INTERRUPTABLE TRANSITION NETWORKS

Sergei Nirenburg
Colgate University
Chagit Attiya
Hebrew University of Jerusalem

ABSTRACT

A specialized transition network mechanism, the interruptable transition network (ITN) is used to perform the last of three stages in a multiprocessor syntactic parser. This approach can be seen as an exercise in implementing a parsing procedure of the active chart parser family.

Most of the ATN parser implementations use the left-to-right top-down chronological backtracking control structure (cf. Bates, 1978 for discussion). The control strategies of the active chart type permit a blend of bottom-up and top-down parsing at the expense of time and space overhead (cf. Kaplan, 1973). The environment in which the interruptable transition network (ITN) has been implemented is not similar to that of a typical ATN model. Nor is it a straightforward implementation of an active chart. ITN is responsible for one stage in a multiprocessor parsing technique described in Lozinskii & Nirenburg, (1982a and b), where parsing is performed in essentially the bottom-up fashion in parallel by a set of relatively small and "dumb" processing units running identical software. The process involves three stages: (a) producing the candidate strings of preterminal category symbols; (b) determining the positions in this string at which higher-level constituents start and (c) determining the closing boundaries of these constituents.

Each of the processors allocated to the first stage obtains the set of all syntactic readings of one word in the input string. Using a table grammar, the processors then choose a subset of the word's readings to ensure compatibility with similar subsets generated by this processor's right and left neighbor.

Stage 2 uses the results of stage 1 and a different tabular grammar to establish the left ("opening") boundaries for composite sentence constituents, such as NP or PP. The output of this stage assumes the form of a string of triads ⟨label x y⟩, where label belongs to the vocabulary of constituent types. In our implementation this set includes S, NP, VP, PP, NP& (the "virtual" NP), Del (the delimiter), etc. x and y are the left and the right indices of the boundaries of these constituents in the input string. They mark the points at which parentheses are to be opened (x) and closed (y) in the tree representation. The values x and y relate to positions of words in the initial input string. For example, the sentence (1) will be processed at stage 2 into the string (2). The '?' in (2) stand for unknown coordinates y.

(1) The very big brick building that sits
1 2 3 4 5 6 7
on the hill belongs to the university.
8 9 10 11 12 13 14

(2) (s 1 ?)(np 1 ?)(s 6 ?)(np& 6 6)
(vp 7 ?)(pp 8 ?)(np 9 ?)(vp 11 ?)
(pp 12 ?)(np 13 ?)

It is at this point that the interruptable transition network starts its work of finding the unknown boundary coordinates and thus determining the upper levels of the parse tree.

An input string n triads long will be allocated n identical processors. Initially the chunk of every participating processor will be one triad long. After these processors finish with their chunks (either succeeding or failing to find the missing coordinate) a "change of levels" interrupt occurs: the size of the chunks is doubled and the number of active processors halved. These latter continue the scanning of the ITN from the point they were interrupted taking as input what was formerly the chunk of their right neighbor. Note that all constituents already closed in that chunk are transparent to the current processor and already closed in that chunk are transparent to the current processor and are not rescanned. The number of active processors steadily reduces during parsing. The choice of processors that are to remain active is made with the help of tne Pyramid protocol (cf. Lozinskii & Nirenburg, 1982). The processors released

after each "layout" are returned to the system pool of available resources. At the top level in the pyramid only one processor will remain. The status of such a processor is declared final, and this triggers the wrap-up operations and the construction of output. The wrap-up uses the original string of words and the appropriate string of preterminal symbols obtained at stage 1 together with the results of stage 3 to build the parse tree.

ITN can start processing at an arbitrary position in the input string, not necessarily at the beginning of a sentence. Therefore, we introduce an additional subnetwork, "initial", used for handling control flow among the other subnetworks.

The list of "closed" constituents obtained through ITN-based parsing of string (2) can be found in (3), while (4) is the output of ITN processing of (3).

(3) (s 1 14)(np 1 10)(s 6 10)(np& 6 6)
(vp 7 10)(pp 8 10)(np 9 10)(vp 11 14)
(pp 12 14)(np 13 14)

(4) (s(np(s(np&)(vp(pp(np)))))(vp(pp)))

3. An ITN Interpreter.

The interpreter was designed for a parallel processing system. This goal compelled us to use a program environment somewhat different from the usual practice of writing ATN interpreters. Our interpreter can, however, be used to interpret both ITNs and ATNs.

A new type of arc was introduced: the interrupt arc INTR. The interrupt arc is a way out of a network state additional to the regular POP. It gives the process the opportunity to resume from the very point where the interrupt had been called, but at a later stage (this mechanism is rather similar to the detach-type commands in programming languages which support coroutines, such as, for instance, SIMULA). Thus, the interpreter must be able to suspend processing after trying to proceed through any arc in a state and to resume processing later in that very state, from the arc immediately following the interrupt arc. For example, if INTR is the fourth of seven arcs in a state, the work resumes from the fifth arc in this state. This is implemented with a stack in which the transitions in the net are recorded. The PUSH and POP arcs are also implemented through this stack and not through the recursion handling mechanisms built into Lisp.

Since it is never known to any processor whether it will be active at the next stage, it is necessary that the information it obtained be saved in a place where another processor will be able to find it. Unlike the standard ATN parsers (which return the parse tree as the value of the parsing function), the ITN parser records the results in a special working area (see discussion below).

Implementation

The ITN interpreter was implemented in YLISP, the dialect of LISP developed at the Hebrew University of Jerusalem. A special scheduler routine for simulating parallel processes on a VAX 11/780 was written by Jacob Levy. The interpreter also uses the pyramid protocol program by Shmuel Bahr.

In what follows we will describe the organization of the stack, the working area, and the program itself.

a) The stack. The item to be stacked must describe a position in the network. An item is pushed onto the stack every time a PUSH or an INTR arc is traversed. Every time a POP arc is traversed or a return from an interrupt occurs one item is popped. The stack item consists of: 1) names and values of the current network registers; 2) the remainder of the arcs in the state (after the PUSH or the INTR traversed); 3) the actions of the PUSH arc traversed; 4) the name of the current network (i.e. that of the latter's initial state); 5) the value of the input pointer (for the case of a PUSH failure).

The working area is used for two purposes: to support message passing between the processors and to hold the findings. The working area is organized as an array, R, that holds a doubly linked list used to construct the output tree. The actions defined on the working area are: a) initialization (procedure init-input): every cell R[i] in R obtains a token from input, while the links R[i].[next-index] and R[i].[previous-index] obtain the values i+1 and i-1, respectively; b) CLOSE, the tool for delimiting subtrees in the input string;

The array R is used in parallel by a number of processors. At every level of processing the active processors' chunks cover the array R. This arrangement does not corrupt the parallel character of the process, since no processor actually seeks information from the chunks other than its own.

The main function of the interpreter is called itn. It obtains the stack containing the history of processing. If an interrupt is encountered, the function returns the stack with new history, to be used for invoking this function again, by the pyramid protocol.

If a call to itn is a return from the interrupt status, then a stack item is popped (it corresponds to the last state entered during the previous run). If the function call is the initial one, we start to scan the network from the first state of the "initial" subnetwork.

At this stage we already know which state of which network fragment we are in. Moreover, we even know the path through the states and fragments we took in order to reach this state and the exact arc in this state from which we have to start processing. So, we execute the test on the current arc. If the test succeeds we perform branching on the arc name.

The INTR arc has the following syntax: (INTR⟨dummy⟩⟨test⟩⟨action⟩*).

The current state is stacked and the procedure is exited returning the stack as the value. ⟨dummy⟩ was inserted simply to preserve the usual convention of situating the test in the third slot in an arc.

The ABORT arc has the syntax (ABORT⟨message⟩⟨test⟩).

When we encounter an error and it becomes clear that the input string is illegal, we want to be able to stop processing immediately and print a diagnostic message.

The actions on the stack involve the movement of an item to and from the stack. The stack item is the quantum value that can be pushed and popped, that is no part of the item is accessed separately from the rest of the values in it. The functions managing the stack are push-on-stack and pop-from-stack.

The push-on-stack is called whenever a PUSH or an INTR arc is traversed. The pop-from-stack is called, first, when the POP arc is traversed and, second, when the process resumes after return from an interrupt.

The close action is performed when we find a boundary for a certain subtree for which the opposite boundary is already known (in our case the boundary that is found is always the right boundary, y). Close performs two tasks: first, it inserts the numeric value for y and, second, it declares the newly built subtree a new token in the input string. For example, if the input string had been

```
⟨s 1 ?⟩⟨np 1 ?⟩⟨vp 4 ?⟩⟨np 6 8⟩⟨pp 9 10⟩
   1       2       3       4       5
```

after the action (close 3 10) is performed the input for further processing has the form:

⟨s 1 ?⟩⟨np 1 ?⟩⟨vp 4 10⟩.

The parameters of close are 1) the number of the triad we want to close and 2) the value for which the y in this triad is to be substituted. The default value for the second parameter is the value of the y in the triad current at the moment a call to close is made.

When the processing is parallel, close is applied multiply at every level, which would mean that a higher level processor will obtain prefabricated subtrees as elementary input tokens. This is a major source of the efficiency of multiprocessor parsing.

The ITN in the current implementation is relatively small. A broader implementation will be needed to study the properties of this parsing scheme, including the estimates for its time complexity, and the extendability of the grammar. A comparison should also be made with other multiprocessor parsing schemes, including those that are based not on organizing communication among relatively "dumb" processors running identical software but rather on interaction of highly specialized and "intelligent" processors -- cf., e.g., the word expert parser (Small, 1981).

Acknowledgments. The authors thank E. Lozinskii and Y. Ben Asher for the many discussions of the ideas described in this paper.

Bibliography

Bates, M. (1978), The theory and practice of augmented transition network grammars. In: L. Bolc (ed.), Natural Language Communication with Computers. Berlin: Springer.

Kaplan, R. M. (1973), A general syntactic processor. In R. Rustin (ed.), Natural Language Processing. NY: Academic Press.

Lozinskii, E. L. and S. Nirenburg (1982a). Locality in Natural Language processing. In: R. Trappl (ed.), Cybernetics and Systems Research. Amsterdam: North Holland.

Lozinskii, E. L. and S. Nirenburg (1982b), Parallel processing of natural language. Proceedings of ECAI, Orsay, France.

Small, S. (1981), Viewing word expert parsing as a linguistic theory. Proceedings of IJCAI, Vancouver, B.C..

Appendix A. ITN: the main function of the interruptable transition network interpreter

```
(def itn
 (lambda ( stack )
; stack - current processing stack
(prog (regs curr-state-arcs net-name
       curr-arc $ test arc-name)
; regs - current registers of the network
; curr-state-arcs - list of arcs not yet
; processed in current state
; net-name - name of network being
: processed
; curr-arc - arc in processing
;(all these are pushed on stack when a
; 'push' arc occurs)
; $ - a special register.
; the function first checks if stack is
; nil; if not then this call is a return
; from interrupt previous values must be
; popped from the stack
[cond (stack (seta ec pn nil)
   ;set end-chunk flag to nil
             (pop-from-stack t))
      (t      (set-net 'al]
loop
 [ cond ((null curr-state-arcs)
        (cond((null (pop nil)) (return nil)]
 (set 'curr-arc (setcdr 'curr-state-arcs))
 ( set 'test (*nth curr-arc 3) )
 ( cond ((eval test)
 ;test succeeds - traverse the arc
 ( set 'arc-name (car curr-arc))
 [cond
   ((eq arc-name 'push ) ; PUSH
     (evlist (*nth curr-arc 4))
     (push-on-stack)
     (set-net (cadr curr-arc))
     (go loop))
   ((eq arc-name 'pop )  ; POP
     (evlist (*nthcdr curr-arc 3))
     (cond
      ((null (pop(eval(cadr curr-arc))))
       (return $)))
      (go loop))
   ((eq arc-name 'jump ) ; JUMP
     (evlist (*nthcdr curr-arc 3))
     (set-state (*nth curr-arc 2))
     (go loop))
   ((eq arc-name 'to)    ; TO
     (evlist (*nthcdr curr-arc 3))
     (set-state (*nth curr-arc 2))
     (get-input)
     (go loop))
   ((eq arc-name 'cat)   ; CAT
     (cond ((eq (curr[AB))
                (*nth curr-arc 2))
           (evlist
            (*nthcdr curr-arc 3))))
     (go loop))
   ((eq arc-name 'abort) ; ABORT
     (tpatom (*nth curr-arc 2))
     (return nil))
   ((eq arc-name 'intr) ; INTeRrupt
     (push-on-stack)
     (return stack))
   (t                        ; error
     (tpatom '"illegal arc")
     (return nil))
     ( go loop ]      ; try next arc
```

Appendix B.

A Fragment of an ITN network
(the "initial" and the sentence subnetworks)

```
;Note that "jump" and "to" can be either
;terminal actions on an arc or separate
;arcs
(def-net '(s-place) '(
 (initial
  (pop t (end-of-sent) (close*))
  (intr nil (end-of-chunk)((to initial)))
  (push S (lab s)
   ((setr s-place (inp-pointer)))
   ((jump initial/DEL)))
  (push NP (lab np) nil ((to initial)))
  (push VP (lab vp) nil ((to initial)))
  (push PP (lab pp) nil ((to initial)))
  (cat np& t (to initial))
  (cat del t (to initial)))
 (initial/DEL
  (cat del t (close* (getr s-place))
                     (to initial))
  (to initial t]

(def-net '( vp-place no-pp pp-place
            np-place)
'(
 (S
  (pop t (is-def (Y))(close (inp-pointer)))
  (to S/ t (setr no-pp 0)))

 (S/
  (intr nil (end-of-chunk)((to S/)))
  (push PP (and (lab pp)
                (le (getr no-pp) 2))
           ((and (gt (getr no-pp) 0)
                 (close* (getr pp-place)))
            (setr pp-place (inp-pointer)) )
            ((setr no-pp (addl
                          (getr no-pp)))
            (jump S/)))
  (abort "more than 2 PPs in S" (lab pp) )
  (cat np& t (to S/NP&))
;(s (pp & pp) ..)
  (cat del (gt (getr no-pp) 0)
           (close* pp-place)
           (setr no-pp 1)
           (to S/))
  (abort "DEL cannot appear at
          beginning of sent" (lab del))
  (jump S/NP& t]
 (S/NP&
  (intr nil (end-of-chunk)((to S/NP&)))
  (push NP
```

```
      (lab np)
       ((and
         (getr pp-place)
         (close* (getr pp-place)))
         (setr np-place (inp-pointer)))
       ((to S/NP)))
;here we can allow PPs after an NP!
  (push VP
      (lab vp)
       ((and (getr pp-place)
             (close* (getr pp-place))))
       ((jump S/OUT)))
  (abort "no NP or VP in
          the input sentence" t)
  (jump S/NP t]
 (S/NP
  (abort "not enough VPs in S"
                 (end-of-sent))
  (intr nil (end-of-chunk)((to S/NP)))
  (push VP (lab vp)
           ((setr vp-place (inp-pointer))
 ;if there is a del
           (close* (getr np-place)))
;close the preceding NP
;and everything in it
           ((jump S/VP)))
```

```
;(s .. (np & np) ..)
  (cat del (lab del)
           (close* (getr np-place))
           (to S/NP&))
  (abort "too many NPs before a VP"
                              (lab np]
 (S/VP
  (cat del (lab del)
           (close* (getr vp-place))
           (jump S/VP/DEL))
  (jump S/OUT t]
 (S/VP/DEL

;standing at 'del' and looking ahead
  (abort "del at EOS?"
     (ge (next-one (inp-pointer))
         sent-len))
     ; the above is a test for eos
  (intr nil (null (look-ahead 1))
            ((jump S/VP/DEL)))
  (to S/NP (eq (look-ahead 1) 'vp))
  (jump S/OUT t]
;exit: it must be an s
 (S/OUT
  (pop t (end-of-sent) (close*))
  (pop t t]
```