

Washington University in St. Louis

## Washington University Open Scholarship

---

All Computer Science and Engineering  
Research

Computer Science and Engineering

---

Report Number: WUCS-82-11

1982-07-01

### ADS Syntax and Command Trees

Will D. Gillett and Takayuki Kimura

Follow this and additional works at: [https://openscholarship.wustl.edu/cse\\_research](https://openscholarship.wustl.edu/cse_research)

---

#### Recommended Citation

Gillett, Will D. and Kimura, Takayuki, "ADS Syntax and Command Trees" Report Number: WUCS-82-11 (1982). *All Computer Science and Engineering Research*.  
[https://openscholarship.wustl.edu/cse\\_research/890](https://openscholarship.wustl.edu/cse_research/890)

Department of Computer Science & Engineering - Washington University in St. Louis  
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

ADS SYNTAX AND COMMAND TREES

W. D. Gillett

T. D. Kimura

WUCS-82-11

Department of Computer Science

Washington University

St. Louis, Missouri 63130

July, 1982

## 1. Introduction

This report describes and relates the ADS syntax and the ADS command trees. The ADS syntax describes the acceptable input to the ADS Compiler, and the ADS command trees are the output of the Compiler. The command trees represent the actual language interpreted by the ADS Interpreter. The ADS syntax represents a meta-language for specifying the ADS command trees, and this report defines the correspondence (or translation) between the two languages.

Section 2 describes the notations used in this report; Section 3 presents the ADS syntax by use of a context-free grammar; Section 4 defines the ADS command trees as a function of the ADS syntax by use of an attributed grammar.

## 2. Notations

A standard mechanism for describing the syntax of a language is a context-free grammar. However, besides describing the syntax of a language, we also want to describe its semantics. Unfortunately, a simple context-free grammar is not capable of expressing semantics. However, a simple extension known as an attributed grammar, is capable of expressing our semantics. An attributed grammar will be used to specify the translation from the ADS syntax to ADS command trees.

In order to describe the ADS command trees, several notations will be used. A very visual notation is simply a drawing of the tree; this will be called the tree form (Section 2.2). This notation will sometimes be used for clarity, but requires a tremendous amount of space (on a printed page). For brevity, an equally precise (but less

intuitive) notation also will be used; this will be called the linearized form (Section 2.3).

## 2.1 Attributed Grammars

Attributed grammars are a simple extension of conventional context-free grammars; there are terminals, nonterminals, context-free productions, and a starting symbol. However, each terminal and nonterminal may have a fixed number of attributes. These attributes represent the semantics or meaning of the corresponding symbol. In any given production rule of the grammar, a given attribute may be defined in terms of other attributes and/or constants. In this way, the semantics of a specific occurrence of a nonterminal in the parse tree can be defined in terms of the semantics of its component terminals and nonterminals.

As an example of an attributed grammar, consider the arithmetic expression grammar below. The attributes are notationally written as subscripts on the nonterminals. An arbitrary symbol written as the subscript is a surrogate or place holder for the actual value of the attributes, which will be determined during the processing of an actual input string. The actions in square brackets ([ ]) below each production specify the relationship between the values of the attributes. The '<->' symbol represents the assignment operator.

Attributed Grammar for Arithmetic Expressions

```

(1)  $\text{exp}_{a_1} ::= \text{exp}_{a_2} '+' \text{term}_{a_3}$ 
       $[a_1 \leftarrow a_2 + a_3]$ 
(2)  $| \text{term}_{a_2}$ 
       $[a_1 \leftarrow a_2]$ 
      ;
(3)  $\text{term}_{a_1} ::= \text{term}_{a_2} '*' \text{prim}_{a_3}$ 
       $[a_1 \leftarrow a_2 * a_3]$ 
(4)  $| \text{prim}_{a_2}$ 
       $[a_1 \leftarrow a_2]$ 
      ;
(5)  $\text{prim}_{a_1} ::= '(' \text{exp}_{a_2} ')'$ 
       $[a_1 \leftarrow a_2]$ 
(6)  $| \text{NUMBER}$ 
       $[a_1 \leftarrow \text{value}(\text{NUMBER})]$ 
      ;

```

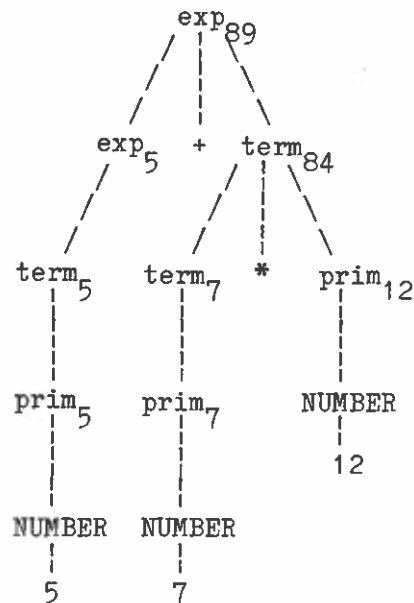
In this grammar, 'NUMBER' corresponds to the generic terminal class of character strings that denote numbers; any character string belonging to this class is identified with NUMBER. The function 'value' extracts the numeric value corresponding to the character string.

If the attributes (subscripts) and actions (things in square brackets) are deleted from the attributed grammar, a standard context-free grammar is obtained; this is known as the input grammar. Of course, this could be used as a generative grammar, but we are interested in it as a parsing grammar. We will use attributed grammars to translate from one language to another.

Conceptually, attributed grammars are used as translation tools in the following way:

- 1) The input grammar is obtained by deleting the attributes and action symbols.
- 2) The input string to be translated is parsed using the input grammar. Any parsing method is applicable. The result of the parse is a parse tree.
- 3) The attributes specified in the attributed grammar are now introduced into the parse tree by placing the values of the attributes on each instantiation of the appropriate nonterminals in the parse tree. This is normally done in a bottom-up manner. The values of unassigned attributes are determined by use of the actions associated with the productions. The attributes and actions are local to a specific production and its corresponding counterpart in the parse tree. In other words, the scope of the value of the attributes and the actions that relate them are restricted to one parent node and its immediate descendants in the parse tree.
- 4) Once the values of all attributes have been determined, the desired semantics (values of attributes) can be extracted from the appropriate nonterminals of the parse tree. Usually, the attribute of the top-most starting symbol is the desired information.

As an example, consider the above grammar for arithmetic expressions and the input string '5+7\*12'. The parse tree with instantiated attributes is given by:

Parse Tree with Attributes

Consider the occurrence of 'prim<sub>12</sub>' in the parse tree. The value 12 corresponds to the attribute  $a_1$  in production (6) of the grammar. Its value is determined by the corresponding action, 'a<sub>1</sub> ← value(NUMBER)', where NUMBER has the value 12. In the occurrence of 'term<sub>84</sub>', the value 84 (attribute  $a_1$ ) is obtained by applying the action of production (3), where  $a_2 = 7$  and  $a_3 = 12$ . Of course, the value of the entire expression, 89, is the attribute of the root node of the parse tree.

Conceptually, we have used the above attributed grammar to translate from character strings representing arithmetic expressions to numeric values. More specifically, if the input string is '5+7\*12', then the output value is 89.

2.2 Tree Form

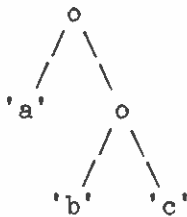
All ADS command trees are binary trees whose structure can be easily described by a simple graphic notation. The root node is represented by the top node which has no ancestors. Internal nodes are represented by a 'o' having one ancestor and two descendants, a left son and a right son. Leaf nodes are represented by an arbitrary character string having one ancestor and no descendants.

Example 1:

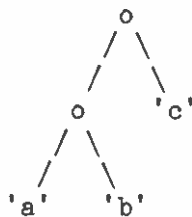
A) A tree with a left son of 'a' and a right son of 'b':



B) A tree with a left son of 'a' and a right son which is a tree with a left son of 'b' and a right son of 'c':



C) A tree whose left son is a tree with a left son of 'a' and a right son of 'b', and whose right son is 'c':





### 2.3 Linearized Form

Although the tree form is very expressive, it is rather verbose. A more concise notation is to "flatten" the two-dimensional tree form into a one-dimensional linearized form. In this notation, the list of (left) subtrees along the (right) spine of the tree are written sequentially from left to right. If the subtree is a leaf node, it is written without parentheses; otherwise, it is embedded in parentheses. This notational scheme is applied recursively to the (left) subtrees.

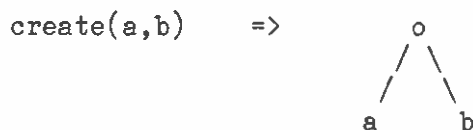
#### Example 2:

The linearized forms of the three trees of Example 1 are:

- A) a b
- B) a b c
- C) (a b) c

An attributed grammar will be used to define the semantics of the linearized form. In the semantics associated with this grammar, there are two primitive functions, encode and create. Create returns an internal node (tree) whose left son is the first argument and whose right son is the second argument.

Example:



Encode returns an arbitrary but fixed encodement of the corresponding TOKEN. In this context, TOKEN is a specific object of interest scanned

from the input. If the TOKEN is a fixed constant, such as punctuation or an operator, the encodement returned is a leaf node. If TOKEN is a generic string, such as a symbol name or a variable, the encodement is a tree representing the generic token.

The following attributed grammar defines the relationship (translation) between the linearized form and the tree form.

Attributed Grammar for Linearized Form

```

(1) treet1 ::= primp treet2
                |
                | [t1 ← create(p,t2)]
(2)          | primp
                | [t1 ← p]
                ;

(3) primp ::= '(' treet ')'
                |
                | [p ← t]
(4)          | TOKEN
                | [p ← encode(TOKEN)]
                ;

```

The input grammar specifies the legal input strings. The desired translation is the value of the attribute of the top-most occurrence of 'tree' in the parse tree corresponding to a specific input string.

Example 3:

<u>Input String</u>	<u>Corresponding Output</u>
-------------------------	---------------------------------

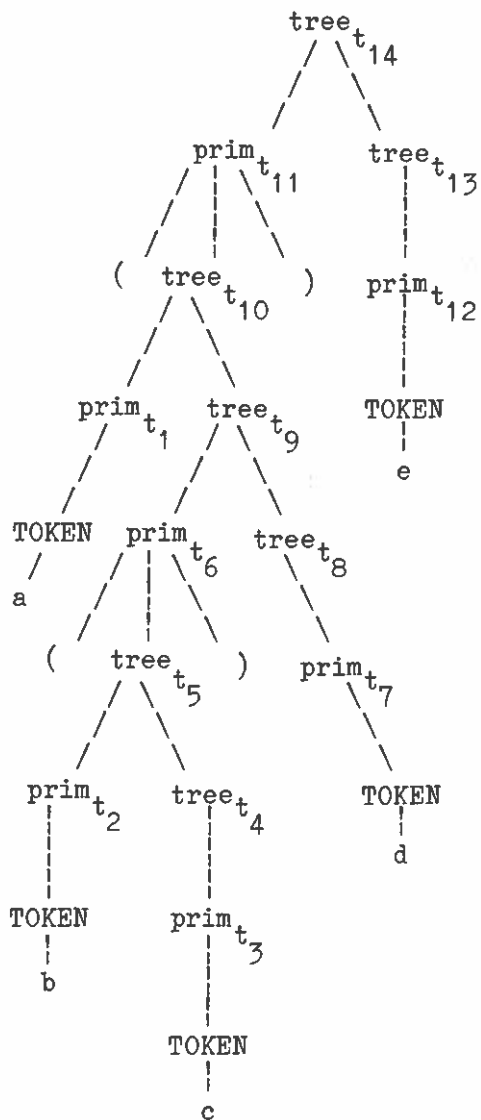
A) a b c	=>	<pre>       o      / \     'a'  o          / \         'b' 'c' </pre>
----------	----	---

B) a (b c) d	=>	<pre>       o      / \     'a'  o          / \         o  'd'        / \       'b' 'c' </pre>
--------------	----	---

C) (a (b c) d) e	=>	<pre>       o      / \     o   'e'    / \   'a'  o        / \       o  'd'      / \     'b' 'c' </pre>
------------------	----	--

The parse tree and attributes for example 3C above are given below. In order to avoid confusion between the parse tree and the attributes, which are also trees, the actual values of the attributes are not presented in the parse tree itself.

Parse Tree for '(a (b c) d) e'



Values of the Attributes

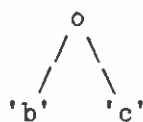
t<sub>1</sub> = 'a'

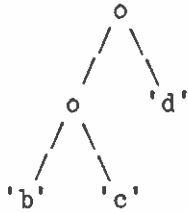
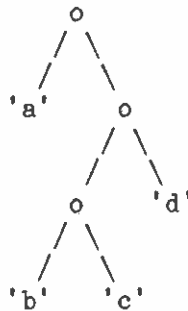
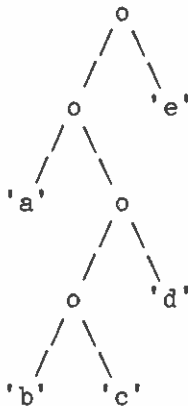
t<sub>2</sub> = 'b'

t<sub>3</sub> = 'c'

t<sub>4</sub> = t<sub>3</sub>

t<sub>5</sub> =



$t_6 = t_5$ 
 $t_7 = 'd'$ 
 $t_8 = t_7$ 
 $t_9 =$ 

 $t_{10} =$ 

 $t_{11} = t_{10}$ 
 $t_{12} = 'e'$ 
 $t_{13} = t_{12}$ 
 $t_{14} =$ 


Consider the occurrence of  $\text{prim}_{t_1}$  in the parse tree and the attribute value  $t_1 = 'a'$ . Attribute  $t_1$  in the parse tree corresponds to the attribute  $p$  in production (4), and its value is determined by the action  $p \leftarrow \text{encode}(\text{TOKEN})$ . Attribute  $t_5$  in the

parse tree corresponds to the attribute  $t_1$  in production (1), and its value is determined by the action ' $t_1 \leftarrow \text{create}(p, t_2)$ ', where  $p$  ( $t_2$  in the parse tree) has the value 'b' and  $t_2$  ( $t_4$  in the parse tree) has the value 'c'. Notice that the attribute on any specific nonterminal is local to its specific instantiation in the parse tree and is not affected by any other instantiation of that nonterminal in the parse tree. The value of the attribute ( $t_{14}$ ) of the root node of the parse tree is the desired binary tree. This is not surprising since this is the intended meaning of the original input string '(a (b c) d) e'. Also notice that  $t_{10}$  corresponds to example 3B above.

### 3. The ADS Syntax Grammar

In the context-free grammar below, many tokens are used but left undefined. Of course, these are defined at the lexical level. Before presenting the grammar, a brief explanation of these tokens is given.

<u>Token</u>	<u>Comment</u>
'<'	begin transaction
'>'	end transaction
'N'	the NIL tree
'T'	the logical value Truth
'F'	the logical value Falsehood
';'	the statement separator; also used in conjunction with the YIELDS and IMPLIES operators
ESCAPE	a special symbol which indicates that the following input is part of the system manipulation language, and not part of the ADS language
'?F'	request for output of a formula
'?S'	request for output of a symbol (binary tree)
'?P'	request for output of a predicate
'?T'	request for output of a transformer
' '	the assert statement symbol
'- '	the deassert statement symbol
'=='	defines the association between an intensional descriptor and a name
'==='	transfers the intensional descriptor from one name to a second name

':='	defines the association between an extensional descriptor and a name
'.'	the APPLY operator; used to evaluate a transformer or predicate at a point
'..'	the IS operator; used to test set membership through use of a predicate name
'(', ')'	parentheses for grouping
'+'	the BUILD operator; used to build binary trees
'"', ''''	transparent and opaque quotes
'@'	the MU operator; extracts the intensional descriptor of a name
'#'	the TAU operator; extracts the extensional descriptor of a name
'=>'	the YIELDS operator; used for conditional evaluation of symbols
'∨', '∧', '∼'	the OR, AND, and NOT operators
'='	the EQUALITY operator; used to compare symbols (binary trees)
'/' o '/'	the DEFINED operator; used to determine if an expression is defined
'\' o '\'	the TRUE operator; used to determine if a formula is both defined and true
'→'	the IMPLIES operator; used for conditional evaluation of formulas
':'	the TYPE operator; used in forms to guarantee type
'[' o ']'	the EVALUATE operator; used to evaluate the meaning of an expression
'λ'	the LAMBDA operator; used to abstract symbols
'ρ'	the RHO operator; used to abstract formulas
'ι'	the IOTA operator; used to describe symbols
'∀'	the FORALL operator
'∃'	the THEREEXISTS operator
SN	a symbol name
FN	a formula name
PN	a predicate name
TN	a transformer name
VAR	a variable name

In the following grammar, 'trans' is the starting symbol. Terminal symbols (tokens) are placed in quotes and/or capitalized as shown in the table above. Nonterminals appear in lower case. This grammar is the input grammar that corresponds to the attributed grammar presented in the next section.

Context-free Grammar for ADS

```

trans      ::= state
           | '<' statelist '>'
           ;

statelist  ::= state
           | state ';' statelist
           ;

state      ::= update
           | query
           | ESCAPE
           ;

query      ::= '?F' fexpE
           | '?S' sexpE
           | '?P' pexpE
           | '?T' texpE
           ;

update     ::= '|-' definition
           | '|-' fact
           | '|-' name
           | '|-' fact
           ;

definition ::= SN '==' sexpE
           | FN '==' fexpE
           | TN '==' texpE
           | PN '==' pexpE
           | SN '===' SN
           | FN '===' FN
           | TN '===' TN
           | PN '===' PN
           ;

fact       ::= SN ':=' sexpE
           | FN ':=' fexpE
           | TN ':=' texpE
           | PN ':=' pexpE
           | ssfact
           | fffact
           ;

ssfact     ::= ssexp ':=' ssexpE
           | ssexp ':=' tterm
           ;

ssexp      ::= stermE '.' TN
           | stermE '.' ssexp
           ;

```



```

ffact ::= ffexp ':' fexpE
      ffexp ':' pterm
      ;

ffexp ::= sterme '.' PN
      sterme '.' ffexp
      sterme '..' PN
      sterme '..' ffexp
      ;

dexp ::= sexp
      fexp
      E
      ;

sexp ::= sterm
      sterme '.' texpE
      ;

sterm ::= '(' sexp ')'
      sfact
      ;

sfact ::= sprim
      '+' sterme sterme
      ;

sprim ::= 'N'
        SN
        NUM
        VAR
        ''' lexunit '''
        "" lexunit ""
        '@' name
        '#' SN
        iotaexp
        '(' fexpE '=>' sexpE ';' sexpE ')'
        '(' fexpE '=>' sexpE ')'
      ;

iotaexp ::= '(' '↑' form ')' fprimE
        ;

fexp ::= fterm
      fexpE '√' ftermE
      ;

fterm ::= ffact
      fterm '^' ffact
      ;

ffact ::= ffact2
      '~' ffact2
      ;

```

```

ffact2      ::=      stermE '.' pexpE
              stermE '=' stermE
              stermE '..' pexpE
              fprim
              ;

fprim       ::=      '(' fexpE ')'
              fprim2
              ;

fprim2      ::=      'T'
              'F'
              FN
              '#' FN
              '/' dexp '/'
              '\' fexpE '\'
              allexp
              existexp
              '(' fexpE '→' fexpE ';' fexpE ')'
              '(' fexpE '→' fexpE ')'
              ;

allexp      ::=      '(' '∀' form ')' fprimE
              ;

existexp    ::=      '(' '∃' form ')' fprimE
              ;

texp        ::=      tterm
              stermE '.' texpE
              ;

tterm       ::=      TN
              '#' TN
              tprim
              ;

tprim       ::=      '(' 'λ' form ')' stermE
              ;

pexp        ::=      pterm
              stermE '.' pexpE
              stermE '..' pexpE
              ;

pterm       ::=      PN
              '#' PN
              pprim
              ;

pprim       ::=      '(' 'ρ' form ')' fpprimE
              ;

```

```

name      ::= SN
           FN
           TN
           PN
           ;

form      ::= form1 ':' pexpE
           form1
           ;

form1     ::= VAR
           '(' form ')'
           '+' form1 form1
           ;

sexpE     ::= sexp
           E
           ;

fexpE     ::= fexp
           E
           ;

texpE     ::= texp
           E
           ;

pexpE     ::= pexp
           E
           ;

stermE    ::= sterm
           E
           ;

ftermE    ::= fterm
           E
           ;

E         ::= '[' sexpE ']'
           ;

fprimE    ::= fprim
           E
           ;

lexunit   ::= sexp
           fexp
           pterm
           tterm
           E
           token
           ;

```

```

token      ::= '
'
           'Δ'
           'Π'
           'λ'
           'ε'
           '@'
           '#'
           '→'
           ';'
           '<'
           '>'
           '.'
           '='
           ':'
           '>'
           ;

sttermE    ::= sterm
           tprim
           E
           ;

fpprimE    ::= fprim
           pprim
           E
           ;

```

#### 4. Definition of ADS Command Trees

The above input grammar specifies the context-free syntax of the ADS language. By adding attributes and actions that relate the values of the attributes, the following attributed grammar is obtained. The grammar below defines the relationship (translation) between the ADS syntax and the ADS command trees. The value of the attribute of 'trans' in the parse tree of a specific input string is the desired translation, i.e., it is the ADS command tree that corresponds to the specific input string.

Attributed Grammar for ADS

```

transa1 ::= statea2
              [a1 ← create(a2, 'N')]
            | '<' statelista2 '>'
              [a1 ← a2]
            ;

statelista1 ::= statea2
                  [a1 ← create(a2, 'N')]
                | statea2 ';' statelista3
                  [a1 ← create(a2, a3)]
                ;

statea1 ::= updatea2
              [a1 ← a2]
            | querya2
              [a1 ← a2]
            | ESCAPE
              [a1 ← special tree]
            ;

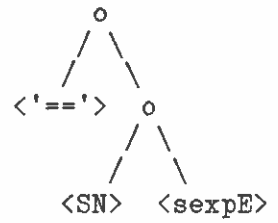
querya1 ::= '?F' fexpEa2
              [a1 ← create(encode('?F'), a2)]
            | '?S' sexpEa2
              [a1 ← create(encode('?S'), a2)]
            | '?P' pexpEa2
              [a1 ← create(encode('?P'), a2)]

```

```

|      '?T' texpEa2
|      [a1 ← create(encode('?T'),a2)]
;

updatea1 ::=
|      '┊' definitiona2
|      [a1 ← create(encode('┊'),a2)]
|      '┊' facta2
|      [a1 ← create(encode('┊'),a2)]
|      '┊' namea2
|      [a1 ← create(encode('┊'),a2)]
|      '┊' facta2
|      [a1 ← create(encode('┊'),a2)]
;

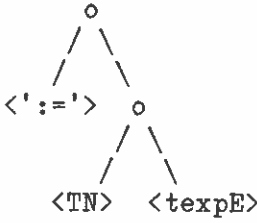
definitiona1 ::= SN '==' sexpEa2
|      [a1 ← create(encode('=='),
|      create(encode(SN),a2))]
|      /*
|      
|      */
|      FN '==' fexpEa2
|      [a1 ← create(encode('=='),
|      create(encode(FN),a2))]
|      TN '==' texpEa2
|      [a1 ← create(encode('=='),
|      create(encode(TN),a2))]

```



```

facta1 ::= SN ':= ' sexpEa2
           [a1 ← create(encode(':='),
                        create(encode(SN),a2))]
|
  FN ':= ' fexpE
           [a1 ← create(encode(':='),
                        create(encode(FN),a2))]
|
  TN ':= ' texpEa2
           [a1 ← create(encode(':='),
                        create(encode(TN),a2))]

           /*
           
           */
|
  PN ':= ' pexpEa2
           [a1 ← create(encode(':='),
                        create(encode(PN),a2))]
|
  ssfacta2
           [a1 ← a2]
|
  fffacta2
           [a1 ← a2]
;

ssfacta1 ::= ssexpa2 ':= ' sexpa3
           [a1 ← create(encode(':='),
                        create(a2,a3))]
|
  ssexpa2 ':= ' tterma3
           [a1 ← create(encode(':='),
                        create(a2,a3))]

```



```

;
ssexpa1 ::= stermEa2 '.' TN
              [a1 ← create(encode('...'),
                           create(a2, encode(TN)))]
              /*
                o
               / \
              /   \
             <'.'>  o
                   / \
                  /   \
                 <stermE> <TN>
              */
              */
|
stermEa2 '.' ssexpa3
              [a1 ← create(encode('...'),
                           create(a2, a3))]
;

ffacta1 ::= ffexpa2 ':' fexpEa3
              [a1 ← create(encode(':', '='),
                           create(a2, a3))]
|
ffexpa2 ':' pterma3
              [a1 ← create(encode(':', '='),
                           create(a2, a3))]
;

ffexpa1 ::= stermEa2 '.' PN
              [a1 ← create(encode('...'),
                           create(a2, encode(PN)))]
|
stermEa2 '.' ffexpa3
              [a1 ← create(encode('...'),
                           create(a2, a3))]
|
stermEa2 '..' PN

```



```

sfacta1 ::= sprima2
              [a1 ← a2]
| '+' stermEa2 stermEa3
              [a1 ← create(encode('+'),
                           create(a2,a3))]
;

sprima1 ::= 'N'
              [a1 ← encode('N')]
| SN
              [a1 ← encode(SN)]
| NUM
              [a1 ← encode(NUM)]
| VAR
              [a1 ← encode(VAR)]
| '''' lexunita2 ''''
              [a1 ← create(encode('''),a2)]
| ''' lexunita2 '''
              [a1 ← create(encode(''),a2)]
| '@' namea2
              [a1 ← create(encode('@'),a2)]
| '#' SN
              [a1 ← create(encode('#'),encode(SN))]
| iotaexpa2
              [a1 ← a2]
| '(' fexpEa2 '=' sexpEa3 ';' sexpEa4 ')'
              [a1 ← create(encode('='),create(

```

```

                                a2,create(a3,a4)))]
|   '(' fexpEa2 '=>' sexpEa3 ')'
    [a1 ← create(encode('=>'),create(
                                a2,create(a3,encode('UNDEF'))))]
;

iotaexpa1 ::= '(' '1' forma2 ')' fprimEa3
              [a1 ← create(encode('1'),
                              create(a2,a3))]
;

fexpa1 ::= fterma2
              [a1 ← a2]
|   fexpEa2 '∨' ftermEa3
    [a1 ← create(encode('∨'),
                              create(a2,a3))]
;

fterma1 ::= ffacta2
              [a1 ← a2]
|   fterma2 '^' ffacta3
    [a1 ← create(encode('^'),
                              create(a2,a3))]
;

```

```

ffacta1 ::= ffact2a2
              [a1 ← a2]
|
  '~' ffact2a2
      [a1 ← create(encode('~'),a2)]
;

```

```

ffact2a1 ::= stermEa2 '.' pexpEa3
              [a1 ← create(encode('.'),
                          create(a2,a3))]
|
  stermEa2 '=' stermEa3
      [a1 ← create(encode('='),
                  create(a2,a3))]
|
  stermEa2 '..' pexpEa3
      [a1 ← create(encode('..'),
                  create(a2,a3))]
|
  fprima2
      [a1 ← a2]
;

```

```

fprima1 ::= '(' fexpEa2 ')'
              [a1 ← a2]
|
  fprim2a2
      [a1 ← a2]
;

```

```

fprim2a1 ::= 'T'
              | [a1 ← encode('T')]
              | 'F'
              | [a1 ← encode('F')]
              | FN
              | [a1 ← encode(FN)]
              | '#' FN
              | [a1 ← create(encode('#'),encode(FN))]
              | '/' dexpa2 '/'
              | [a1 ← create(encode('/'),a2)]
              | '\' fexpEa2 '\'
              | [a1 ← create(encode('\'),a2)]
              | allexpa2
              | [a1 ← a2]
              | existexpa2
              | [a1 ← a2]
              | '(' fexpEa2 '→' fexpEa3 ';' fexpEa4 ')'
              | [a1 ← create(encode('→'),create(
                a2,create(a3,a4)))]
              | '(' fexpEa2 '→' fexpEa3 ')'
              | [a1 ← create(encode('→'),create(
                a2,create(a3,encode('UNDEF'))))]
;

```

```

allexpa1 ::= '(' '∀' forma2 ')' fprimEa3
              [a1 ← create(encode('∀'),
                           create(a2,a3))]
              ;

existexpa1 ::= '(' '∃' forma2 ')' fprimEa3
              [a1 ← create(encode('∃'),
                           create(a2,a3))]
              ;

texpa1 ::= tterma2
              [a1 ← a2]
              |
              stermEa2 '.' texpEa3
              [a1 ← create(encode('.'),
                           create(a2,a3))]
              ;

tterma1 ::= TN
              [a1 ← encode(TN)]
              |
              '#' TN
              [a1 ← create(encode('#'),encode(TN))]
              |
              tprima2
              [a1 ← a2]
              ;

```

```

tprima1 ::= '(' 'λ' forma2 ')' sttermEa3
              [a1 ← create(encode('λ'),
                           create(a2,a3))]
;

```

```

pexpa1 ::= pterma2
              [a1 ← a2]
|
stermEa2 '.' pexpEa3
              [a1 ← create(encode('.'),
                           create(a2,a3))]
|
stermEa2 '...' pexpEa3
              [a1 ← create(encode('...'),
                           create(a2,a3))]
;

```

```

pterma1 ::= PN
              [a1 ← encode(PN)]
|
'#' PN
              [a1 ← create(encode('#'),encode(PN))]
|
pprima2
              [a1 ← a2]
;

```

```

pprima1 ::= '(' 'ρ' forma2 ')' fpprimEa3
              [a1 ← create(encode('ρ'),
                           create(a2,a3))]
;

```



```

namea1 ::= SN
           [a1 ← encode(SN)]
           |
           FN
           [a1 ← encode(FN)]
           |
           TN
           [a1 ← encode(TN)]
           |
           PN
           [a1 ← encode(PN)]
           ;

forma1 ::= form1a2 ':' pexpEa3
           [a1 ← create(encode(':'),
                        create(a2, a3))]
           |
           form1a2
           [a1 ← a2]
           ;

form1a1 ::= VAR
           [a1 ← encode(VAR)]
           |
           '(' forma2 ')'
           [a1 ← a2]
           |
           '+' form1a2 form1a3
           [a1 ← create(encode('+'),
                        create(a2, a3))]
           ;

```

$$\begin{aligned} \text{sexp}_{a_1}^E & ::= & \text{sexp}_{a_2} & [a_1 \leftarrow a_2] \\ & | & E_{a_2} & [a_1 \leftarrow a_2] \\ & ; \end{aligned}$$

$$\begin{aligned} \text{fexp}_{a_1}^E & ::= & \text{fexp}_{a_2} & [a_1 \leftarrow a_2] \\ & | & E_{a_2} & [a_1 \leftarrow a_2] \\ & ; \end{aligned}$$

$$\begin{aligned} \text{texp}_{a_1}^E & ::= & \text{texp}_{a_2} & [a_1 \leftarrow a_2] \\ & | & E_{a_2} & [a_1 \leftarrow a_2] \\ & ; \end{aligned}$$

$$\begin{aligned} \text{pexp}_{a_1}^E & ::= & \text{pexp}_{a_1} & [a_1 \leftarrow a_2] \\ & | & E_{a_2} & [a_1 \leftarrow a_2] \\ & ; \end{aligned}$$

```

stermEa1 ::= sterma2
              [a1 ← a2]
              |
              Ea2
              [a1 ← a2]
              ;

ftermEa1 ::= fterma2
              [a1 ← a2]
              |
              Ea2
              [a1 ← a2]
              ;

Ea1 ::= '[' sexpa2 ']'
        [a1 ← create(encode('['),a2)]
        ;

fprimEa1 ::= fprima2
              [a1 ← a2]
              |
              Ea2
              [a1 ← a2]
              ;

lexunita1 ::= sexpa2
              [a1 ← a2]
              |
              fexpa2
              [a1 ← a2]
              |
              pterma2
              [a1 ← a2]

```

```

| tterma2
|   [a1 ← a2]
|
| Ea2
|   [a1 ← a2]
|
| tokena2
|   [a1 ← a2]
;
tokena1 ::= 'ι'
           [a1 ← encode('ι')]
           | '∀'
           [a1 ← encode('∀')]
           | '∃'
           [a1 ← encode('∃')]
           | 'λ'
           [a1 ← encode('λ')]
           | 'ρ'
           [a1 ← encode('ρ')]
           | '@'
           [a1 ← encode('@')]
           | '#'
           [a1 ← encode('#')]
           | '→'
           [a1 ← encode('→')]
           | ';'
           [a1 ← encode(';')]

```

```

|      '∧'
      [a1 ← encode('∧')]
|      '∨'
      [a1 ← encode('∨')]
|      '.'
      [a1 ← encode('.')]
|      '...'
      [a1 ← encode('...')]
|      '='
      [a1 ← encode('=')]
|      ':'
      [a1 ← encode(':')]
|      '=>'
      [a1 ← encode('=>')]
;

sttermEa1 ::= sterma2 [a1 ← a2]
| tprima2 [a1 ← a2]
| Ea2 [a1 ← a2]
;

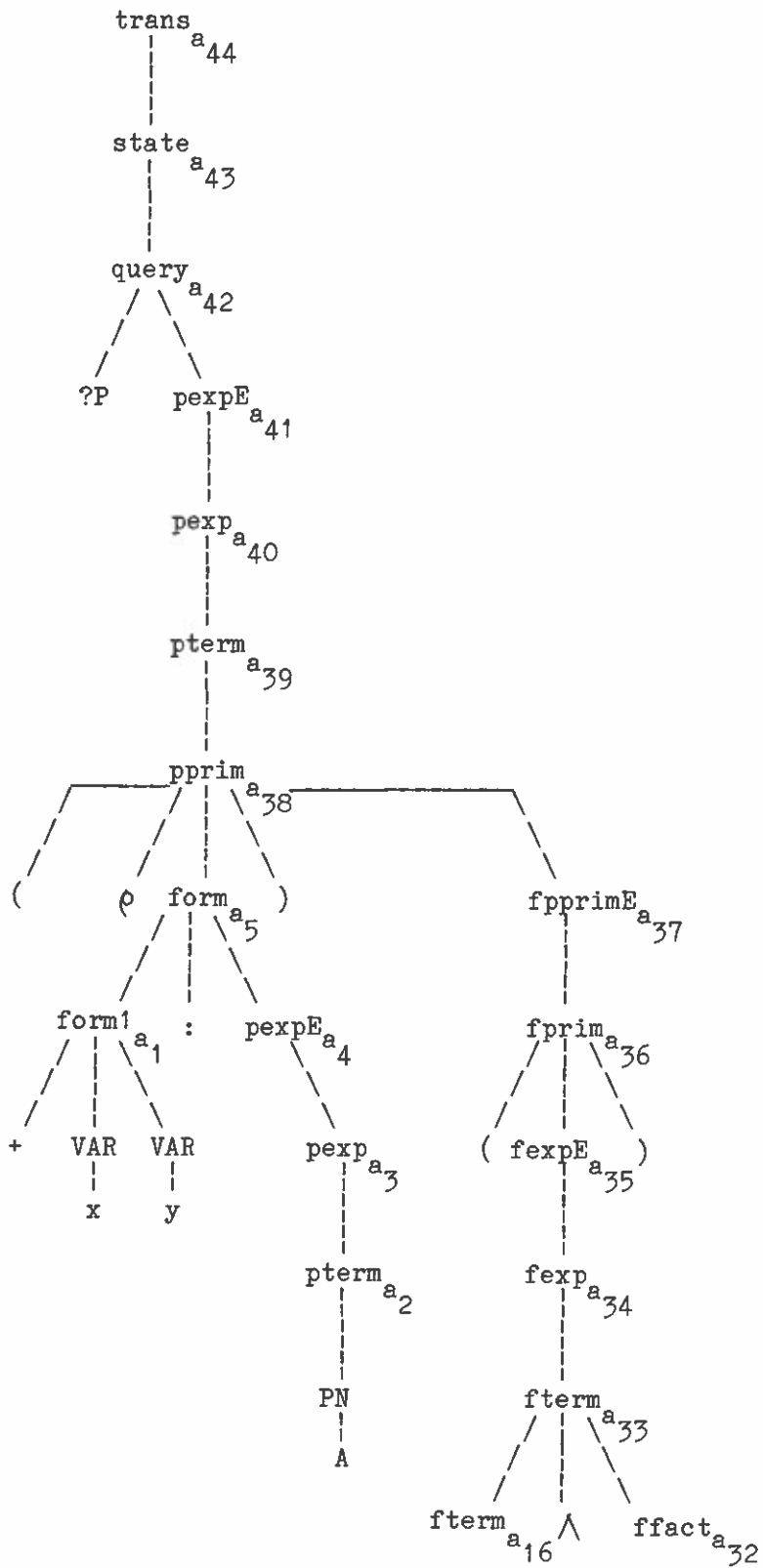
```

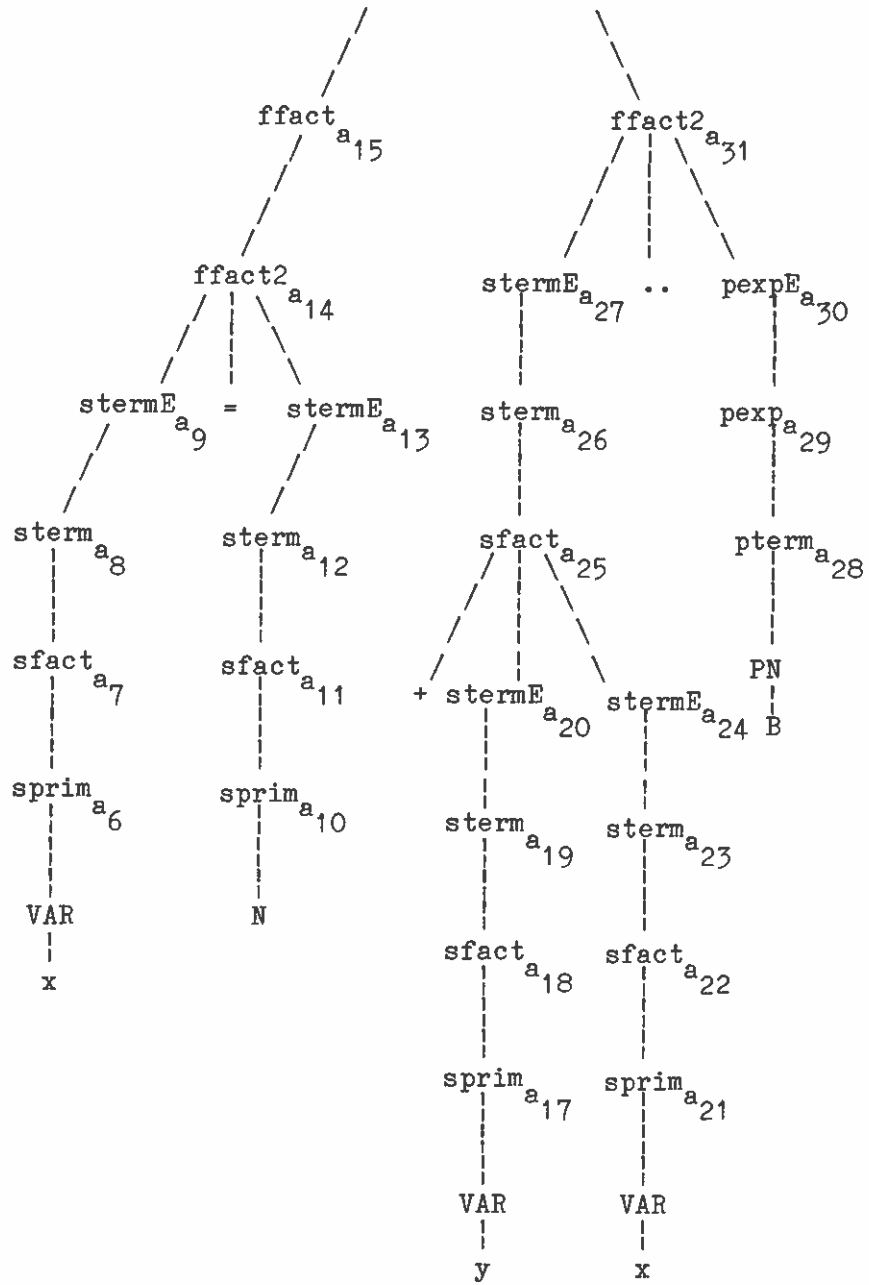
$$\begin{array}{lcl}
 \text{fpprimE}_{a_1} & ::= & \text{fprim}_{a_2} \\
 & & [a_1 \leftarrow a_2] \\
 & & | \\
 & & \text{pprim}_{a_2} \\
 & & [a_1 \leftarrow a_2] \\
 & & | \\
 & & \text{E}_{a_2} \\
 & & [a_1 \leftarrow a_2] \\
 & & ;
 \end{array}$$

For example, consider the following input:

$$?P (\rho + x y : A) (x = N \wedge + x y .. B)$$

Its corresponding parse tree and attributes are given below. Again, because of the complex structure of the attributes, the values of the attributes are not placed directly in the parse tree. Both the linearized form and the tree form of the attributes are presented.

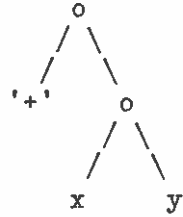
Parse Tree





Attributes

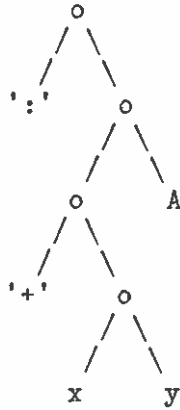
$$a_1 = + x y$$



$$a_2 = A$$

$$a_4 = a_3 = a_2$$

$$a_5 = : (+ x y) A$$



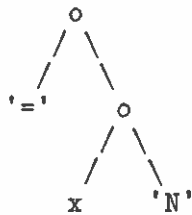
$$a_6 = x$$

$$a_9 = a_8 = a_7 = a_6$$

$$a_{10} = 'N'$$

$$a_{13} = a_{12} = a_{11} = a_{10}$$

$$a_{14} = = x N$$



$$a_{16} = a_{15} = a_{14}$$

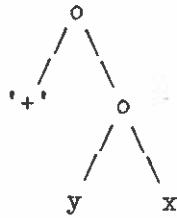
$$a_{17} = y$$

$$a_{20} = a_{19} = a_{18} = a_{17}$$

$$a_{21} = x$$

$$a_{24} = a_{23} = a_{22} = a_{21}$$

$$a_{25} = + y x$$

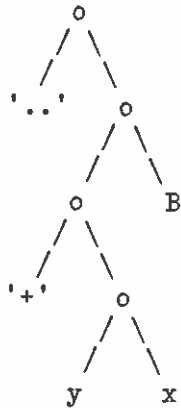


$$a_{27} = a_{26} = a_{25}$$

$$a_{28} = B$$

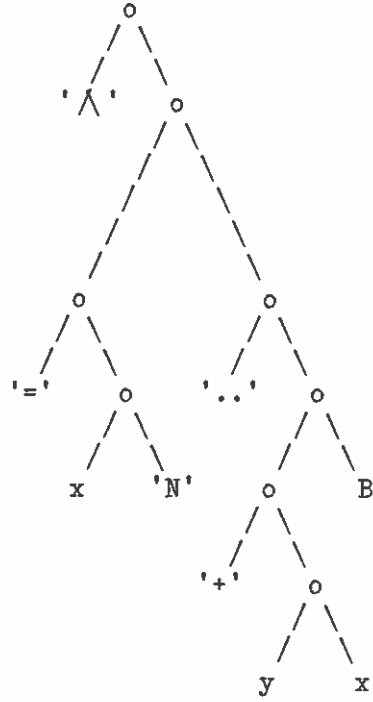
$$a_{30} = a_{29} = a_{28}$$

$$a_{31} = \dots (+ y x) B$$



$$a_{32} = a_{31}$$

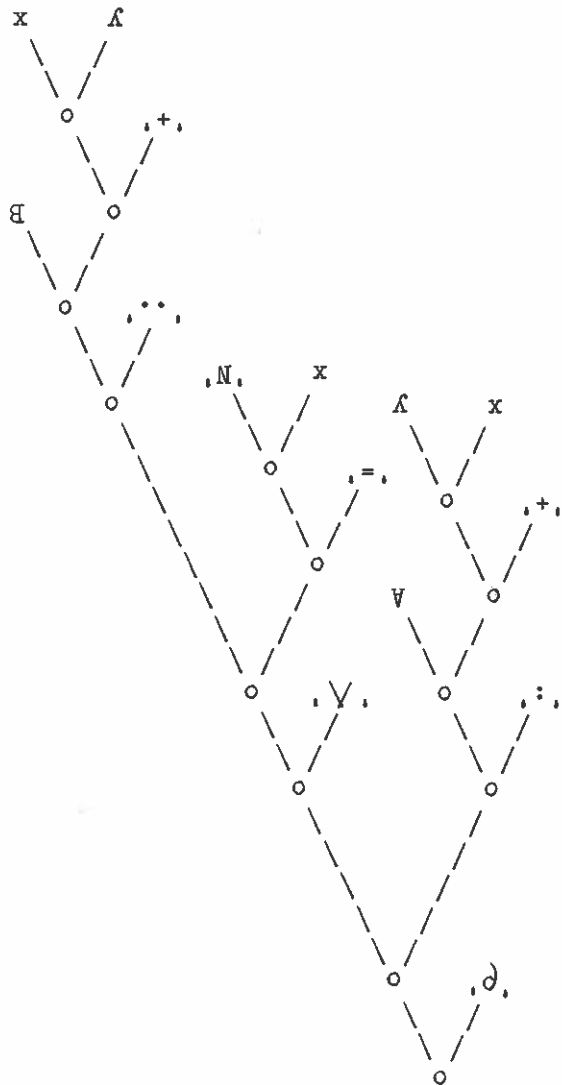
$$a_{33} = \wedge (= x N) \dots (+ y x) B$$

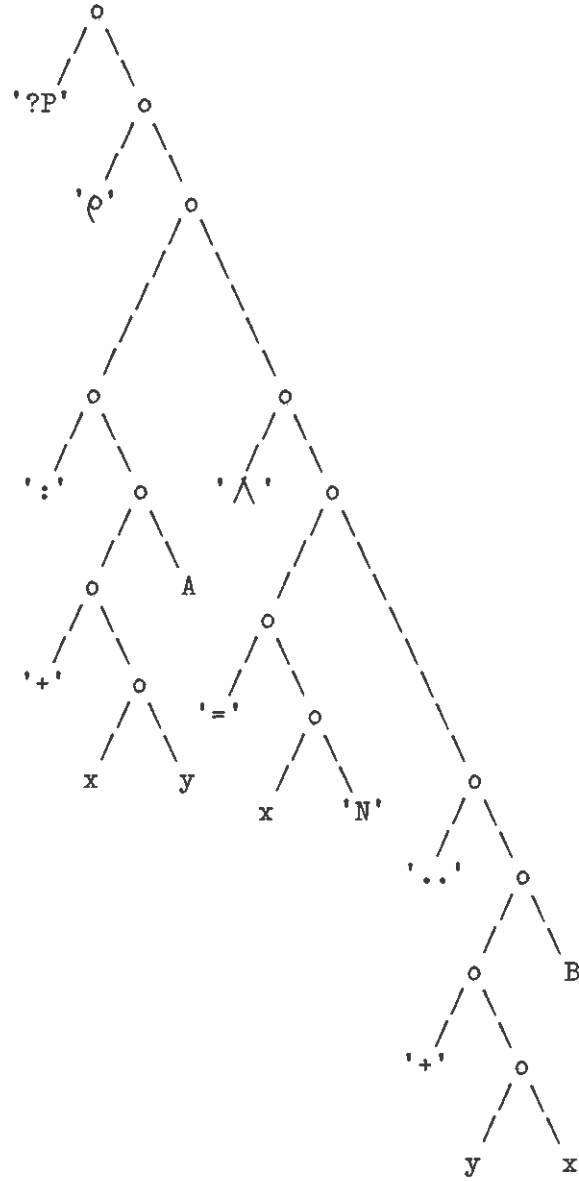


$$a_{37} = a_{36} = a_{35} = a_{34} = a_{33}$$

$$a_{38} = \rho ( : ( + x y ) A ) \wedge ( = x N ) \dots ( + y x ) B$$

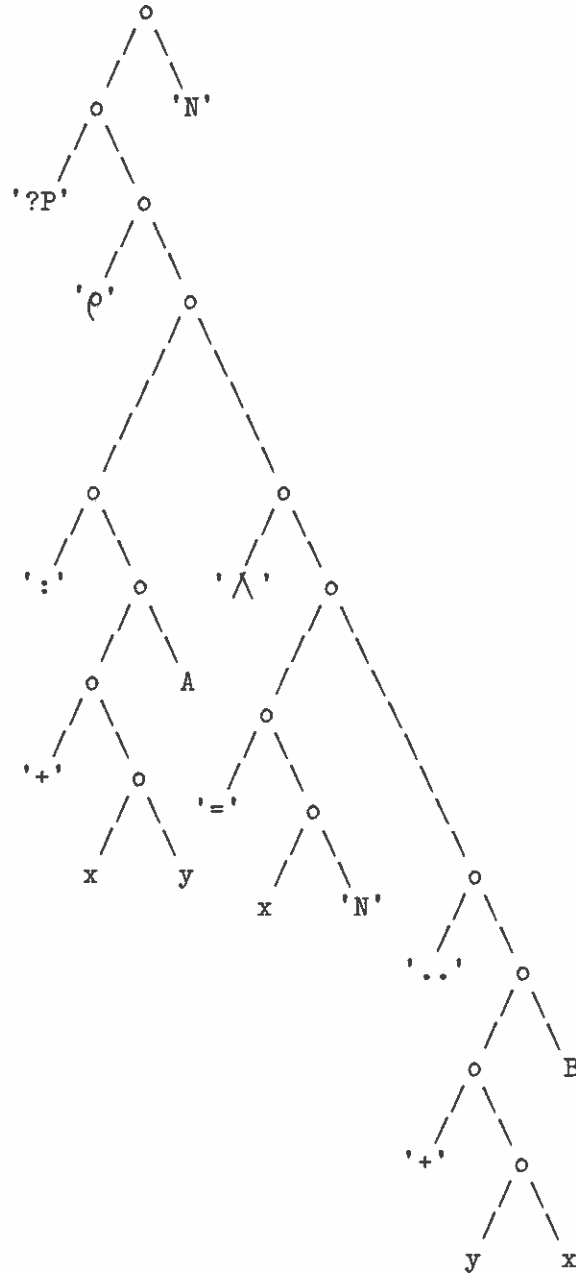
$a_{41} = a_{40} = a_{39} = a_{38}$   
 $a_{42} = ? P \rho ( : (+ x y) A) \vee (= x N) \dots (+ y x) B$





$$a_{43} = a_{42}$$

$$a_{44} = (?P \rho (: (+ x y) A) \wedge (= x N) .. (+ y x) B) N$$



The attribute of trans,  $a_{44}$ , is the desired ADS command tree.

Notice that there is a generic style to the encodement scheme for the ADS command trees; whenever an operator is involved, it is always brought to a superior position and made the most prevalent left subtree.

In the ADS syntax, there are unary, binary and ternary operators; some are prefix, and some are infix. In the command tree form, they are all transformed to a prefix form.

## 5. Summary

The purpose of this report has been to define the ADS syntax and the ADS command trees, and to formally specify the relationship between them. In order to specify the translation, the formalism of attributed grammars has been defined and used. A linearized form of the ADS command trees has also been defined.

The conceptual structure of ADS, the formal semantics of the ADS command trees, and the operational semantics of the ADS command trees appear elsewhere.