DEFINITE CLAUSE TRANSLATION GRAMMARS AND THE LOGICAL SPECIFICATION OF DATA TYPES AS UNAMBIGUOUS CONTEXT FREE GRAMMARS

by

Harvey Abramson

Technical Report 84-11

August 1984

Definite Clause Translation Grammars and the Logical Specification of Data Types as Unambiguous Context Free Grammars

Harvey Abramson

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

ABSTRACT

Data types may be considered as unambiguous context free grammars. The elements of such a data type are the derivation trees of sentences generated by the grammars. Furthermore, the generators and recognizers of non-terminals specified by such grammars provide the composition and decomposition operators which can be used to define functions or predicates over such data types. We present a modification of our Definite Clause Translation Grammars (Abramson 1984) which is used to logically specify data types as unambiguous context free grammars. For example, here is a grammatical specification of binary trees:

leaf :*tree* ::=*string*.

branch :*tree* ::= " (" , *left* :*tree* , " ," , *right* :*tree* , ")".

The decomposition "operators", *left*, *right*, and (implicitly) *string*, are semantic attributes generated by the compiler which translates these grammar rules to Prolog clauses; these operators, together with the parser for *tree*s, and the predicates *leaf* and *branch*, can be used to construct more complex predicates over the data type *tree*. We show how such grammars can be used to impose a typing system on logic programs; and indicate how such grammars can be used to implement Kaviar, a functional programming language based on data types as context free grammars.

August 13, 1984

Definite Clause Translation Grammars and the Logical Specification of Data Types as Unambiguous Context Free Grammars

Harvey Abramson

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

1. Introduction

Several researchers have recently and quite independently converged on the idea that data types can be considered to be unambiguous context free grammars: each non-terminal of a grammar represents a type whose construction is specified by the right hand side of one or more productions; furthermore, the non-terminals in the right hand side of a production act as selectors for decomposing elements of the type, essentially, derivation trees. The constructor and selectors can then be used to define functions or relations over the type and between types specified in this manner.

1.1. Criticism of initial algebra approach.

Kanda and Abrahamson have approached this idea as a result of a critical appraisal of data types considered as initial algebras (Kanda and Abrahamson 1983). To take a very simple example, the initial algebra $I_{(\Sigma,E)}$ for the natural numbers may be specified by:

$$\Sigma = \{0{:}\rightarrow nat\ ,\ s{:}nat \rightarrow nat\ \}$$

$$E = \{\}$$

where 0 is a constant and s is the unary successor operation. This gives the set of natural numbers:

$$\{0, s(0), s(s(0)), \ldots\}$$

and the successor function *succ* as the interpretation of *nat* and s respectively. Since there is nothing in Σ but 0 and s, however, $I_{(\Sigma,E)}$ does not yield any other functions such as, for example, the predecessor operation *pred*.

In order to obtain an interpretation which includes the predecessor function *pred* one must start with a larger signature:

$$\Sigma' = \{0{:}\rightarrow nat\ ,\ s{:}nat \rightarrow nat\ ,\ p{:}nat \rightarrow nat\ \}$$

The initial algebra $I_{(\Sigma',E')}$ generated from this is unsatisfactory in that there are too many terms for the same natural number: $s(0)$, $s(p(s(0)))$, $p(s(s(0)))$ all represent 1. A non-empty set of equational axioms E' must be introduced to enforce equality of terms denoting the same number (see (Kanda and Abrahamson 1983), (Goguen et al. 1978)), and an *undefined* element to totalize the *pred* function.

Kanda and Abrahamson suggest, however, that one could define the natural numbers by the following unambiguous context free grammar:

$$zero{:}nat\ ::=\ "\,0".$$

$$succ{:}nat\ ::=\ "s(",\ nat,\ ")".$$

Here *nat* is the sole non-terminal of the grammar. A *nat* may be 0 or it may be of the form $s(n)$ where n is a *nat*. The names of the productions act as predicates: *zero* is true only when applied to 0; *succ* is true only when it is applied to a *nat* of the form $s(n)$. On the right hand side of the *succ* production, *nat* acts as a selector. Thus we could define the predecessor function using some such clausal notation as:

$$pred(n) = zero(n) \rightarrow undefined.$$

$$pred(n) = succ(n) \rightarrow nat(n).$$

where *undefined* is thrown in to totalize the data type of natural numbers. Using function composition, definition by cases, and general recursive definitions, all functions over the type *nat* can be easily defined without having to respecify initial algebras and augment them with larger and larger sets of equational axioms.

We might also mention here an interesting comment of (Kanda and Abrahamson 1983) which relates polymorphism and language inclusion: if L_1 is a subset of L_2, and f is a function from L_1 to L_1, then f is polymorphic in the sense that it is also a function from L_2 to L_2.

1.2. A grammatical problem of Maluszynski and Nilsson.

Maluszynski and Nilsson reached the conclusion that types might be treated as context free grammars by considering the problem of assigning types to programs of an object language, for example, the lambda calculus as given by the following context free grammar (see (Maluszynski and Nilsson 1981,1982a,b)):

```
prog ::= exp
exp  ::= sexp
exp  ::= appl
sexp ::= const
sexp ::= ident
sexp ::= abstr
sexp ::= "(", exp, ")"
appl ::= appl, sexp
appl ::= sexp, sexp
abstr::= "lambda", ident, "." sexp
```

The problem of assigning types to such programs is the determination or assignment of a unique "type descriptor" to each expression of the language. The program is well-formed if the type assigned to the function part of an application is a "function type" which conforms to the type assigned to the argument expression. (Polymorphism is involved in the assignment of types to some of these lambda expressions.)

In expressing the type assignment rules in the clausal form of logic, Maluszynski and Nilsson note that the string representations of the lambda calculus programs as specified by the grammar above could not be used directly; rather, some representation of these strings as terms had to be introduced, and this, they felt, detracted from the clarity of the clausal description. There was also the problem that the syntax of logic programs did not permit the introduction of any typing of terms other than an arity constraint. If, for example, a term such as *ident*(N) were introduced to represent the *ident* of the grammar above, *ident*$(/1,2,3/)$ might be syntactically correct as far as the logic program for assigning types was concerned, but from the point of view of the computation of types would cause failure of the computation. Such an error could have been ruled out by a static check of the logic program if some type mechanism could have been used to impose a well-formedness condition for the data to which it was applied (see (Maluszynski and Nilsson 1982)).

In order to accomplish this, Maluszynski and Nilsson are led to the notion of using an unambiguous context free grammar (which they call a *metagrammar*) to specify the data domains of a given program. In such a grammar, each non-terminal is associated with the derivation trees whose roots are labelled by the non-terminal. The non-terminals therefore "are sorts of

a many-sorted algebra of derivation trees, whose operations are the production rules of the grammar". In some of the references mentioned, they consider the problem of unifying terms specified by such grammars.

Each of the researchers mentioned above also emphasizes the importance of typing as an aid to the production of correct and reliable software. This is a familiar argument and need not be repeated here.

1.3. Definite Clause Translation Grammars - Background.

Neither of the sources of the notion of using grammars to describe types presents a logical specification (i.e., implementation) of the idea. We show below that a modification of our Definite Clause Translation Grammars can be used as a grammatical typing mechanism. As background to our suggestions, we summarize the original Definite Clause Translation Grammar (DCTG) notation and implementation method. The reader is referred to (Abramson 1984) for a full description of DCTGs.

Definite Clause Translation Grammars (DCTGs) are modelled on the attribute grammars of (Knuth 1968), which, in addition to specifying syntax by context free rules, also specified semantics by attaching "attributes" or properties and rules for evaluating these properties to nodes of a derivation tree. Similarly, a DCTG rule specifies both syntax and semantics. When a string is parsed by the syntactic component of a DCTG, a derivation tree is automatically formed to record the parse; semantic attribute rules, copied from the semantic component of a DCTG, are attached to nodes of this tree. When the tree is traversed the semantic rules are evaluated in order to specify the attributes of a node. The semantic rules are in the form of Horn clauses, and evaluation is accomplished by an interpreter written in Prolog: the semantic rules may be thought of as a local data base containing information as to how global meaning for the entire derivation tree is to be constituted from local meanings attached to nodes of the derivation tree.

The general form of a DCTG rule is:

LeftPart ::= *RightPart* <:> *Semantics*.

The portion of the DCTG rule to the left of the <:> symbol specifies syntax; the portion to the right specifies semantics. For example, here are two rules used in defining *noun_phrase* in a DCTG for a small subset of English (see (Abramson 1984)):

```
noun_phrase ::=
  determiner^^D, noun^^N, rel_clause^^R
<:>
(agree(Num) ::- N^^agree(Num),
               D^^agree(Num),
               R^^agree(Num)),
(logic(X,P1,P) ::- D^^logic(X,P2,P1,P),
                   N^^logic(X,P3),
                   R^^logic(X,P3,P2)).

noun_phrase ::= name^^N
<:>
agree(singular),
(logic(X,P,P) ::- N^^logic(X)).
```

In a DCTG rule, logical variables may be attached by the symbol "^^" to non-terminal symbols in the *RightPart* of the syntactic portion. During parsing, such a logical variable is instantiated to the derivation subtree for the non-terminal to which it is attached. The *Semantics* consist of zero or more Horn clauses specifying various attributes of the non-terminal in the *LeftPart*. Above, the rules for *noun_phrase* have two attributes, *agree* and *logic*. In the second rule, *agree* (*singular*) is a unit clause attribute. In the semantic specifications attached to such a rule, traversal of a subtree to evaluate a semantic attribute is specified by writing, for example, *N^^logic* (*X*,*P*3). This indicates that the *logic* attribute of the *noun* in the rule above is to be

evaluated.

A node in a derivation tree corresponding to use of the first production for *noun_phrase* above has the following representation:

```
node(noun_phrase,
   [D,N,R],
   (agree(Num) ::- N^^agree(Num),
                  D^^agree(Num),
                  R^^agree(Num)),
    (logic(X,P1,P) ::- D^^logic(X,P2,P1,P),
                       N^^logic(X,P3),
                       R^^logic(X,P3,P2)).
```

The second argument is a list of all the subtrees of *noun_phrase* , including terminal symbols, if any; the third argument is accessed by the semantic interpreter during tree traversal.

The basic notion of our typing DCTGs may now be outlined. Given a production defining a non-terminal X, the non-terminals in the right hand side of the production may be treated as *semantic attributes* of X. These attributes may be used either to decompose an object of type X into its constituents or to compose an object of type X from an appropriate set of components. These semantic attributes, in combination with automatically generated type checking predicates, can be used to define relations over a type or between types. The automatically generated type checking predicates may be used either to verify that an object is of a certain type, or to generate an object of a certain type.

Section 2 introduces a modification of Definite Clause Translation Grammars for typing by presenting several simple examples in which we specify types as grammars and subsequently define one or more predicates over or between types. This provides a run-time typing mechanism for Prolog programs. Section 3 explains the implementation as an extension of our Definite Clause Translation Grammars. Section 4 draws some tentative conclusions and indicates future research paths which include the logical specification of *Kaviar* , a functional programming language based on types as context free grammars, and the implementation of a grammatically typed version of Prolog.

2. Typing Definite Clause Translation Grammars: Examples

In this section we shall introduce typing Definite Clause Translation Grammars by presenting several examples, each of them simple, but illustrating various aspects of the notation and different facets of logic programming with types. These typing DCTGs are not identical to the original DCTGs, but there are some similarities of notation, and most importantly, of implementation in Prolog (see Section 3).

2.1. The Natural Numbers.

The natural numbers are specified by the following grammar:

```
zero:natural ::= "0".
succ:natural ::= "s(", natural, ")".
```

Terminal symbols which are strings or lists of characters are enclosed within quotation marks. (Terminals may also be indicated as a list of nullary function symbols, eg, *[symbol].*) The first rule specifies that "0" is a *natural* (*natural* is the only non-terminal of this grammar). It also specifies that the name of this production is *zero* and that there exists a predicate $zero : zero(X)$ is satisfied only if X is a derivation tree whose leaves, read in order from left to right, are in the language generated by this production. The second rule specifies that the other form of a *natural* is "s(" followed by a *natural* followed by ")", for example, s(0) or s(s(0)). The name of this production specifies a predicate $succ : succ(X)$ is satisfied if X is a derivation tree whose leaves read in order from left to right are in the language generated by this grammar.

The relation *pred* specifies that N is the predecessor of X:

```
pred(X,N) :-
  succ(X)^^[natural(N)].
```

succ (X) specifies that X is of type *natural* and is generated by the production named *succ*, ie, X is a derivation tree for some sentence generated by the grammar for *natural*, and the first step in the derivation of the sentence uses the production *succ*. Reading the leaves of X would therefore give us something of the form $s(N)$ where N is a *natural*. The notation introduced consists of an infix operator "^^" from our Definite Clause Translation Grammars which may be read as "with subtrees such that", followed by a list of unary function symbols applied to logical variables. The unary function symbols are in the set of non-terminals which appear in the right hand side of the applicable production, here the production named *succ*. Each such unary function symbol (here only *natural*) selects the relevant sub-derivation tree of the derivation tree named in the predicate to the left of the "^^" operator, and instantiates its argument to it. Thus, if the leaves of X read in order are "s(", "0" and ")", then the natural number N is a subtree of X, consisting of the leaf "0". Since *pred* is a relation, we are also specifying that X is the successor of the natural number N. We can read *succ* (X)^^*[natural* (N)*]* as: for any natural number X not equal to zero, if N is the predecessor of X, then the successor of N is X. In functional notation, we might write that $X = succ(pred(X))$. (We are also being slightly loose here: we should mention "the leaves of X read in order from left to right", etc. Since the grammars we are using are unambiguous we can identify derivation trees with the strings labeling their leaves read in the right order.)

The type *natural* contains all derivation trees generated by the grammar given above. We can specify this by the clauses:

```
type(X,natural) :- zero(X).
type(X,natural) :- succ(X).
```

This is read "X is of type *natural* if it satisfies *zero* or *succ*". We might consider X to be our version of Maluszynski and Nilsson's "grammatical variable of type *natural*", ie, X ranges over derivation trees of sentences in the language generated by the grammar (see (Maluszynski and Nilsson 1981,1982b)). The predicates *zero*, *succ* and *type* are generated automatically as the grammar is compiled into Prolog clauses (see Section 3).

Information about each production for a type is also recorded in unit clauses of *attributes*:

```
attributes(succ,natural,[natural(X)]).
attributes(zero,natural,[]).
```

The first argument is the name of a production, the second a type, and the third a list of the applicable selectors. This is intended for use by an eventual static type checker (see Section 4).

We specify the addition of two natural numbers by the Peano axioms translated into our notation:

```
sum(X,Y,X) :-
  type(X,natural),
  zero(Y).

sum(X,Y,S) :-
  type(X,natural),
  succ(Y)^^[natural(P)],
  succ(S)^^[natural(Q)],
  sum(X,P,Q).
```

If X and Y are instantiated in the second clause of *sum*, that is, if X and Y are derivation trees for natural numbers, then *succ* (S)^^*[natural* (Q)*]* specifies that S is a derivation tree from production *succ* with sub-derivation tree Q, a *natural*; Q is instantiated as a result of the recursive call of *sum*. The predicate *sum* in fact specifies a relation between three naturals. The reader

may verify that if $S = s(s(s(0)))$ then *sum* may be used to find all natural numbers X and Y which add up to S. In this case note that *type* $(X, natural)$ and *succ* (Y)^^*[natural* (P)*]* act as generators of X and Y rather than as type verifiers.

We might note that the two lines of the first clause for *sum* and the first three lines of the second clause are suggestive of the type declarations, with initialization, of some Von Neumann languages; the last line of the second clause is suggestive of the body of a block or procedure.

2.2. Lists.

Here is a grammar which defines simple lists:

```
nonempty:list ::= string, ",", list.
empty:list ::= [].
```

A list is either *empty* or it consists of a *string* followed by a comma followed by a *list*. *string* is a primitive type and consists either of a sequence of numerical characters or of a letter followed by zero or more letters or digits. (The original characters of the sequence are converted to a nullary function symbol.) Thus, 123 and i2 are strings. (The data type *string* is "hand-made" and is defined by a few DCG rules. See Section 3.) The following are, therefore, acceptable lists: 1,2,3, and *abc*,*def*,12,. We shall not list here the clauses of *type* and *attributes* defined for this grammar.

We define the relation *append* between three arguments of type *list*:

```
append(E,X,X) :-
  empty(E),
  type(X,list).

append(X,Y,Z) :-
  nonempty(X)^^[string(A),list(XX)],
  type(Y,list),
  nonempty(Z)^^[string(A),list(ZZ)],
  append(XX,Y,ZZ).
```

The types which have been specified prevent this version of *append* from the poor behaviour of the usual Prolog *append*: with that version of *append*, for example, one may append a list [a,b,c] to a term such as 4, which is not itself a list. This version of *append*, of course, may be used nondeterministically to generate all lists X and Y which when appended yield, for example, *abc*,*def*,12,.

We specify a predicate *length* between the types *list* and *natural* as follows:

```
length(L,Zero) :-
  zero(Zero),
  empty(L).

length(L,N) :-
  type(N,natural),
  nonempty(L)^^[list(L1)],
  length(L1,N1),
  one(One),
  sum(N1,One,N).
```

The predicate *one* specifies the successor of "0":

```
one(One) :-
  zero(Zero),
  succ(One)^^[natural(Zero)].
```

The specifications of the types of $N1$ and *One* could be made explicit by adding *type* $(N1, natural)$, *type* $(One, natural)$ to the definition of the second clause of *length*, but could

also be inferred from the type requirements of *one* and *sum* by a static type checker (see Section 4).

2.3. Trees.

The following grammar specifies the type *tree*:

```
leaf:tree::=string.
branch:tree::= "(", left:tree, ",", right:tree, ")".
```

In the previous grammars we were able to use the names of the non-terminals in the right hand side of a production as the selectors (decomposers) of the type defined by that production, making "puns", so to speak, with the names of the non-terminals in the right hand side: context clarifies whether we are talking of a "selector" or a "type". We cannot do this here since there are two occurrences of the type *tree* in the right hand side of the *branch* production. These occurrences of *tree* are, however, labeled *left* and *right*, and these labels are the selectors of trees which are branches in the *deepreverse* predicate below which reverses a *tree* at all levels. (We need not label both uses of *tree* in the right hand side: labeling one would remove the ambiguity of which subtree was meant). The clauses for *type* are not shown.

```
attributes(branch,tree,[right(R),left(L)]).
attributes(leaf,tree,[string(S)]).

deepreverse(X,X) :- leaf(X).
deepreverse(X,Y) :-
  branch(X)^^[left(Left),right(Right)],
  branch(Y)^^[left(RRight),right(RLeft)],
  deepreverse(Left,RLeft),
  deepreverse(Right,RRight).
```

2.4. Infix and prefix notation.

Our final example is one which was outlined in (Maluszynski and Nilsson 1982b). We specify *infix* and *prefix* expressions and a predicate *fix* which allows one to convert between them. The predicate *fix* may be used, of course, in either direction and may be thought of as specifying a source-to-source translation of a simple kind, translating between expressions such as $a*(b+c)$ and $*a,+b,c$. As in the previous examples, the definition of *fix* is recursive and split into cases depending on the grammatical structure of its arguments. The use of "cut" has not been necessary in any of these examples: the type specifications act as a sort of guard to the "body" of the clausal definitions. The predicates *type* and *attributes* generated for this grammar are not shown.

```
p1:infix ::= expression.
p2:expression ::= term, "+", expression.
p3:expression ::= term.
p4:term ::= primary, "*", term.
p5:term ::= primary.
p6:primary ::= string.
p7:primary ::= "(", expression, ")".

r1:prefix ::= "+", pre1:prefix, ",", pre2:prefix.
r2:prefix ::= "*", pre1:prefix, ",", pre2:prefix.
r3:prefix ::= string.

fix(In,Pre) :-
  p1(In)^^[expression(E)],
  type(Pre,prefix),
  fix(E,Pre).

fix(In,Pre) :-
  p2(In)^^[term(T),expression(E)],
  r1(Pre)^^[pre1(PT),pre2(PE)],
  fix(T,PT),
  fix(E,PE).

fix(In,Pre) :-
  p3(In)^^[term(T)],
  type(Pre,prefix),
  fix(T,Pre).

fix(In,Pre) :-
  p4(In)^^[primary(P),term(T)],
  r2(Pre)^^[pre1(PP),pre2(PT)],
  fix(P,PP),
  fix(T,PT).

fix(In,Pre) :-
  p5(In)^^[primary(P)],
  type(Pre,prefix),
  fix(P,Pre).

fix(In,Pre) :-
  p6(In)^^[string(S)],
  r3(Pre)^^[string(S)].

fix(In,Pre) :-
  p7(In)^^[expression(E)],
  type(Pre,prefix),
  fix(E,Pre).
```

3. Implementation details.

Typing DCTGs differ syntactically from the original DCTGs in that each production must be named (for example, *leaf* and *branch* are the names of the productions defining the type *tree*), and non-terminals in the *RightPart* may optionally be named (for example, *left* and *right* name the different subtrees of a *branch*). Typing DCTGs differ semantically from DCTGs in that the non-terminals (or names of non-terminals) in the *RightPart* are used to construct attributes which may be used for selection and composition. The adaptation of the representation of DCTG parse tree nodes and semantic rules (see Section 2) to typing DCTGs is quite simple and straightforward. A node in a derivation tree which represents use of the production named *branch* in the

grammar of Section 2.3 would look, for example, like:

```
node(branch:tree,
    [ [(], L, [,], R, [)] ],
    (left(L), right(R), true) ).
```

In the DCTG node representation the first argument names only the non-terminal which is at the root of a derivation subtree; in a typing DCTG we also incorporate the name of the production used in forming that node. The second argument is a list of all subtrees of the node. Furthermore, the subtrees indicated above by L and R are *node* structures themselves which insure by unification that L and R are of type *tree* :

```
L = node(N':tree, L', S')
R = node(M':tree, R', T')
```

The semantic attributes of the *branch :tree* node are formed from the names of non-terminals in the right hand side, or if the non-terminals in the right hand side are labeled, from the labels; such attributes are unary function symbols which give access to the relevant subtrees of the node. Considered as Horn clauses for the semantic interpreter, they are unit clauses. We may attach other semantic attributes to typing DCTG rules if we wish. (If a nonterminal z is decorated with a logical variable X as in the original DCTG rules, $z\,\hat{}\,X$, then X is instantiated to the derivation subtree for z, and the selector for z is compiled as $z(X)$. Semantic attribute rules which go into the local data base for such a rule may then traverse X to evaluate attributes, eg, $X\,\hat{}\,logic(A,B)$. Semantic rules are specified as in the original DCTG rule format following a <:> symbol. None of the examples illustrated this possibility, however. A later paper will include such examples. See also Section 4.) The generated attributes are appended to any such specified attributes to form the third argument of a node. The atom *true* is used as an empty marker in a *node* for DCTG productions in which no semantic rules have been explicitly specified.

The method used to compile typing DCTG rules into Prolog clauses is a variant of the one used to compile DCTG rules into Prolog clauses, and need not be detailed here (see (Abramson 1984)). The differences are that:

[1] semantic typing attributes are formed and added to the list of other specified semantic attributes;

[2] a predicate is formed from the name of each production and asserted; and,

[3] clauses for *type* and *attributes* are asserted for each typing DCTG rule.

In a brief appendix we list the predicates *t_lp* and *t_rp* (translate left part, right part) used in compiling typing DCTG into Prolog clauses: these differ from the corresponding predicates in (Abramson 1984); all other predicates used in the compilation are identical to those in the cited reference.

Here is the way the grammar for natural numbers appears as Prolog clauses:

```
natural(node(zero:natural,[[0]],true),S0,S1) :-
  c(S0,0,S1).

natural(node(succ:natural,
   [[s,'('],node(Name:natural,Nodes,Semantics),[')']],
   (natural(node(Name:natural,Nodes,Semantics)),true)),
   S0,S4) :-
   c(S0,s,S1),
   c(S1,'(',S2),
   natural(node(Name:natural,Nodes,Semantics),S2,S3),
   c(S3,')',S4).
```

The first argument to *natural* is instantiated to the derivation tree, the second and third are the "input" and "output" lists used in parsing. The following list of goals shows how the Prolog

parser *natural* of arity 3 may be used:

```
:- natural(N1,"s(s(0))",[]),
   natural(N2,"s(s(s(0)))",[]),
   sum(N1,N2,N3),
   writetype(N3).
```

writetype is a polymorphic predicate which traverses the derivation tree $N3$, printing its leaves from left to right to obtain:

```
s(s(s(s(s(0))))).
```

The interpreter which traverses derivation trees and evaluates semantic attributes of typing DCTG rules follows:

```
node(_,_,Sem)^^Args :- Sem^^Args.

X^^SpecifyList :-
  X =.. [Type,Y],
  X, !,
  specify(Y,SpecifyList).

((Args::-Traverse),Rules)^^Args :-
  !, Traverse.

(Args,Rules)^^Args :- !.

(_,Rules)^^Args :-
  Rules^^Args.

(Args::-Traverse)^^Args :- !, Traverse.

Args^^Args.

specify(X,[]) :- !.

specify(X,[Tree|Trees]) :-
  X^^Tree,
  specify(X,Trees).
```

The second clause of "^^" is the one which is used to evaluate, for example, $branch(X)^^[left(Left),right(Right)]$ in Section 2.3 above. The argument to the left of "^^" is verified to be a unary function symbol and is called. If the call is successful, for example, if $branch(X)$ succeeds, then *specify* accesses the relevant subtrees of the *branch* instantiating $Left$ and $Right$; or, if $branch(X)^^[left(Left),right(Right)]$ is being used to generate a *branch*, then *specify* instantiates the subtrees of X to $Left$ and $Right$. The remaining clauses of "^^" complete the specification of the semantic interpreter, and are applicable also to pure DCTG rules.

As a final point, we mention that the primitive type *string* is defined by several Definite Clause Grammar rules which act as a lexical scanner for identifiers and numbers. Extra arguments attached to non-terminals of these DCG rules are used to produce nodes of the kind described above for the "hand-made" type *string* with the following semantic attributes: if a *string* is a sequence of numerical characters, we may access the numerical atom formed from the characters by:

```
string(S)^^[value(V)]
```

if a *string* begins with a letter which is followed by zero or more letters or digits, we may access the atom formed from those characters by:

```
string(S)^^[id(I)]
```

Listings of the implementation of typing Definite Clause Translation Grammars and all the examples are available from the author.

4. Summary and future research.

Typing DCTGs provide a very simple method of imposing an *optional* type discipline on logic programs. In effect, they allow one to program directly with that most useful of all data structures, the tree. Context free grammars, suitably analysed, provide all the necessary selectors and constructors for manipulating derivation trees of the grammars.

Traditionally, grammars have been viewed as devices for analysing and generating terminal strings of a language. Here, the focus is on the derivation trees, possibly only partially instantiated, which the grammar can analyse and synthesise: the derivation trees of terminal strings are a special case. Typing DCTGs can, of course, be used in the traditional manner to parse an input string: witness in Section 3 the example of how the parser *natural* (*Tree* ,*Input* ,*Output*) is used. Once the string has been parsed, however, the interest generally lies in manipulation of *Tree* .

The compilation of DCTG rules to Prolog clauses, in fact, forces a restriction on the use of typing DCTGs as *parsers* : Prolog's top down left to right strategy rules out left recursive productions. Practically, this is not too important. Just as all practical languages can be parsed by recursive descent using grammars without left recursion, so it is asssumed that all reasonable data structures can be specified by typing DCTGs without left recursive productions. The restriction could be removed by using Earley's general context free parsing algorithm (see (Aho and Ullman 1973)); this would permit one to specify all data structures in their most "natural" form even if it meant using left recursive rules.

The restriction, we repeat, applies to DCTGs used as Prolog *parsers* . If one uses the generators and composers of a grammar to form derivation trees without parsing, then even left recursive rules may be used:

```
leaf:tree ::= string.
branch:tree ::= left:tree, right:tree.
```

These rules specify a more abstract version of *tree* than the grammar of Section 2.3. *type* and *attributes* are:

```
type(X,tree) :- leaf(X).
type(X,tree) :- branch(X).

attributes(leaf,tree,[string(S)]).
attributes(branch,tree,[left(L),right(R)]).
```

This type definition differs from the previous one in the *node* structure which represents *branch* es. Elements of this type *tree* can be formed using *string* , *left*, *right*, *leaf* and *branch* without parsing input strings at all, and indeed, must be so formed! There is a connection to be explored in a later paper between such a grammatical specification (which is used to specify trees rather than a parser) and the Puzzle Grammars of (Sabatier 1984).

We note that we could extend typing DCTGs to specify data types which are *not* context free by placing restrictions or constraints on derivation trees. For example:

```
is_p:prime ::= natural^^P, { is_prime(P) }
```

would specify a natural number which is verified by a Prolog predicate *is_prime* to be a prime number. In writing predicates over this type, we would have to allow Prolog predicates to be included in the specification lists operated on by "^^":

```
is_p(X)^^[natural(P), {is_prime(P)}].
```

In order to keep the compilation of typing DCTG productions to Prolog clauses simple, the generated predicate *is_p* would only require that *X* be a *natural* ; the additional constraint must then be applied. It seems, since the constraints might appear anywhere in a DCTG rule, and might involve any number of nonterminals, that this simple scheme is preferable to trying to incorporate the constraints in the predicate generated from the name of the production.

Another interesting line of investigation which we shall follow is to permit types to be parameterized, eg, *tree* s not only with *string* leaves, but with any other type such as *natural*, *list*, etc. A later report will detail these extensions.

Typing DCTGs provide a run-time type checking mechanism based on unification. This can prove expensive for large data structures. One way to remedy this would be to provide a syntactic sugaring of Prolog programs which would permit types to be specified and statically checked by a grammar based type checker (see (Kanda and Abrahamson 1983) for an indication as to how such a type checker would function; other typing schemes for logic programs and Prolog have been suggested in (Mycroft and O'Keefe 1983), (Mishra 1984)). A well-typed Prolog program could then utilize a simpler and more efficient representation of types than the *node* structure shown above, and potentially expensive unifications could be avoided: the program would work because it had been shown to be well-typed.

One other line of research is the design of a simple functional programming language which utilizes the notion of types as context free grammars. This language is intended for instructional purposes and is to be called *Kaviar* (the name is formed from the italicized letters in its designers' names: Akira *Ka* nda, *Vi* olet Syrotiuk, and Har vey Abramson). A report on its design and implementation both in logic and in *C* is being prepared. The logic implementation will rely on typing DCTGs and the transformation of Kaviar functions into Prolog predicates.

Acknowledgements.

I would like to thank my colleagues on the *Kaviar* project, Dr. Akira Kanda and Violet Syrotiuk, for many helpful discussions. This work was supported by the National Science and Engineering Research Council of Canada. I must also thank the UBC Laboratory for Computational Vision for time on its VAX running Berkeley Unix. The UBC Computing Centre is in no way to be thanked for not supplying modern and adequate Unix-based research computing facilities to the Computer Science Department.

References.

Abramson, H., *Definite Clause Translation Grammars,* Proceedings 1984 International Symposium on Logic Programming, Feb. 6-9, 1984, Atlantic City, New Jersey, pp. 233-241.

Aho, A.V. & Ullman, J.D., *The Theory of Parsing, Translation, and Compiling,* 2 volumes, Prentice-Hall, 1973.

Colmerauer, A., *Metamorphosis Grammars,* in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

Goguen, Thatcher, Wagner & Wright, *Initial algebra approach to specification, correctness and implementation of abstract data types,* in R. Yeh (editor) Current Trends in Programming Methodology, Prentice-Hall, 1978.

Kanda, A. & Abrahamson, K. *Data types as term algebras,* University of British Columbia, Department of Computer Science Technical Report 83-2, March 1983.

Kanda, A. & Abramson, H. & Syrotiuk, V. *Kaviar, a Functional Programming Language Based on Data Types as Context Free Languages,* in preparation.

Knuth, D.E., *Semantics of Context-Free Languages,* Mathematical Systems Theory, vol. 2, no. 2, 1968, pp. 127-145.

Maluszynski, J. & Nilsson, J.F. *A notion of grammatical unification applicable to logic programming languages* Department of Computer Science, Technical University of Denmark, Doc. ID 967, August 1981.

Maluszynski, J. & Nilsson, J.F. *A version of Prolog based on the notion of two-level grammar.* Prolog Programming Environments Workshop, Linkoping University, March 25-27, 1982a.

Maluszynski, J. & Nilsson, J.F. *Grammatical Unification,* Information Processing Letters, vol. 15 no. 4, October, 1982b.

Mishra, P. *Towards a theory of types in Prolog,* Proceedings 1984 International Symposium on Logic Programming, Feb. 6-9, 1984, Atlantic City, New Jersey, pp. 289-298.

Mycroft, A. & O'Keefe, R. *A polymorphic type system for Prolog,* Proceedings Logic Programming Workshop '83, 26 June - 1 July 1983, Praia da Falesia, Algarve, Portugal.

Pereira, F.C.N. (editor), *C-Prolog User's Manual,* University of Edinburgh, Department of Architecture, 1982.

Sabatier, P. *Puzzle Grammars,* Proceedings of the Workshop on Natural Language Understanding and Logic Programming, Rennes, France, Sept. 18-20, 1984.

Appendix

```
t_lp(Name:LP,StL,S,SR,TypeSem,Semantics,H) :-
  add_extra_args([node(Name:LP,StL,Semantics),S,SR],LP,H),
  IsName =.. [Name,node(Name:LP,StL,Semantics)],
  asserta(IsName),
  CallIsName =.. [Name,X],
  assert((type(X,LP) :- CallIsName)),
  reverse(TypeSem,RTypeSem),
  asserta(attributes(Name,LP,RTypeSem)).

t_rp(!,St,St,S,S,!,Sem,Sem) :- !.

t_rp([],St,[[]|St],S,S1,S=S1,Sem,Sem) :- !.

t_rp([X],St,[[NX]|St],S,SR,c(S,X,SR),Sem,Sem) :-
  char(X,NX).

t_rp([X],St,[[X]|St],S,SR,c(S,X,SR),Sem,Sem) :- !.

t_rp([X|R],St,[[NX|NR]|St],S,SR,(c(S,X,SR1),RB),Sem,Sem) :-
  char(X,NX),
  t_rp(R,St,[NR|St],SR1,SR,RB,Sem,Sem).

t_rp([X|R],St,[[X|R]|St],S,SR,(c(S,X,SR1),RB),Sem,Sem) :- !,
  t_rp(R,St,[R|St],SR1,SR,RB,Sem,Sem).

t_rp({Prolog},St,St,S,S,Prolog,Sem,[{Prolog}|Sem]) :- !.

t_rp((T,R),St,StR,S,SR,(Tt,Rt),Sem,TypeSem) :- !,
  t_rp(T,St,St1,S,SR1,Tt,Sem,Sem1),
  t_rp(R,St1,StR,SR1,SR,Rt,Sem1,TypeSem).

t_rp(Name:T,St,[N|St],S,SR,Tt,Sem,[Type|Sem]) :-
  add_extra_args([N,S,SR],T,Tt),
  N = node(Name1:T,Nodes,Semantics),
  Type =.. [Name,N].

t_rp(Name:T^^N,St,[N|St],S,SR,Tt,Sem,[Type|Sem]) :-
  add_extra_args([N,S,SR],T,Tt),
  N = node(Name1:T,Nodes,Semantics),
  Type =.. [Name,N].

t_rp(T^^N,St,[N|St],S,SR,Tt,Sem,[Type|Sem]) :-
  add_extra_args([N,S,SR],T,Tt),
  N = node(Name1:T,Nodes,Semantics),
  Type =.. [T,N].

t_rp(T,St,[N|St],S,SR,Tt,Sem,[Type|Sem]) :-
  add_extra_args([N,S,SR],T,Tt),
  N = node(Name1:T,Nodes,Semantics),
  Type =.. [T,N].
```