Formal Properties of Feature Grammars

Alan W. Black
Department of Artificial Intelligence
University of Edinburgh

Abstract

This paper describes the formal power of feature grammars such as Generalised Phrase Structured Grammars (GPSG) [1]. It shows how such grammars relate to context-free grammar (Chomsky type 2) and which extensions cause the formalism to increase in power to context-sensitive grammar and beyond. Also, it describes how any Turing machine can be implemented as a feature grammar showing that the power of such grammars is beyond context-free. Finally some discussion is given on the relevance of the these results.

The first point to make, and probably the most important, is a distinction between *grammar* and *formalism*. This distinction is often blurred in computational linguistics. To give an example

```
Grammar(1)

   T = { Det Noun Verb }
   N = { S NP VP }
   S =  S
   P = {
         S --> NP VP
         NP --> Det Noun
         VP --> Verb NP
       }
```

is a particular context-free *grammar*. While the *formalism* for context-free grammars is described as ([8] p 80)

> A context-free grammar is 4-tuple (T, N, S, P). Where T is a finite set of terminal symbols; N is a finite set of non-terminal symbols; S is one non-terminal called the *distinguished symbol*; and P is finite set of production of the form $A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals of length one or more.

> The language defined by a context-free grammar is: all strings of terminal symbols that can be derived from the distinguished symbol by rewritting using productions. Rewriting is defined as, $\alpha A\gamma \Rightarrow \alpha\beta\gamma$ where α and γ are lists of categories and $A \rightarrow \beta$ is a production.

This means we can make comments about a formalism (or *class* of grammars) such parsable in n^3 time (where n is the length of string to be parsed), but we can also make comments about *particular* grammars written in a formalism. For example `Grammar(1)` above has no left recursive rules in it but the context-free grammar formalism it is written in does not forbid such grammars.

Feature grammars are quite common in computational linguistics ([1], [2], [3]) One of the problems in these descriptions is that it is often unclear when the *formalism* is being referred to or just a particular *grammar*. So it is not always possible to tell which extensions are outwith the given formalism.

Suppose we define the following feature grammar formalism which is (hopefully accepted as) a subset of the formalism used to describe GPSGs in [1] and similar formalisms.

> A feature grammar consists of a triple (F, R, S). Where F is a finite set of features consisting of a feature name and an enumerated set of atomic values it can take. R is a finite set of rules of the form $A \rightarrow \alpha$. Where A is a category and α is a string of categories (of length one or more), and a category is a set of features (as defined in F) each with a particular value. S is a category called the *distinguished category* from which all strings in the language of grammar can be derived.
>
> The language defined by a feature grammar is defined as all strings of categories that can be derived from the distinguished category by application of the rules. Given the string $\alpha A\gamma$ where α and γ are strings of categories in the grammar and A is a category, the string $\alpha\beta\gamma$ (where β is a string of categories) can be derived if there is a rule $A \rightarrow \beta$.

An example grammar in the above formalism is as follows

`Grammar(2)`

```
F = { (Type {S NP VP V N D}) }
R = {
      {[Type S]} --> {[Type NP]} {[Type VP]}
      {[Type NP]} --> {[Type D]} {[Type N]}
      {[Type VP]} --> {[Type V]} {[Type NP]}
    }
S = {[Type S]}
```

All grammars in this feature grammar formalism have equivalent context-free grammars (CFGs). This is true because we can treat categories as elaborate names for atomic names in the equivalent CFG. In fact `Grammar(1)` is an equivalent CFG to `Grammar(2)`.

The above formalism can be extended to allow *variables* as feature values in categories in rules, with the condition that the range of each variable must be stated as part of the grammar. That is, a grammar now also has (as well as F, R and S) V which is a set of variables each defined with a range which is an enumerated set of atomic values. (For ease of reading, variables will be written as atoms prefixed with a question mark e.g. ?X, ?fred). Also the definition of derivation has to be extended to cater for variables in rules. Rules with variables can be looked upon as a short-hand for a set of rules with no variables. A mapping can be defined from rules containing variables to sets of rules containing no variables.

> A *rule* containing a variable ?X with the range $\{r_1, r_2, ...r_m\}$ represents a set of rules with m members where each $rule_i$ has ?X substituted (throughout the whole *rule*) for r_i.

The above `Grammar(2)` can now easily be extended to allow person number agreement with the use of a variable.

```
Grammar (3)

    F = { (Type {S NP VP V N D})
          (PN {SING1 SING2 SING3 PLUR}) }

    V = { (?X {SING1 SING2 SING3 PLUR}) }

    R = {
          {[Type S]} --> {[Type NP] [PN ?X]} {[Type VP] [PN ?X]}
          {[Type NP] [PN ?X]} -->
                {[Type D] [PN ?X]} {[Type N] [PN ?X]}
          {[Type VP] [PN ?X]} -->
                {[Type V] [PN ?X]} {[Type NP]}
        }

    S = {[Type S]}
```

Grammars in this extended formalism are still context-free as there is a mapping defined from this extended formalism to the original feature grammar formalism, which was shown to be context-free. The important point that makes the mapping possible is the fact that there is a *finite* number of categories.

We can extend this formalism further to include category-valued features (as in GPSG and FUG). This is done by allowing features and variables to be defined as taking categories as their values, instead of just atomic values. This however means that grammars in the formalism are now potentially no longer context-free.

To illustrate this it can be shown that the feature grammar formalism (including category-valued features) can be used to encode any Turing machine (Chomsky type 0) and hence lies outside the power of context-free grammars (Chomsky type 2).

A Turing machine consists of two parts: an infinite tape, partitioned into squares each of which can contain a finite number of symbols, which are initialised to blanks; and a machine, which consists of a finite number of transitions. A transition consists of a current state, current symbol, new symbol, movement (left or right), and a new state. The interpretation is: at each instance the machine reads the current symbol from the tape and depending on what state it is in, the appropriate transition is applied writing the new symbol to the tape, moving left or right to the next position on the tape and changing to the new state. Church's thesis [4] hypothesizes that any function which is computable can be encoded on such a machine. (for more details see chapter 7 of [8]).

Given any Turing machine we can encode it in our feature grammar formalism as follows. The definitions for features and variables required are as follows

```
Feature             Value
   LeftTape           category-valued
   RightTape          category-valued
   State              each state in the Turing machine
   Symbol             each symbol in the Turing machine

Variable            Range
   ?Left              category-valued
   ?Right             category-valued
   ?NextSym           each symbol in the Turing machine
   ?Symbol            each symbol in the Turing machine
```

For each transition in the Turing machine a rule has to be created. There are two cases to consider, transitions that move left, and those that move right.

Consider a transition `<Q,S,NQ,NS,Left>` where `Q` is the old state, `S` is the current symbol, `NQ` is the new state, `NS` is the new symbol and `Left` the direction, we write a grammar rule as follows

```
{ [LeftTape ?Left]
  [State NQ]
  [Symbol ?NextSym]
  [RightTape { [Symbol NS] [RightTape ?Right] } ] }
```

```
-->

{  [LeftTape { [Symbol ?NextSym] [LeftTape ?Left] } ]
   [State Q]
   [Symbol S]
   [RightTape ?Right] }
```

And similarly each right movement transition such as `<Q,S,NQ,NS,Right>` can be encoded by a rule of the form

```
{ [LeftTape { [Symbol NS] [LeftTape ?Left] } ]
  [State NQ]
  [Symbol ?NextSym]
  [RightTape ?Right] }

-->

{  [LeftTape ?Left]
   [State Q]
   [Symbol S]
   [RightTape { [Symbol ?NextSym] [RightTape ?Right] } ] }
```

Halting of Turing machines can be defined in many equivalent ways, here we define that a Turing Machine halts when it reaches some particular state e.g END. The distinguished category of our "Turing machine grammar" will be defined as

```
{[LeftTape ?Left] [State END] [Symbol ?Symbol] [RightTape ?Right]}
```

where END is the ending state in the Turing machine. Running of the machine is analogous to parsing a string. We can define halting of the Turing machine to be equivalent to successfully finding a parse of a string. Unsuccessful parses and parses that do not terminate should be treated as equivalent to non-halting Turing machines. Note that the string that gets parsed represents the the initial state of the Turing machine and tape. This can be done in one category making the parser work on a string of one category as opposed to longer strings in more conventional parsing. The problem of representing an infinite tape in a category can be overcome by specifying the tape with end-markers, e.g. if the initial state is to be a tape holding `A B C` with the machine in state 0 pointing at the A then we can specify the initial category as

```
{ [LeftTape { [Symbol LEFTEND] [LeftTape {}] } ]
  [State 0]
  [Symbol A]
  [RightTape
```

```
      {[Symbol B]
       [RightTape
          {[Symbol C]
           [RightTape {[Symbol RIGHTEND] [RightTape {}] } ]
          }
       ]
      }
   ]
}
```

We can now have effectively an infinite tape by adding the following two rules to our grammar.

```
{ [LeftTape {[Symbol LEFTEND] [LeftTape {}] } ]
  [State ?Q]
  [Symbol BLANK]
  [RightTape ?Right] }

-->

{ [LeftTape {}]
  [State ?Q]
  [Symbol LEFTEND]
  [RightTape ?Right] }


{ [LeftTape ?Left]
  [State ?Q]
  [Symbol BLANK]
  [RightTape {[Symbol RIGHTEND] [RightTape {}] } ] }

-->

{ [LeftTape ?Left]
  [State ?Q]
  [Symbol RIGHTEND]
  [RightTape {}] }
```

Note that this method of encoding would require a bottom-up parser to run. Running top-down means the parser will look at the required result of the function encoded in the Turing machine and try to find which operations could have achieved that result. Then for each operation that could have achieved that result, find which could have achieved that and so on. This would not be guaranteed to terminate, but even running the parse bottom-up does not guarantee termination for all Turing machines. There are interesting questions raised here which are not at issue at present.

Example

Suppose we have a tape with a number of a s and b s on it terminated by a c. we can define a Turing machine that, starting at the left most symbol, will traverse the tape from left to right changing all b s to a s and stopping when it reaches a c. One Turing machine with this function consists of two states $\{Q_0, Q_1\}$; four symbols $\{a, b, c, BLANK\}$ and three transitions $\{< Q_0, a, Q_0, a, LEFT >, < Q_0, b, Q_0, a, LEFT >, < Q_0, c, Q_1, c, LEFT >\}$. When the machine reaches state Q_1 it halts as there are no other transitions it can take. This Turing machine can be encoded in the above extend feature grammar as follows

```
Grammar(4)

     F = { (LeftTape category)
           (RightTape category)
           (State {Q0 Q1} )
           (Symbol {a b c LEFTEND RIGHTEND BLANK})  }

     V = { (?Left category)
           (?Right category)
           (?State    {Q0 Q1} )
           (?NextSym  {a b c LEFTEND RIGHTEND BLANK})
           (?Symbol   {a b c BLANK}) }

     R = {
            { [LeftTape {[Symbol LEFTEND] [LeftTape {}] } ]
              [State ?State] [Symbol BLANK] [RightTape ?Right] }
            -->
               { [LeftTape {}] [State ?State]
                 [Symbol LEFTEND] [RightTape ?Right] }

            { [LeftTape ?Left] [State ?State] [Symbol BLANK]
              [RightTape {[Symbol RIGHTEND] [RightTape {}] } ] }
            -->
               { [LeftTape ?Left] [State ?State]
                 [Symbol RIGHTEND] [RightTape {}] }

            { [LeftTape {[Symbol a] [LeftTape ?Left]}]
              [State Q0] [Symbol ?NextSym] [RightTape ?Right] }
            -->
               { [LeftTape ?Left] [State Q0] [Symbol a]
                 [RightTape {[Symbol ?NextSym] [RightTape ?Right]}] }
```

```
            { [LeftTape {[Symbol a] [LeftTape ?Left]}]
              [State Q0] [Symbol ?NextSym] [RightTape ?Right] }
            -->
               { [LeftTape ?Left] [State Q0] [Symbol b]
                 [RightTape {[Symbol ?NextSym] [RightTape ?Right]}] }

            { [LeftTape {[Symbol c] [LeftTape ?Left]}]
              [State Q1] [Symbol ?NextSym] [RightTape ?Right] }
            -->
               { [LeftTape ?Left] [State Q0] [Symbol c]
                 [RightTape {[Symbol ?NextSym] [RightTape ?Right]}] }
         }

     S = { [LeftTape ?Left] [Symbol ?Symbol]
           [State Q1] [RightTape ?Right] }
```

If the machine is to run on a tape containing `a b a c` the initial string to be parsed would be the following single category

```
 { [LeftTape { [Symbol LEFTEND] [LeftTape {}] } ]
   [State Q0]
   [Symbol a]
   [RightTape
      {[Symbol b]
       [RightTape
          {[Symbol a]
           [RightTape
              {[Symbol c]
               [RightTape
                  {[Symbol RIGHTEND] [RightTape {}]}]}
          }
       ]
      }
   ]
}
```

It now has been shown that the above formalism is Turing computable, that is, it can encode grammars which are *not* context-free. The point where the formalism went beyond context-free was at the introduction of category-valued features. This changes the number of categories from a *finite* number to an *infinite* number, because when category-valued features are allowed, a category can hold a copy of itself in its value. This problem can be overcome though, the extension that should be made is not to allow *any* category-valued features, but either *enumerate* them or *never* allow a category-valued feature to have itself within its value, or the values of its values, and so on recursively. It is this condition that makes the makes GPSG still context-free (p36 of [1]), though actual parsers themsleves rarely make this stipulation.

So What ?

This argument is very similar to Ritchie's proof of Functional Unification Grammar (FUG) being Turing computable [6]. So much so, that GPSG and FUG (and possibly Lexical Functional Grammar (LFG)) appear to be equivalent formalisms. That is, for every grammar in one formalism there exists a grammar in the other that describes the same language. It is not, then surprising that systems like PATR-II [7] and HUG [5] exist in which parsers for all these grammar formalisms can be written.

As was stated above properties can not only be made about the *formalisms* but also about particular *grammars* defined within formalisms. `Grammar(1)` is context-free but is also a valid member of the class of grammars described by the fully extend version of the feature grammar formalism. So we can still have *grammars* that have the property of being context-free, written within *formalisms* that allow more powerful grammars.

One question suggested is, is context-free a desirable property anyway? The usual reason for desiring context-free grammars is to ensure parsing is tractable. Most, if not all, grammars written in GPSG are parsed without expanding them into their equivalent context-free grammars (e.g. [9]) but effectively expand them as required during parsing. It has been pointed out that the equivalent context-free grammar for a reasonable GPSG has of the order of hundreds or millions of rules. Formally strings in context-free languages can be parsed in n^3g^2 time (where n is the length of the string and g is the number of grammar rules [10]). But indexing into such a large number of rules is currently impractical for most computing systems. When the grammar is not expanded the complexity of parsing is a lot worse in that the above function is on the number of context-free rules not the number of feature grammar rules.

The other more important property that is desirable of grammars is *decidability*, that is given any string, can the parser say it is in the language described by the grammar, or say it is not. This unfortunately is not even true of all context-free grammars as a context-free grammar parser given the grammar

```
Grammar(5)

    S --> S
    S --> a
```

and the string `a` would not terminate, as there is an infinite number of parses.

In conclusion, it has been shown where feature grammar formalisms can allow more powerful grammars than context-free, but this does not necessarily mean that particular grammars written in such formalisms an not context-free.

References

[1] Gazdar G., Klein E., Pullum G., and Sag I. *Generalised Phrase Structure Grammar*. Blackwell, 1985.

[2] Kaplan R. and Bresnan J. *Lexical-Functional Grammar: A Formal System for Grammatical Representation* in The Mental Representation of Grammatical Relations, MIT Press, Cambridge 1982.

[3] Kay M. *Functional Grammar* Proceedings Fifth Annual Meeting of Berkeley Linguistics Society, Berkeley, CA 1979.

[4] Church A. *An unsolvable problem in elementary number theory*. American Journal of Mathematics 58, 345-363, 1936.

[5] Karttunen L. *HUG: A development environment for unification-based grammars*. unpublished paper, SRI International, Menlo Park, California, 1985.

[6] Ritchie G. *Simulating a Turing Machine using Functional Unification Grammar*, in Proceedings of sixth European Conference on Artificial Intelligence, pp 127-136, 1984.

[7] Sheiber S., Uszkoriet H., Pereira F., Robinson, and Tyson M. *The Formalism and Implementation of PATR-II*. In Research on Interactive Acquisition and Use of Knowledge, SRI International, Menlo Park, California, 1983. 1985.

[8] Hopcroft J. and Ullman J. *Introduction to Automata Theory, Languages and Computation* Addison-Wesley, 1979.

[9] Phillips J. and Thompson H. *A Parser for Generalised Phrase-Structure Grammar*. to appear in Working Papers in Cognitive Science ed. Ewan Klein & Nick Haddock, 1986.

[10] Earley J. *An Efficient context-free parsing algorithm*. CACM 13: 94-102, 1970.