This is a reformatted version of the paper: Timothy W. Finin and George D. Hadden, Augmenting ATNS, Fifth International Joint Conference on Artificial Intelligence, 1977.

Augmenting ATNs

Tim Finin and George Hadden
Coordinated Science Laboratory
University of Illinois, Urbana, Illinois 61801

Augmented Transition Networks (ATNs) have become a popular and effective tool for constructing natural language understanding systems. Our own system, PLANES [Waltz 76], is based on a large "semantic grammar" which is implemented as an ATN network. In developing our system, we were frustrated by numerous problems and shortcomings in the basic ATN formalism. Consequently, we have augmented and extended the model to overcome some of these problems. We have also developed an optimizing compiler for ATN networks which translates them into LISP functions and an interactive program which allows one to easily extend and modify an ATN grammar.

Extensions to the ATN Model

Some of the extensions are relatively simple: the inclusion of new arc types such as PHRASE (which matches two or more input words) and ROOT (which matches on the root forms of the current input word). Others have involved re-examining the basic ATN mechanism in order to alleviate more fundamental problems.

We have extended our ATN system to provide several control primitives which can be used to dynamically prune the tree of decision points on the developing "parse." One primitive, FAIL (included both as an arc type and as an arc action) can propagate a failure message to any point on the decision tree. Other primitives allow one to save environments, both control and variable binding. This allows one to suspend the processing activated by a specified PUSH arc or action and to continue parsing as if it had failed. If subsequent attempts to find a Parse fail, the suspended processing will automatically be resumed in the same environment.

Even with the ability to control backup, we have found that the ATN model still suffers from a paucity of control primitives. Evaluating an ATN state is conceptually a simple process in which one evaluates each arc leaving a state until one is found which can be taken (leading to a new state to be evaluated) which does not return failure. Thus the basic control primitive is the IF-THEN-ELSE-IF-ELSE... applied to the arcs of a state. We have found it conven-

ient to add the ability to conjoin or disjoin bundles of arcs leaving a state. For example, it is quite common to have a group of arcs leaving a state in which only one arc can possibly be taken.

In the standard ATN formalism, there is no easy way to encode the knowledge that if one of these arcs can be followed but eventually leads to failure, then the other arc(s) cannot possibly be followed. Our ATN system supports a SELECT construct, which will only allow one arc in the bundle to be followed. The SELECT construct has the additional benefit of signaling to the reader that the governed arcs are locally exclusive possibilities, distinguishing some of the deterministic from the non-deterministic aspects of a grammar.

In our own work we use long sequences of single-arc states to parse special constructions or idiomatic phrases. This quickly results in a spaghetti-like network which is difficult to read and understand. To avoid this situation and to keep together code which logically belongs together, we have defined an AND construct which conjoins a bundle of arcs into a single unit. The destination of each arc in the bundle (except for the last arc, of course) is just the next arc in the bundle.

These extensions to the ATN model have been built into an efficient interpreter (written in MACLISP) which uses a simple recursive technique for simulating non-deterministic automata.

Compiler for ATNs

Another major component of our ATN system is an optimizing compiler which translates ATN networks into compact, efficient LISP code. This LISP code can then be passed through a standard LISP compiler to produce machine language code which achieves a speed-up factor of 5 to 10 and a size reduction factor of up to 1/2.

The translation of an ATN network into LISP code is straightforward process, each state in the network resulting in a simple LISP function. Our compiler, however, attempts to produce efficient code by applying three sets of optimization (expressed in a simple pattern matching language) rules at different stages of the translation.

The first set of rules defines source-to-source transformations on ATN networks and individual ATN states. For example, one rule will bundle any adjacent word arcs of a state into a single SELECT arc if the set of words

tested by the arcs are disjoint.

The second set of optimization rules is applied as the ATN network is translated into LISP code. The domain of these rules is the set of ATN arcs. For example, one rule carefully examines the condition and actions on an arc to determine if they might cause a side effect (e.g., set a register). If no side effects are possible, then code for generating a new register context need not be included.

The final set of optimization rules defines LISP source-to-source translations, which result in faster and/or more compact code. This set of rules also includes transformations which can optionally “open code” the more common built-in ATN actions (e.g., setting or accessing a register).

Network Editor

We have developed a special purpose editor, NETEDI [Waltz 76] [Hadden 77], which knows about the structure of ATNs and takes advantage of their redundancies.

NETEDI expects as its main input a list. Each element of this list corresponds to a new arc in the ATN with the following exception: if some initial segment of the input list would correspond to a set of arcs already in the ATN; no new arcs are added for these. The input list can be as simple as a list of words which form a sentence or phrase or complex enough to form any possible arc complete with embedded tests, LISP expressions, and transition modes.

Consider one of the elements of the input list. If it is an atom which begins with a "&", it represents a PUSH arc. When NETEDI sees one of these, it adds register-setting code to the PUSH arc; the register will have the same name as the subnet PUSHed to. Any other atom in the input list represents a WRD arc.

Any arc (including WRD and PUSH) can be represented by including its name as the first element in a list, preceding the name with a “:” and putting the rest of the arguments in the list. For example, "(:CAT PREP)" would form a CAT (for category) arc which recognizes prepositions. So “((:CAT PREP) &NP)” would produce the code required to recognize prepositional phrases assuming the ATN contained a state called “NP” which is the first state of a noun phrase recognizer.

Acknowledgements

The work described in this report was supported by the office of Naval Research under contract number N0001*l-67-A-0305-0026.

References

[Gabriel 75] Gabriel, R.P. and Finin, T.W., The LISP Editor, Working Paper 1, Advanced Automation Group, Coordinated Science Laboratory, University of Illinois, 1975.

[Hadden 77] Hadden, G.D.; NETEDI: An Augmented Transition Network Editor, MS Thesis, University of Illinois: 1977.

[Sussman 72] Sussman, G.J.and McDermott, D.V.: From PLANNER to CONNIVER — A Genetic Approach; Proceedings of the Fall Joint Computer Conference, 1972.

[Waltz 76] Waltz, D.L. et al., The Planes System: Natural Language Access to a Large Data Base, Coordinated Science Laboratory Report No. T-034, University of Illinois, 1976.