Parsing with Logical Variables

T. W. Finin and M. Stone Palmer

1 Introduction

Logic based programming systems have enjoyed increasing popularity in applied AI work in the last few years. One of the contributions that the logic programming paradigm has made to computational linguistics is the *Definite Clause Grammar* (DCG). An excellent introduction to this formalism can be found in [9] in which the authors present the formalism and make a detailed comparison to Augmented Transition Networks (ATN) as a means of both specifying a language and parsing sentences in that language.

We feel that the major strengths offered by the DCG formalism arise from its use of *logical variables* with *unification* as the fundamental operation on them. These techniques can be abstracted from the theorem proving paradigm and adapted to other parsing systems (see [8] and [2]). We have implemented an experimental ATN system which treats ATN registers as logic variables and provides a unification operation over them.

The DCG formalism provides a powerful parsing mechanism based on a context free grammar. The grammar rule

$$S \rightarrow NP\ VP$$

can be seen as the universally quantified logical statement,

$$\forall x\ \forall y\ \forall z:\ NP(x) \wedge VP(y) \wedge Concatenate(x,y,z) \rightarrow S(z).$$

in which the variables x, y, and z range over strings of words, NP and VP are predicates which are true just in the case their arguments are a string of words representing a noun phrase and verb phrase, respectively, and Concatenate(x,y,z) is true if word strings x and y can be concatenated to form word string z. Prolog, a programming language based on predicate calculus, allows logical statements to be input as Horn clauses in the following (reversed) form:

```
s(Z):- np(X), vp(Y), concatenate(X,Y,Z).
```

The resolution theorem prover that "interprets" the Prolog clauses would take the negation of s as the goal and try and produce the null clause. Thus the preceding clause can be interpreted procedurally as, "To establish goal s, try and establish subgoals np, vp, and concatenate." DCGs provide syntactic sugar on top of Prolog so that the arrow can be reversed and the concatenate predicate can be dispensed with. The words in the input string are examined sequentially each time a

"[Word]" predicate is executed which implicitly tests for concatenation (see Fig. 1). DCGs allow grammar rules to be expressed very cleanly, while still allowing ATN-type augmentation through the addition of arbitrary tests on the contents of the variables.

Pereira and Warren [9] argue that the DCG formalism is well suited for specifying a formal description of a language and also for use with a parser. In particular, they assert that it is a significant advance over an ATN approach on both philosophical and practical grounds. Their chief claims are that:

1. DCGs provide a common formalism for theoretical work in computational linguistics and for writing efficient natural language processors.
2. The rule-based nature of a DCG results in systems of greater clarity and modularity.
3. DCGs provide greater freedom in the range of structures that can be built in the course of analyzing a constituent. In particular, the DCG formalism makes it easy to create structures that do not follow the structure implied by the rules of a constituent and to create a structure for a constituent that depends on items not yet encountered in the sentence.

The first two points have been discussed in the past whenever the ATN formalism is compared with a rule-based grammar (see [10], [7], [3], or [1]). The outcome of such discussions varies. It is safe to say that how one feels about these points depends quite heavily on stylistic considerations and past experience in using the two formalisms.

We find the third point to be well founded, however. It is clear that the DCG differs most from previous rule based parsing systems in its inclusion of *logical variables*. The use of logical variables results in greater flexibility in building structures to represent constituents that do not follow the inherent structure determined by the rules themselves. It also allows one to create structures which refer to items that have not yet been discovered in the course of analyzing the sentence.

We have built an experimental ATN system which can treat ATN registers as logical variables and, we feel, capture these important strengths offered by the DCG formalism in the otherwise standard ATN formalism.

The next section of this paper gives a more detailed description of DCGs and presents a simple grammar. In the third section we show an ATN grammar which is "equivalent" to the DCG grammar and discuss the source of its awkwardness. The fourth section then presents an ATN formalism extended to include viewing ATN registers as logical variables which are subject to the standard unification operation. The final section draws a parallel between DCGs and *lexical functional grammars* (LFGs) and suggests that logical variables might be fruitfully introduced into other parsing algorithms and systems.

2 Definite Clause Grammars

Figure 1 shows a simple DCG grammar adapted from [9]. Figure 2 gives a sentence in the language recognized by this grammar together with the associated surface syntactic structure and the semantic structure built by the grammar.

```
/* let & be an infix operator signifying ∧ */
:- op(1000,xfy,'&').

s(Sentence) --> np(X, Verbform, Sentence), vp(X, Verbform).

np(X, Verbform, Sentence) --> det(X, Nounpred, Verbform, Sentence),
                              n(X, Noun),
                              relclause(X, Noun, Nounpred).

np(X, Sentence, Sentence) --> name(X).

relclause(X, Noun, (Noun & Relverb)) --> [that], vp(X, Relverb).
relclause(X, Noun, Noun) --> [].

det(X,Nounpred,Verbform,forAll(X,(Nounpred -> Verbform))) --> [every].
det(X,Nounpred,Verbform,forSome(X, (Nounpred & Verbform))) --> [a].

vp(X, Verbform) --> transv(X, Y, Partverb), np(Y, Partverb, Verbform).

vp(X, Verbform) --> intransv(X, Verbform).

n(X, man(X)) --> [man].
n(X, woman(X)) --> [woman].
n(X, dog(X)) --> [dog].

name(john) --> [john].
name(mary) --> [mary].
name(fido) --> [fido].

transv(X, Y, loves(X,Y)) --> [loves].
transv(X, Y, breathes(X,Y)) --> [breathes].

intransv(X, loves(X)) --> [loves].
intransv(X, lives( X)) --> [lives].
intransv(X, breathes(X)) --> [breathes].
```

Fig. 1. A simple definite clause grammar

Sentence

```
"john loves every woman who breathes"
```

Syntactic Structure

```
(S (NP (NAME john))
   (VP (TRANSV loves)
       (NP (DET every)
           (NOUN woman)
           (REL (VP (INTRANSV breathes))))))
```

Semantic Representation

forAll(X_1, (woman(X_1) & breathes(X_1)) --> loves(john,X_1))

Fig. 2. A sentence, structure, and representation

2.1 The DCG Notation

For readers who may be unfamiliar with this notation for DCGs, we will briefly summarize it here. Each rule in a DCG is of the form:

nt→body.

where *nt* is a non-terminal symbol and *body* is a sequence of one or more terms separated by commas. This specifies a rule in a context free grammar in which the left hand side corresponds to *nt* and the right hand side corresponds to *body*. Note the following important extensions to the standard notation for context free grammars:

- Terminal symbols (i.e., words) in the right hand side of a rule must be enclosed in square brackets, as in the following rule:

  s→[please], vp.

- Non-terminals can be compound terms rather than simple atomic symbols and can include variables. The variables are treated as logical variables (i.e., they may only take on a single value) with a scope equal to the rule in which they appear. We will follow the standard Edinburgh Prolog convention and consider any symbol beginning with a capital letter to be a variable. For example:

  s(s(Np. VP))→nounPhrase (Np), verbPhrase (Vp).

- Procedural tests may be included in the right hand side of a rule by enclosing them in braces. These tests are executed when the terms to their left have been satisfied and cause the rule to fail if they do not succeed. For example, the following rule

  noun(W)→[W]. {rootform(W,Root), category (Root, noun)}.

  might be used to recognize a word as a noun given a data base consisting of the *rootform* relation which links words to their root forms (e.g., rootform(dogs, dog)) and the *category* relation which links root forms to their syntactic categories (e.g., category (dog, noun)).

2.2 Interpreting DCG Rules as Definite Clauses

In order to use a DCG rule it is first transformed into a definite clause. In the previous section we briefly outlined how the *concatenate* predicate could be used to ensure that the constituents on the right hand side of a rule span contiguous portions of the input. A somewhat more efficient technique is to augment the predicates for non-terminals with additional arguments which enforce the contiguity constraint. Non-terminals corresponding to predicates of arity n and are translated into predicates of arity $n+2$. The two extra arguments are appended to the end of the argument list and represent the words in the input *just before* and *just after* the constituent corresponding to the non-terminal. For example, the following two rules:

s(s(Np, Vp))→nounPhrase(Np), verbPhrase(Vp).
noun (Word)→[Word], {rootform(Word, Root), category (Root, noun)}.

become these two definite clauses:

s(s(Np, Vp), X_0, X_1):- nounPhrase (Np, X_0, X_2). verbPhrase (Vp, X_2, X_1).
noun (Word, [Word|X_0], X_0):- rootform (Word, Root), category (Root, noun).

where X_0, X_1, and X_2 are the system generated names of variables.

In order to find a parse corresponding to a DCG non-terminal *nt* of arity *n* it is merely necessary to satisfy the new goal *nt'* formed by appending two extra arguments to *nt:* the list of words to be parsed and the empty list. For example, to find a sentence corresponding to the non-terminal *s(X)* given the words *"the dog chased the cat"* we attempt to prove the goal *S(X, [the, dog, chased, the, cat], []).* If we use a conventional Prolog interpreter to satisfy the top level goal, then the grammar rules will be applied in a top-down, left to right manner. Alternatives will be explored via backtracking.

2.3 An Example in More Detail

Let us turn our attention to how the grammar in Figure 1 discovers a parse for the example sentence in Figure 2. The way in which unification produces the appropriate bindings for this example is actually quite subtle, and requires a detailed analysis. In this grammar, the first determiner encountered in the sentence selects a quantification form for the entire sentence. In the example sentence *"John loves every woman who breathes"* the determiner *every* selects

forAll (X, Nounpred→Verbform)

for the overall sentence form. Had the example been *"John loves a woman who breathes",* the determiner *a* would choose

forSome (X, Nounpred & Verbform)

for the overall sentence form.

Parsing the rest of the sentence will produce predicate representations that will be slotted into the appropriate places in the sentence pattern. The sentence, *"Every woman breathes,"* would produce *"women (X_1)"* for the variable *Nounpred* and *"breathes (X_1)"* for the variable *Verbform,* resulting in a sentence representation of

forall (X_1, woman (X_1)→ breathes (X_1)).

This is explained in more detail below, using the example, "John loves every woman who breathes." This discussion is illustrated by the diagram in Figure 3. In trying to prove the *s(Senctence)* goal, the only applicable rule is:

s (Sentence)→np(X, Verbform, Sentence), vp (X, Verbform).

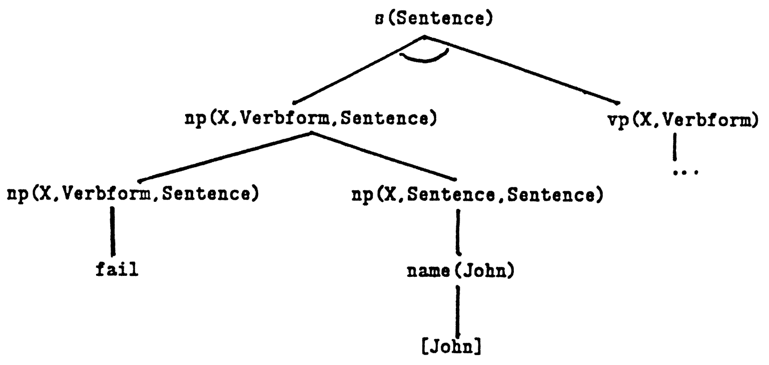

Fig. 3. Finding the subject noun phrase

This rule causes *np(X, Verbform, Sentence)* to be set up as the first subgoal. There are two *np* clauses that match this subgoal:

```
np(X, Verbform, Sentence)→ det(X, Nounpred, Verbform, Sentence).
                           n(X, Noun),
                           relclause (X, Noun, Nounpred).
np(X, Sentence)→           name (X).
```

the first of which eventually fails since "John" is not preceded by a determiner. The second one succeeds, binding *X* to *John,* and identifying the original *Verbform* variable with the original *Sentence* variable. The assumption is that the logical form produced for the verb will be the logical form for the entire sentence.

The second major subgoal in attempting to instantiate a sentence is *vp(X, Verbform),* where *X* is now bound to *John.* Once again there are two possible clauses:

```
vp(X, Verbform)→transv(X, Y, Partverb), np(Y, Partverb, Verbform).
vp(X, Verbform)→intransv(X, Verbform).
```

only one of which will succeed, as Figure 4 shows.

Two new subgoals are set up, *transv(John, Y, Partverb)* and *np(Y, Partverb, Verbform).* The first one succeeds since "loves" is a transitive verb, and results in Partverb being bound to **loves(John, Y),** where *Y* is still unbound. The binding for *Y* is expected to be produced by the successful evaluation of the next subgoal, *np(Y, Partverb, Verbform).* Figure 4 shows the status of the parse at this point.

The variable *Partverb* is also passed as a parameter, so that it can be included in the *Verbform* structure. This time the first *np* clause is applicable, and the presence of the determiner "every" results in the *Verbform* variable, now renamed as *Sentence,* being bound to **forAll(Y, Nounpred→loves (John, Y)).** This is the complete *Verbform* structure, which for this sentence is also the *Sentence* structure (Figure 5). It does, however, still have some unbound variables, namely *Y* and *Nounpred,* which will be filled in as the rest of the noun phrase is processed. The binding for Nounpred will be supplied during the evaluation of the other two subgoals

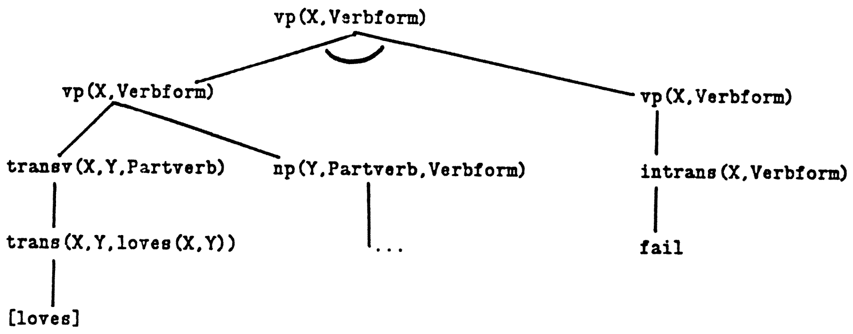

Fig. 4. Finding the verb phrase

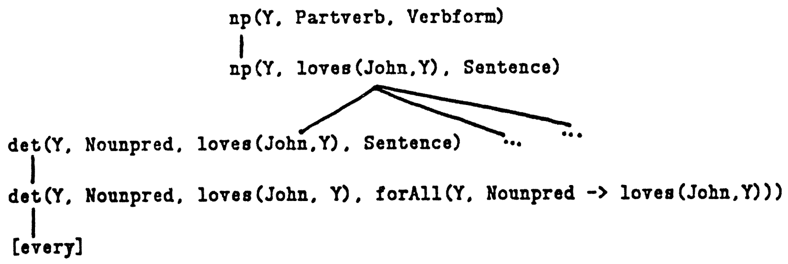
np(Y, Partverb, Verbform)
np(Y, loves(John,Y), Sentence)
det(Y, Nounpred, loves(John,Y), Sentence)
...
...
det(Y, Nounpred, loves(John, Y), forAll(Y, Nounpred -> loves(John,Y)))
[every]

Fig. 5. Determining the sentence form

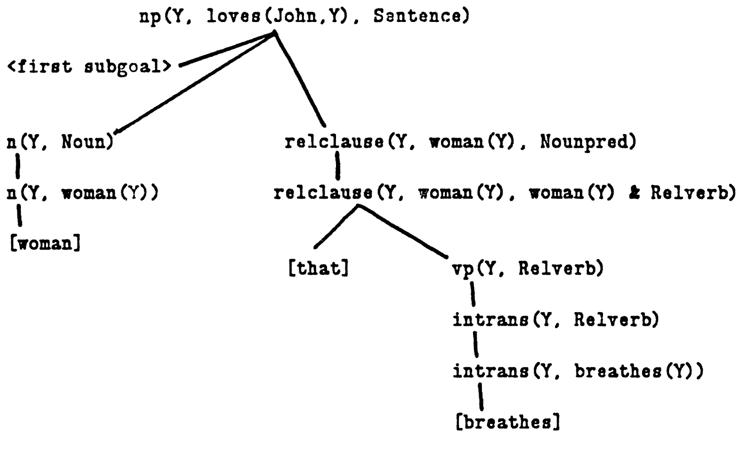

Fig. 6. Finding the relative clause

associated with *np, noun* and *relclause*. The evaluation of the noun goal produces **women(Y)** which is passed to *relclause* as something to be included in *Nounpred.* The final binding for *Nounpred* is **woman(Y) & breathes(Y)** (Figure 6). The variable Y stays unbound, since it is universally quantified.

The result of this processing is that the *relclause* goal is satisfied, with Nounpred bound to:

woman (Y) & breathes (Y)

which caused the *np* goal be completed with *Sentence* bound to:

forAll (Y, woman (Y) & breaths (Y)→loves (John, Y))

which satisfies the *vp* goal with *Verbform* bound to:

forAll (Y, woman (Y) & breaths (Y)→loves (John, Y))

and yields a satisfied s goal with the ultimate binding for *Sentence* of:

forAll (Y, woman (Y) & breaths (Y)→loves (John, Y))

In following the application of this grammar it becomes clear that very strong predictions are made about which parts of the parse will be supplying particular types of information. Determiners provide the quantifiers for the propositional structure of the sentence. The first noun phrase and the noun phrase following the verb are the two participants in the predicate implied by the verb. Obviously the rules given here are part of a toy grammar, but the power of the logical variables can only be made use of through the encoding of these strong linguistic assumptions. DCGs provide a mechanism well qualified for expressing such assumptions and then executing them. The same type of assumptions can be found in LFGs as discussed in the conclusion.

3 Comparing DC and ATN Grammars

Figure 7 shows an ATN grammar which is the "equivalent" of the DCG grammar given in Figure 1. The format used to specify the grammar is the one described in [4] and [5]. There are only two minor ways in which this particular formalism differs from the standard ATN formalism described in [11] or [1]. First, the dollar sign character (i.e., $) followed by the name of a register stands for the contents of that register. Second, the function DEFATN defines a set of arcs, each of which is represented by a list whose first element is the name of the state and whose remaining elements are the arcs emanating from the state.

In addition, this example uses a very simple lexical manager in which a word has (1) a set of syntactic categories to which is belongs (2) an optional set of features, and (3) an optional root form for the word. These attributes are associated with a word using the function LEX, which supplies appropriate default values for unspecified arguments. Thus the expression

(LEX dogs n (animate (number plural)) dog) could be used to indicate that the word dogs belongs to the syntactic category *n* (for noun), has the two features *animate* and *number plural,* and has the word *dog* as its root word.

One immediate difference between the DCG and ATN formalism has to do with communication between different levels of processing. In the DCG formalism, the application of each rule involves a different level of processing. The arguments to the predicates representing a non-terminal serve as communication channels between the levels. As is standard in logic programming languages, each of these variables can be used either for input or output or both.

The standard ATN model adopts a more functional approach. A PUSH arc invokes a sub-computation which takes no arguments and, if successful, returns a single value. One can achieve the effect of passing parameters to a sub-computation by giving a register an initial value via a SENDR action. There are two methods by which one can achieve the effect of returning more than one value from a sub-computation. The first is for the sub-computation to POP with a value which is a list of the items to be passed back to the higher level. Actions on the invoking PUSH arc would then be responsible for decomposing the list into its constituent items. The second method is to use the LIFTR action to directly set registers in the higher level computation. The grammar in Figure 7 makes use of SENDR and LIFTR to pass parameters into and out of ATN computations and thus mimic the actions of the DCG example.

Consider what must happen when looking for a noun phrase. The representation for an NP will be a *predicate* if the noun phrase is indefinite (i.e., "a man" becomes (man X))[1] or a *constant* if the noun phrase is a name (i.e., "John" becomes John). In this simple language, a NP is dominated by a either a sentence (if it is the subject) or by a verb phrase (if it is the object). In either case, the NP determines, or must agree with, the overall structure used to represent the dominating constituent. If the NP is definite[2], then it exerts no additional constraint on the representation of its dominator. If the NP is indefinite it will eventually result in a quantified expression for the dominating sentence or verb phrase. In this case we need to tell the dominating computation what the *predicate, quantifier, connective,* and *variable* name must be. In this ATN grammar, this is done by having the NP network **return** a value to represent the NP predicate and **lift** values for the *quantifier, connective,* and *variable names.*

Similarly, when we are looking for a verb phrase, we must know what token (i.e., variable name or constant) represents the subject (if the verb phrase is dominated by an S) or the head noun (if the verb phrase acts as a relative clause). This is done by sending the **subjvar** register in the sub-computation the appropriate value via the SENDR function. The techniques used to handle quantification and build an overall sentence structure in this ATN grammar are similar to those used in the BBN Lunar Grammar [12].

This heavy use of SENDR and LIFTR to communicate between levels in the grammar violates many accepted programming maxims and results in an ATN

[1] Note that we are switching to a lispy syntax for representing the meanings.
[2] Which in this toy grammar reduces to it being a name.

```
(defatn

 (s (push np t (setr subj *) (to s/subj)))

 (s/subj (push vp t (setr vp *) (sendr subjvar $var) (to s/end)))

 (s/end (pop (list $quant $var (list $connect $subj $vp)) $subj)
        (pop $vp (null $subj)))

 (np (wrd a t (liftr quant 'ForSome) (liftr connect 'And)(to np/det))
     (wrd every t (liftr quant 'ForAll)  (liftr connect '→) (to np/det))
     (cat name t (setr var *) (to np/np)))

 (np/det (cat n t (setr var (gensym)) (setr n (list * $var)) (to np/n)))

 (np/n (wrd (who that which) t (to np/n/who))
       (jump np/np t))

 (np/np (pop $n t (liftr var)))

 (np/n/who
  (push vp t (sendr subjvar $var) (setr n (list 'And $n *)) (to np/np)))

 (vp (cat v t (setr v *) (to vp/v)))

 (vp/v (push np (getf trans $v) (setr obj *) (setr objvar $var) (to vp/vp))
       (pop '(list $v $subjvar) (getf intrans $v)))

 (vp/vp (pop (list $quant $objvar (list $connect
                                         $obj
                                         (list $v $subjvar $objvar)))
             $obj)
        (pop (list $v $subjvar $objvar) (null $obj))))

; The Lexicon
; (lex <word> <category> <features> <rootform>)

(lex man n)
(lex woman n)
(lex loves v (intrans trans))
(lex breathes v (intrans trans))
(lex lives v (intrans))
(lex John name)
  ... etc ...
```

Fig. 7. An equivalent ATN grammar

grammar which is cumbersome and difficult to understand. In the next section we investigate treating ATN registers as logic variables and providing a unification operation on them.

4 Replacing ATN Registers with ATN Variables

Although the previous ATN grammar does the job, it is clearly awkward. We can achieve much of the elegance of the DCG example by treating the ATN registers as **logical variables** and including a unification operation on them. We will call such registers *ATN variables.*

Since our ATN variables must not be tampered with between unifications, assignment operations such as SETR, LIFTR, and SENDR are precluded. Thus the only operations on ATN Registers are **access** and **unify.** It is possible to provide operations similar to the standard SENDR and LIFTR by defining unification operations which do the unification in another environment, but we have not explored these possibilities.

The scheduler component of the ATN parser has been modified to be sensitive to the success or failure of attempted unifications. If a unification operation on an arc fails, the are is blocked and may not be taken.

Figure 8 shows a grammar in the extended ATN formalism. A symbol preceded by a "$" represents an ATN Variable and "*" will again stand for the current constituent. Thus in the state S in the grammar:

```
(S (PUSH NP (UNIFY '($SUBJVAR $VP $S) *) (TO S/SUBJ)))
```

the parser pushes to the state NP to parse a noun phrase. If one is found, it will pop back with a value which will then be unified with the expression ($SUBJVAR $VP $S). If this unification is successful, the parser will advance to state S/SUBJ. If it fails, the arc is blocked causing the parser to backtrack into the NP network. Figure 9 shows the results of applying this ATN grammar to a number of simple sentences.

Although our grammar succeeds in paralleling the behavior of the DCG, there are some open questions involving the use of unification in parsing natural languages. An examination of this ATN grammar shows that we are really using unification as a method of passing parameters. The full power of unification is not needed in this example since the grammar does not try to find "most-general unifiers" for complicated sets of terms. Most of the time it is simply using unification to bind a variable to the contents of another variable. The most sophisticated use involves binding a variable in a term to another copy of that term which also has a variable to be bound as in the "a man loves a woman" example in Figure 9. But even this binding is a simple one-way application of standard unification. It is not clear whether this is due to the simple nature of the grammars involved or whether it is an inherent property of the directedness of natural language parsing.

A situation where full unification might be rquired would arise when one is looking for a constituent matching some partial description. For example, suppose we were working with a syntactic grammar and wanted to look for a singular noun phrase. We might do this with the following PUSH arc:

```
(defatn

 (s (push np (unify '($subjvar $vp $s) *) (to s/subj)))

 (s/subj (push vp t (unify '$vp *) (to s/s)))

 (s/s (pop $s t))

 (np (wrd a t (unify '$np '(ForSome $var (And $pred $hole))) (to np/det))
     (wrd every t (unify '$np '(ForAll $var (→ $pred $hole))) (to np/det))
     (cat name t (unify '$np '$hole) (unify '$var *) (to np/np)))

 (np/det (cat n t (unify '$var (gensym)) (unify '$pred '(* $var)) (to np/n)))

 (np/n (wrd (who that which) t (to np/n/who))
       (jump np/np t))

 (np/np (pop (list $var '$hole $np) t ))

 (np/n/who
  (push vp t (unify '$subjvar '$var) (unify '$pred '(And $pred *)) (to np/np)))

 (vp (cat v (getf trans *) (unify '$v '(* $subjvar $objvar)) (to vp/vtrans))
     (cat v (getf intrans *) (unify '$v '(* $subjvar)) (to vp/vp)))

 (vp/vtrans (push np t (unify '($objvar $v $vp) *) (to vp/vp)))

 (vp/vp (pop $vp t))
```

Fig. 8. An equivalent ATN grammar with ATN variables

```
(PUSH NP T (UNIFY * '(NP (DET $DET) (NUMBER SINGULAR) (ADJ
$ADJS) . . .))
. . .)
```

If we follow the usual schedule of interpreting ATN grammars the unification will not occur until the NP network has found a noun phrase and popped back with a value. This would require a fully symmetric unification operation since there are variables being bound to values in both arguments. It is also highly inefficient since we may know immediately that the noun phrase in the input is not singular. What we would like is to be able to do the unification just after the push is done, which would more closely parallel a Prolog-based DCG parse. Then an attempt to "unify" the number register with anything other than singular would fail immediately.

This could be done automatically if we constrain a network to have only one state which does a pop and place some additional constraints on the forms that can be used as values to be popped. Although we have not explored this idea at any length, it appears to lead to some interesting possibilities.

```
"John loves every woman who breathes"

    (ForAll X₁ (→ (And (woman X₁) (breathes X₁)) (loves John X₁)))


"John loves a woman"

    (ForSome X₁ (And (woman X₁) (loves John X₁)))


"a man loves a woman"

    (ForSome X₁
            (And (man X₁)
                 (ForSome X₂ (And (woman X₂) (loves X₁ X₂)))))


"every man who lives loves"

    (ForAll X₁ (→ (And (man X₁) (lives X₁)) (loves X₁)))


"every man who loves mary loves a woman who loves John"

    (ForAll X₁
            (→ (And (man X₁) (loves X₁ mary))
               (ForSome X₂ (And (And (woman X₂) (loves X₂ John))
                                (loves X₁ X₂)))))


"every man who loves a woman who loves every dog loves every dog"

    (ForAll X₁
      (→ (And (man X₁)
              (ForSome X₂
                (And (And (woman X₂)
                          (ForAll X₃ (→ (dog X₃) (loves X₂ X₃))))
                     (loves X₁ X₂))))
         (ForAll X₄ (→ (dog X₄) (loves X₁ X₄)))))
```

Fig. 9. Example parses with the ATN grammar

5 Conclusions

The particular DCG example we have used in this paper illustrates the similarity between the use of logical variables in DCGs and the evaluation of equations in *lexical functional grammars* (LFGs). The passing of logical variables in the DCG from the *det* clause to the *np* clause to the *vp* clause corresponds to equating the variables from different parts of the syntactic structure being built by an LFG. LFGs also produce functional representations of the sentence at the same time as a syntactic parse is being built, another important similarity. However, they do not build the quanification into the functional structure in the way illustrated by the DCG example. This is done later by the semantic component. Also, in the current implementation of LFG, the functional structure is actually built after the syntactic parsing is completed, but a new implementation designed to build the structures in parallel is in progress.

The DCG example is really a toy example based on Montague grammars. DCGs are a powerful tool for encoding grammar rules that support a particular theory. DCGs do not represent a linguistic theory in themselves, whereas LFGs do. The purpose of this comparison is to simplify the explanation of equating variables in LFGs, and to indicate the suitability of DCGs for implementing this type of approach.

LFGs have two components: 1) grammar rules with associated equations for building up a functional description of the sentence and 2) the lexical rules. The lexical rules consist of rules for the individual lexical entries of words as well as redundancy rules that take advantage of cross-word generalizations such as active-passive forms of verbs. The grammar rules are context free rules that generate several possible syntactic structures, many of which are not grammatical. The resolution of the equations associated with the grammar rules is intended to eventually exclude the ungrammatical structures. The equations contain variables that must be filled in by information supplied by the lexical entries. There are arrows associated with the equations as well as with the lexical entries to guide the instantiation of the variables. The LFG grammar rule in Figure 10 is the standard one for doubly transitive verbs:

GRAMMAR RULE:

```
VP --> Verb (      NP              (       NP       ))
       ↑ ↓  ((↑ Object₁)=↓      ((↑ Object₂)=↓))
```

LEXICAL ENTRY:

```
handed VERB (↑ Tense) = Past
  (↑ Predicate) = 'Hand<(↑ Subject),(↑ Object₁),(↑ Object₂)>
```

Fig. 10. An LFG rule

VP→Verb NP NP

and the equation associated with it indicates which components the NPs will instantiate in the function structure, namely the Object$_1$ and Object$_2$.

The arrow notation has a somewhat involved explanation. LFGs first generate a tree from the context free grammar rules, filling in the leaf nodes with the appropriate lexical items. Every node on the tree is then assigned a variable. The arrows give directions for the instantiation of the variables. Sometimes instantiations are passed upwards in the tree, and sometimes downwards. Tense, for instance is passed up to the parent of VERB node. Not surprisingly, Object$_1$ will eventually get instantiated with the value of the NP corresponding to the direct object of the sentence, and Object$_2$ will eventually be given the value of the indirect object. Object$_1$ and Object$_2$ are the arguments to the verb predicate, and in the final functional description, the verb predicate itself will be assigned a unique identifier, say *f3,* and the Object$_1$ of that predicate will be assigned another unique identifier, *f4.* The use of an equation such as ($\uparrow$Object$_1$) $= \downarrow$ can now be demonstrated, since (*f3* Object$_1$) $=$ *f4* represents just that information, namely, that the Object$_1$ of *f3* is *f4.* The upward arrow indicated that *f3* was the node immediately above in the tree, and the downward arrow indicated that *f4* is associated with the node the equation is associated with.

The point to be made here is simply that the logical variables in DCGs can be used to accomplish the same end as the arrow notation. The example of a DCG parse showed how logical variables could be passed values from other parts of the parse. The DCG also used context free grammar rules to parse a clause while building up a functional structure at the same time. The main difference is that unification offers an advantage in providing a capability for instantiating variables while in the middle of the parse, instead of waiting until the parse is finished to "resolve the equations." The arrows in the original equation, ($\uparrow$ Object$_1$) $= \downarrow$, can be replaced by logical variables such as *F3* and *F4.* These variables would have to be explicitly bound to the variables assigned to the nodes in the tree. Then, when the node variables are assigned unique identifiers, those same identifiers would automatically instantiate the variables in the equations. An extra pass to "resolve" the equations would no longer be necessary.

We have found the use of logical variables and unification to be a powerful technique in parsing natural language. It is one of the main sources of the strengths of the Definite Clause Grammar formalism. In attempting to capture this technique for an ATN grammar we have come to several interesting conclusions. First, the strength of the DCG comes as much from the skillful encoding of linguistic assumptions about the eventual outcome of the parse as from the powerful tools it relies on. Second, the notion of logical variables (with unification) can be adapted to parsing systems outside of the theorem proving paradigm. We have successfully adapted these techniques to an ATN type parser and are beginning to embed them in an existing parallel bottom-up parser [6]. The similarities between a DCG and the LFG formalism have been sketched out and we have sugested that DCGs might be a good vehicle for implementing an LFG system. Third, the full power of unification may not be necessary to successfully use logical variables in natural language parsers.

References

1. Bates, M.: Theory and Practice of Augmented Transition Network Grammars. In: Bolc, L. (ed.): Natural Language Communication with Computers. Berlin-Heidelberg-New York: Springer 1978
2. Bossie, S.: A Tactical Component for Text Generation: Sentence Generation Using a Functional Grammar. Technical report MS-CIS-1982-26. Computer and Information Science, University of Pennsylvania, 1982
3. Codd, E.F., Arnold, R.S., Cadiou, J-M., Chang, C.L., and Roussopoulos, N.: RENDEZVOUS Version 1: An Experimental English-Language Query Formulation System for Casual Users of Relational Data Bases. Report RJ2144. IBM Research Laboratory, San Jose, January, 1978
4. Finin, T.: An Interpreter and Compiler for Augmented Transition Networks. Technical report T-48. Coordinated Science Laboratory, University of Illinois, 1977
5. Finin, T.: Parsing with ATN Grammars. In: Bolc, L. (ed.): Data Base Question Answering Systems. Berlin-Heidelberg-New York: Springer 1982
6. Finin, T. and Webber, B.: BUP - A Bottom Up Parser. Technical Report MS-CS-83-12. Department of Computer and Information Science, University of Pennsylvania, 1983
7. Heidorn, G.: Augmented Phrase Structure Grammar. TINLAP-1, Theoretical Issues in Natural Language Processing, August, 1975
8. Kay, M.: Functional Grammar. Proceedings of the Fifth Annual Meeting of the Berkeley Linguistic Society, Berkeley Linguistic Society, Berkeley, CA, 1979
9. Pereira, F. and Warren, D.: Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. Artificial Intelligence 13, 231-289 (1980)
10. Pratt, V.: LINGOL, A Progress Report. Proceedings of the 4th International Joint Conference on Artificial Intelligence, IJCAI, 4 (1975)
11. Woods, W.: Transition Network Grammars for Natural Language Analysis. Communications of the ACM 13 (10), 591-606 (1970)
12. Woods, W.A., Kaplan, R.M., and Webber, B.L.: The Lunar Sciences Natural Language Information System: Final Report. Report 2378. Bolt, Beranek, and Newman, Inc., Cambridge, Mass., 1972