オブジェクト指向言語 Ld-2
Common Lisp 版
ユーザーズマニュアル
TR-98-19

一杉 裕志
電子技術総合研究所

1998 年 8 月 19 日

目次

1 はじめに **3**

2 稼働システムとインストール方法 **4**
2.1 必要なソフトウエア 4
2.2 コンパイル 4
2.3 ロード 4
2.4 サンプルプログラムの実行 4

3 言語 Ld-2 の概要 **5**
3.1 Ld-2 の特徴 5
3.2 ソフトウエアの部品： system mixin 5
3.3 system mixin の定義: defsystem 5
3.4 system mixin の結合 : setup 6
3.5 プログラムの起動 : start 6
3.6 メソッド探索順序 6
3.6.1 クラスとモジュール 6

4 言語仕様 **8**
4.1 system mixin の定義 8
4.2 大域変数宣言 8
4.3 クラス定義 8
4.3.1 defclass 8
4.3.2 class 8
4.3.3 インスタンス変数宣言 9
4.4 メソッド定義 9
4.4.1 構文 9
4.4.2 メソッド名について 9
4.4.3 abstract method 9
4.4.4 メソッドの役割 9
4.5 メソッド呼び出し 10
4.6 オリジナルのメソッド呼び出し 10
4.7 インスタンス生成 10
4.8 インスタンス変数へのアクセス 10
4.9 変数 *self* 10
4.10 Common Lisp 関数 10
4.11 大域変数 10
4.12 setup 11
4.13 start 11

5 プログラム例 **12**
5.1 最小のプログラム例 12

6 実行開発環境 **13**
6.1 Emacs support 13
6.1.1 かぎかっことまるかっこの混在 13
6.1.2 インデント 13
6.1.3 その他 13
6.2 コンパイルとロード 13
6.3 デバッグ 13
6.3.1 info 文 13
6.3.2 メソッド名 13
6.3.3 Common Lisp のデバッグ機能 13
6.3.4 インスタンス生成関数 14

7 クラスと部品の設計 **15**
7.1 設計・開発の流れ 15
7.2 何をクラスにするか？ 15
7.3 継承関係はどう設計すべきか？ 15
7.4 何を部品 (system mixin) にするか？ 16
7.5 部品間の接続インターフェースは？ 16
7.6 部品間の依存関係をシンプルにするには？ 16
7.7 部品間の競合関係を少なくするには？ 17
7.8 追加部品はどう実装すべきか？ 17

8 実装ノート **18**
8.1 実装の概要 18
8.2 トランスレーター 18
8.3 実行時システム 18
8.3.1 クラス 18
8.3.2 オブジェクト 18
8.3.3 メソッド呼び出し 18
8.3.4 インスタンス変数 18
8.3.5 大域変数 18

9 将来の機能拡張 **20**

第 1 章

はじめに

本マニュアルでは Common Lisp 上で動くオブジェクト指向パッケージ Ld-2 の言語仕様と使い方について述べる。マニュアルを読むには、 Common Lisp のシンタックスとオブジェクト指向に関する基礎的な知識が必要である。

本文中では、 gcl-2.1 (Gnu Common Lisp) を使った実行例を載せている。他の Common Lisp 処理系でもほぼ同じだが、特にデバッグ関係は処理系ごとに異なるので注意してほしい。

プログラミング言語 Ld-2 は、ソフトウエアの部品化をサポートするシンプルなオブジェクト指向言語である。ここで言う部品化とは、ソフトウエア開発者だけでなくソフトウエア利用者にとっても利益のある部品化である。 Ld-2 で書かれたアプリケーションの利用者は、ソフトウエア部品を選択して組み合わせることによって、ソフトウエアを様々なコンフィギュレーションにカスタマイズすることができる。また、既存の部品を変更拡張して新しい機能を追加することも容易である。従来は、エンドユーザによるソフトウエアのコンフィギュレーションは、 #ifdef, patch, プラグインなどの形で行なわれてきたが、これらの方法よりも自然な記述で、はるかに多様性の高いアプリケーションを提供することができる。

Ld-2 は、コンパイラキット Lods の記述言語として設計・開発された。コンパイラキット Lods は、プログラミング言語の進化速度促進を目指し、言語処理系を機能単位で部品化し、部品間の接続インターフェースを標準化することによって、非常に幅広い範囲の言語処理系を容易に構築できるようにすることを目指している。

Lods は、単なる言語処理系作成者のためのクラスライブラリにとどまらず、言語ユーザにとっても利益あるものを目指している。言語ユーザは言語実装部品の中身の詳細をしらなくても、部品を追加・交換するだけで、新しい言語機能や新しい実装方式を利用することができる。

ちょうど PC 互換機を組み立てるユーザのように、実装の詳細に関する知識は必要とせず、何と何が接続可能かという知識さえあればよい。

このような言語処理系と接続インターフェースが提供されていれば、個々の言語処理系部品は比較的小規模な研究室、企業レベルでも実用的なものが提供可能になる。

Ld-2 は現在のところ単一継承で、静的型や情報隠蔽の機能を持たないシンプルなオブジェクト指向言語である。一方、 system mixin と呼ぶソフトウエアの部品化をサポートする特徴的な機構を持っている。メソッドの内部記述言語や実行・開発環境は Common Lisp のものを使う。Ld-2 は Common Lisp のマクロを使って実現されており、 Common Lisp (ただし CLOS は不要) が稼働する環境ならばどこでも利用可能である。

Ld-2 は、オブジェクト指向の継承が持つ機能のうち、部品化の機能にのみ重点を置いて設計されている。また、プログラマーの便宜よりも部品の組み合わせを行なうアプリケーションユーザの便宜を優先した機能をいくつか持っている。プログラマーに特定のプログラミングスタイルを強制しプログラミングの自由度を奪うような機能は一切入れていない。

なお、このドキュメントは最初、 1997 年 2 月に書かれたものだが、少し手直しして、 1998 年 8 月にテクニカルレポートにしたものである。

Ld-2 は、最初コンパイラキット Lods の記述言語として開発されたが、 1997 年 3 月には、拡張可能 Java プリプロセッサ EPP Common Lisp 版の記述に使われた。また、 Ld-2 とほぼ同じ機能を Java 言語に追加する “SystemMixin plug-in” が、 EPP の上で実装されている [18]。

第 2 章

稼働システムとインストール方法

2.1 必要なソフトウエア

いわゆる CLtL の第１版に準じた Common Lisp 処理系が必要である。CLOS (Common Lisp Object System) は必要ない。また、ＯＳには全く依存しない。現在のところ、以下の処理系で動作が確認されている。

- gcl on SunOS, Solaris
- ecl on SunOS
- clisp on SunOS, Windows95

2.2 コンパイル

まず Common Lisp 処理系をインストールする。次に、ファイル ld-2.lsp があるディレクトリに移動し、Common Lisp を起動して、

```
(compile-file "ld-2.lsp")
```

を実行する。

2.3 ロード

Ld-2 を使えるようにするには、ソースファイル ld-2.lsp をコンパイルしてできたファイル ld-2.o があるディレクトリに移動し、Common Lisp を起動して、

`(load "ld-2.o")` あるいは `(require 'ld-2)`

を実行する。ファイルのサフィックスは、OS や Common Lisp 処理系によって違うので注意すること。

2.4 サンプルプログラムの実行

サンプルプログラムが置いてあるディレクトリ（まだ作ってない）に移動し、

```
(load "file-name.lsp")
```

を実行する。

第 3 章

言語 Ld-2 の概要

3.1 Ld-2 の特徴

Ld-2 は、以下の特徴を有する。

- system mixin と呼ぶ部品化を支援する機構を有する。
- Common Lisp 上でマクロを使って実装されたシンプルなオブジェクト指向言語。メソッド内の動作記述には Common Lisp そのものを使う。
- エンドユーザによる「部品」の組み合わせ時のエラーチェックの機構を有する。
- 今のところ、単一継承であり静的型チェックと情報隠蔽の機能はない。
- Smalltalk のクラス変数、クラスメソッドのようなものはない。

3.2 ソフトウエアの部品：system mixin

Ld-2 では、一つのソフトウエアを部品に分割したものを system mixin と呼ぶ。この system mixin を複数組み合わせることにより、一つの完全なアプリケーションとして実行可能な状態にする。

system mixin は他のシステムに対する変更・拡張の差分部分を定義したものである。具体的には、クラスの追加、既存のクラスへのインスタンス変数・メソッドの追加、既存のメソッドの再定義・拡張を記述したものがsystem mixin である。 system mixin は従来の言語の「モジュール」に近い役割も持つ。

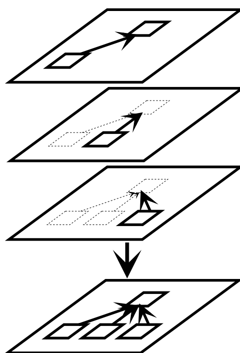

図 3.1: system mixin の結合

```
(defsystem <<S>> ()
  (defclass <super> ()
    (no-orig)                ; super class なし
    (variable !v)            ; インスタンス変数の定義
    (defmethod :init ()      ; 初期化メソッド
       (setf !v 0))
    (defmethod :add (x)      ; メソッドの定義
       (setf !v (+ !v x))))
  (defclass <sub> ()
    (orig <super>)           ; super class の指定
    (method :add (x)         ; メソッドの拡張
      (print x)
      [:original x]) ; super のメソッドの呼び出し
   )
 )
```

図 3.2: システムとクラスの定義

system mixin は、 unix の patch ファイルのようなものだが、変更・拡張の単位は行単位ではなくクラスやメソッド単位である。 patch ファイルでは変更箇所の指定は前後の文脈を用いるため、複数の patch の同時適用が意図した通りに行なわれない可能性がある。 system mixin では、変更箇所はクラス名・メソッド名を用いて指定されるため、そのような問題はない。

複数の system mixin を記述し、それを１列に並べて組み合わせることにより一つのシステムが構築される（図 3.1）。個々の system mixin は、クラス階層の一部分を記述したもので、それらを重ねることによって、クラス階層の全体が完成する。

3.3 system mixin の定義: defsystem

system mixin の定義は図 3.2のように行なう。この例では、クラス `<super>` と、そのサブクラス `<sub>` を持

```
(setq x [:<sub>]); クラス <sub> のインスタンス生成
[x :add 1]        ; x のメソッド呼び出し
[:my-method 123] ; 自分自身のメソッド呼び出し
```

図 3.3: インスタンス生成とメソッド呼び出し

つ system mixin `<<S>>` を定義している。
`(orig <class-name>)` は super class の指定、`(no-orig)` は、 super class を持たないことを示す。

インスタンス変数には名前の先頭に “`!`” をつける。

`defmethod` は新しいメソッドを定義する時に使い、`method` はすでにあるメソッドを拡張する時に使う構文である。`method` 構文の内部では、`[:original arg ...]` という式によって、 super class の同じ名前のメソッドを呼び出せる。

図 3.3は、インスタンス生成とメソッド呼び出しの例である。かぎかっこの第一引数はメソッドを送るオブジェクト、第二引数はメソッド名、残りはメソッドの引数である。かぎかっこの第一引数にメソッド名が来た時には、そのインスタンス自身のメソッド呼び出しを意味する。クラス名をメソッド名として用いた場合は、そのクラスのインスタンスを生成するメソッドと解釈される。

図 3.2 のクラス `<super>` にインスタンス変数 `v2` とその初期化のコードを追加する system mixin は、以下のように定義する。

```
(defsystem <<add-v2>> ()
  (class <super> ()
    (variable !v2)
    (method :init ()
      [:original]
      (setf !v2 0))))
```

`(class ...)` という構文は、すでに定義されているクラスを拡張する時に `defclass` の代わりに使う。

`<<S>>` に `<<add-v2>>` を追加した結果できるシステムでは、クラス `<super>` はあたかも以下のように定義されたかのように振舞う。

```
  (defclass <super> ()
    (no-orig)
    (variable !v)
    (variable !v2)
    (defmethod :init ()
      (setf !v 0)
      (setf !v2 0))
    (defmethod :add (x)
      (setf !v (+ !v x))))
```

通常の継承ではメソッドやインスタンス変数を追加した subclass を定義しても他の兄弟の subclass には全く影響はないが、 system mixin を使った super class への変更は、全ての subclass にそのまま継承される。

3.4 system mixin の結合 : setup

system mixin を一列につなぎあわせることにより、一つのシステムが完成する。これには、トップレベルで以下の式を実行する。

```
(setup (list <<Sn>> ... <<S2>> <<S1>>))
```

リストのもっとも右にある system mixin がオリジナルのシステムであり、それに次々に変更を加えていったものが結果のシステムとなる。

3.5 プログラムの起動 : start

プログラムの起動は、`(start)` という関数で行なう。これにより直前に setup によって定義されたシステムの `<startup>` というクラスが生成され、その初期化メソッド :init が実行される。プログラムの実行は、このメソッド :init から開始される。

system mixin の結合時に、クラスやメソッドの二重定義、未定義のエラーがチェックされる。例えば全てのクラスに対し、構文 `(defclass ...)` による定義がちょうど一つだけないとエラーとなる。クラス内の各メソッドに対しても同様である。

3.6 メソッド探索順序

system mixin の中の defclass または class で定義される「クラス定義の断片」を class mixin と呼ぶ[1]。つまり、 system mixin は複数の class mixin から構成される。同様に class mixin の中にある「メソッド定義の断片」は method mixin と呼ぶ。system mixin を結合した段階で、各クラスの class mixin も一列に結合される。class mixin は、 system mixin 結合順序およびクラス間の継承関係によって、図 3.4のように直列化される。これを class mixin list と呼ぶ。メソッドの探索は、この class mixin list の先頭から行なわれる。すなわち、「 subclass 優先」という従来のルールに加えて、「後から追加された system mixin 優先」（この図での左側）というルールに従う。

3.6.1 クラスとモジュール

再利用性・拡張性の高いアプリケーションの記述にはオブジェクト指向言語が有効である。しかし、多くのオブジェ

[1] これは flavors や CLOS でいう mixin に相当する。

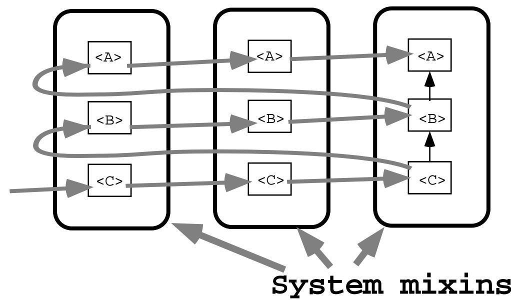

図 3.4: class mixin list

クト指向言語ではクラスの機構がモジュールの機構を兼ねており、その問題点が指摘されている [17]。本来クラスは動的に生成されるデータ構造の雛型あるいはポリモーフィズムの単位であり、モジュールは機能ごとの内部実装の隠蔽や再利用の単位である。アプリケーションによってはこの２つの概念がほぼ一致することもあるが、一般には全く異なる。例えば、複数のクラスが互いの内部実装に依存しながら１つの機能を実現したり、１つのクラスが複数の機能に密接に関わっている場合がある。特にコンパイラ記述においては、クラス間の依存関係は複雑であり、従来のオブジェクト指向言語が持つ継承機構だけでは部品化が困難である。

Ld-2 では、ある機能を実現するコードを１つの system mixin にまとめて記述することによって、モジュールを実現することができる。（ただし、情報隠蔽の機構はまだ実装されていない。）従来のオブジェクト指向言語と違い、複数のクラスを一つのモジュールにしたり、一つのクラスのメソッドを複数のモジュールに分割して記述することができる。

より詳しい議論は、 7章を参照のこと。

第 4 章

言語仕様

4.1 system mixin の定義

```
defsystem ::=
    (defsystem <<system-name>>
       (defsystem-option-list)
       defsystem-toplevel-list)

defsystem-option-list ::=
    | defsystem-option defsystem-option-list

defsystem-option ::=
    decl-dependency | ...

defsystem-toplevel-list ::=
    | defsystem-toplevel defsystem-toplevel-list

defsystem-toplevel ::=
    global-variable | defclass | class

decl-dependency ::=
    (depend dependency-list)
```

dependency-list の部分には、この system mixin が依存する system mixin の名前を書くことができる。 setup 時に、ある system mixin に依存する部品が list の前に来ていないと、エラーとなる。（この機能は未実装。）ここで「A が B に依存する」とは、 A および A が依存する system mixin 中で定義された資源名（大域変数名、クラス名、メソッド名、インスタンス変数名）を B の中で参照していることを言う。

現在の実装では、依存の宣言と実際の依存関係との食い違いをチェックしていない。将来の実装では、宣言された system mixin で定義されていない名前を参照しているとsetup 時にエラーを出すようになるかもしれない。

4.2 大域変数宣言

Ld-2 の大域変数宣言は、 defsystem の中に次のように書く。

```
global-variable ::=
   (global-variable !!gvar-name)
```

なお Common Lisp の defvar 宣言による大域変数（特殊変数）も使えるが、この構文の使用を推奨する。

大域変数値の初期化はされない。クラス <startup> の :init メソッドなどで、明示的に初期化を行なう必要がある。

4.3 クラス定義

4.3.1 defclass

defclass は新しいクラスを定義するための構文である。

```
defclass ::=
   (defclass <class-name> (defclass-option-list)
      decl-orig
      defclass-toplevel-list)

defclass-option-list ::= Not defined yet.

decl-orig ::= (no-orig) | (orig <class-name>)

defclass-toplevel-list ::=
    | defclass-toplevel defclass-toplevel-list

defclass-toplevel ::=
    variable | defmethod | method
```

defclass 構文は、必ず :init という名前のメソッド定義を含んでいなければならない。初期化の必要が特になくても、`(defmethod :init ())` と書いておく必要がある。

4.3.2 class

class は既存のクラスを拡張するための構文である。

```
class ::=
   (class <class-name> (class-option-list)
      defclass-toplevel-list)
```

```
class-option-list ::= Not defined yet.
```

class 構文によって、既存のクラスに対してインスタンス変数の追加、メソッドの追加、既存のメソッドの再定義をすることができる。

4.3.3 インスタンス変数宣言

defclass および class 構文の中で、インスタンス変数の宣言を行なうことができる。

```
variable ::=
    (variable !var-name)
```

値の初期化は、 :init メソッド内で行なう。

4.4 メソッド定義

4.4.1 構文

defmethod は新しいメソッドを定義する構文、 method は既存のメソッドを再定義する構文である。

```
defmethod ::=
    (defmethod :method-name lambda-list body)

method ::=
    (method :method-name lambda-list body)

lambda-list ::= Defined by Common Lisp.
```

lambda-list には、 &optional, &rest など Common Lisp の機能がすべて利用できる。

4.4.2 メソッド名について

defmethod で宣言する名前は、全 system mixin 中で重複しないようにすべきである。（ただし属するクラスが異なれば重複してもよい。）また、名前とその役割の対応は、全 system mixin 中でコンセンサスがとれているものと仮定する。（巨大なアプリケーションを構築するためには名前空間を分ける機構が必要になってくるだろうが、いまのところそれはない。）

4.4.3 abstract method

インターフェースだけを定義し、実装を定義しないいわゆる abstract method は、次のように書く。

```
(defmethod :method-name (args ...) (abstract))
```

C++ や Java では abstract method を一つでも持つクラスに対してインスタンスを生成しようとするとコンパイルエラーとなるが、 Ld-2 ではそのようなチェックはやっていない。その代わりに、実行時に abstract method を呼び出した時に実行時エラーとなる。

なお、 abstract method だけを含む system mixin を abstract system mixin と呼ぶ。

4.4.4 メソッドの役割

この節で述べる機能は、現在完全には実装されていない。前節の abstract method をさらに拡張した機能である。

すべてのメソッド定義は、 defmethod/method の違いとは別に、「メソッドの役割」を持つ。メソッドの役割には、 abstract, default, implement, extend, redefine の 5 種類がある。すべての method mixin は、これらのうちのいずれかの役割を持つ。メソッドの役割は、各メソッド定義の body の最初で宣言する。

例：

```
(defclass <a-data> ()
  (no-orig)
  (defmethod :size ()
     ;; The role of this method mixin is "default".
     (default)
     1)
  ...)
```

メソッドの役割は、継承によるメソッドの再定義に制限をつける。例えば、 implement method を他の implement method で再定義できない。 setup の際このように制限に反するメソッド再定義が見つかった場合、エラーとなる。この制限によって、ある機能を実装する異なる部品が 2 つ指定されて衝突し、正しく動作しないことを未然に防ぐことができる。

それぞれのメソッドの役割の意味は以下の通りである。

- abstract
  いわゆる abstract method 、すなわちインターフェースのみを定義し実装を定義しないメソッド宣言である。defmethod 構文でのみ使える。実行時にこの method mixin が呼び出されるとエラーである。
- default
  default method は、他の method mixin によって全く別のものに再定義されることを許すメソッドである。default method は abstract, default method を再定義できる。 default method は、 defmethod および method 構文で使える。 defmethod 構文では、 (default) 宣言は省略可能である。従って、先の例は、以下のようにも書ける。

  例：

  ```
  (defclass <a-data> ()
  ```

```
(no-orig)
(defmethod :size ()
  ;; Implicitly declared as "default".
  1)
...)
```

default method は、 :original 呼び出しを含んでいてもいなくてもよい。

- implement
  abstract, default method を再定義できるメソッド。:original 呼び出しを含んでいてはいけない。 defmethod および method 構文で使える。
- extend
  default, implement, extend, redefine method を再定義できるメソッド。:original 呼び出しを必ず含まなければならない。 method 構文では、 (extend) 宣言は省略可能である。 method 構文でのみ使える。
- redefine
  abstract, default, imlement, extend, redefine method を再定義できるメソッド。:original 呼び出しを含んでいてはいけない。このメソッドは、「間違った役割宣言をしたメソッド」を再定義したい時などに有効である。 redefine method の仕様は、部品組み合わせの危険度を増すので注意が必要である。 method 構文でのみ使える。

4.5 メソッド呼び出し

`[obj :method-name args ...]` は、 obj のメソッド `:method-name` を呼び出す。

`[:method-name args ...]` は、自分自身のメソッド `:method-name` を呼び出す。

4.6 オリジナルのメソッド呼び出し

`[:original args ...]` という式は、いわゆる「 super class のメソッド呼び出し」に相当する。より厳密に言うと、 class mixin list （3.6参照）にしたがって次の method mixin を探索し、呼び出す。

4.7 インスタンス生成

`[:<class-name> args ...]` という式によってインスタンスが生成される。引数はそのまま、そのクラスの :init メソッドに渡される。

メソッド `:<class-name>` を再定義することはできるが、推奨しない。

4.8 インスタンス変数へのアクセス

インスタンス変数の値の参照は `!var` 、代入は `(setf !var val)` で行なう。（setf を setq と間違いやすいので注意。）実は、 !var は [:var] 、 (setf !var val) は [:set-var val] というメソッド呼び出し式として解釈される。メソッド `:var`, `:set-var` は、クラスに `(variable !var)` という宣言があると自動的に定義される。インスタンス変数へのアクセスがメソッド呼び出しを通して行なわれるため、以下のことが可能である。

- 他のオブジェクトのインスタンス変数にもアクセスできる。例えば [obj :x] というメソッド呼び出しはオブジェクト obj の !x というインスタンス変数の値を取り出す。また、 [obj :set-x 0] は、 !x の値に 0 を代入する。
- サブクラスでアクセスメソッドを再定義することができる。

  例：

```
(method :set-x (val)
  (print "Value of x is changed!")
  [:original val])
```

4.9 変数 *self*

メソッド中では、変数 `*self*` の値が自分自身を指している。つまり、式 `[:foo]` は `[*self* :foo]` とまったく同じ意味である。

4.10 Common Lisp 関数

高階関数を含む全ての Common Lisp 関数をメソッド中から呼び出すことができる。また、インスタンス変数へのアクセスやメソッド呼び出し式を含む関数を defun で定義することができる。

ただし、現在の実装では変数 *self* は dynamic scope に従うので注意すること。例えば、インスタンス変数へのアクセスを含む関数 closure を、他のオブジェクトのメソッドの引数に渡すと、 *self* の値はその受け手のオブジェクトになるので、送り手側のインスタンス変数にはアクセスできない。このように混乱のもとになるので、関数 closure をメソッドの引数に渡すようなインターフェース設計は推奨しない。関数 closure の代わりにオブジェクトを用いること。

4.11 大域変数

大域変数へのアクセスは !!gvar 、 (setf !!gvar val) という式によって行なう。また、動的スコープに従って値を変更するためのマクロとして let-g がある。

例：

```
(let-g ((!!gvar1 val1)
        (!!gvar2 val2)
        ...)
  body ...)
```

4.12 setup

トップレベルで system を再定義した時は、原則として setup し直さなければならない。

setup 時に、以下のチェックが行なわれる。この機能により、エンドユーザが「意味のある」部品の指定を行なったかどうかをある程度検出できる。

- すべてのクラス名に対して、 defclass の前に class が来ていないか。同じ名前の defclass が 2 重になってないか。
- すべてのクラス内のメソッドについて、 defmethod の後に method が来ていないか。 defmethod が 2 重になっていないか。

4.13 start

関数 start は、 setup によって構築されたシステムの起動を行なう。関数 start が呼ばれると、まず <startup> というクラスのインスタンスを生成し、その :init メソッドに関数 start の引数の値を渡す。:init メソッドの実行が終了すると、トップレベルに戻る。:init メソッドの返値は無視される。

第 5 章

プログラム例

5.1　最小のプログラム例

Ld-2 の最小のプログラム例は次のようになる。

```
 (defsystem <<s>> ()
  (defclass <startup> ()
    (no-orig)
    (defmethod :init ()
      (print "Hello World."))
    ))
(setup (list <<s>>))
```

第 6 章

実行開発環境

6.1 Emacs support

6.1.1 かぎかっことまるかっこの混在

Emacs のデフォルトの lisp-emacs では、かぎかっこ が かっこ として認識されない。これに対処するため、付属の paren.el を使用する。このパッケージには、かぎかっこ と まるかっこ の対応が会っていないと beep をならす、などの機能がある。

それでも、開きかっこの削除などによって編集中にかっこの対応が狂うことはある。gcl の場合 load 中にエラーがでたら (read-line) を何度か入力してみると、エラーがおきた場所の次の行が表示され、エラーの場所が特定できる場合がある。

6.1.2 インデント

Ld-2 の構文のいくつかは、デフォルトの lisp-mode ではうまくインデントできない。Emacs Ver.18 の場合、以下のように .emacs に入れる。Emacs のその他のバージョンでも動くかどうかは不明。

```
(load "cl-indent")
(put 'method  'common-lisp-indent-hook
     '(4 (&whole 4 &rest 1) &body))
(put 'class  'common-lisp-indent-hook
     '(4 (&whole 4 &rest 1) &body))
(setq lisp-indent-hook 'common-lisp-indent-hook)
```

6.1.3 その他

その他は Common Lisp の開発環境がそのまま利用できる。

6.2 コンパイルとロード

Ld-2 のプログラムは、通常の Common Lisp のプログラムと全く同じようにコンパイル・ロードすることができる。以下は、gcl での例である。プログラムは、.lsp をサフィックスに持つファイルに記述される。１つのファイルに複数の system mixin を入れてもよい。sys-a.lsp を load するには、(load "sys-a.lsp") を実行する。このファイルをコンパイルするには、(compile-file "sys-a.lsp") を実行する。

6.3 デバッグ

6.3.1 info 文

`(info object)` または `(info <class>)` によって、オブジェクトやクラスの情報を得ることができる。

なお、このとき、値がまだ代入されていないインスタンス変数は表示されない。

6.3.2 メソッド名

Ld-2 のメソッドは、Common Lisp の処理系上では

<<システム名>>/<クラス名>/ メソッド名

という名前の関数として実行される。

もし実行時エラーが起きて Common Lisp のデバッガーに入ったら、backtrace すれば、どのメソッドでエラーが起きたのかを知ることができる。また、Common Lisp の trace 文の引数にこのメソッド名を使うこともできる。

6.3.3 Common Lisp のデバッグ機能

break, trace, step などのデバッグ機能は Common Lisp のものがそのまま使える。メソッド実行中の実行時エラーや break 文によってデバッガに入った場合、そのオブジェクト自身が特殊変数 *self* に束縛されている。従って、デバッガのプロンプトから、インスタンス変数・大域変数へのアクセス、メソッド呼び出しなどが行なえる。

例：

```
[other-obj :mess]
[:print]
!a
(info *self*)
```

6.3.4 インスタンス生成関数

break 等でデバッガに入った場合は、通常のプログラム内と同様に `[:<class-name>]` によってインスタンスを生成できる。

しかし、 toplevel では上記のメソッド呼び出し式はエラーとなる[1]。そのような場合は、代替手段として、インスタンス生成関数が利用できる。`(<class-name> args ...)` を実行すると、インスタンスが生成される。引数は、 :init メソッドの引数に渡される。ただし、このように関数 start を通さずにクラスを生成してメソッド呼び出しを行なうと、大域変数の設定が行なわれていないので、大域変数へのアクセスはエラーとなるので注意。

[1]変数 `*self*` 等の値が設定されていないため。

第 7 章

クラスと部品の設計

7.1 設計・開発の流れ

「極めて多様なコンフィギュレーションが可能なアプリケーション」を Ld-2 で設計・開発する手順は、おおよそ次のようになる。

1. 作りたいアプリケーションが決まったら、何をクラスにするかを設計する。アプリケーションのコンフィギュレーションが変わっても、各クラスの役割は変わらない点に注意する。
2. コンフィギュレーションを１つに固定して、アプリケーションを実装してみる。最初は system mixin の切り分け方は大雑把でよい。
3. 「交換できる機能」を１つずつ増やしていく。それにともなって、ソースコード全体を system mixin に分割していく。必要に応じて "hook" メソッド[1]を入れていく。接続インターフェースも汎用性の高いものに徐々に変えていく。
4. 部品間の依存関係・競合関係を整理する。そのためには、 abstract system mixin （4.4.3節）の導入や、メソッド定義場所の移動などを行なう。
5. ある程度接続インターフェースが安定してくると、システムを公開する。公開された接続インターフェースに依存する実装部品がたくさん作られるようになる。この段階になると、もはや接続インターフェースや部品の切り分け方の大規模な変更は許されなくなる。ただし、メリットが大きければ局所的な接続インターフェースの変更はその後も行なわれ続ける。

7.2 何をクラスにするか？

２つの場合がある。

- 構造体の代わり。つまり、複雑なデータ構造を一つのポインタで指したくなった時。あるいは同じ構造をしたデータを動的に複数生成したい時。もちろん、それに対する操作はメソッドとして定義する。
- ポリモーフィズムが必要な時。

「あるクラスの実装に依存する手続きは、そのクラスのメソッドにする」というルールは、 Ld-2 ではむしろ忘れた方がよい。これはクラスの役割とモジュール (system mixin) の役割を混同している。相互に密接に関連するソースコードは同じ system mixin に入れた方がいいが、同じクラスにする必要はない。クラスは、必ずしもカプセル化の単位として適切ではない。

7.3 継承関係はどう設計すべきか？

- Ld-2 は system mixin の機能を持っているため、コードの再利用、あるいは差分プログラミングという目的だけなら、必ずしも subclass を作る必要はない。system mixin を追加すれば、クラスの名前を変えることなく、差分的に機能を追加することができる。必要もないのに差分的変更のためだけに subclass を作ると、クラス名が別のものになってしまう。そうすると、そのクラス名をソースコード中に埋め込んでいる他の部品は使えなくなってしまう。つまり、部品の組み合わせの自由度が著しく減ってしまう。
- 一般に is-a 関係にあるクラスは subclass にした方がよいが、 Ld-2 では必ずしもそうでない場合もある。例として Lods の <frame-manager> というクラスがある。 Lods では言語のスタックフレームの実装のバリエーションとして、配列を使ったフレームと、 C のスタックをそのまま使ったフレームの２種類を実装した。２つの <frame-manager> クラスを、１つの <abstract-frame-manager> クラスの subclass にする方法もあるが、実際のコンパイラで使用されるのは通常どちらか一方だけである。そこで、 frame-manager のクラスはただ１つのみで、その複数の実装を異なる system mixin で記述して交換できるようにした。
- Ld-2 には今のところ静的型システムはないので、クラスのインターフェースの階層構造は、継承関係に反映させる必要は全くない。「ソースコードの共有」という性質にのみ着目して継承関係や部品化の設計を行

[1] 再定義で拡張されることだけを目的としたメソッドをこう呼んでいる。

なうべきである。

- クラスがモジュールとは違うのと同様に、継承関係はモジュールのネスト構造を表現する手段ではないので注意する。

7.4　何を部品 (system mixin) にするか？

- 「交換」が想定される部分を「部品化」する。部品化とは、ソースコードを複数の system mixin に分割して記述することである。相互に密接に依存し合っている、あるいは特定の実装に依存しているものは、できるだけ１つの system mixin にまとめた方がよい。
- 他の system mixin の実装に依存する system mixin は絶対に存在してはいけない、というわけではない。「部品の内部は隠蔽されていて、公開された固定したインターフェースのみを通してアクセスしなければならない」というルールは保守性を高めるためには必要だが、それを犠牲にすることで部品の種類の多様化が図れる場合がある。例えばある部品 A に追加する機能の実装方法に２種類あり、どちらにするかをエンドユーザが選択可能にしたいならば、追加機能を別の部品 B_1、 B_2 として実装せざるを得ない。このような場合、コンフィギュレーションの多様性と、保守のしやすさとのトレードオフを考えて設計する。
- デザインパターン [9] のうち、 decorator は Ld-2 では必要ない場合が多い。 Ld-2 の system mixin を使えばもっと直接的に、拡張性の高いソフトウエアが書ける。動的に機能を変更できるという点では、 decorator パターンを用いた方が高機能である。しかし、記述は高機能な分、複雑になる[2]。コンパイラキット Lods をもし普通のオブジェクト指向言語で動的に機能変更できるスタイルで書いたら、多数の種類の decorator が入り組んで多分とても保守できない。

7.5　部品間の接続インターフェースは？

接続インターフェースとは、ある機能を実装する側のコードと利用する側のコードが、どの資源名（クラス名、メソッド名、インスタンス変数名など）をどのように用いて相互作用するかを定義したものである。モジュール機能を持つ言語におけるモジュールインターフェースとは少し違うので注意。（モジュールインターフェースは、モジュールの内部を隠蔽し、モジュールの利用者がどのようにしてそのモジュールを利用するかを定義したものである。）モジュールの実装とモジュールインターフェースは１対１に対応する。部品接続インターフェースは、特定の部品と１対１に対応しているわけではない。複数の部品が協力して１つの接続インターフェースを実装することもあるし、１つの部品が複数の接続インターフェースを実装することもある。例えば Lods における heap-manager を実装する部品は、「メモリの割り当てを行なうオブジェクトに関する接続インターフェース」と、「ＧＣを実装する部品の接続インターフェース」の両方を実装する。つまり、１つの部品が、複数の異なった目的の利用者に対する複数の接続インターフェースを実装していることになる。（ java の「interface の多重継承」を使う状況に近いかもしれない。）

接続インターフェースの設計には以下の点に注意する。

- 接続インターフェースはできるだけ抽象度が高く汎用的で安定している方がよい。しかし、完全なインターフェースを最初から設計するのは絶対に不可能である。従っていろいろな部品を実装しながら必要に応じて徐々に変更していく必要がある。
- 需要のない接続インターフェースの設計に時間をかけてはいけない。部品が交換できれば便利そうだと思った機能のインターフェースだけを注意して設計する。交換できても意味のない部分は無理して複数の部品に分ける必要がない。

接続インターフェースは本来、ソースコード上に実体のない抽象的な存在だが、 abstract system mixin の形で接続インターフェースの subset を記述することはできる。これと他の機能（4.1節、 4.4.4節）を組み合わせることによって、ある機能の実装側と利用側が接続インターフェースを正しく満たしているかどうかの必要条件を自動的にチェックできる。（現在はこのチェック機能は実装されていない。）

7.6　部品間の依存関係をシンプルにするには？

モジュール間の依存関係は一般には有向グラフになる。依存関係は、部品の組み合わせの自由度を制限するので、できるだけ依存グラフはシンプルな方がよい。（例えば大規模なループなどは好ましくない。）また、ある部品に依存する場合でも、できるだけ「強くは」依存しないようにした方がよい。それには、いくつかの方法がある。

- 相互に強く依存するメソッドをできるだけ一つの system mixin の中に押し込める。
- 他の system mixin の変化が激しそうなメソッドに依存している場所があったら、より安定したメソッドを通して間接依存する形に書き直す。
- 複数の部品間で相互依存関係が生じる場合は、 abstract system mixin を導入し、相互依存関係を解消する。（図）。例えばＡとＢが相互依存している場合、Ａ、Ｂ内で定義されている資源名を abstract system mixin

[2] なお、部品の記述は現在のままで、動的に system mixin を交換できるような機能を Ld-2 の将来の追加機能として考えている。

Ｉの形で抜き出せば、A、Bがそれぞれ I に依存する木構造に変えることができる。

- 大域変数をうまく使えば、部品間の依存関係を簡単化できる。クラスを設計する段階で、何を大域変数にすべきかも同時に考える。一般にアプリケーション全体に関係した「状態」を表すオブジェクトは、多くのメソッドの引数にしてひきずり回すよりも、大域変数にした方がよい。

理想的には、 abstract method だけを含む system mixin と実装だけを含む system mixin に完全に分けて記述するのがいいのかもしれない。（図）。だが、記述の初期の段階おいては接続インターフェースは絶えず変化するので、最初からそのような記述スタイルで始めるのはあまり得策ではない。 abstract system mixin は接続インターフェースがほぼ固定してきてから導入しても遅くはない。

7.7　部品間の競合関係を少なくするには？

setup 時にある２つの部品を同時に指定すると絶対に正しく動かない、という組み合わせがありうる。そのような部品は「競合関係にある」と呼ぶ。 4.4.4節で述べたように、メソッドの役割を宣言することで競合は setup 時にある程度自動的に検出できる。

競合を減らすには、以下のことに気をつける。

- メソッドを再定義する時にはできるだけ redefine method は使用せず、オリジナルの機能を残した extend method にする。
- ある機能に関連する abstract method は１つの abstract system mixin にできるだけまとめる。同様に、ある機能を実装する implement method や、ある機能を拡張する extend method もそれぞれ system mixin にまとめる。これにより system mixin の「役割」がはっきりして、競合の度合が減るのではないかと思われる。

7.8　追加部品はどう実装すべきか？

システム全体の骨格がすでにできている段階で、新たな部品を追加実装する場合、少しでも再利用性の高い部品として記述した方が望ましい。再利用性を高くするためには、以下の点が重要である。

- 需要が大きく汎用的な機能を提供すること。
- 他の部品との組み合わせの自由度が高いこと。（依存する部品および競合する部品の数が少ないこと。）
- 長い間安定して使われること。（安定していると思われる接続インターフェースにのみ依存すること。）
- 拡張性を高くしておくこと。（多くのメソッドを変更・拡張を想定した形で記述すること。）

もちろんこれらの間にはトレードオフが必要なこともある。

第 8 章

実装ノート

この章では Ld-2 でのクラスやメソッド呼び出しの実装方法について簡単に述べる。普通のユーザは、この章を読む必要はない。

8.1 実装の概要

Ld-2 のシステムはトランスレータと実行時システムからなる。トランスレーターは、 defsystem 構文で定義された system mixin を Common Lisp のプログラムに変換する。メソッド呼び出しおよびインスタンス変数へのアクセスはハッシュ表を通して行なわれる。 setup 時に各クラスが持っているすべてのメソッド探索を探索し、ハッシュ表が構築される。

8.2 トランスレーター

defsystem 構文は Common Lisp のマクロとして定義されている。入力された system mixin 定義は、図 8.1のようにマクロ展開されてから Common Lisp に解釈される。

8.3 実行時システム

8.3.1 クラス

setup によって system mixin のリストから、個々のクラスが生成される。クラスは、 ld-class という名前の構造体で実現され、 class mixin list を保持する。また、インスタンス変数とメソッドキャッシュを保持し、インスタンス生成時およびメソッド呼び出し時に使われる。

8.3.2 オブジェクト

オブジェクトは、 ld-object という構造体で実現される。この構造体は、 class と variables という２つのスロットを持つ。 class は、そのオブジェクトを生成する際に雛型とした ld-class 構造体であり、 variables は、インスタンス変数を保持するハッシュ表である。

8.3.3 メソッド呼び出し

[o :foo] というメソッド呼び出し形式は、 read macro によって (send o :foo) という関数呼び出し形式に展開される。関数 send は、特殊変数 *self* の値を o に束縛して、関数呼び出し (send-self :foo) を実行する。

[:bar] というメソッド呼び出し形式は、 read macro によって (send-self :bar) という関数呼び出し形式に展開される。

関数 send-self は、 *self* が属するクラス（構造体 ld-class ）を取り出し、そこからメソッドを検索し実行する。

実際には、メソッドキャッシュの処理があるのでもう少し複雑である。クラスごとに、メソッド名とクロージャの対応表をハッシュ表として持つ。

現在の実装では、このハッシュ表は、 setup 時に構築される。結局、メソッド呼び出しは１回のハッシュ表検索ですむことになる。

8.3.4 インスタンス変数

!var というインスタンス変数へのアクセスは、 read macro によって (send-self :var) に、 (setf !var val) という代入文は、 (send-self :set-var val) に展開される。

8.3.5 大域変数

すべての大域変数は、実は <global-variables> というクラスのインスタンス変数として実現されている。このクラスは、 setup 時に自動的に定義される。起動時にはクラス <global-variables> のインスタンスが１つ生成され、 Common Lisp の変数 *global-variables* に代入される。（デバッガから (info *global-variables*) と実行してみるとわかる。）

大域変数へのアクセスを行なう !!gvar という式は read macro によって `[*global-variables* :gvar]` を、 (setf !!gvar val) という代入文は `[*global-variables* :set-gvar val]` を行なう式に展開される。

将来的に *global-variables* の値が動的に切り替わるような言語機能の追加を考えている。

入力：

```
>(macroexpand
  '(defsystem <<S>> ()
    (defclass <super> ()
      (no-orig)
      (variable !v)
      (defmethod :init ()
        (setf !v 0))
      (defmethod :add (x)
        (setf !v (+ !v x))))
    (defclass <sub> ()
      (orig <super>)
      (method :add (x)
        (print x)
        [:original x])
      )
    ))
```

展開結果：

```
(progn
  (register-system '<<s>>
      (make-system-mixin :name '<<s>> :classes
          (list (make-ld-class-mixin :name '<sub> :defp 't :mixin-id
                    '<<s>>/<sub> :orig '<super> :vars 'nil :methods
                    '((:add <<s>>/<sub>/add nil)))
                (make-ld-class-mixin :name '<super> :defp 't :mixin-id
                    '<<s>>/<super> :orig ':no-orig :vars '(:v) :methods
                    '((:set-v <<s>>/<super>/set-v t)
                      (:v <<s>>/<super>/v t) (:add <<s>>/<super>/add t)
                      (:init <<s>>/<super>/init t))))))
  (defun <<s>>/<sub>/add (x) (print x) (send-self :original x))
  (defun <<s>>/<super>/set-v (val)
    (setf (gethash ':v (ld-object-variables *self*)) val))
  (defun <<s>>/<super>/v ()
    (gethash ':v (ld-object-variables *self*)))
  (defun <<s>>/<super>/add (x)
    (setf (send-self :v) (+ (send-self :v) x)))
  (defun <<s>>/<super>/init () (setf (send-self :v) 0)))
```

図 8.1: defsystem の展開例

第 9 章

将来の機能拡張

将来、次のような機能拡張を考えている。

- 資源名の rename （parameterized module）
- system mixin の repeated inheritance
- system mixin の依存関係の宣言とエラーチェック（情報隠蔽機能）
- 実行効率改善（インライン展開等）
- system mixin の動的な追加・交換

参考文献

[1] 一杉裕志、平野聡、田沼均、須崎有康: ”無制限に変更・拡張が可能なコンパイラの設計・実現方法の提案”, 第４８回情報処理学会全国大会, 7G-4, March, 1994.

[2] 一杉裕志、平野聡、田沼均、須崎有康、塚本享治: ”拡張可能システムの記述を支援するオブジェクト指向言語”, 日本ソフトウエア科学会第１１回大会, pp.29–32, 1994.

[3] 一杉裕志、平野聡、田沼均、須崎有康、塚本享治：”拡張可能コンパイラキットとその記述言語”, 第１１回オブジェクト指向計算ワークショップWOOC’９５, 1995.

[4] 一杉裕志、平野聡、須崎有康、田沼均、塚本享治: ”言語処理系クラスライブラリと一体化したオブジェクト指向構造エディタ”, 第５１回情報処理学会全国大会, 1L-1, September, 1995.

[5] 一杉裕志, 平野聡, 田沼均, 須崎有康, 塚本 享治: ”コンパイラキット Lods における交換可能なＧＣモジュールの構築法”, プログラミングシンポジウム, Jan, 1997.

[6] Bodin, F. et al: ”Sage++: A Class Library for Building Fortran and C++ Restructuring Tools”, In Proc. of Object-Oriented Numerics Confs., Oregon, Apr.1994.

[7] Bracha, G. and Cook, W.: ”Mixin-based Inheritance”, In Proc. of ECOOP/OOPSLA’90, pp.303–311, 1990.

[8] Chiba, S.: ”A Metaobject Protocol for C++”, In Proc. of OOSPLA’95, pp.285–299, 1995.

[9] Gamma, E., H., R., Johnson, R. and Vlissides, J.: *Design Patterns*, Addison-Welsley (1995).

[10] Gray,R.W., Heuring,V.P., Levi,S.P., Sloane,A.M. and Waite,W.M.: ”Eli: A Complete, Flexible Compiler Construction System”, Communications of the ACM 35 (February 1992), pp.121–131.

[11] Ishikawa, Y.: ”Meta-level Architecture for Extendable C++ Draft Document”, Technical Report TR-94024, RWCP, 1994.

[12] Kiczales, G., des Rivieres, J. and Bobrow, D. G.: The Art of Metaobject Protocol, MIT Press, 1991.

[13] Lamping, J., Kiczales, G., Rodriguez, L. and Ruf., E.: ”An Architecture for an Open Compiler”, In IMSA: Reflection and Meta-Level Architecture, pp.95-106, 1992.

[14] Paakki, J.: ”Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation”, ACM Computing Surveys, Vol.27, No.2, pp.196–256, June 1995.

[15] Smith, B. C., “Reflection and Semantics in Lisp”, In Conference Record of ACM POPL ’84, pp. 23–35, 1984.

[16] Steele, G.: *Common Lisp the Language, 2nd edition*, Digital Press (1990).

[17] Szyperski, C.A.: ”Import is Not Inheritance, Why We Need Both: Modules and Classes”, In Proc. of ECOOP’92, pp.19–32, 1995.

[18] Ichisugi, Y.: EPP home page.
http://www.etl.go.jp/~epp

[19] Gnu Common Lisp
ftp://ftp.ma.utexas.edu/pub/gcl/