CGTM 38
Ingrid Carlbom
February, 1968

Algorithms for Transforming PDL-expressions into Standard Form and into a Primitive Connection Matrix

A. Transformation into Standard Form

A PDL-expression is said to be in standard form (see also Ref. 1, page 2), if

1) only primitives are labelled,
2) the blanking operator operates only on primitives,
3) the tilde operator operates only on primitives or blanked primitives.

The program transforms one or more PDL-expressions describing a picture into standard form. It also gives the whole picture description, where the subpicture-names in the main expression are replaced by their expressions, in standard form.

The Algorithm

1. The main expression is read.
2. The main expression is parsed for well-formedness (see algorithm 1). The primitive names are stored in an array; the expression is stored in a stack in polish postfix form, with the primitive names replaced by pointers to the primitive array.
3. The main PDL-expression is transformed into standard form (see algorithm 2), and stored in an array.
4. The pointers are replaced by their primitive names and parentheses are re-inserted.
5. The transformed expression is written out.
6. The subpicture-expressions are read, parsed, stored in polish postfix form, and transformed into standard form as the main expression.
7. The pointers to the names of the subpicture-expression in the main picture are replaced by their subexpressions.
8. The pointers in the subexpressions are replaced by the primitive names, the parentheses are reinserted, and the transformed subpicture expressions are written out.
9. The total picture-expression is transformed into standard form.

10. A test is performed to determine if the expression is a valid PDL-expression (see Ref 1, page 2).
11. The pointers are replaced by their primitive names, the parentheses are reinserted and the total expression is written out.

The program is written in FORTRAN(H) for the IBM 360 model 75.

Input

The program reads PDL-expressions 80 characters long. The expression should begin with a name assigned to the picture (or subpicture) followed by an equality sign. Each primitivename and its labels may consist of up to four letters or digits. The label, enclosed by parentheses, should follow its primitive name.

Every expression of the form $(S_1 \Theta S_2)$ (where S_1, S_2 are primitives or expressions of the form $(S_1' \Theta S_2')$ and Θ is a concatenation operator) should be enclosed by parentheses. The tilde and blanking operator should precede the primitive or expression it is operating on, and the total expression should be enclosed by parentheses. The tilde operator is punched as &, the blanking operator as / and the x-operator as"•" . Blanks may be inserted anywhere.

Every expression should be followed by a dollar sign. If an expression is followed by a subpicture-expression the dollar sign should be followed by an arbitrary letter and if it is the last subpicture-expression of one expression the dollar sign is followed by another dollar sign. If another picture follows, the last sign on the card should be an arbitrary letter and the very last data card should end with, altogether, three dollar signs.

Output

The program writes the initial expressions, the main-and-subexpressions in standard form and the total expression in standard form.

Error messages

If, when parsed, the expression is found incorrect (e.g. parenthesis missing) "EXPRESSION INCORRECT", the primitive array, the labels and the two stacks are written on the lineprinter.

If a subpicturename is not found in the main picture "NO MATCHING CHARACTER FOUND TO (subpicturename)" is written on the printer.

If a data card does not begin with a picture name followed by an equality sign,

"EXPRESSION INCORRECT" and the two first characters are printed out.

If a data card is not terminated by the right number of dollar signs (see above) "EXPRESSION EXCEEDS ONE CARD" is written on the line printer.

If the PDL-expression is found not valid "NOT A VALID PDL-EXPRESSION" is written on the printer.

B. Transformation into a Primitive Connection Matrix

A connection matrix M describes the connectivity in a graph. If $M_{ij} \neq 0$ then at least one primitive is connected from node i (tail) to node j (head).

The program handles one or more PDL-expressions forming one picture. The program sets up the connection matrix, one array with the primitive names and one with the tail and head nodes. The nonzero elements of the connection matrix are pointers to the array with the primitives. If more than one primitive connects two vertices the nonzero element in the connection matrix points to the first element of a chain of primitive names.

The Algorithm.

1. The main expression is read.
2. The main expression is parsed (see algorithm 1). The primitive names and stored in an array and the expressions in a stack in polish postfix form, with the primitive names replaced by pointers to the primitive array.
3. The subpicture expressions are read.
4. The subpicture expressions are parsed and stored in the same manner as the main-expression.
5. The pointers to the subpicture names in the stack with the main picture are replaced by their expressions.
6. A vertex is assigned to every head and tail node, and the nodes are concatenated according to the rules in Ref. 2, page 4. (see also algorithm 3). The blanked primitives are treated as the other primitives.
7. The head and tail nodes of the blanked edges are contracted to the head and tail nodes of their corresponding labelled primitives. A test is performed to determine if the expression is a valid PDL-expression (see Ref. 1, page 2).
8. The blanked edges are eliminated.
9. The connection matrix is set up.
10. The primitive array, the tail and head vertices, the chain pointer and the connection matrix are written out.

The program is written in FORTRAN(H) for the IBM 360 model 75.

Input

The same as for algorithm A.

Output

The program points the array with the primitive names,one with their labels. arrays with head and tail vertices, the chain pointer and the connection matrix. The 132 characters on the printer limit the number of vertices in one picture to forty.

Error messages

The same as in algorithm A.

References

1. Alan C. Shaw, "A Picture Calculus -- Further Definitions and Some Basic Theorems", GSG Memo No. 46, Computation Group, Stanford Linear Accelerator Center, June 1967.

2. W.F. Miller and Alan C. Shaw,"A Picture Calculus", GSG Memo N . 40, Computation Group, Stanford Linear Accelerator Center, June 1967.

Algorithm 1: Picture Parsing.

S is an array whose first element is set to \$; i is its index, which is initiated to 1. Nextsymbol gets the next symbol from the array the expression was read into.

```
PARSING
x:= Nextsymbol
if x = $ ∧ S_{i-1} = $ then end PARSING
else
if  (x = primitive ∨ x = operator ∨ x= "(")
     then ( i:=i+1;  S_i:=x)
else
if (x = ")" ∧ S_i= label ∧ S_{i-1} = "(" ∧ S_{i-2} = primitive)
     then  i:=i-2;
else
if (x = ")" ∧ S_i = primitive ∧ S_{i-1} ∈ {~, Γ} ∧ S_{i-2} = "(")
     then (S_{i-2} := S_i, i:= i-2)
else
if (x = ")" ∧ S_i = primitive ∧ S_{i-1} ∈ {+, -, *, x} ∧ S_{i-2} = primitive
      ∧ S_{i-3} = "("
     then (S_{i-3} := S_i; i:=i-3)
else
     "EXPRESSION INCORRECT"
Call PARSING.
```

Algorithm 2: Transformation into Standard form.

The expression is a list S which is scanned from the left. $S = S_1 \cdot S_2 \cdot S_3$ where $S_2 = S_2' \cdot \Gamma$, S_1, S_2' are lists of primitive pointers, concatenation and tilde operators, S_3 is a list of primitive pointers, concatenation and unary operators.
The operator "·" concatenates two lists.
Θ is a concatenation operator.

```
S=S_1· (BLANKO(S_2)) · S_3
BLANKO (S_2) =
if S_2' = primitive pointer then S_2
else
```

<u>if</u> $S_2' = S_2'' \cdot \sim$ <u>then</u> $((\mathrm{BLANKO}(S_2'' \cdot \Gamma)) \cdot \sim)$
<u>else</u>
<u>if</u> $S_2' = S_2'' \Gamma$ <u>then</u> S_2'
<u>else</u>
<u>if</u> $S_2' = S_2'' \cdot S_2''' \cdot \theta$ <u>then</u> $((\mathrm{BLANKO}(S_2'' \cdot \Gamma)) \cdot \mathrm{BLANKO}(S_2''' \cdot \Gamma))\ \theta)$

Then the program finds the next leftmost list whose last element is a blanking operator and applies BLANKO on it. This goes on until all operands of Γ are primitive pointers.
Now the list has the form $S = S_1 \cdot S_2 \cdot S_3$ where $S_2 = S_2' \cdot \sim$, S_1, S_2' are lists of primitive pointers, blanked primitive pointers, which are treated like the unblanked ones, and concatenation operators. S_3 is a list of primitive pointers, concatenation and unary operators.

$S := S_1 \cdot (\mathrm{TILDOP}(S_2)) \cdot S_3$
TILDOP $(S_2) :=$
<u>if</u> S_2' = primitive pointer <u>then</u> S_2
<u>else</u>
<u>if</u> $S_2' = S_2'' \cdot \sim$ <u>then</u> S_2''
<u>else</u>
<u>if</u> $S_2' = S_2'' \cdot S_2''' \cdot \theta$ <u>then</u> $((\mathrm{TILDOP}(S_2''' \cdot \sim) \cdot \mathrm{TILDOP}(S_2'' \cdot \sim) \cdot \mathrm{OPER}(\theta))$
$\mathrm{OPER}(\theta) :=$
<u>if</u> $\theta \in \{+, *\}$ <u>then</u> θ
<u>else</u>
<u>if</u> $\theta = -$ <u>then</u> x
<u>else</u>
<u>if</u> θ = x <u>then</u> -

Then the program finds the next leftmost list whose last element is a tilde operator and applies TILDOP on it. This goes on until all operands of ~ are primitive pointers or blanked primitive pointers.

<u>Algorithm 3</u>: Concatenation of the Primitives.

S is a stack that is collapsed as the primitives are connected.
Nextsymbol gets the next symbol from the stack where the expression is stored in polish postfix form.

Con 3 is an array where blanking operators are stored.
Primitive (S_i) gives pointers to all primitives that S_i replaces.

```
CONCATENATE
x := Nextsymbol
if x = "$" then end CONCATENATE
else
if x = primitive pointer
     then (i:= i+1, S_i := x)
else
if x = "+"
     then (tail (S_i) := head(S_i-1), tail(y) := tail(S_i-1);
             head(y) := head(S_i);  i := i-1;  S_i := y)
else
if x = "-"
     then (head(S_i-1) := head(S_i);  tail(y) = tail(S_i-1);
             head(y) := head(S_i); i:=i-1; S_i := y)
else
if x = "*"
     then(tail(S_i-1) := tail(S_i); head(S_i-1) := head(S_i); tail(y) := tail(S_i);
            head(y) := head(S_i); i := i-1; S_i := y)
else
if x = "x"
     then (tail(S_i-1) := tail(S_i); tail(y) := tail(S_i)
            head(y) := head(S_i); i := i-1; S_i := y)
else
if x = "~"
     then (y := tail(S_i); tail(S_i) := head(S_i); head(S_i) := y)
else
if x = " ┌ "
     then (Con3  (primitive(S_i)) := ┌ )
call CONCATENATE.
```

<u>Example 1:</u> Transformation into Standard Form. Given the PDL-expressions:

MAIN=(&((((&A)+(&B))*((&B(ALFA)+(&A(BETA))))+(/SUB)))$F

SUB=(&(B(ALFA)+A(BETA)))$$$

MAIN is parsed and stored in polish postfix form

S= /&2& + 3&4& +*5/+&

The numbers are pointers

to the con-matrix

CON=

1	A	
2	B	
3	B	ALFA
4	A	BETA
5	SUB	

$S=S_1 \cdot S_2 \cdot S_3$

BLANKO(S_2) = BLANKO(5/) = 5/

TILDOP is applied on S

TILDOP (S) = TILDOP(1&2&+3&4&+*5/+&) =

= TILDOP (5/&) · TILDOP(1&2&+3&4&+*&)·+=

=5/& · TILDOP(3&4&+&) TILDOP(1&2&+&)*+=

=5/& 43+21+*+

The pointers are replaced by their primitive names and labels, the parentheses reinserted. The standard form: MAIN=((&(/SUB)) + ((A/BETA)+B(ALFA))*(B+A)))

SUB is parsed and stored in polish postfix form

S'=67+&

CON =

6	B	ALFA
7	A	BETA

TILDOP(S') = TILDOP(67+&) = 7&6&+

In standard form:

SUM =(((&A(BETA))+(&B(ALFA)))

The pointer to SUB in S is replaced by S'

S = 5&6&+/&43+21+*+

$S = S_1 \cdot S_2$

BLANKO (S_1) = BLANKO (5&6&+/)

= 5/&6/&+

S = 5/&6/&+& 43+21+*+

$S = S_1 \cdot S_2$

TILDOP(S_1) = TILDOP (5/&6/&+&) = 6/5/+

S = 6/5/ + 43+21+*+

CON=

1	A	
2	B	
3	B	ALFA
4	A	BETA
5	A	BETA
6	B	ALFA

The pointers are replaced by their primitive names and labels, the parentheses reinserted. In standard form:

MAIN=((((/B(ALFA))+(/A(BETA)))+((A(BETA)+B(ALFA))*(B+A)))

<u>Example 2:</u> Transformation into Connection-matrix.

Given the PDL-expressions:

MAIN=(&((((&A)+(&B))*((&B(ALFA)+(&A(BETA))))+(/SUB)))$F

SUB=(&(B(ALFA)+A(BETA)))$$$

MAIN and SUB are parsed and stored in polish postfix form with SUB replaced by its expression.

S = 1&2&+3&4&+*56+&+

A vertex is assigned to every head-and-tail node, the column vertices in CØN stores pointers to these vertices in vector NODE.

CON=

	prim. names	labels	vertices		blank
1	A		~~1~~ 2	~~2~~ 1	
2	B		~~3~~ 4	~~4~~ 3	
3	B	ALFA	~~5~~ 6	~~6~~ 5	
4	A	BETA	~~7~~ 8	~~8~~ 7	
5	B	ALFA	~~9~~ 12	10	/
6	A	BETA	11	~~12~~ 9	/

Scan from left to right.

The first & causes an interchange between tail-and-head vertices of primitive one. The first + ⟹

Node (tail(2))= Node (head(1))

The second + ⟹

Node (tail(4))=Node(head(3))

The * ⟹

Node(tail(3))=Node(tail(1))

Node (head(4))=Node(head(2))

the next + ⟹

Node (tail(6))=Node(head(5))

The next ~ ⟹

Pointer to tail (5) is interchange with pointer to head(6).

The last + ⟹

Node (tail(12))=Node (head(5))

NODE

1	1			
2	2			
3	3			
4	~~4~~	1		
5	~~5~~	4		
6	~~6~~	2		
7	~~7~~	3		
8	~~8~~	~~5~~	4	
9	~~9~~	3		
10	~~10~~	5		
11	~~11~~	~~10~~	~~5~~	4
12	~~12~~	2		

The head-and-tail nodes to the blanked vectors are contracted to their labelled primitives.

Node (tail(5)) = Node (tail(3))

Node (head(5)) = Node (head (3))

Node (tail(6))= Node (tail(4))

Node (head(6))= Node (head (4))

Finally renumber NODE and set up concatenation matrix.

CONCAT =

	1		
2			4
	3		