はじめに

現在，TEX/LATEX は自然科学，特に数学や物理学などの数式が多く含まれるような分野において，レポートや論文のような文書を作成する際には事実上必須のツールとなっている．しかも，レポートや論文だけでなくセミナーの資料やちょっとしたネット上の解説記事も TEX/LATEX で作成されることが多くなっている．このように，TEX/LATEX は特に自然科学を学ぶ者にとっては馴染みのあるツールであり，当該分野の知識に加えてその使い方を習得する必要に駆られている．

では彼らがどのように TEX/LATEX の使い方を習得するかといえば，広く出版されている入門書を手に取る場合もあるけれども，おおよそ研究室やサークルの先輩からの口伝や，インターネット上の解説記事を頼りにした独学が多いのではないだろうか．もちろん彼らは TEX/LATEX のエキスパートになりたいわけではなく，自分の要求に堪えるレベルで文書が作成できればそれで十分なのではあるが，そのためかあまり推奨されないような使い方をしている場面にしばしば遭遇する．具体例を挙げると，

- `\quad` や `\textbf` などの「見た目系コマンド」の本文中での多用，
- `jarticle` や `jreport` などの時代遅れのドキュメントクラスの利用，
- `eqnarray` 環境や`$$...$$`のような時代遅れの数式環境の利用

などである．

これらの問題のうち，後者 2 つに関しては解説を見かける機会は非常に多

い．時代遅れのドキュメントクラスや環境の使用は，出力がおかしくなったりそもそもタイプセットができないといった致命的な問題を引き起こすからであろう．しかし，1 つ目の問題に対しては後者 2 つに比べれば解説を見かけることは多くない．「見た目系コマンド」が本文中で多用されたからといって，出力がおかしくなるような致命的な問題は生じないからであろう．

TEX/LATEX はマークアップシステムであるから，文書の意味内容を表す「構造」とそれらの構造がどのように紙面に表現されるかの「体裁」はできる限り分離されるべきである．この「構造と体裁をできる限り分離してソースを書こう」という姿勢でなされるマークアップは構造化マークアップと呼ばれ，特に Web ページを作成する際には必須とみなされている[*1]．

本書は LATEX 界隈にもこの流れを取り込むべく執筆された本である．具体的には，TEX/LATEX の基本的な事項を習得した初心者が構造化マークアップを意識した美しいコードを書けるようになることが目標である．従って，TEX/LATEX の基本的な事項は習得済みであることを想定している．TEX/LATEX でレポートや論文を作成した経験があればあまり問題ないと思われる．まずは新しいコマンドや環境の作成方法を述べ，それを使って文章や数式の意味内容を効果的に表現する例を挙げる．

本書で挙げたテクニックを習得できれば，ソースコード中のコマンドの意図が分かりやすくなり可読性が向上するだけでなく，汎用的に使えるコマンドを切り出して別ファイルに保存し，それを別の文書に使いまわすことも容易となる．本書の読者がこのような恩恵を実感できるようになれば何よりである．

本書では，環境に対する依存性が低い話題が中心であるが，具体的な実装面では 2022 年 12 月ごろにインストールされた TEX Live 2022 を想定する．それでも，違いが生じうるのはおおよそ「xparse パッケージを読み込む必要があるかどうか」程度であろう．

[*1] Web ページを作成する際に構造化マークアップが必須となるのはブラウザの多様化や検索エンジンとの関係が大きく，筆者が LATEX で構造化マークアップを勧める理由とはかなり異なるが．

なお，本書ではソース中の空白文字を可視化するため，必要な場合には「\␣」という記法で空白文字を表現することがある．また，ソース中で「\textbf{⟨*argument*⟩}」のように山括弧で囲まれたイタリック体の文字が登場した場合，それは本当に「<argument>」と記述するわけではなく，状況に応じて適切なものに置き換えることを意図して書かれている．このことに注意しながら読み進めてほしい．

目次

第 1 章

予備知識

§1.1　構造化マークアップ

マークアップ方式文書を作成する際，文書の論理的構造とそれを紙面に表示する際の視覚的効果（すなわち体裁）は，明確に分離して記述するのが望ましいとされることが多い．それを強調する意味で，マークアップを特に**構造化マークアップ**と呼ぶことがある．構造化マークアップが重要視される理由は文脈によってさまざまであるが，LaTeX と関係するのはおおむね以下のようなものである：

- 文書の体裁を定義する部分が本文とは分離されているため，体裁のみの変更が容易となる．
- あらかじめ体裁を定義する部分を切り出してパッケージ化しておくことにより，ユーザーが自ら体裁を定義する必要がなくなる．

特に後者は重要である．体裁の定義は処理系の低レベルな部分に触れなければできないことも多く，その処理系に対する深い知識が要求されることが多い．そのような知識を一般ユーザーに求めるようでは，そのツールは不便であると言わざるを得ない．

LaTeX では，基本的にドキュメントクラス内で文書の全体的な体裁が定義

されている．有志がドキュメントクラスを開発してくれるおかげで，ユーザーは文書の論理的構造をソースに記述するだけで「いい感じ」の文書が作成できるのである．このことが理由で LaTeX を使用するユーザーも決して少なくはないだろう．他にも，**パッケージ**[*1]という形で提供される拡張機能を読み込むことにより，LaTeX をさらに便利に使うことができるのは周知のとおりである．

§1.2　コマンドの定義による構造化マークアップ

さて，与えられたドキュメントクラスやパッケージを活用するだけでは，自分のニーズを完璧に満たせるとは限らない．そのニーズ自体が LaTeX を使用するうえで不適切である場合[*1]を除けば，新しいコマンドや環境を定義したり，既存のコマンドや環境の定義を変更することで解決されるべきである．もちろん，技術的な難易度の問題で解決できないこともあるが，簡単に解決できることも多い．例を挙げよう．

例 1.2.1. プログラミング言語「C++」のロゴを出力したいとき，そのまま「C++」と入力しても（Latin Modern Roman フォント採用時には）「C++」と表示されてしまい，非常に不格好である．

採用するフォントや好みによるところも大きいが，例えば

Source Code

```
1 C\nolinebreak\hspace{-.04em}%
2 \raisebox{.4ex}{\tiny\textbf{+}}%
3 \nolinebreak\hspace{-.14em}%
4 \raisebox{.4ex}{\tiny\textbf{+}}%
```

[*1] ここでは，\usepackage コマンドで読み込むことができるパッケージファイルのことを指している．

[*1] 例えば，図を自分の思う通りの場所に配置したいというニーズはそもそも組版上不適切であることが多い．

のようにいくらかの微調整を加えてやることで，出力が（Latin Modern Roman フォント採用時には）「C++」のようになり，いくらか改善する．これを本文に適用したいところであるが，まさかソース中の該当箇所をそのままこれに置き換えようなどとは思うまい．置き換えた後に「微調整」の内容を変更した際，ソース中の該当箇所すべてを変更する必要が生じるからである．当然修正漏れの可能性もある．例えば「\cpp」のようなコマンドを上記微調整の結果として定義し，本文中では「\cpp」のように記述すべきである．

例 1.2.1 で挙げたのは，必要とされた特殊な装飾を本文中から外に隔離する例である．このようにすることで，本文中では「\cpp」のように，すっきりした記述にできる．

重要なのはそれだけではない. 本文中での記述から \hspace や \raisebox といった「余計なもの」が排除され，「\cpp」という「意図」のみになっている．これにより，ソースを読むだけで「何を書きたいのか」がすぐにわかる．書いてすぐならば多少ごちゃごちゃした記述であっても意図を忘れることは多くないだろうが，時間が経った後に自分が書いたソースを見返した場合に何が書いてあるのかまったくわからなくなる．「そのようなことを考えるのは面倒だしどうせ見返すようなことなどないのだから，ソースは書き捨てでよい」などと考えてはいけない．このような tips は定期的に再利用したくなるものである．

「意図をはっきりさせる」ことは，異なる意図ではあるが結果的に体裁が同じになった場合に特に有用である．

例 1.2.2. 新出の単語も強調も太字で書くことはよくある．強調の場合は標準で用意されている \emph コマンドを使うべきである．ここで，新出の単語も \emph コマンドを使ってしまうと，ソース中で両者を見分けることができなくなってしまう．そのため，新出の単語を示すために例えば \term といったコマンドを用意することで，両者を区別することが容易になる．

ただし，「新出の単語を太字表記するのは強調のためで，そこに意味の違

いはない．その証拠に，新出の単語であっても重要でないものは太字表記しないではないか」と考えるのであれば，両者を区別せずに \emph コマンドで統一するのは理にかなっている．

さて，例 1.2.2 で取り上げた \emph コマンドに関して，誤った tips が広まってしまっているようである．

例 1.2.3. 強調の際には \emph というコマンドを使う．ところが，この \emph コマンドは（環境にもよるが）「和文はゴシック体にするが，欧文はイタリック体にする」という挙動を示す．

Source Code

```
1 \emph{なんか重要そうな和文}や\emph{important contents}とか
```

とすれば「**なんか重要そうな和文**や *important contents* とか」のような出力が得られる．

この挙動を嫌がったせいか「強調には直接 \textbf コマンドを使ってしまえ」という誤った tips が横行している．この場合，当然 \emph コマンドの定義を変更するのが正しい．

以降，これを LaTeX で実装するための具体的な方法を考えよう．

第 2 章

TEX の字句解析超入門

マクロの具体的な作り方の前に，まずは TEX の字句解析についてざっくりと復習しておこう．なお，ここではざっくりとした感覚をつかむことを目的としているため，正確性はまるで意識していない．そのため，この節の内容は TEX 処理系や TEX 言語の入門としてはまったく役に立たないということに注意しよう．むしろ，「この後の説明のどこが破綻しているか」を考えることがよい勉強になる可能性すらある．

§2.1　プリミティブとマクロ

TEX において，何らかの動作を実行する命令は**プリミティブ**と**マクロ**に大別される．

プリミティブなのは TEX で定義されている命令の一部であるから，普段我々が目にしている命令のほとんどはマクロであるということになる．これから我々が行うのも，単に「複数の命令にわかりやすい名前を付けよう」というだけにすぎない．しかしそのためには，まず「TEX が命令をどのように認識しているか」を知らなければならない．

§2.2 カテゴリーコードとトークン

TEX では，ソース中の文字に対して表 2.1 に示す 0 から 15 までの**カテゴリーコード**という整数値が割り当てられている．

和文のことを無視すれば，ソース中の文字とカテゴリーコードとの対応は以下のようになっている．

表 2.1 TEX におけるカテゴリーコードとその役割，および通常各カテゴリーコードに属する文字，ここでは和文文字は一切考慮していない．

カテゴリーコード	分類	属する文字
0	制御綴の開始	「\」
1	グルーピング（開始）	「{」
2	グルーピング（終了）	「}」
3	数式の区切り	「$」
4	アライメントタブ	「&」
5	行末	「^^M」（文字コード 13）
6	パラメータ文字	「#」
7	上付き文字（数式中）	「^」
8	下付き文字（数式中）	「_」
9	無視される文字	「^^@」（文字コード 0）
10	空白文字	「␣」（文字コード 32）
11	英字	英語アルファベット
12	記号	数字と「!」,「?」他多数
13	アクティブ文字	「~」
14	コメント開始	「%」
15	無効文字	「^^?」（文字コード 127）

表 2.1 には「^^」から始まる奇妙な文字もあるが，ひとまず気にする必要

はない．また，カテゴリーコード 10 の「␣」は単なるスペースであると考えてよい．

基本的に，カテゴリーコード 11 と 12 以外の文字はソース中で特別な役割を果たす文字であり，そのまま打ち込んでも表示されない．

TEX は，ソースとして受け取ったテキストファイルを**トークン**と呼ばれる塊に分解し，それをもとに動作する．何を 1 つのトークンとみなすかは，おおむね次のような規則に従っていると考えることができる：

1. カテゴリコード 0 の文字の直後にカテゴリーコード 11 の文字がある場合，そこから続くすべてのカテゴリーコード 11 の文字をまとめて 1 つのトークンとみなす．
2. このとき，そのトークン直後のカテゴリーコード 10 の文字は（いくつ続いても）無視される．
3. カテゴリーコード 0 の文字の直後にカテゴリーコード 11 以外の文字がある場合，その文字までを 1 つのトークンとみなす．
4. それ以外の場合，その文字 1 つを単独のトークンとみなす．

例 2.2.1. ソース中にある．「`\LaTeX3␣is␣good.`」という文字列を考える（「␣」は空白を表す）．最初にカテゴリーコード 0 の「`\`」が存在していることから，そこから続く「`\LaTeX`」までが 1 つのトークンとみなされる．そしてその直後に続くトークンは「`3`」である．さらに続く英字や空白は 1 文字につき 1 つのトークンとみなされる．最後のトークンは「`.`」である．すなわち，上の文字列は「`\LaTeX`」「`3`」「␣」「`i`」…「`d`」「`.`」という 11 個のトークンに分解される．

トークンのうち，カテゴリーコード 0 の文字（すなわちバックスラッシュ「`\`」）から始まるものを**制御綴**と呼ぶ．LATEX においては，何らかの動作を行うことが期待される命令を**コマンド**と呼ぶ．コマンドの中で主要なのは，まさにこの制御綴である．

ところで，これまで述べてきたことからもわかるように，カテゴリーコー

ド 0 の文字から続くカテゴリーコード 11 の文字は，まとめて 1 つのトークン，特に制御綴とみなされる．このことからわかるのは，**制御綴の名前に使えるのはアルファベットの大文字・小文字のみである**ということである[*1]．しかし，「\#」や「\$」のような制御綴の存在を考えると，このルールはおかしく見える．このような制御綴が存在しているのは，カテゴリーコード 12 の文字 1 つだけの場合は例外的に制御綴の名前に使えるということにしているからである．従って，「I watch TV\@.」中のトークン「\@」は制御綴とみなされる．そして，その直後に空白があれば，それは無視されるわけではなく空白文字トークンとして認識されるのである．

各文字に割り当てられたカテゴリーコードは変更することができるが，エンドユーザーとしてはあまり行うべきではないということはいうまでもないだろう．しかし，カテゴリーコード 12 の文字「@」だけはカテゴリーコード 11 に変更することがよくある．そのためのコマンドとして，\makeatletter コマンドと \makeatother コマンドが用意されている．\makeatletter コマンドを記述した場所以降では「@」がカテゴリーコード 11 に変更され制御綴の名前に使えるようになり，\makeatother コマンドを記述した場所以降では「@」がカテゴリーコード 12 に戻る．この仕組みは，ユーザー側に触れてほしくない内部でのみ使われるコマンドを定義するのに役に立つ．

§2.3 引数の処理

TEX で扱う制御綴には，引数をともなうものとそうでないものがあった．そして，どこからどこまでが引数であるかはグルーピングを表す文字「{」と「}」で指定するのであった．

引数が複数のトークンからなる場合，グルーピングによってその範囲を指定しなければならない．一方で，引数がただ 1 つのトークンからなる場合，

[*1] 和文対応環境では，和文文字も制御綴の名前に使えることがある．よく使う例として \today コマンドの書式を変更する **\西暦** コマンドと **\和暦** コマンドが挙げられる．

グルーピングによる範囲指定は省略できる（グループ指定を省略した場合，直後のトークンが引数とみなされる）．

例 2.3.1. それぞれの書き方で何が `\textbf` コマンドの引数とみなされているかに注意しよう．`\textbf` コマンドは引数を 1 つとる制御綴であった．

Source Code

```
% 「\LaTeX」の出力結果全体が引数になる
% 「\textbf \LaTeX」のようにスペースを入れても同じ
\textbf\LaTeX

% 「L」のみが引数になる
% 無論「\textbfLaTeX」のように書けばエラーになる
\textbf LaTeX

% 「LaTeX」全体が引数になる
\textbf{LaTeX}
```

出力は，上から順に「**LaTeX**」「**L**aTeX」「**LaTeX**」となる．

マクロを作成していて奇妙な挙動が見られた場合，引数指定が適切であるかどうかも考えておく必要があるだろう．

§2.4 展開と実行

トークンには，処理された結果別のトークン列に変換されるものがある．この変換処理を**展開**という．展開できるトークンは**展開可能**であるという．あるトークンの展開結果として現れたトークンは，展開可能であれば再び展開される．そして，展開可能でないトークンになるまで展開操作は続く．展開可能でないトークンが現れた場合，そのトークンによって定められた何らかの処理が行われることになる．これを**実行**という．

当然のことながら，制御綴のうちで展開可能でないものはプリミティブに限る（が，プリミティブでも展開可能であるものは存在する）．我々が文書

作成でよく使う制御綴はほとんどすべて展開可能である．

以上の事項を念頭に置きながら，マクロの作成方法について学んでいく．

第 3 章

マクロの作り方

§3.1　新しいコマンドの定義

新しいコマンドを定義する場合，\NewDocumentCommand コマンドを用いるのがよい[*1].

\NewDocumentCommand コマンドは新しめの環境であれば追加パッケージを読み込むことなく使えるが，そうでない場合は xparse パッケージを読み込めば使用可能になる.

\NewDocumentCommand コマンドを使って新しいコマンドを定義する場合，（普通は）プリアンブルで以下のように記述する：

Source Code

```
1 \NewDocumentCommand\⟨command name⟩{⟨argument specification⟩}{⟨definition⟩}
```

あるいは，最初の「\⟨*command name*⟩」の部分をかっこでくくって

[*1] もしかしたら \def や \newcommand の方が見慣れているかもしれないが，基本的には \NewDocumentCommand コマンドを用いるべきである．\def は TEX レベルの制御綴で，\newcommand は \NewDocumentCommand よりも機能が劣っているとみなせるからである.

Source Code

```
1 \NewDocumentCommand{\⟨command name⟩}{⟨argument specification⟩}{⟨definition⟩}
```

のように記述してもよい．どちらも意味は同じで「⟨*argument specification*⟩という引数仕様のコマンド \⟨*command name*⟩ を，⟨*definition*⟩ と定義した」ということになる．これにより，ソース中の「\⟨*command name*⟩」は「⟨*definition*⟩」に置き換えて解釈される．

`\NewDocumentCommand` コマンドが引数として要求している「引数仕様」とは，定義したい「\⟨*command name*⟩」がどのような引数をもつかを記述するものである．よく使うものを表 3.1 に示す．

表 3.1　`\NewDocumentCommand` コマンドでよく使う引数仕様の一覧，複数の引数をもつコマンドを定義する場合，順番に並べて記述する．

記号	意味
`m`	必須引数
`o`	オプション引数（既定値なし）
`O{`⟨*default value*⟩`}`	オプション引数（既定値が ⟨*default value*⟩）
`s`	スター（「`*`」）の有無を判定

指定した引数には ⟨*definition*⟩ の部分で，登場順に「`#1`」や「`#2`」のようにアクセスできる．

引数仕様に「`s`」を指定した場合，この引数には「`*`」の有無を示すフラグが付与される．使用時には「\⟨*cmd*⟩`*{`⟨*second argument*⟩`}`」のように記述する．第 1 引数が「`*`」のみ指定可能なオプション引数になったイメージである．

引数仕様に「`s`」を指定した場合，その引数を使って処理を分岐させることができる．そのために役に立つのが `\IfBooleanTF` コマンドである．このコマンドは

Source Code

```
1 \IfBooleanTF{⟨argument⟩}{⟨true expression⟩}{⟨false expression⟩}
```

のように使用する．「*」がある場合には ⟨*true expression*⟩ が，ない場合には ⟨*false expression*⟩ がそれぞれ実行される．このようにして，あるコマンドの「亜種」とも呼ぶべき *付きのコマンドを簡単に定義できる．

LaTeX における \providecommand コマンドのように，「そのコマンドが定義済みでない場合のみ定義を行い，定義済みの場合は何もしない」という処理を行うための \ProvideDocumentCommand コマンドも存在する．使い方はまったく同じで

Source Code

```
1 \ProvideDocumentCommand\⟨command name⟩%
2 {⟨argument specification⟩}{⟨definition⟩}
```

のように使用する．「⟨*argument specification*⟩」の部分も \NewDocumentCommand コマンドとまったく同じである．

§3.2 新しい環境の定義

新しい環境を定義する場合，\NewDocumentEnvironment コマンドを用いる．このコマンドは

Source Code

```
1 \NewDocumentEnvironment{⟨environment name⟩}{⟨argument specification⟩}%
2 {⟨begin process⟩}{⟨end process⟩}
```

のように使用する．これにより「⟨*argument specification*⟩ という引数仕様の ⟨*environment name*⟩ 環境を，開始時に ⟨*begin process*⟩ を，終了時に ⟨*end process*⟩ を行う環境として定義した」ということになる．引数仕様の部分は

もちろん \NewDocumentCommand のものとほぼ同じである[*1].

§3.3　既存の定義の書き換え

既存のコマンドの定義を書き換える場合，\RenewDocumentCommand コマンドを用いるとよい．使い方はやはり \NewDocumentCommand コマンドとまったく同じである．異なるのは，\NewDocumentCommand コマンドでは定義対象となるコマンドが未定義であることが要請されていたが，\RenewDocumentCommand コマンドではその逆で，定義対象となるコマンドが定義済みであることを要請する．

\NewDocumentEnvironment コマンドについても同様に，既存の環境の定義を書き換えるための \RenewDocumentEnvironment コマンドが存在する．

「定義対象が定義済みであろうとなかろうと問答無用で定義を行う」という処理を行うための \DeclareDocumentCommand コマンドや \DeclareDocumentEnvironment コマンドも存在する．しかし，これらは「定義済みとは知らずにうっかり上書きしてしまった」というリスクがあるため，どうしても必要な場合以外は使用を控えるのが望ましい．

[*1] 引数仕様の「 s」は，\NewDocumentEnvironment コマンドで使用した場合には挙動が異なる．詳細は xparse パッケージのドキュメントを参照せよ．

第 4 章

実践例

§4.1　単純な例

まずは単純な例から始めよう．

例 4.1.1. 例 1.2.1 で述べた \cpp コマンドは次のように定義できる：

Source Code

```
\NewDocumentCommand\cpp{}{%
    C\nolinebreak\hspace{-.04em}%
    \raisebox{.4ex}{\tiny\textbf{+}}%
    \nolinebreak\hspace{-.14em}%
    \raisebox{.4ex}{\tiny\textbf{+}}%
}
```

\NewDocumentCommand コマンドの第 2 引数である引数仕様を空にしておくことで，引数を持たないコマンドを定義できる．本文中では「\cpp」とのみ記述すればよいので，保守性や可読性の向上が見込める．

例 1.2.3 で述べた \emph コマンドの再定義を考えよう．

例 4.1.2. \emph コマンドの振る舞いを \textbf コマンドと同じようにしたければ，いっそのこと \textbf コマンドと同じ挙動にしてしまえばいい．

つまり，

Source Code

```
1 \RenewDocumentCommand\emph{m}{\textbf{#1}}
```

のようにしてしまえばいい．こうすれば，\emph{hoge} と書くのと \textbf{hoge} と書くのが等価になる．振る舞いを \textbf コマンドと同じにしたうえで本文中では「\emph{hoge}」のままにしておくことで，「この箇所は強調である」という意図がはっきりと残る．

§4.2　数式が関わる例

数式が関わる文脈では，構造化マークアップはよりその力を発揮する．

例 4.2.1. 実数全体の集合 $\mathbb{R}$ を記述する際，よく見るのは（amsmath パッケージ読み込み時では）「\mathbb{R} と記述せよ」というものである．しかし「\mathbb{R}」では「R を黒板太字で出力せよ」という意図しか伝わらない．そこで

Source Code

```
1 \NewDocumentCommand\realnumbers{}{\mathbb{R}}
```

のように定義し，本文中では「\realnumbers」と記述することで「実数全体の集合を出力せよ」という意図がはっきりと伝わる．さらに，例えば unicode-math パッケージを採用することを後から決めた場合，\realnumber コマンドの定義を

Source Code

```
1 \NewDocumentCommand\realnumbers{}{\symbb{R}}
```

と変更するだけで修正が完了する．本文中で何度実数全体の集合が登場しよ

うと修正は 1 箇所のみである．

例 4.2.2. 数学において，かっこは文脈に応じてさまざな意味で使われる．例えば，x に関数 f を適用した結果を表す「$f(x)$」や，項の構成順序を表す「$a(x+y)$」である．後者に関しては明確な意味があるとは言い難いが，前者には「関数適用」という確固たる意味を有する．そこで，以下のように定義してみよう：

Source Code

```
\NewDocumentCommand\paren{sm}{%
    \IfBooleanTF{#1}{%
        \mathopen{}(#2)\mathclose{}%
    }%
    {%
        \mathopen{}\left( #2 \right)\mathclose{}%
    }%
}
\NewDocumentCommand\apply{smm}{%
    \IfBooleanTF{#1}{%
        #2\paren*{#3}%
    }%
    {%
        #2\paren{#3}
    }%
}
```

*の有無でかっこの大きさを \left と \right で調整するかどうかを指定できる．

このように定義することで，関数適用 $f(x)$ が「 \apply{f}{x}」と記述できるようになる．同時に，コマンド「\paren」も合わせて定義することで，項の構成順序を表す「$a(x+y)$」は「 a\paren{x + y}」のように記述できる．\paren コマンドを定義するのは，意図をはっきりさせるというよりむしろ複雑な実装を隠蔽するという意味が強い．

例 4.2.2 で出てきた \mathopen コマンドと \mathclose コマンドについ

て補足しておこう．これらのコマンドはともに引数を 1 つとり，その引数をそれぞれ数式における開きかっこと閉じかっことして認識させるコマンドである．これらで `\left` と `\right` をはさむことにより，`\left` と `\right` に起因する余白を抑制することができる．

ところで，`\mathopen` や `\mathclose` は，構造化マークアップがマクロを組むことだけではないことを示す好例である．

例 4.2.3. 左半開区間は，$(a,b]$ と書かれることもあれば $]a,b]$ と書かれることもある．とくに後者をきちんと「`\mathopen{]} a, b \mathclose{]}`」と書くことで，本来閉じかっことして振る舞う「]」を開きかっことして認識させることができる．

構造化マークアップとは，自分の意図を明瞭になるようにソースを記述することを志向したマークアップの技法である．したがって，それを実現する手段はなんでもよい．マクロを組むのは多くの場合に有用な解決策であるが，それが唯一ではないのである．

参考文献

[1] 明松真司, “1 週間で LaTeX の基礎が学べる本”. インプレス, 2022.

[2] 奥村晴彦 黒木裕介, “LaTeX 2_ε 美文書作成入門”. 技術評論社, 2017.

コマンド索引

用語索引

構造化マークアップ志向の LaTeX 入門

2022 年 12 月 初版

著者：野口 匠

Twitter：@Nmatician

発行：NOGUTAKU Lab

印刷：株式会社ポプルス