SeSynPro
Towards a Workbench for Semantic Syntax

Henk Schotel

Nijmegen University (KUN)
Dpt. of Philosophy of Language
Postbus 9103
NL-6500 HK Nijmegen
The Netherlands

hschotel@vms.uci.kun.nl
Tel.:++ 31 80 612949/774940;
Fax: ++ 31 80 612200

Keywords: Semantic Syntax, Logic Grammar, Sentence Generation, Graphic Userinterface

Abstract

Semantic Syntax (SeSyn) is a grammatical theory still under development by Pieter Seuren. It describes how logic-based tree structures, called Semantic Analyses (SAs) are mapped onto the surface structure of sentences by a language specific Generator in a two stage process: the cycle (which by applying transformations turns an SA into a shallow structure) and the post cycle (which converts the shallow structure into a surface structure). SeSynPro is the first attempt to implement Semantic Syntax (in Prolog). To implement the formation rules according to which the SAs have to be formed, a logic grammar (SALG, "SA Logic Grammar") is introduced that plays a role in several parts of the implementation. Pattern matching techniques are used to implement the transformations and the postcyclic operations on trees executed by the generator. SeSynPro makes extensive use of the graphic interface possibilities of the Apple Macintosh to make it as linguist friendly as possible. Crucial for this is the DrawTree module that draws trees in accordance with the conventions Seuren uses in his papers. Another feature is a graphic-oriented, formation rule driven SA-editor, that enables the user to draw SAs to be input to the generator, thus freeing him from specifying complicated input lists. Other graphic oriented editors will be added in the future (e.g. for the Formation Rules and the Transformations).

Contents:

1. Semantic Syntax intro

Semantic Syntax is the name Seuren in the early 1970's gave to what was then generally called 'Generative Semantics' (McCawley, 1974). Although Generative Semantics fell on hard times, caused by the linguistics adhering to the Autonomous Syntax type of transformational theories, he continued working along the lines of Semantic Syntax, whose most prominent feature is the coinciding of the deep structure and the semantic representation, (Seuren, 1993).

Semantic Syntax tries to unravel the structure of natural languages, taking into full account semantic as well as syntactic phenomena.

In what follows I shall sketch(!) the theory in the way it has been implemented, thus introducing the workbench and the grammatical theory more or less in parallel.

Nowadays the deep structure is called the Semantic Analysis of a sentence. The general outline of the theory is quite simple (see the Figure 1.): A Semantic Analysis (SA) has a structure which follows a set of (NL specific) formation rules. In the *Cycle* transformation rules convert the SA into a *Shallow Structure* (ShS), which in its turn is changed by the *Postcycle* into a *Surface Structure*. In what follows, I will focus on the design of the workbench, but not after having specified what the aims of our implementation efforts are and were.

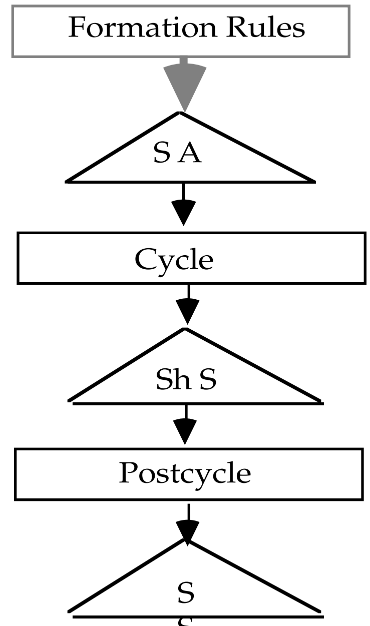


Figure 1.
Outline of Semantic Syntax

2. Aims of the implementation.

Although the basic scheme of the theory is very simple, the details required to make Semantic Syntactic descriptions fit the complexities of a natural language made it worthwhile to find ways to help the research process, which until very recently was done without the use of computers, other than for text processing. The way we work (probably like most readers do) is according to what one could call the *computational modelling research scheme* depicted below (Figure 2). In our research the observations (the fishes in the picture), consist of natural language sentences, the theory is SeSyn and the model developed is the workbench (SeSynPro).

The implementional work resulted in the discovery of inconsistencies and lack of detail (by HS), which were fed back (to PS) to improve the theory. This is an on going process, no details of this were recorded, but particularly in the post cycle there were several occasions where the computational modelling research scheme payed off. The workbench will facilitate the application of the descriptive methods of Semantic Syntax to other languages than the ones from which it was induced (English, Dutch, French, German, Finnish, Sranan Tongo and Mauretian Creole). Such descriptive efforts will undoubtedly lead to the discovery of other shortcomings and give rise to further improvements of the theory. So the workbench can be considered as a tool resulting from a linguistic theory, leading to a faster development of that theory. Another aim of the workbench is to help spread the knowledge about the complex theory that Semantic Syntax is: Experimenting with it on a computer (Apple Macintosh) by linguists and students will greatly facilitate the understanding of it. Of course the Semantic Syntax undertaking as a whole has the potential to contribute to the solution of all kinds of natural language processing problems, including NL front-ends and machine translation (refer to Seuren's paper in this volume for the latter).

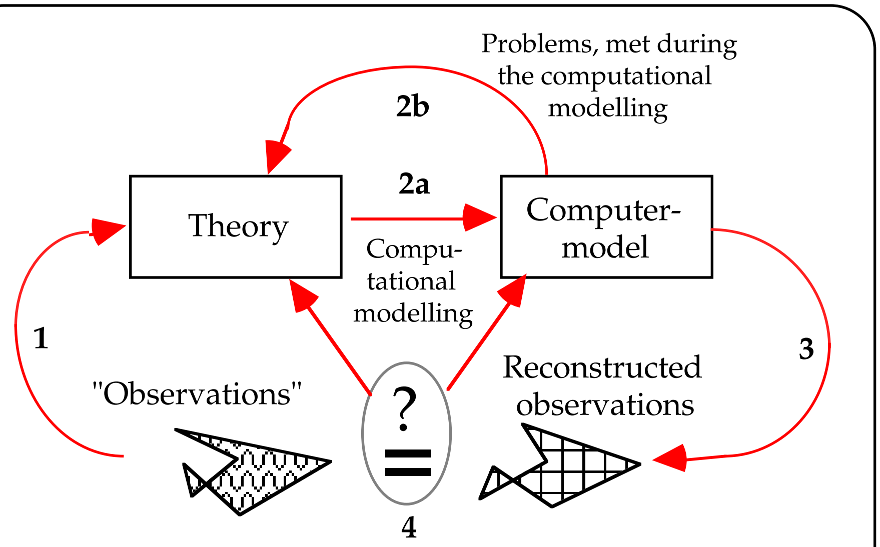


Figure 2

The computational modelling research scheme

1. Theory construction to explain observations
2a. Development of computational model
2b. Theory improvement based on problems, encountered during 2a.
3. The computer model produces output
4. Comparison of output and original observations leads to further improvements of the theory and/or of the computer model (if necessary).

3. Linguistic aspects of the workbench

Although Semantic Syntax at the present stage of its development (with the workbench following closely behind) can handle sentences like:

"Tom seems to be ill"
"It is likely that Tom is ill"
"I expect it to be likely that tom is ill"
"John doesn't always tell the truth"
"Won't the cat eat the mouse in the house?"

or in Dutch:

"Jan heeft de hond de krant weg laten halen"
John has the dog the paper away let take
(John had the dog take the paper away")
"Jan heeft de hond geleerd de krant weg te nemen"
John has the dog taught the paper away to take
("John taught the dog to take away the paper")

or in French:

"Jean n' aurait toujours pas dû te dénoncer à moi si vite"
"Jean not have-past-future still not must$_{PastP}$ you denounce to me so quickly"
("Jean should still not have denounced you to me so quickly")

and many, many more structurally different ones, in this paper we will just be able to illustrate SAs and the generation process by using the sentence "Tom hits the ball" as an example.

3.1 The input: the Semantic Analysis (SA)

An SA represents the meaning of a sentence (in most cases the mapping is 1-1). and consists of a recursive S-structure, in which each S level stands for a predicate meaning (wich carries no tense information) is represented in Semantic Syntax as a tree of which the top is an S-node, below of which are three daughters: the first one represents the predicate (`hit`), and the other two the arguments (`Tom` and the `ball`). Refer to Figure 3. (Quantifiers have not been implemented yet and will not be discussed in this paper. That is why the noun phrase 'the ball' remains unanalysed here.)

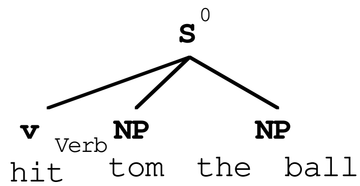
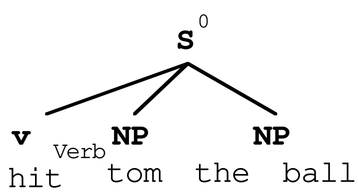


Figure 3. Semantic nucleus of 'Tom hits the ball' in SeSyn

More generallly, the leftmost daughter node of an S stands for the verb and functions as its predicate, the other nodes represent the Subject, the Indirect Object and the Direct Object in this order. Which of the arguments are present in an SA depends on the "frame" of the verb. The frame is found in the lexicon and specifies the argument structure of a verb and restrictions on the types of the arguments. (More about Semantic Analyses and the formation rules to which they must comply can be found below and in Seuren 's paper.)

As promised, we will discuss SeSyn in parallel with SeSynPro, so this is the place to tell you that a tree is represented as *a list*. The list representation for trees was chosen to minimize the number of atoms required and because it allows for the application of pattern-matching techniques to implement the transformations, as will be discussed in 3.5 below.
The first element in the list represents the top of the tree (the mother), and each of the elements in the tail of the list represents a daughter node (which itself can also be a list, representing a *subtree*). An example of a tree and its list representation can be seen in figure 4 below.




```
[s(0,0,'',[]), [v-verb, hit],
               [np, tom],
               [np, the, ball]]
```

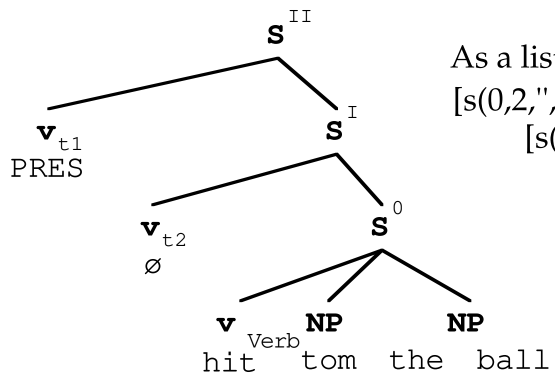
Figure 4.: The nucleus and its Prolog equivalent of 'Tom hits the ball'

S-nodes have a maximum of two subscripts (*sentence unit* (clause) and a *feature list)* and two superscripts (*Tense information* and the *class* to which the predicate verb belongs):

$$\mathbf{S}^{\mathbf{2}:\text{Tense};\ \mathbf{3}:\text{Class}}_{\mathbf{1}:\text{S-Unit};\ \mathbf{4}:\text{Features}}$$

The sub- and superscripts play a role in the transformational process, as will be explained, where necessary, later on. (Also refer to Seuren's paper). In Prolog the s-nodes are represented bij an s-functor, with four arguments, the order of which is indicated with the numbers (1-4) in the scheme above. In other terms, like v-nodes a *superscript* is represented by the **^** operator, a *subscript* by means of a minus-sign. (Look in the text next to the trees above and below for examples like `v-verb` and `v-t1`)

Tenses in English SAs are realized by putting two other Ss, each with a (*non-lexical*) predicate ('v_{t1}' and 'v_{t2}'), above the lexical nucleus. Both of these non -lexical predicates only have two possible realizations. (PRES or PAST and `have` or Ø respectively). Suffice it to remark that the combination of PRES and Ø results in the present tense in English, which makes the complete SA for "Tom hits the ball" as depicted in figure 5:



As a list:
[s(0,2,'',[]), [v-t1, 'PRES'],
[s(0,1,'',[]), [v-t2, ø],
[s(0,0,'',[]), [v-verb,hit],
[np, tom],
[np, the, ball]]]]

Figure 5. The complete SA for 'Tom hits the ball', as a tree and as a Prolog list.

3.2 The SA Formation Rules and their implementation: SALG

The SAs of an NL follow a set of rewrite rules, called the Formation Rules. In Prolog these rules are implemented using the SA Logic Grammar (SALG). As an example a toy set of formation rules to which our English example SA complies is specified using SALG:

```
/* 1 */ s(0,2,'',[]) -=> v-t1, s(0,1,'',[]).
/* 2 */ s(0,1,'',[]) -=> v-t2, s(0,1,pass,[]) or s(0,0,'',[]).
/* 3 */ s(0,0,'',[]) -=> v-verb ++ frame.

v-t1  -=> [T1], {verb(t1,_,T1,_)}. % The open places are for surface
v-t2  -=> [T2], {verb(t2,_,T2,_)}. % category & transformations

np -=> [Prop],        {proper_name(Prop)}.
np -=> [Pronoun],     {pronoun(Pronoun)}.
np -=> [Det],[Noun],  {det(Det),noun(Noun)}.

% Toy Lexicon:

verb(t1,'Aff'-'AH->FV', V_t1 ,_,['L(v)']):-
                                        member(V_t1,['PRES','PAST']).

verb(t2,['V',Ø], '0' ,_,['L(v)']).
verb(t2,['V','have'], 'have' ,_,['PaP','L(v)']).

verb(verb,V):- member(V,[hit,sleep]).
%-----------------------------------------------------
frame(swim,np).            % Frame: A Subject NP only
frame(hit,(np,np)).        % Frame: Subject & Object NP
%-----------------------------------------------------
proper_name(Prop):- member(Prop,[tom,heleen]).
pronoun(Pronoun) :- member(Pronoun,[he,we]).
det(Det)         :- member(Det, [the,a]).
noun(N)          :- member(N,   [ball,man,canal]).
```

A term to the left of the arrow (`-=>`) can be rewritten into the terms at the right. '`or`' indicates alternatives: unlike the semicolon (;) this operator has a precedence higher than that of the comma, so (`a, b or c`), must be read as (`a, (b or c)`). In this example the starting symbol is the S-term s(0,2,'',[]). As is usually the case with logic grammars, (non)terminals in a SALG can have additional parameters. The SALG rules are translated into Prolog clauses, that together form an acceptor that can be used to check whether a list structure complies to the rules for an SA. As a translation example we show the call to the translation predicate that compiles the first rule:

```
s([s(0, 2, '', []), [v-t1| A], [s(0, 1, '', [])|B]]) :-
  v([v-t1| A]),
  s([s(0, 1, '', [])|B]).
```

Prolog code not to be translated is specified between braces {}. A special (meta) feature of this grammar fromalism is the ++ operator, followed by the reserved word '`frame`' that is used to indicate that a part of the right hand side of a rule depends on the *frame* of the lexical verb chosen and this has to be retrieved from the lexicon. This retrieved frame has the same format as the right-hand side of a normal SALG rule, but has to be available at parse-time, not at compile-time. For example, the translation of the third rule:

```
:- translate( s(0,0,'',[]) -=> v-verb ++ frame ), Clause ).
```

produces the clause:

```
s([s(0, 0, '', []), [v-verb, A]|B]) :-
  v([v-verb, A]),
  frame(A, C),
  translate_right(C, D, B),
  call(D).
```

In this clause we see the that the translator produced a call to a part of its own (translate_right) that translates the right-hand side of rules. But compilation of the frame will take place only when this clause figures in the checking of an SA. After that the translated frame will be called. For example the frame for the verb 'hit', will be translated into two calls to the predicate **np**:

```
:-translate_right((np,np),Goal,_).
Goal = (np([np|A]), np([np|B]))
```

Terms that are no formation rule, like in the lexicon above, are not changed by the compiler.

After compiling all of the rules, it can be checked, that the SA of figure 5 is in accordance with the formation rules given above, by calling the one argument predicate **s** that resulted from the translation of rule 1:

```
?-s([s(0,2,'',[]), [v-t1, 'PRES'],
                   [s(0,1,'',[]), [v-t2, Ø],
                   [s(0,0,'',[]), [v-verb,hit],
                                  [np,tom],
                                  [np,the,ball]]]]]).
yes
```

(The SA follows rules 1, 2 and 3 in that order. The frame of the verb 'hit' specified that the structure of the deepest S (headed by `s(0,0,-,-)`) in which two arguments occur, is formed properly).

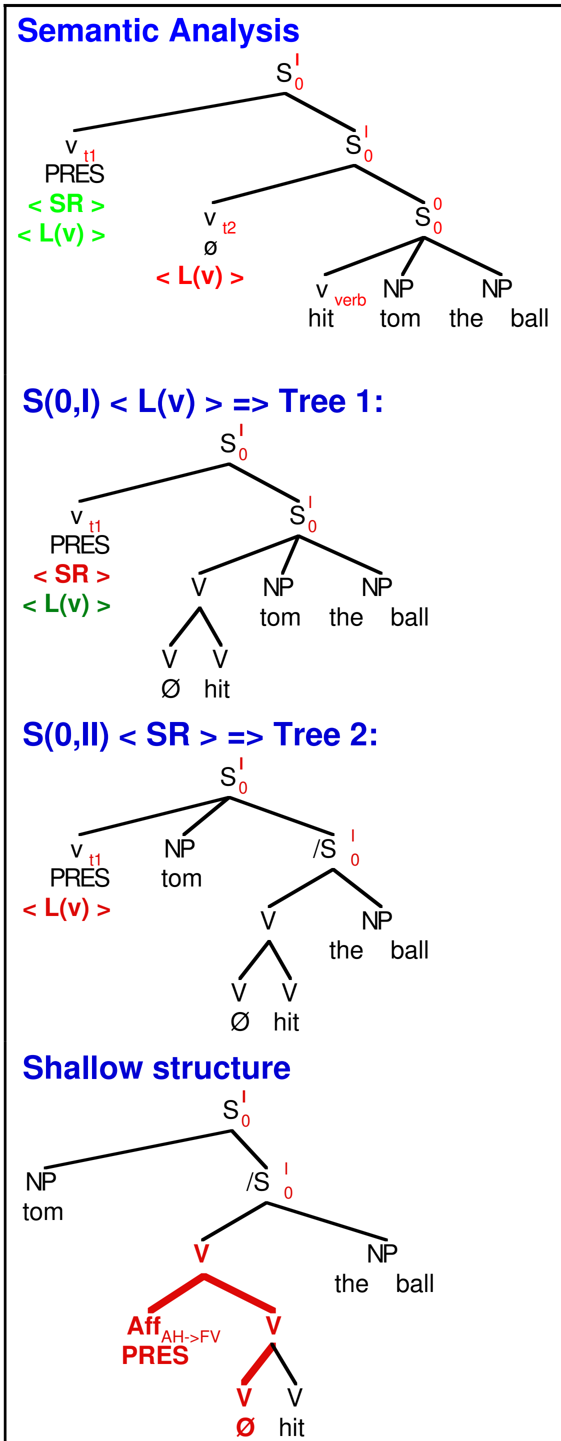
Since according to our toy lexicon the verb `swim` allows for one `np` only, a negative result is obtained, when two arguments are specified for that verb, like in the SA for "He swam the canal", as the following call to **s** shows:

```
?-s([s(0,2,'',[]), [v-t1, 'PAST'],
                   [s(0,1,'',[]), [v-t2, Ø],
                                  [s(0,0,'',[]), [v-lex,swim],
                                                 [np,he],
                                                 [np,the,canal]]]]]).
no
```

One final thing worth noticing about the translation of SALG rules into Prolog clauses variable. There it usually respresents a syntactic output tree (prototypically using a functor notation). Our structure, on the other hand, functions as input and is checked for its acceptability as an SA. So the SALG compiler does not need to add two extra arguments to the terms in the formation rules, as can be verified in the **s** predicates in our example translation of formation rules above.

3.3 The transformational cycle

Since Semantic Syntax is a transformational theory, the heart of the theory consists of the Cycle. So, to understand what we have been implementing, it is necessary to spend some time on this part of the theory. But in spite of the fact that many figures with accompanying texts are used for the explanation, not more than a sketch can be given here. The sentence "Tom hits the ball" is used to illustrate the cycle (figure 6):



This tree is the SA, with the transformations retrieved from the lexicon added to the verbs (shown between pointed brackets). These will be executed starting at the S at the lowest level, the **L(v)** attached to the non-lexical verb for `vt2, Ø`.
A transformation connected to a v always operates on a daughter of the lower S (the v's rightmost sister), and often the v itself.

Lowering (in this case) meant "left adopt" the verb (to which the L is attached) onto the predicate of the lower S. In this case this results in a v-cluster, as the first subtree in the lower S-node. One of the S-nodes has been "pruned" away - the one with the lowered v (S'_0) - since that one had no side branches left. (Its superscript ', was put on the remaining lower S).

This tree is the result of raising the subject, the first argument (= the second daughter = [np,tom]) of the lower S to the level of the S of the predicate that triggered the raising (**SR** (Subject Raising) attached to PRES). The slash (/) in front of the lower S marks the removal of its subject.

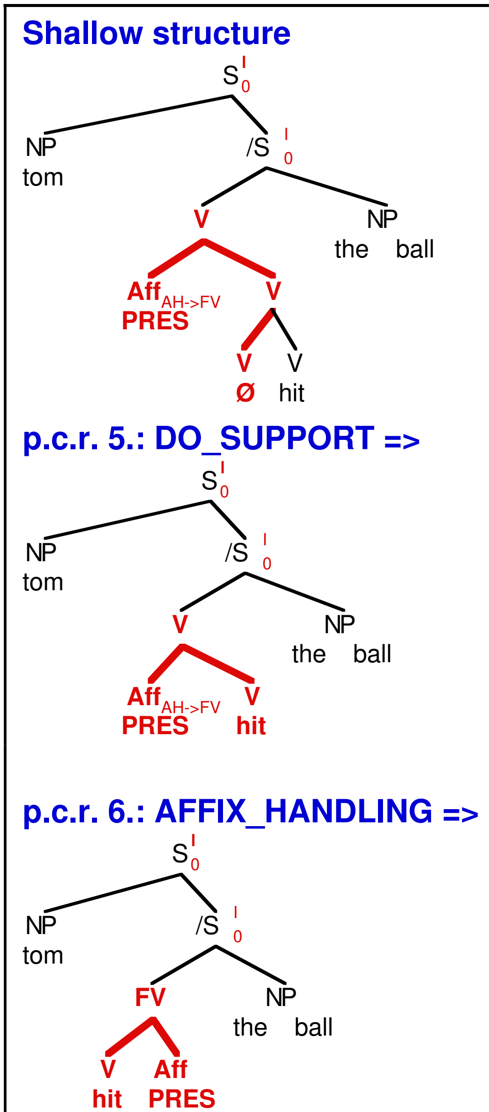
This *Shallow Structure*. ("Shallow", because the Ss are nested less deep than in the SA.).
Here another **L**owering operation was executed, resulting in a yet more complex v-cluster. The lowered v has changed into a new category '`Aff`'. In the post cycle this v-cluster will change into a structure in which PRES is changed into an affix for the verb "hit". Due to the AH->FV sub-script of Aff, a FV node will be created later by the *Affix Handling* stage in the post cycle

Figure 6. The cycle for "Tom hits the ball".

Stated more generally: the transformation executing engine (the *Cycle*) climbs down the SA tree recursively looking for the predicates of Ss and searching the lexicon for the labels of the transformations to be executed - like SR and L(v) - that are associated with the predicates (these labels are present in the toy lexicon above). After reaching the deepest S the Cycle proceeds to execute the transformations (which differ between languages somewhat), starting at the bottom of the tree working its way up, to finally deliver a Shallow Structure.

3.4 The post cycle

The Shallow Structure is put through a *Postcycle*, that unlike the cyclic phase is rather complex and language specific. The postcycle covers variations in verbal end-cluster arrangements in Dutch, French and German of which we will be able to show only the tip of the iceberg. We will illustrate the postcycle by continueing the generation process for "Tom hits the ball" (figure 7).



The part printed fat in the Shallow Structure is called AUX. Aux will result in the auxiliary part of the sentence and plays a role in the movements of sentence parts in the post cycle.

The shallow structure is passed through several postcyclic phases that do not change the tree (*To-insertion*, *that-insertion*, *ø-be-deletion*, *adv-placement, que-attraction*, *wh-fronting*).

Then reached *do-support* is reached, which changes ø into 'do' in some cases (like in questions) and deletes it in other cases.
In our example deletion of the ø was the proper thing to do. A non-branching v-node is pruned, resulting in a simpler v-cluster.

Affix Handling (AH) puts the verb hit and its affix in proper order and executes the AH->FV instruction attached to the Affix node, relabeling the top v-node of the auxiliary into FV.
After AH there is one final postcyclic phase, *mince*, that flattens the v-clusters where possible, but that does not apply here.
So we have reached the endproduct of the generation process: the Surface Structure.

Figure 7. The postcycle for "Tom hits the ball".

3.5 Implementing the operations on trees in Prolog

In implementing the operations on the trees, a major role is played by a *treematcher*, that can search for a pattern in a list in a hierarchical way, by delving into sublists. In the pattern several operators are used to describe the element(s) to be found:

      * means zero or more matching elements,
      ~ means exactly one element.

For example:

```
:-treematch([s, [v-verb, ~]],
            [s, [v-verb,hit],[np,tom],[np,the,ball]]).
no
```

If one of these operator is followed by a term (e.g. a variable), that term will be unified with any matching element(s). For example:

```
?-treematch([s, [v-verb, ~], *X, *Y],
            [s, [v-verb,hit],[np,tom],[np,the,ball]]).
No.1 : X = [], Y = [[np, tom], [np, the, ball]]
No.2 : X = [[np, tom]], Y = [[np, the, ball]]
No.3 : X = [[np, tom], [np, the, ball]], Y = []
No more solutions

:-treematch([s, ~Pred, *Args],
            [s,[v-verb,hit],[np,tom],[np,the,ball]]).
Pred = [v-verb, hit], Args = [[np, tom], [np, the, ball]]

:-treematch([s, ~, ~Subj, *],
            [s,[v-verb,hit],[np,tom],[np,the,ball]]).
Subj = [np, tom]

:-treematch([s, [~(v-Class), ~Verb], *],
            [s,[v-verb,hit],[np,tom],[np,the,ball]]).
Class = verb, Verb = hit
```

Another feature of the matcher is the ability to add "test predicates" to the pattern by means of the `if` operator. `atom` is used as a test predicate in the following examples:

```
:-treematch([s,[v-verb, ~Verb if {atom}], *],
            [s,[v-verb,hit],[np,tom],[np,the,ball]]).
Verb = hit

:-treematch([s,~Verb if {atom}, *],
            [s,[v-verb,hit],[np,tom],[np,the,ball]]).
no
```

The logic variables bound to parts of the tree after a successful tree matching operation, can be recombined and moved around by using *maketree*, the "inverse" of *treematch*, that constructs a tree out of a pattern. Combining these two basic matching predicates, and with the help of a few other (test) predicates, the t-rule *Subject Raising* (SR) in Prolog is defined as follows:

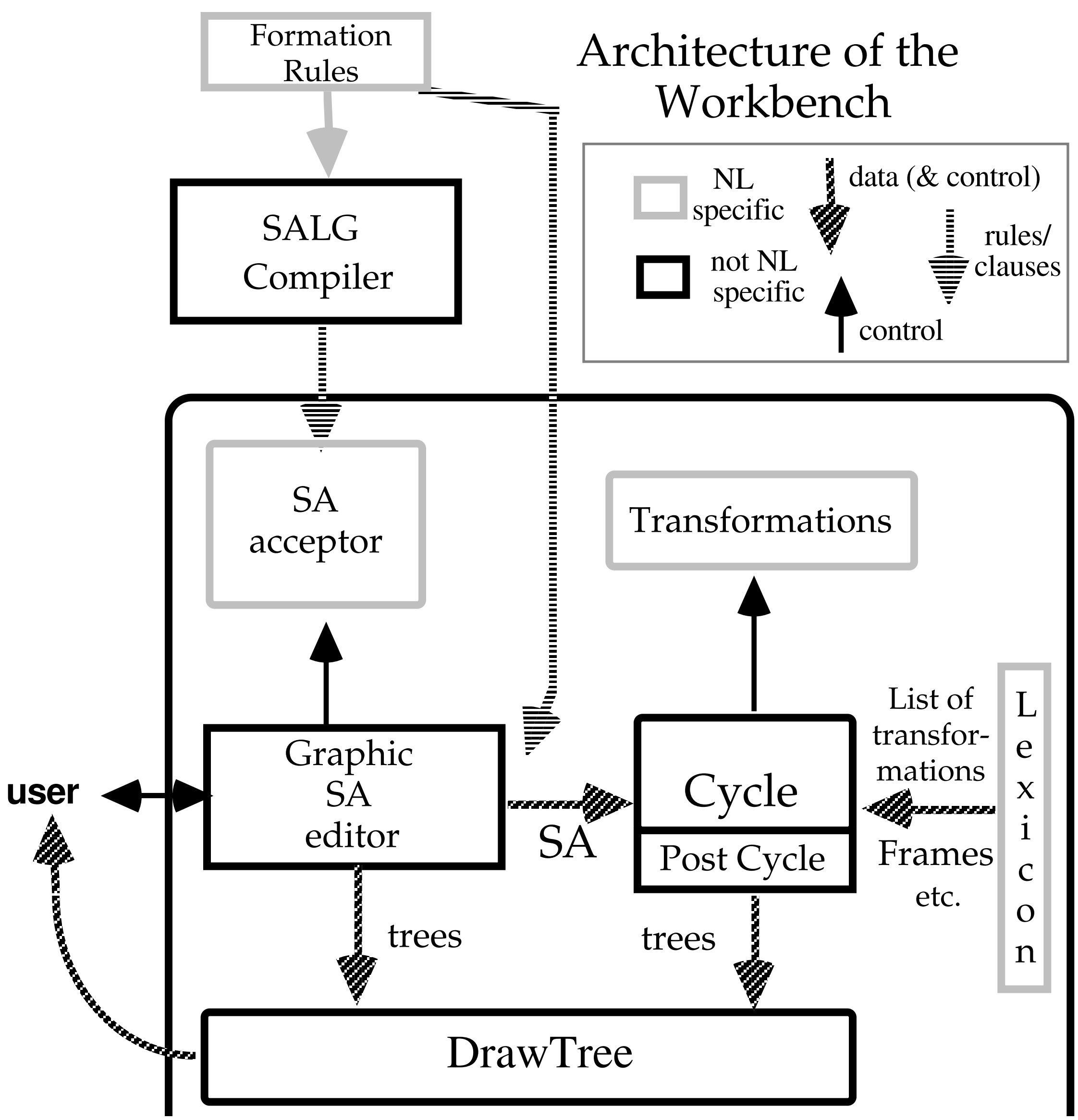
```
%------------------------------------------------------------------
%  SUBJECT RAISING (SR)  (Official Seuren description):
%------------------------------------------------------------------
%  Only if V<SR> has argument-S' or So:
%       Select the first NP to the right of the lower V
%       (this NP may be a prepositional object).
%       Place this NP-constituent in the position of its own S.
%       Move this S (which is now /S) one position to the right.
%------------------------------------------------------------------
t_rule('SR',TreeIn,TreeOut,_):-
 treematch([~S1,~V1 if {v_cluster},*LeftSisters,
                [~S2 if {sr_allowed},~V2 if {v_cluster},
                                     ~Subject if {np_tree},
                                     *Rest2],
             *Rest1],
            TreeIn),
 mark_S_node_as_demoted(S2,DemotedS2), % s -> /s (subj removed)
 sca(V2,V2_sc),  % Surface Cluster arrangement: e.g. v-aff -> 'Aff'
 maketree([~S1,~V1,*LeftSisters,~Subject,
            [~DemotedS2,~V2_sc, *Rest2] , *Rest1],
```

Another important predicate that was used to implement the operations on trees is *submatch* that can match with subtrees within a tree, while remembering the context in which that subtree occurred. Contexts and (parts of) subtrees can be recombined to construct new trees. We will not go into that here.
Also in defining the post cyclic operations, the matching predicates and their inverses are heavily used. The basic ideas for the definition of a matcher in Prolog can be found in Schotel (1987).

4. Other aspects of the workbench

4. 1 The architecture of the workbench

Figure 8. depicts the architecture of the workbench. We see the parts that do the linguistic work already discussed: The *Cycle* and the *Postcycle*. One sees the *lexicon*, from where the *list of transformations* to be executed and the *frames* of the verbs are retrieved. Linked to the transformational engine of the cycle is a box that contains the code for the *transformations*. On the top left is shown how *formation rules* are converted into an *SA acceptor* by the *SALG Compiler*.
The user is supposed to work in one language at a time. If he/she switches to another language, all the NL specific parts (depicted in gray) are exchanged for others.
The *normal user* has to work within the framework of the given sets of formation rules, lexicons, transformations and postcyclic routines. A *super user*, who has to know a certain password, is allowed to modify the NL specific parts just mentioned.



4.2 The graphical user interface

The user interface is represented by two boxes in the architecture: One is for *DrawTree*, the other for the *SA Editor*.

4.2.1 DrawTree

DrawTree does all the treedrawing. It does this by converting a list representation of a tree into LPA MacProlog's so-called Graphic Description Language. DrawTree has drawn all the trees shown in this paper. The basic principle of the algorithm is: For each of the nodes, draw it, unless it has a subtree. In that case: compute the width of its subtree (if any), draw the nodes of the subtree and draw the node right in the middle above its subtree, connecting the node with the its daughters. Writing the Prolog code is left as an exercise to the reader.

The user has some control during the generation process: after each transformation a dialog is popped up that enables him/her to decide whether *DrawTree* has to show the result of the transformation, or skip drawing the whole cycle etc. But the module that offers the most graphically is the SA-editor.

4.2.2 The SA-editor

In the SA EDITOR (developed in co-operation with G. Marttin), the (normal) user can construct an SA, by putting together subtrees. Refer to the figures 9-11 below to see what it looks like on the screen. The figures show a so called graphical window, that has a "tool bar" on its left.

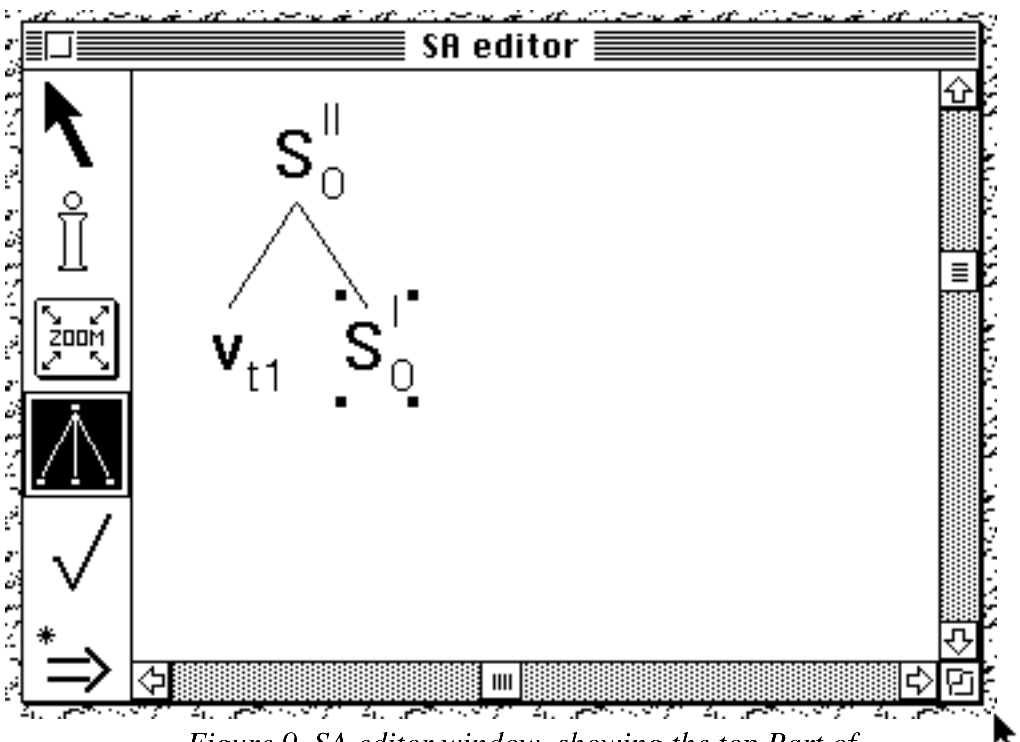


Figure 9. SA editor window, showing the top Part of a future SA, with the right daughter selected.

In figures 9 and 11 one of the tools (looking like a tree) is blackened, which means that it has been selected. This particular tool enables the user to start or extend a tree, which in the end will be an SA. Clicking the mouse while pointing at a node in the drawing area at the right of the tool bar, will pop up a menu that offers a set of trees to choose from. (See figure 10). Each subtree of these corresponds to one of the possibilities allowed for by the formation rules. In other words each formation rule has been converted into a small tree: the top node corresponds to the left hand side, the daughters to the right hand side. In figure 10 the menu shows two possibilities (each having S'_0 as the top, because that was the node clicked on in figure 9). The tree at the right hand side has been selected (as the square around it shows). Now when the user presses the SELECT button, that subtree will be added to the tree under construction, as can be

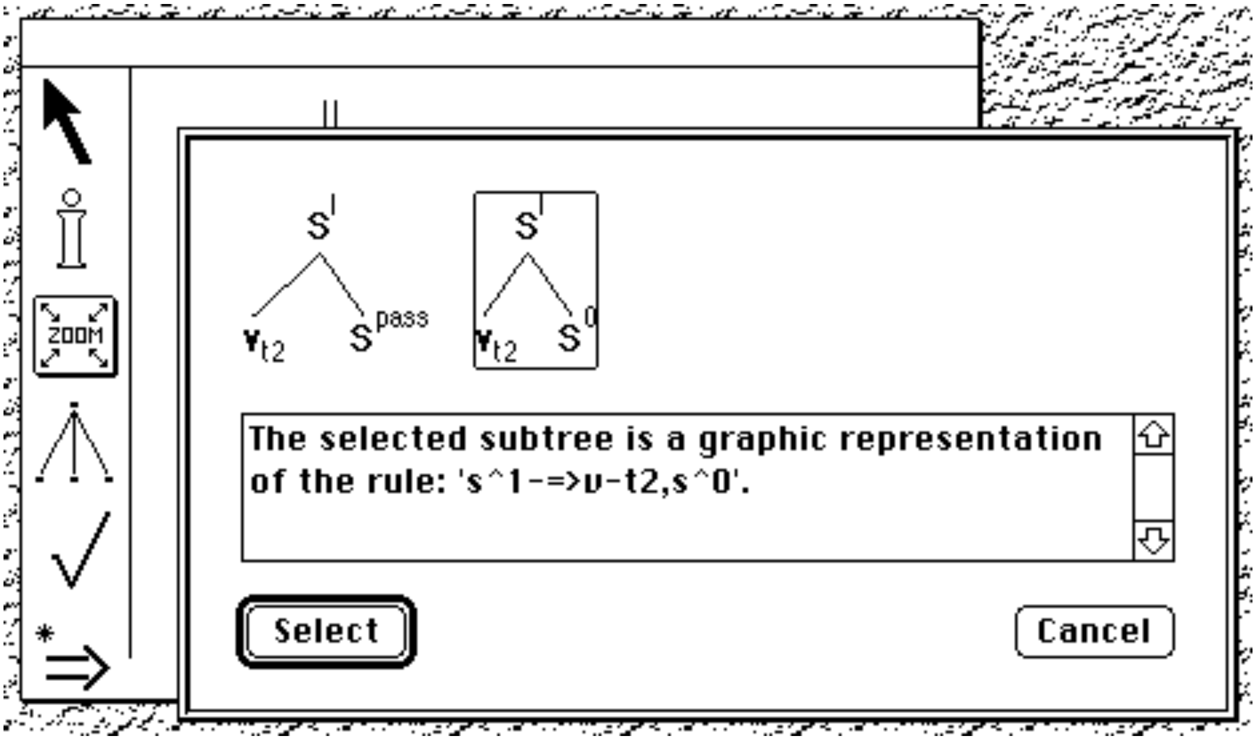


Figure. 10. A menu offering two ways to extend S' into a subtree;
(the second option has been selected).

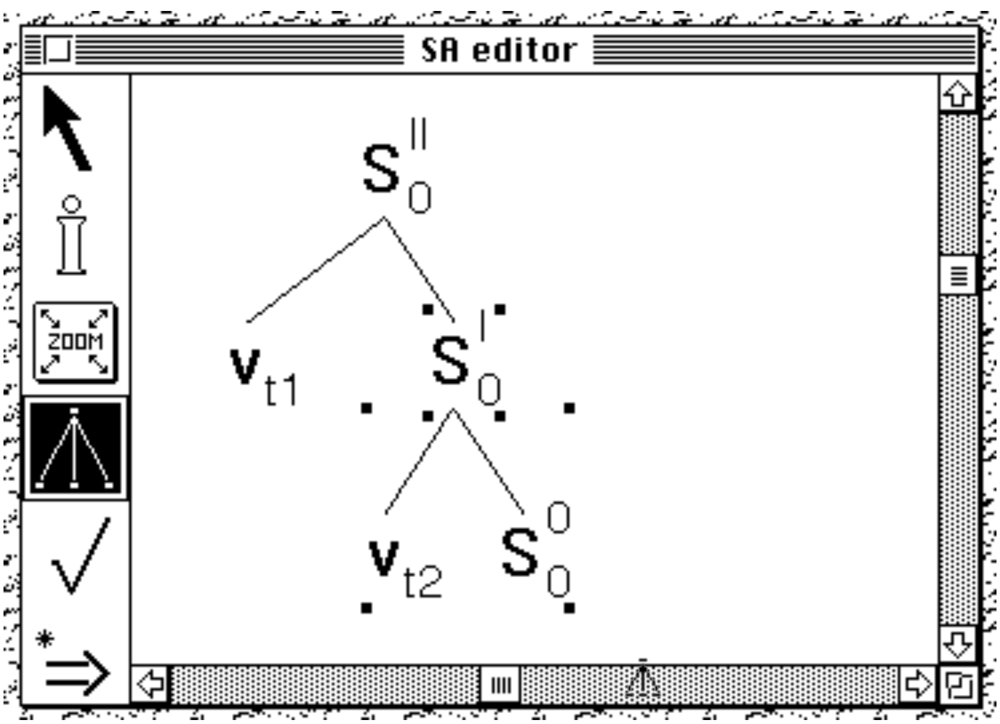


Figure. 11. The SA after adding daughters to S'

Via the Apple Macintosh menu bar, the user is able to move trees around, split up a tree, duplicate trees and combine trees with other ones. After a tree has been completed, the aforementioned SA acceptor can be used to check whether the tree constructed. really is an SA (this is sometimes necessary, because some of the details of the formation rules not discussed here, could not be built into the SA-editor). After the editing has been completed, a button can be pushed to trigger the sentence generator.

This graphical interface greatly improves the user-friendliness of SeSynPro and will

Literature

Abramson, H & Dahl, V. *Logic Grammars*. Springer-Verlag, Berlin, 1989.

Covington, M.A. *Natural Language Processing for Prolog Programmers*, Prentice Hall, Englewood Cliffs, 1994.

Gal, A., Lapalme, G.,Saint_Dizier, P. & Somers, P. *Prolog for Natural Language Processing*, John Wiley & Sons, New York, 1991.

Marttin, G. De SA-editor. Doctoraalscriptie, Nijmegen, 1994.

Mc Cawley, J.D. *English as a VSO Language*, in Seuren 1974.

Pereira, F.C.N & Shieber, S.M.S. *Prolog and Natural-Language Analysis* CSLI Lecture Notes, Stanford, 1987.

Schotel, H.P. *Programmeren in Prolog*. Coutinho, Muiderberg, 1987.

Seuren, P.A.M. *Autonomous versus Semantic Syntax*. In Seuren 1974.

Seuren, P.A.M. (ed) *Semantic Syntax*. Oxford University Press, Oxford, 1974, reprinted 1978.

Seuren, P.A.M. *Discourse Semantics*. Blackwell, Oxford, 1985.

Seuren, P.A.M. *Semantic Syntax*. Prepublication draft, 1994.

Seuren, P.A.M. *Translation relations in Semantic Syntax*. This volume, 1994.