ICOT Technical Memorandum: TM-0690

TM-0690

A'UM(V.1)解説書(第一版)

吉田かおる、近山　隆

March, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

$\mathcal{A}'\mathcal{UM}$ (V.1) 解説書

第 一版

吉田 かおる　　近山 隆

ICOT

1989 年 3 月

目次

図目次

表目次

第 1 章

はじめに

$A'UM$ は大規模な並列問題の解法のために設計された、ストリーム計算を基調とする並列オブジェクト指向プログラミング言語である.

本書は、$A'UM$ の計算モデル、言語文法および支援、そして処理系の実装について述べている. 章の構成は以下のようになっている.

第 2 章に計算モデルを定義する. "並列システムとは事象の半順序集合である" という立場から、ストリームを用いて実現される半順序集合の演算系として計算モデルは定義される.

第 3 章に言語文法および支援を述べる. "並列プログラミングとは事象グラフの作成である." 図 (グラフ) を描くがごとくプログラムが書けるように、"メッセージが右から左に流れる" という原則に基づく関数的文法が定義される. また、計算の終了は半順序集合の完備化として捉えることができる. プログラマが予期せぬデッドロックは最小限に回避し、計算の終了を促すべく、不完備なストリームの自動閉鎖および自動回収を支援している.

第 4 章で処理系について述べる. 計算モデルは、$A'UM$ 抽象機械を定義したものである. $A'UM$ プログラムは計算モデルで定義した基本演算の列に展開される. 現在、この基本演算列から KL1 へ上に処理系が開発されている. 生成する KL1 コード、および実行環境について概略する.

最後に、$A'UM$ に関する参考文献を挙げる.

今後、$A'UM$ は実用化のためにより高い表現力と効率を求めて、計算モデルおよび言語文法に種々の変更、拡張そして追加が施されることになろう.
本書で定義する版を $A'UM$ (V.1) と呼ぶことにする.

第 2 章

計算モデル

2.1 並列システムと半順序集合

そもそも並列システムとは何か? 並列システムとは, 一つ以上の事象が独立に発生しうる系である. ここで"独立"とはその間に順序関係を規定できないことを意味する.

例えば, 三つの事象 a, b, c が発生する系があり,

- a は b より後に起こらなければならない. $(a \prec b)$
- c は a や b とは独立である.

なる事象間の順序関係 $(\prec)$ があるとしよう.

一般に, 任意の要素間に順序関係が規定できる集合は全順序集合あるいは鎖 (chain) と呼ばれ, また順序関係を規定できないような要素を含む集合は半順序集合 (poset) と呼ばれる[Birkhoff79].

a, b, c それぞれ一つの事象からなる三つの集合 $A = \{a\}, B = \{b\}, C = \{c\}$ を考えると, 上記の系 S は半順序集合演算として次のように表現できる.

$$S = (A \dotplus B) + C$$

ここで, $\dotplus$ は順序和 (ordinal sum), $+$ は基本和 (cardinal sum) と呼ばれ, それぞれ半順序集合に対する異なる種類の加算演算子を示す (詳細は後で述べる). それぞれの演算結果は半順序集合である.

すなわち, 並列システムは事象の半順序集合であり, 並列システムは半順序集合の演算系として記述できる. 半順序集合はストリームを用いて自然に表現できる.

2.2 ストリーム

ストリーム (stream) はメッセージの鎖である.

メッセージの順序関係を示すストリームの方向は, 二つの端末である **入力端** (inlet) と **出力端** (outlet) によって表現する. ここで, 入力端は "^" 付き変数 (例えば, ^X) で, 出力端は ただの変数 (例えば, X) で指定する.

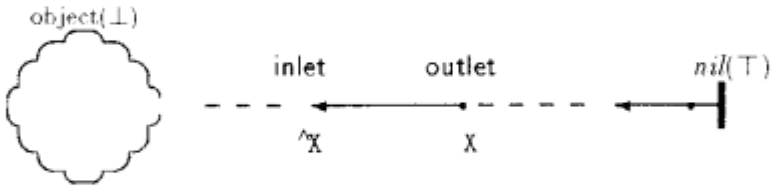

図 2.1: ストリーム表現

ストリーム両端の呼び方: ストリーム両端は, ストリームからメッセージの受信しストリームへメッセージを送信する **オブジェクトから見て** 入ってくる端であるか出ていく端であるかという視点から, 頭部を入力端, 末尾を出力端と呼んでいる.[1]

[1] [Yoshida87, Yoshida88a] では, **ストリーム自身から見て** どちらからメッセージが入りそして出ていくかという視点から, ストリームの末尾を入力端, 頭部を出力端と呼んでいた. この呼び方はいわゆる一般の入出力の捉え方と逆であり混乱を招いたため, 現在の呼び方に変更された.

2.3 ストリーム演算

ストリーム計算は、以下の基本演算で示されるように、ストリームの生産と消費からなる.

送信 (send): 出力端 X へメッセージ m を送信し、これに続くストリームの入力端を ^Y とする.

閉鎖 (close): 出力端 X を閉鎖する. この時、出力端を閉鎖する際に残る状態を 空 (nil) と呼び、このストリームがもはやアクセスできないことを示す.

接続 (connect): 入力端 ^Y を出力端 X へ接続する. この時、入力端 ^Y に続くメッセージは出力端 X より前に位置するメッセージに連結される.

受信 (receive): 入力端 ^X からメッセージを受信し、それに続くストリームの出力端を Y とする.

閉鎖検出 (is closed): 入力端 ^X が閉鎖されていることを検出する.

このように、ストリーム計算は入力端、出力端および空の三種類の用語を使って説明される.

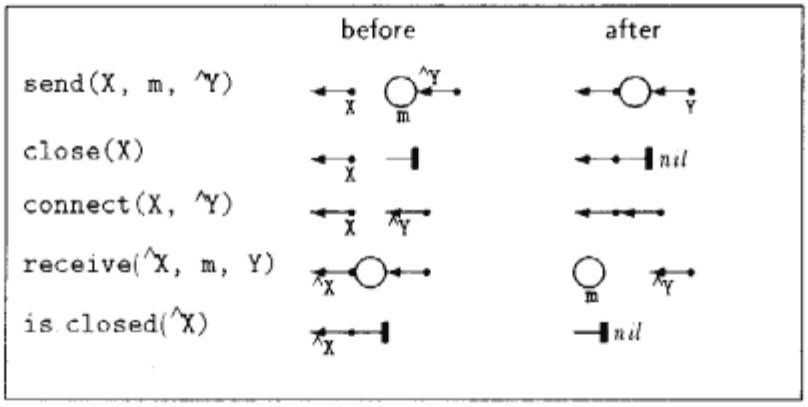

図 2.2: 基本演算

2.4 ジョイント

ストリーム・ツリーを構成するために、ストリームの二項演算を定義し、これらを**ジョイント** (joint) と呼ぶ.

併合 (merge): 入力端 ^X と入力端 ^Y それぞれからのメッセージを非決定的順序で併合し、順次、出力端 Z に送信する. ただし、ストリーム ^X およびストリーム ^Y におけるメッセージの順序関係は保たれる.

連結 (append): 入力端 ^Y からのメッセージが入力端 ^X からのメッセージのどれよりも後になるように、順次、出力端 Z に送信する. ただし、ストリーム X および ストリーム Y におけるメッセージの順序関係は保たれる.

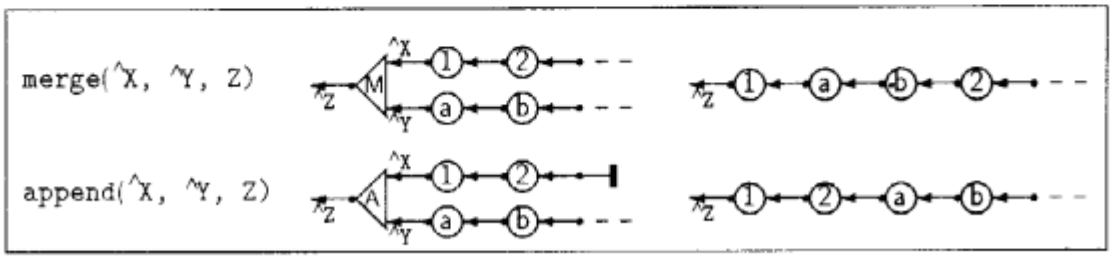

図 2.3: ジョイント

これらの二項演算 (とりわけ、併合演算) は

$$f : Chain \times Chain \to Poset,$$

$$g : Poset \to Chain,$$

の二種類の関数の合成 $g \circ f$ で表される. 鎖を半順序集合に一般化すると、併合演算は基本和 (cardinal sum) に、連結演算は順序和 (ordinal sum) に対応する.

ストリームが鎖であるのに対して、ジョイントにより構成される半順序集合を **チャネル** (channel) と呼ぶ.

また、"ストリーム (の出力端) を分岐する" とは、"Zは X と Y に分岐する" ように二項演算を演算結果から見た時の別の言い方である.

2.5 メッセージ

メッセージ (message) はメッセージ名とストリーム端末 (入力端あるいは出力端) の列をその引数として持つ.

一個も引数を持たないもの, すなわち, メッセージ名だけから成るメッセージを **原子メッセージ** (atomic message), 引数を有するメッセージを **合成メッセージ** (compound message) と呼ぶ.

各メッセージはメッセージ名と端末数および **受信者から見た**それぞれの端末の方向により識別される. すなわち, メッセージ名が一致しても, 端末数あるはそれらの方向が異なれば別のメッセージと解釈する.

各メッセージは, 送信者が実引数として渡すストリーム端と受信者に仮引数として渡されるストリーム端を接続するストリーム接続器として働く. 前述のように, 二本のストリームの接続は一方の入力端と他方の出力端が与えられて可能となるので, メッセージの送信者と受信者はそれぞれの引数に対して, 逆方向を指定する必要がある.

例えば, 下図の m(X, ^Y, Z) はメッセージ名 m を持ち, 二個の出力端 X と Z, 一個の入力端 ^Y の合わせて三個のストリーム端末を保持する. したがって, m(-+-) のように識別名が与えられる. このメッセージの送信者は m(^U, V, ^W) のように指定すれば, ^U は X に, V は ^Y に, そして ^W は Z にそれぞれ接続される.

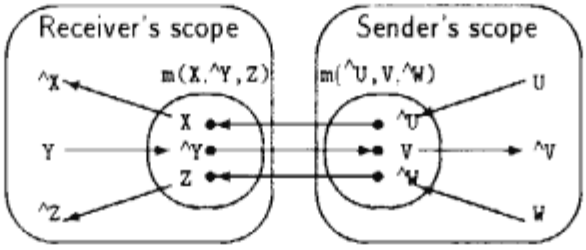

図 2.4: ストリーム接続器としてのメッセージ

2.6 オブジェクト

オブジェクトは繰返し計算を抽象化したものである．オブジェクトは，その生成時に，外部とのインタフェースとして働きインタフェース・ストリームと呼ばれる，一本の入力端を与えられる．またオブジェクトの内部状態を表すスロット群を内部に保持することができる．

送信者の立場から，あるオブジェクトへのストリーム (の出力端) はオブジェクトそのものに見える．あるオブジェクトと知り合いになるということはそのオブジェクトへのストリーム (の出力端) を獲得することである．あるオブジェクト (A) を別のオブジェクト (B) に紹介することは，A へのストリームを分岐し，その片方を B に渡すことである．

2.6.1 生成

クラスにインスタンス生成メッセージ new(^Obj) を送信することにより，そのクラスのインスタンス・オブジェクトが生成される．

この時，生成されたばかりのオブジェクト (これを **第 0 世代** と呼ぶ) には暗黙裏に起動メッセージ initiate が送られ，それに続くストリームの入力端が ^Obj とされる．対応する出力端 Obj をオブジェクト自身と思ってメッセージを送信すればよい．

また，起動メッセージに対するメソッドはオブジェクトの内部状態を後に述べるように初期化するように定義されているが，これはユーザにより再定義可能である．

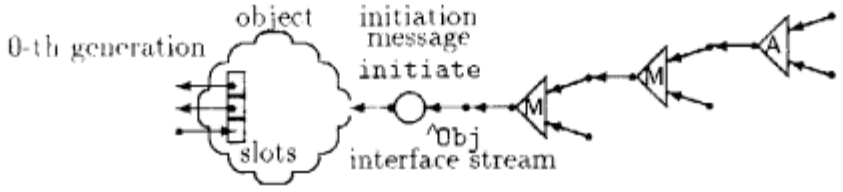

図 2.5: オブジェクトの外観

2.7 世代

オブジェクトは以下に述べる二段階を一周期とする周期の繰返しである．ここで周期のことをオブジェクトの**世代** (generation) と呼ぶ．

1. **受動期**: インタフェース・ストリームに関する事象，すなわちメッセージの受信あるいは閉鎖検出を待つ．

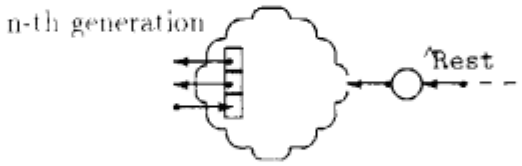

図 2.6: 受動期にある世代

2. **能動期**:

   世代はインタフェース・ストリームに関する事象を観測した後，その観測事象に応じた一連の動作を実行する．一連の動作とは，

   - 任意個の送信，閉鎖，接続，併合，連結および基底オブジェクトの生成動作と
   - 一個以下の世代降下 (次節で述べる)

   から成る．各動作は任意の順序あるい並列に実行される．

   [注] 一般オブジェクトの生成は，クラス・オブジェクトの生成とそれへのメッセージ送信に分解される．

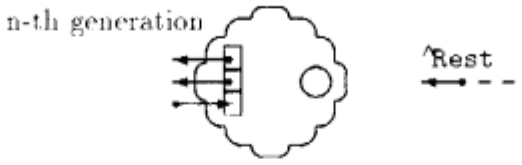

図 2.7: 能動期にある世代

一世代は，インタフェース・ストリームの入力端，オブジェクトの内部状態を表すストリーム端から成るストリーム端の集合体である．

2.7.1 メソッド

一世代の振舞いを定義したものを **メソッド** (method) と呼ぶ．メソッドは観測すべき事象を定義する受動部とその事象に応じて執るべき動作を定義する能動部から成る．

2.7.2 世代降下

ある世代が次の世代を生成する動作を**世代降下** (generation descending)と呼ぶ. 世代降下は他の動作と任意の順序あるいは並列に実行される.

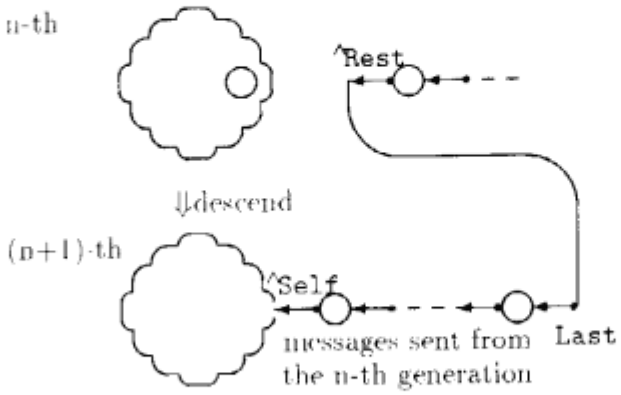

図 2.8: 世代降下

2.7.3 自分自身 (Self)

ある世代のオブジェクトにとって, **自分自身** (Self)とは次世代を意味する. 自分自身にメッセージを送るということは次世代へのストリームの出力端にメッセージを送ることに他ならない.

自分自身の分岐: 自分自身を分岐して, その片方の出力端を受信メッセージあるいは送信メッセージの引数に接続すれば, その送信者あるいは受信者は, 将来この自分自身にメッセージを送信してくることが可能となる.

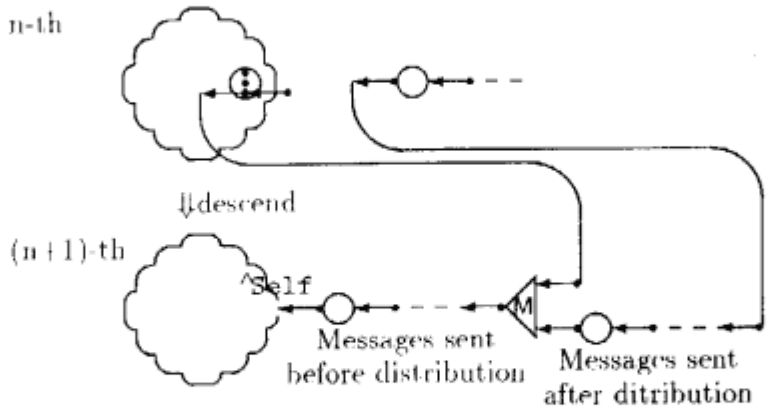

図 2.9: 自分自身の分岐

2.7.4 継続と終了

オブジェクトは，オブジェクトが終了メッセージ $terminate を受信するまで世代降下する．

オブジェクトの終了: オブジェクトが終了メッセージ $terminate を受信すると，それが保持するストリーム端をすべて完備化される．すなわち，出力端は閉鎖し，入力端は回収される．インタフェース・ストリームも同様に回収される．

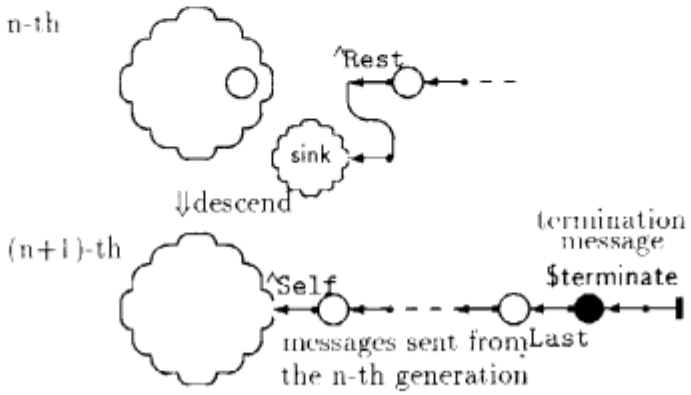

図 2.10: オブジェクト終了の起動

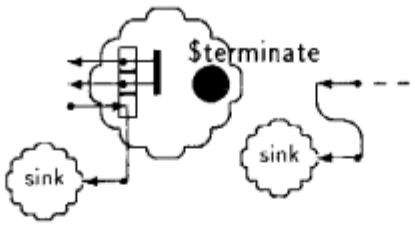

図 2.11: オブジェクトの終了

したがって、オブジェクトの一生は次のような世代連鎖として示される.

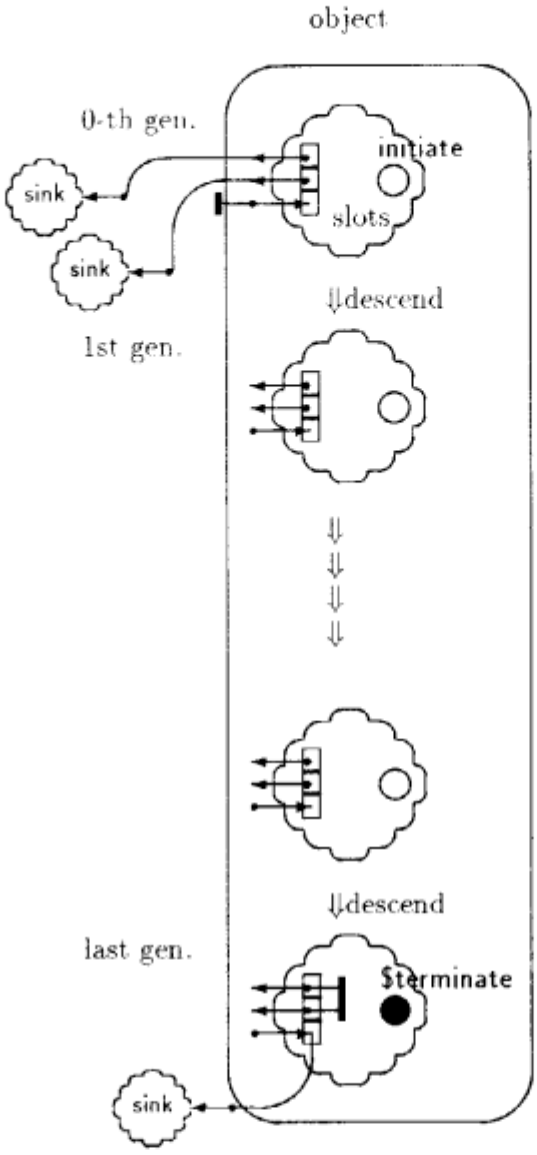

図 2.12: 世代連鎖としてのオブジェクト

2.8 スロット

オブジェクトはその内部状態を表すスロット群を保持できる．各スロットはストリーム端を名前で連想させて保持する．入力端と出力端のどちらを保持するかにより **入力端スロット** あるいは **出力端スロット** と呼ぶ．

スロット名はスロットが定義されるクラス内で一意に決まる．すなわち，各スロットはクラス名とスロット名により識別される．

スロットへのアクセスは，オブジェクト自身すなわち次世代へスロット・アクセスのためのメッセージを送信することにより実現される．

2.8.1 入力端スロット

初期化: オブジェクト生成時，各入力端スロットは閉鎖されている．

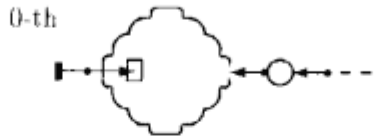

図 2.13: 入力端スロットの初期化

参照: 入力端スロット参照メッセージ get_inlet(Name, Slot) がオブジェクトに送信されると，現在保持している入力端はアクセサが手渡す出力端 (メッセージの第二引数) に接続される．新しく生成されるスロットは閉鎖されている．

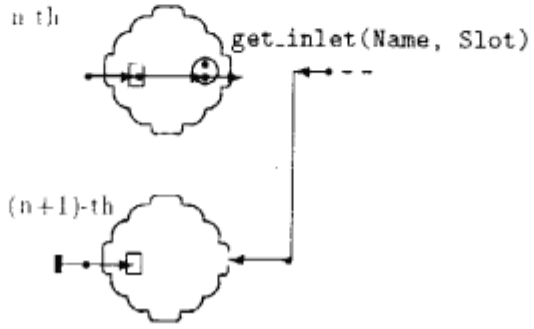

図 2.14: 入力端スロットの参照

更新: 入力端スロット更新メッセージ set_inlet(Name,^Slot) がオブジェクトに送信されると，現在の入力端はシンク・オブジェクトへ接続される．アクセサが手渡す入力端 (メッセージの第二引数) が新しいスロットへ接続される．

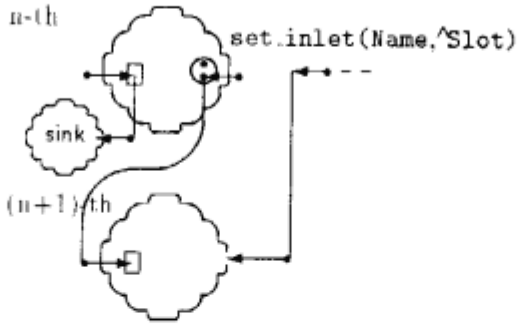

図 2.15: 入力端スロットの更新

後始末: オブジェクト終了時，各入力端スロットはシンク・オブジェクトへ接続される．

2.8.2 出力端スロット

初期化: オブジェクト生成時，各出力端スロットはシンク・オブジェクトへ接続されている．

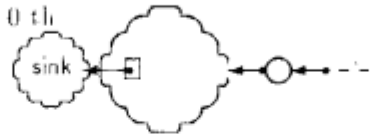

図 2.16: 出力端スロットの初期化

参照: 出力端スロット参照メッセージ get_outlet(Name,^Slot) がオブジェクトに送信されると，現在の出力端は二分され，アクセサが手渡す入力端 (メッセージの第二引数) が片方へ接続される．新しいスロットがもう片方へ接続される．

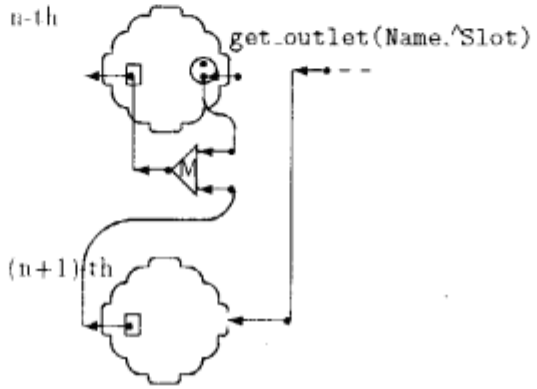

図 2.17: 出力端スロットの参照

更新: 出力端スロット更新メッセージ set_outlet(Name, Slot) がオブジェクトに送信されると，現在の出力端は nil で閉鎖される．アクセサが手渡す出力端 (メッセージの第二引数) は新しいスロットへ接続される．

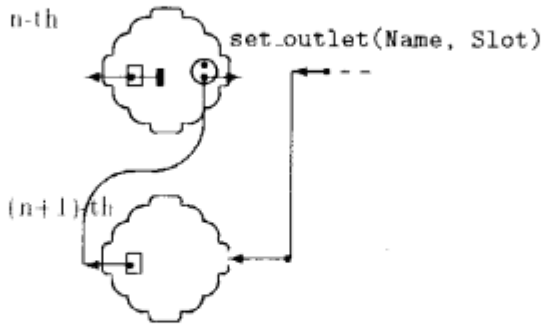

図 2.18: 出力端スロットの更新

後始末: オブジェクト終了時，各出力端スロットは閉鎖される．

2.9 原始オブジェクト

A'UM ではストリームが基底であり，あらゆるものがストリームをインターフェースとしてメッセージ交信しあうオブジェクトである．

一般に原始データとして扱われる．

- 整数 (例えば，5)．
- アトム (例えば，a)．
- ブール値 (例えば，'true)．
- ストリング (例えば，''hi'')

などは **原始オブジェクト** と呼ばれ，他の抽象オブジェクトと全く同様にストリームをインタフェースとするオブジェクトである．クラスもまた原始オブジェクトである．

例えば，整数 5 をプログラムに指定すると，それは 整数オブジェクト 5 が生成され，それへのストリームの出力端を示す．その出力端に加算メッセージadd(1, ^Sum) を送信すると，オブジェクト 5 はその加算メッセージをいつか受信し，加算結果 6 の整数オブジェクトを生成し，それへのストリームの出力端が ^Sum となる．被加算数 1 そして加算結果 6 に対しても同様である．

原始オブジェクトに関する操作に対して，簡易表現が提供されており，ストリームをほとんど意識せずに操作が行なえるようになっている．

リストおよびベクタは原始オブジェクトから合成される抽象オブジェクトであるが，同様に簡易表現が提供されている．

2.10 ストリームの完備化と計算の終了

半順序集合は鎖に分解できる．集合全体として順序関係の下で最小元が存在し，それぞれの鎖に最小上界（鎖中のすべて要素より順序関係で大きい集合の最小元）が与えられるとき，これは完備化された半順序集合（CPO）と呼ばれる．また，集合全体の最小元とその各鎖の最小上界を与えることは半順序集合の完備化と呼ばれる．

最小元は計算の開始，集合自体は計算過程，最小上界は計算の終了とみなすと，最小元が決まらないとは計算が開始しない，また最小上界が決まらないとは計算が終了しないと解釈される．

ストリーム計算においては，それぞれのストリームの入力端と出力端の接続先を決定することを **ストリームの完備化**と呼ぶ．

入力端と出力端のいずれか一方しか現れない不完備なストリームが存在する場合，次のような弊害を引き起こす．

出力端の放置 →

- その入力端からのメッセージ受信あるいは閉鎖検出をあるオブジェクトが期待し続けるかもしれないので，デッドロックを招く可能性がある．

入力端の放置 →

- 形成しているメッセージには誰にも受信されず永久的にゴミ（一括 GC の対象）となる．
- そのストリームを形成しているメッセージ中に引数として含まれるストリームの入力端および出力端も放置されることになる．メッセージの送信者は，受信者がこれら引数として運ばれる入力端を何かのオブジェクトへ接続したり出力端を閉鎖することを期待して送信するから，やはりデッドロックを招く可能性がある．

そこで，不完備なストリームの最上界および最下界は次のように定める．

最上界は空: 放置された出力端は閉鎖する．

最下界はシンク・オブジェクト: 放置された入力端はシンク・オブジェクトに接続する．

シンク・オブジェクト (sink object):　シンク・オブジェクトはメッセージを解釈する能力を有しメッセージを回収する役割を果たすオブジェクトである．すなわち，シンク・オブジェクトの各世代はそのインタフェース・ストリームに関して，

- メッセージを受信した場合，それが引数として保持するストリーム端のそれぞれに対して次のような動作を行なうとともに世代降下する．
  - 入力端であれば，新たにシンク・オブジェクトを生成し，それへ接続する．これを**入力端 (あるいはストリーム) の回収** と呼ぶ．
  - 出力端であれば，閉鎖する．
- 閉鎖を検出した場合，終了する．

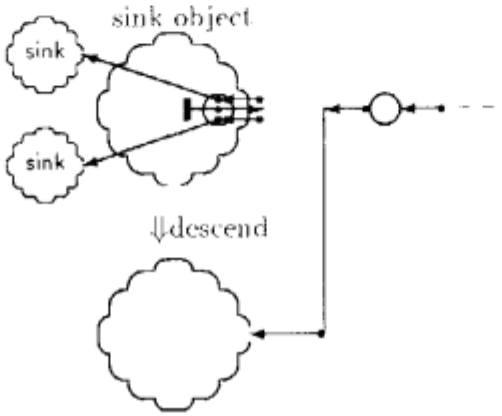

図 2.19: シンク・オブジェクト

2.11 クラス

クラスは、

- 継承すべきクラス群
- 内部状態として保持すべき入力端スロットおよび出力端スロット群
- オブジェクトの世代を定義するメソッド群

から成る、インスタンスの型を定義する。

クラスはストリームをインタフェースとする原始オブジェクトである。そのインタフェース・ストリームを介して、インスタンス生成メッセージを受信すると、その型定義に基づいてインスタンスを生成する。

クラスのクラスであるメタクラスの概念はない。

2.12 継承

プログラムコードを最小化するという目的から、多重クラス継承を支援している。

一つのクラスは、任意個のクラスからメソッド群を継承できる。クラス継承により、オブジェクトに適用可能なメソッド群およびアクセス可能なスロット群の空間は広がるが、継承する上位クラスに対応してインスタンスを生成するわけではない。

2.12.1 継承木

例えば、

1. クラス a がクラス b, c を継承
2. クラス b がクラス p, q を継承
3. クラス c がクラス r, s を継承

と定義されるクラス継承関係は以下のように図示される。

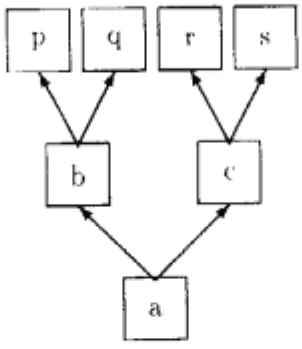

図 2.20: 継承木

このように、クラスで定義されるクラスの継承関係は、一般に、そのクラスを根とする木構造をなす。これを **クラス継承木** と呼ぶ。

クラス継承木は根を起点として**深さ優先かつ左優先**で継承順序が与えられ、継承列が生成される。上の例では、

a → b → p → q → c → r → s →

の継承列となる．このようにユーザにより明示的定義に生成される継承列を **ユーザ定義継承列** と呼ぶ．

このユーザ定義継承列の両端に，

下界クラス → ユーザ定義継承列 → 上界クラス

のように，二つのクラスを暗黙裏に付加したものが各クラスにとっての継承列全体となる．これを **完全継承列** と呼ぶ．

クラス定義において，たとえ明示的には何も継承しなくても，実際には

下界クラス → ユーザ定義クラス → 上界クラス

なる継承列が生成されることになる．

オブジェクトの各世代は，その受動期において観測した事象に対し，この完全継承列の下界クラスから順に探索し適用可能なメソッドを決定した後，能動期に入る．

上界クラスおよび下界クラスは“すべてのクラスのオブジェクトに適用可能なメソッド群”を定義しているクラスであり，それらがユーザ定義クラスによって再定義可能か否かによって，両クラスは異なる．

2.12.2 上界クラス

上界クラスはユーザ定義継承列の最後尾に付加されるクラスである．これはあらゆるクラスのオブジェクトに適用可能であり，かつ，それらユーザ定義クラスにより **再定義可能な** メソッド群を定義している．これらメソッド群として，

- 初期化メッセージ (initiate) に対するメソッド
- 閉鎖検出によるオブジェクトの終了を定義するメソッド (後述)
- 観測事象に対応するメソッドがユーザ定義クラスに未定義である場合のエラー処理を定義するメソッド

が含まれる．

2.12.3 下界クラス

下界クラスはユーザ定義継承列の最前列に付加されるクラスである．これはあらゆるクラスのオブジェクトに適用可能であり，かつ，それらユーザ定義クラスでは **再定義不可能な** メソッド群を定義している．これらメソッド群として，

- スロット・アクセス・メソッド
- 終了メッセージ ($terminate) に対するメソッド

が含まれる．

第 3 章

文法と支援

並列システムは事象の半順序集合であり、並列プログラミングとは事象のグラフを記述することである。したがって、言語にとって、“いかに容易にグラフを描けるようにするか”がその果たすべき第一の課題である。

また、実時間での最大限のゴミの回収とデッドロック回避の問題も合わせ、ストリーム・プログラミングにとってストリームの完備化が必須である。*A'UM* は容易、安全かつ効率のよいプログラムが書けるように、文法を定義するとともに各種支援を行なっている。

以下に、構文および字句を定義する。字句解析はメソッド単位で演算子優先度文法 (operator precedence grammar) に基づいており、演算子の優先度を表に示す。

表 3.1: 構文定義一覧

< *ClassDefinition* > ::=
    class < *ClassName* > “.”
        [< *SuperclassDefition* > “.”]
        [< *InletSlotDefinition* > “.”]
        [< *OutletSlotDefinition* > “.”]
        { < *Method* > “.” }
    end “.”

< *SuperclassDefinition* > ::=**super** < *SuperClassName* > { “,”< *SuperClassName* > }
< *InletSlotDefinition* > ::=**in** < *SlotName* > { “,”< *SlotName* > }
< *OutletSlotDefinition* > ::=**out** < *SlotName* > { “,”< *SlotName* > }

< *ClassName* > ::=< *Name* >
< *SuperClassName* > ::=< *ClassName* >
< *SlotName* > ::=< *SlotName* >

< *Method* > ::=< *Event* > “|” < *Actions* > { “,”< *Actions* > }

< *Event* > ::=< *ReceivingMessage* > |< *DetectingClosed* >
< *MesssageReceiving* > ::=“:” < *MessagePattern* > “=” < *UnorderedOutlet* >
< *DetectingClosed* > ::=“::”

< *Action* > ::=< *NilExpression* >

```
< NilExpression > ::=< StreamClosing > |< VolatileObjectCreation >
< StreamClosing > ::=< OutletExpression > ":"

< OutletExpression > ::=
      < Outlet > |< PrimitiveObjectCreation > |
      < MessageSending > |< StreamMerging > |< OutletMacroExpresion >

< Outlet > ::=< UnorderedOutlet > |< OrderedOutlet >
< UnorderedOutlet > ::=< JointName >
< OrderedOutlet > ::=< JointName > "$" < Number >
< MessageSending > ::=< OutletExpression > ":" < MessaagePattern >
< StreamMerging > ::=< OutletExpression > "=" < InletExpression >

< InletExpression > ::=
      < Inlet > |< StreamAppending > |< InletMacroExpression >

< Inlet > ::="^" < JointName >
< StreamAppending > ::=< InletExpression > "\" < InletExpression >

< VolatileObjectCreation > ::=
      < ImmutableVolatileObjectCreation > |< MutableVolatileObjectCreation >
< ImmutableVolatileObjectCreation > ::=
      < Interface > "?" < ImmutableVolatileClassDefinition >
< MutableVolatileObjectCreation > ::=
      < Interface > "=>" < MutableVolatileClassDefinition >

< Interface > ::=< InletExpression > |< OutletExpression >

< ImmutableVolatileClassDefinition > ::=
      "(" [ < SuperclassDefinition > ";" ]
            < Method > { ";"< Method > } ")"
< MutableVolatileClassDefinition > ::=
      "(" [ < SuperclassDefinition > ";" ]
            [ < InletSlotDefinition > ";" ]
            [ < OutletSlotDefinition > ";" ]
            < Method > { ";"< Method > } ")"
```

表 3.2: 字句定義一覧

```
<Lexicon> ::=
    <Name> |<Number> |<CharacterString> |<Variable> |<Delimiter>

<Name> ::=<NormalName> |<SpecialName> |<QuotedName>
<NormalName> ::=<LowercaseLetter> { <TrailingLetter> }
<SpecialName> ::=<SpecialCharacter> { <SpecialCharacter> }
<QuotedName> ::="'" { <NameStringCharacter> } "'"

<Number> ::=
    <Digit> { <Digit> }

<CharacterString> ::=
    """ { <StringCharacter> } """

<Variable> ::=
    <UppercaseLetter> { <TrailingLetter> }
<TrailingLetter> ::=<Letter> |<Digit> |"_"


<StringCharacter> ::=<NameStringCharacter> |"""" - """"
<NameStringCharacter> ::=<Character> - "'"

<Character> ::=<SpecialCharacter> |<Delimiter> |<Letter> |<Digit>

<SpecialCharacter> ::=
    "!" |"#" |"$" |"&" |"*" |"+" |"-" |"." |"/" |":" |"<" |"="
    ">" |"?" |"@" |"\" |"^" |"_" |"|" |" "

<Delimiter> ::=
    """ |"%" |"'" |"(" |")" |"," |";" |"[" |"]" |"`" |"{" |"}"

<Digit> ::=
    "0" |"1" |"2" |"3" |"4" |"5" |"6" |"7" |"8" |"9"

<Letter> ::=<UppercaseLetter> |<LowercaseLetter>

<UppercaseLetter> ::=
    "A" |"B" |"C" |"D" |"E" |"F" |"G" |"H" |"I" |"J"
    |"K" |"L" |"M" |"N" |"O" |"P" |"Q" |"R" |"S" |"T"
    |"U" |"V" |"W" |"X" |"Y" |"Z"

<LowercaseLetter> ::=
    "a" |"b" |"c" |"d" |"e" |"f" |"g" |"h" |"i" |"j"
    |"k" |"l" |"m" |"n" |"o" |"p" |"q" |"r" |"s" |"t"
    |"u" |"v" |"w" |"x" |"y" |"z"
```

表 3.3: 演算子の優先度

operator	*(precedence, type)*
->	(1050, xfx), (1050, xf)
-|	(1050, xfx), (1050, xf)
|	(1010, xfx)
,	(1000, xfy)
?	(1000, xfx), (500, xf)
=	(700, yfx)
\	(700, yfx)
==	(600, yfx)
\=	(600, yfx)
<	(600, yfx)
>	(600, yfx)
<-	(700, xfx)
+	(500, fx), (500, yfx)
-	(500, fx), (500, yfx)
not	(500, yfx)
/\	(500, yfx)
\/	(500, yfx)
	(400, yfx)
/	(400, yfx)
>>	(400, yfx)
<<	(400, yfx)
mod	(300, yfx)
:	(200, fx)
::	(210, xf)
##	(100,fx)
#	(100, fx)
^	(100, fx)
!	(100, fx)
$	(90, fx), (90, xfx)
'	(10, fx)

3.1 基本文法

3.1.1 クラス定義

クラス定義は、クラス名の定義、継承クラスの定義、入力端スロットおよび出力端スロットの定義、そして一世代の振舞いを記述するメソッドの定義から成る。

<ClassDefinition> ::=
class *<ClassName>* "."
[*<SuperclassDefition>* "."]
[*<InletSlotDefinition>* "."]
[*<OutletSlotDefinition>* "."]
{ *<Method>* "." }
end "."

<SuperclassDefinition> ::=**super** *<SuperClassName>* { ","*<SuperClassName>* }
<InletSlotDefinition> ::=**in** *<SlotName>* { ","*<SlotName>* }
<OutletSlotDefinition> ::=**out** *<SlotName>* { ","*<SlotName>* }

3.1.2 メソッド定義

メソッドは、観測すべき事象を定義する受動部とそれに応じて執るべき動作を定義する能動部の二つの部分から成る。

<Method> ::=*<Event>* "|" *<Actions>* { ","*<Actions>* }

<Actions> ::=*<NilExpression>*

3.1.3 ストリーム変数

ストリームの方向は、次のようなストリーム変数を用いて指定される。

- **入力端変数:** "^" が頭に付いた変数名 (例えば、^X) の出現は入力端を表す。
- **出力端変数:** ただの変数名 (例えば、X) の出現は出力端を表す。

[注] プログラムを読む時、余白に出力端変数から入力端変数へ矢印を引いてみると、メッセージの流れが理解しやすいだろう。

3.1.4 関数的表現式

メソッドの受動部および能動部はそれぞれ関数的表現式によって定義される。各表現式は、その評価値として入力端、出力端あるいは空のいずれかを示し、それに応じて、**入力端表現式**、**出力端表現式** あるいは **空表現式** と呼ぶ。複雑なグラフもこれらの表現式を組み立てることにより簡単に描くことができる。

例えば、C:up:up:up:show(^U) はメッセージ送信式を組み立てたものである．最初の C:up は出力端 C へメッセージ up を送信した後，それに続くストリームの出力端（これを C1 とする）をその評価値とする．したがって，上表現式は C1:up:up:show(^U) と書き換えられる．これを繰り返すと，表現式全体はメッセージ show(^U) を送信した後のそれに続くストリームの出力端を表すことになる．

表 3.4: 基本表現式

relation	*expression*	*result*
receive(^X,m,Y)	':' < *Message* > '=' < *Out* > :m = Y	< *Nil* >
is_closed(^X)	::	< *Nil* >
send(X,m,^Y)	< *Out* > ':' < *Message* > X : m	< *Out* > Y
close(X)	< *Out* > '::' X ::	< *Nil* >
merge(^X,^Y,Z)	< *Out* > '=' < *In* > Z = ^X	< *Out* > Y
append(^X,^Y,Z)	< *Out* > '\' < *In* > Z \ ^X	< *Out* > ^Y
descend(^X,S)	'<==' < *In* > <== ^X	< *Nil* >

3.1.4.1 “右から左”原則

これまで示された図はいずれも，**メッセージが右から左に流れて** 最後に最も左にあるオブジェクトに流れつくように，すなわち左に位置するメッセージはそれより右にあるメッセージより早くオブジェクトに受信されるように描かれている．オブジェクトの受信順序から見れば，**時間が左から右に進む**ようにと言ってもよい．

表現式はこの原則を守るように設計されている．これにより，図を描くようにプログラムを書くことができる．

3.1.4.2 空の導出

基本文法は計算の終了を促すように次のような規則の下で定義されている．

規則 1 (ストリーム変数の対の出現): それぞれのストリーム変数は，入力端変数と出力端変数が対で出現しなければならない．

規則 2 (空表現式のみの出現): メソッドの能動部の トップ・レベルである動作列 (< *Actions* >、フィールド) には，“基本的に” **空表現式しか指定できない**．

これらの規則の目的は，入力端あるいは出力端が放置された不完備なストリームによる弊害を防ぐことにある．ただし，これらは基本文法におけるものであり，後に述べるように，自動閉鎖および自動回収の支援を含む文法の拡張により排除される．

3.1.5 原始オブジェクト

原始オブジェクトに対応する字句の出現は，原始オブジェクトが生成しそのオブジェクトへのストリームの出力端を意味することになる．

以下に，原始オブジェクト(および合成オブジェクト)の種類ととそれらが受信するメッセージ群を 表 3.5 に示す．

3.1.6 共通メッセージ

原始オブジェクトを含めすべてのオブジェクトが受信すべきメッセージがいくつかある．ここに，中でも特徴的なものを示す．一般オブジェクトの場合，これらのメソッドは下界クラスに定義されている．

- **クラス問合せメッセージ:** class(^Class) はオブジェクトにそのクラスを尋ねるメッセージである．例えば，先のカウンタ・オブジェクトへ Counter:class(^Class) のようにこのメッセージを送信すると，^Class はクラス counter への入力端を意味する．

- **誰何メッセージ:** who_are_you(Who) はオブジェクトにその"表示イメージ"を尋ねるメッセージである．例えば，整数 1 へ 1:who_are_you(Who) のようにこのメッセージを送信すると，整数 1 はこの出力端に対して 自分自身が 1 であることを伝えるメッセージ 1 を送信した後，閉鎖する．

  "表示イメージ"とは，オブジェクトをそれとして認識できるものなら基本的に何でもよい．

  一般オブジェクトが誰何メッセージを受信した場合，自分自身へのストリームを引数とするメッセージを返す．すなわち，"君は誰?"の問いかけに"私は私"と答えることになる．

```
:who_are_you(^Who) = Rest | <== ^Self,
                            Who:i_am(AnotherSelf)::,
                            Self = ^AnotherSelf = ^Rest :: .
```

表 3.5: 原始オブジェクト

class	*image*	*acceptable messages*
atom	a	symbol(String)
boolean	'true	not(Negated) and(^Y, LogicalProduct) or(^Y, LogicalSum) xor(^Y, LogicaXsum)
integer	1	plus(Itself) minus(Complement) add(^Y, Sum) sub(^Y, Dif) mul(^Y, Product) div(^Y, Quotient) mod(^Y, Residue) shtl(^Y, LeftShifted) shtr(^Y, RightShifted) lt(^Y, TorF) nlt(^Y, TorF) gt(^Y, TorF) ngt(^Y, TorF) and(^Y, BitwiseProduct) or(^Y, BitwiseSum) xor(^Y, BitwiseXsum)
string	''hi''	size(Size) element_size(ElementSize) make_atom(Atom) element(^Position, Element) set_element(^Position, Element) make_symbol(Atom)
class	##stack	new(Obj)
list	[a|b]	car(Car) cdr(Cdr) set_car(^Car) set_cdr(^Cdr)
vector	{a,b}	size(Size) element(^Position, Element) set_element(^Position, ^Element)
		eq(^Y, TorF) ne(^Y, TorF)

ここに最も簡単な例として、カウンタのプログラムを基本文法で記述してみる.

[応用例-1 (カウンタ)]

カウンタは、メッセージ up あるは down を受信して, それに応じてそのカウンタ値に 1 足すか引くかするオブジェクトである. また, カウンタ値の設定および参照は、メッセージ set および show の受信により行なわれる.

このカウンタに対して二種類のテストをする. クラス test のメソッド testM と testA がそれである. これらのメソッドは共通して,

1. カウンタ・オブジェクト (クラス counter のインスタンス) を生成し,
2. まずカウンタ値を 5 に設定するため、このオブジェクトにメッセージ set(5) を送信し,
3. メッセージ送信後の出力端を分岐し,
4. その片方に二個のup メッセージと一個の show(^U) メッセージを続けて送信し, 二回足した後のカウンタ値を参照してみる
5. もう片方には, 二個のdown メッセージと一個の show(^D) メッセージを続けて送信し, 二回引いた後のカウンタ値を参照してみる

両メソッドの異なる点は,

- メソッド testM では、二本の分岐ストリーム (C1 と C2) は併合され, メッセージ up と down はどちらが先に幾つ到着するかは非決定的である.

  したがって, 参照結果 ^U は {5, 6, 7} のいずれかの値を, ^D は {3, 4, 5} のいずれかの値を示す

- メソッド testA では、二本の分岐ストリーム (C1 と C2) は連結され, C1 からの二個の up メッセージは C2 からの二個の down メッセージより必ず先に到着する.

  したがって, 参照結果 ^U は 7 を, ^D は 5 を示す.

カウンタ・オブジェクトの第一世代がメッセージ set(5) を受信したばかりの能動期の様子を図 3.2 に示す.

```
class   counter.
     out n.
  :up       = Rest | <== ^Self,
                     Self:get_outlet(n, ^N):set_outlet(n, N1) = ^Rest ::,
                     N:add(1, ^N1):: .
  :down     = Rest | <== ^Self,
                     Self:get_outlet(n, ^N):set_outlet(n, N1) = ^Rest ::,
                     N:sub(1, ^N1):: .
  :set(^N) = Rest | <== ^Self,
                     Self:set_outlet(n, N) = ^Rest :: .
  :show(N) = Rest | <== ^Self,
                     Self:get_outlet(n, ^N) = ^Rest :: .
  ::              | <== ^Self,
                     Self:'$terminte':: .
end.

class   test.
  :test         = Rest | <== ^Self,
                          Self:testM(^Um, ^Dm):testA(^Ua, ^Da):nop(^Result)
                              = ^Rest ::,
                          Result\ ^WhoIsUm\ ^WhoIsDm\ ^WhoIsUa\ ^WhoIsDa ::,
                          Um:who_are_you(WhoIsUm)::,
                          Dm:who_are_you(WhoIsDm)::,
                          Ua:who_are_you(WhoIsUa)::,
                          Da:who_are_you(WhoIsDa):: .
  :testM(U, D) = Rest | <== ^Rest,
                          ##counter:new(^Counter)::,
                          Counter:set(5) = ^C ::,
                          C1:up:up:show(^U)::,
                          C2:up:up:show(^D)::,
                          C = ^C1 = ^C2 :: .
  :testA(U, D) = Rest | <== ^Rest,
                          ##counter:new(^Counter)::,
                          Counter:set(5) = ^C ::,
                          C1:up:up:show(^U)::,
                          C2:up:up:show(^D)::,
                          C \ ^C1 \ ^C2 :: .
  :nop(Result) = Rest | <== ^Rest,
                          ##sink:new(^Result)::.
end.
```

図 3.1: 基本文法によるカウンタのプログラム

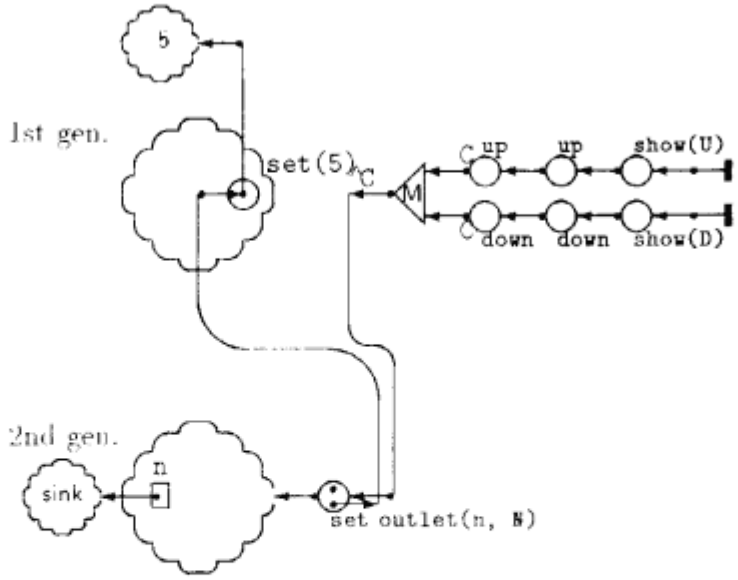

図 3.2: カウンタの第一世代

本プログラムは次節で述べる文法の拡張により以下のように書き直される.

```
class   counter.
     out  n.
  :up   -> !n + 1 = !n.                    % X+Y => S ; X:add(Y,^S)
  :down -> !n - 1 = !n.                    % X-Y => D ; X:sub(Y,^D)
  :set(^N) -> N = !n.                      % !n => $self:set_outlet(n,N)
  :show(N) -> !n = ^N.                     % !n => $self:get_outlet(n,^N)
end.

class  test.
  :test -> :testM(^Um, ^Dm):testA(^Ua, ^Da):nop(^Result),
           Result$1 = (Um?),
           Result$2 = (Dm?),
           Result$3 = (Ua?),
           Result$4 = (Da?).

  :testM(U, D) ->
           #counter:set(5) = ^C, C:up:up:show(^U),
                                 C:down:down:show(^D).
  :testA(U, D) ->
           #counter:set(5) = ^C, C$1:up:up:show(^U),
                                 C$2:down:down:show(^D).
  :nop(Result) -> .
end.
```

図 3.3: 拡張文法によるカウンタのプログラム

3.2 拡張文法

基本文法は，ストリーム計算モデルをそのまま表すように定義された．大規模な並列問題の記述には，より簡潔あるいは抽象化された表現が必要になる．表現力を高め，簡潔かつ安全なプログラムを書くために，基本文法に次の拡張が施されている．

- 簡易(マクロ)表現の導入
- ストリーム変数からチャネル変数への変数の意味の拡張
- ストリーム完備化の支援

3.2.1 簡易表現 (マクロ)

容易にプログラムが書けるように，種々の簡易表現が提供されている．基本文法が関数的表現式から成っているように，これらの簡易表現はいずれも，ある表現式(の列)の展開とともにその評価値として入力端，出力端あるいは空を表す．

3.2.1.1 算術 / 論理演算マクロ

主に、原始オブジェクトの演算操作(例えば整数の四則演算)などのために表 3.6に示すマクロ群が定義されている。それらのほとんどは、評価値として、演算結果オブジェクトへのストリームの出力端を示す。

前述の基本表現式と同様に、これらのマクロを組み立てて複雑な演算式を記述することができる。

例えば、3 + 5 == 8 という演算式に対して、

```
3:add(5, ^Sum)::,
Sum:eq(8, ^True)::,
```

が展開され、真オブジェクト 'true へストリームの出力端 True がその評価値となる。

表 3.6: 算術 / 論理演算マクロ群

macro expression	*mode*	*result*	*expansion*
+ X	<*Out*>	Itself	X:plus(^Itself)::
- X	<*Out*>	Complement	X:minus(^Complement)::
X + Y	<*Out*>	Sum	X:add(Y, ^Sum)::
X - Y	<*Out*>	Difference	X:sub(Y, ^Difference)::
X * Y	<*Out*>	Product	X:mul(Y, ^Product)::
X / Y	<*Out*>	Quotient	X:div(Y, ^Quotient)::
X mod Y	<*Out*>	Residue	X:mod(Y, ^Residue)::
X >> Y	<*Out*>	Shifted	X:shtr(Y, ^Shifted)::
X << Y	<*Out*>	Shifted	X:shtl(Y, ^Shifted)::
not X	<*Out*>	TorF	X:not(^TorF)::
X /\ Y	<*Out*>	Product	X:and(Y, ^Product)::
X \/ Y	<*Out*>	Sum	X:or(Y, ^Sum)::
X xor Y	<*Out*>	Xsum	X:xor(Y, ^Xsum)::
[X|Y]	<*Out*>	List	##list:new(^L)::, L:set_car(X):set_cdr(Y)= ^List::
car X	<*Out*>	Car	X:car(X, ^Car)::
cdr X	<*Out*>	Cdr	X:cdr(X, ^Cdr)::
{X,Y}	<*Out*>	Vector	##vector:new(^V)::, V:set_element(1, X) :set_element(2, Y)= ^Vector::
X == Y	<*Out*>	TorF	X:eq(Y, ^TorF)::
X \= Y	<*Out*>	TorF	X:neq(Y, ^TorF)::
X < Y	<*Out*>	TorF	X:lt(Y, ^TorF)::
X > Y	<*Out*>	TorF	X:gt(Y, ^TorF)::
class_of X	<*Out*>	Class	X:class(^Class)::
X ?	<*In*>	^Who	X:who_are_you(Who)::

3.2.1.2 擬似変数 $self

オブジェクト自身すなわち次世代へのストリーム端は、擬似変数 $self の出現により指定され、出現箇所によりその意味は異なる。

- **出力端では参照**: 擬似変数 $self が出力端フィールドに出現した場合、最新の自分を参照することを意味する。

  メッセージ送信式の出力端を省略した場合、擬似変数 $self が指定されたことを意味する。

- **入力端では更新**: 擬似変数 $self が入力端フィールドに出現した場合、それがこれからの最新の自分となることを意味する。

[自己アクセス順序に関する規則] 自分自身へのアクセス順序は、文脈内での擬似変数 $self、送信先の省略あるいはスロットアクセスなど自己アクセスに関わるマクロの出現順序により決定される。具体的には、

- メソッドは、**事象から動作列へ**
- メソッドの動作列 (*< Actions >* フィールド列) は **左から右へ**
- メッセージの引数列は **左から右へ**
- マクロ展開は、**中から外へ**

という、メソッドを構成する表現式の評価順序に従って、自己アクセス順序が与えられる。

ここに、次のような少し複雑なメソッドを例として挙げよう。

```
:foo(!a, ^X) -> :foo1(!b, !c, ^Y) = $self,
                $self:foo2(Y, ^Z) = $self,
                X + Z = !d.
```

これは、次のメソッドと等価である。ただし、後で述べるが、"!"*< SlotName >* はスロット・アクセス・マクロを、"->" は継続マクロをを表す。

```
:foo(A, ^X) -> $self:get_outlet(a, ^A)
                    :get_outlet(b, ^B)
                    :get_outlet(c, ^C)
                    :foo1(B, C, ^Y)
                    :foo2(Y, ^Z)
                    :set_outlet(d, ^Sum) = $self ::,
               X:add(Z,^Sum):: .
```

3.2.1.3 世代降下マクロ

世代降下を容易に書けるように，次のような継続マクロおよび終了マクロを提供している．

$$<Method> ::= <EventOnly> <DescendingMacro> <Action> \{ \text{","} <Action> \}$$

$$<EventOnly> ::= <ReceivedMessage> | <DetectingClosed>$$
$$<ReceivedMessage> ::= ":" <MessagePattern>$$
$$<DetectingClosed> ::= "::"$$

$$<DescendingMacro> ::= <Succession> | <Termination>$$
$$<Succession> ::= "->"$$
$$<Termination> ::= "-|"$$

継続 (succession) マクロ: 継続マクロは，図 2.8 に示すように，

1. 世代降下する．
2. 現世代でメッセージを受信した場合，受信メッセージの後に続くインタフェース・ストリームの入力端 ^Rest を，次世代へのストリームの末尾にあたる出力端 Last に連結する．

という動作列を添加するマクロである．

例えば，メソッド :m -> :do . は次のように展開される．

```
:m = Rest | <== ^Self,
            Self:do = ^Rest.
```

終了 (termination) マクロ: 終了マクロは，図 2.10 に示すように，

1. 世代降下する．
2. 現世代でメッセージを受信した場合，シンク・オブジェクトを生成し受信メッセージの後に続くインタフェース・ストリームの入力端 ^Rest をこれに接続する．
3. 次世代へのストリームの末尾にあたる出力端 Last に終了メッセージ **$terminate** を送信した後，これを閉鎖する．

という動作列を添加するマクロである．

例えば，メソッド :m -| :do . は次のように展開される．

```
:m = Rest | <== ^Self ,
            ##sink:new(^Rest):: ,
            Self:do:'$terminate':: .
```

3.2.1.4 インスタンス生成マクロ

インスタンス生成マクロは、指定されるクラスへインスタンス生成メッセージを送信し、その時得られるインスタンスへのストリームの出力端を意味する。

表 3.7: インスタンス生成マクロ

macro expression	*mode*	*result*	*attachment*
#ClassName	< *Out* >	Instance	##ClassName:new(^Instance)::

例えば、#counter は

```
##counter:new(^Counter)::
```

と展開され、クラス counter のインスタンスへのストリームの出力端 Counter を表す。

3.2.1.5 スロット・アクセス・マクロ

スロットへのアクセスを容易にするために、入力端スロット・アクセス・マクロ @SlotName および出力端スロット・アクセス・マクロ !SlotName を提供している。

前述の擬似変数 $self と同様に、スロット・アクセス・マクロは出現箇所によりその意味が変わる。以下に出現箇所に対する意味をまとめる。

表 3.8: スロット・アクセス・マクロ群

macro expression	*specified in*	*result*	*expansion*
!SlotName	< *Out* >	Slot	$self:get_outlet(SlotName, ^Slot)
!SlotName	< *In* >	^Slot	$self:set_outlet(SlotName, Slot)
@SlotName	< *Out* >	Slot	$self:set_inlet(SlotName, ^Slot)
@SlotName	< *In* >	^Slot	$self:get_outlet(SlotName, Slot)

- 出力端スロット・アクセス・マクロが出力端フィールドに指定されると、出力端スロットの参照を表す。

  例えば、!n = ^N は

  ```
  $self:get_outlet(n, ^Slot), Slot = ^N
  ```

  に展開される。

  [注] "現スロットはN である" と読めば良い。

- 出力端スロット・アクセス・マクロが入力端フィールドに指定されると、出力端スロットの更新を表す。

  例えば、N = !n は、

  ```
  $self:set_outlet(n, NewSlot), N = ^NewSlot
  ```

に展開される.

[注] "N を新スロットとする" と読めば良い.
スロット更新式を代入に対応づけると, 3 = !n が一般の手続き型言語における代入文 n = 3 と "逆向き" に見えるかもしれない. 将来このスロット n を参照し, そこへ送信されるメッセージは **右から左へ** 3 に向かって流れることからこれを理解されたい.

3.2.1.6 送信先更新マクロ

例えば.

```
:foo -> !a:m1(^X) = !a ::,
        ... (actions related to X) ...
        !a:m2 = !a ::.
```

のように, あるスロットにあるメッセージを送信し, (後で述べる条件分岐動作を含め) その結果に関連する動作を記述した後, そのスロットに別のメッセージを送信するといったスロットおよび自己に関する逐次的な動作列を文脈内で分割して記述したいことがしばしばあるが, その度に自己あるいはスロットの更新の記述をとかく忘れがちになる. そこで.

```
:foo -> !a:m1(^X),
        ... (actions related to X) ...
        !a:m2.
```

のように自己アクセスおよびスロット・アクセスに関して, メッセージ送信後を最新の自己あるいはスロットと暗黙裏に見なせば, 文脈内での出現順序が直接それらの世代として反映されるようになる.

メッセージ送信式における送信先更新マクロは, 送信先として最初に示される出力端の種類に応じて, 送信先の更新を自動的に支援するものである.

表 3.9: 送信先更新マクロ群

macro expressions	*result*	*expansion*
!SlotName:Message	Slot	!SlotName:Message = !SlotName, !SlotName = ^Slot
$global:Message	Global	$global:Message = $global, $global = ^Global
:Message	Self	$self:Message = $self, $self = ^Self

プログラム上で直接メッセージ送信表現式として指定されていなくても, 他のマクロがメッセージ送信表現式に展開される場合もこのマクロが適用される.

例えば, !n + 1 = !n は

```
!n:add(1, ^Sum)::, Sum = !n ::
```

に展開され, ここに送信先更新マクロが適用されて,

```
!n:add(1, ^Sum) = !n ::, Sum = !n ::
```

となる.

3.2.2 チャネル変数

これまでは、変数はストリームすなわち鎖を表すために使われた。ストリーム変数を使って、

> "二人の人物 (X, Y) にある問題 (P) をそれぞれの戦略により解かせ、それぞれが挙げてくる任意個の解答 (A) を集める"

という簡単な問合せ手続きを定義しよう。両者から集める解答の扱い方により、次の二通りがある。

併合: 両者からの解答列をどちらを優先することなく任意順に集める場合、次のように表せる。

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P1, A1)::, Y:solve(P2, A2)::,
    P = ^P1 = ^P2 ::, A = ^A1 = ^A2 ::.
```

解答者 X および Y はそれぞれ将来、

- ストリーム P1 あるいは P2 を介して問題 P へ独立に問い合わせをし、
- 解答をメッセージとして、ストリーム A1 あるいは A2 へ送信し、それらはストリーム A に併合される。

ここで、複数個の出力端変数の出現がストリームの併合演算を表すことにすると、次のようになる。

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P, A), Y:solve(P, A).
```

連結: X からの解答を Y からの解答より優先して集める場合、次のように表せる。

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P1, A1)::, Y:solve(P2, A2)::,
    P = ^P1 = ^P2 ::, A \ ^A1 \ ^A2 ::.
```

解答者に渡されるストリームは、上の場合と変わらない。異なる点は、

- 解答が送信されるストリーム A1 と A2 が連結されて、ストリーム A となる

ことである。

ここで、序数のついた出力端変数がストリームの連結演算を表すことにすると、次のようになる。

```
:consult(^P, ^A, ^X, ^Y, ^Z) ->
   X:solve(P, A$1), Y:solve(P, A$2).
```

この例から明らかなように，これまでのストリーム変数を用いたプログラミングでは，鎖からどのように併合および連結を用いて半順序集合を構成するかがその大きな部分を占めた．このように，変数の意味をストリーム (鎖) からチャネル (半順序集合) に拡張することにより，"解答者に一つの問題 P を渡す" とか "全解答を一つにまとめる" とかいうように，半順序集合全体を一つと見なすことができるようになる．

プログラマは，"ストリームをどう繋ぐか" ではなく，"どのオブジェクトに何をするか" の記述に専念できるようになる．つまり，関心がストリームからオブジェクトに移るわけである．

したがって，チャネルは次のようなチャネル変数の出現により指定される．

- **1 個あるいはそれ以下の入力端変数:** "^" の付いた変数名 (例えば，^X) の出現は入力端を表わす．
- **0 個あるいはそれ以上の非順序出力端変数:** ただの変数名 (例えば，^X) の出現は，併合すべき併合ジョイントの出力端を表す．
- **0 個あるいはそれ以上の順序出力端変数:** "$" と序数が添加された変数名 (例えば，X$1) の出現は，序数の小さい方が先に連結されるような連結ジョイントの出力端を表す．

  序数は正の整数であれば良く，X$1 のように 1 で始める必要も，また X$1, X$2, X$3 のように連続数にする必要もない．

例えば，表現式 X = ^P1 = ^P2 \ ^S1 \ ^S2 \ ^S3 :: は，図 3.4 で示されるチャネルを表す．これは，一個の入力端変数 ^X，三個の非順序出力端変数 X，そして 三個の順序出力端変数 X$1, X$2, X$3 の出現として表すことができる．

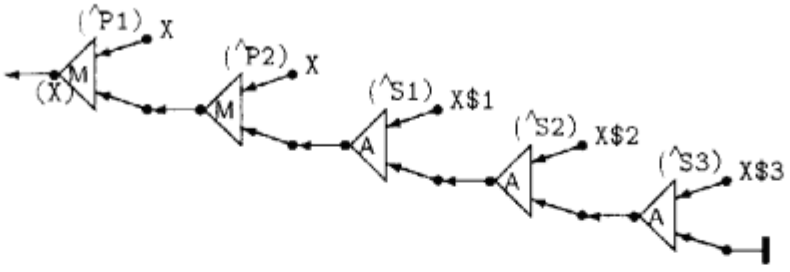

図 3.4: チャネル変数

3.2.3 暗黙のストリーム完備化

基本文法では、不完備なストリームによるデッドロックの発生を防ぎ、計算の終了を促すように、

- ストリーム変数(入力端変数と出力端変数)は対をなして出現すること.
- メソッドの動作列(<*Actions*>フィールド)には空表現式を指定すること.

なる二つの規則が与えられた.

しかし、このように常にストリームが完備するように、特に出力端を常に閉鎖するよう心掛けるのは、プログラマにとってかなりの負担となる. さらに、前節で変数の意味はストリームからチャネルに拡張され、もはや任意個の出力端が出現可能となった.

そこで、次のように文法を拡張して、容易かつ安全なプログラムが書けるようにする.

入力端 / 出力端(表現式)の放置: メソッドの動作列(<*Actions*>フィールド)に入力端表現式および出力端表現式を指定してよい. また、各変数はチャネルを表し、1個以下の入力端と任意個の非順序出力端および順序出力端が出現してよい.

自動完備化: 放置された非空表現式は自動的に完備化される. 自動完備化として、

- 出力端の自動閉鎖
- 入力端の自動回収
- オブジェクトの自動終了

を支援する.

3.2.3.1 出力端の自動閉鎖

放置された出力端は自動的に閉鎖される. 適用を受けるものとして、以下のものがある.

- 動作列に放置された出力端表現式の結果
- 出力端が出現しないチャネルの出力端
- スロット更新時の現スロット
- オブジェクト終了時のすべての出力端スロット
- オブジェクト初期化時のすべての入力端スロット

3.2.3.2 入力端の自動回収

放置された入力端端は自動的に回収される、すなわち、シンク・オブジェクトを生成し、そこへ接続される. 適用を受けるものとして、以下のものがある.

- 動作列に放置された入力端表現式の結果
- 入力端が出現しないチャネルの入力端
- オブジェクト終了時のインタフェース・ストリーム
- オブジェクト終了時のすべての入力端スロット
- オブジェクト初期化時のすべての出力端スロット

3.2.3.3 オブジェクトの自動終了

多くのオブジェクトは，そのインタフェース・ストリームの閉鎖を検出した時点で終了するが，とかくこの指定を忘れがちである．そこで，閉鎖検出により直ちに終了するメソッド

```
:: -> .
```

を任意のクラスが継承する上界クラスで定義しており，これは再定義可能である．

3.2.4 揮発オブジェクト

オブジェクトの各世代は，

1. 観測事象により活性化され，その事象に応じて

2. それと同時に次世代へ降下する．

前者は条件分岐を，後者は繰返しを意味する．すなわち，オブジェクトは元来，コンディション・ハンドラとしての機能を備えている．だからと言って，一つの条件判定に付き一つクラスを定義しているのでは，数多くの小さなクラスを定義しなければならず，プログラムの文脈がばらばらに分断されてしまう．また，条件判定前の文脈と条件判定後の文脈でストリームの変数環境が独立であると，これを渡すために前者から後者へメッセージを送るのはプログラマには大変負担である．これまで述べたオブジェクトの枠組で，これを支援したいというのが揮発オブジェクト導入の発端である．

揮発クラス: 一つのメソッド内に臨時的に定義されるクラスを **揮発クラス** (volatile class) と呼ぶ．一つのメソッド内に任意個の揮発クラスを定義できる．揮発クラスは，以下の点を除けば，一般に定義する外部クラスに比べ何の違いもない．その相違点とは，

- 外部クラスはそれを外部から参照するためのクラス名を持ち，これを用いて揮発クラスを含め他のクラスがこれをアクセスあるいは継承することが可能である．
- 揮発クラスはそれを外部から参照するためのクラス名は持たない．

揮発クラスの定義は何段でもネストすることができる．すなわち，ある揮発クラスのメソッド内に別の揮発クラスを定義できる．一般の手続き型言語で条件判定文 (例えば，`if` 文) がネストするのと同様に考えればよい．

揮発オブジェクト: 揮発クラスのインスタンスを **揮発オブジェクト** (volatile object) と呼ぶ．

生成者オブジェクト: 揮発オブジェクトを生成したオブジェクト，すなわち揮発クラスのを定義しているメソッドを実行しているオブジェクトを **生成者オブジェクト** (creator object) と呼ぶ．揮発クラスの定義がネストする場合，内側の揮発オブジェクトにとってそれを定義している外側の揮発オブジェクトはその生成者オブジェクトである．

以下に述べるが，生成者オブジェクトへのストリームは 擬似変数 `$creator` によって揮発オブジェクトにアクセスされる．

可視領域: スロット名は別々のメソッドに出現して世代を越えて対応するストリームにアクセスできるのに対して，変数名はメソッド内においてのみストリームと一意に対応づけられる．このように名前の有効範囲を **名前の可視領域** (name scope) と呼ぶ．

揮発オブジェクトは，生成者オブジェクトとの間の可視領域の関係によって，不変揮発オブジェクトと可変揮発オブジェクトの 2種類がある．それぞれの詳細を述べる前に，まず両者に共通なクラス定義とオブジェクトの生成の指定方法について述べる．

3.2.4.1 揮発オブジェクトの生成

揮発オブジェクト生成表現式は、生成される揮発オブジェクトのインタフェース・ストリームと揮発クラスの定義からなる. 揮発オブジェクト生成表現式は空表現式である.

```
<VolatileObjectCreation> ::=
        <ImmutableVolatileObjectCreation> |<MutableVolatileObjectCreation>
<ImmutableVolatileObjectCreation> ::=
        <Interface> ''?'' <ImmutableVolatileClassDefinition>
<MutableVolatileObjectCreation> ::=
        <Interface> ''=>'' <MutableVolatileClassDefinition>

<Interface> ::=<InletExpression> |<OutletExpression>

<ImmutableVolatileClassDefinition> ::=
        ''('' [ <SuperclassDefinition> '';'' ]
              <Method> { '';''<Method> } '')''
<MutableVolatileClassDefinition> ::=
        ''('' { <SuperclassDefinition> '';'' }
              [ <InletSlotDefinition> '';'' ]
              [ <OutletSlotDefinition> '';'' ]
              <Method> { '';''<Method> } '')''
```

基本: (入力端をインタフェースとして)
基本的に、<Interface> フィールドには生成される揮発オブジェクトのインタフェース・ストリームの入力端を指定する.

例えば.

```
^Hunger ? (
   : 'true  -> :eat ;
   : 'false -> :sleep
)
```

と指定すると、生成される揮発オブジェクトは入力端 ^Hunger から受信されるメッセージ 'true あるいは 'false により、食べるか眠るかするためのメッセージを自分自身に送信する.

拡張: (出力端には誰何メッセージを送信)
前述の算術マクロを含め多くのマクロは出力端をその評価値としており、その算術結果が何かにより条件分岐したいことがよくある. これを容易に記述できるように、次のような支援をしている.

出力端を指定した場合、暗黙裏にこれに誰何メッセージ who_are_you(Who) を送信し、入力端 ^Who をインタフェースとする.

例えば、奇数か偶数により処理を振り分けたい場合、

```
(X mod 2 == 0) ? ( ... )
```

と指定すると、

```
(X mod 2 == 0):who_are_you(Who)::,
^Who ? ( ... )
```

を意味する.

3.2.4.2 不変揮発オブジェクト

不変揮発オブジェクト (immutable volatile object) は一代限りのオブジェクトであり，生成者オブジェクトと同一の可視領域を有する．

- **変数名の可視領域は同一**: 不変揮発クラスに出現する変数名と，その生成者オブジェクトのメソッド内の出現する同一の変数名は，同一のチャネルを意味する．
- **生成者は自分自身**: 不変揮発オブジェクトにとって，生成者オブジェクトは自分自身である．つまり，不変揮発オブジェクトにとって，自分自身およびスロット群は生成者オブジェクトのそれらを意味する．擬似変数 $creator と $self は同義で使われる

不変揮発オブジェクトの生成: 不変揮発オブジェクト生成表現式の $< Interface >$ フィールドに指定される入力端 (^IVobj) は，生成される不変揮発オブジェクトの第 0 世代に接続される．

また，同一の可視領域であるとは，可視領域を共有することに他ならない．不変揮発オブジェクトはその生成時に，生成者オブジェクト自身へのストリームの他に，**可視領域オブジェクト** (scope object) へのストリームが渡される．生成者オブジェクトも同様に，可視領域オブジェクトへのストリームを渡される．両者が共有するチャネル群は，この可視領域オブジェクトによって管理される．

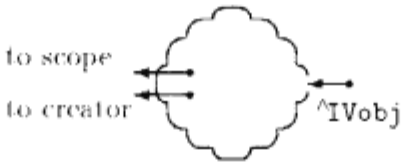

図 3.5: 不変揮発オブジェクトの生成

可視領域オブジェクト: 一般のオブジェクトはスロット名でスロット群を管理する．それと全く同様に，**可視領域オブジェクト** (scope object) は，変数名でチャネル群を管理するオブジェクトである．

可視領域オブジェクトの共有: 不変揮発クラスをあるメソッドに定義すると，生成される不変揮発オブジェクトがそのインターフェース・ストリームからどのメッセージを受信し，どのチャネルをアクセスするかは実行時に決まる．

例えば，生成オブジェクトとの間である共通のチャネル変数を参照しており，あるメソッドにはその出力端が 3 個出現し，別のメソッドにはその出力端が 2 個出現するとしよう．どちらの場合も関係するチャネル (ストリーム) はすべて完備化されなければならない．

さらに，不変揮発クラスの定義はネスト可能である．これを，可能な限りの場合数だけ別々なコードを静的に用意するのはコード最小化の方針から大きくずれてしまう．

問題をまとめると，物理的なコードは最小化し，かつ，チャネルはすべて完備化しなければならないということである．可視領域オブジェクトの導入をもってこれを解決した．つまり，同一の可視領域を有するオブジェクト間で一つの可視領域オブジェクトを共有することにし，実行時にしか決まらないことは実行時に決めることにした．

不変揮発オブジェクトにとっての自分自身:　生成者オブジェクトの自己アクセス順序に関する規則（文脈中の出現順序に依存）は次のようにして守られる．

- 不変揮発オブジェクトには，生成者オブジェクト自身へのストリームを連結分岐した先の片方である．
- 後の片方は生成者オブジェクトが保持し，その不変揮発クラスの定義の後に出現する自分自身へのアクセスに用いられる．
- 不変揮発オブジェクトは自分自身へのすべてのメッセージ送信後，これを閉鎖する．

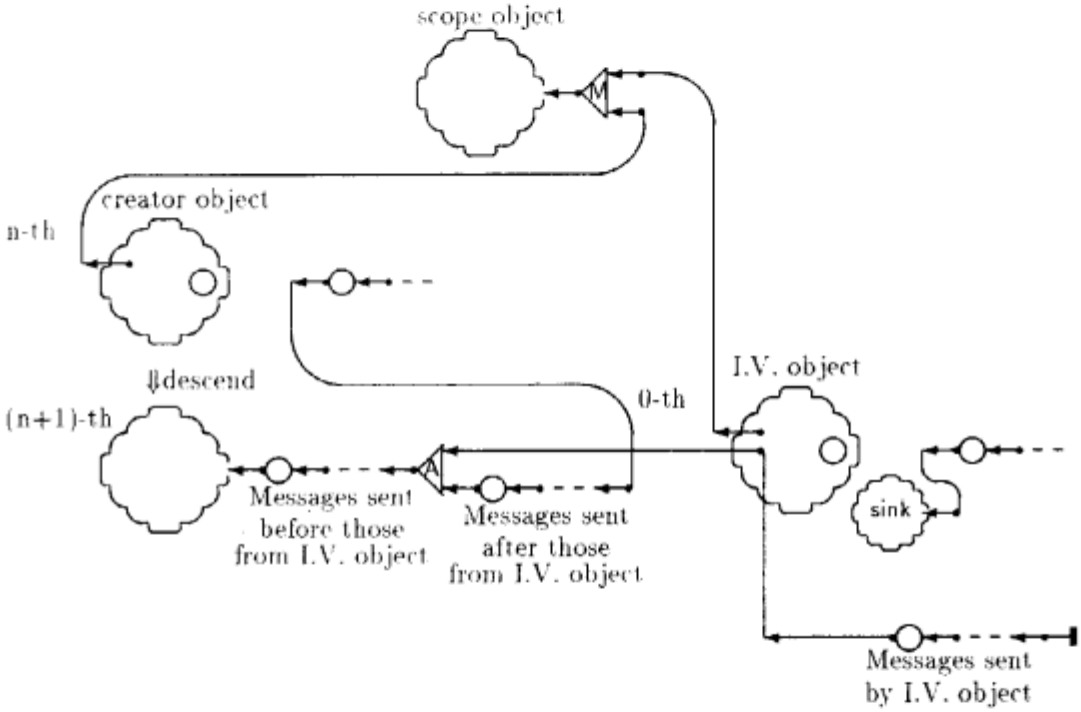

図 3.6: 不変揮発オブジェクトと生成者オブジェクト

不変揮発オブジェクトは，主に条件分岐の目的に使われる．以下に不変揮発オブジェクトの簡単な応用例を示す．

[応用例-2 (二進木ベクタの反転)]

下図に示すように、左側の二進木ベクタを反転して右側の二進木ベクタを得るプログラムを示す.

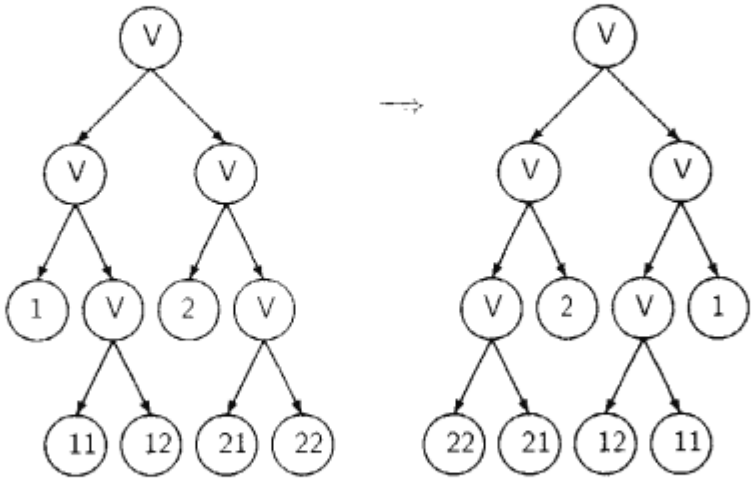

図 3.7: 二進木の反転

```
class  reverse.                                          % 1
    :do(^T, RevT) ->                                     % 2
        (class_of T) ? (                                 % 3
            :integer -> T = ^RevT ;                      % 4
            :vector ->  T:element(0, ^First),            % 5
                        #reverse:do(First, ^RevF),       % 6
                        T:element(1, ^Second),           % 7
                        #reverse:do(Second, ^RevS),      % 8
                       {RevS, RevF} = ^RevT              % 9
        ).                                               %10
end.                                                     %11

class  test.
    :test ->
        #reverse :do({{1,{11,12}}, {2,{21,22}}}, ^RevT),
        WhatIsRevT = (RevT ?),
        :nop(^WhatIsRevT).
    :nop(RevT) -> .
end.
```

図 3.8: 二進木反転のプログラム

各レベルで渡される木が葉であるか節であるかを調べ、

- 葉である場合、それ自身が反転木であり、

- 節である場合、その二つの枝それぞれの反転木を作り、それらの反転木から節の反転木を作ればよい。

クラス reverse の 3行目から10行目に不変揮発オブジェクトが定義されており、2行目から9行目に変数名 T で指定される 3個の出力端と1個の出力端の出現はいずれも同一のチャネルを表している。変数名 RevT に関しても同様である。

3.2.4.3 可変揮発オブジェクト

可変揮発オブジェクト (mutable volatile object) はそれ自身の世代を持ちうる、生成者オブジェクトとは独立のオブジェクトであり、生成者オブジェクトとは独立の可視領域を有する.

- **変数名の可視領域は独立**: 可変揮発クラスに出現する変数名と、その生成者オブジェクトのメソッド内の出現する同一の変数名は、別々のチャネルを意味する.
- **生成者はスロットの一つ**: 可変揮発オブジェクトはそれ自身の内部状態としてスロット群を保持することができる. 生成者オブジェクトは 擬似変数 `$creator` をスロット名とする一つの出力端スロットにすぎない. 一方、擬似変数 `$self` は可変揮発オブジェクトの自分自身を表す. このように、擬似変数 `$creator` と `$self` の意味は異なる.

可変揮発オブジェクトの生成: 可変揮発オブジェクトは、外部オブジェクトと全く同様に、生成された直後、暗黙裏に初期化メッセージが送信される. 初期化メッセージに対するメソッドもやはり同様に再定義可能である.

可変揮発オブジェクト生成表現式の $<$ *Interface* $>$ フィールドに指定される入力端 (`^MVobj`) は、この初期化メッセージの後に接続される.

不変揮発オブジェクトは、その生成時に、生成者オブジェクト自身へのストリームを渡される.

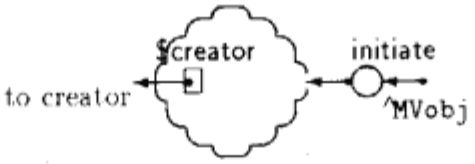

図 3.9: 可変揮発オブジェクトの生成

可変揮発オブジェクトにとっての自分自身:　生成者オブジェクトの自己アクセス順序に関する規則(文脈中の出現順序に依存)は、不変揮発オブジェクトの場合と全く同様にして守られる。すなわち、

- 可変揮発オブジェクトには生成者オブジェクト自身へのストリームを連結分岐した先の片方が渡される。
- 後の片方は生成者オブジェクトが保持し、その可変揮発クラスの定義の後に出現する自分自身へのアクセスに用いられる。
- 可変揮発オブジェクトの終了時、他の出力端スロットとともに出力端スロット `$creator` は閉鎖される。

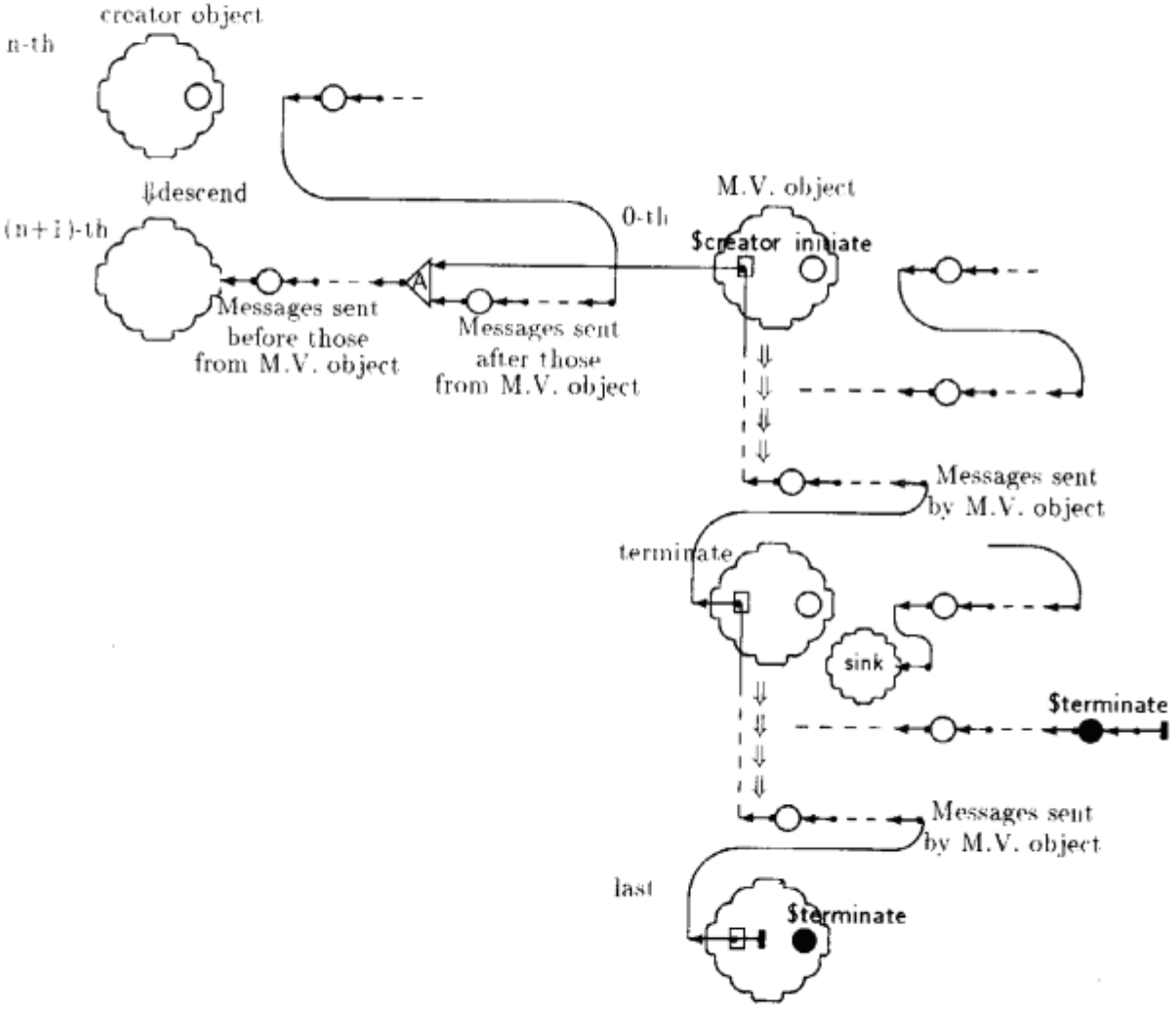

図 3.10: 可変揮発オブジェクトと生成者オブジェクト

可変揮発オブジェクトは、主に繰返しの目的に使われる。以下に、可変揮発オブジェクトの応用例を示す。

[応用例 -3 (素数の生成)]

指定される最大値 (max) までの素数列を生成するプログラムである.

1. まず 2 を送信後に 3 から始まる奇数列を生成する. これをふるいの列に通して得られるのが素数列である.

2. ふるいは新しい素数が見つかる度に一つ作られる. まず 3 のふるいが作られる.

3. ふるいは流れてくる数を自分の素数で割り,

   - 割り切れる (自分の倍数である) 場合, 何もしない.
   - 割り切れない場合, 自分が現時点で最大の素数であるかを調べる.

     自分が最大の素数である場合, 今流れてきた数は素数である. これに対するふるいを作り, これを "自分の次のふるい" とする.

     – そうでなければ, 今流れてきた数を "自分の次のふるい" に渡す.

ふるいは流れてくる数がなくなるまで, これを繰り返す.

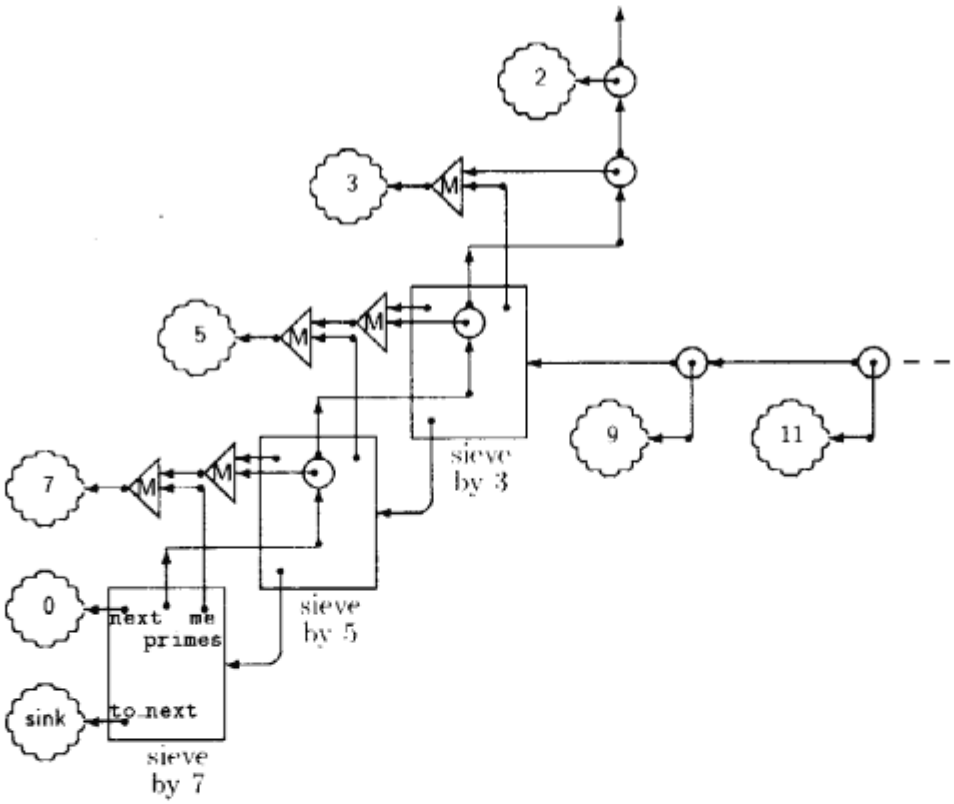

図 3.11: 素数生成のふるい列

```
class   prime.                                                    % 1
    :primes(^Max, ^Ps) ->                                        % 2
        3 = ^X,                                                  % 3
        :generate(X, Max, Ns),                                   % 4
        #sift:do(X, ^Ns, Ps:n(2):n(X) ).                         % 5
    :generate(^X, ^Max, ^Ns) ->                                  % 6
        ( (X+2 = ^NewX) < Max) ? (                               % 7
            : 'true  ->                                          % 8
                :generate(NewX, Max, Ns:n(NewX) ) ;              % 9
            : 'false ->        % end %                          %10
        ).                                                       %11
end.                                                             %12

class   sift.                                                    % 1
    :do(^V, Ns, ^Ps) ->                                          % 2
        S:initialize(V, Ps) = ^Ns,                               % 3
        ^S => (                                                  % 4
                out  me, next, to_next, primes ;                 % 5
            :initialize(^V, ^Ps) ->                              % 6
                V = !me, 0 = !next, Ps = !primes ;               % 7
            :n(^X) ->                                            % 8
                ( (X mod !me) == 0 ) ? (                         % 9
                    : 'true  ->  ;    % if divided, do nothing % %10
                    : 'false ->                                  %11
                        (!next == 0) ? (                         %12
                            : 'true  ->  % there is no next yet % %13
                                X = !next, Ns = !to_next,        %14
                                #sift:do(X, ^Ns, !primes:n(X)) ; %15
                            : 'false ->                          %16
                                !to_next:n(X)                    %17
                        )                                        %18
                )                                                %19
        ).                                                       %20
end.                                                             %21
```

図 3.12: 素数生成のプログラム

```
class  test.
    :test ->
        #prime:primes(20, Ps),
        :nop(^Res),
        Loop:initialize(Res) = ^Ps,
        ^Loop => (   '
                    out  result  ;
            :initialize(^Res) ->
                Res = !result  ;
            :n(^P) ->
                (!result \ (P?)) = !result
        ).
    :nop(Res) -> .
end.
```

図 3.13: 素数生成のテスト・プログラム

クラス sift の4行目から20行目に可変揮発オブジェクトが定義されており、またその中の9行目から19行目に不変揮発オブジェクトが一つ、さらにその中の 12行目から18行目に別の不変揮発オブジェクトが定義されている。可変揮発オブジェクトの外側(2行目から3行目)に出現する変数 V, Ns, Ps は内側(4行目から20行目)に出現する変数 V, Ns, Ps とは独立であり、メッセージ initialize を介して内側に伝えられる。

第 4 章

処理系

4.1 抽象命令

2章で述べた計算モデルは $A'UM$ の抽象機械を示しており、$A'UM$ プログラムは同章に示したように、事象の観測とストリーム演算、原始オブジェクト生成および 次世代生成から成る動作列に展開される。以下に、これらの抽象命令列をまとめる。

表 4.1: 抽象命令列

category	*instruction*
declaration	`entry(X, S)`
event	`receive(X, m, Y)`
	`is_closed(X)`
action	`send(X, m, Y)`
	`close(X)`
	`connect(X, Y)`
	`merge(X, Y, Z)`
	`append(X, Y, Z)`
	`descend(X, S)`

4.2 KL1 上処理系

前節の抽象命令列 を KL1 コードを生成する KL1 上言語処理系を作成した.

4.2.1 KL1 コードの生成

- **クラス**:
  - 外部クラスを KL1 の 1 モジュール (module) として実現した. "@" にクラス名を付けたものをモジュール名としている.
    例えば, クラス reverse のモジュール名は @reverse である.
  - 揮発クラスはそれを定義している外部クラスと同一モジュール内に実現した. 外部クラスのモジュール名にその出現レベル数を付け加えたものを揮発クラス名としている.
    例えば, クラス reverse の 3 行目から 10 行目は一番目のメソッドの一番目に定義される不変揮発クラスであるから, @reverse_1@i1 というクラス名が与えられる.

  ここで, これらの内部表現で使われるクラス名を内部クラス名と呼ぶ.

- **オブジェクト**: オブジェクトは,

  ```
  @reverse(Interface, SlotInfo, Global, Creater_and_Scope, ClassInfo)
  ```

  のように内部クラス名を述語名とう引数の手続き (procedure) で表現している. 外部オブジェクトと揮発オブジェクトは全く引数列からなっており, 区別はない.

- **原始オブジェクト**: 原始オブジェクトはその原始クラス名をモジュール名とするモジュール内に一般オブジェクト同様にプロセスとして定義している.

- **基本ストリーム演算**: ストリームは KL1 のリストを使って実現している.
  - 送信 (send(X, m, ^Y)) は, ボディ部 X = [m|Y]
  - 閉鎖 (close(X)) は, ボディ部 X = []
  - 接続 (connect(X, ^Y)) は, ボディ部 X = Y
  - 受信 (receive(^X, m, Y)) は, ガード部 X = [m|Y]
  - 閉鎖検出 (is_closed(^X)) は, ガード部 X = []

- **ジョイント**:
  - 併合 (merge(^X, ^Y, Z)) は, 前述のように.
    * $f : Chain \times Chain \rightarrow Poset$ に対して, 併合演算の出現毎に, KL1 組込み述語 merge(X, Y, W) すなわち W = {X, Y} を生成する.
    * $g : Poset \rightarrow Chain$ に対して, KL1 組込み述語 merge_in(W, Z) をオブジェクト生成時インタフェース・ストリームの受信口に挿入する.
      ただし, merge_in(W, Z) はベクタ構造を解釈しストリームを生成する.

- 連結(append(^X, ^T, Z))は、*A'UM* 用組込述語の一つとしてモジュール $blt 内に '@append'/3 を定義する. 連結演算出現の際に、'$blt':'@append'(X, Y, Z) を生成する.

- **メッセージ**: メッセージに次のようにメッセージと各引数にタグを付加することによりメッセージを識別している.

  - 各メッセージには、原始メッセージおよび合成メッセージを区別するために、メッセージ・タグ a あるいは c が付加される.
  - 各引数には、それが入力端、出力端あるいは原始データであるかを示すために、引数タグ i, o あるいは m が付加される.

- **クラス継承**:

  オブジェクトの第3引数はメソッド探索表を含む継承情報

  - 下界クラスおよび上界クラスはそれぞれ $OBJECT と $object をモジュール名とするクラスである.

以降に、これまでに挙げた三つの例題プログラム、カウンタ、二進木反転、素数生成のそれぞれに対して生成される KL1 コードを示す.

```
:- module '@counter' .
:- public '@counter'/1 ',' '@counter'/5 .

'@counter'(A) :- true |
        '$OBJECT_x':new($('@counter','@counter',['@counter'],
        ['@counter!n'],[[c(show({o})),c(set({i})),a(down),a(up)]],[n]),A) .

'@counter'([a(up)|H],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@counter!n'),M})|L] ,
        '@integer':new(i(1),{N,C,O}) ,
        M=[c(add(i(o)),{N,P})|Q] ,
        L=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),Q})|R] ,
        R=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),P})|H] ,
        '$OBJECT':descend(G,B,O,$(D,E),F) .

'@counter'([a(down)|H],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@counter!n'),M})|L] ,
        '@integer':new(i(1),{N,C,O}) ,
        M=[c(sub(i(o)),{N,P})|Q] ,
        L=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),Q})|R] ,
        R=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),P})|H] ,
        '$OBJECT':descend(G,B,O,$(D,E),F) .

'@counter'([c(set({i}),{L})|H],B,C,$(D,E),F) :- true |
        G=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),L})|H] ,
        '$OBJECT':descend(G,B,C,$(D,E),F) .

'@counter'([c(show({o}),{L})|H],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@counter!n'),L})|H] ,
        '$OBJECT':descend(G,B,C,$(D,E),F) .
```

図 4.1: クラス counter の KL1 生成コード

```
:- module '@test' .
:- public '@test'/1 ',' '@test'/5 .

'@test'(A) :- true |
        '$OBJECT_x':new($('@test','@test',['@test'],[],
        [[c(nop({o})),c(testA(o(o))),c(testM(o(o))),a(test)]],[n]),A) .

'@test'([a(test)|H],B,C,$(D,E),F) :- true |
        M=[c(testM(o(o)),{N,O})|P] ,
        P=[c(testA(o(o)),{S,T})|U] ,
        U=[c(nop({o}),{X})|H] ,
        N=[c(who_are_you({i}),{BD})] ,
        O=[c(who_are_you({i}),{BI})] ,
        S=[c(who_are_you({i}),{BN})] ,
        T=[c(who_are_you({i}),{BS})] ,
        '$blt':'@append'(BN,BS,CG) ,
        '$blt':'@append'(BI,CG,CF) ,
        '$blt':'@append'(BD,CF,X) ,
        '$OBJECT':descend(M,B,C,$(D,E),F) .
'@test'([c(testM(o(o)),{L,N})|H],B,C,$(D,E),F) :- true |
        '@class':new(m(a(counter)),{P,C,Q}) ,
        P=[c(new({o}),{R})] ,
        '@integer':new(i(5),{T,Q,U}) ,
        R=[c(set({i}),{T})|V] ,
        Y=[a(up)|Z] ,
        Z=[a(up)|BA] ,
        BA=[c(show({o}),{L})] ,
        BC=[a(down)|BD] ,
        BD=[a(down)|BE] ,
        BE=[c(show({o}),{N})] ,
        merge_in(BC,Y,V) ,
        '$OBJECT':descend(H,B,U,$(D,E),F) .
'@test'([c(testA(o(o)),{L,N})|H],B,C,$(D,E),F) :- true |
        '@class':new(m(a(counter)),{P,C,Q}) ,
        P=[c(new({o}),{R})] ,
        '@integer':new(i(5),{T,Q,U}) ,
        R=[c(set({i}),{T})|V] ,
        Y=[a(up)|Z] ,
        Z=[a(up)|BA] ,
        BA=[c(show({o}),{L})] ,
        BC=[a(down)|BD] ,
        BD=[a(down)|BE] ,
        BE=[c(show({o}),{N})] ,
        '$blt':'@append'(Y,BC,V) ,
        '$OBJECT':descend(H,B,U,$(D,E),F) .
'@test'([c(nop({o}),{L})|H],B,C,$(D,E),F) :- true |
        '$$sink':new({L,N,[]}) ,
        merge_in(N,I,C) ,
        '$OBJECT':descend(H,B,I,$(D,E),F) .
```

図 4.2: クラス counter のためのクラスtestの KL1 生成コード

```
:- module '@reverse' .
:- public '@reverse'/1 ',' '@reverse'/5 ','
          '@reverse_1@i1'/1 ',' '@reverse_1@i1'/5 .

'@reverse'(A) :- true |
        '$OBJECT_x':new($('@reverse','@reverse',['@reverse'],[],
        [[c(do(i(o)))]],[n]),A) .
'@reverse'([c(do(i(o)),{L,M})|H],B,C,$(D,E),F) :- true |
        O=[c(class({o}),{P})] ,
        P=[c(who_are_you({i}),{R})] ,
        V=[c('$get_inlet_var'(m(i)),{a('X'),L})|BB] ,
        BB=[c('$get_moutlet_var'(m(o)),{a('X'),O})|BC] ,
        BC=[c('$get_moutlet_var'(m(o)),{a('RevX'),M})] ,
        '$$scope':new({"@reverse_1",['X','RevX'],[ [],[]]},{K,W}) ,
        merge_in(W,I,U) ,
        '@reverse_1@i1'({R,$(C,U),$(G,H),$(K,V)}) ,
        '$OBJECT':descend(G,B,T,$(D,E),F) .

'@reverse_1@i1'(A) :- true |
        '$OBJECT_i':new($('@reverse','@reverse_1@i1',['@reverse_1@i1'],[],
        [[a(vector),a(integer)]],[n]),A) .
'@reverse_1@i1'([a(integer)|G],B,C,$(D,E),F) :- true |
        E=[c('$get_moutlet_var'(m(o)),{a('X'),U})|T] ,
        T=[c('$get_inlet_var'(m(i)),{a('RevX'),U})] ,
        '$OBJECT':terminate(G,B,C,$(D,[]),F) .
'@reverse_1@i1'([a(vector)|G],B,C,$(D,E),F) :- true |
        '@integer':new(i(O),{N,C,O}) ,
        L=[c(element(i(o)),{N,P})] ,
        '@class':new(m(a(reverse)),{R,O,S}) ,
        R=[c(new({o}),{T})] ,
        T=[c(do(i(o)),{P,W})] ,
        '@integer':new(i(1),{Z,S,BA}) ,
        Y=[c(element(i(o)),{Z,BB})] ,
        '@class':new(m(a(reverse)),{BD,BA,BE}) ,
        BD=[c(new({o}),{BF})] ,
        BF=[c(do(i(o)),{BB,BI})] ,
        '@vector':new(v([BI,W]),{BM,BE,BN}) ,
        E=[c('$get_moutlet_var'(m(o)),{a('X'),BW})|BV] ,
        BV=[c('$get_inlet_var'(m(i)),{a('RevX'),BM})] ,
        merge_in(Y,L,BW) ,
        '$OBJECT':terminate(G,B,BN,$(D,[]),F) .
```

図 4.3: クラス reverse の KL1 生成コード

```
:- module '@test' .
:- public '@test'/1 ',' '@test'/5 .

'@test'(A) :- true |
        '$OBJECT_x':new($('@test','@test',['@test'],[],
        [[c(nop({o})),a(test)]],[n]),A) .

'@test'([a(test)|H],B,C,$(D,E),F) :- true |
        '@class':new(m(a(reverse)),{L,C,M}) ,
        L=[c(new({o}),{N})] ,
        '@integer':new(i(1),{P,M,Q}) ,
        '@integer':new(i(11),{R,Q,S}) ,
        '@integer':new(i(12),{T,S,U}) ,
        '@vector':new(v([R,T]),{V,U,W}) ,
        '@vector':new(v([P,V]),{X,W,Y}) ,
        '@integer':new(i(2),{Z,Y,BA}) ,
        '@integer':new(i(21),{BB,BA,BC}) ,
        '@integer':new(i(22),{BD,BC,BE}) ,
        '@vector':new(v([BB,BD]),{BF,BE,BG}) ,
        '@vector':new(v([Z,BF]),{BH,BG,BI}) ,
        '@vector':new(v([X,BH]),{BJ,BI,BK}) ,
        N=[c(do(i(o)),{BJ,BL})] ,
        BL=[c(who_are_you({i}),{BQ})] ,
        BU=[c(nop({o}),{BQ})|H] ,
        '$OBJECT':descend(BU,B,BK,$(D,E),F) .

'@test'([c(nop({o}),{L})|H],B,C,$(D,E),F) :- true |
        '$$sink':new({L,N,[]}) ,
        merge_in(N,I,C) ,
        '$OBJECT':descend(H,B,I,$(D,E),F) .
```

図 1.1: クラス reverse のためのクラス test の KL1 生成コード

```
:- module '@prime' .
:- public '@prime'/1 ',' '@prime'/5 ','
          '@prime_2@i1'/1 ',' '@prime_2@i1'/5 .

'@prime'(A) :- true |
        '$OBJECT_x':new($('@prime','@prime',['@prime'],[],
        [[c(generate(i(i,i))),c(primes(i(i)))]],[n]),A) .
'@prime'([c(primes(i(i)),{L,M})|H],B,C,$(D,E),F) :- true |
        '@integer':new(i(3),{N,C,O}) ,
        S=[c(generate(i(i,i)),{T,L,V})|H] ,
        '@class':new(m(a(sift)),{BA,O,BB}) ,
        BA=[c(new({o}),{BC})] ,
        '@integer':new(i(2),{BG,BB,BH}) ,
        M=[c(n({i}),{BG})|BI] ,
        BI=[c(n({i}),{BJ})|BK] ,
        BC=[c(do(i(o,i)),{BE,V,BK})] ,
        merge_in(BE,T,BU) ,
        merge_in(BJ,BU,N) ,
        '$OBJECT':descend(S,B,BH,$(D,E),F) .
'@prime'([c(generate(i(i,i)),{L,M,N})|H],B,C,$(D,E),F) :- true |
        '@integer':new(i(2),{P,C,Q}) ,
        L=[c(add(i(o)),{P,R})] ,
        U=[c(lt(i(o)),{V,W})] ,
        W=[c(who_are_you({i}),{Y})] ,
        BC=[c('$get_inlet_var'(m(i)),{a('Max'),M})|BI] ,
        BI=[c('$get_moutlet_var'(m(o)),{a('Max'),V})|BJ] ,
        BJ=[c('$get_inlet_var'(m(i)),{a('Ns'),N})|BN] ,
        BN=[c('$get_inlet_var'(m(i)),{a('NewX'),T})] ,
        '$$scope':new({"@prime_2",['Max','Ns','NewX'],[ [],[],[]]},{K,BD}) ,
        merge_in(BD,I,BB) ,
        merge_in(T,U,R) ,
        '@prime_2@i1'({Y,$(Q,BB),$(G,H),$(K,BC)}) ,
        '$OBJECT':descend(G,B,I,$(D,E),F) .

'@prime_2@i1'(A) :- true |
        '$OBJECT_i':new($('@prime','@prime_2@i1',['@prime_2@i1'],[],
        [[b(false),b(true)]],[n]),A) .
'@prime_2@i1'([b(true)|G],B,C,$(D,E),F) :- true |
        L=[c(n({i}),{N})|P] ,
        D=[c(generate(i(i,i)),{U,V,P})|Y] ,
        E=[c('$get_moutlet_var'(m(o)),{a('Max'),V})|BF] ,
        BF=[c('$get_moutlet_var'(m(o)),{a('Ns'),L})|BI] ,
        BI=[c('$get_moutlet_var'(m(o)),{a('NewX'),BO})] ,
        merge_in(U,N,BO) ,
        '$OBJECT':terminate(G,B,C,$(Y,[]),F) .
'@prime_2@i1'([b(false)|G],B,C,$(D,E),F) :- true |
        E=[] ,
        '$OBJECT':terminate(G,B,C,$(D,[]),F) .
```

図 1.5: クラス prime の KL1 生成コード

```
:- module '@sift' .
:- public '@sift'/1 ',' '@sift'/5 ','
          '@sift_1@m1'/1 ',' '@sift_1@m1'/5 ','
          '@sift_1@m1_2@i1'/1 ',' '@sift_1@m1_2@i1'/5 ','
          '@sift_1@m1_2@i1_2@i1'/1 ',' '@sift_1@m1_2@i1_2@i1'/5 .

'@sift'(A) :- true |
        '$OBJECT_x':new($('@sift','@sift',['@sift'],[],
        [[c(do(i(o,i)))]],[n]),A) .

'@sift'([c(do(i(o,i)),{L,M,O})|H],B,C,$(D,E),F) :- true |
        P=[c(initialize(i(i)),{L,O})|M] ,
        '@sift_1@m1'({P,$(C,W),$(G,H)}) ,
        '$OBJECT':descend(G,B,W,$(D,E),F) .

'@sift_1@m1'(A) :- true |
        '$OBJECT_m':new($('@sift','@sift_1@m1',['@sift_1@m1'],
        ['@sift_1@m1!me','@sift_1@m1!next','@sift_1@m1!to_next',
        '@sift_1@m1!primes'],[[c(n({i})),c(initialize(i(i)))]],[n]),A) .

'@sift_1@m1'([c(initialize(i(i)),{L,M})|J],B,C,$(D,E),F) :- true |
        G=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!me'),L})|O] ,
        '@integer':new(i(O),{R,C,S}) ,
        O=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!next'),R})|T] ,
        T=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!primes'),M})|J] ,
        '$OBJECT':descend(G,B,S,$(D,E),F) .

'@sift_1@m1'([c(n({i}),{L})|J],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@sift_1@m1!me'),O})|N] ,
        M=[c(mod(i(o)),{O,P})] ,
        '@integer':new(i(O),{R,C,S}) ,
        P=[c(eq(i(o)),{R,T})] ,
        T=[c(who_are_you({i}),{V})] ,
        Z=[c('$get_inlet_var'(m(i)),{a('X'),L})|BF] ,
        BF=[c('$get_moutlet_var'(m(o)),{a('X'),M})] ,
        '$$scope':new({"@sift_1@m1_2",['X'],[ []]},{K,BA}) ,
        merge_in(BA,H,Y) ,
        '@sift_1@m1_2@i1'({V,$(S,Y),$(N,J),$(K,Z)}) ,
        '$OBJECT':descend(G,B,H,$(D,E),F) .
```

図 1.6: クラス sift の KL1 生成コード (つづく)

```
'@sift_1@m1_2@i1'(A) :- true |
        '$OBJECT_i':new($('@sift','@sift_1@m1_2@i1',['@sift_1@m1_2@i1'],[],
        [[b(false),b(true)]],[n]),A) .

'@sift_1@m1_2@i1'([b(true)|G],B,C,$(D,E),F) :- true |
        E=[] ,
        '$OBJECT':terminate(G,B,C,$(D,[]),F) .

'@sift_1@m1_2@i1'([b(false)|G],B,C,$(D,E),F) :- true |
        D=[c('$get_outlet_slot'(m(o)),{a('@sift_1@m1!next'),M})|L] ,
        '@integer':new(i(0),{N,C,O}) ,
        M=[c(eq(i(o)),{N,P})|Q] ,
        L=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!next'),Q})|R] ,
        P=[c(who_are_you({i}),{U})] ,
        '@sift_1@m1_2@i1_2@i1'({U,$(O,X),$(R,W),$(E,[])}) ,
        '$OBJECT':terminate(G,B,X,$(W,[]),F) .

'@sift_1@m1_2@i1_2@i1'(A) :- true |
        '$OBJECT_i':new($('@sift','@sift_1@m1_2@i1_2@i1',
        ['@sift_1@m1_2@i1_2@i1'],[],[[b(false),b(true)]],[n]),A) .

'@sift_1@m1_2@i1_2@i1'([b(true)|G],B,C,$(D,E),F) :- true |
        D=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!next'),O})|N] ,
        N=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!to_next'),T})|S] ,
        '@class':new(m(a(sift)),{V,C,W}) ,
        V=[c(new({o}),{X})] ,
        S=[c('$get_outlet_slot'(m(o)),{a('@sift_1@m1!primes'),BB})|BA] ,
        BB=[c(n({i}),{BC})|BD] ,
        BA=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!primes'),BF})|BE] ,
        X=[c(do(i(o,i)),{Z,T,BG})] ,
        F=[c('$get_moutlet_var'(m(o)),{a('X'),BQ})] ,
        merge_in(Z,O,BO) ,
        merge_in(BC,BO,BQ) ,
        merge_in(BF,BG,BD) ,
        '$OBJECT':terminate(G,B,W,$(BE,[]),F) .

'@sift_1@m1_2@i1_2@i1'([b(false)|G],B,C,$(D,E),F) :- true |
        D=[c('$get_outlet_slot'(m(o)),{a('@sift_1@m1!to_next'),M})|L] ,
        M=[c(n({i}),{N})|P] ,
        L=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!to_next'),P})|Q] ,
        E=[c('$get_moutlet_var'(m(o)),{a('X'),N})] ,
        '$OBJECT':terminate(G,B,C,$(Q,[]),F) .
```

図 1.7: クラス sift の KL1 生成コード (つづき)

```
:- module '@test' .
:- public '@test'/1 ',' '@test'/5 ','
          '@test_1@m1'/1 ',' '@test_1@m1'/5 .

'@test'(A) :- true |
        '$OBJECT_x':new($('@test','@test',['@test'],[],
        [[c(nop({o})),a(test)]],[n]),A) .

'@test'([a(test)|H],B,C,$(D,E),F) :- true |
        '@class':new(m(a(prime)),{L,C,M}) ,
        L=[c(new({o}),{N})] ,
        '@integer':new(i(20),{P,M,Q}) ,
        N=[c(primes(i(i)),{P,R})] ,
        V=[c(nop({o}),{W})|X] ,
        BA=[c(initialize({i}),{W})|R] ,
        '@test_1@m1'({BA,$(Q,BG),$(X,H)}) ,
        '$OBJECT':descend(V,B,BG,$(D,E),F) .

'@test'([c(nop({o}),{L})|H],B,C,$(D,E),F) :- true |
        '$$sink':new({L,N,[]}) ,
        merge_in(N,I,C) ,
        '$OBJECT':descend(H,B,I,$(D,E),F) .

'@test_1@m1'(A) :- true |
        '$OBJECT_m':new($('@test','@test_1@m1',['@test_1@m1'],
        ['@test_1@m1!result'],[[c(n({i})),c(initialize({i}))]],[n]),A) .

'@test_1@m1'([c(initialize({i}),{L})|J],B,C,$(D,E),F) :- true |
        G=[c('$set_outlet_slot'(m(i)),{a('@test_1@m1!result'),L})|J] ,
        '$OBJECT':descend(G,B,C,$(D,E),F) .

'@test_1@m1'([c(n({i}),{L})|J],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@test_1@m1!result'),N})|M] ,
        L=[c(who_are_you({i}),{P})] ,
        M=[c('$set_outlet_slot'(m(i)),{a('@test_1@m1!result'),T})|J] ,
        '$blt':'@append'(P,T,N) ,
        '$OBJECT':descend(G,B,C,$(D,E),F) .
```

図 4.8: クラス prime ためのクラスtestの KL1 生成コード

4.2.2 処理系の起動

現在、PIMOS 開発支援システム PDSS 上に $\mathcal{A}'\mathcal{UM}$ の試験版処理系が実装されている。本処理系では、$\mathcal{A}'\mathcal{UM}$ プログラムのコンパイル、ロード、テストを行なうための小さな環境 ($\mathcal{A}'\mathcal{UM}$-SHELL) を用意している。ここに、処理系およびプログラムのテスト方法を概略する (処理系のトップレベル・ディレクトリのファイル README を参照)。

1. **処理系の起動:**

   (a) Emacs から `M-x pdss` を入力して、PDSS を起動する。

   (b) まず、カレント・ディレクトリを $\mathcal{A}'\mathcal{UM}$ 処理系のトップレベル・ディレクトリ (ここでは `~yoshida/aum/new`) に移す。

   ```
   :- cd(''~yoshida/aum/new'').
   ```

   (c) 次のコマンドにより、処理系をロードする。

   ```
   :- take(''Startup'').
   ```

   (d) 次のコマンドにより、$\mathcal{A}'\mathcal{UM}$-SHELL を起動する。

   ```
   :- take(''IPL'').
   ```

   すると、$\mathcal{A}'\mathcal{UM}$-SHELL ウィンドウが現れる。

2. **プログラムのコンパイルおよびロード:** $\mathcal{A}'\mathcal{UM}$ SHELL は、

   - ファイル環境の設定、参照
   - KL1 プログラムのコンパイル、ロード
   - $\mathcal{A}'\mathcal{UM}$ プログラムのコンパイル、ロード、KL1 モードでのトレース設定
   - $\mathcal{A}'\mathcal{UM}$ プログラムの起動

   を行なうための環境である。現在 $\mathcal{A}'\mathcal{UM}$-SHELL で支援しているコマンド群を表 4.2 にまとめる。

3. **プログラムの起動:** $\mathcal{A}'\mathcal{UM}$-SHELL から `start` コマンドはクラス `test` のインスタンスを生成し、それにメッセージ `test` を送信する。

   したがって、これまでの例に示したように、$\mathcal{A}'\mathcal{UM}$ プログラムはクラス `test` のメソッド `test` をそのトップレベルとして定義することによりテストできる。

4. **プログラムのデバッグ:** $\mathcal{A}'\mathcal{UM}$ のためのデバッギング環境はまだ用意していない。KL1 デバッガ (PDSS-CONSOLE) を使ってテストされたい。

表 4.2: A'UM-SHELL コマンド群

分類	コマンド	意味
ディレクトリ	cd(DirectoryNameString)	ディレクトリの設定
	pwd	現ディレクトリの場所の確認
	ls	現ディレクトリ下の資源リストの表示
	ls(DirectoryNameString)	指定ディレクトリ下の資源リストの表示
A'UM	aumcmp(AumFileNameStrings)	*A'UM*プログラムのコンパイルかつロード
	load_class(AumClassNameAtoms)	*A'UM*クラス(コンパイル済みプログラム)のロード
	trace_class(AumClassNameAtoms)	ロード済みの*A'UM*クラスへのトレース設定
	notrace_class(AumClassNameAtoms)	ロード済みの*A'UM*クラスのトレース解除
	start	トップレベル・プログラムの起動
KL1	kl1cmp(KL1FileNameStrings)	KL1プログラムのコンパイルかつロード
	load_kl1(Kl1ModuleNameAtoms)	KL1 モジュール(コンパイル済みプログラム)のロード
	trace_kl1(KL1ModuleNameAtoms)	ロード済みのKL1モジュールへのトレース設定
	notrace_kl1(KL1ModuleNameAtoms)	ロード済みのKL1モジュールのトレース解除
その他	halt	*A'UM*-SHELL からの脱出

例えば、素数生成のプログラムをコンパイルして実行してみよう.

```
****************************************
*        Welcome to A'UM World!        *
*                                      *
*    A'UM Shell: Ver.0.5 (01/03/89)    *
*   A'UM Compiler: Ver.0.5 (1/03/89)   *
****************************************

:- cd("@sample").
world = /login3/yoshida/aum/new/@sample

:- aumcmp(["prime.aum", "sift.aum", "test.prime"]).
... compiling A'UM (/login3/yoshida/aum/new/@sample/prime.aum) ...
<<< compiled to KL1-C (/login3/yoshida/aum/new/@sample/@prime.kl1) >>>
<<< compiled to KL1-B (/login3/yoshida/aum/new/@sample/@prime.asm) >>>
<<< done >>>
... compiling A'UM (/login3/yoshida/aum/new/@sample/sift.aum) ...
<<< compiled to KL1-C (/login3/yoshida/aum/new/@sample/@sift.kl1) >>>
<<< compiled to KL1-B (/login3/yoshida/aum/new/@sample/@sift.asm) >>>
<<< done >>>
... compiling A'UM (/login3/yoshida/aum/new/@sample/test.prime) ...
<<< compiled to KL1-C (/login3/yoshida/aum/new/@sample/@test.kl1) >>>
<<< compiled to KL1-B (/login3/yoshida/aum/new/@sample/@test.asm) >>>
<<< done >>>

:- trace_class([test]).

:- start.
```

図 4.9: 素数生成プログラムのコンパイルと実行

KL1 デバッガでトレースした結果を以下に示す.

```
***** PDSS-KL1 V1.65 (Tue Feb 28 15:28:30 GMT+9:00 1989) *****
.................................................
>>>>> Priority: 3987 <<<<<
Call  : [0108]@test:@test({[a(test)]x,A}).  [step]%
Call  : [0108]@test:@test([a(test)x],$({},{}),A,$(B^,C^),$(@test,@test,
$({@test},{{a(...)x,c(...)x}},{n}))).  [step]%
Call  : [0108]@test:@test_1@m1([c(initialize({i}),{A})x,c(n({i})x,{B^}),
c(n({i})x,{C^})|D],$({@test_1@m1!result},{E^}),F,$(G^,H^),$(@test,@test_1@m1,
$({@test_1@m1},{{c(...)x,c(...)x}},{n}))).  [step]%
Call  : [0108]@test:@test_1@m1([c(n({i}),{A^})x,c(n({i})x,{B^})|C],$({
@test_1@m1!result},{Dx}),E,$(F^,G^),$(@test,@test_1@m1,$({@test_1@m1},
{{c(...)x,c(...)x}},{n}))).  [step]%
                              .
                              .
                              .
Call  : [0108]@test:@test_1@m1([c(n({i}),{A^})x|B],$({@test_1@m1!result},
{C}),D,$(E^,F^),$(@test,@test_1@m1,$({@test_1@m1},{{c(...)x,c(...)x}},{n}))).
  [step]%
Call  : [0108]@test:@test([c(nop({o}),{{[],[...]}})x],$({},{}),A,$(B^,C^),
$(@test,@test,$({@test},{{a(...)x,c(...)x}},{n}))).  [step]% w 100 100
Call  : [0108]@test:@test([c(nop({o}),{{[],[i(2),i(3),i(5),i(7),i(11),i(13),
i(17),i(19)]}})x],$({},{}),A,$(B^,C^),$(@test,@test,$({@test},{{a(test)x,
c(nop({o}))x}},{n}))).  [step]%
```

図 1.10: 素数生成プログラムの実行トレース

参考文献

[Birkhoff79] Garrett Birkhoff: *Lattice Theory*, American Mathematical Society, Colloquium Publications, Vol. 25, Third edition, 1979

[Yoshida87] Kaoru Yoshida and Takashi Chikayama: *A'UM — Parallel Object-Oriented Language upon KL1* —, 情報処理学会プログラミング言語研究会 14-4, Dec. 1987

[Yoshida88a] 吉田かおる, 近山隆: "並列オブジェクト指向言語 *A'UM*", 情報処理学会第36回全国大会予稿集, 1988年3月

[Yoshida88b] Kaoru Yoshida and Takashi Chikayama: *A'UM — A Stream-Based Concurrent Object-Oriented Language* —, Proc. of FGCS'88, Nov. 1988

[Yoshida89a] Kaoru Yoshida and Takashi Chikayama: *A'UM = Stream + Object + Relation*, ACM SIGPLAN Notices, Feb. 1989