Parser Template B/parst

Purpose

The parser for turning the text of the typed command into a proposed action by the player.

---

§**1. Grammar Line Variables.** This is the I6 library parser in mostly untouched form: reformatted for template file use, and with paragraph divisions, but otherwise hardly changed at all. It is a complex algorithm but one which is known to produce good results for the most part, and it is well understood from (at time of writing) fifteen years of use. A few I7 additions have been made, but none disrupting the basic method. For instance, I7's system for resolving ambiguities is implemented by providing a `ChooseObjects` routine, just as a user of the I6 library would do.

The I6 parser uses a huge number of global variables, which is not to modern programming tastes: in the early days of Inform, the parser was essentially written in assembly-language only lightly structured by C-like syntaxes, and the Z-machine's 240 globals were more or less registers. The I6 library made no distinction between which were "private" to the parser and which allowed to be accessed by the user's code at large. The I7 template does impose that boundary, though not very strongly: the variables defined in "Output.i6t" are for general access, while the ones below should only be read or written by the parser.

```
Global best_etype;                  ! Preferred error number so far
Global nextbest_etype;              ! Preferred one, if ASKSCOPE_PE disallowed
Global parser_inflection;           ! A property (usually "name") to find object names in
Array pattern --> 32;               ! For the current pattern match
Global pcount;                      ! and a marker within it
Array pattern2 --> 32;              ! And another, which stores the best match
Global pcount2;                     ! so far
Array  line_ttype-->32;             ! For storing an analysed grammar line
Array  line_tdata-->32;
Array  line_token-->32;
Global nsns;                        ! Number of special_numbers entered so far
Global params_wanted;               ! Number of parameters needed (which may change in parsing)
Global inferfrom;                   ! The point from which the rest of the command must be inferred
Global inferword;                   ! And the preposition inferred
Global dont_infer;                  ! Another dull flag
Global cobj_flag = 0;
Global oops_from;                   ! The "first mistake" word number
Global saved_oops;                  ! Used in working this out
Array  oops_workspace -> 64;        ! Used temporarily by "oops" routine
```

```
Global held_back_mode;              ! Flag: is there some input from last time
Global hb_wn;                       ! left over?  (And a save value for wn.)
                                      ! (Used for full stops and "then".)
Global usual_grammar_after;         ! Point from which usual grammar is parsed (it may vary from
                                      ! the above if user's routines match multi-word verbs)
```

§**2. Grammar Token Variables.** More globals, but dealing at the level of individual tokens now.

```
Constant PATTERN_NULL = $ffff;      ! Entry for a token producing no text
Global found_ttype;                 ! Used to break up tokens into type
Global found_tdata;                 ! and data (by AnalyseToken)
Global token_filter;                ! For noun filtering by user routines
Global length_of_noun;              ! Set by NounDomain to no of words in noun
Global lookahead;                   ! The token after the one now being matched
Global multi_mode;                  ! Multiple mode
Global multi_wanted;                ! Number of things needed in multitude
Global multi_had;                   ! Number of things actually found
Global multi_context;               ! What token the multi-obj was accepted for
Global indef_mode;                  ! "Indefinite" mode - ie, "take a brick"
                                      ! is in this mode
Global indef_type;                  ! Bit-map holding types of specification
Global indef_wanted;                ! Number of items wanted (INDEF_ALL_WANTED for all)
Constant INDEF_ALL_WANTED = 32767;
Global indef_guess_p;               ! Plural-guessing flag
Global indef_owner;                 ! Object which must hold these items
Global indef_cases;                 ! Possible gender and numbers of them
Global indef_possambig;             ! Has a possibly dangerous assumption
                                      ! been made about meaning of a descriptor?
Global indef_nspec_at;              ! Word at which a number like "two" was parsed
                                      ! (for backtracking)
Global allow_plurals;               ! Whether plurals presently allowed or not
Global take_all_rule;               ! Slightly different rules apply to "take all" than other uses
                                      ! of multiple objects, to make adjudication produce more
                                      ! pragmatically useful results
                                      ! (Not a flag: possible values 0, 1, 2)
Global dict_flags_of_noun;          ! Of the noun currently being parsed
                                      ! (a bitmap in #dict_par1 format)
Global pronoun__word;               ! Saved value
Global pronoun__obj;                ! Saved value
Constant comma_word = 'comma,';     ! An "untypeable word" used to substitute
                                      ! for commas in parse buffers
```

§**3. Match List Variables.** The most difficult tokens to match are those which refer to objects, since there is such a variety of names which can be given to any individual object, and we don't of course know which object or objects are meant. We store the possibilities (up to `MATCH_LIST_WORDS`, anyway) in a data structure called the match list.

```
Array  match_list --> MATCH_LIST_WORDS;    ! An array of matched objects so far
Array  match_classes --> MATCH_LIST_WORDS; ! An array of equivalence classes for them
Array  match_scores --> MATCH_LIST_WORDS;  ! An array of match scores for them
Global number_matched;              ! How many items in it?  (0 means none)
Global number_of_classes;           ! How many equivalence classes?
Global match_length;                ! How many words long are these matches?
Global match_from;                  ! At what word of the input do they begin?
```

§**4. Words.** The player's command is broken down into a numbered sequence of words, which break at spaces or certain punctuation (see the DM4). The numbering runs upwards from 1 to `WordCount()`. The following utility routines provide access to words in the current command; because buffers have different definitions in Z and Glulx, so these routines must vary also.

The actual text of each word is stored as a sequence of ZSCII values in a `->` (byte) array, with address `WordAddress(x)` and length `WordLength(x)`.

We picture the command as a stream of words to be read one at a time, with the global variable `wn` being the "current word" marker. `NextWord`, which takes no arguments, returns:

(a) 0 if the word at `wn` is unrecognised by the dictionary or `wn` is out of range,
(b) `comma_word` if the word was a comma,
(c) `THEN1__WD` if it was a full stop (because of the Infocom tradition that a full stop abbreviates for the word "then": e.g., TAKE BOX. EAST was read as two commands in succession),
(d) or the dictionary address if the word was recognised.

The current word marker `wn` is always advanced.

`NextWordStopped` does the same, but returns −1 when `wn` is out of range (e.g., by having advanced past the last word in the command).

`MoveWord(at1, b2, at2)` copies word `at2` from parse buffer `b2` – which doesn't need to be `buffer` – to word `at1` in `parse`.

```
#Ifdef TARGET_ZCODE;
[ WordCount; return parse->1; ];
[ WordAddress wordnum; return buffer + parse->(wordnum*4+1); ];
[ WordLength wordnum; return parse->(wordnum*4); ];
[ MoveWord at1 b2 at2 x y;
    x = at1*2-1; y = at2*2-1;
    parse-->x++ = b2-->y++;
    parse-->x = b2-->y;
];
#Ifnot;
[ WordCount; return parse-->0; ];
[ WordAddress wordnum; return buffer + parse-->(wordnum*3); ];
[ WordLength wordnum; return parse-->(wordnum*3-1); ];
[ MoveWord at1 b2 at2 x y;
    x = at1*3-2; y = at2*3-2;
    parse-->x++ = b2-->y++;
    parse-->x++ = b2-->y++;
    parse-->x = b2-->y;
];
#Endif;
```

```
[ WordFrom w p i j wc;
    #Ifdef TARGET_ZCODE; wc = p->1; i = w*2-1;
    #Ifnot; wc = p-->0; i = w*3-2; #Endif;
    if ((w < 1) || (w > wc)) return 0;
    j = p-->i;
    if (j == ',//') j = comma_word;
    if (j == './/') j = THEN1__WD;
    return j;
];
[ NextWord i j wc;
    #Ifdef TARGET_ZCODE; wc = parse->1; i = wn*2-1;
    #Ifnot; wc = parse-->0; i = wn*3-2; #Endif;
    wn++;
    if ((wn < 2) || (wn > wc+1)) return 0;
    j = parse-->i;
    if (j == ',//') j = comma_word;
    if (j == './/') j = THEN1__WD;
    return j;
];
[ NextWordStopped wc;
    #Ifdef TARGET_ZCODE; wc = parse->1; #Ifnot; wc = parse-->0; #Endif;
    if ((wn < 1) || (wn > wc)) { wn++; return -1; }
    return NextWord();
];
```

§**5. Snippets.** Although the idea is arguably implicit in I6, the formal concept of "snippet" is new in I7. A snippet is a value which represents a word range in the command most recently typed by the player. These words number consecutively upwards from 1, as noted above. The correspondence between (w_1, w_2), the word range, and V, the number used to represent it as an I6 value, is:

$$V = 100w_1 + (w_2 - w_1 + 1)$$

so that the remainder mod 100 is the number of words in the range. We require that $1 \leq w_1 \leq w_2 \leq N$, where N is the number of words in the current player's command. The entire command is therefore represented by:

$$C = 100 + N$$

```
[ PrintSnippet snip from to i w1 w2;
    w1 = snip/100; w2 = w1 + (snip%100) - 1;
    if ((w2<w1) || (w1<1) || (w2>WordCount())) {
        if ((w1 == 1) && (w2 == 0)) rfalse;
        return RunTimeProblem(RTP_SAYINVALIDSNIPPET, w1, w2);
    }
    from = WordAddress(w1); to = WordAddress(w2) + WordLength(w2) - 1;
    for (i=from: i<=to: i++) print (char) i->0;
];
[ SpliceSnippet snip t i w1 w2 nextw at endsnippet newlen;
    w1 = snip/100; w2 = w1 + (snip%100) - 1;
    if ((w2<w1) || (w1<1)) {
        if ((w1 == 1) && (w2 == 0)) return;
        return RunTimeProblem(RTP_SPLICEINVALIDSNIPPET, w1, w2);
    }
```

```
    @push say__p; @push say__pc;
    nextw = w2 + 1;
    at = WordAddress(w1) - buffer;
    if (nextw <= WordCount()) endsnippet = 100*nextw + (WordCount() - nextw + 1);
    buffer2-->0 = 120;
    newlen = VM_PrintToBuffer(buffer2, 120, SpliceSnippet__TextPrinter, t, endsnippet);
    for (i=0: (i<newlen) && (at+i<120): i++) buffer->(at+i) = buffer2->(WORDSIZE+i);
    #Ifdef TARGET_ZCODE; buffer->1 = at+i; #ifnot; buffer-->0 = at+i; #endif;
    for (:at+i<120:i++) buffer->(at+i) = ' ';
    VM_Tokenise(buffer, parse);
    players_command = 100 + WordCount();
    @pull say__pc; @pull say__p;
];
[ SpliceSnippet__TextPrinter t endsnippet;
    PrintText(t);
    if (endsnippet) { print " "; PrintSnippet(endsnippet); }
];
[ SnippetIncludes test snippet w1 w2 wlen i j;
    w1 = snippet/100; w2 = w1 + (snippet%100) - 1;
    if ((w2<w1) || (w1<1)) {
        if ((w1 == 1) && (w2 == 0)) rfalse;
        return RunTimeProblem(RTP_INCLUDEINVALIDSNIPPET, w1, w2);
    }
    if (metaclass(test) == Routine) {
        wlen = snippet%100;
        for (i=w1, j=wlen: j>0: i++, j--) {
            if (((test)(i, 0)) ~= GPR_FAIL) return i*100+wn-i;
        }
    }
    rfalse;
];
[ SnippetMatches snippet topic_gpr rv;
    wn=1;
    if (topic_gpr == 0) rfalse;
    if (metaclass(topic_gpr) == Routine) {
        rv = (topic_gpr)(snippet/100, snippet%100);
        if (rv ~= GPR_FAIL) rtrue;
        rfalse;
    }
    RunTimeProblem(RTP_BADTOPIC);
    rfalse;
];
```

§**6. Unpacking Grammar Lines.** Grammar lines are sequences of tokens in an array built into the story file, but in a format which differs depending on the virtual machine in use, so the following code unpacks the data into more convenient if larger arrays which are VM-independent.

```
[ UnpackGrammarLine line_address i size;
    for (i=0 : i<32 : i++) {
        line_token-->i = ENDIT_TOKEN;
        line_ttype-->i = ELEMENTARY_TT;
        line_tdata-->i = ENDIT_TOKEN;
    }
#Ifdef TARGET_ZCODE;
    action_to_be = 256*(line_address->0) + line_address->1;
    action_reversed = ((action_to_be & $400) ~= 0);
    action_to_be = action_to_be & $3ff;
    line_address--;
    size = 3;
#Ifnot; ! GLULX
    @aloads line_address 0 action_to_be;
    action_reversed = (((line_address->2) & 1) ~= 0);
    line_address = line_address - 2;
    size = 5;
#Endif;
    params_wanted = 0;
    for (i=0 : : i++) {
        line_address = line_address + size;
        if (line_address->0 == ENDIT_TOKEN) break;
        line_token-->i = line_address;
        AnalyseToken(line_address);
        if (found_ttype ~= PREPOSITION_TT) params_wanted++;
        line_ttype-->i = found_ttype;
        line_tdata-->i = found_tdata;
    }
    return line_address + 1;
];
[ AnalyseToken token;
    if (token == ENDIT_TOKEN) {
        found_ttype = ELEMENTARY_TT;
        found_tdata = ENDIT_TOKEN;
        return;
    }
    found_ttype = (token->0) & $$1111;
    found_tdata = (token+1)-->0;
];
```

§**7. Extracting Verb Numbers.** A long tale of woe lies behind the following. Infocom games stored verb numbers in a single byte in dictionary entries, but they did so counting downwards, so that verb number 0 was stored as 255, 1 as 254, and so on. Inform followed suit so that debugging of Inform 1 could be aided by using the then-available tools for dumping dictionaries from Infocom story files; by using the Infocom format for dictionary tables, Inform's life was easier.

But there was an implicit restriction there of 255 distinct verbs (not 256 since not all words were verbs). When Glulx raised almost all of the Z-machine limits, it made space for 65535 verbs instead of 255, but it appears that nobody remembered to implement this in I6-for-Glulx and the Glulx form of the I6 library. This was only put right in March 2009, and the following routine was added to concentrate lookups of this field in one place.

```
[ DictionaryWordToVerbNum dword verbnum;
#Ifdef TARGET_ZCODE;
    verbnum = $ff-(dword->#dict_par2);
#Ifnot; ! GLULX
    dword = dword + #dict_par2 - 1;
    @aloads dword 0 verbnum;
    verbnum = $ffff-verbnum;
#Endif;
    return verbnum;
];
```

§**8. Keyboard Primitive.** This is the primitive routine to read from the keyboard: it usually delegates this to a routine specific to the virtual machine being used, but sometimes uses a hacked version to allow TEST commands to work. (When a TEST is running, the text in the walk-through provided is fed into the buffer as if it had been typed at the keyboard.)

```
[ KeyboardPrimitive a_buffer a_table;
#Ifdef DEBUG; #Iftrue ({-value:NUMBER_CREATED(test_scenario)} > 0);
    return TestKeyboardPrimitive(a_buffer, a_table);
#Endif; #Endif;
    return VM_ReadKeyboard(a_buffer, a_table);
];
```

§**9. Reading the Command.** The `Keyboard` routine actually receives the player's words, putting the words in `a_buffer` and their dictionary addresses in `a_table`. It is assumed that the table is the same one on each (standard) call. Much of the code handles the OOPS and UNDO commands, which are not actions and do not pass through the rest of the parser. The undo state is saved – it is essentially an internal saved game, in the VM interpreter's memory rather than in an external file – and note that this is therefore also where execution picks up if an UNDO has been typed. Since UNDO recreates the former machine state perfectly, it might seem impossible to tell that an UNDO had occurred, but in fact the VM passes information back in the form of a return code from the relevant instruction, and this allows us to detect an undo. (We deal with it by printing the current location and asking another command.)

`Keyboard` can also be used by miscellaneous routines in the game to ask yes/no questions and the like, without invoking the rest of the parser.

The return value is the number of words typed.

```
[ Keyboard  a_buffer a_table  nw i w w2 x1 x2;
    sline1 = score; sline2 = turns;
    while (true) {
        ! Save the start of the buffer, in case "oops" needs to restore it
```

```
for (i=0 : i<64 : i++) oops_workspace->i = a_buffer->i;

! In case of an array entry corruption that shouldn't happen, but would be
! disastrous if it did:
#Ifdef TARGET_ZCODE;
a_buffer->0 = INPUT_BUFFER_LEN;
a_table->0 = 15;  ! Allow to split input into this many words
#Endif; ! TARGET_

! Print the prompt, and read in the words and dictionary addresses
PrintPrompt();
DrawStatusLine();
KeyboardPrimitive(a_buffer, a_table);

! Set nw to the number of words
#Ifdef TARGET_ZCODE; nw = a_table->1; #Ifnot; nw = a_table-->0; #Endif;

! If the line was blank, get a fresh line
if (nw == 0) {
    @push etype; etype = BLANKLINE_PE;
    players_command = 100;
    BeginActivity(PRINTING_A_PARSER_ERROR_ACT);
    if (ForActivity(PRINTING_A_PARSER_ERROR_ACT) == false) L__M(##Miscellany,10);
    EndActivity(PRINTING_A_PARSER_ERROR_ACT);
    @pull etype;
    continue;
}

! Unless the opening word was OOPS, return
! Conveniently, a_table-->1 is the first word on both the Z-machine and Glulx

w = a_table-->1;
if (w == OOPS1__WD or OOPS2__WD or OOPS3__WD) {
    if (oops_from == 0) { L__M(##Miscellany, 14); continue; }
    if (nw == 1) { L__M(##Miscellany, 15); continue; }
    if (nw > 2) { L__M(##Miscellany, 16); continue; }

    ! So now we know: there was a previous mistake, and the player has
    ! attempted to correct a single word of it.

    for (i=0 : i<INPUT_BUFFER_LEN : i++) buffer2->i = a_buffer->i;
    #Ifdef TARGET_ZCODE;
    x1 = a_table->9;  ! Start of word following "oops"
    x2 = a_table->8;  ! Length of word following "oops"
    #Ifnot; ! TARGET_GLULX
    x1 = a_table-->6; ! Start of word following "oops"
    x2 = a_table-->5; ! Length of word following "oops"
    #Endif; ! TARGET_

    ! Repair the buffer to the text that was in it before the "oops"
    ! was typed:
    for (i=0 : i<64 : i++) a_buffer->i = oops_workspace->i;
    VM_Tokenise(a_buffer,a_table);

    ! Work out the position in the buffer of the word to be corrected:
    #Ifdef TARGET_ZCODE;
    w = a_table->(4*oops_from + 1); ! Start of word to go
    w2 = a_table->(4*oops_from);    ! Length of word to go
    #Ifnot; ! TARGET_GLULX
    w = a_table-->(3*oops_from);      ! Start of word to go
    w2 = a_table-->(3*oops_from - 1); ! Length of word to go
```

```
            #Endif; ! TARGET_

            ! Write spaces over the word to be corrected:
            for (i=0 : i<w2 : i++) a_buffer->(i+w) = ' ';

            if (w2 < x2) {
                ! If the replacement is longer than the original, move up...
                for (i=INPUT_BUFFER_LEN-1 : i>=w+x2 : i--)
                    a_buffer->i = a_buffer->(i-x2+w2);

                ! ...increasing buffer size accordingly.
                #Ifdef TARGET_ZCODE;
                a_buffer->1 = (a_buffer->1) + (x2-w2);
                #Ifnot; ! TARGET_GLULX
                a_buffer-->0 = (a_buffer-->0) + (x2-w2);
                #Endif; ! TARGET_
            }

            ! Write the correction in:
            for (i=0 : i<x2 : i++) a_buffer->(i+w) = buffer2->(i+x1);

            VM_Tokenise(a_buffer, a_table);
            #Ifdef TARGET_ZCODE; nw = a_table->1; #Ifnot; nw = a_table-->0; #Endif;

            return nw;
        }

        ! Undo handling

        if ((w == UNDO1__WD or UNDO2__WD or UNDO3__WD) && (nw==1)) {
            Perform_Undo();
            continue;
        }
        i = VM_Save_Undo();
        #ifdef PREVENT_UNDO; undo_flag = 0; #endif;
        #ifndef PREVENT_UNDO; undo_flag = 2; #endif;
        if (i == -1) undo_flag = 0;
        if (i == 0) undo_flag = 1;
        if (i == 2) {
            VM_RestoreWindowColours();
            VM_Style(SUBHEADER_VMSTY);
            SL_Location(); print "^";
            ! print (name) location, "^";
            VM_Style(NORMAL_VMSTY);
            L__M(##Miscellany, 13);
            continue;
        }
        return nw;
    }
];
```

§**10. Parser Proper.** The main parser routine is something of a leviathan, and it has traditionally been divided into 11 lettered parts:

(A) Get the input, do OOPS and AGAIN
(B) Is it a direction, and so an implicit GO? If so go to (K)
(C) Is anyone being addressed?
(D) Get the command verb: try all the syntax lines for that verb
(E) Break down a syntax line into analysed tokens
(F) Look ahead for advance warning for `multiexcept`/`multiinside`
(G) Parse each token in turn (calling `ParseToken` to do most of the work)
(H) Cheaply parse otherwise unrecognised conversation and return
(I) Print best possible error message
(J) Retry the whole lot
(K) Last thing: check for THEN and further instructions(s), return.

This lettering has been preserved here, with the code under each letter now being the body of "Parser Letter A", "Parser Letter B" and so on.

Note that there are three different places where a return can happen. The routine returns only when a sensible request has been made; for a fairly thorough description of its output, which is written into the `parser_results` array and also into several globals (see "OrderOfPlay.i6t").

```
[ Parser__parse
    syntax line num_lines line_address i j k token l m;
    cobj_flag = 0;
    parser_results-->ACTION_PRES = 0;
    parser_results-->NO_INPS_PRES = 0;
    parser_results-->INP1_PRES = 0;
    parser_results-->INP2_PRES = 0;
    meta = false;
```

§**11. Parser Letter A.** Get the input, do OOPS and AGAIN.

```
    if (held_back_mode == 1) {
        held_back_mode = 0;
        VM_Tokenise(buffer, parse);
        jump ReParse;
    }
.ReType;
    cobj_flag = 0;
    actors_location = ScopeCeiling(player);
    BeginActivity(READING_A_COMMAND_ACT); if (ForActivity(READING_A_COMMAND_ACT)==false) {
        Keyboard(buffer,parse);
        players_command = 100 + WordCount();
        num_words = WordCount();
    } if (EndActivity(READING_A_COMMAND_ACT)) jump ReType;
.ReParse;
    parser_inflection = name;
    ! Initially assume the command is aimed at the player, and the verb
    ! is the first word
    num_words = WordCount();
    wn = 1;
    #Ifdef LanguageToInformese;
```

```
    LanguageToInformese();
    ! Re-tokenise:
    VM_Tokenise(buffer,parse);
    #Endif; ! LanguageToInformese
    num_words = WordCount();
    k=0;
    #Ifdef DEBUG;
    if (parser_trace >= 2) {
        print "[ ";
        for (i=0 : i<num_words : i++) {
            #Ifdef TARGET_ZCODE;
            j = parse-->(i*2 + 1);
            #Ifnot; ! TARGET_GLULX
            j = parse-->(i*3 + 1);
            #Endif; ! TARGET_
            k = WordAddress(i+1);
            l = WordLength(i+1);
            print "~"; for (m=0 : m<l : m++) print (char) k->m; print "~ ";
            if (j == 0) print "?";
            else {
                #Ifdef TARGET_ZCODE;
                if (UnsignedCompare(j, HDR_DICTIONARY-->0) >= 0 &&
                    UnsignedCompare(j, HDR_HIGHMEMORY-->0) < 0)
                    print (address) j;
                else print j;
                #Ifnot; ! TARGET_GLULX
                if (j->0 == $60) print (address) j;
                else print j;
                #Endif; ! TARGET_
            }
            if (i ~= num_words-1) print " / ";
        }
        print " ]^";
    }
    #Endif; ! DEBUG
    verb_wordnum = 1;
    actor = player;
    actors_location = ScopeCeiling(player);
    usual_grammar_after = 0;
.AlmostReParse;
    scope_token = 0;
    action_to_be = NULL;
    ! Begin from what we currently think is the verb word
.BeginCommand;
    wn = verb_wordnum;
    verb_word = NextWordStopped();
    ! If there's no input here, we must have something like "person,".
    if (verb_word == -1) {
        best_etype = STUCK_PE;
        jump GiveError;
    }
```

```
! Now try for "again" or "g", which are special cases: don't allow "again" if nothing
! has previously been typed; simply copy the previous text across
if (verb_word == AGAIN2__WD or AGAIN3__WD) verb_word = AGAIN1__WD;
if (verb_word == AGAIN1__WD) {
    if (actor ~= player) {
        L__M(##Miscellany, 20);
        jump ReType;
    }
    #Ifdef TARGET_ZCODE;
    if (buffer3->1 == 0) {
        L__M(##Miscellany, 21);
        jump ReType;
    }
    #Ifnot; ! TARGET_GLULX
    if (buffer3-->0 == 0) {
        L__M(##Miscellany, 21);
        jump ReType;
    }
    #Endif; ! TARGET_
    for (i=0 : i<INPUT_BUFFER_LEN : i++) buffer->i = buffer3->i;
    VM_Tokenise(buffer,parse);
    num_words = WordCount();
    players_command = 100 + WordCount();
    jump ReParse;
}
! Save the present input in case of an "again" next time
if (verb_word ~= AGAIN1__WD)
    for (i=0 : i<INPUT_BUFFER_LEN : i++) buffer3->i = buffer->i;
if (usual_grammar_after == 0) {
    j = verb_wordnum;
    i = RunRoutines(actor, grammar);
    #Ifdef DEBUG;
    if (parser_trace >= 2 && actor.grammar ~= 0 or NULL)
        print " [Grammar property returned ", i, "]^";
    #Endif; ! DEBUG
    if ((i ~= 0 or 1) && (VM_InvalidDictionaryAddress(i))) {
        usual_grammar_after = verb_wordnum; i=-i;
    }
    if (i == 1) {
        parser_results-->ACTION_PRES = action;
        parser_results-->NO_INPS_PRES = 0;
        parser_results-->INP1_PRES = noun;
        parser_results-->INP2_PRES = second;
        if (noun) parser_results-->NO_INPS_PRES = 1;
        if (second) parser_results-->NO_INPS_PRES = 2;
        rtrue;
    }
    if (i ~= 0) { verb_word = i; wn--; verb_wordnum--; }
    else { wn = verb_wordnum; verb_word = NextWord(); }
}
else usual_grammar_after = 0;
```

§**12. Parser Letter B.** Is the command a direction name, and so an implicit GO? If so, go to (K).

```
#Ifdef LanguageIsVerb;
if (verb_word == 0) {
    i = wn; verb_word = LanguageIsVerb(buffer, parse, verb_wordnum);
    wn = i;
}
#Endif; ! LanguageIsVerb

! If the first word is not listed as a verb, it must be a direction
! or the name of someone to talk to

if (verb_word == 0 || ((verb_word->#dict_par1) & 1) == 0) {

    ! So is the first word an object contained in the special object "compass"
    ! (i.e., a direction)?  This needs use of NounDomain, a routine which
    ! does the object matching, returning the object number, or 0 if none found,
    ! or REPARSE_CODE if it has restructured the parse table so the whole parse
    ! must be begun again...

    wn = verb_wordnum; indef_mode = false; token_filter = 0; parameters = 0;
    @push actor; @push action; @push action_to_be;
    actor = player; meta = false; action = ##Go; action_to_be = ##Go;
    l = NounDomain(compass, 0, 0);
    @pull action_to_be; @pull action; @pull actor;
    if (l == REPARSE_CODE) jump ReParse;

    ! If it is a direction, send back the results:
    ! action=GoSub, no of arguments=1, argument 1=the direction.

    if ((l~=0) && (l ofclass K3_direction)) {
        parser_results-->ACTION_PRES = ##Go;
        parser_results-->NO_INPS_PRES = 1;
        parser_results-->INP1_PRES = l;
        jump LookForMore;
    }

} ! end of first-word-not-a-verb
```

§**13. Parser Letter C.** Is anyone being addressed?

```
! Only check for a comma (a "someone, do something" command) if we are
! not already in the middle of one.  (This simplification stops us from
! worrying about "robot, wizard, you are an idiot", telling the robot to
! tell the wizard that she is an idiot.)

if (actor == player) {
    for (j=2 : j<=num_words : j++) {
        i=NextWord();
        if (i == comma_word) jump Conversation;
    }
}
jump NotConversation;

! NextWord nudges the word number wn on by one each time, so we've now
! advanced past a comma.  (A comma is a word all on its own in the table.)

.Conversation;

j = wn - 1;
if (j == 1) {
```

```
    L__M(##Miscellany, 22);
    jump ReType;
}
! Use NounDomain (in the context of "animate creature") to see if the
! words make sense as the name of someone held or nearby
wn = 1; lookahead = HELD_TOKEN;
scope_reason = TALKING_REASON;
l = NounDomain(player,actors_location,6);
scope_reason = PARSING_REASON;
if (l == REPARSE_CODE) jump ReParse;
if (l == 0) {
    if (verb_word && ((verb_word->#dict_par1) & 1)) jump NotConversation;
    L__M(##Miscellany, 23);
    jump ReType;
}
.Conversation2;
! The object addressed must at least be "talkable" if not actually "animate"
! (the distinction allows, for instance, a microphone to be spoken to,
! without the parser thinking that the microphone is human).
if (l hasnt animate && l hasnt talkable) {
    L__M(##Miscellany, 24, l);
    jump ReType;
}
! Check that there aren't any mystery words between the end of the person's
! name and the comma (eg, throw out "dwarf sdfgsdgs, go north").
if (wn ~= j) {
    if (verb_word && ((verb_word->#dict_par1) & 1)) jump NotConversation;
    L__M(##Miscellany, 25);
    jump ReType;
}
! The player has now successfully named someone.  Adjust "him", "her", "it":
PronounNotice(l);
! Set the global variable "actor", adjust the number of the first word,
! and begin parsing again from there.
verb_wordnum = j + 1;
! Stop things like "me, again":
if (l == player) {
    wn = verb_wordnum;
    if (NextWordStopped() == AGAIN1__WD or AGAIN2__WD or AGAIN3__WD) {
        L__M(##Miscellany, 20);
        jump ReType;
    }
}
actor = l;
actors_location = ScopeCeiling(l);
#Ifdef DEBUG;
if (parser_trace >= 1)
    print "[Actor is ", (the) actor, " in ", (name) actors_location, "]^";
#Endif; ! DEBUG
jump BeginCommand;
```

§**14. Parser Letter D.** Get the verb: try all the syntax lines for that verb.

```
.NotConversation;
if (verb_word == 0 || ((verb_word->#dict_par1) & 1) == 0) {
    if (actor == player) {
        verb_word = UnknownVerb(verb_word);
        if (verb_word ~= 0) jump VerbAccepted;
    }
    best_etype = VERB_PE;
    jump GiveError;
}
.VerbAccepted;
! We now definitely have a verb, not a direction, whether we got here by the
! "take ..." or "person, take ..." method.  Get the meta flag for this verb:
meta = ((verb_word->#dict_par1) & 2)/2;
! You can't order other people to "full score" for you, and so on...
if (meta == 1 && actor ~= player) {
    best_etype = VERB_PE;
    meta = 0;
    jump GiveError;
}
! Now let i be the corresponding verb number...
i = DictionaryWordToVerbNum(verb_word);
! ...then look up the i-th entry in the verb table, whose address is at word
! 7 in the Z-machine (in the header), so as to get the address of the syntax
! table for the given verb...
#Ifdef TARGET_ZCODE;
syntax = (HDR_STATICMEMORY-->0)-->i;
#Ifnot; ! TARGET_GLULX
syntax = (#grammar_table)-->(i+1);
#Endif; ! TARGET_
! ...and then see how many lines (ie, different patterns corresponding to the
! same verb) are stored in the parse table...
num_lines = (syntax->0) - 1;
! ...and now go through them all, one by one.
! To prevent pronoun_word 0 being misunderstood,
pronoun_word = NULL; pronoun_obj = NULL;
#Ifdef DEBUG;
if (parser_trace >= 1)
    print "[Parsing for the verb '", (address) verb_word, "' (", num_lines+1, " lines)]^";
#Endif; ! DEBUG
best_etype = STUCK_PE; nextbest_etype = STUCK_PE;
multiflag = false;
! "best_etype" is the current failure-to-match error - it is by default
! the least informative one, "don't understand that sentence".
! "nextbest_etype" remembers the best alternative to having to ask a
! scope token for an error message (i.e., the best not counting ASKSCOPE_PE).
! multiflag is used here to prevent inappropriate MULTI_PE errors
! in addition to its unrelated duties passing information to action routines
```

§**15. Parser Letter E.** Break down a syntax line into analysed tokens.

```
line_address = syntax + 1;
for (line=0 : line<=num_lines : line++) {
    for (i=0 : i<32 : i++) {
        line_token-->i = ENDIT_TOKEN;
        line_ttype-->i = ELEMENTARY_TT;
        line_tdata-->i = ENDIT_TOKEN;
    }
    ! Unpack the syntax line from Inform format into three arrays; ensure that
    ! the sequence of tokens ends in an ENDIT_TOKEN.
    line_address = UnpackGrammarLine(line_address);
    #Ifdef DEBUG;
    if (parser_trace >= 1) {
        if (parser_trace >= 2) new_line;
        print "[line ", line; DebugGrammarLine();
        print "]^";
    }
    #Endif; ! DEBUG
    ! We aren't in "not holding" or inferring modes, and haven't entered
    ! any parameters on the line yet, or any special numbers; the multiple
    ! object is still empty.
    inferfrom = 0;
    parameters = 0;
    nsns = 0; special_word = 0;
    multiple_object-->0 = 0;
    multi_context = 0;
    etype = STUCK_PE;
    ! Put the word marker back to just after the verb
    wn = verb_wordnum+1;
```

§**16. Parser Letter F.** Look ahead for advance warning for `multiexcept`/`multiinside`.

There are two special cases where parsing a token now has to be affected by the result of parsing another token later, and these two cases (multiexcept and multiinside tokens) are helped by a quick look ahead, to work out the future token now. We can only carry this out in the simple (but by far the most common) case:

```
multiexcept <one or more prepositions> noun
```

and similarly for `multiinside`.

```
    advance_warning = -1; indef_mode = false;
    for (i=0,m=false,pcount=0 : line_token-->pcount ~= ENDIT_TOKEN : pcount++) {
        scope_token = 0;
        if (line_ttype-->pcount ~= PREPOSITION_TT) i++;
        if (line_ttype-->pcount == ELEMENTARY_TT) {
            if (line_tdata-->pcount == MULTI_TOKEN) m = true;
            if (line_tdata-->pcount == MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN  && i == 1) {
                ! First non-preposition is "multiexcept" or
                ! "multiinside", so look ahead.
                #Ifdef DEBUG;
                if (parser_trace >= 2) print " [Trying look-ahead]^";
```

```
#Endif; ! DEBUG
! We need this to be followed by 1 or more prepositions.
pcount++;
if (line_ttype-->pcount == PREPOSITION_TT) {
    ! skip ahead to a preposition word in the input
    do {
        l = NextWord();
    } until ((wn > num_words) ||
             (l && (l->#dict_par1) & 8 ~= 0));
    if (wn > num_words) {
        #Ifdef DEBUG;
        if (parser_trace >= 2)
            print " [Look-ahead aborted: prepositions missing]^";
        #Endif;
        jump LineFailed;
    }
    do {
        if (PrepositionChain(l, pcount) ~= -1) {
            ! advance past the chain
            if ((line_token-->pcount)->0 & $20 ~= 0) {
                pcount++;
                while ((line_token-->pcount ~= ENDIT_TOKEN) &&
                    ((line_token-->pcount)->0 & $10 ~= 0))
                    pcount++;
            } else {
                pcount++;
            }
        } else {
            ! try to find another preposition word
            do {
                l = NextWord();
            } until ((wn >= num_words) ||
                     (l && (l->#dict_par1) & 8 ~= 0));
            if (l && (l->#dict_par1) & 8) continue;
            ! lookahead failed
            #Ifdef DEBUG;
            if (parser_trace >= 2)
                print " [Look-ahead aborted: prepositions don't match]^";
            #endif;
            jump LineFailed;
        }
        l = NextWord();
    } until (line_ttype-->pcount ~= PREPOSITION_TT);
    ! put back the non-preposition we just read
    wn--;
    if ((line_ttype-->pcount == ELEMENTARY_TT) &&
        (line_tdata-->pcount == NOUN_TOKEN)) {
        l = Descriptors();  ! skip past THE etc
        if (l~=0) etype=l;  ! don't allow multiple objects
        k = parser_results-->INP1_PRES; @push k; @push parameters;
        parameters = 1; parser_results-->INP1_PRES = 0;
```

```
                    l = NounDomain(actors_location, actor, NOUN_TOKEN);
                    @pull parameters; @pull k; parser_results-->INP1_PRES = k;
                    #Ifdef DEBUG;
                    if (parser_trace >= 2) {
                        print " [Advanced to ~noun~ token: ";
                        if (l == REPARSE_CODE) print "re-parse request]^";
                        else {
                            if (l == 1) print "but multiple found]^";
                            if (l == 0) print "error ", etype, "]^";
                            if (l >= 2) print (the) l, "]^";
                        }
                    }
                    #Endif; ! DEBUG
                    if (l == REPARSE_CODE) jump ReParse;
                    if (l >= 2) advance_warning = l;
                }
            }
            break;
        }
    }
}

! Slightly different line-parsing rules will apply to "take multi", to
! prevent "take all" behaving correctly but misleadingly when there's
! nothing to take.

take_all_rule = 0;
if (m && params_wanted == 1 && action_to_be == ##Take)
    take_all_rule = 1;

! And now start again, properly, forearmed or not as the case may be.
! As a precaution, we clear all the variables again (they may have been
! disturbed by the call to NounDomain, which may have called outside
! code, which may have done anything!).

inferfrom = 0;
parameters = 0;
nsns = 0; special_word = 0;
multiple_object-->0 = 0;
etype = STUCK_PE;
wn = verb_wordnum+1;
```

§**17. Parser Letter G.** Parse each token in turn (calling `ParseToken` to do most of the work).

The `pattern` gradually accumulates what has been recognised so far, so that it may be reprinted by the parser later on.

```
for (pcount=1 : : pcount++) {
    pattern-->pcount = PATTERN_NULL; scope_token = 0;
    token = line_token-->(pcount-1);
    lookahead = line_token-->pcount;
    #Ifdef DEBUG;
    if (parser_trace >= 2)
        print " [line ", line, " token ", pcount, " word ", wn, " : ", (DebugToken) token,
        "]^";
    #Endif; ! DEBUG
    if (token ~= ENDIT_TOKEN) {
        scope_reason = PARSING_REASON;
        AnalyseToken(token);
        l = ParseToken(found_ttype, found_tdata, pcount-1, token);
        while ((l >= GPR_NOUN) && (l < -1)) l = ParseToken(ELEMENTARY_TT, l + 256);
        scope_reason = PARSING_REASON;
        if (l == GPR_PREPOSITION) {
            if (found_ttype~=PREPOSITION_TT && (found_ttype~=ELEMENTARY_TT ||
                found_tdata~=TOPIC_TOKEN)) params_wanted--;
            l = true;
        }
        else
            if (l < 0) l = false;
            else
                if (l ~= GPR_REPARSE) {
                    if (l == GPR_NUMBER) {
                        if (nsns == 0) special_number1 = parsed_number;
                        else special_number2 = parsed_number;
                        nsns++; l = 1;
                    }
                    if (l == GPR_MULTIPLE) l = 0;
                    parser_results-->(parameters+INP1_PRES) = l;
                    parameters++;
                    pattern-->pcount = l;
                    l = true;
                }
        #Ifdef DEBUG;
        if (parser_trace >= 3) {
            print "  [token resulted in ";
            if (l == REPARSE_CODE) print "re-parse request]^";
            if (l == 0) print "failure with error type ", etype, "]^";
            if (l == 1) print "success]^";
        }
        #Endif; ! DEBUG
        if (l == REPARSE_CODE) jump ReParse;
        if (l == false) break;
    }
    else {
```

```
! If the player has entered enough already but there's still
! text to wade through: store the pattern away so as to be able to produce
! a decent error message if this turns out to be the best we ever manage,
! and in the mean time give up on this line

! However, if the superfluous text begins with a comma or "then" then
! take that to be the start of another instruction

if (wn <= num_words) {
    l = NextWord();
    if (l == THEN1__WD or THEN2__WD or THEN3__WD or comma_word) {
        held_back_mode = 1; hb_wn = wn-1;
    }
    else {
        for (m=0 : m<32 : m++) pattern2-->m = pattern-->m;
        pcount2 = pcount;
        etype = UPTO_PE;
        break;
    }
}

! Now, we may need to revise the multiple object because of the single one
! we now know (but didn't when the list was drawn up).

if (parameters >= 1 && parser_results-->INP1_PRES == 0) {
    l = ReviseMulti(parser_results-->INP2_PRES);
    if (l ~= 0) { etype = l; parser_results-->ACTION_PRES = action_to_be; break; }
}
if (parameters >= 2 && parser_results-->INP2_PRES == 0) {
    l = ReviseMulti(parser_results-->INP1_PRES);
    if (l ~= 0) { etype = l; break; }
}

! To trap the case of "take all" inferring only "yourself" when absolutely
! nothing else is in the vicinity...

if (take_all_rule == 2 && parser_results-->INP1_PRES == actor) {
    best_etype = NOTHING_PE;
    jump GiveError;
}

#Ifdef DEBUG;
if (parser_trace >= 1) print "[Line successfully parsed]^";
#Endif; ! DEBUG

! The line has successfully matched the text.  Declare the input error-free...

oops_from = 0;

! ...explain any inferences made (using the pattern)...

if (inferfrom ~= 0) {
    PrintInferredCommand(inferfrom);
    ClearParagraphing();
}

! ...copy the action number, and the number of parameters...

parser_results-->ACTION_PRES = action_to_be;
parser_results-->NO_INPS_PRES = parameters;

! ...reverse first and second parameters if need be...

if (action_reversed && parameters == 2) {
    i = parser_results-->INP1_PRES;
```

```
                parser_results-->INP1_PRES = parser_results-->INP2_PRES;
                parser_results-->INP2_PRES = i;
                if (nsns == 2) {
                    i = special_number1; special_number1 = special_number2;
                    special_number2 = i;
                }
            }
            ! ...and to reset "it"-style objects to the first of these parameters, if
            ! there is one (and it really is an object)...
            if (parameters > 0 && parser_results-->INP1_PRES >= 2)
                PronounNotice(parser_results-->INP1_PRES);
            ! ...and return from the parser altogether, having successfully matched
            ! a line.
            if (held_back_mode == 1) {
                wn=hb_wn;
                jump LookForMore;
            }
            rtrue;
        } ! end of if(token ~= ENDIT_TOKEN) else
    } ! end of for(pcount++)
    .LineFailed;
    ! The line has failed to match.
    ! We continue the outer "for" loop, trying the next line in the grammar.
    if (etype > best_etype) best_etype = etype;
    if (etype ~= ASKSCOPE_PE && etype > nextbest_etype) nextbest_etype = etype;
    ! ...unless the line was something like "take all" which failed because
    ! nothing matched the "all", in which case we stop and give an error now.
    if (take_all_rule == 2 && etype==NOTHING_PE) break;
} ! end of for(line++)
! The grammar is exhausted: every line has failed to match.
```

§**18. Parser Letter H.** Cheaply parse otherwise unrecognised conversation and return.

(Errors are handled differently depending on who was talking. If the command was addressed to somebody else (eg, DWARF, SFGH) then it is taken as conversation which the parser has no business in disallowing.)

The parser used to return the fake action `##NotUnderstood` when a command in the form PERSON, ARFLE BARFLE GLOOP is parsed, where a character is addressed but with an instruction which the parser can't understand. (If a command such as ARFLE BARFLE GLOOP is not an instruction to someone else, the parser prints an error and requires the player to type another command: thus `##NotUnderstood` was only returned when `actor` is not the player.) And I6 had elaborate object-oriented ways to deal with this, but we won't use any of that: we simply convert to a `##Answer` action, which communicates a snippet of words to another character, just as if the player had typed ANSWER ARFLE BARFLE GLOOP TO PERSON. For I7 purposes, the fake action `##NotUnderstood` does not exist.

```
.GiveError;
    etype = best_etype;
    if (actor ~= player) {
        if (usual_grammar_after ~= 0) {
            verb_wordnum = usual_grammar_after;
            jump AlmostReParse;
```

```
    }
    wn = verb_wordnum;
    special_word = NextWord();
    if (special_word == comma_word) {
        special_word = NextWord();
        verb_wordnum++;
    }
    parser_results-->ACTION_PRES = ##Answer;
    parser_results-->NO_INPS_PRES = 2;
    parser_results-->INP1_PRES = actor;
    parser_results-->INP2_PRES = 1; special_number1 = special_word;
    actor = player;
    consult_from = verb_wordnum; consult_words = num_words-consult_from+1;
    rtrue;
}
```

§**19. Parser Letter I.** Print best possible error message.

```
! If the player was the actor (eg, in "take dfghh") the error must be printed,
! and fresh input called for.  In three cases the oops word must be jiggled.
if ((etype ofclass Routine) || (etype ofclass String)) {
    if (ParserError(etype) ~= 0) jump ReType;
} else {
    if (verb_wordnum == 0 && etype == CANTSEE_PE) etype = VERB_PE;
    players_command = 100 + WordCount(); ! The snippet variable ‘‘player’s command’’
    BeginActivity(PRINTING_A_PARSER_ERROR_ACT);
    if (ForActivity(PRINTING_A_PARSER_ERROR_ACT)) jump SkipParserError;
}
pronoun_word = pronoun__word; pronoun_obj = pronoun__obj;
if (etype == STUCK_PE) {    L__M(##Miscellany, 27); oops_from = 1; }
if (etype == UPTO_PE) {     L__M(##Miscellany, 28);
    for (m=0 : m<32 : m++) pattern-->m = pattern2-->m;
    pcount = pcount2; PrintCommand(0); L__M(##Miscellany, 56);
}
if (etype == NUMBER_PE)     L__M(##Miscellany, 29);
if (etype == CANTSEE_PE) {  L__M(##Miscellany, 30); oops_from=saved_oops; }
if (etype == TOOLIT_PE)     L__M(##Miscellany, 31);
if (etype == NOTHELD_PE) {  L__M(##Miscellany, 32); oops_from=saved_oops; }
if (etype == MULTI_PE)      L__M(##Miscellany, 33);
if (etype == MMULTI_PE)     L__M(##Miscellany, 34);
if (etype == VAGUE_PE)      L__M(##Miscellany, 35);
if (etype == EXCEPT_PE)     L__M(##Miscellany, 36);
if (etype == ANIMA_PE)      L__M(##Miscellany, 37);
if (etype == VERB_PE)       L__M(##Miscellany, 38);
if (etype == SCENERY_PE)    L__M(##Miscellany, 39);
if (etype == ITGONE_PE) {
    if (pronoun_obj == NULL)
                              L__M(##Miscellany, 35);
    else                     L__M(##Miscellany, 40);
}
if (etype == JUNKAFTER_PE)  L__M(##Miscellany, 41);
if (etype == TOOFEW_PE)     L__M(##Miscellany, 42, multi_had);
```

```
if (etype == NOTHING_PE) {
    if (parser_results-->ACTION_PRES == ##Remove &&
        parser_results-->INP2_PRES ofclass Object) {
        noun = parser_results-->INP2_PRES; ! ensure valid for messages
        if (noun has animate) L__M(##Take, 6, noun);
        else if (noun hasnt container or supporter) L__M(##Insert, 2, noun);
        else if (noun has container && noun hasnt open) L__M(##Take, 9, noun);
        else if (children(noun)==0) L__M(##Search, 6, noun);
        else parser_results-->ACTION_PRES = 0;
        }
    if (parser_results-->ACTION_PRES ~= ##Remove) {
        if (multi_wanted==100)  L__M(##Miscellany, 43);
        else                    L__M(##Miscellany, 44);
    }
}
if (etype == ASKSCOPE_PE) {
    scope_stage = 3;
    if (indirect(scope_error) == -1) {
        best_etype = nextbest_etype;
        if (~~((etype ofclass Routine) || (etype ofclass String)))
            EndActivity(PRINTING_A_PARSER_ERROR_ACT);
        jump GiveError;
    }
}
if (etype == NOTINCONTEXT_PE) L__M(##Miscellany, 73);
.SkipParserError;
if ((etype ofclass Routine) || (etype ofclass String)) jump ReType;
say__p = 1;
EndActivity(PRINTING_A_PARSER_ERROR_ACT);
```

§20. Parser Letter J. Retry the whole lot.

```
! And go (almost) right back to square one...
jump ReType;
! ...being careful not to go all the way back, to avoid infinite repetition
! of a deferred command causing an error.
```

§**21. Parser Letter K.** Last thing: check for THEN and further instructions(s), return.

```
    ! At this point, the return value is all prepared, and we are only looking
    ! to see if there is a "then" followed by subsequent instruction(s).
.LookForMore;
    if (wn > num_words) rtrue;
    i = NextWord();
    if (i == THEN1__WD or THEN2__WD or THEN3__WD or comma_word) {
        if (wn > num_words) {
        held_back_mode = false;
        return;
        }
        i = WordAddress(verb_wordnum);
        j = WordAddress(wn);
        for (: i<j : i++) i->0 = ' ';
        i = NextWord();
        if (i == AGAIN1__WD or AGAIN2__WD or AGAIN3__WD) {
            ! Delete the words "then again" from the again buffer,
            ! in which we have just realised that it must occur:
            ! prevents an infinite loop on "i. again"
            i = WordAddress(wn-2)-buffer;
            if (wn > num_words) j = INPUT_BUFFER_LEN-1;
            else j = WordAddress(wn)-buffer;
            for (: i<j : i++) buffer3->i = ' ';
        }
        VM_Tokenise(buffer,parse);
        held_back_mode = true;
        return;
    }
    best_etype = UPTO_PE;
    jump GiveError;
```

§**22. End of Parser Proper.**

```
]; ! end of Parser__parse
```

§**23. Parse Token.** The main parsing routine above tried a sequence of "grammar lines" in turn, matching each against the text typed until one fitted. A grammar line is itself a sequence of "grammar tokens". Here we have to parse the tokens.

`ParseToken(type, data)` tries the match text beginning at the current word marker `wn` against a token of the given `type`, with the given `data`. The optional further arguments `token_n` and `token` supply the token number in the current grammar line (because some tokens do depend on what has happened before or is needed later) and the address of the dictionary word which makes up the `token`, in the case where it's a "preposition".

The return values are:
(a) `GPR_REPARSE` for "I have rewritten the command, please re-parse from scratch";
(b) `GPR_PREPOSITION` for "token accepted with no result";
(c) $-256 + x$ for "please parse `ParseToken(ELEMENTARY_TT, x)` instead";
(d) 0 for "token accepted, result is the multiple object list";
(e) 1 for "token accepted, result is the number in `parsed_number`";
(f) an object number for "token accepted with this object as result";
(g) -1 for "token rejected".

Strictly speaking `ParseToken` is a shell routine which saves the current state on the stack, and calling `ParseToken__` to do the actual work.

Once again the routine is traditionally divided into six letters, here named under paragraphs "Parse Token Letter A", and so on.

(A) Analyse the token; handle all tokens not involving object lists and break down others into elementary tokens
(B) Begin parsing an object list
(C) Parse descriptors (articles, pronouns, etc.) in the list
(D) Parse an object name
(E) Parse connectives (AND, BUT, etc.) and go back to (C)
(F) Return the conclusion of parsing an object list

```
[ ParseTokenStopped x y;
    if (wn>WordCount()) return GPR_FAIL;
    return ParseToken(x,y);
];

Global parsetoken_nesting = 0;
[ ParseToken given_ttype given_tdata token_n token  i t rv;
    if (parsetoken_nesting > 0) {
        ! save match globals
        @push match_from; @push token_filter; @push match_length;
        @push number_of_classes; @push oops_from;
        for (i=0: i<number_matched: i++) {
            t = match_list-->i; @push t;
            t = match_classes-->i; @push t;
            t = match_scores-->i; @push t;
        }
        @push number_matched;
    }

    parsetoken_nesting++;
    rv = ParseToken__(given_ttype, given_tdata, token_n, token);
    parsetoken_nesting--;

    if (parsetoken_nesting > 0) {
        ! restore match globals
        @pull number_matched;
```

```
        for (i=0: i<number_matched: i++) {
            @pull t; match_scores-->i = t;
            @pull t; match_classes-->i = t;
            @pull t; match_list-->i = t;
        }
        @pull oops_from; @pull number_of_classes;
        @pull match_length; @pull token_filter; @pull match_from;
    }
    return rv;
];
[ ParseToken__ given_ttype given_tdata token_n token
    l o i j k and_parity single_object desc_wn many_flag
    token_allows_multiple prev_indef_wanted;
```

§**24. Parse Token Letter A.** Analyse token; handle all not involving object lists, break down others.

```
    token_filter = 0;
    parser_inflection = name;
    switch (given_ttype) {
    ELEMENTARY_TT:
        switch (given_tdata) {
        SPECIAL_TOKEN:
            l = TryNumber(wn);
            special_word = NextWord();
            #Ifdef DEBUG;
            if (l ~= -1000)
                if (parser_trace >= 3) print "  [Read special as the number ", l, "]^";
            #Endif; ! DEBUG
            if (l == -1000) {
                #Ifdef DEBUG;
                if (parser_trace >= 3) print "  [Read special word at word number ", wn, "]^";
                #Endif; ! DEBUG
                l = special_word;
            }
            parsed_number = l;
            return GPR_NUMBER;
        NUMBER_TOKEN:
            l=TryNumber(wn++);
            if (l == -1000) {
                etype = NUMBER_PE;
                return GPR_FAIL;
            }
            #Ifdef DEBUG;
            if (parser_trace>=3) print "  [Read number as ", l, "]^";
            #Endif; ! DEBUG
            parsed_number = l;
            return GPR_NUMBER;
        CREATURE_TOKEN:
            if (action_to_be == ##Answer or ##Ask or ##AskFor or ##Tell)
                scope_reason = TALKING_REASON;
        TOPIC_TOKEN:
```

```
        consult_from = wn;
        if ((line_ttype-->(token_n+1) ~= PREPOSITION_TT) &&
        (line_token-->(token_n+1) ~= ENDIT_TOKEN))
            RunTimeError(13);
        do o = NextWordStopped();
        until (o == -1 || PrepositionChain(o, token_n+1) ~= -1);
        wn--;
        consult_words = wn-consult_from;
        if (consult_words == 0) return GPR_FAIL;
        if (action_to_be == ##Ask or ##Answer or ##Tell) {
            o = wn; wn = consult_from; parsed_number = NextWord();
            wn = o; return 1;
        }
        if (o==-1 && (line_ttype-->(token_n+1) == PREPOSITION_TT))
            return GPR_FAIL;    ! don't infer if required preposition is absent
        return GPR_PREPOSITION;
    }
PREPOSITION_TT:
    ! Is it an unnecessary alternative preposition, when a previous choice
    ! has already been matched?
    if ((token->0) & $10) return GPR_PREPOSITION;
    ! If we've run out of the player's input, but still have parameters to
    ! specify, we go into "infer" mode, remembering where we are and the
    ! preposition we are inferring...
    if (wn > num_words) {
        if (inferfrom==0 && parameters<params_wanted) {
            inferfrom = pcount; inferword = token;
            pattern-->pcount = REPARSE_CODE + VM_DictionaryAddressToNumber(given_tdata);
        }
        ! If we are not inferring, then the line is wrong...
        if (inferfrom == 0) return -1;
        ! If not, then the line is right but we mark in the preposition...
        pattern-->pcount = REPARSE_CODE + VM_DictionaryAddressToNumber(given_tdata);
        return GPR_PREPOSITION;
    }
    o = NextWord();
    pattern-->pcount = REPARSE_CODE + VM_DictionaryAddressToNumber(o);
    ! Whereas, if the player has typed something here, see if it is the
    ! required preposition... if it's wrong, the line must be wrong,
    ! but if it's right, the token is passed (jump to finish this token).
    if (o == given_tdata) return GPR_PREPOSITION;
    if (PrepositionChain(o, token_n) ~= -1) return GPR_PREPOSITION;
    return -1;
GPR_TT:
    l = indirect(given_tdata);
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Outside parsing routine returned ", l, "]^";
    #Endif; ! DEBUG
    return l;
SCOPE_TT:
```

```
    scope_token = given_tdata;
    scope_stage = 1;
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Scope routine called at stage 1]^";
    #Endif; ! DEBUG
    l = indirect(scope_token);
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Scope routine returned multiple-flag of ", l, "]^";
    #Endif; ! DEBUG
    if (l == 1) given_tdata = MULTI_TOKEN; else given_tdata = NOUN_TOKEN;
ATTR_FILTER_TT:
    token_filter = 1 + given_tdata;
    given_tdata = NOUN_TOKEN;
ROUTINE_FILTER_TT:
    token_filter = given_tdata;
    given_tdata = NOUN_TOKEN;
} ! end of switch(given_ttype)
token = given_tdata;
```

§**25. Parse Token Letter B.** Begin parsing an object list.

```
! There are now three possible ways we can be here:
!     parsing an elementary token other than "special" or "number";
!     parsing a scope token;
!     parsing a noun-filter token (either by routine or attribute).
!
! In each case, token holds the type of elementary parse to
! perform in matching one or more objects, and
! token_filter is 0 (default), an attribute + 1 for an attribute filter
! or a routine address for a routine filter.
token_allows_multiple = false;
if (token == MULTI_TOKEN or MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN)
    token_allows_multiple = true;
many_flag = false; and_parity = true; dont_infer = false;
```

§**26. Parse Token Letter C.** Parse descriptors (articles, pronouns, etc.) in the list.

```
    ! We expect to find a list of objects next in what the player's typed.
.ObjectList;
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Object list from word ", wn, "]^";
    #Endif; ! DEBUG
    ! Take an advance look at the next word: if it's "it" or "them", and these
    ! are unset, set the appropriate error number and give up on the line
    ! (if not, these are still parsed in the usual way - it is not assumed
    ! that they still refer to something in scope)
    o = NextWord(); wn--;
    pronoun_word = NULL; pronoun_obj = NULL;
    l = PronounValue(o);
    if (l ~= 0) {
        pronoun_word = o; pronoun_obj = l;
        if (l == NULL) {
            ! Don't assume this is a use of an unset pronoun until the
            ! descriptors have been checked, because it might be an
            ! article (or some such) instead
            for (l=1 : l<=LanguageDescriptors-->0 : l=l+4)
                if (o == LanguageDescriptors-->l) jump AssumeDescriptor;
            pronoun__word = pronoun_word; pronoun__obj = pronoun_obj;
            etype = VAGUE_PE;
            if (parser_trace >= 3) print "  [Stop: unset pronoun]^";
            return GPR_FAIL;
        }
    }
.AssumeDescriptor;
    if (o == ME1__WD or ME2__WD or ME3__WD) { pronoun_word = o; pronoun_obj = player; }
    allow_plurals = true; desc_wn = wn;
.TryAgain;
    ! First, we parse any descriptive words (like "the", "five" or "every"):
    l = Descriptors(token_allows_multiple);
    if (l ~= 0) { etype = l; return 0; }
.TryAgain2;
```

§**27. Parse Token Letter D.** Parse an object name.

```
! This is an actual specified object, and is therefore where a typing error
! is most likely to occur, so we set:
oops_from = wn;
! So, two cases.  Case 1: token not equal to "held" (so, no implicit takes)
! but we may well be dealing with multiple objects
! In either case below we use NounDomain, giving it the token number as
! context, and two places to look: among the actor's possessions, and in the
! present location.  (Note that the order depends on which is likeliest.)
if (token ~= HELD_TOKEN) {
    i = multiple_object-->0;
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Calling NounDomain on location and actor]^";
    #Endif; ! DEBUG
    l = NounDomain(actors_location, actor, token);
    if (l == REPARSE_CODE) return l;                    ! Reparse after Q&A
    if (indef_wanted == INDEF_ALL_WANTED && l == 0 && number_matched == 0)
        l = 1;  ! ReviseMulti if TAKE ALL FROM empty container
    if (token_allows_multiple && ~~multiflag) {
        if (best_etype==MULTI_PE) best_etype=STUCK_PE;
        multiflag = true;
    }
    if (l == 0) {
        if (indef_possambig) {
            ResetDescriptors();
            wn = desc_wn;
            jump TryAgain2;
        }
        if (etype == MULTI_PE or TOOFEW_PE && multiflag) etype = STUCK_PE;
        etype=CantSee();
        jump FailToken;
    } ! Choose best error
    #Ifdef DEBUG;
    if (parser_trace >= 3) {
        if (l > 1) print "  [ND returned ", (the) l, "]^";
        else {
            print "  [ND appended to the multiple object list:^";
            k = multiple_object-->0;
            for (j=i+1 : j<=k : j++)
                print "  Entry ", j, ": ", (The) multiple_object-->j,
                    " (", multiple_object-->j, ")^";
            print "  List now has size ", k, "]^";
        }
    }
    #Endif; ! DEBUG
    if (l == 1) {
        if (~~many_flag) many_flag = true;
        else {                                  ! Merge with earlier ones
            k = multiple_object-->0;              ! (with either parity)
            multiple_object-->0 = i;
            for (j=i+1 : j<=k : j++) {
```

```
                if (and_parity) MultiAdd(multiple_object-->j);
                else            MultiSub(multiple_object-->j);
            }
            #Ifdef DEBUG;
            if (parser_trace >= 3)
                print "  [Merging ", k-i, " new objects to the ", i, " old ones]^";
            #Endif; ! DEBUG
        }
    }
    else {
        ! A single object was indeed found
        if (match_length == 0 && indef_possambig) {
            ! So the answer had to be inferred from no textual data,
            ! and we know that there was an ambiguity in the descriptor
            ! stage (such as a word which could be a pronoun being
            ! parsed as an article or possessive).  It's worth having
            ! another go.
            ResetDescriptors();
            wn = desc_wn;
            jump TryAgain2;
        }
        if ((token == CREATURE_TOKEN) && (CreatureTest(l) == 0)) {
            etype = ANIMA_PE;
            jump FailToken;
        } !  Animation is required
        if (~~many_flag) single_object = l;
        else {
            if (and_parity) MultiAdd(l); else MultiSub(l);
            #Ifdef DEBUG;
            if (parser_trace >= 3) print "  [Combining ", (the) l, " with list]^";
            #Endif; ! DEBUG
        }
    }
}
else {
! Case 2: token is "held" (which fortunately can't take multiple objects)
! and may generate an implicit take
    l = NounDomain(actor,actors_location,token);       ! Same as above...
    if (l == REPARSE_CODE) return l;
    if (l == 0) {
        if (indef_possambig) {
            ResetDescriptors();
            wn = desc_wn;
            jump TryAgain2;
        }
        etype = CantSee(); jump FailToken;            ! Choose best error
    }
    ! ...until it produces something not held by the actor.  Then an implicit
    ! take must be tried.  If this is already happening anyway, things are too
    ! confused and we have to give up (but saving the oops marker so as to get
    ! it on the right word afterwards).
```

```
        ! The point of this last rule is that a sequence like
        !
        !     > read newspaper
        !     (taking the newspaper first)
        !     The dwarf unexpectedly prevents you from taking the newspaper!
        !
        ! should not be allowed to go into an infinite repeat - read becomes
        ! take then read, but take has no effect, so read becomes take then read...
        ! Anyway for now all we do is record the number of the object to take.
        o = parent(l);
        if (o ~= actor) {
            #Ifdef DEBUG;
            if (parser_trace >= 3) print "  [Allowing object ", (the) l, " for now]^";
            #Endif; ! DEBUG
        }
        single_object = l;
    } ! end of if (token ~= HELD_TOKEN) else
    ! The following moves the word marker to just past the named object...
    wn = oops_from + match_length;
```

§**28. Parse Token Letter E.** Parse connectives (AND, BUT, etc.) and go back to (C).

```
    ! Object(s) specified now: is that the end of the list, or have we reached
    ! "and", "but" and so on?  If so, create a multiple-object list if we
    ! haven't already (and are allowed to).
.NextInList;
    o = NextWord();
    if (o == AND1__WD or AND2__WD or AND3__WD or BUT1__WD or BUT2__WD or BUT3__WD or comma_word) {
        #Ifdef DEBUG;
        if (parser_trace >= 3) print "  [Read connective '", (address) o, "']^";
        #Endif; ! DEBUG
        if (~~token_allows_multiple) {
            if (multiflag) jump PassToken; ! give UPTO_PE error
            etype=MULTI_PE;
            jump FailToken;
        }
        if (o == BUT1__WD or BUT2__WD or BUT3__WD) and_parity = 1-and_parity;
        if (~~many_flag) {
            multiple_object-->0 = 1;
            multiple_object-->1 = single_object;
            many_flag = true;
            #Ifdef DEBUG;
            if (parser_trace >= 3) print "  [Making new list from ", (the) single_object, "]^";
            #Endif; ! DEBUG
        }
        dont_infer = true; inferfrom=0;           ! Don't print (inferences)
        jump ObjectList;                          ! And back around
    }
    wn--;   ! Word marker back to first not-understood word
```

§**29. Parse Token Letter F.** Return the conclusion of parsing an object list.

```
    ! Happy or unhappy endings:
.PassToken;
    if (many_flag) {
        single_object = GPR_MULTIPLE;
        multi_context = token;
    }
    else {
        if (indef_mode == 1 && indef_type & PLURAL_BIT ~= 0) {
            if (indef_wanted < INDEF_ALL_WANTED && indef_wanted > 1) {
                multi_had = 1; multi_wanted = indef_wanted;
                etype = TOOFEW_PE;
                jump FailToken;
            }
        }
    }
    return single_object;
.FailToken;
    ! If we were only guessing about it being a plural, try again but only
    ! allowing singulars (so that words like "six" are not swallowed up as
    ! Descriptors)
    if (allow_plurals && indef_guess_p == 1) {
        #Ifdef DEBUG;
        if (parser_trace >= 4) print "   [Retrying singulars after failure ", etype, "]^";
        #Endif;
        prev_indef_wanted = indef_wanted;
        allow_plurals = false;
        wn = desc_wn;
        jump TryAgain;
    }
    if ((indef_wanted > 0 || prev_indef_wanted > 0) && (~~multiflag)) etype = MULTI_PE;
    return GPR_FAIL;
]; ! end of ParseToken__
```

§**30. Descriptors.** In grammatical terms, a descriptor appears at the front of an English noun phrase and clarifies the quantity or specific identity of what is referred to: for instance, *my* mirror, *the* dwarf, *that* woman. (Numbers, as in *four* duets, are also descriptors in linguistics: but the I6 parser doesn't handle them that way.)

Slightly unfortunately, the bitmap constants used for descriptors in the I6 parser have names in the form `*_BIT`, coinciding with the names of style bits in the list-writer: but they never occur in the same context.

The actual words used as descriptors are read from tables in the language definition. `ArticleDescriptors` uses this table to move current word marker past a run of one or more descriptors which refer to the definite or indefinite article.

```
Constant OTHER_BIT  =   1;     !  These will be used in Adjudicate()
Constant MY_BIT     =   2;     !  to disambiguate choices
Constant THAT_BIT   =   4;
Constant PLURAL_BIT =   8;
Constant LIT_BIT    =  16;
Constant UNLIT_BIT  =  32;

[ ResetDescriptors;
    indef_mode = 0; indef_type = 0; indef_wanted = 0; indef_guess_p = 0;
    indef_possambig = false;
    indef_owner = nothing;
    indef_cases = $$111111111111;
    indef_nspec_at = 0;
];

[ ArticleDescriptors  o x flag cto type n;
    if (wn > num_words) return 0;
    for (flag=true : flag :) {
        o = NextWordStopped(); flag = false;
    for (x=1 : x<=LanguageDescriptors-->0 : x=x+4)
            if (o == LanguageDescriptors-->x) {
                type = LanguageDescriptors-->(x+2);
                if (type == DEFART_PK or INDEFART_PK) flag = true;
            }
    }
    wn--;
    return 0;
];
```

§**31. Parsing Descriptors.** The `Descriptors()` routine parses the descriptors at the head of a noun phrase, leaving the current word marker `wn` at the first word of the noun phrase's body. It is allowed to set up for a plural only if `allow_p` is set; it returns a parser error number, or 0 if no error occurred.

```
[ Descriptors  o x flag cto type n;
    ResetDescriptors();
    if (wn > num_words) return 0;
    for (flag=true : flag :) {
        o = NextWordStopped(); flag = false;
    for (x=1 : x<=LanguageDescriptors-->0 : x=x+4)
            if (o == LanguageDescriptors-->x) {
                flag = true;
                type = LanguageDescriptors-->(x+2);
                if (type ~= DEFART_PK) indef_mode = true;
                indef_possambig = true;
                indef_cases = indef_cases & (LanguageDescriptors-->(x+1));
                if (type == POSSESS_PK) {
                    cto = LanguageDescriptors-->(x+3);
                    switch (cto) {
                    0: indef_type = indef_type | MY_BIT;
                    1: indef_type = indef_type | THAT_BIT;
                    default:
                        indef_owner = PronounValue(cto);
                        if (indef_owner == NULL) indef_owner = InformParser;
                    }
                }
                if (type == light)  indef_type = indef_type | LIT_BIT;
                if (type == -light) indef_type = indef_type | UNLIT_BIT;
            }
        if (o == OTHER1__WD or OTHER2__WD or OTHER3__WD) {
            indef_mode = 1; flag = 1;
            indef_type = indef_type | OTHER_BIT;
        }
        if (o == ALL1__WD or ALL2__WD or ALL3__WD or ALL4__WD or ALL5__WD) {
            indef_mode = 1; flag = 1; indef_wanted = INDEF_ALL_WANTED;
            if (take_all_rule == 1) take_all_rule = 2;
            indef_type = indef_type | PLURAL_BIT;
        }
        if (allow_plurals) {
            if (NextWordStopped() ~= -1) { wn--; n = TryNumber(wn-1); } else { n=0; wn--; }
            if (n == 1) { indef_mode = 1; flag = 1; }
            if (n > 1) {
                indef_guess_p = 1;
                indef_mode = 1; flag = 1; indef_wanted = n;
                indef_nspec_at = wn-1;
                indef_type = indef_type | PLURAL_BIT;
            }
        }
        if (flag == 1 && NextWordStopped() ~= OF1__WD or OF2__WD or OF3__WD or OF4__WD)
            wn--;  ! Skip 'of' after these
    }
    wn--;
```

```
    return 0;
];
[ SafeSkipDescriptors;
    @push indef_mode; @push indef_type; @push indef_wanted;
    @push indef_guess_p; @push indef_possambig; @push indef_owner;
    @push indef_cases; @push indef_nspec_at;
    Descriptors();
    @pull indef_nspec_at; @pull indef_cases;
    @pull indef_owner; @pull indef_possambig; @pull indef_guess_p;
    @pull indef_wanted; @pull indef_type; @pull indef_mode;
];
```

§**32. Preposition Chain.** A small utility for runs of prepositions.

```
[ PrepositionChain wd index;
    if (line_tdata-->index == wd) return wd;
    if ((line_token-->index)->0 & $20 == 0) return -1;
    do {
        if (line_tdata-->index == wd) return wd;
        index++;
    } until ((line_token-->index == ENDIT_TOKEN) || (((line_token-->index)->0 & $10) == 0));
    return -1;
];
```

§**33. Creature.** Will this object do for an I6 `creature` token? (In I7 terms, this affects the tokens "[someone]", "[somebody]", "[anyone]" and "[anybody]".)

```
[ CreatureTest obj;
    if (obj has animate) rtrue;
    if (obj hasnt talkable) rfalse;
    if (action_to_be == ##Ask or ##Answer or ##Tell or ##AskFor) rtrue;
    rfalse;
];
```

§**34. Noun Domain.** `NounDomain` does the most substantial part of parsing an object name. It is given two "domains" – usually a location and then the actor who is looking – and a context (i.e. token type), and returns:

(a) 0 if no match at all could be made,
(b) 1 if a multiple object was made,
(c) k if object k was the one decided upon,
(d) `REPARSE_CODE` if it asked a question of the player and consequently rewrote the player's input, so that the whole parser should start again on the rewritten input.

In case (c), `NounDomain` also sets the variable `length_of_noun` to the number of words in the input text matched to the noun. In case (b), the multiple objects are added to `multiple_object` by hand (not by `MultiAdd`, because we want to allow duplicates).

```
[ NounDomain domain1 domain2 context
    first_word i j k l answer_words marker;
    #Ifdef DEBUG;
    if (parser_trace >= 4) {
        print "   [NounDomain called at word ", wn, "^";
        print "   ";
        if (indef_mode) {
            print "seeking indefinite object: ";
            if (indef_type & OTHER_BIT)  print "other ";
            if (indef_type & MY_BIT)     print "my ";
            if (indef_type & THAT_BIT)   print "that ";
            if (indef_type & PLURAL_BIT) print "plural ";
            if (indef_type & LIT_BIT)    print "lit ";
            if (indef_type & UNLIT_BIT)  print "unlit ";
            if (indef_owner ~= 0) print "owner:", (name) indef_owner;
            new_line;
            print "   number wanted: ";
            if (indef_wanted == INDEF_ALL_WANTED) print "all"; else print indef_wanted;
            new_line;
            print "   most likely GNAs of names: ", indef_cases, "^";
        }
        else print "seeking definite object^";
    }
    #Endif; ! DEBUG

    match_length = 0; number_matched = 0; match_from = wn;

    SearchScope(domain1, domain2, context);

    #Ifdef DEBUG;
    if (parser_trace >= 4) print "   [ND made ", number_matched, " matches]^";
    #Endif; ! DEBUG

    wn = match_from+match_length;

    ! If nothing worked at all, leave with the word marker skipped past the
    ! first unmatched word...

    if (number_matched == 0) { wn++; rfalse; }

    ! Suppose that there really were some words being parsed (i.e., we did
    ! not just infer).  If so, and if there was only one match, it must be
    ! right and we return it...

    if (match_from <= num_words) {
        if (number_matched == 1) {
            i=match_list-->0;
```

```
        return i;
    }
    ! ...now suppose that there was more typing to come, i.e. suppose that
    ! the user entered something beyond this noun.  If nothing ought to follow,
    ! then there must be a mistake, (unless what does follow is just a full
    ! stop, and or comma)
    if (wn <= num_words) {
        i = NextWord(); wn--;
        if (i ~=  AND1__WD or AND2__WD or AND3__WD or comma_word
            or THEN1__WD or THEN2__WD or THEN3__WD
            or BUT1__WD or BUT2__WD or BUT3__WD) {
            if (lookahead == ENDIT_TOKEN) rfalse;
        }
    }
}
! Now look for a good choice, if there's more than one choice...
number_of_classes = 0;
if (number_matched == 1) i = match_list-->0;
if (number_matched > 1) {
    i = true;
    if (number_matched > 1)
        for (j=0 : j<number_matched-1 : j++)
            if (Identical(match_list-->j, match_list-->(j+1)) == false)
                i = false;
    if (i) dont_infer = true;
    i = Adjudicate(context);
    if (i == -1) rfalse;
    if (i == 1) rtrue;       !  Adjudicate has made a multiple
                         !  object, and we pass it on
}
! If i is non-zero here, one of two things is happening: either
! (a) an inference has been successfully made that object i is
!     the intended one from the user's specification, or
! (b) the user finished typing some time ago, but we've decided
!     on i because it's the only possible choice.
! In either case we have to keep the pattern up to date,
! note that an inference has been made and return.
! (Except, we don't note which of a pile of identical objects.)
if (i ~= 0) {
    if (dont_infer) return i;
    if (inferfrom == 0) inferfrom=pcount;
    pattern-->pcount = i;
    return i;
}
! If we get here, there was no obvious choice of object to make.  If in
! fact we've already gone past the end of the player's typing (which
! means the match list must contain every object in scope, regardless
! of its name), then it's foolish to give an enormous list to choose
! from - instead we go and ask a more suitable question...
if (match_from > num_words) jump Incomplete;
! Now we print up the question, using the equivalence classes as worked
```

```
    ! out by Adjudicate() so as not to repeat ourselves on plural objects...
    BeginActivity(ASKING_WHICH_DO_YOU_MEAN_ACT);
    if (ForActivity(ASKING_WHICH_DO_YOU_MEAN_ACT)) jump SkipWhichQuestion;
    j = 1; marker = 0;
    for (i=1 : i<=number_of_classes : i++) {
        while (((match_classes-->marker) ~= i) && ((match_classes-->marker) ~= -i))
            marker++;
        if (match_list-->marker hasnt animate) j = 0;
    }
    if (j) L__M(##Miscellany, 45); else L__M(##Miscellany, 46);
    j = number_of_classes; marker = 0;
    for (i=1 : i<=number_of_classes : i++) {
        while (((match_classes-->marker) ~= i) && ((match_classes-->marker) ~= -i)) marker++;
        k = match_list-->marker;
        if (match_classes-->marker > 0) print (the) k; else print (a) k;
        if (i < j-1)  print (string) COMMA__TX;
        if (i == j-1) {
            #Ifdef SERIAL_COMMA;
            if (j ~= 2) print ",";
            #Endif; ! SERIAL_COMMA
            print (string) OR__TX;
        }
    }
    L__M(##Miscellany, 57);
    .SkipWhichQuestion; EndActivity(ASKING_WHICH_DO_YOU_MEAN_ACT);
    ! ...and get an answer:
.WhichOne;
    #Ifdef TARGET_ZCODE;
    for (i=2 : i<INPUT_BUFFER_LEN : i++) buffer2->i = ' ';
    #Endif; ! TARGET_ZCODE
    answer_words=Keyboard(buffer2, parse2);
    ! Conveniently, parse2-->1 is the first word in both ZCODE and GLULX.
    first_word = (parse2-->1);
    ! Take care of "all", because that does something too clever here to do
    ! later on:
    if (first_word == ALL1__WD or ALL2__WD or ALL3__WD or ALL4__WD or ALL5__WD) {
        if (context == MULTI_TOKEN or MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN) {
            l = multiple_object-->0;
            for (i=0 : i<number_matched && l+i<MATCH_LIST_WORDS : i++) {
                k = match_list-->i;
                multiple_object-->(i+1+l) = k;
            }
            multiple_object-->0 = i+l;
            rtrue;
        }
        L__M(##Miscellany, 47);
        jump WhichOne;
    }
    ! Look for a comma, and interpret this as a fresh conversation command
    ! if so:
```

```
for (i=1 : i<=answer_words : i++)
    if (WordFrom(i, parse2) == comma_word) {
        VM_CopyBuffer(buffer, buffer2);
        jump RECONSTRUCT_INPUT;
    }
! If the first word of the reply can be interpreted as a verb, then
! assume that the player has ignored the question and given a new
! command altogether.
! (This is one time when it's convenient that the directions are
! not themselves verbs - thus, "north" as a reply to "Which, the north
! or south door" is not treated as a fresh command but as an answer.)
#Ifdef LanguageIsVerb;
if (first_word == 0) {
    j = wn; first_word = LanguageIsVerb(buffer2, parse2, 1); wn = j;
}
#Endif; ! LanguageIsVerb
if (first_word ~= 0) {
    j = first_word->#dict_par1;
    if ((0 ~= j&1) && ~~LanguageVerbMayBeName(first_word)) {
        VM_CopyBuffer(buffer, buffer2);
        jump RECONSTRUCT_INPUT;
    }
}
! Now we insert the answer into the original typed command, as
! words additionally describing the same object
! (eg, > take red button
!      Which one, ...
!      > music
! becomes "take music red button".  The parser will thus have three
! words to work from next time, not two.)
#Ifdef TARGET_ZCODE;
k = WordAddress(match_from) - buffer; l=buffer2->1+1;
for (j=buffer + buffer->0 - 1 : j>=buffer+k+l : j--) j->0 = 0->(j-l);
for (i=0 : i<l : i++) buffer->(k+i) = buffer2->(2+i);
buffer->(k+l-1) = ' ';
buffer->1 = buffer->1 + l;
if (buffer->1 >= (buffer->0 - 1)) buffer->1 = buffer->0;
#Ifnot; ! TARGET_GLULX
k = WordAddress(match_from) - buffer;
l = (buffer2-->0) + 1;
for (j=buffer+INPUT_BUFFER_LEN-1 : j>=buffer+k+l : j--) j->0 = j->(-l);
for (i=0 : i<l : i++) buffer->(k+i) = buffer2->(WORDSIZE+i);
buffer->(k+l-1) = ' ';
buffer-->0 = buffer-->0 + l;
if (buffer-->0 > (INPUT_BUFFER_LEN-WORDSIZE)) buffer-->0 = (INPUT_BUFFER_LEN-WORDSIZE);
#Endif; ! TARGET_
! Having reconstructed the input, we warn the parser accordingly
! and get out.
.RECONSTRUCT_INPUT;
num_words = WordCount();
wn = 1;
#Ifdef LanguageToInformese;
```

```
    LanguageToInformese();
    ! Re-tokenise:
    VM_Tokenise(buffer,parse);
    #Endif; ! LanguageToInformese
    num_words = WordCount();
    players_command = 100 + WordCount();
    actors_location = ScopeCeiling(player);
    FollowRulebook(Activity_after_rulebooks-->READING_A_COMMAND_ACT, true);
    return REPARSE_CODE;
    ! Now we come to the question asked when the input has run out
    ! and can't easily be guessed (eg, the player typed "take" and there
    ! were plenty of things which might have been meant).
.Incomplete;
    if (context == CREATURE_TOKEN) L__M(##Miscellany, 48);
    else                           L__M(##Miscellany, 49);
    #Ifdef TARGET_ZCODE;
    for (i=2 : i<INPUT_BUFFER_LEN : i++) buffer2->i=' ';
    #Endif; ! TARGET_ZCODE
    answer_words = Keyboard(buffer2, parse2);
    first_word=(parse2-->1);
    #Ifdef LanguageIsVerb;
    if (first_word==0) {
        j = wn; first_word=LanguageIsVerb(buffer2, parse2, 1); wn = j;
    }
    #Endif; ! LanguageIsVerb
    ! Once again, if the reply looks like a command, give it to the
    ! parser to get on with and forget about the question...
    if (first_word ~= 0) {
        j = first_word->#dict_par1;
        if (0 ~= j&1) {
            VM_CopyBuffer(buffer, buffer2);
            return REPARSE_CODE;
        }
    }
    ! ...but if we have a genuine answer, then:
    !
    ! (1) we must glue in text suitable for anything that's been inferred.
    if (inferfrom ~= 0) {
        for (j=inferfrom : j<pcount : j++) {
            if (pattern-->j == PATTERN_NULL) continue;
            #Ifdef TARGET_ZCODE;
            i = 2+buffer->1; (buffer->1)++; buffer->(i++) = ' ';
            #Ifnot; ! TARGET_GLULX
            i = WORDSIZE + buffer-->0;
            (buffer-->0)++; buffer->(i++) = ' ';
            #Endif; ! TARGET_
            #Ifdef DEBUG;
            if (parser_trace >= 5)
                print "[Gluing in inference with pattern code ", pattern-->j, "]^";
            #Endif; ! DEBUG
```

```
        ! Conveniently, parse2-->1 is the first word in both ZCODE and GLULX.
        parse2-->1 = 0;
        ! An inferred object.  Best we can do is glue in a pronoun.
        ! (This is imperfect, but it's very seldom needed anyway.)
        if (pattern-->j >= 2 && pattern-->j < REPARSE_CODE) {
            PronounNotice(pattern-->j);
            for (k=1 : k<=LanguagePronouns-->0 : k=k+3)
                if (pattern-->j == LanguagePronouns-->(k+2)) {
                    parse2-->1 = LanguagePronouns-->k;
                    #Ifdef DEBUG;
                    if (parser_trace >= 5)
                        print "[Using pronoun '", (address) parse2-->1, "']^";
                    #Endif; ! DEBUG
                    break;
                }
        }
        else {
            ! An inferred preposition.
            parse2-->1 = VM_NumberToDictionaryAddress(pattern-->j - REPARSE_CODE);
            #Ifdef DEBUG;
            if (parser_trace >= 5)
                print "[Using preposition '", (address) parse2-->1, "']^";
            #Endif; ! DEBUG
        }
        ! parse2-->1 now holds the dictionary address of the word to glue in.
        if (parse2-->1 ~= 0) {
            k = buffer + i;
            #Ifdef TARGET_ZCODE;
            @output_stream 3 k;
            print (address) parse2-->1;
            @output_stream -3;
            k = k-->0;
            for (l=i : l<i+k : l++) buffer->l = buffer->(l+2);
            i = i + k; buffer->1 = i-2;
            #Ifnot; ! TARGET_GLULX
            k = Glulx_PrintAnyToArray(buffer+i, INPUT_BUFFER_LEN-i, parse2-->1);
            i = i + k; buffer-->0 = i - WORDSIZE;
            #Endif; ! TARGET_
        }
    }
}
! (2) we must glue the newly-typed text onto the end.
#Ifdef TARGET_ZCODE;
i = 2+buffer->1; (buffer->1)++; buffer->(i++) = ' ';
for (j=0 : j<buffer2->1 : i++,j++) {
    buffer->i = buffer2->(j+2);
    (buffer->1)++;
    if (buffer->1 == INPUT_BUFFER_LEN) break;
}
#Ifnot; ! TARGET_GLULX
i = WORDSIZE + buffer-->0;
(buffer-->0)++; buffer->(i++) = ' ';
```

```
    for (j=0 : j<buffer2-->0 : i++,j++) {
        buffer->i = buffer2->(j+WORDSIZE);
        (buffer-->0)++;
        if (buffer-->0 == INPUT_BUFFER_LEN) break;
    }
    #Endif; ! TARGET_

    ! (3) we fill up the buffer with spaces, which is unnecessary, but may
    !     help incorrectly-written interpreters to cope.

    #Ifdef TARGET_ZCODE;
    for (: i<INPUT_BUFFER_LEN : i++) buffer->i = ' ';
    #Endif; ! TARGET_ZCODE

    return REPARSE_CODE;
]; ! end of NounDomain
```

§**35. Adjudicate.** The `Adjudicate` routine tries to see if there is an obvious choice, when faced with a list of objects (the `match_list`) each of which matches the player's specification equally well. To do this it makes use of the `context` (the token type being worked on).

It counts up the number of obvious choices for the given context – all to do with where a candidate is, except for 6 (`animate`) which is to do with whether it is animate or not – and then:

(a) if only one obvious choice is found, that is returned;
(b) if we are in indefinite mode (don't care which) one of the obvious choices is returned, or if there is no obvious choice then an unobvious one is made;
(c) at this stage, we work out whether the objects are distinguishable from each other or not: if they are all indistinguishable from each other, then choose one, it doesn't matter which;
(d) otherwise, 0 (meaning, unable to decide) is returned (but remember that the equivalence classes we've just worked out will be needed by other routines to clear up this mess, so we can't economise on working them out).

`Adjudicate` returns -1 if an error occurred.

```
[ Adjudicate context i j k good_ones last n ultimate flag offset;
    #Ifdef DEBUG;
    if (parser_trace >= 4) {
        print "   [Adjudicating match list of size ", number_matched,
            " in context ", context, "^";
        print "   ";
        if (indef_mode) {
            print "indefinite type: ";
            if (indef_type & OTHER_BIT)  print "other ";
            if (indef_type & MY_BIT)     print "my ";
            if (indef_type & THAT_BIT)   print "that ";
            if (indef_type & PLURAL_BIT) print "plural ";
            if (indef_type & LIT_BIT)    print "lit ";
            if (indef_type & UNLIT_BIT)  print "unlit ";
            if (indef_owner ~= 0) print "owner:", (name) indef_owner;
            new_line;
            print "   number wanted: ";
            if (indef_wanted == INDEF_ALL_WANTED) print "all"; else print indef_wanted;
            new_line;
            print "   most likely GNAs of names: ", indef_cases, "^";
        }
        else print "definite object^";
```

```
}
#Endif; ! DEBUG
j = number_matched-1; good_ones = 0; last = match_list-->0;
for (i=0 : i<=j : i++) {
    n = match_list-->i;
    match_scores-->i = good_ones;
    ultimate = ScopeCeiling(n);
    if (context==HELD_TOKEN && parent(n)==actor)
    {   good_ones++; last=n; }
    if (context==MULTI_TOKEN && ultimate==ScopeCeiling(actor)
        && n~=actor && n hasnt concealed && n hasnt scenery)
    {   good_ones++; last=n; }
    if (context==MULTIHELD_TOKEN && parent(n)==actor)
    {   good_ones++; last=n; }
    if (context==MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN)
    {   if (advance_warning==-1)
        {   if (context==MULTIEXCEPT_TOKEN)
            {   good_ones++; last=n;
            }
            if (context==MULTIINSIDE_TOKEN)
            {   if (parent(n)~=actor) { good_ones++; last=n; }
            }
        }
        else
        {   if (context==MULTIEXCEPT_TOKEN && n~=advance_warning)
            {   good_ones++; last=n; }
            if (context==MULTIINSIDE_TOKEN && n in advance_warning)
            {   good_ones++; last=n; }
        }
    }
    if (context==CREATURE_TOKEN && CreatureTest(n)==1)
    {   good_ones++; last=n; }
    match_scores-->i = 1000*(good_ones - match_scores-->i);
}
if (good_ones == 1) return last;
! If there is ambiguity about what was typed, but it definitely wasn't
! animate as required, then return anything; higher up in the parser
! a suitable error will be given.  (This prevents a question being asked.)
if (context == CREATURE_TOKEN && good_ones == 0) return match_list-->0;
if (indef_mode == 0) indef_type=0;
ScoreMatchL(context);
if (number_matched == 0) return -1;
if (indef_mode == 0) {
    !  Is there now a single highest-scoring object?
    i = SingleBestGuess();
    if (i >= 0) {
        #Ifdef DEBUG;
        if (parser_trace >= 4) print "   Single best-scoring object returned.]^";
        #Endif; ! DEBUG
        return i;
    }
```

```
}
if (indef_mode == 1 && indef_type & PLURAL_BIT ~= 0) {
    if (context ~= MULTI_TOKEN or MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN
                or MULTIINSIDE_TOKEN) {
        etype = MULTI_PE;
        return -1;
    }
    i = 0; offset = multiple_object-->0;
    for (j=BestGuess(): j~=-1 && i<indef_wanted && i+offset<MATCH_LIST_WORDS-1:
        j=BestGuess()) {
        flag = 0;
        BeginActivity(DECIDING_WHETHER_ALL_INC_ACT, j);
        if ((ForActivity(DECIDING_WHETHER_ALL_INC_ACT, j)) == 0) {
            if (j hasnt concealed && j hasnt worn) flag = 1;
            if (context == MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN && parent(j) ~= actor)
                flag = 0;
            if (action_to_be == ##Take or ##Remove && parent(j) == actor)
                flag = 0;
            k = ChooseObjects(j, flag);
            if (k == 1)
                flag = 1;
            else {
                if (k == 2) flag = 0;
            }
        } else {
            flag = 0; if (RulebookSucceeded()) flag = 1;
        }
        EndActivity(DECIDING_WHETHER_ALL_INC_ACT, j);
        if (flag == 1) {
            i++; multiple_object-->(i+offset) = j;
            #Ifdef DEBUG;
            if (parser_trace >= 4) print "   Accepting it^";
            #Endif; ! DEBUG
        }
        else {
            i = i;
            #Ifdef DEBUG;
            if (parser_trace >= 4) print "   Rejecting it^";
            #Endif; ! DEBUG
        }
    }
    if (i < indef_wanted && indef_wanted < INDEF_ALL_WANTED) {
        etype = TOOFEW_PE; multi_wanted = indef_wanted;
        multi_had=i;
        return -1;
    }
    multiple_object-->0 = i+offset;
    multi_context = context;
    #Ifdef DEBUG;
    if (parser_trace >= 4)
        print "   Made multiple object of size ", i, "]^";
    #Endif; ! DEBUG
```

```
    return 1;
}
for (i=0 : i<number_matched : i++) match_classes-->i = 0;
n = 1;
for (i=0 : i<number_matched : i++)
    if (match_classes-->i == 0) {
        match_classes-->i = n++; flag = 0;
        for (j=i+1 : j<number_matched : j++)
            if (match_classes-->j == 0 && Identical(match_list-->i, match_list-->j) == 1) {
                flag=1;
                match_classes-->j = match_classes-->i;
            }
        if (flag == 1) match_classes-->i = 1-n;
    }
n--; number_of_classes = n;
#Ifdef DEBUG;
if (parser_trace >= 4) {
    print "   Grouped into ", n, " possibilities by name:^";
    for (i=0 : i<number_matched : i++)
        if (match_classes-->i > 0)
            print "   ", (The) match_list-->i, " (", match_list-->i, ")  ---  group ",
            match_classes-->i, "^";
}
#Endif; ! DEBUG
if (indef_mode == 0) {
    if (n > 1) {
        k = -1;
        for (i=0 : i<number_matched : i++) {
            if (match_scores-->i > k) {
                k = match_scores-->i;
                j = match_classes-->i; j = j*j;
                flag = 0;
            }
            else
                if (match_scores-->i == k) {
                    if ((match_classes-->i) * (match_classes-->i) ~= j)
                        flag = 1;
                }
        }
    if (flag) {
        #Ifdef DEBUG;
        if (parser_trace >= 4) print "   Unable to choose best group, so ask player.]^";
        #Endif; ! DEBUG
        return 0;
    }
    #Ifdef DEBUG;
    if (parser_trace >= 4) print "   Best choices are all from the same group.^";
    #Endif; ! DEBUG
    }
}
!  When the player is really vague, or there's a single collection of
!  indistinguishable objects to choose from, choose the one the player
```

```
    !  most recently acquired, or if the player has none of them, then
    !  the one most recently put where it is.
    if (n == 1) dont_infer = true;
    return BestGuess();
]; ! Adjudicate
```

§**36. ReviseMulti.** `ReviseMulti` revises the multiple object which already exists, in the light of information which has come along since then (i.e., the second parameter). It returns a parser error number, or else 0 if all is well. This only ever throws things out, never adds new ones.

```
[ ReviseMulti second_p  i low;
    #Ifdef DEBUG;
    if (parser_trace >= 4)
        print "   Revising multiple object list of size ", multiple_object-->0,
        " with 2nd ", (name) second_p, "^";
    #Endif; ! DEBUG
    if (multi_context == MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN) {
        for (i=1,low=0 : i<=multiple_object-->0 : i++) {
            if ( (multi_context==MULTIEXCEPT_TOKEN && multiple_object-->i ~= second_p) ||
                 (multi_context==MULTIINSIDE_TOKEN && multiple_object-->i in second_p)) {
                low++;
                multiple_object-->low = multiple_object-->i;
            }
        }
        multiple_object-->0 = low;
    }
    if (multi_context == MULTI_TOKEN && action_to_be == ##Take) {
        #Ifdef DEBUG;
        if (parser_trace >= 4) print "   Token 2 plural case: number with actor ", low, "^";
        #Endif; ! DEBUG
        if (take_all_rule == 2) {
            for (i=1,low=0 : i<=multiple_object-->0 : i++) {
                if (ScopeCeiling(multiple_object-->i) == ScopeCeiling(actor)) {
                    low++;
                    multiple_object-->low = multiple_object-->i;
                }
            }
            multiple_object-->0 = low;
        }
    }
    i = multiple_object-->0;
    #Ifdef DEBUG;
    if (parser_trace >= 4) print "   Done: new size ", i, "^";
    #Endif; ! DEBUG
    if (i == 0) return NOTHING_PE;
    return 0;
];
```

§**37. Match List.** The match list is an array, `match_list-->`, which holds the current best guesses at what object(s) a portion of the command refers to. The global `number_matched` is set to the current length of the `match_list`.

When the parser sees a possible match of object `obj` at quality level `q`, it calls `MakeMatch(obj, q)`. If this is the best quality match so far, then we wipe out all the previous matches and start a new list with this one. If it's only as good as the best so far, we add it to the list (provided we haven't run out of space, and provided it isn't in the list already). If it's worse, we ignore it altogether.

I6 tokens in the form `noun=Filter` or `Attribute` are "noun filter tokens", and mean that the match list should be filtered to accept only nouns which are acceptable to the given routine, or have the given attribute. Such a token is in force if `token_filter` is used. (I7 makes no use of this in the attribute case, which is deprecated nowadays.)

Quality is essentially the number of words in the command referring to the object: the idea is that "red panic button" is better than "red button" or "panic".

```
[ MakeMatch obj quality i;
    #Ifdef DEBUG;
    if (parser_trace >= 6) print "    Match with quality ",quality,"^";
    #Endif; ! DEBUG
    if (token_filter ~= 0 && ConsultNounFilterToken(obj) == 0) {
        #Ifdef DEBUG;
        if (parser_trace >= 6) print "    Match filtered out: token filter ", token_filter, "^";
        #Endif; ! DEBUG
        rtrue;
    }
    if (quality < match_length) rtrue;
    if (quality > match_length) { match_length = quality; number_matched = 0; }
    else {
        if (number_matched >= MATCH_LIST_WORDS) rtrue;
        for (i=0 : i<number_matched : i++)
            if (match_list-->i == obj) rtrue;
    }
    match_list-->number_matched++ = obj;
    #Ifdef DEBUG;
    if (parser_trace >= 6) print "    Match added to list^";
    #Endif; ! DEBUG
];
[ ConsultNounFilterToken obj;
    if (token_filter ofclass Routine) {
        noun = obj;
        return indirect(token_filter);
    }
    if (obj has (token_filter-1)) rtrue;
    rfalse;
];
```

§38. ScoreMatchL. `ScoreMatchL` scores the match list for quality in terms of what the player has vaguely asked for. Points are awarded for conforming with requirements like "my", and so on. Remove from the match list any entries which fail the basic requirements of the descriptors. (The scoring system used to evaluate the possibilities is discussed in detail in the DM4.)

```
Constant SCORE__CHOOSEOBJ = 1000;
Constant SCORE__IFGOOD = 500;
Constant SCORE__UNCONCEALED = 100;
Constant SCORE__BESTLOC = 60;
Constant SCORE__NEXTBESTLOC = 40;
Constant SCORE__NOTCOMPASS = 20;
Constant SCORE__NOTSCENERY = 10;
Constant SCORE__NOTACTOR = 5;
Constant SCORE__GNA = 1;
Constant SCORE__DIVISOR = 20;

Constant PREFER_HELD;
[ ScoreMatchL context its_owner its_score obj i j threshold met a_s l_s;
!   if (indef_type & OTHER_BIT ~= 0) threshold++;
    if (indef_type & MY_BIT ~= 0)    threshold++;
    if (indef_type & THAT_BIT ~= 0)  threshold++;
    if (indef_type & LIT_BIT ~= 0)   threshold++;
    if (indef_type & UNLIT_BIT ~= 0) threshold++;
    if (indef_owner ~= nothing)      threshold++;

    #Ifdef DEBUG;
    if (parser_trace >= 4) print "   Scoring match list: indef mode ", indef_mode, " type ",
    indef_type, ", satisfying ", threshold, " requirements:^";
    #Endif; ! DEBUG

    #ifdef PREFER_HELD;
    a_s = SCORE__BESTLOC; l_s = SCORE__NEXTBESTLOC;
    if (action_to_be == ##Take or ##Remove) {
        a_s = SCORE__NEXTBESTLOC; l_s = SCORE__BESTLOC;
    }
    context = context;  ! silence warning
    #ifnot;
    a_s = SCORE__NEXTBESTLOC; l_s = SCORE__BESTLOC;
    if (context == HELD_TOKEN or MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN) {
        a_s = SCORE__BESTLOC; l_s = SCORE__NEXTBESTLOC;
    }
    #endif; ! PREFER_HELD

    for (i=0 : i<number_matched : i++) {
        obj = match_list-->i; its_owner = parent(obj); its_score=0; met=0;
        !      if (indef_type & OTHER_BIT ~= 0
        !          &&  obj ~= itobj or himobj or herobj) met++;
        if (indef_type & MY_BIT ~= 0 && its_owner == actor) met++;
        if (indef_type & THAT_BIT ~= 0 && its_owner == actors_location) met++;
        if (indef_type & LIT_BIT ~= 0 && obj has light) met++;
        if (indef_type & UNLIT_BIT ~= 0 && obj hasnt light) met++;
        if (indef_owner ~= 0 && its_owner == indef_owner) met++;

        if (met < threshold) {
            #Ifdef DEBUG;
            if (parser_trace >= 4)
                print "   ", (The) match_list-->i, " (", match_list-->i, ") in ",
```

```
                (the) its_owner, " is rejected (doesn't match descriptors)^";
            #Endif; ! DEBUG
            match_list-->i = -1;
        }
        else {
            its_score = 0;
            if (obj hasnt concealed) its_score = SCORE__UNCONCEALED;
            if (its_owner == actor) its_score = its_score + a_s;
            else
                if (its_owner == actors_location) its_score = its_score + l_s;
                else
                    if (its_owner ~= compass) its_score = its_score + SCORE__NOTCOMPASS;
            its_score = its_score + SCORE__CHOOSEOBJ * ChooseObjects(obj, 2);
            if (obj hasnt scenery) its_score = its_score + SCORE__NOTSCENERY;
            if (obj ~= actor) its_score = its_score + SCORE__NOTACTOR;
            !   A small bonus for having the correct GNA,
            !   for sorting out ambiguous articles and the like.
            if (indef_cases & (PowersOfTwo_TB-->(GetGNAOfObject(obj))))
                its_score = its_score + SCORE__GNA;
            match_scores-->i = match_scores-->i + its_score;
            #Ifdef DEBUG;
            if (parser_trace >= 4) print "     ", (The) match_list-->i, " (", match_list-->i,
            ") in ", (the) its_owner, " : ", match_scores-->i, " points^";
            #Endif; ! DEBUG
        }
    }
    for (i=0 : i<number_matched : i++) {
        while (match_list-->i == -1) {
            if (i == number_matched-1) { number_matched--; break; }
            for (j=i : j<number_matched-1 : j++) {
                match_list-->j = match_list-->(j+1);
                match_scores-->j = match_scores-->(j+1);
            }
            number_matched--;
        }
    }
];
```

§**39. BestGuess.** `BestGuess` makes the best guess it can out of the match list, assuming that everything in the match list is textually as good as everything else; however it ignores items marked as −1, and so marks anything it chooses. It returns −1 if there are no possible choices.

```
[ BestGuess  earliest its_score best i;
    earliest = 0; best = -1;
    for (i=0 : i<number_matched : i++) {
        if (match_list-->i >= 0) {
            its_score = match_scores-->i;
            if (its_score > best) { best = its_score; earliest = i; }
        }
    }
    #Ifdef DEBUG;
    if (parser_trace >= 4)
    if (best < 0) print "   Best guess ran out of choices^";
    else print "   Best guess ", (the) match_list-->earliest,
        " (", match_list-->earliest, ")^";
    #Endif; ! DEBUG
    if (best < 0) return -1;
    i = match_list-->earliest;
    match_list-->earliest = -1;
    return i;
];
```

§**40. SingleBestGuess.** `SingleBestGuess` returns the highest-scoring object in the match list if it is the clear winner, or returns −1 if there is no clear winner.

```
[ SingleBestGuess  earliest its_score best i;
    earliest = -1; best = -1000;
    for (i=0 : i<number_matched : i++) {
        its_score = match_scores-->i;
        if (its_score == best) earliest = -1;
        if (its_score > best) { best = its_score; earliest = match_list-->i; }
    }
    return earliest;
];
```

§**41. Identical.** `Identical` decides whether or not two objects can be distinguished from each other by anything the player can type. If not, it returns `true`. (This routine is critical to the handling of plurals, and the list-writer requires it to be an equivalence relation between objects: but it is, because it is equivalent to $O_1 \sim O_2$ if and only if $f(O_1) = f(O_2)$ for some function f.)

```
[ Identical o1 o2 p1 p2 n1 n2 i j flag;
    if (o1 == o2) rtrue;  ! This should never happen, but to be on the safe side
    if (o1 == 0 || o2 == 0) rfalse;  ! Similarly
    if (o1 ofclass K3_direction || o2 ofclass K3_direction) rfalse; ! Saves time
    !  What complicates things is that o1 or o2 might have a parsing routine,
    !  so the parser can't know from here whether they are or aren't the same.
    !  If they have different parsing routines, we simply assume they're
    !  different.  If they have the same routine (which they probably got from
    !  a class definition) then the decision process is as follows:
    !
    !     the routine is called (with self being o1, not that it matters)
    !       with noun and second being set to o1 and o2, and action being set
    !       to the fake action TheSame.  If it returns -1, they are found
    !       identical; if -2, different; and if >=0, then the usual method
    !       is used instead.
    if (o1.parse_name ~= 0 || o2.parse_name ~= 0) {
    if (o1.parse_name ~= o2.parse_name) rfalse;
    parser_action = ##TheSame; parser_one = o1; parser_two = o2;
    j = wn; i = RunRoutines(o1,parse_name); wn = j;
    if (i == -1) rtrue;
    if (i == -2) rfalse;
    }
    !  This is the default algorithm: do they have the same words in their
    !  "name" (i.e. property no. 1) properties.  (Note that the following allows
    !  for repeated words and words in different orders.)
    p1 = o1.&1; n1 = (o1.#1)/WORDSIZE;
    p2 = o2.&1; n2 = (o2.#1)/WORDSIZE;
    !  for (i=0 : i<n1 : i++) { print (address) p1-->i, " "; } new_line;
    !  for (i=0 : i<n2 : i++) { print (address) p2-->i, " "; } new_line;
    for (i=0 : i<n1 : i++) {
        flag = 0;
        for (j=0 : j<n2 : j++)
            if (p1-->i == p2-->j) flag = 1;
        if (flag == 0) rfalse;
    }
    for (j=0 : j<n2 : j++) {
        flag = 0;
        for (i=0 : i<n1 : i++)
            if (p1-->i == p2-->j) flag = 1;
        if (flag == 0) rfalse;
    }
    !  print "Which are identical!^";
    rtrue;
];
```

§**42. Print Command.** `PrintCommand` reconstructs the command as it presently reads, from the pattern which has been built up.

If `from` is 0, it starts with the verb: then it goes through the pattern.

The other parameter is `emptyf` – a flag: if 0, it goes up to `pcount`: if 1, it goes up to `pcount`-1.

Note that verbs and prepositions are printed out of the dictionary: and that since the dictionary may only preserve the first six characters of a word (in a V3 game), we have to hand-code the longer words needed. At present, I7 doesn't do this, but it probably should.

(Recall that pattern entries are 0 for "multiple object", 1 for "special word", 2 to `REPARSE_CODE-1` are object numbers and `REPARSE_CODE+n` means the preposition `n`.)

```
[ PrintInferredCommand from singleton_noun;
    singleton_noun = FALSE;
    if ((from ~= 0) && (from == pcount-1) &&
        (pattern-->from > 1) && (pattern-->from < REPARSE_CODE))
            singleton_noun = TRUE;
    if (singleton_noun) {
        BeginActivity(CLARIFYING_PARSERS_CHOICE_ACT, pattern-->from);
        if (ForActivity(CLARIFYING_PARSERS_CHOICE_ACT, pattern-->from) == 0) {
            print "("; PrintCommand(from); print ")^";
        }
        EndActivity(CLARIFYING_PARSERS_CHOICE_ACT, pattern-->from);
    } else {
        print "("; PrintCommand(from); print ")^";
    }
];
[ PrintCommand from i k spacing_flag;
    if (from == 0) {
        i = verb_word;
        if (LanguageVerb(i) == 0)
            if (PrintVerb(i) == 0) print (address) i;
        from++; spacing_flag = true;
    }
    for (k=from : k<pcount : k++) {
        i = pattern-->k;
        if (i == PATTERN_NULL) continue;
        if (spacing_flag) print (char) ' ';
        if (i == 0) { print (string) THOSET__TX; jump TokenPrinted; }
        if (i == 1) { print (string) THAT__TX;   jump TokenPrinted; }
        if (i >= REPARSE_CODE)
            print (address) VM_NumberToDictionaryAddress(i-REPARSE_CODE);
        else
            if (i ofclass K3_direction)
                print (LanguageDirection) i; ! the direction name as adverb
            else
                print (the) i;
    .TokenPrinted;
        spacing_flag = true;
    }
];
```

§**43. CantSee.** The `CantSee` routine returns a good error number for the situation where the last word looked at didn't seem to refer to any object in context.

The idea is that: if the actor is in a location (but not inside something like, for instance, a tank which is in that location) then an attempt to refer to one of the words listed as meaningful-but-irrelevant there will cause "you don't need to refer to that in this game" rather than "no such thing" or "what's 'it'?".

(The advantage of not having looked at "irrelevant" local nouns until now is that it stops them from clogging up the ambiguity-resolving process. Thus game objects always triumph over scenery.)

```
[ CantSee  i w e;
    saved_oops=oops_from;
    if (scope_token ~= 0) {
        scope_error = scope_token; return ASKSCOPE_PE;
    }
    wn--; w = NextWord();
    e = CANTSEE_PE;
    if (w == pronoun_word) {
        w = NextWordStopped(); wn--;
        if ((w == -1) || (line_token-->(pcount) ~= ENDIT_TOKEN)) {
            AnalyseToken(line_token-->(pcount-1));
            !DebugToken(pcount-1); print " ", found_ttype, "^";
            if (found_ttype == ROUTINE_FILTER_TT or ATTR_FILTER_TT)
                e = NOTINCONTEXT_PE;
            else {
                pronoun__word = pronoun_word; pronoun__obj = pronoun_obj;
                e = ITGONE_PE;
            }
        }
    }
    if (etype > e) return etype;
    return e;
];
```

§**44. Multiple Object List.** The `MultiAdd` routine adds object `o` to the multiple-object-list. This is only allowed to hold `MATCH_LIST_WORDS` minus one objects at most, at which point it ignores any new entries (and sets a global flag so that a warning may later be printed if need be).

The `MultiSub` routine deletes object `o` from the multiple-object-list. It returns 0 if the object was there in the first place, and 9 (because this is the appropriate error number in `Parser()`) if it wasn't.

The `MultiFilter` routine goes through the multiple-object-list and throws out anything without the given attribute `attr` set.

```
[ MultiAdd o i j;
    i = multiple_object-->0;
    if (i == MATCH_LIST_WORDS-1) { toomany_flag = 1; rtrue; }
    for (j=1 : j<=i : j++)
        if (o == multiple_object-->j) rtrue;
    i++;
    multiple_object-->i = o;
    multiple_object-->0 = i;
];
[ MultiSub o i j k;
    i = multiple_object-->0;
```

```
    for (j=1 : j<=i : j++)
        if (o == multiple_object-->j) {
            for (k=j : k<=i : k++) multiple_object-->k = multiple_object-->(k+1);
            multiple_object-->0 = --i;
            return 0;
        }
    return VAGUE_PE;
];
[ MultiFilter attr  i j o;
    .MFiltl;
    i = multiple_object-->0;
    for (j=1 : j<=i : j++) {
        o = multiple_object-->j;
        if (o hasnt attr) { MultiSub(o); jump Mfiltl; }
    }
];
```

§**45. Scope.** The scope of an actor is the set of objects which he can refer to in typed commands, which is normally the same as the set of visible objects; but this can be modified. This is how I7 handles tokens like "[any room]".

Scope determination is done by calling `SearchScope` to iterate through the objects in scope, and "visit" each one: which means, carry out some task for each as we get there. The task depends on the current value of `scope_reason`, which is `PARSING_REASON` when the parser is matching command text against object names.

The scope machinery is built on a number of levels, each making use only of lower levels:

(0) Either `NounDomain`, `TestScope` or `LoopOverScope` makes one or more calls to `SearchScope` (on level 1). The point of making multiple calls is to influence the order in which items in scope are visited, which improves the quality of "take all"-style multiple object lists, for instance.

(1) `SearchScope` searches for the objects in scope which are within first one domain, and then another: for instance, first within the current room but not within the current actor, and then within the current actor. It can be called either from level 0, or externally from the choose-objects machinery, but is not recursive. It works within the context of a given token in the parser (when called for `PARSING_REASON`) and in particular the `multiinside` token, and also handles testing commands, scope tokens, scope in darkness, and intervention by the I7 "deciding the scope of" activity. Most of its actual searches are delegated to `ScopeWithin` (level 2), but it also uses `DoScopeActionAndRecurse` (level 3) and `DoScopeAction` (level 4) as necessary.

(2) `ScopeWithin` iterates through the objects in scope which are within one supplied domain, but not within another. It can be called either from level 1, or independently from rules in the "deciding the scope of" activity via the I7 "place the contents of X in scope" phrase. It calls `DoScopeActionAndRecurse` (level 3) on any unconcealed objects it finds.

(3) `DoScopeActionAndRecurse` visits a given object by calling down to `DoScopeAction` (level 4), and recurses to all unconcealed object-tree contents and component parts of the object. The I7 phrase "place X in scope" uses this routine.

(4) `DoScopeAction` simply visits a single object, taking whatever action is needed there – which will depend on the `scope_reason`. The only use made by the parser of `TryGivenObject`, which tries to match command text against the name of a given object, is from here. The I7 phrase "place X in scope, but not its contents" uses this routine.

Two routines are provided for code external to the parser to modify the scope. They should be called only during scope deliberations – i.e., in `scope=...` tokens or in rules for the "deciding the scope of" activity. (At present, `AddToScope` is not used in I7 at all.) Note that this I7 form of `PlaceInScope` has a slightly different specification to its I6 library counterpart of the same name: it can place a room in scope. (In I6, room names were not normally parsed.)

```
[ PlaceInScope O opts; ! If opts is set, do not place contents in scope
    wn = match_from;
    if (opts == false) DoScopeActionAndRecurse(O);
    else DoScopeAction(O);
    return;
];
[ AddToScope obj;
    if (ats_flag >= 2) DoScopeActionAndRecurse(obj, 0, ats_flag-2);
    if (ats_flag == 1) { if (HasLightSource(obj)==1) ats_hls = 1; }
];
```

§**46. Scope Level 0.** The two ways of starting up the scope machinery other than via the parser code above.

```
[ TestScope obj act a al sr x y;
    x = parser_one; y = parser_two;
    parser_one = obj; parser_two = 0; a = actor; al = actors_location;
    sr = scope_reason; scope_reason = TESTSCOPE_REASON;
    if (act == 0) actor = player; else actor = act;
    actors_location = ScopeCeiling(actor);
    SearchScope(actors_location, actor, 0); scope_reason = sr; actor = a;
    actors_location = al; parser_one = x; x = parser_two; parser_two = y;
    return x;
];
[ LoopOverScope routine act x y a al;
    x = parser_one; y = scope_reason; a = actor; al = actors_location;
    parser_one = routine;
    if (act == 0) actor = player; else actor = act;
    actors_location = ScopeCeiling(actor);
    scope_reason = LOOPOVERSCOPE_REASON;
    SearchScope(actors_location, actor, 0);
    parser_one = x; scope_reason = y; actor = a; actors_location = al;
];
```

§**47. SearchScope.** Level 1. The method is:

(a) If the context is a `scope=...` token, then the search is delegated to "stage 2" of the scope routine. This was the old I6 way to override the searching behaviour: while users probably won't be using it any more, the template does, in order to give testing commands universal scope which is exempt from the activity below; and the NI compiler creates `scope=...` tokens to handle Understand grammar such as "[any room]". So the feature remains very much still in use.

(b) The "deciding the scope of" activity is given the chance to intervene. This is the I7 way to override the searching behaviour, and is the one taken by users.

(c) And otherwise:

(1) The I6 `multiinside` token, used as the first noun of its grammar line, has as its scope all of the objects which are inside or on top of the *second* noun of the grammar line. This provides a neat scope for the ALL in a command like GET ALL FROM CUPBOARD, where the player clearly does not intend ALL to refer to the cupboard itself, for instance. The difficulty is that we don't yet know what the second object is, if we are parsing left to right. But the parser code above has taken care of all of that, and the `advance_warning` global is set to the object number of the second noun, or to -1 if that is not yet known. Note that we check that the contents are visible before adding them to scope, because otherwise an unscrupulous player could use such a command to detect the contents of an opaque locked box. If this rule applies, we skip (c.2), (c.3) and (c.4).

(2) For all other tokens except `creature`, searching scope for the room holding the current actor always catches the compass directions unless a definite article has already been typed. (Thus OPEN THE EAST would match an object called "east door", but not the compass direction "east".)
(3) The contents of `domain1` which are not contents of `domain2` are placed in scope, and so are any component parts of `domain1`. If `domain1` is a container or supporter, it is placed in scope itself.
(4) The contents and component parts of `domain2` are placed in scope. If `domain2` is a container or supporter, it is placed in scope itself.
(5) In darkness, the actor and his component parts are in scope. If the actor is inside or on top of something, then that thing is also in scope. (This avoids a situation where the player gets into an opaque box, then pulls it closed from the inside, plunging himself into darkness, then types OPEN BOX only to be told that he can't see any such thing.)

```
[ SearchScope domain1 domain2 context i;
    if (domain1 == 0) return;
    ! (a)
    if (scope_token) {
        scope_stage = 2;
        #Ifdef DEBUG;
        if (parser_trace >= 3) print "  [Scope routine called at stage 2]^";
        #Endif;
        if (indirect(scope_token) ~= 0) rtrue;
    }
    ! (b)
    BeginActivity(DECIDING_SCOPE_ACT, actor);
    if (ForActivity(DECIDING_SCOPE_ACT, actor) == false) {
        ! (c.1)
        if ((scope_reason == PARSING_REASON) && (context == MULTIINSIDE_TOKEN) &&
            (advance_warning ~= -1)) {
            if (IsSeeThrough(advance_warning) == 1)
                ScopeWithin(advance_warning, 0, context);
        } else {
            ! (c.2)
            if ((scope_reason == PARSING_REASON) && (context ~= CREATURE_TOKEN) &&
                (indef_mode == 0) && (domain1 == actors_location))
                    ScopeWithin(compass);
            ! (c.3)
            if (domain1 has supporter or container) DoScopeAction(domain1);
            ScopeWithin(domain1, domain2, context);
            ! (c.4)
            if (domain2) {
                if (domain2 has supporter or container) DoScopeAction(domain2);
                ScopeWithin(domain2, 0, context);
            }
        }
        ! (c.5)
        if (thedark == domain1 or domain2) {
            DoScopeActionAndRecurse(actor, actor, context);
            if (parent(actor) has supporter or container)
                DoScopeActionAndRecurse(parent(actor), parent(actor), context);
        }
    }
    EndActivity(DECIDING_SCOPE_ACT, actor);
];
```

§**48. ScopeWithin.** Level 2. `ScopeWithin` puts objects visible from within the `domain` into scope. An item belonging to the `domain` is placed in scope unless it is being concealed by the `domain`: and even then, if the `domain` is the current actor. Suppose Zorro conceals a book beneath his cloak: then the book is not in scope to his lady friend The Black Whip, but it is in scope to Zorro himself. (Thus an actor is not allowed to conceal anything from himself.)

Note that the `domain` object itself, and its component parts if any, are not placed in scope by this routine, though nothing prevents some other code doing so.

```
[ ScopeWithin domain nosearch context obj next_obj;
    if (domain == 0) rtrue;
    ! Look through the objects in the domain, avoiding "objectloop" in case
    ! movements occur.
    obj = child(domain);
    while (obj) {
        next_obj = sibling(obj);
        if ((domain == actor) || (TestConcealment(domain, obj) == false))
            DoScopeActionAndRecurse(obj, nosearch, context);
        obj = next_obj;
    }
];
```

§**49. DoScopeActionAndRecurse.** Level 3. In all cases, the `domain` itself is visited. There are then three possible forms of recursion:

(a) To unconcealed objects which are inside, on top of, carried or worn by the `domain`: this is called "searching" in traditional I6 language and is suppressed if `domain` is the special value `nosearch`.
(b) To unconcealed component parts of the `domain`.
(c) To any other objects listed in the `add_to_scope` property array, or supplied by the `add_to_scope` property routine, if it has one. (I7 does not usually use `add_to_scope`, but it remains a useful hook in the parser, so it retains its old I6 library interpretation.)

```
[ DoScopeActionAndRecurse domain nosearch context i ad n obj next_obj;
    DoScopeAction(domain);
    ! (a)
    if ((domain ~= nosearch) &&
        ((domain ofclass K1_room or K8_person) || (IsSeeThrough(domain) == 1))) {
        obj = child(domain);
        while (obj) {
            next_obj = sibling(obj);
            if ((domain == actor) || (TestConcealment(domain, obj) == false))
                DoScopeActionAndRecurse(obj, nosearch, context);
            obj = next_obj;
        }
    }
    ! (b)
    if (domain provides component_child) {
        obj = domain.component_child;
        while (obj) {
            next_obj = obj.component_sibling;
            if ((domain == actor) || (TestConcealment(domain, obj) == false))
                DoScopeActionAndRecurse(obj, 0, context);
            obj = next_obj;
        }
```

```
    }
    ! (c)
    ad = domain.&add_to_scope;
    if (ad ~= 0) {
        ! Test if the property value is not an object.
        #Ifdef TARGET_ZCODE;
        i = (UnsignedCompare(ad-->0, top_object) > 0);
        #Ifnot; ! TARGET_GLULX
        i = (((ad-->0)->0) ~= $70);
        #Endif; ! TARGET_
        if (i) {
            ats_flag = 2+context;
            RunRoutines(domain, add_to_scope);
            ats_flag = 0;
        }
        else {
            n = domain.#add_to_scope;
            for (i=0 : (WORDSIZE*i)<n : i++)
                if (ad-->i)
                    DoScopeActionAndRecurse(ad-->i, 0, context);
        }
    }
];
```

§**50. DoScopeAction.** Level 4. This is where we take whatever action is to be performed as the "visit" to each scoped object, and it's the bottom at last of the scope mechanism.

```
[ DoScopeAction item;
    #Ifdef DEBUG;
    if (parser_trace >= 6)
        print "[DSA on ", (the) item, " with reason = ", scope_reason,
            " p1 = ", parser_one, " p2 = ", parser_two, "]^";
    #Endif; ! DEBUG
    @push parser_one; @push scope_reason;
    switch(scope_reason) {
        TESTSCOPE_REASON: if (item == parser_one) parser_two = 1;
        LOOPOVERSCOPE_REASON: if (parser_one ofclass Routine) indirect(parser_one, item);
        PARSING_REASON, TALKING_REASON: MatchTextAgainstObject(item);
    }
    @pull scope_reason; @pull parser_one;
];
```

§**51. Parsing Object Names.** We now reach the final major block of code in the parser: the part which tries to match a given object's name(s) against the text at word position `match_from` in the player's command, and calls `MakeMatch` if it succeeds. There are basically four possibilities: ME, a pronoun such as IT, a name which doesn't begin misleadingly with a number, and a name which does. In the latter two cases, we pass the job down to `TryGivenObject`.

```
[ MatchTextAgainstObject item i;
    if (match_from <= num_words) { ! If there's any text to match, that is
        wn = match_from;
        i = NounWord();
        if ((i == 1) && (player == item)) MakeMatch(item, 1); ! "me"
        if ((i >= 2) && (i < 128) && (LanguagePronouns-->i == item)) MakeMatch(item, 1);
    }

    ! Construing the current word as the start of a noun, can it refer to the
    ! object?

    wn = match_from;
    if (TryGivenObject(item) > 0)
        if (indef_nspec_at > 0 && match_from ~= indef_nspec_at) {
            ! This case arises if the player has typed a number in
            ! which is hypothetically an indefinite descriptor:
            ! e.g. "take two clubs".  We have just checked the object
            ! against the word "clubs", in the hope of eventually finding
            ! two such objects.  But we also backtrack and check it
            ! against the words "two clubs", in case it turns out to
            ! be the 2 of Clubs from a pack of cards, say.  If it does
            ! match against "two clubs", we tear up our original
            ! assumption about the meaning of "two" and lapse back into
            ! definite mode.

            wn = indef_nspec_at;
            if (TryGivenObject(item) > 0) {
                match_from = indef_nspec_at;
                ResetDescriptors();
            }
            wn = match_from;
        }
];
```

§**52. TryGivenObject.** `TryGivenObject` tries to match as many words as possible in what has been typed to the given object, `obj`. If it manages any words matched at all, it calls `MakeMatch` to say so, then returns the number of words (or 1 if it was a match because of inadequate input).

```
[ TryGivenObject obj nomatch threshold k w j;
    #Ifdef DEBUG;
    if (parser_trace >= 5) print "    Trying ", (the) obj, " (", obj, ") at word ", wn, "^";
    #Endif; ! DEBUG
    if (nomatch && obj == 0) return 0;
! if (nomatch) print "*** TryGivenObject *** on ", (the) obj, " at wn = ", wn, "^";
    dict_flags_of_noun = 0;
!  If input has run out then always match, with only quality 0 (this saves
!  time).
    if (wn > num_words) {
        if (nomatch) return 0;
        if (indef_mode ~= 0)
            dict_flags_of_noun = $$01110000;  ! Reject "plural" bit
        MakeMatch(obj,0);
        #Ifdef DEBUG;
        if (parser_trace >= 5) print "    Matched (0)^";
        #Endif; ! DEBUG
        return 1;
    }
!  Ask the object to parse itself if necessary, sitting up and taking notice
!  if it says the plural was used:
    if (obj.parse_name~=0) {
        parser_action = NULL; j=wn;
        k = RunRoutines(obj,parse_name);
        if (k > 0) {
            wn=j+k;
        .MMbyPN;
            if (parser_action == ##PluralFound)
                dict_flags_of_noun = dict_flags_of_noun | 4;
            if (dict_flags_of_noun & 4) {
                if (~~allow_plurals) k = 0;
                else {
                    if (indef_mode == 0) {
                        indef_mode = 1; indef_type = 0; indef_wanted = 0;
                    }
                    indef_type = indef_type | PLURAL_BIT;
                    if (indef_wanted == 0) indef_wanted = INDEF_ALL_WANTED;
                }
            }
            #Ifdef DEBUG;
            if (parser_trace >= 5) print "    Matched (", k, ")^";
            #Endif; ! DEBUG
            if (nomatch == false) MakeMatch(obj,k);
            return k;
        }
        if (k == 0) jump NoWordsMatch;
    }
```

```
    ! The default algorithm is simply to count up how many words pass the
    ! Refers test:
    parser_action = NULL;
    w = NounWord();
    if (w == 1 && player == obj) { k=1; jump MMbyPN; }
    if (w >= 2 && w < 128 && (LanguagePronouns-->w == obj)) { k = 1; jump MMbyPN; }
    if (Refers(obj, wn-1) == 0) {
        .NoWordsMatch;
        if (indef_mode ~= 0) { k = 0; parser_action = NULL; jump MMbyPN; }
        rfalse;
    }
    threshold = 1;
    dict_flags_of_noun = (w->#dict_par1) & $$01110100;
    w = NextWord();
    while (Refers(obj, wn-1)) {
        threshold++;
        if (w)
        dict_flags_of_noun = dict_flags_of_noun | ((w->#dict_par1) & $$01110100);
        w = NextWord();
    }
    k = threshold;
    jump MMbyPN;
];
```

§**53. Refers.** `Refers` works out whether the word at number wnum can refer to the object `obj`, returning true or false. The standard method is to see if the word is listed under the `name` property for the object, but this is more complex in languages other than English.

```
[ Refers obj wnum   wd k l m;
    if (obj == 0) rfalse;
    #Ifdef LanguageRefers;
    k = LanguageRefers(obj,wnum); if (k >= 0) return k;
    #Endif; ! LanguageRefers
    k = wn; wn = wnum; wd = NextWordStopped(); wn = k;
    if (parser_inflection >= 256) {
        k = indirect(parser_inflection, obj, wd);
        if (k >= 0) return k;
        m = -k;
    }
    else
        m = parser_inflection;
    k = obj.&m; l = (obj.#m)/WORDSIZE-1;
    for (m=0 : m<=l : m++)
        if (wd == k-->m) rtrue;
    rfalse;
];
[ WordInProperty wd obj prop k l m;
    k = obj.&prop; l = (obj.#prop)/WORDSIZE-1;
    for (m=0 : m<=l : m++)
        if (wd == k-->m) rtrue;
```

```
    rfalse;
];
```

§**54. NounWord.** `NounWord` (which takes no arguments) returns:
(a) 0 if the next word is not in the dictionary or is but does not carry the "noun" bit in its dictionary entry,
(b) 1 if it is a word meaning "me",
(c) the index in the pronoun table (plus 2) of the value field of a pronoun, if it is a pronoun,
(d) the address in the dictionary if it is a recognised noun.

```
[ NounWord i j s;
    i = NextWord();
    if (i == 0) rfalse;
    if (i == ME1__WD or ME2__WD or ME3__WD) return 1;
    s = LanguagePronouns-->0;
    for (j=1 : j<=s : j=j+3)
        if (i == LanguagePronouns-->j)
            return j+2;
    if ((i->#dict_par1)&128 == 0) rfalse;
    return i;
];
```

§**55. TryNumber.** `TryNumber` takes word number `wordnum` and tries to parse it as an (unsigned) decimal number or the name of a small number, returning

(a) −1000 if it is not a number
(b) the number, if it has between 1 and 4 digits
(c) 10000 if it has 5 or more digits.

(The danger of allowing 5 digits is that Z-machine integers are only 16 bits long, and anyway this routine isn't meant to be perfect: it only really needs to be good enough to handle numeric descriptors such as those in TAKE 31 COINS or DROP FOUR DAGGERS. In particular, it is not the way I7 "[number]" tokens are parsed.)

```
[ TryNumber wordnum   i j c num len mul tot d digit;
    i = wn; wn = wordnum; j = NextWord(); wn = i;
    j = NumberWord(j); ! Test for verbal forms ONE to TWENTY
    if (j >= 1) return j;
    #Ifdef TARGET_ZCODE;
    i = wordnum*4+1; j = parse->i; num = j+buffer; len = parse->(i-1);
    #Ifnot; ! TARGET_GLULX
    i = wordnum*3; j = parse-->i; num = j+buffer; len = parse-->(i-1);
    #Endif; ! TARGET_
    if (len >= 4) mul=1000;
    if (len == 3) mul=100;
    if (len == 2) mul=10;
    if (len == 1) mul=1;
    tot = 0; c = 0; len = len-1;
    for (c=0 : c<=len : c++) {
        digit=num->c;
        if (digit == '0') { d = 0; jump digok; }
        if (digit == '1') { d = 1; jump digok; }
        if (digit == '2') { d = 2; jump digok; }
        if (digit == '3') { d = 3; jump digok; }
```

```
        if (digit == '4') { d = 4; jump digok; }
        if (digit == '5') { d = 5; jump digok; }
        if (digit == '6') { d = 6; jump digok; }
        if (digit == '7') { d = 7; jump digok; }
        if (digit == '8') { d = 8; jump digok; }
        if (digit == '9') { d = 9; jump digok; }
        return -1000;
    .digok;
        tot = tot+mul*d; mul = mul/10;
    }
    if (len > 3) tot=10000;
    return tot;
];
```

§**56. Extended TryNumber.** The same, but recognising verbal forms up to 30.

```
[ I7_ExtendedTryNumber wordnum i j;
    i = wn; wn = wordnum; j = NextWordStopped(); wn = i;
    switch (j) {
        'twenty-one': return 21;
        'twenty-two': return 22;
        'twenty-three': return 23;
        'twenty-four': return 24;
        'twenty-five': return 25;
        'twenty-six': return 26;
        'twenty-seven': return 27;
        'twenty-eight': return 28;
        'twenty-nine': return 29;
        'thirty': return 30;
        default: return TryNumber(wordnum);
    }
];
```

§**57. Gender.** `GetGender` returns 0 if the given animate object is female, and 1 if male, and is abstracted as a routine in case something more elaborate is ever needed.

For GNAs – gender/noun/animation combinations – see the *Inform Designer's Manual*, 4th edition.

```
[ GetGender person;
    if (person hasnt female) rtrue;
    rfalse;
];
[ GetGNAOfObject obj case gender;
    if (obj hasnt animate) case = 6;
    if (obj has male) gender = male;
    if (obj has female) gender = female;
    if (obj has neuter) gender = neuter;
    if (gender == 0) {
        if (case == 0) gender = LanguageAnimateGender;
        else gender = LanguageInanimateGender;
    }
    if (gender == female)   case = case + 1;
    if (gender == neuter)   case = case + 2;
```

```
    if (obj has pluralname) case = case + 3;
    return case;
];
```

§58. Noticing Plurals.

```
[ DetectPluralWord at n i w swn outcome;
    swn = wn; wn = at;
    for (i=0:i<n:i++) {
        w = NextWordStopped();
        if (w == 0 or THEN1__WD or COMMA_WORD or -1) break;
        if ((w->#dict_par1) & $$00000100) {
            parser_action = ##PluralFound;
            outcome = true;
        }
    }
    wn = swn;
    return outcome;
];
```

§59. Pronoun Handling.

```
[ SetPronoun dword value x;
    for (x=1 : x<=LanguagePronouns-->0 : x=x+3)
        if (LanguagePronouns-->x == dword) {
            LanguagePronouns-->(x+2) = value; return;
        }
    RunTimeError(14);
];
[ PronounValue dword x;
    for (x=1 : x<=LanguagePronouns-->0 : x=x+3)
        if (LanguagePronouns-->x == dword)
            return LanguagePronouns-->(x+2);
    return 0;
];
[ ResetVagueWords obj; PronounNotice(obj); ];
[ PronounNotice obj x bm;
    if (obj == player) return;
    bm = PowersOfTwo_TB-->(GetGNAOfObject(obj));
    for (x=1 : x<=LanguagePronouns-->0 : x=x+3)
        if (bm & (LanguagePronouns-->(x+1)) ~= 0)
            LanguagePronouns-->(x+2) = obj;
];
[ PronounNoticeHeldObjects x;
#IFNDEF MANUAL_PRONOUNS;
    objectloop(x in player) PronounNotice(x);
#ENDIF;
    x = 0; ! To prevent a "not used" error
    rfalse;
];
```

§60. Yes/No Questions.

```
[ YesOrNo i j;
    for (::) {
        #Ifdef TARGET_ZCODE;
        if (location == nothing || parent(player) == nothing) read buffer parse;
        else read buffer parse DrawStatusLine;
        j = parse->1;
        #Ifnot; ! TARGET_GLULX;
        KeyboardPrimitive(buffer, parse);
        j = parse-->0;
        #Endif; ! TARGET_
        if (j) { ! at least one word entered
            i = parse-->1;
            if (i == YES1__WD or YES2__WD or YES3__WD) rtrue;
            if (i == NO1__WD or NO2__WD or NO3__WD) rfalse;
        }
        L__M(##Quit, 1); print "> ";
    }
];
```

§61. Number Words. Not much of a parsing routine: we look through an array of pairs of number words (single words) and their numeric equivalents.

```
[ NumberWord o i n;
    n = LanguageNumbers-->0;
    for (i=1 : i<=n : i=i+2)
        if (o == LanguageNumbers-->i) return LanguageNumbers-->(i+1);
    return 0;
];
```

§62. Choose Objects. This material, the final body of code in the parser, is an I7 addition. The I6 parser leaves it to the user to provide a `ChooseObjects` routine to decide between possibilities when the situation is ambiguous. For I7 use, we provide a `ChooseObjects` which essentially runs the "does the player mean" rulebook to decide, though this is not obvious from the code below because it is hidden in the `CheckDPMR` routine – which is defined in the Standard Rules, not here.

```
!Constant COBJ_DEBUG;

! the highest value returned by CheckDPMR (see the Standard Rules)
Constant HIGHEST_DPMR_SCORE = 4;

Array alt_match_list --> (MATCH_LIST_WORDS+1);

#ifdef TARGET_GLULX;
[ COBJ__Copy words from to  i;
    for (i=0: i<words: i++)
        to-->i = from-->i;
];
#ifnot;
[ COBJ__Copy words from to  bytes;
    bytes = words * 2;
    @copy_table from to bytes;
];
```

```
#endif;
! swap alt_match_list with match_list/number_matched
[ COBJ__SwapMatches i x;
    ! swap the counts
    x = number_matched;
    number_matched = alt_match_list-->0;
    alt_match_list-->0 = x;
    ! swap the values
    if (x < number_matched) x = number_matched;
    for (i=x: i>0: i--) {
        x = match_list-->(i-1);
        match_list-->(i-1) = alt_match_list-->i;
        alt_match_list-->i = x;
    }
];
[ ChooseObjects obj code  l i swn spcount;
    if (code<2) rfalse;
    if (cobj_flag == 1) {
        .CodeOne;
        if (parameters > 0) {
            #ifdef COBJ_DEBUG;
            print "[scoring ", (the) obj, " (second)]^";
            #endif;
            return ScoreDabCombo(parser_results-->INP1_PRES, obj);
        } else {
            #ifdef COBJ_DEBUG;
            print "[scoring ", (the) obj, " (first) in ",
                alt_match_list-->0, " combinations]^";
            #endif;
            l = 0;
            for (i=1: i<=alt_match_list-->0: i++) {
                spcount = ScoreDabCombo(obj, alt_match_list-->i);
                if (spcount == HIGHEST_DPMR_SCORE) {
                    #ifdef COBJ_DEBUG;
                    print "[scored ", spcount, " - best possible]^";
                    #endif;
                    return spcount;
                }
                if (spcount>l) l = spcount;
            }
            return l;
        }
    }
    if (cobj_flag == 2) {
        .CodeTwo;
        #ifdef COBJ_DEBUG;
        print "[scoring ", (the) obj, " (simple); parameters = ", parameters,
            " aw = ", advance_warning, "]^";
        #endif;
        @push action_to_be;
        if (parameters==0) {
            if (advance_warning > 0)
                l = ScoreDabCombo(obj, advance_warning);
```

```
            else
                l = ScoreDabCombo(obj, 0);
        } else {
            l = ScoreDabCombo(parser_results-->INP1_PRES, obj);
        }
        @pull action_to_be;
        return l;
    }
    #ifdef COBJ_DEBUG;
    print "[choosing a cobj strategy: ";
    #endif;
    swn = wn;
    spcount = pcount;
    while (line_ttype-->pcount == PREPOSITION_TT) pcount++;
    if (line_ttype-->pcount == ELEMENTARY_TT) {
        while (wn <= num_words) {
            l = NextWordStopped(); wn--;
            if ( (l ~= -1 or 0) && (l->#dict_par1) &8 ) { wn++; continue; } ! if preposition
            if (l == ALL1__WD or ALL2__WD or ALL3__WD or ALL4__WD or ALL5__WD) { wn++; continue; }
            SafeSkipDescriptors();
            ! save the current match state
            @push match_length; @push token_filter; @push match_from;
            alt_match_list-->0 = number_matched;
            COBJ__Copy(number_matched, match_list, alt_match_list+WORDSIZE);
            ! now get all the matches for the second noun
            match_length = 0; number_matched = 0; match_from = wn;
            token_filter = 0;
            SearchScope(actor, actors_location, line_tdata-->pcount);
            #ifdef COBJ_DEBUG;
            print number_matched, " possible second nouns]^";
            #endif;
            wn = swn;
            cobj_flag = 1;
            ! restore match variables
            COBJ__SwapMatches();
            @pull match_from; @pull token_filter; @pull match_length;
            pcount = spcount;
            jump CodeOne;
        }
    }
    pcount = spcount;
    wn = swn;
    #ifdef COBJ_DEBUG;
    print "nothing interesting]^";
    #endif;
    cobj_flag = 2;
    jump CodeTwo;
];
[ ScoreDabCombo a b  result;
    @push action; @push act_requester; @push noun; @push second;
    action = action_to_be;
    act_requester = player;
    if (action_reversed) { noun = b; second = a; }
```

```
    else { noun = a; second = b; }
    result = CheckDPMR();
    @pull second; @pull noun; @pull act_requester; @pull action;
    #ifdef COBJ_DEBUG;
    print "[", (the) a, " / ", (the) b, " => ", result, "]^";
    #endif;
    return result;
];
```

§**63. Default Topic.** A default value for the I7 sort-of-kind "topic", which never matches.

```
[ DefaultTopic; return GPR_FAIL; ];
```