Technical Report

UCAM-CL-TR-33
ISSN 1476-2986

Number 33

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

An island parsing interpreter for Augmented Transition Networks

John A. Carroll

October 1982

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

An Island Parsing

Interpreter for Augmented Transition Networks

John A Carroll

October 1982

Abstract

This paper describes the implementation of an 'island parsing' interpreter for an Augmented Transition Network (ATN). The interpreter provides more complete coverage of Woods' original ATN formalism than his later island parsing implementation; it is written in LISP and has been modestly tested.

University of Cambridge Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG
England

CONTENTS

0. PREFACE

This paper was originally written as an undergraduate project dissertation for the Cambridge University Computer Science Tripos.

An Augmented Transition Network (ATN) is a means of describing a grammar, its main application being to parse natural language sentences. An interpreter for an ATN takes as its input:

(1) the network
(2) a lexicon of words and their associated parts of speech
(3) a natural language sentence;

it outputs a representation of the structure of the sentence, determined by structure-building actions written into the network.

In an ordinary ATN parser, the parsing of a sentence is performed from left to right - the parser traverses each arc in the directed graph of the grammar in the correct 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 sentence (not just at the left end), and parse the rest of the sentence 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 them together as their boundaries meet.

The algorithm for island parsing is very much more complicated than for standard left-to-right parsing. There are nevertheless areas in which there are good reasons for employing island parsing:

(1) When processing speech input, where some words in the utterence may not immediately be recognised by the lexical component, the parsing component can use words that have been recognised to help deduce the others. Thus the first word passed to the parser is not necessarily the first word in the sentence; islands are built with each identified word as a seed, and the islands are coalesced as they 'collide'.

(2) In addition, it has been suggested that scanning the input sentence for conjunctions, and using each as an island seed, will save having to explictly and laboriously put conjunction arcs in grammars, since all conjunctions do is join two constituents of the same type together.

Woods implemented an island parsing ATN interpreter as part of a speech understanding project at BBN. This interpreter departed in some respects from his original formulation of the ATN formalism. Also, extra effort was required to include 'SCOPE clauses' (see section 1.2 below) in ATN grammars written for the interpreter.

The interpreter described in this report was written in an attempt to cover the whole of the ATN formalism and to deal with scoping without explicit representation in the grammar. The interpreter has been tested on only 15 sentences, but using a grammar approximately 300 lines long; the test was therefore a reasonable one, since the size and complexity of grammar used is the critical factor, rather than the number of sentences. Some example runs are given in Appendix 1. The interpreter is written in Cambridge LISP, a not too eccentric dialect of LISP [Winston81].

Chapter 1 gives more background to parsing with ATNs, and the next chapter describes the interpreter's static data structures. The main island parsing algorithm is presented in chapter 3, and chapters 4 to 6 describe the implementation of the features that seem to be unique to this island parsing interpreter. Chapter 7 finishes the description of the interpreter algorithms with details of the top-level driving functions, and chapter 8 draws some conclusions from the development of the interpreter.

1. INTRODUCTION

1.1 The Augmented Transition Network Formalism

The syntax of an Augmented Transition Network (ATN) is defined in [Woods70], the seminal paper on ATNs, and although there have been subsequent alternative (but semantically equivalent) definitions, even by Woods himself, I have adhered to the original syntax with a couple of extensions:

```
<transition network> ::= <arc set> <arc set>*
<arc set> ::= (<state> <arc>*)
<arc> ::= (WRD <word> <test> <action>* (TO <state>)) |
          (CAT <category> <test> <action>* (TO <state>)) |
          (MEM (<word>*) <test> <action>* (TO <state>)) |
          (JUMP <state> <test> <action>*) |
          (PUSH <state> <test> <action>* (TO <state>)) |
          (POP <form> <test> <action>*)
<action> ::= <action1> | (SCOPE <scope spec> <action1>*)
<action1> ::= (SETR <register> <form>) |
              (SENDR <register> <form>) |
              (LIFTR <register> <form>) |
              <any call to a LISP function>
<test> ::= <form> | (SCOPE <scope spec> <form>)
<form> ::= (GETR <register>) |
           * |
           (BUILDQ <buildq list> <register>*) |
           <any call to a feature extracting function> |
           <any call to a LISP function>
<scope spec> ::= (<state> <state>*) | T | NIL
```

Thus the network is in the form of a directed graph with the states of the network as the nodes. An ATN interpreter takes a **network** , a **sentence** to be parsed, and a **lexicon** of words, and traverses the arcs of the network, perhaps 'consuming' the next word in the sentence depending on the type of the current arc. Finally, when all the words in the sentence have been consumed, a set of structures are returned corresponding to each valid parse of the sentence (if any).

Each arc, in addition to its main function described below, can perform structure building actions by setting and modfying the contents of registers. The test on each arc may involve testing the contents of registers, and the arc is followed only if the test returns a non-NIL result. WRD and MEM arcs (the latter being an extension to the original formalism) specify the words they can consume explicitly and exhaustively, whereas CAT arcs specify them via the syntactic category to which they belong, found by consulting the lexicon. JUMP arcs are similar to these three arcs but do not consume any words.

A PUSH arc performs a 'subroutine call' by causing a new transition network to be entered at a state on a lower level. If the lower level network can successfully accept the next segment of the input sentence, it is exited by a POP arc, and processing continues in the higher level network, returning the structure obtained by evaluating the 'form' on the POP arc.

The function SETR sets the contents of a register to the value of a 'form' at the current level in the network; SENDR does this at the next lower level as a PUSH is being performed to that level, and LIFTR does this at the next higher level when a POP arc is being followed. GETR returns the value of a register, and BUILDQ builds a list structure representing a fragment of a parse tree with the contents of the registers specified in the call substituted into the list. The form '*' contains the current word on a consuming arc, or the result from the lower level on a PUSH arc.

A small example of an ATN grammar is:

```
(S/
   (PUSH NP/ T
      (SETR subj (BUILDQ (subj *)))
      (SETR type 'declarative)
      (TO S/NP))
   (JUMP S/NP (CAT VERB)
      (SETR type 'imp)
      (SETR subj '(NP (pro  YOU)))))
(S/NP
   (CAT VERB T
      (SETR verb (BUILDQ (verb *)))
      (TO VP/V)))
(VP/V
   (PUSH NP/ T
      (SETR obj (BUILDQ (obj *)))
      (TO S/POP))
   (JUMP S/POP T NIL))
(S/POP
   (PUSH PP/ T
      (SETR mods *)
      (TO S/POP))
   (POP
      (BUILDQ (S (type +) + (VP + + +))
         type
         subj
         verb
         obj
         mods)
      T))
(PP/
   (CAT PREP T
      (SETR prep *)
      (TO PP/PREP)))
(PP/PREP
   (PUSH NP/ T
      (SETR np *)
      (TO PP/POP)))
(PP/POP
   (POP (BUILDQ (PP (prep +) (np +)) prep np) T))
(NP/
```

```
   (JUMP NP/DET T NIL)
   (CAT DET T
      (SETR det *)
      (TO NP/DET))
   (CAT NPR T
      (SETR noun (BUILDQ (npr *)))
      (TO NP/POP)))
(NP/DET
   (CAT ADJ T
      (SETR adj *)
      (TO NP/DET))
   (CAT NOUN T
      (SETR noun *)
      (TO NP/POP)))
(NP/POP
   (PUSH PP/ (CAT PREP)
      (SETR mods *)
      (TO NP/POP))
   (POP
      (BUILDQ
         (NP (det +) (adj +) (noun +) (mods +))
         det
         adj
         noun
         mods)
      T))
```

When words are put into the lexicon (see 2.2) they can be tagged as having various features to be recalled for computation in actions and tests in the network, so my interpreter contains functions to do this, and also other more general-purpose functions which have been found to be useful in writing ATN grammars which are more complex than the example given above.

1.2 Island Parsing

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

An island ATN parser, on the other hand, can start at any point in the network with a word (the island seed) from anywhere in the input sentence, and parse the rest of the sentence working outwards to the left and right, adding words to each end of the island, as is often needed when processing speech input. (The first word recognised may not be the first word in the sentence).

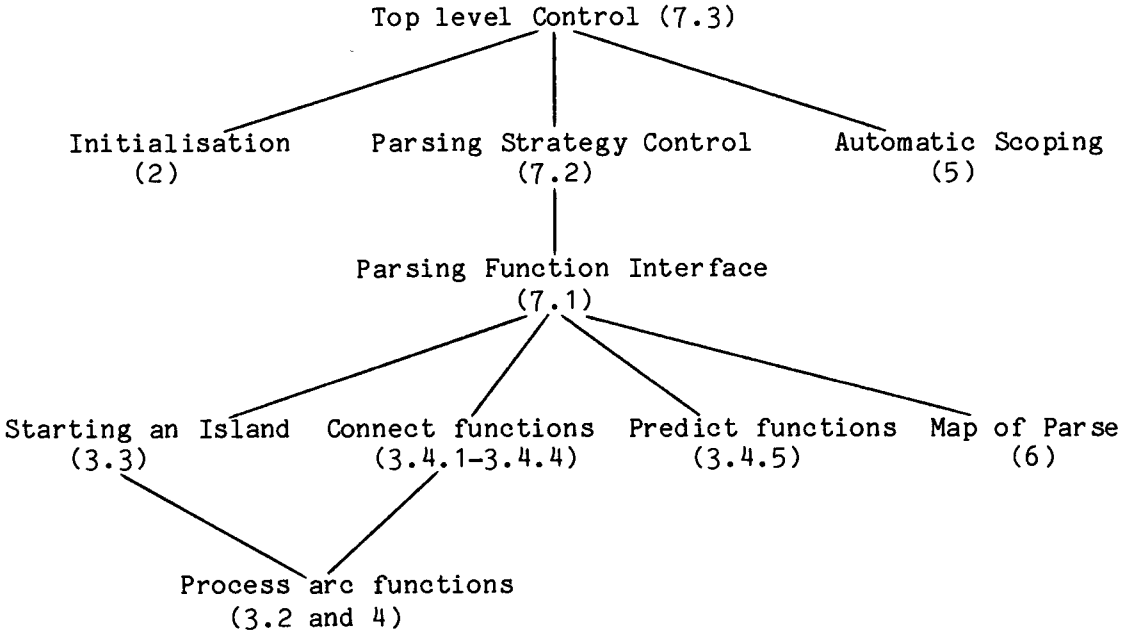
This ability to traverse the network from right to left gives rise to the problem that actions and tests which either require the value of a register that is set somewhere to the left, or change the value of a register that is used somewhere to the left (called context sensitive actions or tests), must have their execution delayed until the parse has passed through a specified set of states to their left. This is the purpose

of the SCOPE clause in the above ATN syntax definition; the scope specification can either be a list of these states, or T, which means delay the execution until this level of the network is joined to a higher level, or NIL, which means that execution can be immediate. Of course the SCOPE clause is irrelevant to an ordinary left to right parser, so it does not appear in [Woods70].

The algorithms required to implement an island parsing ATN interpreter are complex; they exist in a sketchy and incomplete form in [Woods76], and my interpreter is based on this paper. However, Woods' island parsing interpreter departs from his original ATN definition in the treatment of tests on the arcs of the grammar, can deal with only a subset of the full ATN formalism as defined above, and also requires extra work writing grammars to insert SCOPE clauses. These restrictions are quite serious when it comes to writing a grammar for a substantial subset of English. My program interprets the full formalism, and can automatically insert SCOPE clauses, thus "understanding" any grammar written for an ordinary left to right parser. These extensions are believed to be **original work** and chapters 4 and 5 in this paper describe these important and substantial extensions in detail.

1.3 Program Structure

The interpreter is highly structured, and is organised in LISP MODULES, with a very simple user interface provided by 6 top level functions, described in section 7.3. The following is a diagram showing the logical organisation of the main tasks in the interpreter, along with the numbers of the sections containing their descriptions.



2. DATA STRUCTURES AND INITIALISATION

2.1 Storing the Grammar

The connection of two states by a particular type of arc may be thought of as a relation of that name between those two states. The basic relations are J, B, C, and LJ, UB, UC. B represents the relationship between the state at the beginning of a PUSH arc and the state pushed to, C the relationship between the state at the beginning of a POP arc and a state to which control could return at a higher level, and J the relationship between the states at each end of a JUMP arc. The other three relations are the right to left counterparts.

Sets of these basic relations may be concatenated to describe a path over more than one type of arc, and + is used for alternation of relations. Using this notation some more complex relations that are extensively used in the island parsing algorithm can now be named. These are B# and C# for left-to-right parsing, and UB# and UC# [1] for right-to-left:

```
B# = B ((J+B)* B)*
C# = C ((J+C)* C)* ((J+B)* B)*
UB# = UB ((LJ+UB)* UB)* ((LJ+UC)* UC)*
UC# = UC ((LJ+UC)* UC)*
```

So (STATE1 B# STATE2) is equivalent to the statement (STATE1 B((J+B)*B)* STATE2) i.e. there is a path through the network starting at STATE1 with a PUSH arc, followed by a sequence of zero or more JUMP and/or PUSH arcs, and finishing at STATE2.

The sets of states satisfying the '#' relations are not explicitly given in the grammar, and therefore for efficiency they are computed before any sentences are parsed. Both the parsing and also the scoping algorithms need to know, for any state, which arcs enter the state from the left, and which leave to the right. Thus the grammar is pre-processed, and stored in a form alowing for easy and efficient access to this information.

The following items can be extracted straight from the grammar as input:

for each arc - left state, right state, type of arc, test, actions
for each state - right-going arcs

A unique name (LISP atom) is generated for each arc, and it is by this that each arc is known; the left state etc. of each arc is PUT on the property list of the arc's name, and similarly for the information associated with

---

[1] Woods' original 1976 notation was B! etc. for these relations, but using the character '!' was felt to be confusing since it has other special meanings in Cambridge LISP and the language BCPL.

each state. The arc names and states are stored respectively in the global lists *arcs and *states to facilitate processing passes across the whole network during initialisation and scoping.

The rest of the information in the network that is needed is fairly easily computed in passes across the lists of arcs and states:

```
for each state - left-going arcs
                 J* and LJ* sets of states
                 B and UB sets
                 C and UC sets
                 B#, C#, UB#, and UC# sets
```

2.2 Storing the Dictionary

A dictionary can be supplied by the user to the interprter so that any word can be specified as belonging to a set of syntactic categories (more than one for words with several meanings which depend on context); the word can also optionally be given arbitrary features and an uninflected form for recall when the word is encountered in a sentence. (Indeed, the form '*' is set to the uninflected form of a consumed word, and *word to the actual word). The dictionary format is:

```
<dictionary> ::= <word entry>*
<word entry> ::= (<word> <category desc>
                         <feature desc>
                         <uninflected desc>)
<category desc> ::= (cat <syntactic category>*)
<feature desc> ::= (features <feature>*)
<uninflected desc> ::= (uninflected . <word>)
```

An example of a 'word entry' is

```
(likes
   (cat NOUN VERB)
   (features PLURAL TRANSITIVE)
   (uninflected . like))
```

The attributes of a word are stored on its property list. The dictionary format makes this processing particularly simple; 'data-driven' programming can be used by defining functions 'cat', 'features', and 'uninflected', and EVLISing the CDR of each word-entry in the dictionary.

Then finally, for each consuming arc in the network, pointers to the arc are put on the property list of each of the word(s) which can be consumed by that arc. This is for later use in determining which arcs in the network need to be considered when a new word in the input sentence is about to be parsed. (Sections 3.3 and 7.1).

2.3 Reading the Input Sentences

Before any sentences are parsed, the interpreter reads in all the sentences input by the user to the current run, so it does not have to worry about selecting and releasing files each time it wants a new sentence. The sentences are read character-by-character as opposed to atom-by-atom, in order to be able to deal with punctuation, since the LISP atoms ".", "'", and "!" have special meanings. The processing done is:

(1) one of ".", "!", or "?" must be used to terminate each sentence.

(2) "," is made into a separate 'word', i.e.
John, and Mary -> John , and Mary

(3) "'" (denoting possession) is translated into the special marker word POSS, i.e.
John's train -> John POSS train
Bill Jones' house -> Bill Jones POSS house

Each sentence is stored in a vector, so that with a pointer to the current word being parsed, (see 3.1.1), tests in the grammar can be performed on the next word in the sentence without actually consuming it.

One of the parsing strategies available to the user (see 7.2) requires the input, immediately after a sentence, of numbers indicating in what order the words in the sentence are to be parsed. Each number corresponds to one word, so, due to the special treatment of the punctuation "," and "'", extra numbers are generated for any additional 'words' introduced by the above processing. The numbers are then stored in the vector *word-order, with the i'th element containing the position in the sentence of the i'th word to be parsed.

Finally, each word in the sentences is checked to see that its property list contains pointers to at least one consuming arc in the grammar for that word, computed at initialisation (see 2.2). If not, no sentences containing that word are parsed, since it can immediately be seen that there can be no valid parses for them.

3. MAIN PARSER FUNCTIONS

3.1 Dynamic Data Structures

3.1.1 Segment Configurations

In an ATN interpreter the 'state of the parsing process' has to be stored to keep track of the parse. In a standard left to right ATN parser, such a state is described by a configuration, containing (at least) a pointer into the input, the current state in the grammar, the current register settings, and a stack of next states and register settings for the higher levels of the computation. However, this notion of a configuration is inadequate for an island ATN parser, since there can be no stack to store all higher levels of the computation, because higher levels cannot be known with certainty unless the computation begins at the left end.

To overcome this problem, the configuration is redefined as a list of segment configurations (Sconfigs), which represent the state of the computation for each level of the grammar in which any word of the island can be consumed. A segment configuration consists of:

(1) pointer to the word in the sentence at the left end of the segment (lbdry)

(2) pointer to the word in the sentence at the right end of the segment (rbdry)

(3) left-most state of the segment (lstate)

(4) right-most state of the segment (rstate)

(5) list of register settings already made at this level

(6) list of register settings to be carried up to next higher level (liftr-regs)

(7) context-sensitive actions and tests yet to be done

(8) list of states traversed at this level

(9) list of arcs traversed at this level, together with word consumed by each arc (if any)

Items (1)-(4) are required to be able to join 2 segments together, and to extend a segment with a new word to be consumed. Testing whether 2 segments contain adjacent words is paticularly easy: is the difference

between the word pointers into the vector holding the sentence on the 2 closest boundaries equal to 1?

The list of register settings is an association list of pairs of the form (register-name value), so the value of a register can easily be extracted using the LISP function ATSOC. The list of arcs traversed is also of this form, i.e. (arc-name consumed-word) and is used to generate a map of the history of the parse on completion of parsing the current sentence. The list of states traversed is used to determine when a saved scoped action or test can safely be executed.

A member of the list of saved context-sensitive actions and tests consists of:

(1) name of the arc on which the action/test appears

(2) test, or list of actions

(3) scope-specification

(4) list of register contents (the register context) when the actions / test were saved

(5) pointer to the word in the sentence being parsed when the actions / test were saved

The fact that more than one action can be contained in a saved scoped action is due to the ability of scope clauses to contain a list of actions, whereas the ATN syntax allows only one test on an arc, so a SCOPE clause cannot contain more than one test.

The arc name (1) is not necessary for the computation, but is helpful in seeing where the saved action has come from when debugging. Likewise, the register context (4) is only necessary for a parser which is designed to deal with SENDR actions (see 4.2). Item (5) is included to enable tests on the next word to the right in the sentence to be performed before actually being consumed by the parser (look-ahead).

3.1.2 Representation of Sconfigs

The version of LISP in which the project was written has been extended from 'Cambridge LISP' [Norman] to support a record package. Thus the natural way to represent Sconfigs, saved actions and tests is by using records with 9, 5, and 5 fields respectively; the saved actions and tests are declared as different record types, so they can be distinguished, since they require slightly different treatment when they are evaluated (see 3.2).

Mainly to make debugging easier, it was decided to generate a unique name (LISP atom) for each Sconfig as it was created, and hang the record representing the Sconfig off the property list of the atom. The advantages of this against the more obvious method of just manipulating records on their own are:

(1) giving each Sconfig a name makes it easier to follow its progress through stages in the computation.

(2) run-time tracing information of arguments to functions and results of functions is less cluttered, since routines pass names of Sconfigs around, and not the records themselves explicitly.

The extra level of indirection introduced has minimal effect on the execution time of the program.

3.1.3 Organisation of Sconfigs

The names of all Sconfigs are held in the global list *sconfigs. The algorithms in section 3.4 require lists of Sconfigs starting or ending at a particular word in the sentence, so to improve efficiency there are also two vectors *lbdry and *rbdry which, for each word in the sentence hold names of Sconfigs with their 'lbdry' or 'rbdry' at that point.

There are three routines which manage the creation and deletion of Sconfigs:

(1) Generate-sconfig(state word-number) creates a new Sconfig with 'lstate' and 'rstate' equal to state, 'lbdry' and 'rbdry' equal to word-number in sentence, puts the Sconfig into the above data structures, and returns its name.

(2) Copy-sconfig(Sconfig) makes a copy of Sconfig, puts the new Sconfig into the relevant data structures, and returns its name. The routine is called when it is required to extend a Sconfig with one or more new arcs, but the original Sconfig might still be needed at some later stage in the parse.

(3) Kill-sconfig(Sconfig) deletes Sconfig from all the data structures, and is called when the parsing algorithm finds that there is no possibility of fitting this Sconfig into a valid parse of the current sentence.

3.2 Processing an Arc

There are three functions to add an arc to the end of a Sconfig, and which one is used depends on which one of the six types of arc is being processed:

Doarc - processes WRD, CAT, MEM, and JUMP arcs
Doarc-pop - processes POP arcs
Doarc-push - processes PUSH arcs

The routine Doarc will be described here; the other two routines are similar (if SENDR actions are ignored), and so only their main differences will be mentioned. (The changes that were required to implement SENDR will be outlined in 4.2).

3.2.1 Doarc

Doarc takes as arguments the name of a Sconfig to be extended, the extending arc, direction specifying which side of Sconfig the arc is on, and if the arc is consuming, the word consumed and its position in the sentence.

Doarc(Sconfig arc direction word word-number)

(1) Verify that 'arc' is adjacent to Sconfig

(2) Do the context-free actions on 'arc', having set up register * to contain the uninflected form of 'word'

(3) For the scoped test on 'arc'
- if scope is satisfied then evaluate test, and if result is NIL, set flag to delete Sconfig
- otherwise create a saved test and add to Sconfig

(4) For each scoped action on 'arc'
- if scope is satisfied then execute actions
- otherwise create saved actions and add to Sconfig

(5) If 'direction' is left, evaluate those saved actions and tests whose scope is now satisfied

(6) If Sconfig is to be deleted
- then kill Sconfig and return
- else set up new boundaries, change the appropriate boundary array, set up new end states, and states and arcs traversed.

3.2.2 Doarc-pop

Actions cannot be scoped on a POP arc, so (3) above is redundant. Instead, all saved actions and tests are evaluated, and then those on the POP arc. The right-most state of the Sconfig is set to NIL to signify that the Sconfig has been 'popped', and 'liftr-regs' is set to contain the form to be returned to a higher level computation.

3.2.3 Doarc-push

The routine joins a lower level Sconfig to either the left or right of an adjacent one via a PUSH arc. 'Liftr-regs' of the lower level Sconfig are added to the front of the current register context of the higher level one.

3.3 Generating an Island

The function Startisland finds all arcs in the grammar that can consume the word that is to form the island seed, using the list of pointers to arcs that can consume the word (see 2.2). Generate-sconfig is then called to create a Sconfig for each arc pointed to, and Doarc is called to process the arc.

3.4 Connecting a New Word to an Island

This section describes the process of adding a new word to the right of an existing island (by Connect-right). Connecting a new word on the left of an island (Connect-left), connecting an island to both ends of the grammar (Connect-finish-left and Connect-finish-right), and joining together two islands as they meet (Connect-collide) are all similar, and so will not be described.

Connect-right, and the functions it calls, are the heart of the interpreter, in that they connect an island to a set of one or more consuming arcs for the next word. Initially, all Sconfigs on the right of the island are checked to see if they can be extended to one of the arcs via a sequence of JUMP arcs. Then, if any of the Sconfigs can PUSH or POP to one of the arcs, a segment is generated at the higher or lower level and extended to one of the consuming arcs. If a Sconfig has been able to POP, it is checked (by Splitc#q) to see if it can be incorporated into a higher level segment (by Complete-right). What follows is a more detailed description of this algorithm.

In extending the segments at the right end of an old island to meet a new word to be added, the sequence of intervening non-consuming arcs that make the connection can be all at the same level of the network; they can change to a lower level; or they can change levels in a way equivalent to one or more POPs followed by zero or more PUSHes. Depending on the type of connection, and the stage of extension, Connect-right keeps the partially extended segments in one or more of the following six queues:

(1) TODOQ - contains all segments that have not changed levels, initially containing all right boundary segments from the old island.

(2) C#Q - contains all segments that will have to be pushed for in order to reach the new word (which is at a higher level)

(3) C#OPENLEFTQ - contains those segments from the C#Q that are incomplete on the left, i.e. cannot be pushed for yet.

(4) C#COMPLETEQ - contains the remainder of the C#Q, ie those segments that are ready to be incorporated into a higher level segment on the left.

(5) B#Q - contains those segments that are on the same level as that of one of the next consuming arcs, and so must not change levels.

(6) DONEQ - contains those segments whose right end state has a consuming arc for the new word being added to the island, together with left state of the arc.

Any contradictions between the algorithms given below, and the corresponding ones in [Woods76] are fully intentional, and constitute oversights in that original paper.

3.4.1 Connect-right

The function Connect-right joins a set of arcs that can consume the new word (toarcs), to Sconfigs that terminate at the right boundary of the island.

Connect-right(toarcs word word-number)

(1) Create a set of J*TOSTATES, which are states such that each state can J* to the left state of at least one of toarcs.

(2) Put all Sconfigs that are at the right boundary of the old island into the TODOQ.

(3) If there is anything in the TODOQ, call Extend-paths-right with the whole TODOQ, and directions set to 3 (see 3.4.2 for explanation of 'directions').

(4) If there is anything in the C#Q, call Splitc#q to separate it.

(5) If there is anything in the C#COMPLETEQ, call Complete-right to create the constituents and process the PUSH arcs to them. Then go back to (3).

(6) If there is anything in the C#OPENLEFTQ, call Extend-paths-right on the whole C#OPENLEFTQ with directions set to 2.

(7) If there is anything in the B#Q , call Extend-paths-right on the B#Q with directions set to 1.

(8) Call Doarc for each consuming arc and the appropriate Sconfigs in the DONEQ.

3.4.2 Extend-paths-right

Extend-paths-right adds JUMP paths to Sconfigs in the queue supplied to it, to extend them to states that either begin a consuming arc, push for another constituent that contains the consuming arc, or pop from the current constituent to a higher level that contains (or can then push for) the consuming arc. The routine is used in 3 contexts in Connect-right, so it must have a way of deciding which of the above metods of connection are relevant; it does this by being supplied with a numeric code indicating in how many directions to look for a connection.

If directions is equal to 1, then only states on the same level as the input Sconfigs are considered, and if the value of directions is 2 or 3, all ways of connection are considered. In addition, if the value is 2, there is no way for a Sconfig in the queue to be further extended via a POP arc at the moment, so a new Sconfig is generated.

Extend-paths-right(queue directions)

(1) For each Sconfig in the queue do (2) to (5)

(2) For each state, FROMSTATE, in the J* set of the right end state of Sconfig do (3) to (5)

(3) If FROMSTATE is a left state of any arc in toarcs, then a list of new Sconfigs, each extended with a jump path from the 'rstate' of Sconfig to FROMSTATE is put on the DONEQ.

(4) If 'directions' is 2 or 3, and (FROMSTATE B# J*TOSTATE) holds for any state in J*TOSTATES, then a list of Sconfigs extended from that state to FROMSTATE is put in the B#Q.

(5) If directions is 2 or 3, and (FROMSTATE C# J*TOSTATE) holds for any state in J*TOSTATES, then a list of new Sconfigs extended to FROMSTATE is put in the C#Q, and if 'directions' is 2, a new Sconfig is generated at each J*TOSTATE and put on the B#Q.

3.4.3 Splitc#q

The C#Q contains all segments on the right of the island which are at a lower level than at least one of the consuming arcs for the current word. However, there is a major difference between those segments which are complete at the left end, and those that are not. In the former case, the constituent must be created, and the process resumed at the higher level (Complete-right, 3.4.4). In the latter case, however, all states in the C# set of the right state of the segment must be considered for extension, since the segment has no left context.

The routine splits these two types of segments into the C#OPENLEFTQ and the C#COMPLETEQ.

Splitc#q()

(1) For each Sconfig in the C#Q do (2) to (5).

(2) If the lbdry of Sconfig (LBDRY) is the same as the left boundary of the island, then put Sconfig on the C#OPENLEFTQ. Return.

(3) Collect a set of RSTATES from the union of the right end states of the segments that have right boundary one word to the left of LBDRY.

(4) If the intersection of RSTATES and the UB# set of LSTATE (the left end state of Sconfig) is not empty, then put Sconfig on the C#COMPLETEQ.

(5) If the intersection of RSTATES and the UC# set of LSTATE is not empty, and if Sconfig has not just been put on the C#COMPLETEQ, put Sconfig into the C#OPENLEFTQ, otherwise put a copy of Sconfig into the C#OPENLEFTQ.

3.4.4 Complete-right

In the course of adding a new word to the right of an existing island, Complete-right is called if a segment that is complete at its left end encounters a final state on its right (in C#COMPLETEQ). The segment is popped, and joined to the island in one of two ways; it may be added to an existing segment at the next higher level, or if such a segment does not exist, a new intermediate segment will be created for each arc that can push for the popped segment, and which is reachable from at least one Sconfig to the left.

After a segment has been extended to include the new popped constituent, it is placed back in the TODOQ because it may need to be further extended - the new segment behaves very much like the segments that were initially on the right of the old island.

Complete-right()

(1) Apply Doarc-pop to each Sconfig in the C#COMPLETEQ.

(2) For each popped Sconfig do (3) to (9).

(3) Find all Sconfigs in the old island that end one word before the start of Sconfig. Name these the PUSH-SCONFIGS. Find the left states of all arcs that can push for the new Sconfig, and name these the LSTATES.

(4) For LSTATE in the LSTATES do (5) to (9).

(5) For all PUSH-SCONFIGs whose rstate is the same as LSTATE, extend a copy of the segment by calling Doarc-push, joining it to the popped Sconfig. Put the resulting Sconfigs on the TODOQ.

(6) For each PUSH-SCONFIG whose 'rstate' can B#J* to LSTATE, do (7).

(7) Find all states STATE1 such that (rstate of PUSH-SCONFIG B# STATE1) and (STATE1 J* LSTATE). Create a new segment for each path from each STATE1 to LSTATE, and extend each of these new segments by calling Doarc-push to join to the popped Sconfig. Put resulting Sconfigs on the TODOQ.

(8) For each PUSH-SCONFIG whose right end state can C#J* to LSTATE, do (9).

(9) Find all states STATE1 such that (rstate of PUSH-SCONFIG C# STATE1) and (STATE1 J* LSTATE). Create a new segment for each path from each STATE1 to LSTATE, and extend each of these new segments by calling Doarc-push to join to the popped Sconfig. Put the resulting Sconfigs on the TODOQ.

3.4.5 **Predict-right**

Predict-right returns a list of all consuming arcs that can be reached by any path of non-consuming arcs from the right end of an island. This list is used to form the list of toarcs supplied to Connect-right for adding a new word to the island (see 3.4.1).

It is not sufficient for Predict-right to list all arcs that can be reached from end states of the island; these predictions should be restricted by Sconfigs within the island that are at a higher level than segments on the right end.

For each Sconfig at the right boundary of the island, the routine predicts all consuming arcs reachable by (J* + J*B#J*) from its 'rstate', and if the Sconfig cannot be pushed for by any Sconfig to its left, its J*C#J* consuming arcs are also predicted. Otherwise, for each Sconfig that can push for it, any intermediate consuming arcs between the levels of the two Sconfigs are predicted, and the process is then repeated for the higher level Sconfig, imagining that its 'rstate' is each in turn of the states the lower level Sconfig can pop up to without any intervening consuming arcs.

4. EXTENSIONS TO INTERPRET THE FULL ATN FORMALISM

The paper on which this parser is based, [Woods76], describes an island parsing ATN interpreter which can only operate on ATNs with a specified restricted syntax. There are two main areas of difference between this syntax and the original full syntax as defined in [Woods70], which only considers left to right parsing.

(1) The restricted syntax does not allow for the test on each arc of the network; instead there is a VERIFY action into which all tests have to be put. My interpreter uses the original syntax for tests, (allowing them to appear in SCOPE clauses like actions).

(2) The special arc actions LIFTR and SENDR are not mentioned in [Woods76], and have evidently been excluded because although they are very useful actions (it is difficult to write an ATN grammar for a large subset of English without them), there are difficulties when it comes to implementing them in an island parsing interpreter.

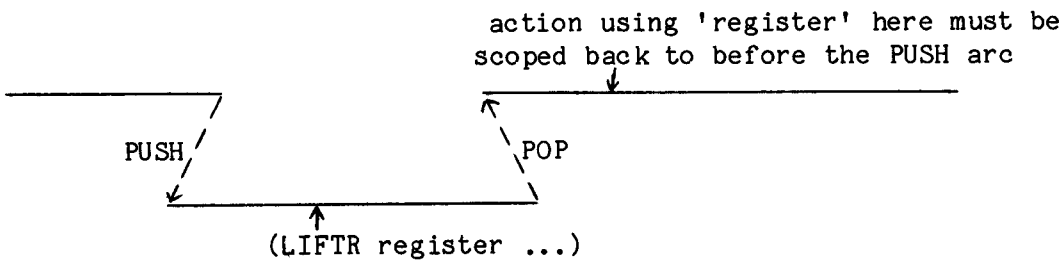
I will now describe the methods I used to implement LIFTR and SENDR and the consequences of introducing these actions into an island parsing interpreter.

4.1 LIFTR Actions

LIFTR can occur on any arc in the network, and it transmits the value of a register up to the next higher level in the network.

When the action is evaluated, it puts the pair (register-name evaluated-form) onto 'liftr-regs' (see 3.1.1) of the Sconfig currently being considered. When the Sconfig is pushed for by a higher level one, 'liftr-regs' are put onto the front of the higher level Sconfig's register context (see 3.2.3). Thus it is easy to implement LIFTR actions.

One has to be careful, however, when inserting scope specifications into the network. If an action (LIFTR register ...) occurs in a sub-network, all actions using that register in all higher level sub-networks that push for the one containing the LIFTR must be scoped so that the actions are not performed in a right to left parse at least until after the PUSH has been executed.



4.2 SENDR Actions

SENDR can only occur on a PUSH arc, and it transmits the value of a register down the push into the lower level network.

Since in an island parser Sconfigs are extended to cover a lower level sub-network before the PUSH to them from a higher level is executed, there is no way of knowing the value of a register that is being SENDRed until at least after the PUSH. Thus all actions involving registers whose values depend on the value of that register must be saved to be executed in the higher level Sconfig. So the actions must be scoped with a special new scope specification, which I shall call scope SENDR. We now have:

<scope-spec> ::= (<state> <state>*) | T | NIL | SENDR

Because there can be any type of interaction at the lower level between registers SENDRed, and registers to be LIFTRed, without doing (in general) a very complicated flow analysis, the only safe scope specification for actions using registers on the PUSH arc (and all actions referencing registers whose values depend on them) is T.

Implementing SENDR involved changes in Doarc-pop and Doarc-push, as I will descibe now.

4.2.1 Changes to Doarc-pop

If any saved actions remain (all of which must have scope SENDR) when all actions have been dealt with, it is likely that the form on the POP arc that is to be returned will need results from the saved actions, so the form is not evaluated immediately, but instead

(SCOPE SENDR (LIFTR * <form on POP arc>))

is added after the last saved action. The current register context is then appended to the context saved with each action, so when the actions are evaluated they pick up the correct register values that were current at the lower level.

4.2.2 Changes to Doarc-push

Before executing any actions on the PUSH arc, any saved actions left in the lower level popped Sconfig are processed.

(1) The scopes of all the actions are changed to that of the SENDR actions on the PUSH arc (if any, otherwise NIL). The scopes will all be the same - either T or SENDR.

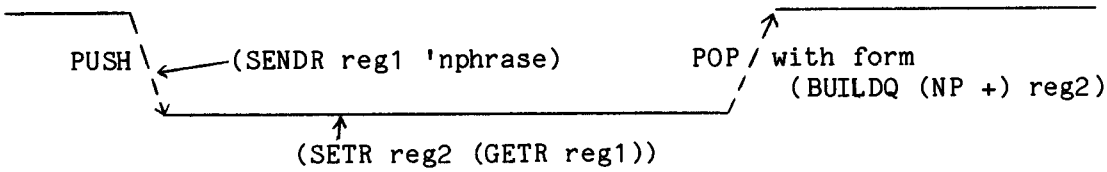
(2) All LIFTR actions are changed to highlvl-setr actions (see below).

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

(4) All the SENDRs on the PUSH arc are put in front of the saved actions.

The rest of the actions on the PUSH arc are then processed as normal. The purpose of the actions lowlvl-start and lowlvl-finish is to 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.

A typical sequence of actions might be:



This would be translated by Doarc-push into the following list of saved actions, and when their scope was satisfied, execution would produce the sequence of operations shown on the right:

	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

5. AUTOMATIC SCOPE COMPUTATION

There is a facility in the interpreter to look at each action in the network, and if it is context-sensitive, to compute its scope, and automatically put the action into a suitable SCOPE clause. [Woods76] claims that:

> "It is possible to construct an algorithm to compute the scopes [of arc actions] automatically if the dependencies of the arc actions are explicitly marked, but we have not yet implemented such a facility".

It is not clear what is meant by "explicitly marking the dependencies of the arc actions", but the algorithm I have developed and implemented requires no work on the part of the user - any ATN grammar written for an ordinary left to right parser can automatically be scoped for island parsing.

5.1 Extracting References to Registers in Actions

Any algorithm to scope ATN grammars will need to find the names of all the registers used in any given action or test on an arc. I have used 'data-driven' programming to do this.

All arguments to each function called in an action or test in the network are assumed to reference one or more registers, either by quoting the name of a register, or via a functional argument. Of course some special functions are exceptions to this rule, and for these an 'extract-registers' function, which when APPLied to each call of the special function returns a list of registers / functional arguments, is PUT on the property list of the special function. For example, the 'extract-registers' function for BUILDQ is CDDR. These functions appear in the same program module as all the network functions, to remind the user that the extraction function of an ATN form/action may need to be changed if the form or action is modified.

5.2 The Scoping Algorithm

The scoping is performed in 5 passes, the first over the whole network at once, dealing with LIFTR actions, the remaining four over each sub-network in turn. The second finds which registers have values which depend on other registers, the third and fourth deal with scoping SENDR actions, and the last does the rest of the scoping. (For the rest of this section, 'action' should be taken to mean 'action and/or test', since they are both treated similarly).

5.2.1 Pass 1

As explained in section 4.1, care has to be taken when scoping a sub-network that pushes to a lower level subnetwork that contains a LIFTR action; it is necessary to ensure that the register that is being LIFTRed is scoped back at the higher level to at least before the PUSH arc.

But if the register is used by an action on the PUSH arc itself, the algorithm should produce correct scope specifications without needing to treat this as a special case. The obvious solution, then, is for each register in a LIFTR on the lower level, to check whether the register appears on the PUSH arc, and if not, the 'dummy' action (SETR register (GETR register)) is added to the PUSH arc actions.

5.2.2 Pass 2

Pass 2 finds, for each sub-network, all the registers whose values depend on other registers, for use in the other scoping passes. It does this by finding for each action the registers used in it, and for each register used PUTs the names of all registers in that action onto the property list of the register under the indicator 'depend-regs'. Thus in the end, 'depend-regs' of each register contains a list of all registers in the sub-network which depend on the value of that register.

Also, for efficiency in the successive passes, each register holds a list of names of arcs which have actions using that register on its property list under the indicator 'arcs-using'. Thus for the actions:

```
(SETR reg1 (BUILDQ (np +) reg2))          on arc G10

(COND
   (reg3 (SETR reg2 reg3)))               on arc G11
```

the data structures built will be:

register	depend-regs	arcs-using
reg1	(reg1 reg2)	(G10)
reg2	(reg2 reg1 reg3)	(G10 G11)
reg3	(reg3 reg2)	(G11)

5.2.3 Pass 3

Pass 3 takes the current sub-network, and scopes all SENDR actions as T. It also scopes as T those actions which reference registers whose values depend on any of the registers used in the actions on the same PUSH arc as a SENDR action.

5.2.4 Pass 4

Pass 4 finds all actions in the current sub-network that use registers that have been passed down from a higher level by a SENDR, and also actions which use registers dependent on those registers SENDRed, giving all the actions scope SENDR.

5.2.5 Pass 5

The remainder of the scoping is performed in pass 5. Each action is considered in turn, collecting the names of registers used in the action, and of all dependent registers. Then the intersection of the states to the left of each action using any of the registers in the list is found (left-states). This list of states represents the common part of all paths from the left to any action dependent on the action being considered. 'Left-states' is used to compute a scope-specification for the action:

(1) if NIL - there are at least two non-intersecting paths to the arc containing the action which reference dependent registers, 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). 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 the action does not use any registers, it obviously is not scoped. 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 <state-list>

5.3 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 be necessary). The main shortcoming is in dealing with networks where there are 2 or more alternative separate paths containing actions using interdependent registers; for example in scoping the following network fragment from the grammar of section 1.1:

```
(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 ... ))
```

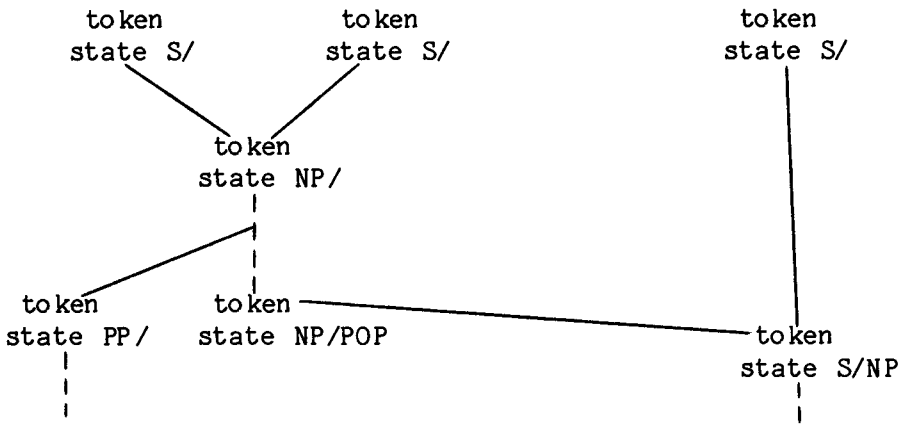
the two actions using register 'noun' are scoped (NP/) but the paths through them are independent, so the optimal scopes are NIL. There does not seem to be any way around this problem by modifying the algorithm, but fortunately scope specifications that are not entirely optimal (as in this case) can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

6. DISPLAYING THE HISTORY OF THE PARSE

On completion of parsing each sentence, the interpreter can optionally generate a lineprinter plot of the paths through the network taken by each complete valid parse.

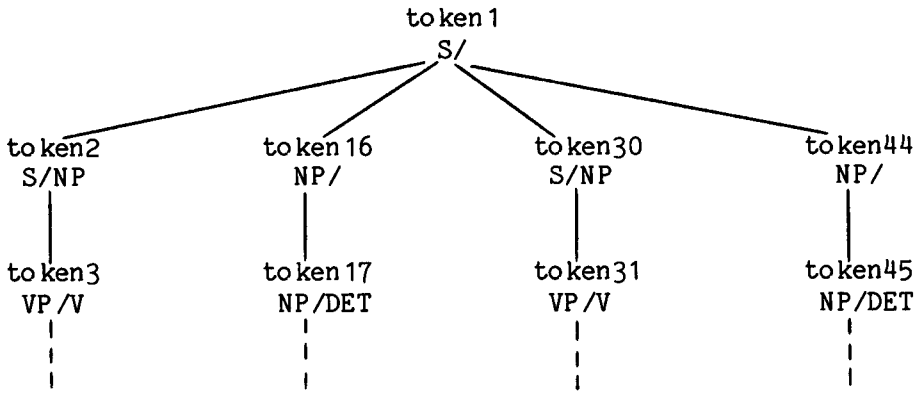
The module that produces the parse-map is a modified version of a collection of routines taken from an existing left to right ATN parser. The routines operate on a set of 'node-tokens' which represent states passed through in the parse, one of which is generated on each arc transition in the original parser to which it belonged. By the nature of depth-first backtracking left-to-right parsing, the tokens can be linked together to form a tree which can be plotted out with the token representing the initial state of the network as the root.

In island parsing, on the other hand, with generating tokens on arc transitions in both directions, tokens have to be saved in Sconfigs, and things become tricky, even before the plotting; the tokens do not form a tree, since a Sconfig (and thus a token) may have more than one 'parent' to its left. So a fragment of a typical set of tokens might be:



and an algorithm to automatically plot this out in general (on a lineprinter) is extremely difficult to devise!

Therefore an alternative approach was implemented, in which each Sconfig carries a list of arcs it has passed through (see 3.1.1), together with the word consumed by each arc (if any). The node-tokens are then generated at the end of each sentence parse, and only for Sconfigs representing valid parses. Only one token is generated for the initial network state, and thereafter all the parses appear as separate chains:



This does not contain any information as to the unsuccessful parse paths, or the common parts of the successful ones, as would the first method described above. However, the valid paths are easier to see in this second method since they are shown as separate, and are not converging and diverging chains of tokens.

The information printed out with each token is:

(1) type of arc in transition
(2) name of state at end of transition
(3) word consumed by the arc (if any).

7. PARSER CONTROL STRUCTURE

7.1 Interface between High and Low Level Functions

All the functions described in chapter 3 perform low level tasks, being only concerned with predicting reachable consuming arcs from the ends of an island, and so on. The functions Synseed, Synevent, Syncollision, and Synend described here form an interface between these low level functions and the high level functions of section 7.2, which deal with whole sentences and allow different global parsing strategies to be experimented with.

Synseed takes a word at position 'word-number' in the sentence, calls Startisland to create an island, and sets up the left and right island boundaries to be at 'word-number'.

Synevent takes a word and the side of an island onto which it is to be added, and calls either Connect-left and Predict-left or Connect-right and Predict-right, forming the toarcs for the Connect functions (see 3.4.1) by computing the intersection between the arcs predicted and the list of arcs which can consume the word. Synevent then sets up the new island boundaries.

Syncollision joins two islands together, calling Connect-collide, supplying it with the right boundary of the left-hand island and a list of Sconfigs on the left of the other island. Connect-collide merges together all compatible Sconfigs from the two islands, and then calls Complete-right as many times as possible on the merged Sconfigs (to splice them into higher levels). Woods, on the other hand, produces the identical effect by individually adding the words from the smaller island onto the end of the larger. This method is obviously less efficient since all the effort in building the smaller island is wasted.

Synend calls Connect and Predict routines to extend an island covering the whole input sentence back to the initial state of the network, and then forward to a set of final POP arcs at the top level. It then prints out the parse trees of all the valid parses (i.e. 'liftr-regs' of all popped Sconfigs that have the initial network state as their left-most state, and whose boundaries cover the whole island), and optionally a history of the parse (chapter 6).

After each call of one of the Connect functions, the interface functions call a routine to eliminate duplicate Sconfigs - to avoid duplicate parses and combinatorial explosions in future extensions of Sconfigs. Duplicates arise when a new word can be connected to an island by more than one combination of pushes and pops, since the low level functions create one Sconfig for each way of reaching a consuming arc for the word.

7.2 High level Parsing Strategies

I have at present implemented two basic parsing strategies, but the program is easily extensible to admit additional ones - in particular an experimental method for parsing conjunctions elegantly was planned [1] but not enough time was available to carry it out. The two current strategies, or modes, in which the parser can work are now described.

7.2.1 **Normal Mode**

Immediately after inputting a sentence, the user specifies the order in which the words in the sentence are to be presented to the parser. If a new word to be parsed is not adjacent to any existing island, it is made into a new island which is merged with others when the intervening words have been processed. Thus the input to the interpreter of

```
Time flies like an arrow.
 1    3      2   5   4
```

means that the interpreter should consider first "time", then "like", and then "flies", "arrow" and "an", in that order. This implies that the program will build islands for "time" and "like", and will then add "flies" onto "time" and merge the island with "like". "Arrow" will be an island, but "an" will join onto "time flies like" and the whole of this will then be merged with "arrow".

An artificial user-driven strategy like this for indicating the order in which words are to be considered is of course necessary for test purposes, where there is no independently generated word ordering as there would be in a speech processing system.

7.2.2 **Default Mode**

If no parse order is specified as in 7.2.1, the sentence is parsed in strictly left to right order, with the first word in the sentence as the seed of the one and only island.

7.3 Control Functions

These are the top level routines called by the user to direct the flow of control of the parser. There are three main ones, corresponding to the virtually independent tasks of initialisation, scoping the network, and parsing a set of input sentences.

---

[1] Although the ATN formalism is quite powerful in expressing natural language grammars, it faces problems dealing with sentences containing conjunctions, since (WRD AND ...) arcs need to be inserted almost everywhere because AND can conjoin any two constituents of the same type together. Bran Boguraev has proposed that an island parsing strategy might be able to overcome this.

Initialise calls routines to read the files containing the network and the dictionary, and to set up the data structures described in sections 2.1 and 2.2.

Scope may be called after initialisation of the network, and inserts suitable SCOPE clauses into it using the algorithms described in Chapter 5.

Parse reads a file containing a set of sentences to be parsed (see 2.3), and calls routines to parse each one with the appropriate strategy. If an error occurs in processing one sentence, the interpreter skips it and goes on to the next one.

The Cambridge LISP system [Norman] allows a core image to be saved to a disc file at the end of a run by the function call PRESERVE; the image can then subsequently be reloaded and computation carried on starting with the state of the system when the image was saved. Two auxiliary top level functions are provided to exploit this facility - if the user has an image from a previous run of the interpreter, Grammar-reload can be called to alter a specified set of states which were previously initialised; Dict-reload performs a similar task for words in the dictionary.

The final top level function is Print-network which prints out the ATN grammar that is stored internally - this can be used as a check that the automatic scoping has worked correctly.

There are also three user-settable flags controlling the amount and form of the output from the interpreter. The switch *debug-level gives the user a means by which variable amounts of information on the contents of the data structures can be output to a separate file at strategic points in the execution of the interpreter.

The generation of a map showing the parse history of each input sentence is controlled by the flag *plot-flag; when set to a non-NIL value, a plot for each sentence is computed, and each is written to a different member of an output pds for subsequent printing.

The user can also control the amount of statistics printed out after each parse of a sentence with the switch *statistics-level. The basic information that is always given is the total number of Sconfigs created during the parse, the number remaining at the end, and the CPU time used in processing the sentence. Additional levels of statistics give information on the number of network arcs processed, and on the number of actions and tests evaluated.

An example of a typical calling sequence for simple use of the interpreter is:

```
(SETQ *debug-level 0)          - no debug output
(SETQ *plot-flag T)            - parse history plots required
(SETQ *statistics-level 2)     - all statistics required

(Initialise)
(Scope)
(Parse)
```

7.4 Errors

During execution, the interpreter recognises three basic types of error:

(1) normal user error - input format errors in grammar, dictionary, or sentences to be parsed; inability to extend an island to any of the next set of consuming arcs for the next word in a parse, etc.

(2) fatal user error - an attempt to perform the scope computation, or parse a set of sentences, before initialisation has been caried out.

(3) internal interpreter inconsistency - checked for at entry-points to certain functions, i.e. Doarc checks that it is called on an arc that is adjacent to the end of a Sconfig.

An appropriate message and a LISP system backtrace are printed on detection of an error, and if the error occured during initialisation or scoping, execution halts with a return code of 20. All errors during parsing a sentence cause the sentence to be abandonned, and the final return code to be set to 16. Fatal user errors cause to interpreter to immediately halt with a return code of 24.

7.5 Interpreter Output

For each sentence parsed, apart from any debug or a plot of the history of the parse (which go to different files), the interpreter outputs the following to the standard output stream:

(1) the parsing strategy and the order of adding words to the island.

(2) the number of parses found, and the parse tree associated with each.

(3) a variable amount of statistics (depending on the value of *statistics-level). With *statistics-level=2, typical statistics might be:

```
Actions and tests evaluated - 1382
   saved actions performed - 741
   saved tests performed - 243

Arcs processed
  consuming and jump arcs - 367
  push arcs - 195
  pop arcs - 65

501 segments created - parse ended with 55 remaining

Parsing sentence took 4295+2187 msecs
```

8. EXPERIENCE OF USING THE INTERPRETER

8.1 Observations

To date, the interpreter has been used to parse only a small number of sentences, but half of these had alternative syntactic structures. Of course, the testing was of the interpreter and not of the grammar, so a small set of sufficiently varied sentences, together with a large enough grammar, was a sufficient test of the robustness of the interpreter.

Two ATN grammars, both provided by Bran Boguraev, were used. The smaller, purely syntactic network of approximately 80 lines was an ideal vehicle for preliminary program testing, although it is not able to describe a sufficiently large subset of English to be useful for any interesting work. The larger grammar is 300 lines long, and it is hoped that the interpreter will prove a useful research tool using this grammar and variants of it.

It was hoped that the parser could interpret a particular well-tested and very comprehensive ATN (available in Cambridge and about 1000 lines long), but this proved to be impossible given any reasonable constraints on CPU time and storage. The problem is that execution times for parsing sentences obviously depend largely on the number of Sconfigs present at any one time; Sconfigs tend to proliferate embarassingly when there are many possible paths of JUMP arcs between states on the same level - which is the case with this grammar. To compound the problem, nearly all the tests on the JUMP arcs have to be scoped T or SENDR, so virtually none of the paths can immediately be eliminated. This problem is not solely a property of my interpreter, but of island parsing in general when using large complicated grammars.

The above is the only limitation in the interpreter that has been found so far (and an unavoidable one at that). This problem is also encountered when parsing sentences using the larger 300 line grammar, but in a more manageable form. A typical sentence takes about 5 secs CPU time in 900 Kbytes of store on the Cambridge IBM 3081D. This is about the minimum amount of store for the interpreter when using this grammar, since it generates of the order of 500 segments for each sentence, finishing with typically about 50 segments remaining. The small (80 line) grammar, however, can be used with the interpreter in 500K, and parsing times are very quick, taking only up to 2 secs CPU time for a sentence of arbitrary complexity.

As might be expected, parsing the same sentence with different orders of adding the words in it to an island can result in differing numbers of Sconfigs being created. For example, when parsing one particular sentence, in a straight right to left parse 15% fewer Sconfigs are created than in a left to right parse, with a corresponding 10% decrease in execution time

for the sentence. However, the numbers of arcs processed in the two cases are almost the same. Some typical program runs are included in Appendix 1.

8.2 Optimisations

The rather large amount of storage required by the interpreter when using the larger 300 line grammar could be reduced by decreasing the size of Sconfigs. The current representation of saved actions and tests includes a copy of the actual actions or test; but if the copy were replaced by a pointer to where the actions / test occur in the grammar, I estimate that the interpreter could be run in only about 700K for the most complex sentences. When designing the parser's data structures in the first place, however, I did not foresee that they might need to be as compact as possible, but due to the modularity of the interpreter, these modifications to saved actions and tests would not be difficult.

8.3 Possible Future Developments

From the outset, the interpreter was designed to be used as a research tool. Island parsing appears to be a promising solution to the problem of parsing sentences containing conjunctions (see 7.2), avoiding the need to explicitly include conjunction consuming arcs in the grammar by using a conjunction as the island seed, and parsing outwards from it.

Island parsing was originally put forward, though, as a solution to problems in speech recognition, where the first word of a sentence may not necessarily be recognised, but could be deduced from starting the parse from one or more places in the middle of the utterance with words that **have** been recognised. This involves a control strategy (7.2) that I have not implemented, but using the flexibility of the interface (7.1) to the low level functions, it would be very easy to implement.

Another extension, argued for persuasively in [Woods80] (in terms of ordinary left to right parsers), is cascaded ATN interpreters; several island parsers could be put on top of each other, each having a separate domain of reponsibility and each passing up completed constituents to the next higher ATN. SENDR actions might cause problems if scoping 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.

9. ACKNOWLEDGEMENTS

I would like to thank Bran Boguraev for all his help in this project: for supplying the 2 ATN grammars I have used; for his comments on this paper in its original form as a dissertation; and for the use of his LISP macro package which greatly simplified the writing of the interpreter.

Thanks also to John Tait for his preliminary runs of my interpreter on the larger ATN grammar in store sizes of up to 1 Megabyte, and to Karen Sparck Jones for her helpful comments on the revision of this paper from the original dissertation.

10. REFERENCES

[Norman]
Norman A. Local Cambridge LISP documentation. See ALG.LISP.DOCUMENT on the IBM 3081.

[Winston81]
Winston P. and Horn B. "LISP". Addison-Wesley, 1981.

[Woods70]
Woods W. "Transition Network Grammars for Natural Language Analysis". Communications of the ACM, vol.13, 1970.

[Woods76]
Woods W. et al. "Parsers". Bolt, Beranek and Newman Inc. report No.3438, vol.4, section A, 1976.

[Woods80]
Woods W. "Cascaded ATN Grammars". American Journal of Computational Linguistics, vol.6, no.1, 1980.

APPENDIX 1

Example runs of the interpreter.

Run 1

This run used the small (80 line) grammar given in section 1.1. The sentence to be parsed (in file .sentence) was 'Time flies like an arrow', in normal mode, with word order as in the example in section 7.2.1. The following top level calls (in file .parse) were employed to perform initialisation, scope the network, print out the scoped network, and parse the sentence:

```
(SETQ *debug-level 0)
(SETQ *plot-flag NIL)
(SETQ *statistics-level 1)

(Initialise)

(Scope)

(Print-network)

(Parse)
```

The dictionary supplied to the parser (in file .dict) was:

```
(Time
   (cat NOUN VERB ADJ))
(flies
   (cat NOUN VERB))
(like
   (cat VERB PREP))
(an
   (cat DET))
(arrow
   (cat NOUN))
```

The fact that three of the words in the sentence belong to more than one syntactic category means that the interpreter can be expected to find more than one valid parse for the sentence. (In fact, it found **four** parses).

The run was performed in a job, in order to use 500K of store, and the Phoenix command to run the interpreter was:

```
alg.lisp.loader(lisp) (sysin .parse
                       %S=500
                       sysprint *A
                       image .image
                       text .sentence
                       dict .dict
                       grammar .grammar)
```

```
CAMBRIDGE LISP SYSTEM ENTERED IN ABOUT 500 KBYTES
CORE IMAGE WAS MADE AT  23.38.14    ON    1 NOV 82
LISP VERSION - VER2 LEV2   IMAGE SIZE = 127680 BYTES

STARTED AT 10.53.24 ON 2 NOV 82 AFTER 0.21 SECS - 58.5% STORE USED


(SETQ *debug-level 0)
(SETQ *plot-flag NIL)
(SETQ *statistics-level 1)

(Initialise)
Initialisation successfully completed after 246+276 msecs


(Scope)
Scoping pass 1 commencing
Scoping commencing for sub-network starting at NP/
Scoping commencing for sub-network starting at PP/
Scoping commencing for sub-network starting at S/

Scoping successfully completed after 262+302 msecs


(Print-network)
(S/
   (PUSH NP/
      T
      (SETR subj (BUILDQ (subj *)))
      (SETR type 'declarative)
      (TO S/NP))
   (JUMP S/NP
      (CAT VERB)
      (SETR type 'imp)
      (SETR subj '(NP (pro YOU)))))
(S/NP
   (CAT VERB
      T
      (SETR verb (BUILDQ (verb *)))
      (TO VP/V)))
(VP/V
   (PUSH NP/
      T
      (SETR obj (BUILDQ (obj *)))
      (TO S/POP))
   (JUMP S/POP
      T
      NIL))
(S/POP
   (PUSH PP/
      T
      (SCOPE (S/NP S/ VP/V) (SETR mods *))
      (TO S/POP))
   (POP
      (BUILDQ
         (S (type !+) !+ (VP !+ !+ !+))
         type
         subj
         verb
```

```
         obj
         mods)
      T))
(PP/
   (CAT PREP
      T
      (SETR prep *)
      (TO PP/PREP)))
(PP/PREP
   (PUSH NP/
      T
      (SETR np *)
      (TO PP/POP)))
(PP/POP
   (POP
      (BUILDQ (PP (prep !+) (np !+)) prep np)
      T))
(NP/
   (JUMP NP/DET
      T
      NIL)
   (CAT DET
      T
      (SETR det *)
      (TO NP/DET))
   (CAT NPR
      T
      (SETR noun (BUILDQ (npr *)))
      (TO NP/POP)))
(NP/DET
   (CAT ADJ
      T
      (SCOPE (NP/) (SETR adj *))
      (TO NP/DET))
   (CAT NOUN
      T
      (SCOPE (NP/) (SETR noun *))
      (TO NP/POP)))
(NP/POP
   (PUSH PP/
      (CAT PREP)
      (SCOPE (NP/ NP/DET) (SETR mods *))
      (TO NP/POP))
   (POP
      (BUILDQ
         (NP (det !+) (adj !+) (noun !+) (mods !+))
         det
         adj
         noun
         mods)
      T))


(Parse)
```

```
Sentence:  Time flies like an arrow.

Normal mode:          Time
                          like
                        flies
                --- coalescing islands ---
                              arrow
                            an
                      --- coalescing islands ---
*** GC 48 (AFTER 0.41+0.36 SECS:INTERPRETER QUOTE) 81.5% STORE USED
*** GC 49 (AFTER 0.50+0.69 SECS:LIST) 84.9% STORE USED


Number of valid parses found : 4

(S
   (type declarative)
   (subj (NP (noun Time)))
   (VP
      (verb fly)
      (PP (prep like) (np (NP (det an) (noun arrow)))) ))

(S
   (type declarative)
   (subj (NP (adj Time) (noun fly)))
   (VP (verb like) (obj (NP (det an) (noun arrow)))) )

(S
   (type imp)
   (NP (pro YOU))
   (VP
      (verb Time)
      (obj
         (NP
            (noun fly)
            (mods
               (PP
                  (prep like)
                  (np (NP (det an) (noun arrow)))) ))) ))

(S
   (type imp)
   (NP (pro YOU))
 . (VP
      (verb Time)
      (obj (NP (noun fly)))
      (PP (prep like) (np (NP (det an) (noun arrow)))) ))


Arcs processed
   consuming and jump arcs - 25
   push arcs - 29
   pop arcs - 14

132 segments created - parse ended with 18 remaining

Parsing sentence took 243+642 msecs
```

Run 2

This second run used the large 300 line grammar, initialising, scoping, and parsing the 4 sentences:

```
JOHN IS EASY TO PLEASE.
 2    1   3   4   5

JOHN IS EAGER TO PLEASE.
  5   4   3   2    1

JOHN NOT EAGER!

THE GIRL'S CAT IS NOT LOST.
 4   2      1  3   5    6
```

with the top level calls (in file .parse):

```
(SETQ *debug-level 0)
(SETQ *plot-flag NIL)
(SETQ *statistics-level 2)

(Initialise)

(Scope)

(Parse)
```

and the dictionary:

```
(JOHN
   (cat NPR))
(IS
   (cat VERB)
   (features TRANS)
   (uninflected . BE))
(BE
   (features COPULA))
(EASY
   (cat ADJ)
   (features LOWOBJ))
(TO
   (cat PREP))
(PLEASE
   (cat VERB)
   (features TRANS INTRANS))
(EAGER
   (cat ADJ)
   (features LOWSUBJ))
(THE
   (cat DET))
(GIRL
   (cat NOUN))
(CAT
   (cat NOUN))
(NOT
   (cat NEG))
(LOST
   (cat VERB ADJ)
   (features TRANS INTRANS)
   (uninflected . LOSE)
   (pastpart))
```

The first two sentences have identical 'surface structure', but the ATN grammar is able to deduce from the feature-list of the adjectives EASY and EAGER that the sentences have a different 'deep structure', viz.

```
(Other people find that) JOHN IS EASY TO PLEASE.
JOHN IS EAGER TO PLEASE (other people).
```

The third (ungrammatical) sentence is not recognised by the grammar, and the fourth is found to have both an active and a passive parse. The error in the third sentence causes a return code of 16 to be set.

```
CAMBRIDGE LISP SYSTEM ENTERED IN ABOUT 900 KBYTES
CORE IMAGE WAS MADE AT  23.38.14    ON    1 NOV 82
LISP VERSION - VER2 LEV2   IMAGE SIZE = 127680 BYTES

STARTED AT 11.04.38 ON 2 NOV 82 AFTER 0.16 SECS - 32.4% STORE USED


(SETQ *debug-level 0)
(SETQ *plot-flag NIL)
(SETQ *statistics-level 2)

(Initialise)
*** GC 35 (AFTER 0.38+0.23 SECS:BUFFER SPACE) 40.5% STORE USED

Initialisation successfully completed after 582+515 msecs


(Scope)
Scoping pass 1 commencing
Scoping commencing for sub-network starting at compl/
Scoping commencing for sub-network starting at pp/
Scoping commencing for sub-network starting at rel/
Scoping commencing for sub-network starting at np/
Scoping commencing for sub-network starting at s/
Scoping commencing for sub-network starting at initial/

Scoping successfully completed after 1540+529 msecs


(Parse)
```

```
Sentence:  JOHN IS EASY TO PLEASE.

Normal-mode:            IS
                      JOHN
                         EASY
*** GC 36 (AFTER 1.68+0.61 SECS:CALL TO DELEQ) 50.0% STORE USED
                               TO
                                  PLEASE
*** GC 37 (AFTER 2.87+1.08 SECS:CALL TO APPEND) 58.4% STORE USED


Number of valid parses found : 1

(S
   (tns present)
   (subj (Np (npr JOHN)))
   (Vp
      (v BE)
      (pred-adj EASY)
      (compl
         (Compl
            to
            (S
               (tns present)
               (subj (Np (pro dummy)))
               (Vp
                  (v PLEASE)))) )))



Actions and tests evaluated - 341
   saved actions performed - 46
   saved tests performed - 12

Arcs processed
   consuming and jump arcs - 274
   push arcs - 140
   pop arcs - 7

388 segments created - parse ended with 35 remaining

Parsing sentence took 1770+998 msecs
```

```
Sentence:  JOHN IS EAGER TO PLEASE.

Normal-mode:                    PLEASE
                              TO
                            EAGER
                          IS
                        JOHN


Number of valid parses found : 1

(S
   (tns present)
   (subj (Np (npr JOHN)))
   (Vp
      (v BE)
      (pred-adj EAGER)
      (compl
         (Compl
            to
            (S
               (tns present)
               (subj (Np (npr JOHN)))
               (Vp
                  (v PLEASE)))) )))



Actions and tests evaluated - 194
   saved actions performed - 46
   saved tests performed - 12

Arcs processed
   consuming and jump arcs - 132
   push arcs - 71
   pop arcs - 7

182 segments created - parse ended with 23 remaining

Parsing sentence took 1088+0 msecs
```

```
Sentence:  JOHN NOT EAGER.

Default-mode:          JOHN
                         NOT
                           EAGER

+++Error: no reachable consuming arcs predicted for word
EAGER attempting to add word onto right of island


ERROR NIL
STYLE 5 BACKTRACE FOLLOWS:
  ARG3: right
  ARG2: EAGER
  ARG1: 3
Report-error (COMPILED CODE: 3 ARGS)
  w-number: 3
  word: EAGER
  ARG1: NIL
Connect-right (COMPILED CODE: 3 ARGS)
  ARG2: right
  ARG1: EAGER
Synevent (COMPILED CODE: 2 ARGS)
Parse-sentence-default (COMPILED CODE: 0 ARGS)
  ARG1: default-mode
Parse-sentence (COMPILED CODE: 1 ARGS)
Parse-sentence (BEING INTERPRETED)

END OF BACKTRACE


Sentence abandonned due to error
```

```
Sentence:  THE GIRL'S CAT IS NOT LOST.

Normal-mode:                CAT
                          POSS
                        GIRL
                             IS
                      THE
                                NOT
*** GC 38 (AFTER 4.63+1.54 SECS:CALL TO COPY) 47.8% STORE USED
                                  LOST
*** GC 39 (AFTER 5.71+1.91 SECS:INTERPRETER QUOTE) 58.1% STORE USED
*** GC 40 (AFTER 6.37+2.45 SECS:CALL TO APPEND) 67.3% STORE USED


Number of valid parses found : 2

(S
   (tns present)
   (subj
      (Np
         (det THE)
         (poss (Np (det THE) (n GIRL)))
         (n CAT)))
   (Vp (v BE) (neg T) (pred-adj LOSE)))

(S
   (aspect passive)
   (tns present)
   (subj (Np (pro dummy)))
   (Vp
      (v LOSE)
      (neg T)
      (obj
         (Np
            (det THE)
            (poss (Np (det THE) (n GIRL)))
            (n CAT)))) )



Actions and tests evaluated - 1241
   saved actions performed - 771
   saved tests performed - 231

Arcs processed
   consuming and jump arcs - 263
   push arcs - 126
   pop arcs - 7

315 segments created - parse ended with 27 remaining

Parsing sentence took 2759+1508 msecs


STOPPING WITH RETURN CODE=16 AFTER 7.20+3.05 SECS - 85.8% STORE USED
```