Lisp を用いた TEX のプリプロセッサ

岩崎 英哉

東京大学 大学院工学系研究科 情報工学 / 計数工学専攻

概要

TEX が提供する様々な機能は，マクロを適切に定義して活用することができるが，大多数の利用者にとっては，マクロを自在に定義するのは難しい作業である．この問題点を克服するため，本論文は，マクロ (関数) の定義に Lisp を利用し，文書中に埋め込まれた Lisp の S 式を，Lisp の評価規則に従って展開して置換文字列を生成するようなシステムを提案する．このシステムは，Lisp の S 式部のみを展開して通常の TEX 文書ファイルを生成するような，TEX の前処理として実現する．利用者は文書展開マクロの定義に Lisp の関数を用いることができ，従来の TEX コマンドとの併用も可能である．提案するシステムにより，TEX マクロによる定義が難しいような機能を，Lisp を用いた関数で容易に柔軟に定義することが可能となった．

1 はじめに

今日，科学技術系の文書作成では，TEX 系の清書系 [3][4] が広く用いられている．TEX 自身は豊富な機能を提供する一種の “言語” であり，その機能を有効に活用するために，利用者は様々なマクロを定義・利用する．ところが多くの利用者にとって，マクロを自在に定義して TEX の提供する機能を使いこなすのは，大変難しい．TEX では，“口” と呼ばれる部分がトークン列を解析してマクロ展開などを行い，その結果を受けて “胃” が代入・定義などの処理を行う．口と胃がうまく協調して作動するように TEX を制御する必要がある点が，TEX マクロの定義を難しくしている要因の一つである．

TEX 文書のように，記号，文字列．数値などを同時に，しかも柔軟に扱う必要のあるものに対しては，記号処理的なアプローチが有効と考えられる．そこで本論文では，マクロ (関数) の定義に Lisp を利用し，文書中に埋め込まれた関数呼出しなどの S 式を Lisp の評価規則に従って展開して置換文字列を生成するようなシステムを提案する．

Lisp-based Preprocessor for TEX Documents
Hideya Iwasaki
Department of Information Engineering / Mathematical Engineering, University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113–8656 Japan
iwasaki@ipl.t.u-tokyo.ac.jp

提案するシステムの特徴は，次のようにまとめられる．

- 本システムは，TEX によらずとも置換文字列を決定できるようなマクロに関して，TEX よりもプログラムが容易な Lisp を用いた定義を可能とする．したがって，ページ番号に代表されるように，TEX の実行時でなければ処理できない部分は TEX に処理を委ねるが，それ以外の部分については本システムを使うこともできるように，役割分担する．
- 本システムは通常の TEX の前処理として位置付ける．本システムの起動により，Lisp の S 式 (関数呼出し部など) だけを展開して通常の TEX 文書ファイルを生成する．その次段階として，TEX 等の起動により DVI ファイルを生成する．
- 本システムは，次段階で起動する TEX 系に対して何ら制限を加えないので，plain TEX [3]，LATEX [4] などを利用者の好みにより選択できる．すなわち，Lisp による記述は，様々なマクロパッケージ・スタイルファイルと共存でき，両者を併用することによる効果的な文書作成が可能である．
- 本システムを実現するために用いる Lisp に特別な機能は必要ないため，様々な Lisp システムの上に本システムを構築することが可能である．我々は，Lisp の一方言である UtiLisp(University of Tokyo Interactive Lisp) [6][5] を用いた．

本論文では，UtiLisp を用いて実現した本システムを UTITEX と呼ぶ．図 1 に，UTITEX を用いた文書作成過程を示す．利用者が用意する文書ファイルは，通常の TEX コマンド，および，UTITEX が評価・展開する (関数呼出しなどの) S 式の両方を含む．このファイルを入力として，UTITEX は S 式部だけを処理して，通常の TEX コマンドだけを含むような文書ファイルに変換する．その後 TEX ・LATEX などの処理により．DVI ファイルを生成する．

以後本論文では，Lisp によって定義され，その呼出しが Lisp 処理系によって処理され置換文字列が生成されるものを “Lisp マクロ” あるいは単に “関数” と呼ぶ．したがって，Lisp マクロは，UtiLisp の関数あるいはマクロとして定義される．また，TEX，LATEX などの世界におけるマクロと組込みの制御シーケンスを合わせて “TEX コマンド” と呼ぶ．

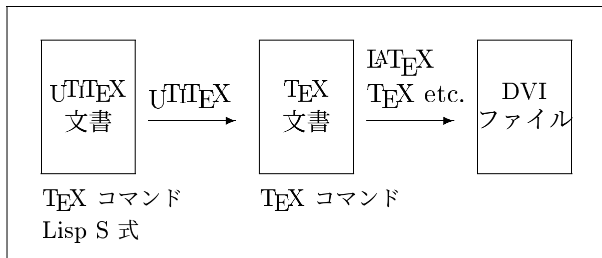


図 1: UTiTEX を用いた文書作成過程

2 基本設計

2.1 Lisp 評価系による展開

UTiTEX では，文書中で Lisp 評価系に展開を委ねる部分を `\(`*name arg1* ··· *argn*) と記述する．このように書くと，S 式 (*name arg1* ··· *argn*) が Lisp 評価系に渡され，その評価結果で `\` から始まる全体が置きかえられる．その他の部分と `%` (コメント開始文字) 以降には UTiTEX は一切手を加えず，そのままの文字列を出力する．

Lisp 評価系による `\(`*name arg1* ··· *argn*) の部分の展開・評価は，周辺の TEX コマンド文脈 (環境) にかかわりなく行われる．たとえば，LaTEX の verbatim 環境の中であっても，`\(` で始まる S 式は UTiTEX によって評価展開される．

2.2 出力関数

置換に使われる “評価結果” には，評価値 (返り値) そのものを使う方法と副作用による出力を使う方法が考えられる．Lisp の評価が一般的には作用的順序 (applicative order) である点を考慮すると，後者の副作用による出力を使う方法は置換文字列の生成と相性が悪い．

たとえば `f` を通常の Lisp 関数とする時，`\(f x y)` の展開を考える．後者の方法では “`x` の評価による出力”，“`y` の評価による出力”，“`f` への引数の適用による出力” という順番で置換文字列に展開され，関数呼出し式の内側・左側にあるものほど置換文字列のはじめの方に出現するという，直観とは異なる展開結果になる．さらに後者の方法では，引数がグループの場合とそうでない場合とで挙動が異なるような関数を定義することは簡単にはできない．なぜなら，引数の展開結果は標準出力に出されるだけなので，引数がグループに展開されたかどうかを容易には判断できないためである．

以上のような理由により，UTiTEX では前者の方法，すなわち評価値そのものから展開結果の置換文字列を生成する方法を採用する．具体的には，評価値 (x とする) を次のような関数 `output` によって標準出力に出力した結果を置換文字列とする．

- x が `nil` の時，何も出力しない．
- x が文字列 (string) の場合，その文字列を `princ` する．(中身だけが出力される．)
- x がシンボル `nl` ならば，単に改行する．
- x がその他のシンボルの場合，そのシンボルの印字名を `princ` する．
- x がその他のアトムならば，`prin1` する．
- x が `group` で始まるリストの場合，`{` を出力後，(`cdr` x) の各要素を左から順に `output` し，最後に `}` を出力してグループを閉じる．
- x が `quote` で始まるリストの場合，(`cadr` x) を `prin1` する．
- x が上であげたような特別なリストでなければ，その構成要素を再帰的に `output` する．

以後本論文では，$exp \to value$は，S 式 exp を Lisp 処理系で評価した値が $value$ であることを，$value \Rightarrow string$は，値 (S 式) $value$ を関数 `output` を用いて出力して得られた置換文字列が $string$ であることを表すものとする．これらを続けて $exp \to value \Rightarrow string$と，また，途中の $value$ に注目しない場合にはこれを省略して$exp \to\Rightarrow string$とも書く．さらに，$string$ の清書出力結果が out であることを$string \rightsquigarrow out$と表すこともある．

以下に，`output` による出力例をいくつか示す．

```
123
⇒ 123

("\begin{center}" xx 12 "\end{center}")
⇒ \begin{center}xx12\end{center}

("\begin{center}" nl "\cs" (group a1)
 (a2 " " a3) nl "\end{center}")
⇒ \begin{center}
  \cs{a1}a2 a3
  \end{center}

("\some" "cs")
⇒ \somecs
```

上のように `output` を定める利点の第一は，構造化されたデータ (`group` で始まるリストなど) を作成し，その構造に忠実な出力を得ることができる点である．これは，S 式という構造を文書作成に用いることの最大の利点でもある．利点の第二は，対応のとれない `{` `}` など，構造化されないものを文字列の形で保持して出力することができる点である．これは “構造” という捉えかただけでは定義が難しいような Lisp マクロを定義する際に威力を発揮する．この特別な場合として，上で掲げた最後の例のように，二つの出力をつなげて一つの TEX コマンドを生成することも可能である．この例では，`\some` と `cs` から `\somecs` という TEX コマンドを出力している．

2.3 TEX コマンドの呼出し

Lisp マクロには，`output` に渡すデータを値として生成するような定義を与える．その定義中に出現する

TEX コマンドに対する実引数を，Lisp による展開結果で与えたいことは，頻繁に起こる．たとえば，引数値の倍の空白を縦方向にあける Lisp マクロ vs は，TEX コマンド \vskip を用いて次のように定義できれば便利である．

```
\(defun vs (s) (\vskip (* s 2) "pt"))
```

そのため，UTiTEX では，\vskip などの \ で始まるシンボルはすべて TEX コマンドとみなし，引数をすべて評価したリストの先頭に自分自身を cons するような結果を返すような UtiLisp マクロの定義を与えている．その結果，たとえば \(vs 20) は

```
\(vs 20)
→ (\vskip 40 "pt")
⇒ \vskip40pt
```

という過程を経て \vskip40pt という置換文字列となる．

2.4 Lisp 評価系利用の切替え

本論文などのような，\(という文字列が随所に出現する文書の作成時など，Lisp 評価系による展開のオン / オフを自由に制御したい場合もある．そこで本システムでは，特別な意味を持つ注釈を，評価系利用の切替に用いる．

具体的には，%- で始まる注釈があれば，それ以降，\(で始まっても Lisp 評価系を呼び出さず，そのまま出力する．(その結果，TEX が処理するファイル中に \(が出現することになる．) これが有効なのは，行頭から %+ で始まる注釈が出現するまでである．%- の入れ子はできない．

3 関数の記述例

3.1 Lisp による計算

いくつかの数値とそれらの数値間の計算結果を表形式にまとめるような場合，その計算を Lisp による評価時に行うことができる．図 2 は，ある処理系におけるインタプリタとコンパイラでの実行時間を比較する表を作る例である．Lisp マクロの tableline は，インタプリタとコンパイラに要した時間を受けとり，これらの比を計算 (簡単のため整数上の計算とする) した上で，次のようにして LATEX の tabular 環境中の行を生成する．

```
\(tableline "(f 10)" 40 390)
→ ("\verb!" "(f 10)" "!&" 40 "&" 390
   "&" 10 "\\ \hline" nl)
⇒ \verb!(f 10)!&40&390&10\\ \hline
```

図 2 (a) の記述を与えると，最終的には図 2 (b) に示すような出力が得られる．

TEX 系だけを用いて文書を作成する場合の多くは，文書作成とは別個に計算を行い，その結果を文書内に書き入れるであろう．そのため，データ更新の際など，誤りが混入する可能性が少なくない．それに対し，UTiTEX で上のような Lisp マクロを用いると，計算の元になるデータだけを関数呼出しの引数として与えるので，計算間違いや入力誤りによる誤り混入の可能性が大幅に減少すると考えられる．

```
\(defun percent (x y) (// (* x 100) y))
\(defun tableline (e x y)
  `("\verb!" ,e "!&" ,x "&" ,y "&"
    ,(percent x y) "\\ \hline" nl))
\begin{tabular}{|c|c|c|c|}      \hline
Expr&Compiler&Interpreter&Ratio(\%)\\
\hline\hline
\(tableline "(f 10)" 40 390)
\(tableline "(g 20)" 411 2512)
\end{tabular}
```

(a) Lisp マクロの定義と呼出し

Expr	Compiler	Interpreter	Ratio(%)
(f 10)	40	390	10
(g 20)	411	2512	16

(b) 清書結果

図 2: tabular 環境の計算を行う Lisp マクロ

もう一つ別の例を示す．LATEX の picture 環境は，描画を文書中に入れる簡便な手段として広く用いられている．picture 環境では，描画面の大きさ，線・箱などの描画対象の位置等を絶対座標として与える必要がある．

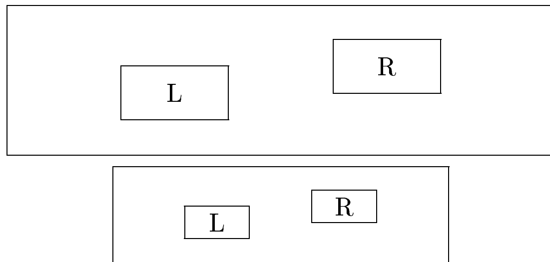
描画面の大きさとの相対的な関係から描画対象の位置や大きさを定めることが多いことを考えると，これらを Lisp による計算に委ねれば，描画面の大きさの変更に柔軟に対応させることができる．図 3 に，横方向 w，縦方向 h の大きさの描画面に対して，横 $w/5$, 縦 $2h/5$ の長方形を 2 個，座標 $(w/5, h/5)$ と $(3w/5, 2h/5)$ の場所に描く例を示す．

pic は描画面の大きさを引数として受けとり，その大きさに即した描画コマンドを生成する．この Lisp マクロにより，\(pic 200 50) は次のように展開され，置換文字列が生成される．

```
\(pic 200 50)
→ ("\begin{picture}"
    ("(" 200 "," 50 ")")
    ("(" 0 "," 0 ")")
    nl
    (\put ("(" 40 "," 10 ")")
          (group \framebox
                 ("(" 40 "," 20 ")")
                 (group "L"))
          nl)
    (\put ("(" 120 "," 20 ")")
```

```
\(defun rect (x y a b text)
  `(\put ,(cord x y)
    (group \framebox ,(cord a b)
           (group ,text))
    ,newline))
\(defun cord (x y) `("(" ,x "," ,y ")"))
\(defun pic (w h)
  (let ((a (// w 5)) (b (// (* 2 h) 5)))
   `("\begin{picture}" ,(cord w h)
     ,(cord 0 0) ,newline
     ,(rect a (// h 5) a b "L")
     ,(rect (// (* 3 w) 5) b a b "R")
     "\end{picture}")))
\begin{center}
\leavevmode\(pic 200 50)
\smallskip\par
\leavevmode\(pic 120 30)
\end{center}
```

(a) Lisp マクロの定義と呼出し



(b) 清書結果

図 3: picture 環境の座標を計算をする Lisp マクロ

```
            (group \framebox
                   ("(" 40 "," 20 ")")
                   (group "R"))
            nl)
      "\end{picture}")
     nil
  ⇒ \begin{picture}(200,50)(0,0)
     \put(40,10){\framebox(40,20){L}}
     \put(120,20){\framebox(40,20){R}}
     \end{picture}
```

図 3 (a) の最終的な清書結果を図 3 (b) に示す．Lisp 評価系が座標を計算するため，Lisp マクロ pic に与える引数，すなわち描画面の広さに応じた描画が可能となった．

3.2 制御構造

TEX で繰返しなどの制御構造を記述するには，独特の技法が必要であったが，UTiTEX では Lisp によって提供される制御構造を利用すればよい．その例として，チェスの駒配置 [2] を記述する．

Lisp マクロ White は，駒の配置の記述 (いくつでも可) をリストとして与えると，各配置に \WPiece をかぶせた形に展開する．ここで，\WPiece は TEX コマンドとして別途定義されているものとする．駒の配置は，“駒の種類を表す英字”，“横軸の座標 (a∼h)”，“縦軸の座標 (1∼8)” から成る文字列で，駒の種類を省略した場合のデフォルト値は P とする．以下に展開例を示す．

```
\(White '("Ke1" "Qd2" "e2" "f4"))
→⇒ \WPiece{K}{e1} \WPiece{Q}{d2}
    \WPiece{P}{e2} \WPiece{P}{f4}
```

```
\(defun White (xs)
  (mapcar xs #'WPieceOrPawn))
\(defun WPieceOrPawn (str)
  (let ((n (string-length str)))
   (cond
    ((= n 2)
     (\WPiece (group "P") (group str)))
    (t (\WPiece
        (group (substring str 0 1))
        (group (substring str 1 n)))))))
```

図 4: 繰返しを用いた Lisp マクロ

図 4 に，White を UTiTEX で記述した例を示す．ここでは，mapcar を用いて，与えられたリストの各要素に下請け関数 WPieceOrPawn を適用しているが，再帰的に定義してももちろん構わない．WpieceOrPawn は文字列の長さを調べ，駒が省略されているか否かを判断している．

3.3 省略可能引数の利用

Lisp マクロに対する引数としてテキストを与える場合には，次のようにして，そのテキストを文字列として渡す．

```
\(somefn "テキスト ")
```

しかしテキストが数行に渡る場合などは，上のようにするよりも，

```
\(somefn)
数行に渡るテキスト
END
```

のように，somefn への引数値を入力ファイルから読む込むようにできる方が，読みやすさの点でも望ましいであろう．ここで END は，somefn への引数であるテキストの終わりを示す目印である．

このような Lisp マクロは，省略可能な引数を使って容易に実現できる．図 5 に，本論文でよく用いている，タイプライタ書体のテキストを枠で囲む Lisp マクロの概略を示す．box-verb は，通常は引数を省略して呼び出す．引数が省略された場合のデフォルト値として，関数 verblines が呼び出される．この関数呼出しは，END が出現するまで入力ファイルから行単位

```
\(defun box-verb ((x (verblines "END")))
  `(,(box-begin)
    "\begin{verbatim}" nl ,x "\end"
    "{verbatim}" nl ,(box-end)))
\(defun verblines (until)
  (do ((x (readline) (readline)) (res))
      ((string-equal x until)
       (nreverse res))
      (push x res) (push 'nl res)))
\(box-verb)
(defun White (xs)
  (mapcar xs #'WPieceOrPawn))
END
```

(a) Lisp マクロの定義と呼出し

```
(defun White (xs)
  (mapcar xs #'WPieceOrPawn))
```

(b) 清書結果 (外枠も清書結果の一部)

図 5: 引数を入力ファイルから読む Lisp マクロ

の読み込みを行う．box-verb は，この値の前後に，LaTeX の verbatim 環境の開始と終了，box 変数への代入操作 (box-begin により生成)，最後に周囲を枠で囲む操作 (box-end により生成) を挿入する．

3.4 属性の利用

UTiTEX では，文書ファイルの処理中に得られた情報を，シンボルの属性として蓄積することができる．その例として，文書中の他の場所への参照関係を解決する例をとりあげる．

図 6 に，節番号，節タイトルなど，節に関する情報を記述し，後に参照する例を示す．関数 section の第一引数は節の名前，第二引数は節情報を保持するラベル (シンボル) である．section の呼出し時に第二引数のラベルが与えられれば，節の種類・番号・タイトルをそのラベルの属性値に登録する．

節情報を参照する関数は labno と labtext である．これらは，前方の (すでに存在している) ラベルの参照時は，登録してある属性値をそのまま返すが，後方の (未出現の) ラベルを参照する場合には，参照関係を解決できないので，2 パス目を必要とする旨を意味するフラグ again を真にし，元の関数呼出し形を結果とする．UTiTEX は，文書ファイルの 1 パス目の処理終了時にフラグ again が真であれば，未解決の参照を解決するために 2 パス目の処理に自動的に入る．図 6 (b) に →⇒ が二つあるのは，2 回のパスそれぞれの結果を示している．2 パス目でも解決できない，すなわち解決不可能な参照に対しては，“???” と出力する．

節情報の他，図表情報・参考文献情報などについて

```
\(setq secno 0 again nil)
\(defun section (name (lab))
   (setq secno (1+ secno) ssecno 0)
   (cond ((and lab (symbolp lab))
          (putprop lab name 'labtext)
          (putprop lab 'sec 'kind)
          (putprop lab secno 'labno)))
   `(\section (group ,name)))
\(defun labno (x) (linfo x 'labno))
\(defun labtext (x) (linfo x 'labtext))
\(defun linfo (lab p)
   (cond ((get lab 'kind) (get lab p))
         (again "???")
         (t (setq again t)
            `("\" '(,p ',lab)))))
```

(a) Lisp マクロの定義

```
\(section "はじめに" 'intro)
\(labno 'design) 節で述べるように..
\(section "設計" 'design)
\(labno 'intro) 節「\(labtext 'intro)」..
→⇒ \section{はじめに}
    \(labno 'design) 節で述べるように..
    \section{設計}
    1 節「はじめに」..
→⇒ \section{はじめに}
    2 節で述べるように..
    \section{設計}
    1 節「はじめに」..
```

(b) 使用例と展開結果

図 6: シンボルの属性を利用する Lisp マクロ

も，シンボルの属性を利用して，同様の処理を行うことができる．このような参照関係の処理は，LaTeX も aux ファイルを介して行うが，UTiTEX では S 式データで情報を保持するので，扱いが容易であり拡張性にも富んでいる．また，LaTeX の参照処理は，利用者による LaTeX の再起動が必要なのに対し，UTiTEX による参照関係だけを含む文書ならば，Lisp 評価系による 2 パス目の処理は UTiTEX によって自動的に行われる．

3.5 文字列処理関数による解析

Lisp を用いる大きな利点のひとつは，豊富な文字列処理関数を利用しての文字列の解析が可能である点である．その例として，Lisp マニュアル中の，次のような形式の関数説明の頭書きを考える．

(putprop *sym value name*)
sym の属性 *name* の値を *value* に設定し，*value* を返す．

```
\(defun fdescr (arg)
  (do ((xs arg (cdr xs)) (x) (res (cons "\verb+(+" nil)))
      ((atom xs) (nreverse `("\verb+)+" . ,res)))
      (setq x (car xs))
      (push (cond ((= (sref x 0) (character "~")) (group "\it " (substring x 1)))
                  ((eq x '/...) "$\cdots$")
                  (t `("\verb+" ,x "+")))
            res)
      (or (null (cdr xs)) (push "\verb+ +" res))))
```

(a) Lisp マクロ fdescr の定義

```
\(fdescr '(putprop ~sym ~value ~name))
→⇒\verb+(+\verb+putprop+\verb+ +{\it sym}\verb+ +{\it value}\verb+ +{\it name}\verb+)+
⇝ (putprop sym value name)

\(fdescr '(and ~arg1 /... ~argn))
→⇒ \verb+(+\verb+and+\verb+ +{\it arg1}\verb+ +$\cdots$\verb+ +{\it argn}\verb+)+
⇝ (and arg1 ··· argn)
```

(b) fdescr の使用例，展開結果と清書結果

図 7: 関数説明の頭書きを生成する Lisp マクロ

ここで，タイプライタ体は “定数”，イタリック体は利用者によるデータが与えられる部分を意味する．このような関数説明の頭書き部 (putprop *sym value name*) を LaTeX で記述するひとつの方法は，

```
\verb+(putprop +{\it sym}\verb+ +
{\it value}\verb+ +{\it name}\verb+)+
```

とすることであるが，UTiTeX では，図 7 (a) に示す関数 fdescr を用いて簡単に記述できる．fdescr は ~ のつく引数をイタリック体に，/... を $\cdots$ に，その他を \verb+ 文字列 + に展開する．図 7 (b) にその使用例・展開結果・清書結果をあわせて示す．

同様の文字列解析を行う Lisp マクロで筆者が重宝しているのは，関数プログラムのテキストをほぼそのままの形で与えると，適切な書体選択・段付け・空白の挿入・縦方向の揃え等を行った清書結果を生成する関数 functional-program である．図 8 (a) にその使用例，(b) に清書結果を示す．

Lisp マクロ functinoal-program は，その呼出しから END までの間を清書すべき関数プログラムと解釈する．END までを処理対象とする機構は，第 3.3 節で説明した省略可能引数で実現している．与えるプログラムには，二項演算子を表す \oplus 等の TeX コマンドを除けば，清書指令は基本的には含まれない．functional-program は，プログラム内の各構成要素 (関数名，変数名，予約語，演算子等) の出現位置 (左から何カラム目か)，名前などを解析し，必要な TeX コマンドに置きかえる．その際，上の行と段付けが合うように (たとえば，図 8 (b) のプログラム [1] の where 直後の *bs* と次行の *as* の先頭が揃うように)，適切量の空白を挿入する．

このような関数プログラムを清書する Lisp マクロは，約 300 行の UtiLisp プログラムで実現することができた．

4 実現

UTiTeX による処理の主ループは，次のような単純なものである．

- 入力ファイルの文字が \(と連続して出現した時に限り (から始まる S 式を一つ読み込み，評価後，出力ファイルに output する，すなわち，(output (eval (read))) を行う．ただし，注釈の中，あるいは，%- の後で %+ が出現する前は，この限りではない．
- それ以外の場合には，入力ファイルの文字をそのまま出力ファイルに書き出す．

実際，本システムの主部のプログラム行数は，わずか 100 行程度にすぎない．

このようにして入力ファイル全体を処理した後，第 3.4 節で述べた，未解決の参照関係が存在することを示すフラグ (again) の値を調べ，真の場合には，2 パス目として上と同じ処理を再度行う．2 パス目が終了すれば，参照は解決しているか，未解決のため “???” と置換されているかのどちらかである．

入力ファイルの文字が \ の時，次の文字が (か否かによって挙動を変える必要があるため，本実現では一文字先を “覗き見” する UtiLisp の組込関数 tyipeek を用いている．同等の機能があれば，本論文で提案するシステムは，UtiLisp 以外の Lisp の上にも構築する

```
\(functional-program)
fact 0 = 1
fact (n+1) =(n+1) * fact n
END
\medskip\par
\(functional-program)
h [] c = g_1 c
h (x:xs) c = k (x,c) \oplus h xs (c \otimes g_2 x)
END
\medskip\par
\(functional-program)
diff (\oplus) (\otimes) k g_1 g_2 xs c = reduce (\oplus) (map k as) \oplus g_1 b
  where bs ++ [b] = map (c \otimes) (scan (\otimes) (map g_2 xs))
        as = zip xs bs
END
```

(a) Lisp マクロ `functional-program` の使用例

$$
\begin{aligned}
&\mathit{fact}\, 0 = 1\\
&\mathit{fact}\,(n+1) = (n+1) * \mathit{fact}\, n
\end{aligned}
$$

$$
\begin{aligned}
&h\,[\,]\,c = g_1\ c\\
&h\,(x : \mathit{xs})\,c = k\ (x, c) \oplus h\ \mathit{xs}\ (c \otimes g_2\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ \mathit{xs}\, c = \mathsf{reduce}\ (\oplus)\ (\mathsf{map}\ k\ \mathit{as}) \oplus g_1\ b\\
&\quad \mathsf{where}\ \ \mathit{bs} +\!\!+ [b] = \mathsf{map}\ (c \otimes)\ (\mathsf{scan}\ (\otimes)\ (\mathsf{map}\ g_2\ \mathit{xs}))\\
&\qquad\qquad \mathit{as} = \mathsf{zip}\ \mathit{xs}\ \mathit{bs}
\end{aligned}
$$

(b) 清書結果

図 8: 関数プログラムを清書する Lisp マクロ

ことが可能である．

本システムを実現する際に問題となり得るのは，第 2.3 節で説明した，\ で始まるシンボルに対し，引数をすべて評価したリストの先頭に自分自身を cons するような結果を返す Lisp 関数の定義を与える方法である．UTiLaTEX では，UtiLisp がシンボルを intern する関数を自由に設定できることを利用し，intern の関数において，一文字目が \ のシンボルに対しては上述のような定義を自動的に与えるようにしている．

5 議論

提案するシステムでは，文書中の S 式を Lisp 評価系で処理する際，% から行末までを注釈と解釈する以外は，S 式周辺の TEX コマンド・環境などを一切考慮せず，機械的な評価・置換を行う．したがって，コメント文字を別の文字に設定するなど，文字のカテゴリコードを変更するような文書に対しては，うまく対応できない場合がある．しかしながら，カテゴリコードの変更が LaTEX の verbatim 環境のような場所でしか行われないような "標準的" な文書に対する利用に関しては，これで十分であると考えている．

本システムで重要なのは，後段の TEX 系の利用に関して制限を加えないという点である．LaTEX では文章中の数式を \(と \) で囲んで記述したり，あるいは，たとえば ifthen パッケージのように \(を TEX コマンドとして用いているものもあるが，第 2.4 節で述べた %-，%+ を用いて Lisp 評価系を制御すれば，UTiLaTEX とこれらの機能を共存させることができる．

利用者は，ある展開結果を実現するマクロを，Lisp マクロで定義してもよいし TEX コマンドで定義してもよい．さらに，Lisp マクロの中で TEX コマンドを呼び出せるので，両者を併用してひとつの機能を実現することもできる．その意味で，本システムは，利用者に対してマクロ定義に関する幅広い選択の可能性を与えている．

本システムの弱点は，エラー処理にある．UTiLaTEX を用いた文書清書中に起こり得るエラーは，

- Lisp 評価系の評価中に起こるエラー
- UTiLaTEX の後段の TEX・LaTEX 等の処理中のエラー

の二通りに分けられる．

前者の部類のエラーに関しては，UtiLisp のブレークパッケージにおける Lisp インタプリタとの対話的なデバッグによって，原因究明作業を行うことが可能である．通常の TEX におけるデバッグと比較すると，Lisp によるデバッグ環境を享受できるのは，本システムの大きな利点である．また，はじめは後段の TEX に

よる処理とは切り離して Lisp マクロのプログラミングだけに専念し， Lisp マクロによる展開動作がある程度確認された後で TEX と組み合わせるという，段階的なプログラム・文書開発も可能である．

後者の TEX 実行時エラーでは，報告される行番号が，元の UTiTEX 文書ファイルにおける行番号ではなく，UTiTEX による展開後の文書のものになる．この点はエラー箇所・原因の特定の際の障害になり得る．この問題点に対処するため，本システムでは，ファイル *xxx*.tex の UTiTEX による展開後の (後段の TEX の入力となる) 文書を (先頭に _ をつけて) _*xxx*.tex というファイルに保持して残すようにしている．このようにすると，エラーの起こった箇所が明確になると同時に， Lisp マクロによる展開結果の置換文字列を確認することができ， Lisp マクロのデバッグにも役立つ．

6 おわりに

本論文は， Lisp の関数を用いて TEX 文書のマクロを定義し， Lisp 評価系によってマクロ展開を行うようなプリプロセッサシステムを提案した． Lisp が提供する S 式という汎用的なデータ構造と豊富な関数，作用的順序という理解しやすい評価機構により，柔軟性の高いシステムを構築することができた．またいくつかの実用的な例を通して，その有効性を確認することができた．

今後の課題としては， Emacs Lisp など幅広く利用されている Lisp 上での実現があげられる．これにより，様々な計算機環境上で本システムを利用することが可能になる．さらに，汎用的な Lisp マクロを拡充するとともに，エラー処理・デバッグ環境を充実させることも重要な課題である．

参考文献

[1] Adachi, S., Iwasaki, H. and Hu, Z.: Diff: A Powerful Parallel Skeleton, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA'2000*) (to appear).

[2] Eijkout, V.: *TEX by Topic*, Addison Wesley, 1992.

[3] Kunth, D. E.: *The TEXbook*, Addison Wesley, 1984.

[4] Lamport, L.: *A Document Preparation System* LATEX — *User's Guide & Reference Manual*, Addison Wesley, 1986.

[5] 田中哲朗: SPARC の特徴を生かした UtiLisp/C の実現法, 情報処理学会論文誌, Vol.32, No.5, pp.684–690 (1991).

[6] Wada, E.: History of UtiLisp Hacking, *Journal of Information Processing*, Vol.13, No.3, pp. 276–283 (1990).