Eurospeech, vol. 5, pp. 2055-2058, ESCA, Budapest, Hungary, 1999

A ROBUST PARSER FOR SPOKEN LANGUAGE UNDERSTANDING

Ye-Yi Wang

Microsoft Research
Speech Technology Group
One Microsoft Way
Redmond, Washington 98052, USA
http://research.microsoft.com/srg

ABSTRACT

This paper describes a robust parsing algorithm for spoken language understanding. Comparing with the other work in robust parsing, we focus on building a parser that is robust to not only ill-formed spontaneous spoken language inputs but also under-specified grammars. Preliminary experiment results show that the parsing performance deteriorates more gracefully than another parser we have used when the grammar is more under-specified.

1. INTRODUCTION

Robust parsing has become an important topic in spoken language understanding ever since the ATIS project. This is because spontaneoue spoken languges are hardly well-formed. Ungrammatical sentences, disfluencies (e.g., repeated words, repairs or false starts) are so pervasive that the traditional parser and NLP grammars will not work well. SLU researchers have been working on robust parsing algorithms in the hope that their performance will deteriorate gracefully when the inputs are ill-formed [4][5][6].

While most of the robust parsing research work has focused on the ill-formed inputs, the robustness of a parser to under-specified grammars has not been well studied. We believe that this is an important issue if we want to allow a non-computational linguist to develop an application with a natural language interface --- in this case, ideally, we can let the developer write high level conceptual relations rather than detailed grammars.

1.1 LEAP Grammar

LEAP (Language Enabled APplications) [1][3] is an effort in the speech technology group in Microsoft Research that aims at spoken language understanding.

LEAP differs from many other NLP systems with its application-centric architecture. In this architecture, a *LEAP entity* is an element of the *real world* that an application has to deal with and wishes to expose to the user via natural language. A Leap entity can be referred to with a *semantic class* --- a semantic class is a way to specify a LEAP entity, and the LEAP entity is called the *type* of that semantic class. Since a LEAP entity can be referred to in many different ways, different semantic classes may have the same type. For example, we can refer to a person with his name (Peter) or his relation to another person (John's advisor); therefore both semantic classes `ByName` and `ByRel` can share the same type, `<PERSON>`.

A LEAP grammar contains the definitions of semantic classes that can be used to refer to LEAP entities. A semantic class is defined as a set of slots that need to be filled with terminal (verbatim) words or with recursive semantic class objects.

The semantic class grammar appears similar to the semantic grammar in CMU Phoenix System [6]. However, there is a philosophical difference between the two grammars. Strictly speaking, our semantic class grammar can hardly be called a grammar, since it is primarily used to define the conceptual relations among LEAP entities rather than the language expressions that are used to refer to the entities. Because of this, a LEAP grammar tends to be under-specified. This can be illustrated with the following example:

```
<PERSON> ByRel {
    <P_RELATION>
    <PERSON> }
<PERSON> ByName {
    [LastName]
    [FirstName] }
<STRING> FirstName verbatim {
    john | john's | peter | … }
<STRING> LastName verbatim {
    smith | smith's | shaw … }
<P_RELATION> PersonalRel verbatim {
    boss | father | mother | son |  … }
```

Here `ByRel` is a semantic class that has the type `<PERSON>`. The LEAP grammar specifies that it has two slots --- one has to be filled with an object of a semantic class having the type `<P_RELATION>`, and the other has to be filled with an object of a semantic class having the type `<PERSON>`. Here little linguistic information is available for the assembly of an expression that can be used to refer to an object of the class `ByRel`. According to this grammar, any sequence that contains a word of `<P_RELATION>` typed class and a word of a `<PERSON>` typed class can be an expression referring to a semantic object of `ByRel`, such as "*John's father*," "*father of John*," or even "*John loves his father*." The slots of the class `ByName` are specified more restrictively with particular semantic classes (`LastName` and `FirstName` in brackets) rather than the types of semantic classes.

Another difference exists between the grammar in Phoenix [6] and our semantic class grammar: the former only allows word skipping between slots (modeled in details with RTNs) in the top frame level, while the latter appeals for robust parsing (word and grammar symbol skipping) in all levels of the semantic class hierarchy.

1.2 Under-Specified vs. Under-Developed

It is important to understand the difference between under-specified grammar and under-developed grammar. Under-specified grammar is the one that has the conceptual relations

among the relevant entities in an application fully defined but lacks the linguistic information (e.g. word order information) regarding how the entities can be glued together to form a legal expression. Under-developed grammar is the one that does not have all these relevant conceptual relations specified. There is not much we can do with the parser for performance improvement when the grammar is under-developed.

1.3 Challenges

Clearly the LEAP grammar is under-specified. While the under-specification makes the grammar versatile enough to cover different expressions with a single semantic class, it poses the following challenges to the parser:

- An under-specified grammar will greatly increase the parse ambiguity --- an input word sequence may match many different semantic classes. How can the parser effectively resolve the ambiguity?
- The ambiguity, together with the requirement of robust parsing in all levels of a semantic hierarchy, results in a much bigger search space. How can the parser effectively prune partial parses to increase its speed?

2. ROBUST CHART PARSING ALGORITHM

Our robust parsing algorithm is an extension of the bottom-up chart parsing algorithm in [2]. To use the algorithm, we first have to convert the LEAP semantic grammar into a CFG rule set. This can be accomplished by introducing a CFG rule A→B for a semantic class A and every slot rule B in A. With the previous example, we introduce the following two rules to the CFG grammar for the semantic class `ByRel`:

```
ByRel →  <P_RELATION>
ByRel →  <PERSON>.
```

Since `<P_RELATION>` and `<PERSON>` are not mutually exclusive in `ByRel`, we need the following rule to glue them together in a semantic class:

```
ByRel →  ByRel ByRel
```

The algorithm uses dotted rules by adding a dot in the right hand side of CFG rules. If the dot appears at the end of a rule like in A→α•, we call it a (partial) parse with symbol A. If the dot appears in the middle of a rule like in A→B•CD, we call it a hypothesis that is expecting a partial parse with a symbol compatible with C: if C is a semantic class or a verbatim, then the symbol of the partial parses must be C. If C is a type, then the symbol of the partial parse has to be a class that has C as its type.

2.1 The Algorithm

The algorithm maintains three major data structures --- A chart (*chart*) holds hypotheses that are expecting partial parses to finish the application of the CFG rules associated with them; an agenda (*agenda*) holds the partial parses that are yet to be used to expand the hypotheses in the chart; and a list (*processed*) holds the parses that have already been used to expand the hypotheses (therefore the parses are sub-trees of some new hypotheses) in the chart. The algorithm requires that the LEAP grammar have been converted to a CFG rule set (*ruleset*).

The general framework of the algorithm is similar to the chart parsing algorithm described in [2]. The differences include

- The requirement that a hypothesis *h* and a partial parse *p* have to cover adjacent words in the input is relaxed here. Instead it is only required that `h.end < p.start` [Line 18]. This effectively skips the words between `h.end` and `p.start`, and makes the parser able to omit noise words in input sentences;
- The combination of a hypothesis with a new partial parse taken from *agenda* results in multiple new hypotheses [iteration in line 21]. Those hypotheses differ from one another in terms of the position of the dot in the dotted rule [line 26]. In other words, those different hypotheses are expecting different partial parses. This effectively skips the symbols in a rule, so the parser can continue its operation even if something expected by the grammar is omitted by the speaker or by the speech recognizer.

```
ALGORITHM: ROBUST_CHART_PARSER
1. agenda = ϕ
2. chart = ϕ
3. foreach (rule r in ruleset)
4.   for i = 0 to r.size
5.     create new hypo h
6.     h.start ← h.end ← 0
7.     h.rule ← r
8.     h.dot_position ← i
9.     chart ← h
10.foreach (w in sentence)
11.  agenda ← w;
12.  foreach (parse p in agenda) {
13.    remove p from agenda
14.    processed ← p
15.    if (p.acceptable()) {
16.      candidates ← p;
17.      foreach (hypo h in chart)
18.        if h.end < p.start and
               h.symbol_after_dot and
               p.symbol are compatible
19.          append(h, p)
20.sort and output all parses in candidates
```

The algorithm calls the procedure `append(h, p)`, which is defined below:

```
PROCEDURE: APPEND(h, p)
21.for i=h.dot_position + 1 to h.rule.size
22.  create new hypo h' with children h, p
23.  h'.start ← h.start
24.  h'.end ← p.end
25.  add h' to the parents list of h and p
26.  h'.dot_position = i
27.  if i ≠ h.rule_size
28.    chart ← h'
29.  else
30.    place(h')
```

The placement of a new partial parse is crucially related to the pruning/disambiguation mechanism of the parser. The algorithm is described in the **PLACE**`(h)` procedure below:

```
PROCEDURE: PLACE(h)
31.if ∃ g (g.start=h.start and g.end=h.end
           and g.dot_position=g.rule.size
           and g.symbol=h.symbol)
32.  case g.score > h.score
33.    discard h
34.  case g.score < h.score
35.    case g ∈ processed
36.      replace occurence of g with h in
         all parses and hypotheses, change
         their scores and propagate the
```

```
        change to all the antecedents
37.     case g ∈ agenda
38.       replace g with h
39.   case g.score = h.score
40.     create an ambigious parse f with
        children h and g.
41.     f.score ← h.score
42.     f.start ← h.start, f.end ← h.end
43.     replace g in agenda/processed with f
44.else
45.   agenda ← h
```

If there is already a parse *g* that has the same symbol and span as the new parse *h*, we then compare their scores. If the existing parse has a higher score, we just discard the inferior new parse [line 32-33]. If the new parse has a higher score, we then replace every occurrence of the existing parse with the new one [line 34-38]. When we do this, we have to check if the existing parse is in *processed* list. If so, we have to accordingly update the score of all hypotheses and parses that used to have the existing parse *g* as their descendents [line 35-36]. For that purpose, a list of parents is maintained for every hypothesis or partial parse [line 25].

2.2 Partial Parse Score

The ***PLACE***(*h*) procedure in the previous subsection depends on the score of partial parses. While we believe that parse scoring should ultimately resort to the likelihood of the parse with respect to a statistical grammar, the amount of data currently available does not license this solution. Instead heuristic scores are used. A parse score here is not a single number. Instead, it is a set of property values of the parse. The set includes, in the order from the most significant to the least significant, the property values in Table 1.

Coverage	+	Number of word covered by a parse
SkippedSym	-	Number of rule symbols skipped in the parse tree
TotalNodes	-	Number of nodes in the parse tree
Depth	-	The depth of the parse tree
Start	-	The leftmost position of the word covered by the parse

Table 1. Properties in parse scores. The signs in the second column indicate the preference on the value of a property: "+" indicates that a larger number is preferred, and "-" indicates the opposite. Combinations of the different properties can result in different parse scores. ***PLACE***(*h*) uses the score of the properties in the shaded background.

The score of a parse can be dynamically computed according to its children's score when the parse is created.

2.3 Pruning and Disambiguation

Pruning is carried out when new parses are created (see the ***PLACE***(*h*) procedure in the algorithm.) The score used in this operation includes the properties that are in the shaded background in Table 1. It is straightforward for the first two properties. They are used to minimize the number of times that input words or grammar symbols are skipped. The preference on fewer parse tree nodes is less obvious. It can be illustrated with the following example: assuming that we have an extra semantic class PC (stands for "Person Container"):

```
<PEOPLE> PC {
        <PERSON>
        <PERSON> <PEOPLE> }
```

Then the phrase "john smith" is ambiguous according to the grammar. It has two parses shown in Figure 1.

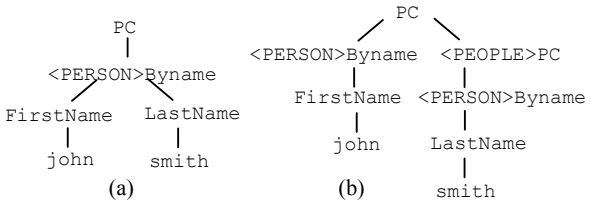

Figure 1. Ambiguous Parse

The "fewer nodes" preference correctly selects (a) from these two parses. Actually the preference is a direct application of the psycholinguistic minimum attachment principle.

A more radical final disambiguation process is carried out in step 20 when we sort the acceptable parses in the candidate list. It uses the score that contains all property values in Table 1.

2.4 Left Recursion

The inclusion of the rule A → A A for a semantic class A in *ruleset* results in unlicensed ambiguity. For example, when the sequence A A A is observed in an input, there are two different parses as shown in Figure 2.

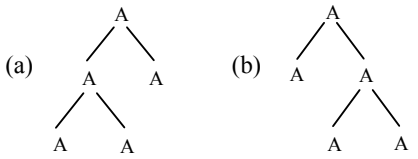

Figure 2. Left recursion (a) vs. right recursion (b)

These two parses are not real ambiguity since the rule A → A A is introduced to concatenate different slots of a semantic class. To avoid this, we only allow left recursion when A → A A is applied to append a hypothesis with a partial parse, so we effectively prohibit structures like (b) in Figure 2.

2.5 Agenda

Unlike a traditional chart parser, the sequential order in which the partial parses are taken out from the agenda is crucial here. This can be illustrated with the example in Figure 3.

Here the number in the bracket following a non-terminal is the number of words in the sentence covered by that non-terminal. Suppose that (a) was taken from the agenda first and combined with the hypotheses in the chart, and then (b) was taken from the agenda. Since (b) has the same symbol and span as (a) but a better score (coverage), it replaces every occurrence of (a) in all parses and hypotheses. Now if (c) is taken from the agenda, it can replace the B[2] sub-tree in (a) and result in a tree A[6], which also spans [*k, l*] and has a better coverage than (b). However, since (a) was already discarded because of its previous inferiority to (b), there will be no chance for (a) to revive and take the place of the inferior A[5] in (b). Therefore it

is crucial to first take out the parse with smaller span (in this example, (c)) from the agenda.

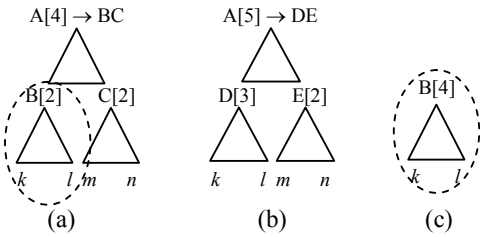

Figure 3. Partial parses from agenda. (a) and (b) have the same symbol (A) and span ([*l, n*]) but different coverage.

In our implementation, `agenda` is a priority queue. A partial parse that has the minimum span and highest score, covers the word closest to the sentence start position (in that order) has the highest priority.

3. PERFORMANCE

The parser was incorporated into the LEAP server and used in *Dr. Who*, a research project that aims the advancement of component research in spoken language technologies. We have collected 9 topics of data in the domain of personal information management, and we have fully developed LEAP grammars for 3 of those topics. The grammars for the rest of the topics are still under-developed.

We conducted our first experiment to evaluate the ambiguity resolution performance of the parser. We selected 121 test sentences that have the one of the 3 topics with the fully developed grammars. We parsed those sentences, collected the number of ambiguities that occurred in the process of parsing. We then manually checked the best parses predicted by the parser, and analyzed how many times the parser had made ambiguity resolution errors that result in incorrect parses. The result is shown in Table 2.

Sentence #	CoParse	Ambi	IncResolution
121	110	1931	6

Table 2. Parser ambiguity resolution performance. 1st column: number of sentences. 2nd column: number of correct parses. 3rd column: total number of ambiguities occurred in the parsing process. 4th column: number of parsing errors due to incorrect ambiguity resolution.

In the second experiment we evaluate the topic identification performance of the parser. We used this metric because (a) it allowed automatic performance evaluation without manually examining the parses; and (b) topic identification performance was extremely important in our application. The evaluation was conducted with a test set of 3000 sentences that covers all 9 topics, and the results are given in Table 3.

Correct TI	Incorrect TI	Accuracy
2220	780	74%

Table 3. Topic identification performance.

In the next experiment we compared the performance of the parser with another top-down heuristic search parser [3] in terms of their robustness to under-specified grammars. This was carried out by replacing our current grammar (Grammar 2) with a more under-specified one that we had in the previous development stage. The topic identification performance of both parsers deteriorated because of this. However, the parser described in this paper had an 11% error rate increase while the other one had a 15% error rate increase.

4. CONCLUSIONS AND FUTURE WORK

We have presented a robust parsing algorithm for spoken language understanding. The algorithm is robust to both ill-formed spoken language inputs and under-specified semantic grammars.

Currently, heuristic scores are used for disambiguation. In the future we will investigate parsing using statistical methods, and parsing systematic framework. We will also investigate a unified approach to CSR and SLU. As a first step towards a unified model, we are applying the robust parsing algorithm to the word graph generated by the Whisper speech recognizer. We also envision a unified model that can use a stochastic semantic grammar directly as the language model for speech recognition.

5. ACKNOWLEDGEMENT

The author would like to thank Kuansan Wang, Joshua Goodman, Peter Mau, Xuedong Huang, Antonio Bagazzi, Bruno Alabiso, Alex Acero and the whole *Dr. Who* team in Microsoft Research.

REFERENCES

[1] Alabiso B and A. Kronfeld, "LEAP: Language Enabled Applications". *Proceedings of the First Workshop on Human-machine Conversation*. Bellagio, Italy, July 1997.

[2] Allen J. "Natural Language Understanding". The Benjamin-Cummings Publishing Company, Inc. 1995.

[3] Bigazzi A. "LEAP Notes". Microsoft Research Internal Document, 1999.

[4] Lavie A. "GLR*: A Robust Parser for Spontaneous Spoken Language". *Proceedings of ESSLLI-96 work-shop on Robust Parsing*, Prague, Czech Republic, August 1996

[5] Miller S., R. Bobrow, R. Ingria and R. Schwartz. "Hidden Understanding Models of natural Language". *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, New Mexico State University, 1994.

[6] Ward W. "The CMU Air Travel Information Service: Understanding Spontaneous Speech". *Proceedings of the DARPA Speech and Natural Language Workshop*, 1990.