ソフトウエア工学 16－4
（1980. 11. 17）

流れ木 flowtree による手続きとデータ型の記述

夜久竹夫（東海大学理学部）・二木厚吉（電子技術総合研究所）・足立暁生（日本アイビーエム）

要約．流れ図に対しては、構造的プログラムの概念を導入することにより各種の改良が試みられている（[6, 1973] など）．我々は流れ図それ自体を改良する最初の、その種の試みとして手続きの流れ木（木フローチャート）とその展開法を提案した[9, 1978]他．その記法の一部は以後の構造的流れ図記法に採り入れられた[3]．

一方、プログラムは手続き記述の部分とデータ記述の部分からなっている．本論文では、流れ木の概念をデータ記述に適用し（[3]に従がう）、流れ木によるデータ型の記述法と展開法を導入し、データ型木 type tree とよぶ．さらにデータ型の例として PASCAL の構文構造の記述（構文木，syntax tree）を考える．そのあとで parser の一部の手続き流れ木（展開）と対応する構文木（展開）との関連を述べることにより、流れ木の「教育」・「文書化」用 tool としての有効性を議論する．

付録で標準 PASCAL の構文木を全て列挙する

key words　programming technique, flowtree, procedure tree, type tree, computation tree, data tree, syntax tree, documentation, education.

1．序

概念の間の関係を表現する手段として図面は広く用いられている．高級言語を前提とせずに考え出された流れ図（Goldstein - von Neuman [5, 1947]）は部分手続きという概念の間の時間的な順序関係により手続きを表現した図面である．以来流れ図は多くのプログラマーによって用いられてきた．実際プログラマーが時間的順序関係に注目していた間は Goldstein - Neuman の流れ図は充分効果的なものであった．

ところで[2]で Dijkstra は手続きを構成する部分手続きの間の上下関係、階層関係にも注目し、構造的プログラミングといわれる方法を提案した．構造的プログラミングの手法はいわゆる goto 論争[2, 1968]などを引き起こしながら日本では 1970 年前後より次第に一般企業、大学でのプログラム教育等に浸透してきた．

同時にその手法はプログラムテキスト（コード）の書法に影響を与え、いわゆる indentation の手法を生んだ．この indentation によりプログラムテキストは格段に解析（解読）しやすく、又作成しやすくなった．それはプログラムテキストが階層構造を視覚的に反映しているからである．このような経過を経て indentation を伴なったプログラムテキストはプログラミングの tool として Goldstein - Neuman 型の流れ図に代わり得るという考え方が生じ、それを裏づける調査も行なわれた[8, 1977]．

一方、流れ図はモジュール間の時間的順序関係とそれに関わる制御構造を明確

に表示している。そのような利点に着目して流れ図に階層関係を反映させる試みが続けられてきた。[1,3,4,6,7,9,11]。それらの記法は総称して構造的流れ図」といわれ、[8]でも、プログラムテキストに対する優位性が予測されている。以下でそれらを概観する。

構造的流れ図には2つの流れがある。1つはプログラムテキストと流れ図を混合させることにより流れ図に階層関係を反映させる方法である。それらには Nash-Schneiderman chart (NSチャート)[6, 1973], Chapin chart [1, 1974] などがある。この2つは制御の流れを示す矢印を取り去ったために、Goldstein-Neuman 型の流れ図とは相当異なる。この系統としては、最近 RSチャート[11, 1980] などが発表されている。

もう一方は制御の流れを示す矢印を保存したまま階層構造を反映させる試みである。この方法に ferstil chart, 木フローチャート(流れ木)[9, 1978], 構造化フローチャート[7, 1979] などがある。又混合型として PAD [3, 1979] がある。

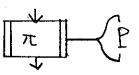
§2で手続きの流れ木(木フローチャート)の記法を解説する。§3で流れ木の記法をデータ型の記述に拡張し、データ型木 type tree を導入する。§4でデータ型の例として文法の構文をとり上げ、それを記述する手法として構文木 syntax tree を導入する。§5で以上の流れ木の有効性について議論する。

2. 手続きの流れ木 procedure tree [9]

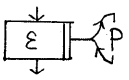
手続きの流れ木(木フローチャート)は以下の(i)-(v)を適当に組み合わせて構成される:

(i) 基本操作記号(介入, 入出力等) JIS等と同じである。

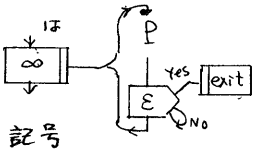
(ii) 手続き定義記号 procedure π; P; は



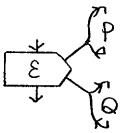
(iii) くり返し記号 for ε do P; と while ε do P; は



(iv) exit 記号 repeat P until ε は

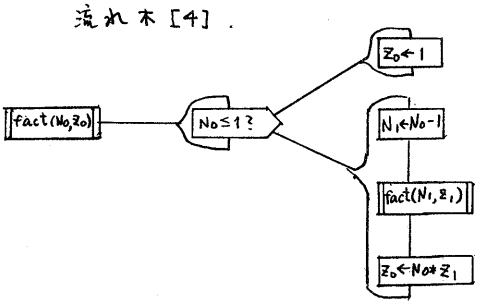


(v) 条件記号 if ε then P else Q; は



(vi) 手続き呼び出し記号 π; は

例 $z_0 \leftarrow N_0!$ を計算する流れ木[4].

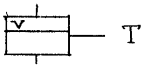


3．　データ型の流れ木：データ型木　type tree

プログラムは手続きだけでなく、対象とするデータの型（構造）を記述する部分を伴なう．　この節で、[9] で提案された流れ木の概念（§2）をデータ型の記述に適用する．　その結果得られる記法を　データ型木　とよぶ．　なおデータ型木は流れ木 [9] を一部とり入れた [二村，3] のデータ記述法を参考にしている．　データ型木を含めて流れ木の有効性は　§5 で展開という概念を通じて議論される．

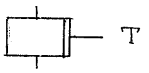
記法．　データ型木は以下の（i）－（v）を適当に組み合わせて構成される（data type tree, type tree）．

(i) データ名：
var V T
「データ名 V は型 T を持つ」はこ



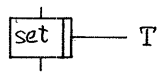
(ii) 基本タイプ
char, number 等

(iii) データ構造
a．くり返し

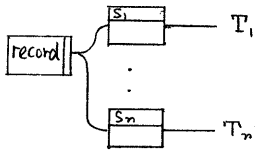


array of T

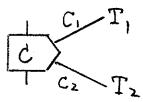
b．set of T



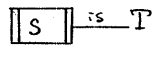
c．record of $S_1 : T_1$; … ; $S_n : T_n$ end



(iv) その他の制御構造
d．selection
case $C : (C_1, C_2)$ of
$C_1 : T_1, C_2 : T_2$ end



(v) 部分定義とその呼び出し
a．type S = T



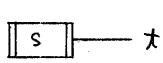
b．type … = … T …

4．　構文の流れ木：構文木 syntax tree

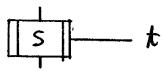
言語の構文を記述する tool として構文グラフが知られている．　一方、言語の構文はデータ型の一種でもある．　この節ではデータ型木の例として、言語の構文を記述することを考える．　その結果得られる流れ木を構文木とよぶ．　ここでは記法を示す．

記法．　構文木 syntax tree は以下の（i）－（v）を適当に組み合わせて構成される．　各記法は BNF あるいは正則表現との関連により示される．

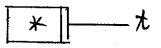
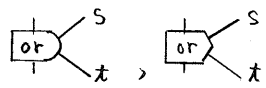
(i) S ::= t は



又は



(ii) terminal symbol t は  (iii) nonterminal symbol n は 

(iv) くり返し t* は  (v) non determinism s | t は 

構文木の例として付録に標準PASCALのそれを列挙する.

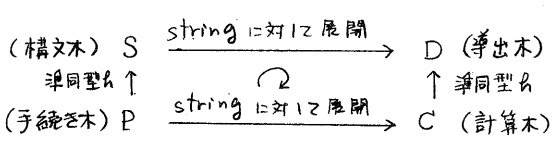
5. 流れ木の展開: 結論

ここでは流れ木の使用例を示すことによりその有効性を議論する. はじめに用語を導入する

用語. 構文木から(通常の)導出木を得る操作を(構文木の)展開という. 又、手続き流れ木を実行して(通常の)計算木 computation tree を得る操作を(手続き流れ木の)展開という[4]

流れ木の使用例としてPASCALの構文の一部と対応するparserの一部をとり上げる.

例. termの構文木をSとする. 列 string に対するSの展開をDとする. 一方、termに対するparserを recursive descendant な方法により構成することを考えるとそれらのparserのうちに次の性質をもつものPが存在する. 即ち(i) Pの流れ木PはSに似ている(グラフ論的に準同型である)、又(ii) Pを入力stringに対して展開して得られる計算木をCとするとCもDに似ている. なお、Dは Cの出力でもある.



一般に次の主張1と2が結論できる.

主張1 上の例は付録の他の構文木についても成り立つ. 従って流れ木とその展開の技法は(recursive descendant な) parserの説明などに有効である.

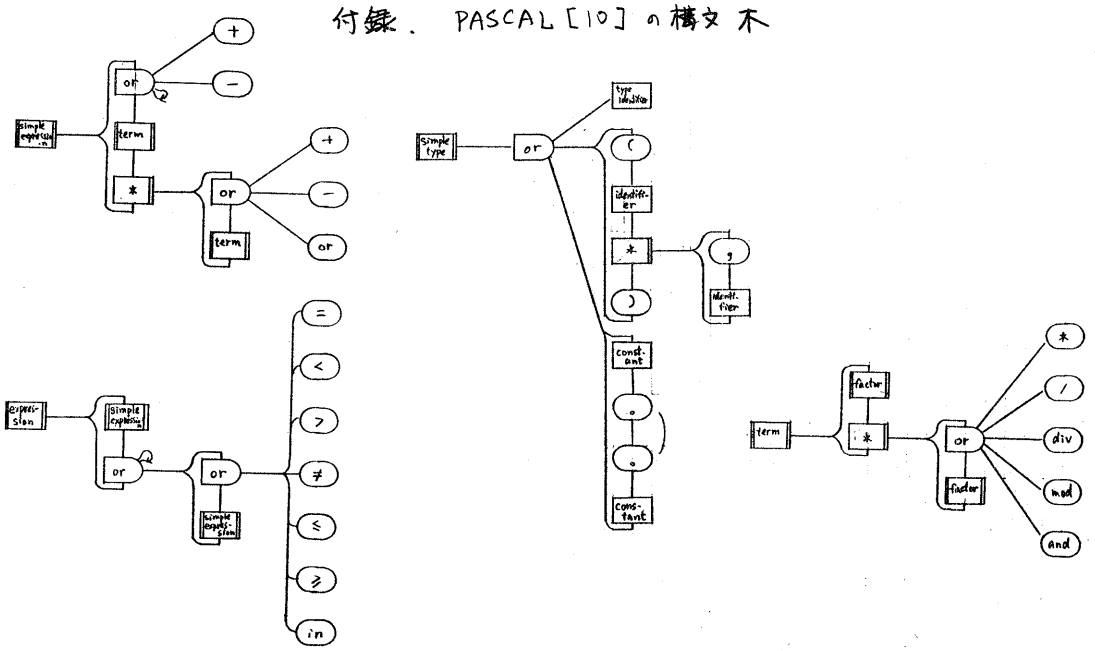
さらに、

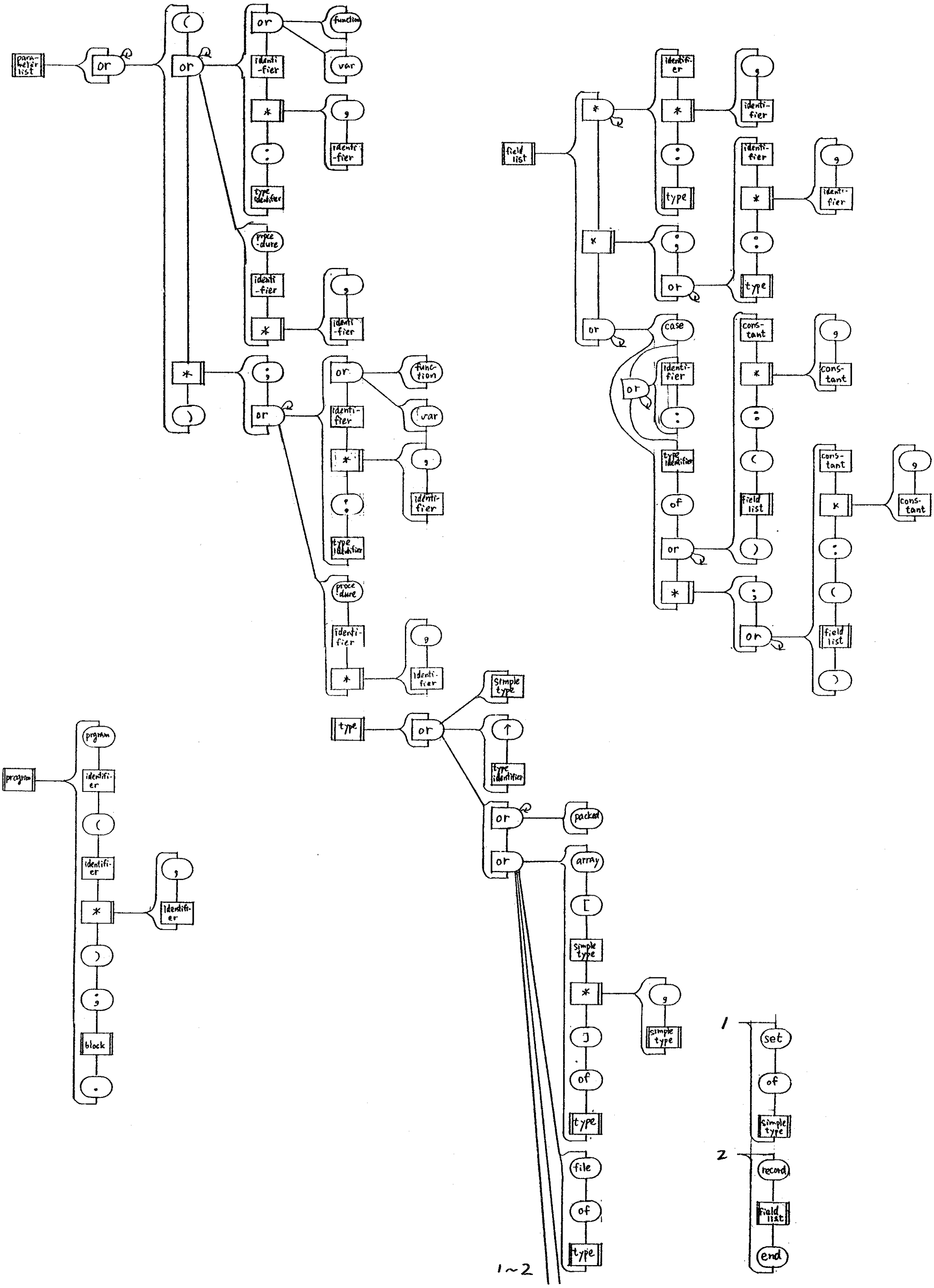
主張2 一般に計算木は計算の構造をあらわす。又、計算木は手続き木と似ている. 従って手続木が計算構造をあらわしている. このことから、流れ木は計算構造が出力をあらわすようなアルゴリズム(上のparser, グラフの生成木構成アルゴリズムなど)の説明に有効である.

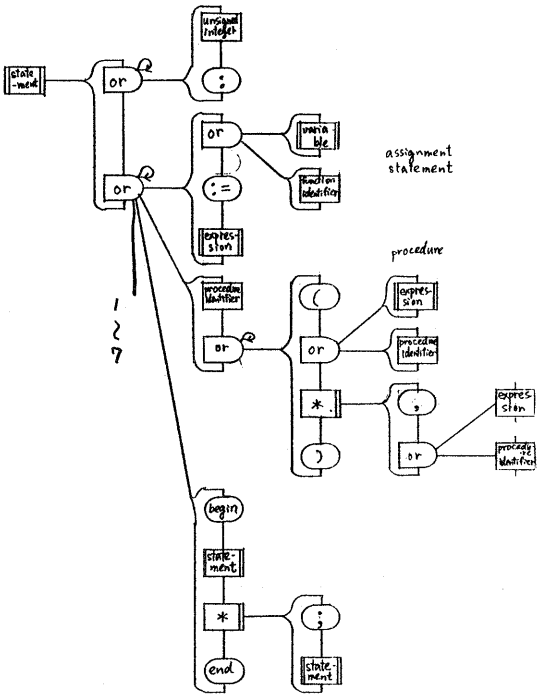
文献

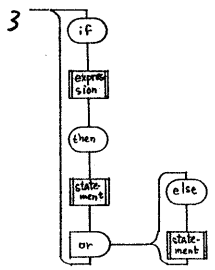
1. N. Chapin, New format for flowcharts, Software – Practice and Experiences 4 (1974), 341 - 357.
2. E. W. Dijkstra, Goto statement considered harmful, Comm. ACM 11 (1968), 147 - 148.
3. 二村良彦他, 問題分析図によるプログラムの設計と作成, 第20回情報処理学会講演論文集 (1979), 269 - 270.
4. 二木厚吉, 夜久竹夫, 木フローチャートとその計算図式について, 79年電気通信学会全国大会予稿集 (1979), 6 : 123.
5. H.H. Goldstein and J. von Neumann, Planning and coding problem for an electronic computing instrument, Part II, vol I. Rep. prepared for the U.S. Army Ordinance Dept., 1947.
6. I. Nassi and B. Schneiderman, Flowchart techniques for structured programming, SIGPLAN Notices 1973 August, 12-26.
7. 大原茂之他, 構造化フローチャートとプログラムの記述について, 第20回情報処理学会講演論文集 (1979), 283 - 284.
8. B. Schneiderman et al., Experimental investigations of the utility of detailed flowcharts in programming, Comm. ACM 20 (1977), 373 - 381.
9. 夜久竹夫, 二木厚吉, フローチャートの木構造型記法, 電子通信学会資料集 AL 78-47 (1978).
10. K. Jensen and N. Wirth, PASCAL, User Manual and Report, Lecture Notes in Comput. Sci. 18 (1974).
11. 鈴木隆一, RSチャートによるプログラム構造化, 第21回情報処理学会講演論文集 (1980), 289 - 290.

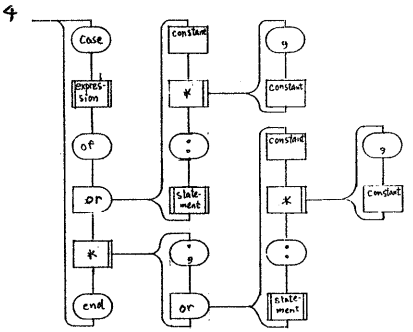
付録. PASCAL [10] の構文木

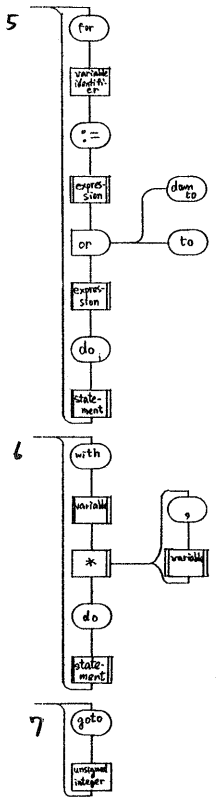



parameter list
or
(
or
or
function
identi-fier
var
*
,
identi-fier
:
type identifier
procedure
identi-fier
*
,
identi-fier
*
;
or
or
function
identi-fier
var
*
,
identi-fier
:
type identifier
procedure
identi-fier
*
,
identi-fier
)
field list
*
identifi-er
*
,
identi-fier
:
type
*
;
or
identi-fier
*
,
identi-fier
:
type
or
case
identi-fier
or
:
type identifier
of
or
*
cons-tant
*
,
cons-tant
:
(
field list
)
;
or
cons-tant
*
,
cons-tant
:
(
field list
)
program
program
identifi-er
(
identifi-er
*
,
identifi-er
)
;
block
.
type
or
simple type
↑
type identifier
or
packed
or
array
[
simple type
*
,
simple type
]
of
type
file
of
type
1~2
1
set
of
simple type
2
record
field list
end


statement
unsigned integer
:
or
variable
function identifier
:=
expression
procedure identifier
procedure
expression
procedure identifier
(
)
*
,
begin
statement
end
;
1 ~ 7
assignment statement

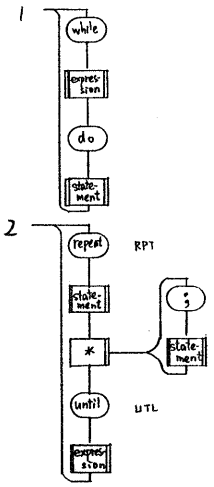

3
if
expression
then
statement
or
else
statement

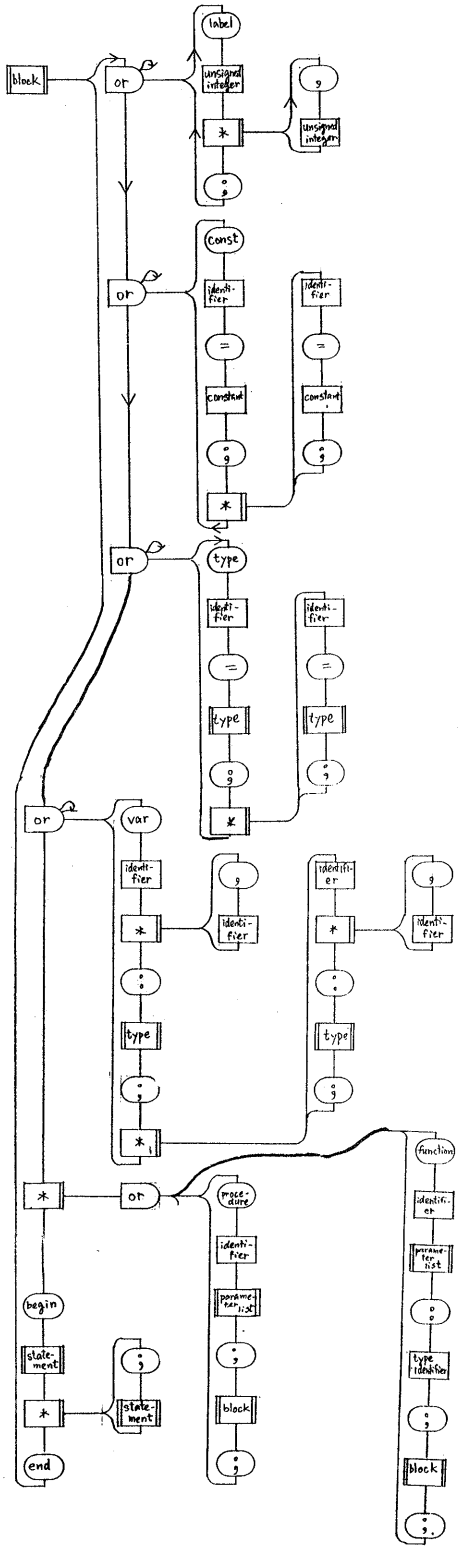

4
case
expression
of
or
*
end
constant
:
statement
,
constant
;

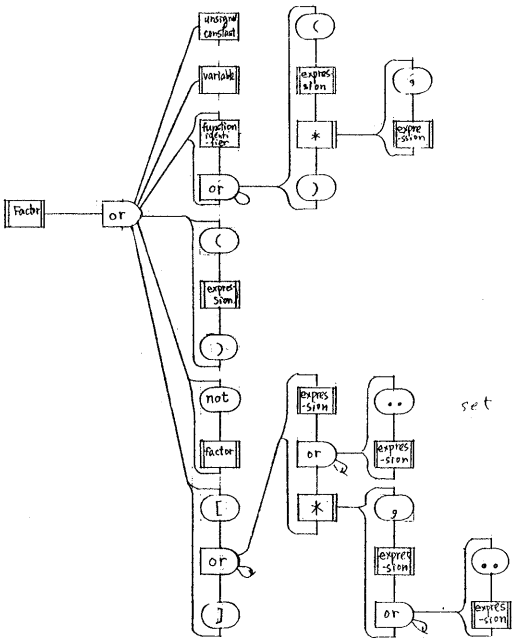

5
for
variable identifier
:=
expression
or
downto
expression
do
statement

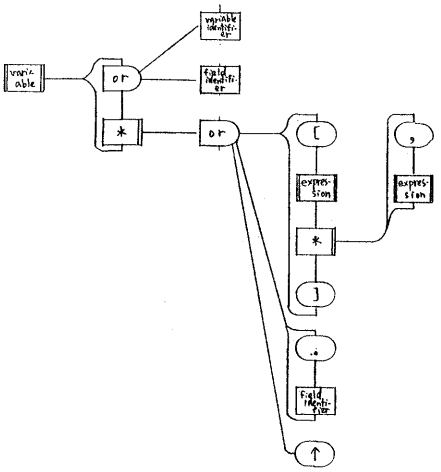
6
with
variable
,
*
do
statement

7
goto
unsigned integer


1
while
expression
do
statement
2
repeat
RPT
statement
;
*
until
UTL
expression


block
or
label
unsigned integer
*
,
unsigned integer
;
or
const
identi-fier
=
constant
;
*
identi-fier
=
constant
;
or
type
identi-fier
=
type
;
*
identi-fier
=
type
;
or
var
identi-fier
*
,
identi-fier
:
type
;
*
identifier
*
,
identi-fier
:
type
;
*
or
procedure
identi-fier
parameter list
,
block
;
function
identifier
parameter list
type identifier
;
block
;
begin
statement
*
;
statement
end


Factor
or
unsigned constant
variable
function designator
or
(
expression
*
)
,
expression
(
expression
)
not
factor
[
or
]
expression
or
*
..
expression
,
expression
or
..
expression
set


variable
or
variable identifier
field identifier
*
or
[
expression
*
]
.
field identifier
↑
,
expression