報告番号 甲第 2235 号

代数的手法に基づく プログラミング言語の仕様記述法と プログラム検証に関する研究

北 英彦

目次

第1章　序論

近年，ソフトウェアの生産性，信頼性の向上が切実に要求されている．これに応えるためには，実用的なプログラム開発方法論，管理手法，プログラミングツール，ソフトウェア開発環境などの研究開発および整備が必要であることはもちろんであるが，それらの理論的根拠を支え，また，指針を与えるソフトウェアの基礎的な理論研究も重要である．

プログラミング言語の形式的仕様記述法および形式的意味論の研究は，ソフトウェア基礎論の一つであり，プログラム検証，プログラム合成，プログラム変換，コンパイラの正当性検証，コンパイラ自動生成などソフトウェアの生産性，信頼性を向上させるために現在進められている研究の基礎を提供する．

プログラミング言語の仕様記述法において，言語の仕様は，通常，シンタクスの記述とセマンティクスの記述に分けられる．シンタクスの記述に関しては，文脈自由言語とその構文解析などに関する多くの研究がなされ，実用的な字句解析プログラムや構文解析プログラムが，それらの形式的仕様から自動的に生成できる程に実用的な段階に達している．しかし，セマンティクスの記述に関しては，最近，種々の研究を通して次第に理解が深められているとはいえ，シンタクスに比べれば全く不十分であって，プログラミング言語のセマンティクスの形式的仕様記述は現在の重要な研究課題である．

プログラミング言語の仕様記述法としては，操作的仕様記述法（Lucas, Walk 71），属性文法（Knuth 68），公理的仕様記述法（Hoare 69），公理的仕様記述法（Hoare 69），表示的仕様記述法（Tennent 76），（中島 82），代数的仕様記述法（ADJ 81），

（Mosses 80）などがある．

操作的仕様記述法は，抽象機械（特に，状態遷移機械）を想定し，その命令系列によってプログラムの意味を記述する方法である．そのため，言語の意味要素の領域（意味領域）はその抽象機械となり，プログラミング言語の意味をその機械と独立に与えることができない．

属性文法は，文脈自由文法の非終端記号に属性を持たせ，構文解析の際にその値を計算する方法で，コンパイラ生成に適しており，Ｐａｓｃａｌ などの仕様記述に用いられたり，属性文法に基づくコンパイラ生成系もいくつか作られている．しかし，もともと意味について形式的に仕様記述することは考えていない．

公理的仕様記述法は，プログラム正当性検証のための公理的検証体系をプログラミング言語を規定するものとして考え，それを仕様記述法とみなしたものであり，公理的な論理体系（公理と推論規則）によって与えられる．当然，プログラム検証を主な目的としている．この方法の特色は，意味の記述の抽象化のレベルが高いことで，そのため，プログラム検証には適しているが，コンパイラなどプログラミング言語の処理系の実現に適しているとはいえない．

表示的仕様記述法は，言語の構成要素に数学的対象物ないしは実在（関数，集合，関数空間）を対応させることで意味を与える方法である．この方法では意味領域は，領域方程式という領域上の再帰方程式の解として与えられる．

代数的仕様記述法は，構文要素の領域（構文領域）と意味領域を多ソート代数（抽象データ型）として捉え，プログラムの意味をこの二つの代数間の準同型写像（意味写像）によって与える方法である．本方法の特徴は，意味領域を抽象データ型として代数的手法によって記述する点である．抽象データ型は，モジュール性，抽象化，情報隠ぺい，局所化というプログラミング方法論を実現する概念で

あり，その代数的仕様記述は，形式性，記述性，理解性を持ち，また，必要最小限の情報しか記述しないという特徴を持つ．また，意味論が等式論理に基づいており，他の仕様記述法に比べて理解がしやすい．

本論文では，このプログラミング言語の代数的仕様記述法に関して考察を行なう．第2章では，プログラミング言語の代数的仕様記述法を提案する．本論文で提案する代数的仕様記述法の特徴は，意味領域の仕様を等式のみを用いて与え，代数的仕様記述法の枠組みの中でその意味が厳密に定まるようにしたことである．さらに，第2章では，本方法による仕様記述から意味が一意に定まること，すなわち，意味写像が一意に存在することを証明するとともに，意味写像の性質を明らかにして，プログラミング言語の仕様からコンパイラを自動生成する方法の基礎を明らかにする．

第3章では，第2章において提案したプログラミング言語の代数的仕様記述法の有効性を実証することを目的として，Ｐａｓｃａｌ風言語 ＰＬ／０＋ の代数的仕様記述を与え，その説明を通して，Ｐａｓｃａｌ 風言語一般に対する代数的仕様記述の技法を明らかにする．

第4章および第5章では，代数的仕様記述法を確立する目的の一つであるプログラム検証について考察する．第4章では，プログラム検証法としてよく知られているホーア流の公理的検証体系と代数的意味論との関係を明らかにする．そのために，代数的意味論に対する公理的検証体系の健全性について考察する．具体的には，ＷＨＩＬＥ プログラムを記述する簡単な言語 ＷＬ に対して，代数的意味論に関する公理的検証体系の健全性を示す．これにより，代数的意味論と公理的検証体系の間に矛盾のないこと，代数的意味論の与えられたプログラミング言語で書かれたプログラムの検証に公理的検証体系を用いてよいことがわかる．また，すでに広く認められて

いるホーア流の公理的検証体系との無矛盾性は，代数的仕様の正しさを相対的に保証する．

第5章では，代数的意味論の枠組の中でプログラムの正当性検証を行うための基本的な考え方について述べる．即ち，プログラムの意味を入力から出力への関数と考えたプログラミング言語の代数的仕様記述法，ならびに，プログラムの機能の代数的仕様記述法を与え，それに基づいてプログラムの正当性の定義を与える．言語の代数的意味論を基礎にしているため，検証の（半）自動化に項書き換え系を直接利用できる．

最後に付録として，第3章で代数的仕様記述の説明に用いたプログラミング言語 ＰＬ／０＋ の全仕様を載せる．この仕様は言語開発支援システム Ｌａｓｓ （Ｌ Anguage development Ｓ upporting Ｓ ystem）（酒井，北，坂部，稲垣 85）の書式を用いて記述されている．

第２章　プログラミング言語の代数的仕様記述法

２．１　はじめに

プログラミング言語の形式的仕様記述法の研究目的は，プログラミング言語の形式的意味論を確立し，プログラム検証の数学的基礎を与えると共に，コンパイラの正当性の証明，コンパイラの自動生成を可能にすることである．

プログラミング言語の仕様は，通常，シンタクスの記述とセマンティクスの記述に分けられる．シンタクスの記述に関しては，文脈自由言語とその構文解析などに関する多くの研究がなされ，実用的な字句解析プログラムや構文解析プログラムがそれらの形式的仕様から自動的に生成できる程に実用的な段階に達している．

しかし，セマンティクスの記述に関しては，最近，種々の研究を通して次第に理解が深められているとはいえ，シンタクスに比べれば全く不十分であって，プログラミング言語のセマンティクスの形式的仕様記述ならびにそれからのコンパイラ自動生成は現在の重要な研究課題である．

属性文法，公理論的意味論，操作的意味論，表示的意味論など種々の意味論が研究されている中で，最近になって，ADJ グループ（ADJ 81）や P.Mosses（Mosses 80）に始まる代数的仕様記述法が注目されている．これは，構文領域ならびに意味領域をそれぞれ抽象データ型，すなわち，代数として捉え，それらの間の写像（意味写像）を与えることによって意味を記述する方法である．現在まで

に，ADJ グループと Mosses の他にも，必ずしも意味写像を用いない方法も含めて，（Gaudel 80），（Goguen 80），（Goguen, Parsaye-Ghomi 81），（Pair 82），（Despryroux 83）など多くの代数的仕様記述法が提案されている．これらの中で，Mosses の方法は，意味写像と意味領域の実現との合成としてコンパイラを捉えることができる点でコンパイラの自動生成を考える場合には便利な考え方である．しかし，Mosses は，構文領域，意味領域，ターゲット言語はいずれも抽象データ型であると考えてはいるが，意味領域 SD を等式を用いて記述するのに SD に属さない演算を用いているなど，意味領域が厳密な代数的手法で記述されていない．例えば，ループ文の意味を記述するための不動点演算子 fix を SD に導入するために，構文上の代入操作（syntactic substitution）の概念を用いている．これは，代数的仕様記述法としての全体の枠組みを破壊するものであり，代数的仕様記述法の分かり易さを損なうばかりでなく，抽象データ型の代数的仕様記述法の理論の成果を直接に用いることを困難にしている．

そこで，著者は，意味写像と意味領域の実現との合成としてコンパイラを捉えられるような枠組みとして，Mosses の考え方は採用するが，上記の困難を克服すべく厳密に代数的手法に基づくプログラミング言語の仕様記述法を確立することを目的として種々の検討を行なった．本論文では，意味領域を抽象データ型と考えその仕様を等式のみを用いて与え，代数的仕様記述法の枠組みの中でその意味が厳密に定まるようなプログラミング言語の仕様記述法を提案する．さらに，本方法による仕様記述から意味が一意に定まること，すなわち，意味写像が一意に存在することを証明するとともに，意味写像の性質を明らかにして，プログラミング言語の仕様からコンパイラを自動生成する方法の基礎を明らかにする．まず，**2．2** では，本論文で用いる抽象データ型とその代数的仕様記述法に関する基本

概念，記法を説明する．2．3から2．6の4つの節で本論文で提案するプログラミング言語の代数的仕様記述法について述べる．2．3で本仕様記述法の枠組みを定め，2．4で構文領域の，2．5で意味領域の，2．6で意味写像の仕様記述法を与える．それと同時に，理解の助けのために，簡単ではあるがループ文を含むプログラミング言語ＳＬを例にとり，2．4，2．5，2．6の各節で各部分の仕様記述を与え，仕様記述法を具体的に説明する．2．7では，本方法で記述された意味写像が意味方程式の解として一意に定まることを示すと共に，その意味写像が，構文領域から，意味領域の演算記号で構成される項集合への写像 δ' とその項を意味領域上で評価する評価写像 eval の合成で表されることを示す．

これらの議論は，Mosses のように不動点演算子 fix という演算を導入することなく，すべて抽象データ型の代数的仕様記述法の枠組の中で取り扱われており，そのため，本論文で提案する方法は，記述性，読解性に優れているばかりでなく十分な抽象性と形式性を備えた仕様記述法となっている．

2．2　抽象データ型の代数的仕様記述に関する記法

本論文では，抽象データ型とその代数的仕様記述法に関する諸概念を用いる．本節ではその基本となる多ソート代数の基本概念の定義と記法のみを記す．それらの詳細については，（稲垣，坂部 84）に詳しい解説があるのでそれを参照されたい．

〔定義2．2．1〕　シグニチャ Σ は，ソートの集合 S と S 上の演算記号領域 F との対 $\langle S, F\rangle$ である．ここで，F は集合族 $\langle F_{w,s}\rangle_{w\in S^*, s\in S}$ であり，S^* は S の元から生成されるすべての有限系列の集合である．ただし，$w\neq w'$ または $s\neq s'$ ならば，$F_{w,s}\cap F_{w',s'}=\phi$（空集合）とする．また，$f\in F_{w,s}$ のとき，$\mathrm{arity}(f)=w$，$\mathrm{sort}(f)=s$ と書き，f をアリティ w，ソート s の演算記号であるという．また，$f\in F_{s1\ldots sn,s}$ のとき，$f:s1,\cdots,sn\rightarrow s$ とも記す．特に，$w=\varepsilon$（空系列）のとき，f は定数記号と呼ばれる．■

〔定義2．2．2〕　$\Sigma=\langle S, F\rangle$ をシグニチャとする．このとき，Σ 代数 A は次のものからなる．

（1）各ソート $s\in S$ に対する空でない集合 A_s．各 $s\in S$ に対して，A_s はソート s の台と呼ばれる．

（2）各演算記号 $f\in F_{s1\ldots sn,s}$（$s1,\cdots,sn,s\in S$，$n>0$）に対する関数 $f_A:A_{s1}\times\cdots\times A_{sn}\rightarrow A_s$．

（3）各定数記号 $f\in F_{\varepsilon,s}$ に対する A_s の元 f_A．

Σ 代数は，しばしば，単に，代数と呼ばれる．■

〔定義2．2．3〕　シグニチャを $\Sigma=\langle S, F\rangle$，変数集合族を $\langle X_s\rangle_{s\in S}$ とする．このとき，$T[\Sigma(X)]$ を次の条件を満たす最小の集合族 $\langle T[\Sigma(X)]_s\rangle_{s\in S}$ とする．

（1）$F_{\varepsilon,s}\cup X_s\subset T[\Sigma(X)]_s$．

（2）$f\in F_{s1\ldots sn,s}$，$t_i\in T[\Sigma(X)]_{si}$（$i=1,\cdots,n$）ならば，$f(t_1,\cdots,t_n)\in T[\Sigma(X)]_s$．

この $T[\Sigma(X)]_s$ の要素をソート s の $\Sigma(X)$ 項，あるい

は，単に，項という．特に，$X = \phi$ のとき，単に，Σ 項と言う．Σ 項の集合族を $T[\Sigma]$ と書く． ■

〔定義2．2．4〕　$\Sigma = \langle S, F \rangle$ をシグニチャとする．Σ 代数 $T = T[\Sigma]$ を次のように定め，Σ 項代数，あるいは，単に，項代数と呼ぶ．

（1）各ソート $s \in S$ に対する台を $T_s = T[\Sigma]_s$ とする．

（2）$f \in F_{\varepsilon,s}$ に対して，$f_T = f$．

（3）$f \in F_{s1...sn,s}$，$t_i \in T_{si}$ $(i = 1, \cdots, n)$ に対して，

$$f_T(t1, \cdots, tn) = f(t1, \cdots, tn).$$ ■

〔定義2．2．5〕　$\Sigma = \langle S, F \rangle$ をシグニチャ，A を Σ 代数とする．このとき，次の条件を満たす関係の族 $\equiv = \langle \equiv_s \subset A_s \times A_s \rangle_{s \in S}$ を A 上の合同関係という．

（1）任意のソート $s \in S$ について，$\equiv_s$ は A_s 上の同値関係である．

（2）任意の $f \in F_{s1...sn,s}$，$a_i, b_i \in A_{si}$ $(i = 1, \cdots, n)$ について，$a_i \equiv_{si} b_i$ ならば，

$$f(a_1, \cdots, a_n) \equiv_s f(b_1, \cdots, b_n).$$

$\equiv$ を合同関係としたとき，任意の $a \in A_s$ について，a の同値類 $\equiv_s[a]$ を，$\equiv_s[a] = \{b \in A_s \mid a \equiv_s b\}$ と定める．文脈から $\equiv_s$ が自明のときには，$\equiv_s[a]$ を単に $[a]$ と書く． ■

〔定義2．2．6〕　$\Sigma = \langle S, F \rangle$ をシグニチャ，A を Σ 代数，$\equiv$ を A 上の合同関係とする．このとき，$\equiv$ による A の

商代数 $A/\equiv$ を次のような Σ 代数とする.

（1）各ソート $s \in S$ に対して,

$$(A/\equiv)_s = \{ [a] \mid a \in A_s \}.$$

（2）各演算記号 $f \in F_{s1...sn,s}$ に対して,

$$f([a_1], \cdots, [a_n]) = [f(a_1, \cdots, a_n)].$$

ここで, $[a_i] \in (A/\equiv)_{si}$ $(i = 1, \cdots, n)$ である. ■

〔定義2．2．7〕 $\Sigma = \langle S, F \rangle$ をシグニチャとする. 任意の Σ 代数 A, 任意の変数集合族 $Y = \langle Y_s \rangle_{s \in S}$ に対して, $\theta = \langle \theta_s : Y_s \to A_s \rangle_{s \in S}$ を Y の A 上での割当という.

θ の定義域を $\Sigma(Y)$ 項に自然に拡張して得られる写像族

$$\theta' = \langle \theta'_s : T[\Sigma(Y)]_s \to A_s \rangle_{s \in S}$$

を次のように定める.

すなわち, $\xi \in T[\Sigma(Y)]_s$ に対して,

$$\theta'_s(\xi) = \begin{cases} \theta_s(\xi) & (\xi = v \in Y_s \text{のとき}) \\ f_A & (\xi = f \in F_{\varepsilon,s} \text{のとき}) \\ f_A(\theta'_{s1}(\xi 1), \cdots, \theta'_{sn}(\xi n)) & (\xi = f(\xi 1, \cdots, \xi n) \text{かつ} \\ & \quad \mathrm{arity}(f) = s1 \cdots sn \\ & \quad \text{のとき}) \end{cases}$$

■

〔定義2．2．8〕 $\Sigma = \langle S, F \rangle$ をシグニチャ, A を Σ 代数とする. Σ 項 ξ を A 上で評価する写像族

$$\mathrm{eval}^A = \langle \mathrm{eval}^A_s : T[\Sigma]_s \to A_s \rangle_{s \in S}$$

を次のように定める. すなわち, $t \in T[\Sigma]_s$ に対して,

$$
\mathrm{eval}^{A}{}_{s}(t) = \begin{cases} f_A & (t = f \in F_{\varepsilon,s} \text{のとき}) \\ f_A(\mathrm{eval}^{A}{}_{s1}(t1), \cdots, \mathrm{eval}_{sn}(tn)) & (\xi = f(t1, \cdots, tn) \text{かつ} \\ & \mathrm{arity}(f) = s1 \cdots sn \\ & \text{のとき}) \end{cases}
$$

以下では，$\mathrm{eval}^{A}{}_{s}(t)$ を，単に t_A と書く． ■

2．3　プログラミング言語の代数的仕様記述法

言語の仕様は，一般に，構文の仕様と意味の仕様からなる．構文の仕様は字面としてのプログラムの集合（以下では，構文領域と呼ぶ）を記述するものである．意味の仕様では，意味を与える基礎となる領域（以下では，意味領域と呼ぶ）と字面のプログラムに意味を与える写像（以下では，意味写像と呼ぶ）の2つを記述する．

本論文で提案する代数的仕様記述法は，構文領域および意味領域を代数（抽象データ型）として，意味写像をこの2つの代数間の写像として記述する．具体的には，構文領域は文脈自由文法で，意味領域は抽象データ型の代数的仕様記述法で，意味写像は原始帰納的な意味方程式系で記述する．

〔定義２．３．１〕　言語の仕様は，３項組

〈ＳＰＥＣ＿ＳＹＮ，ＳＰＥＣ＿ＳＥＭ，ＳＰＥＣ＿ＭＡＰ〉

である．ここで，ＳＰＥＣ＿ＳＹＮは構文領域の仕様（文脈自由文法），ＳＰＥＣ＿ＳＥＭは意味領域の仕様（抽象データ型の仕様），ＳＰＥＣ＿ＭＡＰは意味写像の仕様（原始帰納的な意味方程式系）である．■

以下，**２．４**，**２．５**，**２．６**で，構文領域，意味領域，意味写像について，各々の仕様記述法について述べる．また，これら三つの部分の具体的イメージを得るために，簡単な言語ＳＬを例として，その仕様を各部分ごとに与える．

この言語ＳＬは，非負整数を扱い，演算は加減算だけ，文は代入文とループ文のみであるブロック構造を有する簡単な手続き型言語である．例えば，次のプログラム（１）は，言語ＳＬによるプログラム例である．

```
V0:=0;  V1:=2;  V2:=4;
while V1 > 0 do
    begin
        V1:=V1-1;
        V0:=V0+V2
    end.
```
（１）

この例は，以下の説明で引続き用いられる．

２．４　構文領域の仕様

Ａｌｇｏ１６０以来，プログラミング言語の構文の仕様記述には，文脈自由文法が普通に用いられるようになっている．本論文でも，構文領域の仕様は文脈自由文法で与えることにする．

〔定義２．４．１〕　構文領域の仕様は，文脈自由文法

$$SPEC\_SYN = \langle N,\ T,\ P,\ START \rangle$$

である†．ここで，Ｎは非終端記号の集合，Ｔは終端記号の集合，Ｐは生成規則の集合，ＳＴＡＲＴはＮの要素で開始記号である．生成規則は，$p: m \rightarrow \alpha$ とする．ここで，ｐは生成規則の名前，ｍはＮの要素，α は $(N \cup T)^*$ の要素である．■

（ADJ 77）のように，文脈自由文法をシグニチャとみなす．すなわち，非終端記号の集合Ｎをソートの集合とし，生成規則の集合Ｐから演算記号領域Ｐ'を次のように定める．

$$P' = \langle P'_{w,s} \rangle_{w \in N^*,\ s \in N}$$

$$P'_{w,s} = \{ p \mid p: m \rightarrow \alpha \in P,\ s = m,$$
$$w \text{ は } \alpha \text{ 中の非終端記号を取り出して得られる記号列} \}$$

（注†）言語処理系の作成など実用に際しては，どの文に対してもその構文木が一意に定まるという文法の無あいまい性が必要であるが，以下の議論では構文領域（構文木の集合）から意味領域への意味写像を考えるので，特に文法の無あいまい性を前提とする必要はない．

以下では，生成規則の集合 P と演算記号領域 P' を同一視することにする．このようして，文脈自由文法 SPEC_SYN＝〈N，T，P，START〉をシグニチャ SPEC_SYN＝〈N，P〉とみなすことにより，次の定義のように構文領域を SPEC_SYN 項代数として定めることができる．

〔定義２．４．２〕　文脈自由文法 SPEC_SYN を構文領域の仕様とする．このとき，構文領域はシグニチャ SPEC_SYN＝〈N，P〉の項代数（抽象データ型）T［SPEC_SYN］である．以下では，SPEC_SYN の定める構文領域を L（SPEC_SYN）で表す．ただし，項代数であることを明示するため，T［SPEC_SYN］と書くことがある．■

SPEC_SYN 項代数の台の要素は SPEC_SYN 項であり，それは生成規則を節点とする木とみなすことができる．また，その木は通常の導出木と１対１に対応する．すなわち，ソート m の台 $T[SPEC\_SYN]_m$ は，m を根とする導出木の集合と考えることもできる．以上の方法に基づいて，言語 SL の構文領域の仕様は，図２．１のように与えられる．

```
SL_SPEC_SYN = 〈N, T, P, START〉
  N = {PROGRAM, STATEMENT, STMLIST, EXPRESSION, CONDITION,
       IDENT, NUMBER}
  V = {., :=, while, do, begin, end, ;, +, >,
       V0, V1, V2, 0, 1, …, 9}
  P = {p1: PROGRAM → STMLIST .
       p2: STATEMENT → IDENT := EXPRESSION
       p3: STATEMENT → while CONDITION do STATEMENT
       p4: STATEMENT → begin STMLIST end
       p5: STMLIST → STMLIST ; STATEMENT
       p6: STMLIST → STATEMENT
       p7: EXPRESSION → TERM
       p8: EXPRESSION → TERM + TERM
       p9: EXPRESSION → TERM - TERM
       p10: TERM → IDENT
       p11: TERM → NUMBER
       p12: TERM → ( EXPRESSION )
       p13: CONDITION → EXPRESSION > EXPRESSION
       p14: IDENT → V0
       p15: IDENT → V1
       p16: IDENT → V2
       p17: NUMBER → 0
         :
       p26: NUMBER → 9
       p27: NUMBER → NUMBER 0
         :
       p36: NUMBER → NUMBER 9
     }
  START = PROGRAM
```

図２．１　プログラミング言語ＳＬの構文領域の仕様

Fig. 2.1 Specification of syntactic domain of a simple programming language SL

例えば，式（1）で与えた言語 SL のプログラムに対する SL_SPEC_SYN 項は次式（2）で表される．また，この式（2）を木で表現すると図2．2のようになり，通常の導出木（図2．3）と1対1に対応する．

```
p1(p5(p5(p5(p6(
  p2(p14(), p7(p11(p17()))),
  p2(p15(), p7(p11(p19()))),
  p2(p16(), p7(p11(p21()))),
  p3(p13(
        p7(p10(p15())),
        p7(p11(p17())),
      p4(p5(p6(
        p2(p15(),
          p9(p10(p15()),
                p11(p18()))),
        p2(p14(),
          p8(p10(p14()),
                p10(p16()))))))))
```

（2）

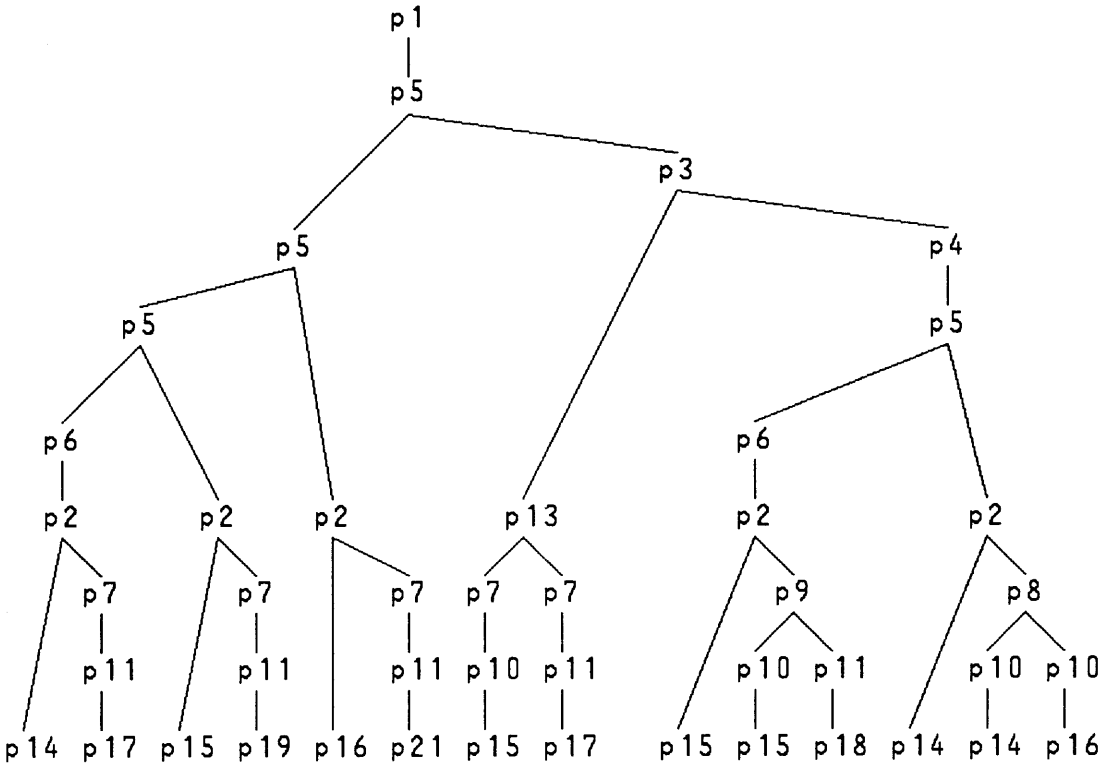

図２．２　式（２）のＳＬ＿ＳＹＮ＿ＳＰＥＣ項の木表現

Fig.2.2 Tree representation of
SL_SYN_SPEC-term of expression (2)

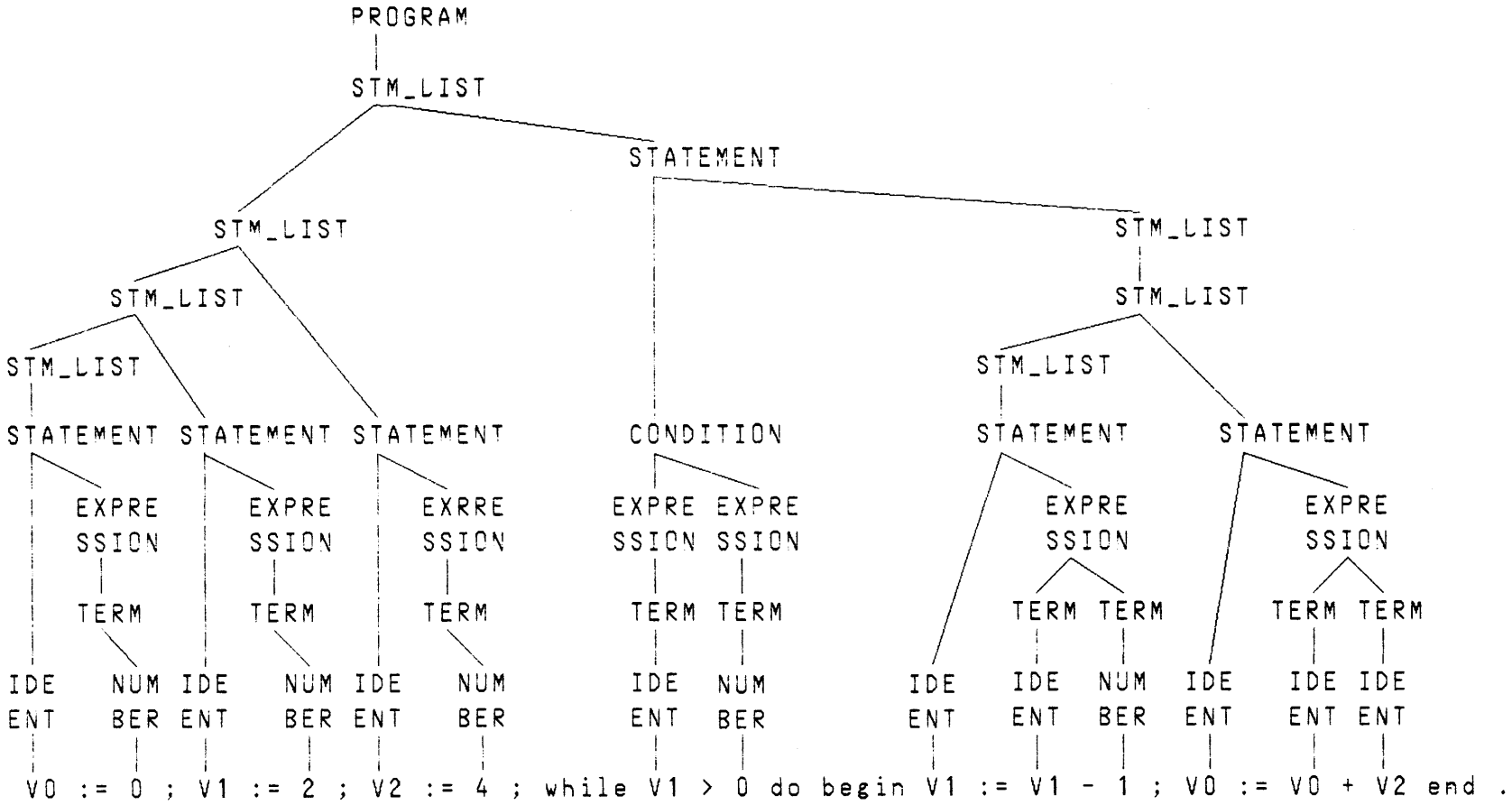

図２．３　式（２）のＳＬ＿ＳＰＥＣ＿ＳＹＮ項に対する導出木

Fig.2.3 Dervivation tree corresponded to SL_SPEC_SYN-term of expression (2).

２．５　意味領域の仕様

プログラミング言語の意味を厳密に定義するためには，意味領域を形式的に仕様記述する必要がある．本節では，意味領域を抽象データ型として捉え，代数的に仕様記述する．

抽象データ型の仕様は，一般に，演算記号の集合と演算間に成り立つ関係を記述した等式（公理）の集合によって与えられる．プログラミング言語の代数的意味論に関する論文の中には，（Mosses 80），（Gaudel 80）のように，仕様記述の能力を高めるために，公理を記述する際，等式以外の形式化が充分にされていない記述方法を用いている場合がある．例えば，Mosses は不動点演算子を定義するのに構文上の代入を，Gaudel は記号表中の変数の位置を割りつけるために副作用を持つ ＬＩＳＰ 関数を用いている．そのため，意味領域が数学的に厳密に定まらず，その実現等の議論を形式的に厳密な代数的理論の枠組みの中で扱うことを困難にしている．

本論文では，理論的明確さのためと形式的扱いに耐えるように，公理は等式のみによって記述し，意味領域を完全に代数的仕様記述法の理論の枠組の中で記述することにする．また，抽象データ型の仕様は，等式の集合で与えるため，意味領域で本来定められるべき演算の他に，公理を記述するために補助的な演算が必要となることが多い．本論文ではこの点を考慮し，意味領域の仕様には必須の演算と補助の演算を陽に分けて記述し，意味領域を本来必要である演算のみからなる抽象データ型となるように定める．

まず，意味領域の仕様記述法とそれが表す領域を形式的に定義する．

〔定義2．5．1〕　意味領域の仕様は，4項組

$$SPEC\_SEM = \langle \Sigma, \Sigma', V, A \rangle$$

である．ここで，Σ はシグニチャ $\langle S, F \rangle$，Σ はシグニチャ $\langle S', F' \rangle$，V は変数集合族 $\langle V_s \rangle_{s \in S}$，$A$ は公理の集合である．ただし，$S \subset S'$，すべての $w \in S^*$，$s \in S$ について，$F_{w,s} \subset F'_{w,s}$ とする．また，公理は，$\xi == \eta$ の形の等式とする．ここに，ξ，η は同一ソートの $\Sigma'(V)$ 項である．■

この定義で，Σ は意味領域が本来持つべきソートと演算記号を指定するもので，Σ に含まれない Σ' のソートと演算記号は補助のものである．このことは，仕様の表す意味領域が Σ' 代数でなく Σ 代数とすべきことを意味している．そこで，意味領域を次のように定める．

〔定義2．5．2〕　$SPEC\_SEM = \langle \Sigma, \Sigma', V, A \rangle$ を意味領域の仕様とする．仕様 $SPEC\_SEM$ の定める意味領域 $SD(SPEC\_SEM)$ は，項代数 $T[\Sigma]$ の合同関係 $\equiv^A$ による商代数 $T[\Sigma]/\equiv^A$ である．ここで，$\equiv^A$ は次のように定められる項代数 $T[\Sigma]$ 上の合同関係である．

任意の $t, t' \in T[\Sigma]_s$ について，

$$t \equiv^A_s t' \quad \text{iff} \quad A \vdash t == t'$$

ここで，$A \vdash t == t'$ は，等式 $t == t'$ が公理の集合 A から等式論理（稲垣，坂部 84）の推論規則を用いて導出されることを意味している．■

この合同関係 $\equiv^A$ は，意味領域の仕様，すなわち，等式の集合と矛盾しないものの中で，最も多くの情報を与えるもの，すなわち，項代数T［Σ］の最も細かい分割を与える合同関係として定義した．これは，抽象データ型の始代数意味論の考え方（稲垣，坂部 84）にそったものである．また，この定義の正当性，すなわち，$\equiv^A$ が合同関係であることは，その定めかたより容易に示される．

さて，上述の仕様記述法に基づいてどのように意味領域を代数的に仕様記述するかについて説明する．手続き型の言語の場合には，言語の各機能は計算機の内部状態の状態変化を引き起こす演算として考え，そのような演算を持つ抽象データ型として意味領域を定義する．このように，言語の機能を演算として捉え抽象データ型を用いて仕様記述すると，言語の意味を形式的で，かつ，明確にとらえることができる．

言語 SL の意味領域は図2．4のように記述される．これは，次のように考えて仕様記述されている．プログラムの意味は，プログラムの実行後の計算機の内部状態であり，また，代入文は，内部状態を別の内部状態に変化させる関数であると考える．そこで，STATE という計算機の内部状態を表すソートを設け，状態に変数を登録する ADD_ID，状態中の変数の値を更新する UPDATE，状態から変数の値を取り出す RETRIEVE という演算を定義する．また，状態から状態への関数の集合を表すソート STATE－STATE を設ける．ただし，抽象データ型では，汎関数が陽には扱えない，すなわち，関数を意味領域の要素とできないので，厳密にいうと，STATE－STATE は関数の表現の集合を表すソートである．この STATE－STATE をソートとして持つ演算として，UPDATE_D 等を考える．これらの演算によって構成される状態から状態への関数の表現の意味は，状態から状態の関数の表現を状態へ適用する演算 APPLY_S

ＴＡＴＥ との関係によって与える．例えは，次章で代入文の意味の記述に用いる ＵＰＤＡＴＥ＿Ｄ は，次の公理によって意味が与えられる．

```
ＡＰＰＬＹ＿ＳＴＡＴＥ（
    ＵＰＤＡＴＥ＿Ｄ（ｉｄ０，ｓｔａｔｅ－ｎａｔ０），
    ｓｔａｔｅ０）
  ＝＝ＵＰＤＡＴＥ（
          ｉｄ０，
          ＡＰＰＬＹ＿ＳＴＡＴＥ－ＮＡＴ（
                ｓｔａｔｅ－ｎａｔ０，
                ｓｔａｔｅ０））
```

ここで，ｉｄ０ は識別子の集合上の変数，ｓｔａｔｅ－ｎａｔ０ は状態から自然数への関数集合上の変数，ｓｔａｔｅ０ は状態集合上の変数，ＡＰＰＬＹ＿ＳＴＡＴＥ－ＮＡＴ は状態から自然数への関数の表現を状態に適用する演算である．

このように，ＵＰＤＡＴＥ＿Ｄ のような関数の表現を構成する演算と，関数の表現を関数と解釈してそれを引数に適用する ＡＰＰＬＹ＿ＳＴＡＴＥ という演算を用いることで，文の意味が状態から状態への関数として扱える．

なお，ＥＱ＿ＩＤ，ＩＦ＿ＩＮＴ 等の演算は，次節で与える言語ＳＬの意味写像の仕様中には現われない演算である．しかし，状態から変数の値を取り出す ＲＥＴＲＥＩＶＥ という演算を等式で記述するために必要となるので，補助の演算として定義してある．

```
SL_SPEC_SEM = ⟨Σ, Σ', V, A⟩
  Σ = ⟨S, F⟩
    S = {STATE-STATE, STATE-NAT, STATE-BOOL, STATE, ID, NAT}
    F = {
      APPLY_STATE: STATE-STATE, STATE → STATE
      COMP: STATE-STATE, STATE-STATE → STATE-STATE
      ITERATE: STATE-BOOL, STATE-STATE → STATE-STATE
      UPDATE_D: ID, STATE-NAT → STATE-STATE
      RETRIEVE_D: ID → STATE-NAT
      INIT_STATE: → STATE
      UPDATE, ADD_ID: STATE, ID, NAT → STATE
      RETRIEVE: STATE, ID → NAT
      MAKE_STATE-NAT: NAT → STATE-NAT
      SUM_D, DIFF_D: STATE-NAT, STATE-NAT → STATE-NAT
      GT_D: STATE-NAT, STATE-NAT → STATE-BOOL
      ID0, ID1, ID2: → ID
        :
    }
```

図２．４　プログラミング言語ＳＬの意味領域の仕様（１）

Fig. 2.4 Specification of semantic domain of a simple programming language SL (1)

$\Sigma' = \langle S', F' \rangle$

$S' = S \cup \{BOOL\}$

$F' = F \cup \{$

IF_NAT: BOOL, NAT, NAT → NAT

IF_STATE: BOOL, STATE, STATE → STATE

APPLY_STATE-NAT: STATE-NAT, STATE → NAT

APPLY_STATE-BOOL: STATE-BOOL, STATE → BOOL

EQ_ID: ID, ID → BOOL

$\}$

$V = \langle V_s \rangle_{s \in S}$

$V_{STATE-STATE} = \{st-st0, st-st1\}$

$V_{NAT} = \{n0, n1\}$

:

図４　プログラミング言語ＳＬの意味領域の仕様（２）

Fig.4 Specification of semantic domain of a simple programming language SL (2)

```
A = {
  APPLY_STATE(COMP(st-st0,  st-st1 ),  st0)
   == APPLY_STATE(st-st0,  APPLY_STATE(st-st1,  st0))
  APPLY_STATE(UPDATED(id0,  st-nat0),  st0)
   == UPDATE(st0,  id0,  APPLY_STATE-NAT(st-nat0,  st0))
  APPLY_STATE(ITERATE(st-b0,  st-st0),  st0)
   == IF_STATE(APPLY_STATE-BOOL(st-b0,  st0),
          APPLY_STATE(
            COMP(ITERATE(st-b0,  st-st0),  st-st0),  st0),
          st0)
  APPLY_STATE-NAT(SUM_D(st-nat0,  st-nat1),  st0)
   == SUM(APPLY_STATE-NAT(st-nat0,  st0),
           APPLY_STATE-NAT(st-nat1,  st0))
  APPLY_STATE-NAT(MAKE_STATE-NAT(n0),  st0) == n0
  APPLY_STATE-NAT(RETRIEVE_D(id0),  st0)
   == RETRIEVE(st0,  id0)
  RETRIEVE(INIT_STATE()),  id0) == ZERO()
  RETRIEVE(ADD_ID(st0,  id0,  n0),  id1)
   == IF_NAT(EQ_ID(id0,  id1),  n0,  RETRIEVE(st0,  id1))
  UPDATE(INIT_STATE(),  id0,  nat0)
   == ADD_ID(INIT_STATE(),  id0,  nat0)
  UPDATE(ADD_ID(state0,  id0,  nat0),  id1,  nat1)
   == IF_STATE(EQ_ID(id0,  id1),
                ADD_ID(state0,  id0,  nat1),
                ADD_ID(UPDATE(st0,  id1,  nat1),  id0,  nat0)
      :
}
```

図２．４　プログラミング言語ＳＬの意味領域の仕様（３）

Fig.2.4 Specification of semantic domain of a simple programming language SL (3)

２．６　意味写像の仕様

文脈自由文法の各生成規則に一つづつ原始帰納方程式を割り当てることで，意味写像の記述を行う．これは，構文領域から意味領域への意味写像を定める原始帰納方程式系となる．構文領域を項代数Ｔ［ＳＰＥＣ＿ＳＹＮ］，意味領域を商代数ＳＤ（ＳＰＥＣ＿ＳＥＭ）＝Ｔ［Σ］／$\equiv^{A}$と定めたから，この原始帰納方程式系の解は，後に証明されるように，Ｔ［ＳＰＥＣ＿ＳＹＮ］からδ（ＳＤ（ＳＰＥＣ＿ＳＥＭ））への一意な準同型写像として定められる．すなわち，前節で述べたように意味領域が厳密に代数として与えられたので，意味写像を自然に代数から代数への準同型写像として与えることができる．このように，構文領域と意味領域を二つの異った代数として厳密に区別し，その間の意味写像を準同型写像で与えるというように，完全に代数的な手法に基づいて意味の仕様記述ができるということは，本仕様記述法の特徴である．

さて，意味写像の仕様を定義する準備として，導出関数を導入する．シグニチャΣ＝〈Ｓ，Ｆ〉，変数集合族Ｙ＝$\langle Y_s \rangle_{s \in S}$に対して，ソートｓのΣ（Ｙ）項ξのΣ代数Ａ上の導出関数ξ_Aは次のように定義される．ξ中に出現する変数を出現する順に，ただし，二度目以降の出現は無視して並べて得られる変数列をｙ１，…，ｙｎ，各変数のｙｉのソートをｓｉとする．このとき，導出関数$\xi_A : A_{s1} \times \cdots \times A_{sn} \rightarrow A_s$は，任意の$a_i \in A_{si}$（ｉ＝１，…，ｎ）に対して，

$$\xi_A(a_1, \cdots, a_n) = \theta'(\xi)$$

と定められる．ここに，θ'は，$\theta_{si}(y_i) = a_i$（ｉ＝１，…，ｎ）で定義されるＹのＡ上の割当θのΣ（Ｙ）項への自然な拡張である．すなわち，$\xi_A(a_1, \cdots, a_n)$は，ξをその各変数ｙｉにａｉを代入して，Ａ上で評価して得られる値である．

この導出関数を用いて，意味写像の仕様は以下のように定義される．

〔定義2．6．1〕　SPEC_SYN＝〈N，T，P，START〉を構文領域の仕様，SPEC_SEM＝〈Σ，Σ'，V，A〉を意味領域の仕様とする．また，Σ＝〈S，F〉とする．このとき，意味写像の仕様は，4項組

SPEC_MAP〈D，M，X，R〉

である．ここで，Dは非終端記号に意味領域のソートを対応させる関数 $D: N \rightarrow S$ であり，Mはソート m の構文領域 $L(SPEC\_SYN)_m$ からソート D（m）の意味領域 $SD(SPEC\_SEM)_{D(m)}$ への関数の集合

$[L(SPEC\_SYN)_m \rightarrow SD(SPEC\_SEM)_{D(m)}]$

の上の関数変数 M_m の集合 $\{M_m \mid m \in N\}$，Xは構文領域上の変数集合族 $\langle X_m \rangle_{m \in N}$，Rは意味方程式の集合 $\{R_p \mid p \in P\}$ である．ただし，意味方程式 R_p（$p \in P_{m1...mn,m}$）は次の形とする．

$$M_m(p(x_1, \cdots, x_n)) = \xi[y_1 \leftarrow M_{m1}(x_1), \cdots, y_n \leftarrow M_{mn}(x_n)]$$

ここで，$x_i \in X_{mi}$（$i=1, \cdots, n$），$M_m, M_{mi} \in M$（$i=1, \cdots, n$）である．ξ は $p \in P_{m1...mn,m}$ に対応して定まるソート D（m）の Σ（Y）項であり，Yはソート D（mi）の相異なる変数 yi（i＝1，…，n）からなる変数集合族 $\langle Y_s \rangle_{s \in S}$ である．また，意味方程式の右辺

$$\xi[y_1 \leftarrow M_{m1}(x_1), \cdots, y_n \leftarrow M_{mn}(x_n)]$$

は，i＝1，…，nについて，Σ 項 ξ 中の変数 yi を M_{mi}（xi）で置き換えて得られる項である．■

この定義における意味方程式の集合は，関数変数の集合 M に関する原始帰納方程式系（Courcelle,Franchi-Zannetacci 82）になっており，**2．7**で示すようにこの方程式系は一意の解をもつ．意味写像の仕様の表す意味写像は，この方程式系の解として定められる．

〔定義2．6．2〕　SPEC_MAP＝〈D，M，X，R〉を意味写像の仕様とする．意味写像は次の条件を満足する写像族

$$SM(SPEC\_MAP) = \langle SM(SPEC\_MAP)_m : L(SPEC\_SYN)_m \rightarrow SD(SPEC\_SEM)_{D(m)} \rangle_{m \in N}$$

である．

任意の $m \in N$ に対して，その意味方程式

$$M_m(p(x1, \cdots, xn)) = \xi[y1 \leftarrow M_{m1}(x1), \cdots, yn \leftarrow M_{mn}(xn)]$$

と任意の $ti \in T[SPEC\_SYN]_{mi}$ $(i = 1, \cdots, n)$ について，

$$SM(SPEC\_MAP)_m(p(t1, \cdots, tn)) = \xi_{SD(SPEC\_SEM)}(SM(SPEC\_MAP)_{m1}(t1), \cdots, SM(SPEC\_SEM)_{mn}(tn)) \quad \blacksquare$$

以上で説明した意味写像の仕様の記述法に従うと，言語 SL の意味は次のように記述される．代入文の意味は，指定された変数の値を更新する状態から状態への関数として，意味領域の演算 UPDATE_D を用いて，次の意味方程式によって記述される．

```
(*  p2: STATEMENT
        → IDENT := EXPRESSION *)
    Mstatement(p2(ident0, exp0))
      = UPDATE_D(Mident(ident0),
                 Mexpression(exp0))
```

また，代入文の系列の意味は，関数合成 COMP を用いて，系列を構成する代入文の意味である状態から状態への関数の合成によって記述される．さらに，ループ文については，演算 ITERATE を用いて，次の意味方程式によって記述されている．

```
(*  p3: STATEMENT
        → while CONDITION do
                STATEMENT *)
    Mstatement(p3(cond0, stm0))
      = ITERATE(Mcondition(cond0),
                Mstatement(stm0))
```

ここで，ループ文の意味である状態から状態への関数

Mstatement(p3(cond0, stm0))

をある状態 st に適用すると，

```
APPLY_STATE(
  Mstatement(p3(cond0, stm0)), st)
```

であり，この式に，ループ文に関する意味方程式と意味領域の仕様（図２．４）中の演算 ITERATE に関する公理を適用すると次のようになる．

```
IF_STATE(
    APPLY_STATE-BOOL(
        MCONDITION(cond0), st),
    APPLY_STATE(
        COMP(
            MSTATEMENT(p3(cond0, stm0)),
            MSTATEMENT(stm0)),
        st),
    st)
```

この式は，次のことを表している．すなわち，ループ文の条件部の意味 $M_{CONDITION}$（cond0）が状態 st で真ならば，ループ文の意味 $M_{STATEMENT}$（p3（cond0，stm0））を状態 st に適用した結果の状態と，ループ文の本体の意味 $M_{STATEMENT}$（stm0）とループ文の意味 $M_{STATEMENT}$（p3（cond0，stm0））の関数合成を状態 st に適用した結果の状態が等しく，また，条件部の意味が偽ならば，ループ文の意味を状態 st に適用した結果の状態は st そのものである．これは，私たちが通常考えているループ文の意味と同じであり，ループ文の意味が等式のみを用いて記述できたことを示している．

以上のようにして，言語 SL の意味写像の仕様は図2．5のように与えられる．

$SL\_SPEC\_MAP = \langle D, M, X, R \rangle$

$D : N \rightarrow S$

$D(PROGRAM) = STATE$

$D(STATEMENT) = D(STMLIST) = STATE\text{-}STATE$

$D(EXPRESSION) = D(TERM) = STATE\text{-}NAT$

$D(CONDITION) = STATE\text{-}BOOL$

$D(IDENT) = ID$

$D(NUMBER) = NAT$

$M = \{M_{PROGRAM}, M_{STATEMENT}, M_{STMLIST}, M_{EXPRESSION}, M_{TERM}, M_{CONDITION}, M_{NUMBER}, M_{IDENT}\}$

$X = \langle X_s \rangle_{s \in S}$

$X_{STM\_LIST} = \{stml0\}, \cdots, X_{NUMBER} = \{num0\}$

$R = \{$

$M_{PROGRAM}(p1(stml0))$

$= APPLY\_STATE(M_{STM\_LIST}(stml0), INI\_STATE())$

$M_{STATEMENT}(p2(id0, exp0))$

$= UPDATE\_D(M_{IDENT}(id0), M_{EXPRESSION}(exp0))$

$M_{STATEMENT}(p3(cond0, stm0))$

$= ITERATE(M_{CONDITION}(cond0), M_{STATEMENT}(stm0))$

$M_{STATEMENT}(p4(stml0)) = M_{STM\_LIST}(stml0)$

$M_{STM\_LIST}(p5(stml0, stm0))$

$= COMP(M_{STATEMENT}(stm0), M_{STM\_LIST}(stml0))$

図２．５　プログラミング言語ＳＬの意味写像の仕様（１）

Fig. 2.5 Specification of semantic mapping of a simple programming language SL (1)

```
M_STMLIST(p6(stm0)) = M_STATEMENT(stm0)
M_EXPRESSION(p7(term0)) = M_TERM(term0)
M_EXPRESSION(p8(term0, term1))
 = SUM_D(M_TERM(term0), M_TERM(term1))
M_EXPRESSION(p9(term0, term1))
 = DIFF_D(M_TERM(term0), M_TERM(term1))
M_TERM(p10(id0)) = RETREIVE_D(M_IDENT(id0))
M_TERM(p11(num0)) = MAKE_STATE-NAT(M_NUMBER(num0))
M_TERM(p12(exp0)) = M_EXPRESSION(exp0)
M_CONDITION(p13(exp0, exp1))
 = GT_D(M_EXPRESSION(exp0), M_EXPRESSION(exp1))
M_IDENT(p14()) = ID0()
M_IDENT(p15()) = ID1()
M_IDENT(p16()) = ID2()
M_NUMBER(p17()) = ZERO()
:
M_NUMBER(p27(num0))
 = MULT(M_NUMBER(num0),
      SUC(SUC(SUC(SUC(SUC(SUC(
        SUC(SUC(SUC(SUC(ZERO()))))))))))))
:
}
```

図２．５　プログラミング言語ＳＬの意味写像の仕様（２）

Fig. 2.5 Specification of semantic mapping of a simple programming language SL (2)

以上で，言語ＳＬの仕様記述ができあがった．この意味記述のもとで，言語ＳＬのプログラムの意味は次のように定められる．例えば，**２．３**で例としてあげたプログラム（１）の意味は，式（２）のＳＬ＿ＳＰＥＣ＿ＳＹＮ項へ意味写像 $M_{PROGRAM}$ を適用することによって得られる．すなわち，その意味は，意味写像の仕様に書かれているの意味方程式に従って再帰的に計算される．

```
MPROGRAM（ｐ１（ｐ５（ｐ５（…），ｐ３（…））））
  ＝ＡＰＰＬＹ＿ＳＴＡＴＥ（
      MSTM_LIST（ｐ５（ｐ５（…），ｐ３（…））），
      ＩＮＩＴ＿ＳＴＡＴＥ（））
  ＝ＡＰＰＬＹ＿ＳＴＡＴＥ（
      ＣＯＭＰ（MSTATEMENT（ｐ３（…）），
                MSTM_LIST（ｐ５（…））），
        ＩＮＩＴ＿ＳＴＡＴＥ（））
                :
  ＝ＡＰＰＬＹ＿ＳＴＡＴＥ（
      ＣＯＭＰ（ＩＴＲＥＡＴＥ（…），ＣＯＭＰ（…）），
      ＩＮＩＴ＿ＳＴＡＴＥ（））
```

この最後の式は，次式と等しいことが意味領域の仕様の公理を使って導出できる．

```
ADD_ID(ADD_ID(ADD_ID(
  INIT_STATE(),
  ID0(), SUC(SUC(SUC(
            SUC(SUC(SUC(SUC(
              SUC(ZERO()))))))))),
  ID1(), ZERO()),
  ID2(), SUC(SUC(
            SUC(SUC(ZERO()))))
```

この式より，プログラム（１）の意味が正しく仕様記述されていることが分かる．

２．７　意味写像の性質

この章では，意味写像が一意に存在することを示し，また，意味写像の性質を調べるとともに，構文領域 L（SPEC_SYN），意味領域 SD（SPEC_SEM），Σ 項代数 T［Σ］の間の関係を明らかにする．そのための準備として，二つのシグニチャ間の関係を記述する道具としてデライバを定義する．

〔定義２．７．１〕　$\Sigma = \langle S, F \rangle$，$\Sigma'' = \langle S'', F'' \rangle$ をシグニチャとする．このとき，写像

$$f : S \rightarrow S''$$

と写像族

$$d = \langle d_{w,s} : F_{w,s} \rightarrow T[\Sigma''(Y^{w})]_{f(s)} \rangle_{w \in S^{*}, s \in S}$$

の対 $\delta = \langle f, d \rangle$ を，Σ から Σ'' へのデライバという．ここで，$Y^w = \langle Y^w{}_s \rangle_{s \in S}$ は，$w = s1 \cdots sn$ のとき，各 $i = 1, \cdots, n$ についてソート $f(si)$ の相異なる変数 yi を選び，ソートごとに分類して得られる変数集合族である． ■

次に，デライバ $\delta = \langle f, d \rangle$ によって，Σ'' 代数 A 上に定められる Σ 代数 $\delta(A)$ を次のように定義し，これを A 上の δ 導出代数，あるいは，導出代数と呼ぶ．

（1）各ソート $s \in S$ に対して，$\delta(A)_s = A_{f(s)}$．

（2）各演算記号 $f \in F_{w,s}$ に対して，

$$f_{\delta(A)} = (d_{w,s}(f))_A.$$

すなわち，関数 $f_{\delta(A)}$ は項 $d_{w,s}(f)$ の A 上の導出関数である．

さて，意味写像の仕様の定義2．6．1の意味方程式系 R は原始帰納方程式であるので，意味写像の仕様 SPEC_MAP = ⟨D, M, X, R⟩ より，構文領域のシグニチャ SPEC_SYN = ⟨V, P⟩ から意味領域のシグニチャ $\Sigma = \langle S, F \rangle$ へのデライバ $\delta = \langle f, d \rangle$ を次のように定めることができる．

（1）写像 $f : N \to S$ は，$D : N \to S$ とする．

（2）写像族

$$d = \langle d_{w,s} : P_{w,s} \to T[\Sigma(Y)]_{D(s)} \rangle_{w \in S^*, s \in S}$$

は，次のように定める．

生成規則 $p \in F_{m1 \ldots mn, m}$ に関する意味方程式

$$M_m(p(x1, \cdots, xn))$$
$$= \xi[y1 \leftarrow M_{m1}(x1), \cdots, yn \leftarrow M_{mn}(xn)]$$

に対して，

$$d_{m1 \ldots mn, m}(p) = \xi$$

このデライバを用いて，次のことが証明できる．

〔命題2．1〕　意味写像の仕様 SPEC_MAP＝〈D，M，X，R〉から上のようして定まるデライバを δ とする．すなわち，δ を構文領域のシグニチャ SPEC_SYN＝〈V，P〉から意味領域のシグニチャ Σ＝〈S，F〉へのデライバとする．このとき，意味写像 SM（SPEC_MAP）は，構文領域 L（SPEC_SYN）から意味領域 SD（SPEC_SEM）上の δ 導出代数 δ（SD（SPEC_SEM））への準同型写像である†．また，逆に，L（SPEC_SYN）から δ（SD（SPEC_SEM））への準同型写像は意味写像である．

（注†）シグニチャ Σ＝〈S，F〉の Σ 代数 A，B に対して，次の条件を満たす写像族 $h = \langle h_s : A_s \rightarrow B_s \rangle_{s \in S}$ を A から B への準同型写像という．

（1）Σ の任意の定数記号 $f \in F_{\varepsilon,s}$ に対して，

$$h_s(f_A) = f_B$$

（2）Σ の任意の演算記号 $f \in F_{s1 \ldots sn,s}$ と任意の $a_i \in A_{si}$（$i = 1, \cdots, n$）に対して，

$$h_s(f_A(a_1, \cdots, a_n)) = f_B(h_{s1}(a_1), \cdots, h_{sn}(a_n))$$

（証明）　まず，意味写像が L（SPEC_SYN）から δ（SD（SPEC_SEM））への準同型写像であることを示す．

意味写像 SM（SPEC_MAP）の定義より，

任意の $p \in P_{m1...mn,m}$，$t_i \in L(SPEC\_SYN)_{mi}$（$i = 1, \cdots, n$）に対して，

$$SM(SPEC\_MAP)_m(p(t_1, \cdots, t_n)) = \xi_{SD(SPEC\_SEM)}(SM(SPEC\_MAP)_{m1}(t_1), \cdots, SM(SPEC\_MAP)_{mn}(t_n))$$

である．デライバ $\delta = \langle f, d \rangle$ の定めかたより，

$$p_{\delta(SD(SPEC\_SEM))} = d_{m1...mn,m}(p)_{SD(SPEC\_SEM)} = \xi_{SD(SPEC\_SEM)}$$

また，項代数の定義から，

$$p_{T[SPEC\_SYN]}(t_1, \cdots, t_n) = p(t_1, \cdots, t_n)$$

ゆえに，

$$SM(SPEC\_MAP)_m(p(t_1, \cdots, t_n)) = p_{\delta(SD(SPEC\_SEM))}(SM(SPEC\_MAP)_{m1}(t_1), \cdots, SM(SPEC\_MAP)_{mn}(t_n))$$

よって，準同型写像の定義より，SM（SPEC_MAP）は構文領域 L（SPEC_SYN）から δ（SD（SPEC_SEM））への準同型写像である．また，逆に，構文領域 L（SPEC_SYN）から δ（SD（SPEC_SEM））への準同型写像が意味写像であることは，上の推論を逆にたどることによって証明できる．■

〔系2．2〕　意味写像SM（SPEC_MAP）は，一意に存在する．

（証明）　構文領域L（SPEC_SYN），すなわち，項代数T［SPEC_SYN］はSPEC_SYN代数のクラスの始代数であるので，任意のSPEC_SYN代数に対して，一意の準同型写像が存在する（稲垣，坂部84）．したがって，δ（SD（SPEC_SEM））はSPEC_SYN代数であるから，L（SPEC_SYN）からδ（SD（SPEC_SEM））へ準同型写像が一意に存在する．この一意の準同型写像は，命題1より，意味写像SM（SPEC_MAP）に一致する．したがって，意味写像SM（SPEC_MAP）は一意に存在すると結論される．

■

次に，意味写像の性質を述べるため，デライバを自然に拡張した写像族δ'を考える．δ＝〈f，d〉をシグニチャSPEC_SYNからシグニチャΣへのデライバとする．このとき，dの定義域をT［SPEC_SYN］に自然に拡張した写像族

$$\delta' = \langle \delta'_m : T[SPEC\_SYN]_m \rightarrow T[\Sigma]_{D(m)} \rangle_{m \in N}$$

を次のように定める．

任意の$p \in P_{w,m}$，$w = m1 \cdots mn$，$ti \in T[SPEC\_SYN]_{mi}$（$i = 1, \cdots, n$）に対して，

$$\delta'_m(p(t1, \cdots, tn))$$
$$= d_{w,m}(p)_{T[\Sigma]}(\delta'_{m1}(t1), \cdots, \delta'_{mn}(tn))$$

ここで，$d_{w,m}(p)_{T[\Sigma]}$は，Σ項$d_{w,m}(p)$のΣ項代数T［Σ］上の導出関数である．この写像族δ'を用いると次の命題が成立する．

〔命題2．3〕　写像族 δ' と評価写像 $eval^{SD(SPEC\_SEM)}$ の合成写像は，意味写像である．すなわち，次式が成り立つ．

$$SM(SPEC\_MAP) = eval^{SD(SPEC\_SEM)} \circ \delta'$$

（証明）　任意の $m \in N$, $p \in P_{w,m}$, $w = m1 \cdots mn$, $ti \in T[SPEC\_SYN]_{mi}$ $(i = 1, \cdots, n)$ に対して，意味写像の定義より，

$$eval^{SD(SPEC\_SEM)}{}_{D(m)} \circ \delta'_m(p(t1, \cdots, tn))$$
$$= eval^{SD(SPEC\_SEM)}{}_{D(m)}(\delta'(p(t1, \cdots, tn)))$$

δ' の定義より，

$$= eval^{SD(SPEC\_SEM)}{}_{D(m)}(d_{w,m}(p)_{T[\Sigma]}(\delta'_{m1}(t1), \cdots, \delta'_{mn}(tn)))$$

$d_{w,m}(p) = \xi$ とすると，

$$= eval^{SD(SPEC\_SEM)}{}_{D(m)}(\xi_{T[\Sigma]}(\delta'_{m1}(t1), \cdots, \delta'_{mn}(tn)))$$

eval の定義（定義2．2．6）より，

$$= \xi_{SD(SPEC\_SEM)}(eval^{SD(SPEC\_SEM)}{}_{D(m1)}(\delta'_{m1}(t1)), \cdots, eval^{SD(SPEC\_SEM)}{}_{D(mn)}(\delta'_{mn}(tn)))$$

故に，$eval^{SD(SPEC\_SEM)} \circ \delta'$ は，定義2．6．2により，意味写像である．■

これは，意味写像がプログラムである SPEC_SYN 項を中間表現とみなすことのできる Σ 項に変換する部分と，その Σ 項を意味写像上で評価する部分に分けられることを示している．すなわち，命題2は，構文領域 L（SPEC_SYN），意味領域 SD（SPEC_SEM），意味写像 SM（SPEC_MAP），Σ 項代数 T［Σ］が図2．6のような関係にあることを言っている．

また，図2．6にはこれらとコンパイラとの関係について明らかにするために，ターゲット言語として，抽象データ型（Σ 代数）TARGET が付け加えられている．$eval^{TARGET}$ は Σ 項を TARGET 上で評価する写像族である．この図は，コンパイラの概念を示す（Mosses 80）の図式を我々の代数的仕様記述法の枠組みの中で位置づけたものになっている．すなわち，この図2．6のように，コンパイラ compile は，意味写像と意味領域の実現の合成として表すことができる．すなわち，

compile＝implement○SM（SPEC_MAP）

である．そこで，

compile

＝implement○SM（SPEC_MAP）

＝$eval^{TARGET}$○δ'

であるので，コンパイラはプログラムである SPEC_SYN 項を，中間表現とみなすことができる Σ 項に変換する部分 δ' と，その Σ 項をターゲット言語 TARGET（抽象データ型）上で評価する $eval^{TARGET}$ との合成と考えることができる．したがって，図6の意味でのコンパイラは，言語の仕様記述から δ' を自動生成し（これは，直構文変換の手法を用いて容易に実現できる），また，意味領域の実現の記述から $eval^{TARGET}$ を自動生成し，それら二つを合成することによって自動的に生成することができる．

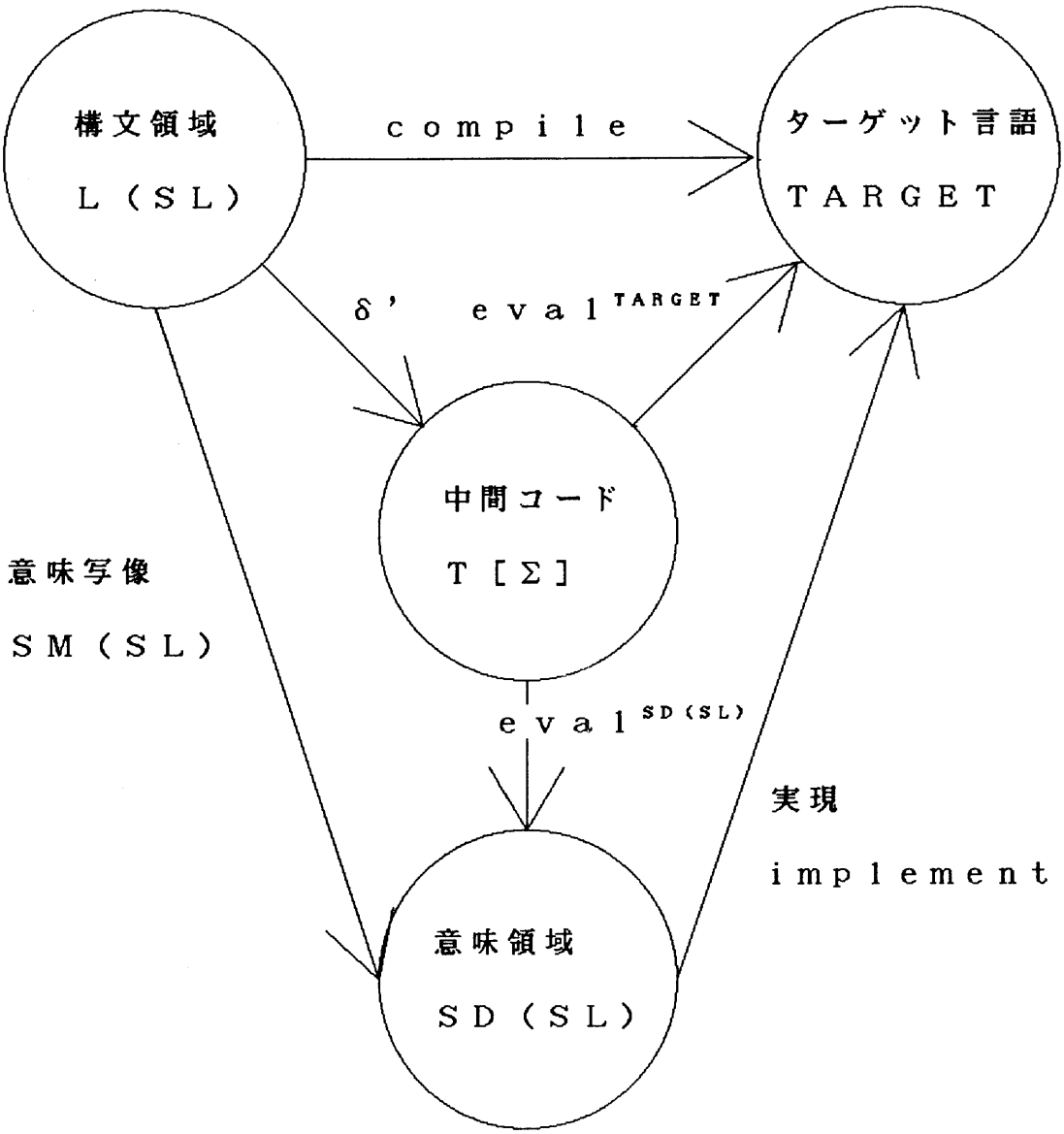

図２．６　プログラミング言語の意味論とそのコンパイラ
Fig.2.6 Semantics and compiler of programming language

2．8　まとめ

本章では，プログラミング言語の代数的仕様記述法を提案し，その意味論を明らかにするとともに，それを用いた例として，簡単な言語の仕様記述を試み，本方法の有効性を示した．さらに，意味写像が一意に定まることを証明すると共に，意味写像の性質を明らかにして，仕様記述からのコンパイラ自動生成法の基礎を明らかにした．

本方法は，意味領域を厳密に抽象データ型として捉え，意味写像を構文領域から意味領域への準同型写像で与えており，完全に代数的理論の枠組の中で言語の意味が記述されているので，目的言語ないしは機械とは独立に言語の仕様記述を可能にしている．そのために，記述性と読解性に優れているばかりでなく，十分な形式性を備えた仕様記述法になっており，これまでに抽象データ型の代数的仕様記述の理論の中で得られてきた成果を直接に利用することができるようになった．

本仕様記述法を用いてプログラミング言語の仕様をどのように記述するかについての説明は第3章で行なう．さらに，形式性の高い代数的仕様記述法が与えられたので，それに基づく代数的プログラム検証法の確立が期待されるが，これについては第5章で述べる．また，（酒井，北，坂部，稲垣 85）では，本章で与えたプログラミング言語の代数的仕様からコンパイラ自動生成するアイデアに基いて，言語開発支援システム Lass（LAnguage development Supporting System）を作成し，それが所期のとおり動作することが確認されている．

第３章　Ｐａｓｃａｌ風言語 ＰＬ／０＋の 代数的仕様記述

３．１　はじめに

本章では，第２章において提案したプログラミング言語の代数的仕様記述法の有効性を実証することを目的として，Ｐａｓｃａｌ風言語ＰＬ／０＋の代数的仕様記述を与え，その説明を通して，Ｐａｓｃａｌ風言語一般に対する代数的仕様記述の技法を明らかにする．なお，ＰＬ／０＋は，（Wirth 76）の中でコンパイラの作成の例に用いられた言語ＰＬ／０を拡張した言語である．すなわち，ＰＬ／０＋は，ＰＬ／０の持つ諸機能に加えて，配列，入出力，パラメタ付き手続きなどの機能を持ち，整数配列のソーティングのプログラムを自然に記述できる程度の実用性のある言語である．

著者の他にも，（Mosses 80），（Desperyrox 83），（Pair 82），（Goguen, Parsaye-Ghomi 81）などに代数的手法に基づくプログラミング言語の意味記述の試みが行われている．（Mosses 80）は，代数的仕様記述法の考え方を最初に提案したものである．しかし，意味領域が特殊な抽象データ型として記述されており理解性を損なっている．（Despeyroux 83）は，Ｐａｓｃａｌフルセットの記述を行っている．しかし，プログラムの状態を抽象データ型として，また，文の実行の意味をデータ型の変換として捉えているため，意味領域は抽象データ型ではなく抽象データ型のクラスとなり，理論的枠組が複雑になっている．（Pair 82）では，状態を文の実行の履歴

によって表現することを考え，プログラミング言語の各機能についてその記述法を示している．しかし，入出力機能の記述法については述べていない．（Goguen, Parsaye-Ghomi 81）は，著者やその他の文献（Mosses 80），（Desperyrox 83），（Pair 82）が採用した構文領域および意味領域を別々の抽象データ型とする考え方をとらず，プログラミング言語の構文と意味を合わせた全体を一つの抽象データ型と考えている．そして，比較的簡単ではあるが基本的な機能を備えた手続き型言語を仕様記述言語 ＯＢＪ（Goguen 80）に用いて記述しており，その仕様は項書き換え系による記号実行が可能である．

これらの仕様記述と比較して，本章で与える ＰＬ／０＋ 仕様の次のような特徴を持っている．第１は，第２章で与えた代数的仕様記述法に従って，意味領域を抽象データ型として純粋に等式のみを用いて記述したことであり，プログラムの正当性検証などの問題を代数的意味論の枠組の中で取り扱うことができる．第２は，プログラムの意味を入力から出力への関数として考え，それを抽象データ型を用いて記述したことである．すなわち，抽象データ型の代数的仕様記述では，関数をそのままデータとして扱えないので工夫が必要であり，本論文では，関数の表現と関数の表現を関数と解釈して引数に適用する関数によって記述を行っている．第３は，意味領域の仕様中の等式の集合が項書き換え系として記号実行ができるように記述したことであり，それに基づいて仕様記述された言語のテストを行うことができる．

以下，**3．2**では，第２章で与えた仕様記述法を簡便でかつ実用的なものにするために，階層化を行なったのでこれについて簡単に説明する．これは，識別子・数字列など基本的なものについては，予め構文と意味が与えられているとするアイデアに基づいている．そして，**3．3**で，この仕様記述法を用いてプログラミング言語

ＰＬ／０＋ の各機能をどのように仕様記述したかについて述べる.

３．２　代数的仕様記述法の階層化

文献（Despeyroux 83），（Pair 82）など代数的仕様記述法に関する文献の多くでは，識別子・数字列など基本的なものついては構文と意味が予め与えられていると仮定して議論をしている．これは，プログラミング言語の仕様の記述性・読解性を向上させるためには不可欠であるものであるが，上記の文献では形式的な取扱いがされていない.

本章では ＰＬ／０＋ の仕様を与えることを目的としているが，本節ではそのための準備として，識別子などの基本的なものに対して構文と意味を予め与えるという考え方を形式的に扱えるように，第２章で与えた仕様記述法を階層化する．以下では，識別子のように予め構文と意味が与えられているものを『原始』という接頭語によって表すことにする.

プログラミング言語の仕様は，構文領域，意味領域，意味写像の各々を規定する仕様の３項組

〈ＳＰＥＣ＿ＳＹＮ，ＳＰＥＣ＿ＳＥＭ，ＳＰＥＣ＿ＭＡＰ〉

である.

構文領域は，基本的には文脈自由文法で記述する．すなわち，構文領域の仕様 ＳＰＥＣ＿ＳＹＮ は，５項組

$$\langle T,\ N^{PRI},\ N,\ P,\ START \rangle$$

である．Ｔ は終端記号の集合である．N^{PRI} は原始非終端記号の集

合であり，各原始非終端記号には文字列の集合が予め与えられているとする．N は非終端記号の集合であり，$N^{PRI} \subset N$ とする．START $\in N - N^{PRI}$ は開始記号である．P は次の形の生成規則の集合である．

$$p : m \rightarrow t0\ m1\ t1\ m2 \cdots mn\ tn$$

ただし，p は生成規則の名前，t i（i = 1，…，n）は終端記号列，m i（i = 1，…n）は非終端記号である．

意味領域は抽象データ型として等式を用いて記述する．すなわち，意味領域の仕様 SPEC_SEM は，6項組

$$\langle S^{PRI},\ F^{PRI},\ S,\ F,\ V,\ A \rangle$$

である．S^{PRI}，S はソートの集合であり，$S^{PRI} \subset S$ とする．F^{PRI} は S^{PRI} 上の演算記号の集合，F は S の上の演算記号の集合であり，$F^{PRI} \subset F$ とする．また，F^{PRI} 中の演算記号には，原始意味領域上の演算として解釈が与えられているものとする．V は変数集合族 $\langle V_s \rangle_{s \in S}$ であり，A は F 中の演算記号と V 中の変数からなる項に関する等式の集合である．

意味写像は，基本的には各生成規則に意味方程式を一つ与えることで記述される．すなわち，意味写像の仕様 SPEC_MAP は，5項組

$$\langle D,\ M^{PRI},\ M^{NEW},\ X,\ R \rangle$$

である．D は非終端記号に意味領域のソートを割り当てる関数である．M^{PRI} は原始意味写像として与えられる原始構文領域から原始意味領域への意味関数の集合 $\{M_m \mid m \in N^{PRI}\}$ である．M^{NEW} は関数変数の集合 $\{M_m \mid m \in N - N^{PRI}\}$ であり，M_m は意味方程式によって定められる構文領域から意味領域への意味関数を表す

変数である．$X = \langle X_m \rangle_{m \in N}$ は意味方程式中で用いられる構文領域上の変数集合族，R は各生成規則規則に対する次のような方程式の集合である．生成規則

$$p : m \rightarrow t0\ m1\ t1\ m2 \cdots mn\ tn$$

に対する意味方程式は，

$$M_m(p(x1, \cdots, xn)) = \xi[y1 \leftarrow M_{m1}(x1), \cdots, yn \leftarrow M_{mn}(xn)]$$

の形の方程式である．ここで，$x\,i \in X\ (i = 1, \cdots, n)$，$\xi$ は変数 $y1, \cdots, yn$ を含むソート $D(m)$ の Σ 項であり，

$$\xi[y1 \leftarrow M_{m1}(x1), \cdots, yn \leftarrow M_{mn}(xn)]$$

は，各 $i = 1, \cdots, n$ に対して ξ 中の変数 $y\,i$ をすべて同時に $M_{mi}(x\,i)$ で置き換えて得られる項である．

3．3　プログラミング言語PL／0＋の仕様記述

本節では，**3．2**で述べた階層化した代数的仕様記述を用いて，プログラミング言語 PL／0＋ の仕様記述を行い，プログラミング言語の機能がどのように仕様記述されるかについて述べるとともに，代数的仕様記述法の記述性，読解性などの有効性を確かめる．

PL／0＋は，（Wirth 76）がコンパイラ構成法の説明の例に用いた言語 PL／0 を拡張した言語である．PL／0 は，単純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おり，整数上の計算手続きを表現することができる．PL／0＋ は，このPL／0 に入出力，一次元整数配列，パラメタ付き手続きなどの機能を加えた言語であり，整数配列のソーティングのプログラムが自

然に記述できる程度の実用性がある.

まず, ＰＬ／０＋ を概観するために, ＰＬ／０＋ の構文を定める構文領域の仕様 ＰＬ＿ＳＰＥＣ＿ＳＹＮ の一部を図３．１に示す.

次に, ＰＬ／０＋ の意味記述の基本的な考え方を以下に述べる. ＰＬ／０＋ プログラムでは, 状態が文の実行によって変化するので, 文の意味は状態から状態への関数と考える. また, 宣言の意味も, 状態に新しい識別子が登録されるので, 文と同様に状態から状態への関数と考える. 入出力については, 入力ファイル, 出力ファイルを考え, それぞれ特別の識別子で名付けられ状態に登録されているとする. そして, プログラムの意味は, 入力ファイルを初期化した状態でプログラムの本体である文の実行を開始し, プログラム実行後の最終状態から出力ファイルを得るもの, すなわち, 入力ファイルから出力ファイルへの関数として考える.

以下では, ＰＬ／０＋ の主要な機能について具体的にどのように代数的に記述したかを述べる. すなわち, **３．３．１**で状態の仕様記述, **３．３．２**でプログラム文の仕様記述, **３．３．３**でパラメタ付き手続きの仕様記述, **３．３．４**で入出力文の仕様記述, そして, 最後に, **３．３．５**で, ＰＬ／０＋ のプログラムの意味の仕様記述を与える. なお, 仕様記述の説明の中で, 非終端記号, ソート名, 生成規則名および演算記号は英小文字列で, 終端記号は英大文字列で, 変数記号は非終端記号またはソート名の後ろに数字列を付けたもので表すことにする. また, 自然数, 整数, ブール型および識別子については, 原始構文領域, 原始意味領域, 原始意味写像によって, 予め構文と意味が与えられているとする.

```
PL_SPEC_SYN=<T, N^PRI, N, P, START>
  T = {PROCEDURE, CALL, READ, WRITE, CONST,
       ARRAY, VAR, =, ., ;, :, :=, (, ), :=, [, ],
       IF, THEN, ELSE, BEGIN, END, WHILE, DO, EOF, <, ・・・}
  N^PRI = {nat_number, number, ident}
  N = N^PRI ∪
      {program, block, const_def_part, var_dcl_part,
       proc_dcl_part, statement, const_def_list, const_def,
       ident, number, var_name_list, var_name,
       para_sec_list, ・・・}
  P = {
    p010: program → block .
    p020: block → const_def_part var_dcl_part
                  proc_dcl_part statement
    p030: const_def_part → CONST const_def_list ;
    p080: const_def → ident = number
    p090: var_dcl_part → VAR var_naume_list ;
    p140: var_name → ident
    p145: var_name → ident : ARRAY
    p150: proc_dcl_part → proc_dcl_list
    p195: proc_dcl → PROCEDURE ident ( para_sec_list ) ;
                     block
    p204: para_sec → value_para_list
    p205: para_sec → VAR var_para_list
```

図3．1　構文領域の仕様（1）

Fig.3.1 Specification of syntactic domain (1)

```
  p210: value_para → ident
  p211: var_para → ident
  p220: statement → ident := expression
  p225: statement → ident [ expression ] := expression
  p235: statement → CALL ident ( call_para_list )
  p236: call_para → expression
  p240: statement → BEGIN statement_list END
  p255: statement → IF cond THEN statement ELSE statement
  p260: statement → WHILE cond DO statement
  p265: statement → READ ( read_para_list )
  p266: statement → WRITE ( write_para_list )
  p283: read_para → ident
  p284: read_para → ident [ expresiion ]
  p287: write_para → expression
  p300: statement_list → statement_list ; statement
  p315: cond → EOF
  p340: cond → expression < expression
  p400: expression → ident
  p405: expression → ident [ expression ]
   :
}
S T A R T = program
```

図3．1　構文領域の仕様（2）

Fig.3.1 Specification of syntactic domain (2)

３．３．１　状態

手続き型言語においては，プログラム中の識別子の持つ属性値は，プログラムの実行と共に変化する．そのため，手続き型言語の意味論を考える場合には，識別子とその属性値の対応関係を表すプログラムの状態というのが重要な概念である．（Pair 82）は，文の実行の系列によって状態を表現している．（Goguen, Parsaye-Ghomi 81）は，状態を環境，記憶入力ファイル，出力ファイルの４項組として考えている．（Despeyrux 83）は，識別子を右辺，その属性値を左辺とする等式，すなわち，『識別子＝＝属性値』を公理として含む代数的仕様により定まる抽象データ型を状態と考えている．このように，状態を何として考えるかには種々の見方があるが，ここでは，プログラムの状態（ソート ｓｔａｔｅ ）を識別子（ソート ｉｄ ）をキーとする表と考えて仕様記述を行う．

識別子の属性としては，基本的には，整数定数，整数変数，ファイル，一次元整数配列がある．このように識別子の属性は複数のデータ型をとりうるので，これら複数のデータ型の直和に当たる属性型（ソート ａｔｔｒ ）を考え，演算と等式によって記述する（図３．２）．例えば，ｍａｋｅ＿ａｔｔｒ＿ｃｏｎｓｔ は整数（ソート ｉｎｔ）を整数定数という属性型にする演算，ｍａｋｅ＿ｉｎｔ は整数定数あるいは整数変数という属性型を整数にする演算であり，それら二つの演算の関係を図３．２中の等式（１）で与える．その他の属性に関しても同様に記述する．ただし，属性型をファイル型にする演算 ｍａｋｅ＿ｆｉｌｅ を整数定数に適用する場合は，本来，その結果は未定義と考えられるが，ここでは，このように値が未定義となる場合には，記述を簡単にするために図３．２中の等式（２）のように適当な値を返すことにする．

```
OPERATIONS
  make_attr_const: int → attr
  make_attr_var: int → attr
  make_int: attr → int
  make_attr_file: file → attr
  make_file: attr → file
  make_attr_proc: state-state, para_list → attr
  make_state-state: attr → state-state
  make_para_list: attr → para_list
    :

AXIOMS
  make_int(make_attr_const(int0)) == int0                    (1)
  make_int(make_attr_var(int0)) == int0
  make_file(make_attr_const(int0)) == empty_file()           (2)
  make_state-state(make_attr_proc(state-state0, para_list0))
   == state-state0
  make_para_list(make_attr_proc(state-state0, para_list0))
   == para_list0
    :
```

図3．2　属性型の仕様記述

Fig.3.2 Specification of type of attributes

このように属性型を記述し，それに基づいて状態を記述する．基本的な演算としては，状態に新しく識別子を登録するａｄｄ＿ｉｄ，識別子の属性値を状態から取りだすｒｅｔｒｉｅｖｅ，識別子の属性が変数あるいは配列であるときにその属性値を更新するｕｐｄａｔｅを考える．これらの演算の意味の基本的な部分は，図３．３の中の等式（３），（４）によって与えられる．

さらに，ＰＬ／０＋は手続き内で局所変数の宣言，局所的な手続きの宣言が可能なので，それらを記述するために状態にブロック構造を設け，局所的に有効な識別子を扱えるようにする．演算は，状態に新しいブロックを設けるｅｎｔｅｒ＿ｂｌｏｃｋ，最新のブロックを取り除くｌｅａｖｅ＿ｂｌｏｃｋを設け，手続きが呼ばれたら局所的な識別子を宣言するために新しいブロックに入り，手続きが終了したらそのブロックを取り除くことができるようにする．

また，ＰＬ／０＋のようなブロック構造を持つプログラミング言語では，しばしば静的スコープルールが採用される．この静的スコープルール（Horowitz 83）とは，プログラム中に現れる識別子の属性を決定する次のような手続きのことである．最初に，識別子の使われている文を含む最も内側のブロック中にその識別子の宣言があるかを調べる．もしなければ，その探索を外側のブロックに向かって続けていく．途中でその識別子の宣言が見つかればその宣言で定義される属性を識別子の属性とする．最も外側のブロックに到達しても見つからなければエラーとする．この静的スコープルールの機能を記述するために，手続きが呼ばれたときその手続きが宣言されていたブロックを，現在のブロックからそのブロックまでのブロック数を用いて示す．すなわち，ｅｎｔｅｒ＿ｂｌｏｃｋの第１引数はそのような静的リンクを表すブロック数とする．これらの演算の意味は，図３．３の中の等式で与える．等式（５）で，変数ｉｎｔ０が１である場合には，静的リンクが直前のブロックを指し

ていることを意味し，状態 state0 に対して更新を行う．そうでない場合は，update_down_block を用いて静的リンクをたどってから更新を行う．update_down_blockの第1引数も，静的リンクを表すブロック数である．このようにすれば，識別子の有効範囲が静的スコープルールに従う場合にも，識別子の属性値の更新を等式を用いて記述できる．識別子の属性値を取り出す演算 retrieve に関しても同様な考え方で記述できる．

```
OPERATIONS
  init_state: → state
  add_id: state, id, attr → state
  update: state, id, attr → state
  retrieve: state, id → attr
  enter_block: int, state → state
  leave_block: state → state
  update_down_block: int, state, id, attr → state
    :

AXIOMS
  retrieve(add_id(state0, id0, attr0), id1)
   == if(equal_id(id0, id1),
         attr0,
         retrieve(state0, id1))                          (3)
  update(add_id(state0, id0, make_attr_var(int0)),
          id1,
          make_attr_var(int1))
   == if(equal_id(id0, id1),
          add_id(state0, id0, make_attr_var(int1)),
          add_id(update(state0, id1, make_attr_var(int1)),
                 id0,
                 make_attr_var(int0)))                   (4)
```

図3．3　プログラムの状態の仕様記述（1）

Fig.3.3 Specification of states of program (1)

```
leave_block(enter_block(int0, state0)) == state0
leave_block(add_id(state0, id0, attr0))
 == leave_block(state0)
update(enter_block(int0, state0), id0, attr0)
 == if(eq(int0, one()),
       enter_block(int0, update(state0, id0, attr0)),
       enter_block(
         int0,
         update_down_block(sub1(int0),
                           state0, id0, attr0)))
update_down_block(int1,
                  enter_block(int0, state0, id1, attr1))
 == if(eq(int1, one()),
       enter_block(int0, update(state0, id1, attr1)),
       enter_block(
         int0,
         update_down_block(sub1(int1),
                           state0, id1, attr1)))          (5)
update_down_block(int0,
                  add_id(state1, id1, attr1), id0, attr0)
 == add_id(update_down_block(int0, state1, id0, attr0),
           id1, attr1)
  :
```

図3．3　プログラムの状態の仕様記述（2）

Fig. 3.3 Specification of states of program (2)

３．３．２　文

次に，ＰＬ／０＋の文の仕様記述について説明する．ここでは，文，宣言の意味は，識別子の属性値を変更したり，新しい識別子を登録したりする状態から状態への関数であると考える．しかし，抽象データ型の代数的仕様記述では，関数をそのままデータとして扱えないので工夫が必要である．ここでは，関数の表現に当たるデータ型と関数の表現を関数と解釈して引数に適用する関数とを考え，それらを代数的に仕様記述するという技法を用いる．

まず，最も基本的な文である変数に対する代入文について述べる．この代入文の生成規則は，

ｐ２２０：ｓｔａｔｅｍｅｎｔ
　　→　ｉｄｅｎｔ　：＝　ｅｘｐｒｅｓｓｉｏｎ

であり，この代入文の意味は，演算ｕｐｄａｔｅ＿ｄを用いて与えられる．この演算は，状態から状態への関数の表現（ソートｓｔａｔｅ－ｓｔａｔｅ）を構成する演算であり，この演算の意味は，状態から状態への関数の表現を関数と解釈して状態に適用する演算ａｐｐｌｙ＿ｓｔａｔｅ－ｓｔａｔｅと状態から整数への関数の表現（ソートｓｔａｔｅ－ｉｎｔ）を関数と解釈して状態に適用する演算ａｐｐｌｙ＿ｓｔａｔｅ－ｉｎｔとを用いて，図３．４の中の等式（６）で与えられる．本論文の仕様記述の約束として，このｕｐｄａｔｅ＿ｄのような関数の表現を構成する演算の名前は，それが表現している演算の名前の後ろに，＿ｄという文字列を付ける．また，関数の表現を関数と解釈して引数に適用する演算の名前には，関数の表現のソートの前に，ａｐｐｌｙ＿という文字列を付ける，式ｅｘｐｒｅｓｓｉｏｎの意味は，状態が

定まると整数値が計算できるもの，すなわち，状態から整数への関数であるので，このｕｐｄａｔｅ＿ｄは，代入文の意味そのものであり，代入文の意味は図３．４の中の意味方程式（７）で与えられる．

このように，文を構成する構文要素に対応して，状態から状態への関数の表現を構成する演算を考え，状態から状態への関数の表現を状態に適用させる演算ａｐｐｌｙ＿ｓｔａｔｅ－ｓｔａｔｅを用いて意味を与える．そして，意味方程式を用いて，構文とその意味との対応を与える．

宣言についても，状態から状態への関数の表現として記述を行う．定数，変数，配列の宣言については，状態から状態への関数の表現を構成する演算ａｄｄ＿ｉｄ＿ｄを用いて仕様記述する．このａｄｄ＿ｉｄ＿ｄの第２引数は，識別子の登録の際の属性値の初期値である．例えば，整数変数宣言に関しては，図３．４の中の意味方程式（８）によって意味を与える．

```
OPERATIONS
  update_d: id, state-attr → state-state
  composition: state-state, state-state → state-state
  add_id_d: id, attr → state-state
  apply_state-state: state-state, state → state
  apply_state-int: state-int, state → int
    :

AXIOMS
  apply_state-state(update_d(id0, state-attr0), state0)
   == update(state0, id0,
             apply_state-attr(state-attr0, state0))        (6)
  apply_state-state(add_id_d(id0, attr0), state0)
   == add_id(state0, id0, attr0)
    :
```

図３．４　文の意味に関する仕様記述（１）

Fig.3.4 Specification of meaning of statements (1)

```
SEMANTIC FUNCTIONS
  M statement: statement → state-state
  M statement_list: statement_list → state-state
  M expression: expression → state-int
  M ident: ident → id
  M var_name: var_name → state-state
    :

SEMANTIC EQUATIONS
  M statement(p220(ident0, expression0))
   = update_d(M ident(ident0),
              make_attr_var_d(
                  M expression(expression0))                (7)
  M statement_list(p300(statement_list0, statment0))
   = composition(M statement(statement0),
                 M statement_list(statement_list0))        (8)
  M var_name(p140(ident0))
   = add_id_d(M ident(ident0), make_attr_var(zero()))
    :
```

図3．4　文の意味に関する仕様記述（2）

Fig.3.4 Specification of meaning of statements (2)

3．3．3＿パラメタ付き手続き

この節では，パラメタ付き手続きの機能の仕様記述について説明する．ＰＬ／０＋の手続きのパラメタには値パラメタと変数パラメタがある．値パラメタについては，手続き呼出しのとき与えられる実パラメタとして一般に式を書くことができ，これは手続き呼出しがなされたときに計算され，仮パラメタがその値を初期値とする局所変数として状態に登録される．これに対して，変数パラメタについては，実パラメタとして変数あるいは配列の要素のみを書くことができ，手続きの呼出しのときに，実パラメタの値でなくその状態における実パラメタの位置が渡され，仮パラメタは渡された実パラメタの場所を共有する．また，手続き内で局所変数や局所的な手続きを宣言することができる．

このようなパラメタ付き手続きを，以下のように仕様記述する．手続き宣言の生成規則は，

```
p195: proc_dcl
        → PROCEDURE ident
              ( para_sec_list ) ;
              block
```

であり．それに対する意味方程式は図３．５の中の方程式（９）である．ここで，ｂｌｏｃｋは，宣言の系列とそれに続く文からなる手続きの本体であり，ｐａｒａ＿ｓｅｃ＿ｌｉｓｔは，仮パラメタのリストで変数パラメタのリストと値パラメタのリストとを任意に結合したものである．また，ｍａｋｅ＿ａｔｔｒ＿ｐｒｏｃは，手続きの本体の意味と仮パラメタのリストの意味との対を属性型にする演算である．手続きの本体の意味は状態から状態への関数

の表現である.

手続き呼出しの生成規則は

```
p235: statement
        → CALL ident
                ( call_para_list )
```

であり，意味方程式は図3．5の中の方程式（10）で与えられる．手続き呼び出しの意味は，状態から状態への関数の表現であり，意味方程式（10）によって多数の関数の表現を構成する演算によって与えられる．

この手続き呼び出しの意味を説明するために，状態 state0 における実行を考える．これは，次の項で表される．

```
apply_state-state(
    Mstatement(
            p235(ident0,
                    call_para_list0)),
    state0)
```

そして，意味方程式と意味領域の仕様の中の公理よりこの式は，次の項と等しいことがわかる．

```
leave_block(
    apply_state-state(
        make_proc(
            retreive(
                state0,
                M_ident(ident0))),
        apply_state-state(
            add_id_list(
                state0, state0,
                make_para_list(
                    retreive(
                        state0,
                        M_ident(ident0))),
                M_call_para_list(
                    call_para_list0)),
            enter_block(
                plus1(
                    find_block(
                        M_ident(ident0),
                        state0)),
                state0))))
```

ここで，手続きの名前 ｉｄｅｎｔ０ の属性値は，手続きの本体の意味と仮パラメタのリストの対であり，ｍａｋｅ＿ｐｒｏｃ は，その属性値から手続き本体の意味，すなわち，状態から状態への関数の表現を取り出す演算，また，ｍａｋｅ＿ｐａｒａ＿ｌｉｓｔ は，仮パラメタのリストを取り出す演算である．ａｄｄ＿ｉｄ＿ｌｉｓｔ は，仮パラメタのリスト，すなわち，識別子のリストを状態に登録する演算である．このとき，仮パラメタが値パラメタなら，それに対する実パラメタである式を状態 ｓｔａｔｅ０ で評価し，その値を属性値とする．また，仮パラメタが変数パラメタなら，それに対する実パラメタの位置（ソート ｌｏｃ ）を属性値として登録する．実パラメータが登録されている位置は，実パラメタが変数のときはその識別子で，配列の要素のときは配列の識別子とその要素の指標の対で表す．

この手続きの実行を表す項は，次のように理解することができる．まず，ｆｉｎｄ＿ｂｌｏｃｋ でその手続きの宣言されていたブロックまでのブロック数を数え静的リンクを表す情報とする．そして，ｅｎｔｅｒ＿ｂｌｏｃｋ で新しいブロックに入り，ａｄｄ＿ｉｄ＿ｌｉｓｔ によって仮パラメタのリストを登録し，その状態に対して，ｍａｋｅ＿ｐｒｏｃ によって得られた手続きの本体である状態から状態への関数を適用し，その後で，ｌｅａｖｅ＿ｂｌｏｃｋ によりこの手続きのブロックを抜け出す．

```
OPERATIONS
  leave_block_d: state-state → state-state
  enter_block_d: state-int → state-state
  find_block: id, state → int
  add_id_list: state, state, para_list, state-int_list
                → state
  add_id_list_d: para_list, state-int_list → state-state
  make_var_para, make_value_para: id → para
  leave_n_block: int, state → state
    :


AXIOMS
  find_block(id1, add_id(state0, id0, attr0))
   == if(equal_id(id0, id1),
         zero(),
         find_block(id1, state0))
  find_block(id1, enter_block(id0, state0))
   == sum(int0,
        find_block(id0, leave_n_block(sub1(int0), state0)))
  leave_n_block(int0, state0)
   == if(eq(zero(), int0),
         state0,
         leave_n_block(sub1(int0), state0))
```

図3．5　手続き機能に関する仕様記述（1）

Fig.3.5 Specification of procedures (1)

```
add_id_list(state0,
      state1,
      cons_para_list(make_value_para(id0), para_list0),
      cons_state-int_list(state-int0, state-int_list0))
 == add_id_list(state0,
      add_id(state1, id0,
             make_attr_var(
                apply_state-int(state-int0, state0))),
      para_list0,
      state-int_list0)
add_id_list(state0,
      state1, nil_para_list(), nil_state-int_list())
 == state1
  :
```

SEMANTIC FUNCTIONS

M_{proc_dcl}: proc_dcl → state-state

$M_{call\_para\_list}$: call_para_list → state-int_list

$M_{para\_sec\_list}$: para_sec_list → para_list

M_{var_para}: var_para → para

M_{value_para}: value_para → para

:

図3．5　手続き機能に関する仕様記述（2）

Fig.3.5 Specification of procedures (2)

```
SEMANTIC EQUATIONS
  M proc_dcl(p195(ident0, para_sec_list0, block0))
   = add_id_d(
        M ident(ident0),
        make_attr_proc(M block(block0),
        M para_sec_list(para_sec_list0)))                    (9)
  M statement(p235(ident0, call_para_list0))
   = leave_block_d(
        apply_state_d(
          make_proc_d(retreive_d(M ident(ident0))),
          composition(
            add_id_list_d(
              make_para_list_d(retreive_d(M ident(ident0))),
              M call_para_list(call_para_list0)),
            enter_block_d(
              plus1_d(
                find_block_d(M ident(ident0)))))))           (10)
  M value_para(p210(ident0))
   = make_value_para(M ident(ident0))
  M var_para(p211(ident0)) = make_var_para(M ident(ident0))
```

図3．5　手続き機能に関する仕様記述（3）

Fig. 3. 5 Specification of procedures (3)

３．３．４　入出力機能

ＰＬ／０＋の入出力は，それぞれ，専用の入力ファイル，出力ファイルを考え，入力ファイルからのデータの読み込みはＲＥＡＤ文によって，出力ファイルへのデータの書込みはＷＲＩＴＥ文によって行う．また，入力ファイルの終わりにきているかどうかは，関数ＥＯＦを用いて検査する．

まず，意味領域中にファイル（ソートｆｉｌｅ）の仕様記述を行う．ファイルに関する演算としては，空のファイルを表すｅｍｐｔｙ＿ｆｉｌｅ，ファイルからデータを読みだすｒｅａｄ＿ｆｉｌｅ，ファイルにデータを書きだすｗｒｉｔｅ＿ｆｉｌｅ，ファイルの先頭のデータを削除するｒｅｍｏｖｅ＿ｄａｔａ，ファイルの終端点かどうかを検査するｅｏｆを考え，等式を用いて仕様記述する．

そして，入出力機能については，次のように考えて仕様記述を行う．入力ファイル，出力ファイルのために，それぞれ，プログラム中では使われない識別子を用意し，ファイルを属性としてそれらの識別子を状態に登録する．そして，ＲＥＡＤ文，ＷＲＩＴＥ文はそれらの識別子の属性値に対する操作と考え，その意味を状態から状態への関数として記述する．ＲＥＡＤ文は，引数で指定された識別子に対して，左から順番に，入力ファイルを表す識別子の属性値であるファイルからデータを読みだし，そのデータで識別子の属性値を更新する．ただし，ファイルからデータを読み込むたびに，先頭のデータを取り除いたファイルで，入力ファイルを表す識別子の属性値を更新する．このように，代数的仕様記述法では，データの読み込みの際におこる副作用を陽に記述する．また，ＷＲＩＴＥ文は，左から順番にその引数である式のリストを評価し，それらの値を順番に出力ファイルを表す識別子のファイルに書き込み，そのファイルで出力ファイルを表す識別子の属性値を更新する．さらに，

条件 ＥＯＦ は，入力ファイルの識別子の属性値であるファイルが空であるかどうかを調べる．

ＲＥＡＤ 文に関する生成規則は，図３．１中の ｐ２６５，ｐ２８３などで与えられる．各々の生成規則に対する意味方程式は，図３．６の中の方程式（１１），（１２）である．これらの意味方程式の中で，＊ｉｎｐｕｔ＊（）は，入力ファイルに対する識別子であり，生成規則 ｐ２８３ に対する意味方程式（１２）の右辺のｃｏｍｐｏｓｉｔｉｏｎ の第２引数が，引数の識別子に入力ファイルからデータを読み込み代入する部分を，第１引数が，データ読み込み後の入力ファイル更新の部分を表している．ＷＲＩＴＥ 文に関しても，同様な方法で仕様記述を行うことができる．

```
OPERATIONS
  read_file: file → int
  write_file: file, int → file
  empty_file: → file
  remove_data: file → file
    :


AXIOMS
  read_file(write_file(write_file(file0, int0)))
   == read_file(write_file(file0, int0))
  read_file(write_file(empty_file(), int0)) == int0
  remove_data(write_file(write_file(file0, int0)))
   == remove_data(write_file(file0, int0))
  remove_data(write_file(empty_file(), int0)) == file0
    :


SEMANTIC FUNCTIONS
  M read_para_list: read_para_list → state-state
  M read_para: read_para → state-state
  M write_para: write_para → id_list
    :
```

図3．6　入出力機能に関する仕様記述（1）

Fig.3.6 Specification of Input/Output functions (1)

```
SEMANTIC EQUATIIONS
  M statement(p265(read_para_list0))
   = M read_para_list(read_para_list0)                    (11)
  M read_para(p283(ident0))
   = M composition(
        update_d(
          *input*(),
          make_attr_var_d(
            remove_data_d(
              make_file_d(retrieve_d(*input*()))))),
        update_d(
          M ident(ident0),
          make_attr_var_d(
            read_file_d(
              make_file_d(retrieve_d(*input*()))))))      (12)
   :
```

図3．6　入出力機能に関する仕様記述（2）

Fig.3.6 Specification of Input/Output functions (2)

3．3．5　プログラム

最後に，ＰＬ／０＋のプログラム全体の意味の仕様記述について述べる．この章の最初に述べたように，ＰＬ／０＋のプログラムは，入力ファイルから出力ファイルへの関数として考えることができる．しかし，プログラムの本体は，宣言と文の系列からなり，この系列の意味は，状態から状態への関数の表現である．そこで，状態から状態への関数の表現からファイルからファイルへの関数の表現（ソート file－file）を構成する演算 program とプログラムの実行を行う演算 run を考え，図3．7の演算と等式によって記述する．等式（13）の中の ＊output＊（）は出力ファイルを表す識別子である．また，initialize は，与えられたファイルを属性値として入力ファイルの識別子を，空のファイルを属性値として出力ファイルの識別子を，空の状態に登録しプログラムの初期状態を構成する．

```
OPERATIONS
  program: state-state → file-file
  run: file-file, file → file
  initilize: file → state
  *input*, *output*: → id
    :

AXIOMS
  run(program(state-state0), file0)
   == apply_state-state(state-state0, initilize(file0))
  initilize(file0)
   == add_id(add_id(init_state(), *input*(),
                    make_attr_file(file0)),
        *output*(), make_attr_file(empty_file()))          (13)
    :

SEMANTIC FUNCTIONS
  M program: program → file-file
  M block: block → state-state
  M const_def_part: const_def_part → state-state
  M var_dcl_part: var_dcl_part → state-state
  M proc_dcl_part: proc_dcl_part → state-state
    :
```

図3．7　プログラムの意味に関する仕様記述（1）

Fig. 3.7 Specification of meaning of programs(1)

```
SEMANTIC EQUATIONS
  M program(p010(block0))
   = program(M block(block0))
  M block(
    p020(const_def_part0,
         var_dcl_part0,
         proc_dcl_part0,
         statement0))
   = composition(
       M const_def_part(const_def_part0),
       composition(
         M var_dcl_part(var_dcl_part0),
         composition(
           M proc_dcl_part(proc_dcl_part0),
           M statement(statement0))))
```

図3．7　プログラムの意味に関する仕様記述（2）

Fig. 3.7 Specification of meaning of programs(2)

３．３．６　検討

プログラミング言語 ＰＬ／０＋ の仕様記述を，３．２．１から３．２．５までに述べた考え方，技法を用いて行った．この仕様の主要な特徴は，以下の３点である．第１は，代数的仕様記述法に従って，意味領域を抽象データ型として純粋に等式のみを用いて記述したことである．これにより，代数的意味論の枠組の中で，コンパイラ自動生成，プログラムの正当性検証などの問題を形式的に取り扱うことができる．第２は，プログラムの意味を入力から出力への関数として考え，それを抽象データ型を用いて記述したことである．すなわち，抽象データ型の代数的仕様記述では，関数をそのままデータとして扱えないので工夫が必要であり，本論文では，関数の表現と関数の表現を関数と解釈して引数に適用する関数によって記述を行っている．第３は，意味領域を規定する等式の集合が項書き換え系（二木，外山 83）として，線形性，無あいまい性を持つように記述したことである．意味領域中の等式は左辺から右辺への書き換え規則と見ることができ，さらに，意味領域の仕様は項書き換え系と見ることができる．また，項書き換え系が線形性，無あいまい性の性質を持てば，書き換え結果が一意に定まる．このことに基づいて，項書き換え系を用いて意味領域の直接実現（記号実行）を行うことができ，さらに，その直接実現を用いて仕様記述された言語のインタプリタを自動生成できる．実際，言語処理系生成システム Ｌａｓｓ（酒井，北，坂部，稲垣 85）により，本章の仕様から ＰＬ／０＋ の処理系を生成し，ＰＬ／０＋ のプログラムを実行してみて，この仕様のテストを行い，仕様の妥当性を確認している．

ＰＬ／０＋ にはない機能のうち，ＧＯＴＯ 文のように制限のない制御の変更の記述には，Scott-Strachey 流の表示的意味論で用

いられた接続法（中島 82）に対応する概念を導入する必要があり，分かり易く記述するのは難しいと思われる．しかし，手続き型言語のその他の機能，例えば，型宣言，一般的なファイルへの入出力等については，本論文で与えたＰＬ／０＋の仕様記述の考え方を基にして，自然な拡張を行えばよい．

３．４　まとめ

プログラミング言語の代数仕様記述法を用いて，Ｐａｓｃａｌ風言語ＰＬ／０＋の仕様記述を与え，Ｐａｓｃａｌ風言語一般に対する仕様記述の技法を示した．そして，代入文，ＩＦ文，ＷＨＩＬＥ文に関しては簡明に，また，入出力機能，パラメタ付き手続きについては多少複雑だが自然な形で，仕様記述ができることを確認した．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．

最後に，本仕様記述の問題点として例外処理の記述があげられる．ＰＬ／０＋の仕様では，未宣言の変数への代入文は効果のない文として記述するなど，意味的なエラーに関し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．

第４章　プログラミング言語の代数的意味論に対する公理的検証体系の健全性

４．１　はじめに

本論文ではこれまでに，第２章でプログラミング言語の代数的仕様記述法を提案し，第３章でそれを用いてＰａｓｃａｌ風のプログラミング言語がいかに記述されるかについて述べた．本章および第５章では，代数的仕様記述法を確立する目的の一つであるプログラム検証について考察する．

プログラム検証法としては，ホーア流の公理的検証体系に基づく論理的手法がよく知られている．代数的意味論に基づくプログラム検証法を考える前に，代数的意味論とホーア流の公理的検証体系との関係を明らかにすることも重要である．

本章では，プログラミング言語の代数的意味論とホーア流の公理的検証体系との関係を明らかにするために，代数的意味論に対する公理的検証体系の健全性について考察する．具体的には，ＷＨＩＬＥプログラムを記述する簡単な言語ＷＬに対して，代数的意味論に関する公理的検証体系の健全性を示す．これにより，代数的意味論と公理的検証体系の間に矛盾のないこと，代数的意味論の与えられたプログラミング言語で書かれたプログラムの検証に公理的検証体系を用いてよいことがわかる．また，すでに広く認められているホーア流の公理的検証体系との無矛盾性は，代数的仕様の正しさを相対的に保証する．

同種の研究としては，表示的意味論と公理的検証体系の関係を明らかにしたものとして，（Donahue 76）がある．（Donahue 76）は，Pascal サブセットの Scott 流の表示的意味論を与え，公理的検証体系の表示的意味論に対する健全性を証明し，両者の間に矛盾がないことを示している．また，ホーア流の公理的検証体系の健全性と完全性の研究については，（Cook 78）がよく知られているが，（Cook 78）はプログラミング言語の意味論としては，関数的意味論を用いて，公理的検証体系の健全性と完全性の証明を行なっている．これらの研究と異なり，代数的意味論に対する公理的検証体系の健全性の証明を行なったことが本論文の特徴である．

以下では，まず，**4．2**で代数的方法を用いてプログラミング言語 WL の仕様を記述し代数的意味論を与える．一方，**4．3**で WL に対するホーア流の公理的検証体系を与える．さらに，**4．4**で**4．3**において表明を記述するのに用いた表明言語 AS の意味論を代数的方法を用いて与え，**4．5**で**4．3**で定義したホーア論理式に対して WL と AS の代数的意味論に基づいて解釈を与える．そして，**4．6**で WL の公理的検証体系が**4．5**で与えたホーア論理式の解釈に対して健全であることを示し，代数的意味論の与えられたプログラミング言語 WL のプログラム検証に公理的検証体系を用いてもよいことを保証する．

4．2　プログラミング言語WLの仕様

本節では，代数的仕様記述法を用いてプログラミング言語 WL の仕様を記述し代数的意味論を与える．WL は，データ型は自然数型，演算は加算と乗算，文は代入文，複合文，WHILE 文を持つ

言語とする．ＷＬ で書かれたプログラムの意味はプログラムの実行後の計算機の状態であり，ＷＬ の文の意味は計算機の状態を変化させる関数とする．

プログラミング言語 ＷＬ の代数的仕様 ＷＬ＿ＳＰＥＣ は，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

＜ＷＬ＿ＳＰＥＣ＿ＳＹＮ，
ＷＬ＿ＳＰＥＣ＿ＳＥＭ，
ＷＬ＿ＳＰＥＣ＿ＭＡＰ〉

で与えられる．以下，各部の仕様を順に与える．

（１）構文領域の仕様 ＷＬ＿ＳＰＥＣ＿ＳＹＮ は，図４．１の文脈自由文法で与えられる．この仕様が定める構文領域を Ｌ（ＷＬ）で表わす．

（２）意味領域の仕様 ＷＬ＿ＳＰＥＣ＿ＳＥＭ は，図４．２の抽象データ型の仕様で与えられる．　この意味領域の仕様において，ΣＤＡ＝〈ＳＤＡ，ＦＤＡ〉は計算機の状態，状態から状態への関数の表現などのプログラミング言語で扱うデータ型のシグニチャを，また，Σ＝〈ＳＤＡ，Ｆ〉（Ｆ⊃ＦＤＡ）は関数の表現を関数と解釈する演算の記号を含めた意味領域全体のシグニチャを表している．この意味領域の仕様が定める意味領域は，部分関数を持つ部分Σ代数 ＳＤ（ＷＬ）として，次のように定められるものである．

```
WL__SPEC__SYN = 〈N, T, P, START〉
  N = {program, statement, statement_list, factor,
       expression, term, condition, ident, number}
  T = {., :=, WHILE, DO, BEGIN, END, ;, +, *, (, ), >,
       V0, …, V9, 0, SUCC}
  P = {p1: program → statement .
       p2: statement → ident := expression
       p3: statement → WHILE condition DO statement
       p4: statement → BEGIN statement_list END
       p5: statement_list → statement_list ; statement
       p6: statement_list → statement
       p7: expression → term
       p8: expression → term + expression
       p9: term → factor * term
       p10: term → factor
       p11: factor → ident
       p12: factor → number
       p13: factor → ( expression )
       p14: condition → expression > expression
       p15: ident → V0
       p16: ident → V1
         : }
  START = program
```

図４．１　プログラミング言語ＷＬの構文領域の仕様（一部）

Fig.4.1 specification of syntactic domain of programming language WL

（i）各ソート $s \in S$ に対して，

$$SD(WL)_s = \{ \equiv [\xi] \mid \xi \in T[\Sigma]_s,\ \exists \eta \in T[\Sigma DA]_s,\ A \vdash \xi == \eta \}$$

ここで，$\equiv$ は $T[\Sigma]$ 上の同値関係 $\langle \equiv_s \rangle_{s \in S}$ で，Σ 項 ξ，η に対して，

$$\xi \equiv_s \eta \quad \text{iff} \quad A \vdash \xi == \eta$$

と定め，$\equiv [\xi]$ は Σ 項 ξ の同値関係 $\equiv$ に関する同値類を表す．また，

$$A \vdash \xi == \eta$$

は，等式集合 A から

$$\xi == \eta$$

が導出できることを意味する．

（ii）各演算記号 $f \in F_{s1 \ldots sn, s}$ に対して，

$$f_{SD(WL)}(\equiv [\xi 1], \cdots, \equiv [\xi n]) = \begin{cases} \equiv [f(\xi 1, \cdots, \xi n)] & (\exists \eta \in T[\Sigma DA]_s,\ A \vdash \eta == f(\xi 1, \cdots, \xi n)) \\ \text{未定義} & (\text{その他}) \end{cases}$$

この意味領域の定義において，FDA に属する演算記号は全域関数として，F－FDA に属する演算記号は部分関数として解釈されていることに注意されたい．

```
WL_SPEC_SEM = ⟨ΣDA, Σ, V, A⟩
  ΣDA = ⟨SDA, FDA⟩
    SDA = {bool, nat, id, state-stata, state-nat,
           state-bool, state}
    FDA = {true, false: → bool
           if_state: bool, state, state → state
           composition: state-state, state-state
                                → state-state
           iterate: state-bool, state-state
                                → state-state
           update, add_id: state, id, nat → state
           retreive: state-id → nat
           update_d: id, state-nat → state-state
           retreive_d: id → state-nat
             :   }
  Σ = ⟨SDA, F⟩
    F = FDA ∪ {
      apply_state-bool: state-bool state → bool
      apply_state-state: state-state state → state
      apply_state-nat: state-nat state → state
        :  }
```

図4．2　プログラミング言語WLの意味領域の仕様（一部）（1）

Fig.4.2 specification of semantic domain of programming language WL (1)

```
V = ⟨ V_s ⟩ s∈S
  V_state-state = {state-state0, …} ,
A = {
  apply_state-state(
      composition(state-state0, state-state1), state0)
    == apply_state-state(state-state0,
          apply_state-state(state-state1, state0))
  apply_state-state(update_d(id0, state-nat0), state0)
    == update(state0, id0,
                apply_state-nat(state-nat0, state0))
  apply_state-state(
      iterate(state-bool0, state-state0), state0)
    == if_state(
          apply_state-bool(state-bool0, state0),
          apply_state-state(
            composition(iterate(state-bool0, state-state0),
                        state-state0),
            state0),
          state0)
    :   }
```

図４．２　プログラミング言語ＷＬの意味領域の仕様（一部）（２）

Fig.4.2 specification of semantic domain of programming language WL (2)

（３）意味写像の仕様 ＷＬ＿ＳＰＥＣ＿ＭＡＰ は，図４．３の意味方程式系で与えられる．この仕様が定める意味写像は，写像族

$$SM(WL) = \langle M_m : L(WL)_m \rightarrow SD(WL)_{D(m)} \rangle_{m \in N}$$

である．

```
WL_SPEC_MAP = 〈D, M, X, R〉
  D : N → S
      D (program) = state-state
      D (statement) = state-state
      D (statement_list) = state-state
      D (number) = nat
      D (ident) = id
      D (expression) = state-nat
      D (term) = state-nat
      D (condition) = state-bool
        :
  M = {M program, M statement, …}
  X = 〈X s〉 s∈S
    X statement = {stm0}
        :
```

図４．３　プログラミング言語ＷＬの意味写像の仕様（一部）（１）

Fig.4.3 specfication of semantic mapping of programming language WL(1)

```
R = {
  M_program( p1(stm0) ) = M_statement(stm0)
  M_statement(p2(id0, exp0))
    = update_d(M_ident(id0), M_expression(exp0))
  M_statement(p3(cond0, stm0))
    = iterate(M_condition(cond0), M_statement(stm0))
  M_statement(p4(stm_list0))
    = M_statement_list(stm_list0)
  M_statement_list(p5(stm_list0, stm0))
    = composition(M_statement(stm0),
                  M_statement_list(stm_list0))
  M_statement_list(p6(stm0)) = M_statement(stm0)
  M_expression(p7(term0)) = M_term(term0)
  M_factor(p11(id0)) = retreive_d(M_idnet(id0))
  M_condition(p14(exp0, exp1))
    = gt_d(M_expression(exp0), M_expression(exp1))
      :   }
```

図４．３　プログラミング言語ＷＬの意味写像の仕様（一部）（２）

Fig.4.3 specfication of semantic mapping of programming language WL(2)

以上の図1～図3の仕様によって，プログラミング言語 WL の意味論が代数的に定まった．

4．3　プログラミング言語WLの公理的検証体系

本章では，プログラミング言語 WL に対するホーア流の公理的検証体系を与える．ホーア流の公理的検証体系は，表明を取り扱うための論理体系 $\mathcal{A}$ とプログラムを取り扱うための論理体系 $\mathcal{H}$ からなる．

まず，表明を取り扱うための論理体系 $\mathcal{A}$ は，古典的な自然数の公理系（Shoenfield 67）から量化記号およびそれに関する公理と推論規則を除いた論理体系であるとする．このように量化記号を取り除くことによって，論理体系 $\mathcal{A}$ の SUCC，+，* などの非論理記号ばかりでなく NOT，AND などの論理記号の意味が代数的に記述することができ，WL，$\mathcal{A}$，および $\mathcal{H}$ が同じ代数的枠組みで取り扱うことができる．（Donague 76）においても同様な論理体系が用いられている．また，論理体系 $\mathcal{A}$ の言語（以下では，AS と呼ぶ）は，WL に関する表明を記述する言語であるから，AS の構文規則には，WL の算術式 expression と条件式 condition に関する構文規則が含まれるとする．すなわち，AS の構文の仕様を図4．4の文脈自由文法で与える．AS の意味論については，次節で与える．

```
AS_SPEC_SYN = 〈N, T, P, START〉
  N = {assertion, as_formula, as_expression, term,
       expression, as_term, as_factor, ・・・ }
  T = {AND, OR, NOT, IMPLY, TRUE, FALSE, (, ), +, *, >, = }
  P = {
    a1: assertion → as_formula AND as_formula
    a2: assertion → as_formula OR as_formula
    a3: assertion → as_formula IMPLY as_formula
    a4: assertion → NOT as_formula
    a5: assertion → as_formula
    a6: as_formula → ( assertion )
    a7: as_formula → as_expression > as_expression
    a8: as_expression → expression
    a9: as_factor → factor
   a10: as_formula → condition
    p7: exression → term
   p10: term → factor
   p11: factor → ident
   p12: factor → ( expression )
   p15: ident → V0
   p16: ident → V1
     :   }
  START = assertion
```

図４．４　表明言語ＡＳの構文領域の仕様（一部）（１）

Fig.4.4 specification of syntactic domain of assertion language AS (1)

構文代入に関する演算

```
sub_assertion:
      assertion → assertion [ ident ← expression ]
sub_as_formula:
      asformula → as_formula [ ident ← expression ]
sub_factor: factor → factor [ ident ← expression ]
```

構文代入に関する公理

```
sub_assertion(
      a1( as_formula0, as_formula1 ),
      ident0, expression0 )
 == a1( sub_as_formula(as_formula0, ident0, expression0),
        sub_as_formula(as_formula1, ident0, expression0) )
as_factor( p11( ident0 ), ident0, expresiion0 )
 == if_expression( equal_idnet( ident0, ident1 ),
                   p12( exression0 ), p11( ident0 ) )
```

図４．４　表明言語ＡＳの構文領域の仕様（一部）（２）

Fig.4.4 specification of syntactic domain of assertion language AS (2)

次に，プログラムを取り扱うための論理体系 H を，その論理式の構文，公理および推論規則の順で与える．

まず，論理式の構文を与える．ホーア流の検証体系で用いられる論理式（以下では，ホーア論理式と呼ぶ）は次の形である．

$\{P\}$ A $\{Q\}$

ここで，P，Q は表明であり，A は WL のプログラムないしは文である．このホーア論理式は，直観的には，『プログラム A の実行前に P が成り立ち，かつ，プログラム A が停止すれば，プログラム A の実行後に Q が成り立つ』という意味である．WL の構文は**4．2**の図4．1で，AS の構文は図4．4ですでに与えた．これらによって，ホーア論理式の構文は定まっている．

次に，論理体系 H の公理と推論規則を与える．これらは，プログラミング言語の文に関する生成規則に対応して与えられる．すべての公理と推論規則を図4．5に与える．図4．5の中で，P［ident←expression］は，構文代入を表す記法であり，ホーア論理式 P 中の変数 ident を算術式 expression で置き換えられて得られる論理式である．

以上のように WL に対する公理的検証体系を定義できるが，次のことに注意をしておく必要がある．本節で与えた公理的検証体系は，この段階では**4．2**で与えた WL の代数的意味論とは（著者の意図としては同じものを記述しようとしているが）独立に与えられている．すなわち，公理的検証体系と代数的意味論が互いに矛盾なく定義されているかどうかはまだ証明されておらず，この公理的検証体系を用いてプログラムの検証を行なった結果が代数的意味論に対して正しいことが保証されていない．公理的検証体系を用いたプログラムの検証が代数的意味論に対して正しいことを保証するためには，代数的意味論と公理的検証体系が互いに矛盾していないことを証明しなければならない．これを以下に説明する．

（代入文に関する公理）

{ $\mathcal{P}$ [ident←expression]} ident := expression { $\mathcal{P}$ }

（ＷＨＩＬＥ文に関する推論規則）

$$\frac{\{\mathcal{P}\ \text{AND condition}\}\ \text{statement}\ \{\mathcal{P}\}}{\{\mathcal{P}\}\ \text{WHILE condition DO statement}\ \{\mathcal{P}\ \text{AND NOT condition}\}}$$

（文の連接に関する推論規則）

$$\frac{\{\mathcal{P}\}\ \text{statement\_list}\ \{\mathcal{Q}\}\ ,\quad \{\mathcal{Q}\}\ \text{statement}\ \{\mathcal{R}\}}{\{\mathcal{P}\}\ \text{statement\_list ; statement}\ \{\mathcal{R}\}}$$

（ＢＥＧＩＮ－ＥＮＤ構造に関する推論規則）

$$\frac{\{\mathcal{P}\}\ \text{statement\_list}\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ \text{BEGIN statement\_list END}\ \{\mathcal{Q}\}}$$

（帰結に関する推論規則）

$$\frac{\{\mathcal{P}\ \text{IMPLY}\ \mathcal{Q}\},\quad \{\mathcal{Q}\}\ \text{A}\ \{\mathcal{R}\},\quad \{\mathcal{R}\ \text{IMPLY}\ \mathcal{S}\}}{\{\mathcal{P}\}\ \text{A}\ \{\mathcal{S}\}}$$

図４．５　公理的検証体系の公理と推論規則

Fig.4.5 axioms and inference rules of axiom system

４．４　表明言語ＡＳの仕様

本節では，次節４．５でホーア論理式の解釈を与える準備として，表明言語ＡＳの意味論を与える．４．２でＷＬの意味論を代数的に与えたので，ホーア論理式の解釈も代数的枠組みの中で行えるようにする．そのために，ＡＳの意味論も代数的に与えることにする．

基本的な考え方としては，ＡＳで記述する表明の意味は状態の集合からブール値の集合への関数であるとする．また，ホーア流の公理的検証体系では，表明の中にプログラミング言語によって記述される条件式，算術式を書くことができる．そのため，条件式，算術式に関しては，ＡＳの仕様とＷＬの仕様が全く同じものでなくてはならない．すなわち，条件式，算術式については，構文領域の仕様中に共通の生成規則を，意味写像の仕様中に共通の意味方程式を持たなくてはならない．さらに，意味領域の仕様中に，表明言語の意味領域がプログラミング言語の意味領域の条件式，算術式に関する領域，ブール型の領域ｂｏｏｌ，計算機の状態の領域ｓｔａｔｅ，状態からブールへの関数の領域ｓｔａｔｅ－ｂｏｏｌ，状態から自然数への関数の領域ｓｔａｔｅ－ｎａｔを矛盾なく含むように記述されなくてはならない．これらのことを考慮にいれて，表明言語ＡＳに対する代数的仕様を以下のように与える．

表明言語ＡＳの仕様は，３項組

＜ＡＳ＿ＳＰＥＣ＿ＳＹＮ，
ＡＳ＿ＳＰＥＣ＿ＳＥＭ，
ＡＳ＿ＳＰＥＣ＿ＭＡＰ＞

として与えられる．構文領域の仕様ＡＳ＿ＳＰＥＣ＿ＳＹＮは，既に，図４．４で与えた．ただし，プログラミング言語の構文領域の仕様記述の場合は，構文を構成する演算，即ち，生成規則の

他には演算が必要ではなかったが表明言語の場合は，構文代入を行う演算とそれに関する公理を表明言語 ＡＳ の構文領域の仕様（図４．４）に加える．

意味領域の仕様 ＡＳ＿ＳＰＥＣ＿ＳＥＭ，意味写像の仕様 ＡＳ＿ＳＰＥＣ＿ＭＡＰは，図４．６，図４．７に与える．

このように，公理的検証体系で用いられる表明言語 ＡＳ に対しても**４．２**で与えた ＷＬ の代数的意味論と矛盾なく代数的意味論を与えることができる．

```
AS_SPEC_SEM = ⟨ΣDA, Σ, V, A⟩
  ΣDA = ⟨SDA, FDA⟩
    SDA = {bool, nat, id, state-nat, state-bool, state}
    FDA = {true, false: → bool
             and_d, or_d:
                       state-bool, state-bool → state-bool
             not_d: state-bool → state-bool
               :  }
  Σ = ⟨SDA, F⟩
    F = FDA ∪ {apply_state-bool:
                       state-bool, state → bool
                 apply_state-nat: state-nat, state → nat
                  :   }
  V = ⟨Vs⟩ s∈S
    Vstate-bool = {state-bool0, …}    …
  A = {apply_state-bool(
                and_d(state-bool0, state-bool1), state0)
          == and(apply_state-bool(state-bool0, state0),
                 apply_state-bool(state-bool1, state0))
        apply_state-bool(not_d(state-bool0, state0))
          == not(apply_state-bool(state-bool0, state0)
          :   }
```

図4．6　表明言語ASの意味領域の仕様（一部）

Fig.4.6 specification of semantic domain of assertion language AS

```
AS_SPEC_MAP = 〈D, M, X, R〉
  D : N → S
    D(assertion) = state-bool
    D(as_formula) = state-bool
    D(condition) = state-bool
    D(ident) = id
    D(number) = nat
    D(expression) = state-nat
    D(term) = state-nat
  M = {M_assertion, M_as_formula, …}
  X = 〈X_s〉_{s∈S}
    X_assertion = {as0, …}   …
  R = {
    M_assertion(a1(as_form0, as_form1))
      = and_d(M_as_formula(as_form0),
              M_as_formula(as_form1))
    M_assertion(a4(as_form0))
      = not_d(M_as_formula(as_forma0))
        :   }
```

図4．7　表明言語ASの意味写像の仕様（一部）

Fig.4.7 specification of semantic mapping of assertion language AS

4．5　ホーア論理式の解釈

4．2で与えたプログラミング言語 WL の代数的仕様記述および4．4で与えた表明言語 AS の仕様記述が定める代数的意味論に基づいて，ホーア論理式の解釈を次のように与える．

ホーア論理式 $\{P\}$ A $\{Q\}$ に対して，

「すべての $s \in SD(WL)_{state}$について，

$$apply\_state\text{-}bool_{SD(AS)}(M_{assertion}(P),\ s)$$

が真であり，かつ

$$apply\_state\text{-}state_{SD(WL)}(M_{statement}(A),\ s)$$

が定義されている（すなわち，初期状態 s に対して プログラム A の実行が停止する）ならば

$$apply\_state\text{-}bool_{SD(AS)}(M_{assertion}(Q),\ apply\_state\text{-}state_{SD(WL)}(M_{statement}(A),\ s))$$

が真である．」

ここで，ＳＤ（ＷＬ）はＷＬの意味領域を，ＳＤ（ＡＳ）はＡＳの意味領域を表している．このように，ホーア論理式には２つの言語の意味領域の演算が混在しているが，ブール型ｂｏｏｌ，計算機の状態ｓｔａｔｅに関しては，ＷＬの意味領域の部分領域とＡＳの意味領域の部分領域は同型であり，上述の論理式ではこれら２つのものを同一視して記述してある．以下では，このホーア論理式の解釈の表記の中で，メタ記号として全称記号 ∀，存在記号 ∃，連言記号 &，含意記号 → を用いる．さらに，意味写像の引数，例えば，$M_{assertion}$ の引数は，厳密にいうと文字列ではなく，それに対応する構文領域の要素，すなわち，ＷＬ＿ＳＰＥＣ＿ＳＹＮ項であるが，理解しやすさと簡単さのため，文字列を使って表す．

ここで，次の二つの事実に注意しよう．

（１）意味領域のｂｏｏｌ型の項の値が真であるのは，それを表わす項がｔｒｕｅ（）という項と等しいことが意味領域の仕様を用いて推論できるとき，かつ，そのときに限る．

（２）意味領域の値が定義されているのは，それを表わす項がある Σ ＤＡ 項と等しいことが意味領域の仕様を用いて推論できるとき，かつ，そのときに限る．

これら二つのことから，上のホーア論理式の解釈は，以下のように言い換えられる．

```
∀ s ∈ T [ΣDA]state,
    apply_state-bool(
        M'assertion(𝒫), s)
      ==true()
&
    ∃ t ∈ T [ΣDA]state,
        apply(M'statement(A), s) == t
→
    apply_state-bool(
        M'assertion(𝒬),
        apply_state-state(
            M'statement(A), s))
      ==true()
```

ここで，等式 $\xi == \eta$ は，プログラミング言語 WL および表明言語 AS の意味領域の仕様中の公理を用いて，ξ と η が等しいと推論されることを表している．また，任意の $m \in N$ に対して，

$$M'_m : T[WL\_SPEC\_SYN]_m \rightarrow T[\Sigma]_{D(m)}$$

は，意味写像の仕様中の意味方程式に従って，構文領域の要素を Σ 項に変換する写像とする．以下の議論では，ホーア論理式の解釈として，この二番目のものを用いる．また，M' を単に M と略記する．

４．６　代数的意味論に対する公理的検証体系の健全性

ＷＬ の代数的意味論と公理的検証体系が互いに矛盾していないことを示すために，代数的意味論に対する公理的検証体系の健全性を証明する．これは，表明を取り扱う論理体系 $\mathcal{A}$ の各公理と推論規則が ＡＳ の代数的意味論のもとで妥当であることを示すこと，および，プログラムを取り扱う論理体系 $\mathcal{H}$ の各公理と推論規則が４．５で与えたホーア論理式の解釈に対して妥当であることを示すことによって証明される．

まず，表明を取り扱う論理体系 $\mathcal{A}$ の各公理と推論規則の妥当性は，ＡＳ の意味領域の仕様を用いて簡単に証明できるので，ここでは，その証明の説明は省略する．

次に，プログラムを取り扱う論理体系 $\mathcal{H}$ については，次の主要な三つのもの，すなわち，代入に関する公理，文の連接に関する推論規則，および，ＷＨＩＬＥ 文に関する推論規則の妥当性の証明について述べる．その他の推論規則についても同様に証明できるが，ここでは省略する．

４．６．１　代入に関する公理の妥当性

４．５で与えたホーア論理式の解釈を用いると，代入に関する公理は，図４．８の中の論理式で表される．この論理式の妥当性は，ａｓｓｅｒｔｉｏｎ と ｅｘｐｒｅｓｓｉｏｎ の構文の構造に関する帰納法を用いて，場合分けが煩雑ではあるが簡単に証明できる．

```
∀ s ∈ T [ Σ D A ] state,
      apply_state-bool(
          M assertion(𝒫 [ ident← expression] ),  s )
       == true()
    & ∃ t ∈ T [ Σ D A ] state,
        apply_state-state(
            M statement(ident := expression),  s ) == t
  → apply_state_bool(
      M assertion(𝒫 ),
      apply_state-state(
          M statement(ident := expression),  s ))
    == true()
```

図４．８　代入文に関する公理の解釈

Fig.4.8 interpretation of axiom for assign statement

４．６．２　文の連接に関する推論規則の妥当性

４．５で与えたホーア論理式の解釈を用いると，文の連接に関する推論規則は図４．９の論理式で表される．この論理式の妥当性は次のようにして示される．まず，１）または４）が偽の場合には，図４．９の論理式が真となるのは自明である．そこで，１）および４）が真であると仮定した場合に，７）が真であることを示す．

ここで，状態 s に対して，８）が真となるかどうか，すなわち，状態 s において

statement_list ; statement

を実行したとき，停止するかどうかで場合分けを行う．

（i）ある s に対して，８）が偽のとき，明らかに７）は成立する．

```
1) 「∀ s ∈ T [Σ D A] state,
      apply_state-bool(M assertion(𝒫), s)
        == true()
    &
    2) ∃ t ∈ T [Σ D A] state,
         apply_state-state(
           M statement_list(statement_list), s)
          == t
    →
    3) apply_state-bool(
         M assertion(𝒬),
         apply_state-state(
           M statement_list(statement_list), s))
        == true()
   」
&
4) 「∀ s ∈ T [Σ D A] state,
      apply_state-bool(M assertion(𝒬), s)
        == true()
    &
    5) ∃ t ∈ T [Σ D A] state,
           apply_state-state(
             M statement(statement), s) == t
    →
    6) apply_state-bool(
         M assertion(ℛ),
         apply_state-state(M statement(statement), s))
        == ture()
→
7) 「∀ s ∈ T [Σ D A] state,
      apply_state-bool(M assertion(𝒫), s) == true()
    &
    8) ∃ t ∈ T [Σ D A] state,
         apply_state-state(
         M statement_list(
             statement_list ; statement), s)
          == t
    →
    9) apply_state-bool(
         M assertion(ℛ),
         apply_state-state(
           M statement_list(
             statement_list ; statement), s))
        == true()
   」
```

図4．9　文の連接に関する推論規則の解釈

Fig.4.9 interpretation of inference rule for sequence of statements

（ii）ある s に対して，8）が真のときは次のようになる．まず，プログラムの停止に関する条件8）が真なので7）からそれを取り除いて次式7’）に変形する．

```
      ｛ apply_state-bool(
          Massertion(𝒫), s)
        ==true()
      →apply_state-bool(
          Massertion(ℛ),
          apply_state-state(
            Mstatement_list(
              statement_list;
              statement),
            s)
        ==true()
```

7’）

意味領域の仕様中の演算 apply_state-state に関する公理の性質から，ある s に対して8）が真，すなわち，状態 s での statement_list ; statement の実行が停止するならば，同じ状態 s に対して2）が真となり，状態

```
apply_state-state(
  Mstatement_list(statement_list), s)
```

に対して5）が真となる，すなわち，状態 s での statement_list の実行が停止し，その後の statement の実行も停止する．このことを用いて，7）を7’）に変形したようにプログラムの停止に関する条件を取り除いて，1），4）

を1’），4’）に変形し，それらに，モードス・ポーネンスを適用すると7’）が導かれる．すなわち，8）が真の場合にも7）が真となる．

このように，すべての場合に対して，図4．9の論理式は真となる．すなわち，文の連接に関する推論規則の妥当性が証明された．

4．6．3　WHILE文に関する推論規則の妥当性

4．5で与えたホーア論理式の解釈を用いると，WHILE 文に関する推論規則は図10の論理式で表される．この論理式の妥当性は次のように示される．まず，10）が偽となる場合には，明らかに図10の論理式は真である．次に，10）が真となる場合について，13）が真となることを示す．ここで，ある状態sに対して，14）が真であるかどうか，すなわち，WHILE 文の実行が停止するかしないかで場合分けを行う．

```
10) 「∀s ∈ T [ΣDA]state,
        apply_state-bool(
          Massertion(𝒫 AND condition), s)
        == true()
      &
11)   ∃t ∈ T [ΣDA]state,
        apply_state-state(
          Mstatement(statement), s) == t
    →
12)   apply_state-bool(
        Massertion(𝒫),
        apply_state-state(Mstatement(statement), s))
      == true()
    」
→
13) 「∀s ∈ T [ΣDA]state,
        apply_state-bool(Massertion(𝒫), s) == true()
      &
14)   ∃t ∈ T [ΣDA]state,
16)     apply_state-state(
          Mstatement(WHILE condition DO statement),
          s)
        == t
    →
15)   apply_state-bool(
        Massertion(𝒫 AND NOT condition),
          apply_state-state(
            Mstatement(WHILE condition DO statement),
            s))
      == true()
    」
```

図4．10　WHILE文に関する推論規則の解釈

Fig.4.10 interpretation of inference rule for WHILE statement

（ｉ）１４）が偽の場合は，明らかに１３）は真である．

（ii）１４）が真である，すなわち，ＷＨＩＬＥ文の実行が停止する状態ｓに対しては，次のようになる．まず，１３）は次式１３’）に書き換えることができる．

```
13')  apply_state-bool(
        Massertion(P), s)
        ==true()
     →apply_state-bool(
          Massertion(
              P AND NOT condition),
          apply_state-state(
              Mstatement(
                  WHILE condition DO
                  statement),
              s))
        ==true()
```

ここで，状態ｓに対して，１４）が真となる条件を考える．図４．１０中の１６）の左辺の項がある ΣＤＡ項ｔと等しいと推論されるためには，必ず，意味領域の仕様中の演算ｉｔｅｒａｔｅに関する公理を一回以上用いなくてはならないことが，意味領域の仕様中の公理から分かる．以下では，ΣＤＡ項ｔと等しいことを推論するために，演算ｉｔｅｒａｔｅに関する公理を用いなければならない回数に関する帰納法を用いて，１３’）が真であることを証明する．

（基底段階）　演算 iterate に関する公理を一回だけ用いて１４）が真と推論できる状態 s0 を考える．このような状態 s0に対して，WHILE 文の条件式 condition は偽であり，１３’）は次式１７）のように変形できる．

```
17) { apply_state-bool(
      Massertion(P), s0))
      ==true()
    → apply_state-bool(
        Massertion(
            P AND NOT condition),
        s0))
      ==true()
```

この式１７）が成立することは，表明言語の仕様中の公理を用いて簡単に証明できる．

（帰納法の仮定）　演算 iterate に関する公理を n（n＞1）回用いると14）が真と推論できる状態 sn を考え，15'）が成立すると仮定する．この式を18）とする．

```
18) { apply_state-bool(
        Massertion(P), sn)
      ==true()
    → apply_state-bool(
          Massertion(
              P AND NOT condition),
            apply_state-state(
              Mstatement(
                  WHILE condition DO
                          statement),
              sn))
      ==true()
```

（帰納段階）　演算 iterate に関する公理を m＝n＋1 回用いれば14）が真と推論できる状態 sm を考える．この場合は，WHILE 文の条件式 condition が状態 sm 真でなくてはならない．このとき，状態 sm での WHILE 文の実行が停止するためには，状態 sm での statement の実行が停止する必要がある．そして，状態 sm に対して，条件式 condition が真であり，statement の実行が停止することより，10）を次式10’）に変形できる．

```
10’）{  apply_state-bool(
        Massertion(P), sm))
        ==true()
     →apply_state-bool(
          Massertion(P),
          apply_state-stat(
            Mstatement(statement), sm)))
        ==true()
```

ここで，状態 sm で statement を実行した後の状態 u と置く．

```
u＝apply_state-state(
        Mstatement(statement), sm)
```

状態 sm での WHILE 文の実行が停止するためには，条件式 condition が状態 sm で真であるので，状態 u での WHILE 文の実行が停止しなくてはならない．そして，

```
apply_state-state(
    Mstatement(WHILE condition DO
                    statement),
    u)
```

と等しい ΣDA 項 t を推論するのに，演算 iterate に関する公理を n 回用いなければならない．ここで，帰納法の仮定18）より，次式18'）を得る．

```
        { apply_state-bool(
          Massertion(P), u)
          ==true()
        →apply_state-bool(
18')      Massertion(
              P AND NOT condition),
          apply_state-state(
            Mstatement(
                WHILE condition DO
                        statement), u)
          ==true()
```

以上の議論で得られた10'）および18'）に，モードスポーネンスを適用すると，15'）中の状態 s を状態 sm で置き換えた式が得られる．すなわち，帰納法の帰納段階が証明された．

このように，演算 iterate に関する公理の適用回数による帰納法により，WHILE 文に関する推論規則の妥当性が証明された．これは，直感的には，WHILE 文のループの繰り返し回数に関する帰納法になっている．

以上では，WL に対する公理的検証体系のプログラムを取り扱う論理体系 $\mathcal{H}$ の主要な公理と推論規則に対してそれが WL の代数的意味論に対して妥当であることを証明した．残りの BEGIN－END 構造に関する推論規則および帰結に関する推論規則も同様にして妥当性を証明することができる．また，論理体系 A の各公理と推論規則の妥当性も証明できる．これにより，WL に対する公理的検証体系のすべての公理と推論規則の代数的意味論に対する妥当性が証明され，したがって，代数的意味論に対して公理的検証体系が健全であることが証明された．

4．7　まとめ

本章では，プログラミング言語の代数的意味論とホーア流の公理的検証体系との関係を明らかにするために，代数的意味論に対する公理的検証体系の健全性について考察した．具体的には，WHILE プログラムを記述する簡単な言語WLを用いて，代数的意味論に対する公理的検証体系の健全性の証明を行なった．これにより，WL に関しては代数的意味論と公理的検証体系の間に矛盾のないことが示された．このように無矛盾性が保証されるので，代数的意味論の与えられたプログラミング言語で書かれたプログラムの検証に公理的検証体系を用いることができる．また，すでによく知られているホーア流の公理的検証体系との無矛盾性は，この章で与えた代

数的仕様の正しさを相対的に保証しているとも考えられる.

本章では，代数的意味論と公理的検証体系が無矛盾であることを示したが，さらに，ホーア流の公理的検証体系が代数的意味論に対して完全であることを証明し，代数的意味論と公理的検証体系がホーア論理式の証明に関して等価であることを示すことが考えられる.

第5章　代数的手法に基づくプログラム検証

5．1　はじめに

第4章では，プログラミング言語の代数的意味論に対するホーア流の公理的検証体系の健全性を示すことにより，代数的に仕様記述されたプログラミング言語で書かれたプログラムの検証にホーア流の公理的検証体系を用いることができることを示した．本章では，プログラムの検証に代数的意味論を直接用いることを考える．すなわち，プログラミング言語の代数的意味論に基づくプログラムの正当性検証を形式的に定義する．

プログラムの検証とは，直観的に言えば，プログラム P とそれが満たすべき仕様 S が与えられたとき，P が確かに S を満たすことを証明することである．しかし，このままでは具体的に何をしたら良いのか分からない．形式的なプログラム検証のためには，『プログラムが仕様を満たす』ということの妥当でかつ厳密な定式化が必要である．この定式化は次の3項目の定義によってなされる．

（1）プログラムの実行によって達成される機能（プログラムの意味）

（2）プログラム仕様の表す機能（プログラムの仕様の意味）

（3）プログラム仕様に対するプログラムの正当性

（1）は，プログラミング言語の形式的意味論を与えることで定義される．（2）は，（1）と同様に，プログラム仕様の記述言語の形式的意味論を与えることで定められる．（3）は，プログラムの

意味する機能がプログラム仕様の意味する機能を満たすことの定義である．これらの定義が定まれば，与えられたプログラムと仕様に対してプログラム検証をするには何をすれば良いのかが明確となる．ここで重要なことは，（3）の妥当な定義のためには，（1）と（2）が同じ枠組の中で定義されなければならないことである．

本章では，プログラム検証を念頭においた場合の（1），（2）の代数的な定義手法，および，（3）の一つの妥当な定式化を提案し，プログラムの代数的な検証法の基礎を明確にする．代数的枠組を用いることの結果として，プログラムの意味ばかりでなく，意図する機能も抽象データ型として仕様記述することとなり，抽象データ型の検証手法，特に，項書き換え系を利用した技法がそのままプログラム検証に応用できる．このことは，本章のプログラム検証の形式化の最も大きな特徴である．

5．2　プログラミング言語の代数的仕様記述法

プログラム検証は，一般に，プログラムの入出力特性について行われる．このことを考慮に入れ，本章では，第2章で提案したプログラミング言語の代数的仕様記述法に，プログラムの入出力特性を陽に記述できる機能を付け加えた仕様記述法を与える．

第2章で述べたプログラミング言語の代数的仕様記述法は，言語の構文領域および意味領域を多ソート代数（抽象データ型）として，また，プログラムの意味を構文領域から意味領域への準同型写像として捉え記述する方法である．この記述法に，プログラムの入出力データ領域に関する記法を組み入れ，プログラムの意味は入力から出力への関数であるようにした仕様記述法を以下のように与える．

プログラミング言語の代数的仕様（以下では単に言語仕様と言う）は，構文領域，入出力領域，意味領域，意味写像の各々を規定する記述からなる4項組

〈SPEC_SYN, SPEC_IO,
SPEC_SEM, SPEC_MAP〉

であるとする．以下に，これらの項目を説明する．

（1）構文領域の記述

SPEC_SYN は文脈自由文法

〈T, N, P, START〉

であり，構文領域を規定する．T は終端記号の集合，V は非終端記号の集合，START∈N は開始記号である．P は生成規則の集合である．

（2）入出力領域の記述

SPEC_IO は抽象データ型の仕様

$\langle \Sigma^{IO}, V^{IO}, A^{IO}, input, output \rangle$

である．Σ^{IO} はシグニチャ $\langle S^{IO}, F^{IO} \rangle$，$A^{IO}$ は Σ^{IO} 上の等式の集合，input, output は S^{IO} の要素である．ただし，S^{IO} にはブール型のソート bool，F^{IO} にはソート bool の定数 true, false が含まれるとする．

SPEC_IO の規定する入出力領域 IO（SPEC_IO）は $\langle \Sigma^{IO}, A^{IO} \rangle$ を抽象データ型の仕様と見なしたとき，それが始代数意味論のもとで意味する抽象データ型であるとする．プログラムの入力と出力のソートは input, output で指定される．

（３）意味領域の記述

ＳＰＥＣ＿ＳＥＭ は次の条件を満たす抽象データ型の仕様

$$\langle \Sigma^{SEM},\ V^{SEM},\ A^{SEM} \rangle$$

である．Σ^{SEM} はシグニチャ $< S^{SEM},\ F^{SEM} >$ である．

（ａ）$S^{IO} \subset S^{SEM}$，$F^{IO} \subset F^{SEM}$，$A^{IO} \subset A^{SEM}$

（ｂ）A^{SEM} は A^{IO} に矛盾しない

（ｃ）input－output $\in S^{SEM}$

（ｄ）apply $\in F^{SEM}$

ＳＰＥＣ＿ＳＥＭ の規定する意味領域 ＳＤ（ＳＰＥＣ＿ＳＥＭ）はそれが始代数意味論で意味する抽象データ型であるとする．ただし，input－output は入力から出力への関数を表すソートであり，

apply：input－output，input

→output

は入力から出力への関数を入力に適用し，その結果を返す関数であるように仕様記述されているものとする．また，（ｂ）は ＩＯ（ＳＰＥＣ＿ＩＯ）が ＳＤ（ＳＰＥＣ＿ＳＥＭ）の部分代数となることを要請するための条件である．

（４）意味写像の記述

ＳＰＥＣ＿ＭＡＰ は３項組

$$\langle D,\ M,\ X,\ R \rangle$$

である．$D: N \rightarrow S$ は非終端記号に意味領域のソートを割り当てる写像である．ただし，

D（ＳＴＡＲＴ）＝input－output

であるとする．

以上（１）～（４）のような言語仕様が与えられたとき，プログラム Ｐ の意味は入力から出力への関数である ＳＭ（ＳＰＥＣ＿Ｍ

AP)$_{START}$(P) で与えられる．すなわち，プログラム P は，入力 $x \in SEM_{input}$ を与えて実行するとその結果としてソート output の意味領域の要素

apply(SM(SPEC_MAP)$_{START}$(P), x)

を出力するものとして解釈される．

5．3 プログラムの仕様とその意味

本論文では，プログラム検証の形式化に代数的枠組を採用しているので，全体の整合性を考え，プログラムに対して意図された機能も代数的に記述することにする．さらに，プログラムの入出力特性に関する検証を前提にしているので，このことも考慮に入れ，プログラム仕様およびその意味を次のように定める．

プログラム仕様は次の条件を満たす抽象データ型の仕様

$$\langle \Sigma^{FUN},\ V^{FUN},\ A^{FUN} \rangle$$

である．Σ^{FUN} はシグニチャ $< S^{FUN},\ F^{FUN} >$ である．

(a) $S^{IO} \subset S^{FUN}$, $F^{IO} \subset F^{FUN}$, $A^{IO} \subset A^{FUN}$

(b) A^{FUN} は A^{IO} に矛盾しない

(c) F^{FUN} は次の関数記号を含む．

Q: input → bool

B: input → output

(a)，(b) は意味領域の仕様の場合と同じ条件である．(c) の Q はプログラムの実行が意味を持つ入力を規定する関数，また，B はプログラムの入出力の動作特性を規定する関数と解釈する．始代数意味論のもとでこの仕様が定める抽象データ型 FUN をプロ

グラム機能と呼ぶ.

5．4　仕様に対するプログラムの正当性

言語仕様によって定められるプログラムの意味は，入力から出力への関数である．一方，プログラム仕様は，プログラムで処理すべき入力データの範囲を述語 Q によって規定し，その入力に対して出力されるべき値をBで規定する．従って，プログラム仕様に対するプログラムの正当性を次のように定義する．

5．2の方法で言語仕様を与えたプログラミング言語のプログラムを P，**5．3**の方法で定めたプログラムの仕様を S とする．このとき，P が S を満たすのは次の条件が成立するときかつその時に限る．

『任意の $x \in IO_{input\text{-}output}$ に対して，

$Q_{FUN}(x) = true_{IO(SPEC\_IO)}$

ならば

$apply_{SD(SPEC\_SEM)}($

$SM(SPEC\_MAP)_{START}(P),\ x)$

$= B_{FUN}(x)$ 』

ただし，Q_{FUN}，$true_{IO(SPEC\_IO)}$，$apply_{SD(SPEC\_SEM)}$，B_{FUN} は関数記号 Q，true，apply，B をそれぞれの抽象データ型 FUN，IO（SPEC_IO），SD（SPEC_SEM），FUN 上で解釈した関数を表す．

具体的な例でこの定義を見てみよう．プログラミング言語は第2

章で言語仕様を与えた言語 SL を用いる．ただし，その言語仕様は本章の形式に合わせるため，

```
input=output=state
```

として，入出力領域の記述部分を加えてあるものとする．非負整数の階乗を計算するプログラム仕様Sの主要な部分は次の通りである．

```
fact: nat→nat
Q: nat→bool
B: input→ouput

fact(zero())
    ==suc(zero())
fact(suc(x))
    ==mult(fact(x), suc(x))

Q(st0)
    ==true()

S(st0)
    ==add_id(
        update(st0, id0(), zero()),
        id1(),
        fact(
            retrieve(st0, id0()))))
```

この仕様を満たすべきプログラム P を

```
v1:=1;
while v0>0 do
    begin
        v1:=v0*v1;
        v0:=v0-1
    end.
```

とする．このとき，P の S に対する正当性は，

Q(st0)==true()

であるので，

『任意の $x \in IO_{state}$ に対して，

$apply_{SD(SEPC\_SEM)}($
$SM(SPEC\_MAP)_{START}(P),\ x)$
$= B_{FUN}(x)$』

を証明することで示される．

言語仕様によれば，意味写像 SM（SPEC_MAP）（の族）は，

```
SM(SPEC_MAP)START(P)
  ==compostion(
        iterate(
            SM(SPEC_MAP)(v0>0),
            composition(
                SM(SPEC_MAP)
                            (v0:=v0-1),
                SM(SPEC_MAP)
                            (v1:=v0*v1))),
        SM(SPEC_MAP)(v1:=1))
```

を満たす．この式がプログラムPの意味を定めるものとして，上述の正当性の条件の検証に用いる．

5．5　まとめ

本章では，代数的意味論の枠組の中でプログラムの正当性検証を行うための基本的な考え方について述べた．即ち，プログラムの意味を入力から出力への関数と考えたプログラミング言語の代数的仕様記述法，ならびに，プログラムの機能の代数的仕様記述法を与え，それに基づいてプログラムの正当性の定義を与えた．

プログラミング言語の形式的意味論に基づくプログラムの正当性検証については，（Aiello, Aiello, Weyhrauch 77）らの表示的意

味論に基づくものがある．これに対して，本章で与えたプログラムの正当性は言語の代数的意味論を基礎にしているため，検証の（半）自動化に項書き換え系を直接利用できる．このことが本枠組の長所である．この長所を利用し，この正当性の定義に基づくプログラムの正当性の検証手法を明らかにし，それを用いてプログラムの正当性検証システムを作成することができる．

第６章　結論

本論文では，プログラミング言語の形式的仕様記述する方法として，代数的仕様記述法の提案を行い，また，代数的に意味論の与えられたプログラミング言語のプログラムの正当性検証について考察を行なった．

第２章では，プログラミング言語の代数的仕様記述法を定義し，その意味論を明らかにするとともに，意味写像が一意に定まることを証明した．この仕様記述法は，すべて抽象データ型の代数的仕様記述法の枠組の中で取り扱われており，そのため，十分な抽象性と形式性を備えている．また，第２章で述べた代数的仕様から，コンパイラを自動生成するアイデアに基いて，言語開発支援システムＬａｓｓ（ＬＡnguage development Ｓupporting Ｓystem）が作成されている（酒井，北，坂部，稲垣 85）．

第３章では，第２章で与えた代数的仕様記述法の有効性を実証することを目的として，Ｐａｓｃａｌ風言語ＰＬ／０＋の代数的仕様記述を与え，その説明を通して，Ｐａｓｃａｌ風言語一般に対する代数的仕様記述の技法を明らかにした．

第４章では，プログラミング言語の代数的意味論とホーア流の公理的検証体系との関係を明らかにするために，代数的意味論に対する公理的検証体系の健全性を考察し，ＷＨＩＬＥプログラム言語ＷＬについてこれを証明した．これにより，代数的意味論と公理的検証体系の間に矛盾のないこと，代数的意味論の与えられたプログラミング言語で書かれたプログラムの検証に公理的検証体系を用いてよいことがわかる．また，すでに広く認められているホーア流の公理的検証体系との無矛盾性は，代数的仕様の正しさを相対的に

保証する.

第5章では，代数的意味論の枠組の中でプログラムの正当性検証を行うための基本的な考え方について述べた．即ち，プログラムの意味を入力から出力への関数と考えたプログラミング言語の代数的仕様記述法，ならびに，プログラムの機能の代数的仕様記述法を与え，それに基づいてプログラムの正当性の定義を与えた.

本研究に残された課題として次のものがある．まず，代数的仕様記述法は，意味論が等式論理に基づいており他の形式的仕様記述法に比較的して分かりやすい．しかし，第3章のＰＬ／０＋の仕様では未宣言の変数への代入文は効果のない文として記述するなど意味的なエラーに関しては便宜的な方法を採用した．これは，エラー処理を完全に記述すると記述量が膨れ理解性を損なうからである．これについては，エラーなどの例外処理を形式的にかつ理解しやすく記述する方法を開発する必要がある．また，第3章では識別子などを既存のものとして扱えるように代数的仕様記述法を改良したが，付録に載せたＰＬ／０＋の仕様ではそれらの部分を分離しても多くの部分が平坦に書かれており理解性を損なっている．これについては，抽象データ型の代数的仕様記述の利点である階層性およびモジュール性を活かすようにプログラミング言語の仕様記述法をさらに改良する必要がある.

また，第5章で述べたプログラムの正当性検証については，プログラミング言語の代数的意味論を基礎としているために，検証の（半）自動化に項書換え系を直接利用することができる．この長所を利用し，本論文で与えたプログラムの正当性に基づくプログラムの検証手法を明らかにし，それを用いてプログラムの正当性検証システムを作成することが望まれる.

謝辞

本論文は，著者が名古屋大学大学院工学研究科博士課程に在学中に行なった研究をまとめたものである．本研究を進めるにあたり，懇切丁寧なご指導とご督励を賜った名古屋大学 稲垣康善 教授，杉江昇 教授，坂部俊樹 助教授に深謝致します．また，豊橋技術科学大学 本多波雄 学長，中京大学 福村晃夫 教授に感謝致します．

日頃熱心にご討論いただいた東北大学 阿曽弘具助 教授，名古屋大学 平田富夫 講師，名古屋大学 杉野花津江 助手に感謝致します．また，ご指導をいただいた名古屋大学 吉田雄二 教授，鳥脇純一郎 教授に感謝します．

第３章の一部は稲垣研究室の高木雄二君（現ＮＴＴ勤務）の助力を得ており感謝を表します．また，酒井正彦君，山中英樹君（現富士通勤務）をはじめとする稲垣研究室の皆様に感謝します．また，研究を進めるのにあたりいろいろとお世話下さった山岸慶子さんに深く感謝します．

最後に，本論文をまとめる際にご支援を下さった沖電気工業（株）研究開発本部 山本正隆 本部長に感謝致します．また，総合システム研究所 似鳥一彦 所長，椎野努 主任調査役，植村昌俊 部長，永田淳次 係長に感謝致します．

文献（英文）

（Aiello, Aiello, Weyhrauch）

L. Aiello, M. Aiello, R. W. Weyhrauch : "Pascal in LCF : Semantics and examples of proof", Theor. Compt. Sci., 5, pp.135-177 (1977)

（ADJ 77）

J. A. Gougen, J. W. Thatcher, E. G. Wagner, J. B. Wright : "Initial algebra semantics and continuous algebras", JACM, 24, pp.68-95 (1977)

（ADJ 81）

J.W.Thatcher, E.G.Wagner, J.B.Wright : "More on advice on structuring compilers and proving them correct", Theor. Comput. Sci., 15, pp.223-249 (1981)

（Cook 78）

S. Cook : "Soundeness and completeness of an axiom system for program verification",SIAM J. Comput. 7, 1 (1978)

（Courcelle, Franchi-Zannetacci 82）

B.Courcelle, P.Franchi-Zannetacci : "Attribute grammars and primitive recursive shemes", Theor. Compt. Sci., 17, pp.163-191, pp.235-257 (1982)

(Despryroux 83)
J.Despryroux : An algebraic specification of a Pascal compiler", SIGPLAN Notice, 18, 12, pp.34-48 (1983)

(Donahue 76)
J. Donahue : "Complementary definitions of programming language semantics", Lecture Notes in Comput.Sci., 42 (1976)

(Gaudel 80)
M.C.Gaudel : "Specification of compilers as abstract data type representations", Proc. of Workshop on Semantics Directed Compiler Generation, Aarhus, Lecture Notes in Comput. Sci., 94, pp.140-164 (1980)

(Goguen 80)
J.A.Goguen : "OBJ-1, A study in executable algebtraic formal spcification", Technical Report, SRI International (1980)

(Goguen, Parsaye-Ghomi 81)
J.A.Goguen, K.Parsaye-Ghomi : "Algebraic denotational semantics using parameterized abstract modules", Lecture Notes in Comput. Sci., 107, pp292-309 (1981)

(Hoare 69)
C. A. R. Hoare : "An axiomatic basis for computer programming", Comm. of ACM, 21 (1969)

(Horowitz 83)
E. Horowitz : "Fundamentals of programming languages", pp.169-178, Computer Science Press (1983)

(Kita, Sakbe, Inagaki 84)
H. Kita, T. Sakabe, Y. Inagaki : "Algebraic spcefication method of progamming languages", Lecture Notes in Comput. Sci., 220, pp144-157 (1984)

(Knuth 68)
D. E. Knuth : "Semantics of context-free languages", Math. Systems Thory, 2, pp127-145 (1968), correction, 5, pp99-96 (1971)

(Lucas, Walk 71)
P. Lucas, K. Walk : "On the formal definition of PL/I", M. Halpern and C. shaw(eds.), Annual Review in Automatic Programming 6, pp.105-182, Pergamon Press (1971)

(Mosses 80)
P.Mosses : "A constructive approach to compiler correctness", Proc. ofWorkshop on Semantics Directed Compiler Generation, Aarhus, Lecture Notes in Comput. Sci., 94, pp.189-210 (1980)

(Shoenfield 67)
J. R. Shoenfield : "Mathematical logic", Addison-Wesiy (1967)

(Tennent 76)
R. D. Tennent : "The denotational semantics of programming languages", Comm. of ACM, 19, pp437-453 (1976)
(Pair 82)
C.Pair : "Abstract data types and algebraic semantics of programming languages", Theor. Compt. Sci., 18, pp.1-31 (1982)

(Wirth 76)
N. Wirth : "Algorithms + Data Structures = Programs", Prentice-Hall (1976)

文献（和文）

（稲垣，北，酒井，坂部 85）
稲垣，北，酒井，坂部：『プログラミング言語の仕様記述法と処理系の自動生成』，情報処理学会，「知識情報処理」シンポジウム（１９８５）

（北，坂部，稲垣，本多 83）
北，坂部，稲垣，本多：『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）

（稲垣，坂部 84）
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）～（４）』，情報処理，２５，Ｎｏ．１，Ｎｏ．５，Ｎｏ．７，Ｎｏ．９（１９８４）

（北，坂部，稲垣 85）
北，坂部，稲垣：『プログラミング言語ＰＬ／０の代数的仕様記述』，信学技報，ＡＬ８４－５６（１９８５）

（北，坂部，稲垣 86）
北，坂部，稲垣：『代数的に仕様記述された言語のプログラム検証』，信学技法，ＳＳ８６－１０（１９８６）

（北，坂部，稲垣 87）
北，坂部，稲垣：『プログラミング言語の代数的意味論に対する公理的検証体系の無矛盾性』，信学技報，ＣＯＭＰ８６－６６（１９８７）

（北，坂部，稲垣 87）
北，坂部，稲垣：『プログラミング言語の代数的仕様記述法』，信学論（Ｄ），Ｊ７０－Ｄ，２，ｐｐ．２４７－２５８（１９８７）

（北，坂部，稲垣，高木 87）
北，坂部，稲垣，高木：『Ｐａｓｃａｌ風言語ＰＬ／０＋の代数的仕様記述』，信学論（Ｄ），Ｊ７０－Ｄ，１２，ｐｐ．２４２８－２４３７（１９８７）

（北，坂部，稲垣 87）
北，坂部，稲垣：『代数的手法によるプログラムの検証』，人工知能学会誌，２，３，ｐｐ．１１９－１２２（１９８７）

（北，坂部，稲垣 88）
北，坂部，稲垣：『プログラミング言語の代数的意味論に対する公理的検証体系の健全性』，信学論（Ｄ），採録予定

（川辺，坂部，稲垣 84）
川辺，坂部，稲垣：『抽象データ型の直接実現システム』，信学技報，ＡＬ８３－６５（１９８４）

（酒井，北，坂部，稲垣 85）
酒井，北，坂部，稲垣：『プログラミング言語の代数的仕様記述からのコンパイラ自動生成』，信学技報，ＡＬ８５－１０（１９８５）

（酒井，坂部，稲垣 86）
酒井，坂部，稲垣：『コンパイラの代数的仕様記述法』，信学技法，ＳＳ８６－９（１９８６）

（酒井，坂部，稲垣 87）
酒井，坂部，稲垣：『抽象データ型の直接実現系Ｃｄｉｍｐｌｅ』，信学技報，ＣＯＭＰ８６－６７（１９８７）

（高木，北，坂部，稲垣 85）
高木，北，坂部，稲垣：『入出力機能とパラメタ付手続きを付加したプログラミング言語ＰＬ／０の代数的仕様記述』，信学技報，ＡＬ８５－４４（１９８５）

（中島 82）
中島：『情報処理入門』，数理科学ライブラリ３，朝倉書店（１９８２）

（二木，外山 83）
二木，外山：『項書き換え型計算モデルとその応用』，情報処理，２４，２（１９８３）

付録　ＰＬ／０＋の仕様

第３章で，プログラミング言語の代数的仕様記述の説明のために用いたＰａｓｃａｌ風言語ＰＬ／０＋の全仕様を付録として掲載する．ただし，以下の仕様は，言語開発支援システムＬａｓｓ（酒井，北，坂部，稲垣 85）の書式で記述してある．

このプログラミング言語ＰＬ／０＋の仕様は，ＰＬ／０の仕様を順次拡張して書かれたものである．また，Ｌａｓｓの書式で書かれているので，この仕様からＬａｓｓを用いて，ＰＬ／０＋のコンパイラと，そのコンパイラの出力コードを実行する仮想機械を自動的に生成することができる．

```
spechead{{
 @(#) Module Name        pl0+.la
 @(#) Identifier         1.5
 @(#) Date               89/02/28
 @(#) SCCS File          /usr1/usrroots/hkita/nu/lass/PL0+/s.pl0+.la
 @(#) Writer             kita
 @(#) Module Type        lass
   PL/0.1 = PL/0 + input/output function, takagi, ver-1.2, 22/Feb/86
   PL/0.2 = PL/0.1 + input/output function, takagi, ver-1.2, 22/Feb/86
   PL/0.2+ = PL/0.2 + 1-dimension array
   PL/0+.la, kita, ver-0.1, 1/Dec/86
   We renamed PL/0.2+ to PL/0+ for simplicity.
   And changed the static scoping rule of identifiers from dynamic one.
   example
        var x, y:array, z ; (* x,z are integers, y is array *)
}}

basesort{{
  id,nat,int,bool
}}

baseoper{{
  ZERO          :                -> int ;
  NEG           : nat            -> int ;
  POS           : nat            -> int ;
  MINUS_INT     : int            -> int ;
  INIT_ID       :                -> id ;
  EQUAL_ID      : id,id          -> bool ;
  NOT_BOOL      : bool           -> bool ;
  OR_BOOL       : bool,bool      -> bool ;
  AND_BOOL      : bool,bool      -> bool ;
  SUM_INT       : int,int        -> int ;
  DIFF_INT      : int,int        -> int ;
  MULT_INT      : int,int        -> int ;
  DIV_INT       : int,int        -> int ;
  EQ_INT        : int,int        -> bool ;
  LT_INT        : int,int        -> bool ;
  GT_INT        : int,int        -> bool ;
  ODDP_INT      : int            -> bool ;
  UNDEF_INT     :                -> int ;
  TRUE          :                -> bool ;
  FALSE         :                -> bool ;
  INPUT         :                -> id ;
  OUTPUT        :                -> id ;
}}

newsort{{
  state,        state-state,    state-int,      state-bool,
  file,         state-file,     file-file,
  list,         state-list,     expr,
  state-attr,   attr,           state-state-state,
  array,        state-array
}}

newoper{{
  INIT_STATE            :                       -> state ;
```

```
I_STATE-STATE          :                        -> state-state ;
APPLY_STATE            : state-state,state      -> state ;
APPLY_STATE_D          : 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               : id,attr                -> state-state ;
UPDATE_D               : id,state-attr          -> state-state ;
RETRIEVE_D             : id                     -> state-attr  ;
SUM_INT_D              : state-int,state-int    -> state-int ;
DIFF_INT_D             : state-int,state-int    -> state-int ;
MULT_INT_D             : state-int,state-int    -> state-int ;
DIV_INT_D              : state-int,state-int    -> state-int ;
MINUS_INT_D            : state-int              -> state-int ;
EQ_INT_D               : state-int,state-int    -> state-bool ;
LT_INT_D               : state-int,state-int    -> state-bool ;
GT_INT_D               : state-int,state-int    -> state-bool ;
ENTER_BLOCK_D          : state-int              -> state-state ;
LEAVE_BLOCK_D          : state-state            -> state-state ;
ODDP_INT_D             : state-int              -> state-bool ;
NOT_BOOL_D             : state-bool             -> state-bool ;
OR_BOOL_D              : state-bool,state-bool -> state-bool ;
AND_BOOL_D             : state-bool,state-bool -> state-bool ;
MAKE_STATE-INT         : int                    -> state-int ;
MAKE_INT_D             : state-attr             -> state-int  ;
MAKE_ATTR_VAR_D        : state-int              -> state-attr ;
MAKE_ATTR_CONST        : int                    -> attr ;
MAKE_ATTR_VAR          : int                    -> attr ;
ADD_ID                 : state,id,attr          -> state ;
UPDATE                 : state,id,attr          -> state ;
RETRIEVE               : state,id               -> attr ;
ENTER_BLOCK            : int,state              -> state ;
LEAVE_BLOCK            : state                  -> state ;
APPLY_STATE_INT        : state-int,state                -> int ;
APPLY_STATE_BOOL       : state-bool,state               -> bool ;
APPLY_STATE_STATE-STATE : state-state-state,state       -> state-state ;
APPLY_STATE_ATTR       : state-attr,state               -> attr ;
MAKE_INT               : attr          -> int         ;
UNDEF_ATTR             :               -> attr ;
EMPTY_FILE             :               -> file ;
READ_FILE              : file          -> int ;
WRITE_FILE             : file,int      -> file ;
REMOVE_DATA            : file          -> file ;
EOF                    : file          -> bool ;
READ_FILE_D            : state-file    -> state-int ;
WRITE_FILE_D           : state-file,state-int  -> state-file ;
REMOVE_DATA_D          : state-file    -> state-file ;
EOF_D                  : state-file    -> state-bool ;
MAKE_FILE              : attr          -> file ;
MAKE_ATTR_FILE         : file          -> attr ;
MAKE_FILE_D            : state-attr    -> state-file ;
MAKE_ATTR_FILE_D       : state-file    -> state-attr ;
```

```
    APPLY_STATE_FILE       : state-file,state      -> file ;
    PROGRAM                : state-state   -> file-file ;
    INITIALIZE             : file          -> state ;
    RUN                    : file-file,file        -> file ;
    LIST_C                 : attr,list     -> list ;
    INIT_LIST              :               -> list ;
    TAKE_OUT_LIST          : list          -> attr ;
    DELETE_LIST            : list          -> list ;
    APPEND_LIST            : list,list     -> list ;
    MAKE_ATTR_PROC-LIST    : state-state,list      -> attr ;
    MAKE_PROC              : attr          -> state-state ;
    MAKE_LIST              : attr          -> list ;
    MAKE_ID                : attr          -> id ;
    MAKE_ATTR_VAL_PARA     : id            -> attr ;
    MAKE_ATTR_VAR_PARA     : id            -> attr ;
    MAKE_ATTR_ID           : id            -> attr ;
    IS_ID                  : attr          -> bool ;
    RETRIEVE_OUT           : state,id      -> attr ;
    UPDATE_OUT             : state,id,attr -> state ;
    MAKE_PROC_D            : state-attr    -> state-state-state ;
    MAKE_LIST_D            : state-attr    -> state-list ;
    ADD_ID_LIST_D          : state-list,list       -> state-state ;
    ADD_ID_LIST            : state,list,list       -> state ;
    APPLY_STATE_LIST       : state-list,state      -> list ;
    CALC                   : expr,state    -> attr ;
    MAKE_EXPR_ID           : id            -> expr ;
    MAKE_EXPR_INT          : int           -> expr ;
    SUM_INT_S              : expr,expr     -> expr ;
    DIFF_INT_S             : expr,expr     -> expr ;
    MULT_INT_S             : expr,expr     -> expr ;
    DIV_INT_S              : expr,expr     -> expr ;
    MINUS_INT_S            : expr          -> expr ;
    MAKE_ID_EXPR           : expr          -> id ;
    MAKE_ATTR_EXPR         : expr          -> attr ;
    MAKE_EXPR              : attr          -> expr ;
    REPEAT                 : state-bool,state-state  -> state-state ;
    ACCESS_ARRAY           : array,int     -> int ;
    ARRAY_C                : array,int,int -> array ;
    INIT_ARRAY             :               -> array ;
    MAKE_ATTR_ARRAY        : array         -> attr ;
    MAKE_ARRAY             : attr          -> array ;
    ACCESS_ARRAY_D         : state-array,state-int         -> state-int ;
    MAKE_ARRAY_D           : state-attr    -> state-array ;
    MAKE_ATTR_ARRAY_D      : state-array   -> state-attr ;
    ARRAY_C_D              : state-array,state-int,state-int
                                   -> state-array ;
    MAKE_EXPR_EMT          : id,expr       -> expr ;
    APPLY_STATE_ARRAY      : state-array,state     -> array ;
    MAKE_ATTR_EMT_ID       : id,int        -> attr ;
    IS_EMT_ID              : attr          -> bool ;
    LEAVE_N_BLOCK          : int,state             -> state ;
    UPDATE_DOWN_BLOCK      : int,state,id,attr     -> state ;
    FIND_BLOCK             : id,state              -> int ;
    FIND_BLOCK_D           : id                    -> state-int ;
  }}
```

```
variable{{
  state0                  : state ;
  state1                  : state ;
  state-state0            : state-state ;
  state-state1            : state-state ;
  state-bool0             : state-bool ;
  state-bool1             : state-bool ;
  state-int0              : state-int ;
  state-int1              : state-int ;
  state-state-state0      : state-state-state ;
  state-state-state1      : state-state-state ;
  state-attr0             : state-attr ;
  attr0                   : attr ;
  attr1                   : attr ;
  int0                    : int ;
  int1                    : int ;
  int2                    : int ;
  bool0                   : bool ;
  id0                     : id ;
  id1                     : id ;
  id2                     : id ;
  id3                     : id ;
  file0                   : file ;
  state-file0             : state-file ;
  state-list0             : state-list ;
  state-list1             : state-list ;
  list0                   : list ;
  list1                   : list ;
  expr0                   : expr ;
  expr1                   : expr ;
  array0                  : array ;
  state-array0            : state-array ;
}}

axiom{{
    LEAVE_N_BLOCK(int0,state0)
 == IF( EQ_INT(ZERO(),int0),
        state0,
        LEAVE_N_BLOCK(DIFF_INT(int0,ONE()),LEAVE_BLOCK(state0))) ;

    FIND_BLOCK(id0,INIT_STATE())
 == UNDEF_INT() ;

    FIND_BLOCK(id0,ENTER_BLOCK(int0,state0))
 == SUM_INT( FIND_BLOCK(id0,LEAVE_N_BLOCK(DIFF_INT(int0,ONE()),state0)),
             int0 ) ;

    FIND_BLOCK(id1,ADD_ID(state0,id0,attr0))
 == IF( EQUAL_ID(id0,id1),
        ZERO(),
        FIND_BLOCK(id1,state0)) ;

    RETRIEVE(INIT_STATE(),id0)
 == UNDEF_ATTR() ;

    RETRIEVE(ENTER_BLOCK(int0,state0),id0)
```

```
== RETRIEVE(LEAVE_N_BLOCK(DIFF_INT(int0,ONE()),state0),id0) ;

   RETRIEVE(ADD_ID(state0,id0,attr0),id1)
== IF( EQUAL_ID(id0,id1),
       attr0,
       RETRIEVE(state0,id1)) ;

   UPDATE(INIT_STATE(),id0,attr0)
== INIT_STATE() ;

   UPDATE(ENTER_BLOCK(int0,state0),id0,attr0)
== IF( EQ_INT(int0,ONE()),
       ENTER_BLOCK(int0,UPDATE(state0,id0,attr0)),
       ENTER_BLOCK(int0,
                   UPDATE_DOWN_BLOCK(DIFF_INT(int0,ONE()),
                                     state0,id0,attr0))) ;

   UPDATE_DOWN_BLOCK(int0,INIT_STATE(),id0,attr0)
== INIT_STATE() ;

   UPDATE_DOWN_BLOCK(int1,ENTER_BLOCK(int0,state0),id1,attr1)
== IF( EQ_INT(int1,ONE()),
       ENTER_BLOCK(int0,UPDATE(state0,id1,attr1)),
       ENTER_BLOCK(
          int0,
          UPDATE_DOWN_BLOCK(DIFF_INT(int1,ONE()),state0,id1,attr1))) ;

   UPDATE_DOWN_BLOCK(int0,ADD_ID(state1,id1,attr1),id0,attr0)
== ADD_ID(UPDATE_DOWN_BLOCK(int0,state1,id0,attr0),id1,attr1) ;

   UPDATE(ADD_ID(state0,id0,MAKE_ATTR_VAR(int0)),id1,attr1)
== IF( EQUAL_ID(id0,id1),
       ADD_ID(state0,id0,attr1),
       ADD_ID(UPDATE(state0,id1,attr1),id0,MAKE_ATTR_VAR(int0))) ;

   UPDATE(ADD_ID(state0,id0,MAKE_ATTR_CONST(int0)),id1,attr1)
== IF( EQUAL_ID(id0,id1),
       ADD_ID(state0,id0,MAKE_ATTR_CONST(int0)),
       ADD_ID(UPDATE(state0,id1,attr1),id0,MAKE_ATTR_CONST(int0))) ;

   UPDATE(ADD_ID(state0,id0,MAKE_ATTR_PROC(state-state0)),id1,attr1)
== IF( EQUAL_ID(id0,id1),
       ADD_ID(state0,id0,MAKE_ATTR_PROC(state-state0)),
       ADD_ID(UPDATE(state0,id1,attr1),id0,
              MAKE_ATTR_PROC(state-state0))) ;

   LEAVE_BLOCK(INIT_STATE())
== INIT_STATE() ;

   APPLY_STATE(ENTER_BLOCK_D(state-int0),state0)
== ENTER_BLOCK(APPLY_STATE-INT(state-int0,state0),state0) ;

   APPLY_STATE-INT(FIND_BLOCK_D(id0),state0)
== FIND_BLOCK(id0,state0) ;

   COMPOSITION(I_STATE-STATE(),state-state0)
```

```
== state-state0 ;

   COMPOSITION(state-state0,I_STATE-STATE())
== state-state0 ;

   APPLY_STATE(I_STATE-STATE(),state0)
== state0 ;

   APPLY_STATE(IF_STATE_D(state-bool0,state-state0,state-state1),state0)
== IF( 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( 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(id0,attr0),state0)
== ADD_ID(state0,id0,attr0) ;

   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(LEAVE_BLOCK_D(state-state0),state0)
== LEAVE_BLOCK(APPLY_STATE(state-state0,state0)) ;

   APPLY_STATE(UPDATE_D(id0,state-attr0),state0)
== UPDATE(state0,
          id0,
          APPLY_STATE_ATTR(state-attr0,state0)) ;

   APPLY_STATE_INT(SUM_INT_D(state-int0,state-int1),state0)
== SUM_INT(APPLY_STATE_INT(state-int0,state0),
           APPLY_STATE_INT(state-int1,state0)) ;

   APPLY_STATE_INT(DIFF_INT_D(state-int0,state-int1),state0)
== DIFF_INT(APPLY_STATE_INT(state-int0,state0),
            APPLY_STATE_INT(state-int1,state0)) ;

   APPLY_STATE_INT(MULT_INT_D(state-int0,state-int1),state0)
== MULT_INT(APPLY_STATE_INT(state-int0,state0),
            APPLY_STATE_INT(state-int1,state0)) ;

   APPLY_STATE_INT(DIV_INT_D(state-int0,state-int1),state0)
== DIV_INT(APPLY_STATE_INT(state-int0,state0),
           APPLY_STATE_INT(state-int1,state0)) ;

   APPLY_STATE_INT(MAKE_STATE-INT(int0),state0)
== int0 ;
```

```
   APPLY_STATE_INT(MINUS_INT_D(state-int0),state0)
== MINUS_INT(APPLY_STATE_INT(state-int0,state0)) ;

   APPLY_STATE_INT(MAKE_INT_D(state-attr0),state0)
== MAKE_INT(APPLY_STATE_ATTR(state-attr0,state0)) ;

   APPLY_STATE_BOOL(GT_INT_D(state-int0,state-int1),state0)
== GT_INT(APPLY_STATE_INT(state-int0,state0),
          APPLY_STATE_INT(state-int1,state0)) ;

   APPLY_STATE_BOOL(EQ_INT_D(state-int0,state-int1),state0)
== EQ_INT(APPLY_STATE_INT(state-int0,state0),
          APPLY_STATE_INT(state-int1,state0)) ;

   APPLY_STATE_BOOL(LT_INT_D(state-int0,state-int1),state0)
== LT_INT(APPLY_STATE_INT(state-int0,state0),
          APPLY_STATE_INT(state-int1,state0)) ;

   APPLY_STATE_BOOL(NOT_BOOL_D(state-bool0),state0)
== NOT_BOOL(APPLY_STATE_BOOL(state-bool0,state0)) ;

   APPLY_STATE_BOOL(OR_BOOL_D(state-bool0,state-bool1),state0)
== OR_BOOL(APPLY_STATE_BOOL(state-bool0,state0),
           APPLY_STATE_BOOL(state-bool1,state0)) ;

   APPLY_STATE_BOOL(AND_BOOL_D(state-bool0,state-bool1),state0)
== AND_BOOL(APPLY_STATE_BOOL(state-bool0,state0),
            APPLY_STATE_BOOL(state-bool1,state0)) ;

   APPLY_STATE_ATTR(RETRIEVE_D(id0),state0)
== RETRIEVE(state0,id0) ;

   APPLY_STATE_ATTR(MAKE_ATTR_VAR_D(state-int0),state0)
== MAKE_ATTR_VAR(APPLY_STATE_INT(state-int0,state0)) ;

   MAKE_INT(MAKE_ATTR_CONST(int0))
== int0 ;

   MAKE_INT(MAKE_ATTR_VAR(int0))
== int0 ;

   MAKE_INT(UNDEF_ATTR())
== ZERO() ;

   REMOVE_DATA(EMPTY_FILE())
== EMPTY_FILE() ;

   READ_FILE(EMPTY_FILE())
== UNDEF_INT() ;

   READ_FILE(WRITE_FILE(EMPTY_FILE(),int0))
== int0 ;

   READ_FILE(WRITE_FILE(WRITE_FILE(file0,int0),int1))
== READ_FILE(WRITE_FILE(file0,int0)) ;
```

```
   REMOVE_DATA(WRITE_FILE(EMPTY_FILE(),int0))
== EMPTY_FILE() ;

   REMOVE_DATA(WRITE_FILE(WRITE_FILE(file0,int0),int1))
== WRITE_FILE(REMOVE_DATA(WRITE_FILE(file0,int0)),int1) ;

   EOF(EMPTY_FILE())
== TRUE() ;

   EOF(WRITE_FILE(file0,int0))
== FALSE() ;

   APPLY_STATE_BOOL(EOF_D(state-file0),state0)
== EOF(APPLY_STATE_FILE(state-file0,state0)) ;

   APPLY_STATE_INT(READ_FILE_D(state-file0),state0)
== READ_FILE(APPLY_STATE_FILE(state-file0,state0)) ;

   APPLY_STATE_FILE(WRITE_FILE_D(state-file0,state-int0),state0)
== WRITE_FILE(APPLY_STATE_FILE(state-file0,state0),
              APPLY_STATE_INT(state-int0,state0)) ;

   APPLY_STATE_FILE(REMOVE_DATA_D(state-file0),state0)
== REMOVE_DATA(APPLY_STATE_FILE(state-file0,state0)) ;

   MAKE_FILE(MAKE_ATTR_FILE(file0))
== file0 ;

   APPLY_STATE_FILE(MAKE_FILE_D(state-attr0),state0)
== MAKE_FILE(APPLY_STATE_ATTR(state-attr0,state0)) ;

   APPLY_STATE_ATTR(MAKE_ATTR_FILE_D(state-file0),state0)
== MAKE_ATTR_FILE(APPLY_STATE_FILE(state-file0,state0)) ;

   INITIALIZE(file0)
== ADD_ID(ADD_ID(INIT_STATE(),
                 INPUT(),
                 MAKE_ATTR_FILE(file0)),
          OUTPUT(),
          MAKE_ATTR_FILE(EMPTY_FILE())) ;

   RUN(PROGRAM(state-state0),file0)
== MAKE_FILE(RETRIEVE(APPLY_STATE(state-state0,
                                  INITIALIZE(file0)),
                      OUTPUT())) ;

   TAKE_OUT_LIST(LIST_C(attr0,list0))
== attr0 ;

   DELETE_LIST(LIST_C(attr0,list0))
== list0 ;

   TAKE_OUT_LIST(INIT_LIST())
== UNDEF_ATTR() ;
```

```
   DELETE_LIST(INIT_LIST())
== INIT_LIST() ;

   APPEND_LIST(LIST_C(attr0,list0),list1)
== LIST_C(attr0,APPEND_LIST(list0,list1)) ;

   APPEND_LIST(INIT_LIST(),list1)
== list1 ;

   MAKE_PROC(MAKE_ATTR_PROC-LIST(state-state0,list0))
== state-state0 ;

   MAKE_LIST(MAKE_ATTR_PROC-LIST(state-state0,list0))
== list0 ;

   APPLY_STATE_STATE-STATE(MAKE_PROC_D(state-attr0),state0)
== MAKE_PROC(APPLY_STATE_ATTR(state-attr0,state0)) ;

   APPLY_STATE_LIST(MAKE_LIST_D(state-attr0),state0)
== MAKE_LIST(APPLY_STATE_ATTR(state-attr0,state0)) ;

   APPLY_STATE(ADD_ID_LIST_D(state-list0,list1),state0)
== ADD_ID_LIST(state0,
               APPLY_STATE_LIST(state-list0,state0),
               list1) ;

   ADD_ID_LIST(state0,INIT_LIST(),list0)
== state0 ;

   ADD_ID_LIST(state0,
               LIST_C(MAKE_ATTR_VAL_PARA(id0),list0),
               list1)
== ADD_ID_LIST(ADD_ID(state0,
                      id0,
                      CALC(MAKE_EXPR(TAKE_OUT_LIST(list1)),
                           state0)),
               list0,
               DELETE_LIST(list1)) ;

   ADD_ID_LIST(state0,
               LIST_C(MAKE_ATTR_VAR_PARA(id0),list0),
               LIST_C(MAKE_ATTR_EXPR(MAKE_EXPR_ID(id1)),list1))
== ADD_ID_LIST(ADD_ID(state0,
                      id0,
                      MAKE_ATTR_ID(id1)),
               list0,
               list1) ;

   ADD_ID_LIST(state0,
               LIST_C(MAKE_ATTR_VAR_PARA(id0),list0),
               LIST_C(MAKE_ATTR_EXPR(MAKE_EXPR_EMT(id1,expr1)),
                      list1))
== ADD_ID_LIST(ADD_ID(state0,
                      id0,
                      MAKE_ATTR_EMT_ID(id1,
                                       MAKE_INT(CALC(expr1,state0)))),
```

```
                    list0,
                    list1) ;

    CALC(SUM_INT_S(expr0,expr1),state0)
 == MAKE_ATTR_VAR(SUM_INT(MAKE_INT(CALC(expr0,state0)),
                          MAKE_INT(CALC(expr1,state0)))) ;

    CALC(DIFF_INT_S(expr0,expr1),state0)
 == MAKE_ATTR_VAR(DIFF_INT(MAKE_INT(CALC(expr0,state0)),
                           MAKE_INT(CALC(expr1,state0)))) ;

    CALC(MULT_INT_S(expr0,expr1),state0)
 == MAKE_ATTR_VAR(MULT_INT(MAKE_INT(CALC(expr0,state0)),
                           MAKE_INT(CALC(expr1,state0)))) ;

    CALC(DIV_INT_S(expr0,expr1),state0)
 == MAKE_ATTR_VAR(DIV_INT(MAKE_INT(CALC(expr0,state0)),
                          MAKE_INT(CALC(expr1,state0)))) ;

    CALC(MINUS_INT_S(expr0),state0)
 == MAKE_ATTR_VAR(MINUS_INT(MAKE_INT(CALC(expr0,state0)))) ;

    CALC(MAKE_EXPR_ID(id0),state0)
 == RETRIEVE_OUT(state0,id0) ;

    CALC(MAKE_EXPR_INT(int0),state0)
 == MAKE_ATTR_VAR(int0) ;

    CALC(MAKE_EXPR_EMT(id0,expr0),state0)
 == MAKE_ATTR_VAR(
      ACCESS_ARRAY(MAKE_ARRAY(RETRIEVE_OUT(state0,id0)),
                   MAKE_INT(CALC(expr0,state0)))) ;

    MAKE_ID_EXPR(MAKE_EXPR_ID(id0))
 == id0 ;

    MAKE_EXPR(MAKE_ATTR_EXPR(expr0))
 == expr0 ;

    MAKE_ID(MAKE_ATTR_ID(id0))
 == id0 ;

    IS_ID(MAKE_ATTR_ID(id0))
 == TRUE() ;

    IS_ID(MAKE_ATTR_VAR(int0))
 == FALSE() ;

    IS_ID(UNDEF_ATTR())
 == FALSE() ;

    IS_ID(MAKE_ATTR_CONST(int0))
 == FALSE() ;

    IS_ID(MAKE_ATTR_PROC-LIST(state-state0,list0))
 == FALSE() ;
```

```
    IS_ID(MAKE_ATTR_FILE(file0))
 == FALSE() ;

    IS_ID(MAKE_ATTR_EMT_ID(id0,int0))
 == FALSE() ;

    IS_ID(MAKE_ATTR_ARRAY(array0))
 == FALSE() ;

    IS_EMT_ID(MAKE_ATTR_ARRAY(array0))
 == FALSE() ;

    IS_EMT_ID(MAKE_ATTR_ID(id0))
 == FALSE() ;

    IS_EMT_ID(MAKE_ATTR_EMT_ID(id0,int0))
 == TRUE() ;

    IS_EMT_ID(MAKE_ATTR_VAR(int0))
 == FALSE() ;

    IS_EMT_ID(UNDEF_ATTR())
 == FALSE() ;

    IS_EMT_ID(MAKE_ATTR_CONST(int0))
 == FALSE() ;

    IS_EMT_ID(MAKE_ATTR_PROC-LIST(state-state0,list0))
 == FALSE() ;

    IS_EMT_ID(MAKE_ATTR_FILE(file0))
 == FALSE() ;

    RETRIEVE_OUT(ADD_ID(state0,id0,attr0),id1)
 == RETRIEVE_OUT(state0,id1) ;

    RETRIEVE_OUT(ENTER_BLOCK(state0),id0)
 == RETRIEVE(state0,id0) ;

    UPDATE_OUT(ADD_ID(state0,id0,attr0),id1,attr1)
 == ADD_ID(UPDATE_OUT(state0,id1,attr1),id0,attr0) ;

    UPDATE_OUT(ENTER_BLOCK(state0),id0,attr0)
 == ENTER_BLOCK(UPDATE(state0,id0,attr0)) ;

    APPLY_STATE(REPEAT(state-bool0,state-state0),state0)
 == IF(APPLY_STATE_BOOL(state-bool0,APPLY_STATE(state-state0,state0)),
          APPLY_STATE(state-state0,state0),
          APPLY_STATE(COMPOSITION(REPEAT(state-bool0,state-state0),
                                  state-state0),state0)) ;

    ACCESS_ARRAY(ARRAY_C(array0,int0,int1),int2)
 == IF(EQ_INT(int0,int2),
          int1,
          ACCESS_ARRAY(array0,int2)) ;
```

```
    ACCESS_ARRAY(INIT_ARRAY(),int2)
 == UNDEF_INT() ;

    MAKE_ARRAY(MAKE_ATTR_ARRAY(array0))
 == array0 ;

    APPLY_STATE_INT(ACCESS_ARRAY_D(state-array0,state-int0),state0)
 == ACCESS_ARRAY(APPLY_STATE_ARRAY(state-array0,state0),
                 APPLY_STATE_INT(state-int0,state0)) ;

    APPLY_STATE_ARRAY(ARRAY_C_D(state-array0,state-int0,state-int1),
                      state0)
 == ARRAY_C(APPLY_STATE_ARRAY(state-array0,state0),
            APPLY_STATE_INT(state-int0,state0),
            APPLY_STATE_INT(state-int1,state0)) ;

    APPLY_STATE_ARRAY(MAKE_ARRAY_D(state-attr0),state0)
 == MAKE_ARRAY(APPLY_STATE_ATTR(state-attr0,state0)) ;

    APPLY_STATE_ATTR(MAKE_ATTR_ARRAY_D(state-array0),state0)
 == MAKE_ATTR_ARRAY(APPLY_STATE_ARRAY(state-array0,state0)) ;

    MAKE_ID(MAKE_ATTR_EMT_ID(id0,int0))
 == id0 ;

    MAKE_INT(MAKE_ATTR_EMT_ID(id0,int0))
 == int0 ;

    MAKE_INT(MAKE_ATTR_FILE(file0))
 == ZERO() ;

    MAKE_FILE(UNDEF_ATTR())
 == EMPTY_FILE() ;

    MAKE_FILE(MAKE_ATTR_CONST(int0))
 == EMPTY_FILE() ;

    MAKE_FILE(MAKE_ATTR_VAR(int0))
 == EMPTY_FILE() ;

    MAKE_INT(MAKE_ATTR_PROC-LIST(state-state0,list0))
 == ZERO() ;

    MAKE_FILE(MAKE_ATTR_PROC-LIST(state-state0,list0))
 == EMPTY_FILE() ;

    MAKE_LIST(UNDEF_ATTR())
 == INIT_LIST() ;

    MAKE_PROC(UNDEF_ATTR())
 == I_STATE-STATE() ;

    MAKE_LIST(MAKE_ATTR_CONST(int0))
 == INIT_LIST() ;
```

```
      MAKE_PROC(MAKE_ATTR_CONST(int0))
    == I_STATE-STATE() ;

      MAKE_LIST(MAKE_ATTR_VAR(int0))
    == INIT_LIST() ;

      MAKE_PROC(MAKE_ATTR_VAR(int0))
    == I_STATE-STATE() ;

      MAKE_LIST(MAKE_ATTR_FILE(file0))
    == INIT_LIST() ;

      MAKE_PROC(MAKE_ATTR_FILE(file0))
    == I_STATE-STATE() ;
  }}

  define{{
    LETTER          = [a-zA-Z] ;
    DIGIT           = [0-9] ;
    LD              = ({LETTER}|{DIGIT}|[_]) ;
  }}

  terminal{{
  begin             =  BEGINN    ;
  end               =  END       ;
  if                =  IF        ;
  then              =  THEN      ;
  else              =  ELSE      ;
  procedure         =  PROCEDURE ;
  call              =  CALL      ;
  while             =  WHILE     ;
  do                =  DO        ;
  odd               =  ODD       ;
  var               =  VAR       ;
  const             =  CONST     ;
  read              =  READ      ;
  write             =  WRITE     ;
  eof               =  EOFF      ;
  repeat            =  REPEAT    ;
  until             =  UNTIL     ;
  array             =  ARRAY     ;
  "<="              =  LE        ;
  ">="              =  GE        ;
  "<>"              =  NE        ;
  ":="              =  ASSIGN    ;
  '+'               =  '+'       ;
  '*'               =  '*'       ;
  ';'               =  ';'       ;
  '-'               =  '-'       ;
  '/'               =  '/'       ;
  '('               =  '('       ;
  ')'               =  ')'       ;
  '='               =  '='       ;
  '>'               =  '>'       ;
  ','               =  ','       ;
  '<'               =  '<'       ;
```

```
'.'                 =  '.'        ;
':'                 =  ':'        ;
'['                 =  '['        ;
']'                 =  ']'        ;
{LETTER}{LD}*       =  ID         //  ID : id
                                  //  ID($0) = quote($0) ;
{DIGIT}+            =  ICONST     //  ICONST : nat
                                  //  ICONST($0) = quote($0) ;
"(*"                              :combegin;
"*)"                              :comend;
}}

prec{{
LE,GE,NE,'=','>','<'      :left;
'+','-'                   :left;
'*','/'                   :left;
ANARY                     :left;
}}

nonterminal{{
  program                 : file-file;
  block                   : state-state;
  const_def_part          : state-state;
  const_def_list          : state-state;
  const_def               : state-state;
  var_dcl_part            : state-state;
  var_name_list           : state-state;
  var_name                : state-state;
  proc_dcl_part           : state-state;
  proc_dcl_list           : state-state;
  proc_dcl                : state-state;
  proc_para_section_list : list ;
  proc_para_section       : list ;
  proc_value_para_list    : list ;
  proc_var_para_list      : list ;
  proc_value_para         : attr ;
  proc_var_para           : attr ;
  statement               : state-state;
  statement_list          : state-state;
  write_para_list         : state-state;
  write_para              : state-state;
  read_para_list          : state-state;
  read_para               : state-state;
  call_para_list          : list;
  call_para               : attr;
  expr_sym                : expr;
  condition               : state-bool;
  expression              : state-int;
  number                  : int;
  ident                   : id;
  nat_number              : nat;
}}

rule{{
  p010: program           ::= block '.'
```

```
                // program[ p010($0) ]
                   = PROGRAM( block[ $0 ] )  ;

p020: block           ::= const_def_part var_dcl_part
                          proc_dcl_part statement

                // block[ p020($0,$1,$2,$3) ]
                   = COMPOSITION(
                         statement[ $3 ],
                         COMPOSITION(
                            proc_dcl_part[ $2 ],
                            COMPOSITION(
                               var_dcl_part[ $1 ],
                               COMPOSITION(
                                   const_def_part[ $0 ],
                                   I_STATE-STATE() ))))  ;

p030: const_def_part  ::= CONST       const_def_list ';'

                // const_def_part[ p030($1) ] = const_def_list[ $1 ]  ;

p040: const_def_part  ::=

                // const_def_part[p040()] = I_STATE-STATE()  ;

p060: const_def_list  ::= const_def

                // const_def_list[ p060($0) ] = const_def[ $0 ]  ;

p070: const_def_list  ::= const_def_list ',' const_def

                // const_def_list[ p070($0,$2) ]
                     = COMPOSITION( const_def[ $2 ],
                                    const_def_list[ $0 ] )  ;

p080: const_def       ::= ident '=' number

                // const_def[ p080($0,$2) ]
                     = ADD_ID_D(
                          ident[ $0 ],
                          MAKE_ATTR_CONST( number[ $2 ] ) )  ;

p090: var_dcl_part    ::= VAR var_name_list ';'

                // var_dcl_part[ p090($1) ] = var_name_list[ $1 ]  ;

p100: var_dcl_part    ::=

                // var_dcl_part[ p100() ] = I_STATE-STATE()  ;

p120: var_name_list   ::= var_name

                // var_name_list[ p120($0) ] = var_name[ $0 ]  ;

p130: var_name_list   ::= var_name_list ',' var_name
```

```
                // var_name_list[ p130($0,$2) ]
                    = COMPOSITION( var_name[ $2 ],
                                   var_name_list[ $0 ] )   ;

p140: var_name        ::= ident

                // var_name[ p140($0) ]
                    = ADD_ID_D(
                          ident[ $0 ],
                          MAKE_ATTR_VAR( ZERO() ) )   ;

p145: var_name        ::= ident ':' ARRAY

                // var_name[ p145($0) ]
                    = ADD_ID_D(
                          ident[ $0 ],
                          MAKE_ATTR_ARRAY(INIT_ARRAY()))   ;

p150: proc_dcl_part   ::= proc_dcl_list ';'

                // proc_dcl_part[ p150($0) ] = proc_dcl_list[ $0 ]   ;

p160: proc_dcl_part   ::=

                // proc_dcl_part[ p160() ] = I_STATE-STATE()   ;

p170: proc_dcl_list   ::= proc_dcl

                // proc_dcl_list[ p170($0) ] = proc_dcl[ $0 ]   ;

p180: proc_dcl_list   ::= proc_dcl_list ';' proc_dcl

                // proc_dcl_list[ p180($0,$2) ]
                    = COMPOSITION( proc_dcl[ $2 ],
                                   proc_dcl_list[ $0 ] )   ;

p190: proc_dcl        ::= PROCEDURE ident ';' block

                // proc_dcl[ p190($1,$3) ]
                    = ADD_ID_D(
                          ident[ $1 ],
                          MAKE_ATTR_PROC-LIST( block[ $3 ],
                                               INIT_LIST() ))   ;

p195: proc_dcl        ::= PROCEDURE ident
                          '(' proc_para_section_list ')' ';' block

                // proc_dcl[ p195($1,$3,$6) ]
                    = ADD_ID_D(
                          ident[ $1 ],
                          MAKE_ATTR_PROC-LIST(
                                  block[ $6 ],
                                  proc_para_section_list[ $3 ] ))   ;

p201: proc_para_section_list ::= proc_para_section
```

```
                // proc_para_section_list[ p201($0) ]
                    = proc_para_section[ $0 ] ;

p202: proc_para_section_list ::= proc_para_section ';'
                                 proc_para_section_list

                // proc_para_section_list[ p202($0,$2) ]
                    = APPEND_LIST( proc_para_section[ $0 ],
                                   proc_para_section_list[ $2 ] ) ;

p203: proc_para_section_list ::=

                // proc_para_section_list[ p203() ]
                    = INIT_LIST() ;

p204: proc_para_section      ::= proc_value_para_list

                // proc_para_section[ p204($0) ]
                    = proc_value_para_list[ $0 ] ;

p205: proc_para_section      ::= VAR proc_var_para_list

                // proc_para_section[ p205($1) ]
                    = proc_var_para_list[ $1 ] ;

p206: proc_value_para_list   ::= proc_value_para

                // proc_value_para_list[ p206($0) ]
                    = LIST_C( proc_value_para[ $0 ], INIT_LIST() ) ;

p207: proc_value_para_list   ::= proc_value_para ','
                                 proc_value_para_list

                // proc_value_para_list[ p207($0,$2) ]
                    = LIST_C( proc_value_para[ $0 ],
                              proc_value_para_list[ $2 ] ) ;

p208: proc_var_para_list     ::= proc_var_para

                // proc_var_para_list[ p208($0) ]
                    = LIST_C( proc_var_para[ $0 ], INIT_LIST() ) ;

p209: proc_var_para_list     ::= proc_var_para ','
                                 proc_var_para_list

                // proc_var_para_list[ p209($0,$2) ]
                    = LIST_C( proc_var_para[ $0 ],
                              proc_var_para_list[ $2 ] ) ;

p210: proc_value_para        ::= ident

                // proc_value_para[ p210($0) ]
                    = MAKE_ATTR_VAL_PARA( ident[ $0 ] ) ;

p211: proc_var_para          ::= ident
```

```
                // proc_var_para[ p211($0) ]
                    = MAKE_ATTR_VAR_PARA( ident[ $0 ] ) ;

p220: statement       ::= ident ASSIGN expression

                // statement[ p220($0,$2) ]
                    = UPDATE_D(
                          ident[ $0 ],
                          MAKE_ATTR_VAR_D( expression[ $2 ] ) )  ;

p225: statement       ::= ident '[' expression ']' ASSIGN expression

                // statement[ p225($0,$2,$5) ]
                     = UPDATE_D( ident[ $0 ],
                                 MAKE_ATTR_ARRAY_D(
                                   ARRAY_C_D(
                                     MAKE_ARRAY_D(
                                          RETRIEVE_D( ident[ $0 ] )),
                                     expression[ $2 ],
                                     expression[ $5 ] )))  ;

p230: statement       ::= CALL ident

                // statement[ p230($1) ]
                    = LEAVE_BLOCK_D(
                          APPLY_STATE_D(
                             MAKE_PROC_D(
                               RETRIEVE_D( ident[ $1 ] ) ),
                             COMPOSITION(
                               ADD_ID_LIST_D(
                                  MAKE_LIST_D(RETRIEVE_D( ident[ $1 ] )),
                                              INIT_LIST() ),
                               ENTER_BLOCK_D(
                                 SUM_INT_D(
                                   FIND_BLOCK_D( ident[ $1 ] ),
                                     MAKE_STATE-INT( ONE() ) ) ) ) ) )  ;

p235: statement       ::= CALL ident '(' call_para_list ')'

                // statement[ p235($1,$3) ]
                    = LEAVE_BLOCK_D(
                        APPLY_STATE_D(
                          MAKE_PROC_D(
                            RETRIEVE_D( ident[ $1 ] )),
                          COMPOSITION(
                            ADD_ID_LIST_D( MAKE_LIST_D(
                                             RETRIEVE_D( ident[ $1 ] )),
                                           call_para_list[ $3 ] ),
                            ENTER_BLOCK_D(
                              SUM_INT_D(
                                FIND_BLOCK_D( ident[ $1 ] ),
                                  MAKE_STATE-INT( ONE() ) ) ) ) ) )  ;

p240: statement       ::= BEGINN statement_list END

                // statement[ p240($1) ] = statement_list[ $1 ]  ;
```

```
p250: statement        ::= IF condition THEN statement

              // statement[ p250($1,$3) ]
                  = IF_STATE_D(
                       condition[ $1 ],
                       statement[ $3 ],
                       I_STATE-STATE() )  ;

p255: statement        ::= IF condition THEN statement ELSE statement

              // statement[ p255($1,$3,$5) ]
                  = IF_STATE_D(
                       condition[ $1 ],
                       statement[ $3 ],
                       statement[ $5 ] )  ;

p260: statement        ::= WHILE condition DO statement

              // statement[ p260($1,$3) ]
                  = ITERATE( condition[ $1 ], statement[ $3 ] )  ;

p261: statement        ::= REPEAT statement_list UNTIL condition

              // statement[ p261($1,$3) ]
                  = REPEAT( condition[ $3 ], statement_list[ $1 ] )  ;

p265: statement        ::= READ '(' read_para_list ')'

              // statement[ p265($2) ]
                  = read_para_list[ $2 ]  ;

p266: statement        ::= WRITE '(' write_para_list ')'

              // statement[ p266($2) ]
                  = write_para_list[ $2 ]  ;

p270: statement        ::=

              // statement[ p270() ] = I_STATE-STATE()  ;

p281: read_para_list   ::= read_para

              // read_para_list[ p281($0) ]
                  = read_para[ $0 ]  ;

p282: read_para_list   ::= read_para ',' read_para_list

              // read_para_list[ p282($0,$2) ]
                  = COMPOSITION( read_para_list[ $2 ],
                                 read_para[ $0 ] )  ;

p283: read_para        ::= ident

              // read_para[ p283($0) ]
                  = COMPOSITION(
```

```
                        UPDATE_D( INPUT(),
                                  MAKE_ATTR_FILE_D(
                                    REMOVE_DATA_D(MAKE_FILE_D(
                                      RETRIEVE_D( INPUT() )))),
                        UPDATE_D( ident[ $0 ],
                                  MAKE_ATTR_VAR_D(
                                    READ_FILE_D(MAKE_FILE_D(
                                      RETRIEVE_D( INPUT() )))))  ;

p284: read_para       ::= ident '[' expression ']'

              // read_para[ p284($0,$2) ]
                   = COMPOSITION(
                        UPDATE_D( INPUT(),
                                  MAKE_ATTR_FILE_D(
                                    REMOVE_DATA_D(MAKE_FILE_D(
                                      RETRIEVE_D( INPUT() )))),
                        UPDATE_D( ident[ $0 ],
                                  MAKE_ATTR_ARRAY_D(ARRAY_C_D(
                                    MAKE_ARRAY_D(
                                        RETRIEVE_D( ident[ $0 ] )),
                                    expression[ $2 ],
                                    READ_FILE_D(MAKE_FILE_D(
                                      RETRIEVE_D( INPUT() )))))))  ;

p285: write_para_list ::= write_para

              // write_para_list[ p285($0) ]
                  = write_para[ $0 ]  ;

p286: write_para_list ::= write_para ',' write_para_list

              // write_para_list[ p286($0,$2) ]
                  = COMPOSITION( write_para_list[ $2 ],
                                 write_para[ $0 ] )  ;

p287: write_para      ::= expression

              // write_para[ p287($0) ]
                  = UPDATE_D( OUTPUT(),
                              MAKE_ATTR_FILE_D(
                                WRITE_FILE_D(
                                  MAKE_FILE_D(
                                    RETRIEVE_D( OUTPUT() )),
                                  expression[ $0 ] )))  ;

p290: statement_list  ::= statement

              // statement_list[ p290($0) ] = statement[ $0 ]  ;

p300: statement_list  ::= statement_list ';' statement

              // statement_list[ p300($0,$2) ]
                  = COMPOSITION( statement[ $2 ],
                                 statement_list[ $0 ] )  ;
```

```
p303: call_para_list  ::= call_para

              // call_para_list[ p303($0) ]
                  = LIST_C( call_para[ $0 ], INIT_LIST() )  ;

p304: call_para_list  ::= call_para ',' call_para_list

              // call_para_list[ p304($0,$2) ]
                  = LIST_C( call_para[ $0 ], call_para_list[ $2 ] )  ;

p305: call_para_list  ::=

              // call_para_list[ p305() ]
                  = INIT_LIST()  ;

p307: call_para       ::= expr_sym

              // call_para[ p307($0) ]
                  = MAKE_ATTR_EXPR( expr_sym[ $0 ] )  ;

p310: condition       ::= ODD expression

              // condition[ p310($1) ]
                  = ODDP_INT_D( expression[ $1 ] )  ;

p315: condition       ::= EOFF

              // condition[ p315() ]
                  = EOF_D(MAKE_FILE_D(
                            RETRIEVE_D( INPUT() )))  ;

p320: condition       ::= expression '=' expression

              // condition[ p320($0,$2) ]
                  = EQ_INT_D( expression[ $0 ], expression[ $2 ] )  ;

p330: condition       ::= expression NE expression

              // condition[ p330($0,$2) ]
                  = NOT_BOOL_D(
                       EQ_INT_D( expression[ $0 ],
                                 expression[ $2 ] ) )  ;

p340: condition       ::= expression '<' expression

              // condition[ p340($0,$2) ]
                  = LT_INT_D( expression[ $0 ], expression[ $2 ] )  ;

p350: condition       ::= expression '>' expression

              // condition[ p350($0,$2) ]
                  = LT_INT_D( expression[ $2 ], expression[ $0 ] )  ;

p360: condition       ::= expression LE expression

              // condition[ p360($0,$2) ]
```

```
                    = OR_BOOL_D(
                         EQ_INT_D( expression[ $0 ],
                                   expression[ $2 ] ),
                         LT_INT_D( expression[ $0 ],
                                   expression[ $2 ] ) )   ;

p370: condition       ::= expression GE expression

              // condition[ p370($0,$2) ]
                    = OR_BOOL_D(
                         EQ_INT_D( expression[ $0 ],
                                   expression[ $2 ] ),
                         LT_INT_D( expression[ $2 ],
                                   expression[ $0 ] ) )   ;

p390: expression      ::= '(' expression ')'

              // expression[p390($1)] = expression[$1]   ;

p400: expression      ::= ident

              // expression[ p400($0) ]
                    = MAKE_INT_D(
                          RETRIEVE_D( ident[ $0 ] ) )   ;

p405: expression      ::= ident '[' expression ']'

              // expression[ p405($0,$2) ]
                     = ACCESS_ARRAY_D(
                          MAKE_ARRAY_D(RETRIEVE_D( ident[ $0 ] )),
                          expression[ $2 ] )   ;

p410: expression      ::= nat_number

              // expression[ p410($0) ]
                    = MAKE_STATE-INT( POS(nat_number[ $0 ]) )   ;

p420: expression      ::= expression '+' expression

              // expression[ p420($0,$2) ]
                    = SUM_INT_D( expression[ $0 ], expression[ $2 ] )   ;

p430: expression      ::= expression '-' expression

              // expression[ p430($0,$2) ]
                    = DIFF_INT_D( expression[ $0 ], expression[ $2 ] )   ;

p440: expression      ::= expression '*' expression

              // expression[ p440($0,$2) ]
                    = MULT_INT_D( expression[ $0 ], expression[ $2 ] )   ;

p450: expression      ::= expression '/' expression

              // expression[ p450($0,$2) ]
                    = DIV_INT_D( expression[ $0 ], expression[ $2 ] )   ;
```

```
p460: expression      ::= '-' expression %prec(ANARY)

            // expression[ p460($1) ]
                = MINUS_INT_D( expression[ $1 ] )  ;

p465: expression      ::= '+' expression %prec(ANARY)

            // expression[ p465($1) ] = expression[ $1 ] ;

p470: expr_sym        ::= ident '[' expr_sym ']'

            // expr_sym[ p470($0,$2) ]
                = MAKE_EXPR_EMT( ident[ $0 ], expr_sym[ $2 ] )  ;

p471: expr_sym        ::= '(' expr_sym ')'

            // expr_sym[ p471($1) ] = expr_sym[ $1 ] ;

p472: expr_sym        ::= ident

            // expr_sym[ p472($0) ] = MAKE_EXPR_ID( ident[ $0 ] ) ;

p473: expr_sym        ::= nat_number

            // expr_sym[ p473($0) ]
                = MAKE_EXPR_INT( POS( nat_number[ $0 ] )) ;

p474: expr_sym        ::= expr_sym '+' expr_sym

            // expr_sym[ p474($0,$2) ]
                = SUM_INT_S( expr_sym[ $0 ], expr_sym[ $2 ] ) ;

p475: expr_sym        ::= expr_sym '-' expr_sym

            // expr_sym[ p475($0,$2) ]
                = DIFF_INT_S( expr_sym[ $0 ], expr_sym[ $2 ] ) ;

p476: expr_sym        ::= expr_sym '*' expr_sym

            // expr_sym[ p476($0,$2) ]
                = MULT_INT_S( expr_sym[ $0 ], expr_sym[ $2 ] ) ;

p477: expr_sym        ::= expr_sym '/' expr_sym

            // expr_sym[ p477($0,$2) ]
                = DIV_INT_S( expr_sym[ $0 ], expr_sym[ $2 ] ) ;

p478: expr_sym        ::= '-' expr_sym  %prec(ANARY)

            // expr_sym[ p478($1) ] = MINUS_INT_S( expr_sym[ $1 ] ) ;

p479: expr_sym        ::= '+' expr_sym  %prec(ANARY)

            // expr_sym[ p479($1) ] = expr_sym[ $1 ] ;
```

```
p480: number          ::= nat_number
                // number[ p480($0) ] = POS( nat_number[$0] )  ;
p490: number          ::= '+' nat_number
                // number[ p490($1) ] = POS( nat_number[$1] )  ;
p500: number          ::= '-' nat_number
                // number[ p500($1) ] = NEG( nat_number[$1] )  ;
p510: ident           ::= ID
                // ident[ p510($0) ] = ID[$0]  ;
p520: nat_number      ::= ICONST
                // nat_number[ p520($0) ] = ICONST[$0]  ;
}}
```