TM-1256

AYA 入門

石田　茂、寿崎　かすみ

April, 1993

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~ 5

Institute for New Generation Computer Technology

AYA 入門

石田 茂　寿崎 かすみ

1993 年 3 月

目次

第 1 章

はじめに

AYA は, 並列論理型言語 KL1[1][2] の上位言語として設計された, プロセス指向プログラミング言語です. プロセス指向プログラミングとは, プログラムをプロセスとプロセス間の通信で記述する方法をいいます. 現在の処理系は, AYA のプログラムを KL1 にコンパイルして実行します. 処理系は, PIMOS3.5 版 [3][4] 上で動作しています.

本書は, これから AYA を学ぼうとしている人たちに, AYA の基礎を理解していただくことを意図しています. 基礎的な KL1 のプログラムを書ける人を読者の対象としているので, KL1 を知らない人は文献 [2] を併せて読まれることをお薦めします. AYA の文法の詳細は, aya 第 1 版解説書 [5] を, プログラムの設計のしかたや処理系の使い方については, AYA プログラミング入門 [6] を参照して下さい.

第 2 章

AYA の基本

ここでは AYA の基本的な概念を説明します.

プロセスとソケット

AYA の基本はプロセスと通信です. AYA ではプロセスをクラスとして定義します. プロセス間の通信はラインを共有することによって行なわれます. プロセスは, 通信用のチャネルあるいは, 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のモードがあり, 入力ソケットはメッセージの受け取りに, 出力ソケットは送信に使用されます. また, 入力ソケットはプロセスの内部メモリとしても使用します.

プログラムの構成

プロセスは, クラス単位で定義され, 各プロセスは状態をあらわすシーンを持ちます. AYA ではプロセスの状態が変わっていく様子をシーンが変わるという形でモデル化しています. 1 つのクラスには複数のシーンが記述でき, また, シーンをネストして記述することもできます. 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値の具体化を待ち合わせ, その値が条件に合うメソッドが選択されます. メソッドは, イベント, コンディション, アクションからなります. 値の待ち合わせおよび条件判定を行なう部分がイベントおよびコンディションで, 条件を満たしたときに行なう処理の記述がアクションです. シーンを移動するときは, 移動先のシーンを指定します.

通信メカニズム

プロセス間の通信はラインを共有することによって行なわれます. 1 つのラインを一方のプロセスが入力ソケットに持ち, 他方のプロセスが出力ソケットに持ちます. 出力ソケットからメッセージを送り, 入力ソケットでそれを受けとります. ラインは一度値を持つと再度異なる値を持つことができません. そのため, プロセス間で連続してメッセージを送信する場合にはストリームを用います. ストリームはラインの使い方の 1 つです.

第 3 章

AYA のプログラミング

3.1 プログラムの構成

3.1.1 簡単なプログラム

次のプログラムは,「0 という入力に対して 1 を, 1 という入力に対して 0 を返すプログラム」の例です.

```
module not .
public not/2 .

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

このプログラムを PIMOS のリスナ上で実行すると, 次のようになります.

実行例

```
[7] not:not(0, X).
X = 1
[8] not:not(1, X).
X = 0
[9]
```

以下, プログラムの内容について説明します.

3.1.2 モジュール宣言・パブリック宣言

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行なうためのパブリック宣言です.

AYA ではプログラムを, モジュールごとに分割して記述します. モジュール宣言はモジュール名を指定し, プログラムの先頭に記述します. モジュールは 1 つ以上のクラス定義の集まりです. プロセスのうちモジュール間にわたって呼びだしを行ないたいものはそのクラス名と引数を指定してパブリック宣言をします.

3.1.3 プロセスの定義

プロセスはクラスとして定義します. class から end class までがクラスの定義です. プロセスのクラスはクラス名と引数の個数で識別します. 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います.

プロセスのクラスの定義は, 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からなります. 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します. この後にソケット宣言, プロセスの初期化処理を記述します.

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しています. ソケットのモードは, 入力は '+' で, 出力は '-' で指定します. プロセスの初期化処理はありません. '\\' はシーンの移動を表します. シーンを移動する時は, '\\' の後に移動先のシーン名を記述します. 初期化処理の後, プロセスを終了したい場合は '\\ .' と記述します. '\\ .' は言語定義シーンの terminate に移動することをあらわします. 移動先の指定がないときは, プロセスは初期化処理終了後, 定常状態になります. 他のプロセスからのメッセージはこの定常状態で受け付けます.

3.1.4 メソッドの定義

実際のプロセスの動作はメソッドで記述します. メソッドの記述は以下の内容からなります.

- イベント
- コンディション
- アクション
- 移動先のシーン名

イベントには, 到着したメッセージとパタンマッチするパタンのみを記述します. イベントを記述するときには, メッセージが到着するソケットを, 基本ソケットとして, 'input' の後に指定します. コンディションには, 到着したメッセージとのパタンマッチや大小比較などの条件判定を記述します. AYA では, 整数や実数の大小比較, データ型のチェック, データ型の変換などの組込みプロセスを記述できます. イベントおよびコンディションで, 条件が満たされるとそのメソッドが選択されます. 複数のメソッドが選択可能な場合は, その中の任意のメソッドが選択されます.

イベントとコンディションは省略することができます. 省略された場合は, ただちにアクションが実行されます. この例ではイベントが省略されています.

アクションには, そのメソッドが選択された場合に行なう処理を記述します. アクションで記述する処理は, ソケットの操作, ユニフィケーション, プロセスの起動です. プロセスの起動とは, クラスの呼び出し, 組込みプロセスの呼び出しです. アクションで複数の処理が記述されている場合, それらの処理は並列に実行されます. また, 何も行なわない時は continue と記述します. プロセスの実行を終了する時には,'\\ .' と記述します. そのままの状態を繰り返す時は, アクションの後には何も記述しません.

プロセス not/2 には 2 つのメソッドが定義されています. ソケット in に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 を出力ソケット out の値とユニフィケーションし, 処理を終了します. ラインを具体化して値をもたせることを, ユニフィケーションと呼びます.

3.1.5 メッセージとライン

プロセス間の通信に使用するデータをメッセージといいます.

AYA では KL1 同様以下のデータを扱えます. 詳しくは, PIMOS のマニュアル [4] を参照して下さい.

整数 ... 123, 16#"ACE", 8#"37"

浮動小数点数 ... 1.23, 1.0e10, 3.0E-30, -2.0

アトム ... abc, 'ABC'

リスト ... [1,2,3], [1|X]

ベクタ ... {a,Y,b}, f(X), {}

ストリング ..."abc", "", string#"abc"

モジュール ... module#foo, module#bar

プロセス間の通信はラインを共有することによって行なわれます.

3.1.6 ソケット

ソケットは, プロセスが値を保持するために使用するホルダで, 通信用のチャネルあるいはプロセスの内部メモリとして使用されます.

ソケットはモードに応じて, 参照・読みだし・更新という操作が行なうことができます. ソケットのアクセスは, ソケット名の前に '@' をつけて指定します. 入力のソケットの場合は値の参照, 出力のソケットの場合は値の読みだしを行ないます. 読み出された後のソケットの値は '_' になります.

not/2 のプログラムを振り返ってみましょう.

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

この例では, コンディションでソケット in を参照し, その値を 1 あるいは 0 と比較しています. アクションで, 出力ソケットにはいっているラインを 0 または 1 とユニフィケーションします. 読み出された後のソケット out の値は '_' になっています.

次にソケットの更新について説明します. 次のプログラムは, 整数を要素とするリストの総和を求めるプログラムです.

```
class sum(+in, -sum)
     with +psum := 0 .
  -> @in = [] |
        @sum <- @psum \\ .
  -> @in = [One|Rest] |
        @psum := ~(@psum + One) ,
        @in := Rest .
end class .
```

ソケットの値の更新は ':=' を用いて行ないます. ':=' の左辺にソケットを, 右辺に新しい値を指定します. あるソケットが 1 つのメソッド中に何度もあらわれ, 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新しくなります. 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ちます.

```
@psum := ~(@psum + One)
```

は, それまで ソケット psum に入っている値を 1 増やし, その値をソケット psum の新しい値とします.

```
@in := Rest
```

は, それまで ソケット in に入っていた値を捨て, Rest を新しい値とします.

ソケットの値の更新を行なうと, それまではいっていた値は捨てられます. このとき, ソケットのモードに応じて次のような後始末を自動的に行ないます.

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 [] に具体化して捨てる

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ります. このとき出力ソケットの中にラインを残したままソケットを捨ててしまうと, ラインの末端を持つプロセスがデッドロックする可能性があります. そこでソケットが最後に保持していたデータに対して次のような後始末を自動的に行なっています.

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

3.2 ストリーム通信

プロセス間で続けてメッセージ通信を行ないたい時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します. このような通信をストリーム通信と呼びます.

次の例は, 整数を生成するプログラムです.

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

ソケット ns は出力のソケットです.

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

が行なわれると, ソケット ns の値は '_' になります. そして,

```
@ns := Ns
```

によって新しく Ns が設定されます.

AYA では, ストリーム型のメッセージをあらわす記法をサポートしています. この記法を用いて gen/3 は次にようにも書けます.

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

ストリーム型のメッセージとそれ以外のものとを区別するためにメッセージの前に ':' をつけて記述します.

1 つめのメソッド中の

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

は, ソケット ns に入っているラインに, メッセージとしてリストセルを送り, その Car 部にソケット no の値を, Cdr 部にソケット ns の新しい値を設定します.

2 つめのメソッド中の ':/' はクローズ・メッセージといい, '[]' と同じ意味です. 通信の終了は, 新しいラインの設定を必要としないので,

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

と, 通常のユニフィケーションとします.

この記法で sum/2 を書き換えると次のようになります.

```
class sum(+in, -sum)
    with +psum := 0 .
  input in .
  :One -> @psum := ~(@psum + One) .
  :/ -> @sum <- @psum \\ .
end class .
```

3 行めは基本ソケット宣言です. イベントに対応するソケット名を記述します. 1 つめのメソッドのイベントの ':One' は, ソケット in に One が到着することをあらわしています. 前についている ':' は, ストリーム型のメッセージであることを示しています. 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.

与えられた数以下の自然数の和を求めるには, これら 2 つのプロセスを組み合わせ次のようにします.

```
class sum_up_to(Nun, Sum) ;
    gen(1, Num, List), sum(List, Sum) \\ .
end class.
```

3.3 状態の移動

最後にシーンを使ったプログラムの説明をします. 次のような例を考えてみましょう.

io_manager プロセスは通信を制御するプロセスです. send(Data, Status) というメッセージを受け取ると, そのメッセージをデバイスへつながるストリームに送ります. ただし, メッセージを送ることができるのは, 前のメッセージが正常に終了した後だけです. 正常にメッセージが終了しなかったときは, それ以後に受け取るメッセージはアボートします.

この例題を AYA で記述すると以下のようになります.

```
module communication .
public io_manager/2 .

class io_manager(+in, -out) \\ idling .

scene idling .
  input in .
    :send(Data, Status) ->
        @out <<= :send(Data, Ack) \\ waiting(Ack, Status) .
    :/ -> continue \\ .

  scene waiting(+ack, -status) .
    input ack.
      normal -> @status <- normal  \\ idling .
      abnormal -> @status <- abnormal \\ abort .
  end scene . % waiting

end scene . % idling

scene abort .
  input in .
    :send(_, Status) -> Status <- aborted .
    :/ -> continue \\ .
end scene . % abort

end class .
```

このプログラムでは, プロセスが 3 つの状態を持ち, それらをシーンを用いてモデル化しています. これらのシーンの関係を図にあらわすと次のようになります.

図 3.1: シーンの関係

シーンの定義は, シーン名, 初期化処理などを記述した部分とメソッド定義からなります. シーン名は予約語 scene のあとに引数とともに記述します. この後にソケット宣言, 初期化処理を記述

します. scene から end scene までがシーンの定義です. シーンは, シーン名によって識別されます. シーンの中からアクセスできるソケットは, そのシーンで宣言されているソケット, 上位のシーンで宣言されているソケットおよびクラスで宣言されたソケットです. この例では, シーン waiting で, ソケット in, ソケット out をアクセスすることができますが, シーン abort でソケット ack, ソケット status をアクセスすることはできません. また, 移動できるシーンの範囲は, 自分より 1 つ下のレベルのシーン, 同じレベルのシーン, 自分より上位のシーンです.

プロセス io_manager には, メッセージの到着を待つ状態 (idling), メッセージの実行結果を待つ状態 (waiting), メッセージをアボートする状態 (abort) の 3 つの状態があります. プロセス io_manager は生成されると, まずはじめにシーン idling に移動します. メッセージ send(Data, Status) の到着を待ち合わせ, メッセージが到着すると, そのメッセージを出力のストリームに送信し, シーン waiting に移動します. シーン waiting では, メッセージの実行結果を待ち合わせます. 実行結果が normal の場合は, シーン idling にもどり処理を繰り返します. abnormal の場合は, シーン abort に移動し, 以後受信するメッセージをアボートします.

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] LPC '91 チュートリアルKL1 入門ICOT 近山 隆

[3] PIMOS マニュアル (第 3.0 版) 操作編

[4] PIMOS マニュアル (第 3.0 版) プログラミング編

[5] ICOT TM-1206 aya 第 1 版解説書ICOT 寿崎かすみ

[6] ICOT TM-1242 aya プログラミング入門ICOT 寿崎かすみ