ICOT Technical Memorandom: TM-1206

TM-1206

aya 第 1 版解説書

寿崎　かすみ、近山　隆

August, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

目次

第 1 章

はじめに

aya はプロセスとプロセス間のメッセージ通信でプログラムを記述する‘プロセス指向プログラミング’のために設計した言語である．aya では，プロセスとそのあいだのメッセージ通信を直接しかも自然に記述できるようにしている．

現在の aya は KL1 にコンパイルして実行する．

この解説書は，KL1 プログラミングの経験のある読者を対象としている．

第 2 章

簡単なプログラム

この章では aya で記述した簡単な例を見ながら, aya でプログラムを記述するさいに必要となる基本的なことを学ぶ.

2.1 簡単なプログラム

aya ではプログラムをプロセスとプロセス間のメッセージにより記述する. プロセスはメッセージが到着するのを待ち, 到着したメッセージに対応した処理を行う.

つぎの例は, 入力が 0 であれば 1 を, 1 であれば 0 を出力するプロセス 'not/2' である.

```
module not.
public not/2.

class not(In,Out)
      with +in := In, -out := Out.
    -> @in = 0 | @out <- 1 \\ .
    -> @in = 1 | @out <- 0 \\ .
end class.
```

はじめの 2 行は, モジュール宣言とモジュール間にわたってプロセスの呼びだしをおこなうためのパブリック宣言である.

```
module not.
public not/2.
```

aya ではプログラムを, モジュールに分割して記述する. モジュールは 1 つ以上のプロセスの集まりである. プロセスのうちモジュール間にわたって呼びだしを行いたいものはプログラムの先頭にそのクラス名と引数を指定してパブリック宣言をする必要が有る.

'class' から 'end class' までがプロセスの定義である. プロセスはこのようにプロセスのクラスで定義する.

プロセスのクラスがプログラム中で呼び出されると, そのクラスに対応したプロセスが生成される. プロセスのクラスはクラス名とクラス呼びだしのときにパラメタを受けとる引数の数で識別する. クラス名が同じでも引数の個数が異なれば, 異なったクラスとして扱う.

2.2 プロセスの記述

プロセスのクラスの定義は 2 つの部分にわかれる. クラス名, 初期化処理などを記述した部分とメソッド定義である.

2.2.1 クラス名・初期化処理

プロセスのクラス名は予約語 'class' のあとに引数とともに記述する．このあとにソケット宣言，プロセスの初期化処理を記述する．ソケットとはプロセスがプロセス内にさまざまな値を保持するためのホルダーである．

```
class not(In,Out)
      with +in := In, -out := Out.
```

この例では 'in' と 'out' の 2 つのソケットを宣言し，引数として受けとった値を初期値として設定するよう指定している．初期化処理は記述していない．プロセスは初期化処理が終了すると，定常状態になる．ほかからのメッセージはこの定常状態で受け付ける．定常状態は，メッセージとそれに対応した処理の定義 (メソッド) のならびとなる．

2.2.2 メソッド定義

受信したメッセージとそれに対応した処理の定義がメソッドである．実際のプロセスの動作はメソッドで記述する．プロセス 'not/2' には 2 つのメソッドが定義されている．

```
    -> @in = 0 | @out <- 1 \\ .
    -> @in = 1 | @out <- 0 \\ .
```

このメソッドはそれぞれ，0 を受けとったときは 1 を，1 を受けとったときは 0 を出力し，プロセスの実行を終了するように指定している．

2.3 ソケットとメッセージ交換

2.3.1 ソケット

プロセスはデータを保持するためにソケットを持つことができることは前に述べた．このソケットにはつぎの 2 通りの使い方がある．

(1) プロセス内の値のホルダー．

(2) プロセス間の通信チャネル．

(1) の場合は，プロセスが自分で値を入れ，必要になったときに読み出して使用する．(2) の場合は，プロセス間で共有する変数を保持し，それを具体化するプロセスと読み出すプロセスが存在することになる．

値が決まるのを待って読み出しをするソケットを入力ソケット，書き込みをするソケットを出力ソケットという．

2.3.2 メッセージ交換

aya では KL1 のすべてのデータ型を使用することができる．そして具体化されたすべてのデータをプロセス間の通信に使用することができる．この通信に使用するデータをメッセージとよぶ．メッセージ通信はプロセス間の共有変数を具体化することで行う．この変数のことを aya ではラインと呼ぶ．ラインを具体化することをメッセージの送信，具体化されることを受信という．メッセージの到着を待ち合わせはソケットを用いて行う．

'not/2' の例では, 入力用のソケット 'in' と出力用のソケット 'out' を持ち in にメッセージが到着するのを待ち合わせている. メッセージが到着すると, 到着したメッセージに対応して 0 あるいは 1 を出力用のソケット out から送信する.

第 3 章

クラス定義

この章ではプロセスのクラス定義について詳しく説明する.

```
class output(Ps)
      with +ps := Ps, -out ;
      shoen:raise(pimos_tag#shell,get_std_out,Out),
      @out := Out.

    -> @ps = [X |Ps] |
      @out <- [putt(X),nl|Out],
      @out := Out, @ps := Ps .
    -> @ps = [] | @out <- [] \\ .
end class.
```

これはプライムナンバ・ジェネレータの出力プロセスである. このプロセスは受けとったメッセージをそのまま出力する.

この例では, 予約語 'class' につづけて名前が 'output' で引数を 1 つもつのクラスを定義している. クラスは, クラス名と引数個数で識別する. クラス名と引数につづけてこのプロセスが使用するソケットの宣言と初期化処理を記述する.

初期化処理のあと移動するシーンを指定するために, 暗黙のメソッドを定義する. 暗黙のメソッドは, 必ず実行される.

この例では, 定常状態に移動することを指定している. 定常状態への移動を指定する場合はなにも記述しない.

3.1 ソケット宣言

'with' から ';' までがソケット宣言である.

```
class output(Ps)
      with +ps := Ps, -out ;
```

このプロセスが使用するソケットは, すべてここで宣言しなくてはならない.

ソケットのモード ソケットの宣言は入力のソケットか出力のソケットかを + と − の符号で示して行う. これをソケットのモードとよぶ. この例では 'ps' は入力のソケットであり, 'out' は出力のソケットである.

ソケットの初期値 ソケットに初期値を指定することもできる．初期値はソケット宣言のあとに ':=' につづけて指定する．この例では ps に対してのみ初期値の設定をしている．初期値としてプロセス起動時に引数として受けとった値を設定している．

初期値が指定されていない場合はソケットのモードに応じた既定値が設定される．既定値はつぎのとおりである．

- + の場合 ⇒ []
- − の場合 ⇒ _

初期値指定の略記法 引数で受けとった値を初期値として設定する場合の略記法として，引数位置にソケット宣言を記述することができる．この記法を使用すると 'output/1' の例はつぎのように記述できる．

```
class output(+ps)
      with -out ;
```

'with' につづくソケットの宣言を行わないときは，';' で区切ってそのまま初期化処理を記述する．

3.2 初期化処理

プロセスの初期化のためにつぎの 3 種類の処理が記述できる．

- プロセスの起動．
- ソケットの操作．
- ラインの具体化 (ユニフィケーション)

先の例では，プロセスの起動とソケットの操作をおこなっている．

```
shoen:raise(pimos_tag#shell,get_std_out,Out),
@out := Out.
```

プロセスの起動は，そのプロセスを定義しているクラス名と引数を指定して行う．起動したいプロセスが他のモジュールに定義されている場合は，クラス名の前にモジュール名を指定する．モジュール名とクラス名の間は，':' で区切る．

この例では，'shoen' というモジュールに定義されている 'raise/3' というプロセスを起動している．

ソケットの操作とはソケットの中身を入れ替えることである．ここではソケット 'out' にライン 'Out' を設定している．

初期化処理としては，メソッド中にメッセージに対応して記述される処理と同様の処理が記述できる．このそれぞれの処理についてはあとで詳しく説明する．

初期化処理を記述しないときは ';' も省略する．

3.3 定常状態への移動

プロセスは初期化処理が終わると通常のメッセージを受け付ける状態に移動する. これは初期化処理実行後に無条件で実行する暗黙のメソッドに定義している.

初期化処理のみでプロセスを終了するときは, '終了状態'(terminate) に移動する. 次のプログラムは, 'terminate' に移動するプロセスの例である.

```
class top(Max) ;
      primes(Max,Ps),
      output(Ps) \\ .
end class.
```

このプロセスはプライムナンバ・ジェネレータのトップレベルのプロセスである. このプロセスは, 初期化処理で 'prime/2', 'output/1' の 2 つのプロセスを生成し, 終了する. 暗黙のメソッドに 'terminate' へ移動することが指定してある. これが最後の '\\' である.

ここでプロセスのクラス定義およびメソッド定義の例として, プライムナンバ・ジェネレータのプログラムを示す.

```
module prime.
public top/1.

class top(Max) ;
      primes(Max,Ps),
      output(Ps) \\ .
end class.

class primes(Max,Ps) ;
      gen(2,Max,Ns),
      sift(Ns,Ps) \\ .
end class.

class gen(+no,+max,-ns).
    -> @no =< @max | @ns <- [(@no)|Ns],
      @ns := Ns,
      @no := ~((@no) + 1).
    -> @no > @max | @ns <- [] \\ .
end class.

class sift(+ns,-ps).
  input ns.
    [P|NS] -> @ps <- [P|Ps],
      filter(P,NS,Ys),
      @ns := Ys,
      @ps := Ps.
    [] -> @ps <- [] \\ .
end class.
```

```
class filter(+p,+xs,-ys).
  input xs.
    [X|Xs] -> ~(X mod @p) \= 0 |
      @xs := Xs,
      @ys <- [X|Ys],
      @ys := Ys .
    [X|Xs] -> ~(X mod @p) = 0 |
      @xs := Xs.
    [] -> @ys <- [] \\ .
end class.

class output(+ps)
      with -out ;
      shoen:raise(pimos_tag#shell,get_std_out,Out),
      @out := Out.
  input ps.
    [X|Ps] -> @out <- [putt(X),nl| Out],
      @ps := Ps,
      @out := Out .
    [] -> @out <- [] \\ .
end class.
```

クラス 'top/1' が最上位のプロセスである．このプロセスは，'primes/2' と 'output/1' の 2 つのプロセスを起動して終了する．'primes/2' も，'gen/3' と 'sift/2' の 2 つのプロセスを起動して終了する．'output/1' は生成した素数を順に受けとり，表示するプロセスである．'gen/3' はプロセス起動時に与えられた最大値までの整数を順に生成するプロセスである．'sift/3' はこの整数列から素数の倍数を取り除くフィルタを生成していくプロセスである．

第 4 章

メソッド

メッセージを受けとったときに, それに対してどのような処理をするかを定義したのがメソッドである. プロセスの動作は, このメソッド定義の集まりで記述する.

つぎの例はさきにあげたプライムナンバ・ジェネレータのフィルタのプロセスである. フィルタのプロセスは 3 つのメソッドで定義されている.

3 つのメソッドはそれぞれ, 受けとった整数がそのフィルタがフィルタする整数の倍数のとき, 倍数でないとき, メッセージ通信が終了するときに対応している.

```
class filter(+p,+xs,-ys).
  -> @xs = [X|Xs], ~(X mod @p) \= 0 |
       @xs := Xs,
       @ys <- [X|Ys],
       @ys := Ys .
  -> @xs = [X|Xs], ~(X mod @p) = 0 |
       @xs := Xs.
  -> @xs = [] | @ys <- [] \\ .
end class.
```

4.1 メソッドの形式

メソッドはつぎの 4 つの部分からなる.

- イベント.
  メソッド定義のはじめから '->' までの部分をイベントという. 'filter/3' の例では, イベントにはなにも記述していない. なにをイベントに書くかについてはあとで述べる. 例にあるようにイベントがなにも記述されていない場合は '->' だけ記述する.

- コンディション.
  '->' から '|' までがコンディションである. ここには, 到着したメッセージとのパタンマッチを記述する. 2 つ以上のパタンマッチも記述できる. '=' によるパタンマッチのほかに, 言語定義のフィルタが記述できる.

  'filter/3' の例では, ソケット xs にリストセルが到着するのをまち, さらにその car とのパタンマッチをしている.

  コンディションを記述しないときは, '|' は省略する.

- アクション.
  | あるいは ->(コンディションが記述されていないとき) のあとにメッセージに対応した処理をを記述する. これがアクションである.

  アクションとしてつぎの 3 種類が記述できる.

  1. ソケットの更新.
  2. プロセスの起動.
  3. ユニフィケーション.

  何も行わないときは 'continue' と記述する.

  この例のはじめのメソッドでは, ソケット xs, ys の値の更新と, ユニフィケーションを行っている.

- 移動先の指定.
  最後に移動先を記述する. これは, このまま定常状態で次のメッセージを待つか, それともプロセスの実行を終了するかの指定である.

  プロセスの実行を終了するときは,'\\' と記述する. '\\' は移動することを示し, そのあとに行き先の 'terminate' を省略している. そのままの状態を繰り返すときはなにも記述しない.

  'filter/3' の例ではソケット xs に [ ] が到着したとき, プロセスの実行を終了するよう指定している. それ以外のときは, そのままつぎのメッセージを待つ.

4.2 入力ソケット宣言とイベント

もう一度 'filter/3' のメソッド定義をみてみよう.

```
-> @xs = [X|Xs], ~(X mod @p) \= 0 |
     @xs := Xs,
     @ys <- [X|Ys],
     @ys := Ys .
-> @xs = [X|Xs], ~(X mod @p) = 0 |
     @xs := Xs.
-> @xs = [] | @ys <- [] \\ .
```

この例では, 'xs' というソケットに到着したメッセージとのパタンマッチによりメソッドの選択を行っている. そこで, 'xs' というソケットに着目しているということを先に宣言して, パタンマッチするパタンのみを記述することもできる.

これを用いると上のメソッド定義がつぎのように記述できる.

```
input xs.
  [X|Xs] -> ~(X mod @p) =\= 0 |
    @xs := Xs,
    @ys <- [X|Ys],
    @ys := Ys .
  [X|Xs] -> ~(X mod @p) =:= 0 |
    @xs := Xs.
  [] -> @ys <- [] \\ .
```

一行目の

```
input xs.
```

が, 着目しているソケットの宣言である. これを '入力ソケット宣言' という. このソケットに到着したメッセージとパタンマッチするパタンをイベントに記述する.

'=' 以外の言語定義プロセスはをイベントとして記述することはできない.

入力ソケット宣言は何度も行うことができる. 新しく入力ソケット宣言を行うとそれ以前に行った宣言は無効になる. 現在有効なソケット宣言を無効にするにはソケット名なしで 'input' とのみ記述する.

4.3 otherwise

'otherwise' は, それ以前に記述してあるイベントのパタンマッチとコンディションの確認がすべて失敗したことを示す略記法である.

'not/2' の例に, 0, 1 以外のメッセージがきた場合の処理, つまりエラー処理を加えると次のようになる.

```
module not.
public not/2.

class not(In,Out)
      with +in := In, -out := Out.
    -> @in = 0 | @out <- 1 \\ .
    -> @in = 1 | @out <- 0 \\ .
    -> @in \= 0, @in \= 1 | @out <- '$error$'.
           % エラー処理
end class.
```

3 番目のメソッドがエラー処理である. これを, 'otherwise' を用いてつぎのように記述することができる.

```
module not.
public not/2.

class not(In,Out)
      with +in := In, -out := Out.
    -> @in = 0 | @out <- 1 \\ .
    -> @in = 1 | @out <- 0 \\ .
  otherwise.
    ->  @out <- '$error$' \\ .
end class.
```

メソッド定義中に任意回数の 'otherwise' を記述できる.

4.4 メソッド間プライオリティ

メソッド定義の選択つまり各メソッドの選択の要件であるイベントおよびコンディションの間にプライオリティを指定することができる. これはメソッドの間に 'alternatively' と記述して指定する.

このプライオリティは, たとえばつぎのように使用する.

```
input abort.
  abort -> アボートされたときの処理.
alternatively.
input request.
  :gett(Term) -> タームの読み込み処理.
  :putt(Term) -> タームの書きだし処理.
```

このように記述すると, メッセージ 'abort' の到着をソケット 'request' にとどく他の (通常の入出力) メッセージより高い優先順位で処理することができる.

'alternatively' の前に定義したメソッドは後ろに定義したものよりプライオリティが高い. 'alternatively' の後ろに定義した 2 つのメソッドの優先度は同じである. 同様に 'alternatively' の前に複数のメソッドが定義されている場合も, それらのプライオリティは同じである.

メソッド定義のなかに複数の 'alternatively' を使用することができる.

第 5 章

ターム・ユニフィケーション

5.1 ターム

aya ではソケットの入力・出力のモードに合わせて入力のタームと出力のタームを考える．入力のタームは現在すでに値をもつもの，あるいはいずれ具体化される変数である．出力のタームはそのプロセスが具体化する変数である．

5.2 ユニフィケーション

ラインを具体化して値を持たせることをユニフィケーションとよぶ．ユニフィケーションは '<-' で記述する．左辺には出力タームを書き右辺には入力タームを書く．'filter/3' の例では，ソケット 'ys' の値とリスト $[X \mid Ys]$, [ ] とのユニフィケーションをそれぞれつぎのように記述している．

```
@ys <- [X|Ys]
@ys <- []
```

5.3 算術演算

aya では算術演算として整数演算・浮動少数点数演算を用意している．これらの演算の結果は式を

- 整数の場合

  ```
  ~( )
  ```

- 浮動少数点数の場合

  ```
  $~( )
  ```

  で囲うことにより得られる．たとえば次のように記述するとソケット 'no' の値を 1 増やすことができる．

  ```
  @no := ~((@no) + 1)
  ```

第 6 章

ソケット

ソケットはそのモードに応じて参照・読みだし・更新の操作が行える. ソケットに対するアクセスは '@' のあとにソケット名を指定して行う. これをソケット名の指定という.

6.1 参照と読みだし

ソケット名の指定は, プロセスの引数および次の節でのべるソケットの更新操作の左辺に記述できる. プロセスの引数として記述することができる. 引数として記述したソケット名の指定は, 入力のソケットの場合は値の参照, 出力ソケットの場合は読みだしになる. 読み出された後のソケットの値は '_' になる.

次のプロセスはプライムナンバ・ジェネレータの, 素数を生成するプロセスである.

```
class gen(+no,+max,-ns).
    -> @no =< @max | @ns <- [(@no)|Ns],
      @ns := Ns,
      @no := ~((@no) + 1).
    -> @no > @max | @ns <- [] \\ .
end class.
```

このプログラムに記述してあるソケットアクセスを例にみてみよう.

```
@no =< @max
```

は, プロセス '=<' の引数としてソケットが 2 つ指定されている. どちらも入力のソケットなので, それぞれのソケットの値が参照され引数としてわたされている.

```
@ns <- [(@no)|Ns],
```

ここでは, '<-' の右辺に現れているソケットは入力のソケットで, リストの car にこの値をわたしている. 左辺のソケットは出力のソケットなのでその値を読みだしてリストセルとのユニフィケーションに使用し, ソケットの値は '_' になる.

6.2 ソケットの更新

ソケットの値を更新するための操作を用意する. 更新操作にはつぎのものがある.

- ソケットの値の入れ替え.
- 算術演算を伴う入れ替え.

このそれぞれについて以下に説明する.

6.2.1 ソケットの値の入れ替え

ソケットの値の入れ替えは':='を用いて行う．':='の左辺にソケットを指定し右辺に新しい値を指定する．例えば次のようになる．

```
@xs := Xs,
@ys := Ys .
```

入力のソケットには入力のタームが，出力のソケットには出力のタームが設定できる．

値の入れ替えを行うと，それまではいっていた値は棄てられる．このとき棄てる値の後始末として次のような処理を行う．

- 入力ソケット ⇒ そのまま棄てる．
- 出力ソケット ⇒ 現在の値を[ ]に具体化して棄てる．

6.2.2 算術演算を伴う入れ替え

ソケットに入っている値を用いて整数演算・浮動少数点数演算をおこない，その結果をまたソケットに入れることが記述できる．

たとえば次のようになる．

```
@no += 1
```

これは，ソケットの値に1を加えそれを新しい値とすること，つまりつぎのように記述するのと同じである．

```
@no := ~((@no) + 1).
```

一般に，左辺に指定したソケットと右辺に指定した値の演算を行いその結果を左辺のソケットの新しい値とする操作を，次の演算子を用いて記述できる．

- 整数演算．

```
+=, -=, *=, /=
```

- 浮動少数点数演算．

```
$+=, $-=, $*=, $/=
```

6.3 1つのソケットが何度もアクセスされるとき

あるソケットが1つのメソッド中になんども現れ，値が更新されていくとき，その値はメソッド定義中で左から右に順に新しくなる．ただし，':='の両辺に同じソケットが現れたときは右辺のソケットのほうが古い値を持つ．

'gen/3'のプロセスのはじめのメソッドについてみる．

```
-> @no =< @max | @ns <- [(@no)|Ns],
   @ns := Ns,
   @no := ~((@no) + 1).
```

ソケット 'no' は入力のソケットなのでコンディション =< の左辺の 'no' と, アクション <- の右辺にリストの car として現れた 'no' は同じ値を持つ. 最後の更新処理では, 右辺の no はそれまでと同じ値をもち, 左辺は右辺の演算結果を値とする.

ソケット 'ns' は出力のソケットである. <- の左辺にあらわれてユニフィケーションがおこなわれるとその値は '_' になる. つぎに ':—' の左辺にあらわれたときの値は '_' である. ここに新しく 'Ns' が設定される.

6.4 ソケットの後始末

プロセスがシーンを移動して, それまで使用していたソケットが不要になることがある. このとき出力ソケットの中にラインが残したままソケットを捨ててしまうと、 ラインの他端を持つプロセスがデッドロックする場合がある.

そこで, ソケットが最後に保持していたデータに対してつぎのような後始末を暗黙のうちに行うようにしている.

- 入力ソケットの場合 ⇒ そのまま捨てる.
- 出力ソケットの場合 ⇒ [ ] に具体化する.

第 7 章

ストリーム

7.1 ストリーム

プロセス間で続けてメッセージ通信を行いたいときはメッセージをリストセルにつめて，つぎに使用するラインと一緒に送る方法を使用することができる．このような通信を，ストリーム通信と呼び，リストセルをストリーム型メッセージという．

'filter/3' もストリーム通信を行うプロセスの例である．

```
class filter(+p,+xs,-ys).
  input xs.
    [X|Xs] -> ~(X mod @p) \= 0 |
      @xs := Xs,
      @ys <- [X|Ys],
      @sys := Ys .
    [X|Xs] -> ~(X mod @p) = 0 |
      @xs := Xs.
    [] ->  @ys <- [] \\ .
end class.
```

メッセージ $[X \mid Xs]$ は，その car, X とパタンマッチをして，メソッドを選択する．cdr の Xs は未使用のラインでソケット Xs の新しい値とし，つぎのメッセージを受けとるのに使用する．

メッセージ [ ] はストリーム通信を終了することを意味している．これを，クローズメッセージとよぶ．

7.2 ストリームのための記法

car のあたまに ':' をつけると，ストリーム型メッセージを表す．クローズメッセージは特別に ':/' と記述する．またストリーム通信のために cdr を新しいラインとすることも同時に記述できるメッセージ送信の記法も用意している．ここで用意する記法はコンパイル時に展開される．この展開を抑制するためには '`'(バックコート) を '`:/' のように用いる．

この記法を用いると 'filter/3' のプロセスはつぎのように記述できる．

```
class filter(+p,+xs,-ys).
  input xs.
    :X -> ~(X mod @p) = 0 | @ys <<= :X.
    :X -> ~(X mod @p) \= 0 | continue.
```

```
    :/ ->  @ys <- :/ \\ .
end class.
```

まず, 1 つめのメソッドをみてみる. イベントの ':X' は, ストリーム型メッセージの X とのパタンマッチを指定している. またこのように記述すると, このリストセルの cdr はソケット xs の新しい値として設定される. アクションの

```
    @ys <<= :X.
```

はストリーム型メッセージ ':X' を送り, その cdr をソケット ys の新しい値として設定することである.

最後のメソッドはクローズメッセージが到着した場合である. クローズメッセージの場合は新しいラインの設定は行わないので通常のソケット参照となる.

出力ソケットに持っていたストリームにもクローズメッセージを送り, 通信を終了することはつぎのように記述する.

```
    @ys <- :/
```

この場合も新しいラインの設定はおこらないので通常のユニフィケーションとなり, ソケットの値は '_' になる.

7.3 ストリームの操作

ストリームそのものの操作も指定することができる. ストリームそのものの操作として,

- マージ
- 2 本のストリームをつなぐ

の 2 種類の操作を記述できる.

7.3.1 マージ

複数本のストリームをマージして 1 本にするプロセスを用意している.

```
    merge(In,Out)
```

'In' が入力ストリーム, 'Out' が出力ストリームである. 入力ストリームを要素とするベクタを 'In' に渡すと, その要素が入力ストリームとして加えられる. このベクタと入力ストリームのユニフィケーションを行うことを ' マージイン ' と呼ぶ.

7.3.2 つなぐ

2 本のストリームをつなぐことができる.

前に説明したメッセージの送信で, 送信メッセージがストリームとして使用される変数の場合を考える.

```
    @ys <<= Mesgs
```

ここでは, ライン Mesgs を具体化したストリーム型メッセージの列をソケット ys の保持するストリームに送っている. これは, メッセージのつまったストリームを ys に入っているストリームのあたまにさしこむこととともみなせる. つまり, 2 本のストリームをつなぐことである. 順番を逆にしてソケットにはいっているストリームの後ろにつなぐことを

```
@ys <=< Mesgs
```

と記述する.

この操作をつかって, 自分自身にメッセージを送ることもできる. たとえば, コマンドの alias 処理を行うプロセスはつぎのように書ける.

```
class alias(+in,-out).
  input in.
    :copy -> @in <=< :cp.
    :cp -> 処理内容.
end class.
```

このプロセスでは, メッセージ 'copy' を受けとると自分に対して 'cp' を送信する. そして, このメッセージ 'cp' を 'copy' に続くメッセージとして受信し処理する.

第 8 章

状態の移動

セマフォ (semaphore) のような問題をプロセスとして記述する場合を考える.

セマフォとは, セマフォに対応づけられた資源を要求する操作と, この資源を解放する操作の2つの操作を持ったオブジェクトである. セマフォの資源を要求する操作をP オペレーション, 解放する操作をV オペレーションと呼ぶ. 資源要求をだされたセマフォで資源がすでに使用中ならばその要求をだしたプロセスは資源が解放されるまで待たされる. V オペレーションが実行されると, 待っていたプロセスがアクセスすることができる.

セマフォのプロセスには, 資源が使用中 (ビジー) の場合と解放されている場合 (アイドル) がある. そして, P オペレーションの要求がきたとき, プロセスがアイドルならばその要求を実行できるがビジーの場合は, 資源が解放されるまで P オペレーションの実行は待たされる.

このようなプロセスを書く方法の1つは, ビジーかどうかを示すフラグを用意し, 毎回それを確認するというものである. しかし aya ではこれを, プロセスにはビジーとアイドルの2つの状態があるというかたちで記述できるようにしている. そしてこれをそれぞれ' シーン' とよぶ.

クラスの初期化処理のあと移動する定常状態はすべてのクラスに存在する名前のないシーンであり, 'terminate' もまた言語定義のシーンである.

8.1 シーン

シーンを用いて記述したセマフォのプログラムを示す.

```
module semaphore.
public semaphore/1 .

class semaphore(+in)
      with +value := 0, +queueh := Queue, -queuet := Queue
    \\ idling.

scene idling.
  input in.
      :p(Ack) -> @value -= 1,
          Ack <- ok, do_operation
          \\ wait_v_operation.
end scene.    % idling

scene wait_v_operation.
  input in.
```

```
    :v -> @queueh = :p(Ack) |
        @in <=< :p(Ack),
        Ack <- ok,
        @value += 1
          \\ idling.
    :v -> @value = 0 | continue \\ idling.
    :p(Ack) -> @value -= 1, @queuet <<= :p(Ack).
end scene.    % wait_v_operation
end class.
```

プロセス 'semaphore' はメッセージの到着を待つ状態, V オペレーションの要求メッセージの到着を待つ状態がある. この 2 つの状態を 'idling', 'wait_v_operation' の 2 つのシーンとして定義している. idling ではメッセージの到着を待ち, P オペレーションの要求がきたら資源が使用中であることを示すカウンタを 1 減らし, 資源に対する処理をはじめ V オペレーションを待つ状態 'wait_v_operation' にうつる. ここでは, V オペレーションがきたら待っていた P オペレーションを実行する. P オペレーションがきたときは待ち行列に加え, カウンタを 1 へらす. 待っている P オペレーションがないときは idling にもどる.

このようにユーザはクラスが既定値として持つシーンにネストしたかたちで, 任意個数のシーンを定義することができる. ユーザが定義したシーンにさらにネストしてシーンを定義することもできる. 任意段数のネストができる.

8.2 シーンの定義

シーンは予約語 'scene' に続けて定義する. シーン定義の終わりは 'end scene' である. シーンはシーン名で識別する. シーンの定義にはクラスの場合と同様に, ソケット宣言, 初期化処理を記述する. そのあとにシーンで定義する状態をあらわすメソッドを定義する.

ソケット宣言 ソケットはクラスのときと同様に入出力のモードとソケット名の組で宣言する. 初期値を指定することもできる.

ソケット宣言は, そのネストの外側でなされたものを継承する. つまり, クラスで宣言されたソケットはそのなかに定義されているすべてのシーンからアクセスできる.

この例でも, シーン idling, wait_v_operation はクラスで定義しているソケット 'in' にアクセスしている.

初期化処理 クラスのときと同様にコンフィケーション, プロセスの起動, ソケットの更新が記述できる.

移動先の指定 初期化処理終了後の移動先を指定する. なにも指定しなければそのシーンで定義されたメソッドを実行する定常状態に移る. プロセスの実行を終了する場合は 'terminate' に移動する. そのほかのユーザ定義のシーンに移動するときはその名前と引数を指定する. 詳細は次節で説明する.

メソッド定義 クラス定義のメソッド定義と全く同様に定義できる.

8.3 シーンの移動

ユーザは定義したシーンの間の移動を明示的に記述する必要が有る.

シーンの移動は, クラス・シーンの初期化処理のあと, およびメソッド定義のなかで指定できる. 指定は '\\' のあとにシーン名と引数を指定して行う. 終了のとき指定する 'terminate' は, シーン名を省略して指定できる.

セマフォの例で見てみる.

```
class semaphore(+in)
      with +value := 0, +queueh := Queue, -queuet := Queue
    \\ idling.
```

セマフォのプロセスはその起動時の初期化処理がおわるとシーン 'idling' に移動する.

```
scene idling.
  input in.
    :p(Ack) -> @value -= 1,
      Ack <- ok,
      do_operation
      \\ wait_v_operation.
end scene.
```

'idling' で p オペレーションの要求を受けとると, シーン 'p_operation/2' に移動する. 'p_operation/2' は引数を 2 つ持つ.

このように, \\ のあとにシーン名と引数を指定して移動する.

8.4 ネストしたシーン

ユーザ定義のシーンがネストした例としてリーダーズ・ライターズ問題を記述したプログラムをみてみる.

この問題は共有資源アクセスの管理を扱ったもので, 次のようなアルゴリズムに基づいている。

- 読みだし要求は, 同時にいくつでも扱える.
- 書き込み操作が行われている間は, 他の要求は一切受け付けない.
- 読みだし操作の間に書き込み要求がくると, その後あらたな読みだし要求は受け付けない. 現在実行中の読みだし操作がすべて終了するのを待って書き込み操作を行う.

このプログラムは, ファイルへの読みだし・書き込み要求を管理するプロセスである. プロセスは, readerswriters という名前のクラスとして定義されている. このプロセスは起動時に, ファイルへの要求がくるストリーム request, このプロセスからファイルデバイスへ要求を送るストリーム tofile, ファイルから管理プロセスへ処理の終了などを伝えるメッセージの流れるストリーム fromfile の 3 本を受けとる.

クラス readerswriters は, idling, reading, writing の 3 つのシーンを持つ. それぞれ, メッセージの到着待ち, 読み込み処理, 書き込み処理を行うシーンである. プロセスを起動するとまず idling のシーンでメッセージの到着を待つ. 読み込み要求ならば reading に, 書き込み要求ならば writing に移る. reading のシーンで読み込み要求を受けとるとそれはただちにファイルデバイス

に送られる. 書き込み要求の場合は reading の中で定義しているシーン waiting に移動し, そのとき実行されている読みだし処理がすべて終了するのを待つ. その後 writing に移って書き込み要求をファイルデバイスに送る. 読みだし操作, 書き込み操作が終了すると, idling に移動して次の要求を待つ.

```
module readerswriters.
public readerswriters/3.

class readerswriters(+request,-tofile,+fromfile)
      \\ idling.

scene idling.
  input request.
    :read(Data) -> @tofile <<= :read(Data) \\ reading.
    :write(Data) -> @tofile <<= :write(Data) \\ writing.
end scene. % idling

scene reading
      with +readers := 1.
    -> @readers = 0 | continue \\ idling.
  input request.
    :read(Data) -> @readers := ~(@readers + 1),
        @tofile <<= :read(Data).
    :write(Data) -> continue \\ waiting(Data).
  input fromfile.
    :readend -> @readers := ~(@readers - 1).

scene waiting(+writedata).
      -> @readers = 0 |
        @tofile <<= :write(@writedata) \\ writing.
    input fromfile.
      :readend -> @readers := ~(@readers - 1).
  end scene. % waiting

end scene. % reading

scene writing.
  input fromfile.
    :writeend -> continue \\ idling.
end scene. %writing

end class. % readerswriters
```

この例で readerswriters というクラスは, idling, reading, writing の 3 つのシーンで定義され, さらに reading は waiting というシーンを内部に持っている.

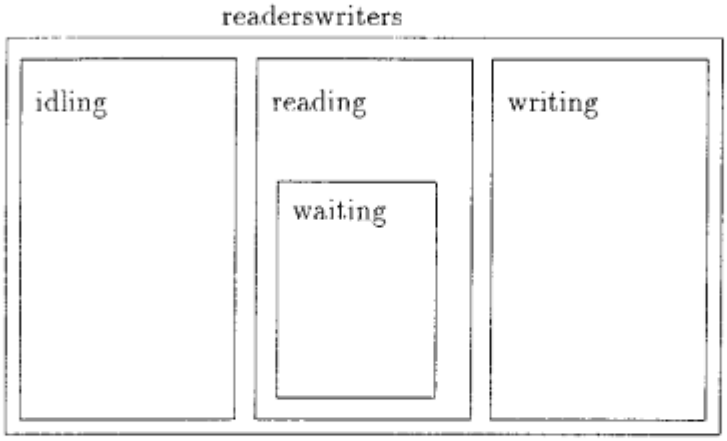

図 8.1: シーンの関係

シーン waiting では, クラスと reading で宣言したソケットにもアクセスできる. シーン waiting を移動先として指定できるのは reading 中のメソッドだけである. waiting 中のメソッドからは,reading, idling, writing に移動することができる.

いいかえると, シーン waiting から移動できるのは, waiting がネストしているシーン 'reading' と, ネストしている reading と同じシーン (クラスが必ず持つシーン) におなじレベルでネストしている 'idling', 'writing' である.

このほか, reading に 'waiting' と同じレベルでネストしたシーンが他にもある場合は, そこに移動することもできる. また, waiting にネストしたシーンがある場合はそこにも移動できる.

第 9 章

プラグマ

プログラムを効率的に実行するために, ユーザはプログラム中に次のことを指定できる.

- レジスタ宣言.
- プライオリティ指定.
- ノード指定.

プライオリティ指定, ノード指定は KL1 プログラムにおけるプライオリティ指定, ノード指定と同じである. レジスタ宣言は KL1 プログラムへのコンパイル時に処理される.

9.1 レジスタ宣言

ソケットに対してレジスタ宣言を行うと, aya のコンパイラはできるかぎり, KL1 プログラムのアクセスの早い引数に展開する

レジスタ宣言はソケットを宣言するときに同時に指定する.

たとえばつぎのようになる.

```
class not(In,Out)
with +in @ register := In, -out := Out.
```

引数位置に記述したソケット宣言にも記述できる.

```
class not(+in@register,-out).
```

9.2 プライオリティ指定

プロセス実行の優先度を制御することができる. 優先度の制御は, プロセス起動時に優先度を指定して行う.

なにも指定しないときはそのプロセスを起動したプロセスと同じ優先度が与えられる. 言語定義のプロセスに優先度を指定することはできない.

優先度の指定の方法には割合指定と相対指定がある. このそれぞれについてつぎに簡単に説明する. プライオリティ指定の詳細は [1] を参照のこと.

9.2.1 割合指定

プロセスの取り得るプライオリティには, 上限と下限がある. この上限と下限に対する割合で指定する方法である. 指定はプロセス名, 引数のあとに次のように指定して行う.

```
@priority(割合)
```

割合としては 0 から 4096 の間の整数を指定する. 実際にはこの指定された値を 4096 で割った商が割合となる.

プライムナンバ・ジェネレータの例で結果を出力するプロセスの優先度を他かより下げるとすると例えば次のように指定する.

```
class top(Max) ;
      primes(Max,Ps),
      output(Ps)@priority(3000) \\ .
end class.
```

9.2.2 相対指定

プロセスは優先度の既定値として自分を起動したプロセスの優先度を持っている. これに対する割合で指定する.

```
@priority(relative, 割合)
```

割合の指定のしかたは, '割合指定' の場合と同じである.

割合指定の場合と同様につぎのように使用する.

```
class top(Max) ;
      primes(Max,Ps),
      output(Ps)@priority(relative,3000) \\ .
end class.
```

9.3 ノード指定

aya によるプログラムを実際のプロセッサで並列に実行するためには, どのプロセスをどのノードで実行するかをプログラム中に指定する必要がある. この指定はプロセスを起動するときと, シーンを移動するときに指定できる. ノードの指定はプロセス名あるいはシーン名, 引数のあとにつぎのように記述して行う.

```
@node(ノード番号)
```

指定の方法は優先度指定と同じである. 詳細は [1] を参照のこと.

第 10 章

ソケットにまつわる略記法

ソケットに対する操作には略記法が用意してあるものが多い．それを紹介する．ここで紹介する記法はストリームにまつわるものと同様に，コンパイル時に展開される．この展開を抑制するためには'\'(バックコート) を使用する必要がある．

10.1 ソケットの参照と更新

ソケットの値の参照あるいは読みだしはソケット名の前に'@'をつけて示す．つぎのように記述すると参照あるいは読みだしと一緒に新しい値を指定することができる．

```
filter(P,@ns!Ys,Ys)
```

これは，第 2 引数にソケット'ns'の値をわたし，そのあとで ns に新しい値として Ys を設定することを指定している．これは，つぎのように記述するのと同じである．

```
filter(P,@ns,Ys),
@ns := Ys,
```

一般に

@ ソケット名 ! 新しい値

と記述することができる．

10.2 ストリームとソケット

10.2.1 cdr を表す記法

つぎのように記述すると，このソケットに入っているラインにこのメッセージを送信した cdr をあらわす．

```
@socket:msg1:msg2
```

このあとに新しい値の指定をすることもできる．

```
@socket:msg1:msg2 ! New
```

この記法を用いてつぎのようなプロセスが記述できる．これは，'child'というプロセスを次々とフォークするプロセスである．

```
class fork(+in,-out).
  input in.
    :/ -> @out <- :/ \\ .
    :fork(In) -> child(In,@out:fork!Out,Out) .
end class.
```

これは, つぎのように記述するのとおなじである.

```
class fork(+in,-out).
  input in.
    :/ -> @out <- :/ \\ .
    :fork(In) -> child(In,Out,Tail),
       @out <<= :fork,
       Out <- @out,
       @out := Tail.
end class.
```

ソケット 'out' からメッセージを送り, そのストリームのテイルを 'child' に渡す. 'child' の第3引数に戻ってきたストリームをソケットの新しい値として設定する.

10.2.2 マージイン

マージインする入力ストリームがソケットにはいっていて, ユニファイするベクタが2要素の場合, 次のように記述することができる.

```
@in <= In2
```

これはつぎのように書くのと同じである.

```
@in ! In1 <- {In1,In2}
```

つぎのように記述してマージインしたストリームを引数としてわたすこともできる.

```
process(@in <=, X,Y)
```

これはつぎのように記述するのと同じである.

```
process(In2,X,Y),
@in <= In2
```

第 11 章

構造体に対するアクセス

ベクタやストリングの要素の読みだしや書き込みを次のように書ける.

11.1 要素の参照

ベクタやストリングの要素の参照を, コンディションおよびアクションにつぎのように記述することができる.

- ベクタの場合.

  ```
  @struct @ 1
  Struct @ 1
  ```

  これらは, ベクタの第一要素に対する参照である. ベクタはソケットに入っていてもそうでなくてもよい.

- ストリングの場合.

  ```
  @struct @@ 1
  Struct @@ 1
  ```

  同様にストリングの第一要素に対する参照である. ストリングはソケットに入っていてもそうでなくてもよい.

ベクタの要素に対するアクセスは '@' のあとに位置を指定しておこなう. 同様にストリングのときは '@@' のあとに要素の位置を指定する.

アクセスしたベクタの要素がさらにベクタあるいはストリングの場合は, さらに位置の指定を行ってその要素を読みだすことができる. たとえばつぎのように記述する.

```
@struct @ 1  @ 2
Struct @ 1  @ 2
```

11.2 要素の更新

ソケットに入っている構造体に対する要素の更新は同様に位置を指定して行える.

```
@struct @ 1 ! Elment
@struct @@ 1 ! Element
```

新しく設定する要素は, '!' につづけて指定する. 更新処理を行うと, ソケットに入っている構造体も更新後の新しいものになる.

更新の記述は, ネストしたベクタあるいはストリングに対しても同様に行える.

この参照・更新の記法を使ってベクタのプロセスを記述するとたとえばつぎのようになる.

```
class vector(+in)
      with +vector := {a, {b1,b2}, c}.
  input in.
    :/ -> continue \\ .
    :vector_element(Pos,Elm) -> Elm <- @vector @Pos .
    :set_vector_element(Pos,NewElm,Elm) -> Elm <- @vector @Pos ! NewElm .
end class.
```

第 12 章

組み込みプロセス

コンディション, アクションの記述のために, aya の組み込みプロセスを用意する.

12.1 コンディション用プロセス

KL1 のガード組み込み述語はすべて記述できる. KL1 の組み込み述語については [2] を参照のこと.

12.2 アクション用プロセス

KL1 のボディ組み込み述語はすべて記述できる. KL1 の組み込み述語については [2] を参照のこと.

付録 A

文法

aya の文法を以下に示す.

A.1 タームの定義

```
<term>                  ::= <plus term> | <minus term>
<condition term>        ::= <condition plus term> | <condition minus term>

<plus term>             ::= <plus socket term> | <arithmetical macro expression> |
                            <plus data term> | <plus structure access>
<minus term>            ::= <minus socket term> | <minus data term> |
                            <minus structure access>

<condition plus term>     ::= <condition plus socket term> |
                          <arithmetical macro expression> |
                          <condition plus data term> |
                          <condition plus structure access>
<condition minus term>    ::= <condition minus socket term> |
                          <condition minus data term> |
                          <condition minus strucure access>

<plus data term>        ::= <instantiated term> | <variable name> |
                            <message send> | <stream close>
<plus socket term>      ::= <plus socket designation> [''!''<plus term>]

<condition plus socket term> ::= <plus socket designation>

<condition plus data term>    ::= <instantiated term> | <variable name> |
                          <condition message send> | <stream close>
```

```
<arithmetical macro expression> ::= <integer arithmetical macro expression> |
                <floating point arithmetical macro expression>

<integer arithmetical macro expression> ::= ''~(''<integer expression>'')''

<floating point arithmetical macro expression> ::=
                ''$~(''<floating point expression>'')''

<integer expression> ::= <integer> | <variable name> |
                         <plus socket designation> [''!'' <plus term>] |
                         <integer term> <op> <integer term> |
                          ''\(''<integer term>'')'' |
                         ''int(''<floating point term>'')''

<integer term> ::= <integer> | <variable name> |
                         <plus socket designation> [''!'' <plus term>]
                         | <integer arithmetical macro expression>

<floating point expression> ::= <floating point> | <variable name> |
                         <plus socket designation> [''!'' <plus term>] |
                <floating point term> <op> <floating point term> |
                ''float(''<integer term>'')''
<floating point term> ::= <floating point> | <variable name>
                         | <plus socket designation> [''!'' <plus term>]
                         | <floating point arithmetical macro expression>
<op> ::= ''+'' | ''-'' | ''*'' | ''/''

<instantiated term>  ::= <integer> | <floating point> | <atom> |
                          <vector> | <list> | <string> | <module>

<minus data term>    ::= <variable name>{<message send>}
<minus socket term>  ::= <lhs socket term>[''!''<minus term>] |
                                 <minus socket designation> ''<=''
<lhs socket term>    ::= <minus socket designation>{<message send>}

<condition minus data term>   ::= <variable name>{<condition message send>}
<condition minus socket term> ::=
                         <minus socket designation>{<condition message send>}

<plus socket designation>  ::= ''@''<plus socket name>
<minus socket designation> ::= ''@''<minus socket name>
<plus socket name>         ::= <atom>
<minus socket name>        ::= <atom>
<message send>             ::= '':''<term>
<stream close>             ::= '':/''
```

```
<condition message send>   ::= ''':''<condition term>

<plus structure access>    ::= <plus term>(<vector pos> | <string pos>)
                                ['''!''<term>]
<minus structure access>   ::= <plus term> <vector pos> ['''!''<term>]
<condition plus structure access>  ::=
                <condition plus term>(<vector pos> | <string pos>)
<condition minus structure access> ::=
                <condition plus term><vector pos>

<vector pos>          ::= ''@''<position>
<string pos>          ::= ''@@''<position>
<position>            ::= <integer expression>
```

- タームにはイベントおよびコンディションに出現するものと，アクション・移動先の指定に出現するものがある．前者を < *condition term* > とよび後者を < *term* > と呼ぶ．< *condition term* > ではソケットの更新を行うことはできない．
- ターム (< *condition term* >, < *term* >) には入力と出力がある．入力および出力のタームはそれぞれ，ソケットへのアクセスを行うターム・データにアクセスするタームからなる．また，入力のタームについては算術演算マクロ・構造体アクセスがある．
- < *lhs socket term* > は，ソケットの更新の左辺に出現できるソケットタームである．
- 構造体アクセスには，その要素の入力・出力によって < *plus structure access* > および < *minus structure access* > がある．
- ストリーム型のメッセージは ":" をつけて示す．close message ([ ]) は ":/" である．

A.2 アクションの定義

```
<action> ::= ''continue'' | <instantiation> | <socket update> |
                <process initiation>[<pragma>]

<instantiation> ::= <minus term> ''<-'' <plus term>

<socket update> ::= <plus socket update> | <minus socket update>

<plus socket update>  ::= <plus socket designation> <update op> <plus term> |
                <plus socket designation> <stream op>
                        (<variable name> | <plus socket term> |
                                <message send>) |
                <plus socket designation> <arithmetical op> <plus expression>
<minus socket update> ::= <lhs socket term> <update op> <minus term> |
                <lhs socket term> <stream op>
                        (<minus term> | <message send> ) |
                <minus socket designation> <arithmetical op> <plus expression>
```

```
<process initiation> ::= <process name>[''('' <actual>{'',''<actual>}'')'']
<process name> ::= <atom> | <vector>
<actual> ::= <term>

<update op> ::= '':=''

<stream op> ::= ''<='' | ''<<='' | ''<=<''

<arithmetical op> ::= ''+='' | ''-='' | ''*='' | ''/='' |
                      ''$+='' | ''$-='' | ''$*='' | ''$/=''
<plus expression> ::= <integer expression> | <floating point expression>
```

- アクションとして記述できるのは，ラインあるいはソケットの具体化，ソケットの更新，プロセスの生成である．また，アクションとしてなにも行わない場合は "continue" を記述する．プロセスの生成には，そのプロセスを実行するノード・プライオリティを指定することができる．

- ソケットの更新処理 (< *update op* >, < *stream op* >) を記述するときは，左辺に！< *next line* > を指定することはできない．

- < *stream op* > による更新処理はそれぞれストリームの merge in(<=), prepend(<<=), append(<=<) にあたる．右辺に記述できるのは，ストリーム型メッセージ，ストリーム型メッセージに具体化されうるソケットおよび変数である．

A.3 コンディション

```
<condition> ::= <language defined condition> | <unification>

<unification> ::= <condition plus term> ''='' <condition plus term>
```

- < *language defined condition* > として，算術比較・タイプチェックなど KL1 のガード組み込み述語に相当するものが記述できる．

- < *unification* > は具体化を伴わない．入力のターム同士の同一性のチェックのみである．

A.4 イベント

```
<event>              ::= <event line exp> | <event stream exp>
<event line exp>     ::= <condition plus term>
<event stream exp>   ::= {<message arrival>}
                         (<message arrival> | <stream close>)
<message arrival>    ::= '':''<condition term>
```

- イベントとして記述できるのは，"input ソケット名." で宣言した入力ソケット (後出) とのユニフィケーションである．ユニフィケーションの対象がストリーム型のメッセージの場合とそれ以外がある．

- ユニフィケーションの対象がストリーム型のメッセージの場合, 複数のメッセージ列とのユニフィケーションも記述できる.

A.5 シーン

```
<scene definition> ::=
        ''scene'' <scene name>[''(''<formal>{'',''<formal>}'')'']
                ['' with'' <socket declaration>[''=''<initial value>]
                {'',''<socket declaration>}[''=''<initial value>]][''.''
                <initiation>] [''\\ '' [<next scene>[<pragma>]]]''.''
        {(<method list item>''.'' | <scene definition>)}
''end scene.''

<scene name> ::= <atom>
<formal> ::= <variable name> | <socket declaration>
<socket declaration>       ::= <plus socket declaration> [''@register''] |
        <minus socket declaration> [''@register'']
<plus socket declaration>  ::= ''+'' <socket name>
<minus socket declaration> ::= ''-'' <socketname>
<mode>           ::= ''+'' | ''-''
<socket name>    ::= <atom>
<initiation>     ::= <action>
<next scene>     ::= <scene name>[''(''<actual>{'',''<actual>}'')''] |
        ''(''<method list item> {'';''<method list item>}'')''
<initial value> ::= <plus term> | <minus term>

<method list item>  ::= <input socket declaration> | <method definition> |
                ''alternatively'' | ''otherwise''
<input socket declaration> ::= ''input''[<input socket name>]
<method definition> ::= [<event>] ''->'' [<condition>{'',''<condition>}''|'']
                <action>{'',''<action>}[''\\'' [<next scene>[<pragma>]]]

<pragma>  ::= ''@node(''<node no>'')'' | ''@priority(''<ratio>'')'' |
                ''@priority(relative,''<ratio>'')''
<node no> ::= <integer expression>
<ratio>   ::= <integer expression>
```

- <socket declaration> と一緒に <initial value> を指定する場合はそれぞれつぎのものを指定しなければならない.
  - <plus socket declaration> については <plus term>.
  - <minus socket declaration> については <minus term>.

  また, <initial value> の指定がない場合は既定値として,
  - <plus socket declaration> には [ ]

– <minus socket declaration> には '_'

が設定される.

- ソケットの宣言と同時に "register 宣言" を行うことができる. この宣言はクラスやシーンの引数として宣言されるソケット・with xxx のかたちで宣言されるソケットのいずれに対しても行える. "register 宣言" されたソケットは KL1 プログラムに変換するさいにできるだけ述語の引数として展開する.

- < *pragma* > としてプライオリティを指定する場合, "@*priority*("< *ratio* > ")" および "@*priority*(*relative*,"< *ratio* > ")" はそれぞれ KL1 の priority(*, 割合), priority($, 割合) に対応する.

- シーンの初期化処理につづいて次のシーンを指定し, かつこのシーンのメソッドを定義することは文法上はできるが, 実行されることの有り得ないメソッドを定義することにあたるので KL1 へのコンパイル時にワーニングとする.

- ''\\'' [<next scene>[<pragma>]]

  の指定をおこなわないときは, 現在のシーンを繰り返すものとみなす. ''\\'' のみ記述するのは

  ```
  ''\\ terminate''
  ```

  の意味である. ここで "terminate" はシステム提供のシーンである. "terminate" と一緒に指定したプラグラマは読み飛ばす.

A.6 クラス

```
<class definition> ::=
        ''class'' <class name>[''(''<formal>{'',''<formal>}'')'']
                ['' with''<socket declaration>['':=''<initial value>]
                {'',''<socket declaration>[''':=''' <initial value>]}[''';''
                <initiation>] [''\\'' [<next scene>[<pragma>]]]''.''
        {(<method list item>''.'' | <scene definition>)}
        ''end class.''

<class name> ::= <atom>
```

A.7 演算子順位

表 A.1: 演算子順位

優先度	種類	演算子
1200	fx	class, scene, public
1180	xfy	;
1150	xfx,fx	->
1130	xfx	|
1100	yfx	\\
	yf	\\
1050	xfy	with
1000	xfy	,
700	xfx	=, \=, <, >, =<, >=, :=, $:=, $<, $>, $=<, $>=, <-, <<=, <=<, +=, -=, *=, /=, $+=, $-=, $*=, $/=
700	xfx,yf	<=
500	yfx,fx	+, -
	yfx	/\, \/, xor
400	yfx	*, /, <<, >>
300	xfx	mod
290	xfy	@, @@
280	xfy	!
280	fy	!
250	xfy	:
	fy	:
	xf	:/
240	fy	@
150	xf	++, --
100	xfx,fx	#
90	xfx	::
80	fx	module, end, input

付録 B

プログラム例

aya によるプログラムの例として,

- カウンタ
- プライムナンバ・ジェネレータ
- n クイーン問題
- 哲学者の食事問題

を次章以降にしめす. プライムナンバ・ジェネレータは本文中に示したものを, ストリームのための記法を用いて書き直したものである.

B.1 カウンタ

カウンタのプロセスである. このプロセスは, clear, up, down, show(State), の 4 種類のメッセージを受けとる. clear を受けとると, カウンタの値を 0 にする. up, down を受けとるとカウンタの値をそれぞれ 1 増減する. show(State) のときはそのときのカウンタの値を引数の State に返す.

```
module counter .     % counter
public counter/2 .

class counter(+in,-out)
      with +count := 0.
  input in .
    :clear ->@count := 0 .
    :up -> @count := ~(@count + 1) .
    :down -> @count := ~(@count - 1) .
    :show(State) ->  State <- @count .
    :/ -> @out <- @count \\ .
end class.
```

B.2 プライムナンバ・ジェネレータ

素数生成プログラムである. ここでは, ストリームメッセージの記法を用いて書き直した.

```
% prime number generator

module prime.
public top/1.

class top(Max) ;
      primes(Max,Ps),
      output(Ps) \\ .
end class.

class primes(Max,Ps) ;
      gen(2,Max,Ns),
      sift(Ns,Ps) \\ .
end class.

class gen(+no,+max,-ns).
    -> @no =< @max | @ns <<= :(@no),
      @no := ~((@no) + 1).
    -> @no > @max | @ns <- :/ \\ .
end class.

class sift(+ns,-ps).
  input ns.
    :P -> @ps <<= :P,
      filter(P,@ns!Ys,Ys).
    :/ -> @ps <- :/ \\ .
end class.

class filter(+p,+xs,-ys).
  input xs.
    :X -> ~(X mod @p) \= 0 |
      @ys <<= :X .
    :X -> ~(X mod @p) = 0 |
      continue.
    :/ -> @ys <- :/ \\ .
end class.

class output(+ps)
      with -out ;
      shoen:raise(pimos_tag#shell,get_std_out,Out),
      @out := Out.
  input ps.
    :X -> @out <<= :putt(X):nl .
    :/ -> @out <- :/ \\ .
end class.
```

B.3 n クイーン問題

$n \times n$ のチェスの盤面にクイーンを互いにとられないように置く全ての答えを求める全解探索問題である．なおクイーンは盤面上を縦横斜めに動くことができる．

```
module queens.
public top/2.

class top(+n,-r) ;
      merge(R0,@r),
      @r := R0.
    -> @n > 0 | queens(@n,@n,[],@r) \\ .
end class.

class queens(+n,+i,+b,-r).
    -> @i > 0 | continue  \\ next_queen(@n).
    -> @i = 0 | @r <- [@b] \\ .

scene next_queen(+j).
    -> @j > 0 | @r <= R1,
      try(@n,@i,@j,@b,R1),
      @j -= 1.
    -> @j = 0 | @r <- :/ \\ .
end scene.
end class.

class try(+n,+i,+j,+b,-r) ;
      check(@b,@j,1,Res)
      \\ if_succeeded(Res).

scene if_succeeded(+res).
  input res .
    yes -> queens(@n,~((@i) -1),[(@j)|(@b)],@r) \\ .
    no -> @r <- :/ \\ .
end scene.
end class.

class check(+b,+j,+d,-res).
  input b.
    :K -> K = @j | @res <- no \\ .
    :K -> @j = ~(K+(@d)) | @res <- no \\ .
    :K -> @j = ~(K-(@d)) | @res <- no \\ .
  otherwise.
    :K -> @d += 1  .
    :/ -> @res <- yes \\ .
end class.
```

'top/2' がトップレベルのプロセスである．クイーンの数と結果を集めるストリームをわたして起動する．マージャを起動し，与えられたクイーンの数が 1 以上ならば n 行目にクイーンを置く探索を行うトップレベルのプロセス 'queen/4' を起動する．

'queen/4' では，ソケット i に行番号を持ち，その行にクイーンを置く探索をする．シーン 'ncxt_queen' では i 行目のクイーンの置ける位置を列をかえながらさがす．実際にさがす処理は 'try/5' が行う．解が見つかればそれを返し，次の行を調べる．みつからなければ終了する．

B.4 哲学者の食事問題

哲学者の食事とはつぎのような問題である．5 人の哲学者がいる．哲学者はそれぞれ食事と思考の 2 つの行動をとる．部屋に丸いテーブルがあり，その真ん中に皿がある．丸いテーブルの周りにそれぞれ哲学者が座る席があり，その前に各自の皿が置かれている．フォークがその皿の間に 1 本ずつおかれている．哲学者は普通思考にふけっているが，空腹になると部屋にはいり，テーブルの自分の席に座って食事をする．料理を食べるのに 2 本のフォークを必要とする．哲学者は，まず 1 本のフォークをとり次にもう 1 本のフォークをとる．そして料理を自分の皿にとりそれを食べる．満腹になるとフォークをもとの場所におき部屋をでる．

このような問題をモデル化するのが哲学者の食事問題である．

```
module dining_philosopher.
public dining_philosopher/0.

class dinig_philosopher ;
      room(Door),
      merge({D0,D1,D2,D3,D4},Door),
      philosopher(D0,FR0,FL0),
      philosopher(D1,FR1,FL1),
      philosopher(D2,FR2,FL2),
      philosopher(D3,FR3,FL3),
      philosopher(D4,FR4,FL4),
      fork(FL4,FR0),
      fork(FL0,FR1),
      fork(FL1,FR2),
      fork(FL2,FR3),
      fork(FL3,FR4).
end class.

class room(+door)
      with +occupancy := 0.
  input door.
    :enter -> @occupancy < 5 | @occupancy += 1.
    :exit -> @occupancy -= 1.
end class.

class fork(+right,+left)
      with +busy := no.
  input right.
```

```
    :pickup(Ack) -> @busy = no |
      Ack <- ok, @busy := yes.
    :pickup(Ack) -> @busy = yes |
      Ack <- wait.
    :putdown -> @busy := no.

  input left.
    :pickup(Ack) -> @busy = no |
      Ack <- ok, @busy := yes.
    :pickup(Ack) -> @busy = yes |
      Ack <- wait.
    :putdown -> @busy := no.
end class.

class philosopher(-door,-right,-left)
    \\ thinking.

scene thinking.
    -> continue.
    -> @door <<= :enter,
        @left <<= :pickup(AckL),
        @right <<= :pickup(AckR)
            \\ should_wait(AckL,AckR).
end scene.  % thinking

scene should_wait(+ackl,+ackr).
    -> @ackl = ok, @ackr = ok | continue
        \\ eating(hunger).
    -> @ackl = wait , @ackr = ok |
        @left <<= :pickup(Ack) \\ wait_left(Ack).
    -> @ackl = ok, @ackr = wait |
        @right <<= :pickup(Ack) \\ wait_right(Ack).
    -> @ackl = wait, @ackr = wait |
        @right <<= :pickup(Ackr),
        @left <<= :pickup(Ackl) \\ should_wait(Ackr,Ackl).
end scene.  % should_wait

scene wait_left(+ack).
  input ack.
    ok -> continue \\ eating(hunger).
    wait -> @left <<= :pickup(Ack) \\ wait_left(Ack).
end scene.  % wait_left

scene wait_right(+ack).
  input ack.
```

```
    ok -> continue \\ eating(hunger).
    wait -> @right <<= :pickup(Ack) \\ wait_right(Ack).
end scene.  % wait_right

scene eating(+appetite).
  input appetite.
    hunger -> @appetite := full.
    hunger -> continue.
    full -> @door <<= :exit,
        @left <<= :putdown,
        @right <<= :putdown \\ thinking.
end scene.  % eating
end class.
```

このプログラムは, 部屋, 哲学者, フォークをそれぞれプロセスとしてあらわしている. はじめのプロセス 'dining_philosopher' は, これらのプロセスを起動し初期化するプロセスである.

部屋の状態をあらわすのがプロセス 'room' である. このプロセスは部屋にいる哲学者の人数を記憶している. 哲学者がはいってくる (enter) とこのカウンタを 1 増やし, 出ていく (exit) と 1 減らす.

哲学者はつぎに示す状態をとる.

- 思考 (thinking)
- 食事 (eating)
- フォークを使用できるかどうかの答えを待つ (should_wait, wait_left, wait_right)

おなかがすくと部屋にはいり, フォークが使用できるか問い合わせる. フォークが 2 本使えないときは使えるようになるのを待つ. フォークが 2 本使えれば食事をはじめる. おなかがいっぱいになるとフォークを置き, 部屋を出て再び思考をはじめる.

フォークには, 使用中 (busy) と未使用 (idling) の 2 つの状態がある. '使用中' の状態のときに問い合わせがくると 'wait' と答える. '未使用' のときは 'ok' と答えて, '使用中' の状態に移る. 哲学者がフォークを置くと, '未使用' の状態に移る.

参考文献

[1] ICOT TM-722 KL1 プログラミング入門編 / 初級編 / 中級編

[2] PIMOS マニュアル (第3.0版) プログラミング編