Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks

Fernando C. N. Pereira and David H. D. Warren

Department of Artificial Intelligence, University of Edinburgh

Recommended by Daniel G. Bobrow and Richard Burton

ABSTRACT

A clear and powerful formalism for describing languages, both natural and artificial, follows from a method for expressing grammars in logic due to Colmerauer and Kowalski. This formalism, which is a natural extension of context-free grammars, we call "definite clause grammars" (DCGs).

A DCG provides not only a description of a language, but also an effective means for analysing strings of that language, since the DCG, as it stands, is an executable program of the programming language Prolog. Using a standard Prolog compiler, the DCG can be compiled into efficient code, making it feasible to implement practical language analysers directly as DCGs.

This paper compares DCGs with the successful and widely used augmented transition network (ATN) formalism, and indicates how ATNs can be translated into DCGs. It is argued that DCGs can be at least as efficient as ATNs, whilst the DCG formalism is clearer, more concise and in practice more powerful.

1. Introduction

The aims of this paper are:

(1) to give an introduction to "definite clause grammars" (DCGs)—a formalism, originally described by Colmerauer (1975), in which grammars are expressed as clauses of first-order predicate logic, providing a natural generalisation of context-free grammars;

(2) to explain how DCGs constitute effective programs of the programming language Prolog, and how they can thereby be used to implement practical systems for language analysis;

(3) to compare DCGs with the augmented transition network (ATN) formalism,

and to show how an ATN can be translated into a DCG. It is NOT our intention to propose any definite solutions to the many unsolved linguistic problems of particular languages such as English; we describe only how DCGs *can* be used, not how they *should* be used. We take an informal approach wherever possible. We start by reviewing some basic concepts, making clear our terminology in the process.

The usual way one attempts to make precise the definition of a **language**, whether it is a natural language or a programming language, is through a collection of **rules** called a **grammar**. (Following normal usage, we restrict the term "grammar" to language definitions of this kind.) The rules of a grammar define which strings of words or symbols are valid **sentences** of the language. In addition, the grammar generally gives some kind of **analysis** of the sentence, into a **structure** which makes its meaning more explicit.

A fundamental class of grammar is the **context-free grammar** (CFG), familiar to the computing community in the notation of "BNF" (Backus–Naur form). In CFGs, the words, or basic symbols, of the language are identified by **terminal** symbols, while categories of **phrases** of the language are identified by **non-terminal** symbols. Each rule of a CFG expresses a possible form for a non-terminal, as a sequence of terminals and non-terminals. The analysis of a string according to a CFG is a **parse tree**, showing the constituent phrases of the string and their hierarchical relationships.

An important idea, due to Colmerauer and Kowalski (cf. Kowalski, 1974b; Colmerauer, 1975), is to translate the special purpose formalism of CFGs into a general purpose one, namely first-order predicate logic. They devised a particular method (having its origins in Colmerauer's (1970) Q-systems) for expressing context-free rules as logic statements of a restricted kind, known as **definite clauses** or "Horn clauses". The problem of recognising, or **parsing**, a string of a language is then transformed into the problem of proving that a certain theorem follows from the definite clause axioms which describe the language.

These ideas might only have been of theoretical interest. However, at the same time, Colmerauer and Kowalski originated a more far-reaching idea. This was that a collection of definite clauses can be considered to be a **program** (see Kowalski (1974a, 1974b); van Emden (1975).) It turns out that automatic deduction can exhibit all the characteristics we associate with effective computation, provided the deduction is pursued in a suitably goal-directed way.

A practical realisation of this concept of "programming in logic" was developed by Colmerauer and his colleagues in the form of the programming language Prolog. (See Roussel (1975), Pereira et al. (1978).) Prolog is based on a very simple but efficient proof procedure. Several implementations of the language have been completed, and these implementations have shown that Prolog can be as efficient as conventional high-level programming languages, cf. Warren et al. (1977). Prolog has been successfully used to write large-scale programs for a number of useful applications, including algebraic "symbol crunching" (Bergman and Kanoui, 1975), architectural design (Markusz, 1977), drug design (Darvas et al., 1977) and compiler implementation (Warren, 1977a, 1977b).

Now if a CFG is expressed in definite clauses according to the Colmerauer–Kowalski method, and executed as a Prolog program, the program behaves as an efficient top-down parser for the language the CFG describes.[1] This fact becomes particularly significant when coupled with another discovery—that the technique for translating CFGs into definite clauses has a simple generalisation, resulting in a formalism far more powerful than CFGs, but equally amenable to execution by Prolog. This formalism—the main subject of our paper—we call **definite clause grammars** (DCGs). DCGs are a special case of Colmerauer's (1975) "metamorphosis grammars", which are for Chomsky type-0 grammars what DCGs are for CFGs. Although metamorphosis grammars can be translated into definite clauses, the correspondence is not nearly so direct as that for DCGs.

DCGs are a natural extension of CFGs. As such, DCGs inherit the properties which make CFGs so important for language theory: the possible forms for the sentences of a language are described in a clear and modular way; it is possible to represent the recursive embedding of phrases which is characteristic of almost all interesting languages; there is an established body of results on CFGs which is very useful in designing parsing algorithms.

Now it is well known that CFGs are not fully adequate for describing natural language, nor even many programming languages. DCGs overcome this inadequacy by extending CFGs in three important ways.

Firstly, DCGs provide for context-dependency in a grammar, so that the permissible forms for a phrase may depend on the context in which that phrase occurs in the string. Secondly, DCGs allow arbitrary tree structures to be built in the course of the parsing, in a way that is not constrained by the recursive structure of the grammar; such tree structures can provide a representation of the "meaning" of the string. Thirdly, DCGs allow extra conditions to be included in the grammar rules; these conditions make the course of the parsing depend on auxiliary computations, up to an unlimited extent.

DCGs, as implemented via Prolog, have been used to write a number of practical systems for language analysis, e.g. for natural language question answering (Dahl, 1977), and in compiler implementation (Warren, 1977b).

DCGs bear some similarities to other formalisms known to computer scientists, notably the "van Wijngaarden grammars" used in the Algol-68 Report (van Wijngaarden, 1974), and the "affix grammars" which Koster (1971) took as the basis for the compiler definition language CDL. Like a van Wijngaarden grammar, a DCG can be viewed as a grammar consisting of an infinite number of context-free rules. Like an affix grammar, a DCG extends a CFG by augmenting non-terminals with arguments. However the three formalisms have significant

[1]The efficiency of the parser also depends on a "suitable" choice of CFG to describe the language.

differences; it seems fair to say that both van Wijngaarden grammars and affix grammars can be viewed as special cases of DCGs.

In this paper we shall be specifically concerned with comparing DCGs with a formalism which at first sight is less obviously similar, namely "augmented transition networks" (ATNs). ATNs were introduced by Woods (1970) as a powerful and practical formalism for natural language analysis. They have been used to implement a number of working natural language systems (Woods et al., 1972, 1976; Bates, 1975; Burton, 1976), and some efficient implementations of the formalism have been developed (Burton and Woods, 1976; Finin and Hadden, 1977). For the reader not familiar with ATNs, we recommend Bates (1978) as a clear and thorough introduction.

We have chosen ATNs for comparison because they are widely known, because they are often considered to represent the "state of the art" in formalisms for practical natural language analysis, and because some of the most interesting natural language systems have been written within the ATN formalism. We shall argue that DCGs can be at least as efficient as ATNs, whilst the DCG formalism is clearer, more concise and in practice more powerful.

The paper begins with a concise introduction to logic as a programming language and to Prolog. We recommend the reader to skim through this section and refer back to it later. The next section explains in detail the basic DCG formalism. This is followed by an account, for the ATN-minded reader, of how ATNs can be translated into DCGs. Finally we give a detailed discussion of the advantages of DCGs relative to ATNs, and conclude with a summary of why we think DCGs represent a significant advance. The appendices contain a full example of the translation of an ATN into a DCG, and also some DCG performance data obtained using our DECsystem-10 implementation of Prolog.

Note that, in describing various formalisms, we shall often use bold-face symbols as meta- or syntactic variables. These symbols are NOT part of the formalism under discussion, but are a device which helps to make our description of the formalism shorter and more precise.

2. Logic as a Programming Language—The Definite Clause Subset

In this section, we define the syntax and semantics of a certain subset of logic ("definite clauses"), which amounts essentially to a dropping of disjunction ("or") from the logic, and we indicate how this subset forms the basis of the practical programming language known as Prolog. Definite clauses have also been called "Horn clauses" or "regular clauses", but we prefer the name coined by van Emden (1975), since it gives at least some indication of their nature. We describe the definite clause subset from a conventional programming standpoint, using the notation and terminology of Prolog.

2.1. Syntax, terminology and informal semantics

2.1.1. *Terms*

The data objects of the language are called **terms.** A term is either a **constant,** a **variable** or a **compound term.**

The constants include **integers** such as:

0 1 999

and **atoms** such as:

a void = := 'Algol-68' []

The symbol for an atom can be any sequence of characters, which in general must be written in quotes unless there is no possibility of confusion with other symbols (such as variables, integers). As in conventional programming languages, constants denote definite elementary objects.

Variables will be distinguished by an initial capital letter, e.g.

X Value A A1

A variable should be thought of as standing for some particular but unidentified object. Note that a variable is not simply a storage location which can be assigned to, as in most programming languages; rather it is a local name for some data object, cf. the variable of pure Lisp and identity declarations in Algol-68.

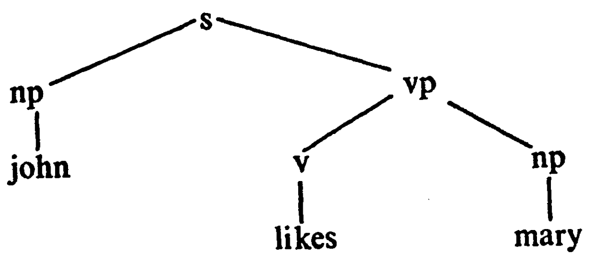
The structured data objects of the language are the compound terms. A compound term comprises a **functor** (called the **principal** functor of the term) and a sequence of one or more terms called **arguments.** A functor is characterised by its **name,** which is an atom, and its **arity** or number of arguments. For example the compound term whose functor is named 'point' of arity 3, with arguments X, Y and Z, is written:

point(X,Y,Z)

One may think of a functor as a record type and the arguments of a compound term as the fields of a record. Compound terms are usefully pictured as trees. For example, the term:

s(np(john),vp(v(likes),np(mary)))

would be pictured as the structure:



Sometimes it is convenient to write a compound term using an optional infix notation, e.g.

    X+Y (P;Q) X<Y

instead of:

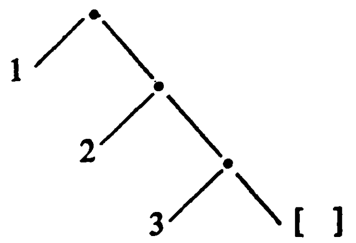
    +(X,Y) ;(P,Q) <(X,Y)

Note that we consider an atom to be a functor of arity 0.

An important class of data structures are the **lists**. These are essentially the same as the lists of Lisp. A list either is the atom:

    []

representing the empty list, or is a compound term with functor '·' and two arguments which are respectively the head and tail of the list. Thus a list of the first three natural numbers is the structure:



This would be written in standard syntax as:

    ·(1,·(2,·(3,[])))

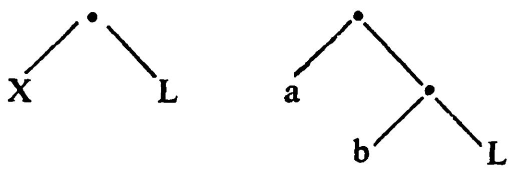
but we shall write it, using a special list notation, as:

    [1,2,3]

Our notation when the tail of a list is a variable is exemplified by:

    [X | L] [a,b | L]

representing respectively:



2.1.2. *Clauses*

A fundamental unit of a logic program is the **goal** or **procedure call**. Examples are:

    gives(tom,apple,teacher) reverse([1,2,3],L) X<Y

A goal is merely a special kind of term, distinguished only by the context in which it appears in the program. The (principal) functor of a goal is called a **predicate**. It corresponds roughly to a procedure name in a conventional programming language.

A logic **program** consists simply of a sequence of statements called **clauses**. A clause comprises a **head** and a **body**. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (i.e., it too may be empty).

If neither the head nor the body of the clause is empty, we call it a **non-unit clause**, and write it in the form:

P :- Q, R, S.

where **P** is the head goal and **Q**, **R** and **S** are the goals which make up the body. We can read such a clause either **declaratively** as:

"**P** is true if **Q** and **R** and **S** are true."

or **procedurally** as:

"To satisfy goal **P**, satisfy goals **Q**, **R** and **S**."

If the body of the clause is empty, we call it a **unit clause**, and write it in the form:

P.

where **P** is the head goal. We interpret this declaratively as:

"**P** is true."

and procedurally as:

"Goal **P** is satisfied."

Finally, if the head of the clause is empty, we call the clause a **question** and write it in the form:

?- P, Q.

where **P** and **Q** are the goals of the body. Such a question is read declaratively as:

"Are **P** and **Q** true?"

and procedurally as:

"Satisfy goals **P** and **Q**."

Clauses generally contain variables. Note that variables in different clauses are completely independent, even if they have the same name – i.e., the "lexical scope" of a variable is limited to a single clause. Each distinct variable in a clause should be interpreted as standing for an arbitrary value. To illustrate this, we give some examples of clauses containing variables, with possible declarative and procedural readings:

(1) employed(X) :- employs(Y,X).
"For any X and Y, X is employed if Y employs X."
"To find whether X is employed, find a Y that employs X."

(2) derivative(X,X,1).
"For any X, the derivative of X with respect to X is 1."
"The goal of finding a derivative for the expression X with respect to X itself is satisfied by the result 1."

(3) ?- ungulate(X), aquatic(X).
"Is it true of any X, that X is an ungulate and X is aquatic?"
"Find an X which is both an ungulate and aquatic."

In a logic program, the **procedure** for a particular predicate is the sequence of clauses in the program whose head goals have that predicate as principal functor. For example, the procedure for a ternary predicate 'concatenate' might well consist of the two clauses:

```
concatenate([X | L1],L2,[X | L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

where 'concatenate(L1,L2,L3)' means "the list L1 concatenated with the list L2 is the list L3".

As we have seen, the goals in the body of a clause are linked by the operator ' , ' which can be interpreted as conjunction ("and"). For convenience, we sometimes also use an operator ' ; ' standing for disjunction ("or"). (The precedence of ' ; ' is such that it dominates ' , ' but is dominated by ' :- '). An example is the clause:

```
grandfather(X,Z) :-
  (mother(X,Y); father(X,Y)), father(Y,Z).
```

which can be read as:

"For any X, Y and Z,
X has Z as a grandfather if
either the mother of X is Y or the father of X is Y,
and the father of Y is Z."

Such uses of disjunction can always be eliminated by defining an extra predicate — for instance the previous example is equivalent to:

```
grandfather(X,Z) :- parent(X,Y), father(Y,Z).
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

— and so disjunction will not be mentioned further in the following, more formal, description of the semantics of clauses.

2.2. Declarative and procedural semantics

The semantics of definite clauses should be fairly clear from the informal interpretations already given. However, it is useful to have a precise definition. The

declarative semantics of definite clauses tells us which goals can be considered true according to a given program, and is defined recursively as follows.

> A goal is **true** if it is the head of some clause instance and each of the goals (if any) in the body of that clause instance is true, where an **instance** of a clause (or term) is obtained, by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

For example, if a program contains the preceding procedure for 'concatenate', than the declarative semantics tells us that:

```
concatenate([a],[b],[a,b])
```

is true, because this goal is the head of a certain instance of the first clause for 'concatenate', namely,

```
concatenate([a],[b],[a,b]) :- concatenate([],[b],[b])
```

and we know that the only goal in the body of this clause instance is true, since it is an instance of the unit clause which is the second clause for 'concatenate'.

Note that the declarative semantics makes no reference to the sequencing of goals within the body of a clause, nor to the sequencing of clauses within a program. This sequencing information is, however, very relevant for the **procedural semantics** which Prolog gives to definite clauses. The procedural semantics defines exactly how the Prolog system will execute a goal, and the sequencing information is the means by which the Prolog programmer directs the system to execute his program in a sensible way. The effect of executing a goal is to enumerate, one by one, its true instances. Here then is an informal definition of the procedural semantics.

> To **execute** a goal, the system searches for the first clause whose head **matches** or **unifies** with the goal. The **unification** process (Robinson, 1965) finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then **activated** by executing in turn, from left to right, each of the goals (if any) in its body. If at any time the system fails to find a match for a goal, it **backtracks**, i.e., it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

For example, let us consider the question:

```
?- concatenate(X,Y,[a,b])
```

which can be read declaratively as:

"Are there lists X and Y which when concatenated yield the list [a,b]?"

If we execute the goal expressed in this question, we find that it matches the head of the first clause for 'concatenate', with X instantiated to [a | X1]. The new variable X1 is constrained by the new goal (or recursive procedure call) which is produced:

```
        concatenate(X1,Y,[b])
```

Again this goal matches the first clause, instantiating X1 to [b | X2], and yielding the new goal:

```
        concatenate(X2,Y,[])
```

Now this goal will only match the second clause, instantiating both X2 and Y to []. Since there are no further goals to be executed, we have a solution:

```
        X = [a,b]
        Y = []
```

i.e., a true instance of the original goal is:

```
        concatenate([a,b],[],[a,b])
```

If we reject this solution, backtracking will generate the further solutions:

```
        X = [a]     Y = [b]
        X = []      Y = [a,b]
```

in that order, by re-matching, against the second clause for 'concatenate', goals already solved once using the first clause.

2.3. Notable features of logic programs

The simplicity of the syntax and semantics of logic programs conceals a number of notable features not found in conventional programming languages. These are discussed in Warren et al. (1977). Here we list briefly those features which are especially relevant to grammar writing.

(1) Pattern matching (unification) replaces the conventional use of selector and constructor functions for operating on structured data.

(2) The arguments of a procedure can serve, not only for it to receive one or more values as input, but also for it to return one or more values as output. Procedures can thus be "multi-output" as well as "multi-input".

(3) The input and output arguments of a procedure do not have to be distinguished in advance, but may vary from one call to another. Procedures can thus be "multi-purpose".

(4) Procedures may generate (via backtracking, in the case of Prolog) a set of alternative results. Such procedures are called "non-determinate". Backtracking amounts to a high-level form of iteration.

(5) Procedures may return "incomplete" results, i.e. the term or terms returned as the result of a procedure may contain variables, which are only filled in later, by calls to other procedures. The effect is similar to the use of assignment in a conventional language to fill in fields of a data structure. Note, however, that

there may be many occurrences of an uninstantiated variable, and that all of these get filled in simultaneously (in a single step) when the variable is finally instantiated. Note also that when two variables are unified together, they become identified as one. The effect is as though an invisible pointer, or reference, linked one variable to the other. We refer to these related phenomena as the "logical variable".

(6) "Program" and "data" are identical in form. A procedure consisting solely of unit clauses is closer to an array, or table of data, in a conventional language.

3. How to Write Grammars in Logic

In this section we describe the formalism of definite clause grammars. The basic idea has been discussed by Kowalski (1974b) and Colmerauer (1975) has given a fully formal treatment.

3.1. Expressing context-free grammars in definite clauses

To describe how grammars can be expressed in logic, we begin by considering context-free grammars (CFGs). For these, we use the following notation, which will prove convenient later.

Each rule has the form:

nt → body.

where **nt** is a **non-terminal symbol** and **body** is a sequence of one or more items separated by commas. Each item is either a non-terminal symbol or a sequence of **terminal symbols**. The meaning of the rule is that **body** is a possible form for a phrase of type **nt**. A non-terminal symbol is written as a Prolog atom, while a sequence of terminals is written as a Prolog list, where a terminal may be any Prolog term. The null string is written as the empty list '[]'. As in the syntax of clauses, this basic notation is extended by allowing alternatives to appear in **body**. Alternative sequences of symbols are separated by semi-colons, with parentheses where necessary.

We now show a simple CFG to illustrate the notation. The grammar covers sentences such as "John loves Mary" and "Every man that lives loves a woman":

```
sentence → noun_phrase, verb_phrase.
noun_phrase → determiner, noun, rel_clause.
noun_phrase → name.
verb_phrase → trans_verb, noun_phrase.
verb_phrase → intrans_verb.
rel_clause → [that], verb_phrase.
rel_clause → [].
determiner → [every].
determiner → [a].
noun → [man].
noun → [woman].
name → [john].
name → [mary].
trans_verb → [loves].
intrans_verb → [lives].
```

We regard each rule of a CFG as "syntactic sugar" for a definite clause of logic. To get the translation, we associate with each non-terminal a 2-place predicate (having the same name). The arguments of the predicate represent the beginning and end points in the string of a phrase for that non-terminal. The first seven rules in the example translate into:

```
sentence(S0,S) :- noun_phrase(S0,S1), verb_phrase(S1,S).
noun_phrase(S0,S) :- determiner(S0,S1), noun(S1,S2), rel_clause(S2,S).
noun_phrase(S0,S) :- name(S0,S).
verb_phrase(S0,S) :- trans_verb(S0,S1), noun_phrase(S1,S).
verb_phrase(S0,S) :- intrans_verb(S0,S).
rel_clause(S0,S) :- connects(S0,that,S1), verb_phrase(S1,S).
rel_clause(S,S).
```

We can read the first clause as "a sentence extends from S0 to S if there is a noun phrase from S0 to S1 and a verb phrase from S1 to S"; we can read the last clause as "a relative clause extends from S to S", i.e., "a relative clause may be empty".

To represent terminal symbols in rules, we use a 3-place predicate, 'connects', where 'connects(S1,T,S2)' means "terminal symbol T lies between points S1 and S2 in the string". Thus the remaining rules translate into:

```
determiner(S0,S) :- connects(S0,every,S).
determiner(S0,S) :- connects(S0,a,S).
noun(S0,S) :- connects(S0,man,S).
noun(S0,S) :- connects(S0,woman,S).
name(S0,S) :- connects(S0,john,S).
name(S0,S) :- connects(S0,mary,S).
trans_verb(S0,S) :- connects(S0,loves,S).
intrans_verb(S0,S) :- connects(S0,lives,S).
```

The first clause, for instance, reads "there is a determiner from S0 to S if the word "every" lies between S0 and S".

Now, to represent a particular sentence to be recognised, say:

```
 Every man that lives loves Mary .
1     2   3    4     5     6    7
```

we tag the sentence with integers as shown, and translate it into the following set of unit clauses:

```
connects(1,every,2).
connects(2,man,3).
connects(3,that,4).
connects(4,lives,5).
connects(5,loves,6).
connects(6,mary,7).
```

Then to determine whether that sentence is grammatical, we try to prove the goal:

```
?- sentence(1,7).
```

The proof procedure used determines the parsing strategy, cf. Kowalski (1974b). This will be discussed further in Section 3.4 with particular reference to the Prolog proof procedure.

We may now notice that the representation of a context-free grammar by clauses is *data-independent*, in the sense that the actual representation of the string to be parsed is not "known" by the clauses—only the predicate 'connects' and the goal to be proved take it into account. If we tag a point in a string, not by an integer, but instead by the list of symbols occurring after that point in the string, it is no longer necessary to provide a separate 'connects' clause for each symbol in the string. Instead we can *define* the 'connects' predicate in a single, general clause:

```
connects([W | S],W,S).
```

which can be read as "The string position labelled by the list with head W and tail S is connected by symbol W to the string position labelled S". The goal of proving the original sentence grammatical is now expressed as:

```
?- sentence([every,man,that,lives,loves,mary],[]).
```

Depending on the proof procedure, one representation or the other may be preferred, for efficiency reasons. In the case of Prolog, the proofs with the two representations will be essentially the same, with integer tags substituted by final segments of the input string.

Note that in cases where the second representation is used, it is possible to execute, or "preprocess", all the calls to 'connects' at "compile-time", thereby dispensing with any need to refer to the predicate at "run-time". For example, preprocessing in this way the clause:

```
rel_clause(S0,S) :- connects(S0,that,S1), verb_phrase(S1,S).
```

we get:

```
rel_clause([that | S1],S) :- verb_phrase(S1,S).
```

Note that in Colmerauer (1975), grammar rules are directly identified with definite clauses of this preprocessed form, and there is therefore no mention of the 'connects' predicate.

The foregoing discussion allows us to identify context-free rules with definite clauses of a certain form. A context-free grammar is thus identified with a set of such clauses.

3.2. Definite clause grammars

We now generalise context-free grammars, in a way that will maintain the correspondence with definite clauses, to obtain the formalism of definite clause grammars.

3.2.1. *Notation*

The notation for DCGs extends our notation for context-free grammars in the following way:

(1) **Non-terminals** are allowed to be compound terms in addition to the simple atoms allowed in the context-free case, e.g.

np(X,S) sentence(S)

(2) In the right-hand side of a rule, in addition to non-terminals and lists of terminals, there may also be sequences of **procedure calls**, written within the brackets '{' and '}'. These are used to express extra conditions which must be satisfied for the rule to be valid, e.g.,

noun(N) → [W], {rootform(W,N), is_noun(N)}.

The last example can be read as "a phrase identified as the noun N may consist of the single word W, where N is the root form of W and N is a noun".

Non-terminals, terminals and procedure calls in the right-hand side of a rule will be referred to collectively as **goals.**

3.2.2. *The meaning of the DCG notation as definite clauses*

A rule of a DCG is again no more than "syntactic sugar" for a certain kind of definite clause. Terminal symbols are translated exactly as before; a non-terminal of arity N translates into an N + 2 place predicate (having the same name), whose first N arguments are those explicit in the non-terminal and whose last two arguments are as in the translation of a context-free non-terminal; procedure calls in the right-hand side of a rule are simply translated as themselves. For example, the rule:

noun(N) → [W], {rootform(W,N), is_noun(N)}

represents the clause:

noun(N,S0,S) :- connects(S0,W,S), rootform(W,N), is_noun(N).

3.3. The use of definite clause grammars

We now discuss how the DCG formalism provides for three important mechanisms in language analysis, namely the building of structures (such as parse trees), the imposing of extra conditions on the constituents of a phrase, and a general treatment of context dependency.

3.3.1. *Building structures*

The extra arguments of non-terminals provide the means of building structure in grammar rules. As non-terminals are "expanded", by matching against grammar rules, structures are progressively built up in the course of the unification process.

Here we just present a simple example. The context-free grammar of Section 3.1 is modified to produce explicitly for each phrase an interpretation which is simply its parse tree. We also take the opportunity of introducing a more compact, and (as we shall see later) more efficient, way of representing the **dictionary**, i.e., the rules defining those non-terminals which correspond to word classes (or parts of speech). In general, instead of having a rule of the form:

category(arguments) → [word].

for each **word** in the class **category**, we write a general rule:

category(arguments) → [W], {cat(W, arguments)}.

and define a "dictionary procedure" **cat** consisting of clauses of the form:

cat(word, arguments).

for each word in **category**.

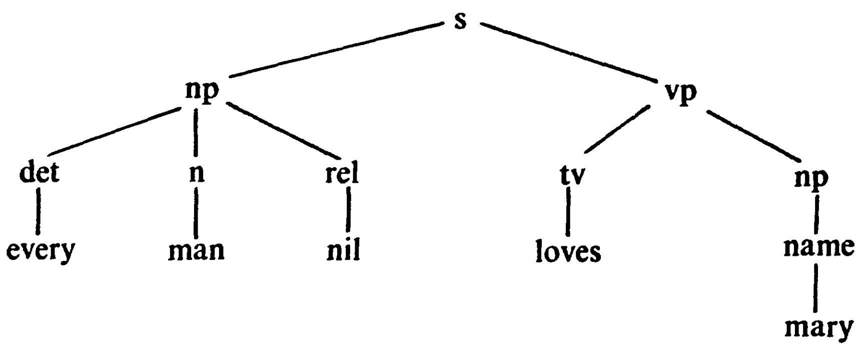
The rules for the modified example are:

```
sentence( s(NP,VP) ) → noun_phrase(NP), verb_phrase(VP).
noun_phrase( np(Det,Noun,Rel) ) → determiner(Det), noun(Noun),
  rel_clause(Rel).
noun_phrase( np(Name) ) → name(Name).
verb_phrase( vp(TV,NP) ) → trans_verb(TV), noun_phrase(NP).
verb_phrase( vp(IV) ) → intrans_verb(IV).
rel_clause( rel(that,VP) ) → [that], verb_phrase(VP).
rel_clause( rel(nil) ) → [].
determiner( det(W) ) → [W], {is_determiner(W)}.
noun( n(W) ) → [W], {is_noun(W)}.
name( name(W) ) → [W], {is_name(W)}.
trans_verb( tv(W) ) → [W], {is_trans(W)}.
intrans_verb( iv(W) ) → [W], {is_intrans(W)}.
```

We read an augmented non-terminal such as 'noun-phrase(NP)' as "a noun phrase with interpretation NP". Thus the first rule is read as "A sentence with interpretation s(NP,VP) may consist of a noun phrase with interpretation NP followed by a verb phrase with interpretation VP". Examples of clauses from the associated dictionary are:

```
is_determiner(every).
is_noun(man).
is-name(mary).
is_trans(loves).
is_intrans(lives).
```

The analysis of the sentence "Every man loves Mary" with these rules produces the following parse tree:



i.e., it follows from the declarative semantics of definite clauses that:

sentence(**theta**,[every,man,loves,mary],[])

is a true term, where **theta** is the term depicted above.

3.3.2. *Extra conditions*

The use of explicit procedure calls in the body of a rule to restrict the constituents accepted, is illustrated by the following rule:

date(D,M) → month(M), [D], {integer(D), 0 < D, D < 32}.

We can read this rule as "A phrase representing the date day D of month M may be written as a phrase representing the month M followed by a symbol D, where D is an integer greater than 0 and less than 32".

3.3.3. *Context dependency*

The arguments of non-terminals in a DCG can be used not only to build structures but also to carry and test contextual information. For instance, we can modify the example of Section 3.3.1 to handle the "number" agreement (singular or plural) required between certain determiners, nouns and verbs. The modified grammar will accept sentences such as "Every man loves some girl" and "All men like girls", but will reject an ungrammatical sentence such as "All men that lives love a woman". To handle the number agreement, certain non-terminals will have an extra argument which can take the values 'singular' or 'plural'; the dictionary predicates will have the "number" argument and also an argument to return the root form of a word. The modified rules are:

```
sentence( s(NP,VP) ) →
  noun_phrase(N,NP), verb_phrase(N,VP).
noun_phrase(N, np(Det,Noun,Rel) ) →
  determiner(N,Det), noun(N,Noun), rel_clause(N,Rel).
```

```
noun_phrase(singular, np(Name) ) → name(Name).
verb_phrase(N, vp(TV,NP) ) →
  trans_verb(N,TV), noun_phrase(N1,NP).
verb_phrase(N, vp(IV) ) → intrans_verb(N,IV).
rel_clause(N, rel(that,VP) ) → [that], verb_phrase(N,VP).
rel_clausc(N, rel(nil) ) → [].

determiner(N, det(W) ) → [W], {is_determiner(W,N)}.
determiner(plural, det(nil) ) → [].
noun(N, n(Root) ) → [W], {is_noun(W,N,Root)}.
name( name(W) ) → [W], {is_name(W)}.
trans_verb(N, tv(Root) ) → [W], {is_trans(W,N,Root)}.
intrans_verb(N, iv(Root) ) → [W], {is_intrans(W,N,Root)}.
```

Examples of clauses from the associated dictionary are:

```
is_determiner(every,singular).
is_determiner(all,plural).
is-noun(man,singular,man).
is_noun(men,plural,man).
is_name(mary).
is_trans(likes,singular,like).
is_trans(like,plural,like).
is_intrans(live,plural,live).
```

3.4. How DCGs are executed by Prolog

So far, we have been discussing DCGs from a *declarative* point of view. To understand a DCG, this is perfectly adequate—for since the DCG is no more than a set of definite clauses, its meaning is independent of any execution mechanism. However, as we have already noted, each proof procedure for definite clauses corresponds to a different parsing strategy for DCGs. We now discuss what this means in the case of the Prolog proof procedure.

From the procedural semantics of Prolog, it follows that, to parse a sentence, the grammar rules are used top-down, one at a time, and that goals in a rule are executed from left to right (i.e. the sentence is parsed from left to right). If there are alternative rules at any point, backtracking will eventually return to them. It is up to the grammar writer to formulate the grammar in such a way that the same work is not repeated unnecessarily on different backtracking alternatives. In practice this is not too difficult for languages intended to be read from left to right, although it often makes the grammar less readable than it would otherwise have been. All the work of the analysis is done by the same uniform mechanism (the Prolog proof procedure) and, in current Prolog implementations, the backtracking is performed very efficiently.

To show how Prolog executes a DCG, and in particular the backtracking and the pattern matching, we will now describe the main steps in parsing the "garden path" sentence:

```
That man that whistles tunes pianos .
1    2   3    4        5     6      7
```

according to the DCG in the previous section, with a dictionary including the following clauses:

```
is_determiner(that,singular).
is_noun(man,singular,man).
is_noun(men,plural,man).
is_noun(pianos, plural,piano).
is_noun(tunes,plural,tune).
is_trans(whistles,singular,whistle).
is_trans(tunes,singular,tune).
is_intrans(whistles,singular,whistle).
```

For greater readability, we will here write Prolog goals coming from DCG non-terminals or terminals in the form:

symbol from **point1** to **point2**

where **symbol** is a non-terminal or list of terminals and **point1**, **point2** are positions in the input string, as labelled above.

The initial goal is:

```
sentence(S)     from 1 to 7
```

This matches the single rule for 'sentence', creating the instantiation:

```
S = s(NP,VP).
```

and the goals:

```
noun_phrase(N,NP)     from 1 to P1,
verb_phrase(N,VP)     from P1 to 7,
```

Prolog next matches the first of those goals against the first rule for 'noun_phrase', producing the instantiation:

```
NP = np(Det,Noun,Rel).
```

and the goals:

```
determiner(N,Det)     from 1 to P2,
noun(N,Noun)          from P2 to P3,
rel_clause(N,Rel)     from P3 to P1,
```

Now the first of these goals expands into two subgoals:

```
[W] from 1 to P2
is_determiner(W,N)
```

both of which succeed immediately, since the word at position 1 in the string is "that" and because the dictionary contains the clause:

```
        is_determiner(that,singular).
```

The solution to the 'determiner' goal is therefore:

```
        determiner(singular, det(that) ) from 1 to 2
```

Prolog now proceeds to the goal for 'noun', which is currently instantiated to:

```
        noun(singular,Noun) from 2 to P3
```

This succeeds in a manner similar to the goal for 'determiner', with solution:

```
        noun(singular, n(man) ) from 2 to 3
```

Note that, had the word at position 2 been "men" instead of "man", the goal would not have succeeded, since no match would be found for the intermediate subgoal:

```
        is_noun(men,singular,Root)
```

Prolog now proceeds to match the 'rel_clause' goal against the first rule for 'rel_clause' yielding the new goals:

```
        [that]                       from 3  to P4,
        verb_phrase(singular,VP1)    from P4 to P1,
```

The first of these goals succeeds trivially, with:

```
        P4 = 4
```

and the second matches the first rule for 'verb_phrase', producing the subgoals:

```
        trans_verb(singular,TV)   from 4  to P5,
        noun_phrase(M,NP1)        from P5 to P1,
```

Both subgoals eventually succeed, with solutions:

```
        trans_verb(singular, tv(whistle) )  from 4 to 5
        noun_phrase(plural,
          np(det(nil),n(tune),rel(nil)) )   from 5 to 6
```

where the second solution is obtained via a match against the first rule for 'noun_phrase'.

We have now obtained a solution to the original 'noun_phrase' goal, corresponding to the phrase "that man that whistles tunes". The next goal to be executed, which comes from the original activation of the rule for 'sentence', is:

```
        verb_phrase(singular,VP)  from 6 to 7.
```

(Now remember that the word at position 6 is "pianos"). Matching this goal against the first rule for 'verb_phrase' leads to the goal:

```
        trans_verb(singular,TV1)  from 6 to P6,
```

which cannot succeed, because "pianos" is not a 'trans_verb'. In the same way, the second rule for 'verb_phrase' fails, because "pianos" is not an 'intrans_verb'.

At this point, Prolog backtracks to the most recent goal for which there is still an alternative rule available, namely:

noun_phrase(M,NP1) from 5 to P1

This goal is now matched against the second rule for 'noun_phrase', leading to the goal:

name(Name) from 5 to P1

which fails because the word at position 5, "tunes", is not a 'name'. Backtracking again, the most recent choice is the use of the first rule for 'verb_phrase' to match the goal:

verb_phrase(singular,VP1) from 4 to P1

So, this goal is now matched against the second rule for 'verb_phrase', producing eventually the solution:

verb_phrase(singular, iv(whistle)) from 4 to 5

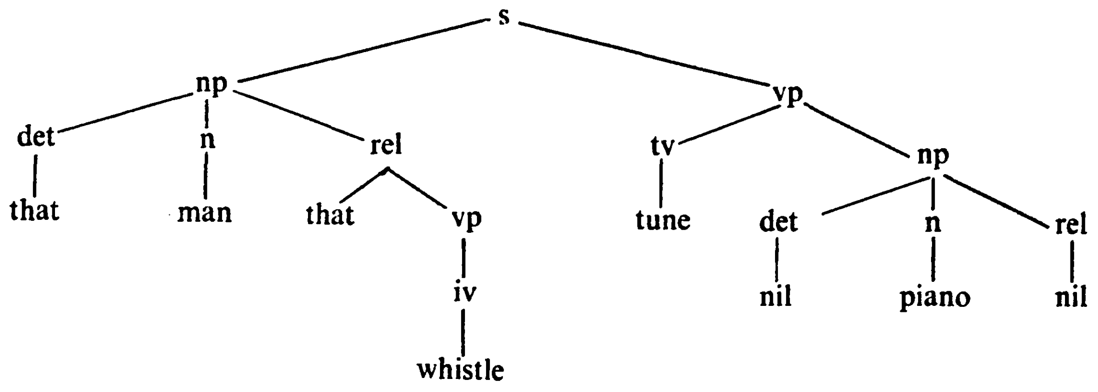
We have now found a second solution to the original 'noun_phrase' goal, corresponding to the phrase "that man that whistles". The only goal still pending is the second goal from the original activation of the rule for 'sentence'. This goal is currently instantiated to:

verb_phrase(singular,VP) from 5 to 7.

Execution of the goal this time succeeds, producing the result:

verb_phrase(singular, vp(tv(tune),np(det(nil),n(piano),rel(nil))) from 5 to 7

thus completing the parsing. Putting together the various instantiations, we obtain, as result S, the structure depicted below:



3.5. The role of the logical variable in DCGs

The feature of logic programs which we have called the "logical variable" makes DCGs a very powerful formalism for implementing practical language analysers.

Structures can be built piecemeal, leaving unspecified parts as variables. The structure can be passed around, and be completed as the parsing proceeds. When the fragments needed are available, the "holes" in the structure represented by variables are filled by unification. Thus it is easy to build terms with structures which do not parallel the parse tree. We illustrate this first with a very simple example, and then in the following section give a much deeper example.

For the first example, we want to recognise sentences comprising a verb 'precedes' or 'follows', and names of months, e.g., "May follows April". The interpretation given to a sentence of this type will be a term of the form 'before(M1,M2)'. For instance, the interpretation of "May follows April" will be 'before(april,may)'. The context-free grammar for this example is given by the rules:

```
sentence → month,verb,month.

verb → [precedes].
verb → [follows].

month → [january].
month → [february].
  etc.
```

To construct the required interpretation, we give the non-terminals extra arguments as follows:

```
sentence(S) → month(M1),verb(M1,M2,S),month(M2).
verb(M1,M2,before(M1,M2)) → [precedes].
verb(M1,M2,before(M2,M1)) → [follows].
month(january) → [january].
month(february) → [february].
  etc.
```

We read the non-terminal 'verb(M1,M2,S)' as "a verb whose interpretation in the context of a subject M1 and object M2, is S".

Notice that, in general, it is not necessary that the first two arguments of 'verb' be known when the proof procedure builds the third argument during the execution of a rule for 'verb'. That is, the relationship between the arguments of a non-terminal is defined independently of any particular order of executing the rules.

Running the example with Prolog, the parsing proceeds from left to right, so in the first rule, the structure S is built before one of its components, M2, is known. This component gets filled in later during the parsing of the rest of the sentence. Note that it is cumbersome and less natural to postpone the building of the structure S until M2 is known, i.e., until the end of the rule for 'sentence', since the form of S depends crucially on the nature of the verb.

3.6. A more sophisticated example

The more sophisticated example of the logical variable in this section has been chosen to illustrate how naturally and concisely one can express complex structure building in a DCG. We leave to the reader the detailed analysis of the example.

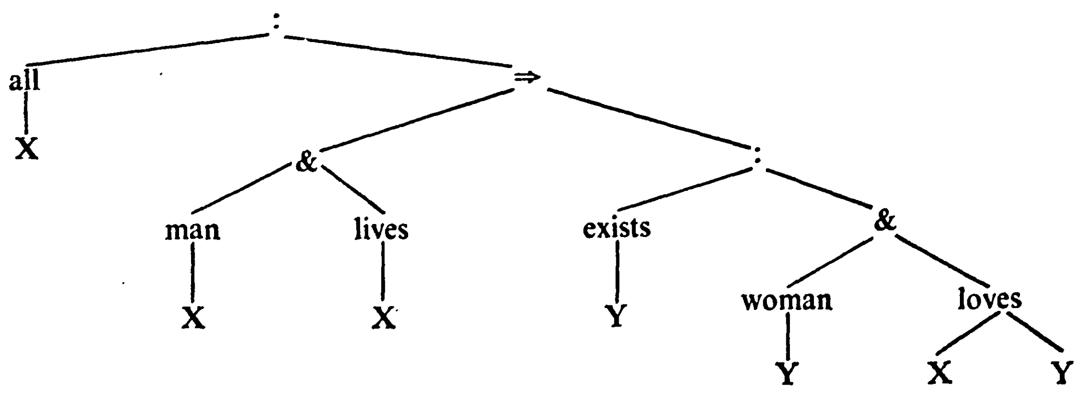
The DCG is again an extension of our original context-free example. It formalises the mapping between English and formulae of classical logic which is usually outlined in introductory logic textbooks. For example, the term constructed as the representation of the sentence:

Every man that lives loves a woman.

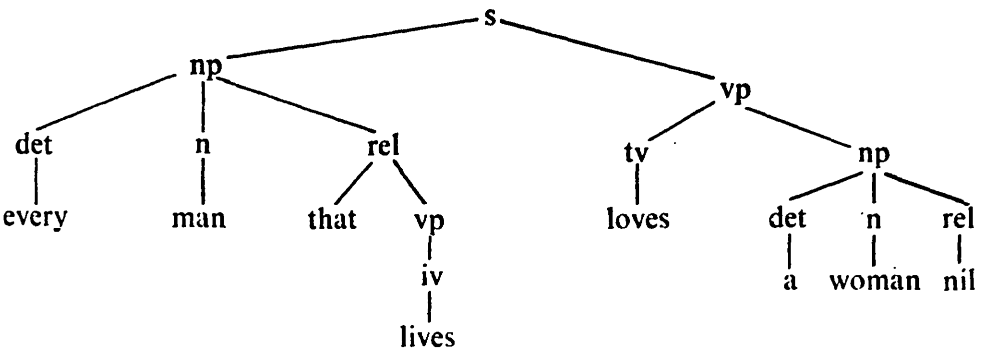
will be:

all(X) : (man(X) & lives(X) ⇒ exists(Y) : (woman(Y) & loves(X,Y)))

(where ' : ', ' & ' and ' ⇒ ' are binary functors written as infix operators). Notice how different the structure of this term is from that of the corresponding parse tree, i.e., compare:



with:



The DCG follows. To avoid details not essential to the purpose of the example, we have not introduced auxiliary predicates for the dictionary.

```
sentence(P) → noun_phrase(X,P1,P), verb_phrase(X,P1).
noun_phrase(X,P1,P) →
  determiner(X,P2,P1,P), noun(X,P3), rel_clause(X,P3,P2).
noun_phrase(X,P,P) → name(X).
```

```
verb_phrase(X,P) → trans_verb(X,Y,P1), noun_phrase(Y,P1,P).
verb_phrase(X,P) → intrans_verb(X,P).

rel_clause(X,P1,P1 & P2) → [that], verb_phrase(X,P2).
rel_clause(X,P,P) → [].

determiner(X,P1,P2, all(X) : (P1 ⇒ P2) ) → [every].
determiner(X,P1,P2, exists(X) : (P1 & P2) ) → [a].

noun(X, man(X) ) → [man].
noun(X, woman(X) ) → [woman].

name(john) → [john].

trans_verb(X,Y, loves(X,Y) ) → [loves].
intrans_verb(X, lives(X) ) → [lives].
```

Each non-terminal has one or more arguments. The last argument gives the interpretation of the corresponding phrase. This interpretation in general depends on other items, as specified by the preceding arguments of the non-terminal. For example, the word "loves" has the interpretation 'loves(X,Y)', which depends on individuals X and Y. A more complex case is the word "every", which has the interpretation:

all(X) : (P1 ⇒ P2)

in the context of two properties P1 and P2 of an individual X. (The property P1 will correspond to the rest of the noun phrase containing the word "every", and the property P2 will come from the rest of the sentence). Observe that the non-terminal for a noun phrase takes the form 'noun_phrase(X,P1,P)', i.e., the interpretation P of the noun phrase will depend on a property P1 of an individual X. This is because in general a noun phrase contains a determiner such as "every". For example, the interpretation of the noun phrase "every man" will be:

all(X) : man(X) ⇒ P1.

The second rule for 'noun_phrase' tells us the interpretation of a noun phrase which consists solely of a name, or "proper noun", e.g., "John". We see that this interpretation, in the context of a property P of some individual X, is simply P itself, provided the individual X is that named by the proper noun (e.g., 'john').

The reader familiar with Montague's (1973) work, will note the similarity of this analysis to Montague's treatment.

Executing this DCG with Prolog brings into play the full power of the logical variable. For the order in which phrases are parsed is such that some parts of the translation of a phrase are not available when that translation is built. For example the interpretation P of a sentence is produced by its subject noun phrase, and, naturally, this interpretation P will also depend on the (as yet unknown) interpretation of the verb phrase which completes the sentence. Such unknown items are left as variables to be filled in later.

4. How to Translate ATNs into DCGs

The purpose of this section is to explain to the ATN-minded reader how an ATN can be translated into a DCG describing the same language and producing the same analysis in essentially the same way. We do not attempt to give a definitive algorithm, as there is always room for ingenuity in producing a good DCG translation. Rather, we indicate the basic ideas underlying the translation process. We take (Bates, 1978) as the definitive reference on ATNs.

4.1. Decomposing the network

A **simple** transition network, i.e., a network without cycles and with a single start and end node, can be directly translated as follows:

(1) the simple network corresponds to a non-terminal;

(2) each distinct path from the start node to the end node translates into a distinct rule for that non-terminal;

(3) the body of each rule is a translation, in order, of the arcs which make up the corresponding path.

Now in general a network has cycles. But it is clear that any network is equivalent to a set of simple networks connected by PUSHes and POPs, what we shall call a **decomposition**. To illustrate all this, we now show how an *un*augmented transition

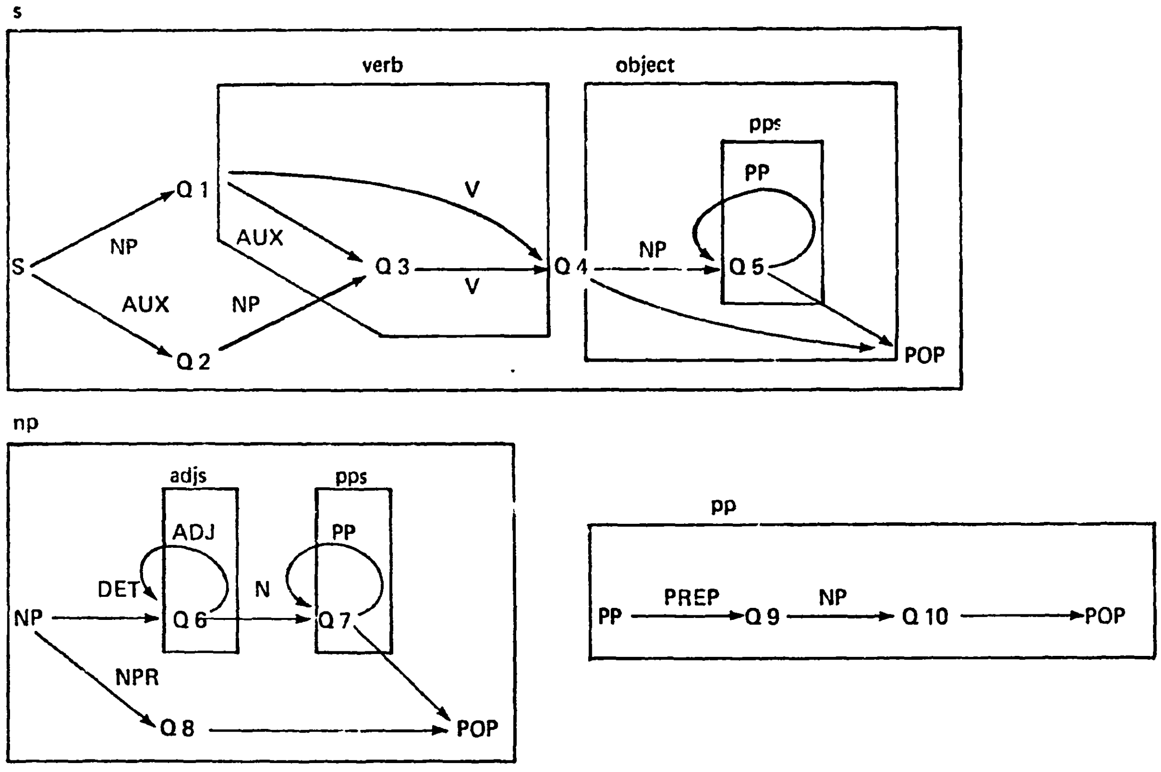


FIG. 1.

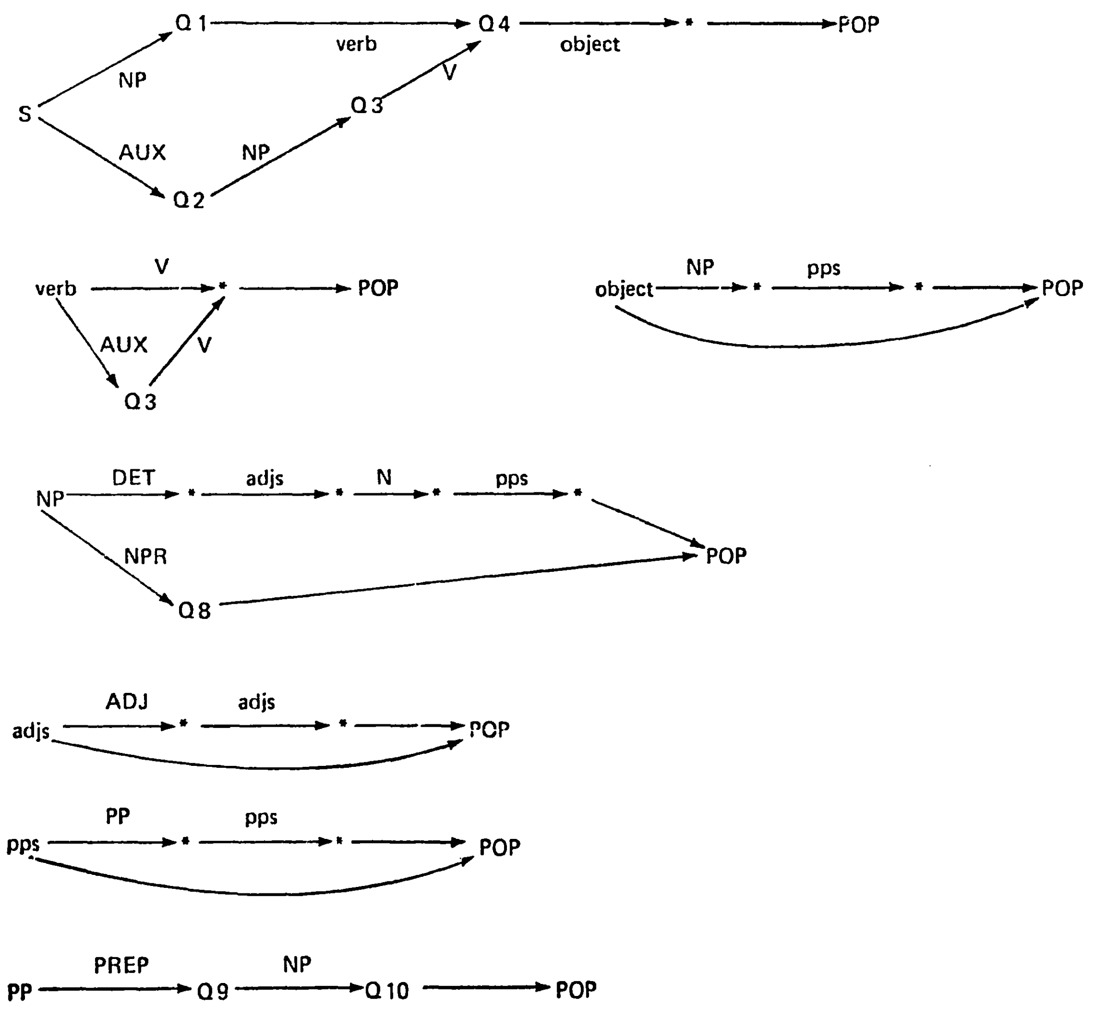


FIG. 2.

network (taken from Woods (1970)) translates into a DCG which is simply a CFG. The network is shown in Fig. 1, where the boxes identify a decomposition into the simple networks of Fig. 2 (where anonymous nodes are labelled by '*'). These in turn translate into the following rules:

```
s → np, verb, object.
s → aux, np, v, object.

verb → v.
verb → aux, v.

object → np, pps.
object → [].

np → det, adjs, n, pps.
np → npr.
```

```
adjs → adj, adjs.
adjs → [].
pps → pp, pps.
pps → [].
pp → prep, np.
```

Note that the V arc from Q3 to Q4 has been translated into two separate occurrences of ' v ', in the second and fourth rules.

Not all decompositions lead to equally concise translations. In general, a more concise translation is obtained if each simple network obeys a **minimality constraint**, that the different paths through the network only meet at the start and end nodes. The decomposition shown above roughly accords with this principle.

4.2. Translating the arcs

We will assume that the Lisp test in an arc is translated into a goal, **test**, and the Lisp actions into a sequence of goals, **actions**. Each arc type is then translated as follows in the body of a rule:

(CAT **category** . . .)	[W], {**test**, **category**(W), **actions**},
(WRD **word** . . .)	[**word**], {**test**, **actions**},
(MEM **list** . . .)	[W], {**test**, in(W,**list**), **actions**},
(TST . . .)	[W], {**test**, **actions**},
(JUMP . . .)	{**test**, **actions**},
(PUSH **subnetwork** . . .)	{**test**}, **subnetwork**, {**actions**},
(POP . . .)	{**test**, **actions**}.

Of course, in cases where the original test or actions are empty, the corresponding goals can be completely omitted from the DCG version. The predicate 'in(X,L)' tests whether X is a member of the list L. Note that these translations are incomplete, as no attention is paid to argument passing in PUSH arcs (SENDRs) and return values in POP arcs. This will be discussed later.

The VIR arc and its associated HOLD action do not have a straightforward translation, and their special-purpose effect must be achieved in DCGs with some of the general purpose mechanisms available. The chief purpose of HOLD and VIR is to handle what transformational grammarians call **constituent extraposition**, that is the phenomenon where a constituent occurs in the "surface" sentence outside the phrase to which it belongs in the "deep structure". A general and powerful means for treating constituent extraposition is provided by an extension of DCGs, called "extraposition grammars" (Pereira, 1979). However, an alternative solution is available in DCGs, through the use of extra arguments to carry the extraposed constituents. A technique directly analogous to this second approach is discussed by Woods (1973) to avoid VIR/HOLD in ATNs, and has been used in situations where parsing cannot proceed from left to right (Bates, 1975; Woods et al., 1976). We shall therefore not go into further detail.

4.3. Treatment of registers, tests and actions

All tests and actions on an arc act upon values kept in **registers**. Now the concept of a register, i.e. a named updatable location where a value can be assigned to or retrieved from, does not exist in logic programming — instead **values** are passed or built in **variables**, where a variable is a local name for a value used in a particular clause, rather than an assignable location. The effects achieved in ATNs using registers are obtained in DCGs through the use of variables. The values of the variables get filled in by the pattern matching which takes place when a goal or non-terminal is executed. Therefore, non-terminals in the translation of an ATN must be augmented with extra arguments, to make use of the pattern matching.

Each non-terminal has at least one argument, which represents the structure returned by the corresponding simple network. This is usually written as the last argument of that non-terminal, and we shall always follow this convention.

For each register of a simple network which is set by a SENDR in some PUSH for that network, there must be a further ("input") argument in the non-terminal representing that network. In the same way, each value sent back to the calling network by a LIFTR in the called network corresponds to an("output") argument in the non-terminal for the called network.

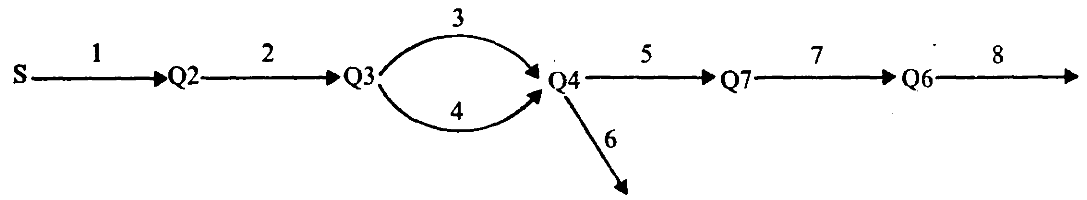
The case of PUSH arcs which were not in the original network, but which arise from its decomposition, is treated as if there were a SENDR for each original network register used in the simple network PUSHed for, and a LIFTR for each original network register which is changed in the invoked network. (And, in the case of a subnetwork introduced by the decomposition, there need be no argument for a return value.)

For a simple network, each register used in a path from the start node to a POP arc is translated into as many variables as the register takes different values in that path. (Usually we name these variables with variants of the name of the corresponding register.)

The value returned at the POP exit of a path in a simple network is just filled into the return value argument of the head of the rule for that path.

BUILDQs are translated in a straightforward and much simpler fashion as terms containing variables corresponding to register values.

The foregoing discussion is better illustrated by a small example. This example is a simplification of the network in Appendix 1 (taken from Woods, 1970). The network covers simple declarative sentences in active and passive form. We depict the network with the arcs labelled by integers, followed by the details of each arc:



```
1: (PUSH NP T
     (SETR SUBJ *))
2: (CAT V T
     (SETR V *))
3: (CAT V (AND (GETF * PPRT)
           (EQ (GETR V) (QUOTE BE)))
     (SETR OBJ (GETR SUBJ))
     (SETR SUBJ (BUILDQ (NP (PRO (SOMEONE)))))
     (SETR V *)
     (SETR AGFLAG T))
4: (PUSH NP T
     (SETR OBJ *))
5: (WRD BY (GETR AGFLAG))
6: (POP (BUILDQ (S + (VP (V+) +))
                SUBJ V OBJ)
     T)
7: (PUSH NP T
     (SETR SUBJ *))
8: (POP (BUILDQ (S + (VP (V+) +))
                SUBJ V OBJ)
     T)
```

The context-free rules corresponding to a decomposition of the network are as follows (where 'a*N*' stands for the translation of arc *N*):

```
s → a1,a2,rest_verb,agent.
rest_verb → a3.
rest_verb → a4.
agent → a5,a7.
agent → [].
```

We now add arguments to the non-terminals in these rules in the way described above, and insert the translation of tests and actions. Tests and actions will be translated whenever possible via by pattern matching in the head of a rule, rather than by actual goals in the body of a rule. The dictionary predicate 'v' resulting from the CAT arcs has two extra arguments, to return the root form of the verb and the tense feature to be used in the GETF test in arc 3. Here are the augmented rules:

```
s(R) →
   np(Subj0),
   [V], { v(V,V0,Tense) },
   rest_verb(Subj0,V0,Subj1,V1,Obj,Agflag),
   agent(Subj1,V1,Obj,Agflag,R).
```

```
rest_verb(Subj0,be, np(pro(someone)),V1,Subj0,t) →
  [V], { v(V,V1,pprt) }.
rest_verb(Subj,V,Subj,V,Obj,f) → np(Obj).
agent(Subj0,V,Obj,t, s(Subj1,vp(v(V),Obj)) ) →
  [by],
  np(Subj1).
agent(Subj,V,Obj,Agflag, s(Subj,vp(v(V),Obj)) ) → [].
```

5. The Advantages of DCGs

Woods (1970) discusses the advantages of the ATN formalism under six headings covering, in our view, just five essentially distinct criteria:

Woods's headings	*essential criteria*
1. Perspicuity	1. Perspicuity
2. Generative Power	2. Power and Generality
3. Efficiency of Representation	3. Conciseness
4. Capturing Regularities	ditto
5. Efficiency of Operation	4. Efficiency
6. Flexibility for Experimentation	5. Flexibility

We argue that, on each of these criteria, and on one extra criterion of our own (6. Suitability for Theoretical Work), DCGs rate at least as highly as ATNs, and that in several respects DCGs represent a significant advance.

5.1. Perspicuity

Practical systems for natural language analysis are necessarily large and complex, and, for the time being at least, writing them is very much an experimental activity. Therefore perspicuity—desirable in any formalism—is particularly important here. The subjective quality of being easy to understand takes on a more objective form—how much real time and effort does it take to modify and extend the system?

Perspicuity is the area where we think DCGs show the most marked improvement over ATNs.

The main reason is that DCGs can be understood in a way which is qualitatively different from the way one understands an ATN. Like an ATN, a DCG can be understood as a *machine* for analysing a particular language. However, unlike an ATN, a DCG can also be understood as a *description* of a language. DCGs share this property with CFGs. As Woods puts it (referring to CFGs), "by looking at a rule, the consequences of that rule for the types of construction that are permitted are immediately apparent".

This can be accounted for informally by noting that it is a straightforward mechanical process to translate each rule of a DCG (or CFG) into a statement of ordinary English, given a glossary of all the symbols (i.e., functors) used in the grammar. The resulting English statements describe what forms are permissible for the phrases which make up the object language in question. We have given examples of this informal translation in our discussion of DCGs. Note that a DCG is unlikely to be readily comprehensible without such a glossary, or some equivalent explanation of the meaning of each symbol and the purpose of its arguments. However, a good choice of names for functors and variables can do much to suggest the intended interpretation.

The immediacy of the relationship between a DCG and the language it describes can also be given a completely formal explication in terms of the declarative semantics of definite clauses. We have previously discussed how a DCG can be identified with a set of definite clauses. The declarative semantics allows us to further identify this set of definite clauses with a (probably infinite) set of "true terms". Each one of these true terms specifies that a certain phrase of the object language occurs between certain points in a certain string. The set of true terms as a whole amounts to an enumeration of all possible occurrences of all possible phrases of the object language. Note that nowhere does this explication involve any notion of *executing* a DCG.

An ATN shares none of the foregoing properties of a DCG. To explain formally how an ATN defines a language seems necessarily to involve the notion of how an ATN is executed. Certainly this is the way ATNs are always explained informally in the literature—see for instance Bates (1978), Section 2, and contrast this with the way we introduce DCGs. Conceptually at least, an ATN is no more than a particular mechanism for parsing a language top-down, left-to-right, and the sequencing imposed by this parsing strategy is implicit in the way registers are operated on. Although it is possible (with ingenuity) to produce other kinds of parsers for ATNs, this requires a re-interpretation of the meaning of arc actions, and necessitates restrictions on register usage.

Bates (1978), in her introduction, claims that "one does not need to know how to program a computer in order to write or use an ATN". However, from the outset, her account of ATNs uses such computing jargon as "pushing the current computation onto a stack". Now although it might be possible to explain the PUSH arc in other terms to non-programmers, it is hard to see how the function of ATN registers could be explained other than by going into some basic computing concepts. Thus, despite Bates's claims to the contrary, it does seem that a knowledge of conventional programming is necessary to properly understand an ATN, whereas the declarative semantics of a DCG is genuinely independent of any notions specific to computing.

However, what we have been discussing so far is not the only aspect in which DCGs are clearer than ATNs. Even without their capability to be understood as language descriptions, and viewed simply as machines, DCGs are in many ways more perspicuous than ATNs.

One of the main reasons for this is that DCGs are more **modular**. The machinery of a DCG is made up of small components (clauses) which communicate only through explicitly passed arguments. There are no global variables—the scope of each variable is limited to a single clause. As a result, the behaviour of each clause in a DCG can be understood independently of any other. In an ATN, on the other hand, the smallest unit which can be isolated in this way is a subnetwork (i.e. a part of the network not connected to the rest except via PUSHes and POPs). No smaller unit can be isolated, since the scope of a register is an entire subnetwork. Now in practice (e.g. LUNAR), subnetworks tend to be very large, and contain too great a mass of detail to be readily assimilated in one piece.

A second factor making DCGs easier to understand is that there is no assignment, i.e. the value of a variable, once fixed, cannot change. No assignment means no side effects, and therefore no possibility of the various sources of confusion which stem from unforseen side effects. Happily, most ATNs actually published only use assignment in a restrained way, and are therefore relatively easy both to understand and to translate into a DCG. In effect, DCGs enforce (and extend) this good practice. It is also worth noting that the ATN writer would lose nothing (in terms of efficiency) by adopting a "single assignment" policy in the style of a DCG. For, given the way ATN register assignment has actually been implemented, it is just as efficient to assign each new value to a fresh register as to update the values of registers already assigned.

Another important feature of DCGs, which helps to make them much more readable than ATNs, is the use of pattern matching in place of explicit tests and BUILDQs. Pattern matching enables what are basically the same underlying operations to be specified in a more concise and "visual" way.

A further point contributing to the clarity of DCGs is that they consist of a single uniform formalism of maximum simplicity. In contrast, ATNs are a more elaborate mixture of two formalisms—transition networks and Lisp. Generally speaking, it does not make for easy comprehension to have a superabundance of ways of saying the same thing, as is the case in ATNs.

5.2. Power and generality

One judges the "power and generality" of a formalism by considering what can, and cannot, be expressed in the formalism—in both a theoretical and a practical sense.

Theoretically, both ATNs and DCGs have the power of a Turing machine, and in that sense are as general as can be. (The adequacy of definite clauses for programming any computable task, without "coding" of the data, is proved by Andreka and Nemeti (1976).)

Of more interest is the question of what tasks can *usefully* be programmed in the two systems. In this context, one of the key features of DCGs is that they provide an essentially more powerful mechanism for building structures than is available in ATNs.

In an ATN, it is impracticable to build structures which do not closely mirror the recursive analysis of the string produced by the PUSH/POP mechanism. This is because a POP arc can only return a single structure, and all of the subcomponents of this structure must be known at the time of the POP. In a DCG, on the other hand, a non-terminal may return more than one structure as its result, and these structures may contain variables which only later get a value. Thus the structure(s) generated in a DCG as the result of the analysis of a phrase may depend on items in the sentence which are **outside** the phrase concerned, and which may not yet have been encountered in the parsing. A good illustration of why this greater generality is useful is provided by the "Sophisticated Example" of Section 3.6.

To simulate such use of the "logical variable" in an ATN, one might be tempted to modify a previously generated structure using **rplaca** and **rplacd**. However, in current ATN implementations at least, this would produce an unwanted side effect on alternative branches of the parser's search space. The other way out would be to use a function such as **substitute**, which involves copying all the structure "above" the point to be modified. However the cost of this copying is likely to be unacceptable in practice.

DCGs are more general than ATNs in that they can be used in a wider variety of ways. This characteristic follows from the fundamental difference between DCGs and ATNs discussed under "Perspicuity", namely that an ATN is a particular machine for parsing a language top-down left-to-right, whereas a DCG is primarily a language description, neutral towards implementation. As a result, a DCG can be executed in a variety of different ways.

For example, Woods (1970) has discussed the question of whether ATNs can be used for *generation* as well as for recognition, i.e. given a "deep-structure", to generate the corresponding surface string(s), instead of the usual inverse process. Now to use an ATN for generation would involve substantial changes in interpretation of the operations labelling the arcs, and the feasibility of this re-interpretation is questionable, particularly if arbitrary Lisp code is involved in arc actions.

In contrast, it is perfectly feasible to program a generation process as a DCG without any change whatsoever to the DCG *formalism*. Moreover, the same proof procedure (e.g., in particular, Prolog) can be adequate for implementing both generation and recognition processes. It is even possible to use the same DCG for both kinds of task, although this will only be practicable in certain cases, and then only with careful design.

A generation problem is specified by presenting an initial goal of the form:

?- sentence(**structure**,S,[]).

where **structure** is a term representing a deep-structure. The result will be to instantiate S to a list representing the surface form of **structure**. Compare this with the usual recognition problem, which is specified in the form:

```
?- sentence(T,string,[]).
```

where **string** is a list representing the initial surface string, and T becomes instantiated to a corresponding deep structure.

If a DCG is to be used for generation, the only clause for the 'connects' predicate should be:

```
connects([W | S],W,S).
```

(and, as described earlier, all calls to 'connects' may be preprocessed away prior to execution). For an example of a generation task programmed as a DCG and executed by Prolog, see Chapter 4 of Colmerauer (1975).

We have been discussing an example of DCG generality where DCGs are used to formalise two quite different kinds of task—generation as well as recognition—using the same proof procedure, Prolog. Another case of DCG generality is that a variety of different processes for solving a given task (such as recognition) can be obtained from the same DCG, by applying different proof procedures to it. Thus the top-down left-to-right parsing entailed by using Prolog is by no means the only way to execute a DCG. Other proof procedures would give different parsing mechanisms (e.g., breadth-first, bottom-up). In particular, Earley's (1970) parsing algorithm can be generalised to give a complete proof procedure for definite clauses (Warren, 1975). Note, however, that a DCG which is efficient for execution by one proof procedure will not necessarily be efficient for another.

A final point concerning the generality of DCGs is that they are not in principle restricted to input consisting of a simple string of atomic symbols. The symbols can be generalised to arbitrary tree structures (possibly with variables) and, more interestingly, instead of a simple list of symbols one can have a "chart" (Kaplan, 1973) catering for alternatives in the input. For example, if part of the input string is:

```
... definite clause grammar ...
   1        2      3       4
```

and the lexical items 'definite', 'clause', 'grammar', 'definite_clause', and 'definite_clause_grammar' are in the dictionary, the following clauses for the 'connects' predicate would represent the possible lexical interpretations:

```
connects(1, definite, 2).
connects(1, definite_clause, 3).
connects(1, definite_clause_grammar, 4).
connects(2, clause, 3).
connects(3, grammar, 4).
```

5.3. Conciseness

In his discussion of "Efficiency of Representation" and "Capturing Regularities", Woods is really concerned with the conciseness of a formalism. This criterion is aptly summed up in his "economy principle"—that the best grammar is that which can characterise a language in the least number of symbols.

If, according to this principle, one compares the *textual* forms of equivalent ATNs and DCGs (counting each identifier as one symbol, and discounting punctuation symbols such as brackets and commas), one generally finds that DCGs are significantly smaller. Typically, the DCG is only around half the size of the ATN.

DCGs are more concise than ATNs for the same reasons that logic programs are in general more concise than programs in conventional languages. The main factor is the use of pattern matching instead of explicit operations for setting and testing registers and building structures.

As has been seen, DCGs are a natural generalisation of context-free grammars. Woods (1970) states that "a major advantage of the transition network model over the usual context-free grammar model is the ability to merge the common parts of many context-free rules, thus allowing greater efficiency of representation". Here Woods is claiming an advantage for the ATN formalism over CFGs, and his subsequent argument to support the claim clearly also applies when comparing ATNs with DCGs. However, we do not think that Woods's argument is correct.

The ability to merge the common parts of many context-free rules is not unique to transition networks, but can be achieved without even going beyond the formalism of context-free rules. For example, the sample grammar which Woods uses to illustrate his argument:

```
s → np,vp.
s → q,np,vp.
s → neg,np,vp.
s → q,neg,np,vp.
```

is better re-expressed as:

```
s → q,s1.
s → s1.

s1 → neg,s2.
s1 → s2.

s2 → np,vp.
```

and this is not so very different from (and in fact it is far more concise than) the *textual* form of the transition network:

$$\begin{tikzcd}
S \arrow[r, bend left, "Q"] \arrow[r] & S1 \arrow[r, bend left, "NEG"] \arrow[r] & S2 \arrow[r, "NP"] & S3 \arrow[r, "VP"] & S4 \arrow[r] & {}
\end{tikzcd}$$

i.e.,

```
(S
  (PUSH Q T (TO S1))
  (JUMP S1 T))
(S1
  (PUSH NEG T (TO S2))
  (JUMP S2 T))
(S2
  (PUSH NP T (TO S3)))
(S3
  (PUSH VP T (TO S4)))
(S4
  (POP NIL T))
```

If one allows, as we do, alternatives to be given in a rule, then the grammar reduces to a single rule, very close to the original regular expression:

```
s → (q;[]),(neg;[]),np,vp.
```

Woods makes much of the ability in ATNs to merge similar parts of a network by recording and testing extra information in registers. There is a direct counterpart of this in DCGs, where similar rules can be coalesced by attaching extra arguments to non-terminals. Whereas Woods seems to favour such merging for ATNs, we think it encourages an intricate and low-level style of language description. Moreover it does not necessarily produce a more concise result. In the case of ATNs, for example, information which was previously explicit in the network is now encoded in Lisp as more complex tests and actions.

The modularity of DCGs encourages the grammar writer to keep separate what are conceptually distinct parts of the grammar, and not to indulge in merging of parts which are superficially similar.

5.4. Efficiency

The operational efficiency of a formalism for language analysis is a matter of crucial importance for applications, such as LUNAR, intended to be genuinely useful. Hence we discuss efficiency at length in this section.

First, let us recall that executing a DCG with Prolog gives a parsing mechanism which can be described as "top-down, left-to-right, depth-first (i.e. one alternative at a time)". Now this is precisely the parsing mechanism used in the majority of ATN applications. Moreover, it is the required mode of operation for recent ATN implementations (Burton and Woods, 1976; Finin and Hadden, 1977) which *compile* the ATN into low-level code (using Lisp as an intermediary). Accordingly, we shall restrict our discussion of efficiency to Prolog implementations of DCGs and to such comparable ATN implementations.

A key property of DCGs, as regards their efficiency, is that a DCG is expressed directly in a general purpose programming language, Prolog. Apart from optional "syntactic sugar", a DCG *is* a Prolog program. DCGs do not need a special interpreter or compiler. To discuss DCG efficiency, therefore, is to discuss Prolog efficiency.

Now Warren, Pereira and Pereira (1977) have described how Prolog can be compiled directly into efficient machine code. They put forward simple reasons why one might expect the speed of the code produced to be comparable with that for more conventional high-level languages, such as Lisp, and argue in particular that pattern matching encourages a better implementation of operations on structured data than the conventional use of selector and constructor functions (such as **car**, **cdr** and **cons**). A practical implementation exists for the DECsystem-10 machine and actual timing data (Warren, 1977a) supports these conclusions. On the basis of this evidence, one can therefore say that a DCG is expressed directly in a general purpose programming language which has an efficiency comparable with Lisp.

An ATN, on the other hand, needs a special interpreter or compiler. Since the ATN formalism relies so heavily on Lisp constructions for expressing tests etc., it is difficult to imagine an ATN compiler which did not generate Lisp code as an intermediary. Therefore it is probably fair to say that ATN efficiency is limited by, and necessarily somewhat inferior to, the efficiency of Lisp for writing grammars.

A disadvantage of ATNs, or at least of the implementations described, is that the system does not have immediate access to the value of a register—the GETR function has to search down an association list of register-value pairs. In Prolog implementations, on the other hand, each variable's value is stored at a known location. This is achieved without any overheads of copying information into or out of variable value cells at procedure call and exit (as happens, for example, in "shallow binding" implementations of Lisp).

The only significant overhead of this kind in Prolog is attributable to its non-determinacy. In certain circumstances when instantiating a variable, the variable's address is remembered on a push-down list, so that the variable can be reset to "uninstantiated" on backtracking. In the DECsystem-10 implementation, these operations are implemented very efficiently at the machine-code level, and only account for a small proportion of the time spent in a typical Prolog computation. Note that the non-determinacy of ATNs has to be achieved using what facilities are provided by the higher level language Lisp, which, unlike Prolog, does not itself incorporate any machinery for non-determinate computation.

A discussion in Warren et al. (1977) attributes much of Prolog's surprisingly competitive speed, compared with Lisp, to the use of "structure sharing" (Boyer and Moore, 1972; Warren, 1977a) to build new data structures. The argument applies a fortiori if we compare with the structure building operations of ATNs. Essentially structure sharing enables arbitrarily large data structures to be constructed with virtually no time cost. Constructing the new object merely involves bringing together two pointers. One is a pointer to a "skeleton" structure, created at compile-time, which corresponds to a term of the source program. The other pointer is to an already existing vector of value cells, called a "frame", which contains the values of variables occurring in the skeleton.

Now compare this one trivial operation with what is involved in the BUILDQ of ATNs. There, space for the new structure has to be allocated from Lisp's "heap" storage (and ultimately garbage-collected), and all the information corresponding to the skeleton structure and the values of its variables has to be copied over into the newly allocated space. It is interesting to note a comment by Woods (1973, p. 133), which, while acknowledging the inefficiencies of register access in ATNs, appears to foresee the advantages of structure sharing: "if the structure returned by the POP arc were merely the list of register contents themselves, then the process of searching for registers by names could be almost totally eliminated".

A feature of the DECsystem-10 Prolog implementation which can make a very significant contribution to the speed of operation of a DCG is the automatic indexing provided for the clauses of each predicate (Warren et al., 1977; Warren, 1977a). When trying to execute a goal, the relevant clauses from the corresponding predicate are accessed through a hash table keyed on the principal functor of the first argument of the goal. In suitable circumstances, the indexing provides for the immediate selection of an appropriate grammar rule from amongst a set of alternatives. This is instead of having to try all the alternatives one by one. A comparable facility does not appear to be available in ATN implementations. Incidentally, the same indexing makes it practicable to implement "dictionary" predicates as sets of unit clauses (cf. many of our examples), since the indexing ensures that the time to look up an individual word in the dictionary is (generally speaking) independent of the number of words in the dictionary.

To recapitulate, we have described a number of aspects in which (compiled) Prolog implementations of DCGs might be expected to be more efficient than current (compiled) ATN implementations:

(1) Compilation of a DCG is only a one stage process, and does not involve an intermediate high level language (Lisp).

(2) Access to variable values is immediate and the overheads attributable to non-determinacy are minimal.

(3) Structure building is done "on the fly", by "structure sharing", at almost no extra cost.

(4) Automatic indexing provides for the immediate selection of appropriate alternatives in the grammar.

Above all, a DCG is merely a particular kind of program in an efficient and general purpose programming language, whereas an ATN is a special purpose formalism.

The decisive test of efficiency is, of course, to compare actual performance data. For the comparison to be meaningful, one must compare equivalent grammars, expressed in equivalent ways, building equivalent structures. The difficulty here is that none of the ATNs for which times have been quoted in the literature has actually been listed in full detail, and these ATNs are in any case unnecessarily big for the purpose simply of making an exact comparison with an equivalent DCG.

Our experience with DCGs, which probably applies equally to ATNs, is that the speed of a grammar depends predominantly on whether the grammar writer chooses to aim at efficiency, or at maximal conciseness and simplicity. The physical size of the grammar (number of rules, or arcs, say) is not, alone, a reliable indicator of the likely parsing times.

In Appendix 2, we give some timing figures for a DCG translation of an early specimen ATN, given by Woods (1970), and also data for a DCG of some complexity covering a sizable subset of English. For what it is worth, bearing in mind our previous remarks: this latter DCG running on a DEC KI10 takes approximately 8 msec. per word to parse an English sentence, while figures quoted for a compiled version of LUNAR on a KA10 (generally reckoned to be only half as fast as a KI10) are of the order of 34 msec. per word on superficially similar sentences.

5.5. Flexibility

In providing a framework for language analysis, a formalism should not be so restrictive that it prevents experimentation with new and diverging ideas. Necessary flexibility of this kind is available in ATNs by virtue of the open-ended use which can be made of Lisp—to build diverse structures, to express special conditions on arcs, etc.

In an exactly analogous way, DCGs have access to the full power of the definite clause subset of logic as a general purpose programming language. As in ATNs, there is wide scope for building different kinds of structures to represent the result of the analysis. Also, by using explicit calls to separately defined procedures, one may easily incorporate into the grammar arbitrarily complex tests, and these tests may depend on auxiliary information passed as extra arguments to non-terminals. As mentioned previously, within the basic DCG formalism one can simulate the effects of special purpose ATN facilities such as the "hold list".

The DCG formalism is more flexible than ATNs in that, as previously discussed, it is in no way tied to a particular parsing or execution mechanism (although the style in which the grammar is written will usually be optimised towards some particular parsing mechanism). Thus writing a grammar as a DCG makes it much easier to experiment with radically different parsing strategies, such as were tried out in the BBN Speech Understanding System (Woods et al., 1976).

5.6. Suitability for theoretical work

In this section we argue that, unlike ATNs, DCGs can also be a useful formalism for theoretical studies of language, and that, as a consequence, they potentially provide a bridge between the work of theoretical linguists and philosophers, such as Chomsky and Montague, and the work of those, such as Woods, concerned with engineering practical natural language systems. To fully justify these claims would call for another paper, so here we merely outline the key points of the argument.

The theorists have (properly) concentrated on describing *what* natural language is, in a clear and elegant way. In this context, details of *how* natural language is actually recognised or generated need not be relevant, and indeed should probably not be allowed to obscure the language definition. This concern with the "what" rather than the "how" of language analysis is reflected in the kinds of formalism developed by the theorists. At the time ATNs were developed, it was not clear how such formalisms could be used as a basis for practical systems to actually carry out language analysis, and the need to achieve workable systems necessitated the more machine-oriented formalism of ATNs.

In consequence, the ATN formalism is fundamentally different from any used in theoretical work. As has already been discussed under "Perspicuity", an ATN is a description of a *process* for recognising a language, rather than a description of the language itself. A symptom of the ATN's process orientation is the use of the assignment operation—a concept virtually unknown outside computing, and one which does not naturally enter into formal descriptions in mathematics or other fields. For these reasons, the ATN formalism is not really suitable for theoretical purposes (except in so far as it is more precise than other semi-formal methods for describing language).

Because of this major difference between the ATN formalism and those normally used by theorists, it has been difficult for the ATN writer to draw directly on theoretical work, and difficult for the outsider to relate what is going on inside an ATN with the kind of language analysis proposed by theorists.

In the years since ATNs were developed, the discovery that logic can be used as a programming language has given us a formalism, DCGs, which can serve both as a description of a language, and, by virtue of the procedural interpretation of logic, as a description of a process for analysing that language. For practical purposes, DCGs, while being less overtly machine-oriented than ATNs, can nevertheless be implemented as efficiently and are a powerful tool for implementing working natural language systems. Furthermore, DCGs seem eminently suitable as a formalism for theoretical work—they are a natural and sufficiently powerful generalisation of CFGs, and they have a clear declarative semantics independent of any execution mechanism. Unlike ATNs, DCGs do not incorporate the concept of assignment.

Indeed it could be argued that DCGs are more suitable as a formalism for theoretical purposes than those in current use. It appears that current theoretical formalisms are either less powerful than DCGs, or else, through being biased towards the process of language generation, incorporate unnecessary notions of execution order.

6. Conclusion

On both practical and philosophical grounds, we believe DCGs represent a significant advance over ATNs.

Considered as practical tools for implementing language analysers, DCGs are in a real sense more powerful than ATNs, since, in a DCG, the structure returned from the analysis of a phrase may depend on items which have not yet been encountered in the course of parsing the sentence. Such use of the power of the "logical variable" is well illustrated by the "Sophisticated Example" of Section 3.6, which we do not believe can be directly mimicked in an ATN.

Also on the practical side, the greater clarity and modularity of DCGs is a vital aid in the actual development of systems of the size and complexity necessary for real natural language analysis. Because the DCG consists of small independent rules with a declarative reading, it is much easier to extend the system with new linguistic constructions, or to modify the kind of structures which are built. Our own experience of just these kinds of problems came from adapting a natural language system written by Veronica Dahl (cf. Appendix 2). The modifications involved substituting English for Spanish as the discourse language, and completely changing the domain of discourse. We found it quite straightforward to make these substantial alterations, and doubt whether this would have been so, had the system not been implemented as a DCG.

Finally, on the philosophical side, DCGs are significant because they potentially provide a common formalism for theoretical work and for writing efficient natural language systems. Note that we are NOT claiming that a DCG formulated as a clear theoretical description of a language is likely to be suitable for execution as a practical language analyser. We have argued only that a common formalism is feasible for both. Normally a substantial transformation would be necessary to turn a DCG conceived as a theoretical description of a language into a practical implementation. It is an interesting problem for future research to see whether such transformations can be performed systematically, possibly by generalising known results on parsing with context-free grammars.

Appendix 1. A Full Example

The ATN from Woods (1970), as amended in Burton and Woods (1976), is listed here, together with a DCG translation of a slightly modified network. The modifications were mainly to prevent the acceptance of ungrammatical sequences of verbs at node Q3/ of the original ATN.

After the DCG proper, there is listed an extract from the dictionary of the DCG; just one clause for each predicate is illustrated. Because of the indexing provided

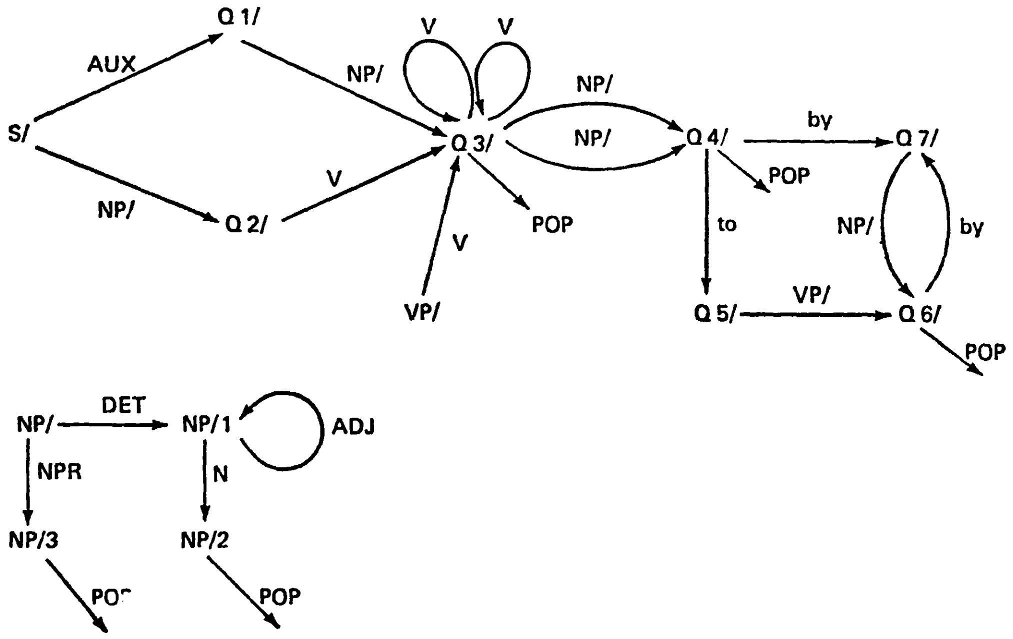


FIG. 3. Diagram of the ATN.

by the DECsystem-10 Prolog implementation, the speed of operation of the DCG is not affected by the size of the dictionary (i.e., by the number of clauses provided for each predicate).

Note the mutually exclusive nature of the three rules for the non-terminal 'complement' in the DCG proper. The Prolog indexing also serves to automatically select the correct alternative from among these three rules.

Listing of the ATN

```
(S/
  (CAT AUX T
      (SETR V *)
      (SETR TNS (LIST (GETF * TENSE)))
      (SETRQ TYPE Q)
      (TO Q1/))
  (PUSH NP/ T
      (SETR SUBJ *)
      (SETRQ TYPE DCL)
      (TO Q2/)))
(Q1/
  (PUSH NP/ T
      (SETR SUBJ *)
      (TO Q3/)))
```

```
(Q2/
  (CAT V T
      (SETR V *)
      (SETR TNS (LIST (GETF * TENSE)))
      (TO Q3/)))
(Q3/
  (CAT V (AND (GETF * PPRT)
              (EQ  (GETR V)
                   (QUOTE BE)))
      (HOLD (GETR SUBJ))
      (SETR SUBJ (BUILDQ (NP (PRO SOMEONE))))
      (SETR AGFLAG T)
      (SETR V *)
      (TO Q3/))
  (CAT V (AND (GETF * PPRT)
              (EQ  (GETR V)
                   (QUOTE HAVE)))
      (SETR TNS (APPEND   (GETR TNS)
                          (QUOTE (PERFECT))))
      (SETR V *)
      (TO Q3/))
  (PUSH NP/ (TRANS (GETR V))
      (SETR OBJ *)
      (TO Q4/))
  (VIR NP (TRANS (GETR V))
      (SETR OBJ *)
      (TO Q4/))
  (POP (BUILDQ (S + + (TNS +) (VP (V +)))
               TYPE SUBJ TNS V)
      (INTRANS (GETR V))))
(Q4/
  (WRD BY (GETR AGFLAG)
      (SETR AGFLAG NIL)
      (TO Q7/))
  (WRD TO (S-TRANS (GETR V))
      (TO Q5/))
  (POP (BUILDQ (S + + (TNS +) (VP (V +) +))
               TYPE SUBJ TNS V OBJ)
      T))
(Q5/
  (PUSH VP/ T
      (SENDR SUBJ (GETR OBJ))
```

```
          (SENDR TNS (GETR TNS))
          (SENDRQ TYPE DCL)
          (SETR OBJ *)
          (TO Q6/)))
(Q6/
   (WRD BY (GETR AGFLAG)
          (SETR AGFLAG NIL)
          (TO Q7/))
   (POP (BUILDQ (S + + (TNS +) (VP (V +) +))
                        TYPE SUBJ TNS V OBJ)
          T))
(Q7/
   (PUSH NP/ T
          (SETR SUBJ *)
          (TO Q6/)))
(VP/
   (CAT V (GETF * UNTENSED)
          (SETR V *)
          (TO Q3/)))
(NP/
   (CAT DET T
          (SETR DET *)
          (TO NP/1))
   (CAT NPR T
          (SETR NPR *)
          (TO NP/3)))
(NP/1
   (CAT ADJ T
          (ADDL ADJS *)
          (TO NP/1))
   (CAT N T
          (SETR N *)
          (TO NP/2)))
(NP/2
   (POP (BUILDQ (NP (DET +) (ADJ +) (N +))
                        DET ADJS N)
          T))
(NP/3
   (POP (BUILDQ (NP (NPR +))
                        NPR)
          T))
```

The DCG Proper

```
sentence(S) →
  [W], {aux_verb(W,Verb,Tense)},
  noun_phrase(G_Subj),
  rest_sentence(q,G_Subj,Verb,Tense,S).
sentence(S) →
  noun_phrase(G_Subj),
  [W], {verb(W,Verb,Tense)},
  rest_sentence(dcl,G-Subj,Verb,Tense,S).
rest_sentence(Type,G_Subj,Verb,Tense,
              s(Type,L_Subj,tns(Tense1),VP) ) →
  rest_verb(Verb,Tense,Verb1,Tense1),
  {verbtype(Verb1,VType)},
  complement(VType,Verb1,G_Subj,L_Subj,VP).
rest_verb(have,Tense,Verb,(Tense,perfect)) →
  [W], {past_participle(W,Verb)}.
rest_verb(Verb,Tense,Verb,Tense) → [].
complement(copula,be,Obj,Subj, vp(v(Verb),Obj1) ) →
  [W], {past_participle(W,Verb), transitive(Verb)},
  rest_object(Obj,Verb,Obj1),
  agent(Subj).
complement(transitive,Verb,Subj,Subj, vp(v(Verb),Obj1) ) →
  noun_phrase(Obj),
  rest_object(Obj,Verb,Obj1).
complement(intransitive,Verb,Subj,Subj, vp(v(Verb)) ) → [].
rest_object(Obj,Verb,S) →
  {s_transitive(Verb)},
  [to,Verb1], {infinitive(Verb1)},
  rest_sentence(dcl,Obj,Verb1,present,S).
rest_object(Obj,Verb,Obj) → [].
agent(Subj) → [by], noun_phrase(Subj).
agent(np(pro(someone))) → [].
noun_phrase(np(Det,adj(Adjs),n(Noun))) →
  [Det], {determiner(Det)},
  adjectives(Adjs),
  [Noun], {noun(Noun)}.
noun_phrase(np(npr(PN))) → [PN], {proper_noun(PN)}.
adjectives([Adj | Adjs]) →
  [Adj], {adjective(Adj)},
  adjectives(Adjs).
adjectives([]) → [].
```

Extract from the Dictionary of the DCG

```
aux_verb(W,V,T) :- verb(W,V,T),auxiliary(V).
auxiliary(be).
verb(is,be,present).
proper_noun(john).
determiner(the).
adjective(nice).
noun(book).
verbtype(be,copula).
verbtype(V,transitive) :- transitive(V).
verbtype(V,intransitive) :- intransitive(V).
transitive(shoot).
intransitive(sleep).
s_transitive(believe).
infinitive(be).
past_participle(been,be).
```

Appendix 2. Performance Data

The DCG timing data which follows is for compiled code produced by the DECsystem-10 Prolog implementation, running on a KI-10 processor. We list the CPU times in milliseconds, averaged over 100 tests for examples in Part 1, and over 10 tests for examples in Part 2.

Part 1

The DCG is that listed in Appendix 1. For each example, there is listed the time to obtain the first parse, followed by the time to exhaust all parses and the total number of parses. For reference, the parse tree(s) obtained are also listed in selected cases.

Observe that in those cases where there is a unique parse, the overhead of going on to seek alternative parses is very low. This is a result of the efficient implementation of backtracking, and of the generally highly determinate nature of this particular grammar for top-down, left-to-right parsing.

(1) fred shot john—3 words
3.0 msec. 3.2 msec. 1 parse
s(dcl,np(npr(fred)),tns(past),vp(v(shoot),np(npr(john))))

(2) mary was liked by john—5 words
3.9 msec. 4.1 msec. 1 parse

(3) fred told mary to shoot john—6 words
5.1 msec. 5.7 msec. 1 parse

(4) john was believed to have been shot by fred—9 words
5.7 msec. 8.3 msec. 2 parses
s(dcl,np(pro(someone)),tns(past),vp(v(believe,s(dcl,np(npr(fred)),
tns((present,perfect)),vp(v(shoot),np(npr(john))))))
s(dcl,np(npr(fred)),tns(past),vp(v(believe),s(dcl,np(pro(someone)),
tns((present,perfect)),vp(v(shoot),np(npr(john))))))

(5) was dave believed to have told mary to tell fred to buy the book by john—16 words
9.6 msec. 12.1 msec. 1 parse

Part 2

Here we attempt to offer some kind of a comparison with the only available published ATN timing data. We list five examples taken from Burton (1976) of sentences with their CPU times for parsing by the compiled LUNAR system running on a DEC KA-10 processor. Each of these examples is followed, for comparison, by a superficially similar sentence accepted by a DCG based on the parsing component of a natural language question-answering system written by Veronica Dahl (1977). This system treats a sizable subset of natural language, approaching in scale that of LUNAR.

All times are to obtain the first parse only. One structure produced as the result of the DCG analysis is listed for reference. Note particularly that the DEC KI-10 processor used for the DCG times is generally reckoned to be nearly twice as fast as a KA-10.

(1) Give me all analyses of S10046.
245 msec.

What are the files of David?
78 msec.

(2) How many breccias contain olivine?
175 msec.

How many files date from Monday?
39 msec.
how_many(X:[] & file,
and(and(pr(file(X)),and(true,true)),
pr(dateof(X,[monday]))))

(4) List modal plag analysis for lunar samples that contain olivine.
265 msec.

Which people are owners of small files that date from Monday?
98 msec.

(5) What is the average composition of olivine?
275 msec.

What is the size of PLC?
40 msec.

(7) How many breccias do not contain Europium?
240 msec.

How many files do not date from Friday?
38 msec.

ACKNOWLEDGEMENTS

We would like to thank all the people who read drafts of this paper, and particularly Chris Mellish, William Woods, Alan Bundy and the referees for their detailed comments and suggestions. The paper was written while Pereira was supported by a British Council Fellowship and Warren by a British Science Research Council Grant (BRG/9445.5).

REFERENCES

Andreka, H. and Nemeti, I. (1976), The generalised completeness of Horn predicate-logic as a programming language, Dept. of AI Research Report 21, Edinburgh (March 1976).

Bates, M. (1975), Syntactic analysis in a speech understanding system, BBN Report 3116 (August 1975).

Bates, M. (1978), The theory and practice of augmented transition networks, in: L. Bolc (Ed.), *Natural Language Communication with Computers* (Springer, Berlin, May 1978).

Bergman, M. and Kanoui, H. (1975), Sycophante: système de calcul formel et d'intégration symbolique sur l'ordinateur, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy (October 1975).

Boyer, R. S. and Moore, J. S. (1972), The sharing of structure in theorem proving programs, in: Meltzer and Michie (Eds.), *Machine Intelligence* 7, (Edinburgh, 1972).

Burton, R. R. (1976), Semantic grammar: an engineering technique for construction of natural language understanding systems, BBN Report 3453 (December 1976).

Burton, R. R. and Woods, W. A. (1976), A compiling system for augmented transition networks, Preprints of COLING-76, Ottawa (June 1976).

Colmerauer, A. (1970), Les systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur, Internal publication no. 43, Départment d'Informatique, Université de Montreal, Canada (September 1970).

Colmerauer, A. (1975), Les grammaires de metamorphose, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy (November 1975). Appears as "Metamorphosis Grammars" in: L. Bolc (Ed.), *Natural Language Communication with Computers*, (Springer, Berlin, May 1978).

Colmerauer, A. (1977), An interesting natural language subset, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy (October 1977).

Dahl, V. (1977), Un systeme déductif d'interrogation de banques de données en Espagnol, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy (November 1977).

Darvas, F., Futo, I. and Szeredi, P. (1977), Logic based program system for predicting drug interactions, *Int. J. Biomedical Comput.* (1977).

Earley, J. (1970), An efficient context-free parsing algorithm, *C. ACM* 13 (February 1970).

van Emden, M. H. (1975), Programming with resolution logic, Report CS-75-30, Dept. of Computer Science, University of Waterloo, Canada (November 1975).

Finin, T. and Hadden, G. (1977), Augmenting ATNs, *Proc. 5th IJCAI, MIT*, Cambridge, MA (August 1977).

Kaplan, R. M. (1973), A general syntactic processor, in: Randall Rustin (Ed.), *Natural Language Processing* (1973).

Koster, C. H. A. (1971), Affix grammars, in: J. E. L. Peck (Ed.), *Algol-68 Implementation* (North Holland, Amsterdam, 1971).

Kowalski, R. A. (1974a), Predicate logic as programming language, *Proc. IFIP* 74, Stockholm (1974).

Kowalski, R. A. (1974b), Logic for problem solving, DCL Memo 75, Dept. of AI, Edinburgh (March 1974). (To be published by North-Holland, Amsterdam as part of a book of the same title.)

Markusz, Z. (1977), Designing variants of flats, *Proc. IFIP Conf.* (1977).

Montague, R. (1973), The proper treatment of quantification in ordinary English, in: R. M. Thomason (Ed.), *Formal Philosophy* (Yale University Press: 1974).

Pereira, F. (1979), Extraposition grammars, Working Paper No. 59, Dept. of AI, University of Edinburgh (June 1979).

Pereira, L., Pereira, F. and Warren, D. (1978), User's guide to DECsystem-10 Prolog, Div. de Informatica, LNEC, Lisbon and Dept. of AI, University of Edinburgh (September 1978).

Robinson, J. A. (1965), A machine-oriented logic based on the resolution principle, *J. ACM* 12 (1965).

Roussel, P. (1975), Prolog: manuel de référence et d'utilisation, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy (September 1975).

Warren, D. H. D. (1975), Implementation of an efficient predicate logic interpreter based on Earley deduction, Research proposal to the Science Research Council, Dept. of AI, University of Edinburgh (1975).

Warren, D. H. D. (1977a), Implementing Prolog—compiling predicate logic programs, Dept. of AI Research Reports 39 and 40, University of Edinburgh (May 1977).

Warren, D. H. D. (1977b), Logic programming and compiler writing, Dept. of AI Research Report 44, University of Edinburgh (September 1977). To appear in *Software Practice and Experience*.

Warren, D. H. D., Pereira, L. M. and Pereira, F. C. N. (1977), Prolog—the language and its implementation compared with Lisp, Proc. ACM Symposium on AI and Programming Languages, *SIGPLAN/SIGART Newsletter* (Rochester NY, August 1977).

van Wijngaarden, A. (Ed.) (1974), Revised Report on the Algorithmic Language Algol-68 (Springer, Berlin, 1976).

Woods, W. A. (1970), Transition network grammars for natural language analysis, *C. ACM* 13 (October 1970).

Woods, W. A. (1973), An experimental parsing system for transition network grammars, in: Randall Rustin (Ed.), *Natural Language Processing* (1973).

Woods, W. A., Kaplan, R. M. and Nash-Webber, B. (1972), The lunar sciences natural language information system: final report, BBN Report 2378 (June 1972).

Woods, W. A. et al. (1976), Speech understanding systems: final report, BBN Report 3438 (December 1976).