SOUP: A PARSER FOR REAL-WORLD SPONTANEOUS SPEECH

Marsal Gavaldà
Interactive Systems, Inc.
1900 Murray Ave. Suite 203
Pittsburgh, PA 15217, U.S.A.
marsal@interactivesys.com

Abstract

This paper describes the key features of SOUP, a stochastic, chart-based, top-down parser, especially engineered for real-time analysis of spoken language with very large, multi-domain semantic grammars. SOUP achieves *flexibility* by encoding context-free grammars, specified for example in the Java Speech Grammar Format, as probabilistic recursive transition networks, and *robustness* by allowing skipping of input words at any position and producing ranked interpretations that may consist of multiple parse trees. Moreover, SOUP is very efficient, which allows for practically instantaneous backend response.

1 Introduction

Parsing can be defined as the assignment of structure to an utterance according to a grammar, i.e., the mapping of a sequence of words (utterance) into a parse tree (structured representation). Because of the ambiguity of natural language, the same utterance can sometimes be mapped into more than one parse tree; statistical parsing attempts to resolve ambiguities by preferring most likely parses. Also, spontaneous speech is intrinsically different from written text (see for example [Lavie 1996]), therefore when attempting to analyze spoken language, one must take a different parsing approach. For instance one must allow for an utterance to be parsed as a *sequence* of parse trees (which cover non-overlapping segments of the input utterance), rather than expect a single tree to cover the entire utterance and fail otherwise.

The SOUP parser herein described, inspired by Ward's PHOENIX parser [Ward 1990], incorporates a variety of techniques in order to achieve both *flexibility* and *robustness* in the analysis of spoken speech: flexibility is given by the lightweight formalism it supports, which allows for rapid grammar development, dynamic modification of the grammar at run-time, and fast parsing speed; robustness is achieved by its ability to find multiple-tree interpretations and to skip words at any point, thereby recovering in a graceful manner not only from false starts, hesitations, and other speech disfluencies but also from insertions unforeseen by the grammar. SOUP is currently the main parsing engine of the JANUS speech-to-speech translation system [Levin et al. 2000, Woszczyna et al. 1998].

Section 2 briefly describes the grammar representation, section 3 sketches the parsing process, section 4 presents some performance results, section 5 emphasizes the key features of SOUP, and section 6 concludes this paper.

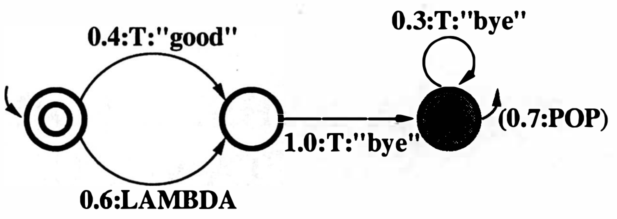


Figure 1: Representation of right-hand side *good +bye as a probabilistic recursive transition network (PRTN). A PRTN has a unique initial node (double circle) and possibly many final nodes (painted gray). A pop arc leaving each final node is implicit.

2 Grammar Representation

The grammar formalism supported by SOUP is purely context-free. Each nonterminal (whose value is simply a label), has a set of alternative rewrite rules, which consist of possibly optional, possibly repeatable terminals and nonterminals. We have found over the years [Mayfield et al. 1995, Woszczyna et al. 1998] that, at least for task-oriented semantic grammars used in speech translation systems, the advantages in parsing speed and ease of grammar construction of such a formalism outweight the lack of the more expressive power offered by richer formalisms (cf., for example, the [Verbmobil Semantic Specification 1994]).

SOUP represents a context-free grammar (CFG) as a set of probabilistic recursive transition networks (PRTNs), where the nodes are marked as initial, regular or final, and the directed arcs are annotated with (*i*) an arc type (namely, specific-terminal (which matches and consumes a particular word), any-terminal (which matches and consumes any out-of-vocabulary word or any word present in a given list), nonterminal (which recursively matches a subnet and, in the parsing process, spawns a subsearch episode) or lambda (the empty transition, which can always occur)), (*ii*) an ID to specify which terminal or nonterminal the arc has to match (if arc type is specific-terminal or nonterminal), and (*iii*) a probability (so that all outgoing arcs from the same node sum to unity). For example, the right-hand side *good +bye (where * indicates optionality and + repeatability and therefore matches *good bye*, *bye*, *bye bye* (and also *good bye bye*, etc)) is represented as the PRTN in Figure 1.

SOUP also directly accepts grammars written in the Java Speech Grammar Format (see section 5.5).

Grammar arc probabilities are initialized to the uniform distribution but can be perturbed by a training corpus of desired (but achievable) parses. Given the direct correspondence between parse trees and grammar arc paths, training the PRTNs is very fast (see section 4).

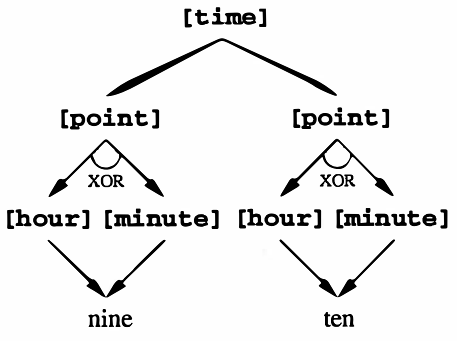
There are two main usages of this stochastic framework of probabilities at the grammar arc level: one is to incorporate the probabilities into the function that scores partial parse lattices, so that more likely ones are preferred; the other is to generate synthetic data, from which, for instance, a language model can be computed.

The PRTNs are constructed dynamically as the grammar file is read; this allows for eventual on-line modifications of the grammar (see section 5.4). Also, strict grammar source file consistency is enforced, e.g., all referenced nonterminals must be defined, warnings for nonterminal redefinitions are issued, and a variety of grammar statistics are provided. Multiple grammar files representing different semantic domains as well as a library of shared rules are supported, as described in [Woszczyna et al. 1998].

The lexicon is also generated as the grammar file is being read, for it is simply a hash table of grammar terminals.

```
s[time]
            ( +[point] )
[point]
            ( [hour] )
            ( [minute] )
[hour]
            ...
            ( nine )
            ( ten )
            ...
[minute]
            ...
            ( nine )
            ( ten )
            ...
```

(a)



(b)

Figure 2: (a) Grammar fragment to illustrate ambiguity packing. The s indicates that nonterminal [time] is a starting symbol of the grammar. (b) Parse lattice for input *nine ten* according to the grammar fragment. It can give rise to four different parse trees, but some will be more likely than others.

3 Sketch of the Parsing Algorithm

Parsing is a particular case of search. In SOUP, parsing proceeds in the following steps:

1. *Construction of the input vector*: Given an utterance to be parsed, it is converted into a vector of terminal IDs. Special terminals <s> and </s> are added at the beginning and end of an utterance, respectively, so that certain rules only match at those positions. Also, user-defined global search-and-replace string pairs are applied, e.g., to expand contractions (as in *I'd like* $\rightarrow$ *I would like*) or to remove punctuation marks. Other settings allow to determine whether out-of-vocabulary words should be removed, or whether the input utterances are case-sensitive.

2. *Population of the chart*: The first search populates the chart (a two-dimensional table indexed by input-word position and nonterminal ID) with parse lattices. (A parse lattice is a compact representation of a set of parse trees (similar to Tomita's shared-packed forest [Tomita 1987]); see Figure 2 for an example).

   This beam search involves top-down, recursive matching of PRTNs against the input vector. All top-level nonterminals starting at all input vector positions are attempted. The advantage of the chart is that it stores, in an efficient way, all subparse lattices found so far, so that subsequent subsearch episodes can reuse existing subparse lattices.

   To increase the efficiency of the top-down search, the set of allowed terminals with which a nonterminal can start is precomputed (i.e., the FIRST set), so that many attempts to match a particular nonterminal at a particular input vector position can be preëmpted by the lack of the corresponding terminal in the FIRST set. This bottom-up filtering technique typically results in a threefold speedup.

   The beam serves to restrict the number of possible subparse lattices under a certain nonterminal and starting at a certain input position, e.g., by only keeping those subparse lattices whose score is

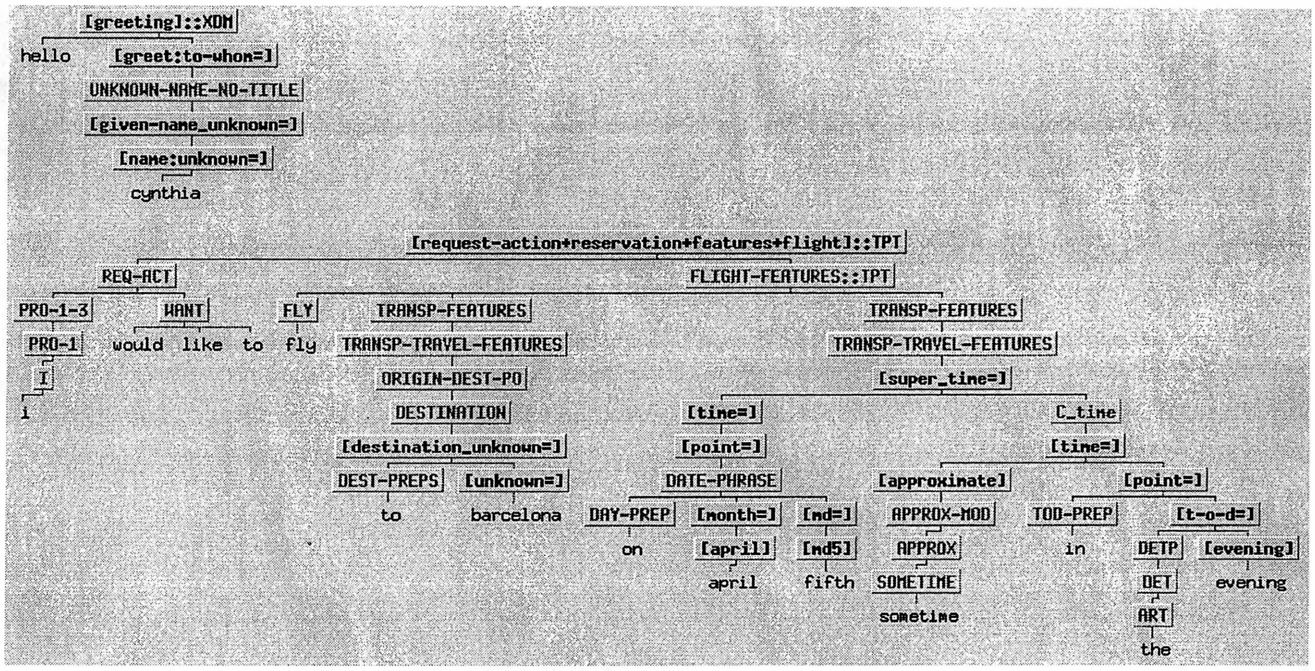


Figure 3: Parse of *Hello Cynthia I'd like to fly to Barcelona on April fifth sometime in the evening.* Uppercased nonterminals such as `PRO-1-3` denote auxiliary nonterminals and are typically removed from the parse trees before backend processing. Note the ability to combine rules from different task domains (`XDM` for cross-domain, `TPT` for transportation) and to parse out-of-vocabulary words (*Cynthia*, *Barcelona*).

at least 30% of the best score. The score function is such that (*i*) coverage (number of words parsed) and (*ii*) sum of arc probabilities are maximized, whereas (*iii*) parse lattice complexity (approximated by number of nonterminals) and (*iv*) usages of the wildcard (approximated by maximal number of any-terminal arcs along the parse lattice) are minimized. Also, pruning of structurally-equal parse lattices is performed, thereby eliminating the redundancy that arises from several right-hand sides matching the same input vector span under the same nonterminal.

3. *Finding the best interpretations*: Once the chart is populated, a second beam search finds the best N interpretations, i.e., the best N sequences of top-level, non-overlapping parse lattices that cover the input vector. Scoring of interpretations adds, to the above scoring function, a fifth factor, namely the minimization of parse fragmentation (number of parse trees per utterance). This search problem can be divided into subproblems (divide and conquer strategy) since both unparsed words and words parsed by a single parse lattice offer a natural boundary to the general problem. A beam search is conducted for each subproblem. In this case, the beam limits the number of active sequences of top-level, non-overlapping parse lattices that form a partial interpretation. Since the single best interpretation is simply the concatenation of the best sequence of each subproblem, even when asked to compute the top N interpretations ($N > 1$), the best interpretation is always computed separately and output immediately so that backend processing can begin without delay. The final result is a ranked list of N interpretations, where the parse lattices have been expanded into parse trees.

Figure 3 shows a sample interpretation in a travel domain.

	Scheduling Grammar	Scheduling + Travel Grammar
Nonterminals	600 (21 top-level)	6,963 (480 top-level)
Terminals	831	9,640
Rules	2,880	25,746
Nodes	9,853	91,264
Arcs	9,866	97,807
Average cardinality of FIRST sets	44.48 terminals	240.31 terminals
Grammar creation time	143 ms	3,731 ms
Training time for 1000 example parse trees	452 ms	765 ms
Memory	2 MB	14 MB
Average parse time	10.09 ms/utt	228.99 ms/utt
Maximal parse time	53 ms	1070 ms
Average coverage	85.52%	88.64%
Average fragmentation	1.53 trees/utt	1.97 trees/utt

Table 1: Grammar measurements and performance results of parsing 606 naturally-occurring scheduling domain utterances (average length of 9.08 words) with scheduling grammar and scheduling plus travel grammar on a 266-MHz Pentium II running Linux.

4 Performance

SOUP has been coded in C++ and Java and compiled for a variety of platforms including Windows (95, 98, NT) and Unix (HP-UX, OSF/1, Solaris, Linux). The upper portion of Table 1 lists some parameters that characterize the complexity of two grammars, one for a scheduling domain and the other for a scheduling plus travel domain; the lower portion lists performance results of parsing a subset of transcriptions from the English Spontaneous Speech Scheduling corpus (briefly described in [Waibel et al. 1996]).

Parsing time increases substantially from a 600-nonterminal, 2,880-rule grammar to a 6,963-nonterminal, 25,746-rule grammar but it is still well under real-time. Also, as depicted in Figure 4, although worst-case complexity for chart parsing is cubic on the number of words, SOUP's parse time appears to increase only linearly. Such behavior, similar to the findings reported in [Slocum 1981], is due, in part, to SOUP's ability to segment the input utterance in parsable chunks (i.e., finding multiple-tree interpretations) during the search process.

Therefore, even though comparisons of parsers using different grammar formalisms are not well-defined, SOUP appears to be faster than other "fast parsers" described in the literature (cf., for example, [Rayner and Carter 1996] or [Kiefer and Krieger 1998]).

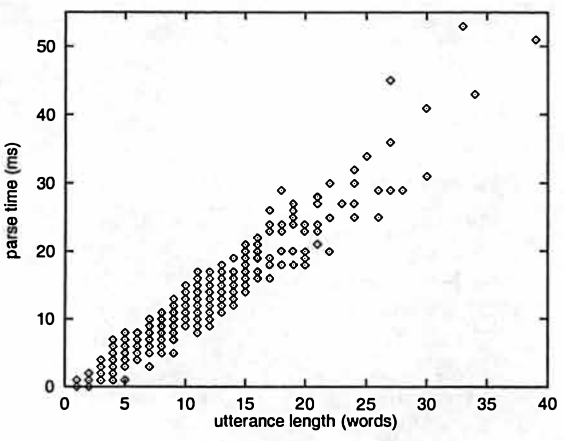
5 Key Features

The following are some of the most interesting features of SOUP.

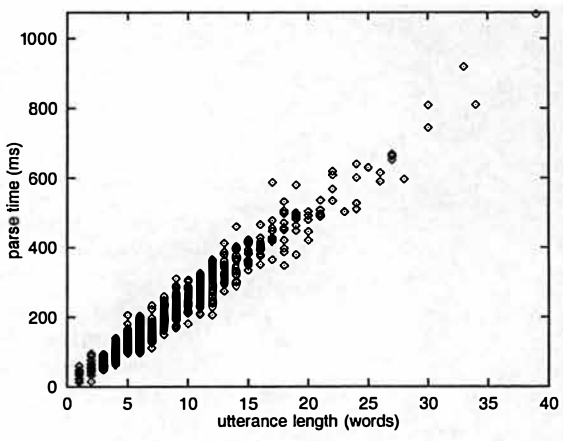
5.1 Skipping

Given the nature of spoken speech it is not realistic to assume that the given grammar is complete (in the sense of covering all possible surface forms). In fact it turns out that a substantial portion of parse errors comes from unexpected insertions, e.g., adverbs that can appear almost anywhere.

SOUP is able to skip words both between top-level nonterminals (inter-concept skipping) and inside any nonterminal (intra-concept skipping). Inter-concept skipping is achieved by the second search step



(a)



(b)

Figure 4: Utterance length vs. parse time for (**a**) scheduling grammar and (**b**) scheduling plus travel grammar. Same test and machine as in Table 1. Parse time appears to increase only linearly with regard to utterance length.

described in section 3 (the search that finds the best interpretation as a sequence of non-overlapping parse lattices), since an interpretation may naturally contain gaps between top-level parse lattices. Intra-concept skipping, on the other hand, occurs during the first search step, by allowing, with a penalty, insertions of input words at any point in the matching of a net. The resulting exponential growth of parse lattices is contained by the beam search. A word-dependent penalty (e.g. one based on word saliency for the task at hand) can be provided but the experiments reported here use a uniform penalty together with a list of non-skippable words (typically containing, for example, the highly informative adverb *not*). The parameter mcs regulates the maximal number of contiguous words that can be skipped within a nonterminal. Figure 5 plots coverage and parse times for different values of mcs. These results are encouraging as they demonstrate that coverage lost by skipping words is offset (up to mcs = 4) by the ability to match longer sequences of words.

5.2 Character-level Parsing

To facilitate the development of grammars for languages with a rich morphology, SOUP allows for nonterminals that operate at the character-level (see Figure 6 for an example). Character-level parsing is achieved using the same functions that parse at the word-level. In fact it is during word-level parsing that character-level parses are spawned by exploding the current word into characters and recursively calling the parse functions. The only difference is that, in a character-level parse, the desired root nonterminal is already known and no skipping or multiple-tree interpretations are allowed.

5.3 Multiple-tree Interpretations

SOUP is designed to support a modular grammar architecture and, as we have seen, performs segmentation of the input utterance into parse trees as part of the parsing process itself. Different interpretations of the same utterance may have different segmentations and the most likely one will be ranked first. Knowledge of the grammar module (which usually corresponds to a task domain) that a nonterminal is from is used in the computation (see [Woszczyna et al. 1998] for details on the statistical model employed).

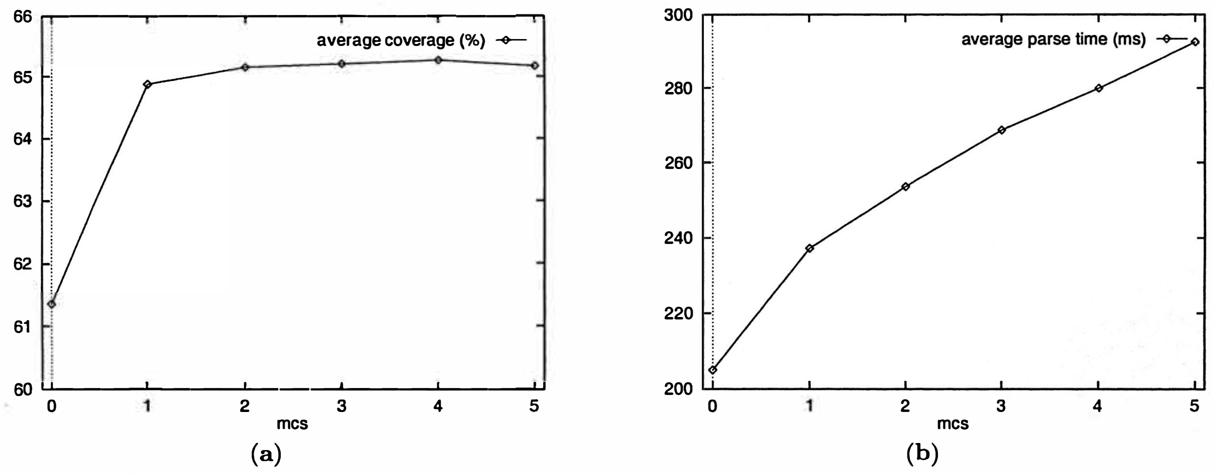


Figure 5: (**a**) Average coverage and (**b**) parse times for different values of mcs (maximal number of contiguous words that can be skipped within a nonterminal). Same test set and machine as in Table 1 but with travel grammar only.

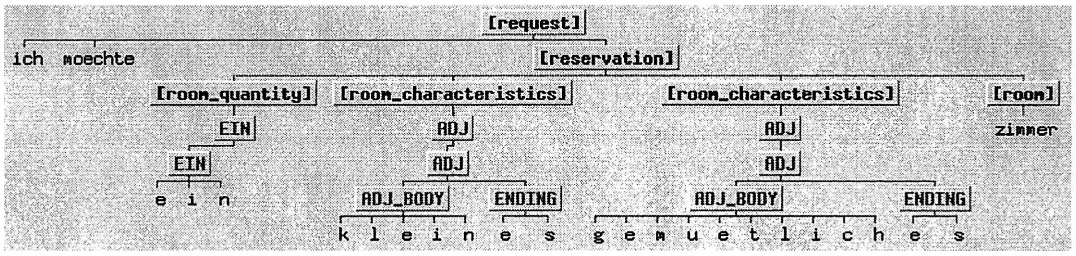


Figure 6: Parse of German *Ich möchte gern ein kleines gemütliches Zimmer* to exemplify character-level nonterminals and skipping. Repeated nonterminals (as in EIN with daughter EIN) indicate the joint between word-level and character-level parse trees. Also note the intra-concept skipping of *gern* (an adverb with practically zero information content).

5.4 Dynamic Modifications

Encoding the grammar as a set of PRTNs gives SOUP the flexibility to activate/deactivate nonterminals and right-hand sides at run-time. For example, grammar nonterminals can be marked as belonging only to a specific speaker side (say, agent vs. client); then, at run-time and for each utterance, nonterminals not belonging to the current speaker are deactivated. Also, in the case of a multi-domain grammar, one could have a topic-detector that deactivates all non-terminals not belonging to the current topic, or at least lowers their probability.

More generally, nonterminals and right-hand sides can be created, modified and destroyed at run-time, which allows for the kind of interactive grammar learning reported in [Gavaldà and Waibel 1998].

5.5 Parsing JSGF Grammars

SOUP has been extended to natively support grammars written according to the specifications of the Java Speech Grammar Format [JSGF 1998]. The JSGF is part of the Java Speech Application Programming Interface [JSAPI 1998] and is likely to become a standard formalism for specifying semantic grammars, at least in the industrial environment.

```
#JSGF V1.0 ISO8859-1 en;
grammar Toy;
public <get> =  <polite>* (get | obtain | request) <obj>+;
    <polite> =  please;
       <obj> =  apple | pear | orange;
```

Figure 7: Toy JSGF grammar used in Figures 8 and 9. In this case * indicates the Kleene star (i.e., optionality and repeatability), + repeatability, and | separates alternatives.

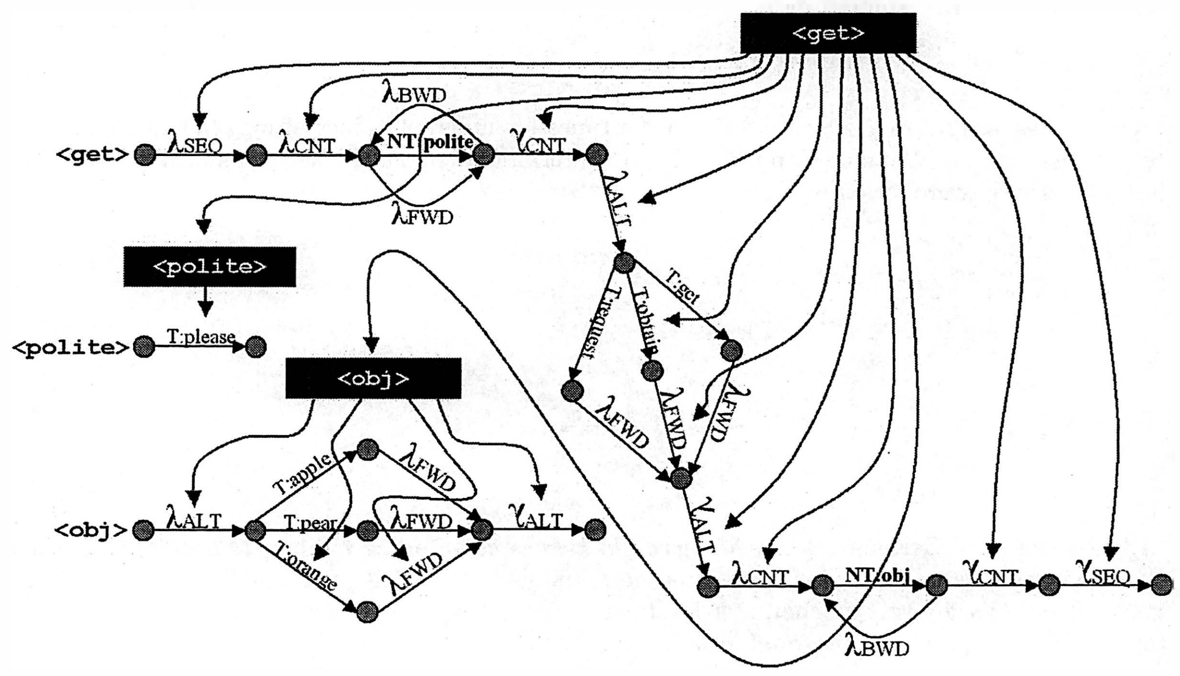


Figure 8: PRTNs for the grammar in Figure 7 and schematic parse of *Please obtain orange*. Upside-down lambdas mark end of JSGF Rule scope.

SOUP is able to represent a RuleGrammar as defined by JSGF with the same underlying PRTNs. This is accomplished by the usage of lambda arcs to encode the JSGF Rule source, so that, out of a parse tree, the corresponding RuleParse (the result of a parse as specified by the JSAPI), can be constructed. In more detail, for each JSGF RuleSequence, RuleAlternatives, RuleCount and RuleTag, a corresponding lambda-SEQ, lambda-ALT, lambda-CNT or lambda-TAG arc is built in the PRTNs, as well as a closing lambda arc (pictured upside-down) to indicate the end of scope of the current JSGF Rule.

Figure 7 shows a toy grammar in the JSGF formalism. Figure 8 depicts the corresponding PRTNs as well as a schematic sample parse tree. Figure 9 shows the resulting RuleParse object constructed from such parse.

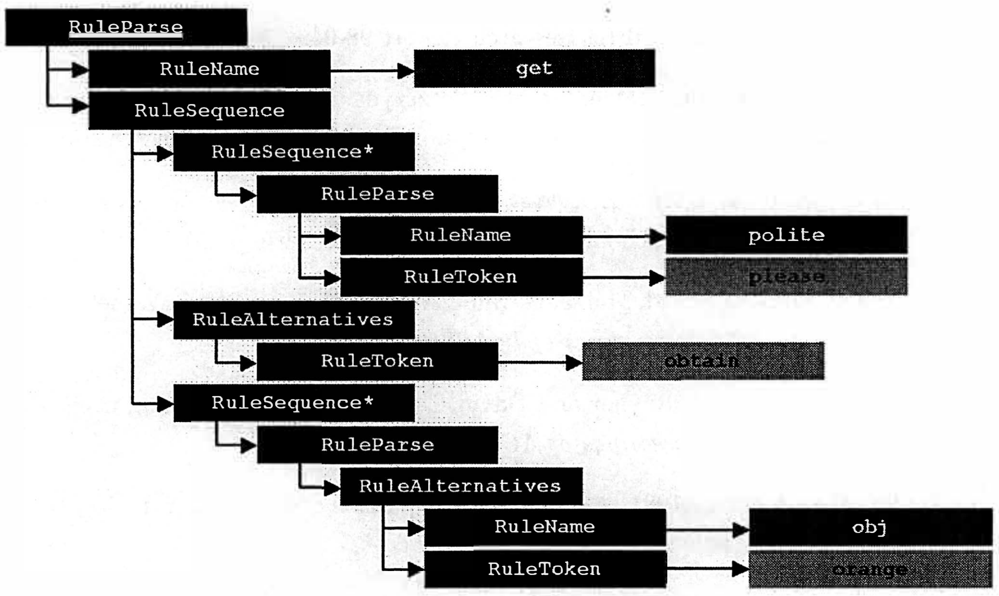


Figure 9: Resulting JSGF `RuleParse` constructed from the parse in Figure 8. Note that `RuleCounts` become `RuleSequences` as specified by the JSAPI (marked with *).

6 Conclusion

We have presented SOUP, a parser designed to analyze spoken language under real-world conditions, in which analysis grammars are very large, input utterances contain disfluencies and never entirely match the expectations of the grammar, and yet backend processing must begin with minimal delay. Given its robustness to ill-formed utterances, general efficiency and support for emerging industry standards such as the JSAPI, SOUP has the potential to become widely used.

Acknowledgements

Laura Mayfield Tomokiyo wrote the Scheduling grammar, Donna Gates and Dorcas Wallace wrote the Travel grammar; Detlef Koll programmed the methods that read in a JSGF `RuleGrammar`. Klaus Zechner, Matthias Denecke and three anonymous reviewers gave helpful comments to earlier versions of this paper.

References

[Gavaldà and Waibel 1998] Marsal Gavaldà and Alex Waibel. 1998. Growing Semantic Grammars. In *Proceedings of COLING/ACL-1998*.

[JSAPI 1998] Java^{TM} Speech API, version 1.0. 1998. `http://java.sun.com/products/java-media/speech/`

[JSGF 1998] Java^{TM} Speech Grammar Format, version 1.0. 1998. `http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/`

[Kiefer and Krieger 1998] Bernd Kiefer and Hans-Ulrich Krieger. 1998. A Bag of Useful Techniques for Efficient and Robust Parsing. DFKI Research Report 98-04.

[Levin et al. 2000] Lori Levin, Alon Lavie, Monika Woszczyna, Donna Gates, Marsal Gavaldà, Detlef Koll and Alex Waibel. 2000. The JANUS-III Translation System. To appear in *Machine Translation.*

[Lavie 1996] Alon Lavie. 1996. *GLR*: A Robust Grammar-Focused Parser for Spontaneously Spoken Language.* Doctoral dissertation. School of Computer Science, Carnegie Mellon University.

[Mayfield et al. 1995] Laura Mayfield, Marsal Gavaldà, Wayne Ward and Alex Waibel. 1995. Concept-Based Speech Translation. In *Proceedings of ICASSP-1995.*

[Rayner and Carter 1996] Manny Rayner and David Carter. 1996. Fast Parsing using Pruning and Grammar Specialization. In *Proceedings of ACL-1996.*

[Slocum 1981] Jonathan Slocum. 1981. A Practical Comparison of Parsing Strategies. In *Proceedings of ACL-1981.*

[Tomita 1987] Masaru Tomita. 1987. An Efficient Augmented-Context-Free Parsing Algorithm. In *Computational Linguistics*, Volume 13, Number 1-2, pages 31–46.

[Verbmobil Semantic Specification 1994] Universität des Saarlandes. 1994. The Verbmobil Semantic Specification. Verbmobil Report 1994-6.

[Waibel et al. 1996] Alex Waibel, Michael Finke, Donna Gates, Marsal Gavaldà, Thomas Kemp, Alon Lavie, Lori Levin, Martin Maier, Laura Mayfield, Arthur McNair, Ivica Rogina, Kaori Shima, Tilo Sloboda, Monika Woszczyna, Torsten Zeppenfeld and Puming Zhan. 1996. JANUS-II: Translation of Spontaneous Conversational Speech. In *Proceedings of ICASSP-1996.*

[Ward 1990] Wayne Ward. 1990. The CMU Air Travel Information Service: Understanding spontaneous speech. In *Proceedings of the DARPA Speech and Language Workshop.*

[Woszczyna et al. 1998] Monika Woszczyna, Matthew Broadhead, Donna Gates, Marsal Gavaldà, Alon Lavie, Lori Levin and Alex Waibel. 1998. A Modular Approach to Spoken Language Translation for Large Domain. In *Proceedings of AMTA-1998.*