SAUMER: SENTENCE ANALYSIS USING METARULES

Fred Popowich
Natural Language Group
Laboratory for Computer and Communications Research
Department of Computing Science
Simon Fraser University
Burnaby, B.C., CANADA V5A 1S6

ABSTRACT

The SAUMER system uses specifications of natural language grammars, which consist of rules and metarules, to provide a semantic interpretation of an input sentence. The SAUMER Specification Language (SSL) is a programming language which combines some of the features of *generalised phrase structure grammars* (Gazdar, 1981), like the correspondence between syntactic and semantic rules, with *definite clause grammars* (DCGs) (Pereira and Warren, 1980) to create an executable grammar specification. SSL rules are similar to DCG rules except that they contain a semantic component and may also be left recursive. Metarules are used to generate new rules from existing rules before any parsing is attempted. An implementation is tested which can provide semantic interpretations for sentences containing topicalisation, relative clauses, passivisation, and questions.

1. INTRODUCTION

The SAUMER system allows the user to specify a grammar for a natural language using rules and metarules. This grammar can then be used to obtain a semantic interpretation of an input sentence. The SAUMER Specification Language (SSL), which is a variation of *definite clause grammars* (DCGs) (Pereira and Warren, 1980), captures some of the features of *generalised phrase structure grammars* (GPSGs) (Gazdar, 1981) (Gazdar and Pullum, 1982), like rule schemata, rule transformations, structured categories, slash categories, and the correspondence between syntactic and semantic rules. The semantics currently used in the system are based on Schubert and Pelletier's description in (Schubert and Pelletier, 1982), which adapts the intensional logic interpretation associated with GPSGs, into a more conventional logical notation.

2. THE SEMANTIC LOGICAL NOTATION

The logical notation associated with the grammar differs from the usual notation of intensional logic since it captures some intuitive aspects of natural language.[1] Thus, individuals and objects are treated as entities, instead of collections of properties, and actions are n-ary relations between these entities. Many of the problems that the intensional notation would solve are handled by allowing ambiguity to be represented in the logical notation. Consequently, as is common in other approaches, (e.g., Gawron, 1982), much of the processing is deferred to the pragmatic stage. The structure of the lexicon, and the appearance of post processing markers (sharp angle brackets) are designed to reflect this ambiguity. The lexicon is organised into two levels. For the semantic interpretation, the first level gives each word a tentative interpretation. During the pragmatic analysis, more complete processing information will result in the final interpretation being obtained from the second level of the lexicon. For example, the sentence *John misses John* could be given an initial interpretation of:

(2.1) [John1 miss2 John3]

with *John1*, *miss2* and *John3* obtained from the first level of the two level lexicon. The pragmatic stage will determine if *John1* and *John3* both refer to the same entry, say JOHN_SMITH1, of the second level of the lexicon, or if they correspond to different entries, say JOHN_JONES1 and JOHN_EVANS1. During the pragmatic stage, the entry of MISS which is referred to by *miss2* will be determined (if possible). For example, does John miss John because he has been away for a long time, or is it because he is a poor shot with a rifle?

Any interpretation contained in sharp angle brackets, <...>, may require post processing. This is apparent in interpretations containing determiners and co-ordinators. The proverb:

(2.2) every man loves some woman

could be given the interpretation:

(2.3) [<every1 man2> love3 <some4 woman5>]

without explicitly stating which of the two readings is intended. During pragmatic analysis, the scope of *every* and *some* would presumably be determined.

---

[1] It should also be noted that, due to the separability of the semantic component from the grammar rule, a different semantic notation could easily be introduced as long as the appropriate semantic processing routines were replaced. The use of SAUMER with "an 'AI-adapted' version of Montague's Intensional Logic" is being examined by Fawcett (1984).

The syntax of this logical notation can be summarised as follows. Sentences and compound predicate formulas are contained within square brackets. So, (2.4) states that *John wants to kiss Mary*:

(2.4) [John1 want2 [John1 kiss3 Mary4]]

These formulas can also be expressed equivalently in a more functional form according to the equivalence

(2.5) $[\ t_n\ P\ t_1\ \ldots\ t_{n-1}\ ]$
$= \langle\ \ldots\ \langle\langle P\ t_1\rangle\ t_2\rangle\ \ldots\ t_n\ \rangle$
$= \langle\ P\ t_1\ \ldots\ t_n\ \rangle$

Consequently, (2.4) could also be represented as:

(2.6) ⟨⟨want2 ⟨⟨kiss3 Mary4⟩ John1⟩⟩ John1⟩

However, this notation is usually used for incomplete phrases, with the square brackets used to obtain a *conventional* final reading Modified predicate formulas are contained in braces. Thus, *a little dog likes Fido* could be expressed as:

(2.7) [<a1 {little2 dog3}> likes4 Fido5]

The lambda calculus operations of lambda abstraction and elimination are also allowed. When a variable is abstracted from an expression as in:

(2.8) λx [x want2 [x love3 Mary4]]

application of this new expression to an argument, say *John1*:

(2.9) ⟨ λx [x want2 [x love3 Mary4]] John1 ⟩

will result in an interpretation of *John wants to love Mary*:

(2.10) [John1 want2 [John1 love3 Mary4]]

Further details on this notation are available in (Schubert and Pelletier, 1982).

3. THE SAUMER SPECIFICATION LANGUAGE

The SAUMER Specification Language (SSL) is a programming language that allows the user to define a grammar of a natural language in terms of rules, and metarules. Metarules operate on rules to produce new rules. The language is basically a GPSG realised in a DCG setting. Unlike GPSGs, the grammars defined by this system are not required to be context-free since procedure calls are allowed within the rules, and since logic variables are allowed in the grammar symbols.

The basic objects of the language are atoms, variables, terms, and lists. Any word starting with a lower case letter, or enclosed in single quotes is an atom. Variables start with a capital letter or an underscore. A term is an atom, optionally followed by a series of objects (arguments), which are enclosed in parentheses and separated by commas. Lastly, a list is a series of one or more objects, separated by commas, that are enclosed in square brackets

3.1 Rules

The rules are presented in a variation of the DCG notation, augmented with a semantic rule corresponding to each syntactic rule. Each rule is of the form "A –> β : γ" where *A* is a term which denotes a nonterminal symbol. β is either an atom list representing a terminal symbol or a conjunction of terms (separated by commas) corresponding to nonterminal symbols, and γ is a semantic rule which may reference the interpretation of the components of β in determining the semantics of *A*. The rule arrow, –>, separates the two sides of the rule, with the colon, :, separating the syntactic component from the semantic component. If the rule is preceded by the word *add*, it can be subjected to the transformations described in section 3.2. The nonterminal symbols can possess arguments, which may be used to capture the flavour of the *structured categories* of GPSGs. β may also possess arbitrary procedural restrictions contained in braces.

γ consists of expressions in the semantic notation. The different terms of this semantic expression are joined by the semantic connector, the ampersand "&". The ampersand differs from the syntactic connector, the comma, since the former associates to the right while the latter associates to the left. The *logical and* symbol, which traditionally may also be denoted by the ampersand, must be entered as "&&". Due to constraints imposed by the current implementation, "⟨ *expr* ⟩" must be entered as "<[*expr*]", "< *expr* >" as "<<[*expr*]", and "λ x *expr*" as "x lmda *expr*." An expression may contain references to the interpretations of the elements of β by stating the appropriate nonterminal followed by the left quote, `. To prevent ambiguity in these references that may arise when two identical symbols appear in β, a nonterminal may be appended with a minus sign followed by a unique integer.

Unlike standard Prolog implementations of DCGs, left recursion is allowed in rules, thus permitting more natural descriptions of certain phenomena (like co-ordination). Since the left recursive rules are interpreted, rather than converted into rules that are not left recursive, the number of rules in the database will not be affected. However, the efficiency of the sentence analysis may be affected due to the extra processing required. Rules of the form "A --> A, A" are not accepted.

An example of a production that derives *John* from a proper noun, *npr*, is shown in (3.1):

(3.1) npr –> ['John'] : 'John'#

The semantic interpretation of this *npr* will be *John#*, with "#" replaced by a unique integer during evaluation. (3.2) illustrates a verb phrase rule that could be used in sentences like *John wants to walk*:

(3.2) vp(Num) –>
v(Num,Root) with Root in [want,like], vp(inf)
x## lmda [x## & v` & [x## & vp`])]

First notice that a restriction on the verb appears within the *with* statement. In the GPSG formalism, this type of restriction would be obtained by naming the rules and associating a list of valid rule names with each lexical entry. Although the *with* restriction may contain any valid procedure, typically the *in* operation (for determining list membership) is used. The double pound, ##, is replaced by the same unique integer in the entire expression when the expression is evaluated. If "#" were used instead, each instance of *x#* would be different. For the above example, if *v'* is *want2* and *vp'* is *run3*, then the semantic expression could evaluate to:

(3.3) x4 lmda [x4 & want2 & [x4 & run3]]

Furthermore, if *np'* is *John1*, then:

(3.4) [np' & vp']

could result in:

(3.5) [John1 & want2 & [John1 & run3]]

3.2 The Metarules

Traditional transformational grammars provide transformations that operate on parse trees, or similar structures, and often require the transformations to be used in sentence recognition rather than in generation (Radford, 1981). However, the approach suggested by (Gazdar, 1981) uses the transformations generatively and applies them to the grammar. Thus, the grammar can remain *context-free* by compiling this transformational knowledge into the grammar. Transformations and rule schemata form the *metarules* of SSL.[2]

Rule schemata allow the user to specify entire classes of rules by permitting variables which range over a selection of categories to appear in the rule. To control the values of the variables, the *forall* control structure can be used in the schema declaration. The schema *forall X in List, Body* will execute *Body* for each element of *List*, with *X* instantiated to the current element. The use of this statement is illustrated in the following metarule that generates the terminal productions for proper nouns:

(3.6) forall Terminal in ['Bob','Carol','Ted','Alice'].
(npr --> [Terminal] : Terminal#) .

Transformations match with grammar rules in the database, using a rule pattern that may be augmented with arbitrary procedures, and produce new rules from the old rules. A transformation is of the form:

(3.7) α --> β : γ ==> α' --> β' : γ'

The metarule arrow, ==>, separates the pattern, α --> β : γ, from the template, α' --> β' : γ'.

[2]Often, metarules are considered to consist of transformations only, while schemata are put into a category of their own. However, since they can both be considered as part of a metagrammar, they are called metarules in this discussion.

The *syntactic pattern*, α --> β, contains nonterminals, which correspond to symbols that must appear in the matched rule, and free variables, which represent *don't care* regions of zero or more nonterminals. The pattern nonterminals may also possess arguments. For each rule symbol, a matching pattern symbol describes properties that *must* exist, but not all the properties that *may* exist. Thus, if *vp* appeared in the pattern, it would match any of *vp*, *vp(Num)*, or *vp(Num,Type) with Type in [trans]*. However, *pp(to)* would not match *pp* or *pp(from)*, but it would match *pp(to,_)*. The matching conditions are summarised in Figures 3-1 and 3-2. In Figure 3-1, *A* and *B* are nonterminals, *X* is a free variable, and α and β are conjunctions of one or more symbols. γ and δ of Figure 3-2 are also conjunctions of one or more symbols. "=" is defined as unification (Clocksin and Mellish, 1981). Parts of the rule contained in braces are ignored by the pattern matcher. The syntactic pattern may also contain arbitrary restrictions,[3] enclosed in braces, that are evaluated during the pattern match. The *semantic pattern*, γ, is very primitive. It may contain a free variable, which will bind to the entire semantics field of the matched rule, or it may contain the structure <[? *x*], which will bind to the entire structure containing the symbol *x*. If <[? *y*] then appears in γ', the result will be the semantic component of the matched rule with *x* replaced by *y*.

	Rule	
Pattern	(B, β)	B
(A, α)	A matches B and α matches β	A matches B and α is a free variable
(X, α)	(X, α) matches β or α matches (B, β)	α matches B
A	No	A matches B
X	Yes	Yes

Figure 3-1: Pattern Matching for Conjunctions

	Rule	
	$b(\beta_1, \ldots \beta_n)$	$b(\beta_1, \ldots \beta_n)$ with δ
Pattern		
$a(\alpha_1, \ldots \alpha_m)$	$a = b$, $m \leq n$, $\alpha_i = \beta_i$, $1 \leq i \leq m$	$a = b$, $m \leq n$, $\alpha_i = \beta_i$, $1 \leq i \leq m$
$a(\alpha_1, \ldots \alpha_m)$ with γ	No	$a = b$, $m \leq n$, $\alpha_i = \beta_i$, $1 \leq i \leq m$, γ matches δ

Figure 3-2: Pattern Matching for Nonterminals

[3]Apparently not present in the Hewlett Packard system (Gawron, 1982) or the ProGram system (Evans and Gazdar, 1984)

The behaviour of patterns can be seen in the following examples. Consider the sentence rule:

```
(3.8)   s(decl) -->   np(nom,Numb),
                      vp(_,Numb) with agreement(Numb)
                      : [ np' & vp' ]
```

The patterns shown in (3.9a) will match (3.8), while those of (3.9b) will not match it.

```
(3.9)   (a) s(A) --> {not element(A,[foo])}, X, vp : Sem
            s --> np(nom), X, vp(pass), Y : Sem

        (b) s(inter) --> np, vp : Sem
            s --> vp : Sem
```

For the verb phrase rule shown in (3.10):

```
(3.10)  vp(active,[MIN]) -->
            v([MIN],Root,Type,_) with (intrans in Type)
            : v'
```

the patterns of (3.11a) will result in a successful match, will those of (3.11b) will not:

```
(3.11)  (a) vp -->  v : <[? v]
            vp -->  v(_,_,Type,_)
                    with (X, intrans in Type, Y),
                    Z : Sem
        (b) vp -->  v(_,_,Type,_)
                    with (X, trans in Type)
                    : Sem
            vp -->  v(_,Root,_,_)
                    with (Root in [foo], X)
                    : Sem
```

For every rule that matches the pattern, the template of the transformation is executed, resulting the creation of a new rule. Any nonterminal, N, that matches a symbol β_i on the left side of the transformation, will appear in the new rule if there is a symbol β_i' in β' that *intra-transformation* (IT) matches with β_i. If there are several symbols in β' that IT-match β_i, the leftmost symbol will be selected. No symbol on one side of the transformation may IT-match with more than one symbol on the other side. Two symbols will IT-match only if they have the same number of arguments, and those arguments are identical. Any *with* expressions and modifiers associated with symbols are ignored during IT-matching. β' may also contain extra symbols that do not correspond to anything in β. In this case, they are inserted directly into the new rule. Once again, if the transformation is preceded by the command *add*, then the resulting rules can be subjected to subsequent transformations.

3.3 Modifiers

Both rules and metarules may contains modifiers that alter the structure of the nonterminal symbols. There are two types of modification, which have been dubbed *external* and *internal* modification.

With external modification, any nonterminal, or variable instantiated to a nonterminal, may be followed by the sequence *@mod*. This will result in *mod* being inserted into the argument list following the *specified* arguments. Thus, if *N@junk* appeared in a rule when *N* was instantiated to *np(more)*, it would be expanded as *np(more,junk)*. Similarly, if the pattern symbol *vp* matched *vp(Numb)* in a rule, then the appearance of *vp@foo* in the template would result in *vp(foo,Numb)* appearing in the new rule. This extra argument, introduced by the modifier, can be useful when dealing with the missing components of *slash* or *derived* categories (Gazdar, 1981).

Internal modification allows the modifier to be put directly into the argument list. If an argument is followed by *@mod*, it will be replaced by *mod*. In the case where *@mod* appears as an argument by itself, *mod* is added as a new argument. For example, if *v(Numb@pastpart)* were contained in a template, it would IT-match *v(Numb)* in the pattern, and would result in the appearance of *v(pastpart)* in the new rule.

4. IMPLEMENTATION

The SAUMER system is currently implemented in highly portable C-Prolog (Pereira, 1984), and runs on a Motorola 68000 based SUN Workstation supporting UNIX[4]. Calls to Prolog are allowed by the system, thus providing useful tools for debugging grammars, and tracing derivations. However, due to the highly declarative nature of SSL, it is not restricted to a Prolog implementation. Implementations in other languages would differ externally only in the syntax of the procedure calls that may appear in each rule. Use of the system is described in detail in (Popowich, 1985).

The current implementation converts the grammar as specified by the rules and metarules into Prolog clauses. This conversion can be examined in terms of how rules are processed, and how the schemata and transformations are processed.

4.1 Rule Processing

The syntactic component of the rule processor is based on Clocksin and Mellish's definite clause grammar processor (Clocksin and Mellish, 1981) which has been implemented in C-Prolog. For a DCG rule, each nonterminal is converted into a Prolog predicate, with two additional arguments, that can be processed by a top-down parser. These extra arguments correspond to the list to be parsed, and the remainder of the list after the predicate has parsed the desired category. With the addition of semantics to each rule, another argument is required to represent the semantic interpretation of the current symbol. Thus, whenever a left quoted category name, *x'*,

[4]UNIX is a trademark of Bell Laboratories

appears in the semantics of the rule, it is replaced by a variable bound to the semantic argument of the corresponding symbol, *x*, in the rule. The semantic expression is then evaluated by the *eval* routine with the result bound to the semantic argument of the nonterminal on the left hand side of the production. For example, the sentence rule:

```
(4.1)   add  s(decl)  -->
                      np(nom,Numb),
                      vp(_,Numb) with agreement(Numb)
                      : [ np' & vp' ]
```

will result in a Prolog expression of the form:

```
(4.2)   s(SemS,decl,_1,_3) :-
                      np(SemNP,nom,Numb,_1,_2),
                      vp(SemVP,_,Numb,_2,_3),
                      agreement(Numb),
                      eval([SemNP & SemVP],SemS).
```

Consequently, to process the sentence *John runs*, one would try to satisfy:

```
(4.3)   :- s(Sem, Type, ['John',runs], []).
```

The first argument returns the interpretation, the second argument returns the type of sentence, the third is the initial input list, and the final argument corresponds to the list remaining after finding a sentence. Any rule *R*, that is preceded by *add* will have the axiom *rule(R)* inserted into the database. These axioms are used by the transformations during pattern matching.

The *eval* routine processes the suffix symbols, # and ## along with the lambda expressions, and may perform some reorganisation of the given expression before returning a new semantic form. For each expression of the form *name#*, a unique integer *N* is created and *name-N* is returned. With "##", the procedure is the same except that the first occurrence of "##" will generate a unique integer that will be saved for all subsequent occurrences. To evaluate an expression of the form:

(4.4) $\langle\ expr_i$ lmda $expr_j$ & X $\rangle$

every subexpression of $expr_j$ is recursively searched for an occurrence of $expr_i$, which is then replaced by *X*.

Left recursion is removed with the aid of a gap predicate identical to the one defined to process gapping grammars (Dahl and Abramson, 1984) and unrestricted gapping grammars (Popowich, forthcoming). For any rule of the form:

(4.5) A --> A, B, α

where A does not equal B, the result of the translation is:

(4.6) $A(\_1,N_n)$:- gap(G,_1,_2), $B(\_2,N_0)$, A(G,[]). $\alpha_1(N_0,N_1)$, . . ., $\alpha_n(N_{n-1},N_n)$.

According to (4.6), a phrase is processed by skipping over a region to find a B — the first non-terminal that does not equal A. The skipped region is then examined to ensure that it corresponds to an A before the rest of the phrase is processed.

4.2 Schema Processing

To process the metarule control structures used by schemata, a *fail* predicate is inserted to force Prolog to try all possible alternatives. The simple recursive definition of *forall X in List*:

```
(4.7)   forall(X in [], Body).
        forall(X in [Y|Rest],Body)  :-
            (X=Y, call1(Body), fail) ;
            forall(X, Rest, Body).
```

uses *fail* to undo the binding of *Y*, the first element of the list, to *X* before calling *forall* with the remainder of the list. The predicate *call1* is used to evaluate *Body* since it will prevent the *fail* predicate from causing backtracking into *Body*.

4.3 Transformation Processing

Execution of transformations requires the most complex processing of all of the metagrammatical operations. This processing can be divided into the three stages of *transformation creation*, *pattern matching*, and *rule creation*.[5]

During the *transformation creation* phase, the predicate *trans(M,X,Y)* is created for the metarule, *M*. This predicate will transform a list of elements, *X*, into another list, *Y*, according to the syntax specification of the metarule. Elements that IT-match will be represented by the same free variable in both lists. This binding will be one to one, since an element cannot match with more than one element on the other side. Symbols that appear on only one side will not have their free variable appearing on the opposite side. Expressions in braces are ignored during this stage. If a transformation like:

(4.8) a --> b, c, X ==> a@foo --> b, X, c(foo)

appears, then a predicate of the form:

(4.9) trans(M, [_1,_2,_3,X], [_1,_2,X,_4])

will be created. Notice that the appearance of a modifier does not cause *a@foo* to be distinguished from *a*, since all modifiers are removed before the pattern-template match is attempted. However, *c* and *c(foo)* are considered to be different symbols. *M* is a unique integer associated with the transformation.

The *pattern match* phase determines if a rule matches the pattern, and produces a list for each successful match which will be transformed by the *trans* predicate. Each element of the list is either one of the matched symbols from the rule, or a list of symbols corresponding to the *don't care* region of the pattern. Any predicates that

[5](Popowich, forthcoming) examines a method of transformation processing that uses the transformations during the parse, instead of using them to generate new rules.

appear in braces in the pattern are evaluated during the pattern match. Consider the operation of an active-passive verb phrase transformation:

```
(4.10)  vp(active.Numb) -->
            v(Numb.R.Type.SType)
            with (X.trans in Type.Y).
            np. Z
              <[? np']
        ==>
        vp(pass.Numb)  -->
            v(Numb.be.T.S)-1 with aux in T.
            v(Numb@pastpart.R.Type.SType)
            with (X.trans in Type.Y).
            Z. pp(by._)
            : x## lmda [pp(by)' & <[? x##]]
```

on the following verb phrase:

```
(4.11)  vp(active.Numb) -->
            v(Numb.R.Type._) with trans in Type.
            np([x.A.x]._._)
            : <[ v' & np' ] .
```

The list produced by the pattern match would resemble:

```
(4.12)  [   vp(active.Numb).
            v(Numb.R.Type._) with [[].trans in Type.[]].
            np([x.A.x]._._).
            [] ]
```

Notice that there was nothing in the rule to bind with *X*, *Y* or *Z*. Consequently, these variables were assigned the null list, []. The pattern match of the semantics of the rule will result in an expression which lambda abstracts *np'* out the of semantics:

```
(4.13)  <[ np' lmda  <[ v' & np' ] ]
```

Finally, the *rule creation* phase applies the transformation to the list produced by the pattern match, and then uses the new list and the template to obtain a new rule. This phase includes conversion of the new list back into rule form, the application of modifiers, and the addition of any extra symbols that appear on the right hand side only. To continue with our example, the *trans* predicate associated with (4.10) would be:

```
(4.14)  trans(N, [_1._2._3.Z], [_3._4._2.Z._5])
```

Notice that the two vp's on opposite sides of the metarule do not match. So the transformed list would resemble:

```
(4.15)  [   _3.
            _4.
            v(Numb.R.Type._) with [[].trans in Type.[]].
            [].
            _5 ]
```

The rule generated by the rule creation phase would be:

```
(4.16)  vp(pass.Numb) -->
            v(Numb.be.T.S)-1 with aux in T.
            v(pastpart.R.Type._) with trans in Type.
            pp(by._)
            : x## lmda [ pp(by)' & <[ v' &  x## ] ]
```

Notice that the expression "<[v' & x##]", which is contained in the semantics of (4.16) was obtained by the application of (4.13) to *x##*.

5. APPLICATIONS

To examine the usefulness of this type of grammar specification, as well as the adequacy of the implementation, a grammar was developed that uses the domain of the Automated Academic Advisor (AAA) (Cercone et.al., 1984). The AAA is an interactive information system under development at Simon Fraser University. It is intended to act as an aid in "curriculum planning and management", that accepts natural language queries and generates the appropriate responses. Routines for performing some morphological analysis, and for retrieving lexical information were also provided.

The SSL grammar allows questions to be posed, permits some possessive forms, and allows auxiliaries to appear in the sentences. From the base of twenty six rules, eighty additional rules were produced by three metarules in about eighty-five seconds. Ten more rules were needed to link the lexicon and the grammar. A selection of the rules and metarules appears in Figure 5-1. The complete grammar and lexicon is provided in (Popowich, 1985).

In the interpretations of some sample sentences, which can be found in Figure 5-2, some liberties are taken with the semantic notation. Variables of the form *wN*, where *N* is any integer, represent entities that are to be instantiated from some database. Thus, any interpretation containing *wN* will be a question. Possessives, like *John's table* are represented as:

```
(5.1)   <table & [John poss table]>
```

Although multiple possessives which associate from left to right are allowed, group possessives as seen in:

(5.2) *the man who passed the course*'s book

and in phrases like:

(5.3) John's *driver's licence*

can not be interpreted correctly by the grammar. Inverted sentences are preceded by the word *Query* in the output. Also, proper nouns are assumed to unambiguously refer to some object, and thus are no longer followed by a unique integer. Analysis times for obtaining an interpretation are given in CPU seconds. The total time includes the time spent looking for all other possible parses.

Results obtained with SAUMER compare favourably to those obtained from the ProGram system (Evans and Gazdar, 1984). ProGram operates on grammars defined according to the current GPSG formalism (Gazdar and Pullum, 1982), but was not developed with efficiency as a major consideration. The grammar used with ProGram, which is given in (Popowich, 1985), is similar to the AAA

```
/* Case is described by a mask, [N,A,G], with free variables for Nom., Acc. and Gen. */

add  vp(active,Numb)   --> v(Numb, Root, T, _) with (Root in [pass,give,teach,offer], indobj in T, trans in T),
                           np([x,D,x],_,_), np([x,A,x],_,_)-1 : <[ v' & np' & np-1' ] .

/* WH-questions in inverted sentences */    eval(y##, Var), NP = np(Case,Numb,Feat)

. ( NP@NP  -->  [], {agreement(Case)} : Var )
. ( s(inv) -->  np([x,A,x],Numb,Feat) with qword in Feat, s(inv)@np([x,A,x],Numb,Feat)
                : <[ (Var lmda s') & np' ] ).

/* passive transformation */

add vp(active,Numb)  -->  v(Numb,R,Type,Subtype) with (X, trans in Type, Y), np, Z : <[? np']
==> vp(pass,Numb)    -->  v(Numb,be,T,S)-1 with  aux in T,
                          v(Numb@pastpart, R, Type, Subtype) with (X, trans in Type, Y),
                          Z, optional(pp(by,_)) : x## lmda [ optional' & <[ ? x## ] ] .
/* sentence inversion */

add  vp(T,[M|N])  -->  v([M|N],R,Type,S) with (X, aux in Type, Y), Z : Sem
==>  s(inv)       -->  v([M|N],R,Type,S) with (X,aux in Type,Y), np([N1,x,x],[M|N],_), Z :[np' & Sem].

/* metarule for the propagation of "holes" in the "slash" categories */

forall Hole in [pp(Prep,Feat),np(Case,Numb,Feat)]
. ( forall Cat1 in [s(Type),vp,pp(Prep,Feat),optional]
  . ( forall Cat2 in [vp,pp(Prep,Feat),np(Case,Numb,Feat),optional]
    . ( Cat1 -->  X, Cat2, Y : Sem   ==>   Cat1@Hole -->  X, Cat2@Hole, Y : Sem ) ) ) .
```

Figure 5-1: Excerpt from Grammar

```
Sentence   did Fred take cmpt101.
Query:     [Fred take5 cmpt101]
Analysis:  2.25 sec.      Total:  4.28334 sec.

Sentence:  who wants to teach Fred's professor's course.
Semantics: [ <w1 & [w1 animate]>
               want4
               [ <w1 & [w1 animate]>
                 teach13
                 <course14 & [ <professor15 & [Fred poss professor15]> poss course14]>
               ]
             ]
Analysis:  6.58337 sec.   Total:  18.9834 sec.

Sentence:  whose course does the student whom John likes want to be taking.
Query:     [ <<the38 student39> & [John like45 <the38 student39>]>
               want46
               [ <<the38 student39> & [John like45 <the38 student39>]>
                 take56
                 <course29 & [<w30 & [w30 animate]> poss course29]>
               ]
             ]
Analysis:  21.9999 sec.   Total:  39.4 sec.

Sentence:  to whom does the professor want which paper to be given.
Query:     [ <the14 professor15>
               want17
               [ x39 give38 <w7 & [w7 animate]> <w21 & [w21 paper22]> ]
             ]
Analysis:  14.3167 sec.   Total:  29.5167 sec.
```

Figure 5-2: Summary of Test Results

grammar used by SAUMER, except that it has a much smaller lexicon, and allows neither relative clauses nor possessive forms. Running on the same machine as SAUMER, ProGram required about 35 seconds to parse the sentence *does John take cmpt101*, with a total processing time of about 140 seconds. SAUMER required just over 2 seconds to parse this phrase, and had a total processing time of about 4 seconds.

As it stands, the semantic notation used by SAUMER does not contain much of the relevant information that would be required by a real system. Tense, number and adverbial information, including concepts like location and time, would be required in the AAA. If the SSL description were to be extended, with the resulting system behaving as a natural language interface of the AAA, a more database directed semantic notation would prove invaluable.

6. PRESENT LIMITATIONS

Although this application of metarules allows succinct descriptions of a grammar, several problems have been observed.

Since each metarule is applied to the rule base only once, the order of the metarules is very important. In our sample grammar, the passive verb phrases were generated before the sentence inversion transformation was processed, and then the slash category propagation transformations were executed. For the current implementation, if a rule generated by transformation T1 is to be subjected to transformation T2, then T1 must appear before T2. Moreover, no rule that is the result of T2 can be operated on by T1. It would be preferable to remove this restriction and impose one that is less severe, such as the finite closure restriction which is described in (Thompson, 1982) and used by ProGram. With this improvement, the only restriction would be that a transformation could only be applied once in the derivation of a rule.

The system can not currently process rules expressed in the Immediate Dominance/ Linear Precedence (ID/LP) format, (Gazdar and Pullum, 1982). With this format, a production rule is expressed with an unordered right hand side with the ordering determined by a separate declaration of *linear precedence*. For example, a passive verb phrase rule could appear something like:

```
(6.1)   vp(pass,[MIN]) -->
            v([MIN], be, _, _).
            v(_, Root, Type, _) with
                    (Root in [pass.carry.give],
                    indobj in Type,
                    trans in Type).
            pp(to),
            optional(pp(by))
            : x## lmda
              [optional' & <[v' & pp(to)' & x##]]
```

with the components having a linear precedence of:

(6.2) v(_,be) < v < pp

The result would be that the *pp(by)* could appear before or after the *pp(to)*, since there is no restriction on their relative positions. If this format were implemented, only one passive metarule would have to be explicitly stated. The direct processing of ID/LP grammars is discussed in (Shieber, 1982), (Evans and Gazdar, 1984), and (Popowich, forthcoming).

7. CONCLUSIONS

SSL appears to adequately capture the flavour of GPSG descriptions while allowing more procedural control. Investigation into a relationship between SSL and GPSG grammars could result in a method for translating GPSG grammars into SSL for execution by SAUMER. Further research could also provide a relationship between SSL and other grammar formalisms, such as *lexical-functional grammars* (Kaplan and Bresnan, 1982). The Prolog implementation of SAUMER, allowing left recursion in rules, should facilitate a more detailed study of the specification language, and of some problems associated with metarule specifications. Due to the easy separability of the semantic rules, one could attempt to introduce a more database oriented semantic notation and develop an interface to a real database. One could then examine system behaviour with a larger rule base and more involved transformations in an applications environment like that of the AAA. However, as is apparent from the application presented here and from preliminary experimentation (Popowich, 1984) (Popowich, 1985), further investigation of the efficient operation of this Prolog implementation with large grammars will be required.

ACKNOWLEDGEMENTS

I would like to thank Nick Cercone for reading an earlier version of this paper and providing some useful suggestions. The comments of the referees were also helpful. Facilities for this research were provided by the Laboratory for Computer and Communications Research. This work was supported by the Natural Sciences and Engineering Research Council of Canada under Operating Grant no. A4309, Installation Grant no. SMI-74 and Postgraduate Scholarship #800.

REFERENCES

Cercone, N., Hadley, R., Martin F., McFetridge P. and Strzalkowski, T. **Designing and automating the quality assessment of a knowledge-based system: the initial automated academic advisor experience**, pages 193-205. IEEE Principles of Knowledge-Based Systems Proceedings, Denver, Colorado, 1984.

Clocksin, W.F. and Mellish, C.S. **Programming in Prolog**. Berlin-Heidelberg-NewYork:Springer-Verlag, 1981.

Dahl, V. and Abramson, H. **On Gapping Grammars.** Proceedings of the Second International Joint Conference on Logic, University of Uppsala, Sweden, 1984.

Evans, R. and Gazdar, G. **The ProGram Manual.** Cognitive Science Programme, University of Sussex, 1984.

Fawcett, B. **personal communication.** Dept. of Computing Science, University of Toronto, 1984.

Gawron, J.M. et.al. **Processing English with a Generalized Phrase Structure Grammar,** pages 74-81. Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics, June, 1982.

Gazdar, G. Phrase Structure Grammar. In P. Jacobson and G.K. Pullum (Ed.), **The Nature of Syntactic Representation,** D.Reidel, Dortrecht, 1981.

Gazdar, G. and Pullum, G.K. **Generalized Phrase Structure Grammar: A Theoretical Synopsis.** Technical Report, Indiana University Linguistics Club, Bloomington Indiana, August 1982.

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

Pereira, F.C.N.(ed). **C-Prolog User's Manual.** Technical Report, SRI International, Menlo Park, California, 1984.

Pereira, F.C.N. and Warren, D.H.D. Definite Clause Grammars for Language Analysis. **Artificial Intelligence,** 1980, 13, 231-278.

Popowich, F. **SAUMER: Sentence Analysis Using MEtaRules (Preliminary Report).** Technical Report TR-84-10 and LCCR TR-84-2, Department of Computing Science, Simon Fraser University, August 1984.

Popowich, F. **The SAUMER User's Manual.** Technical Report TR-85-3 and LCCR TR-85-4, Department of Computing Science, Simon Fraser University, 1985.

Popowich, F. **Effective Implementation and Application of Unrestricted Gapping Grammars.** Master's thesis, Department of Computing Science, Simon Fraser University, forthcoming.

Radford, A. **Transformational Syntax.** Cambridge University Press, 1981.

Schubert, L.K. and Pelletier, F.J. From English to Logic: Context-Free Computation of 'Conventional' Logical Translation. **American Journal of Computational Linguistics,** January-March 1982, 8(1), 26-44.

Shieber, S.M. **Direct Parsing of ID/LP Grammars.** draft, 1982.

Thompson, H. **Handling Metarules in a Parser for GPSG.** Technical Report D.A.I. No. 175, Department of Artificial Intelligence, University of Edinburgh, 1982.