TEX による言語処理系の実装

白田靜哉[a)]

概要：学術文書に広く用いられている LATEX では，実装基盤である TEX がチューリング完全であるため，文書中に任意の計算を埋め込むことができる．しかし，TEX は組版というドメインに特化した言語であるため，その文法や評価規則は汎用的な計算を記述することに適したものではない．我々は，この記述性の問題を解決するために，LISP 処理系を TEX 上に実装した．この言語（*LISP on TEX*）は組み込み言語でありながら，TEX とは独立した文法と評価規則を持つ．また，TEX と LISP の間にインターオペラビリティを確保している．本論文では，上記の要件を達成する LISP on TEX の実装手法を紹介する．

キーワード：TEX, LISP, 言語処理系

1. 背景

文書中にに計算を埋め込むことによって，再利用性や整合性を高め，その生産効率を高めたいという需要がある．例として，HTML と JavaScript の組み合わせが挙げられる．また，プログラミング技術に明るくない人でも，本来は表計算ソフトウェアである Microsoft Excel を用いて，簡単な数値計算などを埋め込んだ文書作成を行うことがある．

本論文では LATEX を用いて作成されることが多い学術文書に注目した．LATEX においても，その実装基盤である TEX はチューリング完全であるため，任意の計算を埋め込むことができる．しかし，TEX は文法や評価規則が難解であり，簡単な計算を埋め込む場合であっても，その記述には困難を伴う．

この問題に対し，LATEX 文書に他のプログラミング言語を埋め込む様々な手法 [2], [3], [4], [6] が提案されてきた．既存の手法のほとんどが外部の言語処理系を呼び出すものである．これらの手法は TEX の外の環境に依存するため，環境構築や文書そのものの可搬性担保に問題がある．また，LuaTEX [3] のように TEX エンジン自身に他の言語処理系を組み込む方法も提案されているが，この手法は既存の TEX 資産，特に pTEX のような日本語 TEX エンジンの資産が利用できなくなるという問題がある．

このような課題に対し，本論文では TEX のマクロを用いて言語処理系を作成するという手法を提案する．本手法は，それ自体が TEX で作成されているため外部環境に依存することはない．また，TEX エンジンを選ばずに動作させることができる．本論文では，実際に LISP 処理系を TEX マクロのみで実装し，*LISP on TEX* と命名した．この言語は，TEX と独立した文法と評価規則を持ちつつ，TEX とのインターオペラビリティを確保している．この実装は TEX のアーカイブである CTAN に既に登録されており[*1]，TEX Live などの主要なディストリビューションに収録されている．

a) hak7a3@gmail.com

[*1] http://www.ctan.org/pkg/lisp-on-tex

本論文の貢献は次の通りである．

- TEX マクロのみで LISP 処理系 LISP on TEX を開発した（3 節）．これにより，TEX プログラミングの難解さ（2 節）に煩わされることなく，既存の TEX 資源を生かしつつ，計算を文書に埋め込むことができる．
- LISP on TEX の実装技法を示した（4 節）．TEX の難解さ，特に文字の取扱いと少ない計算資源の問題のため，汎用的な言語処理系の実装に必要であった，プログラミング上の工夫と配慮について具体的に示した．

2. TEX プログラミングとその難しさ

TEX はマクロや組み込み命令（プリミティブ）を駆使することによって，任意の計算を記述することが可能である．TEX プログラムの簡単な例を図 1(a) に示し[*2]，TEX プログラミングの概要を説明する．このプログラムは，与えられた引数の個数分だけ「急カーブ注意」の記号を出力するマクロ \outputCurve を定義する．実行例を図 1(b) に示す．

マクロ定義は \def などのプリミティブを用いる．\def マクロを束縛する識別子（コントロール・シーケンス）と引数の取り方，およびマクロの本体をとる．ここでは，一つの引数（#1）をとるマクロとして \outputCurve を定義している．

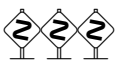
条件分岐は \ifnum などの専用のプリミティブを用いて \ifnum...\else...\fi などと記述する．ここで \ifnum は数値比較の結果により分岐を行うためのプリミティブである．

TEX には繰り返しを実現するための機構は標準で用意されていない．LATEX 文書に繰り返しを行うような計算を埋め込む場合，この例のように再帰呼び出しを用いるか，LATEX の実装で用いられているマクロである \@for などを利用する．

この程度ならば，TEX プログラミングはさほど難解でないように思える．しかし，TEX の字句解析やマクロ展開は複雑なルールを持っており，ある程度 TEX に習熟しないと使いこなすことが困難であることが知られている．Knuth は TEX ブック [5] において，TEX マクロの高度な使用法を紹介する章を「危険地帯」と呼び，「plain TEX[*3] を十分に使いこなすまでは以下の説明は読まないでほしい」としている．また，LATEX では，\def のようなプリミティブを直接使用されないよう，マクロ定義用のインタフェースを用意している．ここでは，TEX プログラミングにおける注意点のうち，改行文字の取扱いと展開制御を紹介する．

```
1  \font\manual=manfnt %フォント読み込み
2  \newcount\tmpInteger
3  \def\outputCurve#1{%
4    \tmpInteger=#1
5    \outputCurveInner
6  }
7  \def\outputCurveInner{%
8    \ifnum\tmpInteger>0
9      {\manual\char127}%記号出力
10     \advance\tmpInteger by -1
11     \outputCurveInner
12   \else
13     \relax
14   \fi
15 }
```

(a)\outputCurve の定義



(b)\outputCurve{3}の実行結果

図 1 「急カーブ注意」記号を出力するマクロ

2.1 改行文字の取扱い

TEX では，字句解析やマクロの展開において改行文字が特別な扱いを受ける．TEX プログラミングにおいて，ユーザはインデントなどの整形を自由に行うことができないという制限を受ける．その例を図 2 に示す．マクロ \showOddEven は引数が偶数ならば「偶数」，奇数ならば「奇数」と出力する．しかし，この定義ではその出力の前に不要な空白ができてしまう．例えば，「2 は \showOddEven{2}」と入力すると「2 は偶数」ではなく「2 は 偶数」とな

[*2] 同様のプログラム例が LuaTEX [3] にある．

[*3] （白田注）Knuth の定義した TEX マクロ集

```
1  \newcount\tmpCheck
2  \def\showOddEven#1{
3    \tmpCheck=#1
4    \divide\tmpCheck by 2
5    \multiply\tmpCheck by 2
6    \ifnum#1=\tmpCheck
7      偶数
8    \else
9      奇数
10   \fi
11 }
```

図 2 改行文字によって意図しない空白が出力される例

り，「は」と「偶数」の間に不要な空白が出力されていることがわかる．これは，1 行目末尾の改行が単語間の空白と認識されることに由来する．これを防ぐには，2 行目の末尾の改行を%を用いてコメントアウトさせればよい．ただし，すべての行末をコメントアウトすればよいというわけではないことに注意する．例えば，図 2 の TEX コードすべての行末に%を付けた場合，\showOddEven{2} は「奇数」と出力する．これは，TEX の字句解析と数値リテラル読み込みのタイミングに由来する．どの改行をコメントアウトすべきで，どの改行はそれをしてはならないかという判断には，ある程度 TEX に習熟する必要がある．

2.2 展開制御

TEX はマクロで計算を記述するため，適切なタイミングでマクロを展開する必要がある．このためには，\expandafter や \edef などの展開制御のためのプリミティブを使用する必要がある．展開制御は複雑な計算だけでなく，単純な計算でも必要になる場合がある．例えば，図 1(a) のプログラムの場合，11 行目にある \outputCurveInner の前に \expandafter を書いていないため，\outputCurve{5000}を実行するとエラー終了する．これには，TEX の入力スタックが関係している．TEX は，条件分岐の評価中，\fi の発見前にマクロを発見すると，現在の入力位置をスタックに積み，発見したマクロの展開を行う．入力スタックの大きさは標準で 5000 と定義されており，それを超えるとエラーが発生する．

3. LISP on TEX の概要

3.1 LISP on TEX の記述例

前節では，TEX による計算の記述が困難であることをいくつかの例を用いて示した．本節では，それに対比させる形で，LISP on TEX の概略を示す．

図 3(a) は，図 1(a) の TEX プログラムを LISP on TEX で実装したものである．図 3(b) はその実行例である．

LISP on TEX のコードは \lispinterp マクロの引数として与えることにより実行される．LISP on TEX では，LISP のシンボルに TEX のコントロール・シーケンスを用いている．また，整数型リテラルには，:10 のように接頭辞として:をつけるようにしている．整数型リテラルの読み込みは TEX によって行われるため，:"ABC0 のように 16 進数表現を利用することもできる．文字列型，すなわち TEX のトークン列のリテラルは’で囲む．ただし，文字列型リテラルは，波括弧（{，}）のバランスが取れている必要がある．

図 1(a) に示した TEX プログラムと異なり，LISP on TEX では改行の処理を意識する必要はない．また，評価規則は TEX と独立していて，call by value になっているので，展開制御を意識せずに通常の LISP プログラムの感覚で計算を記述できる．

LISP on TEX ではクロージャを使用できるため，純粋な TEX と違い，変数名の衝突によるバグの埋め込みを最小限に抑えることができる．また，one-shot-continuation [1] を実装しているため，TEX では困難であった大域脱出などの複雑な制御を行うことも容易である．固定小数点数を利用した計算もサポートしており，図 4 のようにマンデルブロ集合を描画することも可能 [*4] である．

LISP on TEX を，TEX マクロの一部として利用することも可能である．この性質により，TEX

[*4] 本論文に掲載するには長いため，プログラム自体は省略する．実行可能なコードは http://mirrors.ctan.org/macros/latex/contrib/lisp-on-tex/examples/fpnummodule-mandelbrot.tex にある．

```
\font\manual=manfnt %フォント読み込み

\lispinterp{
  (\define \outputCurveLoT
    (\lambda (\n)
      (\lispif (\> \n :0)
        (\begin
          (\texprint %TeXへの出力
            '{\manual\char127}')
          (\outputCurveLoT
            (\- \n :1)))
        ())))
}
```

(a)\outputCurveLoT の定義

(b)(\outputCurveLoT:3) の実行結果

図 3 LISP on TEX による「急カーブ注意」の出力

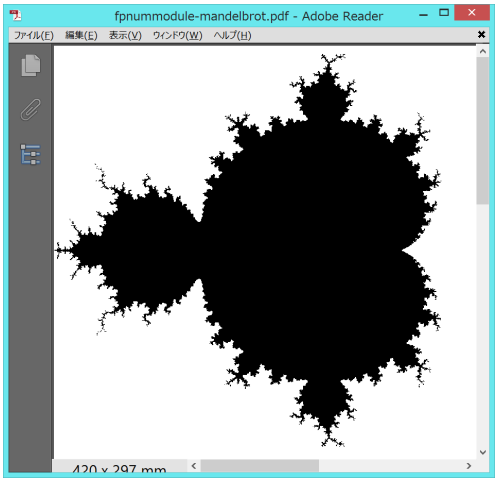


図 4 LISP on TEX によるマンデルブロ集合の描画

での記述が困難な部分を LISP で，TEX の方が記述性に優れている部分を TEX でというように，両者の特徴を生かして混在させてプログラムを記述することができる．図 5 はその例で，TEX マクロの引数として与えられた数に対して，その階乗を漢数字で出力する．TEX での記述が困難な階乗の計算には LISP を，漢数字化には pTEX のプリミティブである \kansuji を用いて，処理の分業を

```
\lispinterp{ % 階乗の定義
  (\define \fact (\lambda (\n)
    (\lispif (\= \n :0) :1
      (\* \n (\fact (\- \n :1))))))
}
\def\printFactKanji#1{%
  \lispinterp{%TeXマクロからLISPを実行
    (\texprint
     (\concat
       '\kansuji' % 漢数字変換はTeXで
       (\intTOstring (\fact :#1))
       '\relax'))
}}
```

(a)\printFactKanji の定義

四〇三二〇

(b)\printFactKanji{8}の実行結果

図 5 階乗の漢数字出力

行っている．

LISP on TEX の実装には，代入などの副作用をもつプリミティブを使用している．そのため，\edef の引数のように副作用を許容しないコンテキストでは使用することができない．

3.2 LISP on TEX の動作例

LISP on TEX がどのように動作しているかの概略を示す．例として，図 5(a) にある関数 \fact を，

```
\lispinterp{(\fact :3)}
```

と呼び出した場合を用いる．まず，LISP on TEX の字句解析により，(\fact :3) が次のような内部表現に変換される[*5]．

```
\@tlabel@cons{\carI\cdrI}
```

内部表現は，「型を示すコントロールシーケンス + {データ}」の形をしている．ここで，\carI は CONS セルの CAR に相当し，次のように定義されている．

```
\@tlabel@symbol{\fact}
```

[*5] 実際には，コントロールシーケンス名を通常の方法ではアクセスできない名前にしている．

\cdrI は CONS セルの CDR に相当し，次のように定義されている．

```
\@tlabel@cons{\carII\cdrII}
```

ここで，\carII は \@tlabel@int{3}，\cdrII は \@tlabel@nil{}と定義されている．

次に，評価器 \@eval によって，内部表現が評価される．これは，次のように呼び出される．

```
\@eval\@tlabel@cons{\carI\cdrI}{}
  \@return
```

\@eval は 4 引数をとる．第 1 および第 2 引数は評価対象となる内部表現，第 3 引数は評価する環境とグローバル環境との差分，第 4 引数は評価結果を格納するためのコントロール・シーケンスである．環境は，「シンボル + {値}」の列で表現される．例えば，{\x ↦ :1, \y ↦ :2} という環境は，

```
\x{\@tlabel@int{1}}
\y{\@tlabel@int{2}}
```

となる．

評価器により，まず \carI が展開，評価される．その結果，次で示されるクロージャの内部表現を得る．

```
\@tlabel@closure{
  {\n:\@@unused}
  {}
  \tlabel@cons{\carIII\cdrIII}
}
```

クロージャは 3 つのデータを持っている．一つ目は引数の取り方，二つ目は自身が定義された環境と大域環境との差分，三つ目が定義部である．

次に，CONS セルの残りの部分が，クロージャに与える引数として，評価および整形される．今回の例では，\@tlabel@int{3}\relax となる．そして，先に得たクロージャの内部表現から，定義部を評価すべき環境とグローバル環境の差分が次のように生成される．

```
\n{\@tlabel@int{3}}
```

そして，定義部を評価するために \@eval を次のように呼び出す．

```
\eval\@tlabel@cons{\carIII\cdrIII}
  {\n{\@tlabel@int{3}}}\@return
```

同様に評価が進み，最終的に \@return が

```
\@tlabel@int{6}
```

と定義され，3! = 6 が計算される．

4. LISP on TEX の実装技法

LISP on TEX では，その機能を実現するために様々な技法を用いている．本節では，LISP on TEX の実装で用いた TEX プログラミング技法を紹介する．

4.1 字句解析

4.1.1 リテラルに対する接頭辞の付与

3 節で述べたように，LISP on TEX では LISP のシンボルに TEX のコントロール・シーケンスを利用している．また，整数型だけでなく，すべての型のリテラルについて，その型に対応する接頭辞を要求する文法になっている．これは，字句解析を行うマクロの肥大化を抑えるためである．TEX には論理演算の概念がないため，例えば，「引数の文字が 1 から 9 ならば \foo を実行する」というマクロを記述する場合，

```
\if1#1%
  \foo
\else\if2#1%
  \foo
\else
  ...
\fi\fi ... \fi
```

のようになる．すなわち，接頭辞を使用しない文法の字句解析を TEX で実装しようとすると，冗長な表現によりマクロが肥大化する．その結果，字句解析処理の保守性や可読性が低下する．以上より，LISP on TEX では各型のリテラルに対して接頭辞を定め，シンボルに TEX のコントロール・シーケンスを採用することとした．

4.1.2 コメント記法

LISP on TEX では，コメントの記法を TEX と同

一にした．これは，TEX マクロからの LISP 呼び出しに対応するためである．TEX では文字ごとにカテゴリコードという文字の役割を示す数値が割り当てられており，動的にそれを変更することができる．この機能を用いれば，字句解析中だけ任意の文字を「その他一般文字」として取り扱わせることにより，TEX とは完全に独立した文法をもつ言語を実装可能 *6 である．しかし，カテゴリコードを変更するようなマクロを定義する場合，他のマクロの一部としてそのマクロを使用することが困難になる．すなわち，図 5 にある \printFactKanji のように，TEX マクロの一部として，LISP on TEX が使用できなくなる．

カテゴリコードの問題は，TEX のもつ大きな制限の一つとして知られており，例えば，LATEX では \verb を引数にとるようなマクロを定義できないといった問題を引き起こしている．カテゴリコードが規定する役割の中に，「コメントの開始」がある．すなわち，TEX のコメント記法を採用しないとするならば，カテゴリコードの変更が必須になる．それでは，TEX マクロの一部として LISP on TEX を使用できない．そのため，コメント記法に TEX の記法をそのまま採用した．

4.1.3 シンボル名の選定

TEX マクロとの親和性のため，LISP on TEX では，シンボル名にも注意を払っている．図 3(a) のソースコードを見るとわかるが，LISP on TEX では条件分岐に使用するシンボルに \if ではなく \lispif を用いている．これは，TEX において \if から始まるコントロール・シーケンスが特別な扱いを受けるためである．TEX では，条件分岐の評価時に，\if から始まるコントロール・シーケンスと \fi が対になっていなければならない．このとき，波括弧の対応関係などはすべて無視される．次の LISP on TEX を利用した TEX マクロを考える．

```
1  \def\exampleOfIf#1#2{%
2   \ifnum#1>0 % TeXの条件分岐
3     \lispinterp{
4      (\lispif ... )}% LISPの条件分岐
5   \fi
6  }
```

この \lispif を \if に変えたものを示す．

```
1  \def\exampleOfIf#1#2{%
2    \ifnum#1>0 % TeX側の条件分岐
3      \lispinterp{
4        (\if ... )}% LISP側の条件分岐
5    \fi
6  }
```

この場合，5 行目にある \fi が LISP 側の条件分岐にある \if と対応しているものとして解釈されるため，このマクロを展開すると TEX 処理系がエラーを起こしてしまう．このような問題を避けるため，LISP on TEX では，組み込み関数やスペシャルフォームに \if で始まるシンボル名を使用していない．

4.1.4 不必要な空白の除去

以上に述べた配慮によって，LISP on TEX では字句解析のほとんどを TEX に任せることができた．しかし，空白や改行の取扱いだけは特別に配慮する必要があった．ここでは，字句解析における「不必要な空白や改行」の除去について述べる．「不必要な空白や改行」とは，各構文要素の間にある空白や改行である．LISP on TEX では，TEX のマクロの引数読み込み時における空白や改行の挙動を用いて，不必要な空白や改行の除去を実現している．図 6 に LISP on TEX の字句解析および構文解析を行うマクロ \@lispread@main の一部を示す．このマクロは，

```
\@lispread@main\callback␣␣␣␣␣(
\foo␣:42)\@end@lispread
```

のように「\@lispread@main\callback 任意の S 式 \@end@lispread」の形で呼び出される．ここで，\callback は構文解析結果を受け取って何らかの処理を行うマクロ，\@end@lispread は入力の終端を示すコントロール・シーケンスである．また，「任意の S 式」には不必要な空白や改行が含まれたままである．しかし，TEX はマクロの引数

*6 ただし，空白や改行の扱いの問題で，インデントに依存するような言語など，空白に意味のある言語は実装できない．

```
\def\@lispread@main#1#2{%
  % CONSセルおよびnil
  \if\noexpand#2(%
    \def\@@next{\@lispread@cell#1}%
  % シンボルもしくは入力終了
  \else\ifcat\noexpand#2\relax
    % ...（省略）...
  \else\if\noexpand#2'% 文字列
      \def\@@next{%
        \@lispread@string#1%
        \@lispread@string@dummy}%
  % ... (他の分岐については省略) ...
  \fi\fi\fi\fi\fi\fi\fi\fi
  \@@next}
```

図 6 LISP on TEX の字句解析器（一部）

読み込み時，先頭の空白や改行を自動的に取り除くという性質[*7] がある．先の例では「(」の前にある空白や改行がすべて取り除かれる．すなわち，LISP on TEX における不必要な空白や改行は，字句読み込みの段階で自然に取り除かれる．

4.1.5 必要な空白の保存

このような TEX の性質により，空白文字や改行文字は自然に除去されるが，これは，「必要な空白」を除去されないようにするためには対策が必要であることも意味する．LISP on TEX において「必要な空白」とは文字列リテラル'...' に含まれる空白である．LISP on TEX では文字列リテラルの空白が消えてしまわないように，わざと不要なマクロを挿入している．先の説明に従えば，文字列リテラル'␣foo' の字句解析および構文解析は，

```
\@lispread@main\callback
'␣foo'\@end@lispread
```

のように呼び出され，図 6 より，

```
\@lispread@string\callback
\@lispread@string@dummy␣foo'%
\@end@lispread
```

と展開される．この時点で，一見不要に見える \@lispread@string@dummy が挿入されていることがわかる．ここで，\@lispread@string は次のように定義されている．

```
\def\@lispread@string#1#2'{%
  \endgroup\expandafter#1%
  \expandafter\@tlabel@string
  \expandafter{#2}}
```

この例では，

- #1 が \callback に，
- #2 が \@lispread@string@dummy foo に

置換される．結果，\@lispread@string@dummy が 1 回展開される．\@lispread@string@dummy は中身が空のマクロとして定義されているので，最終的に，今回の例は

```
\endgroup\callback
  \@tlabel@string{ foo}
```

と展開される．ここで，\@tlabel@string{␣foo} は'␣foo' の内部表現である．

なお，\@lispread@string@dummy を挿入しなかった場合，\@lispread@string の展開時，#2 の内容を決定するタイミングで␣foo の先頭の空白が除去されてしまう．

4.2 式の評価

LISP on TEX では，関数呼び出しのコールスタックに \begingroup と \endgroup によるグルーピングを用いている．すなわち，関数の呼び出しの度にグループがネストされる．TEX にはローカルな定義という機能があり，グループの中でのマクロ定義などは，特別な指定をしない限り外側のグループに影響しない．この機能は言語処理系におけるコールスタックの実装に便利で，スタックを TEX マクロで実装するよりも格段に実装コストが低い．

しかし，2.2 節で述べた \if...\fi の問題と同様に，グループのネストにも制限が設けられている．LISP on TEX では，グループのネストを抑えるため，末尾呼び出しの最適化を実装している．

図 7 に，その一例を示す．この例は，LISP on TEX における条件分岐である \lispif の実装部

[*7] 正確には空白文字や改行文字を読み込んだ後にできる空白トークンを削除する

分である．例えば，(\lispif /t :1 :2) を評価する場合，\@apply@if@normal は

```
\@apply@if@normal\@tlabel@int{1}
  \@tlabel@int{2}{..env..}\@return
```

と呼び出される．ここで，/t は真を表すリテラル，\@tlabel@int{1}，および \@tlabel@int{2}は :1，:2 の内部表現である．..env.. は式を評価するローカルな環境，\@return は評価結果を格納するコントロール・シーケンスで，呼び出し時には条件分岐部分の評価結果が格納されている．このマクロの呼び出し時には，呼び出し元でコールスタックへのプッシュが行われている．すなわち，このマクロ呼び出しは \begingroup と \endgroup に囲まれている．

ここから，\@apply@if@next が

```
\@apply@if@next\@tlabel@bool{t}
  \@tlabel@int{1}
  \@tlabel@int{2}
  {..env..}\@return
```

と呼び出される．ここで，\@tlabel@bool{t}は /t の内部表現である．

\@apply@if@next では，条件が真であれば \@@next を \@apply@if@next@t と，偽であれば \@@next を \@apply@if@next@f と定義し，\@@next を実行する．\@apply@if@next@t は，展開すると \@@tco を帰結部を評価するマクロに定義し，\@apply@if@next@f は，展開すると \@@tco を代替部を評価するマクロに定義する．ここで，\def ではなく \gdef を使用しているため，この定義はグループを超えてグローバルに定義されている．

\@apply@if@next の展開が終了したのち，\@apply@if@normal では \aftergroup\@@tco を実行する．\aftergroup はトークンを一つとり，それを現在のグループの終了時に置く機能を持ったプリミティブ [*8] である．すなわち，\@apply@if@next の呼び出し元で付加されている \endgroup の末尾にあとから \@@tco を配置し

[*8] 同一のグループ内で複数回使用した場合は，使用した順で置かれる．

```
1  \def\@apply@if@normal#1#2#3#4#5#6{%
2   \expandafter
3    \@apply@if@next#6#1{#2}#3{#4}{#5}#6%
4   \aftergroup\@@tco}
5
6  \def\@apply@if@next%
7     \@tlabel@bool#1#2#3#4#5#6#7{%
8    \let\@@next\relax
9    \ifx#1t%
10     \let\@@next\@apply@if@next@t
11   \else\ifx#1f%
12     \let\@@next\@apply@if@next@f
13   \else
14     \errmessage{LISP on TeX [if]:
15       Invalid boolean.
16       It's BUG. Please report.}%
17   \fi\fi\@@next{#1}{#2}{#3}
18                 {#4}{#5}{#6}{#7}}
19 \def\@apply@if@next@t#1#2#3#4#5#6#7{%
20   \gdef\@@tco{\@eval#2{#3}{#6}#7}}
21 \def\@apply@if@next@f#1#2#3#4#5#6#7{%
22   \gdef\@@tco{\@eval#4{#5}{#6}#7}}
```

図 7 LISP on TEX における末尾呼び出し最適化の例

たことになる．結果，コールスタックからのポップ，すなわちグループの終了時に \@@tco が展開され，末尾呼び出しの最適化が実現される．

グルーピングによるコールスタック実装における末尾呼び出しの最適化には，\aftergroup を用いる以外にも，あらかじめグルーピングの終了時にコントロール・シーケンスを置いておく方法も考えられる．すなわち，呼び出し時に

```
\begingroup
  ... 処理の中身 ...
\endgroup
```

として，処理の中身で \endgroup をフックするのではなく，

```
\begingroup
... 処理の中身 ...
\endgroup\@@tco
```

として処理の中身では \@@tco をグローバルに定義することでも，末尾呼び出しの最適化を実現できる．しかし，後者の実装の場合，自己評価型フォームなどの末尾呼び出しの最適化が必要でないときにも \@@tco を何もしないマクロとして定義する必要がある．\aftergroup によるフックの場合はそれが不要なため，潜在的なバグの混入をある程度抑止できる．そのため，LISP on TEX では \aftergroup を採用した．

5. 関連研究

LISP on TEX のように，LATEX で記述された文書に対して TEX ではないプログラミング言語で記述された計算を埋め込む研究がある．本節では，それらの研究について紹介する．

LuaTEX [3] は，TEX エンジン自体に Lua 処理系を組み込んだものである．ユーザは \directlua などの命令を用いて Lua コードを文書中に記述することができる．LuaTEX では TEX エンジンそのものの動作に影響を与えるようなプログラムが記述できるため，より柔軟な組版処理を実現できる．しかし，本質的に TEX エンジンが異なるため，既存の日本語 TEX 資産を LuaTEX では使用できない．LISP on TEX は純粋な TEX マクロのみで記述されているため，このようにエンジンの差異に注意する必要はない．

Carlisle らは，LATEX3 におけるプログラミングレイヤーとして expl3 [2] を作成している．expl3 は LISP on TEX と同様に TEX マクロのみで作成 [*9] されており，空白や改行の取扱いやマクロの引数に対する展開制御の問題を解決している．しかし，目的が TEX マクロの記述を簡略化することであるため，expl3 を使用する際には，TEX のマクロ展開に関する知識を要求される．実際，マクロの引数以外の部分では依然として展開制御の必要性がある．

Iwasaki [4] は，Emacs Lisp を用いて TEX の汎用的なプリプロセッサである UTIITEX を開発した．LISP on TEX と同じく Lisp を採用しているが，Lisp と TEX の間のインタフェースについてよく設計されている．目的がプリプロセスであるため，LISP on TEX のような TEX マクロ中に LISP コードを埋め込むことは想定されていない．

Parkin [6] は，ファイルを通して Perl と LATEX を連携させる手法 PerlTEX を提案した．PerlTEX は，LISP on TEX と同様，TEX エンジンへの非依存性と TEX マクロとの相互運用を実現している．また，Perl の全機能が使用できるため，記述性も高い．TEX の外部環境に依存するため，Windows など Perl が標準でインストールされていない環境では実行できないという問題があったが，Windows での主要 TEX ディストリビューションである TEX Live が Perl バイナリを同時に配布するため，この問題は解決されている．

6. 今後の展望

LISP on TEX の今後の課題として，ごみ集め（GC）とライブラリ関数の実装が挙げられる．

現在の LISP on TEX の実装では，CONS セルが生成されるごとにそれ固有のコントロール・シーケンスを生成する．これは再利用されることはない．しかし，TEX には使用できるコントロール・シーケンスの個数に限界がある．そのため，一定量以上の CONS セルを使用すると TEX 処理系自体が強制終了してしまう．これを回避するためには，不要になったコントロール・シーケンスの再利用，すなわち LISP on TEX における GC が必要になる．

現在の LISP on TEX における組込関数や特殊形式は，決して十分に用意されているとは言えない．特に，文字列操作に関しては，簡単な文字列結合程度しか提供していない．今後の機能拡張としてはフォーマット文字列への対応や，正規表現の実装が挙げられる．

7. まとめ

本論文では，TEX マクロにより実装された言語処理系である LISP on TEX を紹介した．この処理系は，LATEX 文書に埋め込むことが可能で，TEX マクロと LISP 処理系を相互に組み合わせること

[*9] ただし，ε-TEX による拡張を前提としているため，Knuth オリジナルの TEX では動作しない．

もできる．また，本論文では LISP on TEX の実装においてもっとも重要である字句の取扱い，および TEX の資源節約に関する知見を示した．

謝辞　本論文を執筆するにあたり，ご意見をいただいた佐藤重幸氏，岩崎英哉氏に深く感謝いたします．また，論文執筆に協力していただいた電気通信大学岩崎研究室の皆様，LISP on TEX の実装や方針に対し意見をくださった@zr_tex8r 氏と@keno_ss 氏にお礼を申し上げます．

参考文献

[1] Bruggeman, C., Waddell, O. and Dybvig, R. K.: Representing Control in the Presence of One-shot Continuations, *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, New York, NY, USA, ACM, pp. 99–107 (online), DOI: 10.1145/231379.231395 (1996).

[2] Carlisle, D., Rowley, C. and Mittelbach, F.: The LATEX3 Programming Language—a proposed system for TEX macro programming, *Available from CTAN, macros/latex/exptl/project/expl3* (1998).

[3] Hagen, H.: LuaTEX: Howling to the moon, *TUGboat*, Vol. 26, No. 2, pp. 152–157 (2005).

[4] Iwasaki, H.: Developing a Lisp-based preprocessor for TEX documents, *Software: Practice and Experience*, Vol. 32, No. 14, pp. 1345–1363 (2002).

[5] Knuth, D. E.，斎藤信男，鷺谷好輝：改訂新版 TEX ブック (1992).

[6] Pakin, S.: PerlTEX: Defining LATEX macros using Perl, *TUGboat*, Vol. 25, No. 2, pp. 150–159 (2004).