AN ISLAND PARSING INTERPRETER FOR THE FULL AUGMENTED TRANSITION NETWORK FORMALISM

John A. Carroll
University of Cambridge Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG
England

ABSTRACT

Island parsing is a powerful technique for parsing with Augmented Transition Networks (ATNs) which was developed and successfully applied in the HWIM speech understanding project. The HWIM application grammar did not, however, exploit Woods' original full ATN specification. This paper describes an island parsing interpreter based on HWIM, but containing substantial and important extensions to enable it to interpret any grammar which conforms to that full specification of 1970. The most important contributions have been to eliminate the need for prior specification of scope clauses, to provide more power by implementing LIFTR and SENDR actions within the island parsing framework, and to improve the efficiency of the techniques used to merge together partially-built islands within the utterance.

This paper also presents some observations about island parsing, based on the use of the parser described, and some suggestions for future directions for island parsing research.

I INTRODUCTION

A. Island Parsing

In an ordinary ATN parser, the parsing of a sentence is performed unidirectionally (normally left-to-right); the parser traverses each arc in the directed graph of the grammar in the same direction, starting from the initial state.

An island ATN parser, on the other hand, can start at any point in the transition network with a word match from anywhere in the input string, not just at the left end, and parse the rest of the string working outwards to the left and right, adding words to each end of the 'island' formed. Indeed, any number of islands can be built, the parser merging the islands together as their boundaries meet. Clearly, in speech processing, island parsing is well suited to gearing sentence processing to the most solid inputs from the acoustic analyser.

The main problems with previous implementations of island parsing for ATNs have been with scope clauses and LIFTR and SENDR actions; essentially, these problems arise because in island parsing structure determination has to work from right-to-left as well as in the more usual left-to-right direction, i.e. against the normal parsing flow.

B. Scope Clauses

The ATN formalism provides for actions on the arcs of the network which can set and modify the contents of 'registers', and arbitrary tests on an arc to determine whether that arc is to be followed. In an island parser, an action or test is referred to as being context-sensitive when it either requires the value of a register that is set somewhere to the left, or changes the value of a register that is used somewhere also to the left. For each context sensitive action or test, there exists a set of states to its left such that the action can safely be performed if its execution is delayed until the parse has passed through one of these states. This list of states must be expressed, and in the HWIM system (Woods, 1976), this is done when writing the grammar by using a scope clause. The form of a scope clause is

```
(SCOPE <scope specification>
       <list of context-sensitive actions>)
```

where the scope specification is the list of precursor states. This requirement for prior specification of scope clauses clearly adds to the burden of the grammar writer.

I have implemented a more satisfactory treatment of scope clauses. This is described below, following the discussion of LIFTR and SENDR actions, which require special handling in scoping.

II LIFTR AND SENDR ACTIONS

Two important actions (indeed it is difficult to write a grammar of any substantial subset of English without them) defined by Woods (1970), namely LIFTR and SENDR, present implementation difficulties in an island parsing interpreter. These actions were evidently excluded from the HWIM parser since there is no mention of them by Woods (1976).

The action LIFTR can occur on any arc in the network, to transmit the value of a register up to the next higher level in the network, whereas SENDR can only occur on a PUSH arc, to transmit the value of a register down to a lower level.

A. LIFTR

The same mechanism can be used to implement LIFTR actions as is used to transmit the result of each lower level computation up to the next higher level as the value of the special register '*'.

However, LIFTR presents problems with scope clauses in an island parsing ATN interpreter: if an action

(LIFTR <register> ...)

occurs in a sub-network, any action using that register in any higher sub-network that PUSHes for the one containing the LIFTR must be scoped so that the action is not performed in a right-to-left parse at least until after the PUSH has been executed. See figure 1.

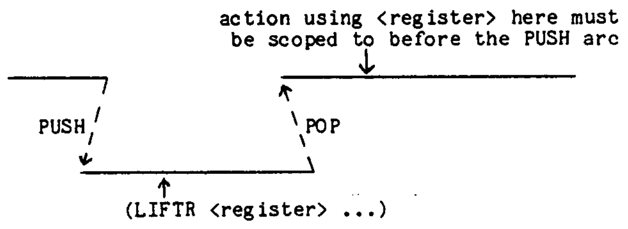
action using <register> here must be scoped to before the PUSH arc
PUSH
POP
(LIFTR <register> ...)

Figure 1. Scoping LIFTR actions.

So, for example, when parsing English from right to left, tests that the verb and subject agree in person and number (if this information is carried in registers) must be postponed until the PUSH for the beginning of the subject noun phrase. Section III describes how my interpreter takes care of this scoping problem.

B. SENDR

1. Treatment of actions using SENDRed registers

Since in a right-to-left parse, lower level sub-networks are traversed before the PUSH to them is performed, there is no way of knowing the value of a register that is being SENDRed at least until after the PUSH. Thus all actions involving registers whose values depend on the value of that register must be saved to be executed at the higher level.

I have dealt with this by putting such actions into SCOPE clauses containing a special new scope specification, which I call scope SENDR. Actions with scope SENDR are never executed at the current level in the network, but are saved and incorporated into the next higher level subnetwork (possibly with a changed scope specification) during processing of the PUSH at that higher level, as follows:-

(1) The form on the POP arc to be returned as the value of the special register '*' on return to the next higher level is put into an explicit LIFTR action.

(2) The scopes of all the saved actions are changed to the same as those of the SENDR actions on the PUSH arc.

(3) All LIFTR actions are changed to highlvl-setr actions (see below).

(4) Scoped calls to lowlvl-start and lowlvl-finish (see below) are put respectively before and after the saved actions.

(5) All the SENDR actions on the PUSH arc are put in front of the lower level saved actions.

The rest of the actions on the PUSH arc are then processed as normal. The purposes of the actions lowlvl-start and lowlvl-finish are to respectively set up and restore a stack of register contexts (hold-regs), each level in the stack holding the register contents of one level in the network, with the base of the stack representing the highest level of saved actions. The action highlvl-setr performs a SETR at the next higher level of register contexts on the stack.

2. An Example

A typical sequence of actions in a fragment of an ATN network might be as in figure 2.

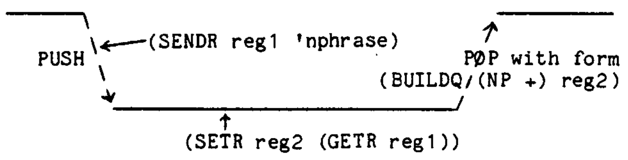

Figure 2. A typical fragment of a network.

	regs: ((reg0 pphrase)) lowlvl-regs: NIL hold-regs: NIL
(SENDR reg1 'nphrase)	lowlvl-regs <- ((reg1 nphrase))

(lowlvl-start)	hold-regs <- (((reg0 pphrase)))
	regs <- ((reg1 nphrase))
	lowlvl-regs <- NIL

(SETR reg2 (GETR reg1))	regs <- ((reg2 nphrase) (reg1 nphrase))
(highlvl -setr * (BUILDQ (NP +) reg2))	hold-regs <- (((* (NP nphrase)) (reg0 pphrase)))

(lowlvl-finish)	regs <- ((* (NP nphrase)) (reg0 pphrase))
	hold-regs <- NIL

Figure 3. Treatment of SENDR actions.

This would be translated into the list of saved actions on the left of figure 3, and when control had passed through a set of states such that the actions' scope specifications were satisfied, execution would produce the sequence of operations shown on the right of the figure.

3. Scope Problems

As with LIFTR, SENDR actions need special scoping treatment: since there can be any type of interaction on a lower level between registers SENDRed and registers to be LIFTRed, the only safe execution time for actions using these registers and for actions referencing registers whose values depend on them (without engaging in full symbolic execution) is when the higher level sub-network has been fully traversed. There is a special scope specification for this - scope T.

III AUTOMATIC SCOPE COMPUTATION

The process of writing scope clauses into the grammar for an island parser is laborious, and therefore prone to error. The implementation described here can automatically detect all context-sensitive actions and tests and put them into scope clauses containing suitable (and usually optimal) scope specifications. Thus the parser can interpret straight off an ATN grammar that has been written for an ordinary left-to-right parser.

The scoping algorithm consists of five passes over the grammar, the first four dealing with the exceptional scoping required by LIFTR and SENDR actions, and the fifth with the rest of the actions and tests in the network. Comments on the algorithm follow the necessarily technical account of it.

A. The Scoping Algorithm

The five passes of the scoping algorithm will now be described, actions and tests in the network being treated identically.

1. Pass 1

Pass one takes care of the scoping problem with LIFTR actions mentioned in the previous section - that a register being LIFTRed must be scoped back at the higher level to at least before the PUSH arc.

But if the register is used on the PUSH arc itself, the scoping algorithm should produce correct scope specifications without needing to treat this as a special case. Thus the solution I have adopted is for the algorithm to check whether the register appears on the PUSH arc, and if not, the dummy action

(SETR <register> (GETR <register>))

is added to the actions on the PUSH arc.

2. Pass 2

The second pass finds, for each sub-network, the names of all the registers whose values depend on other registers (for use in the subsequent scoping passes). It does this by finding the registers used in each register-setting action (SETR, LIFTR, or SENDR), using knowledge of the register usage of each function used, and for each register which is not being assigned to, it appends onto the property-list of the register the name of the register being set in the current action, and a pointer to that register's property-list.

Thus in the end, each register is associated with a list of all the registers in the sub-network which depend on the value of that register.

3. Pass 3

Pass three deals with scoping SENDR actions, giving them the treatment described at the end of the last section - it assigns the scope specification T to all actions which reference registers whose values depend on any of the registers used in actions on the same PUSH arc as a SENDR action.

4. Pass 4

Pass four finds all actions that use registers that have been passed down from a higher level by a SENDR, and also actions which use registers dependent on those SENDRed registers, giving the actions scope SENDR.

5. Pass 5

The rest of the scoping is performed in pass five. Each action is considered in turn, collecting the names of all registers it uses, and the names of those whose values depend on them. The scope specification is then computed depending on the common part of all possible paths from the start of the current sub-network to any action which is dependent on the action under consideration. This list of states ('left-states') is the intersection of the states to the left of each action which uses any of the collected registers.

The algorithm distinguishes the following four cases for the contents of 'left-states':-

(1) If NIL - there are at least two non-intersecting paths from the left to the arc containing the action which reference registers dependent on those in the action, so return scope specification T.

(2) All states in 'left-states' are in loops in the network - it is very difficult to compute the optimal scope specification, so return T (which will always be correct though perhaps not optimal). The problem with loops is that no register should be changed or referenced in a right-to-left parse until control has finally passed out of the loop.

(3) The left state of the arc containing the action being scoped is in 'left-states', and the state is not in a loop - all dependent actions are to the right of the arc, so return NIL.

(4) Otherwise - return as scope specification a list of all states in 'left-states' that are not in loops.

If an action does not use any registers, it obviously does not need scoping, and the algorithm bypasses it. If a scope specification is returned for an action that is already scoped, whether the new scope 'overwrites' the old one depends on what is already there:-

```
scope SENDR overwrites scope T
scope T overwrites scope <list of states>
scope <list of states> is appended to an
                 existing scope <list of states>
```

B. Discussion of the Scoping Algorithm

The algorithm does not produce totally optimal scope specifications in all circumstances: that is, actions may sometimes be scoped so they are saved for longer in the parse before they are executed than may strictly be necessary. The main shortcoming is in dealing with networks where there are two or more alternative separate paths containing actions using registers computed to be interdependent; for example in scoping the network fragment in figure 4,

```
(NP/
   (JUMP NP/DET T)
   (CAT NPR T
      (SETR noun (BUILDQ (npr *)))
      (TO NP/POP)))
(NP/DET
   (CAT ADJ T ... (TO NP/DET))
   (CAT NOUN T
      (SETR noun *)
      (TO NP/POP)))
(NP/POP
   (POP ... ))
```

Figure 4. Scoping with alternative paths.

the two actions using register 'noun' are scoped (NP/) but the paths through them are independent and the register is not used elsewhere, so the actions do not need to be scoped at all. There does not seem to be any way around this problem by modifying the algorithm, but fortunately scope specifications that are not entirely optimal (as in this case) should only minimally affect the performance of the interpreter when parsing a sentence.

IV MERGING PARTIALLY BUILT ISLANDS

In the HWIM system, to join together two adjacent islands to make one island covering them both, the smaller island was broken up and the words from it added onto the end of the larger. This obviously wastes all the effort expended in building the smaller island.

A more efficient method of joining two islands which I have implemented, is to merge all the segment configurations 'Sconfigs'[1] at the boundaries of each island that are compatible, and then splice those that completely cover a sub-network into as many successively higher levels as possible (by calling Woods' 'Complete-right' function as many times as possible). In a real-time speech understanding system (depending on the strategy it employed), the time saved by this method could be critical to the success of the system.

V OBSERVATIONS ON THE INTERPRETER IN USE

The parser has been tested (Carroll, 1982) with various sized (purely syntactic) grammars, simulating speech processing by the arbitrary selection of one or more words in a typed string as parsing starting points, and the arbitrary addition of words to the left and right of these.

It has been observed that the more complex the structure of the sentence being parsed, the more Sconfigs get generated, and consequently the longer the parse takes. There are, however, other less obvious factors influencing the number of Sconfigs generated.

A. Saved Tests

Sconfigs tend to proliferate embarrassingly when there are many possible paths of JUMP arcs between states on the same level of the grammar due to scoped tests having to be saved and not being immediately executable.

If there are no SENDR actions down to the sub-network containing the JUMPs, then none of the saved tests will have to be carried up to a higher level, and so many of the Sconfigs will be filtered out when the POP arc at that level is processed. But if there are SENDR actions, the Sconfigs will not be filtered so effectively, will be carried up to higher levels, and at each higher level the number of Sconfigs will multiply.

This Sconfig proliferation and resulting combinatorial explosion will always be associated with island parsing using large complex grammars that are purely syntactic[2]; unfortunately LIFTR and SENDR actions aggravate the problem. However, the utility of these actions more than outweighs the consequent decrease in parse-time efficiency.

----

[1] The state of the parse in an island parser is held as a list of segment configurations, each of which represents a partial parse covering one or more words in the utterance.

[2] It seems that the HWIM parser also encountered these problems; their solution was to employ semantic grammars, with a large number of WRD arcs, to use both syntactic and semantic categories on CAT arcs, and to expand the set of constituents pushed for to include "semantic constituents".

B. Differing Word-Orders

Parsing the same sentence with differing orders of adding the words in it to islands usually results in differing numbers of Sconfigs being created. For example, two parses of the sentence

JOHN IS EAGER TO PLEASE.

gave the results:-

	run 1	run 2
Sconfigs generated	388	182
parse time (secs.)	1.77	1.08

The difference was caused by the fact that in the first run, 'IS' was used as an initial island, setting up expectations for more possible distinct final sentence structures than in the second run, which started with the word 'PLEASE'. This difference in expectation status reflects the different structuring potential of the two words.

VI SOME FUTURE DIRECTIONS FOR RESEARCH

A. Parsing Conjunctions

Island parsing appears to offer a promising solution to the problem of parsing written as well as spoken sentences containing conjunctions; although the ATN formalism is quite powerful in expressing natural language grammars, it faces problems dealing with sentences containing conjunctions: (WRD AND ...) arcs need to be inserted almost everywhere since AND can conjoin any two constituents of the same type. Boguraev (1982) has suggested that this problem might be overcome by building islands at each conjunction and parsing outwards from them.

B. Cascaded Island Parsers

The nuisance of the combinatorial explosion of Sconfig numbers when using large complex grammars might be amenable to solution with cascaded island ATN interpreters[3] ; several island parsers could be put on top of each other, each having a separate domain of responsibility and each passing up completed constituents to the next higher ATN. In his 1980 paper, Woods explains how cascading gains the "factoring" advantage of allowing alternative configurations in the later stages of the cascade to share common processing in the earlier stages. This processing would otherwise have to be done independently - in the case of an island parser, producing duplicate Sconfigs which would contribute to a possible combinatorial explosion.

SENDR actions might cause problems, however, if scoping due to them caused the actions intended to form complete constituents to be saved so that the actions would not be completely performed before the time came to pass the constituents up to the next ATN. For this reason, restrictions might have to be placed on the ATN grammars used, but this requires further investigation.

[3] The idea of cascading was first put forward by Woods (1980), but only in terms of ordinary left-to-right ATN parsers.

VII ACKNOWLEDGEMENTS

I would like to thank Bran Boguraev for his guidance during the writing of the interpreter, and for supplying the ATN grammars I have used. Thanks also to Karen Sparck Jones and John Tait for their comments on earlier drafts of this paper.

VIII REFERENCES

Boguraev, B. (1982) personal communication.

Carroll, J. (1982) "An Island Parsing Interpreter for Augmented Transition Networks". University of Cambridge Computer Laboratory Technical Report No.33.

Woods, W. (1970) "Transition Network Grammars for Natural Language Analysis". Communications of the ACM, 13, 10, 591-606.

Woods, W. et al. (1976) "Parsers" in "Speech Understanding Systems". Bolt, Beranek and Newman Inc. Report No.3438, Vol.4, 1-21.

Woods, W. (1980) "Cascaded ATN Grammars". American Journal of Computational Linguistics, 6, 1, 1-12.