TM-1242

AYA プログラミング入門

寿崎　かすみ、石田　茂

February, 1993

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

AYA プログラミング入門

寿崎 かすみ　石田 茂

1993 年 1 月

目次

1 はじめに

プログラムをプロセスとプロセス間の通信により記述する方法を‘プロセス指向プログラミング’と呼びます.

AYA は, このプロセス指向プログラミングを自然に行なうために用意した言語です.

本書では AYA の簡単なプログラミングを, 例を用いて簡単に説明します. 詳細は‘aya 第 1 版解説書’(ICOT-TM 1206) を参照してください.

2 特徴

AYA ではプロセスを‘クラス’を単位として記述します. またプロセスが変化していく様子を, シーンが変わるという形でモデル化することにしました. シーンはプロセスのクラスを記述するときの単位で, 1 つのクラスに複数のシーンを記述することができます.

プロセスは, メッセージ通信・プロセスの内部メモリとして使用するためのホルダとして‘ソケット’を持つことができます. ソケットには入力と出力があります. メッセージの受けとりに使用するのが入力ソケット, 出力に使用するのが出力ソケットです.

プロセスの実際の動作は‘メソッド’で記述します. メソッドではソケットの具体化を待ち合わせ, その値ごとにさだめられた処理を行ないます. プロセスは通常, このメソッドの実行を繰り返し行ないます.

プロセス間の通信は, 2 つのプロセスがラインを共有することにより行ないます. 1 つのラインを 2 つのプロセスがそれぞれ入力と出力のソケットにもち, 出力のソケットから値をわたし入力のソケットでそれを受けとります.

メソッド中で使われるデータは, そのメソッドのなかでのみアクセスできます. メソッドを越えてデータをアクセスしたいときは, ソケットにいれておく必要があります. 入力のソケットにいれておくと, 必要なときに参照できます.

現在 AYA のプログラムは KL1 にコンパイルして PIMOS 上で実行します. KL1 のデータ型, 組み込み述語がほとんどそのまま使用できます. AYA から KL1 へのコンパイルは AYA のコンパイラを PIMOS の提供する KL1 のプリプロセッサとして組み込んで行ないます.

このような特徴を持つ言語 AYA でのプログラムの書き方を次章以降で説明します.

3 AYA のプログラミング

はじめに, 素数生成プログラムを AYA で書いてみます. 素数生成プログラムは与えられた最大値までの整数から素数を求め出力するプログラムです.

この問題をプログラムとして記述する方法はいくつかありますが, AYA ではこれをプロセスの集まりとして記述します.

ここでは, 整数を生成するプロセス, 生成された整数をふるうプロセス, 結果を出力するプロセスの 3 種類のプロセスで問題を解いています. 実際にはこのほかに, これらのプロセスを生成するプロセスも必要になります.

それぞれのプロセスをつぎのような名前で用意することにします.

- **top**
  素数生成プログラムのトップレベル・プロセスです. 素数をとりだすプロセスのトッププロセス‘primes’と結果を出力するプロセス‘output’を生成して自分は終了します. このプロセスを生成するときに生成する整数の最大値を指定します.

- **primes**
  整数を生成するプロセス 'gen' と生成されたプロセスをふるう ' ふるい ' のプロセスを生成するプロセス 'sift' を生成して自分は終了します.

- **gen**
  '2' から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す. 生成した整数はプロセス 'sift' に送られます.

- **sift**
  'filter' をすべて通過した整数つまり素数を受けとり, 結果を出力するプロセス 'output' に渡します. また, その素数の倍数をとりのぞく 'filter' を生成します.

- **filter**
  フィルタはプロセス 'gen' から整数を受けとり, すでに生成した素数の倍数かどうかを調べるプロセスです. 各素数が 1 つずつフィルタを持つことになります.

- **output**
  結果として得られた素数を出力するプロセスです. ここでは標準出力に出力することにします.

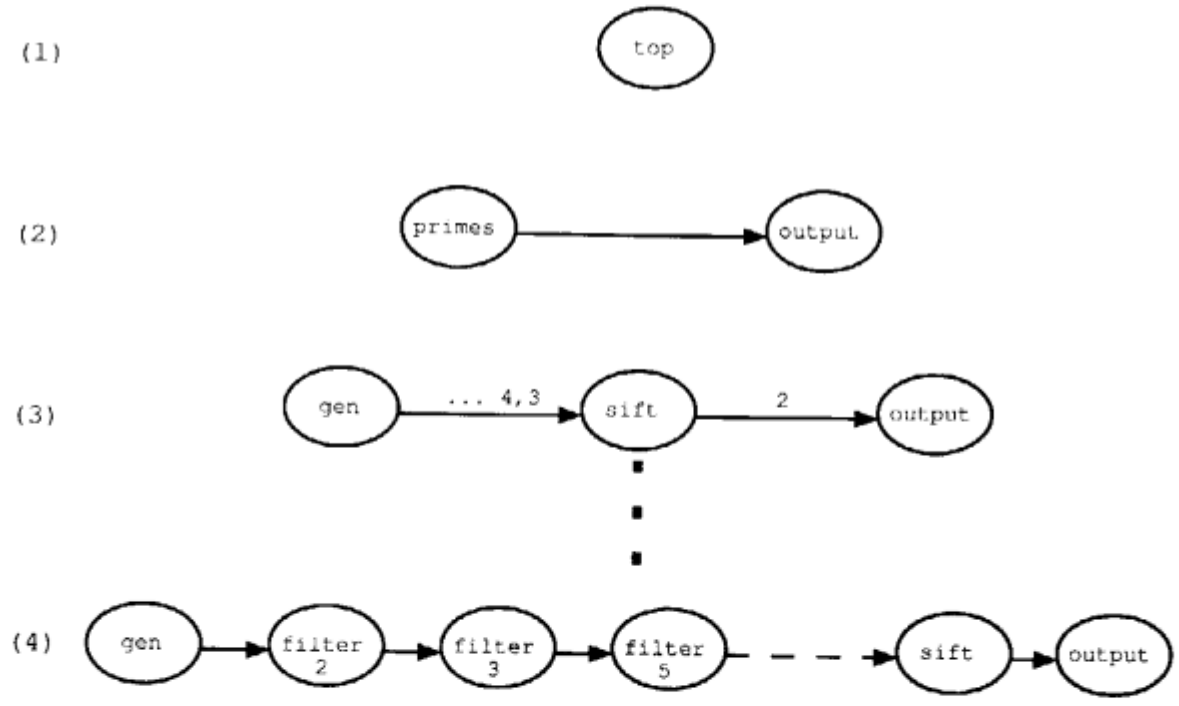

図 1: 素数生成プログラムのプロセス

これらのプロセスの様子を図 1に示します.

はじめ, 'top' が起動され (1), 'top' は 'primes' と 'output' を起動して自分は終了します (2). 'primes' はさらに 'gen' と 'sift' を起動して自分は終了します (3). 'sift' は各素数に対応したフィルタを起動していきます (4).

さてこのようなプロセスを AYA のプログラムで記述してみましょう.

3.1 プロセスの定義

はじめに, プロセスの定義について簡単に説明します.

プロセスはプロセスのクラスで定義します. プロセスのクラスはプロセスの名前と引数の個数の組で識別します. 引数はプロセス生成時に値を受けとるのに使用します。

プロセスのクラス定義はクラス名, 引数個数, 初期化処理を記述したはじめの部分とメソッド定義からなります. プロセスが使用するソケットの宣言はクラス名の後に行ないます.

3.2 クラス名・初期化処理

では, プロセス 'top' から順番に記述していきましょう. 'top' は名前が 'top' で, 生成する整数の最大値を受けとるための引数を 1 つ持ちます. ここまでを AYA で書くと

```
class top(Max)
```

となります. 'class' はプロセスのクラス定義のはじまりを示す予約語です. 'top' はプロセスを 2 つ生成して終了します. これを 'top/1' の初期化として記述することにします. 'top/1' にはメソッドがないのでソケットは不要です. さて, このようなプロセスを AYA で記述するとつぎのようになります.

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

'end class' はクラスの定義の終りを示します. セミコロンの後が初期化処理の記述です. ここでは `primes(Max,Ps)` と `output(Ps)` の 2 つのプロセスを起動しています. プロセスはこのあと終了します. '\\' で終了を指定しています.

プロセス 'primes' も同様に次のように記述できます.

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

'primes' は 2 引数のプロセスとして定義し, プロセス 'gen/3', 'sift/2' を生成して終了します. つぎのプロセス 'gen' は与えられた最大値までの整数を順に生成して 'sift' にわたすプロセスです. このプロセスは上限値と生成した整数の比較をし, 生成した整数のほうが小さい場合はつぎの整数を生成するという処理を繰り返します.

この条件判定と繰り返しはメソッドを使って記述します. 今度はソケットが必要です.

名前は 'gen' で引数を 3 つ用意します. 生成する整数の初期値を受けとる No, 生成する整数の最大値を受けとる Max, そして生成した整数をプロセス 'sift' に送るためのラインを受けとる Ns です. これら 3 つの引数で受けとった値はプロセスが処理を行なう間も参照・更新するので, ソケットにいれておくことにします. それをクラス名・引数のあとに 'with' という予約語に続けて記述します.

```
class gen(No,Max,Ns)
    with +no := No, +max := Max, -ns := Ns.
```

ソケットはソケットの名前, 入出力の指定, 初期値の組で宣言します. 入出力は名前の前に入力のときは'+', 出力のときは'-' を記述して指定します. 初期値は省略することもできます. 初期値を省略すると, 入力のソケットには'_', 出力のソケットには'[ ]' が設定されます.

ソケットの宣言のあとに';' に続けて初期化処理を記述することができます. クラス名・ソケット宣言・初期化処理を記述した部分の最後は'.' (ピリオド) です.

'gen' の例では初期化処理は行なわずメソッドを実行する状態にうつります.

つぎの処理をおこなうメソッドが必要です.

- 与えられた最大値までの整数を生成する.
- 最大値まで生成したらプロセスを終了する.

このメソッドをどのように記述するかについては, 次節で説明します.

3.3 メソッド定義

メソッドではソケットの値を待ち合わせます. そしてその値が条件にあうメソッドが選択され実行されます. この, 値の待ち合わせおよび条件判定を行なう部分をイベントおよびコンディションといい, 条件を満たしたときにおこなう処理の記述をアクションといいます. アクションの実行後プロセスを終了したいときにはそれを記述します. プロセスを終了するときはアクションのあとに'\\' と記述します.

メソッド定義の終りは'.'(ピリオド) です.

さて, プロセス 'gen' のメソッドを記述してみましょう.

はじめは, '与えられた最大値までの整数を生成する' メソッドです. このメソッドは生成した整数と最大値の大小比較を行なう必要があります. これをコンディションとして記述します.

```
-> @no =< @max |
```

'->' から '|' までがコンディションです. イベントは'->' の左に記述しますが, これの使い方はつぎの 'sift' のときに説明します.

ここでソケット 'no' の値と 'max' の値を比較しています. ソケット 'no' には前回のメソッドの実行で生成した整数が, 'max' にはプロセスの呼びだしのときにわたされた最大値がはいっています.

'no' の値が 'max' の値以下ならば条件が成立します. このようにソケットの値はソケット名の前に '@' をつけて記述します.

さてこのコンディションが成り立つとこのメソッドが実行されるつまりアクションに記述した処理が行なわれます. このメソッドではアクションとして

- 整数を 'sift' に送る.
- 1 大きい整数を生成する.

の 2 つの処理を行ないます. 整数を 'shift' に送るのは, プロセス 'shift' との間のラインに整数をわたすことで行ないます. ラインはソケット 'ns' に入っています. これにソケット 'no' にはいっている整数を送ることを

```
@ns <<= :(@no)
```

と記述します．これは，ラインをストリームとして使用する方法で，この方法だと必要なだけのメッセージを続けて送ることができます．ストリームについては次章で説明します．

1 大きい整数の生成はつぎのようになります．

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

ソケット 'no' に入っている値を 1 増やし，その値を再びソケット 'no' にいれます．一般にソケットの更新を ':=' を使ってかくことができます．

これで，メソッドがひとつ記述できました．全体をつなげてかくとつぎのようになります．

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

このメソッドの中ではソケット 'no' が 4 回でてきます．この 4 回のあいだに値の更新も行なわれています．一般にソケットが 1 つのメソッド中に複数回あらわれるとき，その値は左から右に新しくなります．この例ではコンディションの 'no' が一番古く，:(@no) の 'no' がそれと同じです．

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

は例外で右辺の 'no' のほうが古くそれ以前と同じ値をもち，これを用いて更新された値を左辺の '@no' はあらわします．このあとに再び 'no' が参照されると，それは更新後の値を表します．

さて，2 番目のメソッドを記述してみましょう．

'no' の値が 'max' より大きいということをコンディションに記述すると

```
-> @no > @max |
```

となります．この場合はプロセスを終了すればよいのですが，そのまえに整数が送られてくるのを待っているプロセスに，'もう送らない' ということをしらせてやらないとなりません．このためにクローズメッセージ':/' を送ります．これは

```
@ns <- :/
```

と記述します．':/' がクローズメッセージです．ストリーム通信をしていた場合にはかならずこのクローズメッセージを送る必要があります．

最後にプロセスの終了を指定します．これは'\\' と書きます．これまでのことをつなげると 2 番目のメソッドは

```
-> @no > @max |
    @ns <- :/ \\ .
```

となります．

つぎはプロセス 'sift' です．プロセス 'sift' は 'gen' から整数を受けとるラインと結果の素数をプロセス 'output' に送るラインを引数として受けとり起動されます．

はじめに到着する整数は '2' で素数なのでそれを 'output' に渡します．つぎにその素数の倍数をふるうためにプロセス 'filter' を生成し、gen から整数を受けとるラインを filter の入力とします。そして filter の出力を自分の入力とします．このようにすると 'shift' の受けとる整数は 'filter' を通過したものつまり素数となります．クローズメッセージを受けとるとプロセスを終了します．このようなプロセスはつぎのようになります．

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

ここではプロセス名のあとの引数にソケット宣言を書いています. 引数をそのままソケットの初期値とするとき, ソケット宣言をこのように書くことができます.

2行目は基本ソケット宣言です. 'input' のあとにソケット名を指定します. ここで宣言されたソケットの値とパタンマッチする値をイベントとして記述します。ここではソケット 'ns' を基本ソケットとして宣言しています. イベントを記述するためには基本ソケット宣言が必要です.

プロセス 'sift' も2個のメソッドで定義します. クローズ・メッセージが到着したときとそれ以外です. それ以外とはこの場合整数になります.

1つめのメソッドは整数が到着したときで, 到着した整数をソケット 'ps' にしまってあるラインに送ります. イベントに記述してある ':P' は, ソケット ns に 'P' が到着することをあらわしています. 前についている ':' はストリーム型の通信であることを示しています. このメソッドにはコンディションがありません. イベントが条件にあえばアクションが実行されます. プロセス 'filter' の生成は, 第2引数にソケット 'ns' の値をわたし, 第3引数に戻ってくる値を 'ns' の新しい値とします.

2つめのメソッドはクローズメッセージが到着した場合です. このときはソケット 'ps' にクローズメッセージをわたし終了します.

プロセス 'filter' も同様に記述できます.

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

このプロセスの1番目と2番目のメソッドはイベントとコンディションが両方記述してあります. 2番目のメソッドのアクションに記述してある 'continue' はアクションでなにも行なわないことを示します. この場合は何もしないでつぎのメッセージを待つことになります.

最後のプロセス 'output' は初期化処理のあとメソッドを実行する状態に移ります.

```
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.
```

'output' はソケットを 2 つ持ちます. 'ps' は呼び出されるときに受けとるパラメタを初期値とします. 'out' は初期値としては規定値の '_' をとりますが, 初期化処理のなかで標準出力へのストリームが設定されます.

1 つめのメソッドではソケット 'out' に 'putt(X)' と 'nl' の 2 つのメッセージを連続して送っています.

3.4 モジュール宣言・パブリック宣言

以上で素数生成プログラムができたわけですが, これを実際にコンパイルして実行するためには, モジュール宣言とパブリック宣言を加えないとなりません.

モジュールはコンパイルのときの単位で, 1 つ以上のプロセスの集まりを 1 つのモジュールとして記述します. 1 つのファイルに複数のモジュールを記述すること, 1 つのモジュールを複数のファイルに分けて記述することはできません.

また, モジュール間にわたって呼び出しを行ないたいプロセスはパブリック宣言をする必要があります. パブリック宣言は複数個のプロセスに対して行なうことができます。モジュール宣言・パブリック宣言はファイルの先頭に記述します.

モジュール名を 'prime' とし, トップレベル・プロセス 'top/1' をパブリックなプロセスとして宣言すると素数生成プログラムはつぎのようになります.

```
% 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),
      @ns := 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.
```

4 少し進んだ AYA のプログラミング

ここでは AYA の少し進んだプログラミング・テクニックとしてソケット・ライン・ストリームの使い方とシーンを使ったプログラミングについて説明します.

4.1 ソケット・ライン・ストリーム

まず, 前章までに述べたことを簡単に復習します.

ソケットはプロセスが値を持つためのホルダで入力と出力があります. メッセージの受信に使用するのが入力のソケット, 送信に使用するのが出力のソケットです. ソケットはプロセス間通信に使用するほかプロセスが自分のなかで参照・更新する値を持つためにも使用します.

プロセス間の通信は 2 つのプロセスがラインを共有することにより行ないます. 1 つのラインを 2 つのプロセスがそれぞれ入力と出力のソケットに持ち, 出力のソケットから値をわたし入力のソケットでそれを受けとります. 一度値を持ったラインに再度異なる値をわたすことはできません。

ラインの使い方の 1 つにストリームがあります. ストリームの使い方をするとメッセージを連続しておくることができます.

ここではこれらのことを, もうすこし詳しく説明します.

4.1.1 ソケット・ライン

ソケットに対して参照・読みだし・更新ができます. 入力の場合は参照, 出力の場合は読みだしになります. 参照・読みだしおよび更新はソケット名の前に '@' をつけて行ないます.

更新には ':=' を用います. ':=' の左辺に記述したソケットの値を右辺に指定した値に更新します.

```
@no := 2
```

とするとソケット'no'の値は2になります.

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

この場合は右辺の'no'で現在の値を参照し, その値に1を加え, その結果を'no'の新しい値としています.

プロセスが自分のなかに値をもつときは, 入力のソケットをこのように使用します.

ソケットにライン持ったラインを使ってメッセージの送受信をするときは, ソケットの中のラインに値をわたす操作が必要になります. これをつぎのように記述します. これはユニフィケーションと呼ぶ操作です.

```
@out <- message
```

この操作で, ソケット'out'にはいっているラインに'message'というアトムをわたすことができます. 受けとりはコンディションに

```
-> @in = message |
```

と書きます. あるいはソケット'in'を基本ソケットと宣言しておいて

```
message ->
```

と書くこともできます.

このようにしてメッセージの送受信を記述することができますが, この方法では一度しかメッセージを送れないという問題があります. つぎのメッセージを送るためには, なんらかの方法で新しいラインを設定しなくてはなりません.

このような問題を解決するためにストリームという方法を使用します.

4.1.2 ストリーム

ストリームでは, 続けてメッセージを送るためにメッセージとつぎに使うラインを一緒に送るという方法を使います.

```
@out <- [message|Next]
```

メッセージ'message'と次に使うライン'Next'をリストセルにつめて一緒に送ります. リストセルを送りだしたソケットと受けとったソケットに新しいライン'Next'をそれぞれ入れることにすれば続けてつぎのメッセージを送ることができます.

このストリーム・メッセージの送受信をそれ以外のものと区別するためにメッセージの前に':'をつけて記述します.

また, 送信は'<<='を用いてつぎのように記述します.

```
@out <<= :message
```

受信は

```
-> @in = :message |
```

あるいは

```
:message
```

となります.

複数のメッセージをつづけて送るときは

```
@out <<= :message:message2
```

のようにつづけてメッセージを記述します.

この内容をストリームの記述法を用いずに書くとそれぞれつぎのようになります.

```
@out <- [message|Next], @out := Next

-> @in = [message| Next] | @in := Next
[message|Next] -> @in := Next

@out <- [message | [message2| Next]], @out := Next
```

ストリームを用いてメッセージのやりとりをしているときは自動的に次に使用するラインが設定されています. そこで, 通信を終りにする場合には設定されているラインを使ってしまう必要があります.

このために送るメッセージをクローズ・メッセージといい':/' と書きます. クローズ・メッセージはストリーム・メッセージではないのでユニフィケーションを使います. クローズメッセージ':/' を送ることは'[]' を送ることと同じです.

クローズ・メッセージの送受信はつぎのように記述します.

```
@out <- :/

-> @in = :/
:/ ->
```

4.2 シーンを使ったプログラミング

最後に' シーン' を使ったプログラミングの説明をします.

リーダーズ・ライターズ問題のプログラムを書いてみながら' シーン' とはなにかどのように使用するかをみていきます.

はじめにリーダーズ・ライターズ問題について簡単に説明します.

リーダーズ・ライターズ問題は共有資源アクセスの管理を扱っています. つぎのようなアルゴリズムに基づいています.

- 読みだし要求は同時にいくつでも扱えます.
- 書き込み操作が行なわれている間は他の要求は一切受け付けません.
- 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ません. 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ないます.

このような問題を AYA で記述してみます. この共有資源アクセスを管理するプロセスを 'readerswriters' という名前にします. このプロセスの動作は前述のアルゴリズムよりつぎの 3 つの場合で異なります.

- 読み出し要求の処理中. 書き込み要求は受けとっていない.

- 書き込み要求の処理中.
- 読みだし要求の処理中. 書き込み要求も受けとった.

この3つの場合をきりわける方法の1つは, 状態を示すフラグを用意し毎回それをみて動作を決めるというものです. しかし, それは煩わしくまたそのようにして記述したプログラムは読みにくいものになりがちです. そこで, AYA ではこれをプロセスが3つのシーンを持ち, プロセスのシーンが変わることとモデル化します. プロセスの動作はシーンごとに独立に記述します. プロセスはいつもこれらのシーンのいずれか1つとして動作をします.

プロセスは必ず1つのシーンを持ちます. あらためてシーンを記述しないとそれは, プロセスが必ず持つシーンとみなされます. 素数生成プログラムのプロセスは, このプロセスが必ず持つシーンとして記述していました.

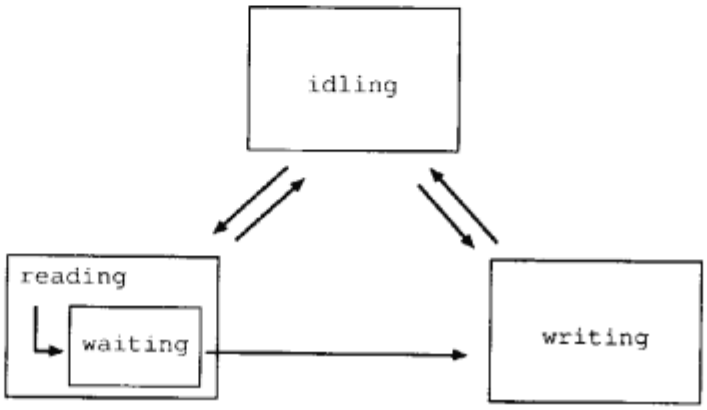

図 2: readers writers

さて複数のシーンで定義されるプロセスを記述してみましょう. readerswriters のプロセスは前に説明した3つのシーンと何もしていないシーンの4つのシーンを持つものとして記述できます.

このシーンの様子を図であらわしたものが図2です. 何もしていないシーン・読みだし処理中・書き込み処理中の3つのシーンがあります. このほかに, 読みだし処理中に書き込み要求を受けとったときを読みだし処理中の特別な場合として, その内側のシーンとしています.

プロセスは呼び出されると何もしていないシーン 'idling' になります. プログラムではつぎのように記述し, つぎのシーンの指定といいます.

```
\\ idling.
```

終了を示す'\\' は'\\terminate.' の省略形です. プロセスの終了は言語定義の 'terminate' というシーンになることとみなします.

このプロセスはクラス定義で request, tofile, fromfile の3つのソケットを宣言します. request はユーザから管理プロセスへの要求がくるラインをしまいます. tofile, fromfile はそれぞれ管理プロセスとデバイスの間の通信に使います. このようにクラス定義で宣言するとこれらのソケットはどのシーンでも共通に使用できます. プロセス 'readerswriters' のクラス定義には初期化処理はなくシーン 'idling' に変わります.

ここまでのプログラムはつぎのようになります.

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

つぎはシーン 'idling' です. シーンの定義はクラス定義のなかに行ないます. 予約語 'scene' のあとにシーン名と引数を指定して, ソケット宣言・初期化処理・次のシーンの指定をクラス定義と同様に記述します. シーン定義は 'end scene' で終ります.

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

idling ではユーザからの 'read' あるいは 'write' の要求を受けとり, ファイルに対してそれぞれの操作を要求するメッセージをおくります. そして読みだし中 'reading'・書きこみ中 'writing' に変わります.

つぎは読みだし中のシーン 'reading' です.

'reading' は読みだし処理がすべて終ったときに他のシーンに変わる必要があるので, 処理中の読みだし要求の数を把握している必要があります. このために 'reading' でだけ使用するソケットを 1 つ宣言します.

```
scene reading
    with +readers := 1.
```

この状態になったときには既に読みだし要求が処理されているので, このソケットの初期値は '1' にします.

このシーンはつぎの 4 種類の動作をします.

- 読みだし処理がすべて終了したら idling になる. 読みだし処理の終了はソケット readers が示している処理中の読みだし要求の数が 0 になったかどうかで確認する.
- ユーザから読みだし要求がきたらファイルに読みだし要求を送り, 読みだし処理の数を 1 増やす.
- ユーザから書き込み要求がきたら, 読みだし処理がすべて終るのを待つシーン waiting になる. このとき, 書き込むデータを引数として渡す. waiting は reading の特殊な場合として waiting の内側に定義してあるので, waiting でソケット readers が 0 になったかどうかを確認することができる.
- ファイルから読みだし処理の終了を伝えるメッセージが来たら読みだし処理の数を 1 減らす.

これをメソッドとして定義するとつぎのようになります.

```
  -> @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).
```

このメソッドの記述には基本ソケット宣言が 2 回あります. このように基本ソケット宣言は何回でも記述できます. 基本ソケット宣言は, 次の基本ソケット宣言までのメソッドに対して有効です.

'reading' の 1 つめのメソッドは基本ソケット宣言を行なわず, コンディションでソケット readers の値を調べます. 2 番目, 3 番目のメソッドはそれぞれ 'request' に:read(Data), :write(Data) が到着するのを待っています. 最後のメソッドは 'fromfile' を基本ソケットとしています.

つぎは waiting です. waiting は reading の中のシーンなので, 定義も内側に記述します.

また reading で宣言したソケットも含めて 'requcst', 'tofile', 'fromfile', 'readers' の 4 個のソケットにアクセスできます.

また, シーン起動時にわたされた書き込みするデータをしまっておくためのソケット writedata を自分のなかに宣言しています.

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

end scene. % reading
```

waiting ではユーザからの要求はうけつけず, ファイルからの処理の終了を伝えるメッセージのみを待って読みだし処理がすべて終了したところでファイルに書き込み要求を送り writing になります.

waiting の定義で reading の定義は終りなので, シーン reading の 'end scene.' をそのあとに記述します.

シーン writing は書き込み要求の終了だけを待ち合わせ, 終了を伝えるメッセージが来たら idling になります.

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

end class.
```

最後にクラス定義のおわりをしめす 'end class' を記述します.

もう一度, readerswriters プロセスの全体を記述してみましょう.

```
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
```

A AYA コンパイラの使い方

AYA コンパイラの使い方を簡単に説明します. この使い方は将来変更されることもあります.

A.1 AYA コンパイラのインストール

AYA のプログラムをコンパイルするためには, AYA のコンパイラをインストールする必要があります. リリースされているアンロード形式のファイルをロードすることでインストールが完了します. 詳しくは, インストールの手引を参照して下さい.

A.2 AYA コンパイラの構成

AYA コンパイラは, KL1 コンパイラのプリプロセッサとして動作します. AYA で記述したプログラムは, AYA コンパイラが読み込み解析し, KL1 プログラムに変換します. この KL1 プログラムを KL1 のコンパイラが受け取りコンパイルします. この様子を図 3 に示します.

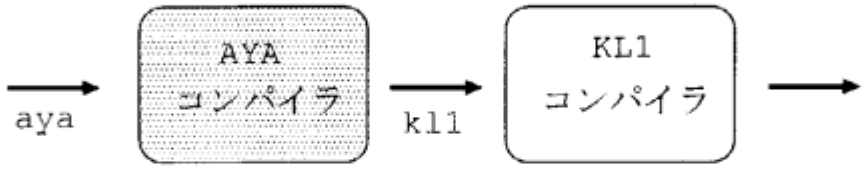

図 3: AYA コンパイラの構成

この機能を使用するために, AYA プログラムのソースファイルは '.aya' で終らなくてはなりません.

A.3 起動方法

それでは, AYA のプログラムをコンパイルし, 実際に動かしてみましょう. この例では, リスナからコンパイラを起動しています. KL1 コンパイラのバッチコンパイル・モード, インタラクティブコンパイル・モードのどちらでも使用できます. それぞれ, 図 4, 図 5に示します.

人力ファイル名は, ".aya" まで指定します.

```
[10] compile("prime.aya").
** KL1 Compiler **
Compile File : icpsi531::>sys>user>ishida>AYA.DIR>SAMPLE>prime.aya.1
!WARNING! no with_macro declaraion. assumed 'pimos'.
[prime.aya@0] Compile Module : prime
=> module prime .
Compile Succeeded : prime

"ishida:: prime" Updated
Total Number of Warning : 1
Compilation Time = 8185 [MSEC]

[11] prime:top(10).
2
3
5
7

[12]
```

図 4: バッチコンパイルの例と実行例

```
[10] compile.
** KL1 Compiler **
COMPILE> prime.aya
Compile File : icpsi531::>sys>user>ishida>AYA.DIR>SAMPLE>prime.aya.1
!WARNING! no with_macro declaraion. assumed 'pimos'.
[prime.aya@0] Compile Module : prime
=> module prime .
Compile Succeeded : prime

"ishida:: prime" Loaded
Total Number of Warning : 1
COMPILE> !exit

[11] prime:top(10).
2
3
5
7

[12]
```

図 5: インタラクティブコンパイルの例と実行例

A.4 エラーメッセージ

AYA コンパイラがコンパイル中に発見した不具合はその内容によりエラーあるいはワーニングとしてメッセージ出力されます. エラーの場合は, !ERROR! というメッセージの後に, ファイル名, ターム番号, エラーの種類, 不具合の箇所を表示します. ワーニングの場合は, !WARNING! というメッセージの後に, ファイル名, ターム番号, ワーニングの種類, 不具合の箇所を表示します. ただし, この内容は今後変更になる可能性があります.

メッセージ出力されるのは次のような場合です.

A.4.1 シンタクッスエラー

プログラム読み込みのときに発見した文法上の誤りを, エラーとして報告します.

A.4.2 構文上のエラー

構文上の誤り, ソケットの後始末忘れなどを報告します. この場合, 内容によって, エラーとして報告するものとワーニングとして報告するものがあります.

A.4.3 モードの不整合

AYA のコンパイラは, 変数およびクラスやシーンの引数の入出力のモードチェックを行なっています. クラスやシーンの呼びだしの引数のモードが整合しているか, 1 回しかあらわれない変数がないか, 出力の変数を複数箇所で具体化することはないかなどを確認し, 不具合が発見されれば警告します. この場合, 内容によって, エラーとして報告するものとワーニングとして報告するものがあります.

B　演算子順位

AYA で使用している演算子とその定義を表 1に示します.

C　文法

AYA の文法を以下に示します.

C.1　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>

<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>

<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りません.
  - <plus socket declaration> については <plus term>
  - <minus socket declaration> については <minus term>
- <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 '_'

表 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

が設定されます.

- ソケットの宣言と同時に "register 宣言" を行うことができます. 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 のあとに宣言されるソケットのいずれに対しても行えます. "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します.

- < *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) に対応します.

- クラス (クラスが既定値として持つシーン) の <initiation> につづいて <next scene> を指定し, かつこのクラスのメソッドを定義することが文法上は記述できます. しかしこれらのメソッドは実行されることがないので KL1 へのコンパイル時にワーニング・メッセージをだします.

- ''\\'' [<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します. ''\\'' のみ記述するのは

  ''\\ terminate''

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

C.2 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>
```

- シーンの <initiation> につづいて <next scene> を指定し, かつこのクラスのメソッドを定義することが文法上は記述できます. しかしこれらのメソッドは実行されることがないので KL1 へのコンパイル時にワーニング・メッセージをだします.

C.3 メソッド

```
<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>]]]
```

C.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とのユニフィケーションです. 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ります.
- 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は, 複数メッセージの列とのユニフィケーションも記述できます.

C.5 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ます.
- < *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行ないます.

C.6 アクション

```
<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，プロセスの生成です．また，アクションとしてなにも行わない場合は "continue" を記述します．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ます．

- 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ません．

- < *stream op* > による更新処理はそれぞれストリームの merge in(<=), prepend(<<=), append(<=<) にあたります．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す．

C.7 ターム

```
<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ります. 前者を < *condition term* > とよび後者を < *term* > と呼びます. < *condition term* > ではソケットの更新を行うことはできません.
- ターム (< *condition term* >, < *term* >) には入力と出力があります. 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ります. 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ります.
- < *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す.
- 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* > があります.
- ストリーム型のメッセージは ":" をつけて示します. close message ([ ]) は ":/" です.