SPEECHACTS: A TESTBED FOR CONTINUOUS SPEECH APPLICATIONS

Paul Martin
Sun Microsystems Laboratories, Inc.
Two Elizabeth Drive
Chelmsford, MA 01824-4195
paul.martin@east.sun.com

Andrew Kehler
Harvard University*
Aiken Computation Laboratory
33 Oxford Street
Cambridge, MA 02138
kehler@das.harvard.edu

Abstract

The SpeechActs system is a testbed for building computer applications utilizing continuous speech input and speech synthesis output. It supports a variety of speech recognition (SR) systems and text-to-speech (TTS) generators; our main goal is to produce a generalized interface scheme allowing new research systems to be substituted easily. The essential components of the system, in addition to the externally-developed SR and TTS modules, are our natural language interpreter SWIFTUS, our Unified Grammar (UG) compiler, and the discourse manager. SWIFTUS provides the tools necessary to easily develop natural language interfaces between SR systems and end applications that exhibit a suitable granularity of analysis. The Unified Grammar compiler creates grammars supporting a variety of SR systems and NLP modules from one, module-independent formalism. The discourse manager is currently an under-developed prototype that coordinates user interactions between a co-existing suite of different speech applications. To date, several prototypes have been developed for a variety of end applications using several different SR and TTS systems.

Introduction

SpeechActs is a prototype system for building computer applications which recognize and synthesize speech. Our goal is to have a general framework to support the use of a speech recognizer (SR) to determine the word string uttered by a user, and have the result passed to a natural language (NL) interpreter which produces commands for a variety of end applications.

Reaching this goal within the current state of technology involves facing significant challenges. First, current SR systems cannot reliably recognize significant amounts of continuous speech without encoding constraints to reduce the perplexity of the recognition task, usually realized by a combination of a limited vocabulary and a constrained grammar. For instance, fully integrated special-purpose systems (the *vertical model*) have demonstrated that sufficiently constrained SR systems can be effective in limited domains such as booking air travel (Kubala et al., 1992; Zue et al., 1992). However, these systems achieve low perplexity by building the application constraints into the SR system, either by hand or by statistical training based on a domain corpus. This specialized approach has two major disadvantages: (1) only the system developers themselves are qualified to maintain and extend the system, and (2) the system cannot be easily ported and adapted to other domains and applications. Hence, developing systems that are accurate enough to be useful while maintaining some amount of portability and adaptability to new domains remains a complex and unsolved problem.

A second challenge is developing an appropriate natural language processing (NLP) component to serve as the interface between the SR system and the command language of the end application. While NLP technology is itself in its infancy with regard to true understanding, the problem is compounded by the fact that current SR systems will often give it an inaccurately-recognized word string as input. Therefore, a major issue is determining the appropriate level of "coarseness" of language analysis for this type of application, as well as the development of mechanisms for dealing with input that is likely to contain misanalyzed words.

A third challenge is managing the knowledge sources for the systems, especially the grammars. Many SR systems utilize simple grammars to lower perplexity, that is, to constrain the word sequences that the system will consider during recognition. NLP modules generally also utilize a grammar, but instead for the purpose of recovering a syntactic structure to serve as the basis for semantic interpretation. For the purpose of development and maintenance, however, it is desirable to have one common source grammar representation that can be compiled into each of the formalisms. This scheme has two main advantages: (1) only one grammar for an application will have to be maintained instead of two; consistency between the SR and NLP system grammars is guaranteed by virtue of both being generated from the same source, and (2) new SR systems can be more easily substituted in the future: the developer need only

*This work was done while the second author was at Sun Microsystems Laboratories, Inc.

write a new compiler to the grammar specification for the new system, avoiding the need to reengineer new special-purpose grammars for each of possibly many existing applications. Unfortunately, SR grammars often must sacrifice generality to reduce perplexity, whereas generality in NLP grammars is required for modularity, extensibility, and portability. Keeping the grammar specification general, while allowing automatic compilation to a low-perplexity grammar for an SR system is a challenging task.

We have addressed these three challenges in our development of the SpeechActs system. In the following section, we outline the architecture of the system.

SpeechActs Modules

This section describes the implemented modules of SpeechActs, along with a brief discussion of SR and TTS modules with which we have integrated to date. We will briefly sketch the Discourse Manager module in the conclusion, as implementation is not yet completed.

Off-The-Shelf Components

Speech Recognizer As we have noted, SpeechActs was developed to support a wide variety of existing SR systems, as well as to be easily adaptable to future SR systems. We use software-only[1] continuous SR systems based on Hidden Markov Model matching networks representing acceptable word sequences as triphones. All are speaker-independent, and use only the triphone models trained by the SR developers. A phoneme dictionary of our vocabulary connects to the trained triphones; we use no preprocessing for coarticulation.

We have experimented with Texas Instruments' DAGGER recognizer (Hemphill, 1993), SphinxII from Carnegie Mellon University (Huag et al., 1993), Sphinx (Lee, 1989), and Bolt Beranek and Newman's HARK system (Smith and Bates, 1993). As our experience is most extensive with the latter two systems, we will concentrate on those.

Sphinx accepts a limited BNF grammar (no explicit optional elements are allowed), which is used to compile a set of allowable word pairs (bigrams). Unfortunately, this reduction allows recognition of strings that are not covered by the BNF but which are composed entirely of allowable word pairs. For example, compiling the grammar $s := abc \mid zbd$ results in Sphinx admitting the string *abd*. Using bigram statistics could help, but cannot rule out the example above.[2]

The grammar used by HARK allows BNFs with regular expression operators in the rules, but does not handle recursive definitions. Therefore, the most natural encoding of certain grammar rules is prohibited, such as specifying prepositional phrase attachment to noun phrases:

```
NP := NP PP
```

Optimizing grammars to share common expressions is required to control the size of their recognition graph, often at the expense of expressing the grammar in the most lucid fashion. HARK enforces all the constraints in its grammar, always returning a grammatical word string. HARK performs well with high-quality microphone input, but the telephone models are not yet as well trained as those from Texas Instruments.

Text-to-Speech Generator As with speech recognizers, we utilize off-the-shelf text-to-speech (TTS) systems. To date, we have used the DecTalk and Centigram systems. DecTalk (Klatt, 1990) is run over a LAN on an IBM-compatible, and produces quite recognizable output (it sounds a bit like Dennis Klatt), but does not respond while speaking, so cannot be interrupted. *Centigram* is the TruVoice product from Centigram Communications Corporation (developed with SUN) that runs directly on a Sparc. It produces recognizable output, and guesses pretty well on words not in its dictionary. It uses the audio device exclusively, refusing to share with an SR system except sequentially. The phoneme dictionary scheme for each SR (except Sphinx/SphinxII) and each TTS is unique, so a small effort is needed to map one form to the other.

SWIFTUS

Integrating SR with an end application requires translating strings of words from the SR into the application's command language. Given that a truly robust NLP capability is beyond the state-of-the-art, system designers must determine the appropriate level of granularity of language processing for the task at hand. A coarse-grained approach (e.g., simple keyword analysis) is simple and can achieve 100% matching accuracy, but is probably too weak to make the distinctions necessary for successful interpretation of utterances. Theoretically, a fine-grained approach (e.g., full syntactic, semantic, and pragmatic analysis) could capture such distinctions, but the inaccuracies inherent in current NL and SR technology results in a system that is too "deep" to be useful. The development of SWIFTUS (SoftWare Instead of Fastus for Text UnderStanding) arose from our attempt to analyze (sometimes inaccurately

[1] The rising computational power of workstations and support for multiple processors with shared memory both argue powerfully against committing to any SR system requiring special DSP hardware. Uhler (personal communication) argues that properly programmed cpu chips in a multiprocessor provide the performance formerly available only from a dedicated DSP chip, and multi-media has prodded most manufacturers to include a digitizing chip as a component of basic workstation hardware.

[2] We can close this loophole by processing our grammar to use variants of the non-terminals based on context, for instance, rewriting the second "b" in our example above as "b-sub-z", where we choose how much preceding context to encode.

recognized) spoken utterances with the proper flexibility and depth of processing for speech-to-command-language translation. Loosely based on the FASTUS system (Hobbs et al., 1993), SWIFTUS is a multi-level finite state automaton pattern recognizer supported by a range of facilities for developing NLP applications.

Lexicon and Knowledge Base Facilities The *Lexical Processor* consists of lexicons as data and a set of morphology routines. Although lexicons are optional, their development eases the process of writing patterns (as well as the grammars from which patterns will automatically be created), and the resulting patterns are more concise, general, extensible, and portable. Separate general and domain-specific lexicons are maintained to facilitate the sharing of general lexicons among applications.

Lexical entries use a standard feature-value pair representation, with common fields including part-of-speech, agreement information, morphological information (such as irregular forms), and references to an optional knowledge base entity. SWIFTUS's morphological component defines variants from a set of rules of English morphology, unless overridden by an explicit irregular form specification. Roots of words are maintained for checking within patterns. Acronyms and alternate spellings can also be declared from within entries. Multi-word phrases that specify a single concept (e.g., "United States of America") are entered separated by hyphens; patterns are automatically defined to recognize the phrase when its words appear unhyphenated in the input string.

The *Knowledge Base* module consists of knowledge bases and simple subsumption and inheritance routines. They are optional, but in both general and domain-specific versions they can capture semantic generalizations inappropriate for the lexicon, such as marking a semantic class (e.g. "display acts": *display*, *show*, *present*, etc.) to support testing in patterns.

Pattern Recognizer The *Pattern Recognizer* is comprised of pattern recognition routines and a set of recognition patterns (i.e., the grammar). Patterns are quadruples: the name, the pattern body, and two sets of Lisp code augmentations. The pattern body is matched against an input string; upon a successful match, the first set of augmentations is executed. If they succeed, then the pattern is recognized and a result form is returned by the second set of augmentations.

The Pattern Recognizer is essentially a cascaded FSA-based system. It supports the standard regular expression operators, including optionality, disjunctive OR, and Kleene plus and star. Patterns can reference patterns to arbitrary levels of depth.

In consuming words, SWIFTUS has primitives for testing: the actual word, its root, its part-of-speech, its knowledge-base class (directly or by subsumption), the value of a feature-value pair in the lexicon, and/or the negation of any of these. SWIFTUS also allows arbitrary functions to serve as predicates for tests. Variables are bound to matched structures; wildcards are supported to help abate the inaccuracy of current SR systems.

Loading rules into SWIFTUS builds corresponding FSAs using standard techniques (Aho et al., 1988). Tests on lexical entries are converted into compiled Lisp functions on FSA arcs. A trivial definition is sent for words not in the lexicon so that predicate checks for words and wildcards can still succeed.

SWIFTUS supports a set of second-pass (and third-pass, etc.) patterns, for use when the first pass returns no matches. This "safety net" of patterns is generally looser than the first-pass analysis, e.g., they may look for selected keywords and wildcards. The passes are separated to allow the preference for a first-pass match.

The feature-value structure produced by the action augmentations of a matched pattern are returned to the calling pattern. At the top level, SWIFTUS feeds the result form to the *Translation* module, which creates commands in the end-application's command language. These application-specific routines and data are separated from the SWIFTUS patterns for pattern modularity and portability. SWIFTUS includes additional tools for checking data integrity and for regression testing.

Assessing SWIFTUS

Our main goal in developing SWIFTUS was to give developers the flexibility necessary to design robust translators from spoken utterances to application commands given the current state of SR and NLP technology. We have found that it succeeds in doing so by providing capabilities that are flexible in several respects:

Looseness/Tightness of Patterns: While the current state of speech recognition technology is less than perfect, it is at the stage where a majority of words can often be successfully recognized. In particular, recognition tends to be more accurate for longer "content" words (e.g., nouns, verbs) than for smaller closed-class words (e.g., determiners, prepositions).

SWIFTUS's use of wildcards in patterns helps alleviate the problems with such recognition errors. In restricted applications, the command expressed in an input string can usually be ascertained from the content words and their ordering; a particular preposition or determiner used is often less important. Thus, whereas a misrecognition of the word "of" as "a" using standard parsers/grammars would generally lead to parsing failure (since the first is a preposition and the second is a determiner), the use of wildcards can make the patterns less sensitive to such recognition errors.

Degree of Linguistic Sophistication: It was desired that SWIFTUS have the capacity for utilizing sophisticated linguistic information, but also that it allow for rapid prototyping by developers with varied degrees of linguistic training. Utilizing lexicons and knowledge bases while not requiring them allows for a first-cut system to be quickly prototyped; more general and comprehensive linguistic data can then be encoded later in the prototyping process when the developers have a better feel for the application. Furthermore, users that don't have the requisite training to encode complex linguistic information can still specify (albeit simple) patterns to get an application up and running.

Dependence on Domain and Application: Some domains and applications are so limited that the best "bang-for-the-buck" comes from a very coarse-grained, domain and application dependent system design. In other cases, a greater degree of linguistic sophistication is more suitable for the task. With SWIFTUS, data sources can be made as domain and application independent or dependent as is deemed appropriate. While its facilities allow adherence to "independence" criteria associated with good system design and portability, these are not imposed on the designer in cases where it is not appropriate.

Fallback strategies: In some cases, more than one level of sophistication of analysis is optimal. For instance, it may be best to attempt a full parse initially, but to fall back to a search for keywords or isolated constituents when the more comprehensive analysis fails. The second-pass parsing mechanism affords this flexibility.

SWIFTUS has been successfully integrated into several working applications, including interfaces into the FrameMaker document management system and the Unix-based CalendarManager system. While no formal benchmarking of speed has been performed, analysis tends to be very fast (under a second) on the 8-12 word commands that are typical for these applications. In sum, we find it to lie at about the correct level of granularity of language processing to support our applications.

The Unified Grammar

A major goal of our work is to provide mechanisms to support the maintainability, portability, and extensibility of our speech applications. As we described in the introduction, methods for generating SR system and NLP system grammars from a common, module-independent formalism help maintain consistency between each as they co-exist in a single system, as well as allow for the easy substitution of new SR (and possibly, NLP) modules without requiring that the grammars for all existing applications be reengineered. Again, however, the fact that the grammars for SR and NLP systems serve different purposes make the design of such a formalism challenging. Whereas NLP systems are sensitive to the syntactic structure that results from parsing (as this is the basis for semantic interpretation), SR systems use grammars only to constrain the allowable word sequences that can be recognized, without regard to this structure. Whereas NLP grammars can (and generally should) remain general by encoding (perhaps domain-dependent) semantic constraints in augmentations instead of in the rules themselves, such constraints need to be "compiled in" to the grammar rules for SR systems. Such constraints are often powerful in limiting the perplexity of the recognition task, so they cannot be put off until the NLP module.

The Unified Grammar allows for the natural expression of general grammars, along with the specification of analysis-constraining and structure-building augmentations. The conflicts noted above are dealt with by preprocessing most of the augmentations at compile time, effecting their constraints in an expanded, more concrete grammar for the SR system.

Unified Grammar Parser

The unified grammar parser accepts definitions of nonterminals as a BNF production followed by augmentations in a Pascal-like form.[3] The augmentations can declare the head of the production (for automatic feature inheritance), supply tests required to accept the rule, and set result features upon acceptance. Both testing and result setting can be done in a simple declarative form or through an escape to Lisp tests or actions for more complex operations. The parser converts these expressions into a Lisp form which is later compiled for the SR system when the full lexicon and all rules have been defined.

Example of a Unified Grammar Rule

The following is an example of a rule in the Unified Grammar formalism; the first statement is the grammar rule itself, the remaining statements form the augmentation:

```
{ date/nop3 := ("next"|"last")/indx
               sem=cal_unit;
   head cal_unit;
   cal_unit.plural = nil;
   index_adj := indx.word; }
```

In this example, the name of the non-terminal *date* is followed by an (optional) identifying tag *nop3* (there to distinguish multiple rules for debugging), followed by a right-hand-side (RHS) specification. The optional tag *indx* is specified for easier reference to the results of

[3]We thank Peter Norvig for substantial suggestions on both the form and the implementation of the grammar parser.

the rule match from within the augmentation; it will be bound to whichever word is matched. The term *sem=cal_unit* is shorthand for a word whose semantics feature (*sem*) is *cal_unit*.

The *head* statement is used to invoke automatic inheritance of lexical head information; it is needed only when the RHS doesn't implicitly specify it through default declarations. The statement *cal_unit.plural = nil* is a test that must be true for the rule to be accepted; in this case it disallows plurals such as "next weeks". The statement *index_adj := indx.word* is an action on the resultant feature structure applying after rule acceptance; it records the lexical choice as the *index_adj* feature of the resultant structure.

Unified Grammar Compiler

The Unified Grammar Compiler compiles the Unified Grammar specification into rules in the format required by a given SR system. The challenge here is to recognize and compile-in as many of the constraints specified in the augmentations as possible, in order to reduce the perplexity to the greatest extent. The internal form is not specialized for any particular SR system; we currently only assume a CFG without augmentations.

Lexicalizing Simple Cases In the simplest case, the compiler replaces a reference to some feature, e.g., *sem = be*, with a newly-created nonterminal category which it defines as the disjunction of all the words in the lexicon which have the correct feature binding (e.g., in this case the set contains *be*, *is*, *was*, etc.). This new nonterminal is named systematically (e.g., *"sem_be"*) and cached so that later uses will share it rather than re-create it. This is the "base-case" for the algorithm; the more challenging issues are handling complex predicates and nonterminal categories.

Complex Predicates Complex predicates arise because we incorporate not just the lexical restrictions that appear in the BNF of the rules, but also the augmented tests, including both the required equality tests (e.g. the requirement that there not be a plural marker on the *cal_unit* feature in our example rule above), and as much of the augmentations expressed as escapes to Lisp code as can practically be preprocessed. Preprocessing of the Lisp code is desirable because the expressive power of the equality constraints falls far short of all the augmentations a grammar designer might wish to include, and it is necessary since the non-analytic solution of treating the Lisp code as a continuation to be evaluated as a black box requires establishing bindings of arbitrary complexity to support a meaningful evaluation.

The parser divides the augmentations into tests and post-matching actions (i.e., structure-building operations); only the test predicates are analyzed. The compiler macro-expands to canonicalize code, and then decomposes compound forms such as *if-then-else* into the multiple branches needed to lexicalize. The nonterminals created are named and cached to reduce duplicate efforts.[4]

Consider the following rule, which rewrites to the lexical entries for "teen numbers":

```
{ teen/cardinal := cat=num;
   num.cardinal = t;
   lispp(and (< 12 (get-v-q num numval))
             (> 20 (get-v-q num
                            numval))); }
```

We lexicalize by creating a nonterminal named *cat_num_eq_cardinal_t_.lt._12_numval_.gt._20_numval*, which will include the words for thirteen through nineteen. Although this predicate comprises four tests, it is a relatively simple case since only one predicate at a time must be tested, and the items being tested are all terminals. Many alternative expressions of this code are too complex to decompose, e.g., we do not analyze the contents of most lambda expressions, so "let-binding" *numval* (to save multiple fetches) might block precompilation.

Pairing Variable Values Let us now consider the following example, covering a small class of questions:

```
{ question/do := "what" root=do cat=pronoun
                  root=have;
   do.3rd-person = pronoun.3rd-person; }
```

The augmentation above requires the *3rd-person* feature to have the same binding on the verb whose root is *do* as on the pronoun, blocking such questions as "what do he have?" and "what does I have?". Since both the verb and the pronoun are lexical, we handle this case by mapping through the lexicon, collecting the possible values for the feature. We then expand this set into all possible combinations; here the feature must be true on both or absent on both. Only cases where an actual lexical item of each type exists are generated, so if no third person singular pronouns were in the lexicon, this rule could be reduced further, allowing only those do-verbs not marked as third-person.

Recursing through Nonterminals The most difficult case arises when the predicates are applied to nonterminals created by rules more complex than disjoined terminals. Here we must "follow the feature" by recursing through the rules to determine whether it is solely rooted in lexical inheritance. For instance, if we were to substitute a nonterminal such as "aux-phrase" for the do-verb in the example above, then we would have such a case. By examining all the rules that produce "aux-phrase", the compiler can determine whether the feature being tested is inherited from the head of each rule or set locally in an augmentation action.

[4] Bob Sproull has suggested an indexing scheme for predicates which disregards their creation order; it will improve the current caching scheme significantly for large grammars, but is not yet urgent for the smaller ones we are using.

When the feature in question is inherited or lexically grounded in all relevant constructions, the compiler recurses through the head categories, and succeeds when it locates a lexical grounding. Once lexical grounding is established, the same expansion techniques used for lexical items are applied.

If a feature value is explicitly set by a rule, the compiler gives up the effort to enforce this restriction at SR time, although the constraint will still be propagated to the NLP grammar as an augmentation.[5]

The Limits of Analysis The shortcomings of the analysis will result in cases where constraints cannot be applied at SR time; this can either be tolerated as a modest price for a more perspicuous grammar, or the constraints in question can be made more explicit in the grammar. For instance, consider the following rule:

```
{date/noprep2 := cardinal sem=cal_unit "ago";
     head cal_unit;
     lispp(if (= 1
                 (get-v-q cardinal numval))
             (not (get-v-q cal_unit
                              plural))
             (get-v-q cal_unit plural));
     num_cal_units := cardinal.numval; }
```

This rule enforces number agreement, allowing "one week ago" or "twenty seven days ago" but disallowing the converse cases. Decomposition of the Lisp predicate is accomplished by the compiler, but in following the source of the *numval* feature through the various constructions that produce cardinals, the compiler encounters the following rule:

```
{ cardinal/big := tensmult [smallcardinal];
   head tensmult;
   lispa(set-hd-q numval
          (+ (get-v-q tensmult numval)
             (if (val smallcardinal)
              (get-v-q smallcardinal numval)
              0))); }
```

Looking at this rule, it is clear that no cardinal formed this way will have a *numval* less than 10, but the current compiler finds the code too complex to decompose. The grammar writer can still get the number agreement by rewriting the original rule to explicitly refer to *one* in the singular case, thus simplifying sufficiently to allow lexicalization at compile time.

Using Unified Grammar Despite the limitations of the current implementation, the use of a Unified Grammar offers a powerful development tool for grammars to be compiled into forms suitable for both SR and NLP. The automatic production of the more concrete grammars required for SR relieves the developer of a thankless chore, and guarantees consistency between the SR and NLP processing. A side benefit is allowing the extra expressive power of full BNF with augmentations and lexical features available for specifying the SR grammar.

[5]There are some cases which could yield to further analysis, for instance, consider those cases where the only value set explicitly is disallowed in the test. However, these cases are in the minority of those resisting pre-compilation.

Conclusions

SpeechActs is serving its designed function as a testbed for developing speech-based applications. The modules developed for it complement the more obviously-necessary SR and TTS components and address interesting issues in integrating a language understanding component. We have been able to develop several speech applications using this framework, and are proceeding apace in producing a non-trivial suite of co-existing applications.

We are currently developing a Discourse Manager component to coordinate such a suite of co-existing applications. Our current, less-ambitious version keeps track of the current application by following along in an FSA script, and maintains such bindings as the current user, current time, and likely pronominal referents on a blackboard shared among all the applications. We are currently extending this module to incorporate theories of discourse segment pushing and popping (Grosz and Sidner, 1986) in order to make movement between applications approach the fluidity of movement between topics in human-to-human conversation.

Acknowledgments

The ideas in SpeechActs and the work required to implement its many component parts were a group effort by the authors and Nicole Yankelovich, Eric Baatz, Gina Levow, and Stuart Adams.

References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1988. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, MA.

Barbara Grosz and Candace Sidner. 1986. Attention, intentions, and the structure of discourse. *Computational Linguistics*, 12(3):175–204.

Charles Hemphill. 1993. DAGGER. Directed acyclic graphs of grammars for enhanced recognition, user's guide and reference manual. Technical report, Texas Instruments, May.

Jerry Hobbs, Douglas Appelt, John Bear, David Israel, Megumi Kameyama, and Mabry Tyson. 1993. FASTUS: A system for extracting information from text. In *Proceedings of the ARPA Human language Technology Workshop*, pages 133–137, Plainsboro, NJ, March.

Xuedong Huag, Fileno Alleva, Mei-Yuh Hwang, and Ronald Rosenfeld. 1993. An overview of the

SPHINX-II speech recognition system. In *Proceedings of the ARPA Human language Technology Workshop*, pages 81–86, Plainsboro, NJ, March.

Dennis Klatt. 1990. DecTalk user's manual. Digital Equipment Corporation Report.

Francis Kubala, Chris Barry, Madeleine Bates, Robert Bobrow, Pascale Fung, Robert Ingria, John Makhoul, Long Nguyen, Richard Schwartz, and David Stallard. 1992. BBN BYBLOS and HARC February 1992 ATIS benchmark results. In *Proceedings of the DARPA Speech and NL Workshop*, pages 72–77, Harriman, NY, February.

Kai Fu Lee. 1989. *Automatic Speech Recognition: The Development of the SPHINX System*. Kluwer Academic Publishers, Boston, MA.

Graeme W. Smith and Madeleine Bates. 1993. Voice activated automated telephone call routing. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications*, Orlando, FL, March.

Victor Zue, James Glass, David Goddeau, David Goodine, Lynette Hirschman, Michael Phillips, Joseph Polifroni, and Stephanie Seneff. 1992. The MIT ATIS system: February 1992 progress report. In *Proceedings of the DARPA Speech and NL Workshop*, pages 84–88, Harriman, NY, February.