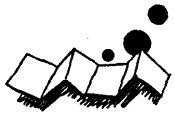
解　説

属性文法による在庫管理システムの記述†



片　山　卓　也††

1. はじめに

属性文法はプログラム言語の意味記述のために，D. E. Knuth によって 1968 年に導入された形式システムであり [Kn]，以来，特にコンパイラ自動生成との関連で多くの研究がなされてきた．属性評価技術の進歩により，属性文法によるコンパイラの記述から実用的なコンパイラの生成も十分に可能な段階に入ったと見ることができる．属性文法はこのように元来プログラム言語やそのコンパイラの形式的記述との関連で生まれた体系であるが，少し見方を変えその形式に多少の変更を加えると，一般の計算を記述するための計算モデルと考えることができる [Ka 1]．本論文は，このような体系によって在庫管理問題の記述を行い，それを通して属性文法のプログラム言語としての特徴を明確にし，その優れた性質を紹介することがその目的である．なお，本論文では，属性文法型プログラム言語についてのみ述べるが，純粋な意味での属性文法との関連については文献を参照されたい [Ka 2]．

2. 在庫管理問題のとらえ方

本在庫管理問題で要求されているのは，出庫依頼を受け取ったときの受付係の仕事である，在庫なし連絡，出庫指示書および在庫不足リストの作成である．本稿は記述用パラダイムの紹介がその主な目的であると考え，基本的には，できるだけ問題を単純化し，プログラムの記述をなるべく見やすくするという方針をとった．このような立場から作成すべきプログラムは，出庫依頼から出庫指示書および在庫不足リストの印刷のみとした．問題には“在庫なし”の連絡の出力も要請されているが，これは在庫不足リストから明らかであるので省略することにした．また，在庫不足の場合は，当該品の積荷が必要量あった時点で不足品を出庫することとされているが，この処理は在庫不足リストを改めて出庫依頼と考えることにより解決されるので，特に考慮することはしていない．

この在庫管理問題の解はどのようなデータベースを設定するかによって，その形が大きく変わると思われる．本稿では解を簡単にするという立場から，次のような図式によって定義される“倉庫”をデータベースとして設定した．

倉庫＝コンテナの集まり

コンテナ＝酒の集まり

このような立場が実用的なシステムにおいて適当であるかどうかは一概にはいえないが，本稿の目的を考えてこのような設定を行った．また出庫指示書や在庫不足リストの印刷書式についても特別な配慮は払っていない．

3. 属性文法型プログラミング言語

本章では，在庫管理問題の記述に用いられる属性文法型プログラミング言語 AG についての紹介を行うが，その特徴は，ひとことでいえば

Rule-based な純関数型プログラミング言語

であり，これにより，副作用のない高い読解性と検証性をもったプログラムを階層的に構成することが可能である．以下では，まず AG の基本原理について述べ，次にそのプログラミング言語としてのいくつかの機能について説明する．

プログラミング言語 AG は，モジュールとモジュール分割という概念にもとづいている．純粋な属性文法との関連では，モジュールは非終端記号に，モジュール分割は生成規則に対応する．

(1)　モジュール

モジュールとは，ある一定の計算を実行する単位であり，その内容はそのモジュールの入出力関係によって規定される．モジュール M が n 個の入力 $x_1, \cdots, x_n$ と m 個の出力 $y_1, \cdots, y_m$ をもつとき，このようなモジュール M を

† Description of an Inventory Control System by Attribute Grammar by Takuya KATAYAMA (Tokyo Institute of Technology).

†† 東京工業大学情報工学科

$M[\downarrow x_1, \cdots, x_n, \uparrow y_1, \cdots, y_m]$

と表記する．AGではこのような x_i や y_i を総称して属性と呼んでいる．Mはこれらの入出力属性間の関係だけによって表現される多入力・多出力関数を表しており，いわゆる副作用は許されない．

（2） モジュール分割

あるモジュールのMの行う計算Cが簡単な場合には，Mの出力値 $y_1, \cdots, y_m$ は入力値 $x_1, \cdots, x_n$ を用いて書き下すことができる．しかし，そうでないときには，Mの計算内容Cをより簡単な部分計算 $C_1, C_2, \cdots, C_k$ に分割し，それらを実行するモジュール群 $M_1, M_2, \cdots, M_k$ を設定し，それらの計算結果を用いて M の計算を表現すればよい．このとき，M は $M_1, \cdots, M_k$ にモジュール分割されると呼び，次のように表す．

$M=M_1M_2\cdots M_k$

このとき，Mをこの分割の親モジュール，$M_1, \cdots, M_k$ を子モジュールと呼ぶ．一般には，どのようなモジュール分割を行うのがよいかは一概にはいえないが，通常は親モジュールMの入出力データの構造と関連した分割が行われることが多い．

AGでは，簡単に実現できるモジュールMが得られるまでモジュール分割を繰り返すことにより計算が進行する．このとき，モジュール分割を停止させるために，特別な空モジュールを子モジュールとしてもつような分割が用いられる．以下ではこのような分割を

$M\equiv$ または単に M

によって表す．

さて，モジュール分割は，親モジュールの行うべき計算を子モジュールに分担させるために行うということを述べたが，これは具体的には，

（1） 子モジュールにどのような入力データを与え，

（2） 子モジュールからの計算結果（出力）から，いかにして親モジュールの出力データを構成するか，

を定めることにより規定される．また，あるモジュールMにはいく通りかのモジュール分割の仕方が許されることがあるので，

（3） どのような条件のもとで，その分割が適用されるか，

も指定しなければならない．（1）と（2）は，属性定義式によって記述され，（3）は分割条件と呼ばれる述語によって定義される．したがって，属性文法型プログラム言語 AG におけるモジュール分割は，一般に次のような形をしている．

$M_0\equiv M_1M_2\cdots M_k$
　when C
　with D

ここで，Cは分割条件であり，Dは属性定義式の集合である．

D中の属性定義式は，考えている分割$M_0\equiv M_1\cdots M_k$中の特定のモジュール M_i の特定の属性 u($i=0$ ならば出力属性，$i\neq 0$ なら入力属性）が他のモジュールの属性 $z_1, \cdots, z_l$ から適当な基本関数あるいはすでに定義されているモジュール（を関数と見たてて）fによって，

$u=f(z_1, \cdots, z_l)$

の形によって定義されるという形で関数的に与えられる．fは属性定義関数あるいは意味関数とよばれる．ここで，uや z_i は特定のモジュール M_r の特定の属性 a を表す表現である．一般に，モジュール分割では同一のモジュールが重複して現れたり（例えば，$M\equiv MR$），異なるモジュールが同一の名前の属性をもつことがあるので，u や z_i は，考えている分割中でのそのモジュール M_r の出現を示す情報と属性名 a の対として，たとえば $a.r$ のように表示しなければならない．理論的な取扱いなどではこの記法は便利であるが，実際のプログラム中では多少見にくいので，本稿では，分割中のすべての属性の出現に異なる識別名を割り当てることにする．これらの識別名は，その分割の中でのみ有効である．また，この識別名を属性生起名と呼ぶ．

分割条件Cは，分割が適用される条件を示す述語であるが，具体的には，そのモジュール分割に含まれる属性生起名と基本述語，基本関数やすでに定義されているモジュール（を述語や関数と見たてて）から構成される．

図-1 はモジュール分割の例である．

この例では，モジュールXがYとZに分割されている．X，Y，Zは，それぞれ1つずつの入力と出力属性をもち，この分割中での，それらの出現は，属性生起名 a，b，c，d，e，fによって表されている．この分割はXへの入力 a が条件 $P(a)$ を満たすときの

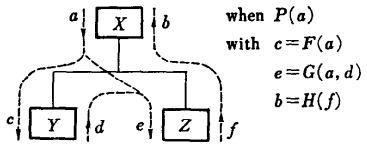


図-1 モジュール分割の例

み行われ，子モジュールYには入力データとして$c=F(a)$が与えられる．Yがその出力dを決定すると（この過程は，Yを親モジュールとする他のモジュール分割によって記述される），このdとXの入力aから$e=G(a,d)$によってZへの入力eが決定される．Xの出力bはZの出力fから$b=H(f)$によって計算される．

この例からXの行うべき計算がYとZによって分担され，最後にそれらを総合してXの計算結果が得られる仕組みが理解されたことと思うが，ここで注意したいことは，XはYとZが所定の計算を行うと仮定した上で，YとZに分解されていることである．この分割の中では，YとZの行う計算の内容を定義することはしない．それらはYとZに関するモジュール分割によって規定される．このようにして，モジュール分割を次々と定義してゆき，計算内容が簡単でその入出力関係を直接書き下せるモジュールに到達したとき，それまでのモジュール分割の総合として，ひとつの関数が定義されたことになる．

［モジュール分割の表記法］

モジュール分割を 図-1 の例のように記述すると，視覚的には理解しやすいが，紙面の都合や計算機への入力などには不便であるので，以下ではこれを次のように表す．

$$X(\downarrow a, \uparrow b) \equiv Y(\downarrow c, \uparrow d), Z(\downarrow e, \uparrow f)$$
$$\text{when } P(a)$$
$$\text{with } c=F(a)$$
$$e=G(a,d)$$
$$b=H(f)$$

属性定義式のかわりに，モジュール分割中の属性生起の場所にその属性の定義式の右辺を書くと with 句は省略できる．

$$X(\downarrow a, \uparrow H(f)) \equiv Y(\downarrow F(a), \uparrow d),$$
$$Z(\downarrow G(a,d), \uparrow f)$$
$$\text{when } P(a)$$

この記述法は分割が簡単な場合には都合がよく，特に属性値が何の変更も受けずに他のモジュールに渡される場合などには便利である．しかし，定義式が長かったりあるいは定義される属性の数が多い場合には見にくくなることがあるので，以下では適宜 with 句も用いる．また，$P(a)$ が常に true，すなわち，分割が常に適用される場合には，when 句を省略することにする．

4. 属性文法型プログラミング言語 AG による在庫管理問題の記述

4.1 属性文法型プログラミング言語 AG のいくつかの機能

在庫管理問題の記述に用いられる属性文法型プログラミング言語 AG の原理については前章で述べた．ここでは在庫管理問題の記述に必要な機能のいくつかについて説明を行う．AG についてのその他の話題については次節で簡単に触れる．

（1） プログラムの形式

AG のプログラムは，基本的には，データ型定義の集合，モジュール分割の集合および初期モジュールの指定からなる．

```
program 初期モジュール名
type
        データ型定義の集合
module
        モジュール分割の集合
```

初期モジュール名は，このプログラムが起動されたとき最初に実行される，すなわち，最初にモジュール分割が適用されるモジュールの名前を指定する．データ型定義ではモジュールの入出力属性のデータ型を指定するためのデータ型名の定義を行う．AG はいわゆる型つき言語であり，すべてのモジュールはその入出力属性のデータ型が陽に指定されていなければならない．データ型は Pascal のそれとほぼ同一であるが，ポインタ型はなく，その代りにリスト型がある．リスト型は，

```
list of 要素型
```

の形で宣言される．リスト型に対しては，nil（空リスト），head（先頭要素），tail（先頭要素を除いた残りのリスト），apendl（リストの先頭に要素を追加）などの基本関数が用意されている．

（2） モジュールおよびモジュール分割の定義法

モジュールはそれを左辺とするモジュール分割がある場合にのみ定義されたと考える．モジュール分割の記法は，前節で与えたものと同一であるが，属性生起名はその属性のデータ型名またはその修飾形を用いて表現する．すなわち，データ型名 t の属性に対する属性生起は，単に

```
t または t_修飾名
```

の形によって表記する．一般に，属性生起名のスコープは各々のモジュール分割という小さい単位であり，

```
program 出庫処理
type
  品名=string
  数量，出庫残=integer
  送り先名=string
  出庫依頼=record          品名：string
                           数量：integer
                       送り先名：string
          end
  インデックス=1..11
  内蔵品=array [1..10] of record  品名：string
                                  数量：integer
                          end
  倉    庫=list of コンテナ
  コンテナ=record  コンテナ番号：integer
                   搬入日時：record     年：integer
                                        月：1..12
                                        日：1..31
                                      時刻：0..23
                                end
                   内蔵品リスト．内蔵品
                   総内蔵量：integer
          end
  搬出指示=[搬出する，搬出しない]
  出庫指示=list of コンテナ出庫指示
  コンテナ出庫指示=record    コンテナ番号：integer
                                数量：integer
                            搬出マーク：搬出指示
                  end
  出庫報告書=text

module
1.  出庫処理[↓出庫依頼，倉庫_出庫前，↑出庫報告書，倉庫_出庫後]
      ≡出庫[↓品名，数量，倉庫_出庫前，↑出庫指示，出庫残，倉庫_出庫後]
        報告書作成[↓出庫指示，出庫依頼，出庫残，↑出庫報告書]
          with  品名＝出庫依頼.品名
                数量＝出庫依頼.数量

2.  出庫[↓品名，数量，倉庫_出庫前，↑出庫指示，出庫残，倉庫_出庫後]
      ≡コンテナ処理[↓品名，数量，コンテナ_処理前，
                    ↑コンテナ出庫指示，コンテナ_処理後，出庫残_1]
        出庫[↓品名，出庫残_1，倉庫_1，↑出庫指示_1，出庫残，倉庫_2]
          when  数量>0 and 倉庫_出庫前≠nil
          with  コンテナ_処理前＝first(倉庫_出庫前)
                倉庫_1          ＝tail(倉庫_出庫前)
                出庫指示＝if コンテナ出庫指示.数量＝0
                             then 出庫指示_1
                             else apndl(コンテナ出庫指示，出庫指示_1)
                倉庫_出庫後＝if コンテナ_処理後.総内蔵量≠0
                                then apndl(コンテナ_処理後，倉庫_2)
                                else 倉庫_2

3.  出庫[↓品名，数量，倉庫，↑出庫指示，出庫残，倉庫]
        when  数量＝0 or 倉庫＝nil
        with  出庫指示＝nil
              出庫残＝if 倉庫＝nil then 数量 else 0

4.  コンテナ処理[↓品名，数量，コンテナ_処理前，↑コンテナ出庫指示，コンテナ_処理後，出庫残]
      ≡コンテナ出庫量決定[↓品名，数量，コンテナ_処理前，↑出庫数量，インデックス]
```

図-2 （次頁へ続く）

```
処理後コンテナ構成[↓コンテナ_処理前, インデックス, 出庫数量, ↑コンテナ_処理後]
  with コンテナ出庫指示=mkrec(コンテナ_処理前.コンテナ番号, 出庫数量, 搬出指示)
       出庫残=数量−出庫数量
       搬出指示=if コンテナ_処理後.総内蔵量=0 then 搬出する else 搬出しない

5. コンテナ出庫量決定[↓品名, 数量, コンテナ, ↑出庫数量, インデックス]
   ≡内蔵量検査[↓品名, 内蔵品, インデックス_初期値, ↑内蔵量, インデックス]
     with 出庫数量=if 内蔵量=0 then 0 else
                    if 内蔵量>数量 then 数量 else 内蔵量
          インデックス_初期値=1
          内蔵品=コンテナ.内蔵品リスト

6. 内蔵量検査[↓品名, 内蔵品, インデックス_初期値, ↑内蔵量, インデックス]
   ≡内蔵量検査[↓品名, 内蔵品[インデックス_初期値+1], ↑内蔵量, インデックス]
     when 内蔵品[インデックス_初期値].品名≠品名 and インデックス_初期値≠11

7. 内蔵量検査[↓品名, 内蔵品, インデックス, ↑内蔵品[インデックス].数量, インデックス]
     when 内蔵品[インデックス].品名=品名

8. 内蔵量検査[↓品名, 内蔵品, インデックス, ↑内蔵量, インデックス]
     when インデックス=11
     with 内蔵量=0

9. 処理後コンテナ構成[↓コンテナ_処理前, 出庫数量, インデックス, ↑コンテナ_処理後]
     with コンテナ_処理後=if 出庫数量=0 then コンテナ_処理前
                          else update(コンテナ_処理前,
                                      (.内蔵品リスト[インデックス].数量, minus(出庫数量)),
                                      (.総内蔵量, minus(出庫数量)))

10. 報告書作成[↓出庫指示, 出庫依頼, 出庫残, ↑出庫報告書]
    ≡出庫指示頭部印刷[↓送り先名, 品名, ↑出庫報告書_1]
     出庫指示印刷[↓出庫報告書_1, 出庫依頼, ↑出庫報告書_2]
     在庫不足リスト印刷[↓出庫報告書_2, 送り先名, 品名, 出庫残, ↑出庫報告書]
       with 送り先名=出庫依頼.送り先名
            品名=出庫依頼.品名

11. 出庫指示頭部印刷[↓送り先名, 品名, ↑出庫報告書]
      with 出庫報告書=write(reset('output'), '出庫指示書', nl, 送り先名, nl, 品名, nl)

12. 出庫指示印刷[↓出庫報告書_1, 出庫指示, ↑出庫報告書_2]
    ≡出庫指示印刷[↓出庫報告書_3, tail(出庫指示), ↑出庫報告書_2]
      when 出庫指示≠nil
      with 出庫報告書_3=write(出庫報告書_1,
                              'コンテナ番号', first(出庫指示).コンテナ番号, nl,
                              '数量', first(出庫指示).数量, nl,
                              'コンテナ搬出指示', first(出庫指示).搬出マーク, nl)

13. 出庫指示印刷[↓出庫報告書, 出庫指示, ↑出庫報告書]
      when 出庫指示=nil

14. 在庫不足リスト印刷[↓出庫報告書_1, 送り先名, 品名, 出庫残, ↑出庫報告書_2]
      with 出庫報告書_2=write(出庫報告書_1, 送り先名, 品名, 出庫残)
```

図-2 在庫管理問題の属性文法型言語によるプログラム

その中では同一のデータ型の属性生起が現れる回数は多くはないのでこの記法は有効である．これにより属性生起名からそのデータ型が直ちに解ると同時に，データ型名と属性生起名に“似て非なる”名前を考え出すという苦痛から逃れることができる．

（3） 構造データ値の更新値

AG は純粋な意味での関数型言語であり，手続き型言語におけるような構造データの部分的更新という概念はなく，したがって，このようなことが必要な場合には新しい値として更新値全体を作り直すことになる．このような操作をデータ構成子だけで記述するのは多くの場合繁雑であり，また更新値の意味が見にくくなる可能性がある．AG ではこれを避けるために構造値更新関数 update が用意されている．

update(v, (p_1, f_1), (p_2, f_2), …)

ここで，v は部分的更新の行われる構造値を，また，p_i は v 中の部分構造を指定するための部分構造指定子の列であり，f_i は p_i によって指定された部分構造値を変換するための関数である．p_i は配列やリストに対する要素指定子 "[i]" とレコード構造の成分指定子 "$.n$" の列である．もちろん，update(v, (p_1, f_1), (p_2, f_2), …) は，v の部分構造 p_1, p_2, … を関数 f_1, f_2, … によって変換した値を表す．

4.2 在庫管理問題の記述

ここでは図-2によって与えられる AG による在庫管理プログラムの解説を行う．

（1） データ型の定義は，問題中に現れる各種データをほぼ忠実に表現したものである．すでに述べたように，倉庫はコンテナの集合として表現するという方針をとっているが，これはコンテナからなるリスト構造によって倉庫を表現することによって実現されている．

（2） 初期モジュールは出庫処理である．出庫処理は出庫依頼と出庫前の倉庫が与えられると，それから出庫報告書と出庫後の倉庫を計算するモジュールとして定義されている．モジュール分割 1. では，このモジュールが出庫モジュールと報告書作成モジュールに分割されることを示している．出庫モジュールには，出庫依頼から得られた品名，数量および出庫前の倉庫が渡され，その実行が終わると出庫指示，出庫残および出庫後の倉庫が返される．一方，報告書作成モジュールは，出庫モジュールから得られた出庫指示と出庫残および出庫処理モジュールの入力である出庫依頼から出庫報告書を作成する．出庫報告書は，出庫指示書と在庫不足リストからなる（10.～14. 参照）．モジュール分割 1. では with 句で 2 つの属性値が定義されているが，これは出庫モジュールに与えられる品名と数量を定めるためのものである．これらは，もちろん右辺の出庫モジュールのそれぞれの場所にそのまま書込んでもよい．出庫前の倉庫は，出庫処理の入力と出庫の入力に，また，出庫後の倉庫は出庫処理の出力と出庫の出力に書かれているが，これは，出庫処理の入力として与えられた出庫前の倉庫がそのまま出庫の入力となること，および，出庫からの出力である出庫後の倉庫がそのまま出庫処理からの出力となることを示している．このモジュール分割には when 句がないが，このことはこのモジュール分割が出庫処理モジュールに対して常に適用されることを示している．

（3） 出庫モジュールは，倉庫の先頭から次々にコンテナを調べてゆき，各コンテナからの出庫指示（コンテナ番号，このコンテナからの出庫数量，コンテナを搬出するか否か）からなる倉庫全体に関する出庫指示を出力すると同時に出庫残を算出する（2., 3. 参照）．このとき，コンテナを次々と調べてゆく過程は出庫モジュールの再帰的分割によって表現されている．再帰は出庫すべき数量が 0 になったか，倉庫を調べつくしたかのいずれかによって終了する．

（4） コンテナ処理では，指定されたコンテナに指定された品名の酒が指定された数量だけあるかを調べ，コンテナ出庫指示，出庫残を計算すると同時に出庫後のコンテナを構成する．これらの作業は，コンテナ出庫量決定と処理後コンテナ構成モジュールによって分担される（4.～9. 参照）．コンテナ出庫量決定には内蔵量検査モジュールが使われるが，これは自分自身を再帰的に使用して，配列で表される内蔵品リストを逐次調べてゆき品名で指定される商品の数量を算出する．処理後コンテナ構成では update 関数を用いて出庫後のコンテナを構成する．なお，minus は次のようなシステム定義のモジュールで

minus(↓x, y, ↑y-x)

あり，minus(x) はその第 1 入力属性の値として x を与えて部分評価したモジュールであり，これが関数として属性定義式のなかで使用されているものである．なお，AG では 1 出力の任意のモジュールは関数として，また論理値をとるような 1 出力の関数は述語として属性定義式や分割条件部に書くことができるようになっている．

（5） 報告書作成モジュールは出庫指示頭部印刷，出庫指示印刷および在庫不足リスト印刷の 3 つのモジュールを用いて出庫指示書と在庫不足リストを印刷する．AG はすでに述べたように純粋な関数型言語であり，入出力も text 型のファイル値を計算することにより行われる．この値は，標準出力ファイル output の空ファイル値 reset('output') に write 関数を用い

て所定のデータを付加（印刷）することにより実現される．write 関数は，

write($f, x_1, x_2, \cdots$)

の形をしているが，これは，ファイルfの後に$x_1, x_2, \cdots$を付加してできるファイルを表している．

5. 属性文法型言語の評価

課題として与えられた在庫管理システムの記述を通して属性文法型言語がプログラミング言語として優れた性質をもっていることが確認できた．属性文法自身がコンパイラの記述において十分な能力を発揮していることから，属性文法型言語がテキストなどの構文的処理には有効であることは明らかであるが，それ以外の一般的な問題にも十分使用できるものであることを示している．

属性文法型言語の特徴は，良くも悪くもすでに重ねて述べた“rule-based”であることおよび“関数型”であることに帰着される．

AG が rule-based であることは，断片的な記述の集合としてプログラムを構成することを意味するが，これは大きなプログラムを少しずつ作成する上で有利であると考えられる．さらに AG は関数型であるので，断片ごとに他の部分とのインタフェースが明確にとれており読解性にすぐれているので，部分的にルール（モジュール分割）を読みながらプログラムの作成や修正を行うことができる．この性質はプログラムの検証や保守の上からも有用である．

一方，AG が rule-based であることの問題点は，FP や λ 記法などの form-based な関数定義法に比べてプログラムの記述量が増えることである．それは，ルールを構成するために導入した中間的なモジュールやその属性生起のための記述によるものであるが，これはある程度エディタなどの工夫によってプログラマに対する負担を軽減することができる．また記述が平板になりやすいのも rule-based な記述の欠点であろう．本在庫管理問題の記述では用いなかったが，AG ではブロック構造を導入してこれを防いでいる．rule-based な方法がよいのか，form-based な方法が良いかは一概には決められないが，複雑なプログラムの記述には rule-based の方が良いのではないかと筆者は考えている[Ka 2]．なお，現在の AG のバージョンには，定形的なモジュール分割をコンパクトに記述する方法とし，継続モジュール，分岐モジュール，繰返しモジュール，選択モジュールなどの form-based なモジュール結合法が用意されている．

AG が純関数型であることの利点は，実行順序をまったく意識せずに，単にデータの依存関係のみを考えればプログラムを読むことができることであり，これも複雑なプログラムを書く上で AG の有利な点であると考えられる．AG のモジュール分割の形は，論理型言語である Prolog の節の形と大変似ており，事実両者の間には深い関係があるが[Ka 2]，AG では人工的な順序づけを極力排しているのに反し，Prolog では節やリテラルの評価についての順序づけを行い，実際上は手続き型言語と差異のない実行順序を導入している．これがパターン・マッチングやバックトラックと組合わされて Prolog に大きな柔軟性を与えていることは事実であるが，プログラムの解読性，保守性や検証性などの点では人為的な実行順序を持たない属性文法型言語の方がはるかに優れている．

最後に AG が純粋に関数型であることの限界は指摘しておく必要がある．AG に限らずいかなる関数型言語も入力値を出力値に変換する関数を定義するものであり，時間や状態といった概念を陽に表現することはできない．したがって，本課題の中心であるデータベースの更新といった操作は，更新前のデータベース値から更新後のデータベース値を得る関数という形でしか定式化することができず，実在物であるデータベースとのインタフェースをとるには，別のメカニズムが必要になる．多くの関数型言語では，関数評価の機構のなかに副作用として状態変化をひき起こすメカニズムを組込んでいるが，これは関数型言語の目ざすところからは望ましくないと筆者は考えている．関数定義とは別の形で状態概念を陽に導入し，両者を調和した形で結合することが記述の明確さを保つ上で重要である．筆者等はこのような目的で関数起動型計算モデル FAM の提案を行っているが[KM]，これによれば出庫処理の過程は図-3 のように記述できる．

図-3 において，太線で囲まれたものは関数を表し，細線で囲まれたものはデータをあらわす．FAM の基

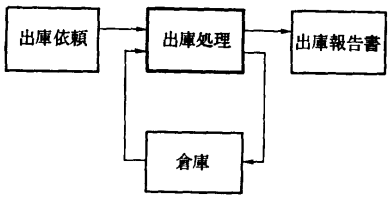


図-3 FAM による出庫処理過程の記述

本動作は，関数はそのすべての入力データがそろうと計算を開始し，計算が終了すると出力データをその結果によって置き換えるというものである．

また，最近オブジェクト指向型言語が注目を集めているが，関数型プログラミングとの結合はこのような目的に適していると考えられる．

参考文献

[Ka 1] Katayama, T.: HFP: A Hierarchical and Functional Programming Based on Attribute Grammar, Proc. of 5th International Conference on Software Engineering, pp. 343-353(1981).

[Ka 2] 片山卓也：属性文法型計算モデル，情報処理，Vol. 24, No. 2, pp. 147-155 (1983).

[KM] 片山卓也，宮地利雄：関数＋共有データ＝プログラム，情報処理学会ソフトウェア基礎論研究会資料 1-2 (1982).

[Kn] Knuth, D. E.: Semantics of Context-Free Language, Math. Sys. Theory, J. 2 pp. 127-145 (1968).

(昭和 60 年 3 月 25 日受付)