講　　座

ALGOL N について

(II) 構文と *program* の静的構造

和　田　英　一*

はじめに

〖今回から ALGOL N の報告第2版をその日本語訳を中心として解説する．第2版 **1.** のうち **1.4** Rough Illustration of Programs の節は全体が pragramtics よりなり，ALGOL N の概説にあてられているが，この部分はすでに前回，岩村により紹介されているので，今回と次回とで **1.4** を除いた **1.** と **2.** 全体を紹介する．〗

1. 構　文

〖ALGOL N の構文は ALGOL **68** のそれにならい，基本的な構文の部分 (*straight language*) と略記法を採用した部分 (*extended language*) とからなる．一般の利用者は *extended language* で program をかいてよい．*extended language* は格好ができるだけ ALGOL **60** に似るようになるよう試みた．*extended language* でかかれた program は一意的に *straight language* に戻すことができるので ALGOL N の文法は主として *straight language* の program に対してのみ規定してある．

straight language の構文は context free grammar の生成文法のような形で与えられており，*extended language* のそれは context sensitive な変形文法のような形で与えられている．*extended language* の program の表現は，まず *straight language* の〈expression〉から出発し，生成文法を繰り返し適用して構成した表現に，*extended language* の変形文法を繰り返し適用して構成したものである．(この構成は Chomsky の Syntactic structure に述べられているものに似ているが，ALGOL N では *straight language* の段階で意味が完全にきまってしまうこと，変形文法の適用の順序が自由であることなどで，Syntactic structure の考え方と異なる．)

報告の **1.1** は *straight language* の構文記述のための *meta-language*，**1.2** はその *meta-language* による *straight language* の構文，**1.3** は *extended language* へ拡張するための変形規則を述べる．〗

1.1 構文の記述のための *meta-language*

〖*meta-language* は以下の報告に述べるようにBNFを若干変更したものになっている(AN 記法とよぶ)．ALGOL **60** の構文の記述に用いられた BNF に COBOL などの構文記述に用いられた(1)同じものの繰り返し (Kleene の ＊operation に近いもので $_{\circ\circ\circ}$ で表わす) や，(2) ある部分のなくてもよいこと (Brooker Morris の ? operation で $\left[\ \right]$ で表わす) をまず採用した．また有沢も指摘するように，〈list〉::=〈element〉|〈list〉〈delimiter〉〈element〉の形のものが多いので (有沢　誠："ALGOL **60** の形式的な文法構造について"，情報処理 Vol. 11, No. 9, Sept. 1970, pp. 509～516)，これも容易に表わせるように工夫した．(ALGOL **68** でいう chain of NOTIONS separated by SEPARATORS は AN 記法では NOTION $\left\{\text{SEPARATOR}\right\}_{\circ\circ\circ}$ となる．) (結果的には PL/I の形式的定義に用いられた拡張された Backus 記法に類似している．情報処理学会 PL/I 研究委員会："PL/I の形式的定義について (2)"情報処理 Vol. 11, No. 9, 1970 Sept. pp. 546～553 の **5.4.1** 拡張された Backus 記法参照)．

BNF をそのまま採用しなかった理由は，(1) 正規表現でも記述できる部分で再帰的に記述しているのは読みにくい．(2) 右にのびてゆく規則 (左再帰的 left recursive，たとえば〈identifier〉::=〈letter〉|〈identifier〉〈letter〉|〈identifier〉〈digit〉) と左にのびてゆく規則 (右再帰的 right recursive，たとえば〈block〉::=〈unlabelled block〉|〈label〉:〈block〉) があり，いか

* 東京大学工学部計数工学科

にもそののびる方向に意味があるようにみせているが，これにはそれほどの意味はない，という点が気に入らないためであった．また ALGOL **60** の BNF でもうひとつ気に入らないのは，上の〈identifier〉の規則で，たとえば MARILYN が〈identifier〉であるのはわかるが，こはれ〈identifier〉MARILY に〈letter〉N がついたものであり，文法でこの program の〈identifier〉といったとき MARLY や MARL や……や M をも〈identifier〉とするのかという疑問も出る点である．もっともこれは構文規則の作り方で避けうる．

一方 ALGOL **68** の構文には matanotion という meta-meta-variable が登場して大変わかりにくい．(ALGOL N でも以下に示すよう α, β などの meta-meta-variable も現われるが，これは metanotion のように再帰的ではない．) これは type (ALGOL **68** では mode) の関係の制限なども構文に入れたかったためで，これはあまりにも複雑になるので，ALGOL N ではそれを採用しなかった．‖

ALGOL N の構文を記述するのに，Backus 記法をわずかに変更した次の *meta-language* を用いる．

1.1.1 *meta-symbol*: $=, (,), [,], \{, \}, |, /, \circ\circ\circ$

{*meta-symbol* を =, (,), [,] のような ALGOL N の basic symbol と区別するため，他の文字よりはるかに大きい字体で印刷する．}

1.1.2 *meta-constant*: **begin, end, real,** (, ;, *a*, *b*, *5* など．

meta-constant は ALGOL N の basic symbol である．

1.1.3 *meta-variable*: 〈expression〉, 〈blok〉, 〈identifier〉, 〈procedure call〉, 〈string〉 など．

meta-variable はある形の構文の要素を表わすのに用いる．

1.1.4 *meta-expression*:

meta-expression は basic symbol の有限の列からなる表現の形を表わし，これを次のように再帰的に定義する．

(1) α が *meta-constant* を表わすとすると，

$$\alpha$$

は *meta-expression* である．表現 φ がただひとつの basic symbol α からなるときのみ，φ は α の形であるという．

(2) α が *meta-variable* を表わすとすると，

$$\alpha$$

は *meta-expression* である．表現 φ が **1.1.5** で定義される意味で *meta-variable* α の形であるときのみ，φ は α の形であるという．

(3) α が *meta-expression* を表わすとすると，

$$(\alpha)$$

も *meta-expression* であり，これを結合の優先度を示すのに用いる．表現 φ が α の形であるときのみ，φ は (α) の形であるという．

(4) α が *meta-expression* を表わすとすると，

$$[\alpha]$$

も *meta-expression* である．表現 φ が空か，α の形であるときのみ，φ は $[\alpha]$ の形であるという．

(5) α, β が *meta-expression* を表わすとすると，

$$\alpha\beta$$

も *meta-expression* である．表現 φ が，α の形の表現と β の形の表現の連結であるときのみ，φ は $\alpha\beta$ の形であるという．

(6) α, β が *meta-expression* を表わすとすると，

$$\alpha|\beta$$

も *meta-expression* である．表現 φ が α の形であるか，β の形であるときのみ，φ は $\alpha|\beta$ の形であるという．

(7) α が *meta-expression* を表わすとすると，

$$\alpha_{\circ\circ\circ}$$

も *meta-expression* である．表現 φ が α の形であるか，$\alpha_{\circ\circ\circ}$ の形の表現と α の形の表現の連結であるときのみ，φ は $\alpha_{\circ\circ\circ}$ の形であるという．したがって，

$$\alpha_{\circ\circ\circ}$$

は

$$\alpha|\alpha_{\circ\circ\circ}\alpha$$

に等価である．

(8) α, β が *meta-expression* を表わすとすると，

$$\alpha\{\beta\}_{\circ\circ\circ}$$

も *meta-expression* である．表現 φ が α の形であるか，$\alpha\{\beta\}_{\circ\circ\circ}$ の形の表現と β の形の表現と α の形の表現の連結であるときのみ，φ は $\alpha\{\beta\}_{\circ\circ\circ}$ の形であるという．したがって $\alpha\{\beta\}_{\circ\circ\circ}$

は $\alpha|\alpha\{\beta\}_{\circ\circ\circ}\beta\alpha$

に等価である．

(9) α, β が *meta-expression* を表わすとすると，

$$\alpha/\beta$$

も *meta-expression* である．表現 φ が

$$\alpha|\beta|\alpha\beta$$

の形であるときのみ φ は $\alpha|\beta$ の形であるという．$|$ による結合と同様，$/$ による結合法則がなりたつ．実際 $(\alpha/\beta)/\gamma$ も $\alpha/(\beta/\gamma)$ も

$$\alpha|\beta|\gamma|\alpha\beta|\alpha\gamma|\beta\gamma|\alpha\beta\gamma$$

に等価であり，このかわりに $\alpha/\beta/\gamma$ とかくことにする．

結合の優先度の順は次のようである．

最強 $\alpha_{\circ\circ\circ}$, $\alpha\{\beta\}_{\circ\circ\circ}$

中間 $\alpha\beta$

最弱 α/β, $\alpha|\beta$

1.1.5 *meta-statement*:

meta-statement は *meta-variable* を定義するのに用いる．α が *meta-variable*, β が *meta-expression* を表わすとすると，

$$\alpha ::= \beta$$

は *meta-statement* である．この *meta-statement* は

「表現 φ が β の形であるときのみ，φ は *meta-variable* α の形であるという」

という文を表わしている．

以下で「φ は α である」という簡単化された文は「φ は α の形である」を意味することにする．

{ ALGOL **60** の ⟨number⟩ と ⟨type declaration⟩ は次のように定義できる．

⟨number⟩ ::= [+|−](⟨digit⟩$_{\circ\circ\circ}$/. ⟨digit⟩$_{\circ\circ\circ}$/$_{10}$[+|−]⟨digit⟩$_{\circ\circ\circ}$)

⟨type declaration⟩ ::= [**own**](**integer**|**real**|**Boolean**)(⟨letter⟩[⟨letter⟩|⟨digit⟩]$_{\circ\circ\circ}$){,}$_{\circ\circ\circ}$ }

【この AN 記法の用例については米田信夫："新算法言語 ALGOL **68**, ALGOL **68** 入門コース(10)" 数理科学，1970 年 7 月号，島内剛一ほか："コンパイラのうちとそと③コンパイラ言語" bit 1971 年 3 月号参照】

1.2 *Straight language* の構文

【以下 **1.2.1** から **1.2.50** に示す構文からわかるように ALGOL N には ALGOL **60** の statement 単位は存在しない．

⟨go to statement⟩ の形はあるが，これは⟨primary⟩ の ⟨expression⟩ である．ALGOL **60** の assignment statement や if statement は ⟨formula⟩ の形の ⟨expression⟩ である．statement がないかわり **effect-***type* の ⟨expression⟩ がある．これは function と subroutine をできるだけ同一に扱いたいという念願による．

ALGOL **60** とのもうひとつの大きい相違は *labelling* の位置である．ALGOL **60** は *labelling* は statement や block の直前につくが ALGOL N では block 内の expression の前につくのみである(**1.2.6** 参照)．そのため一番外の block に *label* がついたときの解釈に困るという事態はおきない．また，これに関連して *variable* や *label* という構文の要素はない．構文の上ではこれは ⟨identifier⟩ として扱われる．

この構文の生成文法には出発記号が明記していないが，利用者の *program* は ⟨expression⟩ を出発記号として作られた ⟨basic symbol⟩ の列だと思ってよい．なお，⟨code body⟩ は ⟨basic symbol⟩ のひとつとして識別されるとする．

⟨secondary⟩ は，たとえば **array** *a* に subscript をつけ "*a*[1]" と element をとりだしたらこれがまた **array** で，さらに subscript をつけ "*a*[1][2]" と element をとりだすというふうに，subscript や selector をつけて element のとられるもの，または *actual parameter* をつけて *call* されるものである．ここで ⟨procedure notation⟩ が (他の notation は ⟨primary⟩ であるのに) ⟨secondary⟩ である．もし，これを ⟨primary⟩ にすると (**1.2.16** 参照)，**procedure** ([⟨expression⟩{,}$_{\circ\circ\circ}$]) ⟨primary⟩ ⟨procedure donor⟩ の ⟨primary⟩ にまた ⟨procedure notation⟩ がかけ，**procedure**([⟨expression⟩{,}$_{\circ\circ\circ}$]) **procedure** ([⟨expression⟩ {,}$_{\circ\circ\circ}$]) ⟨primary⟩ ⟨procedure donor⟩ ⟨procedure donor⟩ となるが，ここで ⟨procedure donor⟩ は省略できるので，ひとつを省略したとするとどちらの ⟨procedure donor⟩ が省略されたかわからなくなるという事情による．他の notation は donor を省略しても構文が曖昧になるとこはない．

⟨extension symbol⟩ の使用法については **1.3** 節を，⟨mark⟩ のそれについては standard declaration の章を参照されたい．】

ALGOL N の program は basic symbol の有限の列の形をとる．basic symbol は構文と意味に関する限り，分解不能な要素として働く，またどの basic symbol も他の basic symbol から分離していなければならない．

ALGOL N の構文との関係において現われる表現も，basic symbol の有限の列の形しかとらない．**1.1** の用法に従うと，

「φ は ⟨basic symbol⟩ である」

という形や，

「φ は ⟨figure⟩ である」

という形の文は，φ が basic symbol か，上で述べた形の表現であることを意味することにする．したがって，次のような *meta-statement* が得られる．

⟨figure⟩═══$\left[\langle\text{basic symbol}\rangle\right]_{\circ\circ\circ}$

$\varphi, \chi, \psi, \omega$ が ⟨figure⟩ であり，φ が連結 $\chi\psi\omega$ であるとき，φ は ψ を「含む」といい，ψ を φ の「部分」とよぶ．さらに ⟨figure⟩ χ, ω のうち少なくともひとつが空でないとき，φ は ψ を「包む」という．

σ を上の文字 $\varphi, \chi, \psi, \omega$ のような表現を代表する記号とする．σ で代表される figure が *meta-variable* の α に対し α の形であるとき，figure の形と代表とを同時に示す便法として，

"σ"

のかわりに

"$\alpha\sigma$"

を用いてよい．

｛　たとえば

「D が

"**let** V **be** E"

（ここに V は ⟨variable⟩，E は ⟨expression⟩ の形の ⟨variable declaratien⟩ であるとする」のかわりに

「⟨variable declaration⟩ D が

"**let** ⟨variable⟩ V **be** ⟨expression⟩ E"

の形であるとする」

とかいてよい．　｝

1.2.1 ⟨expression⟩═══⟨secondary⟩ | ⟨formula⟩

1.2.2 ⟨secondary⟩═══⟨primary⟩ |
⟨procedure notation⟩ |
⟨array element⟩ |
⟨structure element⟩ |
⟨procedure call⟩

1.2.3 ⟨primary⟩═══⟨identifier⟩ |
⟨go to statement⟩ |
⟨block⟩ |
⟨closed expression⟩ |
⟨code⟩ |
⟨effect notation⟩ |
⟨real notation⟩ |
⟨bits notation⟩ |
⟨string notation⟩ |
⟨reference notation⟩ |
⟨array notation⟩ |
⟨structure notation⟩

1.2.4 ⟨declaration⟩═══⟨variable declaration⟩ |
⟨formula declaration⟩ |
⟨mark declaration⟩

1.2.5 ⟨go to statement⟩═══**go to** ⟨identifier⟩

1.2.6 ⟨block⟩═══**begin** $\left[\langle\text{declaration}\rangle;\right]_{\circ\circ\circ}$
$\left(\left[\langle\text{identifier}\rangle:\right]_{\circ\circ\circ}\langle\text{expression}\rangle\right)$
$\left\{;\right\}_{\circ\circ\circ}$ **end**

1.2.7 ⟨closed expression⟩═══(⟨expression⟩)

1.2.8 ⟨code⟩═══**code** ⟨structure donor⟩
⟨primary⟩ : ⟨code body⟩

1.2.9 ⟨effect notation⟩═══**effect**

1.2.10 ⟨real notation⟩
═══**real** ⟨modifier⟩ ⟨real donor⟩

1.2.11 ⟨bits notation⟩
═══**bits** ⟨modifier⟩ ⟨bits donor⟩

1.2.12 ⟨string notation⟩
string ⟨modifier⟩ ⟨string donor⟩

1.2.13 ⟨reference notation⟩═══**reference**

1.2.14 ⟨array notation⟩
═══**array** ⟨array bound⟩ ⟨array donor⟩

1.2.15 ⟨structure notation⟩
═══**structure** ⟨structure donor⟩

1.2.16 ⟨procedure notation⟩
═══**procedure**$\left(\left[\langle\text{expression}\rangle\left\{,\right\}_{\circ\circ\circ}\right]\right)$
⟨primary⟩ ⟨procedure donor⟩

1.2.17 ⟨array element⟩
═══⟨secondary⟩ [⟨expression⟩]

1.2.18 ⟨structure element⟩
═══⟨secondary⟩ [⟨selector⟩]

1.2.19 ⟨procedure call⟩
═══⟨secondary⟩$\left(\left[\langle\text{expression}\rangle\left\{,\right\}_{\circ\circ\circ}\right]\right)$

1.2.20 ⟨formula⟩
==[⟨expression⟩] ⟨mark⟩
[⟨expression⟩]{⟨mark⟩}$_{\circ\circ\circ}$

1.2.1 ⟨variable declaration⟩
==**let** ⟨identifier⟩ **be** ⟨expression⟩

1.2.22 ⟨formula declaration⟩
==**let** ⟨frame⟩ **represent** ⟨expression⟩

1.2.23 ⟨mark declaration⟩
==**let** ⟨mark⟩ **operate** ⟨left priority⟩
⟨right priority⟩

1.2.24 ⟨real donor⟩==[⟨number⟩]

1.2.25 ⟨bits donor⟩==[⟨bits⟩]

1.2.26 ⟨string donor⟩==[⟨string⟩]

1.2.27 ⟨array donor⟩==(⟨expression⟩{, }$_{\circ\circ\circ}$)

1.2.28 ⟨structure donor⟩
==([(⟨selector⟩⟨expression⟩){, }$_{\circ\circ\circ}$])

1.2.29 ⟨procedure donor⟩
==[:([⟨identifier⟩{, }$_{\circ\circ\circ}$])⟨primary⟩]

1.2.30 ⟨modifier⟩==[[[⟨expression⟩]]]

1.2.31 ⟨array bound⟩
==[[⟨expression⟩:][⟨expression⟩]]

1.2.32 ⟨frame⟩
==[( )] ⟨mark⟩ [( )]{⟨mark⟩}$_{\circ\circ\circ}$

1.2.33 ⟨left priority⟩
==[**before**([⟨mark⟩{, }$_{\circ\circ\circ}$]|**all**)**left**]

1.2.34 ⟨right priority⟩
==[**after**([⟨mark⟩{, }$_{\circ\circ\circ}$]|**all**)**right**]

1.2.35 ⟨identifier⟩
==⟨letter⟩ [⟨letter⟩|⟨digit⟩]$_{\circ\circ\circ}$

1.2.36 ⟨selector⟩==(⟨letter⟩|⟨digit⟩)$_{\circ\circ\circ}$:

1.2.37 ⟨number⟩
==⟨digit⟩$_{\circ\circ\circ}$/. ⟨digit⟩$_{\circ\circ\circ}$/(10^{+}|10^{-})
⟨digit⟩$_{\circ\circ\circ}$

1.2.38 ⟨bits⟩==(**0**|**1**)$_{\circ\circ\circ}$

1.2.39 ⟨string⟩==‘[⟨character⟩]$_{\circ\circ\circ}$’

1.2.40 ⟨basic symbol⟩
==⟨non comment⟩|**comment**|**silent**

1.2.41 ⟨non comment⟩
==⟨letter⟩|⟨digit⟩|⟨delimiter⟩|
⟨donor symbol⟩|⟨code body⟩

1.2.42 ⟨letter⟩
==*a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|
n|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*

1.2.43 ⟨digit⟩==*0*|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*

1.2.44 ⟨delimiter⟩==⟨straight symbol⟩|
⟨extension symbol⟩|
⟨mark⟩

1.2.45 ⟨donor symbol⟩
==.|10^{+}|10^{-}|**0**|**1**|‘|’|⟨character⟩

1.2.46 ⟨straight symbol⟩
==**begin**|**end**|**before**|**left**|**after**|
right|(|)|[|]|(|)|[|]|**effect**|
real|**bits**|**string**|**reference**|
array|**structure**|**procedure**|**let**|
be|**represent**|**operate**|**code**|
go to|**all**|,|;|:|:

1.2.47 ⟨extension symbol⟩
==**integer**|**Boolean**|**character**|
name|**quantity**

1.2.48 ⟨mark⟩
==**nil**|**dummy**|**type**|**copy**|**new**|
enproc|**enref**|**deref**|**match**|**as**|
if|**then**|**else**|←|:=|≡|≢|=|≠|
π|*e*|<|≤|>|≥|+|−|×|$^{-1}$|/|
÷|↑|**entier**|**round**|**float**|**sign**|
abs|**arg**|√|**exp**|**log**|**log**$_{10}$|**sin**|
cos|**tan**$^{-1}$|**mod**|**true**|**false**|λ|Λ|
filler|**size**|**lower bound**|**upper**
bound|**conc**|*|**from**|**up to**|**at**|
set|**find**|**in**|**replacing**|**first**|
with|¬|∧|∨|⊕|**complex**|*i*|
Re|**Im**|¯|,|.|**list**|**car**|**cdr**|**for**|
:=|**do**|**step**|**until**|**while**|**case**|
of|**collateral**|**constant**|**scale**|
precision|**exact**|**varying**|**mode**|
⟨etc 1⟩

1.2.49 ⟨character⟩

$$==\mathrm{a}\,|\,\mathrm{b}\,|\,\mathrm{c}\,|\,\mathrm{d}\,|\,\mathrm{e}\,|\,\mathrm{f}\,|\,\mathrm{g}\,|\,\mathrm{h}\,|\,\mathrm{i}\,|\,\mathrm{j}\,|\,\mathrm{k}\,|\,\mathrm{l}\,|\,\mathrm{m}\,|\,\mathrm{n}\,|$$
$$\mathrm{o}\,|\,\mathrm{p}\,|\,\mathrm{q}\,|\,\mathrm{r}\,|\,\mathrm{s}\,|\,\mathrm{t}\,|\,\mathrm{u}\,|\,\mathrm{v}\,|\,\mathrm{w}\,|\,\mathrm{x}\,|\,\mathrm{y}\,|\,\mathrm{z}\,|\,0\,|\,1\,|$$
$$2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9\,|\,(\,)\,|+\,|-\,|.\,|,\,|''\,|\,|$$
$$\hat{\Rightarrow}\,|\,\hat{\Rightarrow}\,|\,\hat{\Rightarrow}\,|\,\langle\text{etc 2}\rangle$$

1.2.50 ⟨comment⟩

==**comment** [⟨non comment⟩]$_{\circ\circ\circ}$
silent

⟨etc 1⟩, ⟨etc 2⟩ と ⟨code body⟩ は指定しない.

1.3 *Extended language* への拡張

program のよみかきをさらに容易にするため, **1.2** で定義した *meta-variable* の意味を次のように拡張することにする. α を構文の要素の形を表わしている *meta-variable*, φ が α の形の表現, ϕ が φ か φ の一部に次の **1.3.1** から **1.3.22** までのある規則を適用して φ から導出された表現とするとき, φ もまた「α の形である.」という. この, 意味の拡張はすべての *meta-variable* に対して, 再帰的かつ同時に実施される. *meta-variable* の前の意味と新しい意味とを明確に述べなければならないときは, 前者を「*straight language* において」, 後者を「*extended language* において」ということばによって示す.

表現 φ が ⟨primary⟩ であり, また

[⟨basic symbol⟩]$_{\circ\circ\circ}$ (⟨straight symbol⟩ | ⟨extension symbol⟩)

の形であるとき, φ は「*primitive*」であるという.

{ 「*primitive*」は構文の要素を表わす形容詞ではあるが, *meta-variable* とはみなされないため, 拡張をこの概念へは適用しない }

1.3.1 "**real** [ ]" を "**integer**" でおきかえてよい.

1.3.2 A が ([⟨digit⟩]$_{\circ\circ\circ}$) の形,
X が (. | 10^+ | 10^-) の形のとき,
"**real** AX" を "AX" でおきかえてよい.

1.3.3 A が (⟨digit⟩$_{\circ\circ\circ}$) の形,
X が ⟨delimiter⟩ のとき,
"**integer** AX" を "AX" でおきかえてよい.

1.3.4 "**bits** [ ]" を "**Boolean**" でおきかえてよい.

1.3.5 X が (**0** | **1**) の形のとき,
"**bits** X" を "X" でおきかえてよい.

1.3.6 **string** [ ]" を "**character**" でおきかえてよい.

1.3.7 "**string** '" を "'" でおきかえてよい.

1.3.8 E が *primitive* のとき,
"(E)" を "E" でおきかえてよい.

1.3.9 "]**array**[" を "," でおきかえてよい.

1.3.10 "[ ]" を空の表現でおきかえてよい.

1.3.11 ":]" を "]" でおきかえてよい.

1.3.12 "][" を "," でおきかえてよい.

1.3.13 X が ( ) | **1**) の形のとき,
"X **effect**:" を "X:" でおきかえてよい.

1.3.14 X_1 が (; | :) の形,
X_2 が (; | **end**) の形のとき,
"X_1 **effect** X_2" を "X_1X_2" でおきかえてよい.

1.3.15 "**let** ⟨identifier⟩ V_1 **be** ⟨expression⟩ E;
let ⟨identifier⟩ V_2 **be** E;
.....
let ⟨identifier⟩ V_n **be** E;"
を "**let** $V_1, V_2, \cdots, V_n$ **be** E;" でおきかえてよい.

1.3.16 V が (⟨identifier⟩ { , }$_{\circ\circ\circ}$) の形,
E が *primitive* のとき,
"**let** V **be** E" を "EV" でおきかえてよい.

1.3.17 G が ⟨frame⟩ のとき,
"**let** G **represent** ⟨expression⟩ E_1;
let G **represent** ⟨expression⟩ E_2;
.....
let G **represent** ⟨expression⟩ E_n;"
を "**let** G **represent** $E_1, E_2, \cdots, E_n$;" でおきかえてよい.

1.3.18 "**let** ⟨mark⟩ M_1 **operate** ⟨left priority⟩ Z_1 ⟨right priority⟩ Z_2;
let ⟨mark⟩ M_2 **operate** Z_1Z_2;
.....
let ⟨mark⟩ M_n **operate** Z_1, Z_2;"
を "**let** $M_1, M_2, \cdots, M_n$ **operate** Z_1Z_2;" でおきかえてよい.

1.3.19 $V_1, V_2, \cdots, V_n$ が ⟨identifier⟩,
⟨primary⟩ E が $1 \leq i_1 \leq i_2 \leq \cdots \leq i_k \leq n, k \geq 0$ に対して $V_{i_1}, V_{i_2}, \cdots, V_{i_k}$ をひとつも含まないとき,
":($V_1, V_2, \cdots, V_n$)
begin let ⟨identifier⟩ U_{i_1} **be** V_{i_1};
let ⟨identifier⟩ U_{i_2} **be** V_{i_2};
.....

let ⟨identifier⟩ U_{ik} **be** V_{ik};
⟨expression⟩ E **end**"

を ":$(A_1, A_2, \cdots, A_n)$ E"

(ここに "A_i" は $1 \leq j \leq k$ のある j に対して $i=i_j$ のとき "**quantity** U_i", $1 \leq j \leq k$ のすべての j に対して $i \neq i_j$ のとき "V_i" または "**name** V_i") でおきかえてよい.

1.3.20 $T_1, T_2, \cdots, T_n$ が ⟨expression⟩,
$A_1, A_2, \cdots, A_n$ が $\left(\left(\textbf{quantity} \middle| \textbf{name}\right)\right.$ ⟨identifier⟩$\left.\right)$
の形のとき,

"**procedure** $(T_1, T_2, \cdots T_n)$ ⟨primary⟩T:
$(A_1, A_2, \cdots, A_n)$ ⟨primary⟩ E"

を "**procedure** $(T_1A_1, T_2A_2, \cdots, T_nA_n)$ T: E" でおきかえてよい.

1.3.21 X が $\left(\left(\middle|,\right)\right.$ の形,
T が primitive のとき,
"XT **quantity** ⟨identifier⟩ V"

を "XTV" でおきかえてよい.

1.3.22 "⟨basic symbol⟩ X" を "⟨comment⟩ AX" か "X ⟨comment⟩ A" でおきかえてよい.

{ 拡張は通常 ⟨expression⟩ の形の表現に適用する. *straight language* における構文の要素の意味, 特に ⟨expression⟩ の意味は, **1.3.1**〜**1.3.22** のどの適用によって *extended language* に移されても, 不変である.}

2. *Program* の静的構造

[前章の規則により構文の上で ALGOL N の資格をそなえた program はこの章で与えられる静的条件を満たさなければならない. ALGOL N は静的に *type* のきまる (Type I の) 言語であるが, *type* をきめるためには parse ができていなければならず, parse するためには formula については parse に必要な mark の facing と priority が block にはいるたびに宣言しうるので, block (および procedure notation) の構造を parse しなければならない.

本章は (1) この parse のしかたと *parsed program*, (2) formula まで parse された *semi-legal program*, (3) *variable declaration* と *formula declaration* により再帰的に *type* のつけれらていった *typed program*, (4) さらに必要な宣言が1回だけあるという条件を満たした *legal program* の順で静的構造の条件をのべる.]

2.1 *Straight language* における *program*

2.1.1 この章では構文の要素はすべて *straight language* であると考え, この意味で ⟨expression⟩ の形の表現をこの章では「*straight language* における *program*」あるいは単に「*program*」とよぶ.

2.1.2 表現 P の部分 A を P における特別の場所との関係で考えるとき, P において何個所にも現われるかも知れぬ表現 A 自身との区別を示すため, 「局所的」という形容詞または棒つき記号 $\bar{A}$ を用いる. したがって P の局所的部分は P における場所をもった P の部分であり, 上の記述における表現 A は, P の局所的部分 $\bar{A}$ の表現である. A の局所的部分 $\bar{A}'$ は, P における A の場所を固定すると, P のある局所的部分によって識別される. つまり, P の局所的部分 $\bar{A}$ の局所的部分 $\bar{A}'$ はまた P の局所的部分である. このような $\bar{A}'$ は, A の局所的部分とか P の局所的部分とよぶよりは, 「局所的」表現とよんでよい. このように形容詞「局所的」を考えている場所のある表現への参照なしに用いるときがある.

{ この概念上の区別を次のように解釈することができる. P を連結 A_1AA_2, n を A_1 における basic symbol の個数とする. 順序対 $\langle A, n\rangle$ を Pの, 表現 A を「表わす」局所的部分とよぶ. A を $\langle A, n\rangle$ の表現, n を P における $\langle A, n\rangle$「の場所」(または A「に対する場所」) とよぶ. }

ある局所的表現がすでに記号 σ で代表され, 場所が文脈により明瞭なときは, その表現を上の記号 $\bar{A}$ のように棒つき記号 $\bar{\sigma}$ で代表する. 逆にはじめに記号 $\bar{\sigma}$ がある局所的表現に対して導入され, 次に棒なし記号 σ がその表現を代表することもある.

2.1.3 *program* を次の条件を満たし, 「*syllable*」とよぶ多くの局所的部分に分割する.

(1) *syllable* は ⟨identifier⟩, ⟨number⟩, ⟨bits⟩, ⟨string⟩, ⟨selector⟩, ⟨code body⟩ または ⟨delimiter⟩ の形である.

(2) *program* における ⟨delimiter⟩ でないふたつの *syllable* はひとつ以上の ⟨delimiter⟩ により分離される.

{ これらの条件のもとで *program* の分割は一意的である. 「*syllable*」の概念はまた *extended language* においても導入される. }

2.1.4 α を *straight language* における構文の要素の形を表わす *meta-exprassion* とする. *program*

P の，P 自身でありうる局所的部分 $\bar{A}$ が α の形であり，ひとつ以上の *syllable* で構成されているとき，$\bar{A}$ を α の形の P の「*constituent*」または P の「*constituent* α」とよぶ．

〖以下の報告に *constituent* ⟨expression⟩, *constituent* ⟨mark⟩, *constituent* ⟨formula⟩, *constituent assenblage* のような形で使用される．〗

｛ $\bar{A}$ がまた *program* であり，$\bar{A}'$ が $\bar{A}$ の *constituent* であるとき，$\bar{A}$ は P の *constituent* である．｝

2.2 Block 構造

2.2.1 同じ形 A のふたつの対象である *variable* A と *label* A とのあいだに区別のあるとき，⟨identifier⟩ の形の表現を「*variable*」とみたり，「*label*」とみたりするときがある．

｛ この区別は前章と同様に，v をあるきまった抽象的対象，l を別のきまった抽象的対象として，*variable* A に対しては順序対 $\langle A, v\rangle$，*label* A に対しては順序対 $\langle A, l\rangle$ をとることができる． ｝

variable と *label* の区別と同様に，次のような区別のあるとき，⟨mark⟩ の形の表現を「*mark*」とよび，⟨frame⟩ の形の表現を「*frame*」とよぶ．表現 M が ⟨mark⟩ の形であるときは，それはまた ⟨frame⟩ の形であるが，*mark* M と *frame* M とを区別の対象とみる．

⟨block⟩ の形か ⟨procedure notation⟩ の形の表現を「*assemblage*」とよび，このような形の *constituent* を「*constituent assemblage*」とよぶ．

2.2.2 E を *assemblage*, $\bar{A}$ を E の *constituent* とする．E が ⟨block⟩, $\bar{A}$ が E に包まれているとき，$\bar{A}$ を「E の *interior* に」あるという．E が ⟨procedure notation⟩

"procodure (⟨expression⟩ $T_1, \cdots,$ ⟨expression⟩ T_n) ⟨primary⟩ T ⟨procedure donor⟩ J"

(ここに $n\geq 0$) であり，$\bar{A}$ が $\bar{J}$ の部分であるとき，$\bar{A}$ を「E の *interior* に」あるという．

$\bar{A}$ が E の *interior* にあり，自分の *interior* に $\bar{A}$ を含んでしかも E の *interior* に含まれるような *assemblage* がひとつもないとき，$\bar{A}$ を「E の *proper interior* に」あるという．

P を (*straight language* における) *program*, $\bar{A}$ を P の *constituent* とすると，P の *constituent* であってしかも $\bar{A}$ を自分の *proper initerior* に含むような *assemblage* は多くともひとつしか存在しない．そのような *assemblage* がひとつもないとき，$\bar{A}$ を「P の *proper exterior* に」あるという．

2.2.3 P を *program*, $\bar{E}$ を P の *constituent assemblage* とする．E が ⟨block⟩

"begin ⟨declaration⟩ D_1;
.....
⟨declaration⟩ D_m;
⟨identifier⟩ L_1^1: ⋯ : ⟨identifier⟩ $L_{i_1}^1$: ⟨expression⟩ E_1;
.....
⟨identifier⟩ L_1^n: ⋯ : ⟨identifier⟩ $L_{i_n}^n$: ⟨expression⟩ E_n **end"**

(ここに $m\geq 0, n\geq 1, i_1\geq 0, i_2\geq 0, \cdots, i_n\geq 0$) であるとき，$j=1, 2, \cdots, m$ に対して $\bar{D}_j$ を P の「*proper declaration*」とよぶ．また $1\leq k\leq n$, $1\leq j\leq i_k$ に対して

"L_j^k:"

の形の *constituent* を P の「*proper labelling*」とよぶ．

$\bar{T}$ が P の *constituent* ⟨expression⟩, $\bar{V}$ が P の *constituent* ⟨identifier⟩ であるとき，順序対

$\langle \bar{T}, \bar{V}\rangle$

を「*parameter-specification*」とよぶ．

E が

"procedure (⟨expression⟩ $T_1, \cdots,$ ⟨expression⟩ T_n) ⟨primary⟩ T: (⟨identifier⟩ $V_1, \cdots,$ ⟨identifier⟩ V_m) ⟨primary⟩ F"

(ここに $n\geq 0$, $m\geq 0$) の形の ⟨procedure notation⟩ であるとき，$1\leq j\leq n, m$ に対して *parameter-specification* $\langle \bar{T}_j, \bar{V}_j\rangle$ を P の「*proper declaration*」とよび，またこの *parameter-specification* を「E の *proper interior* に」あるという．

2.3 *Parsed program*

2.3.1 P を *program* とする．P のある *constituent* に対して，それらの「*immediate constituent*」が割り付けられ，次の(11)までの条件が満たされたとする．

$\bar{E}$ を P の *constituent* とする．

(1) E が ⟨block⟩

"begin ⟨declaration⟩ D_1;
.....
⟨declaration⟩ D_m;
⟨identifier⟩ L_1^1: ⋯ : ⟨identifier⟩ $L_{i_1}^1$:

〈expression〉 E_1;
.....
〈identifier〉 $L_{1^{n}}$: … : 〈identifier〉 $L_{i_n{}^{k}}$;
〈expression〉 E_n **end**"

(ここに $m\geq 0, n\geq 1, i_1\geq 0, i_2\geq 0, \cdots, i_n\geq 0$) であるとき, $\bar{E}$ の *immediate constituent* は $\bar{D}_1, \bar{D}_2, \cdots, \bar{D}_m$ の *immediate constituent* と $\bar{E}_1, \bar{E}_2, \cdots, \bar{E}_n$ である.

〈variable declaration〉

"**let** 〈identifier〉 V **be** 〈expression〉 F"

と 〈formula declaration〉

"**let** 〈frame〉 G **represent** 〈expression〉 F"

の *immediate constituent* は $\bar{F}$ であり, 〈mark declaration〉 には *immediate constituent* はない.

(2) E が 〈closed expression〉

"(〈expression〉 F)"

であるとき, $\bar{E}$ の *immediate constituent* は $\bar{F}$ である.

(3) E が 〈code〉

"**code** (〈selector〉 S_1 〈expression〉 $E_1, \cdots$
, 〈selector〉 S_n 〈expression〉 E_n) 〈primary〉 T
: 〈codebody〉 X"

(ここに $n\geq 0$) であるとき, $\bar{E}$ の *immediate constituent* は $\bar{E}_1, \bar{E}_2, \cdots, \bar{E}_n$ および $\bar{T}$ である.

{ この場合, T を E の「*typifier*」とよぶ. }

(4) E が

"**real** [〈expression〉 F] 〈real donor〉 J"

の形の 〈real notation〉 か

"**bits** [〈expression〉 F] 〈bits notation〉 J"

の形の 〈bits notation〉 か

"**string** [〈expression〉 F] 〈string donor〉 J"

の形の 〈string notation〉 であるとき, $\bar{E}$ の *immediate constituent* は $\bar{F}$ である.

(5) E が 〈array notation〉

"**array** 〈array bound〉 Y (〈expression〉 $E_1, \cdots$,
〈expression〉 E_n)"

(ここに $n\geq 1$) であるとき, $\bar{E}$ の *immediate constituent* は $\bar{Y}$ の *immediate constituent* および $\bar{E}_1, \bar{E}_2, \cdots, \bar{E}_n$ である.

Y が

["〈expression〉 F_1: 〈expression〉 F_2]"

の形であるとき, その *immediate constituent* は $\bar{F}_1$ および $\bar{F}_2$ である. Y が

"[〈expression〉 F_1:]"

か

"[〈expression〉 F_2]"

の形であるとき, その *immediate constituent* はそれぞれ $\bar{F}_1$ か $\bar{F}_2$ である. Y が

"[ ]"

の形のとき, *immediate constituent* はない.

(6) E が 〈structure notation〉

"**structure** (〈selector〉 S_1 〈expression〉 $E_1, \cdots$,
〈selector〉 S_n 〈expression〉 E_n)"

(ここに $n\geq 0$) であるとき, $\bar{E}$ の *immediate constituent* は $\bar{E}_1, \bar{E}_2, \cdots, \bar{E}_n$ である.

(7) E が 〈procedure notation〉

"**procedure** 〈expression〉 $T_1, \cdots$, 〈expression〉 T_n)
〈primary〉 T 〈procedure donor〉 J"

(ここに $n\geq 0$) であるとき, $\bar{E}$ の *immediate constituent* は $\bar{J}$ の *immediate constituent* および $\bar{T}_1, \bar{T}_2, \cdots, \bar{T}_n, T$ である.

J が

": (〈identifier〉 $V_1, \cdots$, 〈identifier〉 V_m)
〈primary〉 F"

(ここに $m\geq 0$) の形であるとき, その *immediate constituent* は $\bar{F}$ である. J が空なら *immediate constituent* はない.

{ $T_1, T_2, \cdots, T_n$ を E の「*parameter* に対する *typifier*」とよぶ. T を E の「結果に対する *typifier*」とよぶ, $V_1, V_2, \cdots, V_m$ を E の「*formal parameter*」とよぶ. F を E の「*procedure body*」とよぶ. }

(8) E が 〈array element〉

"〈secondary〉 F [〈expression〉 E']"

であるとき, $\bar{E}$ の *immediate constituent* は $\bar{F}$ および $\bar{E}'$ である.

{ E' を E の「*subscript*」よぶ. }

(9) E が 〈structure element〉

"〈secondary〉 F [〈selector〉 S]"

であるとき, $\bar{E}$ の *immediate constituent* は $\bar{F}$ である.

(10) E が 〈procedure call〉

"〈secondary〉 F (〈expression〉 $E_1, \cdots$, 〈expression〉
E_n)"

(ここに $n\geq 0$) であるとき, $\bar{E}$ の *immediate constituent* は $\bar{F}, \bar{E}_1, \bar{E}_2, \cdots, \bar{E}_n$ である.

{ この場合, $E_1, E_2, \cdots, E_n$ を E の「*actual parameter*」とよぶ. }

(11) *immediate constituent* を上に述べた *constituent* と, 〈formula〉 の形の *constituent* とにだけ割

り付ける.

〖parse は〈expression〉を, *syllable* を terminal symbol として parsed tree に割り付けようとする. したがって〈expression〉(〈formula〉か〈secondary〉か〈primary〉) の各形の構文の要素で non terminal もの(〈expression〉, 〈secondary〉, 〈primary〉) を *immediate constitute* と称して指定した. この段階の *parsed program* では *formula* 内の parse (これには *mark* の *facing* と *priority* を必要とする.) はおこなわれていない. (*formula* 内の parse までおこなうと *semi-legal*).〗

2.3.2 上のように *immediate constituent* を割り付けた *program* P を, 前節の条件が満たされたときだけ「*parsed program*」とよぶ.

$\bar{E}$ が P の *constituent*〈formula〉であるとき, E は

"$M_1{}^0\cdots M_{i_0}{}^0F_1M_1{}^1\cdots M_{i_1}{}^1F_2M_1{}^2\cdots$
$M_{i_{n-1}}{}^{n-1}F_nM_1{}^n\cdots M_{i_n}{}^n$"

(ここに $n\geq 0, i_0\geq 0, i_1\geq 1, i_2\geq 1, \cdots, i_{n-1}\geq 1, i_n\geq 0$ で $\bar{F}_1, \bar{F}_2, \cdots, \bar{F}_n$ はすべて $\bar{E}$ の *immediate constituent*, $M_1{}^0, \cdots, M_{i_0}{}^0, M_1{}^1, \cdots, M_{i_{n-1}}{}^{n-1}, M_1{}^n\cdots, M_{i_n}{}^n$ はすべて〈mark〉) の形である.

この場合,

"$M_1{}^0\cdots M_{i_0}{}^0$ ( ) $M_1{}^1\cdots M_{i_1}{}^1$ ( ) $M_1{}^2\cdots$
$M_{i_{n-1}}{}^{n-1}$ ( ) $M_1{}^n\cdots M_{i_n}{}^n$"

を E の「*frame*」とよぶ.

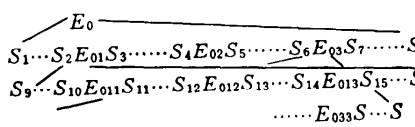
{ $F_1, F_2, \cdots, F_n$ を E の「*operand*」とよぶ. }

2.3.3 *parsed program* P のある *constituent* を「*direct constituent*」とよぶ. これを次のように再帰的に定義する.

(1) $\bar{P}$ は P の *direct constituent* である.

(2) P の *direct constituent* の *immediate constituent* は P の *direct constituent* である.

〖下の図で S_j を *syllable*, E_{ik} を E_i の *immediate constituent* とすると, E_0 は下の図のように*parse* される.



この図の E はすべて E_0 の *direct constituent* である.〗

2.3.4 P を *parsed program*, $\bar{D}$ を P の *proper declaration*, $\bar{E}$ を P の $\bar{D}$ に含まれる *direct constituent* とする. $\bar{E}$ を含んでしかも $\bar{D}$ に含まれているような *constituent*〈procedure donor〉がひとつもないとき, $\bar{E}$ を $\bar{D}$ の「*explicit constituent*」であるという.

〖*explicit constituent* は **2.11.3** で使われる.〗

2.4 mark declaration

2.4.1 〈mark declaration〉は *mark* に対して「*facing*」を, *mark* の順序対に対して「*priority*」を宣言する.

facing は「*double-faced*」,「*left-faced*」,「*right-faced*」か「*non-faced*」である. *priority* は「*natural*」か「*reverse*」である.

〈mark declaration〉 D が

"**let**〈mark〉 M **operate**〈left priority〉 Z
〈right priority〉 Z'"

の形であるとする.

(1) D を「*mark* M に対する宣言」とよぶ.

(2.1) Z と Z' がともに空でないとき, M は D により *double-faced* と「宣言さ」れる.

(2.2) Z が空でなく Z' が空であるとき, M は D により *left-faced* と「宣言さ」れる.

(2.3) Z が空であり Z' が空でないとき, M は D により *right-faced* と「宣言さ」れる.

(2.4) Z と Z' がともに空であるとき, M は D により *non-faced* と「宣言さ」れる.

(3.1) Z が

"**before**〈mark〉 $N_1, \cdots,$〈mark〉 N_n **left**"

(ここに $n\geq 0$) の形のとき, D を $i=1,2,\cdots,n$ の「順序対 $\langle N_i, M\rangle$ に対する *reverse declaration*」とよぶ.

(3.2) Z が

"**before all left**"

の形のとき, D をすべての*mark* N の「順序対 $\langle N, M\rangle$ に対に対する *reverse declaration*」とよぶ.

(3.3) Z' が

"**after**〈mark〉 $N_1', \cdots,$〈mark〉 N_m' **right**"

(ここに $m\geq 0$) の形のとき, D を $i=1,2,\cdots,m$ の「順序対 $\langle M, N_i'\rangle$ に対する *reverse declaration*」とよぶ.

(3.4) Z' が

"**after all right**"

の形であるとき, D をすべての *mark* N' の「順序対 $\langle M, N'\rangle$ に対する *reverse declaration*」とよぶ.

{ D が $\langle M, N\rangle$ に対する *reverse declaration* の

とき，〈M, N〉の *priority* を *reverse* と定義する．しかし D が〈M, N〉に対する *reverse declaration* でないときでも，〈M, N〉の priority が必ずしも *natural* と定義されるわけではない．(→**2.4.4**) }

2.4.2 ある *mark* と，*mark* のある対に対し *facing* と *priority* が **5.** に記述した方法により[standard declaration により，]「前以って」定義されていることがある．どの *mark* に対しても多くともひとつの *facing* しか前以って宣言されておらず，また *mark* のどの順序対に対しても多くともひとつの *priority* しか前以って宣言されていない．そのうえ〈M, N〉に対する *priority* が前以って宣言されているのは，*mark* M, N の少なくともひとつに対して *facing* が前以って宣言されているときのみである．

2.4.3 P を固定された *program*, $\overline{M}$ を P の *constituent* 〈mark〉とする．「$\overline{M}$ の *facing*」は次の場合 (1), (2) により宣言される．

場合 (1)：自分の *interior* に $\overline{M}$ を含み，自分の *proper interior* に *mark* M に対する *proper declaration* を含むような P の *constituent assemblage* が存在する．この場合，$\bar{E}$ を上のようなすべての *constituent assemblage* の間の包み方の順序で最小なものとすると，$\overline{M}$ の *facing* は $\bar{E}$ の *proper interior* における *proper declaration* により M に対して宣言された *facing* である．

場合 (2)：(1)におけるような *constituent assemblage* はひとつもないが，M に対して *facing* が前以って宣言されている．この場合，$\overline{M}$ の *facing* は M に対して前以って宣言された *facing* である．

double-faced が $\overline{M}$ の *facing* のとき，M を「*double-faced*」とよび，他の *facing* に対しても同様のいい方をする．

2.4.4 P を固定された *program*, $\overline{M}$, $\overline{N}$ がその *constituent* 〈mark〉であって，P の同一の *constituent assemblage* の *proper interior* にあるか，ともに P の *proper exterior* にあるとする．順序対〈$\overline{M}, \overline{N}$〉の *priority* は次の場合 (1), (2) により宣言される．

場合 (1)：自分の *interior* に $\overline{M}$ と $\overline{N}$ を含み，自分の *proper interior* に *mark* M か *mark* N に対する *proper declaration* を含むような P の *constituent assemblage* が存在する．この場合，$\bar{E}$ を上のようなすべての *constituent* の間の包み方の順序で最小のものとする．〈M, N〉に対して $\bar{E}$ の *proper interior* における *proper declaration* の間に *reverse declaration* があるとき，〈$\overline{M}, \overline{N}$〉の *priority* は *reverse* であり，ないとき，〈$\overline{M}, \overline{N}$〉の *priority* は *natural* である．

場合(2)：(1)におけるような *constituent assemblage* はひとつもないが，〈M, N〉に対して *priority* が前以って宣言されている．この場合，〈$\overline{M}, \overline{N}$〉の *priority* は〈$M, N$〉に対して前以って宣言された *priority* である．

2.5 *Semi-legal program*

parsed program P を，次のみっつの条件を満たすときのみ「*semi-legal*」とよぶ．

(S 1) P の *constituent* であるどの *assemblage* の *proper interior* にも，どの *mark* に対しても多くともひとつの宣言しかない．

{ したがって *constituent*〈mark〉は多くともひとつの *facing* しかもたない． }

(S 2) $\overline{M}$ が P の *constituent* 〈mark〉であるとき，$\overline{M}$ の *facing* が宣言されている．

(S 3) $\bar{E}$ が P の〈formula〉の形の *direct constituent* であり，$\bar{E}$ の *direct constituent* が $\bar{E}_1$, $\bar{E}_2$, …, $\bar{E}_n$ (ここに $n \geq 0$) であり，そのうえ E が

"E_0'〈mark〉M_1E_1'〈mark〉$M_2E_2'\cdots$〈mark〉M_mE_m'"

(ここに 1, $n-1 \leq m$ で，sequence E_0', E_1', …, E_m' は E_1, E_2, …, E_n と $m-n+1$ 個の空の表現の permutation) の形であるとき，次のことが成り立つ．

(S 3.1) $m=1$ のとき，$\overline{M}_1$ は *double-faced* である．

(S 3.2) $m \geq 2$ のとき，$\overline{M}_1$ は *left-faced*, $\overline{M}_m$ は *right-faced* で，$\overline{M}_2$, …, $\overline{M}_{m-1}$ は *non-faced* である．

(S 3.3) E_0' が〈formula〉であり，

"F_1〈mark〉NF_2"

(ここに F_1 は空か〈expression〉の形のいずれかであり，F_2 は空か P の *direct constituent* のいずれかである．) の形であるとき，〈$\overline{N}, \overline{M}$〉の *priority* が *natural* である．

(S 3.4) E_m' が〈formula〉であり，

"F_1'〈mark〉$N'F_2'$"

(ここに F_1' は空か P の *direct constituent* のいずれかであり，F_2' は空か〈expression〉の形のいずれかである．) の形であるとき，〈$\overline{M}_m, \overline{N}'$〉の *priority* が *reverse* である．

$$\overbrace{E_0' \quad F_1 N F_2}\ M_1 E_1' M_2 E_2' \cdots M_m' \ \overbrace{E_m' \quad F_1' N' F_2}$$

上の図で $\langle N, M_1 \rangle$ は *natural*, $\langle M_m', N' \rangle$ は *reverse* でなければならない.】

{ P を *program* とする. P を *semi-legal program* とするためには P が条件 (S 1), (S 2) および (S 3′) : P において, *left-faced* と *right-faced constituent* 〈mark〉 を左と右の括弧のように対応させ, どの *non-faced constituent* 〈mark〉 をも *left-faced* と対応する *right-faced* の〈mark〉の間におく, を満たすことが必要十分である.

(*straight language* における) *program* P の「*parsing*」とは P を *parsed program* とするような *immediate constituent* の割り付けである. P が上の条件を満たすとき, P を *semi-legal* とするその *parsing* は一意的に決定できる. } (続く)

(昭和 46 年 4 月 27 日受付)