A Framework for the Development of Natural Language Grammars

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

Abstract

This paper describes a parsing system used in a framework for the development of Natural Language grammars. It is an interactive environment suitable for writing robust NL applications generally. Its heart is the SAIL parsing algorithm that uses a Phrase-Structure Grammar with extensive augmentations. Furthermore, some particular parsing tools are embedded in the system, and provide a powerful environment for developing grammars, even of large coverage.[1]

1. Introduction

Every parsing system should embed a set of tools or mechanisms which should provide an aid in treating a minimum set of linguistic phenomena. Designing SAIL we have mainly taken into account the generality of the parsing system in order to give a wide freedom to the grammar designer, so as to investigate many possible solutions in grammar design in order to adopt the best of them. SAIL (System for the Analysis and Interpretation of Language) is the parsing algorithm of the SAIL Interfacing System (SIS) (/Marino 1988a/, /Marino 1988b/, /Marino 1989/), and just because of its features of generality the design has been driven by some general aspects which derive from various theoretical as well as computational accounts.

1. Whatever representation is adopted for the structure of the parsed sentences, it is agreed that complex sets of syntactic and/or semantic features must describe the linguistic units. Therefore, it is necessary to provide feature handling mechanisms. This point has suggested to us a way of providing a very rich language for handling feature structures (FS in the following). FSs are represented as trees where each arc is labelled by an attribute, and nodes can be pointers to the following alternative paths or a pointer to a leaf node where the value for the path spanned so far is found. They can store many kinds of information thanks to their efficient processing provided by a core set of functions.

2. Some linguistic phenomena encountered in parsing NL, such as long-distance dependency or the ability of treating some context-sensitive cases, led us to see the SAIL grammar rules as processes executed by a processor, a role covered by the parser. The rules of a grammar have associated some information related to their status of processes which are scheduled in a priority queue, according to some their priority of execution (/Knuth 1973/, /Aho et al. 1983/). This also allows, for instance, that the execution of some rule can be requested to perform context-sensitive recognition, or some rules can exchange between each other some information under the form of messages to perform the treatment of long-distance dependency.

3. The parser is structured as a bottom-up (shift reduce) all-paths algorithm, and a formalism for the grammar rules was defined to allow syntactic processing in parallel with semantic processing. The grammar of SAIL is a Phrase-Structure Grammar (PSG) with extensive augmentations, so that we also take advantage from the compositionality principle naturally

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

embedded in bottom-up parsers. As mentioned above, the parser is seen as a processor, thus one of its main tasks is to schedule the processes/rules to run in a priority queue. This queue is not completely under control of the parser since the grammar rules and the dictionary can also issue some specific operations or requests about the management of the scheduling task.

4. The need of a flexible front-end for the user is of primary importance to provide a powerful and complete development environment. The user interface built over SAIL, the SIS, is the framework where a user can interact with the underlying parsing system in developing grammars. This interface provides a set of commands, defined by means of a semantic grammar, that are caught and processed by SAIL and can handle many possible requests of the user.

In the following section we give a brief description of the grammar and dictionary format and how a grammar is defined in SAIL. Section 3 gives an overview of the SAIL parsing system, parser organization, and data structures it uses. Section 4 describes the parsing tools available in the system and their purposes. Finally, section 5 shows just one example of a grammar fragment where some parsing tools described in the previous sections are used.

2. The SAIL Grammar

The Grammar Format

The formalism we adopt to express grammar rules, called Complex Grammar Unit (CGU), defines a syntactic and a semantic side called syntactic rule and semantic rule, respectively. The syntactic rule contains the production, the tests, the actions and the recovery actions. The semantic rule contains the semantic counterpart of the syntactic tests and actions. The presence of the syntactic/semantic recovery actions is a very powerful mean to undertake alternative actions whether the rule fails either matching the right-hand side of the production or checking the syntactic/semantic tests. In this way the rules need not to be crudely rejected when they fail but, for instance, they can activate other rules that could be applied successfully.

A rule in SAIL is written defining all the previous CGU's items. In addition, it is also necessary to provide the status of the rule/process, so that it can be properly taken into account by the parser. The status says whether a rule can be scheduled for application or not by the parser. It can be **active** or **inactive**. Active rules always are scheduled by the parser, whereas inactive rules are not (inactive rules can be seen as sleeping rules). The status plays a central role in the organization of a grammar. As an example, if a rule detects some right or wrong conditions in the parsing structure it can either set active or activate an inactive rule.

Summarizing, a grammar rule is composed of three main items: 1) the status: **active** or **inactive**); 2) the production in context-free (CF) format, in the following denoted by $A \leftarrow w_1 \ldots w_n$, $n \geq 1$, where the left-arrow means that the left-hand side is reduced from the right-hand side according to the bottom-up strategy of parsing; 3) the augmentations.

The production is augmented with an additional item, called the son-flag list. This list says for every category in the right-hand side whether the corresponding node matched in the parsing structure must be considered as a son of the left-hand side or not. If a son-flag is set to + for a right-hand side category the corresponding matched node is a son of the left-hand side node, otherwise it is not a son node if the flag is -. We have two types of production depending on its structure: CF and context-sensitive (CS) productions. CF productions, represented by $A \leftarrow w_1 \ldots w_n$, are defined like:

$$(A \quad (w_1 \ \ldots \ w_n)$$
$$(+ \quad \ldots \ +))$$

where all nodes matched by the right-hand side must be sons of the left-hand side node. CS productions represented by: $c_1 \ldots c_p \, A \, c_{p+1} \ldots c_q \leftarrow c_1 \ldots c_p \, w_1 \ldots w_n \, c_{p+1} \ldots c_q$, $1 \leq p \leq q$, $n \geq 1$, are

defined like:

$$(A \quad (c_1 \ldots c_p w_1 \ldots w_n c_{p+1} \ldots c_q)$$
$$(- \ldots - \ + \ldots + \ - \ \ldots -))$$

where only the nodes with a plus flag inside a context of minus-flagged nodes are sons of the left-hand side node.[2]

The augmentations cover the syntactic and semantic tests and actions of the CGU model. They are the body of a rule and are pieces of Lisp code executed by the parser during the application of the rule. Status, production and augmentations is the information provided by the grammar writer for every rule of a grammar. A rule is a named instance of a complex data structure defined according to the following **defrule** format:

```
(defrule
    :gname                  gname
    :rname                  rname
    :production             <production> [<son-flag-list>]
  [ status                  <status>
    :syn-tests              <code>
    :sem-tests              <code>
    :syn-actions            <code>
    :sem-actions            <code>
    :syn-recovery-actions   <code>
    :sem-recovery-actions   <code>] )
```

gname is the grammar name where the rule *rname* is defined. These two names must be provided in every rule definition since in the SIS we can have more than one grammar available which must be referred to by a name. A grammar usually is defined by a **defgramm** declaration of the form:

(**defgramm** *gname* [*root*])

where *root* is the root category of *gname*. This declaration sets up all data structures for the grammar being defined and must be issued before any rule definition.

The Dictionary Format

Any dictionary of a grammar contains a set of forms that are associated with a set of syntactic and semantic information. A **form** is whatever sequence of words w_1 w_2 ... w_n. When n=1 we have a **single form**, otherwise a **multiple form** (n>1). For any form, be it single or multiple, the first word w_1 is called the **key form**. The key form is the mean for storing and retrieving all information of the whole form in the data structures built by the **defgramm** declaration. Any form has associated three kinds of information, forming an **interpretation**: syntactic category; semantic value; a set of features. A form can have more than one interpretation. In this case, a set of interpretations must be defined supplying as the first item the key form; afterwards, for every sequence of words following the key form, the set of interpretations. An entry of the dictionary is defined according to the

[2]This definition leaves free the user of defining rules with discontinuous constituents in the syntactic representation. Currently the parser does not embed any strategy for a full treatment of these cases since the classical definition of adjacency is implemented. This structure was initially motivated in order to define CS rules by only one rule, and not by two (see Section 4.). Furthermore, such a structure allows a faster search in the parsing structure, performed by the matcher of the production, when, for instance, far constituents must be identified for long-distance tasks. Anyway, stated the important role that can be covered by the representation of discontinuous constituents (see /Bunt at al. 1987/), extension of the parser about this topic can be one of our future tasks.

following format:

(**defentry** *keyform gname*
(**defform** *form*
(**set-int** *:category* <*category*>
[*:semval* <*semval*>
:features <*features*>])$^+$)$^+$)

where *keyform* must be a string of just one word, e.g., "dog", "train", etc.; the *form* must be either the null string "" for the single form *keyform*, or a string of one or more words. Every form definition of this kind is said to be in **defentry** format. <*category*> is the syntactic category and <*semval*> is the semantic value. The features must be provided in the following format:

<*features*> ::= (((<*attributes*>) (<*value*>))$^+$)
<*attributes*> ::= a sequence of feature attributes
<*value*> ::= a value for the feature attributes

As an example:

(((GENDER) (MASC))
((NUMBER) (SING))
((KIND-OF ARG1) (THING)))

Here are some examples of dictionary entries. The most trivial of them is:

(**defentry** "train" my_grammar
(**defform** ""
(**set-int** :category Noun)))

where the single form **train** is defined by one interpretation of category Noun. An example of a single form with two interpretations is the following:

(**defentry** "tree" my_grammar
(**defform** ""
(**set-int** :category Noun
:features (((KIND-OF OBJ) (PLANT))))
(**set-int** :category Noun
:features (((KIND-OF OBJ) (DATA-STRUCTURE))))))

where **tree** is defined as a plant and as a data structure. An example of multiple form is:

(**defentry** "in" my_grammar
(**defform** ""
(**set-int** :category Prep))
(**defform** "the"
(**set-int** :category CompPrep)))

where **in** is defined as a preposition and **in the** as a compound preposition.

The Feature Structures

In the current system we have adopted a data structure that can be at the same time efficient to be processed, homogeneous and reusable in various places of the system. This is why the same data structures are processed at different times in different places of the system. For instance, the lexical information looked-up from the dictionary is stored at parsing time in the terminal nodes of the parsing structure the parser builds. Thus, it is obvious to give the same format to the data in the dictionary and in the nodes of the parsing structure. Feature structures, in their classical definition as sets of attribute-value pairs, are associated with each interpretation of any form in the dictionary and of any node in the parsing structure. FSs are treated as trees, and it is possible to manage structures from the bottom of the parsing structure by means of a specific package of functions,

called Feature Structure Handler (FSH), allowing the main operations on FSs as creation, modification, deletion. Currently, this package contains 12 main operations that can be applied on FSs. Over this set of low level operations on FSs we have developed a set of graph functions accessible by the user, which act on the FSs associated with the nodes of the parsing structure.

Rules with Non-Operative Productions (NOP Rules)

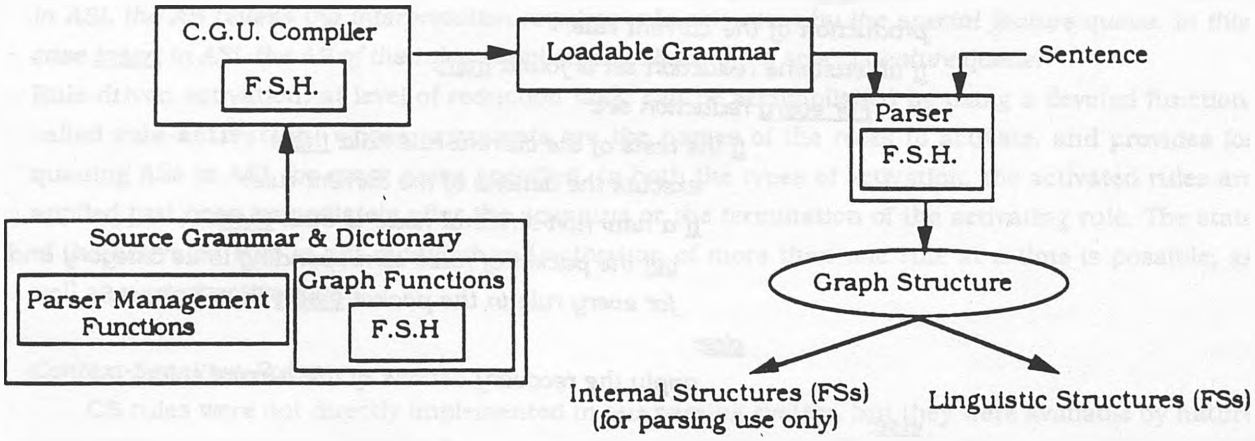
When non-operative productions are defined in some rule they do not build a new node, but can perform various actions, such as activating other rules, or altering semantic structures. There are three types of non-operative productions depending on the NOP category used in the left-hand side:

$$\{ \text{<NOP>} \mid \text{<NOP-ASE>} \mid \text{<NOP-SE>} \} \leftarrow w_1 \ldots w_n$$

If <NOP> is used then only the syntactic rule is applied and the semantic rule is never considered. Only the semantic rule can be applied and the syntactic one is ignored by using the category <NOP-SE>. Finally, both the rules are applied by using the category <NOP-ASE>. As we shall see in Section 4. this kind of production can be useful in CS recognition, providing an alternative way for defining CS rules. Moreover, NOP rules are also useful when it is necessary to control the activation of *real* rules, with the objective of limiting the indeterminism of the parser.

3. Overview of the SAIL Parsing System

In this section we describe briefly the parser, the data structures it handles, and how it works. Starting from the FSH core package, we have adopted this data structure wherever possible inside the parsing system as the figure below shows. The parser builds a parsing structure under the form of a graph, where each node contains two kinds of information: an internal structure of data used by the parsing algorithm only, and the linguistic (syntactic and semantic) information set by the grammar rules. Both these structures are represented in a unique FS managed by the parser and the running grammar by using the underlying FSH functions. Any source grammar must have a set of rules and a set of dictionary forms written in the formats described previously. Grammar rules can make use of two sets of functions: the graph functions, which use the FSH package to update the linguistic structures of the graph, and the parser management functions to handle the various parsing tools and mechanisms (see Section 4.).



The parser is a CF-based one, originally derived from the ICA (Immediate Constituent Analysis) algorithm described in /Grishman 1976/. It is a bottom-up shift-reduce action-based algorithm, performing 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 graph is composed

of nodes that can be terminal or non-terminal. Terminal nodes are built in correspondence to a scanned form, whereas non-terminals are built whenever a rule (other than a NOP rule) is applied.
The parsing system was designed to view the grammar rules as processes to be executed, and the parser as the processor. At any moment, the parser, following a priority schema, handles a queue of processes awaiting execution. In fact we can have different types of rules with different priorities of execution. So it is possible that a rule, when applied, sends a request for execution of another rule inserting the called rule in the appropriate position in the queue. After a scanning or a reduction, the parser gets a set of active rules which are the applicable rules at that moment. When the parser takes such a set - called a packet - for every rule in the packet[3] 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 specification list (ASL). ASs are composed of all the necessary information useful to execute the process on the proper context. ASs in a given 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. The parser performs all possible reductions building more than one node if necessary, extracting one AS at a time before analyzing the next one. After an AS is extracted from ASL the parser searches a match for the right-hand side on the graph. The matching, if successful, returns one or more sets of nodes, called reduction sets. For every reduction set, the application of the rule is tried. In this way we can connect together all possible parses for a sentence in a unique structure. The complete algorithm of the parser is therefore:

Until the end of the sentence is reached:
Scan a form:
build a new terminal node for the scanned form;
For every interpretation of the node:
get the packet of rules corresponding to its category and for every rule in the packet insert in ASL the AS;
For every AS in ASL:
get the first AS from the top of ASL;
get the rule specified in the AS, it is the current rule, and access 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:
if the tests of the current rule hold then:
execute the actions of the current rule;
if a new non-terminal node is built then:
get the packet of rules corresponding to its category and for every rule in the packet insert in ASL the AS;
else:
apply the recovery actions of the current rule;
else:
apply the recovery actions of the current rule;

[3]A packet is a set of active rules. Any grammar is partitioned as a set of packets such that, for every category cat of the grammar, the packet P(cat) is the set of those rules that have cat as the right-most category in their right-hand side. This partitioning is useful for getting the rules applicable at a given moment and it is used by the matcher of the productions.

4. Parsing Tools

Rule Disabling/Enabling Operations

As stated previously, rules can assume two different states, active or inactive. The rule's state is determined at the moment of rule definition. In addition, it is possible to change the state during the parse by using two specific functions. In the application of a rule, others may be changed from active to inactive, performing a disabling operation, or changed from inactive to active, performing an enabling operation. It is possible to change the state of one or more rules at a time and the rules can also perform self-enabling and self-disabling operations. Changes of state effected during a parsing are not permanent. At the end of each parsing the rules are reconfigured as indicated in their original definition.

Dictionary-Driven and Rule-Driven Activation

The mechanism of activation of rules can be used in our parsing system in order to improve the determinism of the parser. We remark that the parsing algorithm is basically a bottom-up parallel **non-deterministic** parser, so that partitioning a grammar as a set of active and inactive rules, and driving their application by an activation mechanism, we can achieve a great control on the parser directly from the grammar, without embedding specific control strategies within the parsing algorithm.

Activation of rules can be effected during the two main phases of the parser activity: scanning and reduction. Dictionary-driven activation can be performed when the parser scans a form defined with an interpretation like the following:

(**set-int**
:category *<category>*
:semval *<semval>*
:features *(((queue) (rule-name^{+}))))*

The special feature *queue* advises the parser of a preference for specific rules to apply when the form is scanned. This preference is independent of the state of the rules specified and the ASs are queued in ASL without considering the packet corresponding to category being scanned. As a consequence of this mechanism of activation, the fifth and sixth line of the parser algorithm must be changed as follows: *get the packet of rules corresponding to its category and for every rule in the packet insert in ASL the AS unless the interpretation requires rule activation by the special feature queue. In this case insert in ASL the AS of the rules supplied as values of the special feature queue.*

Rule-driven activation, at level of reduction task, can be accomplished by using a devoted function, called **rule-activation**, whose arguments are the names of the rules to activate, and provides for queuing ASs in ASL for every name specified. In both the types of activation, the activated rules are applied just once immediately after the scanning or the termination of the activating rule. The state of the activated rule is not modified and activation of more than one rule at a time is possible, as well as nested activations.

Context-Sensitive Rules

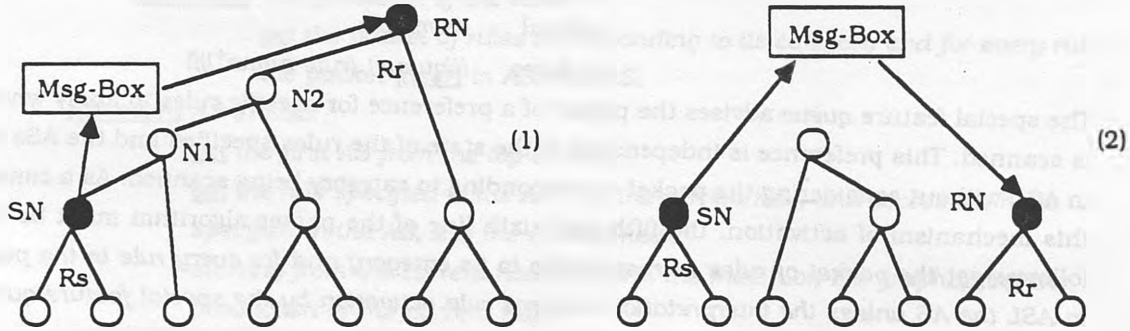
CS rules were not directly implemented in our parsing system, but they were available by nature (in addition to the way currently defined in Section 2.) thanks to the rule-activation mechanism and NOP rules. The complete application for a CS production $\alpha A\beta \leftarrow \alpha\gamma\beta$ is made in two steps. The first one concerns a context determination, the context being represented by the right-hand side of the CS production, $\alpha\gamma\beta$. The second one is just an application of the CF production $A \leftarrow \gamma$, if and only if the first step has determined the context where the CF production is applicable. This can be easily

accomplished by defining a NOP rule for the context determination as first step. Afterwards, this NOP rule must activate the CF rule as second step, building the node A in the proper context.

Message Passing

The message passing mechanism is a parsing tool that makes possible asynchronous operations on linguistic data. This way of processing implies the co-operation between two rules which interact with each other exchanging some information by means of a sending and a receiving task performed at the two independent times of rule application. The sending task is performed by the sending rule at a time T_1, sending a message for another rule. This latter rule must perform the receiving task to receive the message at its execution time T_2, (T_2>T_1). Since the relevant linguistic data the parser works on are stored as FSs, the messages are FSs. We have implemented two approaches of message passing. The first one makes use of a global FS where any rule can store global features. Any rule during a parse can access this global FS and whatever feature value. This type of FS is the global counterpart of the FS stored in every node of the graph structure: the FS of a node is local and can only be accessed by the nodes linked to its node by a direct connection link. Therefore, there being no right of privacy on features in the global FS, this particular structure must be accessed with care by the rules since it can be a place of conflicts among them.

The second approach provides a structure that preserves the right of privacy of the messages. Also in this case the messages are FSs, and are stored in a sort of mailbox, called message-box. Any rule can refer to the message-box to store a message, specifying the destination rule. On the other side, any rule can refer to the message-box to get messages, and only the messages addressed to it will be available. Let us consider the two cases shown in the following partial parse-trees.



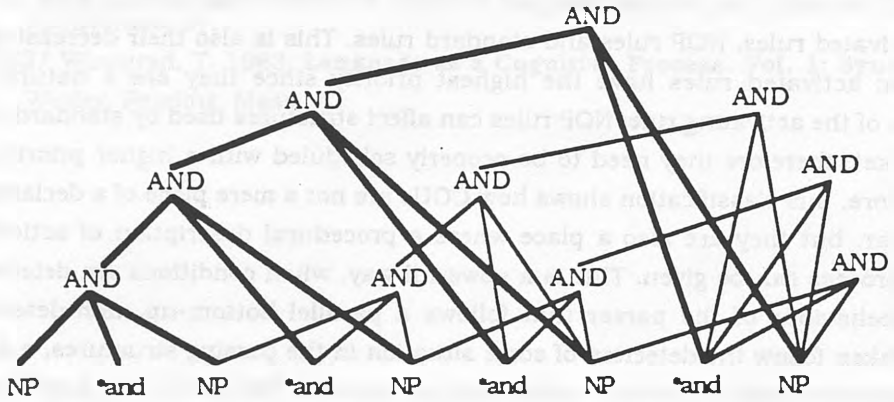
We suppose some information, created or raised in the node SN from the terminal side by the rule Rs, must be used in the node RN built by the rule Rr. (1) shows that the message-box could be used bypassing the nodes N1,N2. This is useful when (some) data from SN are not relevant for processing in N1 and N2, gaining the advantage that no memory space is wasted using the nodes N1,N2 for raising the data from SN to RN. On the other side, (2) shows a case where no path exists between SN and RN. Therefore, the only connection between the nodes can be a common structure accessed by them. The use of the message-box is very easy since all the work is done by two functions. The function **sendmsg** makes a copy of a subset of the FSs of the nodes it can access (i.e., the nodes corresponding to the left- and right-hand side of the production) and stores it in the message-box. The function **receivemsg** gets a message under the form of FS and stores it in the node corresponding to the left-hand side of the production.

All the functions: **sendmsg**, **receivemsg**, and those for handling the global FS are implemented using the FSH package.

5. An Example: SAILing X and X and ... X

The example shows a fragment of a grammar whose aim is to drive the parser according to a specific strategy of recognition achieving as result an optimized parsing structure, i.e., the minimum number of nodes strictly necessary is built.

The recognition of indefinitely long clauses of the form X **and** X **and** ... X could be achieved by using the productions: AND ← NP *and NP, AND ← AND *and NP, where, for instance, X can be an NP and *and is the category of **and**. These productions produce a parsing structure of the kind shown below. Being k the number of conjunctions, the number of the nodes N(k) built by these two productions is given by: N(k) = TN(k) + NTN(k), TN(k) = 2k + 1, NTN(k) = (1/2)k(k + 1).



TN(k) determines the number of the terminal nodes, and NTN(k) the number of the non-terminal nodes. For the graph above N(4) = 19, since TN(4) = 9 and NTN(4) = 10. This kind of parsing structure is not optimized, besides N(k) is a quadratic function of k. In the figure above we have drawn in boldface lines the parsing structure with the minimum number of nodes we want. For this optimized structure NTN(k) is a linear function of k: NTN(k) = k. Therefore, the formula for the optimized case $N_O(k)$ is: $N_O(k) = 3k + 1$.

Our grammar fragment is based on a watch-rule, called Check-and-rule, that checks whether the parser has already built a node of category AND followed by *and NP. This rule has the production: <NOP> ← AND *and NP, and if its right-hand side has no match it means that the first node AND has to be built. Check-and-rule has the following definition.

```
(defrule
    :gname                  my_grammar
    :rname                  Check-and-rule
    :production             (<NOP> (AND *and NP))
    :status                 active
    :syn-actions            (rule-activation '(Make-and-rule NP))
    :syn-recovery-actions   (rule-activation '(Make-first-and-rule NP)))
```

The syn-actions are applied if the right-hand side has a match and the rule Make-and-rule is activated to build a non-terminal node AND. The syn-recovery-actions are applied when the parser has to build for the first time a node AND, and the rule Make-first-and-rule is activated. These two activated rules must be inactive since the watch-rule has the work of activating them.

```
(defrule
    :gname          my_grammar
    :rname          Make-first-and-rule
    :production     (AND (NP *and NP))
    :status         inactive)
```

```
(defrule
        :gname          my_grammar
        :rname          Make-and-rule
        :production     (AND (AND *and NP))
        :status         inactive)
```

6. Final Remarks

Some remarks about the parsing system and the parsing tools described so far are in order. A first point concerns the priority assigned to rules. It is clear that we can have three main kinds of rules: activated rules, NOP rules and standard rules. This is also their decreasing priority order of execution: activated rules have the highest priority since they are a natural completion and extension of the activating rule; NOP rules can affect structures used by standard (non-NOP) rules in their packet, therefore they need to be properly scheduled with a higher priority than the others. Furthermore, this classification shows how CGUs are not a mere place of a declarative description of a grammar, but they are also a place where a procedural description of actions concerning the parsing process can be given. This is a powerful way, when conditions are detected, of altering the natural behaviour of the parser that follows a parallel bottom-up, non-deterministic strategy. Actions taken follow the detection of some situation in the parsing structures, e.g., the activation of a rule instead of another when a misspelling is found in the input, and a parsing process can be driven by a grammar where only the necessary rules for context detection are set active and those devoted to build structures inactive. This way of setting control of the parser places this parsing system in the category of situation-action parsers (/Winograd 1983/).

Unfortunately this paper cannot be a place for a wide description of examples of grammars using the parsing tools of SAIL. Some running examples, as well as that described above, can be found in /Marino 1988a/. Moreover, some ill-formed input cases have been faced, e.g., lexical/syntactic ill-formedness, constraint violation, constituent shuffling, missing constituents, in /Ferrari 1989/. A wide report of the work developed in the framework of the European ESPRIT Project P527 CFID using the SAIL Interfacing System is in /Deliverable 9/ where, among other things, the description of an English grammar and semantics is shown (/Mac Aogain et al. 1989/).

The author is thankful to Giacomo Ferrari who made possible this work.

References

/Aho et al. 1983/ Aho, A., V., Hopcroft, J., E. and Ullman, J., D. 1983. **Data Structures and Algorithms**. Addison-Wesley, Reading, Mass.

/Bunt at al. 1987/ Bunt, H., Thesingh, J. and van der Sloot, K. 1987. Discontinuous Constituents in Trees, Rules, and Parsing. **Proceedings of the 3rd Conference of the European Chapter of the ACL**. Copenhagen, Denmark, pp. 203-210.

/Deliverable 9/ **Deliverable 9: Implementation of Dialogue System**. 1989. Ref. CFID.D9.2. ESPRIT Project 527 (CFID).

/Ferrari 1989/ Ferrari, G. 1989. **The Treatment of Ill-Formed Input within the Frame of SAIL**. Working Paper. ESPRIT Project 527 (CFID).

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

/Knuth 1973/ Knuth, D., E. 1973. **The Art of Computer Programming. Vol.III: Sorting and Searching**. Addison-Wesley, Reading, Mass.

/Mac Aogain et al. 1989/ Mac Aogain, E. and Harper, J. 1989. Semantics and Grammar. In

/Deliverable 9/.

/Marino 1988a/ Marino, M. 1988. The SAIL Interfacing System: A Framework for the Development of Natural Language Grammars and Applications. **Technical Report DL-NLP-88-1**. Department of Linguistics. University of Pisa.

/Marino 1988b/ Marino, M. 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 1989/ Marino, M. 1989. SAIL: A Prototype Environment for Writing NL Applications. In /Deliverable 9/.

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