Partly Specified Priority Patterns in Natural Language Parsing within Dialogue Systems

Gergely Covásznai[1,2], Constantine Kotropoulos[1] and Ioannis Pitas[1]

[1]*Artificial Intelligence & Information Analysis Laboratory,*

Department of Informatics

Aristotle University of Thessaloniki,

Box 451, Thessaloniki 541 24, Greece.

E-mail: costas@zeus.csd.auth.gr , pitas@zeus.csd.auth.gr

Tel: +30-2310-996361, Fax: 30-2310-998453

[2] *On leave from the Institute of Mathematics and Informatics, Univ. of Debrecen, Debrecen Hungary*

E-mail : kovasz@math.klte.hu

Abstract

In this paper, two essential problems that belong to language parsing and have arisen from dialogue management are discussed and solved by implementing a variant of the well-known context-free parsing algorithm [1]. The first problem is the use of partly specified patterns, i.e., the use of wildcards in the right-hand-sides of the rewriting rules of a context-free grammar. The second one is the use of priority patterns, i.e., the assignment of priority values to rewriting rules. These problems are not handled in the majority of the state-of-the-art dialogue systems. The proposed algorithm has been implemented in a dialogue system core application called CAML Core [7], that is used to implement dialogue systems in several domains like conversational and multimodal interfaces for help desk applications, and chat bots.

Keywords

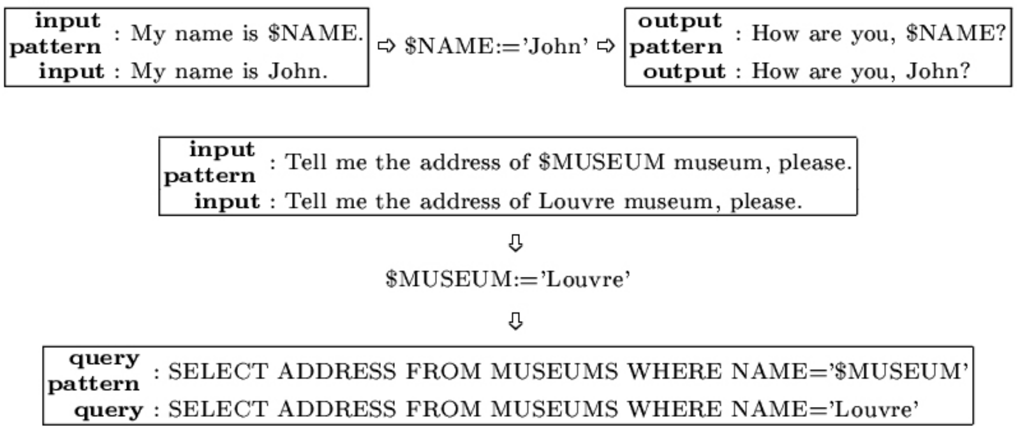
Dialogue Systems, Conversational Agents, Context-Free-Parsing, Early Algorithm.

1. Introduction

Nowadays, human-machine interaction undergoes through spoken dialogues. Many dialogue systems have been developed and used in several domains, e.g., help desk applications, airplane/train ticket reservation systems, chat bots, etc. Each dialogue system includes a module that performs natural language parsing and understanding, i.e., the extraction of semantic information (let us say meaning) from an input spoken by a user. Several techniques exist for natural language parsing. The most frequent technique is a context-free grammar. A problem inherent in a context-free parser is the use of only fully specified patterns, i.e., rewriting rules that have fully specified right-hand-sides. It would be a very useful facility to support *wildcards* within the aforementioned right-hand-sides. A wildcard can get a text value based on the current user input, i.e., a rewriting rule having a wildcard in its left-hand-side is not fixed a priori. Let us see why it is important to use wildcards in natural language parsing:

- Extracting unpredictable information and using it in the dialogue later. There are unpredictable segments in a dialogue, i.e., it is impossible to prepare a context-free grammar including all necessary rewriting rules. For example, the extraction of user's name at the beginning of the dialogue, as can be seen in Figure 1.

- Checking the validity of text out of the context-free parser: The validity of a text assigned to a wildcard can be checked by the dialogue system, which can perform this check even by the contribution of external resources, e.g., the dialogue system can check whether the given text can be found in an external database, as shown in Figure 1. This facility is very useful in order to create easily configurable dialogue systems since not all the data should be incorporated in the dialogue system a priori.

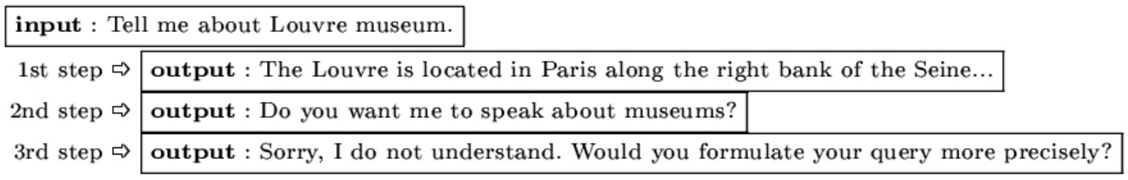


input pattern : My name is $NAME.
input : My name is John.
⇨ $NAME:='John' ⇨
output pattern : How are you, $NAME?
output : How are you, John?

input pattern : Tell me the address of $MUSEUM museum, please.
input : Tell me the address of Louvre museum, please.
⇩
$MUSEUM:='Louvre'
⇩
query pattern : SELECT ADDRESS FROM MUSEUMS WHERE NAME='$MUSEUM'
query : SELECT ADDRESS FROM MUSEUMS WHERE NAME='Louvre'

Fig. 1. The use of wildcards in patterns.

The majority of the state-of-the-art dialogue systems employ natural language parsers that allow only the use of fully specified patterns, like the Phoenix Semantic Parser [8] or the Dialogue Management Tool [9]. Another facility that is quite important and usually not supported in natural language parsing is the use of *priority patterns*. The need of assigning priority numbers to patterns has arisen from dialogue management, where a user input is processed usually in several steps, from the step of the highest priority toward the step of the lowest priority. An example with three steps is shown in Figure 2. The processing of a user input halts at the first step if the input has been recognized completely. It would have halted at the second step if it has not halted at the first step and some a priori keywords have been recognized in the input (e.g., ``museum''). Furthermore, it gets to the last step if it has not halted before.



input : Tell me about Louvre museum.
1st step ⇨ **output** : The Louvre is located in Paris along the right bank of the Seine...
2nd step ⇨ **output** : Do you want me to speak about museums?
3rd step ⇨ **output** : Sorry, I do not understand. Would you formulate your query more precisely?

Fig. 2. The priority in dialogue management steps and related patterns.

In this paper, we are introducing a variant of Jay Earley's context-free parsing algorithm [1] that can handle wildcards and priority rewriting rules. The improvement we have made to this algorithm may serve as an example for improving other context-free parsing algorithms.

2. Context-free Parsing and the Basic Algorithm

2.1. Extending context-free grammars with wildcards

A *context-free grammar* is a quadruple $\langle V_T, V_N, S, \Phi \rangle$ where $V_T \cap V_N = \varnothing$, $S \in V_N$, and $\Phi \subseteq V_N \times (V_T \cup V_N)^*$. A^* denotes $\bigcup_{i=0}^{\infty} A^i$ for any set A. The elements of V_T are called terminal symbols and denoted by a,b,c in this paper. We call the elements of V_N non-terminal symbols, and denote them by $A,B,C.$ We refer to strings that are elements of $(V_T \cup V_N)^*$ as $\alpha,\beta,\gamma.$ The empty string is denoted by $\lambda.$ Any element of Φ is called a rewriting rule, and is written in the form $(A \to \alpha).$ The language generated by a context-free grammar G is denoted by $L(G).$ The aim of parsing is to show whether an input string of the form $w_{1,} w_2,\dots, w_n \in L(G)$, where $w_1 \in V_{T,}$ $i = 1,2,\dots,n$ and $n \geq 0$, and to construct a parse-free grammar if the answer is "yes".

In order to use wildcards in context-free parsing, we have to extend the aforementioned concepts. A grammar can be written as a quintuple $\langle V_T, V_N, S, V_W, \Phi \rangle \cdot$ $V_W \subset V_N$ and its elements are called *wildcards*. During the parsing for an input string, Φ may be extended with new rewriting rules having a wildcard in their left-hand-sides. The right-hand-sides of these rewriting rules depend on the current input string.

2.2. The Sequential Nature of Parse-trees

Since natural languages are based inherently on ambiguous grammars, it usually happens that more than one outputs (i.e., parse-trees) can be constructed for one input string. This is especially true when wildcards are used, since it may happen several possible texts to be assigned to a given wildcard. If priorities are assigned to the rewriting rules in a dialogue system, the parser must emit the parse-trees in the order of their priorities that can be computed on the basis of the priorities of the rewriting rules. This is why we need a parsing algorithm which can deal with a sequence of outputs.

2.3. The Basic Algorithm

The basic algorithm is used to parse in a context-free grammar that does not include wildcards. It emits the parse-trees one after the other. This is why it possesses the ability to emit the parse-trees in a pre-defined order. This algorithm will be improved in Section 3 in order to handle wildcards and priority. In contrast with the algorithm published in Jay Earley's [1], the proposed algorithm is not only a recognizer, but even a parser, i.e., it emits parse-trees instead of acceptance or rejection. Furthermore, our algorithm saves space by erasing those states which are not necessary in the future.

The states are gathered in the ordered set $Parse.$ An ordered set K can be represented in the form $\langle p_{1,} p_{2,} \cdots p_k \rangle, k \geq 0,$ where p_1 is its first element, p_2 is its second element and so on. The number of element of K is written in the form $K.size.$ The i th element of K $(1 \leq i \leq K.size)$ can be denoted by $K[i]$. The operation of appending a new element p to K is denoted by $K.push(p).$ Furthermore, erasing the i th element is denoted by $K.erase(i).$ A state is written in the form $[i, j, (A \to \alpha \bullet \beta), \pi],$ where $0 \leq i \leq n, 0 \leq j \leq n, \bullet \notin V_N \cap V_{T,}$ and π is an ordered set of "pointers" to other states, i.e., their location within $Parse.$ π contains as many pointers as the number of the non-terminal symbols in $\alpha.$

In order to erase the unnecessary states, we shall use the indicator *deletable*, which can get a boolean value *TRUE* or *FALSE.* As can be seen in the algorithm given in Figure 3, the states that are erased are not used for the operation completion. The tasks corresponding to the three basic operations of Earley's algorithm, namely the scanning, the completion, and the prediction, are indicated. Since arbitrary context-free grammars can be used (i.e., they may contain a rewriting rule $(A \to \lambda)$) the completion cannot be applied in a straightforward way as mentioned in [1]. Accordingly the completion is performed in two symmetrical steps.

```
1. $\boxed{\text{initialization}}$
   (a) $Parse := \langle\rangle$, $pointer := 1$;
   (b) for all $(S \rightarrow \alpha) \in \Phi\rangle$:
           $Parse.push([0, 0, (S \rightarrow \bullet\alpha), \langle\rangle])$.
2. If $pointer > Parse.size$ then halt with rejection.
3. If $Parse[pointer] = [0, n, (S \rightarrow \alpha\bullet), \pi]$ and
   CHECK_PARSETREE($Parse, pointer$) then halt with acceptance.
4. $deletable := TRUE$.
5. $\boxed{\text{scanning}}$
   If $Parse[pointer] = [i, j, (A \rightarrow \alpha \bullet a\beta), \pi]$ and $j < n$ and $a = w_{j+1}$ then
       $Parse.push([i, j+1, (A \rightarrow \alpha a \bullet \beta), \pi])$.
6. If $Parse[pointer] = [i, k, (A \rightarrow \alpha \bullet B\beta), \langle p_1, \ldots, p_s\rangle]$ then
   (a) $\boxed{\text{completion}}$
       for all $Parse[p] = [k, j, (B \rightarrow \gamma\bullet), \pi]$, $0 < p < pointer$:
           $Parse.push([i, j, (A \rightarrow \alpha B \bullet \beta), \langle p_1, \ldots, p_s, p\rangle])$;
   (b) $\boxed{\text{prediction}}$
       for all $(B \rightarrow \gamma) \in \Phi$ :
           $Parse.push([k, k, (B \rightarrow \bullet\gamma), \langle\rangle])$;
   and $deletable := FALSE$.
7. $\boxed{\text{completion}}$
   If $Parse[pointer] = [k, j, (B \rightarrow \gamma\bullet), \pi]$ then
       for all $Parse[p] = [i, k, (A \rightarrow \alpha \bullet B\beta), \langle p_1, \ldots, p_s\rangle]$, $0 < p < pointer$:
           $Parse.push([i, j, (A \rightarrow \alpha B \bullet \beta), \langle p_1, \ldots, p_s, pointer\rangle])$
   and $deletable := FALSE$.
8. If $deletable$ then $Parse.erase(pointer)$. Otherwise, $pointer := pointer + 1$.
9. Go to 2.
```

Fig. 3. The basic algorithm.

The basic algorithm provides a check whether a constructed parse-tree is valid. If it is not, the parsing continues until either another parse-tree is constructed, or the algorithm halts with rejection. The validity condition is called *CHECK_PARSETREE*, and it can be customized on-demand.

3. Using Wildcards and Priorities in the Improved Algorithm

The main aim of this section is to improve the basic algorithm in order to make it able to parse in grammars containing wildcards. An additional aim is to handle the priority of rewriting rules. The following changes are necessary in the basic algorithm:

- specifying the dotted rules in the prediction in the case of $B \in V_W$;
- handling the priority of the parse-trees;
- representing distinct types of wildcards.

3.1. Prediction

In the prediction, a dotted rule $(B \rightarrow \bullet\gamma)$ added to $Parse$ must be specified by the content of the input string if $B \in V_W$. If the i th symbol in the input string is the next one which should be parsed then the dotted rules $(W \rightarrow \bullet)$, $(W \rightarrow \bullet w_i)$, $(W \rightarrow \bullet w_i w_{i+1}), \ldots, (W \rightarrow \bullet w_i w_{i+1} \cdots w_n)$ should be added to $Parse$ .Since the right-hand-sides of these rules are constructed based on the input string, all of them will be used in the scanning until the completion is performed on them. Accordingly, the algorithm can append the completed form of these dotted rules to $Parse$ immediately in the prediction, in order to save time. After all, the point 6(b) of the algorithm is modified as follows.

```
6. (b) $\boxed{\text{prediction}}$
    i. if $B \notin V_W$ then for all $(B \to \gamma) \in \Phi$ :
        $Parse.push([k, k, (B \to \bullet\gamma), \langle\rangle])$;
    ii. if $B \in V_W$ then for all $j$, $k \leq j \leq n$:
        if CHECK_WILDCARD$(w_{k+1} \ldots w_j)$ then
        $Parse.push([k, j, (B \to w_{k+1} \ldots w_j \bullet), \langle\rangle])$;
```

The condition *CHECK_WILDCARD* is a check whether a text is valid to be assigned to a wildcard. This facility makes us able to categorize wildcards on-demand.

3.2. Priority

The algorithm must emit the parse-trees in the order of their priority. At any point of parsing, the algorithm must finish the construction of the parse-tree having the highest priority before continuing to the construction of the other ones. Accordingly, the operation *push* used in the basic algorithm cannot be performed in the proposed variant, because it places a newly generated state at the end of *Parse* , i.e., after the states that belong to the parse-trees with a lower priority. From now on, the operation *insert* will be used instead of *push*. For an ordered set K, $K.insert(p,i)$ represents the insertion of a new element p into K at the position i $(1 \leq i \leq K.size+1)$. In the case of $K.insert(p,i)$ each element of K after the position $i-1$ is shifted to one higher position.

In the algorithm, the operations of *Parse.push* must be changed at the points 1.(b), 5, 6.(a), 6.(b).i, 6.(b).ii, and 7. as follows.

```
1.(b) $Parse.insert([0, 0, (S \to \bullet\alpha), \langle\rangle], 1)$
5. $Parse.insert([i, j+1, (A \to \alpha a \bullet \beta), \pi], pointer + 1)$
6.(a) $Parse.insert([i, j, (A \to \alpha B \bullet \beta), \langle p_1, \ldots, p_s, p\rangle], pointer + 1)$
6.(b).i. $Parse.insert([k, k, (B \to \bullet\gamma), \langle\rangle], pointer + 1)$
6.(b).ii. $Parse.insert([k, j, (B \to w_{k+1} \ldots w_j \bullet), \langle\rangle], pointer + 1)$
7. $Parse.insert([i, j, (A \to \alpha B \bullet \beta), \langle p_1, \ldots, p_s, pointer\rangle], pointer + 1)$
```

Actually, *Parse* represents a compound data structure after these changes. This data structure consists of a vector and a stack; its elements before the position *pointer* belong to the vector, and the other ones to the stack. The *pointer* points to the top element of the stack. The vector contains the states which have been already used for constructing a parse-tree, and might be used in the completion in the future. The stack contains the states which have not been used yet, ordered by the priority of parse-trees.

What a parse-tree with a higher priority is, should be specified by the dialogue system. Usually, a measurement is defined in a dialogue system in order to calculate the priority of rewriting rules. Our algorithm can be improved in order to access earlier the rewriting rules having higher priority than the others of lower priority. This improvement is very simple: at each point of the algorithm where a universal quantifier ("for all"') is located (at the points 1.(b), 6.(a), 6.(b).i, 6.(b).ii, and 7.), the quantifier is replaced with a loop. Such a loop must access the elements referred by the quantifier and place them onto the stack in reverse order of priority, i.e., from the element having the lowest priority toward the one having the highest priority. In 6.(a) and 7., it is enough to access the states backwards in *Parse* (from the position *pointe-1* toward 1), since these states are already ordered in *Parse* by the priorities of the parse-trees.

3.3. The Types of Wildcards

Two types of wildcards are distinguished based on the priority of parse-trees. A wildcard is *greedy* if assigning a lengthy string by the parse tree to it has higher priority than assigning a short sting. All the other wildcards are *non-greedy*. Greedy wildcards can be implemented by specifying the priority of rewriting rules used in 6(b)ii in the following way: the rule $(B \to a\alpha)$ has higher priority than $(B \to \alpha)$. Wildcards can be further classified by customizing the condition *CHECK_WILDCARD*, e.g., we can distinguish the wildcards whose length is at least one character from those whose length is at most one word, etc. In the first case, $CHECK\_WILDCARD(\gamma): \gamma \neq \lambda$.

In Figure 4, the parsing in a grammar including greedy and at-least-one-character-long wildcards is demonstrated. The states are enumerated by their location in $Parse$. The non-numbered states are erased during parsing, the numbered ones are located in the final $Parse$. The framed states represent the parse-trees which can be emitted; they are shown in the figure as well. For the sake of completeness, we have not stopped the parsing at any of the framed states, i.e., we have supposed a constantly false condition for *CHECK_PARSETREE*.

$\Phi = \{(S \to W_1 a W_2)\}$

$V_W = \{W_1, W_2\}$

input string: *eagai*

1:	$[0, 0, (S \to \bullet W_1 a W_2), \langle\rangle]$		$[0, 2, (S \to W_1 \bullet a W_2), \langle 8\rangle]$
2:	$[0, 5, (W_1 \to eagai\bullet), \langle\rangle]$	**9:**	$[0, 1, (W_1 \to e\bullet), \langle\rangle]$
	$[0, 5, (S \to W_1 \bullet a W_2), \langle 2\rangle]$		$[0, 1, (S \to W_1 \bullet a W_2), \langle 9\rangle]$
3:	$[0, 4, (W_1 \to eaga\bullet), \langle\rangle]$	**10:**	$[0, 2, (S \to W_1 a \bullet W_2), \langle 9\rangle]$
	$[0, 4, (S \to W_1 \bullet a W_2), \langle 3\rangle]$	**11:**	$[2, 5, (W_2 \to gai\bullet), \langle\rangle]$
4:	$[0, 3, (W_1 \to eag\bullet), \langle\rangle]$	**12:**	$\boxed{[0, 5, (S \to W_1 a W_2\bullet), \langle 9, 11\rangle]}$
	$[0, 3, (S \to W_1 \bullet a W_2), \langle 4\rangle]$	**13:**	$[2, 4, (W_2 \to ga\bullet), \langle\rangle]$
5:	$[0, 4, (S \to W_1 a \bullet W_2), \langle 4\rangle]$	**14:**	$[0, 4, (S \to W_1 a W_2\bullet), \langle 9, 13\rangle]$
6:	$[4, 5, (W_2 \to i\bullet), \langle\rangle]$	**15:**	$[2, 3, (W_2 \to g\bullet), \langle\rangle]$
7:	$\boxed{[0, 5, (S \to W_1 a W_2\bullet), \langle 4, 6\rangle]}$	**16:**	$[0, 3, (S \to W_1 a W_2\bullet), \langle 9, 15\rangle]$
8:	$[0, 2, (W_1 \to ea\bullet), \langle\rangle]$		

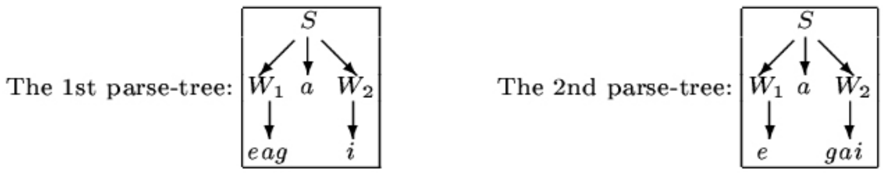


Fig. 4. Examples of parsing with two greedy wildcards.

4. Implementation

In our previous work [7], an XML-compliant dialogue system configuration language called Conversational Agent Markup Language (CAML) and a dialogue system core architecture called CAML Core was proposed. The proposed algorithm was incorporated in the CAML Core. CAML provides facilities to use greedy and at-least-one-character-long wildcards in patterns. Furthermore, it assigns priority numbers to the patterns by user's demand. In Figure 5, an example CAML unit (category) is shown. It specifies two patterns having the same priority. Both of them contain a wildcard. When a parse-tree has been emitted by the language parser of the CAML Core, the text assigned to the wildcard is stored in a variable called `museum_name`. At this point, the system checks the condition *CHECK_PARSETREE*, which is specified in the `condition` tag, i.e., it performs an SQL query in an external database specified by the parameter `database` whether the value of the variable can be found in its table called "*MUSEUMS*". The result of the database query is to be stored in the variable `museum_history`. If the value of this variable does not equal to the empty string, i.e., the SQL query has any result, then the *CHECK_PARSETREE* returns *TRUE*, otherwise *FALSE*. After the parsing has halted with acceptance, the system executes the content of the `action` tag, i.e., the value of the variable `museum_history` is emitted. In Figure 6, the parsing of a spoken user input is demonstrated.

```
<category name="MUSEUM QUERY" priority="10">
  <pattern>
    <li>TELL ME ABOUT THE MUSEUM <wildcard name="museum_name"/></li>
    <li>DO YOU HAVE ANY INFORMATION ABOUT THE MUSEUM <wildcard name="museum_name"/></li>
  </pattern>
  <condition>
    <set name="museum_history" type="sql" database="...">
      SELECT HISTORY FROM MUSEUMS WHERE NAME='<get name="museum_name"/>'
    </set>
    <return>
      <not><eq>
        <get name="museum_history"/>
        <text></text>
      </eq></not>
    </return>
  </condition>
  <action>
    <output><get name="museum_history"/></output>
  </action>
</category>
```

Fig. 5. A CAML category including wildcards.

	$[0, 0, (S \rightarrow \bullet tell\_about\_*\_to\_A), \langle\rangle]$
	$[0, 1, (S \rightarrow t\bullet ell\_about\_*\_to\_A), \langle\rangle]$
	$[0, 2, (S \rightarrow te\bullet ll\_about\_*\_to\_A), \langle\rangle]$
	$[0, 3, (S \rightarrow tel\bullet l\_about\_*\_to\_A), \langle\rangle]$
	$[0, 4, (S \rightarrow tell\bullet\_about\_*\_to\_A), \langle\rangle]$
	$[0, 5, (S \rightarrow tell\_\bullet about\_*\_to\_A), \langle\rangle]$
	$[0, 6, (S \rightarrow tell\_a\bullet bout\_*\_to\_A), \langle\rangle]$
	$[0, 7, (S \rightarrow tell\_ab\bullet out\_*\_to\_A), \langle\rangle]$
	$[0, 8, (S \rightarrow tell\_abo\bullet ut\_*\_to\_A), \langle\rangle]$
	$[0, 9, (S \rightarrow tell\_abou\bullet t\_*\_to\_A), \langle\rangle]$
	$[0, 10, (S \rightarrow tell\_about\bullet\_*\_to\_A), \langle\rangle]$
1:	$[0, 11, (S \rightarrow tell\_about\_\bullet *\_to\_A), \langle\rangle]$
2:	$[11, 24, (* \rightarrow weather\_to\_us\bullet), \langle\rangle]$
	$[0, 24, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 2\rangle]$
3:	$[11, 23, (* \rightarrow weather\_to\_u\bullet), \langle\rangle]$
	$[0, 23, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 3\rangle]$
4:	$[11, 22, (* \rightarrow weather\_to\_\bullet), \langle\rangle]$
	$[0, 22, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 4\rangle]$
5:	$[11, 21, (* \rightarrow weather\_to\bullet), \langle\rangle]$
	$[0, 21, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 5\rangle]$
	$[0, 22, (S \rightarrow tell\_me\_about\_*\_\bullet to\_A), \langle 5\rangle]$
6:	$[11, 20, (* \rightarrow weather\_t\bullet), \langle\rangle]$
	$[0, 20, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 6\rangle]$
7:	$[11, 19, (* \rightarrow weather\_\bullet), \langle\rangle]$
	$[0, 19, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 7\rangle]$
8:	$[11, 18, (* \rightarrow weather\bullet), \langle\rangle]$
	$[0, 18, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 8\rangle]$
	$[0, 19, (S \rightarrow tell\_me\_about\_*\_\bullet to\_A), \langle 8\rangle]$
	$[0, 20, (S \rightarrow tell\_me\_about\_*\_t\bullet o\_A), \langle 8\rangle]$
	$[0, 21, (S \rightarrow tell\_me\_about\_*\_to\bullet\_A), \langle 8\rangle]$
9:	$[0, 22, (S \rightarrow tell\_me\_about\_*\_to\_\bullet A), \langle 8\rangle]$
	$[22, 22, (A \rightarrow \bullet me), \langle\rangle]$
	$[22, 22, (A \rightarrow \bullet us), \langle\rangle]$
	$[22, 23, (A \rightarrow u\bullet s), \langle\rangle]$
10:	$[22, 24, (A \rightarrow us\bullet), \langle\rangle]$
11:	$\boxed{[0, 24, (S \rightarrow tell\_me\_about\_*\_to\_A\bullet), \langle 8, 10\rangle]}$
12:	$[11, 17, (* \rightarrow weathe\bullet), \langle\rangle]$
	$[0, 17, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 12\rangle]$
13:	$[11, 16, (* \rightarrow weath\bullet), \langle\rangle]$
	$[0, 16, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 13\rangle]$
14:	$[11, 15, (* \rightarrow weat\bullet), \langle\rangle]$
	$[0, 15, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 14\rangle]$
15:	$[11, 14, (* \rightarrow wea\bullet), \langle\rangle]$
	$[0, 14, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 15\rangle]$
16:	$[11, 13, (* \rightarrow we\bullet), \langle\rangle]$
	$[0, 13, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 16\rangle]$
17:	$[11, 12, (* \rightarrow w\bullet), \langle\rangle]$
	$[0, 12, (S \rightarrow tell\_me\_about\_*\bullet\_to\_A), \langle 17\rangle]$

Fig. 6. Example of a natural language parsing.

5. Conclusions

A context-free parsing algorithm has been developed that supports the use of wildcards and priority values in the rewriting rules. Both features are desirable and very essential entities in dialogue systems. The algorithm has been incorporated in our own dialogue system core, namely the CAML Core.

Acknowledgments

This work has been by the European Union funded Research Training Network “Multi-modal Human-Computer Interaction” (MUHCI).

References

1. Early J., **“An efficient context-free parsing algorithm”,** Communications of the ACM, vol. 13, no. 2, 1970.
2. Morawietz F., **“Chart parsing and constraint programming”**, in Proc. of Coling 2000, Saarbrüken, 2000.
3. Suereth R., “**Developing Natural Language Interfaces”**, N.Y.: McGraw-Hill, 1997.
4. Ole Bernsen N., Dybkjær H. and Dybkjær L., **“Designing Interactive Speech Systems”,** London: Springer-Verlag, 1998.
5. Huang X., Acero A. and Hon, H.-W., **“Spoken Language Processing”,** Upper Saddle River, N.J.: Prentice Hall PTR, 2001.
6. Gorin A.L., Abella A., Alonso T., Riccardi G. and Wright, J.H., **“Automated natural spoken dialog”**, IEEE Computer, vol. 35, no. 4 pp. 51-56, April 2002.
7. Kovácznai G., Kotropoulos C. and Pitas I., **“CAML: A universal configuration language for dialogue systems”**, in Proc. DEXA-2003 Conf. Prague, 2003, to appear.
8. Ward W., **“Understanding spontaneous speech: The Phoenix system”,** in Proc. ICASSP 1991, pp. 365-367, 1997.
9. Gustavsson C. Strindlund L. and Wilknertz E., “**Dialogue management tool”,** in Proc. OzCHI 2001 Workshop, 2001.