Parsing as Search:
An Easy-to-Understand RTN Interpreter

Robert M. Harlan
David M. Patrone
St Bonaventure University
St. Bonaventure, NY 14778
rharlan@sbu.edu

Abstract

Developing laboratory assignments on natural language processing in an introductory AI course is difficult. One reason is that the interpreter that applies a grammar to input is often a complex, hard to understand system. One is forced either to treat the interpreter as a black box, restricting students to extending the grammar that it applies, or to spend an inordinate amount of time mastering the interpreter. We present an alternative that enables students to master an interpreter in fairly short order. Approaching parsing as a search problem, an interpreter is presented that is based on a search engine with which students have had previous experience. This paper discusses an interpreter for an RTN grammar. A subsequent paper shows how the interpreter can be modified to handle an ATN grammar. The grammar and interpreter are available via ftp.

Overview

Examining the process of recognizing a subset of sentences of a natural language and translating the meaning of the sentences into commands of some other software system is an important topic in an Artificial Intelligence survey course. We examine the first part of this process, the recognition of sentences.

Natural language parsers capable of recognizing a subset of English consist of two components: a grammar, which formally specifies the structure of the sentences which can be recognized by the system, and an interpreter which applies the grammar to an input sentence and computes its structure according to the grammar.

The first component is relatively easy to cover, at least for the sentences which can be generated by context free grammars. Students can be taught quickly to specify grammars using both rewrite rules and recursive transition networks (RTNs), and then given assignments which require the extension of the grammar to new cases.

The interpreter which applies the grammar to sentences and either accepts or rejects them is another story. For the past two offerings of the AI course, we have used a modified version of Winston's augmented transition tree interpreter [WINST 84]. The interpreter is an elegant system, especially in the use of macros to translate LISP representations of transition nets into LISP functions. The price of the elegance is, however, opacity. The first use of the interpreter in a course completely bogged down in a discussion of the interpreter, and the student lab project, extending the grammar for a SHRDLU-like system, was not assisted by this discussion. The interpreter was treated as a black box the second time. Student projects were somewhat more creative, but we felt that they had learned nothing of the process of applying a grammar to a sentence.

A more severe problem is that the interpreter, which interprets a semantic grammar, is not easily extendable. For example, it would be difficult to add cases where additional information is sought from the user, for a parse is either successful, resulting in a call to problem-solving system for which it is the front end, or a parse fails. No mechanism exists for building a representation of an object independent of the slot reserved for it in the semantic grammar.

The solution proposed here is to scrap the interpreter entirely. Following James Allen's suggestion [ALLEN 87], parsing is treated as a search procedure. A generalized depth-first search engine, used by students in the AI course to solve search problems such as missionaries and cannibals, is used to control the search. In particular, it keeps track of the parse states that are currently active, enables backtracking from blind alleys, and checks for a successful or failed parse. The engine calls an expansion function which generates successor parse states by applying the grammar to the current state of the parse.

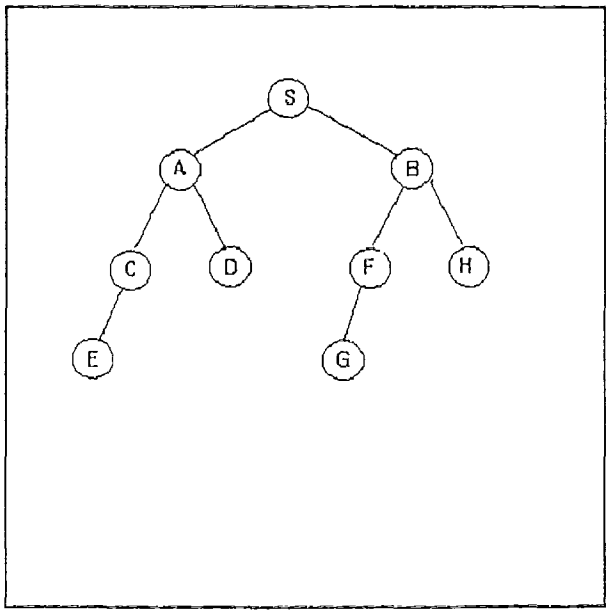

Figure 1
A graphical representation of a problem space.

Because students are familiar with how the search engine controls the search of a problem space, focus is placed on the expansion function which applies the grammar to the sentence and generates successor parse states. As a result, students are able to understand how the syntax of a sentence is computed as well as how to formally specify the grammar used to compute it.

We discuss the search algorithm used to drive the interpreter, Allen's parsing algorithm for an RTN grammar, and our LISP implementation of it. A subsequent paper discusses the extension of the interpreter to augmented transition net grammars.

The Search Algorithm and Implementation

An exhaustive search is the systematic examination of a problem space until either a solution has been found (the goal reached) or no solution is found and no further possibilities remain to be explored. Search begins from some initial state, and on each iteration states that can be reached from the current state are added to the list of states to examine.

A problem space is abstractly illustrated by the graph in Figure 1, where the initial state is represented by node S, the goal state by G, and the intermediate states by the remaining letters. Arcs between nodes indicate that operators exist that enable a transition to be made between the states.

```
(S)
(A B)
(C D B)
(E D B)
(D B)
(B)
(F H)
(G H)
```

Figure 2
Stack during search of graph in figure 1. G is the goal state.

The algorithm for searching the graph is as follows:

1. Initialize the list of states to examine with S

2. While the goal has not been reached and there are still states to examine,
   2a. Generate the states that can be reached from the current state
   2b. Replace the current state with new states

If the list of states is treated as a stack, where the top state on the stack is replaced by the children of the state, the result is a depth first search. The status of the stack during the run of the algorithm is shown in Figure 2. S is replaced by its children, A and B, and A is replaced by its children, C and D.

What is important in this example is how backtracking is handled. The need to backtrack arises when a path leads to a state from which no progress towards a solution can be made, as is illustrated by the path (S A C E). Sufficient information remains in the stack to resume the search at the last branch before the dead end was taken. When A was expanded, both of its children, C and D, were pushed onto the stack. When C leads to the dead end E, it is removed from the stack and the search resumes at D. The path (S A D) fails as well, necessitating a return to B, where a successful path is found.

It is the ability of the search to handle backtracking by retaining unexplored states that will be exploited by our RTN interpreter.

A LISP implementation of the depth first search engine is provided in Figure 3. The stack consists of all currently active states. *Expand*, which is designed to generate the successor states for a particular problem space, takes the first path and returns paths to all states that can be reached from the current state on the path. These are added to the top of the stack. The

```
(defun depthfirst (stack)
  (cond   ;; no more states to examine, return nil
        ((null stack)) nil)
        ;; first state reaches goal, return path to state
        ((goalp (car stack))(car stack)
        (t                ;; add new paths to front of stack
          (append (expand(car stack))(cdr stack)))
  )
)
```

Figure 3
LISP implementation of search

function terminates when either there are no more paths in the stack to examine or the goal has been reached.

The functioning of this search engine is well covered in the unit on search in the AI course. It is modified to illustrate best first, branch and bound, and the A* searches, and is used by students to solve a problem such as missionaries and cannibals.

Parsing as Search

It is a commonplace that most AI problems are search problems. Following Allen, we approach parsing as one as well. In particular, the stack will consist of a set of states representing the current state of a parse. On each iteration the expand function will apply a grammar to the top state in the stack and generate all states that can be reached from that state. Should one of the states lead to a dead end, other paths not tried can be generated from untried states remaining in the stack.

To illustrate the strategy, we use the RTN formalism to specify a grammar for accepting a command, determine the information that must be contained in a state to allow the expand function to generate successor states, and examine how backtracking is handled.

We wish to accept simple commands such as 'pick up the red block', 'pick it up' and 'pick

the red block up'. The grammar which accepts the first command is given in figure 4. Students are asked to modify the grammar to accept the last two commands as an exercise.

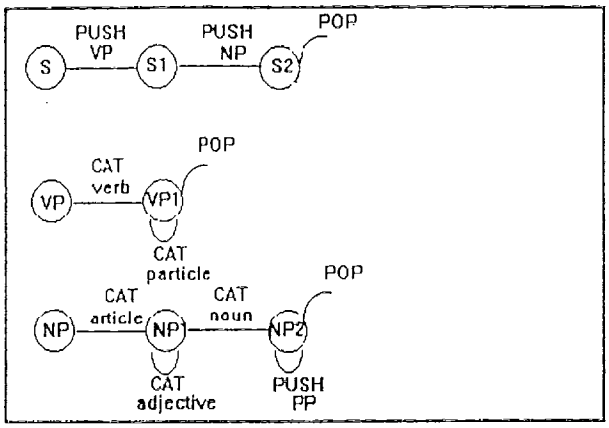

Figure 4
RTN Specification of a grammar.

In this grammar, there are three types of arcs that allow transitions between nodes in the transition nets:

> **CAT arcs,** which test the syntactic category of the next word in the words remaining to be parsed. If the syntactic category of the word matches that required by the arc, the word is removed from the stream and the current node is assigned the next node in the network. An example is the arc between VP and VP1 which is traversed when the next input word has the category verb.
>
> **PUSH arcs,** which require the successful traversal of another network. If the network named in the arc is successfully traversed, the current node is assigned the next node in the network. An example is the arc between S and S1 which is traversed when a VP network is traversed.
>
> **POP arcs,** which signal the successful traversal of the network. If a network pops, the current node is assigned the next node in the parent network's network.

A sentence is accepted if the POP arc of the top-level network S is reached and there are no more remaining words.

The state of the parse is determined by a 3-tuple

(<current node>, <remaining words>, <return points>)

where the current node is the current node in the network being traversed, remaining words is a list of the remaining words in the input stream, and return points is a stack of nodes to return to in previous networks. The expansion function must take the current state and apply the tests corresponding to the arcs emanating from the node of the current state. If a CAT arc is encountered and its condition met, a new state is generated where the current node is set to the next node in the network and the first word in the remaining words list is removed. If a PUSH arc is encountered, the new state is one with the next node in the current network pushed onto the stack of return points and the current node the first node in the network pushed. If a POP arc is encountered, the new state is one in which the current node is the top of the stack of return points and the return points stack is the stack after it is popped. A successful parse is indicated when the parse state has no return points and no remaining words. A failed parse is indicated by a parse state with no return points and some remaining words and no other parse states to which to backtrack.

A trace of the states in the stack of the successful parse of the sentence 'pick up the red block' is provided in figure 5. The name of the arc test used in the transitions is given beside each state.

To illustrate how backtracking is handled, a closer look at figure 5 is required. On the third iteration, two legal transitions from node VP1 are found: the consumption of a particle and remaining in VP1, and a pop, which consumes

(S (pick up the red block) nil)	PUSH VP
(VP (pick up the red block) S1)	CAT verb
(VP1 (up the red block) S1)	CAT particle
(VP1 (the red block) S1)	POP
(S1 (the red block) nil)	PUSH NP
(NP (the red block) S2)	CAT art
(NP1 (red block) S2)	CAT adj
(NP1 (block) S2)	CAT noun
(NP2 nil S2)	POP
(S2 nil nil)	POP

Figure 5
Parse of 'Pick up the red block'.

no words and returns to the preceding network. Since both are viable states, the state (VP1 (up the red block) (S1)) is replaced by two states, (VP1 (the red block) (S1)) and (S1 (up the red block) (nil)). Had the first state not led to a successful parse, the search would have been resumed at the second state.

LISP Implementation of the Expand Function

The LISP implementation of the expand function which applies the grammar to generate successor states is a straight-forward testing of the three possible arc transitions listed in the previous section. On each iteration, the arcs emanating from the current state's node are tested against the next word in the words remaining. For each test that is successful, a new state is generated, which the search engine pushes onto the stack of states.

The LISP implementation of the expand function is given in figure 6. The code for the interpreter and a RTN grammar which handles all three commands discussed in the previous section is available via anonymous ftp from husserl.sbu.edu in the directory /pub/AI.

Conclusion

We have presented an interpreter for an RTN grammar which utilizes a generalized depth-first search engine with which students are familiar to

```
(defun expand (state)
(delete nil
 (let ((remaining_words (cadr state))
            (return_point (caddr state)))
  (mapcar #'(lambda (transition)
           (let ((arc (cadr transition))
                 (next_node (car transition)))
(cond          ;;CAT case
               ((and (equal (car arc) 'CAT)
                          (equal (look-up_catagory (car remaining_words))
                                 (cadr arc))
                )
                (list next_node (cdr remaining_words) return_point)
               )
                ;;PUSH case
               ((equal (car arc) 'PUSH)
                  (list (cadr arc) remaining_words next_node)
               )
                          ;;POP case
               ((and (equal (car arc) 'POP)
                          (not (null return_point))
                )
               (list return_point remaining_words nil)
               )
               (t nil))
  ) ;end mapcar let
           ) (get (car state) 'transitions))  ;argument to mapcar
  ) ;end expand let
)
)  ;end expand function
```

Figure 6
LISP implementation of function *expand*

the search among different possible parse paths. An expansion function applies the grammar to the remaining words on each iteration and generates successor states. Students are able to focus on the process of applying the grammar to input sentences without getting lost in the details of how backtracking from dead ends is handled.

Bibliography

Allen, James. *Natural Language Understanding.* Reading, MA: Benjamin/Cummings Publishing, 1987.

Johnson, Roderick. "Parsing with Transition Networks," *Parsing Natural Language.* New York: Academic Press, 1983.

Winston, Patrick. *Lisp: Second Addition.* Reading, MA: Addison- Wesley Publishing, 1984.