ソフトウェア工学　35－4
（１９８４．５．３０）

言語／システム　ＬＯＯＰＳのオブジェクト指向からのアプローチ

丸一　威雄，石川　裕，所　真理雄
（　慶應義塾大学理工学部　電気工学科　）

１　はじめに

オブジェクト指向プログラミングは、物体（オブジェクト）へメッセージを送ることで計算を進めるという理論を基本にしている。　オブジェクト指向の概念は、データとプログラムの両方の性格をもつオブジェクトにより、データ抽象化，プログラムのモジュール化，情報陰蔽，局所化等を実現することができる点で、注目されてきており、近年これらを実現する言語やアーキテクチャが考案されてきている。　言語としては、Smalltalk-80が有名で、プログラム上に現れるものはすべてオブジェクトとメッセージであるという点に特徴がある。

本論文は、エキスパート・システム構築のツールとしてXerox PARCで開発されたＬＯＯＰＳの基本的な機能と概念を調べることを目的としている。　ＬＯＯＰＳは、プロシジャ指向、オブジェクト指向、データ指向、ルール指向という４つのプログラミング・スタイルを統合した点に特長があるが、その機能はオブジェクト指向を中心に構成されていると考えてよい。　ここでは、ＬＯＯＰＳをオブジェクト指向プログラミング言語としてとらえて、言語としての機能、およびシステムとしての機能を解説する。　現在、我々は発表されているＬＯＯＰＳのマニュアルを参考に、これをFranz Lisp上に作成中である。本論文中のプログラミングの例は、これを使用する。

２　オブジェクト指向プログラミングの概念と機能

２．１　プログラミング・シンタックス

ＬＯＯＰＳはオブジェクト指向についての概念の多くを　Smalltalk から継承している。Smalltalk では、オブジェクト間のメッセージ・パッシングを、

```
names at: names size put: name
```

のように記述した。これと等価なメッセージ・パッシングをＬＯＯＰＳでは、

```
(<- names  at:put: (<- names size) name)
```

のように表す。　つまり、Smalltalk のシンタックスが、

```
ReceiverObj    Selector1:   Obj1   Selector2:  Obj2   ...
```

なのに対し、ＬＯＯＰＳでは、

```
(<- ReceiverObj  Selector:  Obj1  Obj2  Obj3  ... )
```

である。　また、Smalltalk では、カスケードと呼ばれている次の様なシンタックスがあるが、

```
array at: index1 put: 0;  at: index2 put: 1;  size.
```

ＬＯＯＰＳでは、

```
(<- (<- (<- array at:put: index1 0) at:put: index2 1) size)
```

の様に表すことができる。

２．２　オブジェクトの構造

Smalltalk では、すべてのデータ型がオブジェクトであるのに対し、ＬＯＯＰＳでは基本的なデータ型はLispのものを使用している。　従ってこの言語は、Lispのもつデータ型にオブジェクトというデータ型を加えたものとみることができる。

実際のオブジェクトはLispのアトムで表わす。このアトムは、時刻とシステム内のユニーク・ナンバーから生成されるユニーク・ネームであり、オブジェクトの構造およびインヘリタンスに関する情報はすべてこのユニーク・ネームの下へ属性リストとして格納される。この構造を Fig-1に示す。　オブジェクトをユニーク・ネームの下の属性として格納する事は、情報の局所性を高め、オブジェクトの構造を外へ見せない事から情報陰蔽の機能もはたす。このような機能を持つオブジェクトも、どこからも参照されなくなると、ガーベッジ・コレクターにより回収される。

```
(Obj_I00030

  (Self Instance)
  (MetaClass RoadMap)
  (Name MapTest1)
  (:ySize 0)
  (:xSize 0)
  (:roadMapArray nil)
  (Variables
   (roadMapArray xSize ySize))
  (Date
   |Fri Apr 27 02:15:27 1984|)
)
```

Fig-1 属性リストによる表現

２．３　メタクラス，クラス，インスタンス　と　インヘリタンス

オブジェクト指向の概念には、３種のオブジェクトが存在する。　メタクラスはクラスを生成し、クラスはインスタンスを生成することができる。　この結果それぞれのオブジェクト間には親子関係ができ、子は親に従うという性質が生れる。つまり、インスタンスは、クラスで定義されているメソッドに対応するメッセージを受けとることができ、クラスはメタクラスで定義されているメソッドに対応するメッセージを受けとることができる。

Smalltalkでは、クラスを定義することが、クラスとメタクラスの両方を記述することであったが、ＬＯＯＰＳではこれらを分けていて、クラス Class のサブクラスとしてクラスを作成する場合にメタクラスを作成する事を意味し、その他の場合は、クラスを作成する事を意味する。　このためクラス定義の時にメタクラスを指定するという特徴を持つ。Fig-2 にクラスCar の定義例を示す。

```
[DEFCLASS  Car
        (MetaClass        Class)
        (Supers           Object)
        (ClassVariables)
        (InstanceVariables
                (time             0)
                (speed            0)
                (map              nil)
                (direction        0)
                (xPosition        0)
                (yPosition        0)
                (status           nil))
        (Methods
                (SetMap           (map)
                                  Car.SetMap)
                (Initialize       (xPos yPos direction)
                                  Car.Initialize)
                (Move             (map)
                                  Car.Move))
```

Fig-2　　クラス定義の例

この様な３種のオブジェクト間の親子関係の他に、インヘリタンスと呼ばれているクラス間の関係がある。これは、ＡＩの分野から入ってきたもので、スーパークラスからサブクラスへ概念を相続させるというものである。　相続するのは、

ｉ）　メソッド
ii）　インスタンス・バリアブル　（オブジェクトの構造に影響）
iii）　クラス・バリアブル　　（オブジェクトからの参照権）

がある。　インヘリタンスの概念も、Smalltalk では１つの相続しか許されていないが、ＬＯＯＰＳではマルチプル・インヘリタンスの機能もサポートしている。　マルチプル・インヘリタンスで問題になるのは、メソッド名、インスタンス・バリアブル名、クラス・バリアブル名などが重なったときであるが、これらはスーパークラス・チェインがどんなに複雑でも、一意に決定できれば問題はない。　実際には、これは Depth-First Left-to-Rightの縦型探索により解決されている。

２．４　メソッド定義と実行のメカニズム

Fig-2 で示したクラス定義でのMethods には、メッセージ・セレクタと対応して実行するメソッドの対が定義されている。　メソッドは Lisp の関数として定義されるが、この例をFig-3 に示す。　メソッド定義におい

```
[DEFMETHOD  RoadMap  Initialize  (self xSize ySize)
                              ;        initialize receiver object.
                              ;-------------------------------------
        (prog ()
              ( @xSize          <-  xSize)
              ( @ySize          <-  ySize)
              ( @roadMapArray   <-  $Array New: xSize ySize)

              (do ((i 1 (addl i)))
                  ((equal i (addl xSize)))
                  (do ((j 1 (addl j)))
                      ((equal j (addl ySize)))
                      (<- @roadMapArray AtPut: i j (<- $RoadPiece New))
                      (<- (<- @roadMapArray At: i j) SetInclude 'Nothing)
                      (<- (<- @roadMapArray At: i j) SetPosition i j)))

              ( return self ))
```

Fig-3　　メソッド定義の例

て重要なのは、関数の第一引数にレシーバ・オブジェクトが入ってくることである。 この機構により Smalltalkで擬変数と呼ばれていた self と同様に、このレシーバ・オブジェクトを変数 self で受けとる事ができる。(Fig-3 参照)

さらに、このメソッドではレシーバ・オブジェクト内のインスタンス・バリアブル又はクラス・バリアブルを参照する事ができなければならない。 ＬＯＯＰＳのシンタックスでは、インスタンス・バリアブルの参照には ”@”を、クラス・バリアブルの参照には ”@@”をつけて参照する事ができる。 これらは、Lisp のマクロ関数を呼びだし、先のレシーバ・オブジェクトが割り当てられている self を使って属性リスト内の変数に対応する属性値をアクセスする事ができる。 このとき、インスタンス・バリアブル又はクラス・バリアブルの参照が値をフェッチするかセットするかはシンタックス上決定でき、実際に値を参照するときに 後に述べるアクティブ・バリューとよばれるデータ指向的な動作を引き起こすことができる。

２．５ ネームド・オブジェクト と ネーム・テーブル

オブジェクトの構造を示し、Lisp の内部でどの様にオブジェクトを作るかを示した。(２．１) このオジェクトは、すべてシステムで作られたユニーク・ネームにより参照される。 プログラム中では、このユニーク・ネームの他にプログラマが名前をつけて参照したい場合がある。 例えば、クラスなどはユニーク・ネームでなくクラス名で参照したい。 このように名前を付けられたオブジェクトをネームド・オブジェクトと呼び次に説明するネーム・テーブルの機能を使用してネームによるオブジェクトの参照を可能にしている。

ネーム・テーブルは Fig-4に示す Dictionary と呼ばれる構造をしている。

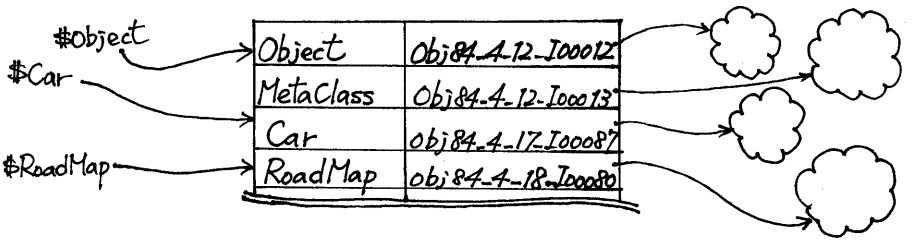

Fig-4 ネーム・テーブルの構造

通常オブジェクトはポインタでさされ、そのポインタはテンポラリ・バリアブルやインスタンス・バリアブルに割りあてられる。 名前でオブジェクトを直接参照したい場合は、”＄”をつける。 名前で参照する場合と変数に割当てられている場合の表現の対応を Fig-5 に示す。 ＬＯＯＰＳには、ここで述べたネーム・テーブルが２種類存在していて、これらにより次に述べるシステム環境をサポートしている。

	通常のオブジェクト	ネームド・オブジェクト
表現 ポインター 実体	変数ｘ (PTR Obj84-4-27I00024) Obj84-4-27I00024	＄MetaClass ＃＄MetaClass Obj84-4-27I00018

Fig-5 ネームド・オブジェクトの表現方法

２．６ システム環境

エキスパート・システムのように高度で複雑なシステムを複数のプログラマーで開発するために、開発環境が考慮されている。 ＬＯＯＰＳでは生成されたオブジェクトを格納するのに知識ベースを用意している。 この知識ベースはそれ自体オブジェクトとしてとらえられていて、バージョン管理や、アップデイトがしやすい様に階層構造をしている。 さらにプログラマーは知識ベース内のある部分を環境というオブジェクトで認識することができる。

実際にはこの環境オブジェクトはローカル・ネーム・テーブルと関係している。 環境オブジェクトをオープンしたりクローズしたりする事により参照できるオブジェクトを変更することもできる。 ネームド・オブジェクトはローカル・ネーム・テーブル、グローバル・ネーム・テーブルの順に検索される。 ＬＯＯＰＳでは環境オブジェクトを複数オープンする事が出来るが常に１つのローカル・ネーム・テーブルのみが有効である。

3　データ指向の概念と機能

3．1　データ指向のモデル

ここで述べるデータ指向はアクセス指向、アクティブ・バリューなどいろいろな名称で呼ばれている。　アクティブ・バリューはモデル的には Fig-6に示される様に実際のオブジェクトとの間に入ったオブジェクトの様なもので表わすことができる。

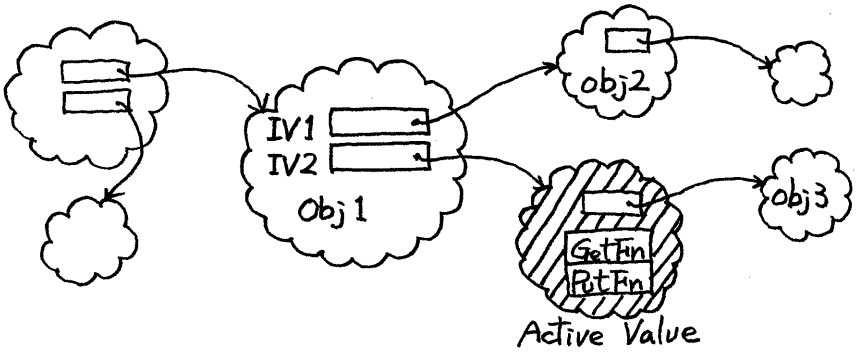

Fig-6　アクティブ・バリューのモデル図

アクティブ・バリュー内には２つの関数 GetFnと PutFnがある。　この図ではObj1内のインスタンス・バリアブルIV1 にはObj2へのポインターが格納されている。　一方、IV2 にはObj3を参照する為にアクティブ・バリューへのポインターが格納されている。　実行時，Obj1のメソッドでIV2 を参照する際に、アクティブ・バリュー内の関数がデーモンとして起動される。　これらはメソッドでIV2 へ値(オブジェクト) をセットしようとする時その値をアクティブ・バリュー内の関数PutFn へ渡しその結果をIV2 の値として格納する。　一方 IV2 より値をフェッチするときは Obj3 をアクティブ・バリュー内の関数GetFn へ渡し実行結果をIV2 の値として受けとる。　アクティブ・バリューは、これを使うプログラマーからは直接見えないことから、デーモン・プロセスとしての性格が強い。

3．2　アクティブ・バリューの定義

Fig-7 にアクティブ・バリューの定義と関数定義の例を示す。この例では、アクティブ・バリューはネスト構造をしている。関数の動く順序は値をセットする時はPutFn1,PutFn2 でフェッチする時はGetFn2,GetFn1 の順である。

```
[DEFCLASS Signal
        (MetaClass  Class)
        (Supers  Object)
        (ClassVariables)
        (InstanceVariables
              (color  #((nil GetFn2 PutFn2) GetFn1 PutFn1)))
        (Methods
              (ChangeColor     (self)
                               Signal.ChangeColor)
```

Fig-7　アクティブ・バリューの定義例

4　ルール指向

4．1　プロダクション・ルール

エキスパート・システム構築を目的としているＬＯＯＰＳは、ＡＩで論じられているフレームをオブジェクトとして実現し、さらにプロダクション・ルールをとり入れる機構を実現した。　このプロダクション・ルールを使用するシステムの多くは、一般にルールを一次元的にもっておりルールが増えすぎると速度が遅くなったり、ルール追加による副作用が生じるという欠点があった。　ＬＯＯＰＳではこのルールをまとめ（ルール・セット）、オブジェクト内のメソッドとして配置することにより、大きくなりすぎるルールを分散させ階層構造をつくる事が出来る。

この様な機能は、新しいルールの追加や削除による副作用の及ぼす影響を小さくすることができ、すべてのルールをサーチしなくてもよい、さらにはルールの体系化ができるなどの利点をもつ。　ルールの起動されるタイミングは次の３通りの場合が考えられる。

ｉ）　オブジェクトのメソッドにルール・セットを付加する。
ii）　アクティブ・バリューの関数としてセットする。
iii）　Lispのプログラムから呼びだす。

4．2　ルール・セット

ルールを起動するタイミングは、上の３通りの場合により異なるが、いずれの場合でもコンパイラにより Lisp の関数に変換される。　ルールの実行制御は、DO形式と ALL形式があり、DO形式はルール・セットの1 回のパスで１つのルールを実行し、 ALL形式は１回のパスで成立つルールをすべて実行するものである。　また動かすルールの回数により　１と ALLがあり、場合分けすると　DO1 、DOALL, WHILE1, WHILEALL となる。

これらはプロダクション・ルールを調べる時にコントロール用のフラグを同時にチェックすることで解決出来る。　ルール指向プログラミングに関する機能は、この他にもあるが、Fig-8 に簡単なルール・セットの定義例を示す。

```
[DEFRULE  CheckCondition

        (WorkSpace              Car)
        (ControlStructure       Dol)
        (TempVariables
                (work   nil))
        (ProductionRule

                [IF  (@direction   =   nil)
                     THEN
                     (<- (<-  self Direction) PrintOut)]

                [IF  (@status   =   'Stop)
                     THEN
                     (<-  self Move)]
```

Fig-8　　ルール・セットの定義例

5　実装について

我々はＬＯＯＰＳのマニュアルに基づき VAX-11 / 750 UNIX 4.2bsd 上の Franz-Lisp を使用して現在実装を行なっている。

5．1　マクロ関数の利用

ＬＯＯＰＳのマニュアルに記載されているシンタックスをLisp上に表現できるようにするのに Read-Macro と呼ばれているマクロ関数を使用した。　これは、 Lisp 外のファイルから Lisp 内へプログラム又はデータを取り込む際に指定した１つの文字に対しマクロ関数を起動できるというものである。具体的にはインスタンス・バリアブルやクラス・バリアブル参照に関係のある”@”、ネームド・オブジェクトで述べた”$”、”#”さらに、オブジェクトの割あてやメッセージ・パッシングに関係のある”<－”などである。

この機能を利用して外部ファイル上にエディタを使用してつくられたＬＯＯＰＳのシンタックスによるメソッドは、Lispで実行可能なシンタックスに変更されながら取り込まれる。

例えば、”<－”に関する Read-Macro では、

```
1)    (A   <-   B)              B をＡに代入する。
2)    (A   <-   B   C   D)      オブジェクトＢへメッセージＣＤを送り結果をＡへ代入
3)    (@X   <-   A)             インスタンス・バリアブルＸへＡを代入
4)    (<-   A   B   C)          オブジェクトＡへメッセージＢＣを送る
```

というような、４通りの異る意味をもつシンタックスを実現している。実際には外部から取り込まれたメソッドは、実行時又はコンパイル時にさらに変換されて完全な Lisp の関数に落される。

５．２　メッセージ・パッシングの実現

クラス、メタクラス、インスタンスのすべてのオブジェクトは Fig-1に示した様なユニーク・ネーム下の属性リストで表されていてオブジェクトはこのユニーク・ネームへのポインタにより参照されている。　５．１で述べたマクロ関数によりメッセージ・パッシングであると判断されたものは、

(SEND　ReceiverObj　'Selector　ArgList)

のような Lisp 関数に変換される。　この関数 SEND の内部では、レシーバ・オブジェクトのクラス又はメタクラス名を使用して FetchMethodという関数を呼だす。この関数FetchMethod は、渡されたクラス・オブジェクト内のメッセージ・セレクタと関数名よりなる連想リストを検索する。見つからない場合はマルチプル・インヘリタンスによるスーパ・チェインを使用してスーパ・クラス・オブジェクトを再帰的に検索する。　すべてのスーパ・チェインをたどっても見つからない場合はレシーバ・オブジェクトへ MsgNotFoundというメッセージが送られエラー情報が出力される。

５．３　アクティブ・バリューの実現

アクティブ・バリューはモデルで示すと Fig-6に示すようにオブジェクトのように表現されるが実装においてはオブジェクトにせずリストを使用した。　インスタンス・バリアブル類をアクセスするのに変数名に”@”をつけるということは２．４で示したが、使われる場合により次のように２つの意味をもつ、

1)　Y <- @X　の場合
　(@ X) のように変換されて
　(GetValue self X) として実行される。

2)　@X <- Y　の場合
　(@<- self X Y)　のように変換されて
　(PutValue self X Y) として実行される。

これら GetValue 、PutValueの関数がまず self で入ってくるオブジェクトのポインターの先の属性リスト内よりインスタンス・バリアブルの値を取り出す。　次にその値がアクティブ・バリュー・リストであるかを調べ、アクティブ・バリュー・リストである時は関数名をその内部より取り出しアーギュメントをそろえて評価する。　この時アクティブ・バリュー・リストのチェックは再帰的に行なえるようになっているのでネスト状のアクティブ・バリューも実現できる。実行順序は

GetValue　の場合は、格納されていた値をGetFn で評価し結果を返す。
PutValue　の場合は、新しい値をPutFn で評価し結果を格納する。

５．４　環境オブジェクトの構造

Lispでは、システムで使用するハッシュ・テーブルが存在するがＬＯＯＰＳでも生成されたオブジェクトやメソッドはこのテーブルに格納される。　この他に Fig-4, Fig-5 で示したテーブルとしてグローバル・ネーム・テーブルが存在する。　一般にクラス、メタクラス・オブジェクトの名前と知識ベースなどの環境に関するインスタンス・オブジェクトの名前がこのテーブルに登録される。　グローバル・ネーム・テーブルは、システム立ち上げ以後どのような状況でも参照可能である。

さらに、ユーザがクラスEnvironment のインスタンスをオープンした時その環境に対応するローカル・ネーム・テーブルが新しく設定され、そのテーブル上にある名前付きの新しいオブジェクトを参照することが出来る。　このテーブル上に存在するオブジェクトは、環境がオープンされた時、対応する外部ファイル（知識ベース）からLisp内へ呼び込まれる。　また、環境をクローズする際には、ローカル・ネーム・テーブル上にあるオブジェクトおよびその内部からポインタでさされるすべてのオブジェクトは外部ファイルへ出される。　従って、ユーザがオブジェクトの名前を付けたい場合は、環境を指定して、そのローカル・ネーム・テーブル又はグローバル・ネーム・テーブルに設定することになる。

６　実行例と考察

６．１　プログラム例

簡単なプログラムの例を示す。　これは与えられた地図（ RoadMapクラスのインスタンス）上を自動車（ Carクラスのインスタンス）が走るというものである。　ここでは３つのクラス Car、RoadMap 、RoadPiece を定義した。　各々のクラスのもつメソッドは、この実行例の中のHasMethod メッセージの結果として示されている。　ここで使用する地図（RoadMap)は RoadPieceオブジェクトの集合体として作られている。　すべてのプログラムは３つのクラスに分散配置されていて、オブジェクトの一部として機能する。　また、プログラムの構造はメッセージ・パッシングにより動的にメソッドの階層として決定される。　この例をオブジェクトのモデルとして見ると Carクラスのインスタンスは、インスタンス・バリアブルとして自分の参照すべきRoadMap オブジェクトへのポインタをもっている。さらにRoadMap のインスタンスは内部構造としてArray のインスタンスをもっており、その要素１つ１つにRoadPiece のインスタンスを保持している。

```
-> (<- $Car HasMethod)クラス Car, RoadMap, RoadPieceで定義されているメソッドのリストを返す。
(SetMap Initialize Move Stop)

-> (<- $RoadMap HasMethod)
(Direction MakeMap Initialize SetDirection PrintMap)

-> (<- $RoadPiece HasMethod)
(Direction SetDirection Include SetInclude Position SetPosition)

-> (<- $Environment Old: 'Simulation1)        過去に作成した環境 Simulation1を呼びだす。
Name <Simulation1> is set in Global Name Table .
(PTR Obj_I00030)

-> (<- $Simulation1 Open)                     環境 Simulation1をオープンする。
Name <Map1> is set in Simulation1 Environment
(|#$| Simulation1)

-> (x <- $Car New)          クラス Carのインスタンスを生成し、変数 Xに割りあてる。
(PTR Obj_I00032)

-> (<- x SetName 'MyCar)    変数Xで参照されるオブジェクトに名前をつける。
Name <MyCar> is set in Simulation1 Environment
(PTR Obj_I00032)

-> (<- $MyCar SetMap $Map1)                   自動車が参照すべき地図を割りあてる。
(|#$| MyCar)

-> (<- $MyCar Initialize 1 1 nil)             自動車の状態を設定する。
(|#$| MyCar)

-> (<- $Map1 PrintMap)                        地図の状態を表示する。

 ***
** *
* ***
*** *
@ ***

(|#$| Map1)
-> (<- $MyCar Move)                           自動車へ Move というメッセージを送る。
(DEFINST MyCar
        (CreateDate |Sat May  5 00:12:13 1984|)
        (OwnerClass Car)                                    ->(<- $Map1 PrintMap)
        (InstanceVariables (status Move)
                           (speed 1)                         ***
                           (yPosition 2)                    ** *
                           (xPosition 1)                    * ***
                           (direction north)                @** *
                           (time 0)                         * ***
                           (map (|#$| Map1))))
                                                            (|#$| Map1)
(|#$| MyCar)
```

Fig-9　　プログラムの実行例

6 .2　オブジェクト指向プログラミングについて

先に示したこのようなプログラム（６．１）を他の手続型言語でプログラミングする場合には、１つあるいは複数のプロシジャを用いてつくることになる。　それは外部に共有データとして地図、自動車　に関する物を用意し、他に用意したプロシジャで、それぞれのデータを比較して判断を加え状態を更新するというものになる。　このようにプロシジャでプログラミングした場合は、プロシジャによる階層が固定され、１つのまとめられた構造を決定することになる。　さらに、各々のプロシジャから外部データ参照が行なわれるので、あるプロシジャで発生したデータの誤操作は、他のプロシジャーへ副作用を与えやすい。

このようなプロシジャ型のプログラミング言語に対し、オブジェクト指向型の言語では、プロシジャはメソッドとしてクラスにまとめられ、機能的に配置され、データも意味をもつグループに分けられクラスとして定義される。　こうしてできたクラスはプログラムとして独立したモジュールとなる。　プログラムの構造はメッセージ・パッシングにより動的に生じたメソッドの階層であり、具体的には他のオブジェクトのメソッド　又は、スーパ・チェイン上のクラスのメソッドによる階層である。　またオブジェクトが他のオブジェクトを内部構造としてもつことでデータとしての構造もきまる。　この例では、自動車へ Move というメッセージを送ると自動車は、自分の参照すべき地図へメッセージを送りどの方向へ進めるかを問合せる。　地図は自分の道の要素により道のある方向を返す。つまり、プログラムは、地図と自動車のインターフェイスの部分と、自動車と自動車を動かす側とのインターフェイスの部分に分けることができる。　また、個々の情報は各々のオブジェクトで自己管理しているので、プログラマは自分の記述しているクラスのことに神経を集中できる。

以上のように、プログラミングにオブジェクト指向の概念を取り入れることは、プログラマにとっては、必然的にプログラムをメソッドとして、各オブジェクトへまとめなければならず、これが結果的に問題に意識を集中してプログラムできるという効果を生む。また、完成したクラス（メソッドを含む）は、それ自体有効なプログラムのパッケージ化されたものと考えることができ、他のプログラムで使用する場合にもメッセージによるイ

ンターフェイスだけを意識すればよいことになる。さらに、人間の思考においても物体を中心に概念を抽象化することは比較的楽である、また新しくオブジェクトを定義する時も従来のオブジェクトの積み重ねで考えることができる。　このようにして作られたプログラムは、メッセージ・パッシングにより階層化された構造を生み、プログラムも有効に配置されたことになる。

6．3　データ指向、ルール指向プログラミングについて

オブジェクト指向によってプログラムの抽象化作業が支援される事を述べてきたが、データ指向の概念であるアクティブ・バリューも同様に効果がある。　これは、データに対するデバック、アクセスされた事に対するテストやデバック、値の変更による表示画面の書き変えなどをデータをアクセスするだけで実現させる事ができる。　従ってデバックの終ったデータのチェック・ルーチンなどをアクティブ・バリューの内部へ閉じ込める事で、プログラマはそれらの手続を意識しないですみ、他のプログラミングに専念できる。　この様なデータ指向プログラミングは、データが主体であるということと、データ参照に手続を用意している点でデータと手続を一体化させたオブジェクト指向プログラミングと近い関係にある。

この他にＬＯＯＰＳでは、データやオブジェクト主体のプログラミング・スタイルとは異なるルール指向プログラミングが用意されている。　前者が物体主体であるのに対し、後者は物体間の関係が強調されている。従来のエキスパート・システムの多くは、ルールの追加や削除が考慮されていたが、ＬＯＯＰＳでは追加や削除は考えておらず、そのかわりルールの階層化やグループ化を実現している。　この場合の階層化はオブジェクトのメソッドの階層化と等価であると考えてよい。　また、１つのルール・セットで調べられる事は、そのオブジェクトから参照できるオブジェクトの状態のみであるという特徴をもつ。

7　まとめ

オブジェクト指向を中心に、いくつかのプログラミング・スタイルを統合化したＬＯＯＰＳについて述べてきた。　ＡＩなど高度なアプリケーションに使用されてきた Lisp をベースにしているので、 Lisp の関数内でメッセージ・パッシングを記述することができ、従来のシステムを生かしながらプログラムをクラスとしてパッケージ化することができる。　さらにシステムを、データ（オブジェクト）の構造に意識を集中して構築できる点で有効であるように思われる。

ＬＯＯＰＳは多くの新しい概念を提案している点で今後さらに注目を集めるシステムになるであろう。

REFERENCES

[Shapiro & Takeuchi 83] E. Y. Sapiro, Akikazu Takeuchi.
"Object Oriented Programming in Concurrent Prolog", New Generation Computing, Vol. 1, No. 1, 1983.

[ICOT 83] K.Furukawa, A.Takeuchi, S.Kunifuji.
"MANDARA : A Concurrent Prolog Based Knowledge Programming Language/System", Knowledge Engineering and AI 23-Nov, 1983.

[Bobrow 76] Daniel G. Bobrow, Terry Winograd.
"An Overview of KRL, a Knowledge Representation Language", Xerox Palo Alto Research Center, csl-76-4 July 4, 1976.

[Bobrow & Stefik 83] Daniel G. Bobrow, Mark J. Stefik.
"The Loops Manual (Preliminary Version)", Knowledge-based VLSI Design Group Technical Memo, kb-vlsi-81-13 (working paper), January, 1983.

[Bobrow & Stefik & Bell 83] D. G. Bobrow, M. j. Stefik and Alan g. Bell.
"Rule-Oriented Programming in LOOPS (Preliminary Version)", Knowledge-based VLSI Design Group Memo, kb-vlsi-82-22 (working paper), January, 1983.

[Goldberg & Robson 83] Adele Goldberg, David Robson.
"SMALLTALK-80 The Language and its Implementation", Addison-Wisley 1983.

[Smalltalk 81] byte, vol. 6, no. 6 McGrawHill, August, 1981.

[Winston 77] Patrik H. Winston.
"Artificial Intelligence", Addison-Wesley, Reading, Massachusetts 1977.