RegExp Template B/regxt

Purpose

Code to match and replace on regular expressions against indexed text strings.

---

§**1. Head.** As ever: if there is no heap, there are no indexed texts, and if there are no indexed texts then there is no point in compiling any of this code.

```
#IFDEF MEMORY_HEAP_SIZE; ! Will exist if any use is made of indexed texts
Global IT_RE_Trace = false; ! Change to true for (a lot of) debugging data in use
[ IT_RE_SetTrace F; IT_RE_Trace = F; ];
```

§**2. Algorithm.** Once Inform 7 supported indexed text, regular-expression matching became an obvious goal: regexp-based features offer more or less the gold standard in text search and replace facilities, and I7 is so concerned with text that we shouldn't make do with less. But the best and most portable implementation of regular expression matching, PCRE by Philip Hazel, is about a hundred times larger than the code in this section, and also had unacceptable memory needs: there was no practicable way to make it small enough to do useful work on the Z-machine. Nor could an I6 regexp-matcher compile just-in-time code, or translate the expression into a suitable deterministic finite state machine. One day, though, I read one of the papers which Brian Kernighan writes every few years to the effect that regular-expression matching is much easier than you think. Kernighan is right: writing a regexp matcher is indeed easier than you think (one day's worth of cheerful hacking), but debugging one until it passes the trickiest hundred of Perl's 645 test cases is another matter (and it took a whole week more). Still, the result seems to be robust. The main compromise made is that backtracking is not always comprehensive with regexps like `^(a\1?){4}$`, because we do not allocate individual storage to backtrack individually through all possibilities of each of the four uses of the bracketed subexpression – which means we miss some cases, since the subexpression contains a reference to itself, so that its content can vary in the four uses. PCRE's approach here is to expand the expression as if it were a sequence of four bracketed expressions, thus removing the awkward quantifier `{4}`, but that costs memory: indeed this is why PCRE cannot solve all of Perl's test cases without its default memory allocation being raised. In other respects, the algorithm below appears to be accurate if not very fast.

§**3. Class Codes.** While in principle we could keep the match expression in textual form, in practice the syntax of regular expressions is complex enough that this would be tricky and rather slow: we would be parsing the same notations over and over again. So we begin by compiling it to a simple tree structure. The tree is made up of nodes, and each node has a "class code": these are identified by the `*_RE_CC` constants below. Note that the class codes below are all negative: this is so that they are distinct from all valid ZSCII or Unicode characters. (ZSCII is used only on the Z-machine, which has a 16-bit word but an 8-bit character set, so that all character values are positive; similarly, Unicode is (for our purposes) a 16-bit character set on a 32-bit virtual machine.)

```
! Character classes
Constant NEWLINE_RE_CC = -1;
Constant TAB_RE_CC = -2;
Constant DIGIT_RE_CC = -3;
Constant NONDIGIT_RE_CC = -4;
Constant WHITESPACE_RE_CC = -5;
Constant NONWHITESPACE_RE_CC = -6;
Constant PUNCTUATION_RE_CC = -7;
Constant NONPUNCTUATION_RE_CC = -8;
Constant WORD_RE_CC = -9;
Constant NONWORD_RE_CC = -10;
Constant ANYTHING_RE_CC = -11;
Constant NOTHING_RE_CC = -12;
Constant RANGE_RE_CC = -13;
Constant LCASE_RE_CC = -14;
Constant NONLCASE_RE_CC = -15;
Constant UCASE_RE_CC = -16;
Constant NONUCASE_RE_CC = -17;
! Control structures
Constant SUBEXP_RE_CC = -20;
Constant DISJUNCTION_RE_CC = -21;
Constant CHOICE_RE_CC = -22;
Constant QUANTIFIER_RE_CC = -23;
Constant IF_RE_CC = -24;
Constant CONDITION_RE_CC = -25;
Constant THEN_RE_CC = -26;
Constant ELSE_RE_CC = -27;
! Substring matchers
Constant VARIABLE_RE_CC = -30;
Constant LITERAL_RE_CC = -31;
! Positional matchers
Constant START_RE_CC = -40;
Constant END_RE_CC = -41;
Constant BOUNDARY_RE_CC = -42;
Constant NONBOUNDARY_RE_CC = -43;
Constant ALWAYS_RE_CC = -44;
Constant NEVER_RE_CC = -45;
! Mode switches
Constant SENSITIVITY_RE_CC = -50;
```

§**4. Packets.** The nodes of the compiled expression tree are stored in "packets", which are segments of a fixed array. A regexp complicated enough that it cannot be stored in `RE_MAX_PACKETS` packets will be rejected with an error: it looks like a rather low limit, but in fact suffices to handle all of Perl's test cases, some of which are works of diabolism.

A packet is then a record containing 14 fields, with offsets defined by the constants defined below. These fields combine the compilation of the corresponding fragment of the regexp with both the tree structure holding these packets together and also the current state of the temporary variables recording how far we have progressed in trying all of the possible ways to match the packet.

```
Constant RE_MAX_PACKETS = 32;
Constant RE_PACKET_SIZE = 14; ! Words of memory used per packet
Constant RE_PACKET_SIZE_IN_BYTES = WORDSIZE*RE_PACKET_SIZE; ! Bytes used per packet
Array RE_PACKET_space --> RE_MAX_PACKETS*RE_PACKET_SIZE;
Constant RE_CCLASS = 0;      ! One of the class codes defined above
Constant RE_PAR1 = 1;        ! Three parameters whose meaning depends on class code
Constant RE_PAR2 = 2;
Constant RE_PAR3 = 3;
Constant RE_NEXT = 4;        ! Younger sibling in the compiled tree
Constant RE_PREVIOUS = 5;    ! Elder sibling
Constant RE_DOWN = 6;        ! Child
Constant RE_UP = 7;          ! Parent
Constant RE_DATA1 = 8;       ! Backtracking data
Constant RE_DATA2 = 9;
Constant RE_CONSTRAINT = 10;
Constant RE_CACHE1 = 11;
Constant RE_CACHE2 = 12;
Constant RE_MODES = 13;
```

§**5. Nodes.** The routine to create a node, something which happens only during the compilation phase, and also the routine which returns the address of a given node. Nodes are numbered from 0 up to $M-1$, where M is the constant `RE_MAX_PACKETS`.

```
[ IT_RE_Node n cc par1 par2 par3  offset;
    if ((n<0) || (n >= RE_MAX_PACKETS)) rfalse;
    offset = RE_PACKET_space + n*RE_PACKET_SIZE_IN_BYTES;
    offset-->RE_CCLASS = cc;
    offset-->RE_PAR1 = par1;
    offset-->RE_PAR2 = par2;
    offset-->RE_PAR3 = par3;
    offset-->RE_NEXT = NULL;
    offset-->RE_PREVIOUS = NULL;
    offset-->RE_DOWN = NULL;
    offset-->RE_UP = NULL;
    offset-->RE_DATA1 = -1; ! Match start
    offset-->RE_DATA2 = -1; ! Match end
    offset-->RE_CONSTRAINT = -1; ! Rewind edge
    return offset;
];
[ IT_RE_NodeAddress n;
    if ((n<0) || (n >= RE_MAX_PACKETS)) return -1;
    return RE_PACKET_space + n*RE_PACKET_SIZE_IN_BYTES;
];
```

§**6. Match Variables.** A bracketed subexpression can be used as a variable: we support `\1`, ..., `\9` to mean "the value of subexpression 1 to 9", and `\0` to mean "the whole text matched", as if the entire regexp were bracketed. (PCRE and Perl also allow `\10`, `\11`, ..., but we don't, because it complicates parsing and memory is too short.)

`RE_Subexpressions-->10` stores the number of subexpressions in use, not counting `\0`. During the compiling stage, `RE_Subexpressions-->N` is set to point to the node representating `\N`, where `N` varies from 1 to 9. When matching is complete, and assuming we care about the contents of these variables – which we might not, and if not we certainly don't want to waste time and memory – we call `IT_RE_CreateMatchVars` to allocate indexed text variables and fill them in as appropriate, memory permitting.

`IT_RE_EmptyMatchVars` empties any such variables which may survive from a previous match, setting them to the empty text.

```
Array RE_Subexpressions --> 11; ! Address of node for this subexpression
Array Allocated_Match_Vars --> 10; ! Indexed text to hold values of the variables

[ IT_RE_DebugMatchVars indt
    offset n i;
    print RE_Subexpressions-->10, " collecting subexps^";
    for (n=0:(n<RE_Subexpressions-->10) && (n<10): n++) {
        offset = RE_Subexpressions-->n;
        print "Subexp ", offset-->RE_PAR1,
            " = [", offset-->RE_DATA1, ",", offset-->RE_DATA2, "] = ";
        for (i=offset-->RE_DATA1:i<offset-->RE_DATA2:i++)
            print (char) BlkValueRead(indt, i);
        print "^";
    }
];

[ IT_RE_CreateMatchVars indt
    offset n i ch cindt cl csize;
    for (n=0:(n<RE_Subexpressions-->10) && (n<10): n++) {
        offset = RE_Subexpressions-->n;
        if (Allocated_Match_Vars-->n == 0)
            Allocated_Match_Vars-->n = INDEXED_TEXT_TY_Create(); ! Permanently
        cindt = Allocated_Match_Vars-->n;
        csize = BlkValueExtent(cindt);
        cl = 0;
        for (i=offset-->RE_DATA1:i<offset-->RE_DATA2:i++) {
            ch = BlkValueRead(indt, i);
            if (cl+1 >= csize) {
                if (BlkValueSetExtent(cindt, 2*cl, 6) == false) break;
                csize = BlkValueExtent(cindt);
            }
            BlkValueWrite(cindt, cl++, ch);
        }
        BlkValueWrite(cindt, cl, 0);
    }
];

[ IT_RE_EmptyMatchVars indt
    n;
    for (n=0:(n<RE_Subexpressions-->10) && (n<10): n++)
        if (Allocated_Match_Vars-->n ~= 0)
            BlkValueWrite(Allocated_Match_Vars-->n, 0, 0);
];
```

```
[ IT_RE_GetMatchVar indt vn
    offset;
    if ((vn<0) || (vn>=10) || (vn >= RE_Subexpressions-->10)) jump Nope;
    offset = RE_Subexpressions-->vn;
    if (offset == 0) jump Nope;
    if (offset-->RE_DATA1 < 0) jump Nope;
    if (Allocated_Match_Vars-->vn == 0) {
        print "*** ", vn, " unallocated! ***^";
        jump Nope;
    }
    BlkValueCopy(indt, Allocated_Match_Vars-->vn);
    return indt;
    .Nope;
    BlkValueWrite(indt, 0, 0);
    return indt;
];
```

§**7. Markers.** At each node, the `-->RE_DATA1` and `-->RE_DATA2` fields represent the character positions of the start and end of the text matched by the node and its subtree (if any). These are called markers.

Thus `IT_MV_End(N, 0)` returns the start of `\N` and `IT_MV_End(N, 1)` the end of `\N`, according to the current match of subexpression `N`.

```
[ IT_MV_End n end
    offset;
    offset = RE_Subexpressions-->n;
    if (end==0) return offset-->RE_DATA1;
    return offset-->RE_DATA2;
];
[ IT_RE_Clear_Markers token;
    for (: token ~= NULL: token = token-->RE_NEXT) {
        if (token-->RE_DOWN ~= NULL) IT_RE_Clear_Markers(token-->RE_DOWN);
        token-->RE_DATA1 = -1;
        token-->RE_DATA2 = -1;
        token-->RE_CONSTRAINT = -1;
    }
];
```

§**8. Debugging.** Code in this paragraph simply prints a convenient screen representation of the compiled regexp, together with the current values of its markers. It is invaluable for debugging purposes and, touch wood, may not be needed again, but it is relatively compact and we keep it just in case.

```
[ IT_RE_DebugTree findt detail;
    print "Pattern: ", (INDEXED_TEXT_TY_Say) findt, "^";
    IT_RE_DebugSubtree(findt, 1, RE_PACKET_space, detail);
];
[ IT_RE_DebugSubtree findt depth offset detail
    cup;
    if (offset ~= NULL) {
        cup = offset-->RE_UP;
        if (offset-->RE_PREVIOUS ~= NULL) print "*** broken initial previous ***^";
    }
    while (offset ~= NULL) {
        if (offset-->RE_UP ~= cup) print "*** broken up matching ***^";
        spaces(depth*2);
        IT_RE_DebugNode(offset, findt, detail);
        if (offset-->RE_DOWN ~= NULL) {
            if ((offset-->RE_DOWN)-->RE_UP ~= offset)
                print "*** broken down/up ***^";
            IT_RE_DebugSubtree(findt, depth+1, offset-->RE_DOWN, detail);
        }
        if (offset-->RE_NEXT ~= NULL) {
            if ((offset-->RE_NEXT)-->RE_PREVIOUS ~= offset)
                print "*** broken next/previous ***^";
        }
        offset = offset-->RE_NEXT;
    }
];
[ IT_RE_DebugNode offset findt detail
    i par1 par2 par3;
    if (offset == NULL) "[NULL]";
    print "[", (offset-RE_PACKET_space)/(RE_PACKET_SIZE_IN_BYTES), "] ";
    ! for (i=0:i<RE_PACKET_SIZE:i++) print offset-->i, " ";
    par1 = offset-->RE_PAR1;
    par2 = offset-->RE_PAR2;
    par3 = offset-->RE_PAR3;
    switch (offset-->RE_CCLASS) {
        DIGIT_RE_CC: print "DIGIT";
        NONDIGIT_RE_CC: print "NONDIGIT";
        UCASE_RE_CC: print "UCASE";
        NONUCASE_RE_CC: print "NONUCASE";
        LCASE_RE_CC: print "LCASE";
        NONLCASE_RE_CC: print "NONLCASE";
        WHITESPACE_RE_CC: print "WHITESPACE";
        NONWHITESPACE_RE_CC: print "NONWHITESPACE";
        PUNCTUATION_RE_CC: print "PUNCTUATION";
        NONPUNCTUATION_RE_CC: print "NONPUNCTUATION";
        WORD_RE_CC: print "WORD";
        NONWORD_RE_CC: print "NONWORD";
        ALWAYS_RE_CC: print "ALWAYS";
        NEVER_RE_CC: print "NEVER";
```

```
        START_RE_CC: print "START";
        END_RE_CC: print "END";
        BOUNDARY_RE_CC: print "BOUNDARY";
        NONBOUNDARY_RE_CC: print "NONBOUNDARY";
        ANYTHING_RE_CC: print "ANYTHING";
        NOTHING_RE_CC: print "NOTHING";
        RANGE_RE_CC: print "RANGE"; if (par3 == true) print " (negated)";
            print " ";
            for (i=par1:i<par2:i++) print (char) BlkValueRead(findt, i);
        VARIABLE_RE_CC: print "VARIABLE ", par1;
        SUBEXP_RE_CC:
            if (par1 == 0) print "EXP";
            else print "SUBEXP ";
            if (par1 >= 0) print "= V", par1;
            if (par2 == 1) {
                if (par3 == 0) print " (?=...) lookahead";
                else print " (?<=...) lookbehind of width ", par3;
            }
            if (par2 == 2) {
                if (par3 == 0) print " (?!...) negated lookahead";
                else print " (?<!...) negated lookbehind of width ", par3;
            }
            if (par2 == 3) print " uncollecting";
            if (par2 == 0 or 3) {
                if (par3 == 1) print " forcing case sensitivity";
                if (par3 == 2) print " forcing case insensitivity";
            }
            if (par2 == 4) print " (?>...) possessive";
        NEWLINE_RE_CC: print "NEWLINE";
        TAB_RE_CC: print "TAB";
        QUANTIFIER_RE_CC: print "QUANTIFIER min=", par1, " max=", par2;
            if (par3) print " (lazy)"; else print " (greedy)";
        LITERAL_RE_CC: print "LITERAL";
            print " ";
            for (i=par1:i<par2:i++) print (char) BlkValueRead(findt, i);
        DISJUNCTION_RE_CC: print "DISJUNCTION of ", par1, " choices";
        CHOICE_RE_CC: print "CHOICE no ", par1;
        SENSITIVITY_RE_CC: print "SENSITIVITY";
            if (par1) print " off"; else print " on";
        IF_RE_CC: print "IF"; if (par1 >= 1) print " = V", par1;
        CONDITION_RE_CC: print "CONDITION"; if (par1 >= 1) print " = V", par1;
        THEN_RE_CC: print "THEN";
        ELSE_RE_CC: print "ELSE";
    }
    if (detail)
        print ": ", offset-->RE_DATA1, ", ", offset-->RE_DATA2, ", ", offset-->RE_CONSTRAINT;
    print "^";
];
```

§**9. Compiling Tree For Substring Search.** When we search for a literal substring, say looking for "per" in "Supernumerary", we will in fact use the same apparatus as when searching for a regular expression: we compile a very simple node tree in which `\0` as the root contains just one child node, a `LITERAL_RE_CC` matching exactly the text "per". We return 2 since that's the number of nodes in the tree.

```
[ IT_CHR_CompileTree findt exactly
    root literal fto no_packets token attach_to;
    fto = IT_CharacterLength(findt);
    root = IT_RE_Node(0, SUBEXP_RE_CC, 0, 0, 0);
    literal = IT_RE_Node(1, LITERAL_RE_CC, 0, fto, 0);
    root-->RE_DOWN = literal;
    literal-->RE_UP = root;
    if (exactly) {
        no_packets = 2;
        if (no_packets+3 > RE_MAX_PACKETS) return "regexp too complex";
        exactly = RE_PACKET_space-->RE_DOWN;
        token = IT_RE_Node(no_packets++, START_RE_CC, 0, 0, 0);
        RE_PACKET_space-->RE_DOWN = token; token-->RE_UP = RE_PACKET_space;
        attach_to = IT_RE_Node(no_packets++, SUBEXP_RE_CC, -1, 3, 0);
        token-->RE_NEXT = attach_to; attach_to-->RE_PREVIOUS = token;
        attach_to-->RE_UP = RE_PACKET_space;
        attach_to-->RE_NEXT = IT_RE_Node(no_packets++, END_RE_CC, 0, 0, 0);
        (attach_to-->RE_NEXT)-->RE_PREVIOUS = attach_to;
        (attach_to-->RE_NEXT)-->RE_UP = RE_PACKET_space;
        attach_to-->RE_DOWN = exactly;
        while (exactly ~= NULL) {
            exactly-->RE_UP = attach_to; exactly = exactly-->RE_NEXT;
        }
    }
    no_packets = IT_RE_ExpandChoices(RE_PACKET_space, no_packets);
];
```

§**10. Compiling Tree For Regexp Search.** But in general we need to compile a regular expression string into a tree of the kind described above, and here is the routine which does that, returning the number of nodes used to build the tree. The syntax it accepts is very fully documented in *Writing with Inform*, so no details are given here.

```
Array Subexp_Posns --> 20;
[ IT_RE_CompileTree findt exactly
    no_packets ffrom fto cc par1 par2 par3
    quantifiable token attach_to no_subs blevel bits;
    fto = IT_CharacterLength(findt);
    if (fto == 0) {
        IT_RE_Node(no_packets++, NEVER_RE_CC, 0, 0, 0); ! Empty regexp never matches
        return 1;
    }
    attach_to = IT_RE_Node(no_packets++, SUBEXP_RE_CC, 0, 0, 0);
    RE_Subexpressions-->0 = attach_to; RE_Subexpressions-->10 = 1; no_subs = 1;
    quantifiable = false; blevel = 0;
    for (ffrom = 0: ffrom < fto: ) {
        cc = BlkValueRead(findt, ffrom++); par1 = 0; par2 = 0; par3 = 0;
```

```
if (cc == '\') {
    if (ffrom == fto) return "Search pattern not terminated";
    cc = BlkValueRead(findt, ffrom++);
    switch (cc) {
        'b': cc = BOUNDARY_RE_CC;
        'B': cc = NONBOUNDARY_RE_CC;
        'd': cc = DIGIT_RE_CC;
        'D': cc = NONDIGIT_RE_CC;
        'l': cc = LCASE_RE_CC;
        'L': cc = NONLCASE_RE_CC;
        'n': cc = NEWLINE_RE_CC;
        'p': cc = PUNCTUATION_RE_CC;
        'P': cc = NONPUNCTUATION_RE_CC;
        's': cc = WHITESPACE_RE_CC;
        'S': cc = NONWHITESPACE_RE_CC;
        't': cc = TAB_RE_CC;
        'u': cc = UCASE_RE_CC;
        'U': cc = NONUCASE_RE_CC;
        'w': cc = WORD_RE_CC;
        'W': cc = NONWORD_RE_CC;
        default:
            if ((cc >= '1') && (cc <= '9')) {
                par1 = cc-'0';
                cc = VARIABLE_RE_CC;
            } else {
                if (((cc >= 'a') && (cc <= 'z')) ||
                    ((cc >= 'A') && (cc <= 'Z'))) return "unknown escape";
                cc = LITERAL_RE_CC;
                par1 = ffrom-1; par2 = ffrom;
            }
    }
    quantifiable = true;
} else {
    switch (cc) {
        '(': par2 = 0;
            !if (BlkValueRead(findt, ffrom) == ')') return "empty subexpression";
            if (BlkValueRead(findt, ffrom) == '?') {
                ffrom++;
                bits = true;
                if (BlkValueRead(findt, ffrom) == '-') { ffrom++; bits = false; }
                else if (BlkValueRead(findt, ffrom) == '<') { ffrom++; bits = false; }
                switch (cc = BlkValueRead(findt, ffrom++)) {
                    '#': while (BlkValueRead(findt, ffrom++) ~= 0 or ')') ;
                        if (BlkValueRead(findt, ffrom-1) == 0)
                            return "comment never ends";
                        continue;
                    '(': cc = BlkValueRead(findt, ffrom);
                        if ((cc == '1' or '2' or '3' or '4' or
                            '5' or '6' or '7' or '8' or '9') &&
                            (BlkValueRead(findt, ffrom+1) ==')')) {
                            ffrom = ffrom + 2;
                            par1 = cc - '0';
                        } else ffrom--;
```

```
                cc = IF_RE_CC; ! (?(...)...) conditional
                quantifiable = false;
                if (blevel == 20) return "subexpressions too deep";
                Subexp_Posns-->(blevel++) = IT_RE_NodeAddress(no_packets);
                jump CClassKnown;
            '=': par2 = 1; ! (?=...) lookahead/behind
                par3 = 0; if (bits == false) par3 = -1;
            '!': par2 = 2; ! (?!...) negated lookahead/behind
                par3 = 0; if (bits == false) par3 = -1;
            ':': par2 = 3; ! (?:...) uncollecting subexpression
            '>': par2 = 4; ! (?>...) possessive
            default:
                if (BlkValueRead(findt, ffrom) == ')') {
                    if (cc == 'i') {
                        cc = SENSITIVITY_RE_CC; par1 = bits; ffrom++;
                        jump CClassKnown;
                    }
                }
                if (BlkValueRead(findt, ffrom) == ':') {
                    if (cc == 'i') {
                        par1 = bits; par2 = 3; par3 = bits+1; ffrom++;
                        jump AllowForm;
                    }
                }
                return "unknown (?...) form";
        }
    }
    .AllowForm;
    if (par2 == 0) par1 = no_subs++; else par1 = -1;
    cc = SUBEXP_RE_CC;
    quantifiable = false;
    if (blevel == 20) return "subexpressions too deep";
    Subexp_Posns-->(blevel++) = IT_RE_NodeAddress(no_packets);
')': if (blevel == 0) return "subexpression bracket mismatch";
    blevel--;
    attach_to = Subexp_Posns-->blevel;
    if (attach_to-->RE_DOWN == NULL) {
        if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
        attach_to-->RE_DOWN =
            IT_RE_Node(no_packets++, ALWAYS_RE_CC, 0, 0, 0);
        (attach_to-->RE_DOWN)-->RE_UP = attach_to;
    }
    quantifiable = true;
    continue;
'.': cc = ANYTHING_RE_CC; quantifiable = true;
'|': cc = CHOICE_RE_CC; quantifiable = false;
'^': cc = START_RE_CC; quantifiable = false;
'$': cc = END_RE_CC; quantifiable = false;
'{': if (quantifiable == false) return "quantifier misplaced";
    par1 = 0; par2 = -1; bits = 1;
    while ((cc=BlkValueRead(findt, ffrom++)) ~= 0 or '}') {
        if (cc == ',') {
            bits++;
```

```
            if (bits >= 3) return "too many colons in ?{...}";
            continue;
        }
        if ((cc >= '0') || (cc <= '9')) {
            if (bits == 1) {
                if (par1 < 0) par1 = 0;
                par1 = par1*10 + (cc-'0');
            } else {
                if (par2 < 0) par2 = 0;
                par2 = par2*10 + (cc-'0');
            }
        } else return "non-digit in ?{...}";
    }
    if (cc ~= '}') return "{x,y} quantifier never ends";
    cc = QUANTIFIER_RE_CC;
    if (par2 == -1) {
        if (bits == 2) par2 = 30000;
        else par2 = par1;
    }
    if (par1 > par2) return "{x,y} with x greater than y";
    if (BlkValueRead(findt, ffrom) == '?') { ffrom++; par3 = true; }
    quantifiable = false;
'<', '[': par3 = false; if (cc == '<') bits = '>'; else bits = ']';
    if (BlkValueRead(findt, ffrom) == '^') { ffrom++; par3 = true; }
    par1 = ffrom;
    if (BlkValueRead(findt, ffrom) == bits) { ffrom++; }
    while (cc ~= bits or 0) {
        cc = BlkValueRead(findt, ffrom++);
        if (cc == '\') {
            cc = BlkValueRead(findt, ffrom++);
            if (cc ~= 0) cc = BlkValueRead(findt, ffrom++);
        }
    }
    if (cc == 0) return "Character range never ends";
    par2 = ffrom-1;
    if ((par2 > par1 + 1) &&
        (BlkValueRead(findt, par1) == ':') &&
        (BlkValueRead(findt, par2-1) == ':') &&
        (BlkValueRead(findt, par2-2) ~= '\'))
        return "POSIX named character classes unsupported";
    bits = IT_RE_RangeSyntaxCorrect(findt, par1, par2);
    if (bits) return bits;
    if (par1 < par2) cc = RANGE_RE_CC;
    else cc = NOTHING_RE_CC;
    quantifiable = true;
'*': if (quantifiable == false) return "quantifier misplaced";
    cc = QUANTIFIER_RE_CC;
    par1 = 0; par2 = 30000;
    if (BlkValueRead(findt, ffrom) == '?') { ffrom++; par3 = true; }
    quantifiable = false;
'+': if (quantifiable == false) return "quantifier misplaced";
    cc = QUANTIFIER_RE_CC;
    par1 = 1; par2 = 30000;
```

```
            if (BlkValueRead(findt, ffrom) == '?') { ffrom++; par3 = true; }
            quantifiable = false;
        '?': if (quantifiable == false) return "quantifier misplaced";
            cc = QUANTIFIER_RE_CC;
            par1 = 0; par2 = 1;
            if (BlkValueRead(findt, ffrom) == '?') { ffrom++; par3 = true; }
            quantifiable = false;
    }
}
.CClassKnown;
if (cc >= 0) {
    quantifiable = true;
    if ((attach_to-->RE_CCLASS == LITERAL_RE_CC) &&
        (BlkValueRead(findt, ffrom) ~= '*' or '+' or '?' or '{')) {
        (attach_to-->RE_PAR2)++;
        if (IT_RE_Trace == 2) {
            print "Extending literal by ", cc, "=", (char) cc, "^";
        }
        continue;
    }
    cc = LITERAL_RE_CC; par1 = ffrom-1; par2 = ffrom;
}
if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
if (IT_RE_Trace == 2) {
    print "Attaching packet ", no_packets+1, " to ";
    IT_RE_DebugNode(attach_to, findt);
    IT_RE_DebugTree(findt);
}
token = IT_RE_Node(no_packets++, cc, par1, par2, par3);
if ((token-->RE_CCLASS == SUBEXP_RE_CC) && (token-->RE_PAR2 == 0)) {
    RE_Subexpressions-->(token-->RE_PAR1) = token;
    (RE_Subexpressions-->10)++;
}
if ((attach_to-->RE_CCLASS == SUBEXP_RE_CC or CHOICE_RE_CC or IF_RE_CC) &&
    (attach_to-->RE_DOWN == NULL)) {
    attach_to-->RE_DOWN = token; token-->RE_UP = attach_to;
} else {
    if ((token-->RE_CCLASS == CHOICE_RE_CC) &&
        ((attach_to-->RE_UP)-->RE_CCLASS == CHOICE_RE_CC)) {
        no_packets--; token = attach_to-->RE_UP;
    } else {
        if (token-->RE_CCLASS == CHOICE_RE_CC) {
            while (attach_to-->RE_PREVIOUS ~= NULL)
                attach_to = attach_to-->RE_PREVIOUS;
        }
        if (token-->RE_CCLASS == QUANTIFIER_RE_CC or CHOICE_RE_CC) {
            token-->RE_PREVIOUS = attach_to-->RE_PREVIOUS;
            token-->RE_UP = attach_to-->RE_UP;
            if ((attach_to-->RE_UP ~= NULL) && (attach_to-->RE_PREVIOUS == NULL))
                (attach_to-->RE_UP)-->RE_DOWN = token;
            token-->RE_DOWN = attach_to;
            bits = attach_to;
```

```
                    while (bits ~= NULL) {
                        bits-->RE_UP = token;
                        bits = bits-->RE_NEXT;
                    }
                    attach_to-->RE_PREVIOUS = NULL;
                    if (token-->RE_PREVIOUS ~= NULL)
                        (token-->RE_PREVIOUS)-->RE_NEXT = token;
                } else {
                    attach_to-->RE_NEXT = token; token-->RE_PREVIOUS = attach_to;
                    token-->RE_UP = attach_to-->RE_UP;
                }
            }
        }
        if (token-->RE_CCLASS == CHOICE_RE_CC) {
            if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
            token-->RE_NEXT = IT_RE_Node(no_packets++, CHOICE_RE_CC, 0, 0, 0);
            (token-->RE_NEXT)-->RE_PREVIOUS = token;
            (token-->RE_NEXT)-->RE_UP = token-->RE_UP;
            token = token-->RE_NEXT;
        }
        attach_to = token;
        if (IT_RE_Trace == 2) {
            print "Result:^";
            IT_RE_DebugTree(findt);
        }
    }
    if (blevel ~= 0) return "subexpression bracket mismatch";
    if (exactly) {
        if (no_packets+3 > RE_MAX_PACKETS) return "regexp too complex";
        exactly = RE_PACKET_space-->RE_DOWN;
        token = IT_RE_Node(no_packets++, START_RE_CC, 0, 0, 0);
        RE_PACKET_space-->RE_DOWN = token; token-->RE_UP = RE_PACKET_space;
        attach_to = IT_RE_Node(no_packets++, SUBEXP_RE_CC, -1, 3, 0);
        token-->RE_NEXT = attach_to; attach_to-->RE_PREVIOUS = token;
        attach_to-->RE_UP = RE_PACKET_space;
        attach_to-->RE_NEXT = IT_RE_Node(no_packets++, END_RE_CC, 0, 0, 0);
        (attach_to-->RE_NEXT)-->RE_PREVIOUS = attach_to;
        (attach_to-->RE_NEXT)-->RE_UP = RE_PACKET_space;
        attach_to-->RE_DOWN = exactly;
        while (exactly ~= NULL) {
            exactly-->RE_UP = attach_to; exactly = exactly-->RE_NEXT;
        }
    }
    no_packets = IT_RE_ExpandChoices(RE_PACKET_space, no_packets);
    if (IT_RE_Trace) {
        print "Compiled pattern:^";
        IT_RE_DebugTree(findt);
    }
    bits = IT_RE_CheckTree(RE_PACKET_space, no_subs); if (bits) return bits;
    return no_packets;
];
```

```
[ IT_RE_RangeSyntaxCorrect findt rf rt
    i chm;
    for (i=rf: i<rt: i++) {
        chm = BlkValueRead(findt, i);
        if ((chm == '\') && (i+1<rt)) {
            chm = BlkValueRead(findt, ++i);
            if (((chm >= 'a') && (chm <= 'z')) ||
                ((chm >= 'A') && (chm <= 'Z'))) {
                if (chm ~= 's' or 'S' or 'p' or 'P' or 'w' or 'W' or 'd'
                    or 'D' or 'n' or 't' or 'l' or 'L' or 'u' or 'U')
                    return "Invalid escape in {} range";
            }
        }
        if ((i+2<rt) && (BlkValueRead(findt, i+1) == '-')) {
            if (chm > BlkValueRead(findt, i+2)) return "Invalid {} range";
            i=i+2;
        }
    }
    rfalse;
];
[ IT_RE_ExpandChoices token no_packets
    rv prev nex holder new ct n cond_node then_node else_node;
    while (token ~= NULL) {
        if (token-->RE_CCLASS == IF_RE_CC) {
            if ((token-->RE_DOWN)-->RE_CCLASS == CHOICE_RE_CC) {
                for (nex=token-->RE_DOWN, n=0: nex~=NULL: nex=nex-->RE_NEXT) n++;
                if (n~=2) return "conditional has too many clauses";
                if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
                cond_node = IT_RE_Node(no_packets++, CONDITION_RE_CC, 0, 0, 0);
                if (token-->RE_PAR1 >= 1) {
                    cond_node-->RE_PAR1 = token-->RE_PAR1;
                }
                then_node = token-->RE_DOWN;
                then_node-->RE_CCLASS = THEN_RE_CC;
                else_node = then_node-->RE_NEXT;
                else_node-->RE_CCLASS = ELSE_RE_CC;
                if (cond_node-->RE_PAR1 < 1) {
                    cond_node-->RE_DOWN = then_node-->RE_DOWN;
                    then_node-->RE_DOWN = (then_node-->RE_DOWN)-->RE_NEXT;
                    if (then_node-->RE_DOWN ~= NULL)
                        (then_node-->RE_DOWN)-->RE_PREVIOUS = NULL;
                    (cond_node-->RE_DOWN)-->RE_NEXT = NULL;
                    (cond_node-->RE_DOWN)-->RE_UP = cond_node;
                }
                token-->RE_DOWN = cond_node; cond_node-->RE_UP = token;
                cond_node-->RE_NEXT = then_node; then_node-->RE_PREVIOUS = cond_node;
            } else {
                if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
                cond_node = IT_RE_Node(no_packets++, CONDITION_RE_CC, 0, 0, 0);
                if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
                then_node = IT_RE_Node(no_packets++, THEN_RE_CC, 0, 0, 0);
                if (token-->RE_PAR1 >= 1) {
                    cond_node-->RE_PAR1 = token-->RE_PAR1;
```

```
            then_node-->RE_DOWN = token-->RE_DOWN;
        } else {
            cond_node-->RE_DOWN = token-->RE_DOWN;
            then_node-->RE_DOWN = (token-->RE_DOWN)-->RE_NEXT;
            (cond_node-->RE_DOWN)-->RE_NEXT = NULL;
            (cond_node-->RE_DOWN)-->RE_UP = cond_node;
        }
        token-->RE_DOWN = cond_node;
        cond_node-->RE_UP = token; cond_node-->RE_NEXT = then_node;
        then_node-->RE_PREVIOUS = cond_node; then_node-->RE_UP = token;
        then_node-->RE_NEXT = NULL;
        if (then_node-->RE_DOWN ~= NULL)
            (then_node-->RE_DOWN)-->RE_PREVIOUS = NULL;
        for (nex = then_node-->RE_DOWN: nex ~= NULL: nex = nex-->RE_NEXT) {
            nex-->RE_UP = then_node;
        }
    }
    if (cond_node-->RE_DOWN ~= NULL) {
        nex = cond_node-->RE_DOWN;
        if ((nex-->RE_CCLASS ~= SUBEXP_RE_CC) ||
            (nex-->RE_NEXT ~= NULL) ||
            (nex-->RE_PAR2 ~= 1 or 2)) {
            !IT_RE_DebugSubtree(0, 0, nex, true);
            return "condition not lookahead/behind";
        }
    }
}
if ((token-->RE_CCLASS == CHOICE_RE_CC) && (token-->RE_PAR1 < 1)) {
    prev = token-->RE_PREVIOUS;
    nex = token-->RE_NEXT;
    while ((nex ~= NULL) && (nex-->RE_CCLASS == CHOICE_RE_CC))
        nex = nex-->RE_NEXT;
    holder = token-->RE_UP; if (holder == NULL) return "bang";
    if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
    new = IT_RE_Node(no_packets++, DISJUNCTION_RE_CC, 0, 0, 0);
    holder-->RE_DOWN = new; new-->RE_UP = holder;
    if (prev ~= NULL) {
        prev-->RE_NEXT = new; new-->RE_PREVIOUS = prev;
    }
    if (nex ~= NULL) {
        nex-->RE_PREVIOUS = new; new-->RE_NEXT = nex;
    }
    new-->RE_DOWN = token;
    token-->RE_PREVIOUS = NULL;
    ct = 1;
    while (token ~= NULL) {
        token-->RE_PAR1 = ct++;
        token-->RE_UP = new;
        if ((token-->RE_NEXT ~= NULL) &&
            ((token-->RE_NEXT)-->RE_CCLASS ~= CHOICE_RE_CC))
            token-->RE_NEXT = NULL;
        token = token-->RE_NEXT;
    }
```

```
            new-->RE_PAR1 = ct-1;
            if (token ~= NULL) token-->RE_NEXT = NULL;
            token = new; continue;
        }
        if (token-->RE_DOWN ~= NULL) {
            no_packets = IT_RE_ExpandChoices(token-->RE_DOWN, no_packets);
            if ((no_packets<0) || (no_packets >= RE_MAX_PACKETS)) break;
        }
        token = token-->RE_NEXT;
    }
    return no_packets;
];
[ IT_RE_CheckTree token no_subs
    rv;
    while (token ~= NULL) {
        if (token-->RE_CCLASS == VARIABLE_RE_CC) {
            if (token-->RE_PAR1 >= no_subs) return "reference to nonexistent group";
        }
        if ((token-->RE_CCLASS == SUBEXP_RE_CC) &&
            (token-->RE_PAR2 == 1 or 2) &&
            (token-->RE_PAR3 == -1)) {
            token-->RE_PAR3 = IT_RE_Width(token-->RE_DOWN);
            if (token-->RE_PAR3 == -1) return "variable length lookbehind not implemented";
        }
        if (token-->RE_DOWN ~= NULL) {
            rv = IT_RE_CheckTree(token-->RE_DOWN, no_subs);
            if (rv) return rv;
        }
        token = token-->RE_NEXT;
    }
    rfalse;
];
[ IT_RE_Width token downwards
    w rv aw choice;
    while (token ~= NULL) {
        switch (token-->RE_CCLASS) {
            DIGIT_RE_CC, NONDIGIT_RE_CC, WHITESPACE_RE_CC, NONWHITESPACE_RE_CC,
            PUNCTUATION_RE_CC, NONPUNCTUATION_RE_CC, WORD_RE_CC, NONWORD_RE_CC,
            ANYTHING_RE_CC, NOTHING_RE_CC, RANGE_RE_CC, NEWLINE_RE_CC, TAB_RE_CC,
            UCASE_RE_CC, NONUCASE_RE_CC, LCASE_RE_CC, NONLCASE_RE_CC:
                w++;
            START_RE_CC, END_RE_CC, BOUNDARY_RE_CC, NONBOUNDARY_RE_CC, ALWAYS_RE_CC:
                ;
            LITERAL_RE_CC:
                w = w + token-->RE_PAR2 - token-->RE_PAR1;
            VARIABLE_RE_CC:
                return -1;
            IF_RE_CC:
                rv = IT_RE_Width((token-->RE_DOWN)-->RE_NEXT);
                if (rv == -1) return -1;
                if (rv ~= IT_RE_Width(((token-->RE_DOWN)-->RE_NEXT)-->RE_NEXT))
                    return -1;
                w = w + rv;
```

```
            SUBEXP_RE_CC:
                if (token-->RE_PAR2 == 1 or 2) rv = 0;
                else {
                    rv = IT_RE_Width(token-->RE_DOWN);
                    if (rv == -1) return -1;
                }
                w = w + rv;
            QUANTIFIER_RE_CC:
                if (token-->RE_PAR1 ~= token-->RE_PAR2) return -1;
                rv = IT_RE_Width(token-->RE_DOWN);
                if (rv == -1) return -1;
                w = w + rv*(token-->RE_PAR1);
            DISJUNCTION_RE_CC:
                aw = -1;
                for (choice = token-->RE_DOWN: choice ~= NULL: choice = choice-->RE_NEXT) {
                    rv = IT_RE_Width(choice-->RE_DOWN);
                    !print "Option found ", rv, "^";
                    if (rv == -1) return -1;
                    if ((aw >= 0) && (aw ~= rv)) return -1;
                    aw = rv;
                }
                w = w + aw;
            SENSITIVITY_RE_CC:
                ;
        }
        if (downwards) return w;
        if (token ~= NULL) token = token-->RE_NEXT;
    }
    return w;
];
```

§**11. Parser.** The virtue of all of that tree compilation is that the code which actually does the work – which parses the source text to see if the regular expression matches it – is much shorter and quicker: indeed, it takes up fewer lines than the compiler part, which goes to show that decoding regular expression syntax is a more complex task than acting upon it. This would have surprised the pioneers of regexp, but the syntax has become much more complicated over the decades because of a steady increase in the number of extended notations. The process shows no sign of stopping, with Python and PCRE continuing to push boundaries beyond Perl, which was once thought the superest, duperest regexp syntax there could be. However: to work.

The main matcher simply starts a recursive subroutine to perform the match. However, the subroutine tests for a match at a particular position in the source text; so the main routine tries the subroutine everywhere convenient in the source text, from left to right, until a match is made – unless the regexp is constrained by a `^` glyph to begin matching at the start of the source text, which will cause a `START_RE_CC` node to be the eldest child of the `\0` root.

```
Global IT_RE_RewindCount;
[ IT_RE_PrintNoRewinds; print IT_RE_RewindCount; ];

Constant CIS_MFLAG = 1;
Constant ACCUM_MFLAG = 2;

[ IT_RE_Parse findt indt ipos insens
    ilen rv root i initial_mode;

    ilen = IT_CharacterLength(indt);
```

```
    if ((ipos<0) || (ipos>ilen)) return -1;
    root = RE_PACKET_space;
    initial_mode = 0; if (insens) initial_mode = CIS_MFLAG;
    IT_RE_Clear_Markers(RE_PACKET_space);
    for (:ipos<=ilen:ipos++) {
        if ((RE_PACKET_space-->RE_DOWN ~= NULL) &&
            ((RE_PACKET_space-->RE_DOWN)-->RE_CCLASS == START_RE_CC) &&
            (ipos>0)) { rv = -1; break; }
        if (ipos > 0) IT_RE_EraseConstraints(RE_PACKET_space, initial_mode);
        IT_RE_RewindCount = 0;
        rv = IT_RE_ParseAtPosition(findt, indt, ipos, ilen, RE_PACKET_space, initial_mode);
        if (rv >= 0) break;
    }
    if (rv == -1) {
        root-->RE_DATA1 = -1;
        root-->RE_DATA2 = -1;
    } else {
        root-->RE_DATA1 = ipos;
        root-->RE_DATA2 = ipos+rv;
    }
    return rv;
];
```

§**12. Parse At Position.** `IT_RE_ParseAtPosition(findt, indt, ifrom, ito)` attempts to match text beginning at position `ifrom` in the indexed text `indt` and extending for any length up to position `ito`: it returns the number of characters which were matched (which can legitimately be 0), or -1 if no match could be made. `findt` is the original text of the regular expression in its precompiled form, which we need partly to print good debugging information, but mostly in order to match against a `LITERAL_RE_CC` node.

```
[ IT_RE_ParseAtPosition findt indt ifrom ito token mode_flags
    outcome ipos npos rv i ch edge rewind_this;
    if (ifrom > ito) return -1;
    ipos = ifrom;
    .Rewind;
    while (token ~= NULL) {
        outcome = false;
        if (IT_RE_Trace) {
            print "Matching at ", ipos, ": ";
            IT_RE_DebugNode(token, findt, true);
        }
        if (ipos<ito) ch = BlkValueRead(indt, ipos); else ch = 0;
        token-->RE_MODES = mode_flags; ! Save in case of backtrack
        switch (token-->RE_CCLASS) {
            ! Should never happen
            CHOICE_RE_CC: return "internal error";
            ! Mode switches
            SENSITIVITY_RE_CC:
                if (token-->RE_PAR1) mode_flags = mode_flags | CIS_MFLAG;
                else mode_flags = mode_flags & (~CIS_MFLAG);
```

```
    outcome = true;
! Zero-length positional markers
ALWAYS_RE_CC:
    outcome = true;
NEVER_RE_CC:
START_RE_CC:
    if (ipos == 0) outcome = true;
END_RE_CC:
    if (BlkValueRead(indt, ipos) == 0) outcome = true;
BOUNDARY_RE_CC:
    rv = 0;
    if (BlkValueRead(indt, ipos) == 0 or 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') rv++;
    if (ipos == 0) ch = 0;
    else ch = BlkValueRead(indt, ipos-1);
    if (ch == 0 or 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') rv++;
    if (rv == 1) outcome = true;
NONBOUNDARY_RE_CC:
    rv = 0;
    if (BlkValueRead(indt, ipos) == 0 or 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') rv++;
    if (ipos == 0) ch = 0;
    else ch = BlkValueRead(indt, ipos-1);
    if (ch == 0 or 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') rv++;
    if (rv ~= 1) outcome = true;
! Control constructs
IF_RE_CC:
    i = token-->RE_PAR1; ch = false;
    if (IT_RE_Trace) {
        print "Trying conditional from ", ipos, ": ";
        IT_RE_DebugNode(token, findt, true);
    }
    if (i >= 1) {
        if ((i<RE_Subexpressions-->10) &&
            ((RE_Subexpressions-->i)-->RE_DATA1 >= 0)) ch = true;
    } else {
        rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito,
            (token-->RE_DOWN)-->RE_DOWN, mode_flags);
        if (rv >= 0) ch = true;
    }
    if (IT_RE_Trace) {
        print "Condition found to be ", ch, "^";
```

```
    }
    if (ch) {
        rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito,
            ((token-->RE_DOWN)-->RE_NEXT)-->RE_DOWN, mode_flags);
        !print "Then clause returned ", rv, "^";
    } else {
        if ((((token-->RE_DOWN)-->RE_NEXT)-->RE_NEXT) == NULL)
            rv = 0; ! The empty else clause matches
        else rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito,
            (((token-->RE_DOWN)-->RE_NEXT)-->RE_NEXT)-->RE_DOWN, mode_flags);
        !print "Else clause returned ", rv, "^";
    }
    if (rv >= 0) {
        outcome = true;
        ipos = ipos + rv;
    }
DISJUNCTION_RE_CC:
    if (IT_RE_Trace) {
        print "Trying disjunction from ", ipos, ": ";
        IT_RE_DebugNode(token, findt, true);
    }
    for (ch = token-->RE_DOWN: ch ~= NULL: ch = ch-->RE_NEXT) {
        if (ch-->RE_PAR1 <= token-->RE_CONSTRAINT) continue;
        if (IT_RE_Trace) {
            print "Trying choice at ", ipos, ": ";
            IT_RE_DebugNode(ch, findt, true);
        }
        rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito,
            ch-->RE_DOWN, mode_flags);
        if (rv >= 0) {
            token-->RE_DATA1 = ipos; ! Where match was made
            token-->RE_DATA2 = ch-->RE_PAR1; ! Option taken
            ipos = ipos + rv;
            outcome = true;
            if (IT_RE_Trace) {
                print "Choice worked with width ", rv, ": ";
                IT_RE_DebugNode(ch, findt, true);
            }
            break;
        } else {
            if (mode_flags & ACCUM_MFLAG == false)
                IT_RE_FailSubexpressions(ch-->RE_DOWN);
        }
    }
    if (outcome == false) {
        if (IT_RE_Trace) {
            print "Failed disjunction from ", ipos, ": ";
            IT_RE_DebugNode(token, findt, true);
        }
        token-->RE_DATA1 = ipos; ! Where match was tried
        token-->RE_DATA2 = -1; ! No option was taken
    }
SUBEXP_RE_CC:
```

```
    if (token-->RE_PAR2 == 1 or 2) {
        npos = ipos - token-->RE_PAR3;
        if (npos<0) rv = -1; ! Lookbehind fails: nothing behind
        else rv = IT_RE_ParseAtPosition(findt, indt, npos, ito, token-->RE_DOWN,
            mode_flags);
    } else {
        switch (token-->RE_PAR3) {
            0: rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito, token-->RE_DOWN,
                mode_flags);
            1: rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito, token-->RE_DOWN,
                mode_flags & (~CIS_MFLAG));
            2: rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito, token-->RE_DOWN,
                mode_flags | CIS_MFLAG);
        }
    }
    npos = ipos;
    if (rv >= 0) npos = ipos + rv;
    switch (token-->RE_PAR2) {
        1: if (rv >= 0) rv = 0;
        2: if (rv >= 0) rv = -1; else rv = 0;
    }
    if (rv >= 0) {
        token-->RE_DATA1 = ipos;
        ipos = ipos + rv;
        token-->RE_DATA2 = npos;
        outcome = true;
    } else {
        if (mode_flags & ACCUM_MFLAG == false) {
            token-->RE_DATA1 = -1;
            token-->RE_DATA2 = -1;
        }
    }
    if (token-->RE_PAR2 == 2) IT_RE_FailSubexpressions(token, true);
QUANTIFIER_RE_CC:
    token-->RE_DATA1 = ipos;
    if ((token-->RE_DOWN)-->RE_CCLASS == SUBEXP_RE_CC) {
        (token-->RE_DOWN)-->RE_CACHE1 = -1;
        (token-->RE_DOWN)-->RE_CACHE2 = -1;
    }
    if (IT_RE_Trace) {
        print "Trying quantifier from ", ipos, ": ";
        IT_RE_DebugNode(token, findt, true);
    }
    if (token-->RE_PAR3 == false) { ! Greedy quantifier
        !edge = ito; if (token-->RE_CONSTRAINT >= 0) edge = token-->RE_CONSTRAINT;
        edge = token-->RE_PAR2;
        if (token-->RE_CONSTRAINT >= 0) edge = token-->RE_CONSTRAINT;
        rv = -1;
        for (i=0, npos=ipos: i<edge: i++) {
            if (IT_RE_Trace) {
                print "Trying quant rep ", i+1, " at ", npos, ": ";
                IT_RE_DebugNode(token, findt, true);
            }
```

```
            rv = IT_RE_ParseAtPosition(findt, indt, npos, ito, token-->RE_DOWN,
                mode_flags | ACCUM_MFLAG);
            if (rv < 0) break;
            if ((token-->RE_DOWN)-->RE_CCLASS == SUBEXP_RE_CC) {
                (token-->RE_DOWN)-->RE_CACHE1 = (token-->RE_DOWN)-->RE_DATA1;
                (token-->RE_DOWN)-->RE_CACHE2 = (token-->RE_DOWN)-->RE_DATA2;
            }
            if ((rv == 0) && (token-->RE_PAR2 == 30000) && (i>=1)) { i++; break; }
            npos = npos + rv;
        }
        if ((i >= token-->RE_PAR1) && (i <= token-->RE_PAR2))
            outcome = true;
    } else { ! Lazy quantifier
        edge = token-->RE_PAR1;
        if (token-->RE_CONSTRAINT > edge) edge = token-->RE_CONSTRAINT;
        for (i=0, npos=ipos: (npos<ito) && (i < token-->RE_PAR2): i++) {
            if (i >= edge) break;
            if (IT_RE_Trace) {
                print "Trying quant rep ", i+1, " at ", npos, ": ";
                IT_RE_DebugNode(token, findt, true);
            }
            rv = IT_RE_ParseAtPosition(findt, indt, npos, ito, token-->RE_DOWN,
                mode_flags | ACCUM_MFLAG);
            if (rv < 0) break;
            if ((token-->RE_DOWN)-->RE_CCLASS == SUBEXP_RE_CC) {
                (token-->RE_DOWN)-->RE_CACHE1 = (token-->RE_DOWN)-->RE_DATA1;
                (token-->RE_DOWN)-->RE_CACHE2 = (token-->RE_DOWN)-->RE_DATA2;
            }
            if ((rv == 0) && (token-->RE_PAR2 == 30000) && (i>=1)) { i++; break; }
            npos = npos + rv;
        }
        if ((i >= edge) && (i <= token-->RE_PAR2))
            outcome = true;
    }
    if (outcome) {
        if (token-->RE_PAR3 == false) { ! Greedy quantifier
            if (i > token-->RE_PAR1) { ! I.e., if we have been greedy
                token-->RE_DATA2 = i-1; ! And its edge limitation
            } else {
                token-->RE_DATA2 = -1;
            }
        } else { ! Lazy quantifier
            if (i < token-->RE_PAR2) { ! I.e., if we have been lazy
                token-->RE_DATA2 = i+1; ! And its edge limitation
            } else {
                token-->RE_DATA2 = -1;
            }
        }
        ipos = npos;
        if ((i == 0) && (mode_flags & ACCUM_MFLAG == false))
            IT_RE_FailSubexpressions(token-->RE_DOWN);
        if ((token-->RE_DOWN)-->RE_CCLASS == SUBEXP_RE_CC) {
            (token-->RE_DOWN)-->RE_DATA1 = (token-->RE_DOWN)-->RE_CACHE1;
```

```
            (token-->RE_DOWN)-->RE_DATA2 = (token-->RE_DOWN)-->RE_CACHE2;
        }
        if (IT_RE_Trace) {
            print "Successful quant reps ", i, ": ";
            IT_RE_DebugNode(token, findt, true);
        }
    } else {
        !token-->RE_DATA2 = -1;
        if (mode_flags & ACCUM_MFLAG == false)
            IT_RE_FailSubexpressions(token-->RE_DOWN);
        if (IT_RE_Trace) {
            print "Failed quant reps ", i, ": ";
            IT_RE_DebugNode(token, findt, true);
        }
    }
! Character classes
NOTHING_RE_CC: ;
ANYTHING_RE_CC: if (ch) outcome = true; ipos++;
WHITESPACE_RE_CC:
    if (ch == 10 or 13 or 32 or 9) { outcome = true; ipos++; }
NONWHITESPACE_RE_CC:
    if ((ch) && (ch ~= 10 or 13 or 32 or 9)) { outcome = true; ipos++; }
PUNCTUATION_RE_CC:
    if (ch == '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') { outcome = true; ipos++; }
NONPUNCTUATION_RE_CC:
    if ((ch) && (ch ~= '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}')) { outcome = true; ipos++; }
WORD_RE_CC:
    if ((ch) && (ch ~= 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}')) { outcome = true; ipos++; }
NONWORD_RE_CC:
    if (ch == 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') { outcome = true; ipos++; }
DIGIT_RE_CC:
    if (ch == '0' or '1' or '2' or '3' or '4'
        or '5' or '6' or '7' or '8' or '9') { outcome = true; ipos++; }
NONDIGIT_RE_CC:
    if ((ch) && (ch ~= '0' or '1' or '2' or '3' or '4'
        or '5' or '6' or '7' or '8' or '9')) { outcome = true; ipos++; }
LCASE_RE_CC:
    if (CharIsOfCase(ch, 0)) { outcome = true; ipos++; }
NONLCASE_RE_CC:
    if ((ch) && (CharIsOfCase(ch, 0) == false)) { outcome = true; ipos++; }
UCASE_RE_CC:
    if (CharIsOfCase(ch, 1)) { outcome = true; ipos++; }
NONUCASE_RE_CC:
```

```
        if ((ch) && (CharIsOfCase(ch, 1) == false)) { outcome = true; ipos++; }
    NEWLINE_RE_CC: if (ch == 10) { outcome = true; ipos++; }
    TAB_RE_CC: if (ch == 9) { outcome = true; ipos++; }
    RANGE_RE_CC:
        if (IT_RE_Range(ch, findt,
            token-->RE_PAR1, token-->RE_PAR2, token-->RE_PAR3, mode_flags & CIS_MFLAG))
            { outcome = true; ipos++; }
    ! Substring matches
    LITERAL_RE_CC:
        rv = IT_RE_MatchSubstring(indt, ipos,
            findt, token-->RE_PAR1, token-->RE_PAR2, mode_flags & CIS_MFLAG);
        if (rv >= 0) { ipos = ipos + rv; outcome = true; }
    VARIABLE_RE_CC:
        i = token-->RE_PAR1;
        if ((RE_Subexpressions-->i)-->RE_DATA1 >= 0) {
            rv = IT_RE_MatchSubstring(indt, ipos,
                indt, (RE_Subexpressions-->i)-->RE_DATA1,
                (RE_Subexpressions-->i)-->RE_DATA2, mode_flags & CIS_MFLAG);
            if (rv >= 0) { ipos = ipos + rv; outcome = true; }
        }
        .NeverMatchIncompleteVar;
}
if (outcome == false) {
    if (IT_RE_RewindCount++ >= 10000) {
        if (IT_RE_RewindCount == 10001) {
            style bold; print "OVERFLOW^"; style roman;
        }
        return -1;
    }
    if (IT_RE_Trace) {
        print "Rewind sought from failure at pos ", ipos, " with: ";
            IT_RE_DebugNode(token, findt, true);
    }
    if ((token-->RE_CCLASS == QUANTIFIER_RE_CC) &&
        (IT_RE_SeekBacktrack(token-->RE_DOWN, findt, false, ito, false)))
        jump RewindFound;
    if (mode_flags & ACCUM_MFLAG == false) IT_RE_FailSubexpressions(token);
    token = token-->RE_PREVIOUS;
    while (token ~= NULL) {
        if (IT_RE_SeekBacktrack(token, findt, true, ito, false)) {
            .RewindFound;
            ipos = token-->RE_DATA1;
            mode_flags = token-->RE_MODES;
            if (mode_flags & ACCUM_MFLAG == false)
                IT_RE_FailSubexpressions(token, true);
            if (ipos == -1)
                IT_RE_DebugTree(findt, true);
            if (IT_RE_Trace) {
                print "^[", ifrom, ",", ito, "] rewinding to ", ipos, " at ";
                IT_RE_DebugNode(token, findt, true);
            }
            jump Rewind;
        }
```

```
                token = token-->RE_PREVIOUS;
            }
            if (IT_RE_Trace)
                print "^Rewind impossible^";
            return -1;
        }
        token = token-->RE_NEXT;
    }
    return ipos - ifrom;
];
```

§**13. Backtracking.** It would be very straightforward to match regular expressions with the above recursive code if, for every node, there were a fixed number of characters (depending on the node) such that there would either be a match eating that many characters, or else no match at all. If that were true, we could simply march through the text matching until we could match no more, and although some nodes might have ambiguous readings, we could always match the first possibility which worked. There would never be any need to retreat.

Well, in fact that happy state does apply to a surprising number of nodes, and some quite complicated regular expressions can be made which use only them: `<abc>{2}\d\d\1`, for instance, matches a sequence of exactly 6 characters or else fails to match altogether, and there is never any need to backtrack. One reason why backtracking is a fairly good algorithm in practice is that these "good" cases occur fairly often, in subexpressions if not in the entire expression, and the simple method above disposes of them efficiently.

But in an expression like `ab+bb`, there is no alternative to backtracking if we are going to try to match the nodes from left to right: we match the "a", then we match as many "b"s as we can – but then we find that we have to match "bb", and this is necessarily impossible because we have just eaten all of the "b"s available. We therefore backtrack one node to the `b+` and try again. We obviously can't literally try again because that would give the same result: instead we impose a constraint. Suppose it previously matched a row of 23 letter "b"s, so that the quantifier `+` resulted in a multiplicity of 23. We then constrain the node and in effect consider it to be `b{1,22}`, that is, to match at least 1 and at most 22 letter "b"s. That still won't work, as it happens, so we backtrack again with a constraint tightened to make it `b{1,21}`, and now the match occurs as we would hope. When the expression becomes more complex, backtracking becomes a longer-distance, recursive procedure – we have to exhaust all possibilities of a more recent node before tracking back to one from longer ago. (This is why the worst test cases are those which entice us into a long, long series of matches only to find that a guess made right back at the start was ill-fated.)

Rather than describing `IT_RE_SeekBacktrack` in detail here, it is probably more useful to suggest that the reader observe it in action by setting `IT_RE_Trace` and trying a few regular expressions.

```
[ IT_RE_SeekBacktrack token findt downwards ito report_only
    untried;
    for (: token ~= NULL: token = token-->RE_NEXT) {
        if ((IT_RE_Trace) && (report_only == false)) {
            print "Scan for rewind: ";
            IT_RE_DebugNode(token, findt, true);
        }
        if ((token-->RE_CCLASS == SUBEXP_RE_CC) &&
            (token-->RE_PAR2 == 1 or 2 or 4)) {
            if (downwards) rfalse;
            continue;
        }
        if (token-->RE_DOWN ~= NULL) {
            if ((IT_RE_Trace) && (report_only == false)) print "Descend^";
            if (IT_RE_SeekBacktrack(token-->RE_DOWN, findt, false, ito, report_only)) rtrue;
```

```
        }
        untried = false;
        switch (token-->RE_CCLASS) {
            DISJUNCTION_RE_CC:
                if ((token-->RE_DATA2 >= 1) &&
                    (token-->RE_DATA2 < token-->RE_PAR1) &&
                    (token-->RE_CONSTRAINT < token-->RE_PAR1)) { ! Matched but earlier than last
                    if (report_only) rtrue;
                    if (token-->RE_CONSTRAINT == -1)
                        token-->RE_CONSTRAINT = 1;
                    else
                        (token-->RE_CONSTRAINT)++;
                    untried = true;
                }
            QUANTIFIER_RE_CC:
                if (token-->RE_CONSTRAINT ~= -2) {
                    if ((IT_RE_Trace) && (report_only == false)) {
                        print "Quant with cons not -2: ";
                        IT_RE_DebugNode(token, findt, true);
                    }
                    if (token-->RE_DATA2 >= 0) {
                        if (report_only) rtrue;
                        token-->RE_CONSTRAINT = token-->RE_DATA2;
                        untried = true;
                    }
                }
        }
        if (untried) {
            if (IT_RE_Trace) {
                print "Grounds for rewind at: ";
                IT_RE_DebugNode(token, findt, true);
            }
            IT_RE_EraseConstraints(token-->RE_NEXT);
            IT_RE_EraseConstraints(token-->RE_DOWN);
            rtrue;
        }
        if (downwards) rfalse;
    }
    rfalse;
];
```

§**14. Fail Subexpressions.** Here, an attempt to make a complicated match against the node in `token` has failed: that means that any subexpressions which were matched in the course of the attempt must also in retrospect be considered unmatched. So we work down through the subtree at `token` and empty any markers for subexpressions, which in effect clears their backslash variables – this is important as, otherwise, the contents left over could cause the alternative reading of the `token` to be misparsed if it refers to the backslash variables in question. (If you think nobody would ever be crazy enough to write a regular expression like that, you haven't see Perl's test suite.)

If the `downwards` flag is clear, it not only invalidates subexpression matches below the node but also to the right of the node – this is useful for a backtrack which runs back quite some distance.

```
[ IT_RE_FailSubexpressions token downwards;
    for (: token ~= NULL: token = token-->RE_NEXT) {
        if (token-->RE_DOWN ~= NULL) IT_RE_FailSubexpressions(token-->RE_DOWN);
        if (token-->RE_CCLASS == SUBEXP_RE_CC) {
            token-->RE_DATA1 = -1;
            token-->RE_DATA2 = -1;
        }
        if (downwards) break;
    }
];
```

§**15. Erasing Constraints.** As explained above, temporary constraints are placed on some nodes when we are backtracking to test possible cases. When we do backtrack, though, it's important to lift any constraints left over from the previous attempt to parse material which is part of or subsequent to the token whose match attempt has been abandoned.

```
[ IT_RE_EraseConstraints token;
    while (token ~= NULL) {
        switch (token-->RE_CCLASS) {
            DISJUNCTION_RE_CC: token-->RE_CONSTRAINT = -1;
            QUANTIFIER_RE_CC: token-->RE_CONSTRAINT = -1;
        }
        if (token-->RE_DOWN) IT_RE_EraseConstraints(token-->RE_DOWN);
        token = token-->RE_NEXT;
    }
];
```

§**16. Matching Literal Text.** Here we attempt to make a match of the substring of the indexed text `mindt` which runs from character `mfrom` to character `mto`, looking for it at the given position `ipos` in the source text `indt`.

```
[ IT_RE_MatchSubstring indt ipos mindt mfrom mto insens
    i ch;
    if (mfrom < 0) return 0;
    if (insens)
        for (i=mfrom:i<mto:i++) {
            ch = BlkValueRead(mindt, i);
            if (BlkValueRead(indt, ipos++) ~= ch or IT_RevCase(ch))
                return -1;
        }
    else
        for (i=mfrom:i<mto:i++)
            if (BlkValueRead(indt, ipos++) ~= BlkValueRead(mindt, i))
                return -1;
    return mto-mfrom;
];
```

§**17. Matching Character Range.** Suppose that a character range is stored in `findt` between the character positions `rf` and `rt`. Then `IT_RE_Range(ch, findt, rf, rt, negate, insens)` tests whether a given character `ch` lies within that character range, negating the outcome if `negate` is set, and performing comparisons case insensitively if `insens` is set.

```
[ IT_RE_Range ch findt rf rt negate insens
    i chm upper crev;
    if (ch == 0) rfalse;
    if (negate == true) {
        if (IT_RE_Range(ch, findt, rf, rt, false, insens)) rfalse;
        rtrue;
    }
    for (i=rf: i<rt: i++) {
        chm = BlkValueRead(findt, i);
        if ((chm == '\') && (i+1<rt)) {
            chm = BlkValueRead(findt, ++i);
            switch (chm) {
                's':
                    if (ch == 10 or 13 or 32 or 9) rtrue;
                'S':
                    if ((ch) && (ch ~= 10 or 13 or 32 or 9)) rtrue;
                'p':
                    if (ch == '.' or ',' or '!' or '?'
                        or '-' or '/' or '"' or ':' or ';'
                        or '(' or ')' or '[' or ']' or '{' or '}') rtrue;
                'P':
                    if ((ch) && (ch ~= '.' or ',' or '!' or '?'
                        or '-' or '/' or '"' or ':' or ';'
                        or '(' or ')' or '[' or ']' or '{' or '})) rtrue;
                'w':
                    if ((ch) && (ch ~= 10 or 13 or 32 or 9
                        or '.' or ',' or '!' or '?'
                        or '-' or '/' or '"' or ':' or ';'
```

```
                    or '(' or ')' or '[' or ']' or '{' or '}')) rtrue;
            'W':
                if (ch == 10 or 13 or 32 or 9
                    or '.' or ',' or '!' or '?'
                    or '-' or '/' or '"' or ':' or ';'
                    or '(' or ')' or '[' or ']' or '{' or '}') rtrue;
            'd':
                if (ch == '0' or '1' or '2' or '3' or '4'
                    or '5' or '6' or '7' or '8' or '9') rtrue;
            'D':
                if ((ch) && (ch ~= '0' or '1' or '2' or '3' or '4'
                    or '5' or '6' or '7' or '8' or '9')) rtrue;
            'l': if (CharIsOfCase(ch, 0)) rtrue;
            'L': if (CharIsOfCase(ch, 0) == false) rtrue;
            'u': if (CharIsOfCase(ch, 1)) rtrue;
            'U': if (CharIsOfCase(ch, 1) == false) rtrue;
            'n': if (ch == 10) rtrue;
            't': if (ch == 9) rtrue;
        }
    }
    if ((i+2<rt) && (BlkValueRead(findt, i+1) == '-')) {
        upper = BlkValueRead(findt, i+2);
        if ((ch >= chm) && (ch <= upper)) rtrue;
        if (insens) {
            crev = IT_RevCase(ch);
            if ((crev >= chm) && (crev <= upper)) rtrue;
        }
        i=i+2;
    } else {
        if (chm == ch) rtrue;
        if ((insens) && (chm == IT_RevCase(ch))) rtrue;
    }
  }
  rfalse;
];
```

§**18. Search And Replace.** And finally, last but not least: the routine which searches an indexed text `indt` trying to match it against `findt`. If `findtype` is set to `REGEXP_BLOB` then `findt` is expected to be a regular expression such as `ab+(c*de)?`, whereas if `findtype` is `CHR_BLOB` then it is expected only to be a simple string of characters taken literally, such as `frog`.

Each match found is replaced with the indexed text in `rindt`, except that if the blob type is `REGEXP_BLOB` then we recognise a few syntaxes as special: for instance, `\2` expands to the value of subexpression 2 as it was matched – see *Writing with Inform* for details.

The optional argument `insens` is a flag which, if set, causes the matching to be done case insensitively; the optional argument `exactly`, if set, causes the matching to work only if the entire `indt` is matched. (This is not especially useful with regular expressions, because the effect can equally be achieved by turning `ab+c`, say, into `^ab+c$`, but it is indeed useful where the blob type is `CHR_BLOB`.)

For an explanation of the use of the word "blob", see "IndexedText.i6t".

```
[ IT_Replace_RE findtype indt findt rindt insens exactly
    cindt csize ilen i cl mpos cpos ch chm;
    ilen = IT_CharacterLength(indt);
```

```
IT_RE_Err = 0;
switch (findtype) {
    REGEXP_BLOB: i = IT_RE_CompileTree(findt, exactly);
    CHR_BLOB: i = IT_CHR_CompileTree(findt, exactly);
    default: "*** bad findtype ***";
}
if ((i<0) || (i>RE_MAX_PACKETS)) {
    IT_RE_Err = i;
    print "*** Regular expression error: ", (string) IT_RE_Err, " ***^";
    RunTimeProblem(RTP_REGEXPSYNTAXERROR);
    return 0;
}
if (IT_RE_Trace) {
    IT_RE_DebugTree(findt);
    print "(compiled to ", i, " packets)^";
}
if (findtype == REGEXP_BLOB) IT_RE_EmptyMatchVars();
mpos = 0; chm = 0; cpos = 0;
while (IT_RE_Parse(findt, indt, mpos, insens) >= 0) {
    chm++;
    if (IT_RE_Trace) {
        print "^*** Match ", chm, " found (", RE_PACKET_space-->RE_DATA1, ",",
            RE_PACKET_space-->RE_DATA2, "): ";
        if (RE_PACKET_space-->RE_DATA1 == RE_PACKET_space-->RE_DATA2) {
            print "<empty>";
        }
        for (i=RE_PACKET_space-->RE_DATA1:i<RE_PACKET_space-->RE_DATA2:i++) {
            print (char) BlkValueRead(indt, i);
        }
        print " ***^";
    }
    if (rindt == 0) break; ! Accept only one match, replace nothing
    if (rindt ~= 0 or 1) {
        if (chm == 1) {
            cindt = BlkValueCreate(INDEXED_TEXT_TY);
            csize = BlkValueExtent(cindt);
        }
        for (i=cpos:i<RE_PACKET_space-->RE_DATA1:i++) {
            ch = BlkValueRead(indt, i);
            if (cl+1 >= csize) {
                if (BlkValueSetExtent(cindt, 2*cl, 7) == false) break;
                csize = BlkValueExtent(cindt);
            }
            BlkValueWrite(cindt, cl++, ch);
        }
        BlkValueWrite(cindt, cl, 0);
        IT_Concatenate(cindt, rindt, findtype, indt);
        csize = BlkValueExtent(cindt);
        cl = IT_CharacterLength(cindt);
    }
    mpos = RE_PACKET_space-->RE_DATA2; cpos = mpos;
```

```
        if (RE_PACKET_space-->RE_DATA1 == RE_PACKET_space-->RE_DATA2)
            mpos++;
        if (IT_RE_Trace) {
            if (chm == 100) { ! Purely to keep the output from being excessive
                print "(Stopping after 100 matches.)^"; break;
            }
        }
    }
    if (chm > 0) {
        if (rindt ~= 0 or 1) {
            for (i=cpos:i<ilen:i++) {
                ch = BlkValueRead(indt, i);
                if (cl+1 >= csize) {
                    if (BlkValueSetExtent(cindt, 2*cl, 8) == false) break;
                    csize = BlkValueExtent(cindt);
                }
                BlkValueWrite(cindt, cl++, ch);
            }
        }
        if (findtype == REGEXP_BLOB) {
            IT_RE_CreateMatchVars(indt);
            if (IT_RE_Trace)
                IT_RE_DebugMatchVars(indt);
        }
        if (rindt ~= 0 or 1) {
            BlkValueWrite(cindt, cl, 0);
            BlkValueCopy(indt, cindt);
            BlkFree(cindt);
        }
    }
    return chm;
];
```

§**19. Concatenation.** See the corresponding routine in "IndexedText.i6t": this is a variation which handles the special syntaxes used in search-and-replace.

```
[ IT_RE_Concatenate indt_to indt_from blobtype indt_ref
    pos len ch i tosize x y case;
    if ((indt_to==0) || (BlkType(indt_to) ~= INDEXED_TEXT_TY)) rfalse;
    if ((indt_from==0) || (BlkType(indt_from) ~= INDEXED_TEXT_TY)) return indt_to;
    pos = IT_CharacterLength(indt_to);
    tosize = BlkValueExtent(indt_to);
    len = IT_CharacterLength(indt_from);
    for (i=0:i<len:i++) {
        ch = BlkValueRead(indt_from, i);
        if ((ch == '\') && (i < len-1)) {
            ch = BlkValueRead(indt_from, ++i);
            if (ch == 'n') ch = 10;
            if (ch == 't') ch = 9;
            case = -1;
            if (ch == 'l') case = 0;
            if (ch == 'u') case = 1;
```

```
            if (case >= 0) ch = BlkValueRead(indt_from, ++i);
            if ((ch >= '0') && (ch <= '9')) {
                ch = ch - '0';
                if (ch < RE_Subexpressions-->10) {
                    x = (RE_Subexpressions-->ch)-->RE_DATA1;
                    y = (RE_Subexpressions-->ch)-->RE_DATA2;
                    if (x >= 0) {
                        for (:x<y:x++) {
                            ch = BlkValueRead(indt_ref, x);
                            if (pos+1 >= tosize) {
                                if (BlkValueSetExtent(indt_to, 2*tosize, 11) == false) break;
                                tosize = BlkValueExtent(indt_to);
                            }
                            if (case >= 0)
                                BlkValueWrite(indt_to, pos++, CharToCase(ch, case));
                            else
                                BlkValueWrite(indt_to, pos++, ch);
                        }
                    }
                }
                continue;
            }
        }
        if (pos+1 >= tosize) {
            if (BlkValueSetExtent(indt_to, 2*tosize, 12) == false) break;
            tosize = BlkValueExtent(indt_to);
        }
        BlkValueWrite(indt_to, pos++, ch);
    }
    BlkValueWrite(indt_to, pos, 0);
    return indt_to;
];
```

§**20. Stubs.** This time, there are no stubs: if there are no indexed texts, none of these routines is ever referred to.

```
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```