6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Algorithmic Improvements in Natural Language Parsing within Dialogue Systems: Priority Patterns and Wildcards

Gergely Kovásznai

Department of Computer Science, University of Debrecen, Hungary
e-mail: kovasz@inf.unideb.hu

Abstract

Nowadays, human-machine interaction is changing dramatically toward spoken dialogue. One of the subtasks performed by a dialogue system (DS) is the natural language (NL) parsing in order to extract meaning from spoken user input. The most regular tool used for NL parsing is a context-free (CF) grammar. Jay Earley's parser is a well-known algorithm for parsing in an arbitrary CF grammar. In this paper, an improved variant of Earley's parser is proposed in order to solve two essential problems that belong to NL parsing and arise from dialogue management. One of them is the use of partially-specified patterns, i.e., the use of wildcards in the right-hand-sides of the rewriting rules. The second one is the use of priority patterns, i.e., the assignment of priorities to rewriting rules. These problems mostly are not handled in the state-of-the-art DSs. The proposed parser has been implemented in a DS core application called the CAML Core, that is used to implement DSs in several domains like conversational and multi-modal interfaces for help desk applications, and for chat bots.

1. Introduction

In our days, many dialogue systems (DSs) are developed and used in several domains, e.g., help desk applications on airplane/train ticket reservation, chat bots, etc. Each DS includes a module that performs natural language (NL) parsing and understanding, i.e., the extraction of semantic information (let us say meaning) from a user input in a spoken language [2, 3, 4]. The most frequent technique for NL parsing is a context-free (CF) grammar. A problem inherent in a CF parser is the use of only fully-specified patterns, i.e., rewriting rules (rules) that have fully-specified right-hand-sides. From a practical point of view, it is impossible to prepare a CF grammar for each possible user input since it can include only a finite

set of rules. The use of partially-specified patterns, i.e., the support of *wildcards* within the right-hand-sides can solve this problem. By the use of wildcards, unpredictable information (e.g., names) can be extracted from a user input, furthermore a validity check on a text can be performed out of the parser (i.e., by a dialogue manager module). The majority of the state-of-the-art DSs employ NL parsers which support only fully-specified patterns, like the Phoenix Semantic Parser and the Dialogue Management Tool (see [5]).

Another facility which is quite important and usually not supported in NL parsing is the use of *priority patterns*. The need of placing rules in a priority order has arisen from dialogue management, where the analysis of a user input is performed usually in hierarchical steps, from the step with the highest priority toward the step with the lowest priority.

In Section 2, the basic concepts that are used in this paper are introduced. In Section 3, the basic variant of Jay Earley's parser [1] is proposed, which will be improved in Section 4 in order to use wildcards and priority patterns. This parser will be further improved in Section 5 for the sake of the soonest possible emission of parse-trees.

2. Context-free Parsing

Basic concepts.

Let a *CF grammar* be a quintuple $\langle V_T, V_{NT}, S, V_W, \Phi \rangle$, where $S \in V_{NT}$ and $V_W \subset V_{NT}$. V_T is the set of terminal symbols denoted by a, b, c in this paper. The non-terminal symbols, i.e., the elements of V_{NT}, are denoted by A, B, C. We refer strings that are elements of $(V_T \cup V_{NT})^\star$ as α, β, γ. The empty string is denoted by λ. We call the elements of V_W *wildcards*. Φ is an ordered set of rules. A rule $(A \to \alpha)$ located before another rule $(B \to \beta)$ in Φ is said to have higher *priority* than $(B \to \beta)$. During the parsing for $w_1 \dots w_N$ $(N \geq 0,\ w_1, \dots, w_N \in V_T)$, Φ may be extended with some rule that has a wildcard in the left-hand-side. The right-hand-side of such a rule depends on $w_1 \dots w_N$.

Since *ordered sets* are used in this paper, a notation is needed to be introduced. An ordered set K can be represented in the form $\langle s_1, \dots, s_k \rangle$ $(k \geq 0)$, where s_1 is its first element, s_2 is its second element, etc. The number of elements of K is written in the form $K.size$. The ith element of K $(1 \leq i \leq K.size)$ is denoted by $K[i]$.

During parsing, so-called *states* are constructed. A *state* is written in the form $[i, j, (A \to \alpha \bullet \beta), \pi]$, where $0 \leq i, j \leq N$, $\bullet \notin V_T \cup V_{NT}$, and π is an ordered set of states. π contains as many states as the number of the non-terminal symbols in α (see [1]). Each state represents a semi-tree, i.e. a possibly incomplete parse-tree. A *completed form* of an $[i, j, (A \to \alpha \bullet \beta), \langle s_1, \dots, s_m \rangle]$ is a state $[i, k, (A \to \alpha\beta\bullet), \langle s_1, \dots, s_n \rangle]$ $(k \geq j,\ n \geq m)$.

The sequential nature of the parse-trees.

Since natural languages are inherently based on ambiguous grammars, it usually happens that more than one 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 rules, the parser must emit the trees in the order of their priorities that can be computed on the basis of the priorities of the rules. This is why we need a parsing algorithm which can deal with a sequence of outputs. In contrast with Earley's original parser, the proposed parser constructs one single ordered set (not sets) of states.

3. The Basic Parser

The basic parser is actually Earley's original algorithm in a form which will be suitable for our further improvements. It is used to parse in a CF grammar that does not include wildcards. It emits the trees one after the other, this is why it possess the ability to emit the trees in a pre-defined order, e.g., in the order of their priorities (see Section 4 and 5).

As can be seen in the algorithm introduced in Figure 1, the constructed ordered set of states is called $Parse$. In the variable $index$, the position of the state currently focused (within $Parse$) is stored. $Parse$ and $index$ are initialized in the phase initialization (item (I)). In the initialization, only one state gets to be stored in $Parse$, namely $[0, 0, (S' \to \bullet S), \langle\rangle]$ where $S' \notin V_T \cup V_{NT}$.

(I) initialization
  (A) $Parse := \langle\rangle$, $Parse.push([0, 0, (S' \to \bullet S), \langle\rangle])$;
  (B) $index := 1$.
(II) If $index > Parse.size$ then halt with rejection.
(III) $state := Parse[index]$.
(IV) If $state = [0, N, (S' \to S\bullet), \pi]$ then
  if CHECK_PARSETREE($state$) then halt with acceptance.
(V) If $state = [i, j, (A \to \alpha_1 \bullet a\alpha_2), \pi]$ then
  scanning
  if $j < N$ and $a = w_{j+1}$ then
  (A) $Parse.modify(index, [i, j+1, (A \to \alpha_1 a \bullet \alpha_2), \pi])$;
  (B) go to (III).
(VI) If $state = [i, k, (A \to \alpha_1 \bullet B\alpha_2), \langle s_1, \ldots, s_m\rangle]$ then
  (A) prediction
  for all $(B \to \beta) \in \Phi$:
    $Parse.push([k, k, (B \to \bullet\beta), \langle\rangle])$;
  (B) completion1
  for all $s := [k, j, (B \to \beta\bullet), \pi] \in Parse$ where $Parse.index(s) < Parse.index(state)$:
    $Parse.push([i, j, (A \to \alpha_1 B \bullet \alpha_2), \langle s_1, \ldots, s_m, s\rangle])$.
(VII) If $state = [k, j, (B \to \beta\bullet), \pi]$ then
  completion2
  for all $s := [i, k, (A \to \alpha_1 \bullet B\alpha_2), \langle s_1, \ldots, s_m\rangle] \in Parse$ where $Parse.index(s) < Parse.index(state)$:
    $Parse.push([i, j, (A \to \alpha_1 B \bullet \alpha_2), \langle s_1, \ldots, s_m, state\rangle])$.
(VIII) $index := index + 1$.
(IX) Go to (II).

Figure 1: The basic parser.

Thee new operations must be defined for an ordered set K. $K.push(s)$ denotes the operation of appending s to K. $K.modify(i, s)$ denotes the operation of

modifying $K[i]$ to s. $K.index(s)$ returns the position $s \in K$ in K.

At the items (II) and (IV), the halt conditions are formulated. If the parser is focusing a so-called final state then it can halt with the acceptance of $w_1 \dots w_N$ (at (IV)). If the parser has not constructed any final state and there is no state left to be focused then it halts with rejection (at (II)). The parser can be customized by specifying the condition CHECK_PARSETREE which is an additional check if the constructed tree represented by $Parse[index]$ is valid and can be emitted.

The phases corresponding to the three basic operations of Earley's parser, namely the scanning, the completion, and the prediction, are indicated in Figure 1. Since arbitrary CF grammars can be used (i.e., they may contain even a rule $(A \rightarrow \lambda)$), the completion cannot be applied in a straightforward way (as mentioned in [1]), this is why the completion is performed in two symmetrical phases, namely in the completion1 and the completion2.

Actually, $Parse$ represents a compound data structure, namely a composition of a FIFO (first-in, first-out) data structure and a LIFO (last-in, first-out) one. FIFO consists of the states that have already been focused, and LIFO contains the ones that have not been yet. The variable *index* separates the two data structures since *index* points to the first element of LIFO and the first element of FIFO is located at the position $index - 1$.

4. Wildcards and Priority in the Improved Parser

The basic parser is going to be improved in this section in order to handle wildcards and the priorities of rules. In the improved parser, the following elements should be incorporated:

- a modified *prediction* in order to specify new states in the case of $B \in V_W$;
- distinct *types of wildcards*, e.g., greedy and at-least-one-character-long ones;
- *former emission* of a tree with higher priority than other ones.

Handling wildcards.

In terms of the predicition, if w_i is the next symbol which should be parsed then the dotted rules $(B \rightarrow \bullet), (B \rightarrow \bullet w_i), (B \rightarrow \bullet w_i w_{i+1}), \dots, (B \rightarrow \bullet w_i w_{i+1} \dots w_N)$ should be used to generate new states. 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, their completed form can be added (to $Parse$) immediately in the prediction (as can be seen at (III)(C)(2)(a) in Figure 2), in order to save time.

The improved parser can be customized by specifying the condition CHECK_-WILDCARD, which is a check if a $(B \rightarrow \beta)$ where $B \in V_W$ is valid. This is how we can use at-least-one-character-long wildcards (CHECK_WILDCARD(β): $\beta \neq \lambda$), for example. Another support for customizing wildcards is the customization of the priority of $(B \rightarrow \beta)$ rules. This is how we can use greedy wildcards ($(B \rightarrow \beta_1)$ has higher priority than $(B \rightarrow \beta_2)$ if $|\beta_1| > |\beta_2|$), for example.

```
(I)  [initialization]
   (A)  Parse := ⟨[ENDSTATE]⟩, Parse.insert(1, [0, 0, (S′ → •S), ⟨⟩]);
   (B)  D_Root := ∅, Root(S′, 0) := [ENDSTATE];
   (C)  index := 1.
(II)(A)  state := Parse[index];
   (B)  if state = [ENDSTATE] then halt with rejection.
(IV)  If state is INACTIVE then
   (A)  if state = [0, N, (S′ → S•), π] then
          if CHECK_PARSETREE(state) then halt with acceptance;
   (B)  if state = [i, j, (A → α1 • aα2), π] then
      (1)  [scanning]
           if j < N and a = w_{j+1} then
        (a)  Parse.modify(index, [i, j + 1, (A → α1a • α2), π]);
        (b)  go to (II);
      (2)  set state USED;
   (C)  if state = [i, k, (A → α1 • Bα2), ⟨s1, ..., sm⟩] then
      (1)  set state USED;
      (2)  if Root(B, k) is undefined then
           [prediction]
        (a)  ∘ if B ∉ V_W then
                 for all (B → β) ∈ Φ in reverse order of their priority:
                   Parse.insert(index, [k, k, (B → •β), ⟨⟩]);
             ∘ else
                 for all j where k ≤ j ≤ N in reverse order of the priority of (B → w_{k+1} ... w_j) rules:
                   if CHECK_WILDCARD(w_{k+1} ... w_j) then
                     Parse.insert(index, [k, j, (B → w_{k+1} ... w_j•), ⟨⟩]);
        (b)  Root(B, k) := state;
        (c)  if Parse[index] ≠ state then go to (V);
      (3)  [completion1]
           for all USED s := [k, j, (B → β•), π] ∈ Parse:
           a  s′ := [i, j, (A → α1B • α2), ⟨s1, ..., sm, s⟩];
           b  insert(state, s′);
   (D)  if state = [k, j, (B → β•), π] then
      (1)  set state USED;
      (2)  [completion2]
           for all USED s := [i, k, (A → α1 • Bα2), ⟨s1, ..., sm⟩] ∈ Parse:
           a  s′ := [i, j, (A → α1B • α2), ⟨s1, ..., sm, state⟩];
           b  insert(s, s′).
(V)(a)  i := min{Parse.index(t)|t ∈ Parse is INACTIVE};
   (b)  if i > index then i := index + 1;
   (c)  index := i.
(VI)  Go to (II).
```

Figure 2: The improved parser.

Concepts.

Let us introduce the function *Root* which assigns a state to a non-terminal symbol A and a position i, $0 \leq i \leq N$. A state $s = Root(A, i)$ if some $[i, i, (A \rightarrow \bullet\alpha), \langle\rangle]$ was added when s was being focused (in the prediction). The state s is said to be the root of each $[i, j, (A \rightarrow \alpha_1 \bullet \alpha_2), \pi]$. Let $s \xrightarrow{\text{root}} s'$ denote that s is the root of a state s', and let $s \xrightarrow{\text{root}}_k s'$ $(k \geq 1)$ denote that there is a sequence of $s_1, \ldots, s_k$ where $s_1 = s$, $s_k = s'$, and $s_i \xrightarrow{\text{root}} s_{i+1}$ for all i $(1 \leq i \leq k-1)$.

It can be seen that each state in $Parse$ has a root except $[0, 0, (S' \rightarrow \bullet S), \langle\rangle]$, so that a special state denoted by [ENDSTATE] must be introduced as $Root(S', 0)$. The state [ENDSTATE] must differ from any possible state that can be generated. Like $[0, 0, (S' \rightarrow \bullet S), \langle\rangle]$, [ENDSTATE] is added in the initialization (at (I)). Futhermore, [ENDSTATE] is used in the new halt condition for rejection (at (II)(B)).

Handling priority.

Since a tree having higher priority than other ones should be emitted before them, the ordering of the states in *Parse*, as introduced in the basic algorithm, is not suitable any more, i.e., *Parse* cannot be splitted to FIFO and LIFO data structures hereafter. The set of states that have already been focused (used states) and the set of states that have not been focused yet (inactive states) will be general ordered sets. Since the states should be ordered according to their priorities, the used and the inactive states can be located mixed in *Parse* (i.e., *index* does not separate them). Accordingly, the *activation-flag* which can get the values either INACTIVE or USED is being introduced for a state, denoting if the given state is an inactive state or a used one. Every state is INACTIVE by default.

Since a new state can be added to *Parse* not only at the end, the operation $K.insert(i, s)$ will be used for this insertion. For an ordered set K, $K.insert(i, s)$ represents the insertion of s into K at the position i ($1 \leq i \leq K.size+1$). Each element of K located not before the position i is shifted to one higher position, before s is placed to the position i.

New states can be inserted (into *Parse*) in two different ways: either in the *prediction* or in the completion. As can be seen at (III)(C)(2)(a), all the new states are inserted at the position *index* in the prediction, i.e., just before the currently focused state. They are inserted one after the other in reverse order of their priorities, in order to get them ordered (in *Parse*) by their priorities. Actually, all the states are not needed to be entirely ordered, only the states that have the same root must be ordered as compared to each other.

In terms of the *completion*, a special function is needed to be introduced in order to ensure the aforementioned root-based ordering. This function is denoted by $insert(state, s)$ (shown in Figure 3) where $state \in Parse$, and represents the root-based insertion (i.e., an insertion method that preserves root-based ordering) of s into *Parse* after the position of *state*.

$insert(state, [i_1, i_2, (A \to \alpha_1 \bullet \alpha_2), \langle s_1, \ldots, s_m \rangle])$:

(1)(a) $p := Parse.index(state)$;
(b) $ins\_pos := 0$.
(2)(a) $p := p + 1$;
(b) let $[j_1, j_2, (B \to \beta_1 \bullet \beta_2), \langle t_1, \ldots, t_n \rangle]$ denote $Parse[p]$.
(3)(a) If $Parse[p] = Root(A, i_1)$ then go to (7);
(b) if $Parse.index(Root(A, i_1)) = Parse.index(Root(B, j_1))$ then
- if $\alpha_1\alpha_2 \neq \beta_1\beta_2$ go to (7);
- else go to (4);

(c) if $Parse.index(Root(A, i_1)) > Parse.index(Root(B, j_1))$ then go to (5);
(d) if $Parse.index(Root(A, i_1)) < Parse.index(Root(B, j_1))$ then go to (6).
(4)(a) $r := 1$;
(b) if $r > n$ then go to (7);
(c) if $Parse.index(s_r) > Parse.index(t_r)$ then go to (6);
(d) if $Parse.index(s_r) < Parse.index(t_r)$ then go to (7);
(e) $r := r + 1$;
(f) go to (4)(b).
(5)(a) If $ins\_pos = 0$ then $ins\_pos := p$;
(b) go to (2).
(6)(a) $ins\_pos := 0$;
(b) go to (2).
(7)(a) If $ins\_pos = 0$ then $ins\_pos := p$;
(b) $Parse.insert(ins\_pos, [i_1, i_2, (A \to \alpha_1 \bullet \alpha_2), \langle s_1, \ldots, s_m \rangle])$.

Figure 3: The insertion function.

Root-based insertion.

As can be seen in Figure 3, the insertion function tries to find the proper position for inserting the state s (let $[i_1, i_2, (A \to \alpha_1 \bullet \alpha_2), \langle s_1, \ldots, s_m \rangle]$ denote s) by comparing s to all the states after *state* one after the other (by the use of the position variable p). By the use of the variable ins_pos, the latest possible position for insertion can be stored (if $ins\_pos = 0$ then there has been no position to remember).

Let s' denote $Parse[p]$. The search immediately ends (and the insertion gets performed at (7)) when s' is the root of s (at (3)(a)). Something similar is done if s' has the same root as s (at (3)(b)): if s and s' have been generated from the same rule $(A \to \alpha_1 \alpha_2)$ then the priorities of the semi-trees represented by s and s' are compared (at (4)). If this comparison shows that s has higher priority than s', the insertion gets performed immediately. In all the other cases, the search is being continued.

One of them is the case when the root of s is located after the root of s' (at (3)(c)). It means that $s' \xrightarrow{\text{root}}_k Root(A, i_1)$ where $k > 1$, i.e., $[j_1, j_1, (B \to \bullet \beta_1 \beta_2), \langle\rangle]$ was inserted later than $[i_1, i_1, (A \to \bullet \alpha_1 \alpha_2), \langle\rangle]$.

The other case (at (3)(d)), when $s \xrightarrow{\text{root}}_k Root(B, j_1)$ where $k > 1$, cannot happen in the improved parser. Since it may happen in the expedited parser, look for further explanation in Section 5.

5. Immediate Emission in the Expedited Parser

It can be easily seen that the improved parser proposed in Section 4 is quite ineffective and has been created only for a purpose of further improvements made in this section, which will result a parser called the expedited parser (shown in Figure 4 where a phase name followed by an asterisk refers to a phase with the same name in the improved parser in Figure 2). The improved parser emits the trees only at the end of the parse. This functionality could be easily realised by simply collecting the trees generated by the basic parser and emitting them in priority order after the basic parser has ended. Contrarily, the expedited parser emits a tree as soon as possible, i.e., if no other tree having higher priority can be constructed thereafter.

Let us introduce the expression "*immediate emission of a state*", which means that a state generated from a state s and from the root of s (in the completion2) can be focused immediately (see (IV)(E)(2)). In order to observe if a state can be emitted immediately, the new value ,is introduced for the activation-flag. For a state s, being active (i.e., the state has the activation-flag set to ,) means that its completed form $s' \notin Parse$ may be inserted thereafter (the activation-flag can be set to ,at (IV)(D)(1)). Furthermore, we are introducing a new flag called the *collection-flag*. If this flag is set for a state then the state is said to be collected (a state is non-collected by default and its collection flag can be set at (IV)(B), (IV)(D)(4)(b), and (IV)(E)(2)(b)). For a state s, being collected means that an $s' \notin Parse$ may

be inserted thereafter where the roots of s and s' are the same and s' has higher priority then s. It can be seen that a non-collected state generated from a state s and from the root of s can be emitted immediately, as done at (IV)(E)(2)(c) where such a state is inserted before all the inactive states (i.e., it will be focused next time).

```
(I)(A)  initialization ⋆ ;
   (B)  D_RRoot := ∅.
(II)(A) state := Parse[index];
   (B)  if state = [ENDSTATE] then halt with rejection.
(III)  If state is ACTIVE then
   (A)  let [i, k, (A → α1 • Bα2), π] denote state;
   (B)  ∘ if RRoot(B, k) in undefined then set state USED;
        ∘ else for all s := [k, j, (C → γ1 • γ2), φ] ∈ Parse:
              if RRoot(C, k) is defined and RRoot(C, k) = state then
            (1) set s USED and non-COLLECTED;
            (2) set state USED.
(IV)  If state is INACTIVE then
   (A)  if state = [0, N, (S' → S•), π] then
            if CHECK_PARSETREE(state) then halt with acceptance;
   (B)  if state = [k, k, (A → •α), ⟨⟩] then
            if there is an ACTIVE [k, k, (A → •γ), ⟨⟩] then set state COLLECTED;
   (C)  if state = [i, j, (A → α1 • aα2), π] then
      (1)  scanning ⋆ ;
      (2)  set state USED;
   (D)  if state = [i, k, (A → α1 • Bα2), ⟨s1, . . . , sm⟩] then
      (1)  set state ACTIVE;
      (2)  if Root(B, k) is undefined then
               prediction ⋆ ;
      (3)  ∘ if there is an ACTIVE [k, k, (B → •β), ⟨⟩] ∈ Parse then
             (a) – if RRoot(B, k) is defined then r := RRoot(B, k);
                 – else r := Root(B, k);
             (b) for all ACTIVE s := [k, k, (C → •γ), ⟨⟩] ∈ Parse where Parse.index(Root(C, k)) ≤
                 Parse.index(r):
                     RRoot(C, k) := r;
           ∘ else set state USED;
      (4)  completion1
           for all USED s := [k, j, (B → β•), π] ∈ Parse:
             (a) s' := [i, j, (A → α1B • α2), ⟨s1, . . . , sm, s⟩];
             (b) if any of state and s is COLLECTED then set s' COLLECTED;
             (c) insert(state, s');
   (E)  if state = [k, j, (B → β•), π] then
      (1)  set state USED;
      (2)  completion2
           for all ACTIVE s := [i, k, (A → α1 • Bα2), ⟨s1, . . . , sm⟩] ∈ Parse in ascendent order of
           Parse.index(s):
             (a) s' := [i, j, (A → α1B • α2), ⟨s1, . . . , sm, state⟩];
             (b) if any of state and s is COLLECTED then set s' COLLECTED;
             (c) ∘ if s = Root(B, k) and s' is non-COLLECTED then
                   (i) i := min{Parse.index(t)|t ∈ Parse is INACTIVE};
                  (ii) if i > index then i := index + 1;
                 (iii) Parse.insert(i, s');
                 ∘ else insert(s, s').
(V)(a) i := min{Parse.index(t)|t ∈ Parse is INACTIVE};
   (b) if i > index then i := index + 1;
   (c) index := i.
(VI)  Go to (II).
```

Figure 4: The expedited parser.

Since a state $[i, k, (A \to \alpha_1 B \bullet \alpha_2), \phi]$ can be inserted before the location of $[i, j, (A \to \alpha_1 \bullet B\alpha_2), \pi]$ (at (IV)(E)(2)(c)), the case mentioned at the end of Section 4 and handled at (3)(d) in Figure 3 can happen in the expedited parser, in contrast with the improved parser.

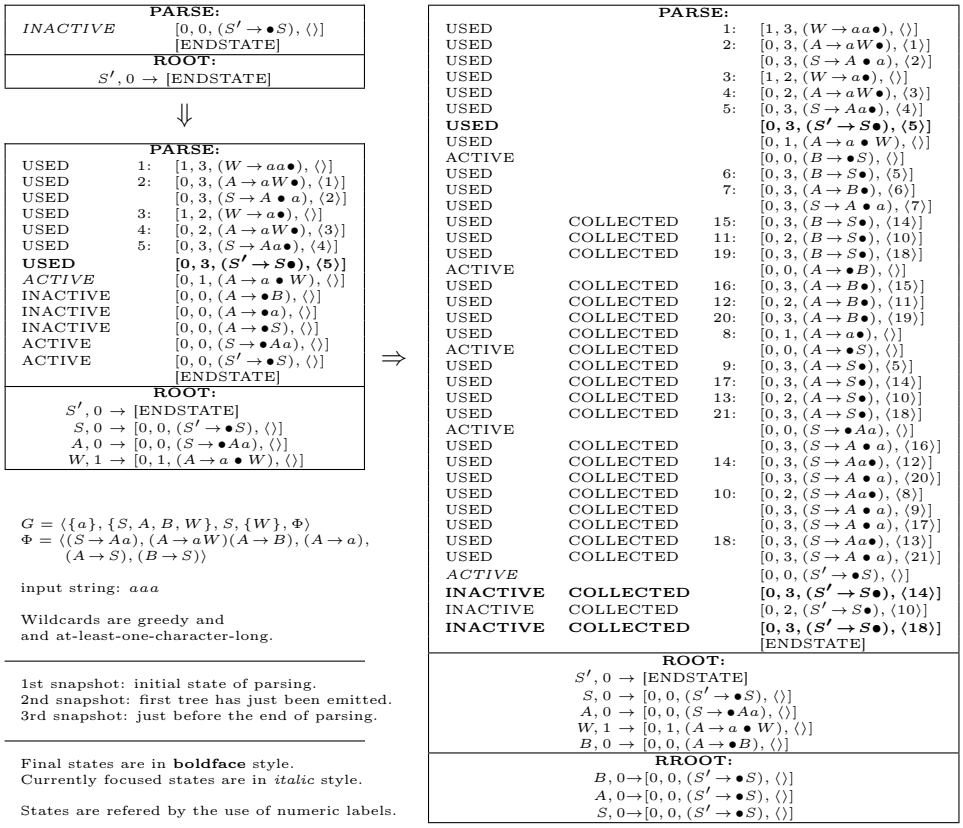

Figure 5: An example parsing.

Handling recursion.

The straightforward way of the generation of states gets absolutely impossible because of the existence of $[i, i, (A \to \alpha_1 \bullet A\alpha_2), \pi]$ states, which we call recursive states. The situation gets even more complicated if a kind of indirect recursion occurs, i.e., there is a subset $\{[i, i, (A_1 \to \alpha_1 \bullet A_2\beta_1), \pi_1], [i, i, (A_2 \to \alpha_2 \bullet A_3\beta_2), \pi_2], \ldots, [i, i, (A_k \to \alpha_k \bullet A_1\beta_k), \pi_k]\}$ of $Parse$, where $k \geq 1$. Such a set we will call a *recursive chain*.

Let us introduce the term of a *recursive set*. A recursive chain itself is a recursive set. Futhermore, $R_1 \cup R_2$ is a recursive set where R_1 and R_2 are recursive sets, if there is an $[i, i, (A \to \alpha_1 \bullet \alpha_2), \pi_1] \in R_1$ and there is an $[i, i, (A \to \beta_1 \bullet \beta_2), \pi_2] \in R_2$. From a pratical point of view, all the elements of a recursive set depend on each other in the following sense: the generation of a completed form of an element can cause the generation of a completed form of any element.

A *maximal recursive set* is a recursive set R_1 for that no recursive set R_2 can be found where $R_1 \cup R_2$ is a recursive set and $|R_1 \cup R_2| > |R_1|$. In order to

represent maximal recursive sets, the function *RRoot* is introduced. Similarly to *Root*, it can assign a state to a non-terminal symbol A and an index i ($0 \leq i \leq N$), and it is initialized at (I). $RRoot(A, i)$ can be specified in the following way: $RRoot(A, i) = r$ if there is an $\{s_1, \ldots, s_k\}$ recursive set where $Parse.index(r) = max\{Parse.index(r_i) | r_i$ is the root of $s_i, 1 \leq i \leq k\}$. The assignment pairs for *RRoot* are specified at (IV)(D)(3). *RRoot* is used at (III) in order to check if an active state can be transformed to a used one, i.e., no completed form of the given state may be generated thereafter.

In Figure 5, the parse of a grammar that includes a wildcard and causes recursivity is illustrated.

6. Realization and Future Plans

The proposed parser has been implemented and incorporated in the CAML Core [5, 6]. Our parser could be further improved by employing lookahead in the prediction [7], and by avoiding infinite loops in parsing.

References

[1] J. Earley, "An Efficient Context-Free Parsing Algorithm", *Communications of the ACM*, 1970, Vol. 13, Issue 2.

[2] R. Suereth, "Developing Natural Language Interfaces". N.Y., McGraw-Hill, 1997.

[3] N. Ole Bernsen, H. Dybkjær, and L. Dybkjær, "Designing Interactive Speech Systems". London, Springer-Verlag, 1998.

[4] X. Huang, A. Acero, and H.-W. Hon, "Spoken Language Processing". Upper Saddle River, N.J., Prentice Hall PTR, 2001.

[5] G. Kovásznai, C. Kotropoulos, and I. Pitas, "CAML – A Universal Configuration Language for Dialogue Systems", *Lecture Notes in Computer Science*. Springer, 2003, Vol. 2736, p. 896 - 906.

[6] Web page of the CAML and the CAML Core: `http://caml.no-ip.com` .

[7] M. Bouckaert, A. Pirotte, M. Snelling, "Efficient parsing algorithms for general context-free parsers", *Information Sciences*. 1975, Vol. 8, p. 1-26.