複数の最上位環境をサポートするLispモジュール機能

安本太一[†] 湯淺太一[††]

複数の名前空間を持つLispのためのモジュール機能を提案する．提案するモジュール機能の特徴は，モジュールによって最上位環境（top-level environment）とともに記号空間を分割し，モジュール間で（記号ではなく）束縛の可視性制御を行うことにある．既存のLisp言語を自然でかつ容易な方法で拡張してモジュール機能を追加できるうえ，Lispにおけるプログラム開発効率の高さも損なわない．さらに，マクロの束縛捕捉問題を，単純ではあるが効果的に解決できる．

A Lisp Module System to Support Multiple Top-level Environments

TAICHI YASUMOTO[†] and TAIICHI YUASA[††]

A module system is proposed for Lisp dialects with multiple name-spaces. A module in this module system is characterized by its own top-level environment and its own symbol space. By partitioning a single symbol-space, as well as a single top-level environment, into modules, the module system allows to extend existing Lisp languages in a natural and easy way, while preserving the efficiency of program development in Lisp. It also provides simple but effective solutions to the binding-capturing problems of macros.

1. はじめに

モジュラープログラミングは大規模なプログラムを開発するために重要なプログラミングパラダイムであり，1つのプログラムを，関連した機能の集合をまとめた複数のモジュール（module）から構成する．各モジュールの大きさは，プログラムの開発や保守が容易になるように，適度に小さい方が良いとされている．モジュールは，他のモジュールで定義された変数や関数の束縛のうち，**export**されている束縛を**import**することによって利用できる．このようなモジュール間の関係は各モジュールの**インタフェース**で指定する．束縛の可視性を制御する概念を導入することにより，変数や関数の名前の衝突を回避することができ，モジュールを集めて1つのプログラムを容易に構築することが可能である．

本稿では，Lispのためのモジュール機能を提案する．Lispにおいてモジュラープログラミングをサポートするためには，以下の機能が必要である．

(1) プログラム全体をモジュールに分割し，各モジュールで定義した束縛をexportしたり，各モジュールにおいて使用されるこれらの束縛を選択してimportするための言語構文．

(2) モジュールを効率良く開発するための対話的なプログラミング環境．

(3) モジュラープログラミングにおける束縛の可視性制御能力を利用して，各モジュールについて効率的なコードを生成するコンパイラ．

(4) 分割コンパイルを支援するためにモジュール間の依存関係を自動保守するようなモジュール管理ツール．

本稿は，最初の3点を対象とする．最後の1点については，提案するモジュール機能とは分離して論じることが可能なので，別の機会に譲ることにする．

Lispには，Scheme[2]のような単一の名前空間を備えたものと，Common Lisp[16]のような複数の名前空間を備えたものに分けられる[8]．名前空間とは，変数や関数の名前とこれらの名前が特定する実体（記憶領域中の場所や関数本体）の対応づけ（束縛）を行うときの空間である[7]．Common Lispでは，変数の名前空間と関数の名前空間が独立して存在するので，同名の変数と関数を独立に定義できる．複数の名前空間を

† 愛知教育大学教育学部総合科学課程総合理学コース数理科学選修
Department of Mathematical Science, Faculty of Education, Aichi University of Education
†† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

サポートするLisp言語が多く存在するにもかかわらず，Lispにおけるモジュラープログラミングの研究は，単一の名前空間を持ったLisp言語を前提としているものが多い．そこで，本稿では，複数の名前空間を備えたLisp言語のためのモジュール機能について議論する．このような言語におけるモジュラープログラミングにおいては，変数束縛と関数束縛の区別は必要不可欠である．変数束縛はモジュール間の通信手段として他のモジュールによって更新され，関数束縛はその関数を定義したモジュールによってのみ更新されると考えるのが自然である．複数の名前空間を備えたLispのためのモジュール機能は，以下を目標として設計されるべきである．

(1) 名前空間を区別したexport.
たとえば，モジュールが同じ名前の変数束縛と関数束縛を定義していても，公開したい束縛のみをexportすることが可能である．

(2) 名前空間を区別したimport.
たとえば，モジュールは，あるモジュールから変数束縛のみをimportし，別のモジュールからはこれと同名の関数束縛のみをimportすることが可能である．

(3) 名前空間ごとに独立したimport.
たとえば，変数束縛をimportし，同じ名前の関数束縛を**内部束縛**（importしたものではなく，モジュール自身によって設定した束縛）とできる．

(4) importされた束縛の属性の継承.
たとえば，変数が定数として宣言されている場合は，その変数をimportしたモジュールは“定数”という属性も継承し，値の変更は許されない．また，マクロとして定義された関数束縛をimportした場合は，“マクロ”という属性も継承し，マクロ展開に使用できる．

さらに，Lispの対話性を維持するためには，以下も考慮すべきである．

(1) import，exportおよび束縛定義の順序.
未定義の束縛であっても，その束縛のexportやimportが可能である．定義される前の束縛は，未束縛な状態にある．

(2) exportされた束縛の更新.
更新の結果は，その束縛をimportしたすべてのモジュールにただちに反映される．

(3) インタフェース指定と束縛定義の自由な変更.
プログラマは，任意のモジュールを選択して，インタフェース指定や束縛定義を変更できる．

(4) 不完全なプログラムの実行.
まだ定義されていない束縛があっても，その束縛が使用されない限りはプログラムを実行できる．

提案するモジュール機能は，特定のLisp言語を対象にしたものではない．多くのLisp言語において，このモジュール機能を付加することにより，モジュラープログラミングが可能となるように設計を行った．拡張された言語は，もとになった言語（基本言語）の上位互換であり，基本言語で記述されたプログラムは，拡張言語の構文規則を満たす．そして，そのプログラムは，拡張言語のためのLisp処理系上では，基本言語のためのLisp処理系上で実行した場合とまったく同じように動作する．

本稿の構成は以下のとおりである．まず，2章では，関連した研究を概観することによって，過去に提案されたモジュール機能との相違を明確にする．3章では，具体的な例をあげながらモジュール機能を説明するために，仮想的な基本言語を導入し，モジュール機能に関連したLispの基本的な用語を再確認する．4章では，モジュール機能のために基本言語に付加する構文を提案し，拡張した基本言語処理系の動作を説明する．5章と6章では，マクロと記号操作について，それぞれ議論する．7章では，モジュール機能実現にともなう言語処理系拡張と，束縛の可視性制御を利用したコンパイラのコード生成における最適化について述べる．

2. 関連研究

Common Lispは，記号をパッケージ（package）に分割して登録するための機構を備えている．記号は名前を持ったデータであり，変数や関数の名前を示す識別子は記号オブジェクトによって内部表現される．そして，束縛は，記号オブジェクトによって実現されている．パッケージ間で記号の可視性制御を行う機能が提供されているが，次に示す多くの問題があり，我々が目指すモジュール機能にとって代わるものではない．

(1) 一見隠蔽されているかのようにみえる内部記号（internal symbol）は，“パッケージ名 :: 記号名”という構文によって参照可能である．

(2) 記号単位の可視性制御機能なので，変数束縛と関数束縛の双方を独立して扱うことができない．たとえば，関数束縛のみのimportが必要な場合でも，意図しない変数束縛のimportが強いられる．

(3) exportされた関数束縛が他のパッケージによって変更されるのを防ぐことができない．

(4) 束縛のimport先を動的に変更できない．import先はテキストの入力時に決定されるので，テキストを再入力しない限り，import先を変更できない．

(5) 束縛のimport手続きに柔軟性がない．importしたい束縛に対応する記号が存在していなければimportできない．また，その記号が存在していても，exportされていなければimportできない．このように，プログラマはexportやimportの順序に煩わされる．

(1)，(2)，(3)のために，モジュール化の目的である束縛の可視性制御が十分に達成されているとはいい難い．また，(4)と(5)はLispの持つ対話性とはそぐわない．このような不具合は，記号を操作の対象とする限り避けられない．

TALK[4]は複数の名前空間を持つLisp言語であり，モジュール機能を備えている．このモジュール機能は応用プログラムの配布を強く意識したものである．マクロはコンパイル時（応用プログラムの配布前）のみに必要であり，関数は実行時（配布後）に必要となる．このために，関数とそれに関連したマクロを同一のモジュールに定義することを禁止している．また，TALKのモジュール機能では，exportした束縛がプログラム全体からアクセスできてしまう．この問題をパッケージ機能を採用して解決していることから，Common Lispのパッケージ機能と同様な問題を抱えている．

FelleisenとFriedman[6]は，Schemeのための簡潔なモジュール機能を提案している．これは対話的なプログラミングを考慮して設計されてはいるが，インタフェース指定が明示的に記述できないうえ，マクロもサポートしていない．Tung[17]もSchemeにおける対話的なモジュラープログラミングのための機能を提案しているが，このシステムはマクロをサポートしていないうえ，Xウィンドウシステム上に作られたプログラミング環境を前提としているので汎用性に欠ける．

CurtisとRauen[3]は，syntactic closure[1]を基本に，マクロをサポートするSchemeのためのモジュール機能を提案している．QueinnecとPadget[14]も，マクロをサポートするモジュール機能を提案し，モジュラープログラミングにおける諸問題について論じており有益な提案を行っている．しかし，これらのシステムは対話的なプログラミングについて触れていない．実際，これらのモジュール機能を対話的に使用することは困難であると推測される．

Standard MLは，構造化されたモジュール機能[11),12)]を備えており，静的な型チェックを可能にしている．SheldonとGifford[15]は，静的多様型（static polymorphic type）を扱うLisp方言のためのモジュール機能を提案している．しかし，これらのモジュール機能は，静的な型判定ができない多くのLisp言語には適さない．

3. 基本言語

本章では，仮想的なLisp言語を定義し，それによってモジュール機能に関連する基本的なLisp用語を再確認する．この言語をモジュール機能を付加する基本言語として使用する．

図1に，基本言語の主な構文を示す．この言語はCommon Lispのサブセットに近いものであり，特に断らない限りCommon Lispの用語を用いることにする．この言語は，2つの名前空間，すなわち関数（およびマクロ）の名前空間と変数の名前空間をサポートしている．この特徴を基本言語に採用することは非常に重要である．なぜなら，2つの名前空間を備えたLisp言語のためのモジュール機能を定義しておけば，より多くの名前空間を備えたLisp言語のためのモジュール機能に拡張するのはたやすいからである．

以下では，この基本言語の処理系の動作について簡単に説明する．この動作は，インタプリタを備えたほとんどのLisp言語処理系に共通するものである．

(1) 処理系は，*program*の終わりに達するまで以下のステップを繰り返す．

(2) 処理系は，キーボードあるいはファイルから**最上位式**（*top-level expression*）を読む．最上位式のテキスト表現（S-式）は，Lispオブジェクトに変換される．このとき，各識別子は記号に変換され，同じ識別子はすべて同一の記号に変換される．テキスト表現とそれに対応するLispオブジェクトを区別するために，後者を**フォーム**と呼ぶ．特に，最上位式に対応するフォームを**最上位フォーム**と呼ぶ．

(3) 最上位フォームが*definition*ならば，**最上位束縛**（top-level binding）が設定される．基本言語では，**defun**と**defmacro**は最上位の関数束縛を設定し，**defvar**は最上位の変数束縛を設定する．これら最上位束縛が**最上位環境**（top-level environment）を構成する．基本言語は複数の名前空間を備えているので，同じ名前の関数束縛と変数束縛が，最上位環境に共存することが可能である．

(4) 最上位フォームが*expression*ならば，空の**静的環境**（lexical environment）の下で，このフォー

```
program        ::=  top-level-expr*
top-level-expr ::=  definition | expr
definition     ::=  (defun function-name parameters expr*)
                 |  (defmacro macro-name parameters expr*)
                 |  (defvar variable-name expr)
expr           ::=  constant
                 |  variable-name
                 |  special-expr
                 |  (macro-name object-expr*)
                 |  (function-name expr*)
special-expr   ::=  (setq variable-name expr)
                 |  (if expr expr expr)
                 |  (let variable-binding-spec expr*)
                 |  (flet function-binding-spec expr*)
                 |  ⋮
        ⋮
```

図1 基本言語の構文
Fig. 1 Syntax of the base language.

ムが評価される．let（あるいはflet）で始まるフォームは，静的変数束縛（あるいは静的関数束縛）を静的環境に設定する．
関数名（あるいはマクロ名）で始まるフォームならば，その名前の関数束縛を静的環境内で探す．もし，そのような束縛が見つからなければ，次に最上位環境が探索される．変数名の場合も同様である．このような探索は記号の比較によって行われるので，同じ識別子をすべて同一の記号で表現することが重要である．

4. モジュール機能

提案するモジュール機能は，モジュールごとに，その最上位環境である**モジュール環境**（module environment）を導入する．従来の単一の最上位環境を複数のモジュール環境に分割することによって，最上位束縛の可視性を制御する．プログラム全体をモジュールに分割するために，**図2**に示すように，*program*の定義にモジュールヘッダ（*module-header*）を追加して基本言語の構文を拡張する．

4.1 モジュールヘッダ

各モジュールヘッダは，モジュールヘッダの後に入力される最上位式が，*module-name*によって指定されたモジュールの本体を構成することを示している．モジュールヘッダでは，他のモジュールとのインタフェースを指定する．ヘッダ中の *export-spec* は，**公開束縛**の指定を行う．公開束縛というのは export される最上位束縛のことで，他のモジュールからのアクセスが可能である．これに対して，export されていない最上位束縛は，**非公開束縛**であり，他のモジュールからアクセスすることはできない．基本言語は2つの名前空間を持っているので，名前空間を付して export 指定を行う．function は関数束縛を，variable は変数束縛を指定する．この名前空間の指定は，次に述べる *import-spec*でも行う．

*import-spec*では，**外部束縛**の指定を行う．外部束縛とは，他のモジュールの公開束縛にアクセスするために使われる最上位束縛である．*selection*における，モジュール名と名前空間の指定に続く（$name_1$ $name_2$）は，$name_2$ という公開束縛を，$name_1$ という外部束縛の下でアクセスすることを示している．$name_1$ が $name_2$ と同じ名前であれば，単に $name_2$ と省略してもよい．*name*や*module-name*は内部的には記号オブジェクトで表現される．後述するように提案するモジュール機能では同じ名前を持った記号が複数存在する可能性があるが，これらの記号オブジェクトはその名前だけが意味を持つ．

import 指定において，モジュール名のみを指定したときは，そのモジュールのすべての公開束縛が import される．たとえば，

```
(module a-stack-application
  (import (stack (function
```

program	::=	{ *module-header* | *top-level-expr* }*
module-header	::=	(module *module-name* { *import-spec* | *export-spec* }*)
import-spec	::=	(import { *module-name* | *selection* }+)
selection	::=	(*module-name* (*kind* { *name* | (*name name*) }+)+)
export-spec	::=	(export (*kind* *name*+)+)
kind	::=	function | variable
module-name	::=	*name*

図2 モジュールシステムの構文
Fig. 2 Syntax of the module system.

```
            (stack-push push)
            (stack-pop pop)
            empty-stack))
      base))
```

は，モジュール a-stack-application が，モジュール stack から公開されている関数束縛 push，pop，empty-stack と，モジュール base 内のすべての公開束縛を import することを指定する．push と pop の関数束縛については，それぞれ stack-push と stack-pop というように名前を変えて import する．その他の束縛は，同じ名前で import する．1つのモジュールにおいて，同じ名前空間に属する2つの外部束縛を同じ名前で import すると名前の衝突が起きるが，上のように名前を変更して import できる機能を用いれば，名前の衝突を容易に回避できる．複数の類似するモジュールを組み合わせてプログラムを構築するときには特に起こりがちな問題であり，モジュラープログラミングには欠かせない機能である．上の例では，モジュール base が push と pop の公開束縛を持っていても，名前の衝突は避けられる．

各モジュール環境は，**外部環境**と**内部環境**という2つの最上位環境から構成される．外部環境は外部束縛から構成され，内部環境はモジュール中の束縛定義によって設定された内部束縛から構成される．同じ名前空間において，2つの環境が同名の束縛を持った場合は，内部環境が優先され，外部束縛は隠蔽される．

対話的なモジュラープログラミングを可能にするために，モジュールに対してモジュールヘッダを複数回入力できることにする．既存のモジュールにモジュールヘッダが与えられた場合には，既存のインタフェース指定への追加や更新が行われる．次のように，モジュール名のみでインタフェース指定がないモジュールヘッダも入力できる．

(module *module-name*)

このようなモジュールヘッダは，既存のインタフェース指定には影響を与えず，指定したモジュールにカレントモジュール（現在，最上位束縛が登録されるモジュール）が切り替わるだけであり，プログラマがモジュールを自由に行き来するために使用される．

カレントモジュールは，フォームの評価中にも切り替わることがある．モジュール M において，最上位式が評価されるときは M がカレントモジュールである．一方，フォームの評価中に他のモジュール N で定義された関数 f が呼び出された場合，その本体を評価するときは，カレントモジュールは N に切り替わる．関数 f の定義がモジュール N の下で本体が評価されることを前提としていることを考えれば，これは自然な解釈である．関数 f から戻ったときには，関数 f を呼び出す前のカレントモジュールが，再びカレントモジュールとなる．

4.2 組込みモジュール

モジュール機能を追加した処理系には，base と default という2つのモジュールがあらかじめ定義されている．モジュール base は基本言語が定義されているモジュールであり，基本言語において定義されているすべての関数束縛（マクロを含む）と変数束縛を export している．

組込みの束縛を使用する場合は，モジュールは base を明示的に import しなければならない．すべてのモジュールが，組込みの束縛のすべてを必要とするとは限らないからである．もし，組込みの束縛のサブセットで事足りるモジュールが多いならば，必要な束縛だけを export したモジュールを次のように定義すればよい．

```
(module subset (import base)
  (export (function car cdr cons ...)
          (variable t ...)))
```

base ではなく subset を import することにより，指定した束縛だけを他のモジュールで利用することができる．

モジュール default は，処理系が起動した直後のカレントモジュールであり，base の束縛のみを import

し，束縛はいっさいexportしていない．つまり，次のモジュールヘッダによって定義される．

```
(module default (import base))
```

プログラマがモジュール機能を必要としないときは，モジュール**default**のみを使用することになる．

4.3 記号空間

各モジュールは，それぞれに固有の記号空間（記号の名前と記号の対応づけを行う空間）を1つ有する．すなわち，提案するモジュール機能は，単一の記号空間ではなく，複数の記号空間をサポートする．原則として各記号は1つのモジュールのみに属し，1つのモジュール内に同じ名前を持った記号が複数存在することはない．ただし，唯一性という観点から，1つの**nil**がすべてのモジュールに属するものとする．**nil**は，空リストや真偽値の偽を表現する特殊な記号であるからである．一方，モジュールが異なれば，複数の記号が同じ名前を持つことは可能である．この特徴は，後述するマクロのサポートにおいて重要である．今後の議論で，記号が属するモジュールを明示する必要があるときは，モジュール名を記号名の後に書くことにする．たとえば，$\mathtt{foo}_M$は，モジュールMに属する記号**foo**を示す．

拡張されたLisp処理系がモジュールMにおいて最上位式を読むとき，基本言語の処理系と同様に，テキスト表現はLispオブジェクトに変換される．このとき，識別子はM中の記号によって表現される．たとえば，定義式

```
(defun foo (x) (* x x))
```

は，

$$(\mathtt{defun}_M\ \mathtt{foo}_M\ (\mathtt{x}_M)\ (*_M\ \mathtt{x}_M\ \mathtt{x}_M))$$

という定義フォームに変換される．また，式

```
(foo 10 y)
```

は，

$$(\mathtt{foo}_M\ 10\ \mathtt{y}_M)$$

というフォームに変換される．すなわち，フォームの先頭の要素が，$\mathtt{defun}_M$，$\mathtt{defmacro}_M$あるいは$\mathtt{defvar}_M$のいずれかであれば，M中の最上位フォームは定義式と見なされる．

式中のすべての識別子は，M中の記号に変換されるので，**quote**式

```
(quote (ten (little indians)))
```

は，

$$(\mathtt{quote}_M\ (\mathtt{ten}_M\ (\mathtt{little}_M\ \mathtt{indians}_M)))$$

に変換される．この**quote**フォームを評価すると，その値は，

$$(\mathtt{ten}_M\ (\mathtt{little}_M\ \mathtt{indians}_M))$$

というリストである．もし，同じ**quote**式が他のモジュールNに現れるならば，その値は

$$(\mathtt{ten}_N\ (\mathtt{little}_N\ \mathtt{indians}_N))$$

である．これは，前者と同じ構造であるが，異なる記号から構成されている．

4.4 拡張動作

モジュール機能の付加は，最上位式の構文をまったく拡張しない．拡張されるのは，基本言語Lisp処理系の動作である．

モジュールMにおいて，最上位束縛が設定されるとき，Mのモジュールヘッダによってその束縛がすでにexportされているなら，それは公開束縛である．さもなければ，その最上位束縛は非公開束縛である．もし，後に入力するMのモジュールヘッダがこの束縛をexportすれば，非公開束縛であった最上位束縛は公開束縛になる．

基本言語と同様に，定義式以外の最上位式は，空の静的環境の下で評価され，その評価中に静的束縛が設定される．拡張されるのは束縛の探索処理である．たとえば，f_Mという関数名に出会うと，f_Mの関数束縛が見つかるまで，静的環境，Mの内部環境，Mの外部環境という順に探索される．その束縛が外部環境において見つかった場合は，importした束縛が使われる．すなわち，静的環境や内部環境に隠蔽されない場合に限って，importした束縛にアクセスできる．

importした束縛は，基本言語の最上位束縛と同様にアクセスできる．関数束縛であれば関数呼び出しや関数オブジェクトの取り出しが可能であり，変数束縛であれば参照や代入が可能である．ただし，importした関数束縛の変更はできない．

モジュール機能は，公開束縛のみが他のモジュールからアクセスできることを保証する．しかし，あるモジュール内で作成されたオブジェクトが他のモジュールで使用されることは制限していない．この特徴は関数について混乱を招くかもしれないので，次に例をあげて説明する．

```
(module m (export (function foo)))
(defun bar () ...)
(defun foo () #'bar)

(module n (import m))
... (foo) ...
```

モジュール**n**から**foo**を呼び出すと，モジュール**m**の非公開束縛である**bar**の関数オブジェクトを返す．**m**で作成されたオブジェクトは**n**で自由に使ってよいの

で，これは問題ない．一方，

```
(module n (import m))
... #'bar ...
```

では，モジュールmで定義された関数barの関数オブジェクトを得ることができない．なぜならば，その束縛はモジュールnには見えないからである．

5. マクロ

マクロは，プログラマが新しい構文を追加することを可能にする強力な機能であり，ほとんどのLisp処理系がサポートしている．しかし，マクロはスコープ（scope）の問題を引き起こすことが指摘されている．本章では，提案するモジュール機能を装備した処理系が，この問題を自然に回避できることを示す．この解決策は，文献1)，5)，9)のようにマクロの新たな実現方法を提案して問題を解決するものではなく，プログラムのモジュール化により有効となるものである．

5.1 束縛捕捉の問題

文献1)では，マクロに関する問題のいくつかの例があげられている．このうちの2つを次に示す．最初の例は，最上位束縛への参照が，マクロの利用者が導入した静的束縛によって捕捉され，期待どおりにマクロが動作しない例である．今，マクロが次のように定義されているとする．

```
(defmacro push (expr var)
  `(setq ,var (cons ,expr ,var)))
```

マクロフォーム

```
(flet ((cons (x y) (+ x y)))
  (push 10 stack))
```

は，

```
(flet ((cons (x y) (+ x y)))
  (setq stack (cons 10 stack)))
```

と展開される．マクロ定義では，consは組込み関数を参照することを意図しているが，展開されたフォーム中のconsは局所関数を参照してしまう．

2つ目の例は，マクロフォーム外の束縛への参照が，マクロ展開によって導入された静的束縛により，捕捉されてしまう例である．今，マクロが次のように定義されているとする．

```
(defmacro or (expr1 expr2)
  `(let ((temp ,expr1))
     (if temp temp ,expr2)))
```

マクロフォーム

```
(or (member x y) temp)
```

は，

```
(let ((temp (member x y)))
  (if temp temp temp))
```

と展開される．ifフォーム中の最後のtempはマクロフォームの外部で設定されている変数束縛を参照することを意図しているが，マクロ展開によって導入される静的束縛を参照してしまう．

5.2 束縛捕捉の解決

提案するモジュール機能を備えた処理系では，モジュール M の中で定義されるマクロが別のモジュール N で使用されるならば，前節の問題は自動的に解決される．なぜならば，同じ綴りの識別子が，M と N の中の異なる記号によって表現されるからである．最初の例の展開フォームは次のようになる．

```
(flet_N ((cons_N (x_N y_N) (+_N x_N y_N)))
  (setq_M stack_N (cons_M 10 stack_N)))
```

$\mathtt{cons}_M$ は $\mathtt{cons}_N$ とは異なる記号であり，$\mathtt{cons}_M$ への参照は $\mathtt{cons}_N$ の静的束縛によって捕捉されることはない．同様に，2つ目の例の展開フォームは次のようになる．

```
(let_M ((temp_M (member_N x_N y_N)))
  (if_M temp_M temp_M temp_N))
```

$\mathtt{temp}_M$ のために導入された束縛は，外側の $\mathtt{temp}_N$ の束縛への参照を捕捉しない．

もちろん，モジュール機能を備えた処理系を使ったとしても，マクロを定義するモジュールにおいてマクロを不用意に使用すれば，マクロが予期せぬ結果を起こす可能性は残る．しかしながら，モジュールを記述するプログラマは，マクロが利用される状況を把握できるので，従来よりは誤りの可能性は大幅に減ることになる．ちなみに，Common Lispのパッケージ機能では，問題は解決しない．なぜならば，記号のimportにより，複数のパッケージに属する記号が存在する可能性があるからである．たとえば，pushの例では，consが組込み関数であることから，問題は解決されない．

5.3 束縛アクセスの特例

先述のように，マクロを定義したモジュールと，そのマクロを利用するモジュールが異なる場合は，展開されたマクロフォームには，カレントモジュールに属さない記号が含まれる．この記号が関数（およびマクロ）や変数を示すものであれば，その束縛は他のモジュールから利用されることになるので，当然exportされるべきである．先述のマクロpushを例にあげると，$\mathtt{cons}_M$ の関数束縛は，モジュール M においてexportされていなければならない．

この事例では，importせずに他のモジュールの束縛を利用することになるが，カレントモジュール内の同名の束縛と衝突しないので，問題にならない．この

機能は，マクロによる抽象化をサポートするために不可欠である．マクロの利用者は，マクロ展開の結果，どのような関数が呼び出されるか一般には分からないからである．

6. 記号操作

提案しているモジュール機能を備えた処理系は，複数の記号空間を備えている．したがって，基本言語において単一の記号空間を前提としていた記号操作関数は，複数記号空間に対応できなければならない．

6.1 記号を生成する関数

たいていのLisp言語は，新たな記号を生成するための関数**intern**や**gensym**をサポートしている．**gensym**はつねに新しい記号を作成し，**intern**はすでに存在する記号があればそれを返す．モジュール機能を追加した場合は，これらの関数は，カレントモジュールに属する記号を返すことにする．そこで，これらの関数に対しては，特別な扱いが必要である．このために，モジュールごとに同じ名前の関数を組込み関数として用意することにする．

モジュールはモジュール内の記号のみを生成することができるので，モジュール M が他のモジュール N に属する記号を生成する必要がある場合は，N の助けを借りることになる．たとえば，N の**intern**を**intern-N**という名前でimportして参照すればよい．

```
(module N (export (function intern)))

(module M
  (import
    (N (function (intern-N intern)))))
```

6.2 記号の等値性

関数**eq**は，2つの引数が同一のオブジェクトであるかどうかを比較する．モジュール機能を追加しても，**eq**の定義は不変である．単一の記号空間を持つ従来のLispでは，2つの引数が同じ名前を持つ記号であれば**eq**は真を返す．一方，モジュール機能を追加した処理系では，同じ名前を持った記号でも所属するモジュールが異なれば別の記号になるので**eq**は偽を返す．名前による一致が必要な場合は，次に示す**eq***のような関数を定義して用いることになる．

```
(defun eq* (x y)
  (or (eq x y)
      (and (symbolp x)
           (symbolp y)
           (string= (symbol-name x)
                    (symbol-name y)))))
```

しかし，この定義では，文字列を比較するので，非常に効率が悪い．実行効率の低下を避けるためには，次に示す“汎用”の記号空間を用意する．そして，プログラム中で記号が実際に使用される前にすべてのデータオブジェクトをこの記号空間に変換しておき，比較に**eq**を用いるのが賢明である．

```
(module universal-symbols
  (export (function convert)))

(defun convert (x)
  (if (symbolp x)
      (intern (symbol-name x))
      (if (consp x)
          (cons (convert (car x))
                (convert (cdr x)))
          x)))
```

7. 実　　装

Common Lispのサブセットを基本言語として拡張し，インタプリタおよびコンパイラへモジュール機能の試験的な実装を行ったので，その概要を述べる．拡張の対象とした処理系は，KCL（Kyoto Common Lisp）[18]である．パッケージ機能や，束縛を直接操作する関数（**symbol-function**や**set**）などが除かれている．

7.1 データ構造

モジュール機能に関連したデータ構造と変数を次に示す．

(1) **記号**　次のスロットを持つ．
- (a) **関数束縛**　最上位関数束縛を保持する．次のいずれかの値をとる．
  - (i) 未束縛（記号が生成された直後の値）．
  - (ii) 内部束縛（関数オブジェクトへのポインタ）．
  - (iii) 外部束縛（関数束縛をimportしている記号へのポインタ）．
- (b) **変数束縛**　最上位変数束縛を保持する．その内容は，関数束縛と同様である．
- (c) **importフラグ**　束縛をimportしているときにオンになるフラグ．初期値はオフである．束縛がないときは意味を持たない．関数束縛と変数束縛のために計2つある．
- (d) **exportフラグ**　束縛をexportしているときにオンになるフラグ．初期値はオ

フである．関数束縛と変数束縛のために計2つある．

(e) **束縛属性フラグ** 定数やマクロであることを示すためのフラグ．内部束縛のときにのみ意味を持つ．

(2) **モジュール** 次の2つから構成される．

(a) **インタフェース指定** モジュールヘッダによって逐次与えられるimportやexportに関する情報を格納する．

(b) **記号表（symbol table）** 記号の名前から記号への対応づけを行う．

(3) **current-module** カレントモジュールを指す変数．記号表を指定できるように拡張されたreaderが参照する．

(4) **モジュールテーブル** モジュール名を表す記号からモジュールへの対応づけを行う表．

(5) **モジュールヘッダ記号表** モジュールヘッダ解析のための記号表．

これらは，既存のデータオブジェクトを拡張したり，既存のデータオブジェクトをリストやベクタを使って組み合わせることによって簡単に実装できる．

トップレベルにおいて，先頭の要素が**module**という名前の記号で始まるフォームが入力されると，モジュールヘッダの読み込みが開始される．このフォームを構成するリスト中の記号は，このモジュールヘッダが入力される前のカレントモジュールに属しているので，構文解析を簡単にするためにモジュールヘッダ記号表に属する記号に変換しておく．リストの第2要素である記号は，モジュールテーブルにおける探索のためのキーとして使用する．探索により見つかったモジュールが，新しいカレントモジュールとなる．そのような名前を持ったモジュールがなければ，新しいモジュールが生成されてモジュールテーブルに登録される．

current-moduleの値を新しいカレントモジュールにセットした後，importやexportを次のように行う．

(1) importやexportの対象となる束縛のうち，対応する記号がないものについては，それぞれのモジュールにおいて記号を生成する．

(2) カレントモジュールをMとし，importしようとしている束縛が属するモジュールをNとする．M側の束縛が未束縛あるいは外部束縛でかつ，N側の束縛がexportされている場合に限り，M側の記号の束縛スロットにN側の記号へのポインタをセットするとともに，その束縛スロットに対応するimportフラグをオンにする．M側の束縛が内部束縛ならば，何もしない．また，N側の束縛がexportされていないときは，その外部束縛の導入あるいは更新は保留し，M側の記号の束縛スロットは変化しない．

(3) カレントモジュールが束縛をexportするときは，対応する記号のexportフラグをオンにする．この束縛がexportされていないために保留されていたimportがあれば，引き続いて上記(2)を行うので，importやexportの順序を問わずに束縛をimportできることになる．

内部束縛を定義するフォームは，importフラグの設定というごく簡単な処理が加わったことを除けば，従来とまったく同じように内部束縛を設定する．たとえば，**defun**は，その関数名の記号の関数束縛スロットに関数オブジェクトへのポインタをセットし，importフラグをオフにする．

evaluatorも，内部束縛の場合は従来とまったく同じように束縛の値をとりだす．付加されたのは，外部束縛の場合に，記号オブジェクトの束縛スロットをたどって最終的な内部束縛を担う記号を探索するといった前処理だけである．この前処理までに，束縛が設定されていれば未束縛エラーにはならない．また，**setq**などの特殊フォーム（special form）においても，同様な探索を行う．内部束縛を担う記号が探索された後は，従来と同じ処理を行う．したがって，束縛の属性は探索された内部束縛のものが採用されることになり，外部束縛においても定数やマクロといった属性が継承される．

以上のように，処理系全体としては，大きな変更なくモジュール機能を付加できる．筆者らのグループは，この他に，TUTSchemeというScheme処理系に対して，ほぼ同様の対応を行ったが，短期間のうちに作業を終えた[13]．このScheme処理系における試みの実績は，単一名前空間のみの場合にも適応できることを示し，今回提案するモジュール機能の高い汎用性を示すものである．

7.2 束縛探索の最適化

モジュール機能の付加によって実行効率の低下が懸念されるのは，4.2節のモジュール**subset**の利用例のようにimportが多段になる場合，内部束縛の探索に時間を要することである．そこで，**図3**に示すように，あらかじめ束縛のimportのリンクをたどっておき，1段のimportに帰着させてしまうのが賢明である．この多段のimportの最適化は，モジュールヘッダや定義フォームによって，外部束縛が設定された直

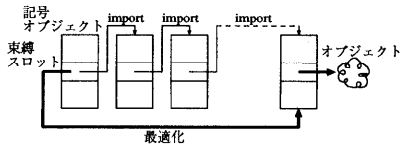


図3 多段importの最適化
Fig. 3 Optimization for multi-level imports.

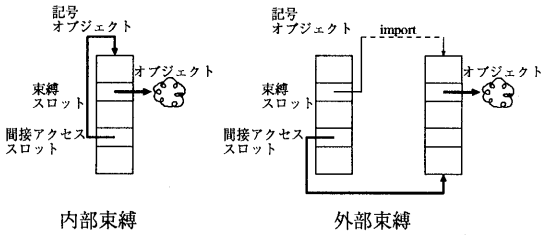


図4 間接アクセススロット
Fig. 4 Indirect access slot.

後に行えばよい．対話的なプログラミングを行っているときは，キー入力のわずかな合間に最適化は終了してしまうであろうから，最適化作業にともなう実行効率の低下は問題とならない．

多段importの最適化により，実際のimportの段数にかかわらず，1つの記号を経由すれば必ず内部束縛にアクセスできる．これを前提とするならば，図4に示すように記号に間接アクセススロットを設け，内部束縛か外部束縛によってアクセス方法を切り替える処理を，間接アクセスに置き換えて簡素化することが可能である．その結果，内部束縛か外部束縛かの相違に関係なく，記号の間接アクセススロット → 記号 → 束縛スロット → オブジェクトという経路で必ずオブジェクトにたどりつくことができるので，特にコンパイラにおけるコード生成が容易になる．対話的なプログラミングによって外部束縛が内部束縛に隠蔽されたときに必要な間接アクセススロットのポインタの付け変えは，多段importの最適化と同様に，定義フォームの実行直後に行えばよい．

7.3 コンパイラ

モジュール機能を付加した場合における束縛へアクセスする部分のコード生成について述べる．以下では，記号経由のアクセスとアドレスによる直接アクセスの2通りを検討する．

拡張の対象としているコンパイラは，KCLのコンパイラをもとにしており，Lispの関数をCの関数に翻訳する．変数束縛や関数束縛へのアクセスは，記号オブジェクトを介して行うコードを生成する．ただし，関数呼び出しについては，関数束縛スロットからとりだした関数本体であるC関数へのポインタをキャッシュする工夫がなされている．2回目以降の関数呼び出しは，このポインタが格納されたCポインタ変数の値を使った間接呼び出しになる．関数が再定義されなければ，記号オブジェクトへアクセスする必要はない．

7.3.1 記号経由のアクセス

プログラムの実行速度を向上させてデバッグの効率を高めることを目的とし，コンパイル前と変わりなく束縛にアクセスできるようにコードを生成する．生成されたコードの実行効率は，基本言語のコンパイラによって生成されたコードの実行効率と同程度でもかまわない．したがって，記号オブジェクトを介して束縛にアクセスすることを基本とする．変数束縛については，前節の簡素化された束縛アクセスを前提としたコード生成を行う．さらに，関数束縛については，束縛をキャッシュする機能を利用するコード生成を行う．つまり，最初に関数本体を得る処理が少し異なるほかは，従来と変わりない．

なお，本稿では触れないが，マクロの再定義についてはweak consを用いてマクロの再定義時の自動再コンパイルを行う手法によって対処できる[10]．

7.3.2 アドレスによる直接アクセス

非公開束縛は，束縛が属しているモジュールで閉じている．プログラマがこのモジュールのトップレベルから，対話的に，この束縛へアクセスあるいはこの束縛をexportしないならば，この束縛のアクセスのために記号オブジェクトを介する必要はない．完成した応用プログラムを利用に供する場合などに想定される状況である．したがって，次のような最適化が可能である．変数アクセスについては，変数束縛スロットを，オブジェクトを指すCポインタ変数として独立させ，変数アクセスにはこのC変数を用いる．関数呼び出しについては，キャッシュによる間接関数呼び出しではなく，Lisp関数に対応するC関数を直接呼び出すようにする．

7.3.3 評価実験

記号経由のアクセスあるいはアドレスによる直接アクセスのコードの性能を評価するために，モジュール機能を付加する前のコードとその実行時間を比較した．コンパイルされたコードの実行は，SUN SS5（microSPARC II 70 MHz）上で行った．

まず，変数にアクセスするときの性能評価を行う．以下の単純な加減算を行うプログラムにおいて，関数**incdec**を実行し，大域変数**x**へのアクセスを2×10^7回行った．大域変数**x**は非公開束縛なので，その変数束縛だけをC変数として扱うコード生成が可能である．実行時間を**表1**に示す．なお，変数束縛アクセス

表1 変数束縛のアクセスコスト
Table 1 Cost of accessing variable bindings.

コード生成	実行時間（秒）	比
モジュール機能なし	10.52	1.00
直接アクセス	8.44	0.80
記号経由アクセス	11.64	1.11

表2 関数束縛のアクセスコスト
Table 2 Cost of accessing function bindings.

コード生成	実行時間（秒）	比
モジュール機能なし	4.60	1.00
直接アクセス	4.31	0.94
記号経由アクセス	4.60	1.00

のコストが明確になるように，関数=，+，-をインライン展開している．

```
(module m (import base))
(defvar x 0)
(defun incdec ()
  (do ((i 0 (+ i 1)))
      ((= i 1000))
    (do ((j 0 (+ j 1)))
        ((= j 1000))
      (setq x (+ x 1))
      (setq x (- x 1))
      ..............
      (setq x (+ x 1))
      (setq x (- x 1)))))
```

直接アクセスの場合は，記号を介するコストが削減され，実行時間が短縮されている．一方，記号経由アクセスの場合は，ポインタを1つたどるコスト（間接アクセススロットから内部束縛を保持する記号を求める処理）が加算され実行時間が増加している．このような実行時間の増加は，提案するモジュール機能の実装上やむをえないものであるが，公開する束縛を必要不可欠なものに限定するモジュラープログラミングのスタイルによって抑制される．

次に，関数呼び出しの性能評価を行う．以下の単純な加算を行うプログラムにおいて，関数 inc を実行し，plus1 の関数呼び出しを 10^6 回行った．大域関数 plus1 は非公開束縛なので，記号やキャッシュを介さずに本体の C 関数を直接呼び出すコード生成が可能である．実行時間を**表2**に示す．なお，plus1 の関数束縛へのアクセスのコストが明確になるように，関数 = および + をインライン展開している．

```
(module m (import base))
(defun plus1 (n) (+ n 1))
(defun inc ()
  (do ((i 0 (+ i 1)))
      ((= i 1000))
    (do ((j 0 (+ j 1)))
        ((= j 1000))
      (plus1 0))))
```

直接アクセスの場合は，モジュール機能がない場合や記号経由アクセスの場合のようにポインタをたどらないので，実行時間が短縮されている．モジュール機能がない場合と記号経由アクセスの場合の実行時間に差がみられないのは，キャッシュの効果である．このキャッシュ機能は従来からあったものであるが，プログラムの実行中に関数束縛が更新されることは現実にはほとんどないことを考えれば，モジュール機能を追加した場合にも束縛を得るコストを削減するのに有効である．

8. まとめ

既存の Lisp 言語を拡張して容易に実現できるモジュール機能を提案した．この機能は，モジュールごとに固有の環境を提供し，さらにモジュールごとに固有の記号空間を提供するものである．モジュール間では（記号ではなく）束縛の可視性制御を行うので，名前空間ごとに柔軟な対応ができるうえ，モジュール間での束縛の衝突を完全に避けることができる．この基本原理により，マクロの束縛捕捉問題も自然に解決する．モジュール間でやりとりされる束縛は，その束縛が使用される前に設定していればよいので，モジュール機能を追加しても Lisp の対話性を損なうことがない．コンパイラにおけるコード生成では，束縛の可視性制御を利用し，束縛にアクセスするためのコストを削減する最適化が可能である．

今後の課題としては，提案したモジュールの形式的意味論を示すこと，モジュラープログラミングに関連したコンパイラ最適化のさらなる研究，およびモジュール機能に適したプログラミング環境の設計があげられる．

謝辞 有益なコメントをくださった査読者の方々に深謝する．

参考文献

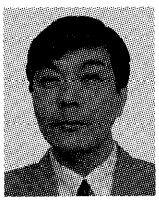
1) Bawden, A. and Rees, J.: Syntactic Closures, *Conference Record of the 1988 ACM Lisp and Functional Programming*, pp.86–95 (1988).
2) Clinger, W. and Rees, J. (Eds.): The Revised^4 Report on the Algorithmic Language Scheme, Technical Report, CIS-TR-90-02, University of Oregon (1990).

3) Curtis, P. and Rauen, J.: A Module System for Scheme, *Conference Record of the 1990 ACM Lisp and Functional Programming*, pp.13–19 (1990).
4) Davis, H., Parquier, P. and Séniak, N.: Talking about Modules and Delivery, *Conference Record of the 1994 ACM Lisp and Functional Programming*, pp.113–120 (1994).
5) Dybvig, R.K., Friedman, D.P. and Haynes, C.T.: Expansion-Passing Style: Beyond Conventional Macros, *Conference Record of the 1986 ACM Lisp and Functional Programming*, pp.143–150 (1986).
6) Felleisen, M. and Friedman, D.P.: A closer look at export and import statements, *Computer Languages*, Vol.11, No.1, pp.29–37 (1986).
7) ISO/IEC: Information Technology – Programming Languages, their environments and system software interfaces, Programming Language ISLISP, International Standard, Reference Number ISO/IEC 13816:1997 (E) (1997).
8) 伊藤貴康：LISP 言語国際標準化と日本の貢献，情報処理，Vol.38, No.10, pp.932–937 (1997).
9) Kohlbecker, E., Friedman, D.P., Felleisen, M. and Duba, B.: Hygienic Macro Expansion, *Conference Record of the 1986 ACM Lisp and Functional Programming*, pp.151–161 (1986).
10) 小宮常康，伏見明浩，湯淺太一：Weak Cons を用いたマクロ再定義時の自動再コンパイル，情報処理学会論文誌，Vol.36, No.6, pp.1407–1414 (1995).
11) MacQueens, D.: Modules for Standard ML, *Conference Record of the 1984 ACM Lisp and Functional Programming*, pp.198–207 (1984).
12) MacQueens, D.: An Implementation of Standard ML Modules, *Conference Record of the 1988 ACM Lisp and Functional Programming*, pp.212–223 (1988).
13) 森山崇元：TUTScheme におけるモジュール機能の設計と実現，卒業論文，豊橋技術科学大学 (1996).
14) Queinnec, C. and Padget, J.: A Detailed Summary of a Deterministic Model of Modules and Macros for Lisp, Technical Report, LIX/RR/90/01, École Polytechnique, Laboratoire d'Informatique, pp.212–223 (1989).
15) Sheldon, M.A. and Gifford, D.K.: Static Dependent Types for First Class Modules, *Conference Record of the 1990 ACM Lisp and Functional Programming*, pp.20–29 (1990).
16) Steele, Jr., G.L.: *Common Lisp the language*, Digital Press (1984).
17) Tung, S.H.: Interactive Modular Programming in Scheme, *Conference Record of the 1992 ACM Lisp and Functional Programming*, pp.86–95 (1992).
18) Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol.13, No.3, pp.284–295 (1990).

(平成 9 年 10 月 2 日受付)
(平成 10 年 7 月 3 日採録)

安本 太一（正会員）

1966 年生．1986 年詫間電波工業高等専門学校電子工学科卒業．1988 年豊橋技術科学大学情報工学課程卒業．1990 年同大学大学院工学研究科修士課程情報工学専攻修了．同年愛知教育大学教育学部総合科学課程総合理学コース数理科学選修助手．1997 年同大学助教授になり現在に至る．1998 年豊橋技術科学大学大学院工学研究科博士課程電子・情報工学専攻（社会人）修了．博士（工学）．プログラミング言語処理系，並列計算機に興味を持つ．日本ソフトウェア科学会，電子情報通信学会各会員．



湯淺 太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学理学研究科博士課程修了．同年同大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，超並列計算に興味を持っている．著書「Common Lisp 入門（共著）」，「Scheme 入門」，「C 言語によるプログラミング入門」他．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．