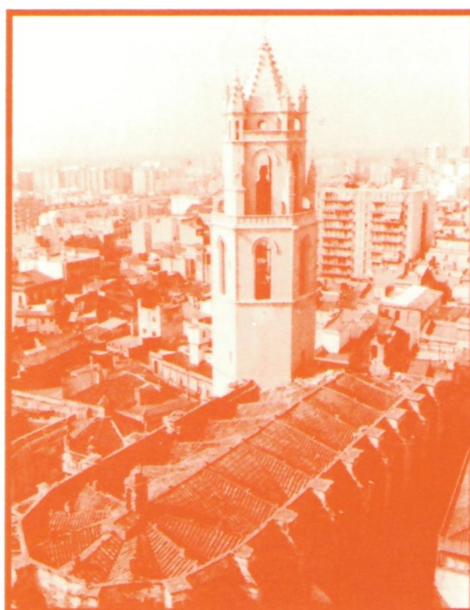
LENGUAJES NATURALES
Y
LENGUAJES FORMALES
IX

Carlos Martín Vide, Ed.



Actas del

IX CONGRESO DE LENGUAJES NATURALES Y LENGUAJES FORMALES

IX CONGRÉS DE LLENGUATGES NATURALS I LLENGUATGES FORMALS

UNIVERSITAT ROVIRA I VIRGILI
SECCIÓ DE LINGÜÍSTICA GENERAL

Understanding English Specification of Finite State Devices

Robert Wall and Jim Talley
Dept. of Linguistics
University of Texas at Austin
lifz050@utxvms.cc.utexas.edu
jtalley@emx.cc.utexas.edu

This paper describes a system which receives a specification of a regular language, in English (for example, "all strings over the alphabet $\{a, b\}$ which do not contain the string *bab*") and constructs a representation of a finite-state automaton (FSA) which accepts the language specified. The system uses a strategy for domain-specific natural language understanding (NLU) where syntactic processing is based on a grammar which conflates syntactic, semantic, and pragmatic concerns and relies on a strong underlying semantic model to simplify the parsing task. The system has possible applications as an aid in teaching automata theory and might also be regarded as a miniature system for constructing computer programs in natural language.

Introduction

In a first course in automata and formal language theory, one learns a number of algorithms for manipulating finite-state automata (FSA's): conversion of a non-deterministic FSA to an equivalent deterministic one; minimizing the number of states of a deterministic FSA; construction of an FSA accepting the union, intersection, complement, concatenation, Kleene closure, prefix, etc. of the languages accepted by some given FSA's; and so on. One also learns how to convert between a regular expression, such as $(a \cup b^*)ba$, and an FSA accepting the regular language so represented. All this is tidy, algorithmic, and constructive. The beginning student quickly learns, however, that when natural language intrudes--when a regular language is specified in English, for example--one has to rely, for the most part, on guesswork and rules of thumb to solve the problem. For example, how does one construct an FSA accepting "all strings not containing the substring *bab* and ending in at least two *b*'s"?

The Approach

Our goal was to construct a natural language understanding (NLU) system that would bring English definitions of regular languages into the algorithmic domain. Specifically, we wanted to build a system that would, for example, when given a specification such as:

(1) $L_1 = \{w \in \{a, b\}^* \mid w \text{ contains no occurrence of } bab\}$

produce a FSA which accepts exactly L_1[1]. In our system we assume that the input specifications are always given (effectively) in the form:

(2) $L = \{w \in \Sigma^* \mid \ldots\phi(w)\ldots\}$

where $\phi(w)$ is an English expression giving necessary and sufficient conditions for any string w to be a member of L. Our approach is to parse $\phi(w)$ with the help of a semantically-based grammar and then translate this parse into an expression in an intermediate "operator" language, from which the desired FSA is constructed. We will give more details of the parsing and translation routines later, but to illustrate, the specification in (1) would be translated into the expression shown in (3):

(3) (Complement (Contains "bab"))

To convert this into an FSA, the program proceeds from the inside out. First "bab" is evaluated as a deterministic FSA accepting just the string *bab*, i.e., the language {bab}. (The program does not deal with diagrams--at least not yet--but instead represents FSA's as a LISP structure containing the alphabet, the set of states, the transitions between states, etc.)

The one-place operator `Contains`, given any FSA M as argument, places the FSA Σ^*, accepting all strings over the alphabet, both before and after M connected to it by e-transitions (reading the empty string). The result is a (non-deterministic) FSA accepting the set of all strings containing *bab* as a sub-string.

In preparation for the next step, this automaton is then converted to an equivalent deterministic FSA and the number of states minimized. (Of course it is not absolutely necessary to minimize the states of M here, but it helps to avoid explosions in running time as the complexity of the FSA's increases.) The one-place operator `Complement`, given an FSA M as argument, then simply reverses accepting and non-accepting states to give a (deterministic, minimum-state) FSA accepting L_1.

As another example, take the language L_2 in (4).

(4) $L_2 = \{w \in \{a, b\}^* \mid w \text{ ends in at least two } b\text{'s}\}$

This is translated into the following operator expression:

(Ends-With (At-Least-N 2 "b"))

To produce the FSA from this expression, first "b" is evaluated as a (deterministic) FSA accepting exactly `{b}`. The two-place operator `At-Least-N`, expects a pair (n, M) as argument, where n is a positive integer and M is an FSA. It then, in effect, chains n copies of M together followed by an automaton accepting $L(M)^*$, i.e., the Kleene closure of the language accepted by M. In the case at hand, this gives an FSA which accepts bbb^*, i.e., the set of all strings consisting of at least two b's.

However, there is a problem here which we have glossed over. The English expression "at least two b's" is ambiguous--must the b's be contiguous or may they be separated by some string of symbols? We will return to this and some similar problems below, but for now let us assume that the intended construal here is "at least two contiguous b's." The one place operator `Ends-With` then precedes the FSA accepting bbb^* with Σ^*. The result is an FSA accepting L_2.

The treatment of "at least 2 b's" above can be easily extended to handle similar quantifier expressions. For example, the English phrases on the left

in (5) can be translated into the operator expression shown on the right hand side:

(5)

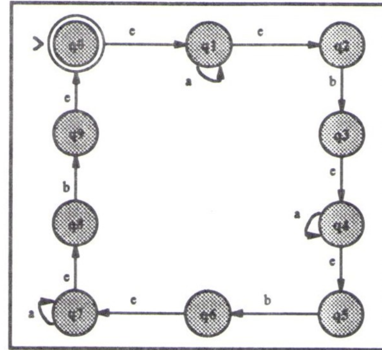
fewer/less than n x's $\Rightarrow$
    (Complement (At-Least-N n M_x))
more than n x's $\Rightarrow$
    (At-Least-N n+1 M_x)
at most/no more than n x's $\Rightarrow$
    (Complement (At-Least-N n+1 M_x))
no fewer/no less than n x's $\Rightarrow$
    (At-Least-N n M_x))

where M_x is an FSA accepting exactly the language {x}. Note that in principle *x* might range over strings of any length, e.g., "at least 3 *bab*'s". Not only does the contiguous-vs.-separated ambiguity arise here, but another one is added as well: overlapping vs. non-overlapping. For example, does the string *bababab* contain at least 3 occurrences of *bab* or not? Yes, if overlapping occurrences are counted; no, if not.

Other expressions which are easily handled are "even number of *x*'s", "odd number of *x*'s", "number of *x*'s which is divisible by/is a multiple of *n*", "is of length less than/more than *n*", and so on (again modulo the ambiguities just mentioned). The language L_3, for example,

$L_3 = \{w \in \{a, b\}^* \mid$ the number of b's in w is a multiple of $3\}$

is accepted by an FSA constructed as follows. Three copies of the deterministic FSA {b} are alternated with three copies of $(\Sigma\text{-}b)^*$, an FSA accepting all strings containing no occurrences of *b*. These are formed into a loop around an initial state, which is designated as the only accepting state.



FSA#1 (absorbing states omitted for clarity)

This could of course be made deterministic and minimized to yield an equivalent 3-state FSA.

English specifications joined by "and," "or," "if...then," etc. are handled straightforwardly by algorithms for constructing from any FSA's M_1 and M_2 an FSA accepting the intersection, union, etc. of $L(M_1)$ and $L(M_2)$. For example, the specification of L_4,

$L_4 = \{w \in \{a, b\}^* \mid w$ does not contain *bab* and ends with at least 2 b's$\}$

which is the intersection of L_1 and L_2 above, is translated into the operator expression:

(Intersection
    (Complement (Contains "bab"))
    (Ends-With (At-Least-N 2 "b")))

With the `Intersection` operator we could handle a quantified expression such as "exactly 2 *b*'s" by translating it as:

(Intersection
    (At-Least-N 2 "b")
    (Complement (At-Least-N 3 "b")))

i.e., “at least 2 and no more than 2.” However, we have redundantly added the operator `Exactly-N`, making such a construction unnecessary.

When we come to languages such as L_5:

(6) $L_5 = \{w \in \{a, b, c\}^* \mid$ every a in w is followed by 2 b’s$\}$

the problems of ambiguity in the English specification become acute. (6) is ambiguous (or perhaps vague) on at least four counts:

1) Does “followed” mean “immediately followed (without intervening letters)” or “followed, but not necessarily immediately”?
2) Does “2” mean “at least 2” or “exactly 2”?
3) Does “2 b’s” mean “two contiguous b’s” or “two not necessarily contiguous b’s”
4) Does “every a” have a distributive or a group interpretation? That is, does each a have the property of being followed by 2 b’s (however that may be interpreted), or does it suffice if the substring which contains all the a’s has the property?

Certain readings are more plausible that others in specific contexts. For example “begins with 2 b’s” would almost certainly have to be interpreted as “contiguous b’s” but in addition might reasonably be read either as “at least 2 b’s” or “exactly 2 b’s”. (Students of automata theory often give varying responses in such cases, as anyone knows who has ever set such a question on an examination.) Further, not all the possibilities can be realized independently. Our current system assumes a (modifiable) default reading for each source of ambiguity, but also is prepared, in the presence of disambiguating information, to produce any of the possible readings.

Suppose, for example, that we construe L_5 as “each a is followed (not necessarily immediately) by at least 2 not necessarily contiguous b’s” We might represent a typical string in this language schematically as follows:

...a...b...b...(b...b)...a...b...b...(b...b)...
a...b...b...(b...b)...

where it is understood that there are no occurrences of a’s before the first a, no others in the regions between a’s, and none after the last b given. The regions between an a and the next b might be empty (if the b follows immediately), and the regions between unparenthesized b’s might also be empty (if the b’s were contiguous). Otherwise the dotted regions are assumed to contain occurrences of other letters -- in this case c’s.

An FSA for this language would be constructed as shown schematically in Figure 1:

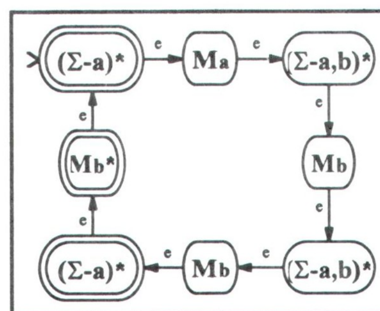


Figure 1

M_a and M_b are two-state FSA's accepting {a} and {b}, respectively. As above, (Σ - a)* is the FSA accepting all strings over the alphabet Σ which do not contain *a*. (Σ - a,b)* is the FSA accepting all strings over Σ which contain neither *a* nor *b*. Both of the latter two FSA's are easily constructed, and in the present case will accept $(a \cup b)^*$ and c^*, respectively. The unlabeled transitions are assumed to be *e*-transitions.

On the other hand, if we interpret L_5 as "each *a* is followed immediately by at least two contiguous *b*'s," schematically,

...abb...abb...abb...(no more *a*'s)

the FSA schema (Figure 2) is much simpler:

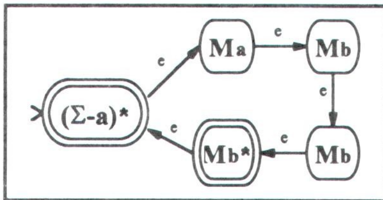


Figure 2

We leave it to the reader to construct schema for the other cases.

The Xerox PARC Approach

Lauri Karttunen, Ron Kaplan, and their co-workers at Xerox PARC (personal communication) have been working on a system which is similar to ours in some respects, although it is motivated by different concerns. Their system translates from an expanded notation for regular expressions into FSA's, and it of course makes use of many of the same basic algorithms for making FSA's deterministic, minimizing, concatenating, and so on.

For cases such as "every *x* is followed by *n* *y*'s", they reason as follows. The language contains all strings (over the alphabet) which do not fail the specification. How could a string fail? By having at least one occurrence of *x* which is not followed by *n* *y*'s. Therefore, find the language

{*w* | *w* contains at least one *x* followed by something other than *n* *y*'s}

and take its complement. This is the desired language.

For example, for L_5, "every *a* is followed by 2 *b*'s," they would look for the language representing the negation of this expression:

L_6 = {*w* ∈ {a, b, c}* | *w* contains at least one *a* followed by something other than 2 *b*'s}

They then complement L_6 to produce an automaton recognizing the desired language. Note that the ambiguity problems arise here as well -- "followed" may be construed as "immediately" or "not necessarily immediately" and so on. We have checked FSA's generated by our procedure against those constructed by the Xerox PARC procedure in a number of cases and found them to be equivalent on the "distributive" reading (i.e., each *a* has the property of being followed by at least two *b*'s before the next *a* occurs). It is not yet clear whether there is a real difference between the two methods in the case of the "group" reading (i.e., all of the *a*'s as a group have the property of being followed by at least two *b*'s).

Further Issues

There are English specifications which have not yet been incorporated into our system but which should present no problem in principle. Some examples are:

every substring of length 3 contains at least 2 *a*'s

an even number of *a*'s between every two *b*'s

The schema shown in Figure 1 is easily generalized from single letters to strings, e.g., "the number of occurrences of *bab* is a multiple of three." An FSA accepting {bab} replaces the one accepting {b}, and $\Sigma^*\text{-}\Sigma^* bab \Sigma^*$ is interleaved instead of $(\Sigma^*\text{-}b)^*$. This gives the correct result if we are counting only non-overlapping occurrences, but is clearly inadequate if overlapping should be taken into account.

There are some types of English expressions which we have not attempted to incorporate because they presuppose knowledge from domains other than the mathematical theory of symbols and strings. For example,

the set of chess moves, such as p-k4 or kbp x qn

all strings of letters that contain the five vowels in order

And we have not attempted to deal with specifications of languages which are not regular, such as:

an equal number of *a*'s and *b*'s

exactly *n a*'s followed by exactly *n b*'s

It has been pointed out to us that certain Boolean combinations of such specifications will be regular. For example,

$\{w \in \{a, b\}^* \mid w$ contains an equal number of a's and b's or w does not contain an equal number of a's and b's$\}$ $(= (a \cup b)^*)$

Indeed, one could make such expressions out of any specifications whatever, even for specifications of non-r.e. sets. It is not clear how to deal with such cases short of adding general routines for detecting tautologies and contradictions.

We now proceed to a discussion of the implementation of the system.

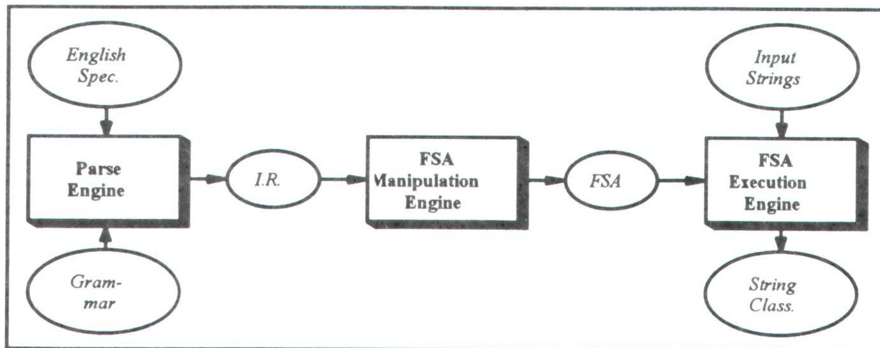


Figure 3 Overall structure of system

```
1. <compound-spec> → <spec>
|              Sem.: result of <1>
|  2. <spec> → which <compound-VP-restr>
|  |              Sem.: result of <1>
|  |  3. <compound-VP-restr> → <VP-restr> and <VP-restr>
|  |  |              Sem.: intersect <1> and <2>
|  |  |  4. <VP-restr> → "end"-<verb-complex> <comp-re-list>
|  |  |  |              Sem.:    if (<1> = 'negative')
|  |  |  |                           complement of (concatenate Σ* and <2>)
|  |  |  |                       else
|  |  |  |                           concatenate Σ* and <2>
|  |  |  |  5. "end"-<verb-complex> → "end"-<passive-verb-complex>
|  |  |  |  |              Sem.: result of <1>
|  |  |  |  |  6. "end"-<passive-verb-complex> → are "end"-<passive-verb>
|  |  |  |  |  |              Sem.: 'positive'
|  |  |  |  |  |  7. "end"-<passive-verb> → terminated by
|  |  |  |  |  |                 Sem.:
|  |  |  |  8. <comp-re-list> → <compound-re>
|  |  |  |  |              Sem: intersect all elements of list
|  |  |  |  |  9. <compound-re> → a <multi-contig-seq> <re>
|  |  |  |  |  |              Sem.: <1> or more sequential repetitions of <2>
|  |  |  |  |  |  10. <multi-contig-seq> → double
|  |  |  |  |  |                 Sem.: 2
|  |  |  |  |  |  11. <re> → " <simple-re> "
|  |  |  |  |  |  |              Sem.: result of <2>
|  |  |  |  |  |  |  12. <simple-re> → <string>
|  |  |  |  |  |  |                 Sem.: construct FSA to accept <string>
|  |  |  13. <VP-restr> → "contain"-<verb-complex> <comp-re-list>
|  |  |  |              Sem.:    if (<1> = 'negative')
|  |  |  |                           complement of (concatenate Σ* , <2>, and Σ*)
|  |  |  |                       else
|  |  |  |                           concatenate Σ* , <2>, and Σ*
|  |  |  |  14. "contain"-<verb-complex> → "contain"-<active-verb-complex>
|  |  |  |  |              Sem.: result of <1>
|  |  |  |  |  15. "contain"-<active-verb-complex> → do not "contain"-<active-V>
|  |  |  |  |  |              Sem.: 'negative'
|  |  |  |  |  |  16. "contain"-<active-V> → contain
|  |  |  |  |  |                 Sem.:
|  |  |  |  17. <comp-re-list> → <compound-re>
|  |  |  |  |              Sem: intersect all elements of list
|  |  |  |  |  18. <compound-re> → <quan-one> <re-unit> <re>
|  |  |  |  |  |              Sem.: result of <3>
|  |  |  |  |  |  19. <quan-one> → the
|  |  |  |  |  |                 Sem.:
|  |  |  |  |  |  20. <re-unit> → string
|  |  |  |  |  |                 Sem.:
|  |  |  |  |  |  21. <re> → " <simple-re> "
|  |  |  |  |  |  |              Sem.: result of <2>
|  |  |  |  |  |  |  22. <simple-re> → <string>
|  |  |  |  |  |  |                 Sem.: construct FSA to accept <string>
```

Figure 4 Parse Trace with Compositional Semantics

A Simple Example

Here we will trace an example through the system. In order to keep the presentation feasible, we will assume the English language input from the user was the following simple sentence:

Construct a finite state automaton which generates the set of all strings, over the alphabet... ***Σ = {a,b}, which are terminated by a double "b" and do not contain the string "bab".***

(The portion of the input sentence which is actually specified by the user is given in boldface.)

Parsing

Figure 4 illustrates the successful parse of the English input sentence. The category *<compound-spec>* (compound specification) is the distinguished symbol of the grammar and the start point (line #1) for the parse. In the case of our example sentence, the simplest expansion of *<compound-spec>* was selected, a single *<spec>* (specification). The attached semantic interpretation of this rule (given in the following line) is also very simple: i.e., pass on the I.R. expression which is constructed in the evaluation of the first variable (*<spec>*) of the right hand side (RHS) of the rule. Line #2, the rule expanding *<spec>*, consumes the literal `which` and expands to *<compound-VP-restr>* (compound verb phrase restriction).

In line #3, we see that *<compound-VP-restr>*, in this example, expands to a conjunction of *<VP-restr>*'s, and the compositional semantic template dictates that the result of the first should be intersected with the result of the second. The first *<VP-restr>* (line #4), matches the rule whose RHS consists of an *"end"-<verb-complex>* followed by a *<comp-re-list>* (compound regular expression list). (The second *<VP-restr>* is taken up in line #13.) Note that the category *"end"-<verb-complex>* captures the class of English expressions which denote the property of "ending in (some string) *w*". This is a good example of a category which is much more coherent semantically than syntactically (although it does have some purely syntactic properties). Note also that the semantics of this class of English expressions is captured at this (high) level, not "down in the trenches" during the dissection of the particular expression of the "ends with" concept.

Finally, let us draw attention to the last line of Figure 4 (line #22), where the attached semantics indicate that construction of an FSA to recognize a string (here "*bab*") is called for. That operation is actually built into the I.R. language -- any string is assumed to be an extended regular expression. The remainder of Figure 4 will be left to the reader's inspection. A more graphical (but less informative) parse tree representation is also provided (Figure 5).

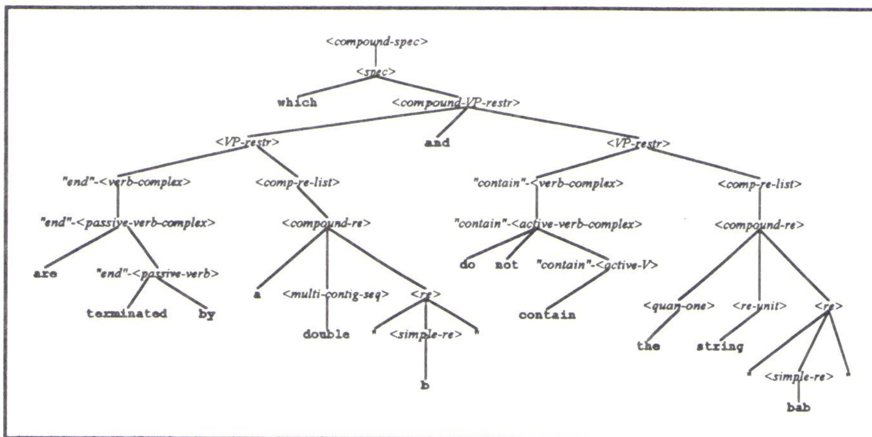


Figure 5 Parse tree for the example

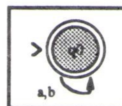
The final result of the parse is the Intermediate Representation expression which is the composition of the grammar rules' attached semantics:

```
(Produce-FSA '((#\a #\b)
  (Intersection
    (Ends-With
      (At-Least-N 2 "b"))
    (Complement
      (Contains "bab")))))
```

In many ways the I.R. is an "undigested" composition -- more a prescription for arriving at the meaning of the sentence via composition than a synthesis itself. This is precisely the strength of parsing in domains such as this, which have a strong underlying formal model. The semantic representation at the parse level can be much less intricate since the parser can rely on the formal operations of the model for interpretation.

I.R. Interpretation

The very first step in the process of interpretation is always formation of a finite state automaton to represent Σ^* :

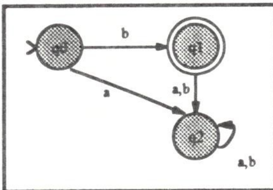


FSA#2

Σ^* is employed in a host of operations which are part of the I.R. interpretation. (We will see some examples of this below.)

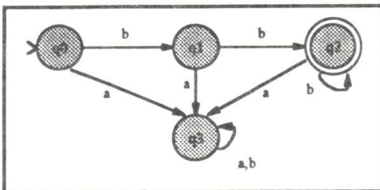
After forming Σ^* (and binding it to a global variable), we simply submit the remaining portion of the I.R. expression to the LISP interpreter for evaluation (since as discussed above, it forms a LISP S-expression and the I.R. language operators are encoded as LISP functions). The LISP evaluation proceeds left-to-right, recursing at parentheses embeddings. We trace execution at the FSA formation points:

We start off with the first branch of the intersection. At the bottom of the recursion is the interpretation of the regular expression *"b"*, yielding:



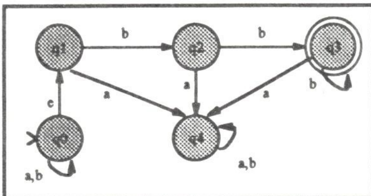
FSA#3

One step back up the recursion tree, we interpret *(At-Least-N 2 FSA#3)*, yielding:



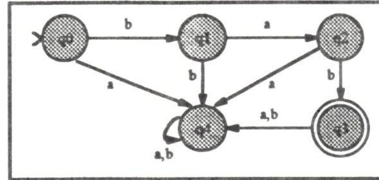
FSA#4

Interpretation of *(Ends-With FSA#4)* as the concatenation of FSA#2 (Σ*) with FSA#4, yields:



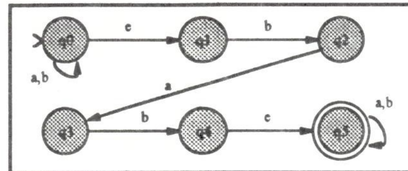
FSA#5

Starting on the second branch of the intersection, the regular expression, *"bab"* is interpreted, yielding:



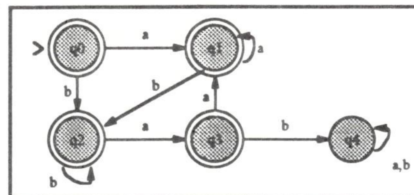
FSA#6

Interpretation of *(Contains FSA#6)* as the concatenation of FSA#2 (Σ*) with FSA#6 and FSA#2 again, yields (with the absorbing state left off for simplicity):



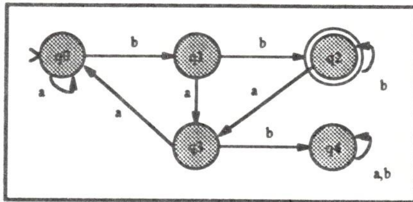
FSA#7

And, popping up the recursion tree one level, interpretation of *(Complement FSA#7)*, yields:



FSA#8

And finally, the two branches are intersected; *(Intersection FSA#5 FSA#8)* yields the final deterministic, minimal FSA:



FSA#9

Verification

It is fairly easy, given the simplicity of the resulting final automaton[4], to verify that FSA#9 does, in fact, implement the original English specification. In more complex cases, the ability to test strings against the device as a means of verification, as illustrated below, is much more crucial.

(recognize "aba" *FSA#9*) ⇒ NIL
(recognize "abb" *FSA#9*) ⇒ (2)
(recognize "babb" *FSA#9*) ⇒ NIL
(recognize "baabb" *FSA#9*) ⇒ (2)

Summary

This system for automatically converting English specifications of finite state automata (FSA's) into functional (simulated) devices originated as, more than anything else, a mathematical curiosity. It certainly has practical utility as an educational device in the context of an introductory course in automata and formal language theory (if nowhere else). There has been considerable work (principally by the AI community) in the area of limited domain (sub-language) NLU (e.g., see Grishman & Kittredge, 1986). The most common target of such work has historically been database access (e.g., Waltz, 1978), but a variety of other areas have also been represented.

To our knowledge this is the first attempt at NLU in the domain of FSA specifications. It represents a limited capability to produce a working computational device from English language specifications. The general capability to program computers via natural language has been a desideratum in the area of Human-Computer Interface for many years, and although there has been some effort in this area (e.g., Flatau, 1987), it seems that, relative to its importance, little effort has been applied to this goal. While we are aware that difficulties exist in applying the techniques embodied in our system to broader classes of computational devices and languages, e.g. push-down automata and context-free languages, the possibility of applying these techniques directly to interesting sub-classes of higher-level devices and languages still exists. And even without such generalization, finite state devices represent an important class in and of themselves.

[1]We have not yet attacked the converse of this problem -- deriving an English description from an FSA. It seems both easier and harder. One could always render a regular expression, say, into some sort of "mathematical English," but it is not obvious how to get a compact and intuitively revealing interpretation.

[2]The user first supplies the alphabet (Σ), then completes the sentence, "Construct a finite state automaton which generates the set of all strings, over the alphabet Σ = {<user_supplied>}..." The decision was made to keep the parsing task on the interesting part of the problem.

[3]Although the FSA *Manipulation* Engine always outputs a deterministic, minimal FSA, the FSA *Execution* Engine can also work with non-deterministic FSA's. With a non-deterministic FSA, the result of testing an input string which is in the language is the set of states that the machine was in at the end of the string.

[4]It should be noted that part of the simplicity of the final FSA of the example is due to its graphical presentation (which is not yet automatically produced in our system). It is considerably more difficult to get an intuitive grasp on the language of the FSA when given only the textual listing of states and transitions.

Bibliography

Bates, M. (1978) "The Theory and Practice of Augmented Transition Network Grammars" in L. Bolc (ed.) *Natural Language Communication with Computers*, Springer: New York, pp.191-260

Brobrow, D.G. (1973) "Natural Language Input for a Computer Problem Solving System" in M. Minsky (ed.) *Semantic Information Processing*, MIT Press: Cambridge, MA, pp.133-215

Flatau, A. (1987) "A Program for Translating English Sentences into Lisp Programs", Artificial Intelligence Lab -- Univ. of Texas at Austin, Tech. Rpt. #AI TR87-48 (February 1987)

Green, B., A.K. Wolf, C. Chomsky, & K. Laughery (1963) "BASEBALL: An Automatic Question Answerer" in Feigenbaum & Feldman (eds.) *Computers and Thought*, McGraw-Hill: New York, pp.207-216

Grishman, R. & R. Kittredge (1986) *Analyzing Language in Restricted Domains: Sublanguage Description and Processing*, Erlbaum

Hopcroft, J.E. & J.D. Ullman (1979) *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley: Reading, MA

Lewis, H.R. & C.H. Papadimitriou (1981) *Elements of the Theory of Computation*, Prentice-Hall: Englewood Cliffs, NJ

Lindsay, R. (1973) "In Defense of Ad Hoc Systems" in Schank & Colby (eds.) *Computer Models of Thought and Language*, Freeman: San Francisco, pp.372-395

Partee, B.H., A. ter Muelen, & R.E. Wall (1990) *Mathematical Methods in Linguistics*, Kluwer: Dordrecht

Waltz, D.L. (1978) "An English Language Question-Answering System for a Large Relational Data Base", *Comm. of the ACM* 21, pp.526-539

Wilks, Y. (1973) "An Artificial Intelligence Approach to Machine Translation" in Schank & Colby (eds.) *Computer Models of Thought and Language*, Freeman: San Francisco, pp.114-151

Winograd, T. (1972) *Understanding Natural Language*, Academic Press: New York