Some transformations on trees and codes which preserve equivalences of the type $((A\last B)\circ C)\sim((A\last C)\circ B)$

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics

Memorandum 1976-10

Issued November, 1976

SOME TRANSFORMATIONS ON TREES AND CODES WHICH PRESERVE EQUIVALENCES OF THE TYPE $((A*B)\circ C) \sim ((A*C)\circ B)$

by

R.P.Nederpelt

University of Technology
Department of Mathematics
PO Box 513, Eindhoven
The Netherlands

1. INTRODUCTION.

1.1 Abstract.

In this paper we shall proceed from product formulae like ((a-b)*(c-(dxe))), based on binary operations. In particular we shall be interested in formulae of this type which are partitioned into classes by an equivalence relation of the form ((AσB)τC) ∿ ((AσC)τB), A,B and C being formulae and σ, τ being symbols for binary operations. In 2.6 we give some well-known examples of such classes of formulae.

Our aim will be to describe a manner of representing each of these classes by a (labelled) oriented tree. Since an oriented tree is an equivalence class of ordered trees, it suffices to give a bijection of product formulae onto ordered trees in such a manner, that the equivalence is preserved. This means that equivalent product formulae are mapped onto equivalent ordered trees, and vice versa. Precise definitions of these notions will be given in section 2.

There are well-known 'codings' of ordered trees, being bijective mappings of ordered trees to strings of symbols (e.g. parentheses). We describe some of these codings in 2.3.

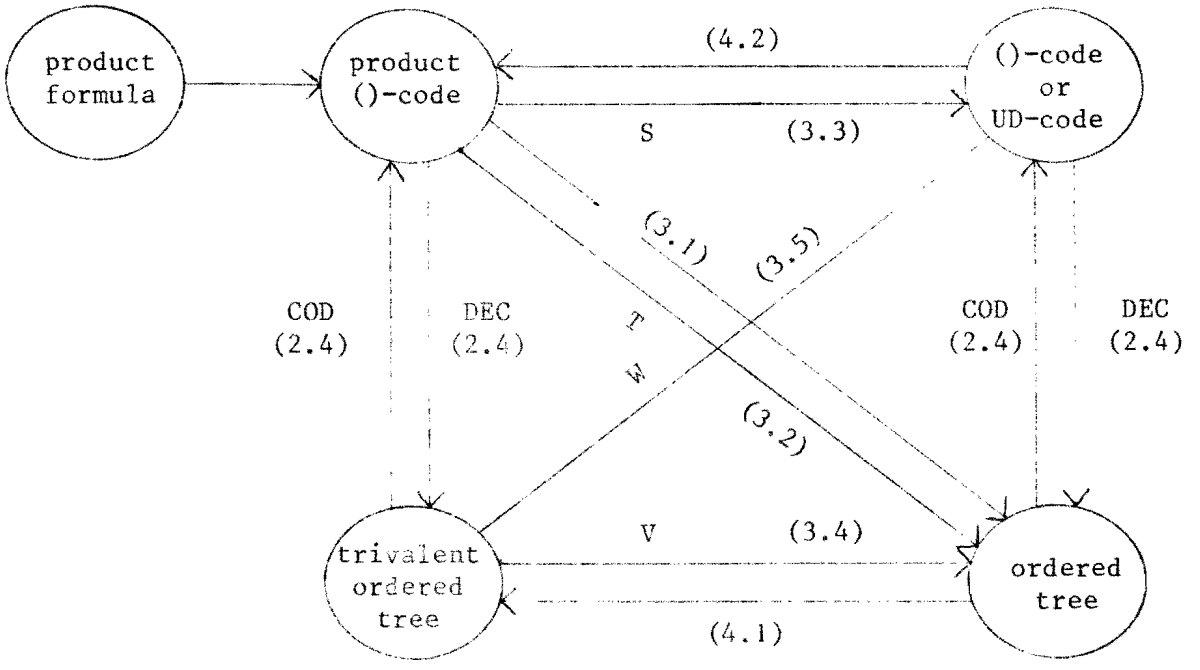
A product formula can be represented by a (labelled) trivalent ordered tree. The coding of such a tree yields a symbol string of a special form; we shall call it a product code.

In section 3 we describe a number of mappings which preserve the equivalence in the sense as explained above. As a domain for these mappings we take trivalent ordered trees or product codes rather than product formulae. As images we take ordered trees or codes of such trees.

In section 4 we describe some inverse mappings.

1.2 A picture of the transformations.

The transformations which we shall describe can be graphically represented in the following figure. The numbers refer to paragraphs.



2. DEFINITIONS.

2.1 Product formula.

Let X and S be alphabets, i.e. sets of symbols. We assume that neither X nor S contains the opening parenthesis or the closing parenthesis as a symbol. A product formula is a string of symbols, with the following recursive definition:

(i) if $x \in X$, then the string x is a product formula;

(ii) if A and B are product formulae and $\sigma \in S$, then the string $(A\ \sigma\ B)$ is a product formula.

Example:

let $X := \{a,b,c,\ldots,z\}$ and $S := \{+,\times,\uparrow,\sin\}$ then $((a\times c) + ((x+y) \uparrow b))$ and t are product formulae. Note that (a sin b) is a product formula, but (sin b) is not. We think of S as being a set of symbols for binary operations, and X as being a set of symbols representing variables.

2.2 Product ()-code.

Each product formula has a frame of parentheses, which one can obtain by deleting all symbols of alphabets X and S. So from $(u \times (v+w))$ one obtains (()). This string of parentheses is not very informative as to the product structure of the formula, since $((u \times v)+w)$ yields the same string. We retain much more information by embracing each letter of X occurring in the product formula by parentheses, before deleting all symbols of X and S. Then $(u \times (v+w))$ becomes successively $((u) \times ((v) + (w)))$ and ((()(()())).The formula $((u \times v)+w)$, however, yields (((())())()).

The product structure of the formula is fully described by the parentheses strings thus obtained. If one takes the precaution of storing the string $u \times v + w$ (or the lists $\langle u,v,w \rangle$ and $\langle \times,+ \rangle$) in a safe place, one can reconstruct the original product formula.

We call parentheses strings like those above: product ()-codes. A recursive definition of product ()-codes is:

(i) () is a product ()-code;

(ii) if A and B are product ()-codes, then so is (AB).

2.3 Some codes for ordered trees.

It is wellknown (see e.g. Read [4] p. 157) that ordered trees may be coded by binary strings, i.e. strings of symbols from a two-letter alphabet. One can define this coding recursively: if an ordered tree consists of a root which bears n subtrees coded $C_1,\ldots,C_n$ (in order), then the tree itself is coded $0,C_1,\ldots,C_n,1$, the comma's standing for concatenation. A sole root is coded by 01. Here we use the two-letter alphabet $\{0,1\}$.

Example: see figure 1a. The leftmost tree has two subtrees, as suggested in the figure.

We shall adopt the existing convention of using the left parenthesis instead of 0 and the right parenthesis instead of 1. Thus 00010100111011 is written as (((()(()))(())). We call this code the ()-code.

De Bruijn and Morselt (see [1]) use the alphabet {U,D} instead of {0,1}. We call their code the UD-code. It differs only slightly from the codes above. The letters U and D, standing for 'up' and 'down', refer to a simple way of obtaining the code of an ordered tree by following a path around the tree. One writes a U if an edge is followed in the 'up' direction, a D if it is followed in the 'down' direction. Example: see figure 1b. Apart from the obvious renamings, one obtains the same code as the ()-code by putting an extra U in front and an extra D behind.

We note that the ()-code can be obtained directly from the tree by enclosing each node in parentheses and following the "De Bruijn-Morselt-path". See figure 1c.

The codes described have two nice properties. The first, which we shall call the balance condition, states that the number of left parentheses (or U's) in a code is equal to the number of right parentheses (or D's). The codes also obey the so-called level condition: the number of opening parentheses preceding a certain entry in a ()-code is greater than the number

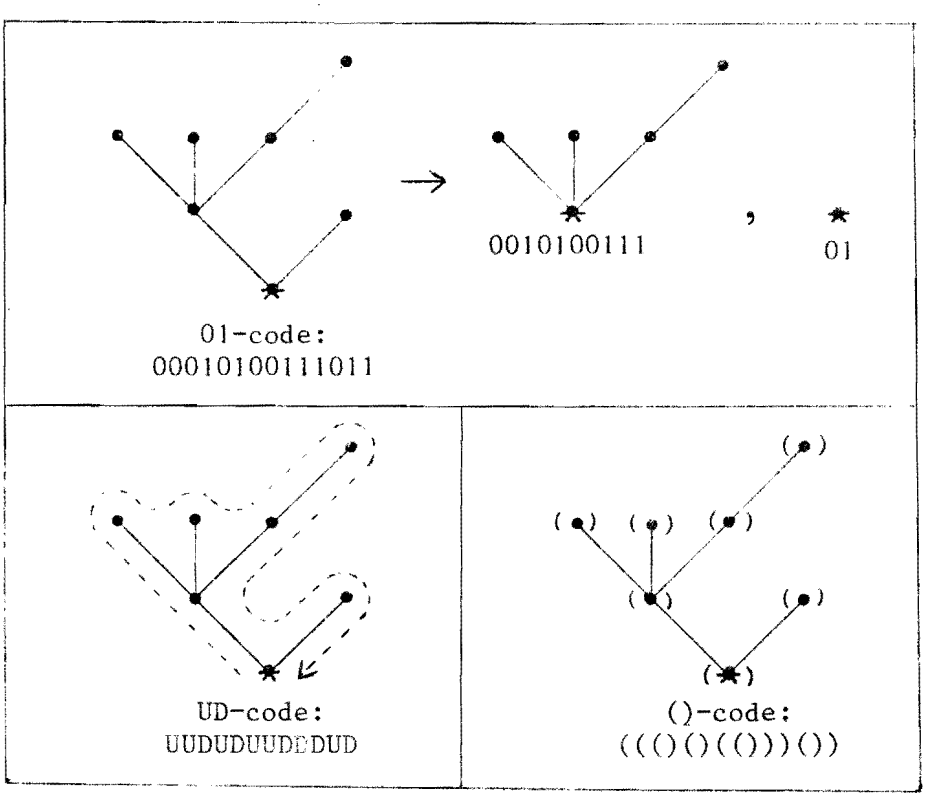


figure 1a

figure 1b, figure 1c

of closing parentheses preceding this entry. (For a UD-code: the number of U's preceding an entry is greater than or equal to the number of D's preceding this entry).

It holds that a string on the basis of the two-letter alphabet {(,)} and obeying both the balance condition and the level condition, is a ()-code of some ordered tree. An analogous statement holds for the UD-code. One may regard a ()-code as a linear nesting, a nest being a connected part of the string which has 'matching' parentheses as begin and end. For example, the nest structure of (((()(()))(())) is expressed by the following underlinings:

In this example the (unique) outer nest has two so-called subnests. The first of these subnests in its turn has three subnests, the second has none.

We call a ()-code, each nest of which has two subnests or none, a doubly-nested ()-code.

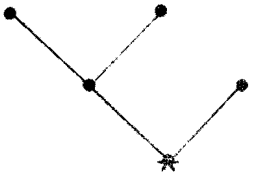
We note that, in particular, all product ()-codes are ()-codes. In paragraph 2.5 we shall state conditions, necessary and sufficient for ()-codes to be product ()-codes.

Product ()-codes are doubly nested. This follows by induction on the length of construction.

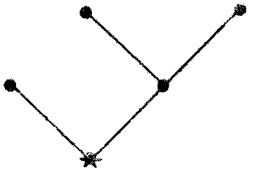
The ()-code of a trivalent ordered tree is a product ()-code. (A trivalent or bifurcant tree is a tree in which each node but for the root has valency 1 or 3; the root has valency 0 or 2).

2.4 Decoding.

The process of decoding, i.e. the construction of an ordered tree from a code which is a linear nesting, is easy. Since, in particular, product ()-codes are linear nestings, one can decode such a code as an ordered tree. Such a tree is trivalent. For example, the product ()-codes ((()(())) and (((())(()) yield the trivalent ordered trees:



and



The coding and decoding procedures being unique, we have the bijections:
ordered tree $\leftrightarrow$ ()-code or UD-code,
and in particular:
trivalent ordered tree $\leftrightarrow$ product ()-code.
We call the mappings involved the natural coding or decoding respectively, abbreviated as COD and DEC.

2.5 When is a ()-code a product ()-code?

A trivial answer to this question is: a ()-code is a product ()-code if and only if it is doubly nested.

In some cases it may be more appropriate to regard a ()-code not as a nesting, but merely as a string of symbols. For such cases we shall develop another criterion, summarized in Theorem 2.

Let A be a ()-code. Partition code A in parts of the form $(^{k})^{\ell}$ with $k \geq 1$ and $\ell \geq 1$. Here $(^{k}$ stands for a sequence of k succeeding opening parentheses. This partitioning can be done in only one way. Now it may occur that for each of these parts $(^{k})^{\ell}$ at least one of the exponents k or ℓ, is equal to one. We then say that A has the pruned tree property. (The name will be clear if one imagines what the tree corresponding to an A with this property looks like).

Theorem 1: Each product ()-code has the pruned tree property.
Proof: Induction on the length of construction of the product ()-code. □

We shall define two auxiliary notions: partition code and priority code.
(1) Let A be a ()-code such that $A \equiv (^{k_1})^{\ell_1} \ldots (^{k_m})^{\ell_m}$ with $k_i \geq 1$ and $\ell_i \geq 1$. The partition code of A is the string $\langle a_1, a_2 \ldots, a_{m-1} \rangle$, where $a_1 = k_1 - \ell_1$ and $a_i = a_{i-1} + k_i - \ell_i$ $(i=2,\ldots,m-1)$. Note: all a_i are natural numbers.

(2) Let A be a string of natural numbers. Then $\bar{A}$ is obtained from A by adding 1 to each number of A. Example: if $A \equiv \langle 2,1,3 \rangle$, then $\bar{A} \equiv \langle 3,2,4 \rangle$. (If A is the empty string, then of course $\bar{A} \equiv A$).
We define priority codes inductively by:

(i) The empty string is a priority code.

(ii) If A and B are priority codes, then $\langle \bar{A} \mid \bar{B} \rangle$ is a priority code.

Theorem 2: Let A be a ()-code. Then A is a product ()-code if and only if (i) the partition code of A is a priority code and (ii) A has the pruned tree property.
Proof: The if-part is proved by induction on the length of the partition code of A, the only-if-part by (i) induction on the length of construction of A and (ii) Theorem 1. □

Note: If A^* is a product formula with corresponding product code A, then the partition code of A (which is a priority code) gives a precise description of the relative priorities of the operations in A^*.

2.6 Some equivalences.

One may be interested in certain equivalences defined in the set of all product formulae based on certain alphabets X and S. For example, one may agree that each part (A+B) may be replaced by an equivalent part (B+A), A and B being product formulae and + $\in$ S. A class for the corresponding equivalence relation contains all product formulae which are 'equal but for transposition with respect to addition'.

We shall concern ourselves with a slightly more complicated equivalence by considering $((A\sigma B)\tau C)$ and $((A\sigma C)\tau B)$ to be equivalent, A, B and C being product formulae and $\sigma,\tau \in S$. More precisely: the equivalence relation $\sim$ is the equivalence relation generated by:

(i) if A, B and C are product formulae and $\sigma,\tau \in S$, then $((A\sigma B)\tau C) \sim ((A\sigma C)\tau B)$

(ii) (monotony:) if A, B, C and D are product formulae such that $A \sim C$ and $B \sim D$ and if $\sigma \in S$, then $(A\sigma B) \sim (C\sigma D)$.

We give two examples of this type of equivalence.

I. Consider algebraic formulae, the only (binary) operation allowed being exponentiation. Then the formula $(A^B)^C$ would yield the same outcome as the formula $(A^C)^B$, after calculation for specific values of the variables. Hence, one may define these formulae to be equivalent.
Formally: X := {a,b,...}, S := {↑} and for each triple of product formulae A, B and C it holds that: $((A\uparrow B)\uparrow C) \sim ((A\uparrow C)\uparrow B)$, the equivalence being monotonous.

II. Consider formulae from the implication calculus. Then the formulae $(C\rightarrow(B\rightarrow A))$ and $(B\rightarrow(C\rightarrow A))$ are logically equivalent. One may consider the corresponding equivalence relation.
Formally: X := {a,b,...}, S := {→} and for each triple of product formulae A, B and C it holds that $(C\rightarrow(B\rightarrow A)) \sim (B\rightarrow(C\rightarrow A))$, the equivalence being monotonous again. One notes the similarity between this equivalence relation and the previous ones, by reading the formulae from right to left.

The induced equivalence relation for product ()-codes obeys:

(i) if A, B and C are product ()-codes, then $((AB)C) \sim ((AC)B)$,

(ii) monotony.

For the corresponding trivalent trees one obtains:
if 

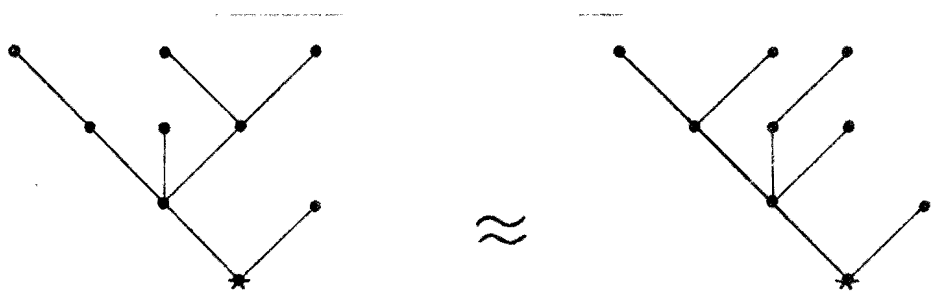
and 

represent trivalent ordered trees, then

For the so defined equivalence relations on product formulae, product ()-codes and trivalent ordered trees we shall use the same symbol $\sim$. We speak of $\sim$-equivalence.

We shall consider another equivalence between ordered trees, denoted $\approx$. We shall say that two ordered trees are $\approx$-equivalent, if they represent the same oriented tree, that is to say, if they only differ in the manner in which they are embedded in the plane.

Example:



We also call the corresponding ()-codes or UD-codes $\approx$-equivalent:
$((((())(()(()))))()) \approx ((((())(()))(()))(()))$.

2.7 Preservation of equivalence.

Let A and B be sets, let R_1 be an equivalence relation on A and R_2 on B. We shall call a bijective mapping $F : A \to B$ equivalence preserving if $\forall_{x\in A}\forall_{y\in A}[xR_1y \iff F(x)\ R_2\ F(y)]$. Hence, an equivalence preserving mapping F has the property that the image of a class of an element is the same as the class of the image of that element. Also: the mapping F induces a bijection from the set of classes of A to the set of classes of B.

In section 3 we take R_1 to be $\sim$ and R_2 to be $\approx$. For set A we take the

set of all product ()-codes or the set of all trivalent ordered trees, for B we take the set of all ()-codes, of all UD-codes or the set of all ordered trees. The mappings which we shall describe are equivalence preserving. This makes it possible to represent a class of algebraic formulae as described in 2.6, example I, by a single labelled oriented tree. An analogous statement holds for a class of implication formulae as described in 2.6, example II. The labelling of these trees occurs in an obvious manner, each node being labelled with an element of alphabet X.

3. EQUIVALENCE PRESERVING TRANSFORMATIONS FOR TRIVALENT ORDERED TREES AND PRODUCT ()-CODES.

3.1 Transformation via standard form.

A transformation from product formulae of a special type to oriented trees was discussed in [2]. In that paper X = {x}, x representing a real number greater than 1, and S = {↑}, ↑ standing for exponentiation. The formulae obtained are a special case of the algebraic formulae of 2.6, ex. I. In [2], the corresponding product formulae such as $(((x\uparrow(x\uparrow x))\uparrow x)\uparrow(x\uparrow x))$ were called bracketings.

For given x, the authors formed classes of bracketings, each class consistingof all bracketings with the same number n of exponentiation symbols and the same numerical outcome. They were interested in the number of classes for given x and n.

From the paper it follows that, for almost all x, such a class coincides with the $\sim$-equivalence class described in 2.6. The authors describe a manner of mapping bracketings onto oriented trees. The algorithm given intuitively amounts to taking logarithms, reordering and exponentiating again in order to obtain a so-called standard form. The conversion of a standard form into an ordered tree is the second step.

The latter transformation is equivalence preserving, so that the number of classes to be calculated is (for almost all x) the same as the number of oriented trees on n nodes.

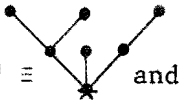
One can also easily adapt the transformation described in [2] in such a manner, that it maps product ()-codes onto ordered trees. The procedure is then, however, somewhat cumbersome. Therefore we now propose other transformations, to be described in the following. Of course, these transformations can also be applied to the case of [2].

We note that Guy and Selfridge reconsidered the problem of [2] and gave some further results. See [3].

3.2 A direct transformation from product ()-codes to ordered trees.

We use symbolic notations for ordered trees, which we shall explain by means of an example:

let a ≡  and b ≡  , then [a] ≡ a,

[a,b] ≡  and $\underline{b}$ ≡  .

The tree  will be denoted by [].

We shall describe a direct transformation from product ()-codes to ordered trees.

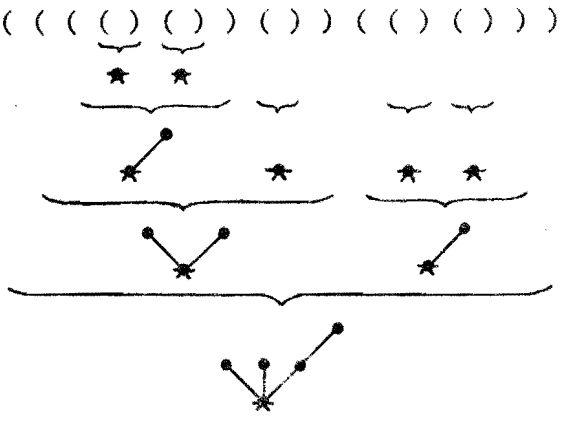
A transformation T from product ()-codes to ordered trees is recursively defined by:

(i) $() \xmapsto{T} []$,

(ii) if $A \xmapsto{T} a$ and $B \xmapsto{T} b$, then

$(AB) \xmapsto{T} [a,\underline{b}]$.

Example. Consider the product ()-code ((((())())())(()())), which one may obtain from a product formula like (((x+y)×z) + (u-v)). This product ()-code is transformed by T into an ordered tree as follows:



By induction on the size of the tree one can easily show that T is a bijection. The difference between T and the natural decoding DEC of a product ()-code into a trivalent ordered tree as described in 2.4, is:

$$(AB) \xmapsto{T} [a,\underline{b}], \text{ and } (AB) \xmapsto{DEC} [\underline{a},\underline{b}].$$

Transformation DEC is not equivalence preserving with respect to $\sim$ and $\approx$. We shall now show that T does preserve the equivalence.

Lemma 1. Let E and F be product ()-codes. If $E \sim F$, then $T(E) \approx T(F)$.

Proof: Induction on the length of proof of $E \sim F$. Let a,b,ab ... be the ordered trees corresponding to A,B,(AB) ... .

(1) Suppose $E \equiv ((AB)C)$ and $F \equiv ((AC)B)$. Then

$$E \xmapsto{T} [ab,\underline{c}] = [a,\underline{b},\underline{c}]$$

and

$$F \xmapsto{T} [ac,\underline{b}] = [a,\underline{c},\underline{b}]$$

and the right-hand sides are $\approx$-equivalent.

(2) Suppose $E \equiv (AB)$, $F \equiv (CD)$, $A \sim C$ and $B \sim C$. Then the ordered trees a and c are $\approx$-equivalent by induction hypothesis and the same holds for b and d. Now

$$E \xmapsto{T} [a,\underline{b}] \text{ and } F \xmapsto{T} [c,\underline{d}]$$

and the right-hand sides are $\approx$-equivalent again.

(3) The other cases are very easy: $E \sim F$ by virtue of

(i) $E = F$,

(ii) $F \sim E$ or

(iii) there is a G such that $E \sim G$ and $G \sim F$. □

We define recursively a mapping ℓ from product ()-codes to natural numbers:

(i) $() \xmapsto{\ell} 1$;

(ii) if $A \xmapsto{\ell} \alpha$ and $B \xmapsto{\ell} \beta$, then $(AB) \xmapsto{\ell} \alpha+\beta$.

Lemma 2: Let E and F be product ()-codes. If $T(E) \approx T(F)$, then $E \sim F$.

Proof: Induction on $\mu := \max\{\ell(E),\ell(F)\}$. Let $T(E) \approx T(F)$.

(1) $\mu = 1$. Then $E \equiv F \equiv ()$ and $E \sim F$.

(2) If $E \equiv ()$, then $T(E) = [] = T(F)$, so $F \equiv ()$. Hence $E \sim F$.

The same holds if $F \equiv ()$. So let $E \equiv (AB)$ and $F \equiv (CD)$. Then $E \equiv (AB) \overset{T}{\longmapsto} [a,\underline{b}]$ and $F \equiv (CD) \overset{T}{\longmapsto} [c,\underline{d}]$, where $A \overset{T}{\longmapsto} a$, etc. Since $T(E) \approx T(F)$, one of the two following statements holds:

(i) $a \approx c$ and $b \approx d$;

(ii) $[a,\underline{b}] \approx [c,\underline{d}] \approx [g,\underline{b},\underline{d}]$ for some ordered tree g.

In the first case, by induction hypothesis, $A \sim C$ and $B \sim D$, so also $E \equiv (AB) \sim (CD) \equiv F$.

In the second case $a \approx [g,\underline{d}]$ and $c \approx [g,\underline{b}]$. There is a product ()-code G such that $G \overset{T}{\longmapsto} g$. Moreover, $(GD) \overset{T}{\longmapsto} [g,\underline{d}] \approx a \overset{T}{\longleftarrow} A$, so by induction hypothesis: $(GD) \sim A$. Also $(GB) \sim C$. Then $E \equiv (AB) \sim ((GD)B) \sim ((GB)D) \sim (CD) \equiv F$. □

As a consequence of these two lemmas, T is equivalence preserving: $E \sim F \iff T(E) \approx T(F)$.

Note: The ℓ-map of all product ()-codes in an $\sim$-equivalence class is the same; this number is equal to the number of nodes (the root inclusive) in each of the T-maps. These observations can be used as a second proof of Theorem 2.1 of [2].

3.3 A transformation from product ()-codes to UD-codes.

There is a very simple recipe for transforming a product ()-code into the UD-code of its T-map:

(i) Substitute U for each opening parenthesis and D for each closing parenthesis in the product ()-code.

(ii) Partition the UD-code obtained into parts of the form U^kD^ℓ, with $k \geq 1$ and $\ell \geq 1$.

(iii) Replace each part U^kD^ℓ thus obtained by $UD^{\ell-1}$. (Here D^0 stands for the empty string).

(iv) Omit the first U.

As an example we again take the product ()-code ((((())())())((()))), which we considered in 3.2. We obtain successively:

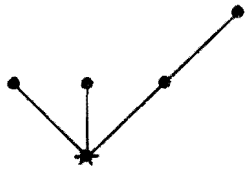
(i) U U U U D U D D U D D U U D U D D D,

(ii) $U^4D \mid UD^2 \mid UD^2 \mid U^2D \mid UD^3$,

(iii) U | UD | UD | U | UD^2 and

(iv) U D U D U U D D.

This is the UD-code for the tree:



This tree is, indeed, the T-map of the product ()-code.

Let S(E) stand for the result of applying the above recipe to product ()-code E. Let COD(X) be the UD-code of tree X which code we obtain by the natural coding as described in 2.3. We shall prove that S(E) is indeed the natural coding of T(E).

Theorem 3: Let E be a product ()-code. Then S(E) ≡ COD(T(E)).

Proof: Induction on $\ell(E)$.

(i) Let E ≡ (). Then S(E) is the empty string, being the UD-code of T(E) ≡ [] .

(ii) Let E ≡ (AB), where A ≡ $(^{k_1})^{\ell_1}(^{k_2})^{\ell_2}\ldots(^{k_m})^{\ell_m}$ and B ≡ $(^{v_1})^{w_1}(^{v_2})^{w_2}\ldots(^{v_n})^{w_n}$, k_i, ℓ_i, v_i and w_i being natural numbers.

Then $S(A) \equiv D^{\ell_1-1}UD^{\ell_2-1}\ldots UD^{\ell_m-1}$ and $S(B) \equiv D^{w_1-1}UD^{w_2-1}\ldots UD^{w_n-1}$.

Also: $S(E) \equiv D^{\ell_1-1}UD^{\ell_2-1}\ldots UD^{\ell_m-1}UD^{w_1-1}\ldots UD^{w_{n-1}-1}UD^{w_n}$, which is S(A) followed by U, followed by S(B), followed by D.

By induction hypothesis: S(A) ≡ COD(T(A)) and S(B) ≡ COD(T(B)). Let T(A) = a and T(B) ≡ b. Then COD(T(E)) = COD([a,b]), which is COD(T(A)) followed by U, followed by COD(T(B)), followed by D.

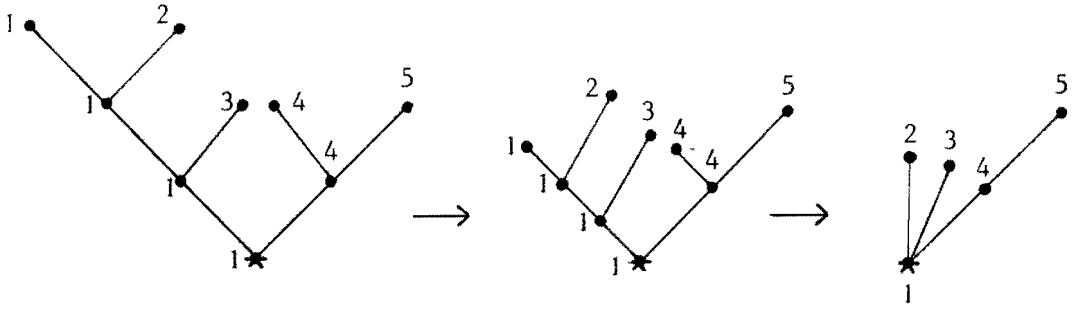
Hence COD(T(E)) = S(E). □

It is now trivial that S is an equivalence preserving bijection: $E \sim F \iff S(E) \approx S(F)$.

3.4 A transformation from trivalent ordered trees to ordered trees.

In 3.2 we described in passing the difference between the natural decoding DEC(A) of a product ()-code A, and its T-map T(A). One can ask what transformation V looks like which maps DEC(A) onto T(A). In general: we shall describe a simple transformation V from trivalent ordered trees such that V = T ∘ COD. Of course, V is an equivalence preserving bijection.

We give a pictorial representation of this correspondence, again taking as an example product ()-code A := ((((())())())(()())), which we considered in 3.2 and 3.3. We shall add numbers 1,2,...,5 for elucidating the process of this transformation.



One could call this process a "left branch contraction".

3.5 A transformation from trivalent ordered trees to UD-codes.

This transformation, which we call W, is also very easy. Given a trivalent ordered tree X, we follow the De Bruijn-Morselt path around the tree in the usual way. Now we write a U if we pass an edge upwards and to the right and a D if we pass an edge downwards and to the left. We write nothing in other cases. So we only take into account edges in the S.W.-N.E. direction, and not those in the S.E.-N.W. direction. The UD-code obtained is the UD-code of V(X).

So W = S ∘ COD = COD ∘ V and W is an equivalence preserving bijection.

The correspondence between this transformation W, and the transformation V described in 3.4, is obvious.

4. THE INVERSE TRANSFORMATION.

4.1 The inverse transformation from ordered trees to trivalent ordered trees.

It is not hard to invert the "left branch contraction" explained in 3.4, in order to devise the direct transformation $V^{\leftarrow}$ from ordered trees to trivalent ordered trees. We shall not describe this transformation. Instead, we shall give the corresponding transformation for codes in the next paragraph.

4.2 The inverse transformation from UD-code to product ()-code.

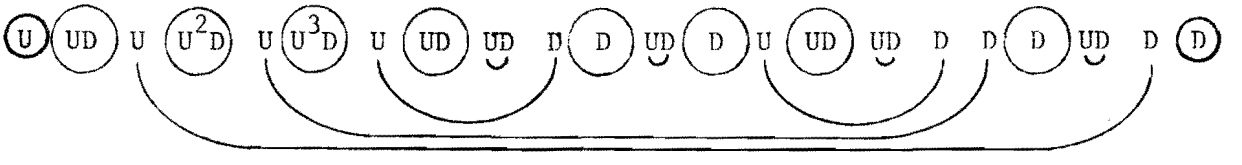
We shall describe the mapping $S^{\leftarrow}$, the inverse of S which was explained in 3.3.

Starting from a UD-code, one can form the UD-nests by connecting corresponding U's and D's.

Example:

Now we may obtain another string of U's and D's from a given UD-code by the following procedure. Add a symbol U in front and a symbol D behind. Insert a symbol D between each adjacent pair DU in the given UD-code and insert U^mD between each adjacent pair UU in the given UD-code, m being the number of subnests of the nest corresponding to the first U of the pair.

The UD-code of the example is thus transformed into

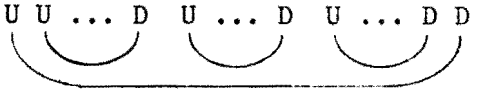


Lemma 3. The above procedure transforms UD-codes into UD-codes. The latter UD-code is, but for the obvious renamings, a product ()-code which is the $S^{\leftarrow}$-map of the original UD-code.

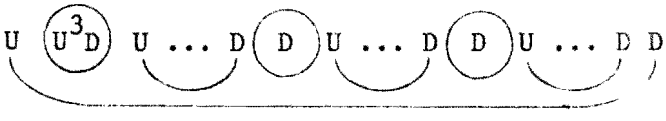
Proof. (1) Between each adjacent pair DU one finds a border of two adjacent subnests. So the procedure adds one D between each pair of adjacent subnests. Between each adjacent pair UU one may imagine the starting point

of a full sequence of consecutive subnests, belonging to the nest of the first U of the pair. Here the procedure adds another D, and as many U's as there are subnests. Thus, altogether, we add as many U's as D's, in such a manner, that the level condition is obeyed (we never "descend below the zero level"). So the string obtained is a UD-code, since level condition and balance condition (see 2.3) are obeyed.

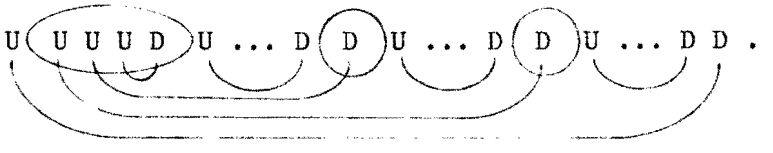
(2) We shall now show that the UD-code obtained is doubly nested. One may depict one nest plus its subnests, as occurring in the original UD-code, by:



(For convenience we took a nest with 3 instead of n subnests). After insertion of D between adjacent pairs DU and of $U^m D$ between the pair UU, we obtain:

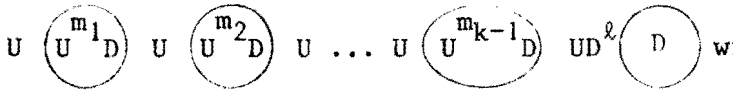


which we can write as the nested code:



Assuming that the three nests U ... D are doubly nested, it clearly follows that this holds for the entire code obtained. Hence, by induction, the new UD-code is doubly nested. It follows that one obtains a product ()-code after replacing U and D by parentheses.

(3) Let $U^k D^\ell$ be one of the parts ($k \geq 1$, $\ell \geq 1$) in which the original UD-code can be partitioned. After insertion, instead of $U^k D^\ell$ we obtain:



The last string may be rewritten as

$$U^{m_1+1} D\, U^{m_2+1} D \ldots U^{m_{k-1}+1} D\, UD^{\ell+1}$$

which has the pruned tree property.

Apart from this insertion of symbols U and D in the parts U^kD^ℓ of the original code, a new part (U) (UD) is added in front of the entire string. This string may be rewritten as U^2D.

(4) Finally we show that the S-map of the code obtained is the original UD-code. We derived above that a part U^kD^ℓ is transformed into $U^{m_1+1}D\,U^{m_2+1}D \ldots U^{m_{k-1}+1}D\,UD^{\ell+1}$. Mapping S replaces each part $U^{m_i+1}D$ by a single U, and $UD^{\ell+1}$ by UD^ℓ, so $U^{m_1+1}D \ldots U^{m_{k-1}+1}D\,UD^{\ell+1}$ is replaced by U^kD^ℓ. By the procedure the part U^2D was added in front of the entire string. Mapping S first replaces U^2D by a single U and next omits this U. It follows that S is the inverse of the transformation described above. □

We note that the above transformation, which is $S^{\leftarrow}$, can be easily algorithmised. The only non-trival part is to devise an algorithm which counts the number of subnests of a given nest. However, this is not hard; one may for example make use of the level numbers of a UD-code.

5. REFERENCES

[1] N.G. de Bruijn and B.J.M. Morselt, A note on plane trees, Journal of Combinatorial Theory 2, 27-34, 1967.

[2] F. Göbel and R.P. Nederpelt, The number of numerical outcomes of iterated powers, Amer.Math.Monthly, vol. 78, p.1097-1103, December, 1971.

[3] R.K. Guy and J.L. Selfridge, The nesting and roosting habits of the laddered parenthesis, Amer.Math.Monthly, vol. 80, p. 868-876, October, 1973.

[4] R.C. Read, The coding of various kinds of unlabeled trees, in Graph Theory and Computing, Ed. R.C. Read, Academic Press, New York, 1972.

Typing: C.I.T. Kolk-Koenraat

English correction: A.V. Zimmermann