Weak Consを用いたマクロ再定義時の自動再コンパイル

小宮常康[†] 伏見明浩[†,☆] 湯淺太一[†]

Lisp処理系には，インタプリタとコンパイラの両方を備えた処理系が多いが，その両方を併用する場合は，マクロを使用した関数のコンパイルに注意を要する．インタプリタでは，マクロ展開はマクロ式の評価時に行うため，いつでも最新のマクロ定義が使われる．一方，コンパイラでは，コンパイル時に展開を行うため，コンパイル時におけるマクロ定義が使われる．そのため，マクロを使用した関数をコンパイルする場合は，必要なマクロがあらかじめ定義されていなければならない．またコンパイルされた関数では，マクロ式はすでに展開されているので，マクロを再定義してもその変更はコンパイルされた関数には反映されない．このようにコンパイラでは，マクロの定義時期やマクロの再定義によって，動作がインタプリタと変わってしまうことがある．そこで本論文では，関数とマクロの依存関係グラフを保持し，マクロの（再）定義時にそのマクロに依存する関数やマクロを自動再コンパイルまたは自動再定義することにより，コンパイラを使用した場合でもインタプリタと同じように動作させる方法を提案する．提案する方法では，weak consを用いることによって，不必要な再コンパイルを避けることができ，また不要な再コンパイル情報を捨てることができる．本方法は，本文で述べるように，インタプリタと同じ動作をさせることができない場合があるが，それはきわめて特殊な場合であり，実用上は問題ないと考えられる．

Automatic Re-Compilation on Macro Redefinition, by Making Use of Weak Cons

TSUNEYASU KOMIYA,[†] AKIHIRO FUSHIMI [†,☆] and TAIICHI YUASA [†]

Some Lisp systems have both an interpreter and a compiler. In such systems, compilation of those functions that contain macro calls sometimes causes a trouble. The interpreter always uses the latest macro definition since macro forms are expanded at execution time. On the other hand, the compiler expands macros at compilation time and thus uses those macro definitions that exist at compilation time. Therefore, when a function is compiled, all macros that the function references must have already been defined. If a macro is re-defined, the change cannot be reflected to compiled functions since macro forms have already been expanded. As the result, the behavior of a compiled function may be different from an interpreted function, depending on the time of macro definition. In this paper, we propose a mechanism to guarantee that a compiled function behaves in the same way as the interpreted function. A Lisp system with this mechanism keeps a dependence graph of functions and macros. When a macro is defined (or re-defined), the system automatically re-compiles or re-defines those functions and macros that depend on the (re-)defined macro. The proposed mechanism can avoid redundant re-compilation and discard unnecessary information for re-compilation, by making use of weak conses. Although some compiled functions may still behave in a different way as the interpreted function, such functions rarely appear in realistic programs and thus cause no realistic problem.

1. はじめに

Lisp処理系には，インタプリタとコンパイラの両方を備えた処理系が多い．一般にそのような処理系ではインタプリタによって実行される関数とコンパイルされた関数は相互に呼び出すことができる．そのためコンパイラを備えたLisp処理系では，インタプリタでプログラムを対話的に開発・デバッグし，動作の安定した関数についてはコンパイルして高速化を図るといった形態でプログラムの開発が行われることが多い．

インタプリタとコンパイラを併用する場合，マクロを使用した関数のコンパイルには注意を要する．Lispにおけるマクロ式(第1要素がマクロ名であるリスト)は，マクロ名に付随するマクロ展開関数によってマクロ式を別の式に変換（マクロ展開）し，その結果を評価する．インタプリタでは，このマクロ展開をマクロ

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology
* 現在，NTN株式会社
Presently with, NTN Corporation

式の評価時に行うが，コンパイラでは，コンパイル時にマクロ展開を行う．すなわち，インタプリタではいつでも最新のマクロ定義が使われるのに対し，コンパイラではコンパイル時におけるマクロ定義が使われる．そのため，マクロを使用した関数をコンパイルする場合は，必要なマクロがあらかじめ定義されていなければならない．またコンパイルされた関数では，マクロ式はすでに展開されているので，マクロを再定義してもその変更はコンパイルされた関数には反映されない．

このようにコンパイラでは，マクロの定義時期やマクロの再定義によって，動作がインタプリタと変わってしまうことがある．これを避けるには，以下の方法がある．

1. 開発段階でのコンパイラの使用を避ける
2. マクロを定義または再定義したときにそのマクロを使用している関数をコンパイルし直すか，もしくは定義し直してインタプリタで動作するようにする

開発段階でのコンパイラの使用を避けるのはプログラムの実行時間が遅くなるため開発効率が低下する場合がある．また，後者の方法もユーザ自身が行うのでは間違いやすく，また煩わしいためLisp処理系のすぐれた開発環境が損なわれてしまう．さらに後者の方法では，関数オブジェクトが他のLispデータ構造から指されている場合，その関数を再コンパイルまたは再定義しても，Lispデータ構造は変更前の古い関数オブジェクトを指したままになる．

そこで本論文では，関数とマクロの依存関係グラフを保持し，マクロの（再）定義時にそのマクロに依存する関数やマクロを自動再コンパイルまたは自動再定義することにより，コンパイラを使用した場合でもインタプリタと同じように動作させる方法を提案する．またScheme[1)~3)]処理系であるTUTScheme[4)]上に実現する方法について述べる．提案する方法では，weak cons[4)~8)]を用いることによって，不必要な再コンパイルを避けることができ，また不要な再コンパイル情報を捨てることができる．

本論文では，まずプログラムをインタプリタとコンパイラで実行する場合の相違について具体的に述べ，次に自動再コンパイル機能の基本原理とweak consによる依存関係グラフについて述べる．そして，実現方法について述べ，最後に提案する方法では対応することができない問題について述べる．

2. インタプリタとコンパイラの相違

以下の式を例に，関数をコンパイルする場合にマクロと関数の定義順序によって動作が異なることを示す．なお，ここではマクロの記法としてCommon Lisp[9)]の表記法を用いる．

```
(defmacro foo (n) …)
(defun bar (x) … (foo y) …)
```

ここで関数barの本体に現れる（foo y）はマクロ式である．マクロfooを定義した後に関数barをコンパイルすると，fooはマクロ名であることがわかるので，(foo y)はマクロ式としてマクロ展開された後コンパイルされる．しかし，マクロfooを定義する前に関数barをコンパイルすると，その時点でfooはマクロ名として定義されていないため(foo y)は関数呼び出しの式としてコンパイルされる．また，マクロfooを定義した後に関数barをコンパイルしても，マクロfooが再定義されると問題が生じる．関数barの本体に現れるマクロ式(foo y)はすでにマクロ展開されているので，マクロfooの変更は関数barに反映されない．一方，インタプリタでは，実行時にマクロ展開を行うため，上記のような問題は生じない．

3. 自動再コンパイル機能の基本原理

前章の問題を解決するには，マクロを（再）定義したときにそのマクロを使用している関数やマクロをコンパイルし直すか，インタプリタ上で動作するように定義し直せばよい．そこで本論文では，関数とマクロの依存関係グラフを保持し，マクロの（再）定義時にそのマクロに依存する関数やマクロを自動再コンパイルまたは自動再定義することにより，コンパイラを用いた場合でもインタプリタと同じように動作させる方法を提案する．自動再コンパイルを行うか自動再定義を行うかはコンパイル時間とコンパイルされた関数の実行時間とのトレードオフなので必要に応じてユーザがどちらかを選べばよい（ただし自動再定義は本章の終りで述べるように実現が困難な場合がある）．以下では提案する方法の基本原理について述べる．

再コンパイルまたは再定義が起こり得るのは，以下の場合である．

- マクロを定義した場合
- マクロを再定義した場合
- マクロを関数として定義し直した場合（つまりマクロ名と同じ名前の関数を定義した場合）

これらのいずれかが行われると，そのマクロを使用している関数とマクロの再コンパイルまたは再定義を行

う．さらにマクロを再コンパイルまたは再定義した場合は，その再コンパイル（再定義）されたマクロを使用している関数とマクロも再コンパイル（再定義）する必要がある．再コンパイル（再定義）を必要とする関数とマクロは，関数とマクロの依存関係グラフから検索する．依存関係グラフとは，関数やマクロ間の依存関係を表したものであり，ある関数またはマクロがどの関数やマクロに依存するかを調べることができる．ここで，マクロaがマクロbに依存するとは，マクロaの展開形の中にマクロbが現れるのではなく，マクロaの展開関数がマクロbの展開関数を呼び出すことを意味するものとする．この情報の記憶は関数やマクロの（再）定義時に行う．

図1と**図2**に関数とマクロの依存関係グラフの例を示す．図中においてf1～f3は関数，mとm1～m5はマクロである．破線の矢印は関数またはマクロが関数を呼び出すことを表し，実線の矢印は関数またはマクロがマクロを呼び出すことを表している（呼び出す側→呼び出される側）．関数を再コンパイル（再定義）した場合，（直接的または間接的に）その関数を呼び出す関数やマクロは再コンパイル（再定義）する必要がない．従って，実線の矢印だけが再コンパイルを必要とする依存関係を表している．図1において，マクロm3が再定義されたとすると自動再コンパイルは以下のように行われる．

1. マクロm3を参照しているマクロm2がコンパイルされたものならm2を再コンパイル（再定義）する．
2. 次にマクロm2を参照している関数f2がコンパイルされたものならf2を再コンパイル（再定義）する．
3. f2は関数なのでマクロm1および関数f1は再コンパイル（再定義）する必要がない．

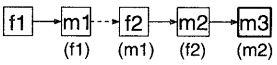


図1 関数とマクロの依存関係グラフ（1）
Fig. 1 A dependence graph of functions and macros (1).

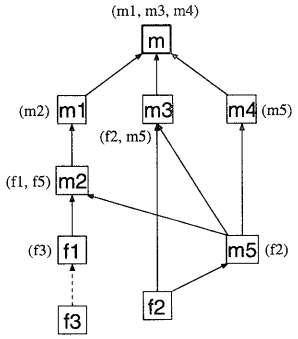


図2 関数とマクロの依存関係グラフ（2）
Fig. 2 A dependence graph of functions and macros (2).

一方，図2の依存関係の場合は，再コンパイル（再定義）の順序に注意する必要がある．図2中のすべての関数とマクロがすでにコンパイルされているとすると，mの再定義によって，マクロm1～m5および関数f1とf2が再コンパイル（再定義）される．このとき，深さ優先でm1→m2→f1→m3→f2→m5→m4という順序で再コンパイル（再定義）してはいけない．f2はm3とm5を参照しているのでm3とm5を再コンパイル（再定義）した後にf2を再コンパイル（再定義）する必要がある．すなわち，ある関数（マクロ）を再コンパイルする際には，その関数（マクロ）が依存するすべてのマクロを先に再コンパイルしなければならない．いま，マクロまたは関数aがマクロbに依存するときa>bと書くことにする（これは図1と図2の実線の矢印に相当する）．この＞が半順序関係であれば，＞のもとでトポロジカル・ソートを行うことにより正しい再コンパイル（再定義）の順序を求めることができる．しかし＞が半順序であるためには，依存関係グラフの実線の矢印で表される部分グラフにループが含まれないようにしなければならない．そこで＞に以下の規則を付け加える．

規則 ループしたマクロの依存関係a＞aを発見した場合は，a＞aが成り立たないものとして扱い，警告を発する．

これによりマクロ展開関数の再帰呼び出しによるループが取り除かれる．この規則は自動再コンパイルが停止しなくなるのを防ぐために必要である．

一方，自動再コンパイル機能が行う再コンパイルと再定義は，ユーザが行うコンパイルや定義とは動作が異なる．ユーザが行うコンパイルは，新たな関数またはマクロ展開関数を生成するが，自動再コンパイル機能が行う再コンパイルでは新たに生成せずに既存の関数オブジェクトの中身（コンパイルされたコード列やコード列の開始アドレスなど）を更新する．再定義の場合は，新たに関数オブジェクトを生成せずに既存のオブジェクトの型をインタプリタで実行する関数に変更する必要がある．従って，再定義はコンパイルされた関数オブジェクトをインタプリタで実行する関数オブジェクトに変更可能な処理系でなければ実現することができない．

このように関数オブジェクトを生成するのではなく更新することによって，関数オブジェクトが別のLisp

データ構造から指されている場合（つまり複数のLispデータ構造が1つの関数オブジェクトを共有している場合）でもマクロ定義の変更を正しく反映させることができる．自動再コンパイル機能を用いずにユーザ自身がコンパイルし直したり定義し直す方法では，新たに関数オブジェクトを生成するため，Lispデータ構造が指す古い関数オブジェクトにマクロ定義の変更を反映させることができない．

4. Weak consによる依存関係グラフの保持

関数とマクロの依存関係である依存関係グラフで保持する情報の内容は，図1に示した括弧内の記号（自分を呼び出す関数やマクロ）である．いま，図1において，マクロm1に依存する関数f1をユーザが再定義したとする．すると，依存関係グラフは**図3**のようになる．m1に依存するのは，再定義して新たに生成された関数f1だけではなく，再定義前の関数（f1′とする）もm1に依存する．なぜならば，このf1′は別のLispデータ構造から指されていて，まだ使われているかもしれないからである．この再定義前の関数がまだ使われているかどうかは，依存関係グラフの更新時や再コンパイル時には効率よく知ることはできない．しかし，依存関係グラフに再定義前の関数を必ず残すようにすると，メモリ効率の点や余分な再コンパイルが行われてしまう点で問題がある．そこで，依存関係グラフには再定義前の関数を必ず残すようにし，weak consを用いて依存関係グラフを管理する．weak consは，consと同様にcar部とcdr部から成るデータである．weak consのcdr部はconsのそれと同じ働きを持つがcar部の働きはconsと異なる．もしcar部のデータが他のデータから参照されていなければ，car部のデータはごみ集めによって回収される（**図4**参照）．weak consを使えば，再定義前の関数がどこからも参照されていなければ，ごみ集めによって再定義前の関数を回収することができる．このweak consによる依存関係グラフの具体的な実現方法については，5.3節で述べる．

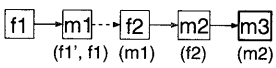


図3 f1を再定義したときの依存関係グラフ
Fig. 3 A dependence graph after f1 was re-defined.

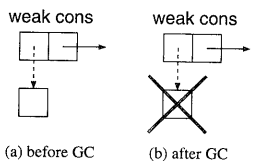


図4 Weak cons
Fig. 4 Weak cons.

5. 実現方法

本章ではScheme処理系TUTScheme[4]上への実現方法について述べる．TUTScheme上の本機能は，Schemeによって記述されており，処理系に大きく依存するのは関数オブジェクトを操作する関数（関数オブジェクトの中身を更新する関数）だけである．

5.1 インクリメンタル・コンパイラ

インクリメンタル・コンパイラを有する処理系では，トップ・レベルから入力されたプログラムは，直ちにコンパイルされ，実行される．この方式では，会話的なプログラム開発・デバッグ環境を損なわずにプログラムの高速化を達成することができるため，多くの処理系で採用されている[4),6)~8),10),11)]．しかし，これらの処理系は通常インタプリタを持たないため，マクロの定義時期やマクロの再定義によって生じる問題を回避することができない．そのためこれらの処理系では，本論文で述べる自動再コンパイル機能が特に有効である．また，インタプリタベースの処理系の中には，一度マクロを展開すると元のマクロ式を破壊的にその展開形に置き換えてしまうものもある[12)]．一度置き換えられると，マクロを再定義してもその変更は反映されないが，本機能を用いることによってこの問題を解決することができる．

本章で述べる自動再コンパイル機能の実現で用いたScheme処理系TUTSchemeは，インクリメンタル・コンパイラを有するコンパイラベースの処理系である．入力されたプログラムは，コンパイラによって特定のハードウェアに依存しないバイトコード列に変換された後，バイトコードインタプリタで実行される．

5.2 TUTSchemeのマクロ

Schemeにおけるマクロという概念は，Revised[4] Report on Scheme[3)]で述べられているが，IEEEのScheme仕様[2)]では定義されていない．

TUTSchemeのマクロはRevised[4] Report on Schemeのマクロとは異なり，Common Lisp[9)]のマクロに似た仕様である．その実現はマクロ名を与える記号の値としてマクロ展開関数を格納している．またマクロの定義はトップ・レベルでのみ可能である．

5.3 依存関係グラフの実現

関数とマクロの依存関係である依存関係グラフは表を用意して管理することも考えられるが，TUTSchemeではすべての記号が属性リストを持っているのでそれを利用する．保持する情報の内容は，図3に

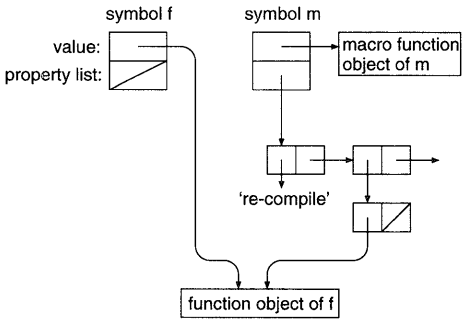


図5 Cons による依存関係グラフの表現
Fig. 5 A representation of dependence graph, by making use of cons.

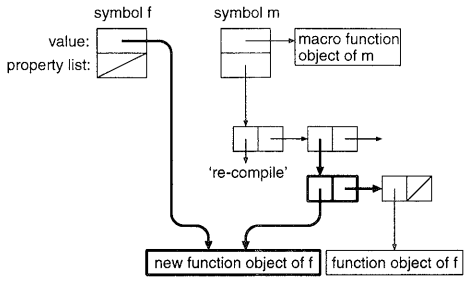


図6 関数 f を再定義した場合の依存関係グラフの表現
Fig. 6 A representation of dependence graph after redefinition of f.

示した括弧内の記号（自分を呼び出す関数やマクロ）であり，ユーザが再定義する前の古い関数も保持する．具体的には，関数またはマクロを（再）定義したときに，その定義の中に現れる関数呼び出しの式（マクロ式）の関数名（マクロ名）の属性リストに re-compile をキー（属性）として今定義された関数（マクロ展開関数）をリストの形で保持すればよい．**図5** に関数 f がマクロ m を呼び出す場合の状態を示す．

しかし，4章で述べたように通常のリストで依存関係グラフを保持すると，関数を再定義した場合に，不要な情報を保持してしまうことがあるので，メモリ効率の点や余分な再コンパイルが行われてしまう点で問題がある．例えば，図5において，マクロ m を呼び出す関数 f を再定義すると**図6**のようになる．記号 m はその依存関係として新しく生成された関数オブジェクトだけでなく，再定義前の古い関数も記憶する．もし，この古い関数がどこからも参照されていなければ，これを依存関係グラフに残しておくのは無駄である．

そこで，呼び出し側の関数とマクロ展開関数の保持に cons によるリストではなく，weak cons によるリストで保持するようにする．weak cons を使えば，**図7**のように，再定義前の関数がもしどこからも参照されていなければ，ごみ集めによって再定義前の関数を

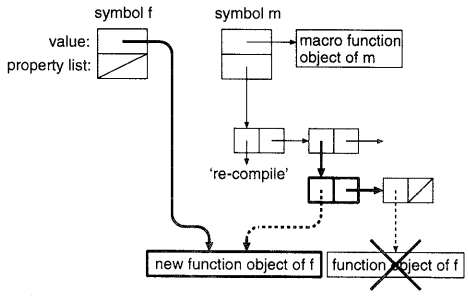


図7 Weak cons による依存関係グラフの表現
Fig. 7 A representation of dependence graph, by making use of weak cons.

回収することができる．また，回収された関数を指していた weak cons はもはや必要がない．そこで依存関係のリストを探索中にそのような weak cons を発見したらそれをリストから外すようにする．

5.4 再コンパイル

TUTScheme の関数オブジェクトは

- 関数名
- ソース・プログラム（ラムダ式）
- コンパイルされたコード列
- コード列の開始アドレス
- コード列から参照される Scheme オブジェクト列
- 環境（局所変数の値からなるリスト）
- 引数情報

から構成される．再コンパイルは，関数オブジェクトからソース・プログラムを取りだし，コンパイルされたコード列，開始アドレス，コンパイルされたコード列から参照される Scheme オブジェクト列，引数情報を生成する．そして関数オブジェクトのコード列，開始アドレス，コンパイルされたコード列から参照される Scheme オブジェクト列，引数情報を新しいものに置き換える．

再コンパイルする関数オブジェクトが関数閉包の場合（空でない環境を持つ場合）は，関数閉包の生成時に用いた環境情報（局所変数名など）のもとでソース・プログラムをコンパイルする．従って関数閉包の再コンパイルを行うには，その情報を関数オブジェクトに持たせる必要がある．そして環境が空のときと同様に関数オブジェクトのコード列，開始アドレス，コンパイルされたコード列から参照される Scheme オブジェクト列，引数情報を新しいものに置き換える．関数閉包が持つ環境は再コンパイルによって関数閉包に閉じ込めるべき変数が変更されることがあるので，関数は，その関数が参照するしないに関わらずその時点の環境のすべてを含んでいる必要がある．

しかしTUTSchemeでは,閉じた変数の参照のコストはそうでない変数の参照のコストより高い．TUTSchemeでは，閉じた変数とset!によって副作用を受ける局所変数の値はヒープに，それ以外の局所変数の値はスタックに置かれる．副作用を受ける局所変数の値をヒープに置くのは，call/ccによる継続の生成に備えるためである．TUTSchemeにおける継続の生成は，スタックの内容をヒープにコピーするため，副作用を受ける局所変数の値をスタックに置くとヒープ上の継続にはその変更が反映されない．そこで副作用を受ける可能性のある局所変数の値はヒープに置き，副作用が起きてもそれがヒープ上の継続にも及ぶようにしている．TUTSchemeの局所変数の参照には次の4通りがある．

1. 関数閉包が持つ環境内の変数に対してその関数閉包が行う参照
2. 関数閉包が持つ環境内の変数に対してその関数閉包の外で行われる参照
3. set!によって副作用を受ける局所変数に対する参照
4. 上記以外の参照

1の参照は環境リストの何番目かによって行う．2と3の参照はスタック上のオフセットから値を格納するセルを指すポインタを得て，そのポインタから値を間接的に参照する．4の参照はスタック上のオフセットによって値を直接的に参照する．もし関数が生成時点の環境のすべてを含まなければならないとすると，閉じ込められない変数に対しても上記の1または2の参照を行わなければならず，参照の効率が悪くなる．そのため，TUTSchemeへの実現では関数閉包の再コンパイルをサポートせずに，局所変数の参照の効率を優先した．

一方，継続の生成に備えるため，変数の束縛をスタックを一切用いずにヒープ上でリストとして実現するような処理系では，関数閉包の再コンパイルをサポートすることによって局所変数の参照の効率が悪くなることはない．

5.5 自動再コンパイル機能

TUTSchemeのトップ・レベルは，インタプリタベースの処理系と同様にread-eval-printループを形成している．関数evalはおおまかに書くと次のように実現されている．

```
(define (eval exp)
  (bcload-eval (compile exp)))
```

ここで(compile exp)はexpをコンパイルし，コンパイルされたプログラム(S式)を返す．関数bcload-evalはコンパイルされたプログラムを読み込んでバイトコード列を生成し，それをバイトコードインタプリタで評価して結果を返す．自動再コンパイル機能では，このevalを次のように変更する．

```
(define (eval exp)
  (if (not (definition? exp))
      (bcload-eval (compile exp))
      (let* ((dname (definition-name exp))
             (pre (macro? dname))
             (result (bcload-eval
                       (compile exp))))
        (if (function?
              (get-symbol-value dname))
            (begin
              (record-caller
                (get-symbol-value dname)
                (get-receiver-name exp))
              (if (or (macro? dname) pre)
                  (for-each
                    re-compile
                    (lookup dname)))))
        result)))
```

ここでdefinition?は式が大域変数の定義（define式）またはマクロ定義かどうかを調べる述語，definition-nameは定義式によって定義される変数名を返す関数，macro?は引数の記号がマクロ名かどうかを調べる述語，get-symbol-valueは記号の値を返す関数，get-receiver-nameは式中に現れる関数呼び出し式（マクロ式）の関数名（マクロ名）をリストにして返す関数である．record-callerは引数に与えられた関数（マクロ展開関数）を依存関係グラフに登録する関数である．この際，すでに登録されていた場合は登録しない．関数lookupは再コンパイルすべき関数やマクロを再コンパイルすべき順序でリストにして返す．re-compileは前節で述べた方法で再コンパイルを行う関数である．

上に示したevalでは，まずexpが大域変数の定義*またはマクロ定義かを調べる．もしそうならば，expをコンパイルして大域変数の値を定義した後に，定義された変数の値が関数またはマクロ展開関数かどうかを調べる．もしそうであれば，定義中に現れる関数呼び出し式（マクロ式）の関数名（マクロ名）の属性リストに今生成した関数（マクロ展開関数）をrecord-callerによって登録する．そして，今定義したものがマクロであるかまたは定義する前はマクロであった

* Schemeでは関数は変数の値として定義される．

(つまりマクロ名と同じ名前の関数を定義した場合)ならば再コンパイルを行う．再コンパイルを行う際は，再コンパイルされる関数名またはマクロ名をユーザにわかるように表示する．こうすることによって，ユーザはプログラム・ファイル中の関数とマクロの危険な定義順序を発見することができ，自動再コンパイル機能を使わずに動作させたときの誤った定義順序によるエラーを防ぐことができる．そして最後にexpの評価結果を返す．

6. 提案する方法の問題点

この章では，プログラムをインタプリタと提案する方法で実行する場合の相違について述べる．

本論文で提案する方式は，依存関係 > が半順序であれば正しく再コンパイルすることができる．> が半順序関係を満たさなくなるのは，図1〜図3の依存関係グラフの実線で表される部分グラフにループが含まれる場合である．3章で述べた > の定義より，ループのある依存関係 a>…>a となるのは，aがマクロであり，途中に関数の呼び出しがないときだけである．つまり，マクロ式の展開時にマクロ展開関数を再帰的に呼び出すマクロを定義するとループが作られる．以下にそのようなマクロ定義の例を示す．

```
(defmacro foo (n)
  (if (zero? n) 0 (bar)))
(defmacro bar () (foo 0))
```

3章で述べた依存関係 > を半順序とするための規則を設けているのは，このようなマクロ定義の自動再コンパイルが停止しなくなるのを防ぐためである．このようなマクロの使い方は現実には皆無に近いが，もし誤って上記のようなマクロを定義すると，発見の難しい誤りがつくられる．しかし本機能では，依存関係 > を半順序とするための規則によりマクロの定義時に警告が発せられるため，ユーザはこのような発見の難しい誤りを防ぐことができる．なお，上記の例は，自動再コンパイル機能のないインクリメンタル・コンパイラを用いても，マクロ foo の定義に現れる(bar)またはマクロ bar の定義に現れる(foo 0)が関数呼び出し式としてコンパイルされてしまうため，正しくコンパイルすることができない．

一方，次のマクロ定義

```
(defmacro my-or exps
  (cond ((null? exps) #f)
        ((null? (cdr exps)) (car exps))
        (else
          `((lambda (temp)
              (if temp
                  temp
                  (my-or ,@(cdr exps))))
            ,(car exps)))))
```

のように展開形の中に再帰的に現れるマクロ式は，マクロ展開関数が再帰的に呼び出されるわけではないので正しくコンパイルすることができる．

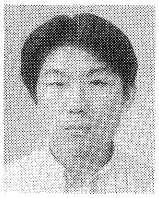
7. おわりに

本論文では，コンパイラを用いたときに生じる関数とマクロの定義順序の問題ついて述べ，この問題を解決する方法を提案した．この方法を用いると，インタプリタのように関数とマクロの定義順序を気にせずに済む．特に，インクリメンタル・コンパイラを有する会話的なプログラム開発・デバッグ環境を提供するコンパイラベースの処理系において有効である．また，weak consを用いることで不必要な再コンパイルを避けることができ，また不要な再コンパイル情報を捨てることができることを示した．

参考文献

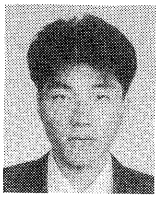
1) 湯淺太一：Scheme入門，岩波書店 (1991).
2) IEEE Standard for the Scheme Programming Language, IEEE (1991).
3) Clinger, W. and Rees, J., eds. : Revised^4 Report on the Algorithmic Language Scheme, *MIT AI Memo 848b*, MIT (1991).
4) 湯淺太一ほか：TUTSchemeのマニュアル，豊橋技術科学大学湯淺研究室 (1994).
5) 寺田 実：拡張wpによる記憶管理，情報処理学会論文誌，Vol. 34, No. 7, pp. 1610-1617 (1993).
6) MIT Scheme Reference Manual, Edition 1.0 for Scheme Release 7.1 (1991).
7) Kelsey, R. and Rees, J.: Scheme 48 (1992).
8) Rees, J. A., Adams, N. L. and Meehan, J. R.: *The T Manual Fifth Edition—Pre-Beta Draft—*, Computer Science Department, Yale University (1990).
9) Steele, G. L. et al.: *Common Lisp : The Language*, pp. 143-152, Digital Press (1984).
10) Bartley, D. H. and Jensen, J. C.: The Implementation of PC Scheme, *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pp. 86-93 (1986).
11) ILOG: ILOG Talk Reference Manual Version 3.0, ILOG Inc. (1994).
12) Jaffer, A.: Manual for Scm (1993).

(平成6年8月 3 日受付)
(平成7年4月14日採録)



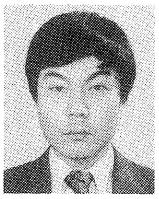
小宮　常康

1969年生．1989年育英工業高等専門学校電気工学科卒業．1991年豊橋技術科学大学工学部情報工学課程卒業．1993年同大学院工学研究科情報工学専攻修士課程修了．現在，同大学院工学研究科システム情報工学専攻博士課程に在学中．記号処理言語と並列プログラミング言語に興味を持ち，現在は並列記号処理言語に関する研究に従事．



伏見　明浩

1971年生．1991年沼津工業高等専門学校卒業．1993年豊橋技術科学大学工学部情報工学課程卒業．1995年同大学院工学研究科情報工学専攻修士課程修了．現在，NTN(株)勤務．ネットワークによる並列処理に興味を持っている．



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．同年京都大学数理解析研究所助手．1987年豊橋技術科学大学講師．現在，同大学教授．理学博士．記号処理と超並列計算に興味を持っている．著書「Common Lisp入門（共著）」ほか．ソフトウェア科学会，電子情報通信学会，IEEE, ACM各会員．