数理解析研究所講究録
556巻 1985年 177-186

プログラミング言語ＰＬ／０の代数的仕様記述

北　英彦　　坂部　俊樹　　稲垣　康善
名古屋大学　工学部

1. まえがき

プログラミング言語の意味を形式的に与える方法の一つに代数的仕様記述法がある。これは、言語の構文領域および意味領域を代数（抽象データ型）として定義し、プログラムの意味をこの二つの代数間の準同型写像で与える方法である。代数的仕様記述法を用いると、言語の意味領域を抽象データ型として形式的に取り扱うことができ、また、意味領域の実現に関する形式的な議論が可能である。そのため、代数的仕様記述法は、プログラム検証のための基礎ならびにコンパイラ自動生成のための理論的枠組みとして、有望なアプローチと思われる。

著者らは、既に、文献(6)で代数的仕様記述法の一方法を提案した。そこで、その有効性を確認するため、プログラミング言語として基本的な制御機能を備えた言語ＰＬ／０[12]の仕様記述を行った。本報告では、ＰＬ／０の仕様記述の考え方を述べ、代数的仕様記述法によってプログラミング言語の意味がどのように記述できるかを示す。

なお、ＰＬ／０の仕様記述を行なう際、コンパイラの字句解析部で意味処理が行われることを考慮し、文献(6)の仕様記述法を修正した代数的仕様記述法を用いたので、２章でこれについて簡単に説明し、３章でＰＬ／０の仕様をいかに記述したか説明する。

2. プログラミング言語の代数的仕様記述法

代数的仕様記述法では、言語の仕様を、構文領域の仕様、意味領域の仕様、意味写像の仕様の３項組として定義し、各々、文脈自由文法、抽象データ型の仕様、意味方程式の集合で与える。

ところで、通常、コンパイラの作成の際には、識別子・数字列等については、意味処理を含めて字句解析部で扱うことが多い。そのため、代数的仕様記述法に関する文献[2]，[3]，[7]では、これらのものは構文と意味が既に与えられたものとして議論をしている。しかしながら、形式的な扱いはされていない。そこで、識別子のような基本的なものは既に構文と意味が与えられているという考えかたを形式化した代数的仕様記述法を考えたので、本章で説明する。以下、構文領域、意味領域、意味写像の順にその仕様記述法を定義する。ただし、本報告で用いる抽象データ型に関する概念、記法についての詳細は文献(5)を参照されたい。

2.1. 構文領域の仕様記述法

構文領域は、基本的には文脈自由文法を用いて仕様記述を行うが、ここでは、識別子などの構文

は既存のものとするように変更を行う。

【定義 2.1.1 (構文領域の仕様)】 構文領域の仕様Gは、6項組 $\langle V_T, V^B, V, P, S_0, \mathfrak{L}^B \rangle$ ある。ここで、V_Tは終端記号の集合、V^Bは組み込み非終端記号の集合、Vは被定義非終端記号の集合、Pは生成規則の集合、S_0はVの要素で開始記号、$\mathfrak{L}^B$は組み込み構文領域である。生成規則は、$p: N \to \alpha$の形とする。pは生成規則の名前、NはVの要素、αは$(V^B \cup V \cup V_T)^*$の要素である。また、組み込み構文領域$\mathfrak{L}^B$は、文字列の集合族 $\langle \mathfrak{L}^B{}_N \rangle$ $N \in V^B$である。□

構文領域の仕様Gが定める言語を、文脈自由文法で生成される言語の定義を用いて定義する。即ち、構文領域の仕様Gから文脈自由文法G'を、

$G' = \langle V_T \cup \{w \mid {}^{\exists}N \in V^B,\ w \in \mathfrak{L}^B{}_N\},\ V',\ P',\ S_0 \rangle$

$V' = V^B \cup V$、　$P' = P \cup \{w: N \to w \mid w \in \mathfrak{L}^B{}_N,\ N \in V^B\}$

と定め、仕様Gの定める言語は、文脈自由文法G'で生成される言語$L(G')$であるとする。

ADJ[12]で指摘されているように、文脈自由文法をシグニチャとみなすと文脈自由文法で生成される言語を抽象データ型として代数的に定義できる。本仕様記述法でも、言語を抽象データ型として代数的に定義する。まず、文法G'をシグニチャ $\langle V', P'' \rangle$ とみなす。即ち、非終端記号の集合V'をソートの集合とし、演算記号領域P'' を生成規則の集合P'に対して、

$P'' = \langle P''_{z,s} \rangle\ z \in V'^*, s \in V'$、　$P''_{z,s} = \{p \mid p: N \to \alpha \in P',\ s = N,\ z = nt(\alpha)\}$

と定める。ここで、$nt(\alpha)$は記号列α中のすべての終端記号を空系列で置き換えて得られる非終端号列を表す。このように、文脈自由文法G' をシグニチャとみなし、構文領域の仕様Gが定める構文領域を次のように定義する。

【定義 2.1.2 (構文領域)】 構文領域の仕様Gの定める構文領域$\mathfrak{L}$は、項代数(抽象データ型)$T[G']$である。□

2.2. 意味領域の仕様記述法

代数的仕様記述法では、意味領域は抽象データ型の代数的仕様記述で与えられる。抽象データ型の仕様記述の議論では、新しいデータ型を構成する場合は、あらかじめ与えられたデータ型の上に構成することを考えるのが普通である[10]。ここでも、識別子・数字列に対する既存の意味領域を前提とする意味領域の仕様を定義する。

【定義 2.2.1 (意味領域の仕様)】 意味領域の仕様$\mathcal{D}$は、5項組 $\langle \Sigma^B, \Sigma, \mathcal{V}, \mathcal{A}, SD^B \rangle$ である。ここで、Σ^Bはシグニチャ $\langle \mathcal{S}^B, \mathcal{F}^B \rangle$、$\Sigma$はシグニチャ $\langle \mathcal{S}, \mathcal{F} \rangle$ である。ただし、$\mathcal{S} \supseteq \mathcal{S}^B$、すべての$z \in \mathcal{S}^{B*}, s \in \mathcal{S}^B$について、$\mathcal{F}_{z,s} \supseteq \mathcal{F}^B{}_{z,s}$ とする。また、$\mathcal{V}$は変数集合族 $\langle \mathcal{V}_s \rangle$ $s \in \mathcal{S}$、$\mathcal{A}$は公理の集合、SD^Bは組み込み意味領域で、Σ^B代数である。公理は、等式$\xi \approx \eta$である。ここで:ξ, ηは同一ソートの$\Sigma(\mathcal{V})$項である[5]。□

意味領域の仕様$\mathcal{D}$の定める意味領域を、組み込み意味領域SD^Bを部分代数とする抽象データ型として定義する。

【定義 2.2.2 (意味領域)】 意味領域の仕様$\mathcal{D}$が完全性[5]、無矛盾性[5]を満たすとき、仕様$\mathcal{D}$の定める意味領域SDは、商代数$T[\Sigma(SD^B)]/\equiv$である。ここに、$\equiv$は、$\xi \equiv \eta \overset{\text{iff}}{\mathcal{A}} \vdash^{SD^B} \xi \approx \eta$で

定められる合同関係であり、記号$\vdash^{SD^B}$は組み込み意味領域SD^Bで成り立つ等式を公理として許したときの推論可能性を表す。□

2.3. 意味写像の仕様記述法

代数的仕様記述法では、意味写像は文脈自由文法の各生成規則に意味方程式を一つ与えることで仕様記述される。本報告でも、基本的には同じだが、組み込み構文領域に対する意味は、組み込み意味写像として与えられているものとして意味写像の仕様の定義を行う。

【定義2.3.1(意味写像の仕様)】 $G=\langle V_T, V^B, V, P, S_0, \mathfrak{L}^B\rangle$を構文領域の仕様、$\mathfrak{D}=\langle \Sigma^B, \Sigma, \mathfrak{V}, \mathcal{A}, SD^B\rangle$を意味領域の仕様とする。意味領域の仕様$\Gamma$は、5項組$\langle D, M^B, M, X, R, M^B\rangle$である。ここで、Dは関数$D: V^B \cup V \to \mathfrak{D}$である。ただし、$N\in V^B$について、$D(N)\in \mathfrak{D}^B$とする。また、$M^B$は組み込み構文領域$\mathfrak{L}^B{}_N$から組み込み意味領域$SD^B{}_{D(N)}$への関数集合上の変数(関数変数)$M_N$の集合$\{M_N \mid N\in V^B\}$、Mは構文領域$T[G']_N$から意味領域$SD_{D(N)}$への関数集合上の関数変数$M_N$の集合$\{M_N \mid N\in V\}$、Xは構文領域上の変数集合族$\langle X_N\rangle\ N\in V$、Rは意味方程式の集合$\{R_p \mid p\in P\}$、$M^B$は、組み込み意味写像で、写像族$\langle M^B{}_N: \mathfrak{L}^B{}_N \to SD^B{}_{D(N)}\rangle\ N\in V^B$である。意味方程式は、各生成規則$p: N\to\alpha$, $nt(\alpha)=N_1\cdots N_n$に対して、

$$M_N[p(x_1,\cdots,x_n)] = \xi_{SD}(M_{N1}[x_1], \cdots, M_{Nn}[x_n])$$

の形とする。ここで、$M_N\in M$、$M_{Ni}\in M\cup M^B (i=1,\cdots,n)$、$x_i\in X_{Ni} (i=1,\cdots,n)$、$\xi$はソートD(N)の$\Sigma(SD^B)$項、$\xi_{SD}$は$\xi$を意味領域SD上で評価して得られる導出関数(derived function)である。□

意味写像の仕様は、組み込み意味写像をベースとして、関数変数の値(関数)を決めるための再帰方程式系になっている。このように意味写像の仕様は、方程式系として与えられるので、仕様の意味、即ち、意味写像を方程式系の解として定義するのは自然であろう。

【定義2.3.2(意味写像)】 意味写像の仕様Γの定める意味写像Mは、次の条件(1)、(2)を満たす写像族$\langle M_N: \mathfrak{L}_N \to SD_{D(N)}\rangle\ N\in V\cup V^B$である。

(1) $N\in V^B$について、$M_N = M^B{}_N$

(2) 各意味方程式R_pについて、$M_N[p(x_1,\cdots,x_n)] = \xi_{SD}(M_{N1}[x_1],\cdots,M_{Nn}[x_n])$□

3. プログラミング言語PL/0の仕様記述

2章で述べた代数的仕様記述を用いて、プログラミング言語PL/0の仕様記述を行う。PL/0は、Wirthが文献(12)のなかでコンパイラ作成の例のために用いた言語で、簡単ではあるがプログラミング言語として基本的な制御機能を備えている。もう少し詳しく述べると、PL/0は整数上の計算を行うための言語で、入出力文、GOTO文の機能は持っていないが、整数定数宣言、整数変数宣言、引数なしの手続き宣言、IF文、WHILE文、CALL文(手続き呼び出し文)、局所変数なしのBEGIN-ENDブロック構造の機能を備えている。

このプログラミング言語PL/0の意味を次のような考え方に基づいて仕様記述する。PL/0は、本来、状態遷移形言語であり、また、入出力文の機能がないので、PL/0プログラムの意味

は、プログラムを実行し終えた後の計算機の内部状態として考えられる。そして、プログラムの各ステートメントの意味は、計算機の内部状態を別の内部状態へ変化させるもの、即ち、状態から状態への関数として考えられる。また、宣言の意味についても、ステートメントと同様に状態から状態への関数として考えることにする。

以下、ＰＬ／０の構文領域、意味領域、意味写像について、どのように記述を行ったか記す。

3.1. ＰＬ／０の構文領域の仕様

文献(12)では、ＰＬ／０の構文は構文図で与えている。ここでは、その構文図を、本仕様記述法の構文領域の仕様となるように等価な文脈自由文法に直した。また、文献(12)で走査プログラムで処理を行なうようにしている識別子、数字列の構文については、２章で定めた組み込み構文領域$\mathfrak{L}^B$として既に与えられていると仮定する。ＰＬ／０の構文領域の仕様の一部を下図に与える。

```
（＊　ＰＬ／０の構文領域の仕様　＊）
G＝〈V_T, V^B, V, P, S_0, 𝔏^B〉
  V_T＝{ begin, end, if, then, else, procedure, call, while, do, var, const,  …… }
  V^B＝{ IDENT, NAT_CONST }
  V  ＝{ PROGRAM, BLOCK, CONST_DEF_PART, VAR_DCL_PART, PROC_DCL_PART,
         STATEMENT, STATEMENT_LIST, CONDITION, EXPRESSION,  …… }
  P  ＝{ p010 : PROGRAM → BLOCK .
         p020 : BLOCK → CONST_DEF_PART ; VAR_DCL_PART ;
                        PROC_DCL_PART ; STATEMENT
         p030 : CONST_DEF_PART → const CONST_DEF_LIST ;
         p040 : CONST_DEF_PART →
         p050 : CONST_DEF_LIST → CONST_DEF , CONST_DEF_LIST
         p060 : CONST_DEF_LIST → CONST_DEF
         p070 : CONST_DEF → IDENT = NUMBER
         p080 : VAR_DCL_PART → var VAR_NAME_LIST ;
           ……
         p120 : VAR_NAME → IDENT
         p130 : PROC_DCL_PART → PROC_DCL_LIST ;
           ……
         p170 : PROC_DCL → procedure IDENT ; BLOCK
         p180 : STATEMENT → IDENT := EXPRESSION
         p190 : STATEMENT → call IDENT
         p200 : STATEMENT → begin STATEMENT_LIST end
         p210 : STATEMENT → if CONDITION then STATEMENT
```

```
        p220 : STATEMENT → while CONDITION do STATEMENT
        p230 : STATEMENT →
        p240 : STATEMENT_LIST → STATEMENT ; STATEMENT_LIST      ····· }
  S₀= PROGRAM
```

図3.1　ＰＬ／０の構文領域の仕様

3.2. ＰＬ／０の意味領域の仕様

文献(12)では、ＰＬ／０の意味領域は、ＰＬ／０に適した仮想計算機として与えられおり、その仮想計算機自身は解釈プログラムによって与えられている。ここでは、その解釈プログラムとは直接的には関係なく、ＰＬ／０の機能を記述するのに必要な演算を持つ抽象データ型を仕様記述することで意味領域を与える。なお、自然数、整数、ブール、識別子については、組み込み意味領域SD^Bとして既にあるものと仮定する。ＰＬ／０の意味写像の仕様を図3.2に与える。

```
(*  ＰＬ／０の意味領域の仕様  *)
𝒟= ⟨Σ^B, Σ, 𝒱, 𝒜, SD^B⟩
  Σ^B= ⟨𝒮^B, ℱ^B⟩
    𝒮^B= { ID,NAT,INT,BOOL}
    ℱ^B= { POS, NEG : NAT → INT    EQ, LT, GT, LE, GE : INT INT→ BOOL   ····· }
  Σ= ⟨𝒮, ℱ⟩
    𝒮=𝒮^B∪ { STATE, STATE-STATE, STATE-INT, STATE-STATE-STATE, ATTR,  ····· }
    ℱ=ℱ^B∪ { I_STATE-STATE : → STATE-STATE
             APPLY_STATE : STATE-STATE STATE → STATE
             APPLY_STATE_D : STATE-STATE-STATE STATE-STATE → STATE-STATE
             APPLY_STATE_STATE-STATE : STATE-STATE-STATE STATE → STATE-STATE
             IF_STATE_D : STATE-BOOL STATE-STATE STATE-STATE → STATE-STATE
             ITERATE : STATE-BOOL STATE-STATE → STATE-STATE
             COMPOSITION : STATE-STATE STATE-STATE → STATE-STATE
             ADD_ID_D : STATE-STATE ID ATTR → STATE-STATE
             UPDATE_D : STATE-STATE ID STATE-ATTR → STATE-STATE
             RETRIEVE_D : STATE-STATE ID → STATE-STATE
             ENTER_BLOCK_D, LEAVE_BLOCK_D : STATE-STATE → STATE-STATE
             INIT_STATE : → STATE
             ADD_ID, UPDATE : STATE ID ATTR → STATE
             RETRIEVE : STATE ID→ ATTR
             ENTER_BLOCK, LEAVE_BLOCK : STATE → STATE      ····· }
  𝒱= ⟨𝒱_s= { s0, s1, s2 } ⟩ s ∈ 𝒮
```

```
A = { APPLY_STATE(I_STATE-STATE(),state0) ≈ state0
      APPLY_STATE(APPLY_STATE_D(state-state-state0,state-state0),state0)
       ≈ APPLY_STATE(APPLY_STATE_STATE-STATE(state-state-state0,state0),
                     APPLY_STATE(state-state0,state0))
      APPLY_STATE(IF_STATE_D(state-bool0,state-state0,state-state1),state0)
       ≈ IF_STATE(APPLY_STATE_BOOL(state-bool0,state0),
                  APPLY_STATE(state-state0,state0),
                  APPLY_STATE(state-state1,state0))
      APPLY_STATE(ITERATE(state-bool0,state-state0),state0)
       ≈ IF_STATE(APPLY_STATE_BOOL(state-bool0,state0),
                  APPLY_STATE(COMPOSITION(ITERATE(state-bool0,state-state0),
                                          state-state0),
                              state0),
                  state0)
      APPLY_STATE(COMPOSITION(state-state0,state-state1),state0)
       ≈ APPLY_STATE(state-state0,APPLY_STATE(state-state1,state0))
      APPLY_STATE(ADD_ID_D(state-state0,id0,attr0),state0)
       ≈ ADD_ID(APPLY_STATE(state-state0,state0),id0,attr0)
      APPLY_STATE(UPDATE_D(state-state0,id0,state-attr0),state0)
       ≈ UPDATE(APPLY_STATE(state-state0,state0),id0,
                APPLY_STATE_ATTR(state-attr0,state0))
      APPLY_STATE(RETRIEVE_D(state-state0,id0),state0)
       ≈ RETRIEVE(APPLY_STATE(state-state0,state0),id0)
      APPLY_STATE(ENTER_BLOCK_D(state-state0),state0)
       ≈ ENTER_BLOCK(APPLY_STATE(state-state0,state0))
      APPLY_STATE(LEAVE_BLOCK_D(state-state0),state0)
       ≈ LEAVE_BLOCK(APPLY_STATE(state-state0,state0))
      APPLY_STATE_STATE-STATE(MAKE_STATE-STATE_D(state-attr0),state0)
       ≈ MAKE_STATE-STATE(APPLY_STATE_ATTR(state-attr0,state0))
                                                    ..... }
```

図3.2　ＰＬ／０の意味領域の仕様

ＰＬ／０はプログラムの意味を、プログラムの実行後の計算機の状態としたので、意味領域にソートSTATE を、また、ステートメント宣言の意味を状態から状態への関数としたので、状態から状態への関数空間を表すソートSTATE-STATE を準備する。演算としては、識別子を登録するADD_ID、識別子の属性値を状態から取りだすRETRIEVE、識別子の値を更新するUPDATE、状態から状態への関

数の関数合成を行う演算COMPOSITION、状態に状態から状態への関数を適用する演算APPLY_STATE等を考える。これらの演算に関する公理は図3.2を参照されたい。

代入文の意味を記述するために、状態から状態への関数UPDATE_D を準備する。UPDATE_D は、状態から状態への関数state-state0を基にして、新しく状態から状態への関数を作る演算である。そして、新たに作られた関数というのは、与えられた状態に対して、元の関数state-state0を適用し、さらに、その結果の状態を更新する関数である。

```
APPLY_STATE( UPDATE_D(state-state0,id0,state-attr0), state0 )
  ≈ UPDATE( APPLY_STATE(state-state0,state0), id0, APPLY_STATE_ATTR(state-attr0,state0))
```

図3.2に与えたＰＬ／０の意味領域の仕様において、_D のついた演算がいくつか出てくるが、上と同様な意図で導入した。また、それらの演算に対する公理についても、同様に記述される。

ＷＨＩＬＥ文については、演算ITERATE : STATE-BOOL STATE-STATE → STATE-STATE を準備する。ITERATE は、状態から状態への関数と状態からブールへの関数とから、状態から状態への関数を作る演算であり、その意味は次の公理で与えられる。

```
APPLY_STATE( ITERATE(state-bool0,state-state0),state0)
≈ IF_STATE( APPLY_STATE_BOOL(state-bool0,state0),
            APPLY_STATE(COMPOSITION(ITERATE(state-bool0,state-state0),
                                    state-state0
                        state0),
            state0)
```

この公理から、ITERATE はＷＨＩＬＥ文そのものであることがわかる。即ち、state-bool0 が状態state0で真のときは、ITERATE で作られた関数は、ITERATE で作られた関数とstate-state0を関数合成したものと等しい結果をだす関数として記述されており、これは、ＷＨＩＬＥ文の条件式が真で１回まわるのに対応している。また、state-bool0が状態state0で偽の場合は、ITERATEで作られた関数は、状態state0をそのまま結果としており、条件式が偽のときのＷＨＩＬＥ文は何もしないのに対応している。

3.3. ＰＬ／０の意味写像の仕様

文献(12)では、コード生成プログラムで、意味写像に対応する部分が与えられている。ここでは、それとは直接的には関係なく、意味方程式を用いて意味写像を仕様記述する。なお、識別子・数字列については、組み込み意味写像M^B で意味が与えられているものとする。ＰＬ／０の意味写像の仕様の全体は、図3.3に与える。

```
（＊　ＰＬ／０の意味写像の仕様　＊）
Γ = ⟨D, M^B, M, X, R, M^B⟩
  D : V^B ∪ V → 𝒮
    D (PROGRAM) = STATE
```

D (BLOCK) = D (STATEMENT) = D (STATEMENT_LIST) = D (CONST_DEF_PART)
=…= D (VAR_DCL_PART) =…= D (PROC_DCL_PART) =…=STATE-STATE
D (CONDITION) =STATE-BOOL ……

M^B= { M_{IDENT}, M_{NAT_CONST} }

M= { $M_{PROGRAM}$, M_{BLOCK}, $M_{STATEMENT}$, $M_{STATEMENT\_LIST}$, …… }

X= ⟨X_N= {x0, x1, x2, x3} ⟩ N∈ V^B∪V

R= {(* p010 : PROGRAM → BLOCK . *)

$M_{PROGRAM}$[p010(x0)] = APPLY_STATE(M_{BLOCK}[x0], INIT_STATE())

(* P020 : BLOCK → CONST_DEF_PART ; …… ; STATEMENT *)

M_{BLOCK}[p020(x0, x1, x2, x3)]

= COMPOSITION($M_{STATEMENT}$[x3]

COMPOSITION($M_{PROC\_DCL\_PART}$[x2],

COMPOSITION($M_{VAR\_DCL\_PART}$[x1],

$M_{CONST\_DEF\_PART}$[x0])))

(* p030 : CONST_DEF_PART → const CONST_DEF_LIST ; *)

$M_{CONST\_DEF\_PART}$[p030(x0)] = $M_{CONST\_DEF\_LIST}$[x0]

……

(* p070 : CONST_DEF → IDENT = NUMBER *)

M_{CONST_DEF}[p070(x0, x1)]

= ADD_ID_D(I_STATE-STATE(), M_{IDENT}[x0],

MAKE_ATTR_CONST(M_{NUMBER}[x1]))

……

(* p120 : VAR_NAME → IDENT *)

M_{VAR_NAME}[p120(x0)]

= ADD_ID_D(I_STATE-STATE(), M_{IDENT}[x0], MAKE_ATTR_VAR(ZERO()))

……

(* p170 : PROC_DCL → procedure IDENT ; BLOCK *)

M_{PROC_DCL}[p170(x0, x1)]

= ADD_ID_D(I_STATE-STATE(), M_{IDENT}[x0], MAKE_ATTR_PROC(M_{BLOCK}[x1]))

(* p180 : STATEMENT → IDENT := EXPRESSION *)

$M_{STATEMENT}$[p180(x0, x1)]

= UPDATE_D(I_STATE-STATE(), M_{IDENT}[x0],

MAKE_ATTR_VAR_D($M_{EXPRESSION}$[x1]))

(* p190 : STATEMENT → call IDENT *)

$M_{STATEMENT}$[p190(x0)]

= LEAVE_BLOCK_D(APPLY_STATE_D(

```
                    MAKE_STATE-STATE_D(
                        RETRIEVE_D(I_STATE-STATE()),M_IDENT[x0]),
                    ENTER_BLOCK_D(I_STATE-STATE())))
(* p200 : STATEMENT → begin STATEMENT_LIST end *)
 M_STATEMENT[p200(x0)] = M_STATEMENT_LIST[x0]
(* p210 : STATEMENT → if CONDITION then STATEMENT *)
 M_STATEMENT[p210(x0,x1)]
   = IF_STATE_D(M_CONDITION[x0],M_STATEMENT[x1],I_STATE-STATE())
(* p220 : STATEMENT → while CONDITION do STATEMENT *)
 M_STATEMENT[p230(x0,x1)] = ITERATE(M_CONDITION[x0],M_STATEMENT[x1])
(* p230 : STATEMENT → *) M_STATEMENT[p230()] = I_STATE-STATE()
(* p240 : STATEMENT_LIST → STATEMENT ; STATEMENT_LIST *)
 M_STATEMENT_LIST[p240(x0,x1)]
   = COMPOSITION(M_STATEMENT_LIST[x1],M_STATEMENT [x0])   …… }
```

図3.2　ＰＬ／０の意味領域の仕様

代入文の意味は、代入文の生成規則に次の意味方程式を対応させることで与えられる。

```
(* p180:STATEMENT → IDENT := EXPRESSION *)
 M_STATEMENT [p180(x0,x1)]
   = UPDATE_D( I_STATE_STATE(), M_IDENT [x0] , MAKE_ATTR_VAR(M_EXPRESSION [x1] ) )
```

代入文は、計算機の状態を更新するためのステートメントであり、状態が与えられると状態を更新する。それゆえ、代入文の意味はUPDATEそのものではなく、状態が与えられると更新を行う演算UPDATE_Dによって記述される。

ＷＨＩＬＥ文の意味は、意味領域の演算ITERATE をそのまま用いて次のように記述できる。

```
(* p220:STATEMENT → while CONDITION do STATEMENT *)
 M_STATEMENT [p220(x0,x1)] = ITERATE( M_CONDITION [x0] , M_STATEMENT [x1] )
```

ここで、条件式は変数を含むので、単なるブールでなく状態からブールへの関数として意味が記述されている。

4. まとめ

代数的仕様記述法を用いて、プログラミング言語としての基本的な制御機能を持つ言語ＰＬ／０の仕様記述を行った。代入文、ＩＦ文、ＷＨＩＬＥ文等の意味記述については簡潔に、また、手続きの機能については多少複雑だが自然な形で、仕様記述できた。このように、簡単ではあるが本質的な機能を備えた言語ＰＬ／０の意味を、形式的で、かつ、自然に記述できることは、代数的仕様記述法の利点である。

ＰＬ／０にはない機能のうち、入出力、パラメータを持つ手続きについては、ここで述べたのと

同様な考え方で記述できる。ただし、ＧＯＴＯ文については、表示的意味論でいう接続法のようなものを考えねばならず、自然な形での記述は難しいと思われる。

本仕様記述法の問題点として例外処理の記述があげられる。ＰＬ／０の仕様記述の際は、未宣言な変数への代入等の意味上のエラーについては便宜的な方法を採用した。この点については、形式的で、かつ、例外処理の記述がしやすい代数的仕様記述法を開発する必要がある。

なお、著者らの研究室において、本代数的仕様記述法に基づくコンパイラ生成系を作成中である。そこでは、具体的な仕様記述言語の設計、コンパイラの各部分の生成系の作成、意味領域（抽象データ型）の直接実現システムの作成等を行っている。

謝　辞

日頃御指導下さる豊橋技術科学大学本多波雄学長、名古屋大学福村晃夫教授、並びに御討論下さる研究室の皆様に感謝致します。

文　献

(1) Courcelle,B., Franchi-Zannettacci,P. : " Attribute grammars and primitive recursive schemes ", Theor. Comput. Sci., Vol.17, pp235-257 (1982)

(2) Despryroux,J. : " An algebraic specification of a Pascal compiler ", SIGPLAN Notice, Vol.18, No.12, pp34-48(1983)

(3) Gaudel,M.C.: "Specification of compilers as abstract data type representations", Proc. of Workshop on Semantics-Directed Compiler Generation, Aarhus. LNCS94, pp140-164(1980)

(4) Goguen,J.A., Parsaye-Ghomi,K.: " Algebraic denotational semantics using parameterized abstract modules", LNCS107, pp292-309 (1981)

(5) 稲垣，坂部：『抽象データタイプの代数的仕様記述法の基礎(1), (2), (3), (4)』， 情報処理, Vol.25, No.1, No.5, No.7 No.9 (1984)

(6) 北他：『抽象データタイプに基づく形式言語の意味記述』，信学技報，AL83-23, (1983)

(7) Mosses,P. : " A constructive approach to compiler correctness ", Proc. of Workshop on Semantics-Directed Compiler Generation, Aarhus. LNCS94 (1980)

(8) 中島：『情報処理入門』，数理科学ライブラリ３,朝倉書店 (1982)

(9) Pair,C. : " Abstract data types and algebraic semantics of programming languages ", Theor. Comput. Sci., Vol.18, pp1-31 (1982)

(10) 杉山，谷口，嵩：『基底代数を前提とする代数的仕様記述』,信学会，論文誌(D)，Vol.J64-D, No.4, pp324-331 (1981)

(11) Thacher,J.W., Wagner,E.G., Wright,J.B. : " Initial algebra semantics and continuous algebra ", Journal of ACM, Vol.24, NO.1,pp68-95 (1977)

(12) Wirth,N. : " Algorithms + Data Structures = Programs ", Prentice-Hall (1976)