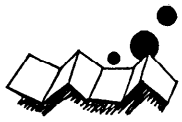
解　説

Lisp上のオブジェクト指向プログラミング†



梅村恭司†† 大里延康††

1. はじめに

現在のオブジェクト指向の考え方は Smalltalk によってその骨格が固まり，Lisp によって強化されてきたといえるだろう．Lisp によるオブジェクト指向のほぼ中心となってきたのは MIT 系の Lisp マシン上の Lisp である Zetalisp にオブジェクト指向のプログラミング機能として提供された Flavor system である．Flavor system では，Smalltalk でいうクラスに対応する概念を，あえて Flavor と呼ぶ．Smalltalk のクラスの継承関係は，概念の抽象度という観点に重点を置いた階層構造をなしていた．これに対し，Flavor system における Flavor の継承関係は，より単純な部品とそれを組み合わせて作られる，より複雑なものとの関係という観点に重点が置かれている．Flavor system の特徴は，いわゆる多重継承の機能によって徹底したモジュール化の能力を追求している点にある．Flavor system に続く多くのオブジェクト指向言語は，こぞってこの多重継承を取り入れている．

一方，AI の研究の隆盛とともに Lisp の標準化の機運が高まり，Guy Steele をはじめ米国の研究者を中心として，Common Lisp の提案が行われた．現在では，Common Lisp が Lisp の事実上の標準となっていることは衆知のとおりである．さらに，この Common Lisp にオブジェクト指向の機能を追加するための討論が精力的に行われ，いくつかの提案がなされた中で，Xerox 社が提出した Common Loops (Common Lisp Object-Oriented Programming System) が最有力なものとして検討されてきた．この案は，現在 CLOS (Common Lisp Object System) という後継案となって引き続き検討されている．これに対し，MIT 系の Lisp マシンを商品化した Symbolics 社も，これまでの Flavor system の使用実績を背景として，新 Flavors を Common Lisp に対応するものとして提案した．本稿では，Lisp におけるオブジェクト指向として重要なものと考えられる新旧の Flavors と CLOS を中心に解説を試みる．そして，オブジェクト指向の概念と Lisp のもつ機能とが親和性の高いものであることを説明する．

なお，本稿ではオブジェクト指向に関する一般的な概念や用語についてはほぼ既知のものとし，特に網羅的な解説はしないこととする．

† Object oriented programming in Lisp by Kyoji UMEMURA and Nobuyasu OSATO (Nippon Telegraph and Telephone corporation).

†† NTT 電気通信研究所

2. Flavor system

Flavor system は，MIT の D. Weinreb, D. Moon らのグループによって Zetalisp 上に実現された．このシステムは，Lisp におけるオブジェクト指向の先駆けであると同時に，多重継承とメソッド結合という概念を与え，かつそれによって実用的なシステムを開発してその有用性を示したという点で，きわめて重要なシステムである．本章では，その特徴を概観し，Common Lisp に対応するために提案された新しい Flavor system (新 Flavors と呼ぶ) にも触れながらその意義について述べる．なお，Flavor というのは Flavor system 独特の用語であるが，以下では，特に混乱の可能性がないかぎり Flavor とクラスという用語を適宜取りまぜて用いる．

2.1 オブジェクト (object)

Flavor system では，Flavor と呼ばれるユーザ定義データ型に属するデータ (これを Flavor のインスタンスと呼ぶ) をオブジェクトとみなす．Flavor そのものは DEFFLAVOR というマクロで定義される．Smalltalk-80 などと異なり，Flavor system では，整数やリストなど通常の Lisp のデータ型のデータはオブジェクト指向の意味でのオブジェクトとはみなされない．Flavor のインスタンスであるオブジェクトに対して，メソッドと呼ばれる手続きを定義することができる．この手続きは，オブジェクトに対して「メッセージ」を送ることによって起動される．

2.2 継承 (inheritance)

一般にオブジェクト指向プログラミングでは，オブジェクトの構造や動作などプログラムの扱う概念になんらかの階層性があるとき，上位の概念を用いて，それより下位の概念を定義する，いわゆる「継承」の機能が提供される．継承の機能を用いることによってモジュール性の高いプログラムを作成することができ，プログラムの保守性，読解性が向上するとともに，プログラムの部品化を図ることができる．

クラスの階層構造について述べるとき，より上位のクラスをスーパクラス，より下位のクラスをサブクラスと呼ぶ．クラスの階層は型の階層と考えればよい．たとえば，整数の上位概念として数というものがあると考えると，数のクラスは整数のクラスのスーパクラスであるということになる．数というものがもつ性質は，整数もまたもっている．すなわち，整数のクラス（型）は数のクラス（型）の性質を継承している．

なお，継承によるモジュール化は，オブジェクト間の情報隠蔽によるモジュール化とは意味が異なることに注意されたい．継承は，スーパクラスの側で共通の性質を一元管理するという意味でのモジュール化である．継承される情報は，サブクラス側からみて必ずしも隠蔽されてはいない．

2.3 多重継承 (multiple inheritance)

多重継承というのは，スーパクラスを複数個もつことが許されるようなクラス間の継承方法のことである．なぜ多重継承が必要となるのかを説明するために，まず，一個のスーパクラスしか許さないような継承方法（これを単純継承と呼ぶことにする）で生ずる問題点を明らかにする．

まず，クラスAがあるとしよう．このAのサブクラスとして，Aがもともともっているインスタンス変数のほかにインスタンス変数 b をももつクラス A-with-b を定義する．クラス A-with-b には，b に関係する処理をする新たなメソッド α を定義するとする．一方，Aにもう一つ別のインスタンス変数 c を加えたクラス A-with-c も必要であるとする．このクラスに対しても，A-with-b の場合と同様に c に関する処理をするためにメソッド β を定義する．これら b をもつという性質と c をもつという性質は互いに独立であるとする．

さてここで，インスタンス変数 b と c の両方をもつようなAのサブクラス A-with-bc を定義する必要が生じたとする．すると，単純継承のシステムでは，クラス A-with-bc を定義するには，それを A-with-b のサブクラスとして定義してインスタンス変数 c を追加し，A-with-c にあるものと全く同じメソッド β を追加するか，逆に，A-with-c のサブクラスとして A-with-bc を定義し，同様に A-with-b のインスタンス変数 b とメソッド α を追加するかの二つの方法がある．しかし，b をもつという性質と c をもつという性質は互いに独立であるにもかかわらず，これらのいずれの方法をとっても，それら二つの性質の間にあたかも概念上の上下関係があるかのようになる．のみならず，いずれの方法でも，同じインスタンス変数とメソッドが二カ所に存在することになってしまう．たとえば，前者の方法では，クラス A-with-c にもクラス A-with-bc にも全く同じインスタンス変数 c とメソッド β とを置かざるをえない．したがって，たとえば β になんらかの変更を加える必要が生ずると，A-with-c の β と A-with-bc の β の両方を修正しなければならなくなる．すなわち，β は，このプログラムのモジュール性を損なうものとなる．

この問題は，スーパクラスを複数個許す多重継承を採用することによって解決する．上の例では，クラスAのほかにクラスBとクラスCの二つのクラスを用意し，A-with-b はAとBの二つをスーパクラスとしてもつクラスとして，A-with-c はAとCの二つをスーパクラスとしてもつクラスとして，さらに A-with-bc はAとBとCの三つをスーパクラスとしてもつクラスとして定義すればよい．こうすれば，BとCに上下の関係がないので自然にクラスの階層を記述でき，かつ全く同じインスタンス変数やメソッドが異なるクラスに存在するという事態は生じない．なお，この方法ではBやCはAとは独立なのでBやCのメソッドからのAの情報の参照は通常できないが，それらのメソッドが継承によって下位のクラスでAとともに使用されることを前提とする場合には，適当な宣言を入れればそれが可能なようになっている．

このように，プログラムを局在化させ，手続きのモジュール化を図る手段として「継承」というものを考えるならば，多重継承は不可欠のものとなる．ここで注意すべきは，上記の例でいうならば，A-with-bc のスーパクラスであるBとCとの間にはオブジェクトとしての抽象度という意味の階層性はないということである．むしろ，A-with-bc からみるとBとCは，A-with-bc を構成する「部品」と考えるのが自然である．これに対し，単純継承では，クラス間の階層は，各ク

ラスが表すオブジェクトの抽象度を反映するという性質が強く出る．たとえば，ヒトというクラスは動物というスーパクラスをもち，動物というクラスは生物というスーパクラスをもつ，というぐあいである．多重継承では，スーパクラス側に近いクラスほど単純で基本的な性質をもつという意味での階層性は残っているが，抽象度という意味での階層性はいわばクラス階層の脊骨にあたるところに認められるだけである．上の例でいえば，Aと A-with-*bc* の間には，一般的な性質をもつAからそれを特化した A-with-*bc* に階層性があるのみで，AとB，Bと A-with-*bc* などの間に階層性はないと考えるのが自然であろう．

このように，Flavor system ではスーパクラスに対して上位の抽象概念という見方ではなく，部品 (component) という見方をする．だから，新たになんらかの Flavor を作るときは，より単純な Flavor を混合して作るという考え方をする．この，Flavor の混合が，上で述べた継承に対応する．継承の対象となるのはインスタンス変数とメソッドである．Flavor の特色は，多重継承による徹底したモジュール化である．Flavor 間で共通する性質はくくり出されて新たな Flavor として定義され，それを継承する．これによって，同じものは一カ所にしか存在しないようにするのである．

2.4 メッセージ送信

オブジェクトに対して定義されたメソッドの起動は，そのオブジェクトにメッセージを送ることによって行われる．メッセージを送るには，メッセージ送信のための式を評価する．新 Flavors ではメッセージ送信のシンタクスとして CLOS とよく似たジェネリック関数 (generic function) を採用している．ジェネリック関数によるメッセージ送信式は，一般の関数呼び出しと見かけ上同じであるが，新 Flavors では，CLOS の場合と異なり，第一引数がこのメッセージの受け手（レシーバ）とみなされる，すなわちメソッドは，あくまである特定のクラスに属するという点に注意しておく．

メッセージ送信によるアルゴリズムは，扱う対象が何であろうと，メッセージの表す一般的な操作が可能なものという認識だけで設計でき，それを汎用アルゴリズムとして「部品化」することができる．

2.5 メソッド結合

メソッドの継承で問題になるのは，継承関係にある複数のクラスの中に同じ名前のものがあったらそれらのメソッドはどのように実行されるか，ということである．Flavor system では，部品 Flavor から継承してきたメソッドは，一定の規則に従って組み合わせを行い，新しいメソッドとして動作させる．これをメソッド結合 (method combination) と呼ぶ．メソッド結合を行うために，継承する Flavor には順序づけが行われる．また，メソッドの方には呼び出される順序や値の返しかたなどを指定するため，primary や before などという型 (type) を与える．ユーザは，Flavor の順序とメソッドの型を用い，Flavor の定義式の中で，メソッドの結合方法を宣言する．デフォルトでは daemon 型とよばれる結合方法がとられる．daemon 型のメソッド結合では，before 型のメソッド primary 型のメソッド，after 型のメソッドの順に呼び出される．すなわち，primary メソッドの前後に前処理と後処理を行う，という手続きになる．

新 Flavors では，新たなメソッド結合をユーザが定義できるような道具だてが整備された．旧 Flavors のメソッド結合のさまざまなバリエーションは，複雑すぎて分かりにくく，もっとも単純な daemon しかほとんど使われないというのが現実であったと思われる．これには，メソッド結合を支援する十分な道具が整っていなかったこともあろう．実際，旧 Flavors の初期のマニュアルには，ユーザが自分の好きなようにメソッド結合をすることができるが，詳しくはコードを見よなどということが書かれていた．

メソッド結合の概念は Flavor の大きな特徴であり，非常に便利な反面，自分のプログラムがどのように自動的に加工されたかを常に把握しておくことを強いられ，よい支援ツールなしでは使いこなすのに骨が折れるというのが実際のところであろう．つまり，メソッド結合というのは，一種の自動手続き加工であるから，ユーザは，それをコントロールするために一段高いレベルの管理をしなければならないのである．

2.6 Flavor system の問題点

本章では Flavor system の提供する特徴的な機能を概観した．Flavor は，Lisp のオブジェクト指向において多重継承とメソッド結合という機能を組み込んでモジュール性を高めたが，多重継承における Flavor の順序づけやメソッド結合を駆使したプログラミングの困難さなどの問題を新たに導入したともいえる．さらに，Flavor 間の構造の設計は，多重継承によって困難さが増大した．これらの問題は，CLOS に受け継がれ，部分的には解決されているが，なお十分ではない．

3. Common Lisp Object System (CLOS)

CLOS は Common Lisp にオブジェクト指向の機能を取り入れるために検討された言語仕様である．オブジェクト指向の機能を取り入れたとはいえ，既存のオブジェクト指向のもつデータとアルゴリズムの複合体の考え方と対立するものを含む．基本哲学は関数呼び出しにあり，オブジェクト指向の分野で議論されたことを関数呼び出しの枠に組み込んだものとなっている．

CLOS は現時点では規格化されたものではないが，Common Lisp の設計に携わった人物により作成され，Lisp への影響が大きいと予想できる．CLOS は Gregor Kiczales の作成した Common Loops から発展したものである．CLOS のドキュメントが Daniel G. Bobrow, Linda G. Demichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, David A. Moon により著されている．このうち，R. P. Gabriel, D. A. Moon は Common Lisp 取りまとめの中心人物である．さらに，寄与している人物リストのなかに，D. L. Weinreb も載っている．そして，CLOS は Flavor system に影響を与え，新 Flavors は CLOS に近いものに変化した．

CLOS の設計目標はオブジェクト指向の基本機能を明らかにして，さまざまなオブジェクト指向のシステムの実際的なベースを実現することである．すなわち，CLOS では多重継承の考え方を整理し，複数の上位クラスの処理を曖昧さのないものにした．さらにクラスの定義の変更に対する振舞いを定義した．さらに，Common Lisp 本来のデータ型と定義されたクラスとの関係についても考慮している．そして，CLOS を導入しても Lisp としての速度が低下しないようにオブジェクト指向の機能に一定の制限をつけることも提案している．CLOS は旧 Flavors システムを一般化するとともに不明確な部分を明確にしたものであり，現在実現されているいろいろなオブジェクト指向のシステムを，CLOS の枠組みの中で実現できるように統合したものといえる．

3.1 ジェネリック関数 (generic function)

ジェネリック関数とは引数のクラスによって呼び出される定義体が動的に決まるような関数のことである．Smalltalk でいうメッセージセレクタ名に相当するものがジェネリック関数の関数名である．通常のオブジェクト指向言語では一つの引数だけに定義体を決定する権利があったのだが，ジェネリック関数の考え方ではすべての引数が定義体の決定に参加できる．たとえば「1+2」という演算に対して，通常のオブジェクト指向の考え方では「1」に対して「+2」というメッセージを送ると考える．ジェネリック関数では「1」と「2」の組に対して「+」というメッセージが送られると考えるのである．今までは，「整数」，「実数」のクラスと同じ数の定義体しか存在しえなかったが，ジェネリック関数では「整数＋整数」，「整数＋実数」，「実数＋整数」，「実数＋実数」のすべての組み合わせに対して別々の定義体が存在しうる．引数によって定義体を別にするかどうかは選択可能なので，定義体は一つのこともあるし，第一引数のクラスだけで選択が行われるようにすることもできる．第一引数で定義体を選択する場合は今までのオブジェクト指向の定義と同一になる．このように，特別の場合として今までのものを含み，それを一般化したことになる．

定義体の選択がすべての引数のクラスに関連するため，実現方法に大きな変更が加わる．定義体の決定が一つの引数でなされる場合には定義体はクラスに属するものとして考え，実行時には呼び出しの名前をもとに定義体を捜す動作が行われた．ジェネリック関数では，定義体がクラスに属するものと考えることができない．定義体はクラスのべき集合に属すものと考えるのである．実現上はべき空間をハッシュを使って実現してしまうとか，呼び出し名に定義体を属させてクラス検査をしながら定義体を選択していくことになる．いずれにしろ，クラスと定義体の結合が疎なものとなるので「データ形式＋アルゴリズム」の複合体としてのオブジェクトというものとはイメージが異なる．クラスのべき集合と操作定義体との対応がクラスと操作定義体との対応より分かりにくいことは，CLOS の利用上の問題点となるおそれがある．CLOS ではメッセージを送る形式は通常の関数呼び出しと同じ形式を使う．ジェネリック関数の「関数」という名の由来は呼び出し形式にある．形式が関数と同じだけではなくて，ジェネリック関数名から関数の定義体を取り出して操作することも通常の関数と同じように可能である．通常のオブジェクト指向の考え方では，定義体はメッセージセレクタではなくてデータクラスに属すと考えるところを，メッセージセレクタに属させるのである．CLOS はオブジェクトシステムと命名されていて，オブジェクト指向の機能を実現している．しかし，基本の哲学は Lisp と同じ関数呼び出しにあり，

オブジェクト指向のメッセージパッシングの考え方と異なっている．ジェネリック関数の名に象徴されるようにCLOSは関数呼び出し指向のシステムである．オブジェクトを取り扱えるがオブジェクト指向のシステムではない．

3.2 CLOSにおける多重継承

多重継承における最大の問題点は，上位のものが共通のさらに上位のものをもつ場合の処理である．旧Flaversでは共通の上位の定義体が二回起動されて，動作がおかしくなるなどの問題があった．CLOSでは，上位のクラスを一元的に並べるリスト・Class Precedence Listをクラス定義時に考えて，このリストに従って継承を行う．多重継承を単純な継承に対応させて処理をするのである．このようなリストを生成できない場合，言いかえれば矛盾した継承の指定によりクラス間の上下関係が決定できない場合には，エラーとする．一元的な順序を導入するので，共通の上位クラスが重複することはない．

Class Precedence Listの決定方法を説明する．図-1のように継承関係が存在するとしよう．Aが(B C)を継承することで A<B, A<C という関係が存在するだけでなく，Aからみて B<C という関係が指定される．このような関係で A, B, C, F の関係は一意に定まる．問題はCとD，DとEの関係である．このような場合，双方の下位のクラスの中で最小の番号をもつクラスを下位と定めることとする．この方法で下位のほうから番号を定めていけば一意の番号が定まる．クラスの上下関係が半順序を保っており，あるクラスの直接の上位のクラスの間に全順序が定まっていればClass Precedence Listが一意に定まる．注意すべきことはAに対しての上位クラスとしての全順序が定まるのであって，一般に全順序が定義できるわけではないことである．処理上は，異なるクラスであれば，それぞれのClass Precedence Listの中に矛盾を含む順序が存在してよい．しかし，このような階層は混乱のもとであり，勧められない．

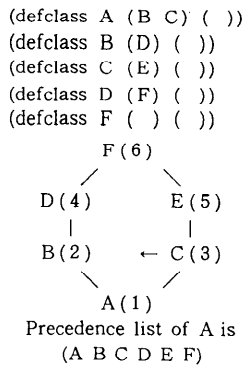


図-1 クラス順序の決定

3.3 クラス定義の変更

オブジェクト指向のシステムの共通の問題として，実行の途中でクラスの定義を変更したときのインスタンスのふるまいがある．旧Flaversなどの従来の多くのオブジェクト指向システムでは，古いクラスのインスタンスに送られたメッセージは古いクラスの定義のまま動作したり，あるいはメッセージに無反応になったり，エラーが起きたりした．クラスの定義の変更に対する動作は決められておらず，会話的にプログラムする場合に大きな障害になっている．CLOSではクラス定義が変更されたら，その結果はすべてのインスタンスに反映されることを要求している．そのクラスが下位のクラスをもてば下位のクラスのインスタンスにも反映される．そして，インスタンスの同一性は保存されなければならない．

クラス定義の変更がクラスの形を変える場合，たとえばスロットを追加したりした場合には自動的にインスタンスの内容を更新するのは不可能である．このため，Class-changedというジェネリック関数が定められている．ユーザは，この関数を用いてクラス再定義によってインスタンスが変更されたときの処理を定義する．Class-changedはクラスの定義が変更されたとき，変更されたクラスのすべてのインスタンスに作用する．第一引数には，もとの情報を得るために，更新しようとしているインスタンスのコピーが渡される．第二引数には，変更の結果を格納するために，もとのインスタンスが渡される．そして，Class-changedの中で，第一引数の情報をもとに第二引数を変更するものである．これを定義しておくことでクラスを変更したときのインスタンスの変更方法を指定できる．重要な点は，過去に生成したオブジェクトを定義変更後もできるだけ利用しようとしていることである．

クラス定義の変更がすべてのインスタンスに及ぶことから，単純に実現するとクラス定義からインスタンスへのリンクが必要であると分かる．すなわち，すべてのインスタンスはクラスから利用される可能性がある．このような状況ではインスタンスはゴミにはなりえず，一度生成されると永久に存在しつづけることになる．これを避けるには，ガーベージコレクタがクラスからインスタンスをたどるポインタを区別して特別な処理をするか，メッセージを送るときに特別な処理

をしてポインタを使わないようにしなければならない．このように，CLOS の仕様には，表面上に現れない部分でシステムの根幹に変更が必要となる場合がある．

3.4 基本データ型の非オブジェクト

CLOS ではオブジェクト指向の機能を関数と同一の形式で呼び出すけれども，普通の意味の関数とジェネリック関数とは厳格に区別する．たとえば「car」という名前でメソッドを定義するのはエラーである．また，すべてのクラスのメタクラスとして standard-class が存在するが，Lisp 本来の型については standard-type-class という別のメタクラスを導入している．メタクラスは計算の枠組を決める作用があるので，メタクラスが異なるということは，オブジェクト指向の枠の外で処理されることを意味している．

これらは，基本データ型の処理にはオブジェクト指向によるオーバヘッドが許されないという配慮の結果である．ここにもオブジェクト指向よりも関数指向である CLOS の側面をみることができる．ただ，CLOS では基本データ型もできるかぎりオブジェクトに近い扱いをするように工夫している．そのために，standard-type-class というものを導入した．旧 Flavors はオブジェクトとデータ型を全く異なるものとしていたが，CLOS は旧 Flavors よりも基本データ型とオブジェクトが統合化されている．

実行効率を考えたオブジェクト指向のシステムでは基本データ型の計算メカニズムとオブジェクトの計算メカニズムが異なっているのが普通である．CLOS はメカニズムが異なっているけれども統一的に使える方法を考え出している．この方法により，オブジェクト指向のオーバヘッドを押さえている．これは，すべてをオブジェクトに統合しすぎて実行速度の向上が難しい Smalltalk の場合と，オブジェクトと基本データ型とが分離して使用しにくい旧 Flavors との場合のうまい妥協点となっている．

3.5 CLOS の問題点

現存するオブジェクト指向で不明であった多重継承やクラス定義の変更に明確な意味と動作を指定したのはオブジェクト指向の分野での議論をふまえてのうえの進展である．旧 Flavors で動作が不明であったり，応用上に障害となっていた部分はかなり改善されている．オブジェクト指向の機能を使わないときには，Lisp としての速度が低下しないことが考慮されている．しかし，オブジェクト指向の処理系として考えたときの効率に問題がある．この文章の範囲では解説しきれないが，詳しく CLOS の実現方法を検討すると，クラス定義のときに処理系が行わなければならないことの多さと，インスタンス変数へのアクセスの効率の低下とが最大のものと判明するはずである．インプリメントを工夫することで効率を向上させるのがこれからの課題である．

効率の良い CLOS 処理システムを構築する場合には仕様の外見からは予測できない工夫が必要である．たとえば Class-changed を効率よく実現するには，すでに Common Lisp の処理系が存在していても，システムの根幹に変更を加えなければならなくなる．Class-changed 以外にも実現のためにトリックが必要なものが存在し，効率のよい CLOS 処理システムを実現するには，核となる部分から処理系を再構築することが不可避であろう．

CLOS はオブジェクト指向の最大の哲学である「データとアルゴリズムの融合」に合致しない哲学をもって設計されている．データのクラスによるアルゴリズムの選択がオブジェクト指向の基本要素であるのは事実であるが，「メッセージが送られて動作が行われる」というイメージが薄れてしまっている．ジェネリック関数の評価はジェネリック関数で記述されたシステムが実現されたとき複数の定義体による分岐が有効に使われるかどうかによる．これが有効に使われなければ定義体の選択のための苦労は無になってしまう．この部分が CLOS と新 Flavors とで意見が分かれているところである．

ジェネリック関数は今までのオブジェクト指向を特別な場合として含んでいるが，複数の引数によるアルゴリズムの選択が人間の認識のパターンに合致するかが考慮されるべきである．while や repeat を特別な場合として含む goto と if が，人間の認識の方法に合致するように while や repeat に進化した経緯に学ぶべきであろう．

4. Lisp とオブジェクト指向との相性

Lisp はオブジェクト指向の言語と相性がよい．まず，変数の型が実行時に決まることがあり，次に領域の確保がやはり実行時に行われることである．さらに，定義を実行時に変更できる．ほとんどのことが実行時になされる．Smalltalk-80 にもこの性質があり，オブジェクト指向のシステムの重要な要素となっている

4.1 変数の型

Fortran や Pascal などでは変数に型があり，一つの変数に「実数」と「整数」を同時に代入することはできなかった．オブジェクト指向言語では変数に型を付加することが難しい．ある変数の保持するデータの型を制限しないことで継承の機能が使え，内部構造の異なるデータが統一的に操作できるのである．Lisp の変数には型がなく，一つの変数には任意のデータが代入できる．この性質のため，オブジェクト指向の機能を入れる場合に，Lisp 本来の機能を変更せずに済んだ．

オブジェクト指向のメソッドの検索は実行時に行うのが普通である．メソッドの検索は実行時の型検査の発展としてみることもできる．型を判定してエラーを検出するのみならず，適切なアルゴリズムを選択するという実質的な計算に還元している．メソッドを選び出すきっかけは，実行時の型検査となる．このため，変数に型がなく，データに型があるのが本質的となる．

4.2 ダイナミックアロケーション

オブジェクトはデータとアルゴリズムの融合体であるが，データの領域を確保する時点が問題である．Fortran などでは実行の始まる前にすべてのデータ領域を見積もって用意することが必要であった．オブジェクト指向の計算，たとえば「複素数」をオブジェクトと定義すると「加算」などの演算のたびに新しい「複素数」を生成することが不可欠となる．インスタンスの変更のみならず似たインスタンスの生成が必要となる場面が多い．このためインスタンスの生成が実行を開始した後に行える性質が必要となる．Lisp においてはあらゆる領域が実行開始の後に取られる．そのためのしくみとしてガーベージコレクションの機構が整備されている．新しいインスタンスを実行時に取ることが必要なオブジェクト指向の機能を実現するのに Lisp 本体の機能を変更する必要がない．

4.3 Smalltalk-80 と Lisp との比較

Smalltalk-80 は Lisp で記述されたシステムを母体に設計された．このため，言語に内包されている基本機能は類似性が高い．このため，Lisp を用いてオブジェクト指向のシステムを記述するのは比較的容易である．しかし，計算実行に至るまでの考え方が異なるために，単純な Lisp でのオブジェクト指向のプログラムは，Lisp に翻訳されるものでなく Lisp により解釈されるものとなっていた．このため，単純な Lisp を用いてオブジェクト指向のシステムを構築した場合にはシステムの効率が問題となる．CLOS にしても，Flavors にしても，関数の実行メカニズムを補強して，効率よく性能のよいオブジェクト指向のシステムをめざしている．

Smalltalk-80 の速度を低下させている最大のものは，オブジェクトとして扱うことが難しいものもオブジェクトとしていることである．たとえば整数などもオブジェクトとして扱うために，プリミティブへの制御の移行が間接的になる．これは，初期の Lisp でCAR などを関数として扱っていたのと等価の状態である．また，繰り返しなどの制御構造も Block へのメッセージで実現しており，これも，初期の Lisp でマップ関数を関数呼び出しで実現しているのと等価である．言語処理系の完成度では Lisp のほうが先行している．概念の整理と純粋性の観点からは Smalltalk-80 が優れているが，実際的な表現力とシステムとしての処理能力を与えているのは Lisp のほうである．しかし，Lisp はオブジェクト指向のシステムとしてみたときの完成度と概念整理は不十分である．Smalltalk-80 の速度向上が早いか，Lisp の概念整理が先行するか興味のあるところである．

5. まとめ

本稿では，Lisp におけるオブジェクト指向として重要なものと考えられる新旧の Flavors と CLOS を中心に解説を試みた．そして，オブジェクト指向の概念と Lisp のもつ機能とが親和性の高いものであることを説明した．

参考文献

1) Keene, S. E. et al.: Flavors: Object-oriented Programming on Symbolics Computers, Symbolics, Inc. (1985).
2) 大里延康: Smalltalk-80 vs. Flavor, Loops. TAO, Computer Today, No. 4, サイエンス社 (1984).
3) 梅村恭司: Smalltalk-80 入門, サイエンス社 (1986).
4) Bobrow, D. G. et al.: CommonLoops: Merging Lisp and Object-Oriented Programming, Proc. of OOPSLA '86, ACM (1986).
5) Bobrow, D. G. et al.: Common Lisp Object System Specification., ANSI Draft X 3 Document 87-003 (1987).

(昭和 62 年 10 月 29 日受付)