vicinity (any actions which take place when this object is in scope).
Routine returns `true` to print nothing further; `false` to carry on.

`react_before` *Routine*

For objects Acts like a `before` rule, but detects any actions in the vicinity (any actions which take place when this object is in scope).
Routine returns `true` to stop the action, printing nothing; `false` to carry on.

`short_name` *Routine*

For objects The short name of an object (like "brass lamp"), or a routine to print it.
Routine returns `true` to stop here, `false` to carry on by printing the object's 'real' short name (the string given at the head of the object's definition). It's sometimes useful to print text like `"half-empty "` and then return `false`.

`short_name_indef` *Routine*

For objects If set, this form of the short name is used when the name is prefaced by an indefinite article. (This is not useful in English-language games, but in other languages adjectival parts of names agree with the definiteness of the article.)

`time_left` *Number*

Number of turns left until the timer for this object (if set, which must be done using `StartTimer`) goes off. Its initial value is of no significance, as `StartTimer` will write over this, but a timer object must provide the property. If the timer is currently set, the value 0 means "will go off at the end of the current turn", the value 1 means "... at the end of next turn" and so on.

`time_out` *Routine* `NULL` ⊕

Routine to run when the timer for this object goes off (having been set by `StartTimer` and not in the mean time stopped by `StopTimer`).
Warning A timer object must also provide a `time_left` property.

`when_closed` *String or routine*

For objects Description, or routine to print one, of something closed (a `door` or `container`) in a room's long description.
No return value

`when_open` *String or routine*

For objects Description, or routine to print one, of something open (a `door` or `container`) in a room's long description.
No return value

`when_on` *String or routine*

For objects Description, or routine to print one, of a `switchable` object which is currently switched on, in a room's long description.
No return value

`when_off` *String or routine*

For objects Description, or routine to print one, of a `switchable` object which is currently switched off, in a room's long description.
No return value

`with_key` *Object* `nothing`

The key object needed to lock or unlock this `lockable` object. A player must explicitly name it as the key being used and be holding it at the time. The value `nothing`, or 0, means that no key fits (though this is not made clear to the player, who can try as many as he likes).

§A3 Library routines

The Inform library files contain about three hundred routines, almost all of which are "private" in the sense that they are difficult for designers to use, not useful anyway, and subject to change without notice as the library is maintained and rewritten. The routines in this appendix are those which are "open to the public". Designers are encouraged to make use of them.

`Achieved(tasknum)` *see §22*

Signals to the library that task number `tasknum` has been achieved, so that points may be awarded if it has not been achieved before.
No return value

`AfterRoutines()` *see §6*

This should be called in the action routine, such as `TakeSub`, of a group 2 action, such as `Take`, once the action has taken place but before anything is printed. It runs through the `after` rules as laid out in §6.
Routine returns `true` if the action has been interrupted by some other rule, `false` if not.

`AllowPushDir()` *see §15*

Used only inside the `before` rule for a `PushDir` action, this routine signals to the library that the attempt to push an object from one place to another should be allowed.
No return value

`Banner()` *see §21*

Prints the game banner. Normally unnecessary, but should be used soon after if your game suppresses the banner at the `Initialise` stage.
No return value

`ChangePlayer(obj,flag)` *see §21*

Cause the player at the keyboard to play as the given object `obj`, which must provide a `number` property. If the `flag` is `true`, then subsequently print messages like "(as Ford Prefect)" in room description headers. This routine, however, prints nothing itself.
No return value

`CommonAncestor(obj1,obj2)` *see §3*

A routine used internally by the library when working out visibilities, and which might as well be available for public use. Returns the nearest object in the object tree which (directly or indirectly) contains both `obj1` and `obj2`, or else returns `nothing` if no such object exists. For instance if `Bedquilt` contains `bottle` and the `player` carrying a `lamp`, the common ancestor of lamp and bottle is `Bedquilt`.
Routine returns The common ancestor or `nothing`.

`DictionaryLookup(word,length)` *see §34*

Takes the word stored character by character in the array `word->0`, `word->1`, ..., `word->(length-1)` and looks it up in the story file's dictionary.
Routine returns The dictionary value (e.g., `'t'`, `'a'`, `'k'`, `'e'` will return `'take'`) or zero if the word isn't in the dictionary.

`GetGNAOfObject(obj)` *see §37*

Determines the gender-number-animation of the short name of the given object `obj`.
Routine returns The GNA, which is a number between 0 and 11: see table of GNA values in §34.

`HasLightSource(obj)` *see §19*

Determines whether or not `obj` "has light", i.e., casts light outward to objects containing `obj`: see §19 for a more exact definition.
Routine returns `true` or `false`.

`IndirectlyContains(obj1,obj2)` *see §3*

The condition `obj2 in obj1` only tests whether `obj2` is directly contained in `obj1`, so that `lamp in player` would fail if the lamp were in a rucksack carried by the player. `IndirectlyContains(player,lamp)` would return `true`. Formally, the test is whether `obj2` is a child of `obj1`, or is a child of a child of `obj1`, or ... and so on. See also the library routine `CommonAncestor` above.
Routine returns `true` or `false`.

`IsSeeThrough(obj)` *see §19*

Determines whether or not `obj` "is see-through", i.e., allows light to pass through it. An object is see-through if it has `transparent`, or `supporter`, or `enterable` (unless it is also a closed `container`).
Routine returns `true` or `false`.

`Locale(obj,tx1,tx2)` *see §26*

Prints out the paragraphs of room description which would appear if `obj` were the room: i.e., prints out descriptions of objects in `obj` according to the usual rules. After describing the objects which have their own paragraphs, a list is given of the remaining ones. The string `tx1` is printed if there were no previous paragraphs, and the string `tx2` otherwise. (For instance, you might want "On the ledge you can see" and "On the ledge you can also see".) After the list, nothing else is printed, not even a full stop.
Routine returns The number of objects printed in the list, possibly zero.

`LoopOverScope(R,actor)` *see §32*

Calls routine `R(obj)` for each object `obj` in scope for the given `actor`. If no `actor` is given, the actor is assumed to be the player.
No return value

`LTI_Insert(position,character)` *see §36*

Inserts the given `character` at the given `position` in the standard library array `buffer` used to hold the text typed by the player, moving subsequent text along to make room. (This is protected against overflowing the buffer.)
No return value

`MoveFloatingObjects()` *see §8*

"Floating objects" is Inform library jargon for "objects which use `found_in` to be present in several locations at once". This routine adjusts the positions of objects across the whole game, ensuring that they are consistent with the current states of the property `found_in` and the attribute `absent`, and should be called after any game event which has changed these states.
No return value

`NextWord()` *see §28*

Finds the next dictionary word in the player's input, that is, the word at position `wn` in the input, moving the word number `wn` on by one. (The first word is at position 1.)
Routine returns The dictionary value, or 0 if the word is not in the dictionary or if the word stream has run out, or the constant `THEN1__WD` if the "word" was a full stop, or the constant `COMMA_WORD` if it was a comma.

`NextWordStopped()` *see §28*

Finds the next dictionary word in the player's input, that is, the word at position `wn` in the input, moving the word number `wn` on by one. (The first word is at position 1.)
Routine returns The dictionary value, or 0 if the word is not in the dictionary, or the constant `THEN1__WD` if the "word" was a full stop, or the constant `COMMA_WORD` if it was a comma, or -1 if the word stream has run out.

`NounDomain(o1,o2,type)`

This routine is one of the keystones of the parser, but see also `ParseToken` below: the objects given are the domains to search through when parsing, almost always the location and the actor, and the `type` indicates a token. The only tokens safely usable are: `NOUN_TOKEN`, for `noun`, `HELD_TOKEN`, for `held` and `CREATURE_TOKEN`, for `creature`. The routine parses the best single object name it can from the current position of `wn`.
Routine returns `nothing` for "no match", or the object matched for a success, or the constant `REPARSE_CODE` to indicate that it had to ask a clarifying question: this

reconstructed the input drastically and the parser must begin all over again. NounDomain should only be used by general parsing routines and these should always return GPR_REPARSE if NounDomain returned REPARSE_CODE, thus passing the buck upwards.

ObjectIsUntouchable(obj,flag) *see §32*

Determines whether any solid barrier, that is, any container that is not open, lies between the player and obj. If flag is true, this routine never prints anything; otherwise it prints a message like "You can't, because ... is in the way." if any barrier is found.

Routine returns true if a barrier is found, false if not.

OffersLight(obj) *see §19*

Determines whether or not obj "offers light", i.e., contains light so that its contents are visible to each other: see §19 for a more exact definition.

Routine returns true or false.

ParseToken(tokentype,tokendata) *see §31*

This is the library's own "general parsing routine", and parses the word stream against the specified token. Because some of these tokens require too much setting-up work to parse, and anyway are not very useful, only two token types are open to the public. If tokentype is ELEMENTARY_TT, then tokendata must have one of the following values: NOUN_TOKEN, HELD_TOKEN, MULTI_TOKEN, MULTIHELD_TOKEN, MULTIEXCEPT_TOKEN, MULTIINSIDE_TOKEN, CREATURE_TOKEN and NUMBER_TOKEN. Alternatively, tokentype can be SCOPE_TT and tokendata must then be a "scope routine".

Routine returns GPR_FAIL if parsing fails; GPR_PREPOSITION if a match is made but results in no data; GPR_NUMBER if a match is made, resulting in a number; GPR_MULTIPLE if a match is made, resulting in a multiple object; GPR_REPARSE if the parser has had to rewrite the text being parsed and would now like parsing to begin again from scratch; otherwise, an object which the parser has matched against the text.

PlaceInScope(obj) *see §32*

Used in "scope routines" (only) when scope_stage is set to 2 (only). Places obj in scope for the token currently being parsed. No other objects are placed in scope as a result of this, unlike the case of ScopeWithin.

No return value

PlayerTo(obj,flag) *see §21*

Moves the player to obj, which can either be a location or something enterable. If flag is false, then run a Look action to print out a room description: but if flag is true, print nothing, and if flag is 2, print a room description but abbreviate it if the room has been visited before.

No return value

`PronounNotice(obj)` *see §33*

Resets the pronouns to the object `obj`. This means that all pronouns which can match against the object are set equal to it: for instance, "Aunt Jemima" would match `'her'` but not `'it'`, "the grapes" would match `'them'` and so on.
No return value

`PronounValue(pronoun)` *see §33*

Finds the current setting of `pronoun`, which has to be the dictionary value of a legal pronoun in the current language: in the case of English, that means `'it'`, `'him'`, `'her'` or `'them'`.
Routine returns The setting, or `nothing` if it is unset.

`ScopeWithin(obj)` *see §32*

Used in "scope routines" (only) when `scope_stage` is set to 2 (only). Places the contents of `obj` in scope for the token currently being parsed, and applies the rules of scope recursively so that contents of see-through objects are also in scope, as is anything added to scope.
No return value

`SetPronoun(pronoun,obj)` *see §33*

Changes the current setting of `pronoun`, which has to be the dictionary value of a legal pronoun in the current language: in the case of English, that means `'it'`, `'him'`, `'her'` or `'them'`.
No return value

`SetTime(time,rate)` *see §20*

Set the game clock (a 24-hour clock) to the given `time` (in seconds since the start of the day), to run at the given `rate` r: $r = 0$ means it does not run, if $r > 0$ then r seconds pass every turn, if $r < 0$ then $-r$ turns pass every second.
No return value

`StartDaemon(obj)` *see §20*

Makes the daemon of the object `obj` active, so that its `daemon` routine will be called at the end of every turn.
No return value

`StartTimer(obj,period)` *see §20*

Makes the timer of the object `obj` active. Its `time_left` property is set initially to `period`, then decreased by 1 at the end of every turn in which it was positive. At the end of the turn when it was zero, the timer is stopped and the object's `time_out` property is called.
No return value

`StopDaemon(obj)` *see §20*

Makes the daemon of the object `obj` no longer active, so that its `daemon` routine will no longer be called at the end of every turn.
No return value

`StopTimer(obj)` *see §20*

Makes the timer of the object `obj` no longer active, so that its `time_left` routine will no longer be decreased and `time_out` will not be called as originally scheduled.
No return value

`TestScope(obj,actor)` *see §32*

Tests whether the object `obj` is in scope to the given `actor`. If no `actor` is given, the actor is assumed to be the player.
Routine returns `true` or `false`.

`TryNumber(wordnum)` *see §28*

Tries to parse the word at `wordnum` as a non-negative number, recognising decimal numbers and English ones from "one" to "twenty".
Routine returns -1000 if it fails altogether, or the number, except that values exceeding 10000 are rounded down to 10000.

`UnsignedCompare(a,b)`

The usual `>` condition performs a signed comparison, and occasionally, usually when comparing addresses in memory of routines or strings, you need an unsigned comparison.
Routine returns Returns 1 if $a > b$, 0 if $a = b$ and -1 if $a < b$, regarding a and b as unsigned numbers between 0 and 65535. (That is, regarding `-1` as 65535, `-2` as 65534, . . ., `-32768` as 32768.)

`WordAddress(wordnum)` *see §28*

Find where word number `wordnum` from what the player typed is stored.
Routine returns The `->` array holding the text of the word.

`WordInProperty(word,obj,prop)` *see §34*

Tests whether `word` is one of the dictionary values listed in the array given as the property `prop` of object `obj`. (Most often used to see if a given dictionary word is one of the `name` values.)
Routine returns `true` or `false`.

`WordLength(wordnum)` *see §28*

Find the length (number of letters) of the word numbered `wordnum` from what the player typed.
Routine returns The length.

`WriteListFrom(obj,st)` *see §27*

Write a list of `obj` and its siblings, with the style being `st`. To list all the objects inside `X`, list from `child(X)`. The style is made up by adding together some of the following constants:

`NEWLINE_BIT`	New-line after each entry
`INDENT_BIT`	Indent each entry according to depth
`FULLINV_BIT`	Full inventory information after entry
`ENGLISH_BIT`	English sentence style, with commas and 'and'
`RECURSE_BIT`	Recurse downwards with usual rules
`ALWAYS_BIT`	Always recurse downwards
`TERSE_BIT`	More terse English style
`PARTINV_BIT`	Only brief inventory information after entry
`DEFART_BIT`	Use the definite article in list
`WORKFLAG_BIT`	At top level (only), only list objects which have the `workflag` attribute
`ISARE_BIT`	Prints " is " or " are " before list
`CONCEAL_BIT`	Misses out `concealed` or `scenery` objects

`YesOrNo()`

Assuming that a question has already been printed, wait for the player to type "yes", "y", "no" or "n".
Routine returns `true` for "yes" or "y", `false` for "no" or "n".

§A4 Library message numbers

Answer: "There is no reply."
Ask: "There is no reply."
Attack: "Violence isn't the answer to this one."
Blow: "You can't usefully blow that/those."
Burn: "This dangerous act would achieve little."
Buy: "Nothing is on sale."
Climb: "I don't think much is to be achieved by that."
Close: 1. "That's/They're not something you can close." 2. "That's/They're already closed." 3. "You close ⟨x1⟩."
Consult: "You discover nothing of interest in ⟨x1⟩."
Cut: "Cutting that/those up would achieve little."
Dig: "Digging would achieve nothing here."
Disrobe: 1. "You're not wearing that/those." 2. "You take off ⟨x1⟩."
Drink: "There's nothing suitable to drink here."
Drop: 1. "The ⟨x1⟩ is/are already here." 2. "You haven't got that/those." 3. "(first taking ⟨x1⟩ off)" 4. "Dropped."
Eat: 1. "That's/They're plainly inedible." 2. "You eat ⟨x1⟩. Not bad."
EmptyT: 1. ⟨x1⟩ " can't contain things." 2. ⟨x1⟩ " is/are closed." 3. ⟨x1⟩ " is/are empty already." 4. "That would scarcely empty anything."
Enter: 1. "But you're already on/in ⟨x1⟩." 2. "That's/They're not something you can enter." 3. "You can't get into the closed ⟨x1⟩." 4. "You can only get into something freestanding." 5. "You get onto/into ⟨x1⟩."
Examine: 1. "Darkness, noun. An absence of light to see by." 2. "You see nothing special about ⟨x1⟩." 3. "⟨x1⟩ is/are currently switched on/off."
Exit: 1. "But you aren't in anything at the moment." 2. "You can't get out of the closed ⟨x1⟩." 3. "You get off/out of ⟨x1⟩."
Fill: "But there's no water here to carry."
FullScore: 1. "The score is/was made up as follows:^" 2. "finding sundry items" 3. "visiting various places" 4. "total (out of `MAX_SCORE`)"
GetOff: "But you aren't on ⟨x1⟩ at the moment."
Give: 1. "You aren't holding ⟨x1⟩." 2. "You juggle ⟨x1⟩ for a while, but don't achieve much." 3. "⟨x1⟩ doesn't/don't seem interested."
Go: 1. "You'll have to get off/out of ⟨x1⟩ first." 2. "You can't go that way." 3. "You are unable to climb ⟨x1⟩." 4. "You are unable to descend ⟨x1⟩." 5. "You can't, since ⟨x1⟩ is/are in the way." 6. "You can't, since ⟨x1⟩ leads nowhere."
Insert: 1. "You need to be holding ⟨x1⟩ before you can put it/them into something else." 2. "That/Those can't contain things." 3. "⟨x1⟩ is/are closed." 4. "You'll need to take it/them off first." 5. "You can't put something inside itself." 6. "(first taking it/them off)^" 7. "There is no more room in ⟨x1⟩." 8. "Done." 9. "You put ⟨x1⟩ into ⟨second⟩."

Inv: 1. "You are carrying nothing." 2. "You are carrying"

Jump: "You jump on the spot, fruitlessly."

JumpOver: "You would achieve nothing by this."

Kiss: "Keep your mind on the game."

Listen: "You hear nothing unexpected."

LMode1: " is now in its normal brief printing mode, which gives long descriptions of places never before visited and short descriptions otherwise."

LMode2: " is now in its verbose mode, which always gives long descriptions of locations (even if you've been there before)."

LMode3: " is now in its superbrief mode, which always gives short descriptions of locations (even if you haven't been there before)."

Lock: 1. "That doesn't/They don't seem to be something you can lock." 2. "That's/They're locked at the moment." 3. "First you'll have to close ⟨x1⟩." 4. "That doesn't/Those don't seem to fit the lock." 5. "You lock ⟨x1⟩."

Look: 1. " (on ⟨x1⟩)" 2. " (in ⟨x1⟩)" 3. " (as ⟨x1⟩)" 4. "^On ⟨x1⟩ is/are ⟨list⟩" 5. "[On/In ⟨x1⟩] you/You can also see ⟨list⟩ [here]." 6. "[On/In ⟨x1⟩] you/You can see ⟨list⟩ [here]."

LookUnder: 1. "But it's dark." "You find nothing of interest."

Mild: "Quite."

ListMiscellany: 1. " (providing light)" 2. " (which is/are closed)" 3. " (closed and providing light)" 4. " (which is/are empty)" 5. " (empty and providing light)" 6. " (which is/are closed and empty)" 7. " (closed, empty and providing light)" 8. " (providing light and being worn" 9. " (providing light" 10. " (being worn" 11. " (which is/are " 12. "open" 13. "open but empty" 14. "closed" 15. "closed and locked" 16. " and empty" 17. " (which is/are empty)" 18. " containing " 19. " (on " 20. ", on top of " 21. " (in " 22. ", inside "

Miscellany: 1. "(considering the first sixteen objects only)^" 2. "Nothing to do!" 3. " You have died " 4. " You have won " 5. (The RESTART, RESTORE, QUIT and possibly FULL and AMUSING query, printed after the game is over.) 6. "[Your interpreter does not provide undo. Sorry!]" 7. "Undo failed. [Not all interpreters provide it.]" 8. "Please give one of the answers above." 9. "^It is now pitch dark in here!" 10. "I beg your pardon?" 11. "[You can't "undo" what hasn't been done!]" 12. "[Can't "undo" twice in succession. Sorry!]" 13. "[Previous turn undone.]" 14. "Sorry, that can't be corrected." 15. "Think nothing of it." 16. ""Oops" can only correct a single word." 17. "It is pitch dark, and you can't see a thing." 18. "yourself" (the short name of the `selfobj` object) 19. "As good-looking as ever." 20. "To repeat a command like "frog, jump", just say "again", not "frog, again"." 21. "You can hardly repeat that." 22. "You can't begin with a comma." 23. "You seem to want to talk to someone, but I can't see whom." 24. "You can't talk to ⟨x1⟩." 25. "To talk to someone, try "someone, hello" or some such." 26. "(first taking `not_holding`)" 27. "I didn't understand that sentence." 28. "I only

understood you as far as wanting to " 29. "I didn't understand that number." 30. "You can't see any such thing." 31. "You seem to have said too little!" 32. "You aren't holding that!" 33. "You can't use multiple objects with that verb." 34. "You can only use multiple objects once on a line." 35. "I'm not sure what "⟨pronoun⟩" refers to." 36. "You excepted something not included anyway!" 37. "You can only do that to something animate." 38. "That's not a verb I recognise." 39. "That's not something you need to refer to in the course of this game." 40. "You can't see "⟨pronoun⟩" (⟨value⟩) at the moment." 41. "I didn't understand the way that finished." 42. "None/only ⟨x1⟩ of those is/are available." 43. "Nothing to do!" 44. "There are none at all available!" 45. "Who do you mean, " 46. "Which do you mean, " 47. "Sorry, you can only have one item here. Which exactly?" 48. "Whom do you want [⟨actor⟩] to ⟨command⟩?" 49. "What do you want [⟨actor⟩] to ⟨command⟩?" 50. "Your score has just gone up/down by ⟨x1⟩ point/points." 51. "(Since something dramatic has happened, your list of commands has been cut short.)" 52. "Type a number from 1 to ⟨x1⟩, 0 to redisplay or press ENTER." 53. "[Please press SPACE.]"

No: see **Yes**

NotifyOff: "Score notification off."

NotifyOn: "Score notification on."

Objects: 1. "Objects you have handled:^" 2. "None." 3. " (worn)" 4. " (held)" 5. " (given away)" 6. " (in ⟨x1⟩)" [without article] 7. " (in ⟨x1⟩)" [with article] 8. " (inside ⟨x1⟩)" 9. " (on ⟨x1⟩)" 10. " (lost)"

Open: 1. "That's/They're not something you can open." 2. "It seems/They seem to be locked." 3. "That's/They're already open." 4. "You open ⟨x1⟩, revealing ⟨children⟩" 5. "You open ⟨x1⟩."

Order: "⟨x1⟩ has/have better things to do."

Places: "You have visited: "

Pray: "Nothing practical results from your prayer."

Prompt: 1. "^>"

Pronouns: 1. "At the moment, " 2. "means " 3. "is unset " 4. "no pronouns are known to the game."

Pull: 1. "It is/Those are fixed in place." 2. "You are unable to." 3. "Nothing obvious happens." 4. "That would be less than courteous."

Push: see **Pull**

PushDir: 1. "Is that the best you can think of?" 2. "That's not a direction." 3. "Not that way you can't."

PutOn: 1. "You need to be holding ⟨x1⟩ before you can put it/them on top of something else." 2. "You can't put something on top of itself." 3. "Putting things on ⟨x1⟩ would achieve nothing." 4. "You lack the dexterity." 5. "(first taking it/them off)^" 6. "There is no more room on ⟨x1⟩." 7. "Done." 8. "You put ⟨x1⟩ on <second>."

Quit: 1. "Please answer yes or no." 2. "Are you sure you want to quit? "

Remove: 1. "It is/They are unfortunately closed." 2. "But it isn't/they aren't there now." 3. "Removed."

Restart: 1. "Are you sure you want to restart? " 2. "Failed."

Restore: 1. "Restore failed." 2. "Ok."

Rub: "You achieve nothing by this."

Save: 1. "Save failed." 2. "Ok."

Score: "You have so far/In that game you scored ⟨score⟩ out of a possible `MAX_SCORE`, in ⟨turns⟩ turn/turns"

ScriptOn: 1. "Transcripting is already on." 2. "Start of a transcript of"

ScriptOff: 1. "Transcripting is already off." 2. "^End of transcript."

Search: 1. "But it's dark." 2. "There is nothing on ⟨x1⟩." 3. "On ⟨x1⟩ is/are ⟨list of children⟩." 4. "You find nothing of interest." 5. "You can't see inside, since ⟨x1⟩ is/are closed." 6. "⟨x1⟩ is/are empty." 7. "In ⟨x1⟩ is/are ⟨list of children⟩."

Set: "No, you can't set that/those."

SetTo: "No, you can't set that/those to anything."

Show: 1. "You aren't holding ⟨x1⟩." 2. "⟨x1⟩ is/are unimpressed."

Sing: "Your singing is abominable."

Sleep: "You aren't feeling especially drowsy."

Smell: "You smell nothing unexpected."

Sorry: "Oh, don't apologise."

Squeeze: 1. "Keep your hands to yourself." 2. "You achieve nothing by this."

Strong: "Real adventurers do not use such language."

Swim: "There's not enough water to swim in."

Swing: "There's nothing sensible to swing here."

SwitchOff: 1. "That's/They're not something you can switch." 2. "That's/ They're already off." 3. "You switch ⟨x1⟩ off."

SwitchOn: 1. "That's/They're not something you can switch." 2. "That's/ They're already on." 3. "You switch ⟨x1⟩ on."

Take: 1. "Taken." 2. "You are always self-possessed." 3. "I don't suppose ⟨x1⟩ would care for that." 4. "You'd have to get off/out of ⟨x1⟩ first." 5. "You already have that/those." 6. "That seems/Those seem to belong to ⟨x1⟩." 7. "That seems/Those seem to be a part of ⟨x1⟩." 8. "That isn't/Those aren't available." 9. "⟨x1⟩ isn't/aren't open." 10. "That's/They're hardly portable." 11. "That's/They're fixed in place." 12. "You're carrying too many things already." 13. "(putting ⟨x1⟩ into `SACK_OBJECT` to make room)"

Taste: "You taste nothing unexpected."

Tell: 1. "You talk to yourself a while." 2. "This provokes no reaction."

Touch: 1. "Keep your hands to yourself!" 2. "You feel nothing unexpected." 3. "If you think that'll help."

Think: "What a good idea."

Tie: "You would achieve nothing by this."

ThrowAt: 1. "Futile." 2. "You lack the nerve when it comes to the crucial moment."

Turn: see **Pull**

Unlock: 1. "That doesn't seem to be something you can unlock." 2. "It's/ They're unlocked at the moment." 3. "That doesn't/Those don't seem to fit the lock." 4. "You unlock ⟨x1⟩."

VagueGo: "You'll have to say which compass direction to go in."

Verify: 1. "The game file has verified as intact." 2. "The game file did not verify properly, and may be corrupted (or you may be running it on a very primitive interpreter which is unable properly to perform the test)."

Wait: "Time passes."

Wake: "The dreadful truth is, this is not a dream."

WakeOther: "That seems unnecessary."

Wave: 1. "But you aren't holding that/those." 2. "You look ridiculous waving ⟨x1⟩."

WaveHands: "You wave, feeling foolish."

Wear: 1. "You can't wear that/those!" 2. "You're not holding that/those!" 3. "You're already wearing that/those!" 4. "You put on ⟨x1⟩."

Yes: "That was a rhetorical question."

§A5 Entry point routines

By definition, an "entry point routine" is a routine which you can choose whether or not to define in your source code. If you do, the library will make calls to it from time to time, allowing it to change the way the game rules are administered. The exception is `Initialise`, which is compulsory.

`AfterLife()` *see §21*

When the player has died (a condition signalled by the variable `deadflag` being set to a non-zero value other than 2, which indicates winning), this routine is called: by setting `deadflag` to be `false` again it can resurrect the player.
No return value

`AfterPrompt()` *see §22*

Called just after the prompt is printed: therefore, called after all the printing for this turn is definitely over. A useful opportunity to use `box` to display quotations without them scrolling away.
No return value

`Amusing()` *see §21*

Called to provide an "afterword" for players who have won: for instance, it might advertise some features which a successful player might never have noticed. This will only be called if you have also defined the constant `AMUSING_PROVIDED` in your own code.
No return value

`BeforeParsing()` *see §30*

Called after the parser has read in some text and set up the `buffer` and `parse` tables, but has done nothing else yet except to set the word marker `wn` to 1. The routine can do anything it likes to these tables provided they remain consistent with each other, and can leave the word marker anywhere.
No return value

`ChooseObjects(obj,c)` *see §33*

When `c` is `false`, the parser is processing an "all" and has decided to exclude `obj` from it; when `c` is `true`, it has decided to include it. When `c` is 2, the parser wants help in resolving an ambiguity: perhaps using the `action_to_be` variable the routine must decide how appropriate `obj` is for the given action.
Routine returns When `c` is `false` or `true`, return `false` to accept the parser's decision, 1 to force inclusion of `obj`, 2 to force exclusion. When `c` is 2, return a numerical score between 0 and 9, with 0 being "inappropriate" and 9 "very appropriate".

`DarkToDark()` *see §19*

Called when a player goes from one dark room into another one, which is a splendid excuse to kill the player off.

`DeathMessage()` *see §21*

If the player's death occurs because you have set `deadflag` to a value of 3 or more, this entry point is called to print up a suitable "You have died"-style message.
No return value

`GamePostRoutine()` *see §6*

A kind of super-`after` rule, which applies to all actions in the game, whatever they are: use this only in the last resort.
Routine returns `false` to allow the action to continue as usual, `true` to stop the action and print nothing further.

`GamePreRoutine()` *see §6*

A kind of super-`before` rule, which applies to all actions in the game, whatever they are: use this only in the last resort.
Routine returns `false` to allow the action to continue as usual, `true` to stop the action and print nothing further.

`Initialise()` *see §21*

An opportunity to set up the initial state of the game. This routine is compulsory and has one compulsory task: to set `location` to the place where the player begins, or to the `enterable` object in or on which the player begins. It's usual to print a welcoming message as well.
Routine returns `true` or `false` to continue as usual; 2 to suppress the game banner, which would otherwise be printed immediately after this routine is called.

`InScope()` *see §32*

An opportunity to change the library's definition of what is in scope. This acts as a sort of global version of a scope token routine: it should use the library routines `ScopeWithin` and `PlaceInScope` to define what scope should be. It may want to look at the library variable `et_flag`. If this is `true`, the scope is being worked out in order to run through `each_turn`. If it's `false`, then the scope is being worked out for everyday parsing.
Routine returns `false` to tell the parser to add all the usual objects in scope as well, or `true` to tell the parser that nothing further is in scope.

`LookRoutine()` *see §26*

Called at the end of every `Look` action, that is, room description.
No return value

`NewRoom()` *see §21*

Called when the room changes, before any description of it is printed. This happens in the course of any movements or uses of `PlayerTo`.
No return value

`ParseNoun(obj)` *see §28*

To do the job of parsing the `name` property (if `parse_name` hasn't done it already). This takes one argument, the object in question, and returns a value as if it were a `parse_name` routine.
Routine returns The number of words matched, or 0 if there is no match, or −1 to decline to make a decision and give the job back to the parser. Note that if −1 is returned, the word number variable `wn` must be left set to the first word the parser should look at -- probably the same value it had when `ParseNoun` was called, but not necessarily.

`ParseNumber(text,n)` *see §28*

An opportunity to parse numbers in a different, or additional, way. The text to be parsed is a byte array of length `n` starting at `text`.
Routine returns 0 to signal "no match", 1 to 10000 to signal "this number has been matched".

`ParserError(pe)` *see §33*

The chance to print different parser error messages such as "I don't understand that sentence". `pe` is the parser error number and the table of possible values is given in §33.
Routine returns `true` to tell the parser that the error message has now been printed, or `false` to allow it to carry on and print the usual one.

`PrintRank()` *see §22*

Completes the printing of the score. Traditionally, many games have awarded the player a rank based on the current value of the variable `score`.
No return value

`PrintTaskName(n)` *see §22*

Prints the name of task `n`, which lies between 0 and `NUMBER_TASKS` minus 1.
No return value

`PrintVerb(v)` *see §30*

A chance to change the verb printed out in a parser question (like "What do you want to (whatever)?") in case an unusual verb via `UnknownVerb` has been constructed. `v` is the dictionary address of the verb.
Routine returns `true` to tell the parser that the verb has now been printed, or `false` to allow it to carry on and print the usual one.

`TimePasses()` *see §20*

Called after every game turn, which is to say, not after a group 1 action such as "score" or "save", but after any other activity. Use this entry point only as a last resort, as it's almost certainly easier and tidier to use timers and daemons.
No return value

`UnknownVerb(word)` *see §30*

Called by the parser when it hits an unknown verb, so that you can transform it into a known one. `word` is the dictionary value of this unknown verb.
Routine returns `false` to allow the parser to carry on and print an error message, or the dictionary value of a verb to use instead.