Hastings, John Douglas, Design and Implementation of a Speech Recognition Database Query System, M.S., Department of Computer Science, August 1991.

This thesis introduces CONQUEST, a constrained natural language speech recognition database query system. The objective was to improve on previous natural language database query systems by designing and implementing a more user-friendly query system through the integration of speech and nondeterministic syntactic processing. This paper will discuss the areas in which improvements were attempted, the components required along with a discussion of each, an illustration of system operation, and an evaluation of the final product.

DESIGN AND IMPLEMENTATION OF A SPEECH RECOGNITION DATABASE QUERY SYSTEM

by
John Douglas Hastings

A thesis
submitted to the
Department of Computer Science and
The Graduate School of The University
of Wyoming in Partial Fulfillment of Requirements
for the Degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Laramie, Wyoming
August, 1991

ACKNOWLEDGEMENTS

I wish to express my deepest appreciation for the additional time and effort that Dr. S. R. Petrick offered in helping me to complete this thesis.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 NATURAL LANGUAGE QUERY SYSTEMS

Computational linguistics has probably been most successfully applied to providing natural language front ends to database systems, relational database systems in particular. There are several reasons that might be suggested for this. First and foremost is the relatively small size of the natural language subsets that are required for query in a very narrowly limited database domain. These small natural language subsets allow queries which are limited syntactically without removing the ways by which a wide variety of requests for structured information can be stated. It is also often the case that system users can be advised in such a way that they are allowed to observe and avoid the syntactic and semantic limitations of a system, while still being able to communicate effectively. This situation with respect to the required coverage of English for database retrieval contrasts markedly with that for translation of one natural language to another. With rare

exceptions (such as the French METEO System for translating weather reports) people do not write reports, articles, books, manuals, etc. with the specific intent of their being able to be translated to another language. Consequently, even such mundane literature as equipment instruction manuals have been found to contain the entire range of syntactic and semantic complexity.

A large number of natural language database system front ends have been implemented in the past decade with individual query success rates of 70% to 80% being quite common [1, 2, 3, 4]. This might seem like a rather modest accomplishment, but the corresponding success rates for users of even the latest and best formal query languages are apparently much lower [5]. In addition, there are many users and potential users of natural language query systems that are either unwilling or unable to use formal languages. The movement of query system research and development has progressed in recent years from the experimental research laboratory to the corporate development facility (in the past five years), with most of the current research being conducted in the area of customizing the lexical, syntactic, and semantic demands of a system to accommodate a specific database application.

Useful as these recent database systems are, they nevertheless have some shortcomings. Herschmann et al [6]

reported that their users were not disturbed by the length of time required to answer successful queries, but they were bothered when a similar length of time was required to signal unprocessable queries. These and most other natural language query system users have also complained about the lack of helpful advice when a query cannot be answered; all too many systems just give a message such as "Unable to process your query. Please rephrase and try again" without identifying the likely reason for its failure. This is particularly frustrating if the user has input a long sentence and waited a long time for a response, only to be told that the sentence wasn't understood with no hint of why not or how to rephrase. If 30% of a user's queries cannot be processed correctly he or she expects some assistance in reformulating those queries to get the desired information from the system.

One approach to eliminating queries that lie outside of a system's capabilities is to allow the user to input only "legal" queries that the system can process. Ross and Tennant first proposed and implemented such a system at Texas Instruments [7, 8]. They used a context-free grammar to form the basis for their internal translation from English to SQL (a database query language), and constrained the user to entering only words, chosen from among the set of possible words, that constituted a syntactically valid

continuation of the words previously input to the system. At each stage of the input and processing, a list of all the words that produced a syntactically well-formed continuation was displayed on a computer monitor menu, and the user was required to choose among them via scrolling, mouse pointing and clicking. Their claim was that their system gives its users sufficient flexibility to ask for any desired information, while at the same time guiding and constraining them to input only syntactically well-formed input queries.

This approach has been adopted by at least two other groups since its proposal [personal communication]. It has a serious drawback, however. This has to do with the time requirements and inconvenience of successive word selection via menus, even if a mouse is used. Particularly when there are a lot of possible choices, it is tiresome and time-consuming to scroll through a long list of words, hunting (perhaps vainly) for the word one wants to use next in formulating an intended query. Even if the user has generated the same query at some earlier point and knows the exact words which are possible, he must still search for them (or at the least move to them) on the menu screen.

This onerous nature of using large menus is compounded by a property common to all natural language-based database systems. They are not provably complete in the sense of guaranteeing that every query which is expressible in first

order logic has a natural language query equivalent that is accepted by the system. For systems making use of the Texas Instruments approach this means that there may be no sequence of menu selections that constitutes a query that expresses a user's intention to retrieve certain data. This causes user frustration in a laborious but futile search for a menu system path that asks for the desired information. The next section contains an outline of our proposal for alleviating these drawbacks while retaining the advantages of the TI grammar-constrained approach.

1.2 A CONSTRAINED NATURAL LANGUAGE QUERY SYSTEM PROPOSAL

The proposed system is called CONQUEST for **Co**nstrained **N**atural Language **Que**ry **S**yst**e**m and attempts in two main ways to alleviate the shortcomings of the TI approach. First, input spoken word recognition is incorporated to allow users to enter their queries by speaking, which is perhaps faster and easier than searching for the words on a menu screen and possibly more user friendly than communicating through the use of a keyboard or mouse. Second, a nondeterministic parsing approach is used to parse each word as it is entered according to a grammar and in the context of the previously entered words. This allows CONQUEST to detect the exact point of non well-formedness in a natural language database

query and to flag the error (optionally with a speech output system if the user so desires) so that the user can see precisely where in the query an error occurs and what the nature of the problem is. In addition, instead of cluttering the screen up with unnecessary and sometimes confusing information (such as menus), CONQUEST displays nothing on the screen unless an error is made or the user executes a special command. After a grammatical query is successfully input to the system it is converted to the SQL database query language, suitable for retrieval of data in any database system which provides interpretation of that language. Figure 1 shows the overall architecture of CONQUEST. The choice of specific components is discussed in the next chapter.

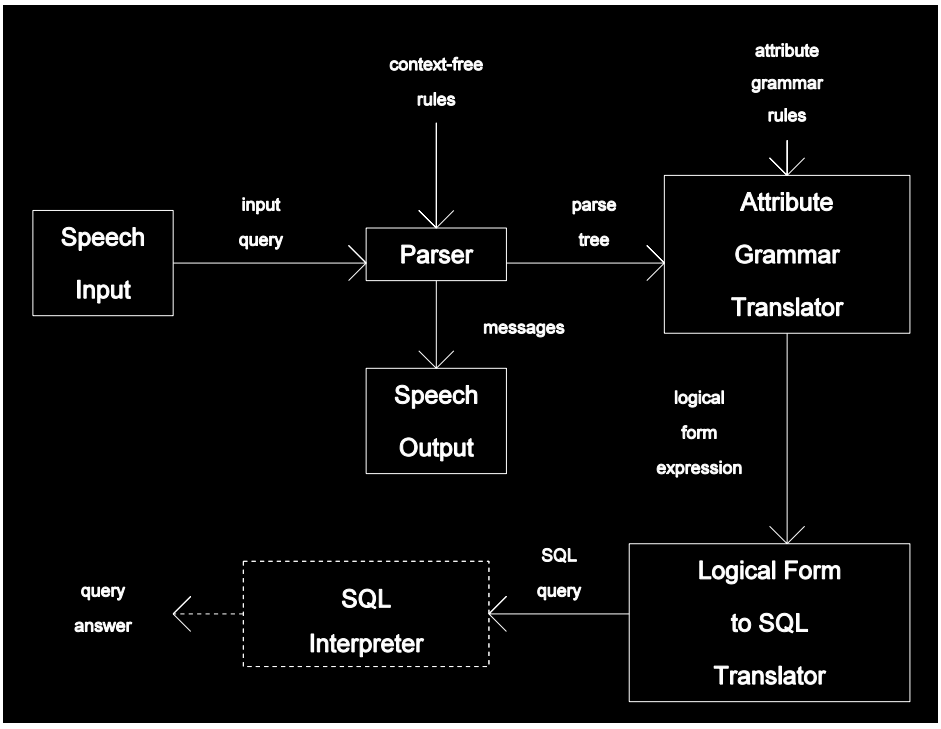

Figure 1. CONQUEST Architecture

CHAPTER II

COMPONENTS AND STRUCTURES

2.1 SPEECH SYSTEMS

After determining the general approach that would be taken, a choice of specific components was made, beginning with input spoken word recognition and text-to-speech generation facilities. These choices were made on the basis of availability rather than functionality. Specifically, a COVOX Voice Master Key Version 2.00 System [9] was already installed on a laboratory 33 MHz IBM compatible PC, and it had been successfully interfaced to work with the Goldworks I Developer LISP System [10] that was also installed on that machine. The COVOX System is designed to perform isolated word recognition and to initiate the equivalent of a sequence of keystrokes (i.e., a macro) upon recognition of a word for which it has previously been trained. Another COVOX-supplied hardware-software system, the Speech Thing and the SmoothTalker software (designed and written by First

Byte, Inc.) [11], is used for text-to-speech generation, and was also chosen for reasons of availability.

These COVOX speech I/O software/hardware packages were available because their performance to cost ratio is exceptionally high. The combined cost of software and hardware for both speech recognition and generation was just $225. In spite of this, the Voice Master Key System provides recognition of isolated spoken words and phrases from a limited vocabulary with a success rate that can under favorable circumstances exceed 95%. The system's performance is less than that of more expensive commercial alternatives and experimental research systems in two principal ways: (1) it is less robust in providing a high individual word recognition success rate for a range of speakers without separate "training" for each speaker, and (2) it is quite limited as to the size of the vocabulary that can be recognized. Several commercially available systems recognize 1000 word vocabularies [12] and systems with an active vocabulary of up to 30,000 words exist [13]. The COVOX Voice Master Key System, on the other hand, is limited to recognizing 256 words, and they must be organized into groups of no more than 16 words. Only one group can be active for recognition purposes at a given time, and switching between groups can be done by voice command or by typed command. For the purpose of a small demonstration

system this restriction can be tolerated. However, if one were thoroughly evaluating the practical utility of the type of spoken input query system created in this thesis, a system capable of recognizing a several hundred word vocabulary would be required.

The COVOX Speech Thing together with the SmoothTalker text-to-speech software system provides sufficiently intelligible English output to be used for feedback to users in cases of detected ambiguity or ungrammaticality.

2.2 GRAMMAR FORMALISM

A syntactic linguistic formalism (theory) upon which to base the system was required for two reasons. First, to provide each query with a linguistic structure that can be used as an adequate basis for semantic interpretation in the form of translation to a first order logical form. The latter systematically represents the meaning of a query. Second, to provide a filter for detecting the point within a query at which it becomes syntactically ill-formed and to provide a means of suggesting well-formed continuations from earlier points in the query.

These two requirements eliminate certain grammatical formalisms from consideration. The transformational grammar was eliminated because of the complexity of its parsers [14,

15] and because it is not suitable for determining the point at which an input string (query) becomes ungrammatical. The augmented transition network (ATN) [16] is one formalism that does appear to be suitable. Its transitions to the set of next states or failures to do so indicate the syntactic admissibility of the next word.

Another candidate is the context free grammar, preferably augmented by the addition of complex syntactic/semantic features as in GPSG (General Phrase Structure Grammar) [17] or DCL (Definite Clause Grammar [18]. Such extensions of a context free grammar provide for the parsimonious expression of various types of syntactic agreement and a means of semantic interpretation. We have opted instead for a simple context free grammar (CFG) syntactic component for several reasons. First, it is simpler than its augmented CFG counterparts. Second, although it is not suitable for specifying a sizeable natural language subset, it nevertheless can be used to specify a small subset which is large enough to demonstrate the utility of our proposed query interface. And finally, its choice permits the use of various available software implementations of system components. In particular, a CFG parser, a semantic interpreter (translator), and a logical form to SQL (Structured Query Language) translator were all available after necessary modification for our purposes.

2.3 PARSING ALGORITHM

Having selected a simple CFG linguistic model, alternative CFG parsing algorithms were next considered in order to obtain the associated structural description(s) of a query with respect to a particular CFG. The first candidate considered was a left corner parser because: (1) an implementation supplied by Petrick [19] was available, (2) Ross [8] has demonstrated that this parsing algorithm can be modified to determine ill-formedness at the intermediate points of a query, and (3) various authors [20, 21] have given experimental support for the practical efficiency of left corner parsers relative to that of chart parsers even though the latter have better (n^3 as opposed to exponential) worst case complexity bounds [22, 23, 24]. However, initial attempts to modify Petrick's left corner parser to let it accept terminal symbols (words) in a strictly sequential, word-at-a-time fashion proved harder than expected. This was due to the organization of the parser with respect to its handling of non-determinism. Its depth-first, pushdown stack implementation of following left corner non-deterministic choice points was not conducive to determining points within a query at which no continuation is possible. This problem with handling nondeterminism

could also cause difficulties with other parsers that appear at first inspection to be suitable, ATNs in particular.

Nondeterminism in chart parsers, on the other hand, is easy to implement in either a depth first manner or in a manner that processes all continuations consistent with a particular next word prior to moving on to the next word. The latter type of implementation is actually more common. After considering several chart parsers, Tomita's [22, 25] was selected to be modified for the purposes of this thesis. Even though its worst case bound is no better than that of other chart parsers, its implementations offer possibilities for efficiency that have been empirically validated. In particular, its treatment of data structures for the sharing of substructures common to ambiguous parses is particularly efficient. The principal reason for selecting this parser, however, is that it seemed to require the least amount of modification to achieve the type of interaction that was required. Tomita's original parser, whose coding is given in his doctoral thesis, begins parsing only after an entire sentence has been specified. Nevertheless, it is organized to do all of the parsing associated with the next word in the input sentence nondeterministically with no lookahead prior to going on to the next word, so his code required minimal conversion for our purposes. The primary effort taken was in translating from the MACLISP dialect in which

the procedures were given to the available version of Common LISP.

Tomita's parser works by building on to a forest of interconnected (pointer-sharing) trees which are alternately collapsed and built upon. This allows two very useful things. First, the user, whenever desired, can back up in the forest of parse trees. This might be useful, for example, when the user wishes to retract his previous word. More generally, it is useful when difficulty is encountered in producing a query that is both syntactically well-formed and semantically responsive to the user's desire for specific information to be retrieved. Consider the situation where a word is uttered that has no subsequent continuation to a well-formed query (and is flagged as erroneous), and none of the alternative possible words (which are suggested by the system, and which do not fail) seem to be appropriate for producing a query that reflects the user's intent. The user must then back up to a prior point in his query (possibly its beginning) and attempt to go on from there. To do this it is possible to back the Tomita parser up to any requested prior point, and to determine the set of terminal symbols that constitute well-formed continuations onward from that point. Many other parsers would be forced to reparse the entire query to

earlier points, repeating the same computation over and over.

Details of the action of the modified Tomita parser are presented in the next chapter. They explain more clearly how the parser works in order to satisfy the interactive requirements of this thesis.

2.4 TRANSLATION TO LOGICAL FORM

Once a syntactically correct (with respect to the system's grammar) query has been produced and assigned phrase structure, it is necessary to translate that structure into some computer-interpretable form. Rather than translating directly into a formal query language such as SQL many investigators have found it worthwhile to produce an intermediate representation of meaning using some variant of relational calculus [2, 3, 26]. One reason for this is that it is easier to produce something that is in a standard logical form before doing the final translation. In addition, going to this intermediate form requires less modification of code in the event that the database query language is changed. A set domain relational calculus representation devised by Petrick for the IBM Research TQA System [1] was chosen as our logical form. Adoption of this particular representation made it possible to use a subset

of Petrick's LISP functions for subsequent translation of logical forms to SQL. An explanation and illustration of the nature of this logical form appears in a subsequent section of this chapter. The remainder of this section is concerned with the translation from surface syntactic structures to logical form.

The task of translating English structures to computer interpretable form is similar to that of converting programming language syntactic structures to an intermediate or a low level programming language, and some of the same formal mechanisms have been used for both. The usual translation mechanism used is a bottom-to-top, single pass translation procedure devised originally by Irons [27] for the translation of ALGOL to assembly language. In its simplest form a single translation rule is associated with each production of a CFG. Each translation rule defines the translation of that portion of a parse tree structure corresponding to its CFG production. The translation of a nonterminal node expanded by some production is defined to be some function of the resulting translations of the production's daughter nodes. Terminal symbols are defined to be their own translations, and hence the translation of every node in a syntactic tree can be recursively determined by applying translation rules corresponding to the productions embedded in the tree in a bottom-to-top fashion.

The translation of the tree's root node is taken as the translation of the tree.

This procedure is used in the semantic interpretation of the Montague grammar [28] and GPSG [17] as well as a number of other natural language systems, and it is the basis of most computer compilers. A natural extension is to replace the single translation of each node described above with multiple translations, each identified as the value of a set of semantic attributes associated with a node.

A property of a compositional semantic system such as the one described above is that a given subtree will always be translated the same way regardless of the larger structure in which it is embedded. Often however, this is not desirable. One solution is to tolerate this restriction, producing translations interlaced with variables that are substituted for at higher levels when the necessary information is available. GPSG takes this approach. Another approach is to allow attribute value information to be passed both up and down a tree, thus allowing for the translation of subtrees as a function not only of their own content but also as a function of the larger context in which they occur. This is the approach devised by Knuth for programming language compilation purposes and subsequently adopted by researchers in natural language interpretation as well [29].

Translation of the syntactic structures assigned to queries by our simple subset of English did not require the use of Knuth's inherited (top-down) attributes, only his bottom-to-top Irons-like synthesized attributes. Nevertheless, we made use of a full-fledged Knuth Attribute Grammar translation component for two reasons. A LISP-based attribute grammar translator was available [30] and its use gives our system greater possibilities for future extension to more realistic subsets of English than would have been the case had an Irons-type translator been used.

2.5 LOGICAL FORM TO SQL TRANSLATION

To complete the task of mapping English queries to a computer interpretable form, a target database query language had to be chosen. SQL was an obvious choice due to its universality, with many database systems providing an SQL compatibility feature in addition to their main database query language. Choosing SQL allows CONQUEST to work with a large number of database systems. Also, considerable work has supposedly been done on optimizing SQL queries. Probably the most convincing argument was the existence of LISP functions (that required only slight modification) written by Petrick to accomplish the translation from logical form to SQL in the TQA System. Petrick's functions cover the

full range of logical form possibilities, and they are further complicated by individually handling many separate cases so as to produce very efficient SQL translations. Our simple English query subset produced very few of the different types of logical form query cases treated by Petrick, and so only a portion of his functions had to be translated from the IBM LISP dialect in which they were originally written to equivalent Common LISP functions. Nevertheless, 73 functions in all had to be converted to our Common LISP dialect. Those functions are included with the others of CONQUEST in Appendix C.

2.6 LOGICAL FORM NOTATION USED

It will be recalled that a set domain relational calculus logical form is used as an intermediate step in the conversion from the parse tree structure to SQL. The LISP representation used for this simple type of logical form expression is illustrated by the following example:

```
(SETX 'X1
   '(RELATION 'ZP '(PNO PNAME) '(X1 'SCREW) '(= =)))
```

which denotes the set of all values X1 from the relation 'ZP (perhaps a database file containing an inventory of parts), such that there is a tuple whose PNO attribute has the value X1, and whose PNAME attribute has the value 'SCREW. More complex logical form expressions are possible, but the

simple CFG used by CONQUEST is capable of producing very few of them. Some further examples of logical forms CONQUEST is able to translate to SQL are:

```
(SETX 'X1 '(TOTAL X1
   (SETX 'X2
      '(RELATION 'ZP '(PNO COLOR) '(X2 'RED) '(= =)))))

(SETX 'X1
   '(AND
      (RELATION 'ZP '(PNO COLOR) '(X1 'RED) '(= =))
      (RELATION 'ZSP '(PNO SNO) '(X1 'S2) '(= =))))
```

Data structures used for the different grammars and tables appear during the discussion of components in Chapter 3.

CHAPTER THREE

TECHNICAL DISCUSSION OF SYSTEM COMPONENTS

3.1 SPEECH SYSTEM

As noted before, CONQUEST allows use of the COVOX Voice Master Key Version 2.00 speech recognition system to speak in the words of an English sentence, or the user can type them in. If the speech recognizer is employed, the system is first calibrated to adjust for noise surrounding the microphone. Next, the user specifies which words the system should be prepared to recognize by training a voice template. The user then specifies a macro template indicating the text (sequence of keystrokes) that should be entered when a spoken word is recognized. In our case the word to be recognized and the text to be entered are the same, so the system simply types the characters of the word followed by a carriage return (the ENTER key). For example if the user speaks "the", the text "the" should be typed in. Once the templates have been created and saved, they no longer have to be respecified.

Speech recognition can now begin. The user wanting a word to be recognized just hits the "hot key" (the SHIFT key in our configuration) and speaks a word. If all goes as planned, the word is recognized and the system enters the proper keystrokes.

The COVOX system does have some limitations. First, the templates in the system must be grouped in small sets of 16 words. This limits the number of words which can be recognized without switching via voiced or typed command to a different set of templates. Altogether, there can be 16 such different sets of 16 word templates. This is sufficient for our demonstration purposes when the words for each distinct sentence can be included in a separate template set, but it would not be suitable for practical applications.

A second problem is the recognition of numbers. It is impossible to set up the speech system to recognize all possible spoken numbers as there are an infinite number. One alternative would be to set up the voice and macro templates with individual digits. However, that would further restrict an already small speech vocabulary. Probably the best option is to type in the numbers separately. If the user is involved in a database application which relies heavily on numbers, the advantages of speech input are lost. It should be noted that

significant successes in the recognition of continuous speech constrained to numeric information have begun to take place [31].

Another problem is voice recognition itself. The COVOX system is usually very speaker specific. To achieve good recognition for more than one user, separate voice templates must be used. Even in the case of a single user, day to day variances in a person's voice and articulation sometimes cause recognition to fail for certain words. Existing but more expensive systems make stronger claims to speaker independence, but the performance of any system is enhanced if its parameters are optimized for a single speaker.

One last problem was the use of the memory resident COVOX speech recognition programs with the desired LISP interpreter. To handle an attribute grammar (stored in list form) that was both large enough and complex enough to show a variety of natural language queries, a Lisp interpreter of sufficient power such as GOLDWORKS II was required. However, GOLDWORKS II is not compatible with the COVOX programs, so GOLDWORKS I had to be used. As it works out, GOLDWORKS I can not manage very large lists, so this severely limited the attribute grammar size.

In addition to speech input, CONQUEST allows spoken output with the COVOX Speech Thing. This output device is designed to pronounce arbitrary ASCII text, and can speak

most phrases including numbers (except those including symbols) reasonably well. The Speech Thing will speak a phrase by calling the Common Lisp interface procedure **speak** and passing it the phrase as a string parameter.

3.2 CONTEXT-FREE GRAMMARS

Context-free grammar productions (or rules) are of the type:

```
G: S -> S B
   S -> B
   B -> 0
   B -> 1
```

where G is the set of productions, S and B are nonterminal symbols (or variables), S is the distinguished start symbol, and 0 and 1 are terminal symbols. These productions generate a language, made up of all the possible terminal symbol sentences derived by the productions from the distinguished nonterminal start symbol. In this case, the productions in G with respect to the start symbol S generate the language containing all binary numbers. For example, the sentence 011 belongs to the language generated by G by the following derivation: S -> SB -> SBB -> BBB ->

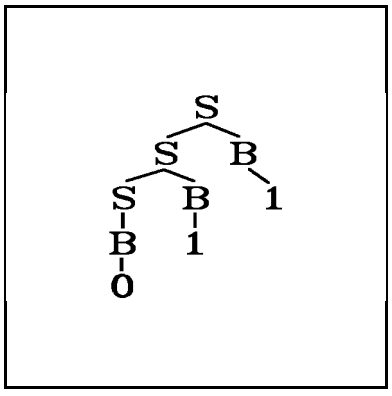

Figure 2. Parse Tree

BB1 -> B11 -> 011. The tree structure thus assigned to 011 is shown in Figure 2.

3.3 TOMITA'S PARSING ALGORITHM

A modified form of Tomita's parsing algorithm is used to interactively determine the legality of input terminals in the context of a sentence. This algorithm uses an LR parsing table containing an action table and a goto table that are constructed from a given context-free grammar to build a parse forest for a sentence with respect to that grammar. This parse forest shows the context-free productions employed to legally produce the input sentence.

Using the example context-free grammar G, limit the application of the Tomita parser to the productions:

(0) S -> S *B
(1) S -> *B
(2) START -> S

where START specifies the starting nonterminal (required in Tomita's algorithm), S is a nonterminal, and *B is a category nonterminal. Tomita requires the productions to be in an alphabetically ordered array in order to facilitate faster binary searches while building the LR parsing table, and to allow quick array index references to specific productions while parsing. In CONQUEST these rules are bound to the array **rules** using LISP as follows:

```
(setf rules '# (
     ((S -> (S *B))
     ((S -> (*B))
     ((START -> (S)) ))
```

What about the productions:

```
*B -> 0
*B -> 1
```

where *B is a category nonterminal, and 0 and 1 are category terminals? In the case that a subset of the English language is to be represented, it is best to avoid representing in the context-free grammar each production where English grammatical categories (such as *determiner) go to grammatical terminals (such as 'the). Otherwise, with these productions included, the LR parsing table generated from the context-free grammar would include each grammatical category terminal, making the LR parsing table too large and unnecessarily slowing down the system. So for "category" rules a separate terminal lookup table is maintained to show the category to which a terminal belongs. For the above productions the lookup table would contain (0 *B) and (1 *B). In LISP this lookup table is set up as follows:

```
(setf lookup '#(
     (0 *B)
     (1 *B) ))
```

The LR parsing table, containing the action and goto tables, generated by Tomita's algorithm for the above grammar is:

State	*B	$	S
0	sh2		1
1	sh3	acc	
2	re1	re1	
3	re0	re0	
	action table		goto table

where *B is a category preterminal, and S is a nonterminal. Two stacks are utilized by the Tomita parser, a state stack and a parse tree stack, which are maintained so as to always contain the same number of elements. Actually, there is no separate parse tree stack. The parse tree stack is just the set of pointers into the parse forest which coincide with state stack elements. Table entries such as "sh **n**" mean shift a new node (which contains the current preterminal such as *B along with the corresponding terminal such as 0) onto the parse forest, and push state **n** onto the state stack (along with a pointer to the new top forest element). Entries such as "re **n**" mean add rule **n** (referenced by index **n** in the array) to the parse forest (along with pointers to the parse forest node(s) it now dominates), reduce the state stack elements (and parse tree stack elements) which correspond to the now reduced node(s), and look at the top state stack element and the new top parse forest production to determine the new state. The entry "acc" stands for accept the input sentence. The symbol **$** signals the end of

an input sentence. Although a period would have been a more intuitive end of sentence symbol, the LISP programming language doesn't allow it. In LISP the previous action table would be bound to the array **table1** by the following expression:

```
(setf table1 '#(
     ((*B (S 2)))
     ((*B (S 3)) ($ (A)))
     ((@ALL (R 1)))
     ((@ALL (R 0))) ))
```

and the goto table would be bound to **table2** by:

```
(setf table2 '#(
     ((S 1))
     NIL
     NIL
     NIL ))
```

It should be noted that, even though Tomita's variant of LR parsing requires only one entry per table position as in the case of unambiguous grammars, it allows several entries making it suitable for use with ambiguous grammars. This is especially useful, since natural language grammars are usually ambiguous, requiring some form of nondeterminism. In comparison, regular LR parsing algorithms allow only one entry per table position.

Here is a trace of the input **0 1 $** according to the above context-free grammar's rules and LR parsing table. The trace will proceed through just enough steps that the parsing process becomes clear, making the remaining steps

fairly self explanatory (refer to Table 1). Initially, the system starts out with no current symbol, an empty parse forest, an empty parse tree stack, and a stack state of 0. The user enters the category terminal 0 and, by use of the lookup table, the category preterminal *B becomes the current symbol. With a current symbol of *B and a current state of 0, the action table determines that "sh 2" is the action to perform. The action "sh 2" shifts the preterminal *B (along with the corresponding terminal 0) from its current symbol status onto the parse forest (referenced by pointer **a**), and pushes state 2 (along with a parse tree stack pointer to element **a** of the parse tree stack) onto the state stack.

Next, the user enters the terminal 1, so the preterminal *B becomes the current symbol. With a current symbol of *B and a current state of 2, the action table specifies "re 1" as the next action. The action "re 1" uses rule one (S -> *B) from the context-free grammar to reduce the parse tree stack elements which point to the parse forest nodes. In this case rule one reduces only one node, so pop state stack element 2(a) which refers to the parse forest element **a**, and add the production [S (a)] for the reduced parse stack element **a** (corresponding to [S] -> [*B->0]) to the parse forest. Use the top parse forest

Table 1. PARSING STEPS FOR THE SENTENCE **0 1 $.**

State stack (with parse tree stack)	Current symbol	Next action	Parse forest
0	———	input 0	———
0	*B (0)	sh 2	———
0,2(a)	———	———	a [*B->0]
0,2(a)	———	input 1	a [*B->0]
0,2(a)	*B (1)	re 1	a [*B->0]
0	*B (1)	goto 1	a [*B->0], b [S (a)]
0,1(b)	*B (1)	sh 3	a [*B->0], b [S (a)]
0,1(b),3(c)	———	———	a [*B->0], b [S (a)], c [*B->1]
0,1(b),3(c)	———	input $	a [*B->0], b [S (a)], c [*B->1]
0,1(b),3(c)	$	re 0	a [*B->0], b [S (a)], c [*B->1]
0	$	goto 1	a [*B->0], b [S (a)], c [*B->1], d [S (b c)]
0,1(d)	$	acc	a [*B->0], b [S (a)], c [*B->1], d [S (b c)]

nonterminal S, the current state of 0, and the goto table to push a state of 1 (along with parse tree stack pointer **b** referring to the new top parse forest element) onto the state stack. The rest of the parse proceeds in the same manner.

The final parse forest in the bottom row and right most column of Table 1 corresponds in graphical form to the parse tree in Figure 3. The subscripts on nodes in Figure 3 are used to tell which parse forest elements the nodes appear in. For example the node S_d in the parse tree appears in parse forest element d [S (b c)].

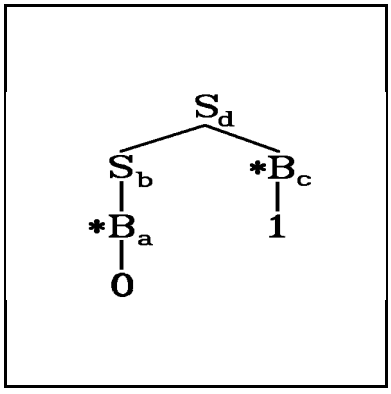

Figure 3. Parse Tree

3.4 KNUTH'S ATTRIBUTE GRAMMAR TRANSLATION ALGORITHM

As noted before, the Knuth attribute grammar translation algorithm is used to generate a possible semantic meaning from a syntactically correct parse tree produced by Tomita's parsing algorithm. To demonstrate Knuth's technique the same simple example context free grammar and parse tree from above will suffice. One possible translation would be from the original binary form to its decimal equivalent. In order to perform translations

using Knuth's algorithm the context-free grammar rules must be modified to include semantic rules, thereby producing an attribute grammar. One possible set of rules for the conversion to decimal is:

Syntactic Rules	Semantic Rules
S_0 -> S_1 *B	$n(S_0) = n(S_1) * 2 + n(*B)$
S -> *B	n(S) = n(*B)
START -> S	———

The subscripts on variables in the first syntactic rule are used only to demonstrate their correspondence to the variables in the first semantic rule. For this binary to decimal translation only one attribute (**n**) is required. The last rule, START -> S, requires no semantic rule, since it is not a part of the parse tree. This attribute grammar is represented in LISP [30] by binding it to the atom **translationrules**:

```
(setq translationrules
     '((s s b)    (((n 0) (+ (* (n 1) '2) (n 2))))
       (s b)      (((n 0) (n 1)))
       (start s)  nil))
```

where syntactic rules occupy the even numbered positions of the list and corresponding semantic rules hold the odd numbered positions. The unquoted numbers in the semantic rules along with the attribute name refer to the attribute values (**n** in this case) of the variables in the syntactic rules. Zero refers to the first variable, one to the second

variable, and so on. Thus (((n 0) (+ (* (n 1) '2) (n 2)))) is the representation of $n(S_0) = n(S_1)*2 + n(*B)$.

To translate simply start at the preterminals and proceed upwards in the tree, computing the attribute values for each node as a function of its childrens' attribute values. The attribute value for each preterminal for this grammar is just the value of its child (0 or 1). The attribute value of the root node (S_d in this case) is the result. Using the above parse tree, here is a possible sequence of the translation steps for attribute **n** going from node $*B_a$ to node S_d:

1) $n(*B_a) = 0$
2) $n(S_b) = n(*B_a) = 0$
3) $n(*B_c) = 1$
4) $n(S_d) = n(S_b)*2 + n(B_c) = 0*2 + 1 = 1$

The value of **n** at the top node is 1, therefore the result of the translation is 1.

One distinct advantage of Knuth's algorithm is the ability to pass attribute values both up and down a parse tree. Information can be passed from a child to its parent through synthesized attributes, and from a parent to its children through inherited attributes. Although only synthesized attributes were used in CONQUEST, a more ambitious treatment of a larger (and more complex) English subset would undoubtedly require the use of inherited attributes as well.

3.5 LOGICAL FORM TO SQL CONVERSION PROCEDURE

For the purposes of CONQUEST, the translation process above yields a logical form expression which must be converted to an SQL query. How does the procedure that accomplishes this work? Take a look at an example logical form expression and the corresponding SQL expression that should be produced. For this example the logical form expression is:

```
(SETX 'X1
     '(RELATION 'ZP '(PNO PNAME) '(X1 'SCREW) '(= =)))
```

The matching SQL expression is:

```
SELECT DISTINCT A.PNO, A.PNAME
FROM ZP A
WHERE A.PNAME = 'SCREW
```

What is the correspondence between this SQL expression and the above logical form expression? If you recall from Chapter 2, the logical form expression above denotes the set of all values X1 from the relation 'ZP, such that there is a tuple whose PNO attribute has the value X1, and the value of its PNAME attribute is 'SCREW. For the translation to SQL of such simple expressions, only three basic steps need be followed. First, in the SQL clause specify which relation is to be used. The logical form expression specifies that the relation ZP will be used, so in the SQL clause include the FROM clause, FROM ZP A, which specifies the relation ZP, but afterwards (in this SQL expression) refer to it with the

alias A. Therefore, in this instance A is just another name for a tuple of the relation ZP.

Second, in the logical form expression the attribute values PNO and PNAME are the ones in question, so list the (PNO, PNAME) attribute value tuples that meet the required conditions by including SELECT DISTINCT A.PNO, A.PNAME in the SQL clause. In actuality the logical form expression really only specifies the set of PNO attribute values that meet the condition, but also list the corresponding PNAME attribute value to provide added clarification.

Lastly, specify the conditions that must be met in order for the desired (PNO, PNAME) attribute value tuples to be listed. The above logical form expression specifies that only those (PNO, PNAME) attribute tuple values where the PNAME attribute value is 'SCREW should be listed, so in the SQL clause indicate this by including WHERE A.PNAME = 'SCREW.

In this simple case the conversion from logical form to SQL was fairly straightforward. Less simple cases involve embedded relations and SETX expressions, and arithmetic (such as SUM) and logical expressions (such as OR).

CHAPTER FOUR

ILLUSTRATION OF SYSTEM OPERATION

4.1 DATABASE STRUCTURE USED

The database relation around which the system was designed is named **courses** and is set up with the following structure:

NEWNO	OLDNO	CRED_HRS	NAME	PREREQ	DESCRIP
1200	320	3	MIS	1010	Info systems
2100	410	3	ASSL	1200	Assembly lang
3300	530	4	OS	2300	Op systems

where **newno**, **oldno**, **cred_hrs**, **name**, **prereq**, and **descrip** are attribute names corresponding to the hypothetical attribute values in the tuples immediately below them.

4.2 SYSTEM INITIALIZATION

For CONQUEST to perform translations from English to SQL, the main procedure **parsesentence** (see Appendix C) is called. However, before translations can begin, several variables must be initialized. First, the array **rules** (see Appendix A) is set to contain the context-free grammar rules

used by Tomita's algorithm. These rules are derived from the original attribute grammar rules (see **att_rules** in Appendix A) by calling the procedure **convert_gram** (see Appendix C).

If the LR parsing table (consisting of the action and goto tables) has not yet been determined, it must be developed, otherwise just loaded. The LR parsing table is constructed from the context-free productions bound to **rules** by calling the procedure **construct** (see Appendix C). The resulting action and goto tables (belonging to the LR parsing table) appear in Appendix A.

Next, the original attribute grammar rules (**att_rules)** are converted to **translationrules** (see Appendix A) by the procedure **convert_trans** (see Appendix C) for use later on by the Knuth translation procedures. This is done during initialization to avoid causing delays later on during execution of the interactive portion of the system.

A full trace of system operation to which references will be made from now on appears in Appendix B, Example Query 1. The first initialization the user sees is when CONQUEST asks whether or not speech input should be recognized. If speech input will be used, the COVOX speech recognition system is loaded and calibrated. Next, the user specifies whether or not spoken computer output is preferred

(CONQUEST will only speak during parsing since SQL queries aren't very well suited for speech), and if it is, the necessary programs are loaded, and the user must turn on the speaker. Lastly, the user answers whether or not translation should be timed.

4.3 PARSING PROCESS

After the system is initialized, parsing begins with the user giving his English sentence or query one word at a time, either by means of the speech recognition system or by typing. Special commands include **back** which causes the system to erase the last word of the current sentence, **clear** which clears out the entire sentence, and **quit** which causes the procedure to abort. As can be seen from the example, the user continues entering words (category terminals) with no system interference as long as no errors are made, or no special commands are entered. If an error occurs, CONQUEST signals the error, checks all the possible category preterminals (bound to the variable **preterms** in Appendix A) to see which ones offer a legal continuation, then prompts the user with a list of the possible grammatical category preterminals and an example of the terminals that fall within each category (determined from the array **lookup** in Appendix A). If the user issues a **back** or **clear**, the system

prompts the user with the new sentence, and says to begin entering words again.

If all goes as planned, parsing continues with the entire sentence accepted as soon as the user enters the symbol **$**. In the case of Example Query 1, the accepted sentence is "Give a description of course number 1200" and the resulting parse tree is bound to **fa**.

4.4 TRANSLATION PROCESS

As soon as the sentence is syntactically accepted, user interaction with the system has basically ended, and translation begins. CONQUEST now calls the procedure **converttree** (see Appendix C). This converts the parse tree in the array **fa** to a parse tree in the list **parses**, in order to be compatible with Petrick's translation procedures. The procedure **ktransbt** (see Appendix C) is invoked to translate the parse tree to logical form, according to the attribute grammar rules in **translationrules**, and the result is bound to the atom **translation1**.

4.5 LOGICAL FORM TO SQL CONVERSION PROCESS

To finish the conversion to SQL, **translation1** is fed as input to the procedure **lftosql1** (see Appendix C), and the result is bound to the atom **translation2**. The atom

translation2 is in turn passed to the procedure **printsql1** (see Appendix C) which prints the SQL query.

4.6 OTHER EXAMPLES

Traces of some other example sentences which are handled by CONQUEST appear in Appendix B, Example Queries 2 through 11. The output appears precisely as it would during normal system operation with no variable trace information cluttering up the screen. Other sentences of the same types shown are also possible by exercising a slight variation of the terminals chosen.

CHAPTER FIVE

CONCLUSION

At the present time CONQUEST is a relatively user-friendly system due to several pronounced advantages. First, it allows communication by speaking, a method by which most users are likely to be more comfortable. Second, CONQUEST allows users to communicate in a fairly immediate manner. The speed of parsing within CONQUEST is very fast, taking only seconds for the simple queries allowed (see Appendix B, Example Query 1 for the timing). Either with error-free queries or with error-filled queries, the user experiences only minimal delay before being allowed to enter the next in a sequence of words. This is perhaps close enough to continuous speech that the perception of talking with someone rather than something is enhanced.

Another advantage is that CONQUEST provides helpful error diagnostics to the user during query entry. This obviously makes CONQUEST more user-friendly than systems which simply signal an error, but offer no suggestions about what next to try. It seems in this way that the user

communicates more with the system (especially when speech output is used), rather than at the system.

In addition, by no means rivaling high speed typing, the COVOX speech recognition system does function at a relatively high speed, offering CONQUEST the advantage (in the event that the spoken word is recognized properly) of mistake free data entry. It seems more natural to speak a word correctly and be understood, than to incorrectly type a word that will most definitely be misunderstood.

Despite these advantages, CONQUEST does have some weaknesses. As you can see, there is little variety in the sentences CONQUEST can currently handle. As mentioned before, there were two main reasons for this restriction. First, in order to use the COVOX speech input system, Goldworks I had to be chosen, limiting the grammar size to a small set of simple rules, and, as a consequence, allowing representation of very few sentence complexities and paraphrases. Second, the low vocabulary size of the COVOX speech recognition input system also limited sentence complexity and paraphrasing. These two restrictions led to the incorporation of a context-free grammar of limited size. This, in turn, restricted the complexity and variety of sentences that could be processed. As it now stands, this restricted set of sentences severely limits the practicality of CONQUEST, because it both limits the number of query

types a person can make and limits the number of available paraphrases a user can make. Since Petrick's procedures were already programmed to handle a high variety of complex queries, getting CONQUEST to handle more advanced sentences through the use of more advanced grammar representations and more powerful speech recognition input, would be a logical improvement. As mentioned in Chapter 3, it would also be advantageous to integrate with the speech input system, the ability to recognize numbers, especially in the case of numerically oriented database relations.

Time limitations also restricted CONQUEST development, although they were not as critical. In order to get CONQUEST in working order, no great deal of time was spent making any one part of the system especially sophisticated, or making components well suited to each other. Since the simple grammar type was already the dominant limiting factor and required no additional component efficiency, there was no reason to make the rest of the system particularly efficient. One example of this was the different grammar and tree structure representations used throughout CONQUEST. It would be preferable to modify some of the components (procedures) so that it isn't necessary to convert grammars from Tomita's array type to Petrick's list type, or from Tomita's parse forest to a list of parse trees for Petrick's procedures. Practical situations requiring larger grammars

and larger parse forests (trees) would absolutely demand optimizations such as these, in order to avoid unduly slowing down the system. All in all, it seemed that a great deal of time and effort was spend translating procedures from their original LISP dialects to the Common LISP dialect dictated by the Goldworks I system, and integrating components so that they could correctly operate together.

With these time limitations CONQUEST never evolved to the point where the generated SQL queries were fed as input to an actual database system, in turn yielding information from a database relation. In the future an obvious next step would be to find an appropriate interface between CONQUEST and some SQL compatible database system (such as dBASE IV).

If and when CONQUEST is interfaced with a database system, and the user is able to enter a natural language query and receive information back, how does he know that the information is valid? An apparent solution would be the inclusion of a feature to convert the generated SQL query back to English so that the user can determine by an alternate wording whether or not the meaning of the original English sentence was taken as intended. [32]

These limitations notwithstanding, the abilities of CONQUEST as it now stands and the future possibilities of a system of this type demonstrate the benefits of integrating

speech and nondeterministic syntactic processing to produce a more user friendly system in comparison to menu-cluttered and/or deterministic natural language front end systems.

LIST OF REFERENCES

1. Plath, W. J. "REQUEST: A Natural Language Question-Answering System." IBM Journal of Research and Development 20 (July 1976): 326.

2. Hendrix, G. G. et al. "Developing a Natural Language Interface to Complex Data." ACM Transactions on Database Systems 3, No. 2 (June 1978): 105-147.

3. Woods, W. A., Kaplan, R. M., and Nash-Webber, B. "The Lunar Sciences Natural Language Information System." BBN Report. 2378 Cambridge, MA: Bolt Beranek and Newman, Inc., 1972.

4. Thompson, F. B. et al. "REL: A Rapidly Extensible Language System." In Proceedings of the Twenty-fourth National Conference of the ACM New York: (1969), 399.

5. Reisner, P. "Human Factors Studies of Database Query Languages: A Survey and Assessment." Computing Surveys 13, No. 1 (March 1981): 13-31.

6. Herschmann, R. L., Kelley, R. T., and Miller, H. C. "User Performance with a Natural Language Query System for Command Control." NPRDC TR 79-7 San Diego, CA: Navy Personnel Research and Development Center, 95152, January 1979.

7. Tennant, Harry R. et al. "Menu-based Natural Language Understanding." In 21st Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference. Cambridge, MA: MIT (June 1983), 151-158.

8. Ross, Kenneth. "Parsing English Phrase Structure." Ph.D. dissertation, University of Massachusetts, Amherst, September 1981.

9. Voice Master Key User Manual. Version 2.00, Eugene, OR: COVOX, Inc., 1989.

10. Goldworks Expert System User's Guide. Version 1.0, Cambridge, MA: Gold Hill Computers, Inc., 1987.

11. Speech Thing User Manual, 4th ed., Eugene, OR: COVOX, Inc., 1989.

12. The VoiceScribe-1000 User System and The VoiceScribe-1000 Developer's System. Dragon Systems, Inc., Newton, MA (March 19, 1990).

13. DragonDictate. Dragon Systems, Inc., Newton, MA (March 19, 1990).

14. Petrick, S. R. "A Recognition Procedure for Transformational Grammars." Ph.D. dissertation, MIT, Cambridge, MA, 1965.

15. Peters, P. S., and Ritchie, R. W. "On the Generative Power of Transformational Grammars." Information Sciences 6 (1973): 49-83.

16. W. A. Woods. "Transition Network Grammars for Natural Language Analysis." C. Association for Computing Machinery 13, No. 10 (Oct. 1970): 591-606.

17. Gazdar, G. et al. Generalized Phrase Structure Grammar. Oxford, UK: Blackwell, and Cambridge, MA: Harvard University Press, 1985.

18. Pereira, Fernando C. N., and Warren, David H. D. "Definite Clause Grammars for Language Analysis: A Survey of the Formalism and a Comparison with Augmented Transition Networks." Artificial Intelligence 13 (1980): 231-278.

19. Griffiths, T. V., and Petrick, S. R. "On the Relative Efficiencies of Context-Free Grammar Recognizers." Communications of the ACM 8, No. 5 (May 1965): 289-300.

20. Slocum, J. "A Practical Comparison of Parsing Strategies." In Proceedings of the 19th Annual Meeting of the ACL, Stanford, CA: Stanford University (1981), 1-6.

21. Petrick, S. R. "Parsing." In Encyclopedia of Artificial Intelligence. Ed. S. Shapiro. John Wiley & Sons, Inc. (1987), 687-696.

22. Tomita, Masaru. "An Efficient Augmented-Context-Free Parsing Algorithm." Computational Linguistics 13, Nos. 1-2 (Jan.-June 1987): 31-46.

23. Younger, D. "Recognition and Parsing of Context-Free Language in Time n^3." Information and Control 10 (1967): 189-208.

24. Earley, J. "An Efficient Context-Free Parsing Algorithm." Communications of the ACM 6, No. 8 (1970): 94-102.

25. Tomita, Masaru, "An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications," Ph.D. dissertation, Carnegie Mellon University, May 1985, distributed by University Microfilms International, Ann Arbor, Mich.

26. Petrick, S. R. "Semantic Interpretation in the REQUEST System." In COMPUTATIONAL AND MATHEMATICAL LINGUISTICS, Proceedings of the International Conference on Computational Linguistics. Eds. A. Zampolli and N. Calzolari, Pisa 27 VIII-I/IX 1973, Casa Editrice Olschki, Firenze, Vol. II., 585-610.

27. Irons, E. T. "A Syntax-Directed Compiler for ALGOL 60." Communications of the ACM 4, No. 1 (January 1961): 51-55.

28. Montague, R. "The Proper Treatment of Quantification in Ordinary English." In Formal Philosophy. Ed. R. M. Thomason. Yale University Press: (1973).

29. Knuth, Donald E. "Semantics of Context-Free Languages." Mathematical Systems Theory 2, No. 2 (New York: Springer-Vertag, June 1968), 127-145.

30. Petrick, S. R. "On the Use of Syntax-Based Translators for Symbolic and Algebraic Manipulation." In Proceedings of the SECOND SYMPOSIUM ON SYMBOLIC AND ALGEBRAIC MANIPULATION. Ed. S. R. Petrick. New York : ACM SIGSAM (March 1971), 224-237.

31. Picone, Joseph. "Continuous Speech Recognition Using Hidden Markov Models." IEEE ASSP Magazine 7, No. 3 (July 1990): 26-41.

32. Miller, Todd A. "The SQLENG System: An SQL To English Translator Using Attribute Grammars." University of Wyoming, Laramie, August 1989.