BOTTOM-UP PARSING EXTENDING CONTEXT-FREENESS IN A PROCESS GRAMMAR PROCESSOR

Massimo Marino
Department of Linguistics - University of Pisa
Via S. Maria 36 I-56100 Pisa - ITALY
Bitnet: massimom@icnucevm.cnuce.cnr.it

ABSTRACT

A new approach to bottom-up parsing that extends Augmented Context-Free Grammar to a Process Grammar is formally presented. A Process Grammar (PG) defines a set of rules suited for bottom-up parsing and conceived as processes that are applied by a PG Processor. The matching phase is a crucial step for process application, and a parsing structure for efficient matching is also presented. The PG Processor is composed of a process scheduler that allows immediate constituent analysis of structures, and behaves in a non-deterministic fashion. On the other side, the PG offers means for implementing specific parsing strategies improving the lack of determinism innate in the processor.

1. INTRODUCTION

Bottom-up parsing methods are usually preferred because of their property of being driven from both the input's syntactic/semantic structures and reduced constituents structures. Different strategies have been realized for handling the structures construction, e.g., parallel parsers, backtracking parsers, augmented context-free parsers (Aho et al., 1972; Grishman, 1976; Winograd, 1983). The aim of this paper is to introduce a new approach to bottom-up parsing starting from a well known and based framework - parallel bottom-up parsing in immediate constituent analysis, where all possible parses are considered - making use of an Augmented Phrase-Structure Grammar (APSG). In such environment we must perform efficient searches in the graph the parser builds, and limit as much as possible the building of structures that will not be in the final parse tree. For the efficiency of the search we introduce a Parse Graph Structure, based on the definition of adjacency of the subtrees, that provides an easy method of evaluation for deciding at any step whether a matching process can be accomplished or not. The control of the parsing process is in the hands of an APSG called Process Grammar (PG), where grammar rules are conceived as processes that are applied whenever proper conditions, detected by a process scheduler, exist. This is why the parser, called PG Processor, works following a non-deterministic parallel strategy, and only the Process Grammar has the power of altering and constraining this behaviour by means of some Kernel Functions that can modify the control structures of the PG Processor, thus improving determinism of the parsing process, or avoiding construction of useless structures. Some of the concepts introduced in this paper, such as some definitions in Section 2., are a development from Grishman (1976) that can be also an introductory reading regarding the description of a parallel bottom-up parser which is, even if under a different aspect, the core of the PG Processor.

2. PARSE GRAPH STRUCTURE

The Parse Graph Structure (PGS) is built by the parser while applying grammar rules. If $s = a_1 a_2 \ldots a_n$ is an input string the initial PGS is composed by a set of terminal nodes $<0,\$>$, $<1,a_1>$, $<2,a_2>, \ldots, <n,a_n>$, $<n+1,\$>$, where nodes 0,n+1 represent border markers for the sentence. All the next non-terminal nodes are numbered starting from n+2.

Definition 2.1. A PGS is a triple (N_T,N_N,T) where N_T is the set of the terminal nodes numbers $\{0, 1, \ldots, n, n+1\}$; N_N is the set of the non-terminal nodes numbers $\{n+2,\ldots\}$, and T is the set of the subtrees.

The elements of N_N and N_T are numbers identifying nodes of the PGS whose structure is defined below, and throughout the paper we refer to nodes of the PGS by means of such nodes number.

Definition 2.2. If $k \in N_N$ the node $i \in N_T$ labeling a_i at the beginning of the clause covered by k is said to be the **left corner leaf** of k lcl(k). If $k \in N_T$ then lcl(k)=k.

Definition 2.3. If $k \in N_N$ the node $j \in N_T$ labeling a_j at the end of the clause covered by k is said to be the **right corner leaf** of k rcl(k). If $k \in N_T$ then rcl(k) = k.

Definition 2.4. If $k \in N_N$ the node $h \in N_T$ that follows the right corner leaf of k rcl(k) is said to be the **anchor leaf** of k al(k), and al(k) = h = rcl(k)+1. If $k \in N_T-\{n+1\}$ then al(k) = k+1.

Definition 2.5. If $k \in N_T$ **the set of the anchored nodes** of k an(k) is $an(k) = \{j \in N_T \cup N_N \mid al(j) = k\}$.

From this definition it follows that for every $k \in N_T-\{0\}$, an(k) contains at the initial time the node number (k-1).

Definition 2.6. a. If $k \in N_T$ the subtree rooted in k T(k) is represented by T(k) = <k,lcl(k),rcl(k),an(k),cat(k)>, where k is the root node; lcl(k) = rcl(k) = k; an(k) = {(k-1)} initially; cat(k) = a_k, the terminal category of the node.

b. If $k \in N_N$ the subtree rooted in k T(k) is represented by T(k)=<k,lcl(k),rcl(k),sons(k),cat(k)>, where k is the root node; $sons(k) = \{s_1,\ldots,s_p\}$, $s_i \in N_T \cup N_N$, $i = 1,\ldots,p$, is the set of the direct descendants of k; cat(k) = A, a non-terminal category assigned to the node.

From the above definitions the initial PGS for a sentence $s=a_1a_2...a_n$ is: $N_T=\{0,1,...,n,n+1\}$, $N_N=\{\}$, $T=\{T(0),T(1),...,T(n),T(n+1)\}$; and: $T(0)=<0,0,0,\{\},\$>$, $T(i)=<i,i,i,\{i-1\},a_i>$ for $i=1,...,n$, and $T(n+1)=<n+1,n+1,n+1,\{n\},\$>$. With this PGS the parser starts its work reducing new nodes from the already existing ones. If for some $k\in N_N$, $T(k)=<k,lcl(k),rcl(k),\{s_1,...,s_p\},A>$, and $T(s_i)=<s_i,lcl(s_i),rcl(s_i),\{s_{i1},...,s_{ip_i}\},z_i>\in T$, for $i = 1,...,p$, are the direct descendants of k, then k has been reduced from $s_1,...,s_p$ by some grammar rule whose reduction rule, as we shall see later, has the form $(A\leftarrow z_1...z_p)$, and the following holds: $lcl(k) = lcl(s_1)$, $rcl(s_1) = lcl(s_2)-1$, $rcl(s_2) = lcl(s_3)-1$, ..., $rcl(s_{p-1}) = lcl(s_p)-1$, $rcl(s_p) = rcl(k)$. From that we can give the following definition:

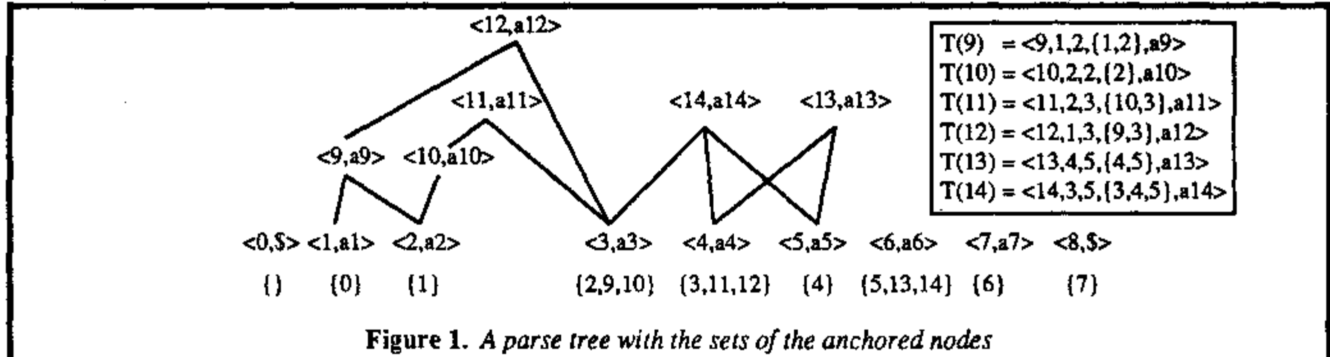

Figure 1. *A parse tree with the sets of the anchored nodes*

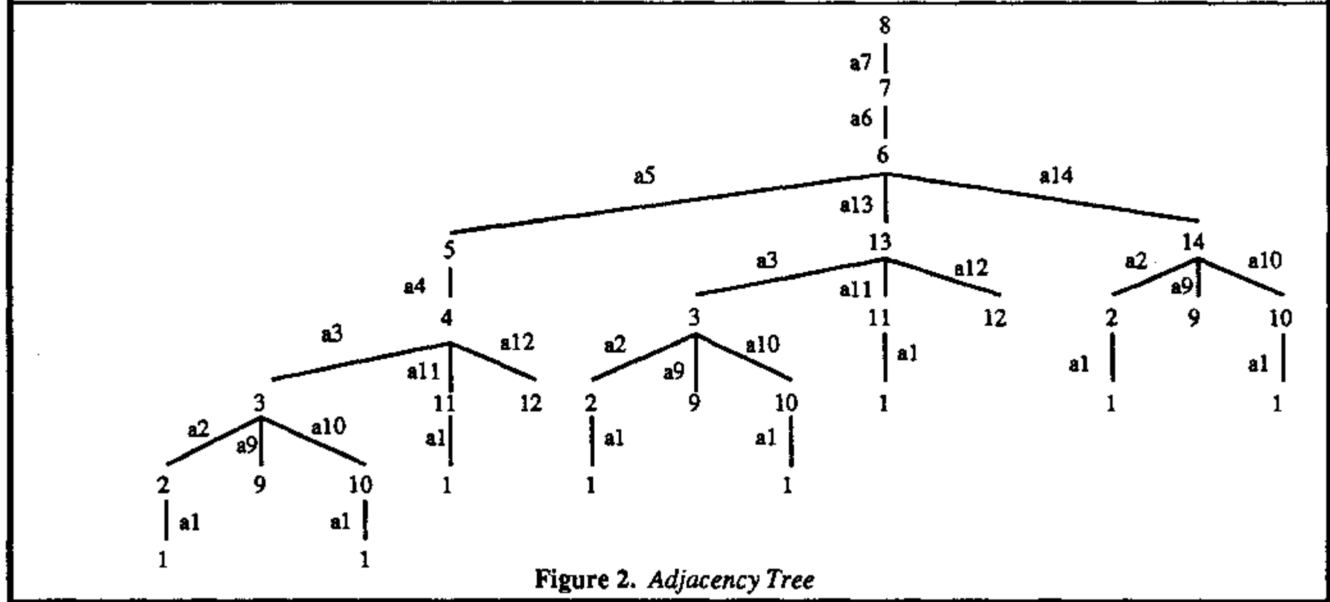

Figure 2. *Adjacency Tree*

Definition 2.7. If $\{s_1,...,s_p\}$ is a set of nodes in the PGS, then their subtrees $T(s_1),...,T(s_p)$ are said to be **adjacent** when $rcl(s_i) = lcl(s_{i+1})-1$ or, alternatively, $al(s_i) = lcl(s_{i+1})$, for $i = 1,...,p-1$.

During a parsing process a great effort is made in finding a set of adjacent subtrees that match a right-hand side of a reduction rule. Let $(A\leftarrow z_1\, z_2\, z_3)$ be a reduction rule, then the parser should start a match process to find all possible sets of adjacent subtrees such that their categories match $z_1\, z_2\, z_3$. The parser scans the input string left-to-right, so reductions grow on the left of the scanner pointer, and for the efficiency of the match process the matcher must start from the last scanned or built node z_3, finding afterwards z_2 and z_1, respectively, sailing in the PGS right-to-left and passing through adjacent subtrees. Steps through adjacent subtrees are easily accomplished by using the sets of the anchored nodes in the terminal nodes. It follows from the above definitions that if $k\in N_N$ then the subtrees adjacent to T(k) are given by an(lcl(k)), whereas if $k\in N_T$ then the adjacent subtrees are given by an(k). The lists of the anchored nodes provide an efficient way to represent the relation of adjacency between nodes. These sets stored only in the terminal nodes provide an efficient data structure useful for the matcher to accomplish its purpose. Figure 1 shows a parse tree at a certain time of a parse, where under each terminal node there is the corresponding list of the anchored nodes. A useful structure that can be derived from these sets is an **adjacency tree**, recursively defined as follows:

Definition 2.8. If (N_T,N_N,T) is a PGS for an input sentence s, and $|s| = n$, then the adjacency tree for the PGS is so built:
- n+1 is the root of the adjacency tree;
- for every $k\in N_T-\{0,1\}\cup N_N$, the sons of k are the nodes in an(lcl(k)) unless an(lcl(k))={0}.

Figure 2 shows the adjacency tree obtained from the partial parse tree in Figure 1. Any passage from a node k to one of its sons h in the adjacency tree represents a passage from a

subtree T(k) to one of its adjacent subtrees T(h) in the PGS. Moreover, during a match process this means that a constituent of the right-hand side has been consumed, and matching the first symbol that match process is finished. The adjacency tree also provides further useful information for optimizing the search during a match. For every node k, if we consider the longest path from k to a leaf, its length is an upper bound for the length of the right hand side still to consume, and since the sons of k are the nodes in an(lcl(k)), the longest path is always given by the sequence of the terminal nodes from the node 1 to the node lcl(k)-1. Thus its length is just lcl(k)-1.

Property 2.1. If (N_T,N_N,T) is a PGS, $(A\leftarrow z_1...z_p)$ is a reduction rule whose right-hand side has to be matched, and $T(k)\in T$ such that $cat(k) = z_p$, then:

a. the string $z_1 \ldots z_p$ is **matchable** iff $p \leq lcl(k)$;

b. for $i = p,\ldots,1$, z_i is **partially matchable** to a node $h\in N_N \cup N_T$ iff $cat(h) = z_i$ and $i \leq lcl(h)$.

Property 2.1. along with the adjacency relation provides a method for an efficient navigation within the PGS among the subtrees. This navigation is performed by the matcher in the PGS as visiting the adjacency tree in a pre-order fashion. It is easy to see that a pre-order visit of the adjacency tree scans all possible sequences of the adjacent subtrees in the PGS, but Property 2.1 provides a shortcut for avoiding useless passages when matchable conditions do not hold. When a match ends the matcher returns one or more sets of nodes satisfying the following conditions:

Definition 2.9. A set RSet = $\{n_1,\ldots,n_p\}$ is a match for a string $z_1\ldots z_p$ iff $cat(n_i) = z_i$, for $i = 1,\ldots,p$, and $T(n_i)$ is adjacent to $T(n_{i+1})$, for $i = 1,\ldots,p-1$. The set RSet is called a **reduction set.**

The adjacency tree shows the hypothetical search space for searching the reduction sets in a PGS, thus it is not a representation of what memory is actually required to store the useful data for such a search. A more suitable representation is an **adjacency directed graph** defined by means of the lists of the anchored nodes in the terminal nodes, and by the pointers to the left corner leaf in the non-terminal nodes.

Definition 2.10. If (N_T,N_N,T) is a PGS, an adjacency digraph can be represented as follows:

a. for any $k\in N_T$, k has outgoing arcs directed to the nodes in an(k);

b. for any $k\in N_N$, k has one outgoing arc directed to lcl(k).

In the classic literature the lists of the anchored nodes are called adjacency lists, and are used for representing graphs (Aho et al., 1974). A graph G=(V,E) can be usually represented by |V| adjacency lists. In our representation we can obtain an optimization representing an adjacency digraph by n adjacency lists, if n is the length of the sentence, and by $|N_N|$ simple pointers for accessing the adjacency lists from the non-terminal nodes, with respect to $n+|N_N|$ adjacency lists for a full representation of an adjacency digraph composed of arcs as in Definition 2.10.a.

Figure 3 shows how a new non-terminal node is connected in an adjacency digraph, and Figure 4 shows the adjacency digraph for the parse tree of Figure 1.

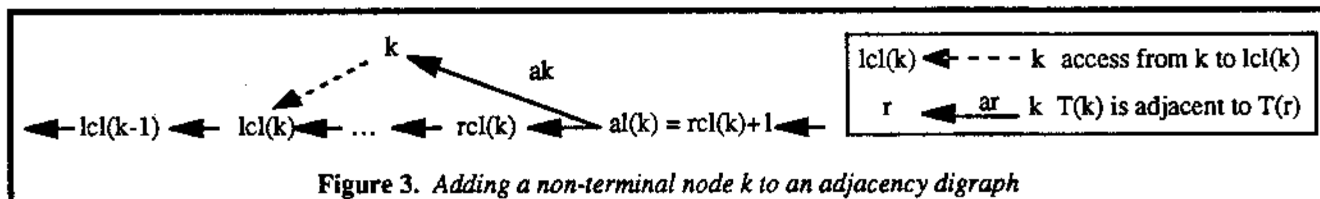

Figure 3. *Adding a non-terminal node k to an adjacency digraph*

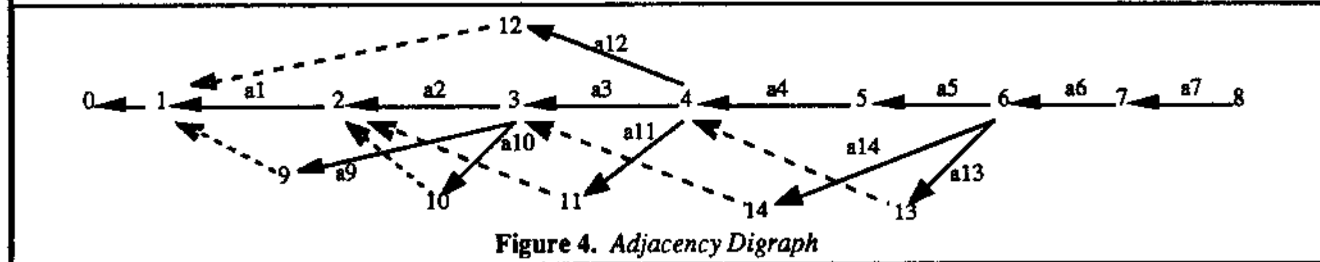

Figure 4. *Adjacency Digraph*

3. PROCESS GRAMMAR

The Process Grammar is an extension of the Augmented Context-Free Grammar such as APSG, oriented to bottom-up parsing. Some relevant features make a Process Grammar quite different from classical APSG.

1. The parser is a PG processor that tries to apply the rules in a bottom-up fashion. It does not have any knowledge about the running grammar but for the necessary structures to access its rules. Furthermore, it sees only its internal state, the Parse Graph Structure, and works with a non-deterministic strategy.

2. The rules are conceived as processes that the PG processor schedules somehow. Any rule defines a reduction rule that does not represent a rewriting rule, but rather a statement for search and construction of new nodes in a bottom-up way within the Parse Graph Structure.

3. The rules are augmented with some sequences of operations to be performed as in the classical APSG. In general, augmentations such as tests and actions concern manipulation of linguistic data at syntactic and/or semantic level. In this paper we are not concerned with this aspect (an

informal description about this is in Marino (1989)), rather we examine some aspects concerning parsing strategies by means of the augmentations.

In a Process Grammar the rules can have knowledge of the existence of other rules and the purpose for which they are defined. They can call some functions that act as filters on the control structures of the parser for the scheduling of the processes, thus altering the state of the processor and forcing alternative applications. This means that any rule has the power of changing the state of the processor requiring different scheduling, and the processor is a blind operator that works following a loose strategy such as the non-deterministic one, whereas the grammar can drive the processor altering its state. In such a way the lack of determinism of the processor can be put in the Process Grammar, implementing parsing strategies which are transparent to the processor.

Definition 3.1. A **Process Grammar** PG is a 6-tuple (V_T,V_N,S,R,V_S,F) where:

- V_T is the set of terminal symbols;
- V_N is the set of non-terminal symbols;
- $S \in V_N$ is the Root Symbol of PG;
- $R = \{r_1,\ldots,r_g\}$ is the set of the rules. Any rule r_i in R is of the form $r_i = \langle red(r_i),st(r_i),t(r_i),a(r_i)\rangle$, where $red(r_i)$ is a reduction rule $(A \leftarrow \alpha)$, $A \in V_N$, $\alpha \in (V_T \cup V_N)^+$; $st(r_i)$ is the state of the rule that can be **active** or **inactive**; $t(r_i)$ and $a(r_i)$ are the tests and the actions, respectively;
- V_S is a set of special symbols that can occur in a reduction rule and have a special meaning. A special symbol is ε_1, a null category that can occur only in the left-hand side of a reduction rule. Therefore, a reduction rule can also have the form $(\varepsilon_1 \leftarrow \alpha)$, and in the following we refer to it as **ε-reduction**;
- $F = \{f_1,\ldots,f_s\}$ is a set of functions the rules can call within their augmentations.

Such a definition extends classical APSG in some specific ways: first, a Process Grammar is suited for bottom-up parsing; second, rules have a state concerning the applicability of a rule at a certain time; third, we extend the CF structure of the reduction rule allowing null left-hand sides by means of ε-reductions; fourth, the set F is the strategic side that should provide the necessary functions to perform operations on the processor structures. As a matter of fact, the set F can be further structured giving the PG a wider complexity and power. In this paper we cannot treat a formal extended definition for F due to space restrictions, but a brief outline can be given. The set F can be defined as $F=F_{Ker} \cup F_{fs}$. In F_{Ker} are all those functions devoted to operations on the processor structures (Kernel Functions), and, in the case of a feature-based system, in F_{fs} are all the functions devoted to the management of feature structures (Marino, 1989). In what follows we are also concerned with the combined use of ε-reductions and the function **RA**, standing for Rule Activation, devoted to the immediate scheduling of a rule. $\mathbf{RA} \in F_{Ker}$ and a call to it means that the specified rule must be applied, involving the scheduling process we describe in Section 4. Before we introduce the PG processor we must give a useful definition:

Definition 3.2. Let $r \in R$ be a rule with $t(r)=[f_{t1};\ldots;f_{tq}]$, $a(r)=[f_{a1};\ldots;f_{ar}]$ be sequences of operations in its augmentations, $f_{t1},\ldots,f_{tq},f_{a1},\ldots,f_{ar} \in F$. Let $\{n_1,\ldots,n_p\}$ be a reduction set for $red(r) = (A \leftarrow z_1\ldots z_p)$, and $h \in N_N$ be the new node for A such that T(h) is the new subtree created in the PGS, then we define the **Process Environment** for t(r) and a(r), denoted briefly by **ProcEnv(r)**, as:

$$\mathrm{ProcEnv}(r) = \{h,n_1,\ldots,n_p\}$$

If red(r) is an ε-reduction then $\mathrm{ProcEnv}(r) = \{n_1,\ldots,n_p\}$.

This definition states the operative range for the augmentations of any rule is limited to the nodes involved by the match of the reduction rule.

4. PG PROCESSOR

Process Scheduler. The process scheduler makes possible the scheduling of the proper rules to run whenever a terminal node is consumed in input or a new non-terminal node is added to the PGS by a process. By *proper rules* we mean all the rules satisfying Property 2.1.a. with respect to the node being scanned or built. These rules are given by the sets defined in the following definition:

Definition 4.1. $\forall c \in V_N \cup V_T$ such that $\exists\, r \in R$ where $red(r) = (A \leftarrow \alpha c)$, $A \in V_N \cup \{\varepsilon_1\}$, being c the right corner of the reduction rule, and $|\alpha c| \leq L$, being L the size of the longest right-hand side having c as the right corner, the sets P(c,i), $P_{\varepsilon 1}(c,i)$ for $i = 1,\ldots,L$, can be built as follows:

$$P(c,i) = \{r \in R \mid red(r)=(A \leftarrow \alpha c),\ 1 \leq |\alpha c| \leq i,\ st(r)=\textbf{active}\}$$
$$P_{\varepsilon 1}(c,i) = \{r \in R \mid red(r)=(\varepsilon_1 \leftarrow \alpha c),\ 1 \leq |\alpha c| \leq i,\ st(r)=\textbf{active}\}$$

Whenever a node $h \in N_T \cup N_N$ has been scanned or built and k=lcl(h), then the process scheduler has to schedule the rules in $P(cat(h),k) \cup P_{\varepsilon 1}(cat(h),k)$. In the following this union is also denoted by $\Pi(cat(h),k)$. Such a rule scheduling allows an efficient realization of the immediate constituent analysis approach within a bottom-up parser by means of a partitioning of the rules in a Process Grammar.

The process scheduler sets up a *process descriptor* for each rule in $\Pi(cat(h),k)$ where the necessary data for applying a process in the proper environment are supplied. In a Process Grammar we can have three main kinds of rules: rules that are activated by others by means of the function **RA**; ε-reduction rules; and standard rules that do not fall in the previous cases. This categorization implies that processes have assigned a priority depending on their kind. Thus activated rules have the highest priority, ε-reduction rules have an intermediate priority and standard rules the lowest priority. Rules become scheduled processes whenever a process descriptor for them is created and inserted in a priority queue by the process scheduler. The priority queue is divided into three stacks, one for each kind of rule, and they form one of the structures of the processor state.

Definition 4.2. A **process descriptor** is a triple PD=[r,h,C] where: $r \in R$ is the rule involved; $h \in N_T \cup N_N \cup \{NIL\}$ is either the right corner node from which the matcher starts or NIL; C is a set of adjacent nodes or the empty set. A process descriptor of the form $[r, NIL, \{n_1,\ldots,n_c\}]$ is built for an activated rule r and pushed in the stack s_1. A process descriptor of the form [r,h,{}] is built for all the other rules and is pushed either in the stack s_2 if r is an ε-reduction rule or in the stack s_3 if a standard rule. Process descriptors of these latter forms are handled by the process scheduler, whereas process descriptors for activated rules are only created and queued by the function **RA**.

State of Computation. The PG processor operates by means of an operation Op on some internal structures that define the processor state ProcState, and on the parsing structures accessible by the process environment ProcEnv. The whole state of computation is therefore given by:

$[Op, ProcState, ProcEnv] = [Op, pt, [s_1, s_2, s_3], PD, pn, RSet]$

where $pt \in N_T$ is the input pointer to the last terminal node scanned; $pn \in N_N$ is the pointer to the last non-terminal node added to the PGS. For a sentence $s=a_1\ldots a_n$ the computation starts from the initial state [**begin**,0,[NIL,NIL,NIL], NIL,n+1,{}], and terminates when the state becomes [**end**,n,[NIL,NIL,NIL],NIL,pn,{}]. The aim of this section is not to give a complete description of the processor cycle in a parsing process, but an analysis of the activation mechanism of the processes by means of two main cases of rule scheduling and processing.

Scheduling and Processing of Standard Rules. Whenever the state of computation becomes as [**scan**, pt, [NIL,NIL,NIL],NIL,pn,{}] the processor scans the next terminal node, performing the following operations:

```
scan:  sc1 if pt = n  then  Op ← end
       sc2            else  pt ← pt + 1;
       sc3                  schedule (Π(cat(pt),lcl(pt)));
       sc4                  Op ← activate.
```

Step sc4 allows the processor to enter in the state where it determines the first non-empty higher priority stack where the process descriptor for the next process to be activated must be popped off. Let suppose that $cat(pt)=z_p$, and $\Pi(z_p, lcl(pt))=\{r\}$ where r is a standard rule such that $red(r)=(A \leftarrow z_1\ldots z_p)$. At this point the state is [**activate**, pt,[NIL,NIL,[r,pt,{}]],NIL,pn,{}] and the processor has to try reduction for the process in the stack s_3, thus Op←**reduce** performing the following statements:

```
reduce: r1  PD←pop (s3);
     [reduce,pt,[NIL,NIL,NIL],[r,pt,{}],pn,{}]
        r2  C←match (red(r), pt);
     C = {n1,...,np-1,pt}
        r3  PD←[r, pt, C];
     [reduce,pt,[NIL,NIL,NIL],[r,pt,C],pn,{}]
        r4  ∀ rset∈ C:
        r5     RSet ←rset;
     [reduce,pt,[NIL,NIL,NIL],[r,pt,{}],pn,RSet]
        r6     if t(r) then   pn←pn + 1;
        r7                    add_subtree(pn,red(r),RSet);
        r8                    a(r);
        r9                    schedule (Π(cat(pn),lcl(pn)));
     [reduce,pt,[NIL,s2,s3],[r,pt,{}],pn,RSet]
        r10 Op←activate.
```

Step r9, where the process scheduler produces process descriptors for all the rules in Π(A,lcl(pn)), implies immediate analysis of the new constituent added to the PGS.

Scheduling and Processing of Rules Activated by ε-Reduction Rules. Let consider the case when an ε-reduction rule r activates an inactive rule r' such that: $red(r)=(\varepsilon_i \leftarrow z_1\ldots z_p)$, a(r)=[**RA** (r')], $red(r')=(A \leftarrow z_k\ldots z_h)$, $1 \le k \le h \le p$, and st(r')=**inactive**. When the operation **activate** has checked that an ε-reduction rule has to be activated then Op←**ε-reduce**, thus the state of computation becomes: [**ε-reduce**,pt,[NIL,[r,m,{}],NIL],NIL,pn,{}], and the following statements are performed:

```
ε-reduce:  εr1  PD←pop (s2);
     [ε-reduce,pt,[NIL,NIL,NIL],[r,m,{}],pn,{}]
           εr2  C←match (red(r), m);
     C = {n1,...,np-1, m}
           εr3  PD←[r,m,C];
     [ε-reduce,pt,[NIL,NIL,NIL],[r,m,C],pn,{}]
           εr4  ∀ rset∈ C:
           εr5     RSet←rset;
     [ε-reduce,pt,[NIL,NIL,NIL],[r,m,{}],pn,RSet]
           εr6     if t(r) then a(r)=[RA (r')];
     [ε-reduce,pt,[[r',NIL,{nk,...,nh}],NIL,NIL],
     [r,m,{}],pn,RSet]
           εr7  Op←activate.
```

In this case, unlike that which the process scheduler does, the function **RA** performs at step εr6 the scheduling of a process descriptor in the stack s_1 where a subset of ProcEnv(r) is passed as the ProcEnv(r'). Therefore, when an ε-reduction rule r activates another rule r' the step εr2 does the work also for r', and **RA** just has to identify the ProcEnv of the activated rule inserting it in the process descriptor. Afterwards, the operation activate checks the highest priority stack s_1 is not empty, therefore it pops the process descriptor $[r', NIL, \{n_k,\ldots,n_h\}]$ and Op←**h-reduce** that skips the match process applying immediately the rule r':

```
h-reduce:  hr1  RSet←C;
     [h-reduce,pt,[NIL,NIL,NIL],[r',NIL,{}],pn,RSet]
           hr2 through hr6 as r6 through r10.
```

From the above descriptions it turns out that the operation **activate** plays a central role for deciding what operation must run next depending on the state of the three stacks. The operation **activate** just has to check whether some process descriptor is in the first non-empty higher priority stack, and afterwards to set the proper operation. The following statements describe such a work and Figure 5 depicts graphically the connections among the operations defined in this Section.

activate: a1 **if** s_1=NIL
a2 **then if** s_2=NIL
a3 **then if** s_3=NIL
a4 **then** Op ← **scan**
a5 **else** Op ← **reduce**
a6 **else** Op ← **ε-reduce**
a7 **else** PD ← pop (s_1);
PD = [r,NIL,C]
a8 Op ← **h-reduce**.

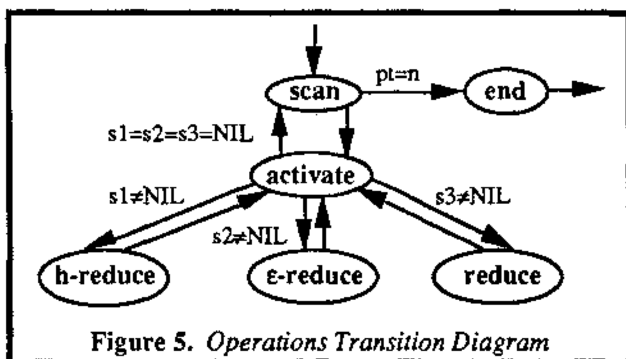

Figure 5. *Operations Transition Diagram*

5. EXAMPLE

It is well known that bottom-up parsers have problems in managing rules with common right-hand sides like X → ABCD, X → BCD, X → CD, X → D, since some or all of these rules can be fired and build unwanted nodes. A strategy called *top-down filtering* in order to circumvent such a problem has been stated, and it is adopted within bottom-up parsers (Kay, 1982; Pratt, 1975; Slocum, 1981; Wirén, 1987) where it simulates a top-down parser together with the bottom-up parser. The PG Processor must face this problem as well, and the example we give is a Process Grammar subset of rules that tries to resolve it. The kind of solution proposed can be put in the family of top-down filters as well, taking advantage firstly of using ε-reduction rules. Unfortunately, the means described so far are still insufficient to solve our problem, thus the following definitions introduce some functions that extend the Process Grammar and the control over the PGS and the PG Processor.

Definition 5.1. Let r be a rule of R with red(r)=(ε_i←$z_1 \ldots z_p$), and RSet={$n_1 \ldots n_p$} be a reduction set for red(r). Taken two nodes $n_i,n_j \in$ RSet where $n_i \in N_N$ such that we have cat(n_i)=z_i, cat(n_j)=z_j, and T(n_i), T(n_j) are adjacent, i.e., either j=i+1 or j=i-1, then the function **Add_Son_Rel** of $F_{K\sigma}$ when called in a(r) as **Add_Son_Rel** (z_i,z_j) has the effect of creating a new parent-son relation between n_i, the parent, and n_j, the son, altering the sets sons(n_i), and either lcl(n_i) or rcl(n_i) as follows:

a) sons(n_i) ← sons(n_i) ∪ {n_j}
b) lcl(n_i) ← lcl(n_j) if j=i-1
c) rcl(n_i) ← rcl(n_j) if j=i+1

Such a function has the power of making an alteration in the structure of a subtree in the PGS extending its coverage to one of its adjacent subtrees.

Definition 5.2. The function **RE** of $F_{K\sigma}$, standing for Rule Enable, when called in the augmentations of some rule r as **RE** (r'), where r, r' are in R, sets the state of r' as **active**, masking the original state set in the definition of r'.

Without entering into greater detail, the function **RE** can have the side effect of scheduling the just enabled rule r' whenever the call to **RE** follows the call **Add_Son_Rel** (X,Y) for some category $X \in V_N, Y \in V_N \cup V_T$, and the right corner of red(r') is X.

Definition 5.3. The function **RD** of $F_{K\sigma}$, standing for Rule Disable, when called in the augmentations of some rule r as **RD** (r'), where r, r' are in R, sets the state of r' as **inactive**, masking the original state set in the definition of r'.

We are now ready to put the problem as follows: given, for instance, the following set P1 of productions:

P1 = {X → ABCD, X → BCD, X → CD, X → D}

we want to define a set of PG rules having the same coverage of the productions in P1 with the feature of building in any case just one node X in the PGS.

Such a set of rules is shown in Figure 6 and its aim is to create links among the node X and the other constituents just when the case occurs and is detected. All the possible cases are depicted in Figure 7 in chronological order of building.

The only active rule is r0 that is fired whenever a D is inserted in the PGS, thus a new node X is created by r0 (case (a)). Since the next possible case is to have a node C adjacent to the node X, the only action of r0 enables the rule r1 whose work is to find such an adjacency in the PGS by means of the ε-reduction rule red(r1)=(ε_1← C X). If such a C exists r1 is scheduled and applied, thus the actions of r1 create a new link between X and C (case (b)), and the rule r2 is enabled in preparation of the third possible case where a node B is adjacent to the node X. The actions of r1 disable r1 itself before ending their work. Because of the side effect of RE cited above the rule r2 is always scheduled, and whenever a node B exists then it is applied. At this point it is clear how the mechanism works and cases (c) and (d) are handled in the same way by the rules r2 and r3, respectively.

As the example shows, whenever the rules r1,r2,r3 are scheduled their task is realized in two phases. The first phase is the match process of the ε-reduction rules. At this stage it is like when a top-down parser searches lower-level constituents for expanding the higher level constituent. If this search succeeds the second phase is when the

red(r0) = (X ← D)
st(r0) = **active**
a(r0) = [**RE** (r1)]

red(r1) = (εl ← C X)
st(r1) = **inactive**
a(r1) = [**Add_Son_Rel** (X,C); **RE** (r2); **RD** (r1)]

red(r2) = (εl ← B X)
st(r2) = **inactive**
a(r2) = [**Add_Son_Rel** (X,B); **RE** (r3); **RD** (r2)]

red(r3) = (εl ← A X)
st(r3) = **inactive**
a(r3) = [**Add_Son_Rel** (X,A); **RD** (r3)]

Figure 6. *The Process Grammar of the example*

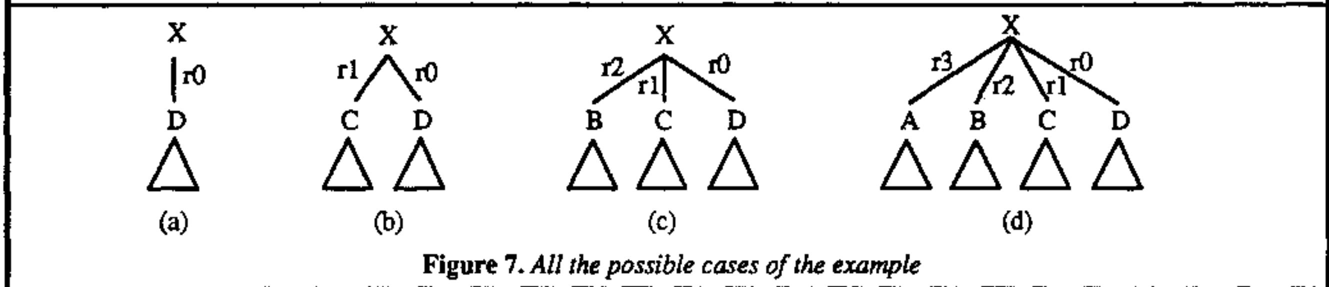

Figure 7. *All the possible cases of the example*

appropriate links are created by means of the actions, and the advantage of this solution is that the search process terminates in a natural way without searching and proposing useless relations between constituents.

We terminate this Section pointing out that this same approach can be used in the dual case of this example, with a set P2 of productions like:

$$P2 = \{X \rightarrow A, X \rightarrow AB, X \rightarrow ABC, X \rightarrow ABCD\}$$

The exercise of finding a corresponding set of PG rules is left to the reader.

6. RELATED WORKS

Some comparisons can be made with related works on three main levels: the data structure PGS; the Process Grammar; the PG Processor.

The PGS can be compared with the chart (Kaplan, 1973; Kay, 1982). The PGS embodies much of the information the chart has. As a matter of fact, our PGS can be seen as a denotational variant of the chart, and it is managed in a different way by the PG Processor since in the PGS we mainly use classical relations between the nodes of the parse-trees: the dominance relation between a parent and a son node, encoded in the non-terminal nodes; the left-adjacency relation between subtrees, encoded in the terminal nodes. Note that if we add the right-adjacency relation to the PGS we obtain a structure fully comparable to the chart.

The Process Grammar can embody many kinds of information. Its structure comes from the general structure stated for the APSG, being very close to the ATN Grammars structure. On the other hand, our approach proposes that grammar rules contain directives relative to the control of the parsing process. This is a feature not in line with the current trend of keeping separate control and linguistic restrictions expressed in a declarative way, and it can be found in parsing systems making use of grammars based on situation-action rules (Winograd, 1983); furthermore, our way of managing grammar rules, i.e., operations on the states, activation and scheduling mechanisms, is very similar to that realized in Marcus (1980).

7. DISCUSSION AND CONCLUSIONS

The PG Processor is bottom-up based, and it has to try to take advantage from all the available sources of information which are just the input sentence and the grammar structure. A strong improvement in the parsing process is determined by how the rules of a Process Grammar are organized. Take, for instance, a grammar where the only active rules are ε-reduction rules. Within the activation model they merely have to activate inactive rules to be needed next, after having determined a proper context for them. This can be extended to chains of activations at different levels of context in a sentence, thus limiting both calls to the matcher and nodes proliferation in the PGS. This case can be represented writing $(\varepsilon_i \leftarrow \alpha\gamma\beta) \Rightarrow (A \leftarrow \gamma)$, reading it as *if the ε-reduction in the lhs applies then activate the rule with the reduction in the rhs*, thus realizing a mechanism that works as a context-sensitive reduction of the form $(\alpha A \beta \leftarrow \alpha\gamma\beta)$, easily extendable also to the general case $(\varepsilon_i \leftarrow \alpha_1\gamma_1\alpha_2 \ldots \alpha_p\gamma_p\alpha_{p+1}) \Rightarrow (A_1 \leftarrow \gamma_1) \ldots (A_p \leftarrow \gamma_p)$.

This is not the only reason for the presence of the ε-reduction rules in the Process Grammar. It also becomes apparent from the example that the ε-reduction rules are a powerful tool that, extending the context-freeness of the reduction rules, allow the realization of a wide alternative of techniques, especially when its use is combined together with Kernel Functions such as **RA** getting a powerful mean for the control of the parsing process. From that, a parser driven by the input - for the main scheduling - and both by the PGS and the rules - for more complex phenomena - can be a valid

framework for solving, as much as possible, classical problems of efficiency such as minimal activation of rules, and minimal node generation. Our description is implementation-independent, it is responsive to improvements and extensions, and a first advantage is that it can be a valid approach for realizing efficient implementations of the PG Processor.

Extending the Process Grammar. In this paper we have described a Process Grammar where rules are augmented with simple tests and actions. An extension of this structure that we have not described here and that can offer further performance to the parsing process is if we introduce in the PG some recovery actions that are applied whenever the detection of one of the two possible cases of process failure happens in either the match process or the tests. Consider, for instance, the reduction rule. Its final aim is to find a process environment for the rule when scheduled. This leads to say that whenever some failure conditions happen and a process environment cannot be provided, the recovery actions would have to manage just the control of what to do next to undertake some recovery task. It is easy to add such an extension to the PG, consequently modifying properly the reduction operations of the PG processor.
Other extensions concern the set F_{Ker}, by adding further control and process management functions. Functions such as **RE** and **RD** can be defined for changing the state of the rules during a parsing process, thus a Process Grammar can be partitioned in *clusters* of rules that can be enabled or disabled under proper circumstances detected by "low-level"(ε-reduction) rules. Finally, there can be also some cutting functions that stop local partial parses, or even halt the PG processor accepting or rejecting the input, e.g., when a fatal condition has been detected making the input unparsable, the PG processor might be halted, thus avoiding the complete parse of the sentence and even starting a recovery process. The reader can refer to Marino (1988) and Marino (1989) for an informal description regarding the implementation of such extensions.

Conclusions. We have presented a complete framework for efficient bottom-up parsing. Efficiency is gained by means of: a structured representation of the parsing structure, the Parse Graph Structure, that allows efficient matching of the reduction rules; the Process Grammar that extends APSG by means of the process-based conception of the grammar rules and by the presence of Kernel Functions; the PG Processor that implements a non-deterministic parser whose behaviour can be altered by the Process Grammar increasing the determinism of the whole system. The mechanism of rule activation that can be realized in a Process Grammar is context-sensitive-based, but this does not increase computational effort since processes involved in the activations receive their process environments - which are computed only once - from the activating rules. At present we cannot tell which degree of determinism can be got, but we infer that the partition of a Process Grammar in clusters of rules, and the driving role the ε-reductions can have are two basic aspects whose importance should be highlighted in the future.

ACKNOWLEDGMENTS

The author is thankful to Giorgio Satta who made helpful comments and corrections on the preliminary draft of this paper.

REFERENCES

Aho, Alfred, V. and Ullman, Jeffrey, D. (1972). *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing.* Prentice Hall, Englewood Cliffs, NJ.

Aho, Alfred, V., Hopcroft, John, E. and Ullman, Jeffrey, D. (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

Grishman, Ralph (1976). A Survey of Syntactic Analysis Procedures for Natural Language. *American Journal of Computational Linguistics.* Microfiche 47, pp. 2-96.

Kaplan, Ronald, M. (1973). A General Syntactic Processor. In Randall Rustin, ed., *Natural Language Processing*, Algorithmics Press, New York, pp. 193-241.

Kay, Martin (1982). Algorithm Schemata and Data Structures in Syntactic Processing. In Barbara J. Grosz, Karen Sparck Jones and Bonnie Lynn Webber, eds., *Readings in Natural Language Processing*, Morgan Kaufmann, Los Altos, pp. 35-70. Also CSL-80-12, Xerox PARC, Palo Alto, California.

Marcus, Mitchell, P. (1980). *A Theory of Syntactic Recognition for Natural Language.* MIT Press, Cambridge, MA.

Marino, Massimo (1988). A Process-Activation Based Parsing Algorithm for the Development of Natural Language Grammars. *Proceedings of 12th International Conference on Computational Linguistics.* Budapest, Hungary, pp. 390-395.

Marino, Massimo (1989). A Framework for the Development of Natural Language Grammars. *Proceedings of International Workshop on Parsing Technologies.* CMU, Pittsburgh, PA, August 28-31 1989, pp. 350-360.

Pratt, Vaughan, R. (1975). LINGOL - A Progress Report. *Proceedings of 4th IJCAI*, Tbilisi, Georgia, USSR, pp. 422-428.

Slocum, Johnathan (1981). A Practical Comparison of Parsing Strategies. *Proceedings of 19th ACL*, Stanford, California, pp. 1-6.

Winograd, Terry (1983). *Language as a Cognitive Process. Vol. 1: Syntax.* Addison-Wesley, Reading, MA.

Wirén, Mats (1987). A Comparison of Rule-Invocation Strategies in Context-Free Chart Parsing. *Proceedings of 3rd Conference of the European Chapter of the ACL*, Copenhagen, Denmark, pp. 226-233.