A Process-Activation Based Parsing Algorithm for the Development of Natural Language Grammars

Massimo MARINO
Department of Linguistics - University of Pisa
Via S. Maria 36 - 56100 Pisa - ITALY
Electronic Mail: MASSIMOM@ICNUCEVM.BITNET

ABSTRACT

A running system, named SAIL, for the development of Natural Language Grammars is described. Stress is put on the particular grammar rule model adopted, named Complex Grammar Units, and on the parsing algorithm that runs rules written in according to this model. Moreover, the parser is like a processor and sees grammar rules as processes which can be activated or inactivated, and can handle exchange of information, structured as messages, among rules for long distance analysis. A brief description of the framework of SAIL a user can interact with, named SIS, is also given. Finally, an example shows that different grammar formalisms can be implemented into the frame of SAIL.

1. INTRODUCTION

Most recent research in the field of grammar formalisms and parsers for natural language has seen the flourishing of various theoretical as well as computational accounts, which, however, bring into consideration the same facts. The most relevant ones seem the following:
- whatever representation is adopted for the structure of the parsed sentence (basically f-structures or trees), it is agreed that (complex) sets of features must describe the linguistic units. It is, therefore, necessary to provide feature handling mechanisms;
- long distance dependency, or, more generally, dependency, requires a specific treatment, which is to be naturally embedded in the theoretical or computational model of syntax, and must be subject to language dependent constraints. In any case, the treatment of dependencies takes the form of a differently constrained search for a referent;
- a certain amount of context-sensitiveness is to be allowed in natural language parsing.

As an additional feature of recent research, the inclination towards the one-to-one correspondence between semantic and syntactic rules has to be mentioned.

SAIL is the parsing algorithm of a development environment, called SAIL Interfacing System, where different grammars corresponding to different grammatical theories can be implemented (/Marino 1988/). Its basic features, which allow full implementation of grammars and their debugging are as follows:
- a rich language for the handling of features;
- grammar rules are seen as processes which can be activated or inactivated, and can exchange messages; this mechanism allows a natural treatment of dependencies and the running of context-sensitive rules;
- the format of the rules is such as to allow semantic processing in parallel with syntactic processing;
- the traditional structure of the parser, a bottom-up all-paths algorithm, allows relative efficiency and the easy integration of a diagnostic component for debugging;
- the development environment is based on different layers of rules, which are processed by the same parser and can handle the external interface, the particular application, and the debugger. This enables the user to modify also the front-end of SAIL, by modifying the corresponding grammar.

2. THE GRAMMAR RULE FORMALISM

The grammar rules are expressed in a formalism called Complex Grammar Units (C.G.U.s) having the following BNF:

CGU ::=	<Syntactic-Rule> <Semantic-Rule>
<Syntactic-Rule> ::=	<Production> <Syn-Tests> <Syn-Actions> <Syn-Recovery-Actions>
<Production> ::=	<LHS> <RHS>
<LHS> ::=	A non-terminal symbol of the grammar
<RHS> ::=	A pattern string of terminal and/or non-terminal symbols
<Syn-Tests> ::=	Some tests on the applicability of the syntactic actions
<Syn-Actions> ::=	Arbitrary syntactic actions
<Syn-Recovery-Actions> ::=	Syntactic actions for the recovery in case of match-fail or test-fail
<Semantic-Rule> ::=	<Sem-Tests> <Sem-Actions> <Sem-Recovery-Actions>
<Sem-Tests> ::=	Some tests on the applicability of the semantic rule
<Sem-Actions> ::=	Arbitrary semantic actions
<Sem-Recovery-Actions> ::=	Semantic actions for the recovery in case of match-fail or test-fail

In each grammar rule the syntactic interpretation is directly connected with the corresponding semantic interpretation: in this way the parser processes in parallel both interpretations.
Inside the augmentations we can do several things:
- the tests are evaluated before the application of a rule and through them we can check its applicability;
- every node of the parsing structure contains structural information about the part of sentence it covers; this information is local to each node and is stored as a feature structure tree. The features are classically stored as attribute-value pairs, with the possibility that a value is itself an attribute-value pair; several feature handling functions are defined inside the system, so we can use them with the augmentations to create, delete, test, get, copy and raise features;
- the semantic rule acts on the semantic part of the system, which can be, for example, a KB handled by a knowledge representation language; this side is dependent on the system application;
- the semantic actions are a sequence of semantic operations, including the possibility of assigning a semantic value to the new node built by the rule. The

semantic value assigned to a node represents, in general, the meaning of the part of the sentence the node covers, according to the chosen formalism;
- the syntactic and semantic recovery actions allow alternative actions if the rule fails during the matching phase or the test checking, so the rules need not be crudely rejected when they fail;
- some built-in system functions are available; these tools handle, for example, the particular execution of a rule, or modify the parsing processing, etc.; these mechanisms are discussed below.

The production in each rule is classically represented as a Context-Free production: $A \rightarrow w_1\ w_2 \ldots w_n$, where A is a non-terminal symbol and $w_1\ w_2 \ldots w_n$ is a string of terminal and/or non-terminal symbols.
The rules of the grammar are applied by the parser in a bottom-up way: it starts from the sentence and builds over it the parsing structure as a graph.

In our system we also have a dictionary D. Each item in the dictionary is called a **form**. We distinguish between **single forms** and **multiple forms**. The first matches the general concept of a word; the second defines a multi-word expression of the language, typically an idiom.
One or more interpretations are associated with each form and they consist of the syntactic category, the semantic value and a feature structure.
A sentence is a compound of forms. For every sentence $f_1\ f_2 \ldots f_n$ such that every form $f_i \in D$, and a grammar G defined in our model, we say that $f_1\ f_2 \ldots f_n$ is parsable if we can build a structure through a finite sequence of rule applications, where at least one node covers the entire sentence and its category is the root symbol of the grammar.

Rule applications are performed by the parser in a bottom-up strategy whenever:
a. at least one sequence of nodes exists in the structure the parser has been building, matching the <RHS> part of the rule;
b. if the above condition holds the parser verifies the tests of the rule; if they are verified the rule is applied;
c. if the match fails, or the tests are not verified, then the parser executes recovery actions.

The core application of a CGU consists in:
d. building a new node corresponding to the <LHS> part of the rule;
e. assigning features to the new node by executing the syntactic actions of the rule;
f. executing the semantic actions of the rule, and possibly assigning the semantic value to the new node.

In the following we always represent the production in the standard way as above; the feature structures associated with a node of category w_i are represented as $[\beta_i]$ and the semantic value as $[[w_i]]$.
We can specify the complete process of a rule application by means of a PASCAL-like statement as follows.

```
if Match ( w_1, ..., w_n, graph )
    /* search one or more sets of nodes matching the <RHS> */
    then if   Syn-Tests( [β_1], ..., [β_n] )  and
              Sem-Tests ( [[w_1]], ..., [[w_n]], SEM )
              /* SEM represents the semantic model */
                then    begin
                        Build (A, w_1, ..., w_n );
                /* build a new node A over the matched nodes */
                        Syn-Actions ( [β], [β_1], ..., [β_n] );
                        Sem-Actions ( [[w_1]], ..., [[w_n]], SEM )
                        end
                else    begin
                        Syn-Recovery-Actions;
                        Sem-Recovery-Actions
                        end
    else    begin
            Syn-Recovery-Actions;
            Sem-Recovery-Actions
            end;
```

The rules are grouped in such a way that the parser accesses to a restricted number of them, i.e. only the currently applicable ones, when it tries to apply some. This is accomplished by partitioning the rules into **packets** discriminated by the last category in the right-band side. If a grammar is partitioned as $P_1,\ldots,P_k$ then for every $i,j=1,\ldots,k$, $i\neq j$, we must have that $P_i \cap P_j = \{\}$. So when the parser accesses a packet through the category of a node, the rules in that packet are the only ones applicable at that moment.

Now let us introduce the concept of **Not Operative Productions.**
In general such productions do not build a new node if one of the three special categories <NOP>, <NOP-ASE>, <NOP-SE> is the left-hand side. A Not Operative Production is one of the following:

{ <NOP> | <NOP-ASE> | <NOP-SE> } $\rightarrow w_1\ w_2 \ldots w_n$

Rules with such productions are called **NOP Rules**.
Depending on the NOP-category used, the rule application is performed in a special way.
A NOP rule with <NOP> as left-hand side is applied as follows if the syntactic tests succeed: 1) no new node is built; 2) only the syntactic rule is taken into account by the parser; 3) the semantic rule is never considered. Therefore the application of such a rule type is performed as in the following PASCAL-like statement:

```
if Match ( w_1, ...,w_n, graph )
    then  if Syn-Tests ( [β_1], ..., [β_n] )
                then      Syn-Actions( [β_1], ..., [β_n] )
                else      Syn-Recovery-Actions
    else  Syn-Recovery-Actions;
```

This kind of NOP rule is useful when we are interested in performing modifications or particular constructions or analyses on features inside a certain context without building a new node. Such a kind of NOP rule is purely syntactic.
In a production with <NOP-ASE> as left-hand side, if both syntactic and semantic tests succeed: 1) no new node is built; 2) the rule application is performed in the standard way, including feature handling if it does not involve the non-existent parent node.
In a production with <NOP-SE> as left-hand side, if only the semantic tests succeed: 1) no new node is built; 2) only the semantic rule is taken into account by the parser; 3) the syntactic rule is never considered. From thereon application is the dual of that defined for the <NOP> category.

3. RULES AS PROCESSES

The rules defined in our system are viewed as processes to be executed by the parser which has the role of the processor. As a consequence, a state is assigned to each rule which is determined at the moment of grammar definition. Rules can assume two different states: **active** or **inactive** state. A rule is **active** when the parser normally takes it into account for application; rules are active when their names are in their corresponding packets. A rule is **inactive** when the parser does not normally take it into consideration for application; rules are inactive when their names are not in any packet.
It is possible to modify the state of a rule by means of two

functions within the augmentations during a rule application.
A rule R_i changes its state from active to inactive if some rule R_j calls within its augmentations the function **rule-disable** for R_i, performing a **disabling** operation; on the termination of the disabling rule R_j, the disabled rule name R_i is removed from the corresponding packet, and the parser does not take into account R_i. Conversely, a rule R_i changes its state from inactive to active if some rule R_j calls the function **rule-enable** for R_i within its augmentations, performing an **enabling** operation. On the termination of the enabling rule R_j, the enabled rule name R_i becomes present into the corresponding packet. It is possible to change the state of one or more rules at a time through these functions and the rules can perform self-enabling and self-disabling operations. Changes of state effected during the parsing are not permanent. At the end of each parse the rules are reconfigured as indicated in their original definition.

In addition we can invoke an inactive rule for just one application from another rule. We say that an inactive rule R_i is **activated** to be applied just once, when a call to the function **rule-activation** is in some augmentation of another rule R_j. The activation of an inactive rule R_i allows just one application of it by the parser, immediately after the termination of the activating rule R_j. The state of the activated rule is not modified. The activation of more than one rule at a time is possible, and once a rule is activated it can activate other rules.

4. CONTEXTUAL RULES

Rule activation by means of the **rule-activation** function, together with NOP rules can be used to handle context sensitive languages. However, this is entirely done by means of CF productions and the augmentations.

A typical CS production is: $\mu_1 A \mu_2 \rightarrow \mu_1 \beta \mu_2$ where μ_1, μ_2, β are strings of symbols, and A is a non-terminal symbol. A bottom-up application of such a production is possible if it happens in two steps: 1) individuation of the context $\mu_1 \beta \mu_2$; the right-hand side must match a sequence of sub-trees that covers $\mu_1 \beta \mu_2$; 2) inside this context we can perform the application of the CF production $A \rightarrow \beta$ building the node A over the sequence of nodes characterized by β. So the complete application for a CS production is made in two steps: the first one concerns context determination, the context being represented by the right-hand side of the CS production; the second step is just the application of a CF production if and only if the first step has determined the context where the CF production is applicable. These considerations allow us to say that: step 1 can be performed by the application of a NOP rule using the NOP-special categories; in fact this kind of rule is useful in determining the context by defining a NOP rule with production:

$$\{ \text{<NOP>} \mid \text{<NOP-SE>} \mid \text{<NOP-ASE>} \} \rightarrow \mu_1 \beta \mu_2$$

Step 2 can be performed by the application of an activated rule; in fact, when the rule at step 1 determines the context it can activate an inactive rule with a production $A \rightarrow \beta$, indicating in the call to **rule-activation** the last node in the sequence β.
Now we can give the definition of **contextual rule**.
We say that a rule is contextual if it is a NOP rule with production:

$$\{ \text{<NOP>} \mid \text{<NOP-SE>} \mid \text{<NOP-ASE>} \} \rightarrow w_1 \dots w_n$$

and inside the augmentations there is a rule activation of <u>at least</u> one inactive rule which has a production:

$$A \rightarrow w_k\, w_{k+1} \dots w_m \qquad 1 \leq k \leq m \leq n$$
$$A \in VN \cup \{\text{<NOP>}, \text{<NOP-SE>}, \text{<NOP-ASE>}\}$$

where VN is the set of the non-terminal symbols of the grammar.
This definition allows a nesting of contextual rules: in fact an activated rule can be a contextual rule itself. In addition, we can activate more than one rule at a time; in this way we can access several contexts inside a main context.

5. MESSAGES

We suggest a method to make possible asynchronous operations, i.e., how two independent rules can interact with each other in order to perform long distance operations. All this is based on the fact that we must be sure that a certain rule will be applied after another and the earlier rule wants to communicate some information to the other one. To this end we have adopted a communication mechanism, that we call **message passing**, which is not based on matching as all the previously explained operations, but on executing two basic tasks: sending and receiving. The sending task is firstly performed by the sending rule that sends a message to a receiving rule; afterwards the receiving rule must perform the receiving task to receive the message. These two tasks are executed by the two rules at two independent times, i.e., when the rules are applied. In the following we denote the sending rule as Rs and the receiving rule as Rr, and we assume they are standard rules: so we denote with SN the node built by Rs and with RN the node built by Rr.
We state two different approaches for what a <u>message</u> is: 1) the rules access a global feature structure where they store global features. Each rule can access this structure and whatever feature value in it; 2) a Message-Box exists where a rule can send a message to another specified rule. The Message-Box is accessible from every rule but the messages are accessible only by receiving rules. A message is composed as follows: a reference to the feature structure of SN: Rs makes available its feature structure to Rr; a sequence of operations, possibly empty, that Rr must execute.
It is not necessary that both these items are present in a message.

In the case of the global feature structure all the rules have access to it. We recall that all the feature structures included in the nodes of the graph are local to their own node. Each rule can store in or get from the global structure features that are global for the sentence: then the messages are feature structures and the same type of operations allowed on the feature structures of the nodes of the graph is possible on this structure.

The Message-Box is a structure referred to by all rules that want to send or receive messages. A rule Rs, building the node SN, sends a message which is automatically inserted in the Message-Box specifying: its name Rs, the receiving rule Rr, a reference to the feature structure of SN which is made available to Rr, a list of operations, possibly empty, to be performed by Rr. Until the messages are sent, they are the exclusive property of Rs. When they are sent Rs loses its property rights, and only the rule Rr specified in the messages is authorized to get them. In addition, Rr finds in the message a reference to a feature structure and this structure is available only to it and always local to its own node.

Message passing, in either of the two realizations, is a way to facilitate the individuation and treatment of existing relations among phrases or parts of them. It is certainly flexible and not expensive because it avoids searches, i.e., matches, inside the graph, and it can be a valid alternative to NOP rules that require a certain number of matches to find particular nodes in the graph.

In fact, if there was not overlapping of the sub-trees rooted in SN and RN, then we can solve relations between SN and RN by applying a proper NOP rule, but, more efficiently, message passing allows us to avoid a certain computational overhead performing proper operations directly in Rs and Rr.

When NOP rules are applied they act upon a structure already built. It is also possible to activate rules that perform further building (contextual rules) and/or featuring operations within a context. This process of activation can be nested many times inside a certain structure. This analysis performs a kind of operation that is virtually directed toward the bottom, in depth. If there was a partial or total overlapping between the sub-trees rooted in SN and RN, then - in this case - when Rs sends a message, assumes that Rr will be applied above its node SN; in this way it is possible to evaluate the consequences of certain operations on a structure which is not yet but it could be built. In this case we act toward the top of the parsing structure, through as many levels as we want. In contrast, using NOP rules, we only act on an existing structure representing deeper levels.

So we can distinguish two ways of operation for long distance analysis among phrases or parts of them: breadth analysis, using both NOP rules or message passing; depth analysis which can be top-down with NOP rules or bottom-up with message passing.

The mechanism of the messages so described is performed through functions that can be used within the augmentations.

6. THE PARSER

Our parser is a CF-based one, derived from the ICA (Immediate Constituent Analysis) algorithm described in /Grishman 1976/, designed to run CGU rules, carrying out the syntactic and semantic analysis in parallel. It is a bottom-up algorithm, and it performs left-to-right scanning and reduction in an immediate constituent analysis. The data structure it works on is a graph where all possible parse trees are connected. The complete parse tree(s) is (are) extracted from the graph in a subsequent step. Therefore, the parser is also able to create structure fragments for ill-formed sentences, thus returning, even in this case, partial analyses. This is particularly useful for diagnosis and debugging.
Parsing termination occurs in a natural way, when no more rule can be applied and the input string is completely scanned.

Before entering the parser a preprocessor scans the sentence from left to right, performs the dictionary look-up for each form in the input string, and returns a structure, the preprocessed sentence, with the syntactic and semantic information taken from the dictionary.

The graph is composed of nodes: the nodes can be either terminals or non-terminals. Terminal nodes are built in correspondence to a scanned form, whereas non-terminal ones are built whenever a rule is applied, obviously the rule must not be a NOP rule.

As stated above the parser is seen as a processor and it sees the rules as processes. It handles a queue of waiting processes/rules to be executed. When the parser takes a packet, for every rule it builds a process descriptor and inserts it in the queue. We call such a process descriptor an **application specification** (AS), while the queue is called the **application specifications list** (ASL). ASs are composed of:
- a node identifier, through this node the parser starts the matching;
- the name of the rule that the parser will apply;
- only in the case of an AS of an activated rule this item is the context where the named activated rule will be applied, i.e. the nodes that matched the right-hand side of the activating rule, otherwise this item is left empty.

ASs in ASL are ordered depending upon the rule involved in an AS. In general, if standard active rules have to be executed, ASL is handled with a LIFO policy. If we consider the case of NOP rules, then these rules must be ordered before the others, since feature modifications they may produce can serve as input to other rules of the same packet, which are applied after them. An inactive rule can be activated just for one application by means of **rule-activation** function: the activated rules must be applied immediately after the end of the activating rule. So this kind of rules has the highest priority of execution with respect to NOP rules and standard active rules. Then **rule-activation** inserts an **activation specification** on the top of ASL for the activated rule. Summarizing, the rules have the following decreasing priority order of execution: 1) activated rules; 2) active NOP rules; 3) standard active rules.

Once a node is created, be it terminal (in correspondence to a scanned form) or non-terminal (in correspondence to a reduction), the parser inserts in the ASL an AS for every rule in the packet corresponding to the category of the new created node: i.e. the new node is the one specified in every inserted AS. The parser performs all possible reductions building more than one node if possible, extracting one AS at a time before analyzing the next one. After an AS is extracted from the ASL, the parser gets the specified rule: the first step is to match the right-hand side on the graph. The nodes matching a right-hand side are searched by the matcher: it returns one or more sets of these nodes, called **reduction sets**. For every reduction set, the application of the current rule is tried. In this way we can connect together all possible parses for a sentence in a unique structure. Termination occurs when the ASL is empty and the preprocessed string is completely scanned. Afterwards the parser returns the graph, from which all parse trees satisfying the following conditions are extracted: a node covers the entire sentence and its category is the root symbol of the grammar. Here is the complete algorithm of the parser:

```
♦ Until the end of the sentence is not reached:
    ♦ Scan a form:
        • Build a new terminal node for the scanned form;
        • For every interpretation of the node:
            ° get the packet corresponding to its category and for every
              rule in the packet insert the AS in the ASL ;
    ♦ For every AS in the ASL:
        • get the first AS from the top of the ASL;
        • get the specified rule in the AS, it is the current rule, and
          access to the node specified in the AS, it is the current node;
        • starting from the current node perform the match on the
          graph using the production of the current rule;
        • if at least one reduction set is found then:
            ° For every reduction set:
                - Apply the current rule;
                - if a new non-terminal node is built then get the
                  corresponding packet to its category and for every rule
                  in it insert the AS in the ASL;
          else: ° Apply recovery actions of the current rule;
```

In this algorithm by **match** we mean the operation of searching the reduction sets and by '**apply** the current rule' we mean the standard rule application starting from the test checking as stated for the CGU model; particular ways of application, e.g. NOP rules, depend on the particular rule definition.

7. AN EXAMPLE

The example concerns a simple fragment of a LFG written in SAIL according to the CGU model. Our example is taken from /Kaplan 1982/and /Winograd 1983/.
The lexical entries for this grammar in SAIL are the following:

a	((Determiner	NIL	(Definiteness) (Indefinite) (Number) (Singular)))
baby	((Noun	NIL	(Number) (Singular) (Predicate) (Baby)))
girl	((Noun	NIL	(Number) (Singular) (Predicate) (Girl)))
handed	((Verb	NIL	(Tense) (Past) (Predicate) (Hand)))
the	((Determiner	NIL	(Definiteness) (Definite)))
toys	((Noun	NIL	(Number) (Plural) (Predicate) (Toys)))

Rules in SAIL are written using a **defrule** format where all the fields appearing in the CGUs can be defined; in addition two fields are devoted to the state definition (STATUS field) and the rule type definition, that is if the rule is a standard rule or a contextual or a NOP rule (CNTXTLORNOPR field). The rules are the following:

```
(defrule NPRule        ; NP → Determiner Noun
  (STATUS active)
  (CNTXTLORNOPR NIL)
  (PRODUCTION (NP (Determiner Noun)))
  (SYN-TESTS T)
  (SEM-TESTS T)
  (SYN-ACTIONS
     (raisef `(* Definiteness Determiner)
               ;raise the values of the specified features from the
               ;son node into the parent node
               `(* * Noun))))
               ; second * means all features of the son node
               ;first * means the storing of the features as they are
               ;in the son node into the parent node

(defrule VPRule  ; VP → Verb NP NP
  (STATUS active)
  (CNTXTLORNOPR NIL)
  (PRODUCTION (VP (Verb NP NP)))
  (SYN-TESTS T)
  (SEM-TESTS T)
  (SYN-ACTIONS
     (raisef  `(* * Verb)      ;all features of the Verb node are
                               ;copied in the parent node
         `((Object Definiteness)     Definiteness   NP) ;1st NP
         `((Object Number)           Number         NP)
         `((Object Predicate)        Predicate      NP)
         `((Object-2 Definiteness)   Definiteness   NP 2) ;2nd NP
         `((Object-2 Number)         Number         NP 2)
         `((Object-2 Predicate)      Predicate      NP 2))))

(defrule TOPRule  ; S →NP VP
  (STATUS active)
  (CNTXTLORNOPR NIL)
  (PRODUCTION (S (NP VP)))
  (SYN-TESTS T)
  (SEM-TESTS T)
  (SYN-ACTIONS
     (raisef  `((Subject Definiteness) Definiteness NP)
              `((Subject Number)       Number       NP)
              `((Subject Predicate)    Predicate    NP)
              `(* * VP)))
  (SEM-ACTIONS
     (put-sem-val  ;stores the EVALuation of the following
                   ;expression as the semantic value of the
                   ;parent node S
        (append (getf-pn `Predicate)
                (getf-pn `(Subject Predicate))
                (getf-pn `(Object Predicate))
                (getf-pn `(Object-2 Predicate))))))
        ; getf-pn gets feature values from the parent node
```

The graph built by the parser applying these rules to the sentence **'a girl handed the baby the toys'** is equivalent to the c-structure built by the corresponding LFG as shown in /Winograd 1983/. The top node S contains the following feature structure:

$$\begin{bmatrix} \text{Subject} : & \begin{bmatrix} \text{Definiteness} & : \text{Indefinite} \\ \text{Number} & : \text{Singular} \\ \text{Predicate} & : \text{Girl} \end{bmatrix} \\ \text{Object} : & \begin{bmatrix} \text{Definiteness} & : \text{Definite} \\ \text{Number} & : \text{Singular} \\ \text{Predicate} & : \text{Baby} \end{bmatrix} \\ \text{Object-2} : & \begin{bmatrix} \text{Definiteness} & : \text{Definite} \\ \text{Number} & : \text{Plural} \\ \text{Predicate} & : \text{Toys} \end{bmatrix} \\ \text{Tense} & : \text{Past} \\ \text{Predicate} & : \text{Hand} \end{bmatrix}$$

with the semantic value: (Hand Girl Baby Toys).
Comparing the solution of the LFG version with the feature structure and the semantic value of the SAIL version we have that the LFG solution is equivalent to the above feature structure plus the semantic value.

8. THE SAIL INTERFACING SYSTEM

The SAIL Interfacing System (S.I.S.) is the framework where a user can interact with SAIL in developing NL applications. In fact SIS is organized in Interface Levels (I.L.s): in SIS we commonly speak of Interface Level Applications (I.L.A.s) which are the association of an IL with a grammar.
If IL-Name is the name of an IL, and G-Name is the name of a grammar which defines a particular language through a dictionary and a set of CGU rules, then the pair <IL-Name, G-Name> defines an ILA inside the SIS: this application is a task performed by that particular IL.
In this way the development environment is based on different layers of rules, which are processed by the same parser and can handle the external interface, the particular application, and any request issued by the user. In fact, the grammar of an ILA defines a language which can be used by the user for sending to the system his requests so that are caught by the parsing system and immediately satisfied.

SIS is structured in 2 main ILs: the Kernel Interface Level (K.I.L.) and the Natural Language IL (N.L.I.L.).
When the system runs only two ILAs are active and available to the user: the KIL, associated to the Kernel Grammar (K.G.) and the Current Running Interface Level (C.R.I.L.). The KIL is always active because it is the core ILA of SIS and its purpose is to handle the overall system, so when the system is started the user is introduced to the Kernel Interface Level. The Kernel Grammar is a semantic grammar associated with the KIL and defines a kernel language of commands and through them the user can use all the functionality of the system such as grammar building, parse checking, running other ILAs.
When SAIL starts up, the KIL is also the CRIL, but when the user wants to load as CRIL another ILA defined in the system, for example a NLIL application, then a KIL command allows this and NLIL becomes the CRIL by

loading a grammar associated to the NLIL: in this way the CRIL is updated to the new application and the loaded grammar becomes the current running grammar.
A subset of KIL commands defines a language through which the user can examine the parsing structures generated by the parser for all the sentences input until that moment. This tool, named ANAPAR (ANAlysis of PARsing), is useful for the grammar and parse checking in developing NL applications.
Finally, we want to point out that the particular structure given to SIS enables the user to modify the front-end to SAIL by modifying the corresponding grammar of the KIL; in fact, all the files involved in their definition are accessible to the user who can modify those files as he wishes, or extend the language by introducing new grammar rules.

CONCLUSIONS

The example has shown the possibility of implementing different grammar formalisms into the frame of SAIL and also the searching of standard procedures for building grammars in the CGU model starting from Categorial Grammars is planned.

An experimental component has also been implemented, which performs some diagnosis of ill-formed input, and confirmed that the chosen parsing algorithm easily supports such a component.

A full evaluation of some of the described mechanisms (such as message passing) has not been carried yet, as application to real linguistic cases has not been designed, but theoretically.

However, a whole view of the system, and the described example show that SAIL is a valuable tool for the development of concrete grammars, even of large coverage.

The whole system described in this paper is currently implemented in Common Lisp and runs on Sun and Orion workstations.

This work has been carried out within the framework of the ESPRIT Project P527 CFID (Communication Failure in Dialogue: Techniques for Detection and Repair).

ACKNOWLEDGMENTS

The author is thankful to Giacomo Ferrari and Irina Prodanof for their helpful support. The author is also grateful to Ronan Reilly who read the draft of the paper.

REFERENCES

/Aho 1972/ Aho, A. E. and Ullman, J. D. (1972). **The theory of parsing, translation and compiling. Vol 1: Parsing**. Prentice-Hall Inc.

/Grishman 1976/ Grishman, R. (1976). A survey of syntactic analysis procedures for natural language. **AJCL**, Microfiches 47, pp. 2-96.

/Kaplan 1982/ Kaplan, R. and Bresnan, J. (1982). Lexical-Functional Grammar: A Formal System for Grammatical Representation. In **The Mental Representation of Grammatical Relations**, Bresnan, J, Ed. Cambridge, MA: MIT Press, pp.173-281.

/Marino 1987a/ Marino, M., Spiezio, A., Ferrari, G. and Prodanof, I. SAIL: a natural language interface for the building of and interacting with knowledge bases. In **Proceedings of the 2nd International Conference on Artificial Intelligence: Methodology, Systems and Applications (AIMSA '86), Varna, Bulgaria, 1986**, Jorrand, P. and Sgurev, V., Eds. North-Holland, 1987, pp. 349-356.

/Marino 1987b/ Marino, M., Spiezio, A., Ferrari, G. and Prodanof, I. An efficient context-free parser for augmented phrase-structure grammars. In **Proceedings of 1987 ACL Europe Conference**, Copenhagen, Denmark, 1987.

/Marino 1988/ Marino, M. (1988). The SAIL Interfacing System: a Framework for the Development of Natural Language Grammars and Applications. **Technical Report DL-NLP-1988-1**, Department of Linguistics, University of Pisa, Italy.

/Robinson 1980/ Robinson, J., J. (1980). Interpreting natural-language utterances in dialogues about tasks. **Techn. Note 210**, SRI International, Menlo Park, CA.

/Robinson 1982/ Robinson, J., J. (1982). DIAGRAM: A grammar for dialogues. **CACM**, 25, 1, pp. 27-47.

/Winograd 1983/ Winograd, T. (1983). **Language as a Cognitive Process. Vol.1: Syntax**. Addison-Wesley.