3. User-Level Language Crafting

3.1 Introduction

CLOSの設計の基本的なアイディアとは，言語の実装のモデルを特定し，標準化することである。それによって実装の内部の動作を操作可能になる。 本章ではこの機能(すなわちMOP)を説明する。

3.2 The Metaobject Protocol

CLOSの目標:

- ユーザーやプログラムからCLOS環境の内部を調べられるようにする
- プログラムが既存のコードを変更することなしに CLOS自身を拡張できるようにする

前者はクラスブラウザやデバッガを作るのに使われる。 後者は新しいアプリケーションやシステム環境にCLOSの動作を適応 可能にするために必要。

3.2.1 Design Premise and Challenges

CLOS内部を static part と dynamic part に大きく分けて考える。これらは それぞれ先の2つの目標とおおまかに対応している(図3.1)。

static part
: metalevel architecture と呼ばれ，システムの基礎的要素とその構造のこと。 主な要素としては，言語の実装上に おけるクラスやスロット，メソッドの表現(manifestation)がある。

dynamic part
: プロトコル(protocol)と呼ばれ，実行時に言語の振る舞いに影響する基本的要素の 操作を規定する。

言語の振る舞いのそれぞれに一つ以上のプロトコルが対応する。例えば，新しいオブジェクトの定義はクラスの iniialization protocol と finalization protocol に支配される。

CLOSを変更することは言語の一部であり，それゆえ言語自身と同じ 移植性を持つ。

こうしたシステムは言うは易しいが，設計するのが難しい。なぜなら，

- 適切な基本的要素に分解するのが大変
- プロトコルをどの程度詳細に規定すべきかを判断するのはさらに大変

なため。適切なバランスを見つけるには試行錯誤が必要となる。

3.2.2 Implementation of the Design Premise

CLOSの作るメタレベルの世界(CLOS-producing metalevel world)は それ自身がCLOSで書かれている。このことは*meta-circularity*と呼ばれる。 言語自身がオブジェクト指向プログラ

ミング技術で扱われるような CLOSプログラムである。 言語の実装を言語の領域としてほったらかさずに変更できるような 能力を*reflection*という。

メタレベルアーキテクチャはCLOSのクラス階層上に定義され，実装される。 例えば，CLOSのクラスやメソッドはメタレベルのクラス(*metaobject class*)の インスタンス(*metaobjects*)である。

CLOSのダイナミクスはこうした(メタレベルの)クラスに特化した総称関数群に よって捉えることができる。また，そうした総称関数の働きはプロトコルによって 記述されている。

3.2.3 A More Detailed View

図3.2の説明。ふつうのプログラマは`defclass`や`defmethod`, `defgeneric`を使うことで無意識にメタオブジェクトを作っている。 また，クラスのインスタンスを 作ったり，総称関数を起動するなどの際にメタオブジェクトを使っていることになる。

メタレベルプログラマは同様のことを意識的に行なう。

find-class
: プログラマレベルのクラス(の名前)を受けとって，対応するメタオブジェクト を返す。

find-method
: (後から出てくるので省略)

symbol-function
: シンボルから関数実体を取り出す

```
;; emacs-lisp での例(^_^;
(defun double (x)
  (* 2 x))
=> double

(symbol-function 'double)
=> (lambda (x) (* 2 x))

(symbol-function 'symbol-function)
=> #<subr symbol-function>
```

また，システムのクラスからサブクラスを作ったり，システムの総称関数に メソッドを加えたりする。

3.2.4 Curbing Chaos

モジュール(クラスライブラリ?)を作るプログラマは言語が不変であると 考える。そこで，あるモジュールをロードしたら，言語が変わってしまう というのでは，他のモジュールが正しく動かないかもしれない。

しかし，MOPはオープンにしてあって，衝突の回避策を強制していない。 そのかわりにメタレベルの変更に関する規則がある。

オープンになっているのはラジカルな変更が必要な場合もあるからである。しかし，一般にはプログラムはポータブル，つまり他のプログラムと共存可能で ないとならない。

規則はAppendix Aに載っている。メタレベルに手を加えるときは，他の プログラムがあてにしている既存のシステム動作を変えないというのが基本。

3.3 An Example Problem

ここからはCLOSにオブジェクトの永続性(persistence)を持たせるという拡張を 行なう例をとって，MOPの使い方を見ていく。

- オブジェクトは永続性を持つものと，一時的(transient)なものに分かれる とする。`make-persistent`というメッセージを送ることで オブジェクトを永続化できる。
- 永続性を持つオブジェクトもメモリ上にキャッシングされる。
- 永続性をもつオブジェクトであっても，いくつかのスロットが一時的であって もよい。
- CLOSではクラスの再定義が出来るが，このためにクラス階層が変化してしまう ので，永続オブジェクト用データベースのスキーマをそれに合わせて進化させ なくてはならない。

本稿はデータベースでの実際までは踏み込まない。しかし，MOPを使って 以下のようにCLOSの振る舞いを変えるのだという点は理解しよう。

さて，永続化を実装するには...，

- プログラムでクラスを調べて，適切なスキーマを作れないといけない。
- クラスの定義・再定義をトラップして，スキーマの生成と発展(? evolution) を可能にしないといけない。
- 継承を操作できなくてはならない。
- スロットを一時的であると宣言するためのオプションを導入しないとならない。
- 実行時にどのスロットが一時的であるか調べる必要がある。
- プログラマに気付かれることなく，インスタンスが生成されるたびに内部情報 (スロットが一時的であるか否かなど)を管理しないといけない。
- クラスごとに付加的な情報が必要になる。例えば，データベースがどのように アクセスされるかや，あるクラスのインスタンスに対する特別なキャッシング ポリシーなどがある。
- データベースへの faulting を実装するためにスロットアクセスは intercept されなくてはならない(?)
  (Slot access must be intercepted to implement faulting to the database)

`persistent-metalevel-class`という *class*メタオブジェクトを定義し，プログラマに使ってもらうことにする。 プログラマレベルのクラスとして， `cache`や`make-persistent`といった永続化のためのメソッド をもった`persistent-root-class`を定義する。

3.4 Metalevel Statics

3.4節はCLOSのstatic partの説明。構成要素としては，

1. Classes
2. Slots
3. Methods
4. Generic Functions
5. Method combination

がある(図3.3)。

3.4.1 The *Class* Metaobject Class

一番，重要な部分(?)。経験則として，defclassで指定された 情報は，のちほどその defclassで作られる*class*メタオブジェクト から見つけることができる。

standard-classのインスタンスは プログラマがdefclassマクロを使って， 定義するようなクラスを実現するメタオブジェクトになる，ので重要である。 新たにメタクラスを作るときはstandard-classから派生させるのが 普通。

built-in-classのインスタンスはdefclassで作られないような システム組み込みのクラスを実現し，再定義されないなどの特徴を持つ。
forward-referenced-classのインスタンスは 自分のスーパークラスが定義されていないようなクラスを定義するときに使われる。

standard-classに格納されている情報:

スロット
: *slot*メタオブジェクトのリストとして格納されており， class-slots, class-direct-slotsで調べることが出来る

スーパークラスとサブクラス
: *class*メタオブジェクトのリストとして格納されており， class-direct-superclasses とclass-direct-subclasses で調べることが出来る

デフォルト初期化引数
: class-default-initargsあるいは class-direct-default-initargsで調べることが出来る

クラスが最終化(finalize)されているかどうか
: class-finalized-p で調べることが出来る

以下，永続化の詳細を見ていく。まずは，*class*メタオブジェクトに スキーマチェック用のスロットを加える。

次にプログラマレベルの永続クラスとしてhypertext-nodeを定義する。 これはプログラマレベルのクラスであると同時に，メタレベルで見ると persistent-metalevel-classという*class*メタオブジェクトクラス のインスタンスであるようなメタオブジェクトである。

3.4.2 The Slot-definition Metaobject Class

OOPにとって，スロットは単に値を保存するだけのものではなく， スロットの初期化やアクセスは管理されなくてはならない。そこで，CLOSでは スロットもメタオブジェクト(slot-definition)になっている。 スロット定義メタオブジェクトにはスロットの値は格納されていない。

standard-direct-slot-definition
: クラス定義フォーム(defclass)から直接得られるような 情報を保持している

standard-effective-slot-definition
: CLOSの継承規則が適用された後の実行時のスロットの情報を保持している

3.4.3 The Method Metaobject Class

メソッドメタオブジェクトは`defmethod`で定義されるあらゆる情報を 保持する(スロットがインスタンスごとの値を持たなかったのを踏まえた言い方か)。

```
(find-method <generic-function-meta-object>
             <list-of-qualifier-keywords>
             <list-of-class-metaobjects>)
```

`find-method`
: メソッドからメソッドメタオブジェクトを得る

`method-generic-function`
: メソッドが結びつけられている総称関数を探し，総称関数メタオブジェクトを 返す。

`method-lambda-list`
: メソッドの引数の単なる名前のリストを返す

`method-specializers`
: メソッドのそれぞれの引数のクラスに対応するメタオブジェクトクラスのリストを返す

`method-qualifiers`
: メソッドの修飾子を返す。主メソッドなら`NIL`。

ここまでが statics の説明。次回以降で dynamics を見ていく。

---

harada@graco.c.u-tokyo.ac.jp