Lexical Functional Grammar (LFG) describes linguistic structure in terms of a Functional structure (F-structure) that may be computed by the evaluation of equations written in the grammar and lexicon. Constraints can be placed on F-structure features to restrict the instantiation or non-instantiation of features and ranges of possible values assigned to features. F-structures must have qualities of coherency and completeness so as to be well-formed. An LFG compiler is briefly described and it is shown how constraints may be compiled so as to improve post-parse checking speed. Coherency and completeness are shown to be re-interpretable as constraints checks in their own right (although this is not explicit in LFG) and thus they may be treated in a way uniform with LFG’s own constraints.

Lexical Functional Grammar (LFG)

LFG is one of the family of approaches to grammar that share the characteristic of using unification as their main information combining mechanism. LFG is made distinctive by having a phrase structure backbone (with the formal power of a Recursive Transition Network (RTN) rather than a Context-Free Grammar (CFG)). This is known as the Constituent structure, abbreviated to C-structure. Equations attached to C-structure rules describe how feature information is to be communicated through the parse tree so as to build a resulting Functional structure (abbreviated to F-structure). The emphasis is on grammatical function.

Equations are attached to lexical entries and are the primary source of values for features and are usually instantiations of features (including long-distance dependencies), but may also be specifications of constraints on the F-structure. The PRED feature is used in verb entries to specify subcategorization information. Equations attached to grammar rules more usually specify the nesting of features, in particular giving nested feature structures for grammatical functions such as SUBJ, OBJ, etc (Shieber, 1985).

The following is a small lexicon from Kaplan and Bresnan (1982, p. 185):
Lexicon 1

a: DET, (↑ SPEC) = A
(↑ NUM) = SG
girl: N, (↑ NUM) = SG
(↑ PRED) = ‘GIRL’
handed: V, (↑ TENSE) = PAST
(↑ PRED) = ‘HAND((↑ SUBJ) (↑ OBJ2) (↑ OBJ))’
the: DET, (↑ SPEC) = THE
baby: N, (↑ NUM) = SG
(↑ PRED) = ‘BABY’
toy: N, (↑ NUM) = SG
(↑ PRED) = ‘TOY’

An equation such as (↑ NUM) = SG may be read as “the NUM feature of the dominating feature structure has the value ‘SG’”.

The following is a small grammar from Kaplan and Bresnan (1982, p. 184):

Grammar 1

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow DET \ N \\
VP & \rightarrow V \ NP \ NP
\end{align*}
\]

The “trivial assignment” equation, \(↑ = ↓\), is read as “the current node is instantiated with the dominated node”. This is the most common equation and is therefore frequently omitted from written grammars.

The F-structure that would be computed for the input *A girl handed the baby a toy* would be:

```
S
  NP
    DET
      SPEC: A
      PRED: 'GIRL'
    NUM: SG
  VP
    V
      TENSE: PAST
      PRED: 'HAND((↑ SUBJ) (↑ OBJ2) (↑ OBJ))'
    NP
      SPEC: THE
      PRED: 'BABY'
      NUM: SG
    NP
      SPEC: A
      PRED: 'TOY'
```
Note how the SUBJ, OBJ and OBJ2 are formed of subsidiary F-structures and that each subsidiary F-structure and the main F-structure itself has its own PRED feature. A well-formed F-structure has to meet the criteria of uniqueness (where a feature can have only one value), completeness and coherency (Kaplan & Bresnan, 1982, pp. 180-183, 203-214). Completeness requires the definition of local completeness. A locally complete F-structure contains all the governable functions that its PRED specifies. (In this example this means that the main F-structure must contain the governable functions, SUBJ, OBJ and OBJ2.) An F-structure is complete if it and all its subsidiary F-structures are complete. Coherence requires the definition of local coherence. A locally coherent F-structure is one in which all the governable functions that the F-structure contains are governed by its local PRED. (In this example this means that the main F-structure must not contain any governable functions other than SUBJ, OBJ and OBJ2.) An F-structure is coherent is it and all its subsidiary F-structures are coherent.

Well-formedness testing is a contentious issue. It is often thought that completeness and coherency can only be tested once the whole F-structure has been built, particularly if long-distance dependencies have played a part in computing the F-structure. From a theoretical point-of-view, LFG has been criticised as having a “cognitively unrealistic” post-parsing process and from an implementational point-of-view, well-formedness checks are frequently carried out in the post-parsing phase in practical implementations of LFG. Some of the following is intended to show that it is possible to move checking in part from post-parsing to a more obviously satisfactory place during parsing.

An LFG Compiler

There were two motivations for building an LFG compiler; speed and correctness. As a broad over-simplification, unification-based formalisms fall into two groups; linguistic grammars and computational grammars. Linguistic grammars allow the expression of sophisticated linguistic descriptions in an elegant fashion but require specially written interpreters to implement parsers. These tend (like programming language interpreters) to run slowly and also require the use of “linguistic unification” algorithms. A computational grammar (eg Definite Clause Grammar) is close to some underlying programming language and so descriptions tend to run quickly. The price paid for the speed is that descriptions of linguistic phenomena have to be coerced into the linguistically unsophisticated language. Computational grammars often have the advantage of being able to use a programming language’s own matching algorithm, of which the most notable example is Definite Clause Grammar’s use of Prolog’s term unification algorithm. The first advantage of using in-built features of a programming language is that the speed built into the programming
language implementation is directly available. The second advantage is that in-built features are more likely to be correct than corresponding algorithms programmed by a user. For example, Prolog’s in-built unification algorithm is more likely to be correct than unification algorithms written for linguistic grammars.

The reasoning that led to building the compiler was that LFG has many of the advantages of a sophisticated linguistic formalism but lacked the advantages of a computational grammar. A compiler should allow a linguist to easily express ideas in the LFG formalism and then compile them into a form that would run quickly and accurately in Prolog (although, of course, there remains the problem of ensuring that the compiler is correct). Prolog is a language that lends itself to writing such compilers; indeed, under the name of meta-interpretation, the activity is a field of study in its own right. The compiler was developed as part of a larger speech understanding system for a publicly available database.

Meta-interpretation is a technique widely used, for example as here, to transform a convenient natural representation of knowledge (LFG rules and lexical entries) into another form more suited to machine processing (cf Van Harmelen, 1989). Meta-interpretation using Prolog was employed in the P-PATR system (Hirsh, 1988) to transform PATR-II grammars into Prolog’s CFG-equivalent formalism, Definite Clause Grammar (DCG). The PATR-II source grammars were unexecutable grammars and the target executable Prolog code. In the BUP system (Matsumoto et al, 1983), the source consisted of executable DCGs that are an inherently top-down search with depth-first control (as is Prolog). The DCGs were transformed into an alternative parsing strategy, the primarily bottom-up Left-Corner algorithm (actually also represented in BUP as DCGs), although control remained depth-first.

Hirsh and (to a lesser extent) Matsumoto et al’s executable target code could have been used as convenient re-representations of the source form of the linguistic knowledge. These re-representations could themselves be used as the rule-base for alternative parsers that might take the form of implementations in Prolog of standard algorithms (eg the Active Chart Parser family) or implementations of Prolog in Prolog designed to give alternative control, eg breadth-first. Such re-representations are useful because they can take advantage of the strengths of the underlying programming language.

In the case of the work being reported, both the execution and re-representation themes were present. LFG grammars and lexicons were written in a form
recognisably similar to “real” LFG (possible by use of \textit{op/3}) and compiled into various forms (some directly executable, some not), depending on the present requirements. The compilation took advantage of Prolog’s in-built term unification; the power of the logic variable to model LFG’s uniqueness requirement, and the convenience of difference lists to collect cumulative information, most notably constraints and long-distance dependency control information.

There were two basic outputs from the compiler. First, DCG rules that were suitable when the C-structure was restricted both in size and in form to a CFG and, secondly, RTNs. The latter were attractive because they allowed common paths to be merged (thus saving parsing time) and because it was easier to represent in particular the Kleene star. The RTN output could be run directly, but was used in conjunction with a primarily bottom-up Active Chart Parser with the intention of gaining the economies of chart parsing (but see Ramsay (1989)) and the ability to match partial constituents for ellipsis and substitution processing.

The LFG compiler had two-stages, with the first stage being a lexical analyser. This collected several types of information, for instance the names of all features employed by the linguist together with each feature’s possible values and all governable functions. For instance, a feature’s possible values would be recorded as a Prolog structure:

\begin{verbatim}
lex_term(aux, [have, there_be]).
\end{verbatim}

The second stage was the code generator, the heart of which compiled each equation in turn and output the compiled version of grammar rules and lexical entries. The essentials were that each feature (eg \textsc{num}) was formed into a structure of an arity of one, eg:

\begin{verbatim}
num(_12345).
\end{verbatim}

All features were organised into a sorted, fixed-length list and all references to a given feature in equations were treated as references to the variable (ie \_12345), thus using the speed of Prolog’s in-built unification and making for a greater likelihood of correctness. Although the equation compiler was generally integrated into compilers for DCGs and RTNs, it was also used within an interpreter for tracing grammars in development, thus ensuring that the interpreter and compiler shared the same semantics.

\textbf{Constraints in LFG}

LFG allows \textit{constraining equations} that constrain a named feature to have a given value in the final F-structure but doesn’t itself instantiate the value of that feature and \textit{existential constraints} that require a named feature to have some value in the final
F-structure without making any statement of what this value might be. Both kinds of constraints can be negated, so it is possible to specify that a feature isn’t instantiated to a particular value or that a given feature doesn’t have an instantiation in the final F-structure.

Examples of these constraints are as follows:

Constraining equation:

\[ (\downarrow \text{AUX}) = c \text{ THERE}_BE \]
\[ (\uparrow \text{Q ADVERB}) = c + \]

Negated constraining equation:

\[ \neg (\downarrow \text{AUX}) = c \text{ HAVE} \]

Existential constraint:

\[ (\uparrow \text{TENSE}) \]

Negated existential constraint:

\[ \neg (\uparrow \text{TENSE}) \]

Implementation of constraints in an LFG Compiler

There were two stages in handling constraints. The first stage was their representation. This was done by simply compiling each constraint attached to a C-structure category into a Prolog structured object with two arguments. The first argument was the variable used as the value for the feature named in the constraint, and the instantiation of the feature would automatically instantiate the variable held in the constraint structure. The second argument represented the possible values that the feature was constrained to take. For instance, consider an example given above:

\[ (\downarrow \text{AUX}) = c \text{ THERE}_BE \]

This would be represented as:

\[ \text{const}(_{12367}, \text{[there be]}). \]
where \_12367 would be the variable which the compiler had assigned to this instance of the feature AUX.

A negated constraining equation was represented in a similar way. In the examples above, there are two instances of the AUX feature. As has been seen, the lexical analyser would the following information about the feature:

\[ \text{lex_term(aux, [have, there be])}. \]

The compiled representation of the equation:
\[ \neg ( \downarrow \text{AUX}) = c \text{ HAVE} \]

would be:

\[
\text{const}(_{53872}, \{\text{there}_\text{be}\}).
\]

where the second argument represents all possible values except that specified in the negated constraining equation.

With an existential constraint, the second argument would be all the possible values collected by the lexical analyser. So if the constraint was:

\[
(\uparrow \text{AUX})
\]

the compiled constraint would be:

\[
\text{const}(_{53877}, \{\text{have, there}_\text{be}\}).
\]

For a negative existential constraint the second argument would be a list with a single member that was an atom constructed by the compiler and which occurred nowhere else as a feature’s value in the grammar and lexicon. In the following example, the compile has generated the atom \text{nec} when compiling the equation:

\[ \neg (\uparrow \text{SUBJ FORM}) \]

into the form:

\[
\text{const}(_{45307}, \{\text{nec}\}).
\]

The second stage was the processing of constraints in parsing. As has been seen above, constraint checking was apparently intended to be a post-parsing process both because in LFG constraints specify how features should be instantiated in the final F-structure and also because of the delayed effects of LFG’s long-distance constituent control. Constraint checking could be a lengthy process depending on how constraints were recorded during parsing. For instance, if constraints were stored with “normal” features and their values, it would be necessary to search through the F-structure to find all constraints (and to recursively search subsidiary F-structures). To speed constraint checking, it was necessary to avoid searching through F-structures for named constraints (especially as features could be nested by the application of functions not adjacent to the constraint attached to the C-structure node). Constraints were placed in difference lists with the intention that they be collected into a list of constraints to be checked after parsing had been completed. Because the structure that recorded a constraint also included the variable associated with the given feature name, it meant that the instantiation of a feature was stored with the list of permissible values. Thus checking became merely a process of examining each member of a difference list and as such was a serial search.
Completeness and coherency in LFG

As has been seen above, LFG includes the concept of governable grammatical functions. A limited number of features may have the status of governable functions by virtue of their being specified in lexical entries. For instance, the verb handed might have a lexical entry that includes the predicate feature:

\[(↑\text{PRED}) = \text{‘hand}〈↑\text{SUBJ} \ (↑\text{OBJ2}) \ (↑\text{OBJ})〉\]

The utterance The girl handed would be ungrammatical because OBJ2 and OBJ wouldn’t be satisfied. An intransitive verb in an utterance such as The girl fell the apple the dog would be ungrammatical because more grammatical functions would be instantiated than would be specified in the lexical entry for fell. (For present purposes, these are referred to as invalid governable functions.)

In summary, an F-structure is complete if there is an instantiated feature corresponding to every governable function specified in its predicate and coherent if no more governable functions occur in the F-structure than is specified in its predicate (Kaplan and Bresnan, 1982, pp. 210-214). The function is deemed to occur if it contains its own PRED feature that is instantiated.

Completeness and coherency in an LFG Compiler

The number of governable functions used in LFG is quite small and the set of their names is a proper subset of the set of feature names. For any given lexicon (and any associated grammar) it is possible to determine the features that act as governable functions by simply listing every feature named in those PRED features that have the form in the above example. This kind of analysis was easily accomplished within the lexical analyser, which included amongst its output a list of all features that could be used as governable functions and also rendered it unnecessary for the grammar writer to specify this information manually (cf Hurt, 1991).

It has been shown that constraints were re-represented so as to improve speed of processing by avoiding search for features in the final F-structure. A similar tactic was used here, eg when:

\[(↑\text{PRED}) = \text{‘hand}〈↑\text{SUBJ} \ (↑\text{OBJ2}) \ (↑\text{OBJ})〉\]

was compiled, each governable function specified (ie SUBJ, OBJ2, OBJ) would be formed into a constraint that had the same structure as for true LFG constraints compiled as described above. The first argument was the variable associated with the nested PRED of the feature name. However, as has been seen, the number of possible features could be very large in number, so the second argument of the compiled constraint is treated merely as an instruction to ensure the PRED had been instantiated.
to some value. This method ensured that completeness could be checked as part of the constraint checking process.

\[
\begin{align*}
\text{const}(_{45208}, \{\text{pred}(_{45200})\}), & \quad \text{const}(45277, \{\text{pred}(_{45203})\}), \\
\text{const}(_{45318}, \{\text{pred}(_{45207})\})
\end{align*}
\]

The constraint checking algorithm had to be written in such a way as to ensure that when the constraint value list was in the form \{\text{pred}(_{45207})\}, a check was made to ensure that the variable associated with the PRED feature (ie _45318) was instantiated to some (unspecified) value.

Coherency checking was more involved. The method devised was to compile constraint structures for each governable function found by the lexical analyser but not specified in a given PRED equation. A valid governable function has an instantiated PRED feature. An invalid governable function would have a feature in the feature data structure, but the feature’s variable would be uninstantiated.

As an example, suppose that the grammar that included the PRED specification given above also included another lexicon entry with the PRED specification:

\[(\uparrow \text{PRED}) = ‘\text{persuade}(\uparrow \text{SUBJ})(\uparrow \text{VCOMP})’\]

When the previous example was compiled, it would have now to have a set of constraints including one that specifically disallowed VCOMP to take a value. The compiled version of this constraint would look like:

\[
\text{const}(_{45326}, \{\text{nec}\}).
\]

The examples given above show that the governable functions specified in the PRED entry of an F-structure can be seen as equivalent to positive existential constraints. Thus positive existential constraints are used to enforce completeness. On the other hand, the specification of invalid governable functions can be seen as equivalent to negative existential constraints. Thus negative existential constraints are used to enforce coherency.

**The positioning of evaluation procedures of compiled constraints**

As has been seen, LFG’s constraints refer to the final F-structure. The obvious point at which constraints could be checked was after parsing was completed and the F-structure computed, and this was the easiest way of programming because of the use of difference lists. In terms of speed, the checking process has the (reversed) speed characteristics of a serial search, in as much as a successful search through the constraints collected in parsing would have to be examine every member of the list, whereas an unsuccessful search would be terminated as soon as a violated constraint was found.
A unification-based grammar formalism emphasises the matching component of a parsing algorithm. Matching is more than the comparison of categories, as it is with simple, single category CFGs. Unification requires the matching of complex feature structures, with the attendant problems of cyclic (and therefore infinite) unification. It is tempting to add a constraint checking process to the matching component of an LFG parser, if only to avoid situations where violated constraints are discovered only long after the constraint was added to the parse record.

There are arguments against on-the-fly checking. Most seriously, LFG’s use of long-distance constituent control to model movement (without recourse to transformations) means that features may have constraints applied to them early in the parsing process, but these features may only be instantiated late in parsing. (This is an issue linked to the separate and more complicated topic of implementing long-distance constituent control, a matter that is, in part, [natural] language dependent.) The other major argument against on-the-fly checking is the amount of time involved in checking and rechecking the constraints at each stage of the matching process. It has been shown that checking constraints has the inverse speed characteristics of a serial search. For a successful analysis, the complete set of constraints would have to be checked at each stage at the matching process. In reply to the argument that LFG grammars contain few constraints, it should be countered that the handling of completeness and coherency in this implementation has moved the burden of these checks into the realm of LFG’s normal constraint checking, although on the other hand these well-formedness checks have to be made in some way or another at some point in parsing.

The argument in favour of on-the-fly checking is best seen in the context of an Active Chart Parser. Each edge in the chart represents a possible contributor to an extension of a parse. The algorithm works by generating pairs of active and inactive edges and then testing their validity. The inclusion of edges that include unsatisfiable constraints would waste considerable parsing time. In addition, it has to be remembered that the implementation using difference lists does allow quick checking speeds in comparison with other implementations that require the (possibly recursive) scanning of F-structures.

An ultimate evaluation of the advantages and disadvantages of on-the-fly checking is not possible unless experiments are conducted, and the conclusions may well be that it is in large part natural language dependent.
Optimisation
As has been stated above, it was necessary to search the entire list of compiled constraints in order to ensure that no constraint was violated and that the average time taken for an ill-formed F-structure to be rejected is half the time taken to scan the list of constraints. It is tempting to attempt to imagine methods of altering compiled constraints in some way so as to gain a better speed performance.

First we will consider the organisation of LFG constraints in lexical entries and grammar rules. The linguist is free to specify more than one constraint on a feature, for instance the redundant:

\[(↑ \text{TENSE}) = \text{c PAST} \]
\[(↑ \text{TENSE})\]

With a grammar and lexicon that included values for these features of past and present, these would be compiled into:

\([\text{const}(_{-12547}, \text{[past]}), \text{const}(_{-12547}, \text{[past, present]}), ...]\]

At compilation these constraints could be merged to form one:

\([\text{const}(_{-12547}, \text{[past]}), ...]\]

Arguably, this example is poor LFG in as much as the constraints are redundant and it is not a major requirement of the compiler to overcome deficiencies in the linguistic writing.

Other reorderings of constraints in lexical entries are possible. It is attractive to imagine ways in which the serial search of constraints could be converted into something with more of the speed character of a sequential search. For instance, the constraints could be ordered in alphabetical order or, more attractively, the constraints that are most discriminating (and thus most likely to fail early) placed first in the difference list of constraints. The common difficulty with both these schemes is that the constraints are collected during parsing in difference lists, which are essentially unordered. Both schemes would require that the chosen order of constraints be preserved during parsing: itself a time-consuming operation. The second option also has the disadvantage of having to decide a criterion by which to order constraints.

During parsing, it is possible for the difference list in which constraints are collected to include two or more constraints on the same feature contributed by different parts of the grammar and lexicon. It is tempting to write the parser in such a way as to allow the merging of common constraints in the hope that the newly merged
constraint would be even more specific. This is a vain hope, in as much as the process of merging constraints would be at least as lengthy as that of scanning the constraints for violations.

There remains one possible optimisation, partly used in the current implementation. Negative existential constraints and, as argued above, the related coherency checks can be optimised by taking advantage of their use of a value that doesn’t appear in the grammar and lexicon. Instead of recording the constraint in the constraint difference list, the variable belonging to the feature to be constrained can be directly instantiated with the compiler-generated value. This ensures that the feature can never be instantiated by any other value. The complication is that any software that displays or manipulates the resulting F-structure has to know what the compiler’s generated value is for that particular grammar and lexicon. There is also the difficulty that optimisations such as these make the target code produced even less similar to the source LFG that means that it is difficult to use the same code generators for both compilers and interpreters/tracers thus increasing the risk that the semantics of compilers and tracers will be different.

A fuller example
The following example is intended to illustrate more clearly the detailed workings of the compilation of constraints. It consists of the lexical entries compiled from Lexicon 1, given above, with the addition of an entry for persuaded as shown in Kaplan & Bresnan (1982, p. 220):

\[
\text{persuaded: } V, \quad (↑ \text{TENSE}) = \text{PAST} \\
(↑ \text{PREDE}) = '\text{PERSUADE}(↑ \text{SUBJ}) (↑ \text{OBJ}) (↑ \text{VCOMP})' \\
(↑ \text{VCOMP TO}) = \text{c +} \\
(↑ \text{VCOMP}) = (↑ \text{OBJ})
\]

Several of the compiled lexical entries are shown. They have been slightly modified to exclude information about long-distance constituent control and debugging information has been included, which means that all features are listed in the form of a name and value structure. Also, for clarity, constraints are shown with the name of the feature that they reference. The first entry is for \textit{a}. The first three arguments represent the word and syntactic category, followed by a list of features. The following two arguments (both being the variable _43074) is the difference list pair for constraints, while the last two arguments are the conventional difference list pair for the input to be consumed.

\[
\text{lexicon}(a, \text{det}, [\text{num(sg)}, \text{obj(_43788)}, \text{obj2(_43789)}, \text{pred(_43790)}], \\
\text{spec(a), subj(_43792), tense(_43793), to(_43794)}, \\
\text{vcomp(_43795)]}, \\
\text{_43783, _43783, [a | _43939], _43939})
\]
The lexical entries for the, girl and baby are in essence the same as for a. The other two lexical entries are for the two verbs of which that for persuaded is shown:

```
lexicon(persuaded,v,[num(_43787)],
  obj([num(_44138),obj(_44139),obj2(_44140),
    pred(_44126),spec(_44142),subj(_44143),
    tense(_44144),to(_44145),
    vcomp(_44146))],
  obj2(nec),
  pred(persuade([up subj,up obj,up vcomp])),
  spec(_43791),
  subj([num(_44096),obj(_44097),obj2(_44098),
    pred(_44068),spec(_44100),subj(_44101),
    tense(_44102),to(_44103),
    vcomp(_44104)]),
  tense(past),
  to(_43794),
  vcomp([num(_44208),obj(_44209),obj2(_44210),
    pred(_44168),spec(_44212),
    subj([num(_44138),obj(_44139),
      obj2(_44140),pred(_44126),
      spec(_44142),subj(_44143),
      tense(_44144),to(_44145),
      vcomp(_44146))],
    tense(_44214),to(_44215),
    vcomp(_44216))],
  _43783,
  [const(vcomp-to,_44215,[+])],
  const(subj(_44121),_44068,[pred(_44060)]),
  const(obj(_44163),_44126,[pred(_44060)]),
  const(vcomp(_44233),_44168,[pred(_44060)]|_43783],
  [persuaded|_44613],_44613).
```

Notice here how the invalid governable functions, such as OBJ2 in the entry for persuaded are represented by the compiler-generated value nec.

**Evaluation and conclusions**

The method of compiling constraints outlined above has the main advantages of simplifying and speeding-up checking algorithms at parse time by obviating the need to check (possibly recursively) the contents of an F-structure. More importantly, it has been shown that the well-formedness conditions of coherency and completeness can themselves be interpreted and represented as LFG constraints, with the implication that the same advantages of speed and simplicity can be achieved. This method is suitable for use in a compiler but less suitable for use in an interpreter/tracer because, as there is no obvious explicit specification of the imposition of coherency and completeness checks given in the written grammar, there is nothing in the grammar to be highlighted in a tracer.
References


---

1 The work described herein was supported by the Speech Research Unit, RSRE (Malvern) under contract D/ER1/9/4/2027/195.

2 This might be used to specify that a particular kind of S clause must have a tensed verb, thus disallowing forms such as *hanging*.

3 The set of names of governable functions is a proper subset of the set of feature names because PRED is not a governable function, but has to be present in the F-structure of a well-formed governed function.