The SATySFI book

The SATySFI book

Takashi Suwa 著

物理書籍が以下から入手できます：

https://booth.pm/ja/items/1046747

The SATYSFIbook（Web 公開版 第 1 版）

by

Takashi SUWA

序文

本書は，2017 年度未踏事業のひとつとして筆者が開発し，2018 年 12 月現在も発展を続けている新しい組版処理システム SATYSFI について解説することを目的としたものである．SATYSFI はマークアップ言語（およびその下層にあるプログラミング言語）とそれに対する処理系からなっており，ユーザがコードとして文書を執筆してそれを処理系に渡すとその指定に従って組版処理が行なわれ PDF が生成される，という仕組みをもつ．

このような形態をもつ既存の組版処理システムとしては，主に理工系分野の論文組版等に広く用いられている TEX/LATEX がある．「既に広く使われている実績のある同様の役割のソフトウェアがあるのに，新たに創る必要があったのだろうか？」と思われるかもしれない．単純な疑問として，SATYSFI のどのような性質が「未踏」だったのだろうか？

結論から言うと，最大の違いは組版処理機能そのものではなく，

- コマンド[*1]定義がどの程度簡単にでき，どの程度その定義の可読性が高いか
- 不適格な入力に対してどのようにエラーが報告されるか

にある．SATYSFI はマークアップ言語であると同時にいわゆる函数型言語らしい特徴も多く備えており，そのおかげでコマンド定義を通常のプログラミングの感覚で読み書きできるようになっている．これは，TEX/LATEX の「少し凝ったコマンドを定義するには TEX 言語独特の項書き換えの仕組みを理解し，多くの場合直観に反した実装を書かねばならず，デバッグも難しい」という性質とは対照的である．さらに，函数型らしい言語設計を採用していることの恩恵として強力な型システムを搭載しており，SATYSFI はユーザが或る種の不適格な入力を与えた際にはその型システムを用いて組版処理を行なうよりも前にそれを検出し，（多くの場合）入力ミスの箇所が特定しやすいエラーを提示してくれる．これもやはり TEX/LATEX が「ひとまず受け取ったソースに基づいて組版処理をしはじめ，その途中で不整合に出くわしたらそのとき初めてエラーを出す」という仕組みをとっているのとは対照的であり，このようなエラー報告の仕組みにより SATYSFI には従来システムに

1　LATEX に親しみのない読者には奇異に聞こえるかもしれないが，文書中のマークアップや組版処理の指定が関連づけられたタグのようなものを広く「コマンド」と呼ぶ．

比べて高い執筆効率が期待できるのである．

以上が SATYSF$_{\mathrm{I}}$ の開発動機の非常に大雑把な概要だが，おそらくまだまだ「本当に有用なのか？」という疑問を払拭しきれない読者もいらっしゃるかと思う．どのような経緯で現在の SATYSF$_{\mathrm{I}}$ の形態に至ったのか，どのような点で既存システムと差別化されるのかについては，第 1 章でさらに詳細に述べたので参照されたい．なお，既に「早く SATYSF$_{\mathrm{I}}$ を使いたい，有用性云々をくどくど述べられるよりは先にコードの書き方を知りたい」というお気持ちの読者は，第 1 章は読み飛ばしても構わない．SATYSF$_{\mathrm{I}}$ コードの入力で画面を凝視し続けたためになかなか寝つけない夜，ふと本書の第 1 章に目を通していなかったことを思い出し，カモミールティーでも淹れて眠気がくるまでの時間に一瞥し，「なるほど」と感じていただければ筆者としては本望だ．

さて，本書の構成について述べよう．SATYSF$_{\mathrm{I}}$ を構成する各部分は複雑に依存し合っており，各部分を別個に，つまり他の部分から完全に独立したものとしていきなり理解するのはかなり困難である．そのため，本書は全体として「前の方では大雑把な理解のしかたを伝え，後ろへ行くにしたがってより一般性の高い説明に改める」という方針で構成されている．したがって，第 2 章以降はできるだけ前から順に読み進めることをおすすめする．

第 2 章ではまず `stdjareport` というクラスファイルが定義しているコマンド類の用法の説明を通じて SATYSF$_{\mathrm{I}}$ で文書をどのように執筆すればよいのかという感覚を掴んでもらう．この章を参考にすれば，読者は（自前で定義したコマンドを含まない）最低限の簡素な文書が記述できるようになる．

第 3 章ではユーザが自らの手でコマンドを定義する方法のごく初歩を説明する．この章の後半ではプログラミング言語としての SATYSF$_{\mathrm{I}}$ に触れ始めるので，いわゆる函数型プログラミングに馴染みのない読者にとっては新たな直観を獲得する場となり，多少腰を据えて読んでもらうことになるかもしれない（といっても，世間で噂されているほど難易度の高い概念では決してない）．また，この章では SATYSF$_{\mathrm{I}}$ の組版処理システムとしての最大の特徴である型システムについて触れはじめる．すなわち，どのような文書やパッケージの記述を型がつく整合したものとして組版処理に渡し，一方でどのような記述を何らかの不整合を含むものとして拒否するのかを規定した仕組みである型検査器について軽く触れる．といっても，ここで触れるのは SATYSF$_{\mathrm{I}}$ の型システムのうちでも OCaml や F# といったいわゆる函数型言語のそれと大枠を同じくする部分がメインである．

第 4 章では SATYSF$_{\mathrm{I}}$ の処理系の全体像について説明し，各部分がどんな処理を担っているかを概観する．処理系がやっていることの全体像は，TEX の処理系やブラウザのレンダリングエンジンがやっていることと比べてきわめて顕著な差があるわけではないが，

SATYSFI での種々のデータの型による扱いがどのような定式化を経て確立されているか，組版処理の根幹となるプリミティヴ[*2]がどんな処理に対応しているのかを理解するのに把握しておくことが必須となる．

第 5 章では，SATYSFI の各部分の機能を読者が把握している前提で，組版処理そのものに手を加えるような高度なコマンド定義の方法について解説する．文字サイズ，フォント指定，文字色，行送りといった文字組版に関する基礎的設定をいじったり，行分割に関する制御などについても説明し始める．また，SATYSFI の型システムに備えられた，（汎用の計算機言語には見られない）組版処理に由来する独特の型の扱いもここで詳しく触れる．SATYSFI のあらゆる仕組みはこの型システムでできるだけ静的に不整合を検出することに合わせて構築されており，ここ以前の説明で読者が「どうしてこんな少し回りくどい仕組みになっているのだろう」と抱いた疑問の伏線は，おそらくここで回収されるだろう．

第 6 章から第 12 章までは各論的で，それぞれ行分割処理，システムと設定ファイル，文字組版，グラフィックス，数式組版，相互参照，ページ分割処理について解説する．各章で関連するいくつものプリミティヴが紹介され，ユーザやパッケージ開発者はこれを用いてより柔軟に組版処理を制御するコマンドを定義することができる．

残りの章はかなり詳細に SATYSFI を知りたい，さらには SATYSFI 本体の開発に手を入れてみたい，と考える方に向けて掲載した内容である．第 13 章では SATYSFI の現在のあり方の改善点や今後拡張・追加したい機能について述べる．特に，現在実装を進めている**テキストモード** (*text mode*)，つまり（PDF ではなく）テキストファイルを出力するモードについても触れる．このモードのおかげで，同一の SATYSFI 文書から PDF と（HTML や LaTeX などの）ソースファイルとが（適切なパッケージが揃っている下で）共に出力できるようになる．第 14 章では，SATYSFI のごく小さなサブセットの言語について，構文，操作的意味論，型システムなどを厳密に与える．これは本当に物好きな読者や，或いは計算機言語理論を専門としていて SATYSFI がどのように形式化されているのか気になる方々に向けた章であり，おそらく全く読まなくても実用上差し支えないだろう．

以上が本書の構成であるが，future work について記載した第 13 章が大々的に設けられていることからもわかるように，SATYSFI はまだまだ試行錯誤の段階にある．したがって，これは著者の或る種のワガママなのだが，読者には本書を「今後ずっと参照できる SATYSFI の参考書」のように捉えるよりは「SATYSFI が辿る歴史の 1 つの節目をスナップショットとして記録したもの」という程度に大らかに受け止めて頂きたいと考えている．勿論，現状の SATYSFI でも著者が修士論文の執筆に使った程度にはシステムとして成熟

2　言語機能として備わっている，ビルトインの函数などのこと．

しているし，言語機能としても全体的にはそれなりに筋の良い形式化になっているのではなかろうかと自負しているので，言語機能の根幹を著しく改めるような抜本的な変更は行なわないつもりではいる．

SATYSF$_{\mathrm{I}}$ の開発にあたっては，様々な方々からの支援を受けました．まず未踏事業に於いて PM の竹迫良範さん，統括 PM の竹内郁雄先生をはじめとして多くの方から意見を頂き，また事業として多大な経済的支援を賜りました．また，成果報告会での発表以降，多くの方が善意から SATYSF$_{\mathrm{I}}$ の改善や拡張に参加してくださいました．

- スタックマシンへのコンパイル機能，パターンマッチの網羅性検証，未定義変数名のエラーに対する修正候補のサジェスト機能など，処理系に種々のきわめて重要な拡張を施していただいた Daiki Matsunaga (`matsud224`) さん，
- 字句解析器・構文解析器を整頓し [8]，かつ Windows 版へのクロスコンパイル [7] を整備していただいた Masaki Hara (`qnighy`) さん，
- 仕様書のない状態から SATYSF$_{\mathrm{I}}$ でのスライド作成用クラスファイル Steamer [13] を実装され，かつそれに伴い種々のプリミティヴの追加を提案していただいた Hiromi Ishii (`konn`) さん，
- SATYSF$_{\mathrm{I}}$ Online [20] を実装された `miettal` さん，
- SATYSF$_{\mathrm{I}}$ Playground [25] を実装された `pandaman64` さん，
- ocamlbuild から dune への移行など OCaml での開発に関して種々の改善を頂いた Daichi Oohashi (`leque`) さん，
- インストール手順の整備，処理系の動作の整頓等に多大に協力してくださった Takuma Ishikawa (`nekketsuuu`) さん，Yuichi Nishiwaki (`nyuichi`) さん，`sakas--` さん，`objmagic` さん，Masanori Ogino (`omasanori`) さん，Noriaki Sakamoto (`na4zagin3`) さん，
- 数式用パッケージを拡張していただいた `ToruNiina` さん，`forestaa` さん，
- SATYSF$_{\mathrm{I}}$ の前身である Macrodown に Travis CI を導入していただいた YOSHIMURA Yuu (`y-yu`) さん，
- その他注意深い観察により修正を提示いただいた `theoldmoon0602` さん，`equal-l2` さん，`sunaemon` さん，`KilledByNLP` さん

にはここに感謝の意を表明したいと思います．

2018 年 9 月　Takashi SUWA

目次

1. SATYSFI の開発動機 1

1.1. マークアップ方式の利点と弱点 1

1.2. 既存システムの問題点 1：エラー報告能力 3

1.3. 既存システムの問題点 2：コマンド定義の可読性 4

1.4. 問題意識の総括 4

1.5. SATYSFI が選んだ解決策 5

1.6. エラー報告機能 6

2. SATYSFI 簡易入門 11

2.1. ファイルの種別と拡張子 11

2.2. 文書構造 12

2.3. 段落 14

2.4. 章節 15

2.5. 新出単語 16

2.6. 文中数式 16

2.7. 別行立て数式 17

2.8. コメントアウト 18

2.9. 箇条書き 18

2.10. 定理環境 19

2.11. 脚註 20

2.12. 相互参照 20

3. コマンド定義：基礎編 23

3.1. 簡単なエイリアス 23

3.2. プログラミング言語としての SATYSFI 26

3.2.1. 式と値 26

3.2.2. 変数とその束縛 28
3.2.3. 基本的なデータ 30
3.2.4. 型システムの基礎 33
3.2.5. 長さ 34
3.2.6. 函数定義 36
3.2.7. 真偽値と条件分岐 42
3.2.8. 型システムと型推論の恩恵 43
3.2.9. タプル 45
3.2.10. 代数的データ型 47
3.2.11. 再帰函数 50
3.2.12. レコード 56
3.2.13. 手続き 57
3.2.14. 算術に関するプリミティヴ 58
3.2.15. OCaml との顕著な差異 61

4. 文書が処理される仕組み 63
4.1. フロントエンド 63
4.2. バックエンド 66
4.2.1. インラインボックス列への変換 66
4.2.2. 行分割処理 66
4.2.3. ページ分割処理 67

5. コマンド定義：発展編 69
5.1. テキスト処理文脈 69
5.2. コマンド定義の糖衣構文 75
5.3. 行分割を行なうコマンド 76
5.4. ブロック変換器 82

6. 行分割処理 84
6.1. 行分割処理の概要 84
6.2. 行分割に関するプリミティヴ 85
6.2.1. 行分割候補箇所 85
6.2.2. ペナルティ値 87

6.2.3. グルー 89
6.3. ブロック方向の長さの制御 90
6.4. ブロックボックス列の行分割内容への挿入 91
6.5. 長さの取得に関するプリミティヴ 93

7. システムと設定ファイル 94
7.1. 各ディレクトリの役割 95
7.2. フォント設定 96
7.3. デフォルトのフォント設定 97
7.4. 分割ファイルの読み込み 98

8. 文字組版 100
8.1. 文字の扱いの基本 100
8.2. 文字の分類 101
8.3. フォントの扱い 104
8.4. 異なる文字体系間の空白 105
8.5. 寡占的文字体系 105

9. グラフィックス 107
9.1. パス 107
9.2. グラフィックス指定 110
9.3. 囲み枠 113
9.4. 表組版 117
9.5. 画像 119

10. 数式組版 120
10.1. 数式の構文と扱い 121
10.2. 文中数式の扱い 122
10.3. 数式を構築するプリミティヴ 123
10.4. 文字の扱いと文字クラス 125
10.5. 括弧類 127

11. 相互参照 129

12. ページ分割処理 131
12.1. ページ分割に関連するプリミティヴ 134
12.2. クラスファイルの基礎 135

13. Future Work 137
13.1. 機能の拡張 137
13.1.1. テキストモード 137
13.1.2. 文字組版機能の拡張 137
13.1.3. 言語機能やシステム上の拡張 139
13.2. 既存機能の破壊的な変更 141

14. 言語仕様 142
14.1. テキストの扱いの形式的記述 142
14.1.1. 構文 142
14.1.2. 操作的意味論 144
14.1.3. 型システム 145
14.1.4. 考察 147

Chapter 1
SATYSFI の開発動機

まずは「SATYSFI がそもそもなぜマークアップ言語とその処理系という形態をとることを選んだのか」，そして「既存のマークアップ言語やその処理系では不満足だった，SATYSFI で解決されるべき課題は何か」という点から書こう．そのためには，マークアップ言語＋処理系という形態が一般論として他の形態と比べどんな利点をもち得てどんな弱点を本質的にもたざるを得ないか，そして既存のマークアップ言語はその利点を活かせていてその弱点を或る程度克服できているか，ということについて見ていく必要がある．

1.1. マークアップ方式の利点と弱点

この本を手に取った方ならおそらくよくご存知のとおり，一口に組版処理システムといっても大きく分けて

- マークアップ言語によってテキストファイルとして文書を記述し，言語処理系に入力として与えて最終的に PDF などの形式で版面を出力する方式
- GUI の画面上に版面がプレヴューされた状態で文書を記述していく WYSIWYG エディタ方式

の 2 つがある．具体的には TEX/LATEX, troff, Markdown などが前者にあたり，Microsoft Office Word, Adobe InDesign, QuarkXpress などが後者に該当する．

“マークアップ言語方式と WYSIWYG 方式とのいずれが優れているのか” に関する議論は（少なくとも一方には触れるユーザが多いためか）頻繁に目にし，もはや大喜利の様相を呈している場合もあるが，実際にはそれぞれの方式に利点と弱点のトレードオフがあるといってよいと考えられる．WYSIWYG 方式に比べたマークアップ言語方式一般の利点としては，あくまでも傾向ではあるものの

- 入力がテキストファイルなので差分管理が簡単

- ユーザ定義コマンドにより：
  - 複雑な自動処理が実現できる
  - 文書の内容を変更せずに後から体裁を柔軟に変更しやすい

といったことが挙げられる．したがって，これらの恩恵を活かして文書を作成したい場合がマークアップ言語の想定ユースケースということになる．特に，文書内容を文書の体裁からできるだけ独立したものとして記述したいという**意味マークアップ**の志向が強いユーザにとってはこれらの性質は望ましいものである．

一方で弱点としては

- 直接版面を見ながら執筆できるわけではないため，執筆作業がしばしば直観的でない
- ユーザが不適格なコードを入力として与えやすい

といった点が挙げられるであろう．1 点目については，組版処理システムを WYSIWYG 方式，マークアップ言語方式と大別せず両者を融合したようなハイブリッド方式のソフトウェアを創るという方針によって緩和できるように考えられる．これは既存のソフトウェアでは LyX [5] のような仕組みが該当し，今後組版処理システムを発達させていく上でも有望な方針といえそうである．ただし，LyX が LaTeX というマークアップ方式の組版処理システムの存在を前提としているように，順序としてはまず純粋なマークアップ言語とその処理系が発達してからこのようなハイブリッド方式が発達していく余地ができるもの，という側面が強い（先走って述べると，SATYSFI はこのハイブリッド方式の方針を採らずに純粋なマークアップ方式として組版処理システムを実現したが，それはこのような側面を念頭に置いたためである）．

より注目すべきマークアップ方式の弱点は 2 点目である．WYSIWYG 方式ではそもそも妥当な操作しか GUI 上でユーザに提供しないことによってユーザが不適格な入力をしえない仕組みを構築することができるのに対し，マークアップ方式ではその性質上任意のテキストが入力として渡せるので，ユーザが不適格なコードを与えてしまえるのである．さらに広く言えば不適格なコードだけでなく意図しないコード全般を与えてしまいうるのだが，ユーザが意図していない入力のうち処理系にとっては適格なコードは間違いを検出することが（少なくとも機械学習のような統計的な手法によらない限り原理的に）できないし，これはユーザ自身が出力された版面を見て確かめるしかない．これに対し，処理系にとっても不適格なコード，すなわちコマンドの引数の与え方がおかしいといったミスは，処理系が（非統計的な，“クリスプな” 手法により）見つけることができるから，これをユーザに報告して修正を仰ぐことができる．つまり，ユーザが意図しないコードを与えてしまいやすいとい

う弱点が本質的にあるマークアップ言語という方式では，処理系にとって不適格なコードが入力として渡された場合にどのようなエラー報告を行ない修正方法を示唆するかがユーザの執筆効率に多大な影響をもたらすと考えてよいであろう．

ところが，このエラー報告に関して既存のマークアップ方式の組版処理システムには不十分な点が見受けられる．これを次節でより詳細に扱う．

1.2. 既存システムの問題点 1：エラー報告能力

前節で既に述べたように，マークアップ言語には

- ユーザが不適格なコードを入力として与えやすい

という本質的に避けにくい弱点があり，不適格な入力に対してどのようにエラー報告するかがユーザの執筆効率に大きく関わる．しかし，既存のマークアップ言語方式の組版処理システムでは，この不適格な入力に対するエラー報告がわかりにくいという傾向がある．特に LaTeX の場合を例に挙げると，以下のようなものはユーザにとって親の顔よりも見たエラーログだ：

```
! Undefined control sequence.
```

```
! Missing $ inserted.
```

```
! Missing number, treated as zero.
```

場合によっては原因がすぐわかることもあるが，基本的には「組版処理中に初めて不整合が生じた」ときにエラーが報告される仕組みなので，ユーザの与り知らない，コマンドの実装内の記述と与えた引数との不整合によるエラーが直接報告されることも多い．「第 n 行目までで何かがおかしい」という程度の情報しか直ちには得られないことも多く，このような状況におかれてはもはやコードをコメントアウトしながら二分探索して原因の場所を特定するしかない．

エラー報告は不親切なだけでなく提示されるまでに時間を要することも多い．やはり「組版処理中に初めて不整合が生じた」ときにエラーが報告されるのが要因で，例えば 500 ページの文書を LaTeX で組む際に 490 ページ目にあたる部分に不適格な記述を書いてしまった場合は 490 ページ分の組版処理の後にエラーが提示されるため，ユーザは入力ミスの存在を知らされるまで短めに見積もっても数十秒から数分程度は待つことになる．

また，このエラーの原因が直ちに汲み取りにくいという傾向はエンドユーザだけでなくパッケージ開発者にとっても深刻な問題である．パッケージ開発者は多くの場合複雑な処理を簡潔なインターフェイスとして提供するためにパッケージを実装し，当然ながらその実装過程に於いて複雑な処理が意図どおり記述できているか具体的な文書例を作成してデバッグする．この際に，エラー報告が不親切であるがゆえに，どのように修正すればよいかがエラー報告からまるで見当がつかないことも極めて多く，パッケージ開発者は修正に四苦八苦することを強いられてしまう．このようにして不親切なエラー報告はパッケージ開発者の実装にも大きく支障をきたすのである．

1.3. 既存システムの問題点 2：コマンド定義の可読性

ここまで述べたエラー報告についての弱点とは別に，マークアップ方式をとっている既存の組版処理システムに対する別の不満点としてはコマンド定義等を記述したコードの可読性が挙げられる．マークアップ言語の利点の傾向のひとつとして

- ユーザ定義コマンドにより：
  - 複雑な自動処理が実現できる
  - 文書の内容を変更せずに後から体裁を柔軟に変更しやすい

という性質を挙げていたものの，注意すべきはこのコマンド定義は必ずしも簡単に書けたり読んで修正できたりするとは限らないということである．例えば TEX/LATEX ではコマンド定義機能が提供されていて，多少の自動処理であれば簡単にコマンド定義として書くことができるが，少し凝った自動処理を書くにはかなりの工夫を要し，とても直観的に単純とはいえないコードを書く必要がある．複雑な自動処理を行なうコマンド定義は書く側にとっても負担であるが，他人の手によって書かれたコマンド定義を読んで修正したい人にとってはさらに大変な苦労を強いられることになる．TEX/LATEX の場合，もはや書いた本人以外誰も読んで把握することができないのではなかろうかと思われるようなコードはごく普通にパッケージ群の奥底にある．

1.4. 問題意識の総括

結局，ここまで追ってきた議論を簡潔にまとめると，以下のようになる：

- マークアップ言語は（論文の執筆等の文書作成時に積極的に援用される志向である）意味マークアップの観点からは WYSIWYG エディタと比べて望ましい形態である

- マークアップ言語と WYSIWYG エディタを融合した形態で両者の利点をいいとこ取りした組版処理システムが将来的には実現できるかもしれないが，LyX の例を見るに，そのようなソフトウェアは堅牢なマークアップ言語が土台にあってはじめて実現できるように思われるから，先に便利なマークアップ言語を確立しておく必要がある
- しかし，既存のマークアップ言語は
  - エラー報告機能が不親切で遅いため，ユーザの文書執筆やパッケージ開発の効率があまり良いとはいえない
  - コマンド定義を記述したコードの可読性がしばしば著しく低く，変更や修正がしづらい

そこでこれを解決した便利なマークアップ言語として SATYSFI を実現しよう，というわけである．

1.5. SATYSFI が選んだ解決策

1.4 節で述べた問題点を解消するために開発した，

- コマンド定義を記述したコードの可読性が高くカスタマイズしやすい
- ユーザが不適格な入力を与えたときのエラー報告が素早く，わかりやすく提示される

という性質を備えた新たなマークアップ言語方式の組版処理システムが SATYSFI である．“主にコマンド定義用の言語” と “主に文書を書くための言語” の 2 層を切り分けて言語を設計し，この言語に対する処理系として SATYSFI が実装されている．文書を書くための層は見た目上おおよそ LaTeX に近い設計にした一方で，コマンド定義用の層はいわゆる函数型プログラミングに便利な言語仕様，すなわち函数型言語らしい言語仕様をベースとしてコア言語を設計・実装することでシステムの基礎づけを与え，これにより

- グローバルな状態をあまり意識せず，局所的な処理にだけ気を遣ってコマンド定義が読み書きできる
- 強力な型システムを搭載しやすい

といった函数型言語であることによる恩恵を一気に享受することで上記の性質を達成した．この言語は，函数型言語の中でも特に OCaml や F# に近い言語設計となっている．処理系は大きく分けて

- ユーザやパッケージ開発者による不適格な入力を検出してエラーとして報告するフロントエンド（構文解析器や型検査器）
- フロントエンドでの検査を通った入力を実際に組版処理を施して PDF を出力するバックエンド

からなっている．

また，このような構成の組版処理システムを実装するだけでなく，実装した成果をオープンソースなソフトウェアとして公開することも重要だ．主な理由は

- ユーザが無償で組版処理システムを使えるようにすることに重大な意義があると考えられるため
- 開発した組版処理システムの改善・拡張を活発化させるため

なのだが，このうち特に 2 点目が重要である．多言語組版の実現などを念頭に置くと組版処理システムに必要な機能はきわめて多岐にわたり，あらゆる観点で万全なものとして完成させるには著者 1 人で創りきれる規模とはいえず，オープンソースにすることでシステム内部でどのような処理が行なわれているか誰でも実装を読んで理解でき，改善が必要であると感じた場合には積極的に開発に参加しうるような環境が必要なのである．

最初からあらゆる観点で万全な組版処理システムにすることが現実的に困難なのは，これまで数十年規模で多くの開発者の手によって開発・保守されてきた TEX の派生システムでも未だにバグ修正や機能の拡張などに取り組まれていることからほぼ明らかであろう．SATYSFI は多言語組版が可能な組版処理システムの基礎づけを与えることを主目的とし，まずは日本語を母語とする著者にとっても個人的に馴染みの深い和文・欧文の混植が実用に堪える程度満足にできることを目標として開発することとした．

1.6. エラー報告機能

以上のような動機のもとに開発した SATYSFI によって実現された，ユーザの高い執筆効率を支える縁の下の力持ちとなるところのエラー報告機能についての特徴を具体例とともに紹介する．ユーザが以下のような出力をしようとしてわずかに間違った記述を入力に含めてしまった場合を考え，LATEX と SATYSFI でどのようにエラー報告が異なるかを見てみる．

いろは歌：

色は匂へど散りぬるを，吾が世誰ぞ恒ならむ．有為の奥山今日越えて，浅き夢見じ，酔ひもせず．

まず，LaTeX の場合（特に pLaTeX を用いることにする）の正しい出力を得られるコードは以下のとおりである：

```
% -*- coding: utf-8 -*-
\documentclass{jsarticle}
\usepackage[T1]{fontenc}
\usepackage{lmodern}
\begin{document}
  \indent
    いろは歌：
    \fbox{\parbox[t]{5cm}{%
      色は匂へど散りぬるを，吾が世誰ぞ恒ならむ．
      有為の奥山今日越えて，浅き夢見じ，酔ひもせず．
    }}
  \par
\end{document}
```

ここでユーザが `\parbox` コマンドの幅指定を忘れ，8 行目を以下のように間違ってしまったとする：

```
\fbox{\parbox[t]{%
```

このとき pLaTeX は不適格な入力であるとして何らかのタイミングでエラーを報告するのだが，そのエラーとして標準出力 / 標準エラー出力に提示されるのが次のようなメッセージである：

```
! Argument of \@iiiparbox has an extra }.
<inserted text>
                \par
l.11     }}

?
```

おそらくほとんどのユーザはこのエラーログから「11 行目までで何かおかしい」ということ以上に詳しい情報を引き出せないだろう．これは `\parbox` コマンドの実装内部でエラーが生じており，そこで初めてユーザの与えた不適格な引数が害を及ぼしてこのようなエラーログを引き起こしているのであり，ユーザにとってはほとんど与り知らないようなところのエラーが報告されていてわかりづらいと言わざるを得ない．

同様に，高さ指定のオプション引数をユーザが間違えて必須の引数だと思い込んで記述してしまった時のエラーも見てみる．これは 8 行目を以下のように間違えてしまった場合である．

```
\fbox{\parbox{t}{5cm}{%
```

そのエラーが次のようなメッセージである：

```
! Missing number, treated as zero.
<to be read again>
                   t
l.11     }}

?
```

やはりコマンドの実装内で初めて破綻した箇所でのエラーが出ており，ユーザにとってはまるで直接の原因がわからないといってよいようなエラーログである．小さい文書であればすぐ気づくかもしれないが，大きな文書を組んでいる場合は原因を探すのにもひと苦労である．場合によっては，“書いたコードの一部をコメントアウトしながら地道に二分探索して原因箇所をつきとめる”ことになってしまう．

では SATYSFI で等価な実験をするとどうだろうか．ここからが要点である．まず，正しい出力を得る SATYSFI のコードは以下のとおり：

```
% -*- coding: utf-8 -*-
@require: stdjareport
@import: local

document (|
  title = {Test Document};
  author = {Takashi SUWA};
|) '<
  +p{
    いろは歌：
    \fbox{\parbox?:(Top)(5cm)<
      +p{
        色は匂へど散りぬるを，吾が世誰ぞ恒ならむ．
        有為の奥山今日越えて，浅き夢見じ，酔ひもせず．
      }
```

```
    >}
  }
>
```

念のために述べると，`\fbox` や `\parbox` といったコマンドはわかりやすさのために LaTeX と名称を揃えてあるだけで，SATYSFI のプリミティヴのみを用いて実装されている．内容は今は逐一解説しないが，以下のようにしてコマンドとして定義できる：

```
let-inline ctx \fbox it =
  let pads = (2pt, 2pt, 2pt, 2pt) in
  let t = 0.5pt in
  let deco (x, y) w h d =
    [ stroke t Color.black
        (Gr.rectangle (x, y -' d) (x +' w, y +' h)) ]
  in
    inline-frame-inner pads deco (read-inline ctx it)

type position-spec = Top | Bottom

let-inline ctx \parbox ?:specopt wid ib =
  let blockf =
    match specopt with
    | None         -> embed-block-top ctx wid
    | Some(Top)    -> embed-block-top ctx wid
    | Some(Bottom) -> embed-block-bottom ctx wid
  in
    blockf (fun ctx -> read-block ctx ib)
```

では，もとの `\parbox` の幅指定をやはりユーザが誤って記述しなかったときはどうなるであろうか．13 行目を以下のように変えてみる：

```
\fbox{\parbox?:(Top)<
```

これを処理すると入力として不適格ゆえにエラーが報告されるのであるが，そのエラー報告が以下のようなものとなる：

```
! [Type Error] at line 13, character 26 to line 16, character 1:
    this expression has type
      block-text,
    but is expected of type
```

```
      length.
```

13 行目の 26 文字目から block-text 型の引数が与えられているが，ここには length 型の引数，つまり長さを表す指定がくるべきだ，というエラーが出ているのである．直接的な原因がわかりやすいため，ユーザはこのエラーにしたがってすぐに修正することができる．

同様に，オプション引数を必須の引数だと間違えてユーザが入力を与えてしまった場合のエラーも見てみる．やはり 13 行目を以下のように変えて処理してみることにする：

```
\fbox{\parbox(Top)(5cm)<
```

以下のようなエラー報告が返ってくる：

```
! [Type Error] at line 13, characters 18-23:
    this expression has type
      position-spec,
    but is expected of type
      length.
```

position-spec 型の引数が与えられているが，必須の引数なら length 型の引数が与えられるべき箇所である，というエラーが出ている．やはり直接的な原因がわかりやすく，ユーザはすぐに修正に向かうことができる．このようなエラー報告は型検査器に於いてコマンド `\parbox` に型推論の結果つけられた型から判定することによって為されており，静的に型をつける恩恵はまさにここにあるといってよいだろう．以上のような型を用いた修正に便利な仕組みが，既存のマークアップ言語方式の組版処理システムでは実現されていない，SATYSFI の大きな強みである．

Chapter 2
SATYSFI 簡易入門

ひとまず読者がすぐにさわれるものとして`stdjareport`クラスがあるので，これを土台に説明することとしよう．おそらくLaTeXの使用経験がある読者ならコード例を見ただけでおおよそ何を記述しているかは察しがつくであろうが，LaTeX（の一般的なクラスファイルが提供するマークアップ）と顕著に違いのある箇所についてはその都度述べておくことにした．

なお，LaTeXの使用経験がない読者， そのうち特にMarkdownやreStructuredTextといったいわゆる軽量マークアップ言語に慣れ親しんだ読者からすれば「こんな冗長な記述をしなければならないのか」と感じられる点も見受けられるかもしれないが，組版処理中にエラーを生じないことを組版処理よりも前にできるだけ検出すること，ユーザにもコマンドが柔軟に定義できるようにすることを志向している都合上冗長にならざるを得ない部分も多いのだろうとひとまずは理解されたい．

2.1. ファイルの種別と拡張子

SATYSFIがユーザからの入力として直接的に扱うファイルには

- **文書ファイル** (*document file*)：文書の内容を記述するファイル
- **パッケージファイル** (*package file*)：コマンド定義などを記述するファイル

の2種類がある．パッケージファイルのうちの一部には文書全体のマークアップ方法や体裁等を規定しているものがあり，これは**クラスファイル** (*class file*) と呼ぶ．パッケージファイルは拡張子を`.satyh`にしておく必要がある[*3]． 文書ファイルの拡張子は現在のところ特に決まりはないが， 慣習的には`.saty`を使用している． 特別な理由がない限り， 拡張

3　実際には`.satyh`のほかに`.satyg`などの拡張子が扱われるが， 今のところはひとまず`.satyh`だけと認識してもらって差し支えない．詳しくはテキストモードで扱う．

子はこの .saty にすることを強く推奨する．

2.2. 文書構造

全体構造として文書ファイルは以下のように記述する：

```
@require: stdjareport

document (|
  title = {サンプル文書};
  author = {組 版太郎};
|) '<
  +chapter{はじめての章}<
    +p{
      \SATySFi; でのはじめての段落の記述です．
      段落は \dfn{ブロックコマンド} で実現されています．
    }
  >
>
```

インデントは構文解析上特に意味を持たず単に無視されるので，どのようにつけても構わない[*4]．文書ファイルの `document` よりも手前の部分は，LaTeX に準じて便宜的に**プリアンブル** (*preamble*) と呼ぶことにしよう．ここに

```
@require:（パッケージ名）
```

と書くことでシステムにインストールされたパッケージを読み込める．パッケージの実体はソースコードが記述された単一のパッケージファイルであり，例えば `stdjareport` の本体は `stdjareport.satyh` というファイルなのだが，読み込みの指定は拡張子なしで書く（`.satyh` が自動で補われる）．文書ファイルと同じディレクトリに置いているような“ローカルな”パッケージファイルや分割ファイルは

```
@import:（ファイルへの相対パス）
```

4　勿論，習慣としては可読性のために括弧の入れ子構造に応じてインデントをつけておくことを薦める．

で読み込む．やはり拡張子は自動で補われるので書かない[*5]．

`(| … |)` にタイトルや著者名など文書に必要な情報を与え，本文は `'< … >` に記述する．この形式は SATYSFI の GitHub リポジトリの `lib-satysfi/dist/packages/` に置いている他のクラスファイル，例えば `stdjabook` クラスなどでもおおよそ同様である．SATYSFI での文書の記述は主に

- **ブロックテキスト** (*block text*)：ページ方向に積み重なるもの
- **インラインテキスト** (*inline text*)：文字の進行方向へ連なるもの

に分かれており，このうちインラインテキストが基本的に通常の文字列としてのテキストを書くことが想定されている箇所である．ブロックテキストは `< … >`，インラインテキストは `{ … }` で囲って記述する．ただし，上に掲げたコードで文書全体の内容にあたるブロックテキストを囲う開始側の括弧が `<` ではなく例外的に `'<` であることには注意されたい[*6]．TEX の用語でいえば，ブロックテキストは垂直モードにおける記述，インラインテキストは水平モードにおける記述におおよそ対応する．

ひとまず，上に掲げた例から PDF を生成してみよう．まずは適当なディレクトリに例示した文書ファイルを保存する．ファイル名は `hello.saty` とする．ターミナルを開き，`hello.saty` が置いてあるディレクトリに移動して

```
$ satysfi hello.saty
```

を実行する．文書の記述に誤りがなければ，同じディレクトリに `hello.satysfi-aux` と `hello.pdf` が生成されるはずである．このうち `hello.satysfi-aux` は**補助ファイル** (*auxiliary file*) あるいは**ダンプファイル** (*dump file*) と呼ばれる副産物で，今は無視して構わない．`hello.pdf` が肝腎の生成物であり，これを開くとたしかにそれらしい小さな文書ができていることが確認できるだろう．

---

5 `@require` と `@import` で拡張子を省略する仕組みは単に手間を省くためにあるわけではない．補われる拡張子は `.satyh` とは限らないのである．これについては第 13 章でのテキストモードについての説明で後述する．

6 詳細は後述するが，この括弧の前後で字句解析規則が変わりうる都合上，“コマンドの引数ではない部分” でブロックテキストを書くときは `<` ではなく `'<` にする必要がある．単に `<` と書くと不等号の 2 項演算子だと認識されてしまう．

さて，上に掲げた例には，`+chapter{ … }< … >` やら `+p{ … }` やら `\dfn{ … }` やら，なんとなくわかりそうだが妙に多くの記号が駆使された怪しげな記述がある[*7]．これらについて以下の節で見ていくこととしよう．

2.3. 段落

段落は“文字の連なりを受け取ってページ方向に連なる行のリストにするもの”と捉え，インラインテキストを引数にとってブロックを返すコマンドで定式化されている．以下のように `+p` コマンドでブロックテキスト内に記述する：

```
+p{
  明日は明日の風が吹く
}
```

明日は明日の風が吹く

上で「ブロック」と書いたものは正確には「ブロックテキスト」ではないが，ひとまずはブロックテキストとおおよそ同じものだと考えて差し支えない．ブロックテキスト内で使うコマンド，つまり“ブロックを返す”コマンドは名前が `+` で始まり，**ブロックコマンド** (*block command*) と呼ぶ．一方インラインテキスト内で使うコマンドは名前が（LaTeX のコマンドと同様に）`\` で始まり，これを**インラインコマンド** (*inline command*) と呼ぶ（後述）．いずれのコマンドの名前も，2 文字目は大小ラテン文字，3 文字目以降は大小ラテン文字，数字，ハイフンのいずれかである．

コマンドに与える引数の構文は

- `( … )`：**式引数** (*expression argument*)
- `{ … }`：**インライン引数** (*inline argument*)
- `< … >`：**ブロック引数** (*block argument*)

7 念のため書いておくと，`+chapter` や `+p` といったコマンドは `stdjareport` が規定しているもので，“SATYSFI の言語機能に本質的に備わっているもの”ではない．その気になればクラスファイル `stdjareport.satyh` の中を覗いて実装を見ることができる．

の 3 種類に大別される[8]．コマンドに引数を与えることをコマンドの**適用** (*application*) と呼び，適用の構文は，正規表現風の擬似的なメタ記法を用いて

$$cmd\big[(\cdots)\big]^*\Big(;\ \Big|\ \big[(\{\cdots\}\mid\langle\cdots\rangle)\big]^*\Big)$$

で表される．ここで *cmd* は，インラインテキスト中ではインラインコマンドを，ブロックテキスト中ではブロックコマンドをそれぞれ表す．要するに，コマンドの適用はまずコマンドの後ろに 0 個以上の式引数が並び，そしてその後ろに `;` か或いは各要素がインライン引数かブロック引数であるような長さ 1 以上の列かがきて終わる．

ブロックテキスト，すなわち `<…>` の中身はブロックコマンドの適用の 0 個以上の列である．インラインテキスト，すなわち `{…}` の中には通常の文字列とインラインコマンドの適用とが書ける．ただし，いくつかの特殊な機能をもつ文字は，通常の文字列に含めるには `\` によるエスケープを要する．エスケープする必要のある文字は `{`，`}`，`<`，`>`，`|`，`%`，`*`，`;`，`#`，`$`，`\`，`@`，`` ` `` である．このうち `;` だけは欧文でそれなりの頻度で出現するため，多少注意を要する．

2.4. 章節

`+chapter`，`+section`，`+subsection` を使う．名称は LaTeX に準じているが用法は少し異なり，ブロックテキスト内に以下のように記述する：

```
+chapter{（章題）}<
  （内容）
>
```

章題だけではなく，その章の内容をブロックテキストとして第 2 引数に与えるという形で定式化されている．章題は文字の連なりなのでインラインテキスト，内容は段落のページ方向への積み重なりなのでブロックテキストである．`+section`，`+subsection` の用法も全く同様．

8　このうち式引数には `(…)` の形ではない多少の変種があり，`([…])` および `((|…|))` という具合に直下が四角括弧であるものやレコード式であるものはそれぞれ単に `[…]` および `(|…|)` と書いて式引数として扱わせることができるのだが，今は気にしなくてよい．

2.5. 新出単語

新たに定義される名称や言い回しは，インラインコマンド `\dfn` で囲む．インラインテキスト中で以下のように使う：

```
+p{
  原稿とデザインにしたがって活字や図版を配置して
  版面を構成することを総称して \dfn{組版処理} と呼ぶ.
}
```

> 原稿とデザインにしたがって活字や図版を配置して版面を構成することを総称して**組版処理**と呼ぶ.

引数として囲まれた部分のラテン文字がイタリック体になる．このように，インラインコマンドとは“（何らかの加工が施された）文字の連なり”を返すコマンドなのである．なお，既に述べたコマンドの適用の構文からわかるように，引数のないインラインコマンドはセミコロンをつける：

```
+p{ \SATySFi; is a new typesetting system. }
```

> SATYSFI is a new typesetting system.

2.6. 文中数式

文中数式は，インラインテキスト中に `${ ··· }` で囲って書く．記法はかなり LaTeX に近く，以下のような例だと `${ ··· }` の中身は見かけ上 LaTeX コードと完全に一致する：

```
+p{
  数式は \LaTeX; と同様に
  ${E = m c^2}, ${F = G \frac{m_1 m_2}{r^2}},
  ${x^x = e^{x \log x}}
  といった要領で書くことができます.
}
```

> 数式は LaTeX と同様に $E = mc^2$, $F = G\frac{m_1 m_2}{r^2}$, $x^x = e^{x \log x}$ といった要領で

書くことができます．

数式中のコマンドである**数式コマンド** (*math command*) はインラインコマンドと同様に `\` から名前が始まる．同一の名前をインラインコマンドと数式コマンドの両方で使うことはできない（つまり，名前空間は分かれていない）．インラインコマンドの場合とは違い，引数のない数式コマンドの使用に `;` はつけない．

LaTeX の数式の記法とよく似ているもののいくつかの違いがあり，特に括弧類の扱いは顕著に LaTeX とは異なる．LaTeX の `\left`・`\right`・`\middle` に相当するものはなく，代わりに括弧が左右合わせてひとつのコマンドとして定義されている．丸括弧で囲いたい場合は `\paren{（囲う式）}` と記述する．丸括弧のほか，集合の外延記法は `\set{…}`，内包記法は `\setsep{…}{…}`，絶対値は `\abs{…}`，距離やノルムに使う 2 本線の括弧は `\dist{…}` という数式コマンドにより提供されている．また，函数の適用は `\app{f}{x}` という具合に書けるようになっている．

なお，四角括弧，（集合に限らない一般の）波括弧，三角括弧もそれぞれ `\sqbracket{…}`，`\brace{…}`，`\angle-bracket{…}` で書けるようになっているが，これらのコマンドは数式の“意味”を指しているわけではないので，一応直接数式に書くというよりはユーザ定義コマンド（後述）の定義中で使うことを主たる用法として想定している[9]．

その他，数式について詳しくは第 10 章で扱う．

2.7. 別行立て数式

```
+p{
  別行立て数式は，少し前後がゴチャッとしているが
  \eqn($${
    \int_a^b \frac{\ordd x}{\sqrt{x^2 + 1}}
      = \sqbracket{\log \paren{x + \sqrt{x^2 + 1}}}_a^b
  }$);%
  という具合に書く．位置を揃えたい場合は，さらにゴチャッとするが
  \align[
```

9　と言いつつ，次節の例で `\sqbracket` を数式中に直接書いているが．

```
    ${| \paren{x - a} \paren{x - b}
          |= x^2 - a x - b x + a b |};
    ${|   |= x^2 - \paren{a + b} x + a b |};
  ];%
  という要領で書く．
}
```

別行立て数式は，少し前後がゴチャッとしているが

$$\int_a^b \frac{\mathrm{d}x}{\sqrt{x^2+1}} = \left[\log\left(x+\sqrt{x^2+1}\right)\right]_a^b$$

という具合に書く．位置を揃えたい場合は，さらにゴチャッとするが

$$\begin{aligned}(x-a)(x-b) &= x^2 - ax - bx + ab \\ &= x^2 - (a+b)x + ab\end{aligned}$$

という要領で書く．

`\align` の引数の `[ … ]` は直下に四角括弧があるプログラム引数 `([ … ])` の略記であり，プログラムに於ける `[ … ]` はリスト (*list*) である[*10]．

2.8. コメントアウト

単行コメントアウトは LaTeX と全く同様に `%` によって行なう．単純にコメントアウトとして使うだけでなく，インラインテキスト中で改行文字を入力として与えずにソース上で改行するのに使うことができる．複数行のコメントアウトは現在は備わっていないが，明らかにあると便利なので近い将来追加するつもりでいる．

2.9. 箇条書き

箇条書きは `itemize` パッケージを読み込むと使える．ブロックテキスト中で

```
+listing{
```

10 リストといっても，箇条書きの意味のリストとは特に関係がないことに注意．これについては第 3 章で説明する．

```
  * 東京
    ** 本郷
    ** 弥生
    ** 浅野
    ** 駒場
  * 千葉
    ** 柏
}
```

と書くほか，インラインテキスト中でも

```
\listing{
  * 東京
    （中略）
    ** 柏
}%
```

という具合に書ける． 箇条書きは“ページ方向に積み重なるもの”なので基本的には“ブロック”なのだが，マークアップの都合上段落中などでも使えるようにインラインコマンド版も提供されている．ただし，インラインコマンド版は末尾にコメントアウトの文字 `%` をつける必要がある[*11]． 見てのとおり多重構造も扱え， 階層は `*` の連続する個数によって判定される（インデントは単に見やすさのためにつけてあるだけである）．既に述べたように，`*` はこの機能のために特殊な文字として扱われている都合上通常のテキストとして書くには `\*` とエスケープする必要がある．

ちなみに， 上に掲げた例の箇条書きの部分 `{ … }` はインラインテキストではなく，itemize 型という箇条書き専用の型がつくテキストになっており， 通常のインラインテキストを与えると型エラーとなる．これについては第 3 章で後述する．

2.10. 定理環境

`stdjareport` には定義・補題・定理・系・例のための環境一式が備わっており， コマンドはそれぞれ `+definition`，`+lemma`，`+theorem`，`+corollary`，`+example` である．いずれも用法は同様で，最も簡素な使い方としてはブロックテキスト中で

---

11 これがないと，`}` 直後の改行文字に由来して次の文字の前に余分なスペースが挿入されてしまう．やや微妙な性質なので今後 `%` を書かなくてよいように改善したいと検討している．

```
+theorem{
  （内容）
}
```

などと書く．環境中で改段落することは想定されていないので，内容はインラインテキストで与える．オプション引数としてタイトルを与えることができる：

```
+theorem?:({進行}){
  ${t} が型のつく項ならば，以下のいずれかが成り立つ：
  \listing{
    * ${t} は値.
    * 或る ${t'} が存在して ${t \to t'} を満たす.
  }%
}
```

定理 2.1 (進行). t が型のつく項ならば，以下のいずれかが成り立つ：

- t は値.
- 或る t' が存在して $t \to t'$ を満たす.

`?:(…)` がオプション引数を与える構文である．

2.11. 脚註

脚註の用法はほぼ LaTeX と同様で，インラインテキスト中に `\footnote{…}` と記述することで挿入できる．ただし，脚註の中にさらに脚註の記述を設けることなどはできない．

2.12. 相互参照

相互参照は第 1 章でも述べたとおりマークアップ方式の組版処理システムが簡潔に実現することを得意としやすい機能だ．SATYSFI も相互参照機能を備えており，`stdjareport` はそれを利用して相互参照のインターフェイスを実現している．実は，既に紹介した章節や定理環境のコマンドたちは，オプション引数として相互参照のラベルを受け取ることができ

る[*12]：

```
+chapter?:(`introduction`){はじめに}<
  （中略）
>
```

そして，章番号を使いたい箇所で `\ref-chapter` を用いて挿入する：

```
+p{
  相互参照は \ref-chapter(`introduction`); でも述べたとおり
  マークアップ方式の組版処理システムが
  簡潔に実現することを得意としやすい機能だ.
  （以下略）
}
```

> 相互参照は第 1 章でも述べたとおりマークアップ方式の組版処理システムが簡潔に実現することを得意としやすい機能だ.（以下略）

定理類のコマンドも，2 番目のオプション引数として相互参照のラベルが挿入できる：

```
+lemma?:({保存})?:(`preservation`){
  ${\Gamma \vdash t : \tau} かつ ${t \to t'} ならば,
  ${\Gamma \vdash t' : \tau} が成り立つ.
}
```

> **補題 2.1** (保存). $\Gamma \vdash t : \tau$ かつ $t \to t'$ ならば，$\Gamma \vdash t' : \tau$ が成り立つ.

連続するオプション引数でより手前側のものを省略し後ろ側のものを与えたい場合は，オプション引数の “省略の明示” である `?*` を使う．ここでは補題のタイトルが不要だが相互参照のラベルをつけたい場合は以下のように改めればよい：

```
+lemma?*?:(`preservation-without-title`){
  ${\Gamma \vdash t : \tau} かつ ${t \to t'} ならば,
  ${\Gamma \vdash t' : \tau} が成り立つ.
}
```

12 勿論，相互参照のラベルを渡しても章節そのものの組版結果は特に変わらない.

補題 2.2　$\Gamma \vdash t : \tau$ かつ $t \to t'$ ならば，$\Gamma \vdash t' : \tau$ が成り立つ．

そして実際に番号を挿入する場所では `\ref-theorem`，`\ref-lemma`，... などを使う：

```
+proof{
  \ref-lemma(`preservation`); よりすみやかにしたがう.
}
```

Proof.　補題 2.1 よりすみやかにしたがう． ■

Chapter 3
コマンド定義：基礎編

第 2 章で stdjareport を用いて SATYSFI の基礎的なコマンドの用法を知った．ところで，マークアップ言語の大きな強みのひとつは自前でコマンドを定義することによって種々の自動処理を実現したり独自の意味マークアップを規定したりできる点にあったはずである．実際，`+chapter` とか `+p` とか `\dfn` とかいったコマンドたちは単に stdjareport クラスによって定義が提供されているだけであり，その気になればユーザでも同様の定義を書いたりできるはずだ．そろそろ自前でコマンドを定義したくてうずうずしている読者も多いかと思う．この章では，コマンド定義のごく基礎的な方法を紹介する．あらかじめ言ってしまうとここでの説明はとてもアドホックなものになっており，SATYSFI をよく理解されたのちにこの章を読めば「こんな子供騙しな説明をしていたのか」と驚かれるかもしれないが，読者の様々なバックグラウンドを鑑み，まず最初の一歩としておおまかに把握してもらうには良い説明だろうと判断してのことなのでご容赦願いたいと思う．より深い理解を獲得し，より柔軟で幅広い種類の処理をコマンドとして定義するには，第 4 章で SATYSFI が行なっている組版処理の全体像を把握した上で第 5 章に臨んでいただきたい．

3.1. 簡単なエイリアス

まずは非常に簡素な例から考える．例えば stdjareport の提供する `\dfn` というコマンド名が気に入らず，`\new-word` という名前で書きたいとしよう．要するに，`\new-word{（新出単語）}` と書いたら `\dfn{（新出単語）}` に “勝手に変わってくれる” ようなコマンド `\new-word` を定義したいとする．

これを実現するために，プリアンブルに手を加えよう．文書ファイルの `@require` や `@import` が並ぶ箇所と `document` との間には，コマンド定義を書くことができる．第 2 章で扱った例の本文中の `\dfn` を `\new-word` に書き換え，`\new-word` の定義をプリアンブルに追加してみよう：

```
@require: stdjareport
```

```
let-inline \new-word it =
  {\dfn{#it;}}

in
document (|
  title = {サンプル文書};
  author = {組 版太郎};
|) '<
  +chapter{はじめての章}<
    +p{
      \SATySFi; でのはじめての段落の記述です.
      段落は \new-word{ブロックコマンド} で実現されています.
    }
  >
>
```

`let-inline` 云々とあるのが `\new-word` の定義である． コマンドの定義はプログラミング言語で言うところの函数の定義によく似ており，`it` がインラインテキストが格納される引数にあたる． そして受け取った引数をインラインテキスト中で使っているのが `#it;` である．コマンド定義は

```
let-inline（定義したいインラインコマンド名）（0 個以上の引数）=
  {（置き換え後の記述）}
```

という形で書くことになる．おそらく中級以上の LaTeX ユーザなら `\newcommand` を想起するだろう．実際，SATYSFI での上記のような `\new-word` の定義は，`\dfn` が定義されている下で `\newWord` を定義する以下の LaTeX コードとほぼ等価なものと考えてよい．

```
\newcommand{\newWord}[1]{\dfn{#1}}
```

この定義中の `#1` が SATYSFI での `#it;` に相当するのである． 引数を要しないコマンドも同様に定義できる．例えば

```
let-inline \Cde =
  {ともあれ，カルタゴは滅ぶべきであると考える次第である．}
```

と定義すれば， インラインテキスト中で `\Cde;` と打つだけで「ともあれ， カルタゴは滅ぶべきであると考える次第である． 」と書いているのと等価になってくれる． 日本語を読

み書きできる古代ローマ人にとっては重宝するだろう．

ブロックコマンドの自前定義についても，以下のような構文によってインラインコマンドと同様に行なえる：

```
let-block（定義したいブロックコマンド名）（0 個以上の引数）=
  '<（置き換え後の記述）>
```

ただし，またも `<` ではなく `'<` であることに注意されたい．不恰好だが残念ながらこればかりは気をつけてもらうしかない．この機構を用いると，例えば段落末に必ず「ともあれ，カルタゴは滅ぶべきであると考える次第である．」という 1 文をつけ加えて段落をつくるコマンド `+pCde` は

```
let-block +pCde t = '<
  +p{
    #t;
    ともあれ，カルタゴは滅ぶべきであると考える次第である．
  }
>
```

という具合に定義できる．以降 `+p` の代わりにこの `+pCde` を使えば自動で末尾に文が挿入されるからやはり便利である．このほか，古代ローマ人以外にも便利な例としては次のようなブロックコマンドが定義できる：

```
let-block +no-proof bt = '<
  #bt;
  +p{証明は読者への演習課題とする．}
>
```

今度は引数がブロックテキストであることに注意されたい．ブロックテキストが格納された変数 `bt` も，ブロックテキスト中に `#bt;` と書くことでそこに内容を埋め込むことができる．この定義は要するに，与えられたブロックテキストの後ろに「証明は読者への演習課題とする．」と書かれた段落をくっつけるコマンドの定義になっている．以下のようにして理工書の執筆に使えば大変便利である．

```
+no-proof<
  +lemma{
    （証明がめんどくさい補題の内容）
  }
```

```
>
```

3.2. プログラミング言語としての SATYSFI

既に述べたように，SATYSFI はいわゆる函数型らしい特徴を兼ね備えた組版処理向けのマークアップ言語である．むしろ，より正確を期して言い表すならば値呼びの函数型言語の上に組版処理用の種々の言語機能が分厚く搭載された DSL（domain-specific language, 領域特化言語）だ．プログラミング言語としての SATYSFI の基礎的部分には OCaml に大変よく似た体系が採用されており，基礎的部分の理解を得るのに [11] や [2] といった OCaml に関する参考書，或いは [12] や [26]（およびその和訳 [27]）といった計算機言語や型システムに関する基礎理論を扱った理工書で代用することも十分可能なほどである．とはいえ，OCaml などの他言語から回り道せずとも SATYSFI での最低限の直観を身につければ十分 SATYSFI でのプログラミングが可能であり，熱心に函数型プログラミングの基礎を習得したい読者ばかりではないであろうから，ここでも簡単にプログラミング言語としての SATYSFI について解説する．また，OCaml と SATYSFI との間には目的意識や美意識の違いに起因した多少の差異があるので，これについて（既に OCaml で流暢にプログラミングできる読者に対しても）3.2.15 節で簡単に補足する．

3.2.1. 式と値

SATYSFI には対話環境が特に整備されていないので，プログラムの計算結果を PDF に出力して確かめることにしよう．ひとまず以下のようなコードを見てほしい：

```
@require: stdjareport

let-inline \show-int n = embed-string (arabic n)
let-inline \show-float x = embed-string (show-float x)
let-inline \show-string s = embed-string s
in

document (|
  title = {Programming in \SATySFi;};
  author = {計 算太郎};
|) '<
  +p{ Answer: \show-int(40 + 2); }
>
```

`let-inline` から始まる，未知でありながら前節で見たインラインコマンドを定義する方法によく似た構文が 3 つ連なっているが，実際これはそれぞれ `\show-int`，`\show-float`，`\show-string` というインラインコマンドの定義である．この定義をどう読むのかは後々わかることになるので，今の段階ではそれぞれ「与えられた(整数 | 浮動小数点数 | 文字列)を版面に（数値の場合十進法で）出力するコマンド」を定義するオマジナイだと思っていただきたい．注目されたいのは `\show-int` を使う部分，すなわち

```
+p{ Answer: \show-int(40 + 2); }
```

である．丸括弧に囲まれた `40 + 2` という怪しげな記述がある．とりあえず，この文書を SATYSFI 処理系にかけてみると，この段落に相当する部分は

> Answer: 42

と出力される．そう，40 と 2 の和が計算されているのである．平たく言えば，こういった“計算をする処理”がプログラムの仕事の根幹であり，“計算をする処理”をプログラムとして記述することをプログラミングと呼ぶ．`40 + 2` のような記述は**式** (*expression*) またはしばしば**項** (*term*) と呼ばれ，式を適切な構文にしたがって組み立てることでプログラムが構築される[*13]．

`40 + 2` の“計算が進んで” `42` になったように，式は一般に“それ以上計算しない式”へと“うつりかわっていく”．この“うつりかわり”の過程を**評価** (*evaluation*) またはしばしば**簡約** (*reduction*) と呼び，“それ以上計算しない式”を**構文的値** (*syntactic value*) 或いは単に**値** (*value*) と呼ぶ．プログラムの動作とは，式を評価して値を求めることなのである．

もう少し複雑な計算をしてみよう．

```
+p{ Answer: \show-int(1 + 2 * 3 - 4); }
```

> Answer: 3

13 C 言語や Python などの“手続き的な”観点に強く基づいている計算機言語に馴染みのある読者は「式から文をつくり，文からプログラムを構築するのではないか？」と思われるかもしれないが，SATYSFI に文の概念はない．というか，式の特殊な場合が“手続き的な”言語に備わっている文に相当するのである．これは OCaml をはじめとする多くのいわゆる函数型言語に共通する性質である．

`*` は乗算のための 2 項演算子である．通常我々が「$1 + 2 \times 3 - 4$」と書いた時には $\big(1 + (2 \times 3)\big) - 4$ と扱われるのと同様に，SATYSF$_{\mathrm{I}}$ 言語でも 2 項演算子の結合には優先順位がある．やはり日常的な数式と同様に，括弧で囲うと優先順位を指示できる：

```
+p{ Answer: \show-int((1 + 2) * 3 - 4); }
```

Answer: 5

なお，SATYSF$_{\mathrm{I}}$ での整数は 64bit 環境であれば 63bit 符号つき整数，32bit 環境であれば 31bit 符号つき整数である[*14]．

3.2.2. 変数とその束縛

ただ単に整数の計算をするだけでは電卓を叩いているのと大して違いはないから，もう少し“プログラミングらしいこと”に踏み込もう．すなわち，**変数** (*variable*) の使用である．変数は，**変数名** (*variable name*) によって識別され，何らかの値を**束縛する** (*bind*) ことのできる機構だ．変数名として使えるのは，

- 1 文字目はラテン小文字（`a`–`z`）
- 2 文字目以降は大小ラテン文字（`a`–`z`，`A`–`Z`），数字（`0`–`9`），またはハイフン（`-`）

を満たす，長さが 1 文字以上のコード中の文字列である．ただし，これに該当する文字列のうち以下に列挙した 39 種類は**予約語** (*reserved word*) と呼ばれ，構文を示すなどの特別な用途で用いられることが定められたトークンなので，変数名には使えない[*15]：

```
and as before block-cmd command constraint controls cycle do
direct else end false fun if in inline-cmd let let-block let-inline
let-math let-mutable let-rec match math-cmd mod module not of
open sig struct then true type val when while with
```

変数名に関して少しだけ注意を喚起しておくと，“整数を格納した変数 `x` から 1 引く式”を

14 この仕様は SATYSF$_{\mathrm{I}}$ の処理系の実装に使われている OCaml の `int` 型の仕様に起因する．

15 `controls` と `cycle` は今では特に使われないのだが，かつてグラフィックスに用いられていた TikZ 風の構文の名残として予約語のままになっている．

意図して `x-1` と書くと，意図とは異なる結果になる．というのも，これは `x-1` 全体でひとつの変数名とみなされてしまうためだ[*16]．回避策として，2 項演算子を使うときには `x - 1` のように必ず前後に空白文字をひとつずつ入れることを習慣づけるとよいだろう．前後に空白文字を入れていないことによりトークンの切り分け方が意図とは異なる結果になることは `-` 以外にも多くのパターンがあるので，2 項演算子を式と式の間に中置して使うときには前後に空白文字を入れることを強く推奨する．

さて，肝腎の変数への値の束縛は

```
let x = expr1 in expr2
```

という形をもつ let 式 (*let-expression*) で行なう[*17]．x が束縛される変数名であり，$expr_1$ および $expr_2$ は式である．この let 式は，$expr_1$ を評価した結果の値を変数 x に束縛し，そして $expr_2$ を評価するのに使う．要するに，$expr_2$ 中では変数 x が $expr_1$ を評価した結果の値として使えるのである．実際に使ってみよう：

```
+p{ Answer:
  \show-int(
    let n = (1 + 2) * 3 - 4 in
      n * n * n
  );
}
```

Answer: 125

あまりにも大したことのない例だが，期待どおり計算できていることがわかる．

ここですでに C 言語や Python などの “手続き的な” 言語に慣れ親しんでいる人は特によ

16 このような記述をすると，多くの場合は型検査器が「`x-1` という未定義の変数が使われている」というエラーを出すが，`x-1` という名前の変数が整数を格納する変数として本当に使われている場合はとても厄介で，発見困難なバグの原因になるだろう．

17 ここにイタリック体で x とか $expr_1$ などと書いたのは “メタ文字” であり，実際にコード中に `x` とか `expr1` といった文字列を書いているわけではなく，適宜あてはめるべきものに読み替えよ，ということを指す記述である．以降もこの “メタ文字” は頻繁に登場するので是非慣れて頂きたい．

く注意されたいのだが，束縛は「代入」とは似て非なる概念である．SATYSF$_{\mathrm{I}}$ に於ける変数は（OCaml などその他のいわゆる函数型言語の変数にも共通するのだが）“値を入れておき，適宜変更できる箱” ではない．むしろ，“値に別名をつけたもの” という直観の方が正確である．

3.2.3. 基本的なデータ

SATYSF$_{\mathrm{I}}$ で扱えるデータは勿論整数だけではない．浮動小数点数，文字列，真偽値，さらには組版指定のための**長さ** (*length*) の値などが扱える．長さについては3.2.5 節で，真偽値は 3.2.7 節で条件分岐とともに解説するので，ここではひとまず浮動小数点数と文字列について紹介する．

まずは小数などの値を扱うためのデータ，**浮動小数点数** (*floating-point number*) である．SATYSF$_{\mathrm{I}}$ での浮動小数点数の扱いは IEEE754 [10] に準拠するものとする．また，先走って型について述べておくと，浮動小数点数は float 型をつけて扱われる．浮動小数点数の値を PDF に出力して確認するために，最初に掲げたテンプレートの段落の部分を以下のように `\show-float` に変更しよう：

```
+p{ Answer: \show-float(42.195); }
```

整数の `\show-int` のときと同様に，上の `42.195` と書いてあるところを変えて遊んでみる．浮動小数点数のリテラルは単純に `42.195` とか `3.14` などと書く．現在のところ「`1e-5`」のような指数表記は用意していない．小数部分が `.0` の場合は `5.0` を省略して `5.` といった具合に書けるが，整数部分が `0.` の場合は `.5` といった具合には省略できない．浮動小数点数も整数と同様に加減乗除などの計算ができるが，演算子は `+` や `*` ではなく，かわりに `+.` や `*.` を使う：

```
+p{ Answer: \show-float((1.1 +. 0.7) *. 2.4); }
```

Answer: 4.32

整数値と浮動小数点数との演算は，直接はできない．型については後述するが，`1 +. 2.0` のような記述は型エラーとなる．評価されて整数値になる式 *expr* を浮動小数点数に関する計算に用いたい場合は `float` *expr* で浮動小数点数に変換する：

```
+p{ Answer: \show-float((float (1 + 2 * 3 - 4)) *. 2.4); }
```

Answer: 7.2

`float` とそれに伴う式との結合は 2 項演算子より強いので，`float 1 + 2 * 3 - 4` と書くと `(float 1) + 2 * 3 - 4` と等価な式として解釈されてしまうことに注意．反対に，浮動小数点数へと評価される式 *expr* を小数部分を切り捨てて整数値として使いたい場合は `(round` *expr*`)` という形で書く：

```
+p{ Answer: \show-int((round ((1.1 +. 0.7) *. 2.4)) * 2); }
```

Answer: 8

ただし，`(round` *expr*`)` は *expr* の評価結果の浮動小数点数が正負無限，NaN，または小数部分を切り捨てた整数が int 型に収まらない場合の動作は未定義である．なお，ここで用いた `float` および `round` は実は**函数** (*function*) と呼ばれるものであり，後の3.2.6 節でこれらについてより一般性の高い説明を与える．

続いては文字列である．文字列の “計算結果” を実際に PDF に出力するため，やはり最初に掲げたテンプレートの段落の部分を `\show-string` に変えて使おう：

```
+p{ Answer: [\show-string(`this is a string`);] }
```

この `` `this is a string` `` の部分を色々変えて遊ぶのである．`\show-string(…);` のさらに外側が `[` と `]` で囲まれているが，これは単にインラインテキストの一部として書かれた四角括弧の文字で，`\show-string` によって表示される文字列の端がどこなのかを見やすくするためにつけているだけである．要するに表示させたい文字列値のいずれか一方の端が空白文字であるとき，本当にその端に空白文字が入っているかを確認する目的でつけている．

さて，先走って登場してしまったが，SATYSF$_{\mathrm{I}}$ での文字列リテラルは両端をバッククォート（`` ` ``）で囲って記述する．多くのプログラミング言語では文字列内容を二重引用符で囲い，キャリッジリターンや二重引用符などいくつかの特殊文字はバックスラッシュでエスケープして書くような方式が採られるが，SATYSF$_{\mathrm{I}}$ にはそのような文字列リテラルの記法はなく，代わりに一切エスケープの要らないバッククォートによるリテラルだけが備わっている．一切エスケープが要らないといっても，「`foo`bar」のようにバッククォートを文字列内容に含めたい場合はどうすればよいのだろう？　これは Markdown を書いた経験があれば想像しやすいが，バッククォートが内容に現れる場合は両端のバッククォートを 2 個ずつにするのである．これにより文字列内容の終端がどこであるかがわかる．試しにテンプ

レートの段落部分を

```
+p{ Answer: [\show-string(``foo`bar``);] }
```

にしてみると，結果は以下のようになる：

Answer: [foo`bar]

一般化して言うと，文字列内容として n 個隣接したバッククォートを含めたいとき，左右で同数の $(n+1)$ 個以上の連続したバッククォートで両端を囲めばよい．

ここで，良い意味で性格の悪い読者はコーナーケースに気づいたかもしれない： 文字列内容の（少なくとも一方の）端にバッククォートがくる場合はどうすればよいのだろう？ これは簡単で，文字列内容の端がバッククォートの場合は，文字列を括るためのバッククォートの連続と文字列内容の端のバッククォートとの間に空白文字を挿入するとよい．バッククォートで括られた内容は，両末端の空白文字の 0 個以上を除去して文字列内容にするのである：

```
+p{ Answer: [\show-string(`` `bar``);] }
```

Answer: [`bar]

たしかに空白文字が除去され，左端がバッククォートになっていることがわかる．さて，読者が次に思うことはもう私にはよくわかっている： 文字列内容の（いずれかの）末端を本当に空白文字にしたい場合はどうすればいいのだろう？ この疑問にもしっかり答えられるので安心してほしい．もし左端の空白文字の 0 個以上の連続を無視させたくないのであれば，文字列の括る左側のバッククォートを `` ` `` ではなく `#` ` にするのである．バッククォートが 2 個以上の場合も同様に `#``` の要領で `#` を左につける．右端の空白文字の 0 個以上の連続を無視させたくない場合も同様に文字列を括る右側を `` ` `` ではなく `` `# `` とする．バッククォートが 2 個以上の場合も同様である：

```
+p{ Answer: [\show-string(#`` `bar``);] }
```

Answer: [`bar]

たしかに空白文字が残っている．これでようやく安心して文字列リテラルが使える．

文字列リテラルが書けるようになったところで，文字列を構築する方法を伝授しよう．文

字列の結合は 2 項演算子 `^` でできるほか，整数値に評価される式 *expr* に対し，`(arabic` *expr*`)` で整数値を十進法文字列にすることができる：

```
+p{ Answer: [\show-string(
  `The result is `# ^ (arabic ((1 + 2) * 3 - 4)) ^ `!`
);] }
```

Answer: [The result is 5!]

ここで使った `arabic` はやはり函数であり，これに関しては3.2.6 節でしっかりと扱う．

なお，インラインテキスト `{…}` と文字列は異なる概念であることに注意されたい．前者はコマンドなどを内容に含んでいるが，後者は純粋に Unicode コードポイント列である．ただし，文字列を（等価なテキストで構成された，コマンドを含まない）インラインテキストへと変換することは `embed-string` という函数で行える．これについても3.2.6 節で扱う．

3.2.4. 型システムの基礎

ところで，“文字列を整数値に足そうとする” とどうなるだろうか？　結論から言うと，そのような計算は意味をなさず，しかも型検査器に弾かれて “計算を試みることさえさせてもらえない”．整数の計算に使った `\show-int` を含む最初のテンプレートで，試しに以下のように記述して処理してみよう：

```
+p{ Answer: [\show-int(
  42 + `added string`
);] }
```

次のようなエラーが出るはずである：

```
! [Type Error] at "test.saty", line 12, characters 9-23:
    this expression has type
      string,
    but is expected of type
      int.
```

ファイル名 `test.saty` や行数などは実際にどう記述したかで変わるのだが，　どうであ

れ実際に `` `added string` `` が書かれている位置を指しているはずだ． 上記のエラーは SATYSF$_{\mathrm{I}}$ の型検査器が出した**型エラー** (*type error*) であり，その意味するところは（ほぼ直訳だが）「`` `added string` `` という式は string 型の式だが，ここには int 型の式が書かれることが期待されている」 だ． ここでようやく SATYSF$_{\mathrm{I}}$ の特徴たる**型** (*type*) の登場となる[*18]．一旦上のような具体的な型エラーを離れ，型についての一般論の時間にしよう．型とは，非常に大雑把に言えば“値の種類による分類”，さらには“評価した結果どんな種類の値になることが事前にわかっているかに基づく式の分類”である．ここで“事前に”と言っているところが非常に重要だ．要するに，“計算を実際に行なわなくともどんな種類の値が結果として出てくるのかが式の字面だけからわかる”ような仕組みが型によって確立されているということであり，この仕組みが**型システム** (*type system*) と呼ばれるものだ．型システムを反映して与えられたプログラムが整合しているかどうかを判定する機構が型検査器であり，この“整合しているかの検査”はプログラムをなす各部分の式に型がつくかどうかを判定することで行なわれる．

式に型をつけることの直観は単純である． まず， 整数値の型として （上の型エラーにも出てきたように）int 型が用意されている． 最初から値であるような `1` とか `2` とか `500000000` といった整数リテラルの式は単純にそのまま int 型がつけられる． 重要なのは， まだ値でない， 評価すると“何かしら計算が進む”式の型つけだ． 例えば， 加算の式 $expr_1$ + $expr_2$ はどのように型で扱うかというと，「$expr_1$ + $expr_2$ に何らかの型がつくならそれは int 型であり，そのためには $expr_1$ と $expr_2$ にそれぞれ int 型がつかねばならない」という規則が設けられているのである． これが型システムという仕組みの大枠だ． 本当はもう少し一般性の高い仕組みなのだが，それは次節での函数の型つけの説明に譲る．

さて， 上で出た型エラーの話に戻る． このエラーの要因は， 要するに $expr_1$ + $expr_2$ という形の式に型をつけるには上記の説明どおり $expr_1$ と $expr_2$ のそれぞれに int 型がつくことが要請されるのだが， ここでは $expr_2$ が `` `added string` `` という文字列リテラルであり， これは（string 型はつくが）int 型がつかないと定められている． したがって元の加算の式全体には型がつかないことになり，型検査で弾かれるのである．

3.2.5. 長さ

型システムの基礎を把握してもらったところで， 組版処理上基礎的なデータのひとつで

---

18 といっても，プログラミング言語のもつ特徴としては型はごく一般的な概念である．型システムを有していることではなく，型システムを組版処理を行なう処理系に持ち込んだことが SATYSF$_{\mathrm{I}}$ の顕著な特徴なのである．

ある**長さ** (*length*) とその型である length 型を紹介しよう．これはおそらく汎用のプログラミング言語にはそれ以上分解できない機能として搭載されているのをまず見ないデータだろう．長さの値は振舞いの上ではほぼ浮動小数点数のようなものではある[*19]が，“長さ同士を乗算したものを長さとして使う” といった “意味をなさない計算” が安易にはできないように，つまりそういった計算をしようとする記述が型検査を通らないように，浮動小数点数とは型の上で区別されている．長さの定数リテラルは，`2.5cm` とか `-1inch` といった，次に該当する構文で書く：

(-｜)(〈非負で数値の浮動小数点数〉 ｜ 〈非負整数〉)〈単位〉

ただし，〈単位〉の部分は変数名と同一の規則の文字列である．これ全体が長さの定数リテラルをなす 1 トークンであり，負の符号と数値部分の間や数値部分と単位の間に空白文字を入れたりしてはいけない．具体的な単位については，あらかじめ定義された以下の（“状況” に依存しない，絶対的な）単位のみが使える：

- `mm`： ミリメートル
- `cm`： センチメートル
- `inch`： **国際インチ** (*international inch*)（1 inch を厳密に 2.54 cm と定めた単位）
- `pt`： **DTP ポイント** (*DTP point*)（1 pt を厳密に $(1/72)$ inch と定めた単位）

em や ex などといった，“現在の文字サイズ” に依存する相対的な長さの単位は使えないことに注意されたい．勿論実質的に em 単位で長さを指定する方法は存在し，それについては後々わかることになる[*20]．さらに引っかかりやすいポイントは，ポイントという単位の定義だ．“point” と呼ばれる単位には歴史的に様々な定義が与えられてきた．代表的なものを掲げる：

---

19 現状，長さに関する演算がどのように振舞うかは厳密には規定しない．厳密に規定しなければどのように行分割されるかが “処理系依存” になってしまうので将来的には規定した方が望ましいのであろうが，今のところ公開されている SATYSFI の処理系は著者が実装した 1 つ（およびそれを他者が fork し長さの扱いに関係しない箇所を改変したもの）しかないと思われるのでひとまず気にしなくてよい．なお，現在の処理系の実装では（後述する）DTP ポイント単位の浮動小数点数で内部的に表現されている．

20 大幅に先走って述べておくと，そのような処理は `get-font-size` というプリミティヴを用いて実現する．

- Didot ポイント (*Didot's point*)（1783 年）： pied du Roi（royal foot，王のフィート）に基づくフランスのインチの厳密に $(1/72)$ 倍.
- アメリカンポイント（1886 年）： インチの厳密に 0.013837 倍.
- JIS Z 8305 に基づくもの（1962 年）： 厳密に 0.3514 mm.
- TEX ポイント `pt`（1982 年）： 国際インチの厳密に $(1/72.27)$ 倍.
- TEX での Didot ポイント `dd`（1964 年）： TEX ポイントの厳密に $(1157/1238)$ 倍.
- DTP ポイント（1984 年）： 国際インチの厳密に $(1/72)$ 倍. PostScript [1] の仕様で導入され，PostScript ポイント (*PostScript point*) とも呼ばれる．CSS で扱われる `pt`，および TEX のビッグポイント (*big point*) `bp` も，これに等しい[*21].

CSS，PostScript，PDF（および，`bp` という単位だが，TEX）などに広くまたがって互換性に優れたデファクトスタンダードとして確立されていることを鑑み，SATYSFI は `pt` の定義として DTP ポイントを採用した.

長さに関する基礎的な演算を行なう式は次のとおりである：

- l_1 `+'` l_2： 長さ同士の加算.
- l_1 `-'` l_2： 長さ同士の減算.
- l `*'` r： 長さ l と浮動小数点数 r を受け取り，l を r 倍した長さを返す.
- l_1 `/'` l_2： 長さ l_1 が長さ l_2 の "何倍の長さであるか" の浮動小数点数を返す.

3.2.6. 函数定義

大抵のプログラミング言語には，頻出する計算手順に名前をつけ，その計算手順を 1 箇所にまとめて記述する方法が備わっている．多くの言語と同様に，SATYSFI では函数 (*func-*

21 ただし，TEX での長さの扱いはスケールドポイント (*scaled point*) `sp` という TEX ポイントの厳密に $(1/65536)$ 倍の長さの整数倍として表現され，ビッグポイントも TEX ポイントの厳密に $(65781/65536)$ 倍として扱われているので，定義と実装で差がある．といっても `sp` は 10^{-6} mm のオーダーなので，視覚表現上はこの単位に対する丸め誤差を気にする必要がない.

tion) がその役割を担う[*22]．函数は，何らかの値を受け取り，それを加工して新たな値をつくる処理である．例として，円単位の金額を整数で受け取り，それをドル単位に換算する函数 `to-dollar` を定義してみよう．レートはひとまず 9 月 24 日 19 時 23 分 (UTC) に基づいて 1 ドルを 112.73 円とする．これは以下のようにして定義できる：

```
let to-dollar m-yen =
  let rate = 112.73 in
    (float m-yen) /. rate
in
（続きの式）
```

函数の定義も let 式を用いて行なうのである．ここで登場する `m-yen` は**仮引数**または単に**引数**と呼ばれるもので，函数 `to-dollar` の受け取る値が格納される変数である．ここで例示した `to-dollar` はひとつだけ引数をとる函数だが，一般には任意に正整数個とれる．函数定義は，次のような構文によって為される：

let $f\ x_1 \cdots x_n$ = $expr_1$ in $expr_2$

f が定義したい函数に与える名前であり，各 x_i は第 i 引数である．

さて，上で定義した函数 `to-dollar` を実際に使ってみよう．函数の定義は，パッケージの読み込みの記述から `document` までの間の部分で行なえる．すなわち，以下のようになる：

```
@require: stdjareport

let-inline \show-int n = embed-string (arabic n)
let-inline \show-float x = embed-string (show-float x)
let-inline \show-string s = embed-string s
in

let to-dollar m-yen =
  let rate = 112.73 in
    (float m-yen) /. rate
in
```

22 函数のほかには，Fortran90 などの言語に見られるように**サブルーティン** (*subroutine*) と呼ばれる機構がこの役割を担っている言語もある．

```
document (|
  （以降略）
```

それでは実際に `to-dollar` を使って円単位の金額をドル単位に換算してみよう． 函数 f に式 *expr* の計算結果を渡したいときは f *expr* と書く． これを函数 f の式 *expr* への**適用** (*application*) とか式 *expr* による函数 f の**呼び出し** (*call*) と呼ぶ．また，ここで評価結果が f に渡される式 *expr* のことを f の**実引数**と呼ぶ．やや厄介なことに，仮引数だけでなくこの実引数も単に**引数**と呼ばれることが多い．しかし「引数」が仮引数の意であるか実引数の意であるかはほとんどの場合使用されている文脈から判断できるので，特に違いを強調する必要のない場合は両者ともに引数と書くことにする．

函数 f の式 *expr* への適用は，次のような流れで評価される：

- 実引数 *expr* が評価されて値 *value* になる．
- 函数 f の定義 `let` $f\,x$ = e `in` $\cdots$ が “思い返される”．
- 仮引数 x が *value* に束縛された下で “定義の本体” である式 e が評価され，その結果が適用の式全体の評価結果となる．

不正確ながら大雑把に言ってしまえば，函数定義中の仮引数が実引数で置き換えられたものを計算するのが函数適用である．さて，実際に `to-dollar` を適用したのが以下だ：

```
+p{ Answer: \show-float(to-dollar 10000); }
```

Answer: 88.7075312694

やはり大雑把に言えば，`to-dollar` の “定義本体” の中の `m-yen` がすべて `10000` に置き換わったものの計算結果が出ているわけである．というわけで 10000 円は大体 88.7 ドルくらいであることが計算できた．

函数それ自体にも， 整数などと同様に型がつけられる．τ_1 型の引数をとって τ_2 型の値を返す函数には $\tau_1 \to \tau_2$ という型がつくと定められている． 例えば， 上で定義した `to-dollar` は int $\to$ float 型をもつ．このことを以降

`to-dollar` : int $\to$ float

とコロン（:）を挟んで式とそれにつく型を並べることで表記する．

さて，引数が 1 個の函数はよいとして，複数の引数をとる函数も同様の例を通じて説明する．先ほどの `to-dollar` は交換のレートが決め打ちされていた．今度はレートも引数にして，その都度変えられるようにしてみたい．その定義が以下である：

```
let to-dollar-2 rate m-yen =
  (float m-yen) /. rate
```

レート `rate` が第 1 引数に，日本円の額面 `m-yen` が第 2 引数になるように改めたのである．さて，9 月 26 日 16 時 44 分 (UTC) のレートは 1 ドル 112.95 円であった．このレートであらためて計算してみよう．引数が n 個あるとき，函数適用は $f\ expr_1 \cdots expr_n$ という形で書く：

```
+p{ Answer: \show-float(to-dollar-2 112.95 10000); }
```

Answer: 88.5347498893

ところで，引数が複数個ある函数はどのような型がつくのだろう？　結論から言うと，引数を n 個とり，各第 i 引数の型が τ_i，返り値の型が τ であるような函数には

$$\tau_1 \to \cdots \to \tau_n \to \tau$$

という型がつく．例えば上で定義した `to-dollar-2` は float → int → float という型をもつという具合である．この矢印が連なった型は実は新たな記法などではなく，括弧を省略せずに書くと

$$\tau_1 \to \Big(\tau_2 \to \Big(\cdots \to \big(\tau_n \to \tau\big)\Big)\Big)$$

という型である．つまり“矢印で構築される函数の型 $\tau \to \tau'$ の τ' の側がさらに函数の型になっている”という構造が繰り返されてできているのが上記の型なのである．だが，どうして n 引数函数の型がこのように表されるのだろうか？　これは次のような考察に基づくものだ：

> τ_1 型の第 1 引数と τ_2 型の第 2 引数を受け取って τ 型の値を返す函数は，τ_1 型の引数を受け取って「τ_2 型の引数を受け取って τ 型の値を返す函数」を返す函数ともみなせる．

これはいわゆる函数型言語が基盤とする理論の中でも初歩となる重要な点で，この考察に基

づく“変換”を**カリー化** (*currying*) と呼ぶ．このことは次に説明する**部分適用** (*partial application*) を念頭に置くととても自然に理解できる．部分適用とは，簡単に言えば“一部の引数しか渡さないこと”である．先ほど定義した `to-dollar-2` に，第 1 引数であるレートの値のみ渡して部分適用することにより別の函数を定義してみよう：

```
let to-dollar-par = to-dollar-2 112.73
```

この `to-dollar-par` は「日本円での額面を受け取り，それをレート 112.73 円 / ドルに基づいてドルの額面にして返す函数」であり，型として int → float をもつ．int → string 型の函数に引数として int 型を渡すと返り値の型が string 型であるのと全く同様に，float → (int → float) 型の `to-dollar-2` に float 型の式を渡す記述だから int → float 型なのである．こうして定義した `to-dollar-par` は，最初に掲げた `to-dollar` と等価な振舞いをする函数になっている．また，部分適用と対応するように，実は複数の実引数の適用

$$f\ expr_1 \cdots expr_n$$

も，括弧を略さずに書くと

$$(((f\ expr_1)\ expr_2) \cdots)\ expr_n$$

である．これは函数適用 $expr_1\ expr_2$ に於いて $expr_1$ が $\tau \to \tau'$ 型を，$expr_2$ が τ 型をもつことが検査されることから考えると自然だろう．平均的な型システムの入門的解説に比べるとかなり駆け足になってしまったが，以上が函数につけられる型と部分適用の仕組みだ．

ところで，これもいわゆる函数型言語が基盤とする理論としては重要な点なのだが，函数はいちいち let 式で名前をつけて扱う必要はない．例えば，上で let 式で定義した `to-dollar-2` は `to-dollar-2` と名前をつけずとも以下のように**無名函数**[*23]で書ける：

```
(fun rate m-yen -> (float m-yen) /. rate)
```

勿論この例ではわざわざこう書く必要はないのだが，後述する高階函数では役立つことが多い．「役立つことが多い」などと，あたかも無名函数が“後から言語機能に足された瑣末なもの”であることを示唆するかのような言い方になってしまったが，実はこの無名函数は let 式による函数の定義よりも言語機能として基盤にあるものだ．函数を定義する let 式

23 理論的には**ラムダ抽象** (*λ-abstraction*) と呼ばれるものである．

```
let f x1 ··· xn = expr1 in expr2
```

は，実は以下の糖衣構文にすぎない：

```
let f = (fun x1 ··· xn -> expr1) in expr2
```

つまり，前者は処理系で内部的に（構文解析が終わり，型検査が始まる前に）後者に変換されているのである．後者の let 式は3.2.2 節で最初に紹介した let 式そのものである．すなわち，函数定義は単に変数束縛の一種だったのである．“函数” と “変数” の垣根がないことは，SATYSFI が拠り所とする ML 系言語の重要な特徴である[*24]．ちなみに，もっといえば `(fun x1 ··· xn -> expr)` も `(fun x1 -> (fun x2 -> ··· (fun xn -> expr) ···))` の糖衣構文である．これは先ほど部分適用で説明したこととも対応しており，要するに n 引数をとる函数とは “1 引数函数が n 重入れ子になったもの” である，ということだ．

すでに述べたが，浮動小数点数や文字列などのデータの説明で登場した `arabic`，`float`，`round` は，それぞれ以下のような型をもつプリミティヴの函数である：

- `arabic` : int → string
- `float` : int → float
- `round` : float → int

函数につく型については上記の説明ですでによく把握されているだろうから，それぞれにこのような型がつくことはもはや当然のように感じられるだろう．

さらには，2 項演算子も実は函数である．`expr1 + expr2` は実は `(+) expr1 expr2` という函数適用の糖衣構文であり，2 項演算子 `+` の正体は `(+)` という 3 トークンからなる特殊な名前をもつ変数なのである．`(+)` は int → int → int という型をもち，それゆえに int 型のつかない引数を与えると型検査器にエラーを返されていたのである．さらに，2 項演算子は（通常の変数名の函数と同様に）自分で定義することもできる．使える名前は，`(+ | - | * | / | ^ | & | | | ! | : | = | < | > | ~ | ' | . | ?)` を *opsymb* と表すとして

24 いわゆる函数型言語にあっても，“函数” と “変数” とが別個の概念としてある言語は存在する．Common Lisp などが有名である．

- `(* | /)`[*opsymb*]*
- `(+ | -)`[*opsymb*]*
- `~`[*opsymb*]*
- `(= | > | <)`[*opsymb*]*
- `&`[*opsymb*]$^+$
- `|`[*opsymb*]$^+$

にマッチするもののうち，以下に該当しないものである[*25]：

```
= -- ->
```

結合の強さは先に挙げたものほど強いとする．いずれも左結合である．

3.2.7. 真偽値と条件分岐

真偽値は“正しいか正しくないか”を表すデータであり，bool 型をつけて扱う．bool 型のつく式はおおよそ“命題”に対応し，bool 型の値としては真に対応する true と偽に対応する false との 2 種類のみがある．

真偽値に関連する言語機能として，“何が成り立っているかによってその先の処理を変える”記述， すなわち**条件分岐** (*conditional branching*) がある． 例として整数の絶対値函数 `abs` を定義してみよう：

```
let abs n =
  if n < 0 then - n else n
```

ここで使われる `if` $expr_0$ `then` $expr_1$ `else` $expr_2$ が条件分岐に使われる if 式 (*if-expression*) であり，直観的には「$expr_0$ が成り立つならば $expr_1$ を，そうでないならば $expr_2$ を評価する」を意味する． 型つけの規則としては，$expr_0$ は bool 型であることと，$expr_1$ と $expr_2$ とに

25 `--` は実質的に使われなくなっているが，歴史的経緯から特殊なトークンとして残存している．

同一の型がつくことが要請される．ここでの例は，整数を比較する 2 項演算子 `<` を用いて与えられた引数が `0` 未満であるかどうかを判定しており，負の値であった場合は符号を反転し，非負の値であった場合はそのまま返す，ということを記述している．ちなみに，条件分岐の話とは関係ないが，注意深い人は `- n` という部分が新しい構文であることに気づいただろう．これは `-` という名前の 2 項演算子のみに与えられた特殊な機能で，大雑把に言えば“左側の引数を省略した場合は `0` として扱う”というものだ．要するに符号の反転を自然に書くための機能である[*26]．ともあれ，この `abs` の動作を確認してみる：

```
+p{ Answer: \show-int(abs (abs (1 - 2) * 3 - 4)); }
+p{ Answer: \show-int(abs (abs (1 + 2) * 3 - 4)); }
```

```
Answer: 1

Answer: 5
```

論理演算として選言 `||` : bool → bool → bool，連言 `&&` : bool → bool → bool，否定 `not` : bool → bool などが用意されており，これを用いてより複雑な条件の指定ができる．最初から使える種々の演算は3.2.14 節に掲載したので参照されたい．

3.2.8. 型システムと型推論の恩恵

式に型がつくことで，その式を評価しても“値の種類に由来するエラー”は起きないということが保証される．というか，型システムはこういったエラーをプログラムを評価せず事前に検出して弾くためにあるわけである．型なしの言語（いわゆる動的型つき言語[*27]）に馴染みの深い読者なら，もしかすると「静的型を入れず，整数値に文字列を足す処理が書かれたら前者を十進文字列にして文字列結合にするように `+` を定義すればよいのでは？　逐一 `arabic` で整数値を文字列値へ変換する処理を明示的に書く必要もないし楽になるだろ

26 浮動小数点数用の `-.` や長さ用の `-'` には，この機能は現状備わっていない．将来的に追加したいとは考えている．

27 「動的型」と言ったときの「型」は，ここで説明した型の概念とは似て非なるもので，“実行時にデータの種類を示すものとして値についているタグ”であることに注意されたい．繰り返しになるが，この本で使用しているのと同様の意で「型」と言った時は“式に対して事前にわかる分類”である．動的型との混同を避けて俗に静的型と呼ばれることもある．歴史的には動的型を指して「型」という語を使う方がずっと後発の用法である．

う」などと考えるかもしれない．だが，$expr_1$ + $expr_2$ の形の式の計算結果が整数値にも文字列値にもなりうるというのはそんなに“うれしいこと”ではない．式に型をつけないという言語設定を仮定した下で，次のような函数を考えよう：

```
let f x =
  let y = 42 + x in
    y / 2
```

型なしの言語だったりして，整数値と文字列値の加算が許容されるとしよう．このとき，上記の `f` に渡される `x` の“値の種類”によっては，除算 `y / 2` に於いて `y` が文字列値など整数値でない場合も扱わねばならないことになる．除算の定義をどうするか，ここで 2 通りの選択肢がある：

- 整数値以外が左側にきた場合は実行時エラーとする．
- 整数値以外が左側にきた場合，結果を必ず `0` にしたり，何も処理せずその値を返したりと，とにかく何かしらそれらしい値を返して実行時エラーにはしない．

前者を採用した場合，これはまさに型システムを入れることで除外したかった“値の種類による実行時の破綻”だ．「この実行時エラーを許容してもよいから静的に型をつけたくない」という主義の人もいるかもしれないが，ほとんどの場合その決断は割りに合わないと言ってよい．型なしの言語で数千行規模のプログラムを書くと（静的型つきの言語で書いた場合に比べて）急速に保守が困難になることは想像に難くないし，経験的にもそうであることが多いはずだ．もし貴方が数千行規模のプログラムを型なしの言語で書いて保守しており，かつ特に苦労していると思っていないなら，それは単に型なし言語のデバッグの労苦に飼い慣らされていて，静的型つきの言語の世界観から見れば大部分は要らぬ苦労をしているだけ，という可能性は非常に高い．一方で後者を採用した場合，計算が失敗することはないが，些細な間違いに対してもデバッグが非常に困難になる．どこで意図しない計算が生じているか事細かに追わないとわからないからだ．特に組版処理のように“扱う値の種類”が多い場合，“種類の違い”による意図しない計算は生じるリスクが大きく，致命的に修正困難な状況に陥りやすいだろう．要するに，“値の種類によらず計算に失敗しないようにする”ことも“そんなに嬉しくない”のである．

とにかく，人間はプログラムを書く際にはひたすら間違い続ける．しかもその間違いは，かなりしょうもないものであることが経験的に多そうだ．型検査器は人間の犯すミスのうち或る種の不整合を丁寧にも事細かに指摘してくれる．決してお節介なお婆さんではない．「“式の種類”に関して不整合のあるプログラムを書いたときに型検査器がその不整合を教えてくれ，それに基づいて該当箇所をスムーズに修正できる」という利点は，「“異なる種類”へ

の値の変換を `arabic` のような形で明示的に書かなければならない」という多少の厄介さを補ってあまりがありすぎるほどの恩恵である．また，これは非常に誤解が多いのだが，プログラムに静的に型をつけるのに**型註釈** (*type annotation*)，つまり各変数にどのような型をつけるかを明示した註釈が必要なわけでは必ずしもない[*28]．適切に型システムを設計すれば，与えられたプログラムが“整合している”限り，全部または一部の変数にどのような型がつくか，型推論で求められるのである．特に $\mathrm{SATYSF_I}$ を含め OCaml，F#，Standard ML，Haskell など多くの言語の型システムがその理論的基盤としている Damas–Hindley–Milner **多相**[*29] [9, 21] は，型推論が**完全性** (*completeness*) をもつ，すなわち変数に一切型註釈のない式に対しても必ずその式につく“最も一般的な型”である**主要型** (*principal type*) が存在してかつそれを求めるアルゴリズムが知られているという性質があり，それゆえに特に $\mathrm{SATYSF_I}$ や OCaml ではよほど複雑な機能を使わない限り基本的に型註釈を書かなくてよいという状況が達成されている．つまり型註釈が現実的にプログラマの手を煩わせることはほとんどない．むしろ型註釈が省略できすぎるので，（2 週間後の自分が見てプログラムを読めなくなることのないように）型検査器からすればなくてもよい型註釈を可読性のために入れるくらいである[*30]．こうした考察から多くの言語には型システムによる静的な検査が搭載されており，我々のコーディングを型検査器が支えてくれるようになっている．$\mathrm{SATYSF_I}$ も，そんな言語のひとつである．

3.2.9. タプル

タプル (*tuple*) とは“複数の式を並べてひとまとめにした式”であり，数学的に言えば直積に相当するものだ．タプルは式として次のような形をもつ：

---

28 「静的型つきの言語は逐一変数につく型を書かないといけないから面倒」という誤解は，おそらく C 言語のような型註釈が省略できない（或る意味特殊な）静的型つき言語の存在によるところが大きそうだ．一般に静的型つき言語で変数の型註釈が省略できないわけでは決してない．

29 Hindley–Milner **多相**，Damas–Milner **多相**，let **多相**，**冠頭多相**などとも呼ばれる．

30 Haskell は let 多相を基盤としていながらも**型クラス** (*typeclass*) がある影響で型註釈が省略できない場合が顕著にみられ，それゆえかトップレヴェルで定義する函数には必ず型註釈を入れる慣習がある．たまに「型が函数定義の本体であって，実装はオマケにすぎない」という主張をする人がいるが，これは（**篩型** (*refinement type*) [6, 15, 30] のような，通常の型よりもずっと強力な型を扱う型システムでもない限り）過言気味だ．型から実装が決まるわけでは決してないためだ．例えば返り値の型が bool の函数なら，“最後に `not` をかませるような実装に変更する”だけで元と同一の型がつくが振舞いの全く異なる函数の実装ができてしまう．

```
(expr_1, ..., expr_n)
```

ただし $n \geq 2$．各式 $expr_i$ が τ_i をもつとき，上記のタプルは以下のような型をもつ：

$$\tau_1 * \tau_2 * \cdots * \tau_n$$

代表的な用途として，“複数の値を返り値にしたい時” がある．例として，2 つの整数を受け取ってその除算の商と剰余を一緒に返す函数 `div-rem` を定義してみる：

```
let div-rem m n =
  (m / n, m mod n)
```

こうして定義した `div-rem` には int → int → int ∗ int 型がつく[*31]．一方で，これを使う側はどうしたらよいのだろう？　このために，タプルを “分解” する方法が用意されている．**パターンマッチ** (*pattern matching*) と呼ばれる構文である．パターンマッチは条件分岐を強化したようなもので，“値の形” によって分岐させる機構である．`div-rem` を用いて，“第 1 引数の整数以上の値にするには第 2 引数を少なくとも何倍すればよいか” を返す函数 `make-exceed` を定義してみる：

```
let make-exceed n q =
  match div-rem n q with
  | (m, 0) -> m
  | (m, _) -> m + 1
```

ここで登場した次のような構文の式がパターンマッチを実現する match 式 (*match-expression*) である：

```
match expr with | pat_1 -> expr_1 | ... | pat_n -> expr_n
```

ただし $n \geq 1$ で，`with` の直後の `|` は省略してもよい．各 pat_i は**パターン** (*pattern*) と呼ばれるものであり，“値の形” を表す．パターンの中には変数または**ワイルドカード** (*wildcard*) である `_` を含めてよく，そこは “どんな形でもマッチする” ものと扱われる．match 式は，まず *expr* を評価し，その結果得られた値が各パターン pat_i のどれに合致する “形” をしているかを先に挙げたパターンほど優先で選択し，パターン中の変数を該当箇所の実際の

31 型の記法では，∗ は → より結合が強いとする．すなわちここでは int → int → (int ∗ int) を指す．

値に束縛してから，それに対応する式 $expr_i$ を評価する．上記の例では，`div-rem` が返すタプルの 2 番目の値，つまり剰余に相当する部分が `0` である場合は，変数 `m` をタプルの 1 番目の値に束縛し，そのパターンに対応する式である `m` を返す．つまり，この場合は単に割った商を返している．一方で，最初のパターンにマッチしなかった場合は，2 番目のパターンにマッチさせ，同様に変数 `m` を用いて商を取り出し，それに `1` を加えて返す．パターンの順番は重要である．もし上の例で `| (m, _) -> m + 1` を `| (m, 0) -> m` よりも先に書いていたら，定義される `make-exceed` は単に "商に 1 足して返す函数" になってしまう（つまり，`n` が `q` で割り切れる場合の返り値が異なる）．型の観点からは，各 $expr_i$ の型はすべて一致する必要があり，その一致した型が match 式全体のもつ型になる．また，パターンがひとつだけの match 式 `match` *expr* `with` pat_1 `->` $expr_1$ は `let` を用いて

```
let pat1 = expr in expr1
```

という糖衣構文で書くことができる．別の例として "長さと平面座標を受け取り，その座標を $+x$ 方向に与えられた長さだけ移動させる函数" `shift-x` を定義してみよう：

```
let shift-x len point =
  let (x, y) = point in
  (x +' len, y)
```

このように `let` でタプルを分解できるのである．こうして定めた `shift-x` は length → length ∗ length → length ∗ length 型をもつ．さらには，パターンを引数に直接書くこともできる．以下のように書いても，上での定義と等価に振舞う函数が定義できる：

```
let shift-x len (x, y) =
  (x +' len, y)
```

駆け足気味であったが，以上がタプルとパターンマッチの基礎である．

3.2.10. 代数的データ型

代数的データ型 (*algebraic data type, ADT*) は，非常に大雑把に言えば "構成に独立な複数の方法があるデータ" を扱う型である．**ヴァリアント型** (*variant type*) とか**合併型** (*union type*)

と呼ばれることもあり[*32]，数学的には直和に対応する．

代数的データ型を扱う必要性を感じるために，まず「長方形または円からなる図形のデータを扱いたい」という場面を考える．長方形は対角線をなす 2 点を記録すればよく，円は中心座標と半径を記録すればよいだろう．それぞれ図形に対して一意的な表現方法にはなっていないが，それは特に問題にせず，単にデータとして不足なく保持できればよい．ここまでに紹介した言語機能を用いて長方形と円を一緒に扱うナイーヴな方法のひとつは，string ∗ length ∗ length ∗ length ∗ length という型のタプルを用いる方法だ．$\left(x_1, y_1\right)$ と $\left(x_2, y_2\right)$ とが対角線をなす長方形は `(`rectangle`,` x_1`,` y_1`,` x_2`,` y_2`)` で，(x, y) を中心とする半径 r の円は `(`circle`,` x`,` y`,` r`, 0pt)` でそれぞれ表すことにするのである．しかし，これはかなり“荒っぽい”方法であり，いくつもの問題を孕んでいる．まず `(`hoge`, 2pt, -5pt, 0pt, 3pt)` のような“意味をなさないデータ”が同じ型で作れてしまう．勿論図形を受け取る各函数でそのデータが正しいか逐一確認することはできるが，複雑な処理では確認を忘れてしまうことも十分ありうるし，何より正しくないデータだった場合に実行時エラーにするしかないのは不便である．このほか，図形が格納された変数 `geom` に対して `let (_, _, _, x, _) = geom in` ⋯ として `x` に値を取り出せるが，この `x` は図形ごとに全然違う役割を担っている．このような操作が安易にできてしまうと，複雑化するにしたがって見つけにくい実装のミスに繋がる．

代数的データ型を用いれば，このような状況をはるかにスマートに扱える．まずは具体例として，上で挙げた図形を扱うための型 geometry を定義してみる：

```
type geometry =
  | Rectangle of (length * length) * (length * length)
  | Circle    of (length * length) * length
```

この定義は，新たに定義する geometry 型に於いて

- $\left(x_1, y_1\right)$ と $\left(x_2, y_2\right)$ とが対角線をなす長方形は `Rectangle(`(x_1, y_1)`,` (x_2, y_2)`)` という形で扱われる
- (x, y) を中心とする半径 r の円が `Circle(`(x, y)`,` r`)` という形で扱われる
- この形以外のデータは扱わない

32 「合併型」や“union type”は Elm などの言語で用いられる用語だが，型システムの議論ではこれらの語がよく似た別物を指すこともあるので注意を要する．

ということを宣言している． この宣言以降，`Rectangle` *expr* および `Circle` *expr* で geometry 型のつく式を書ける．一般に代数的データ型の定義は以下のような構文で宣言される：

type $'a_1 \cdots 'a_m$ *tyname* = | Ctr_1 of ty_1 | ⋯ | Ctr_n of ty_n

ただし $m \geq 0$ かつ $n \geq 1$ で，また `=` の直後の `|` は省略でき，宣言される型の名前 *tyname* に使えるのは変数名に使える文字列である．各 Ctr_i は**コンストラクタ** (*constructor*) と呼ばれ，コード中の相異なる文字列である．コンストラクタとして使えるのは

- 1 文字目はラテン大文字
- 2 文字目以降は大小ラテン文字，数字，ハイフンのいずれか

を満たす 1 文字以上の長さの文字列である．*Ctr* of *ty* は *Ctr* と書く構文もあり，これは *Ctr* of `unit` と等価である．unit 型は唯一の値 `()` をもつ型で，3.2.13 節で説明するような手続き的処理の返り値や，手続き的処理を行なう函数の引数などにダミー的に利用される[*33]．$'a_j$ は**型引数**と呼ばれるものだが，今は触れず，$m = 0$ とする．

さて，代数的データ型とそれに属するデータの作り方はわかったが，そのデータをどうやって使えばいいのだろう？ 実は代数的データ型のデータも，タプルと同様にパターンマッチによって分解する．例として，上で定義した geometry 型のデータを受け取ってその（平方センチメートル単位の）面積を浮動小数点数で返す函数 `area` : geometry → float を定義してみる：

```
let pi = 3.1415926535

let length-abs len =
  if len <' 0pt then 0pt -' len else len

let area geom =
  match geom with
  | Rectangle((x1, y1), (x2, y2)) ->
      let w = (length-abs (x1 -' x2) /' 1cm) in
      let h = (length-abs (y1 -' y2) /' 1cm) in
        w *. h
```

33 このような用法を**サンク** (*thunk*) などと呼ぶ．

```
  | Circle(_, r) ->
      let rf = r /' 1cm in
        rf *. rf *. pi
```

与えられた図形が長方形なのか円なのかパターンマッチで分岐して処理している．*Ctr* が何らかのデータ型に属するものとして定義済みのコンストラクタであるとき，*Ctr pat* がパターンとして使え，*Ctr v* の形の値が（*v* が *pat* にマッチするときに）マッチするのである．上で定義した `area` を実際に使ってみよう：

```
+p{ Answer:
  \show-float(area (Rectangle((1cm, 2.4cm), (-3cm, 10cm)))); }
+p{ Answer:
  \show-float(area (Circle((5cm, 0cm), 2cm))); }
```

Answer: 30.4

Answer: 12.566370614

長方形と円のそれぞれについて正しく面積が計算できていることがわかる．

SATYSFI には，いくつかの代数的データ型がデフォルトで備わっている．特に組版処理特有のデータは次のように定義される color 型である：

```
type color =
  | Gray of float
  | RGB  of float * float * float
  | CMYK of float * float * float * float
```

`Gray`，`RGB`，`CMYK` はそれぞれグレイスケール，RGB，CMYK の色空間で色を表現するためのコンストラクタで，PDF の仕様に於ける `/DeviceGray`，`/DeviceRGB`，`/DeviceCMYK`にそれぞれ対応する．引数の各数値は `0.` で最小，`1.` で最大を表し，例えば `Gray(0.)` は黒，`Gray(1.)` は白，`RGB(1., 0., 0.)` は赤，といった具合である．この color 型は，第 9 章で紹介するグラフィックス関連の種々の記述で使われるほか，`color` パッケージではこの型を用いていくつかの頻出の色が定義されている．

3.2.11. 再帰函数

この節では**再帰函数** (*recursive function*) を扱う．再帰函数とは，簡単に言えば“定義中で

自分自身を呼び出すように記述されている函数”である．再帰函数の説明としてあまりにも月並みではあるが，例として整数の階乗を求める函数 `fact` を定義する（ただし，ここでは $0! := 1$ と定めるものとする）：

```
let-rec fact n =
  if n <= 0 then 1 else n * fact (n - 1)
```

`let` の代わりに `let-rec` から始める構文である．とりあえず使ってみて，結果が期待どおりであることを確認しよう：

```
+p{ Answer: \show-int(fact 6); }
```

Answer: 720

再帰函数の定義は，一般には以下のような形をしている：

`let-rec` $f\, x_1 \cdots x_n$ = $expr_1$ `in` $expr_2$

`let` の場合とは違い，$expr_1$ の中でも定義しようとしている f が使えるのが特徴である．ただし，この `let-rec` による定義では，糖衣構文を“はがした”上で最終的に f が束縛される式は函数の形の式 `(fun` x `->` $expr$`)` でなければならず，したがって `let-rec n = n + 1` のような定義は書けないようになっている．さて，もとの例である `fact` の定義を理解しよう．これはたしかに `fact` n で階乗 $n!$ を返すような定義になっており，慣れれば一目瞭然なのだが，“プログラムを手続き的に理解するという観念に拘束されている人”には少し飛躍があるかもしれない．再帰函数はいわゆる函数型プログラミングを最大に特徴づける重要な点とさえ言えるので是非とも理解されたい．再帰を理解する有用な直観は，いわゆる数学的帰納法によく似ている．すなわち，“より小さいものに対する解決がより大きいものの解決に使える”のである．上記の `fact` の定義は，次のような直観に基づいて理解できる：

- 与えられた引数が `0`（またはそれ未満）の場合は，返り値として即座に `1` を返してよい．
- 与えられた引数 n が `1` 以上の場合，返すべき値は n と n `- 1` の階乗との積である．したがって，n `- 1` の階乗を（`fact` 自分自身を呼び出すことで）計算し，その計算結果に n を掛けて返り値とすればよい．

或る意味では，階乗函数の（数学的な）定義：

$$
\begin{aligned}
0! &:= 1 \\
n! &:= n \times (n-1)! \quad (n \geq 1)
\end{aligned}
$$

に近い記述ができているとも言える．ともかく，「再帰は再帰がわかるとわかる」のだ．

さて，ここで少し**多相型** (*polymorphic type*) の話をしよう．まず以下のような函数を見てほしい：

```
let proj1 tuple =
  let (x, _) = tuple in x
```

この函数につく型はなんだろうか？　ひとまず，例えば int ∗ string → int などはつくはずであるが，よくよく見直すと $\tau_1 * \tau_2 \to \tau_1$ という形の型は全部つくということがわかる．これを反映して，let 多相では上記の `proj1` に

$$\forall\alpha.\ \forall\beta.\ \alpha * \beta \to \alpha$$

という型をつける．ここでの α や β は**型変数** (*type variable*) と呼ばれ，実際に `proj1` が使われるたびに（“整合する” 限りに於いて）適切な型をあてはめて使われる．例えば上記の `proj1` の定義以降で `(proj1(1cm,2cm),proj1(`foo`,1))` などと書いても（ふたつの `proj1` は異なる型の引数を受け取っているにもかかわらず）問題なく型がつく．多相型をつける一般的な規則を述べると長くなるため事細かには述べないが，基本的には let で束縛される変数の型に関して型推論の結果特に型が決定しない部分は型変数として一般化され，多相型がつくという仕組みになっている．

多相型について把握してもらったところで，代数的データ型で一旦無視していた型引数の説明をしよう．型引数は，多相な代数的データ型を定義するための機構である．例として，最初から備わっている代数的データ型のひとつである α option 型について見てみる．この型は，以下のように定義できるものだ：

```
type 'a option = None | Some of 'a
```

`'a` というひとつの型引数をとる定義である．型引数は，その代数的データ型の定義中で型として使うことができる．α option 型は，直観的には “α 型の値ひとつか，或いは何もないか” を表現する型である．もう少し具体的な用途を挙げて直観を説明するなら，“成功した時は α 型の値を返したいが，失敗するかもしれない函数” の返り値の型として使ったりする．例えば “実行時に失敗しない安全な除算” は以下のように定式化できる：

```
let div-safe m n =
  if n == 0 then None else Some(m / n)
```

この `div-safe` は int → int → int option 型をもつ．

`div-safe` は型引数に具体的に当てはまる型が int 型に決定していたが，重要なのはこれを多相なままにできるということである．

```
let from-option d opt =
  match opt with None -> d | Some(v) -> v
```

`from-option` は，第 2 引数に与えられた値が `None` のときは“デフォルト値”として第 1 引数 d を返し，`Some(`v`)` の形のときはその v を取り出して返すことで τ option 型のデータを τ 型へと“潰す”函数である．これは τ という型によらず同様であり，実際この `from-option` には $\forall\alpha.\ \alpha \to \alpha$ option $\to \alpha$ という多相型がつく．

さらに代数的データ型の話を続けよう．実は，代数的データ型の重要な特徴として，再帰的なデータを扱う型を定義できるというものがある．例として，二分木を扱う代数的データ型は以下のように定義できる：

```
type 'a bin-tree =
  | Empty | Node of 'a * 'a bin-tree * 'a bin-tree
```

α bin-tree 型の定義の中に α bin-tree 型自身が現れている．この型に基づく二分木を文字列に変換する函数 `show-tree` は以下のように定義できる：

```
let-rec show-tree pr tr =
  match tr with
  | Empty -> `_`
  | Node(x, trL, trR) ->
      `(` ^ (pr x) ^ #` `# ^
        (show-tree pr trL) ^ #` `# ^ (show-tree pr trR) ^ `)`
```

この `show-tree` につく型は $\forall\alpha.\ (\alpha \to \text{string}) \to \alpha$ bin-tree $\to$ string である．引数 `pr` も函数であることに注目されたい．`show-tree` は函数を受け取ってそれを使う函数になっており，このような函数を**高階函数** (*higher-order function*) と呼ぶ．型も，それに対応して $\alpha \to$ string が引数側にきている．

最も汎用性の高い再帰的なデータ型は，おそらくリスト (*list*) だろう．リストとは `[3; 1; 4; 1; 5; 9; 2]` といったように基本的に“同じ種類データが有限個並んだもの”であり，τ 型のデータが並んだリストを τ list 型がつくものとして扱うのだが，リストを構築・分解する方法の直観とは少し飛躍がある：　リストの構築方法は

- 空リスト `[]`
- 要素 x とリスト xs の結合 x `::` xs

の 2 種類があり，`[3; 1; 4; 1; 5; 9; 2]` は `3 :: (1 :: (4 :: (1 :: (5 :: (9 :: (2 :: []))))))` の糖衣構文なのである[*34]．要するに，リストとは（模擬的には）以下で定義される代数的データ型に相当する：

```
type 'a list = [] | (::) of 'a * 'a list
```

リストに対する簡単な処理をいくつか見てみる．まずリストを受け取ってその長さを返す函数 `list-length` は以下のとおり：

```
let-rec list-length l =
  match l with
  | []      -> 0
  | x :: xs -> 1 + list-length xs
```

ふたつのリストを結合する函数 `list-append` は以下のとおり：

```
let-rec list-append l1 l2 =
  match l1 with
  | []      -> l2
  | x :: xs -> x :: list-append xs l2
```

リストを前後反転する函数 `list-reverse` は，前 2 つの例に比べて少し技巧的だが以下のように実装できる：

```
let list-reverse l =
  let-rec aux acc l =
    match l with
```

34 `::` は右結合なので，ここでつけている括弧は全て省略できる．

```
    | []      -> acc
    | x :: xs -> aux (x :: acc) xs
  in
  aux [] l
```

かなり駆け足の説明のため基礎的な函数のみの紹介となってしまったが，リストに対する操作，例えばマップ函数や畳み込み函数は，内容の充実した函数型プログラミングの入門書では必ず触れるものなので，是非参照されたい．なお，リストに対する操作は `list` パッケージでいくつも定義されており，ここで例示した `list-length`，`list-append`，`list-reverse` も `List.length`，`List.append`，`List.reverse` としてそれぞれ実装されている．

第 2 章で登場した itemize 型も，実はあらかじめ定義されている以下のような再帰的な代数的データ型である：

```
type itemize = Item of inline-text * itemize list
```

そして，`\listing` などのコマンドに渡される

```
{ * 東京
    ** 本郷
    ** 弥生
    ** 浅野
    ** 駒場
  * 千葉
    ** 柏}
```

という記述は，以下のような構造の糖衣構文である：

```
(Item(` `, [
  Item({東京}, [
    Item({本郷}, []);
    Item({弥生}, []);
    Item({浅野}, []);
    Item({駒場}, []); ]);
  Item({千葉}, [
    Item({柏}, []); ]);
]))
```

`\listing` などのコマンドは，この再帰的なデータ構造の上を走査して組版処理しているのである．

3.2.12. レコード

レコード (*record*) は，タプルと同様に“複数の式をまとめて扱う機構”なのだが，タプルがデータを“何番目のデータか”に基づいて構築・分解していたのに対し，レコードではそれぞれのデータに**ラベル** (*label*) がつけられ，これに基づいて構築したり取り出したりする．おおよそ，C 言語でいうところの構造体にあたる概念である．例として，人物の情報を扱うレコードをまず掲げよう：

```
(| given-name = `Takashi`; family-name = `Suwa`; age = 25; |)
```

ここでの `given-name`，`family-name`，`age` がラベルである．ラベルに使える文字列は変数名に使えるそれと同じ範囲である．レコード *r* からの内容の取り出しは *r*#*label* という式によって行なう．例えば上記のレコードが `r-tsuwa` という変数に格納されているとき，`r-tsuwa#age` は `25` へと評価される．実は，レコードは SATYSF$_{\mathrm{I}}$ で文書を書くユーザならほぼ全員使っているはずである．というのも，大抵のクラスファイルでは `document` の直後に文書に関する情報をレコードで渡すようなインターフェイスになっているからだ．そう，以下の `(| … |)` の部分はレコードだったのである．

```
@require: stdjareport

document (|
  title = {はじめての文書};
  author = {組 版太郎};
|) '<
（以降略）
```

レコードのラベルという仕組みはタプルに比べて各要素がどのような役割を果たしているかが明瞭であり，プログラムのドキュメント性を向上させる効能があるが，型つけに関してさらに顕著な違いがある．SATYSF$_{\mathrm{I}}$ でのレコードに関する型つけ規則は，SML# を基礎づける型システム [23] を大々的に参考にしている．この型システムについて定義をすべて述べるのはあまりにも長いので，簡素に直観だけを与える．詳細は元論文 [23] を参照されたい．まず，基本的にレコード：

(| $label_1$ = $expr_1$; ⋯; $label_n$ = $expr_n$; |)

は，各式 $expr_i$ に型 τ_i がつく下で

$$(\!| label_1 : \tau_1;\ \cdots;\ label_n : \tau_n |\!)$$

という型をつけて扱う．レコードそのものの型つけは単純なのだが，重要なのはラベルへのアクセスに関する型つけである．レコードを受け取り，そのレコードのラベル `foo` にアクセスする函数を考える：

```
let f r = r#foo + 1
```

この函数 `f` にどんな型をつければよいかをまず素朴に考えよう．この函数の式が型の制約として要請するのは，`r` という変数が束縛されているレコードが `foo` というラベルをもち，かつそれに対応する値が int 型であるということだけである．この函数を SATYSFI では（SML# に倣って）

$$\forall \alpha :: (\!| \mathrm{foo} : \mathrm{int} |\!).\ \alpha \to \mathrm{int}$$

という型をつけて扱う．これは α が一般化されている或る種の多相型なのだが，重要なのは「$:: (\!| \mathrm{foo} : \mathrm{int} |\!)$」の部分であり，これは α が（任意の型ではなく）ラベル foo に int 型をもつレコードの型のみを動く型変数であるということを記述している．同様にして

```
let g r =
  let (x, _) = r#bar in r#foo + (round x)
```

で定義される `g` には以下のような型がつく：

$$\forall \beta.\ \forall \alpha :: (\!| \mathrm{foo} : \mathrm{int};\ \mathrm{bar} : \mathrm{float} * \beta |\!).\ \alpha \to \mathrm{int}$$

一般的な説明は煩雑なので省くが，おおよそこのような仕組みでレコードに関して多相型がつけられる．

3.2.13. 手続き

いわゆる函数型言語ではあるものの，SATYSFI にも手続き的処理を行なう機構は備わっている．すなわち，書き換え可能な変数に相当する**可変参照** (*mutable reference*) である．可変参照は，次のような構文で宣言する：

```
let-mutable x <- expr1 in expr2
```

`x` が宣言される可変参照であり，$expr_1$ がその初期値，$expr_2$ が `x` の使えるスコープである．可変参照は基本的には通常の変数と同様の扱いであるものの，τ 型の値が格納される可変参照は τ ref 型がつけられる．可変参照 `x` の現在の内容は `!x` で取得することができ，また内容を破壊的に書き換える処理は，新たに格納させる内容に評価される式を *expr* として `x <- expr` と書く．この式が評価されて `x` の内容が上書きされると，式全体はユニット値 `()` となる．

可変参照は，章節番号の挿入など，“本質的に手続き的な処理”をクラスファイル内で扱うのに利用されているので，各種クラスファイルの中を覗くと多くの用例が確認できるだろう．ただし，多くの場合可変参照を多用するとプログラムの挙動を把握困難にする要因となる[*35]ので，止むを得ない場合を除いてできる限り使わないように心がけた方が良さそうである．

3.2.14. 算術に関するプリミティヴ

主な“算術演算”用のプリミティヴについてまとめて紹介する：

`+`, `-`, `*` : int → int → int
整数の加算，減算，乗算．

`/`, `mod` : int → int → int
整数の除算と剰余．`mod` は通常の変数名のように見えるが，例外的に 2 項演算子である．いずれも第 2 引数に `0` が与えられた場合は実行時エラーとなる．

`==`, `<>` : int → int → bool
整数の等号と不等号．

`>`, `<`, `>=`, `<=` : int → int → bool
“大なり”，“小なり”，“大なりイコール”，“小なりイコール”．

`&&`, `||` : bool → bool → bool
連言と選言．

35 だからこそいわゆる函数型プログラミングの流儀は破壊的代入を積極的に避けるのである．

`not` : bool → bool

否定.

`float` : int → float

整数を浮動小数点数に変換する.

`round` : float → int

浮動小数点数を，小数部分を切り捨てて整数に変換する．引数が正負無限や NaN，あるいは小数部分を切り捨てた整数が int 型で表現不可能なときの結果は未定義.

`+.`, `-.`, `*.`, `/.` : float → float → int

浮動小数点数の加算，減算，乗算，除算.

`sin`, `cos`, `tan`, `asin`, `acos`, `atan` : float → float

正弦，余弦，正接函数，およびアークサイン，アークコサイン，アークタンジェント（すなわち正弦，余弦，正接函数の “逆函数”）.

`atan2` : float → float → float

`atan2` y x で y `/.` x のアークタンジェントを返すが，y と x の符号が象限の決定に用いられる．返される値は $-\pi$ から π の間の値である.

`+'`, `-'` : length → length → length

長さの加算と減算.

`*'` : length → float → length

長さのスカラー倍.

`/'` : length → length → float

長さの “除算”.

`>'`, `<'` : length → length → bool

長さの“大なり”と“小なり”.

`^` : string → string → string
文字列の結合.

`arabic` : int → string
整数を十進文字列に変換する.

`show-float` : float → string
浮動小数点数を十進文字列に変換する.

`string-same` : string → string → bool
文字列の等価性の判定． バイト列として等価かどうかの判定であり，Unicode によって規定された正規化等は行なわない.

`string-sub` : string → int → int → string
`string-sub` s i l で文字列 s の第 i 番目の文字を先頭とする長さ l の部分文字列を取り出す． 文字数は Unicode コードポイント単位. i と l による指定が部分文字列の範囲でない場合は実行時エラーが出る.

`string-length` : string → int
与えられた文字列が Unicode コードポイント単位で何文字かを返す.

`string-unexplode` : int list → string
与えられた整数列の各整数を Unicode コードポイントと解釈した文字列を返す. Unicode コードポイントの範囲にない整数が与えられた場合の動作は未定義.

`embed-string` : string → inline-text
文字列をインラインテキストに変換する.

`split-into-lines` : string → (int ∗ string) list
与えられた文字列を各行に分解してリストを返す． 返されるリストの各要素 `(`i`,`

`s)` は“インデントが `i` 個挿入された内容文字列 `s` の行”に相当する．

`!` : $\forall\alpha.\ \alpha\ \mathrm{ref} \to \alpha$

`!r` で可変参照 `r` の参照する現在の値を返す．

`::` : $\forall\alpha.\ \alpha \to \alpha\ \mathrm{list} \to \alpha\ \mathrm{list}$

リストの“コンス”，すなわち第 1 引数を第 2 引数のリストの先頭につけてできるリストを返す．

`|>` : $\forall\alpha.\ \forall\beta.\ \alpha \to (\alpha \to \beta) \to \beta$

前後逆の函数適用，すなわち `x |> f` で `f x` に同じ．ただし，式 `x` と `f` の評価にともに“副作用”がある場合は評価順序の逆転に注意を要する．

`display-message` : string → unit

与えられた文字列を 1 行として標準出力に表示する．

3.2.15. OCaml との顕著な差異

すでに述べたようにプログラミング言語としての SATYSF$_{\mathrm{I}}$ は OCaml にかなり近いものとなっているが，非互換な違いがいくつか顕著にある．

変数名　おそらくコードの見た目上これが最も目につく違いである．変数名はラテン小文字から始まり，2 文字目以降は大小ラテン文字，数字，ハイフンのいずれかでなければならない．アンダースコア（`_`）や引用符（`'`）は使えないことに注意．

再帰函数の定義　再帰函数の定義には，OCaml の `let rec` の代わりに `let-rec` を用いる．これで 1 トークンである．

型コンストラクタの構文　SATYSF$_{\mathrm{I}}$ では，型コンストラクタ *tyconstr* の“適用”は ty_1 ⋯ ty_n *tyconstr* という形で書く．これは OCaml の（ty_1，…，ty_n）*tyconstr* および $n = 1$ の場合の ty_1 *tyconstr* という形の構文とは明確に異なる．そのかわり，OCaml での `int list list` のような括弧を省いた“適用”を書くことはできず，必ず `(int list) list` と括弧をつけることになる．

コンストラクタ　SATYSF$_{\mathrm{I}}$ では，ヴァリアント型のコンストラクタはすべて 1 引数である．`None` のように引数のないコンストラクタの使用は `None()` のようにユニット値が省略されているものとして扱われているほか，OCaml では

```
type foo = Foo of int * int
```

と

```
type foo = Foo of (int * int)
```

とで定義される型が異なるが，SATYSF$_{\mathrm{I}}$ では完全に一致する．

レコード　まずレコード式を囲む括弧が `{ … }` の代わりに `(| … |)` である[*36]．また，レコード型は nominal ではなく structural であり，したがってレコード式 `(| … |)` はそれにつく型を `type …` で宣言せずとも書くことができる．これは SATYSF$_{\mathrm{I}}$ の型システムがレコードに関して SML# と同様のもの [23] を採用しているからである（詳細は元論文を参照せよ）．これに加え，OCaml のように特定のフィールドを mutable にするような機能はなく，必ず全てのフィールドが immutable である．

可変参照　可変参照は OCaml のように多相函数 `ref` で作るのではなく，

```
let-mutable x <- expr1 in expr2
```

という構文で作るように限定されている．これは OCaml の

```
let x = ref expr1 in expr2
```

と同等である．破壊的代入も，OCaml のように `:=` ではなく，`x <- expr` と書く．`<-` が多相な 2 項演算子なのではなく，`x <- expr` 全体が式をなすひとつの構文である．

36　これは `{ … }` の記法をインラインテキストのために明け渡したからである．

Chapter 4
文書が処理される仕組み

SATYSF$_{\mathrm{I}}$ のおおよその用法は既に第 2 章で説明したが，もしかすると他のマークアップ言語ないし組版処理システムではあまり意識することのない“インライン”なる概念と“ブロック”なる概念の区別を不思議に感じたかもしれない．この区別は SATYSF$_{\mathrm{I}}$ がどのように組版処理の工程を定式化して扱っているかに起因しており，本章ではその定式化を概観することにする．SATYSF$_{\mathrm{I}}$ での組版処理工程の定式化は，各工程に対応するプリミティヴにどのように型をつけて静的な検査を実現しているかという点とも密接に関わってくるので，是非第 5 章と併せて読んでいただきたい．

模式的に描くと，SATYSF$_{\mathrm{I}}$ の全体像は図 4.1 のようになっている． SATYSF$_{\mathrm{I}}$ 処理系の構造は左右に大別されて描かれているが，このうち入力に近い左側を**フロントエンド** (*front-end*)，右側を**バックエンド** (*back-end*) と呼ぶ．

4.1. フロントエンド

フロントエンドは入力側から順に**字句解析器** (*lexer*)，**構文解析器** (*parser*)，**型検査器** (*type checker*) からなっている．ソースファイルが SATYSF$_{\mathrm{I}}$ へ渡されると，まず字句解析器にしたがってソースファイルの内容がトークン列に切り分けられる．字句解析器には

- プログラムを読み取る状態
- インラインテキストを読み取る状態
- ブロックテキストを読み取る状態
- コメントを読み取る状態
- ⋮

といったいくつかの状態があり，ソースファイルを読み取る過程でスタックを用いて括弧の対応などを検査しつつこれらの状態を適宜遷移する．明らかに括弧の対応が取れていな

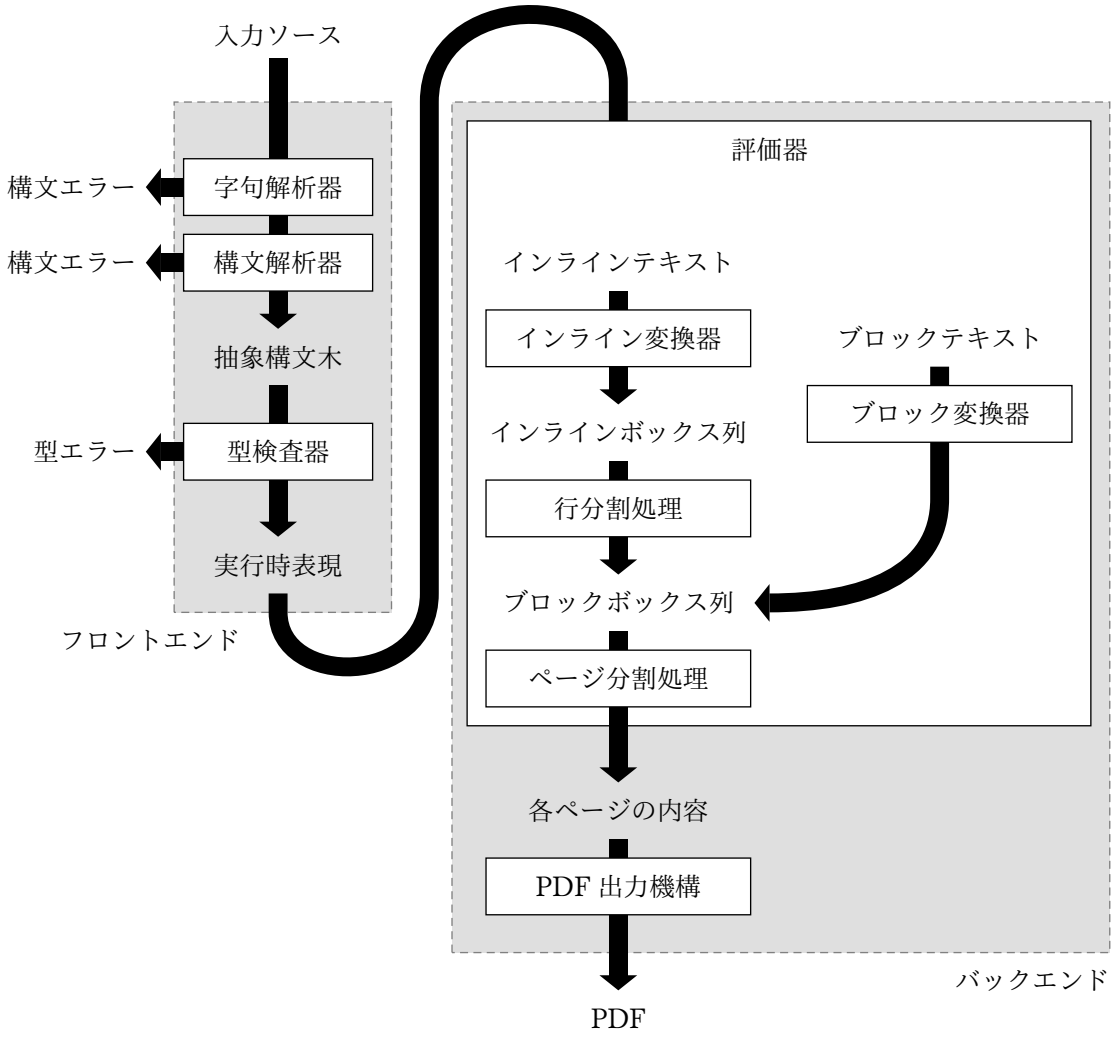


図 4.1 SATYSFI の全体像

かったり，現在の状態に対して正当なソースファイルで出現するはずのない文字に遭遇したりすると，その時点で

```
! [Syntax error at Lexer]（エラー位置とその内容）
```

という形でエラーが報告される．続いてトークン列化されたデータは構文解析器によって（SATYSFI の文法にしたがって）抽象構文木に組み上げられる．この時点で構文木に組み上

げられないトークン列の組み合わせが見つかった場合は

```
! [Syntax error at Parser]（エラー位置）
```

という形でエラーが報告される．字句解析器でエラーが出るか構文解析器でエラーが出るかには，ユーザにとってはそれほど大きな違いはない．いずれも何らかのミスを含んでいるために抽象構文木に組み上げられないソースファイルが処理系に渡され，型検査器にかけるまでもなく入力がおかしいとわかったため弾かれたということである．ここまでの処理は，ほとんどのマークアップ言語やプログラミング言語に共通の処理である[*37]．字句解析でも構文解析でもエラーが出ず抽象構文木に組み上げられた入力は，次に型検査器に通されることになる．型検査器は SATYSFI のフロントエンドに於いて根幹的役割を果たす重要な部分であり，抽象構文木を受け取って**型検査** (*type checking*)[*38]を行ないつつ，同時にその抽象構文木を**実行時表現**へと変換する[*39]．型検査とは，引数の種類や個数が合っていないとかリストと真偽値を足し合わせる計算を記述しているといった“或る種の不整合”が入力に含まれていないかを検査することを指す．ここでの検査に通れば実際に組版処理を行なっている最中に“或る種の不整合”に基づく処理の失敗が起こらないことが（数学的に）保証されており，それゆえに実際に組版処理を行なわずとも入力ミスを素早く検出できるである．何らかの不整合が含まれている場合，型検査器はその不整合を

```
! [Type Error]（エラー位置とその内容）
```

37 字句解析と構文解析でのエラーを区別せず“syntax error”として提示する言語も多い．ちなみに TEX 処理系は構文解析に相当する処理は行なわないほか，字句解析に相当する処理すら組版処理と同時進行で行なう，言語処理系としてはかなり珍しい方式である．また，ブラウザなどマークアップ言語を扱う処理系が構文に対するエラーを明示的に提示しているのを見た経験は少ないかもしれないが，これは大抵の場合処理系がミスに対してかなり“寛容”につくられていて，なんとかミスを整合させようと何らかの強引な修正を行なうからだ．ちなみに，処理系がミスに“寛容”なのは別に望ましいことではなく，むしろ逆である．それは税関が“寛容”だと密輸が横行するのと同様の意味に於いてであり，実際この世界にはブラウザの“寛容さ”にかまけた杜撰なソースの Web ページに溢れている．

38 実際には型検査というよりは**型推論** (*type inference*) と呼ばれる処理を行なっている．

39 実行時表現は，単に型情報などが落ちて糖衣構文などが除去された，実行用に少しだけ形式が整えられた抽象構文木と考えてもらって差し支えないが，実際の実装ではバイトコードなどのより低級な表現になっていることもありうる．

という形で報告する．ユーザはこれにしたがってすみやかに修正することができる．

4.2. バックエンド

さて，フロントエンドでの各種処理と検査をめでたく通過した入力データは，バックエンドによって実際に組版処理にかけられる．ここでの処理は各コマンドの定義の記述に基づいて各工程を行ったり来たりするので実際には複雑に依存し合っているのだが，概して図 4.1 の右側のような処理を経る．バックエンドでの処理の流れを以下順を追って見ていくことにしよう．

4.2.1. インラインボックス列への変換

まず型検査を通過して抽象構文木から変換して得られた実行時表現は**評価器** (*interpreter*) に渡され，ここでの評価の過程でインラインテキストになった部分は（各コマンドの定義に基づいて）**インライン変換器** (*inline converter*) にかけられる．変換器では，どのフォントをどんな文字サイズ，どんな文字色で使うかといった“組版処理に必要なテキスト以外の情報”を用いてインラインテキストから**インラインボックス列** (*inline box row*) をつくる．“組版処理に必要なテキスト以外の情報”がどのように指定されるのかについては5.1 節の**テキスト処理文脈** (*text-processing context*) についての解説で後述するので，今は「コマンド定義の記述とそのコマンドが使われている環境によってそれらの情報が決まる」という程度の認識で構わない．インラインボックス列は，直観的には“既に組まれたテキスト”に相当し，紙面上でどの程度の幅・高さ・深さを占めるかなどのデータも保持している．インラインテキストからインラインボックス列への変換は，図 4.2 のように表すことができる．「ボックス列」という命名は，この図のような直観から来ている．

4.2.2. 行分割処理

さて，テキストからインラインボックス列を生成してしまえばもう組版処理を達成したも同然だろうか？　勿論そんなことはない．古今東西を問わず，長い文章が書かれる際に 1 行がひたすら切れずに細長く連なっていることはまずなく，装飾性が実用性に圧倒的に優先されるような媒体ではない限り，基本的には適切に行として切り分けた単位で文字が排列されている．すなわち，今度の相手は紙面の幅である．文字の排列を適切に行単位に切り分けて紙面の所望の幅に収める処理を**行分割処理** (*line breaking*) と呼ぶ．行分割処理は，段落の幅や行送りの長さといった各種パラメータを用いて，受け取ったインラインボックス列を適切に切り分けて**ブロックボックス列** (*block box row*) として段落を生成する処理である．ブロックボックス列とは，段落が連なる方向に並んだ（おおよそ行単位の）ボックス列である．行

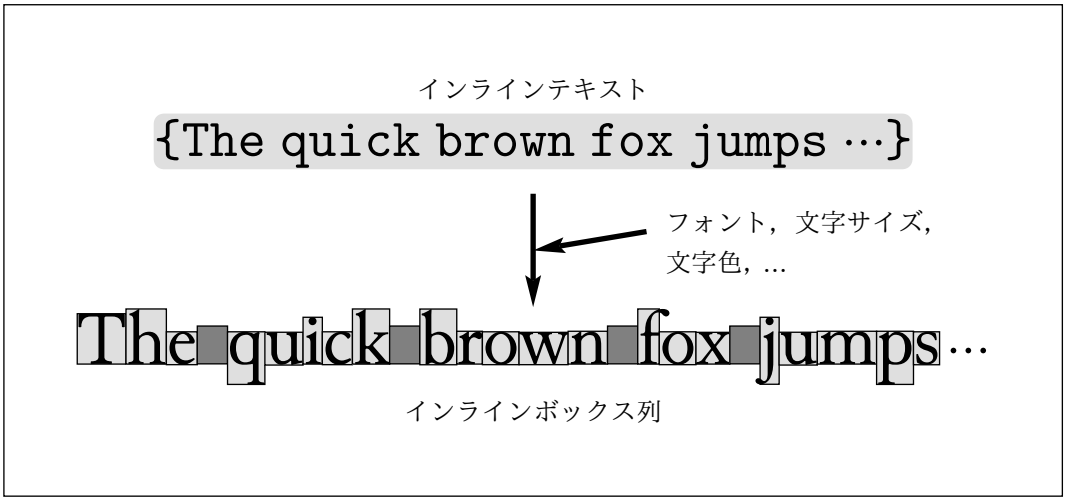


図 4.2 インラインボックス列への変換

分割処理がインラインボックス列を受け取ってブロックボックス列を返す処理として定式化されていることは，図 4.3 のような直観による．図の下側の 1 行にあたる横に細長い枠がいわば“ひとつのボックス”であり，それが段落の進む方向に積み重なっているので行分割された後の内容はブロックボックス列なのである．

行方向に連なった内容をどの箇所で段落の各行に切り分けるかを決めるのはそれほど自明な処理ではない．例えば欧文なら単語中で切り分けることは（ハイフネーションしてよい箇所でない限り）できないし，和文なら鉤括弧の直後や句点の直前といった箇所での行分割を禁止する**行頭禁則・行末禁則**という制約がある．両端揃えで段落を組む場合は欧文の単語間の空白や和文の字間を（あらかじめ定めた許容範囲内で）適切に伸ばしたり縮めたりして調整する必要があり，図 4.2 でいえば濃い灰色で示された部分のボックスが横方向に伸縮することになる．そして何より“詰められるだけ前から順に詰めていく”という処理では一般に見栄えのよい段落になるとは限らない．行分割処理がどのように行なわれているかの詳細については第 6 章で述べる．

4.2.3. ページ分割処理

文書の内容がブロックボックス列になっても，PDF として出力するにはまだ処理すべきことが残っている．紙面の横幅を相手にした後は，紙面の縦幅も相手にすることになるので

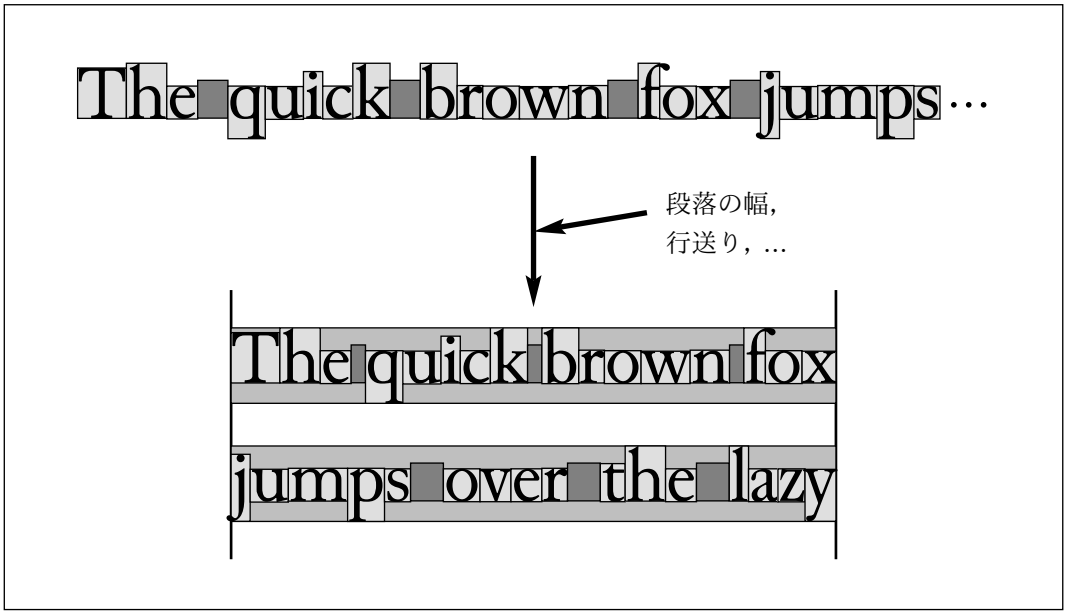


図 4.3 行分割処理の直観

ある[*40]．すなわち，今度は縦方向に連なった段落を適切にページ単位に切り分ける**ページ分割処理** (*page breaking*) が必要になる．ページ分割は行分割と比べると視認上の要請がゆるやかなこともあって切り分け箇所の決定自体は比較的簡素な処理で済む．実際，SATYSFI のページ分割処理がやっていることは本質的には単に“前から詰められるだけ詰めて切り分ける”処理である．それでも章題の直後などページが切り分けられるのを避けねばならない箇所はやはりあるので，“各箇所でのページの分割しやすさ”を制御するための仕組みが備わっている．このほか，脚註の挿入などもページ分割時に処理される．ページ分割の詳細についても，章を分けて第 12 章で解説する．

40 現在の SATYSFI は縦書きには対応していないが，縦書きの場合は（横書きと同様の行分割処理とページ分割処理を蹈襲すれば）横幅と縦幅を相手にする順番が逆になる．要するに，横書き・縦書きいずれの場合も文字が進む方向に関する処理が先になり，次いで段落が進む方向の処理が行なわれるのである．

Chapter 5
コマンド定義：発展編

第 3 章で基礎的なコマンド定義の方法およびプログラミング言語としての SATYSFI について習得し，第 4 章で SATYSFI の処理系の全体像，および各工程がどのように定式化されているかについて概観した．これらの前提知識を把握した下で，この章ではようやくフォントや文字色，行送りといった，“組版上の装飾的な視覚情報” にも関わる処理を行なうコマンドの実装方法を習得する．

5.1. テキスト処理文脈

第 4 章でも既に述べたが，組版処理はインラインテキスト・ブロックテキストといった “テキストの情報” だけが与えられてもできない．すなわち，どのフォントのどんなサイズで組むか[*41]，段落の幅や行送りはどの程度の長さにするか，といった情報が “テキストの情報” に加えて必要である．これらの “テキストには表れていないが組版処理上必要な情報” を，SATYSFI は**テキスト処理文脈** (*text-processing context*) という機構に保持させて扱う．抽象的な説明を続けてもおそらく要領を得ないであろうから，具体例を用いて説明しよう．例えば，文字サイズを 1.5 倍に変更するコマンド `\enlarge` を考える：

```
+p{内側の \enlarge{文字が \enlarge{1.5 倍} の大きさに} なります．}
```

内側の文字が 1.5 倍の大きさになります．

このような `\enlarge` は次のように定義できる：

41 活版印刷の慣習としては，“font” という語は各字形が同一のデザインに則っているだけでなくサイズも相等しいひと揃いの活字のセットを指したようだが，SATYSFI でフォントと言う場合は（字形やそれに関するメトリックのひとかたまりに紐づけられた）フォント名，拡大縮小比率，ベースライン調整率の 3 つ組を指す．詳しくは第 8 章で述べる．

```
let-inline ctx \enlarge it =
  let size = get-font-size ctx in
  let ctx-enlarge = set-font-size (size *' 1.5) ctx in
    read-inline ctx-enlarge it
```

唐突に全部見せられてもあまり実感が湧かないであろうから少しずつ説明するが，まず 1 行目から見慣れない文法である．`let-inline` を用いてインラインコマンドを定義する構文は第 3 章で既に紹介したが，ここでは `let-inline` と定義されるコマンド名 `\enlarge` との間に `ctx` なる何かがある．実はこれこそがテキスト処理文脈を受け取る“第 0 引数”の変数だ．コマンドの適用が評価されるとき，ユーザが与えた通常の引数に加えて，コマンドが使われている箇所のすぐ外側のテキスト処理文脈がこの変数 `ctx` に渡され，コマンドの実装中で使える，というわけである．ここまで `stdjareport` など何らかのクラスファイルが提供するマークアップ等の方式に従って SATYSFI を使うにあたってはテキスト処理文脈を全く意識する必要がなかったであろうけれども，実のところユーザからは見えない裏側で密かに各コマンドがテキスト処理文脈を受け渡ししていたのである．なお，`ctx` は単に変数名なので，第 0 引数に `x` とか `y` とか `i-love-satysfi` といった違う名前を使っても構わない[*42]．

さて，続いてコマンド定義の“中身”を見てみる．第 0 引数の `ctx` と（各適用でユーザが与える）第 1 引数 `it` を使うのだが，まず `size` に外から渡されてきたテキスト処理文脈 `ctx` が保持しているフォントサイズを取り出して束縛する．これはプリミティヴ：

`get-font-size` : context → length

を用いて行なう．取得したフォントサイズ `size` は length 型であり，`*'` を用いて `size *' 1.5` で 1.5 倍の長さを得る．フォントサイズの設定は

`set-font-size` : length → context → context

を用いて行なう．すなわち，古いテキスト処理文脈 `ctx` と設定するフォントサイズ `size` に対して `set-font-size size ctx` でフォントサイズの更新された新しいテキスト処理文脈が返ってくる．`\enlarge` の実装では，この新しいテキスト処理文脈を変数 `ctx-`

42 ただし，既存のパッケージでは慣習的に第 0 引数を必ず `ctx` という変数名にしているので，複数人で開発したり，既存パッケージを改造したりする場合は `ctx` という変数名で統一した方が理解に齟齬を生じにくいかもしれない．

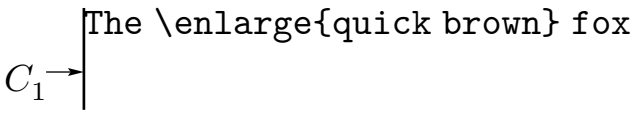
`enlarge` に束縛している．なお，ここの一連の処理で `ctx` の内容が“書き換えられる”わけではないことに注意されたい．単に `ctx` の内容のうちフォントサイズに関するデータだけが `size` に変更された新しいテキスト処理文脈がつくられて `ctx-enlarge` という変数に束縛されているだけである．実際，`ctx-enlarge` が束縛されたスコープでも依然として `ctx` を使うことができ，かつそのテキスト処理文脈には元々のフォントサイズが保持されている．

最も重要なのは最終行だ．ここで使われているプリミティヴ

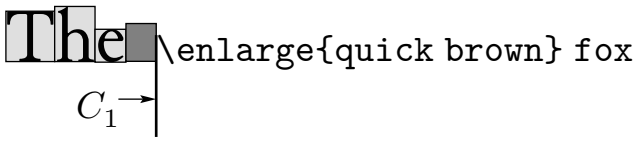
`read-inline` : context → inline-text → inline-boxes

は第 4 章で説明したインライン変換器に相当し，図 4.2 に示されたような処理をしている．すなわち，`read-inline` *ctx it* でテキスト処理文脈 *ctx* の保持するフォントや文字サイズなどの情報をもとにインラインテキスト *it* をインラインボックス列へと変換する．そしてこれがきわめて重要な事項なのだが，インラインテキスト *it* に含まれているインラインコマンドの適用が評価されるのも，この `read-inline` によるインラインボックス列への変換の最中である．このとき，`read-inline` に第 1 引数として渡したテキスト処理文脈が，第 2 引数の *it* 中の各コマンド適用を評価するときに第 0 引数として渡されるのである．興味のある読者は，これについて第 14 章を眺めるとよいかもしれない．

さて，実際にコマンドが処理される様子を“コマ送りで”眺めてみる．まずは（コマンドを含む）普通のテキストがある．これに対して `read-inline` がテキスト処理文脈 C_1 を用いてインライン変換を始めた状況を考えよう：



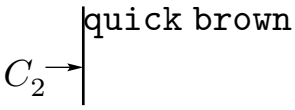
縦棒は“現在インライン変換器が読んでいるところ”である．現在の位置にあるのが“単なるテキストとしての文字”であるとき，インライン変換器はそのとき読み進めるのに使っているテキスト処理文脈が保持しているフォントや文字サイズ，文字色といった情報を使って“テキストとしての文字”を“アウトライン化された，組める状態になったグリフとしての文字”に変換する：



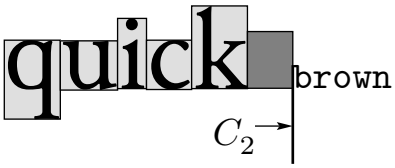
コマンドに出会うと，そのコマンドの定義を“思い返す”：

```
let-inline ctx \enlarge it =
  let size = get-font-size ctx in
  let ctx-enlarge = set-font-size (size *' 1.5) ctx in
    read-inline ctx-enlarge it
```

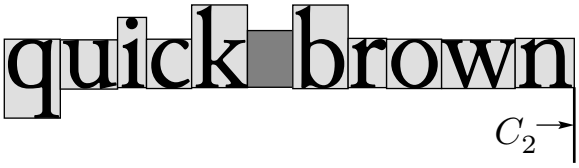
そしてこのコマンド定義に基づいて第 0 引数に現在のテキスト処理文脈（ここでは C_1）を，第 1 引数以降もインラインテキスト中にユーザが与えたものを，それぞれ渡して評価する．`set-font-size` によって文字サイズが更新された C_1 を C_2 と書くことにすると，続いて `\enlarge` の定義中にある `read-inline` が再びインライン変換器をテキスト処理文脈を C_2 にして起動する：



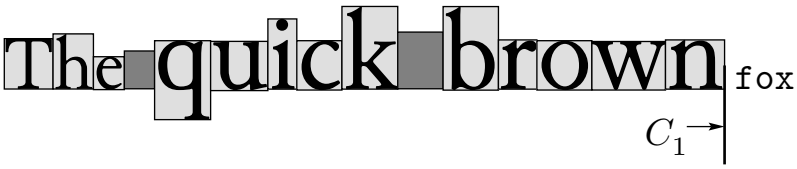
そして C_2 は文字サイズが大きくなっているので次のように変換され，



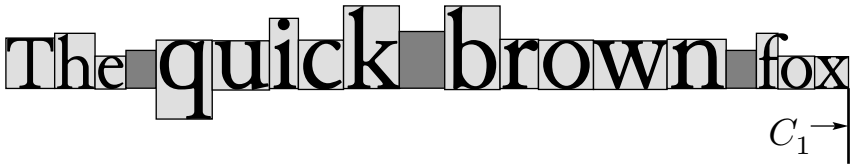
やがて末尾まで到達する：



末尾まで到達すると，`\enlarge` がそれを返り値として戻し，もともとコマンドの適用があった箇所が返り値で置き換えられる：



再びテキスト処理文脈 C_1 を用いたインライン変換に戻り，継続する：



こうして末尾まで到達し，インライン変換の処理が再帰的に行なわれた結果のブロックボックス列が完成するのである．

さて，コマンドにつく型の話をすると，基本的に

`let-inline` *ctx cmd* x_1 x_2 $\cdots$ x_n = *expr*

という構文でインラインコマンドが定義されるとき，第 0 引数 *ctx* は context 型がつくとされ，また各変数 x_i が τ_i 型をもつ下で *expr* は inline-boxes 型であることが要請される．こうして定義されたインラインコマンド *cmd* は

$$\left[\tau_1;\ \tau_2;\ \cdots;\ \tau_n\right]\ \text{inline-cmd}$$

という型をもつと定められている[*43]．この型は感覚的には

$$\text{context} \to \tau_1 \to \tau_2 \to \cdots \to \tau_n \to \text{inline-boxes}$$

という函数の型とほぼ同等なのだが，函数とは違って部分適用することが想定されていないことや，将来実装予定のテキストモードでは

$$\text{context} \to \tau_1 \to \tau_2 \to \cdots \to \tau_n \to \text{string}$$

という函数のように扱われることが期待され，PDF モードとテキストモードとで共通の型になっていることが望ましいという観点から，このように統一的な型をつけて扱うことにしている．

ちなみに，ここで説明の例に挙げた `get-font-size` および `set-font-size` とほぼ同様の感覚で使えるプリミティヴが SATYSFI にはいくつも備わっている．この本でそれらを順を追って説明していくのだが，`set-font-size` 程度の比較的簡単な部類のものには

43　この説明は正確ではない．詳しくは5.2 節および第 14 章を参照されたい．

以下がある：

`set-text-color` : color → context → context

ctx `|> set-text-color` *color* で現在の文字色を *color* に更新したテキスト処理文脈を返す．例えば *ctx* `|> set-text-color (RGB(0., 0., 0.5))` は文字を紺色で“描く”ように更新したテキスト処理文脈を返す．

`get-text-color` : context → color

テキスト処理文脈に保持されている文字色を返す．

`set-manual-rising` : length → context → context

ctx `|> set-manual-rising` *l* で“文字が描かれる位置”を本来よりも *l* だけ高い位置にする．

例 5.1　「SATYSFI」を打ち出すインラインコマンド `\SATySFi` : [ ] inline-cmd は `set-manual-rising` を用いて実装されている：

```
let no-break ib =
  let pads = (0pt, 0pt, 0pt, 0pt) in
    inline-frame-outer pads (fun _ _ _ _ -> []) ib

let-inline ctx \SATySFi =
  let size = get-font-size ctx in
  let f = read-inline ctx in
  let fd =
    read-inline
      (ctx |> set-manual-rising (0pt -' (size *' 0.25)))
  in
  let ib =
    f {SAT} ++ kern (size *' 0.15) ++ fd {Y}
      ++ f {SF} ++ kern (size *' 0.05) ++ fd {I}
  in
    script-guard Latin (no-break ib)
```

`inline-frame-outer` については第 9 章で触れるが，今は気にしなくてよく，単にここで定義された `no-break` : inline-boxes → inline-boxes が与えられたブ

ロックボックス列を途中で行分割不可能にする函数だと把握すればよい．`script-guard` についても第 8 章で触れるが，これはこのコマンドが使用された箇所の前後に挿入される和欧間スペースなどに関わるものである．

5.2. コマンド定義の糖衣構文

ところで，インラインコマンドを定義する構文：

```
let-inline ctx cmd x_1 ··· x_n = expr
```

は第 3 章で解説した let 式と同様に

```
let-inline ctx cmd = (fun x_1 ··· x_n -> expr)
```

の糖衣構文である．これはあまり意識する必要はないが，知っていると部分適用などを駆使してコマンド適用が簡潔に書けたりして便利である．より詳しく知りたい場合は第 14 章を参照されたい．さらに言うと，おそらくこちらは少し意外に感じられるかと思うが，第 3 章で登場したテキスト処理文脈を受け取らない `let-inline` によるコマンド定義の構文：

```
let-inline cmd x_1 ··· x_n = expr
```

は，実は

```
let-inline ctx cmd = (fun x_1 ··· x_n -> read-inline ctx expr)
```

の糖衣構文である．ただし，*ctx* は *expr* に自由出現しない変数名をひとつ選んだものである．要するに外から渡されるテキスト処理文脈をそのままインラインテキスト *expr* のインライン変換に使うだけ，ということである．こちらの第 0 引数を取らない構文は `=` より前に書いてある変数の個数で `read-inline` が“挿入”される場所が決定されるので，

```
let-inline cmd = (fun x_1 ··· x_n -> expr)
```

と書いても意図通りには脱糖衣されないことに注意されたい．

5.3. 行分割を行なうコマンド

行分割処理に相当するプリミティヴは

`line-break` : bool → bool → context → inline-boxes → block-boxes

である．これは `line-break` b_1 b_2 *ctx ib* の形で使われ，行分割された結果であるブロックボックス列を返してくれる．引数はそれぞれ以下を指示する[*44]：

- b_1 ： 組み上がった段落の直前でページ分割してよいか否か
- b_2 ： 組み上がった段落の直後でページ分割してよいか否か
- *ctx* ： 行長や行送りなどを取り出すためのテキスト処理文脈
- *ib* ： 行分割対象となるインラインボックス列

b_1 および b_2 はあくまで組み上がった段落の前後でページ分割してよいかどうかの指定であり，段落中での改ページには干渉しないことに注意されたい．典型的な利用場面としては，章節の題の直後で改ページされないように指定することなどが挙げられる．この文脈で「段落」が指すのはあくまで“文字が適切に行分割されて行が連なったもの”であり，文書上のマークアップで段落と扱われるデータであるとは限らない．章題も文字が一度行方向に並べられ，（ほとんどの場合は 1 箇所も行分割されないとはいえ，形式上は）行分割処理を経て行がページの進行方向に連なるものへと組み上げられた結果という意味では立派な「段落」なのである．

行分割が実際にどのようなアルゴリズムで行なわれるかについては第 6 章を参照してもらうとして，ここでは行分割処理が“インターフェイスとしてどのように使えるのか”を扱う．和文組版での段落を組むブロックコマンド `+p` の，最もシンプルな定義は次のようなものだ：

```
let-block ctx +p it-inner =
  let ib-indent = inline-skip (get-font-size ctx *' 0.88) in
  let ib-inner = read-inline ctx it-inner in
    line-break true true ctx
```

44 このインターフェイスは設計した著者もあまり美しいとは思えないのだが，頻出のプリミティヴであるため互換性の都合上存続している．将来の非互換なヴァージョンで姿を変えるかもしれない．

```
    (ib-indent ++ ib-inner ++ inline-fil)
```

`let-inline` と同様，`let-block` によるブロックコマンドの定義も“テキスト処理文脈を外から渡してもらう”ための第 0 引数 `ctx` をとることができる．ここで定義した `+p` はユーザが与える引数 `it-inner` としてインラインテキストを受け取る．最初に定義している `ib-indent` はいわゆる字下げのための“白いインラインボックス列”だ[*45]．これの定義には

`inline-skip` : length → inline-boxes

というプリミティヴを用いている．`inline-skip` l という形で使い，長さ l だけ文字の進行方向に幅をもつ，伸縮しない空白をつくる[*46]．ここの例で与えられている長さは `get-font-size` を用いて取得した現在の文字サイズを 0.88 倍したものである．0.88 という数字を決め打ちでハードコードしてしまっているが，これは欧文に対する和文のデフォルトの**拡大率**を表している．フォントに於ける拡大率の概念については第 7 章および第 8 章で扱うので，今は特に気にしなくてよい．こうして段落の字下げ部分ができたところで，続いて段落の本領であるテキスト内容の処理にとりかかる．ここはそれほど変哲はなく，前述の `read-inline` と現在の文脈 `ctx` を用いてインラインテキスト `it-inner` を“字幅などをもつ，組める状態”であるインラインボックス列へと変換し，変数 `ib-inner` をそれに束縛する．

最後はこれらをつなげて行分割して仕上げである．インラインボックス列同士を結合させる処理は

`++` : inline-boxes → inline-boxes → inline-boxes

というプリミティヴによって行なう．これを用いて `ib-indent ++ ib-inner` とすることで，行分割して段落に組み上げたい内容が完成する．と言いたいところだが，実はこれを `line-break` の第 4 引数として渡すと意図どおりにはならない．というのも，`line-break` が行なうのは“各行を最後の端まで使い切っている段落への整形”だからだ．つまり，`ib-indent ++ ib-inner` を `line-break` の第 4 引数に渡すと，最終行も行長いっぱいに伸びるように組まれた段落になっているブロックボックス列が返ってくる．通常の段

45 活版印刷でいうところの**クワタ**の役割をするものである．

46 何らかの“許容範囲”で伸縮する空白は `inline-glue` というプリミティヴを用いてつくる．これについては第 6 章を参照されたい．

落を組むとき，常に最終行を最後までぴっちりと行長を使い切って組むことは全く意図していない．「最終行は途中でも終わってよい」ということを行分割処理を行なう機構に教えてあげるにはどうすればよいのだろう？　こういった使えるのが次のプリミティヴである：

`inline-fil` : inline-boxes

これは“途中で行分割されることはないが，周囲の要請に応じてどこまでも伸ばせる，任意の非負の長さをもてる空白”である[*47]．これをインラインボックス列の最後に結合させることで，「最終行は途中のどこで終わってもよい」ということを行分割処理を担う機構に間接的に教えられるのである．したがって行分割させるインラインボックス列は `ib-indent` `+` `+` `ib-inner` `++` `inline-fil` ということになる．この `+p` の使用例はあらためて掲げる必要もないだろう．この本の本文中の段落は上に掲げた定義と本質的に同等な定義の `+p` を用いて組まれており，いたるところに使用例があるからだ．以上が行分割処理の非常にオーソドックスな用例である．

ブロックコマンドもインラインコマンドと同様の方法で型がつけられる．すなわち，

`let-block` *ctx cmd* $x_1\ x_2 \cdots x_n$ = *expr*

でブロックコマンド *cmd* を定義する際は第 0 引数 *ctx* には context 型がつけられ，*expr* は各引数 x_i が τ_i 型をもつ下で block-boxes 型がつくことが要請される．以上によって定義した *cmd* は

$[\tau_1;\ \tau_2;\ \cdots;\ \tau_n]$ block-cmd

という型がつけられる．例えば上で定義した `+p` は [inline-text] block-cmd という型がつく．

`inline-fil` は“単に自由にどこまでも伸びる”だけではなく，同一の行に複数個挿入されていると全部同じ長さで組まれる，という機能を備えている．これを使えばいわゆる中央揃えなども行分割処理を用いて実現できる：

```
let-block ctx +centering it =
```

47 TEX ユーザなら，`\fil` に相当する機構だと言えばおそらくすぐわかるだろう．ちなみに現在のところは `\fill` に相当する機構，つまり“より強い無限伸長グルー”は SATYSFI には備わっていない．

```
  line-break true true ctx
    (inline-fil ++ read-inline ctx it ++ inline-fil)
```

この `+centering` を用いた例が以下である：

```
+centering{The \SATySFi;book}
```

The SATYSFIbook

ところが，このシンプルな実装は内容が長くなると困ったことになる：

```
+centering{
  本書は，2017 年度未踏事業のひとつとして筆者が開発し，
  2018 年 8 月現在も発展を続けている新しい組版処理システム \SATySFi; について
  解説することを目的としたものである．
}
```

本書は，2017 年度未踏事業のひとつとして筆者が開発し，2018 年 8 月現在も発展を続けている新しい組版処理システム SATYSFI について解説することを目的としたものである．

おそらく，複数行のときに期待される最も自然な挙動は「適切な長さで行に分割され，それぞれ中央揃えになっている」だろう．実はこれは案外複雑な話で，完全に自動で処理する方法は今のところおそらくない．第 6 章で紹介する `set-every-word-break` を使えば一応それらしいことはできるが，ハイフネーションのない欧文にしか対応できない．プリミティヴの挙動の多少の非互換な変更を許すなら，それほど大きな変更なく可能になるだろう．とりあえず今の我々にできるシンプルな回避策は，人間がどこで切って各行にするかを指定する方式にするというものだ：

```
let-block ctx +centerings its =
  let l = get-font-size ctx *' 0.25 in
  let bbs =
    its |> List.map (fun it ->
      line-break true true (ctx |> set-paragraph-margin l l)
        (inline-fil ++ read-inline ctx it ++ inline-fil))
  in
    bbs |> List.fold-left (+++) block-nil
```

この `+centerings` は [inline-text list] inline-cmd という型をもつ．なにやら一気に複雑そうな定義が出てきてしまったが，とりあえず用例を見て正しく動くことを確認しよう：

```
+centerings{
| 本書は，2017 年度未踏事業のひとつとして筆者が開発し，
|  2018 年 8 月現在も発展を続けている新しい組版処理システム \SATySFi; について
| 解説することを目的としたものである．
|}
```

本書は，2017 年度未踏事業のひとつとして筆者が開発し，
2018 年 8 月現在も発展を続けている新しい組版処理システム SATYSFI について
解説することを目的としたものである．

用例にも

{|*text*$_1$|⋯|*text*$_n$|}

という新しい構文が出てきているが，これはそれほどヘンテコなものではなく，一種のインライン引数であって単にインラインテキストのリストを簡素に書くために用意されている糖衣構文であり，以下の式引数と等価である：

[{*text*$_1$}; ⋯; {*text*$_n$}]

それではあらためてコマンド定義を見てみる．`l` が文字サイズの 0.25 倍の長さに束縛されるのはよいとして，重要なのは `bbs` : block-boxes list の定義だ．これは受け取ったインラインテキストのリスト `its` : inline-text list を要素ごとにブロックボックス列に変換している処理であり，要素ごとの処理を記述しているのが函数

`List.map` : $\forall\alpha.\ \forall\beta.\ (\alpha \to \beta) \to \alpha\ \mathrm{list} \to \beta\ \mathrm{list}$

だ．これは第 3 章で説明をサボってしまったいわゆるマップ函数である．型からもわかるように，マップ函数は函数とリストを受け取ってリストの各要素にその函数を適用して結果のリストを返すという高階函数である．イメージを簡単に書けば

$$\texttt{List.map}\, f\, [v_1;\ v_2;\ \cdots;\ v_n] \longrightarrow [f\, v_1;\ f\, v_2;\ \cdots;\ f\, v_n]$$

という具合である．`+centerings` の定義中の `List.map` の第 1 引数は，インラインテキ

ストを受け取ってそれをインラインボックス列に変換したものの前後に `inline-fil` をつけて `line-break` に渡すことで“中央揃えの 1 行”であるブロックボックス列をつくる函数であり，これを各行に対してやった結果のリストが `bbs` ということだ．そして最後にやることはこの `bbs` の要素をすべて結合してひとつのブロックボックス列にすることである．これは

$$\texttt{List.fold-left} : \forall\alpha.\ \forall\beta.\ (\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \mathrm{list} \to \alpha$$

といういわゆる左畳み込み函数を用いて行なう．これは“2 引数函数”，“畳み込みの初期値”，“畳み込まれるリスト”を受け取って“畳み込み”を行なう高階函数である．“2 引数函数”を $\oplus$ と 2 項演算子で書くことにすると，“畳み込み”は以下のイメージのような処理である：

$$\begin{aligned}
&\texttt{List.fold-left}\ (\oplus)\ i\ [v_1;\ v_2;\ v_3;\ \cdots;\ v_n] \\
&\longrightarrow \texttt{List.fold-left}\ (\oplus)\ (i \oplus v_1)\ [v_2;\ v_3;\ \cdots;\ v_n] \\
&\longrightarrow \texttt{List.fold-left}\ (\oplus)\ ((i \oplus v_1) \oplus v_2)\ [v_3;\ \cdots;\ v_n] \\
&\ \vdots \\
&\longrightarrow ((\cdots(((i \oplus v_1) \oplus v_2) \oplus v_3) \oplus \cdots) \oplus v_n)
\end{aligned}$$

したがってこの $\oplus$ として `+++` を用いると `bbs` の要素を順に結合してひとつのブロックボックス列にすることができる．これを記述したのが最終行なのである．ここで“初期値”として用いた

`block-nil` : block-boxes

は，任意のブロックボックス列 *bb* に対して *bb* `+++` `block-nil` と `block-nil` `+++` *bb* と *bb* とがまったく等価になるようなプリミティヴである．ちなみにインラインボックス列にも対応するものがあり，

`inline-nil` : inline-boxes

はやはり任意のインラインボックス列 *ib* に対して *ib* `++` `inline-nil` と `inline-nil` `++` *ib* と *ib* とがまったく等価になるようなプリミティヴである．

なお，`+centerings` の定義中にあった `set-paragraph-margin` は `line-break` が返す段落の上下の**マージン** (*margin*) を変更するプリミティヴであり，これについては第 6

章で扱う．

5.4. ブロック変換器

ところで，`+p` のようなブロックコマンドを含むブロックテキストはどのようにブロックボックス列へと変換されているのだろう？　実はブロックテキストをブロックボックス列に変換するのも専用の機構があり，**ブロック変換器** (*block converter*) と呼ぶ．本文中では触れてこなかったが，すでに図 4.1 にチラッと登場しているので気づいた方もいるかもしれない．インライン変換器が `read-inline` というプリミティヴで提供されているように，ブロック変換器にも対応するプリミティヴがある：

`read-block` : context → block-text → block-boxes

使い方もほぼ `read-inline` と同様だ．例えば，`+section` の簡単な実装は以下のように書ける：

```
let-mutable section-number <- 0

let-block ctx +section it-title bt-content =
  let () = section-number <- !section-number + 1 in
  let it-num = embed-string (arabic !section-number) in
  let bb-title =
    let ib-title =
      read-inline (ctx |> set-section-context)
        {#it-num; #it-title;}
    in
      line-break true false ctx
        (ib-title ++ inline-fil)
  in
  let bb-content =
    read-block ctx bt-content
  in
    bb-title +++ bb-content
```

`section-number` : int ref は節番号を記録しておくための可変参照である．これは `+section` が呼び出されるたびにインクリメントして使っている．`arabic` と `embed-string` を使って節番号をインラインテキストに変換して `it-num` をこれに束縛し，この `it-num` をユーザが第 1 引数として与えた節のタイトル `it-title` と `{#it-num; #it-title;}` でインラインテキストとして結合する．inline-text 型の変数 x の内容がインラ

インテキスト中で `#x;` という構文により埋め込めるのである． 実はこれは初出ではなく， すでに第 3 章で見た簡単なエイリアスの定義の際に使った記法である． このインラインテキストを `read-inline` で変換したインラインボックス列に `ib-title` を束縛し，`inline-fil` を結合させて `line-break` に渡すことにより節のタイトル `bb-title` をつくる．タイトル直後ではページ分割してほしくないので，`line-break` の第 2 引数には `false` が与えられている．ここまではそれほど新しいことはやっていない．途中で `set-section-context` という函数が出てくるが，これは単に `+section` の定義よりも手前のどこかで定義されている節のタイトル用の文字サイズやフォント等の設定を行なってテキスト処理文脈に反映させる函数だ．具体的には，例えば以下のような実装になっていることだろう：

```
let set-section-context ctx =
  ctx |> set-font-size 18pt
      |> set-font Latin (`lmsans`, 1., 0.)
      |> set-font Kana (`ipaexg`, 0.88, 0.)
      |> set-font HanIdeographic (`ipaexg`, 0.88, 0.)
```

`read-block` が使われるのは節の本文，すなわち第 2 引数として与えられたブロックテキスト `bt-content` をブロックボックス列 `bb-content` に変換する部分だ． この過程で，節の本文である `+p` で書かれた段落や `+subsection` などを処理する変換が走る．そして最終的に節のタイトルと本文のブロックボックス列が

`+++` : block-boxes → block-boxes → block-boxes

というプリミティヴによって結合してめでたく最終結果となる．「簡素なクラスファイルをつくるのにもこんな複雑な記述をしなければならないのか」と率直に感じた読者もいらっしゃるかもしれないが，慣れると特に負担に感じることはない．何より，処理として意味をなさない， 不整合のある記述をしていたら型検査器が該当箇所に型エラーを出して注意してくれるので，それほど苦労せずに修正でき，スムーズに書き上げられるのである．以上がブロック変換器の仕組みである．

Chapter 6
行分割処理

6.1. 行分割処理の概要

行分割処理に於いて，見た目が十分整うように行を段落に切り分けるのはそれほど自明ではない．まずは行を切り分けて良い箇所の決定である．例えばラテン文字やキリル文字といった“欧文”は基本的に単語間の空白かハイフネーション上許容される単語の途中の箇所かのいずれかでしか行を分けてはいけないという制約があるし，和文でも開き括弧の直後や小書きの「っ」や音引き「ー」の直前など，行を分けてはいけない箇所の“規定”がいくらかある．このほか，例えば読点の直後で行を分けた場合は行の途中の読点とは違って読点の後ろに 2 分アキを入れない，つまりいわゆる“全角幅”の半分の空白を設けないなどといった，行末が揃っているように見せるための“規則”もある．こういった，行を切り分けてよい箇所・いけない箇所の区別と切り分けた場合の特別な処理を SATYSFI で実装するにあたっては，まず多言語対応の組版を意図して Unicode Line Breaking Algorithm [32] に原則従うことにし，特にひとまず和文・欧文の混植を優先して実現することを目的としていたのでこの和文の組版規則については『日本語組版処理の要件』[34]，通称 JLreq にできるだけ従って実装した．ハイフネーションについては，やはり偉大な先駆者である TEX 処理系に倣って Liang–Knuth アルゴリズム [19] をもとに実装した．

また実際に切り分ける箇所を決定する処理も自明ではない．前から順に詰められるだけ詰めて切り分けるという単純な貪欲アルゴリズムでは後ろの行が前に比べて過度に間延びして美しく見えないような出力もできてしまうから，“段落全体を見渡してから行分割箇所を決める”ような処理を効率的に行なう必要があるのである．この行分割に関しては，Knuth–Plass アルゴリズム [17] と呼ばれる，TEX で採用されたものとおおよそ同様の処理を独自に拡張して用いることにした．このアルゴリズムでは，端的に言えば**最短経路問題**に帰着することで適切な行分割箇所を選んでいる．独自に拡張したのは，入れ子になった枠のついた部分でも途中で行分割できるようにしたことに関してである．

Knuth–Plass アルゴリズムに於ける行分割処理の最短経路問題への帰着はシンプルにして巧妙であり，実用上もかなり多くの組版上の要請に応えられるものとなっている[*48]．行分割処理を担う機構は，まず段落に組み上げるインラインボックス列を受け取って，その中の行分割候補となる各箇所とボックス列の先頭および末尾を頂点とし，この間に重みつきの辺を張って有向グラフをつくることを考える．この有向グラフは，頂点 u と v の間に「u で行分割したあとに v で行分割するとその間にどの程度体裁の悪い行が生じるか」という罰則値を重みとしてもつ辺を張って構築する．この罰則値は

- 単語間の空白があらかじめ決められた自然な長さと比べてどの程度伸縮することになり，それは許容できる伸縮量の基準値に対してどの程度の割合か
- v それ自体がもつ “行分割してほしくなさ” である**ペナルティ値** (*penalty*)

に基づいて計算される[*49]．そして，この有向グラフ上でボックス列の先頭から末尾までを結ぶ経路のうち，長さ（すなわち各辺の重みの合計）が最小となるものを選ぶと，段落全体として最も端整な行のセットが得られ，その経路をなす頂点で行分割すればよいことがわかる，という仕組みである．前述のとおり，SATYSFI も TEX で採用されたこの巧妙な仕組みを継承して用いている．以下でこの行分割処理を制御するプリミティヴを紹介する．

6.2. 行分割に関するプリミティヴ

6.2.1. 行分割候補箇所

第 5 章に於いて，行分割処理そのものを担うプリミティヴが

`line-break` : bool → bool → context → inline-boxes → block-boxes

であることはすでに第 5 章で紹介したが，分割される箇所の候補がどのように決定され，その処理にユーザやパッケージ開発者がどのように介入できるかについてはまだ触れていなかった．これについて以下で少し解説する．本章のほかに，行分割の仕組みを扱っているわかりやすい文献としては [18, 4] などがあるので是非参照されたい．

---

48 Knuth 自身も “TEX's line-breaking algorithm has proved to be general enough to handle a surprising variety of different applications; this, in fact, is probably the most interesting aspect of the whole TEX system.” と述べている [16].

49 あまりにも体裁が悪い行ができるとわかる場合（例えば単語間の空白を自然な長さにすると行長に全く届かないような，短い数単語しか含まない行が生じる場合）はそもそも辺を張らないことによって最短経路を求める計算効率を上げる．

SATYSFI では，行分割の候補となる箇所がインラインボックス列の中に含まれており，そのいずれも

- そこで行分割されなかった場合の内容 b_0
- そこで行分割された場合に直前の行末に入る内容 b_1
- そこで行分割された場合に直後の行頭に入る内容 b_2
- そこで行分割された場合に加算されるペナルティ値 p

からなる 4 つの情報 $\left(b_0, b_1, b_2, p\right)$ をもつ．逆に，これらの情報をもたない箇所では行分割しない[*50]．この 4 つ組を備えた箇所を以降単に**行分割候補箇所** (*line break opportunity*) と呼ぶことにしよう．ペナルティ値は実際の行分割箇所を決定するにあたって参考にされ，この値が大きいほどその候補箇所での行分割が抑制される[*51]．これについては6.2.2 節で後述する．

行分割候補箇所は，インラインテキストからインラインボックス列に変換する過程で現れる．この際に，例えば空白文字は b_0 が（単語間の空白として適切な程度に）多少伸縮する空白，b_1 と b_2 が空っぽであるような行分割候補箇所に変換される．“table” という文字列は，b_0 が table，b_1 が ta–，b_2 が ble であるような行分割候補箇所へと変換される．すなわち，途中で行分割する必要がなければ単に行中で table と組まれるように，途中でハイフネーションを行なって分割する方が周辺の要素の並びから望ましいと判断される場合はその行末が ta– で次の行頭が ble になるように，それぞれ指定した行分割候補箇所になるのである[*52]．インラインテキストがどのようにインラインボックス列に変換され，どのように行分割候補箇所が現れるかは，前述のとおり基本的には Unicode Line Breaking Algorithm [32] と『日本語組版処理の要件』[34] に基づいているが，多言語組版を想定して将来的にはユーザやパッケージ開発者が介入して変更できるようにしたい

---

50 厳密に言うと例外がある．ブロックボックス列を “段落の一部として埋め込む” 機能を担う `embed-block-breakable` というプリミティヴによって生成されたインラインボックス列は，その直前で必ず行分割される．

51 一応（行分割を促進する意図で）負の値を指定することもできるが，現在の実装は負の値が渡されることを想定していないので，行分割が実際に促進される保証はない．

52 実際のハイフネーション処理はもう少しだけ複雑である．単語中でハイフネーションが可能な箇所は 2 箇所以上あったりするからだ．

と考えている．

行分割候補箇所はインラインテキストからインラインボックス列への変換過程で生成されるだけでなく，“手動で” 挿入することもできる．これに用いるのがプリミティヴ：

`discretionary` : int → inline-boxes → inline-boxes → inline-boxes → inline-boxes

である．使い方は至極単純で，`discretionary` p ib_0 ib_1 ib_2 でペナルティ値 p が p，各 b_i がそれぞれ ib_i であるような行分割候補箇所が生成される．なお，`discretionary` の引数に渡した ib_i の中にさらに入れ子で行分割候補箇所が含まれている場合，その箇所では決して行分割されない．つまり，そこでは必ず b_0 の内容が選ばれる．

6.2.2. ペナルティ値

インラインテキストからインラインボックス列への変換の過程で与えられる各行分割候補箇所のペナルティ値 p は，基本的に `read-inline` の第 1 引数として渡したテキスト処理文脈に基づいて決定される．単語間空白から変換される行分割候補箇所 (p, b_0, b_1, b_2) は，デフォルトでは[*53] $p = 100$ であり，b_1 と b_2 は空である．b_0 は伸縮する空白であり，**グルー** (*glue*) と呼ばれる．これについては6.2.3 節で後述する．p はプリミティヴ：

`set-word-break-penalty` : int → context → context

により変更できる．すなわち，テキスト処理文脈 *ctx* と単語間空白の行分割候補箇所に対して設定したいペナルティ値 p に対して `set-word-break-penalty` p *ctx* でペナルティ値が p に更新されたテキスト処理文脈が返される．b_1 と b_2 も変更でき，

`set-every-word-break` : inline-boxes → inline-boxes → context → context

によって行なう．すなわち `set-every-word-break` ib_1 ib_2 で ib_i をそれぞれ単語間空白の行分割候補箇所の b_i に設定する．現在単語間空白の b_1 と b_2 に設定されている内容を取得するのは

`get-every-word-break` : context → inline-boxes ∗ inline-boxes

による．

---

53 プリミティヴ `get-initial-context` が返すテキスト処理文脈では，の意．

例 6.1　フランス語の表記の或る流儀では，引用符であるギュメに囲まれた部分が複数行に亘って続く場合，2 行目以降の先頭にも開く側の引用符を置く．このような処理を自動でやってくれる引用符のコマンド，すなわち以下のような記述と組版結果を実現するようなコマンド `\q` を定義したい（本文は [28] より）：

```
+p{
  \q{
    The original problem to be solved was
    for fine French typesetting,
    in which guillemets are placed
    running down the left side of a paragraph,
    as in this paragraph,
    so long as material is being quoted.
  }
  Since \TeX; breaks paragraphs in arbitrary places,
  it was impossible to develop a robust macro package
  that could, in a single pass, place the guillemets
  in the right positions.
}
```

« The original problem to be solved was for fine French typesetting, in « which guillemets are placed running down the left side of a paragraph, as in this « paragraph, so long as material is being quoted. » Since TEX breaks paragraphs in arbitrary places, it was impossible to develop a robust macro package that could, in a single pass, place the guillemets in the right positions.

これは `set-every-word-break` を使えばおおよそ実現できる：

```
let-inline ctx \q inner =
  let ib-after = read-inline ctx {«\ } in
  let ctxq =
    ctx |> set-every-word-break inline-nil ib-after
  in
    read-inline ctxq {« #inner; »}
```

要するに「単語間空白で行分割された場合は次の行頭にギュメの開く側をつける」と

いう指示をギュメの内側のインラインテキストを変換するときに使うテキスト処理文脈に与えることで所望の処理を実現している．ただし，これはハイフネーションで行分割された場合には残念ながら対応できていない．現在の SATYSFI はまだハイフネーション時に挿入される文字ないしボックス列を変更するプリミティヴを提供していないためである[*54]．

ハイフネーションによるペナルティもテキスト処理文脈が保持しており，

`set-hyphen-penalty` : int → context → context

によって変更できる．デフォルトでは 100 に設定されている．

6.2.3. グルー

行分割の際，単語間の空白や和文の文字の間などの箇所は両端を揃えるために多少伸縮するのであった．これは**グルー** (*glue*) と呼ばれる仕組みによって実装されている．グルーは

- 自然な長さ l_0
- 収縮の基準となる非負の長さ l_1
- 伸長の基準となる非負の長さ l_2

の 3 つの情報 (l_0, l_1, l_2) からなっている．インラインテキストからインラインボックス列へ変換される過程で単語間空白などから生じるが，プリミティヴ：

`inline-glue` : length → length → length → inline-boxes

を用いて "手づくりする" こともできる．すなわち `inline-glue` l_0 l_1 l_2 で各 l_i が l_i のグルーが返される．なお，段落先頭の字下げなど伸縮せず一定の長さのものは**スキップ** (*skip*) と呼ばれ[*55]，プリミティヴ：

`inline-skip` : length → inline-boxes

---

54 実装は簡単だが，互換性を損なう変更であるため未実装となっている．互換性のない更新をまとめて行なう際に追加する予定である．

55 **カーン** (*kern*) と呼ばれることの方が一般的かもしれないが，現在の SATYSFI では慣習的にスキップと呼んでいる．

でつくることができる[*56].

インラインテキストがインラインボックス列に変換される過程で単語間空白がどのようなグルーに変換されるかは，`read-inline` に第 1 引数として渡されたテキスト処理文脈が保持しているフォントサイズ s および浮動小数点数のパラメータ r_0, r_1, r_2 によって決まり， それぞれ $l_i := r_i s$ で定められる． デフォルトでは $r_0 = 0.33$, $r_1 = 0.08$, $r_2 = 0.16$ になっているが，プリミティヴ：

`set-space-ratio` : float → float → float → context → context

によって変更できる．すなわち `set-space-ratio` r_0 r_1 r_2 *ctx* でテキスト処理文脈 *ctx* を各 r_i が r_i になるように更新したものを返す．

すでに第 3 章で登場したが，`inline-fil` も機能上はグルーの変種である．おおらかには，罰則なしに任意の長さまで伸ばすことのできる，自然長 0，縮み許容量 0 のグルーと捉えられるのである． 行分割処理にかけるインラインボックス列の末尾に `inline-fil` をくっつけることで “行の途中でテキストが終わってよいこと” を表現できたり， また 1 行になるとわかっているインラインボックス列の前後に `inline-fil` を 1 つずつ附加して `line-break` に渡すことで中央揃えの行をつくることができたりする． では， 複数行になるインラインボックス列を適切に切り分けて中央揃えにするにはどうすればよいだろうか？ これは `set-every-word-break` と `inline-fil` を使えばおおよそ実現できる[*57]ので，考えてみてほしい．

6.3. ブロック方向の長さの制御

標準の行送りや行間の最小の長さなどを指定する機能を紹介する． **行送り**とは段落中に於けるベースラインから次のベースラインまでの段落方向の距離であり，行の内容の深さや高さが邪魔しない限り一定の行送りで行が並べられる． 行間とは行の下端とその次の行の上端との距離であり， これには許容する最小の長さ t を決めておく． これは行の深さを d,

---

56 生成されるインラインボックス列の内部表現が異なるものの，振舞いの上では `inline-skip` l は `inline-glue` l `0pt 0pt` と等価である．このため，`inline-skip` は将来的にプリミティヴからパッケージで定義される函数へと “降格” するかもしれない．

57 「おおよそ」というのは「現状の SATYSFI だと， 単語間で行分割された場合は問題ないが， ハイフネーション箇所で行分割されたときには残念ながら対処できない」ことを指している．これは例 6.1 で言及した問題と同様である．

次の行の高さを h としたとき，その行から次の行への行送りを少なくとも $d + t + h$ にするのに使う．つまり，或る行の深さと次の行の高さの和が長すぎて標準の行送りでは詰まりすぎたり重なってしまったりするのを防ぐ意図でこの t を設定する．これらの情報もやはりテキスト処理文脈が保持しており，`line-break` に第 3 引数として与えたものから取り出して使われる．これらの値を変更するプリミティヴは以下のとおり：

`set-leading` : length → context → context

標準の行送りを第 1 引数の値に設定するよう更新したテキスト処理文脈を返す．

`set-min-gap-of-lines` : length → context → context

行間の最小値を第 1 引数の値に設定するよう更新したテキスト処理文脈を返す．

インライン方向のスペーシングの指定は基本的に文字サイズに対する割合で行ない，文字サイズだけ変更すればそれに伴って自動で拡大・縮小されたのだが，ブロック方向に関する長さの指定は絶対的な長さで指定するインターフェイスであることに注意．

また，`line-break` が返すひとまとまりの段落には上下に**マージン** (*margin*) という部分がついている． これは CSS に於けるマージンと類似したもので， ブロック方向の長さを持っている． マージン同士が隣り合うと， より長い側の長さでブロック方向のスペースがつくられる． つまり， マージンとは “隣り合うものに少なくともこれだけ離れていてほしいという距離” である． マージンの大きさはテキスト処理文脈が保持しており，`line-break` に渡されたテキスト処理文脈から取り出して使われる[*58]．マージンを変更するプリミティヴは次のとおり：

`set-paragraph-margin` : length → length → context → context

ctx `|> set-paragraph-margin` l_1 l_2 で段落の上のマージンを l_1 に， 段落の下のマージンを l_2 にするように設定を更新したテキスト処理文脈を返す．

6.4. ブロックボックス列の行分割内容への挿入

第 2 章の説明に於いて， 直観的にはインラインテキストやインラインボックス列が “文

58 第 5 章でも言及したが，段落の前後で改ページしてよいか否かの情報は `line-break` の引数として与えておきながらマージンの長さはテキスト処理文脈が保持しているということについてはやや一貫性に乏しいインターフェイスにしてしまったと感じている． 将来的に非互換な変更でこれに関して整理することを検討している．

字が進む方向に連なるデータ”，ブロックテキストやブロックボックス列が“段落が進む方向に連なるデータ”であるという旨の説明をした．しかし実際には箇条書きや別行立て数式など“段落の進む方向に連なるデータ”としか思えないものが `\listing` や `\align` などのインラインコマンドの適用で書けた．これらの内容は途中でページ分割すらできるから，“大きな 1 文字になっている”という言い逃れもできないはずだ．第 2 章ではこの話題を避けていたのだが，そろそろ読者もこの SATγSF$_{\mathrm{I}}$ の矛盾した態度には疑問を呈さざるを得ないのではなかろうか．実際，この“矛盾”を実現するプリミティヴが用意されている：

`embed-block-breakable` : context → block-boxes → inline-boxes

`embed-block-breakable` *ctx bb* でブロックボックス列 *bb* を途中でページ分割可能な部分としてインラインボックス列に擬態させる．つまり，`embed-block-breakable` の返り値の正体はブロックボックスが擬態した特殊なインラインボックス列である．これを含むインラインボックス列が `line-break` に渡されると，行分割処理を担う機構はこの擬態したブロックボックス列の直前では行長の最後まできっちり使い切るように行分割する．そして綺麗に揃った状態でブロックボックス列になった下端に，ヴェールを被って擬態していたブロックボックス列をヴェールの中から取り出し，そのままそこにブロックボックス列として（`+++` による結合と同じ具合に）くっつける．そしてこの擬態ブロックボックスより後方の内容はまた左端から通常の要領で行分割され，そしてその結果できた，（左端から始めたので当然）綺麗に上端が揃ったブロックボックス列を，擬態ブロックボックス列由来の部分の後ろにやはりくっつけるのである．こうした経緯で，インラインボックス列中にヴェールを被って潜んでいたブロックボックス列が途中でページ分割可能な状態のままシレッと行分割中に挿入されるのだ．

実際，`itemize` パッケージの `+listing` と `\listing` の定義の概要部分だけ抜き出して簡略化した以下のような定義を見ると，

```
let listing ctx (Item(_, itmzs)) =
  let bbs = List.map (listing-item ctx 0) itmzs in
    block-skip item-gap-outer
      +++ (List.fold-left (+++) block-nil bbs)
      +++ block-skip item-gap-outer

let-block ctx +listing item = listing ctx item

let-inline ctx \listing item =
  inline-fil ++ embed-block-breakable ctx (listing ctx item)
```

と両者に共通の補助函数 `listring` : context → itemize → block-boxes を呼んでおり，`embed-block-breakable` を通しているかくらいしか違いがない．ただし，ここで手前についている `inline-fil` は重要だ．これは第 5 章での `+p` の定義で末尾に `inline-fil` を結合させたのと同様の理由による．また

`block-skip` : length → block-boxes

は受け取った長さの“ブロック方向の空白”を返すプリミティヴである．

6.5. 長さの取得に関するプリミティヴ

文字幅などを取得する便利なプリミティヴがいくつかある：

`get-natural-metrics` : inline-boxes → length ∗ length ∗ length

与えられたインラインボックス列を行分割せずに 1 行で組んだ時の自然な長さ，高さ，深さの 3 つ組を返す．

`get-natural-length` : block-boxes → length

与えられたブロックボックス列のブロック方向の長さを返す．

`get-text-width` : context → length

テキスト処理文脈に格納されている行長を取り出す．

Chapter 7
システムと設定ファイル

この章では SATYSFI がどのようにフォントやパッケージや設定ファイルを読み込んで使用するかといったシステムに関する説明を提供する．SATYSFI が使用するフォントファイル，標準パッケージ，設定ファイルなどの一式は，次のような構造をもつ**ライブラリルート** (*library root*) と呼ばれるディレクトリの下に収められている：

```
LIBROOT/
├─ dist/
│   ├─ fonts/
│   │   └─ ⋮
│   ├─ hash/
│   │   ├─ default-font.satysfi-hash
│   │   ├─ fonts.satysfi-hash
│   │   └─ mathfonts.satysfi-hash
│   ├─ hyph/
│   │   ├─ english.satysfi-hyph
│   │   └─ ⋮
│   ├─ packages/
│   │   ├─ gr.satyh
│   │   ├─ list.satyg
│   │   ├─ stdjareport.satyh
│   │   └─ ⋮
│   ├─ unidata/
│   │   ├─ EastAsianWidth.txt
│   │   ├─ LineBreak.txt
│   │   ├─ Scripts.txt
│   │   └─ ⋮
```

LIBROOT/ は，UNIX 系 OS では

- `~/.satysfi/`
- `/usr/local/share/satysfi/`
- `/usr/share/satysfi/`

のいずれかであり，SATYSFI は設定ファイル *foo* が必要となった際に `~/.satysfi/`*foo*，`/usr/local/share/satysfi/`*foo*，`/usr/share/satysfi/`*foo* をこの優先度で探し，最初に存在したものを用いる．すなわち，*LIBROOT/* はファイル *foo* ごとに決定される．上記パスの 2 箇所以上にファイルを置く場合は，この点に注意する必要がある．Windows の場合は *LIBROOT/* として

- `%userprofile%\.satysfi\`
- `%SATYSFI_RUNTIME%\`

がこの優先度で使われる [7]．以降，設定ファイル類は *LIBROOT/* からの相対パスで書くこととする．

7.1. 各ディレクトリの役割

まず，*LIBROOT/* の直下には `dist/` がある．これは内容物が将来的にパッケージマネージャやディストリビューションによって操作される（すなわち，ユーザが手で内容物を変更するべきでない）ことを意識して “distribution” から命名されている．現在はこのディレクトリ以下の内容をユーザが操作することは非推奨ではなく，また（カレントディレクトリが *LIBROOT/* 以下の状態で SATYSFI を起動しない限り）SATYSFI が内容を変更することはない．将来的には `dist/` の他に `local/` を設け，これにユーザが自身が作成したパッケージ類を格納できるようにすることを想定している．

`dist/hash/` と `dist/unidata/` は設定ファイルの格納場所であり，どのような名前のファイルが直下に置かれている必要があるかも決められている．他のディレクトリは直下にデータを格納する．

`dist/hash/` には SATYSFI コードからデータにアクセスするためのいくつかの設定ファイルが格納されており，それぞれ YOJSON 形式（yojson パッケージ [22] により拡張された JSON 形式をこう呼ぶことにする）で記述されている．

`dist/unidata/` は Unicode コードポイントを組版処理中にどのように扱うかを規定したファイルを格納しており，これは Unicode Database として公開されているテキストファイル群の一部である．

`dist/fonts/` は SATYSFI で使用する OpenType フォントファイルを直下に格納する．

`dist/hyph/` はハイフネーション辞書を直下に格納する． 現状では英語のハイフネーションのみに対応しており，`dist/hyph/english.satysfi-hyph` というデータのみを利用するが，将来的に多言語へと拡張し，`dist/hash/` 以下でハイフネーション辞書の設定を扱う想定である．

`dist/packages/` はパッケージ類を格納する．パッケージファイルを

`dist/packages/`*path/to/foo*(`.satyh` | `.satyg`)

というパスに置くと，パッケージとして

```
@require: path/to/foo
```

で読み込めるようになる． 拡張子が `.satyh` のものと `.satyg` のものが両方存在する場合は `.satyh` が優先される[59]．

7.2. フォント設定

SATYSFI で使用できるフォントは**フォントハッシュファイル** (*font hash file*)：

`dist/hash/fonts.satysfi-hash`

に設定の記載のあるフォントである． ユーザが新しいフォントを SATYSFI で使えるように設定したい場合は，`dist/fonts/` 直下に所望のフォントファイル *bar*`.otf` を置き，フォントハッシュファイル内の YOJSON のディクショナリに

```
"bar": <Single {"src-dist": "bar.otf"}>
```

というエントリを追記する． この設定を加えることでフォント名 `` `bar` `` が SATYSFI で使えるようになる． フォントの（拡張子を除いた）ファイル名とフォント名は一致していなくても構わない． 所望のフォントが TrueType Collection のフォントファイル *baz*`.ttc` の

59 拡張子 `.satyg` は，テキストモードと PDF モードとで共通に使えるパッケージファイルにつけることを想定している．これは将来的な拡張の都合であり，現在はどちらでも特に違いはない．

n 番目の要素[*60]である場合は

```
"baz": <Collection {"src-dist": "baz.ttc", "index": n}>
```

と記述することでやはり `` `baz` `` という名前でそのフォントが使えるようになる．数式フォントについても**数式フォントハッシュファイル** (*math font hash file*)：

```
dist/hash/mathfonts.satysfi-hash
```

に設定を記載する．記述する形式はフォントハッシュファイルと同様である．

7.3. デフォルトのフォント設定

組版処理中で，特にフォント指定がない場合に使われるフォント[*61]は**デフォルトフォントハッシュファイル** (*default font hash file*)：

```
dist/hash/default-font.satysfi-hash
```

で設定されているものになる．デフォルトフォントハッシュファイルの内容は文字体系ごとに初期値とするフォントを指定したディクショナリであり，現状では文字体系として漢字，かな，ラテン文字，その他，の 4 つが定められている．勿論今後より多くの文字体系をサポートする想定だが，今のところは手前 3 つにあてはまらない Unicode コードポイントはその他扱いになる．デフォルトフォントハッシュファイルは，例えば以下のようにして記述する：

```
{
  "han-ideographic": {
    "font-name": "ipaexm",
    "ratio"    : 0.88,
    "rising"   : 0.0
  },
  "kana": {
    "font-name": "ipaexm",
    "ratio"    : 0.88,
    "rising"   : 0.0
  },
```

60 番号は 1-origin である．

61 `get-initial-context` が返すテキスト処理文脈に格納されているフォント設定の意．

```
  "latin": {
    "font-name": "Junicode",
    "ratio"    : 1.0,
    "rising"   : 0.0
  },
  "other-script": {
    "font-name": "Junicode",
    "ratio"    : 1.0,
    "rising"   : 0.0
  }
}
```

`ratio` エントリは拡大率，`rising` エントリは行方向に垂直な位置調整の指定である．詳しくは第 8 章で扱うが，SATYSFI ではフォントを規定するのに OpenType フォントファイルだけでなく拡大率とベースライン調整率という 2 つの値を用いる．上の設定では漢字と仮名を（ラテン文字との混植を想定して）0.88 倍の大きさに調整している例である．

なお，現状ではデフォルトの数式フォントは `` `lmodern` `` で固定されてしまっている．勿論，近い将来に変更可能なように拡張する想定である．

7.4. 分割ファイルの読み込み

ここでは `@require:` *foo* および `@import:` *foo* がどのようにファイルを読み込んでいるかを説明する．

SATYSFI は，文書ファイルが与えられるとまずそのファイルを「すでに見たファイルのリスト」である**既読リスト**に入れ，構文解析する．構文解析が（エラーを出さずに）終わると，そのファイルに書かれていた `@require:` *foo* および `@import:` *foo* をそれぞれ見る．`@require:` *foo* の場合は *LIBROOT*/`dist/packages/`*foo*(`.satyh` | `.satyg`)の形の候補から，`@import:` *foo* の場合はそれが書かれているファイルからの相対パスとしてパッケージファイルを見つけ，そのパスが既読リストに入っていれば何もせず，既読リストにまだ入っていないファイルならば既読リストに入れて構文解析する．そこにも `@require:` や `@import:` が書かれていたらその指し示す先のファイルを（既読リストに入っていなければ）また既読リストに入れて構文解析する．こうして再帰的にファイルを辿っていく．

この処理を行なう際，既読リストの構築と同時にファイル *foo* に `@require:` *bar* が書かれていたら「*foo* は *bar* に依存する」という依存関係も記録しておく．こうして，読んだファイルを頂点とし，どのファイルがどのファイルに依存しているかを辺で張った有向グラ

フができあがる．今度はこの有向グラフを用いて，列挙されたパッケージファイルを実際に読み込む順序を決定する処理に移る．このとき，グラフに有向閉路が見つかった場合，SATYSFI はパッケージ間に循環した依存があって依存解決できない旨のエラーを出して終了する．有向閉路がなかった場合，つまり依存関係が有向非巡回グラフ (DAG) だった場合は，トポロジカルソートを行なって依存順に並べる．なお，複数の依存解決の可能性がある場合，どれが選ばれるかには特に決まりがない．したがって，2 つのファイルのいずれを先に読み込むかによって結果が異なる場合は必ず依存関係を明示する必要がある．

こうして依存関係を解決して整列されたパッケージファイルたちは，後方から順に型検査を行なって評価するという処理を繰り返し，逐次読み込んでいく[*62]．

なお，以上の説明からわかるように，ひとつのファイル内で先頭に `@require:` や `@import:` をどのような順序で書いてもそれらは互いに区別されないので，特に気にする必要はない．

62 すべてのファイルの型検査を終えてからすべてのファイルを評価するという方式でもよいのだが，現在の実装では各ファイル単位で同時並行で型検査と評価を行なう．この実装上の選択には特に理由はないし，変更しても特に問題は生じない．

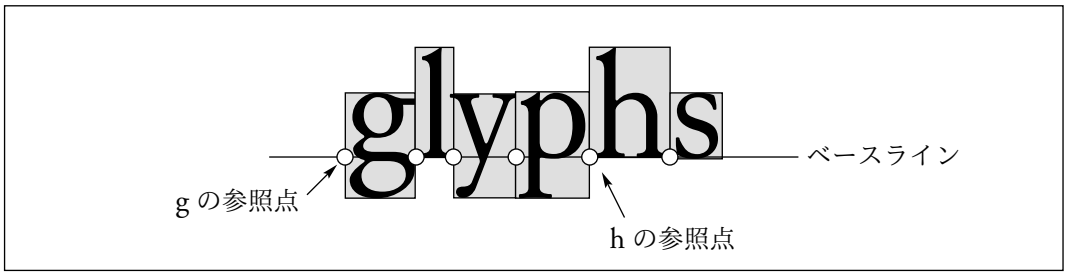
Chapter 8
文字組版

文字は組版処理の主役であり，そして最も人間くささゆえの複雑性を備えた組版処理対象だ．現在の SATYSF$_{\mathrm{I}}$ はひとまず和欧混植がそれらしくできることを志向して開発したためかなりアドホックな面も目立つのだが，ひとまず文字をどのように分類して扱っているかについてここで説明する．

8.1. 文字の扱いの基本

SATYSF$_{\mathrm{I}}$ に限らず，一般に横組みの文字組版（とりわけラテン文字の組版）に於いては文字を排列させる基準として**ベースライン** (*baseline*) という水平線が想定される．“文字に対応する形状” を**字形**または**グリフ** (*glyph*) と呼び，入力されたテキストデータからそれに対応づけられたグリフの列をベースライン上に一定の規則で排列させていくことを広く組版処理と言う．排列は，ベースライン上に**参照点** (*reference point*) と呼ばれる点を想定して行なわれる．フォントは（テキストを構成する）各文字に対して，その文字のグリフを参照点を基準にしてどんな輪廓線で描くかの情報と，その文字がどれだけの**字幅** (*advance width*) をもつかなどの情報を関連づけており，基本的には以下の図のように「或る参照点 p_n に基づいて文字 c_n のグリフ g_n を描いたら，p_n から文字 c_n の字幅 w_n だけ版面上で右に[*63]移動した位置を新たな参照点 p_{n+1} とし，その参照点から次の文字 c_{n+1} のグリフ g_{n+1} を描く」という処理を繰り返すことによりテキストをグリフの列として描画する，というモデルが採られている：

63 アラビア文字やヘブライ文字など RTL (right-to-left) な文字体系では「左に」だが，SATYSF$_{\mathrm{I}}$ が RTL の文字体系に未対応なこともあって，ここでは簡単のため考えないことにする．



ベースラインや参照点は SATYSFI では文字組版だけでなく第 9 章で説明するグラフィックスの取り扱いに関しても必要となる概念なので，よく意識されたい．

8.2. 文字の分類

SATYSFI は Unicode コードポイントに対して以下の方法でそれぞれ分類する：

- どの**文字体系** (*script*) に属するか
- “東アジア的観点から” **広い** (*wide*) 文字か**狭い** (*narrow*) 文字か
- どの**行分割クラス** (*line breaking class*) に属するか

文字体系は Unicode Script Property [33] により規定されているが， 現在の SATYSFI は `Hani`，`Kana`，`Latn` と「それ以外」の 4 種類を

```
type script = HanIdeographic | Kana | Latin | OtherScript
```

で定義されるような script 型で扱う． 勿論扱える文字体系は今後増やしていく想定であり，或いは script 型のように文字体系があらかじめ言語にハードコードされた方法ではない，より一般的な方法で文字体系を区別する方法に変更するかもしれない．

文字が “東アジア的観点から” 広いか狭いかは East Asian Width [31] に基づいて判断される．East Asian Width は東アジアのレガシーな文字表示に於いて各文字が “いわゆる「全角」 で表示されるものであるか， いわゆる「半角」 で表示されるものであるか” といったことを規定しているプロパティなのだが，SATYSFI ではこのプロパティを主に属する文字体系のない Unicode コードポイントを扱うのに有用な情報として転用している．East Asian Width は各 Unicode コードポイントを Fullwidth (F)，Wide (W)，Ambiguous (A)，Halfwidth (H)，Narrow (Na)，Neutral (N) の 6 種類のいずれかに分類しており，SATYSFI はこのうちの前 2 分類に該当する Unicode コードポイントを**広い** (*wide*) 文字， 後ろ 4 分類に該当するものを**狭い** (*narrow*) 文字として扱う．

残る分類方法は行分割クラスである．これは Unicode Line Breaking Algorithm [32] で定義された line breaking class とおおよそ一致する分類方法だが，『日本語組版の要件』[34]，通称 JLreq で定義されている**文字クラス**の要素を取り込んで独自に少し上書きしたものだ．すべて掲載すると長いので，Unicode Line Breaking Algorithm が定義する line breaking class との差分を以下に述べる：

- BK (Mandatory Break)，CR (Carriage Return)，LF (Line Feed)，NL (Next Line) を廃止し，代わりにいずれも INBR (Input Break) という新しい分類に統一する．
- JLreq が定義する文字クラス cl-01（始め括弧類），cl-02（終わり括弧類），cl-06（句点類），cl-07（読点類），cl-10（長音記号）に相当する JLOP (JL Open Punctuation)，JLCP (JL Close Punctuation)，JLFS (JL Full Stops)，JLCM (JL Commas)，JLPL (JL Prolonged Sound Mark) という分類をそれぞれ新たに用意する．
- 表 8.1 のとおりに上書きする．

また，これに伴って Unicode Line Breaking Algorithm も一部拡張する．具体的には

- LB13: $\times$ (CL | CP | EX | IS | SY | JLCP | JLCM | JLFS)
- LB14: (OP | JLOP) SP* $\times$
- LB16: (CL | CP) SP* $\times$ (NS | CJ | JLPL)
- LB21: $\times$ (BA | HY | NS | CJ | JLPL),　BB $\times$

と変更する．規則の読み方は [32] を参照されたい．なお，INBR クラスと SP クラスに該当する文字，つまり “改行文字” や “空白文字” は，行分割候補箇所を決定する前に以下に挙げるような規則に基づく正規化処理により除去される場合がある．すなわち，IDs という略記で(ID | CJ | IN | SA | JLOP | JLCP | JLFS | JLCM | JLPL)を表すとして，各 SP または INBR の文字は前後が以下のどのパターンに（先に挙げたものほど高い優先度で）マッチするかによって，除去するか，残すか，置換するか等の処理を決定する．

- 前方が IDs，後方が AL にそれぞれマッチする SP または INBR は除去する．
- 前方が AL，後方が IDs にそれぞれマッチする SP または INBR は除去する．
- 前方が IDs，後方が IDs にそれぞれマッチする INBR は除去する．
- 前方が IDs，後方が IDs にそれぞれマッチする SP は残す．
- 前方が IDs に合致する SP または INBR は除去する．

U+2018	LEFT SINGLE QUOTATION MARK	‘	OP
U+2019	RIGHT SINGLE QUOTATION MARK	’	CP
U+201B	SINGLE HIGH-REVERSED-9 QUOTATION MARK	‛	OP
U+201C	LEFT DOUBLE QUOTATION MARK	“	OP
U+201D	RIGHT DOUBLE QUOTATION MARK	”	CP
U+201F	DOUBLE HIGH-REVERSED-9 QUOTATION MARK	‟	OP
U+3001	IDEOGRAPHIC COMMA	、	JLCM
U+3002	IDEOGRAPHIC FULL STOP	。	JLFS
U+3008	LEFT ANGLE BRACKET	〈	JLOP
U+3009	RIGHT ANGLE BRACKET	〉	JLCP
U+300A	LEFT DOUBLE ANGLE BRACKET	《	JLOP
U+300B	RIGHT DOUBLE ANGLE BRACKET	》	JLCP
U+300C	LEFT CORNER BRACKET	「	JLOP
U+300D	RIGHT CORNER BRACKET	」	JLCP
U+300E	LEFT WHITE CORNER BRACKET	『	JLOP
U+300F	RIGHT WHITE CORNER BRACKET	』	JLCP
U+3010	LEFT BLACK LENTICULAR BRACKET	【	JLOP
U+3011	RIGHT BLACK LENTICULAR BRACKET	】	JLCP
U+3014	LEFT TORTOISE SHELL BRACKET	〔	JLOP
U+3015	RIGHT TORTOISE SHELL BRACKET	〕	JLCP
U+3016	LEFT WHITE LENTICULAR BRACKET	〖	JLOP
U+3017	RIGHT WHITE LENTICULAR BRACKET	〗	JLCP
U+3018	LEFT WHITE TORTOISE SHELL BRACKET	〘	JLOP
U+3019	RIGHT WHITE TORTOISE SHELL BRACKET	〙	JLCP
U+301A	LEFT WHITE SQUARE BRACKET	〚	JLOP
U+301B	RIGHT WHITE SQUARE BRACKET	〛	JLCP
U+30FC	KATAKANA-HIRAGARA PROLONGED SOUND MARK	ー	JLPL
U+FF08	FULLWIDTH LEFT PARENTHESIS	（	JLOP
U+FF09	FULLWIDTH RIGHT PARENTHESIS	）	JLCP
U+FF0C	FULLWIDTH COMMA	，	JLCM
U+FF0E	FULLWIDTH FULL STOP	．	JLFS
U+FF3B	FULLWIDTH LEFT SQUARE BRACKET	［	JLOP
U+FF3D	FULLWIDTH RIGHT SQUARE BRACKET	］	JLCP
U+FF5B	FULLWIDTH LEFT CURLY BRACKET	｛	JLOP
U+FF5D	FULLWIDTH RIGHT CURLY BRACKET	｝	JLCP
U+FF5F	FULLWIDTH LEFT WHITE PARENTHESIS	｟	JLOP
U+FF60	FULLWIDTH RIGHT WHITE PARENTHESIS	｠	JLCP

表 8.1 行分割クラスの上書き定義

- 後方が IDs に合致する SP または INBR は除去する．

- INBR は SP に置換する．

この正規化の処理は，和文の途中にある改行文字は特に何も書いていないのと同じだが，欧文の途中にある改行文字は単語間空白と扱われることが期待されている，という区別が必要であることを反映したものになっている．

8.3. フォントの扱い

SATYSF$_{\mathrm{I}}$ に於ける「フォント」を規定するのは，単に OpenType フォントファイル[*64]に関連づけられたフォント名 F ではなく，これに**拡大率** r と**ベースライン調整率** r' という 2 つの数値が備えつけられた 3 つ組 $\left(F, r, r'\right)$ である．フォントは文字体系ごとに別個に設定され，次のようなプリミティヴを用いる：

`set-font` : script → string ∗ float ∗ float → context → context

ctx `|> set-font` *script* (*fontname*, *r*, *rb*) で文字体系 *script* に対するフォント設定を (*fontname*, *r*, *rb*) に更新したテキスト処理文脈 *ctx* を返す．

`get-font` : context → script → string ∗ float ∗ float

`get-font` *script ctx* で *ctx* に格納されている *script* のフォント設定を取り出す．

拡大率が r に設定されている場合，テキスト処理文脈設定されている文字サイズを s としたとき，F のもつグリフやメトリックの情報は rs を 1em とするようなサイズで使われる．すなわち，$r < 1$ なら“標準的な大きさ”から縮小補正がかかり，$r > 1$ なら拡大補正，$r = 1$ ならばそのまま，ということである．さらに，ベースライン調整率が r' に設定されている場合，F のグリフは通常よりも $r's$ だけベースラインを持ち上げた位置に描かれる．すなわち，$r' = 0$ なら補正せず，$r' < 0$ ならば基準よりも下に，$r' > 0$ ならば上に補正される．これら拡大率とベースライン調整率は，ラテン文字と漢字などベースラインに対する捉え方の異なる文字体系をスムーズに共存させるための補正である．実際，ラテン文字と漢字で異なるフォントデータ F_1, F_2 を用いて組み，フォントサイズ s をそのまま F_1 と F_2 の両方で素直に 1em にすると，大抵の場合はラテン文字の方が小さく見えてしまう．そこで，通常は和文の方を小さくするように補正する．本書では $r_1 = 1$, $r_2 = 0.88$ という拡大率をラテン文字に使うフォントと漢字や仮名に使うフォントにそれぞれ設定している．なお，ベースライン調整率は両者ともに 0 にしている．和文フォントは和欧混植を見越し

64 TrueType Collection の場合も加味すると，正確には「フォントファイル内の要素のひとつ」．

てかもともとベースラインを仮想ボディの下端よりも上に設定しているものが多く，改めて補正する必要がなかったためである．

8.4. 異なる文字体系間の空白

一般に，ラテン文字と漢字など，異なる文字体系の文字が隣接する際には字間を空けないと詰まりすぎて見える．SATYSF$_{\mathrm{I}}$ は，互いに異なる文字体系に属する文字が隣接している箇所に自動でグルーを挿入する．このグルーの各長さに関する情報は単語間空白と同様にフォントサイズに対する比率でテキスト処理文脈が保持している．デフォルトでは `Latin` と `HanIdeographic` の間，`Latin` と `Kana` の間，およびそれらの前後逆の 4 種類の組み合わせに対して `(0.24, 0.08, 0.16)` が設定されているが，以下のプリミティヴによって変更できる：

`set-space-ratio-between-scripts` : script → script → float → float → float → context → context

ctx `|> set-space-ratio-between-scripts` $script_1$ $script_2$ r_0 r_1 r_2 で前方が文字体系 $script_1$ の文字，後方が文字体系 $script_2$ の文字の隣接箇所で自然長，収縮基準長，伸長基準長の文字サイズに対する比率がそれぞれ r_0，r_1，r_2 であるようなグルーを挿入するように *ctx* を更新したテキスト処理文脈を返す．

スペーシングに関してインラインボックス列を特定の文字体系のテキストに “擬態させる” 方法として，次のプリミティヴが使える：

`script-guard` : script → inline-boxes → inline-boxes

`script-guard` *script ib* でインラインボックス列 *ib* を前後のスペーシングに関して文字体系 *script* のテキストとして扱う．

8.5. 寡占的文字体系

文字の広い / 狭いの区別は，文字体系が特に定まっていない約物などの Unicode コードポイントを特定の文字体系へと “吸収” するのに用いる．SATYSF$_{\mathrm{I}}$ では，テキストは基本的に “代表となる文字体系” が意識されて書かれているという仮定のもと，テキスト処理文脈に “現在の寡占的な文字体系” の情報を保持させるようになっている．“寡占的な文字体系” は 2 つ設定することができ，ひとつは特定の文字体系に属さない広い文字をどの文字体系に “吸収” して扱うかで，もうひとつは特定の文字体系に属さない狭い文字をどの文字体系へと “吸収” して扱うかである．前者に設定された文字体系を**広い寡占的文字体系** (*dominant*

wide script)，後者を**狭い寡占的文字体系** (*dominant narrow script*) と呼ぶ．例えば和欧混植の場合，前者は `Kana`，後者は `Latin` に設定するとよい．こうすることで，いわゆる“全角用の”約物は仮名の仲間として扱われることになってフォントも仮名と揃えられ，一方で“アルファベット用の”約物はラテン文字の仲間として扱われ，フォントがラテン文字のものと揃えられることになる．このような設定に関与するのが以下のプリミティヴたちである：

`set-dominant-wide-script` : script → context → context

第 1 引数に与えられた文字体系を広い寡占的文字体系に設定する．

`get-dominant-wide-script` : context → script

広い寡占的文字体系に設定されている文字体系を取得する．

`set-dominant-narrow-script` : script → context → context

第 1 引数に与えられた文字体系を狭い寡占的文字体系に設定する．

`get-dominant-narrow-script` : context → script

狭い寡占的文字体系に設定されている文字体系を取得する．

以上の仕組みを知って「なんとアドホックな仕組みであることか」と驚かれる読者も多いかもしれない．正直なところ著者もそう思うが，一応現状では和欧混植に関してよく効力を発揮している．おそらく“より多言語な組版”を扱うには耐えない仕組みであろうから，この仕組みが将来の非互換な変更で大きく形を変える可能性は高い．

Chapter 9
グラフィックス

版面を構成する文字以外の要素で重要なのがグラフィックスである．本章では，本文の囲み枠，表の罫線，幾何図形，直交座標系のグラフ，グラフ理論のグラフ，可換図式，さらには数式中の“連続的に伸縮する”括弧といった多様な対象を描画する用途で活用できる SATYSFI のグラフィックスの機能について紹介する．一読すればわかるようにグラフィックス用のプリミティヴは非常にプリミティヴなのだが，そのプリミティヴさゆえに，思い立てばユーザの手によって非常に柔軟な描画が指定できる．また，頻出する図形や囲み枠の指定については `gr`，`deco`，`hdecoset`，`vdecoset` などのパッケージで提供されており，図形定義に活用できる．

9.1. パス

パス (*path*) は曲線や閉曲線およびそれが占める領域を定めた情報であり，グラフィックスを構成する最も基本的な単位だ．パスの構築は

- point 型： length $*$ length と同義，“2 次元座標” や “2 次元ベクトル” の型として使う

という型シノニムと

- pre-path 型： **未完パス** (*pre-path*)，すなわちまだ閉じたり終了したりしていない伸ばす途中のパスの型
- path 型：完成したパスの型

という 2 つの基本型を用いて，以下に挙げるようなプリミティヴで行なう．座標はページの左下を $(0, 0)$ とする絶対座標で，右向きが $+x$ 方向，上向きが $+y$ 方向である[*65]．これだけ聞くと絶対座標でパスの各点を指定することについてはあまりにもプリミティヴす

65 y 座標が上向きに伸びるのは違和感があるかもしれないが，これは PDF の仕様の “最も基本的な座標系” にそのまま従ったものだ．

ぎるのではないかと違和感を覚えるかもしれないが，以下で紹介するプリミティヴ `shift-path` を用いて実質的に相対座標を扱うこともできるのでひとまず心配しないで頂きたい．

`start-path` : point → pre-path

第 1 引数に指定した点から未完パスを開始する．

`line-to` : point → pre-path → pre-path

prepath `|> line-to` v で未完パス *prepath* を終点から点 v へと線分で延長したものを返す．

`bezier-to` : point → point → point → pre-path → pre-path

prepath `|> bezier-to` u_1 u_2 v で未完パス *prepath* を終点から点 v へと Bézier 曲線で延長したものを返す．u_1 と u_2 は制御点である．

`close-with-line` : pre-path → path

未完パスを受け取り，起点と終点を線分で結んで閉じてできるパスを返す．

`close-with-bezier` : point → point → pre-path → path

prepath `|> close-with-bezier` u_1 u_2 で未完パス *prepath* の起点と終点を制御点 u_1，u_2 の Bézier 曲線で結んで閉じてできるパスを返す．

`terminate-path` : pre-path → path

未完パスを受け取り，開いたままのパスとして返す．

`unite-path` : path → path → path

2 つのパスを統合して 1 つにする． これはドーナツ形など中空のパスをつくるのに必須である[*66]．

`shift-path` : point → path → path

path `|> shift-path` v でパス *path* の位置を v だけずらす．

66 点がパスの “内側” であるか否かは， その点を囲んでいるパスの回転数の総和の偶奇によって決まっている．

`get-path-bbox` : path → point ∗ point

パスを受け取り，そのパスのバウンディングボックスの左下と右上の座標の組を返す．すなわち，与えられたパスが占める領域のうち x 座標がとる最小値と y 座標がとる最小値の組 $\left(x_{\mathrm{min}}, y_{\mathrm{min}}\right)$ と，x 座標がとる最大値と y 座標がとる最大値の組 $\left(x_{\mathrm{max}}, y_{\mathrm{max}}\right)$ との組を返す．

パスの定義のたびに毎回上記のプリミティヴのみを用いて記述していてはあまりに非効率なので，典型的な図形については `gr` パッケージの `Gr` モジュールがより簡素なインターフェイスでパスを定義する方法を提供している．代表的なものをここに挙げる：

`Gr.line` : point → point → path

与えられた 2 点を結ぶ線分を返す．

`Gr.rectangle` : point → point → path

与えられた 2 点が対角線をなす，x 軸に平行な辺をもつ長方形を返す．

`Gr.rectangle-round` : length → point → point → path

角丸の長方形を返す．第 1 引数が角の半径の指定である．

`Gr.polygon` : point → point list → path

第 1 引数を開始点とし[*67]，第 2 引数の要素を順に辿って開始点に戻る多角形を返す．要するに `Gr.polygon` p `[`p_1`;` ⋯`;` p_n`]` は

```
start-path p |> line-to p1 |> ⋯ |> line-to pn |> close-with-line
```

と同じ．

`Gr.poly-line` : point → point list → path

第 1 引数を開始点とし，第 2 引数の要素を順に辿ってできる折れ線を返す．要する

67 パスは少なくとも 1 点を含むようにしか定義できないので，空リストが与えられないように開始点をリストの要素ではなく独立した引数として与えるようなインターフェイスになっている．

に `Gr.poly-line` p [p_1; …; p_n] は

```
start-path p |> line-to p1 |> … |> line-to pn |> terminate-path
```

と同じ．

`Gr.circle` : point → length → path

第 1 引数を中心とし，第 2 引数を半径とする円を返す[*68]．

9.2. グラフィックス指定

パス，つまり path 型の値は，単に曲線や閉曲線のデータ，さらにいえばどんな領域を占めているかの情報であり，それ自体が描けるものではない．実際にそのパスに基づいて文書中に図形を挿入するには，それがどんな色で塗られるかとか，あるいはどんな枠線で描かれるかといった情報を与える必要がある．パスに加えてそれらの情報を持っているのが graphics 型の値である．これは以下のプリミティヴを用いてつくる．

`fill` : color → path → graphics

`fill` *color path* でパス *path* の内側を色 *color* で塗ったグラフィックスを返す．

`stroke` : length → color → path → graphics

`stroke` *t color path* でパス *path* を幅 *t*，色 *color* の線として描いたグラフィックスを返す．

`draw-text` : point → inline-boxes → graphics

`draw-text` *v ib* で位置 *v* を参照点としてインラインボックス列 *ib* を置いたグラフィックスを返す．

`shift-graphics` : point → graphics → graphics

第 1 引数の分だけ第 2 引数のグラフィックスの座標をずらす．

68 実際に描画して視覚上の違和感を催すことはほぼないであろうが，この函数が返すのは 3 次 Bézier 曲線によって近似的に表現される円なので真円ではない．これは PDF 1.7 の仕様に円弧のパスを記述する直接的な方法が存在しないことに起因する．したがって，精密な製図を目的として使う場合は注意を要する，

`get-graphics-bbox` : graphics → point ∗ point

`get-path-bbox` と同じ要領で与えられたグラフィックスのバウンディングボックスを返す．ストロークの太さは考慮されないことに注意．

さて，グラフィックスの構成は上に掲げたプリミティヴを用いればよいが，実際に本文中にグラフィックスを挿入するにはどうすればいいのだろうか．このために用意されているのがプリミティヴ：

`inline-graphics` : length →
length → length → (point → graphics list) → inline-boxes

である．これは `inline-graphics` *w h d grf* という形で使うことでインラインボックス列を得る．最初の 3 つの引数は寸法の指定で，*w* が横幅，*h* が高さ，*d* が深さである．要するにグラフィックスは大きい 1 文字のようなものとして扱うのだが，どのような寸法の文字として扱って欲しいかを指定しているのである．そして重要なのは第 4 引数 *grf* で，これは座標を受け取ってグラフィックスのリストを返す函数である．SATYSF$_{\mathrm{I}}$ はまずグラフィックスの内容が確定していない段階で寸法だけを用いて（行分割処理やページ分割処理を経て）実際に版面のどの位置に配置されるかを決定し，位置が確定してからその参照点 (x, y) を指定されていた函数 *grf* に引数として渡す．そしてその返り値としてできたグラフィックスのリストをそこに描き込む，という仕組みになっている．

例 9.1　指定されたテキストが現在のフォントで占める領域と同じ範囲をバツ印つきの長方形の枠にして出力するコマンド，すなわち以下のような記述と組版結果を実現するコマンド `\area` を考える：

```
+p{ The quick brown fox jumps
      over the lazy dog. }
+p{ The quick \area{brown} fox jumps
      over the \area{lazy} dog. }
```

The quick brown fox jumps over the lazy dog.

The quick ☒ fox jumps over the ☒ dog.

このような `\area` は，次のように `inline-graphics` および `get-natural-metrics` を用いて定義することができる：

```
let-inline ctx \area it =
  let ib = read-inline ctx it in
  let (w, h, d) = get-natural-metrics ib in
  let thickness = 0.5pt in
  let color = Color.black in
    inline-graphics w h d (fun (x, y) ->
      [
        stroke thickness color
          (start-path (x, y -' d)
            |> line-to (x +' w, y -' d)
            |> line-to (x +' w, y +' h)
            |> line-to (x, y +' h)
            |> close-with-line);

        stroke thickness color
          (start-path (x, y -' d)
            |> line-to (x +' w, y +' h)
            |> terminate-path);

        stroke thickness color
          (start-path (x, y +' h)
            |> line-to (x +' w, y -' d)
            |> terminate-path);
      ]
    )
```

前節で紹介した gr パッケージの函数を使えば，`inline-graphics` の部分をもっと短く記述できる：

```
inline-graphics w h d (fun (x, y) ->
  List.map (stroke thickness color) [
    Gr.rectangle (x, y -' d) (x +' w, y +' h);
    Gr.line (x, y -' d) (x +' w, y +' h);
    Gr.line (x, y +' h) (x +' w, y -' d);
  ]
)
```

なお，上での説明からもわかるとおり，`inline-graphics` の第 4 引数 *grf* の“中身”が評価されるタイミングはページ分割処理が行なわれるときであり，注意を要する．ページ分割処理が始まる直前の，“文書のすべての字面上の内容”が確定するときまでは決して評価されないのである．したがって，グラフィックスの描画中に何らかの“副作用”を起こす処理は内容の決定には基本的に影響を及ぼさない．例えば

```
let-mutable flag <- false

let-inline ctx \test =
  inline-graphics 5cm 3cm 0cm (fun (x, y) -> (
    let () = flag <- true in
    [ fill Color.black
        (Gr.rectangle (x, y) (x +' 5cm, y +' 3cm)); ]
  ))

let-inline ctx \show-flag =
  if !flag then {T} else {F}
```

などという定義があり，ここ以外では可変参照 `flag` の内容は書き換えられないとする．この定義に基づいて文書中で 1 箇所 `\test;` を使い，その前後いたるところに `\show-flag;` を挿入しても，それらは全部「F」と出力されるはずである．`\show-flag;` が評価されるときにはまだ `\test` の定義中の `inline-graphics` の第 4 引数の“中身”は評価されておらず，したがって `flag` は依然として `false` を参照しているからだ．もし `\test` の定義が

```
let-inline ctx \test =
  let () = flag <- true in
  inline-graphics 5cm 3cm 0cm (fun (x, y) ->
    [ fill Color.black
        (Gr.rectangle (x, y) (x +' 5cm, y +' 3cm)); ]
  )
```

だったなら，`\test;` より手前に書かれた `\show-flag;` は「F」に，`\test;` より後方に書かれた `\show-flag;` は「T」になる．両者の定義の違いはしっかりと意識されたい．よくわからないうちは「基本的にグラフィックスを返す函数の中で“副作用”を起こそうとするべきではない」と認識していてもかまわない．

9.3. 囲み枠

行や段落を囲う枠も，グラフィックス機能を用いて創ることができる．余白の指定に

はpaddings 型を用いる．これは単に length ∗ length ∗ length ∗ length の型シノニムであり，paddings 型の値は左，右，上，下の順番でパディングの長さの指定として使われる．また，実際の描画の指定にはdeco 型が用いられる．これも point → length → length → length → graphics list の型シノニムであり，この型の値は要するに参照点，幅，高さ，深さの 4 引数を受け取り，それを使って描画内容を返す函数である．

`inline-frame-outer` : paddings → deco → inline-boxes → inline-boxes

`inline-frame-outer` *pads deco ib* でパディング指定 *pads*，描画指定 *deco*，内容 *ib* の枠で囲まれた部分をつくる．枠内の内容で行分割されることはないが，“外側の要請に合わせて” 内容に含まれるグルーが伸縮する．

`inline-frame-inner` : paddings → deco → inline-boxes → inline-boxes

`inline-frame-outer` と同様だが，枠内の内容は自然な長さで組まれる．すなわち，内容に含まれるグルーは自然長になる．

`inline-frame-fixed` : length → paddings → deco → inline-boxes → inline-boxes

`inline-frame-outer` および `inline-frame-inner` とよく似ているが，枠の幅を第 1 引数で指定する．この幅に合わせて内側の内容のグルーが伸縮する．

一般に，描画指定によってはパディングは “実際の枠と内容との距離” とはならないことに注意されたい．描画指定である deco 型の函数は，与えられた各引数を（一定の長さを足すなど）自由に加工して使うことができるからである．パディングはあくまで枠で囲まれた全体がどのようなバウンディングボックスをもっていると行分割処理で扱われるかの決定，および枠の描画指定である deco 型の函数にどのような値を第 2–4 引数として与えるかの計算のみに用いられる．

内容の途中で行分割できる枠も定義できる．枠の描画指定は，分割されないもの（独立形），手前は切れていないが後ろで切れているもの（先頭形），両方切れているもの（中途形），手前で切れていて後ろは切れていないもの（末尾形）の 4 種類をこの順番で指定する．deco-set 型はこのための型であり，deco ∗ deco ∗ deco ∗ deco 型の型シノニムである．

`inline-frame-breakable` : paddings → deco-set → inline-boxes → inline-boxes

途中で行分割できる枠で内容を囲む．枠は各行で与えられた 4 種類から該当するものが選択される．中身のグルーなどは `inline-frame-outer` と同様に行分割に合わせて伸縮する．

`inline-frame-breakable` の第 2 引数に与えられる deco-set 型の 4 つ組は，典型的にはそれぞれ「ロ」形，「匚」形，「ニ」形，「コ」形をしたグラフィックスを返すような函数である．

例 9.1　`hdecoset` パッケージの `HDecoSet` モジュールには，函数

`simple-frame-stroke` : length → color → deco-set

が以下のように定義されている：

```
let simple-frame-stroke t scolor =
  let strokef = stroke t scolor in
  let decoS (x, y) w h d =
    [ strokef (Gr.rectangle (x, y -' d) (x +' w, y +' h)); ]
  in
  let decoH (x, y) w h d =
    [ strokef (Gr.poly-line (x +' w, y -' d)
        [(x, y -' d); (x, y +' h); (x +' w, y +' h);]); ]
  in
  let decoM (x, y) w h d =
    [ strokef (Gr.line (x, y -' d) (x +' w, y -' d));
      strokef (Gr.line (x, y +' h) (x +' w, y +' h)); ]
  in
  let decoT (x, y) w h d =
    [ strokef (Gr.poly-line (x, y -' d)
        [(x +' w, y -' d); (x +' w, y +' h); (x, y +' h)]); ]
  in
    (decoS, decoH, decoM, decoT)
```

これは `simple-frame-stroke` *thk color* で太さ *thk*，色 *color* の線でできた，途中で行分割可能な枠の描画の指定になる．例えば

```
let-inline ctx \simple-frame it =
  let pads = (2pt, 2pt, 2pt, 2pt) in
  let decos = HDecoSet.simple-frame-stroke 1pt (Gray(0.5)) in
    inline-frame-breakable pads decos (read-inline ctx it)
```

で `\simple-frame` を定義すると，

与えられたインラインテキストを，左右上下 2pt ずつのパディングを設け，太さ 1pt，50% の灰色の線でできた，途中で行分

割できる簡素な枠で囲むインラインコマンド になる．

途中で行分割可能な枠の高さと深さ，つまり deco 型の函数に渡される第 3 引数と第 4 引数は，分割前ではなく分割後の内容にしたがって決められるので，上の例の組んだ結果から見てもわかるようにすべての行で揃うとは限らない．もしすべての行で枠の高さと深さを揃えたいならどうすればよいだろうか？　それ専用のプリミティヴを SATYSFI 言語が提供しなくとも，一応 `get-natural-metrics` と `set-every-word-break` などを用いて実現できる[*69]ので，練習として考えてみてほしい．

さて，ブロック方向の囲み枠はインライン方向の囲み枠に比べると幾分か複雑である．というのも，枠内の内容を組むにあたっては“段落の幅”を枠のためのパディングの分だけ狭くしなければならないからだ．以下で説明するように，これは“テキスト処理文脈に関する継続渡し”として実現している．

`block-frame-breakable` : context → paddings → deco-set → (context → block-boxes) → block-boxes

`block-frame-breakable` *ctx pads decos k* の形で使う．*k* はテキスト処理文脈を受け取ってブロックボックス列を返す継続であり，`block-frame-breakable` は第 1 引数として受け取ったテキスト処理文脈 *ctx* を第 2 引数として受け取ったパディングの 4 つ組 *pads* に基づいて段落幅の値だけ更新したものを継続 *k* に適用し，その結果のブロックボックス列を枠内の内容とする．枠は `inline-frame-breakable` と同様に独立形，先頭形，中途形，末尾形の 4 つ組である *decos* を受け取り，これをもとにページ分割時に描画する．独立形，先頭形，中途形，末尾形は，典型的にはそれぞれ「ロ」形，「冂」形，「| |」形，「凵」形である[*70]．インライン方向とは違い，ブロック方向には深さの概念がないので，枠の描画時に deco 型の値に渡される引数のうち深さにあたる第 4 引数の値は常に長さ 0 である．

以上のような仕組みによって，SATYSFI では囲み枠が実現されている．

---

69 「一応」はやはり例 6.1 で言及した問題と同様の問題による．

70 たとえ“同様な見た目のデザインの枠”であっても `inline-frame-breakable` と `block-frame-breakable` とで同一の deco-set 型の値を枠の指定に使いまわせるわけではないことに注意．インライン用とブロック用でグラフィックスを別々に定義しなければならない．

9.4. 表組版

表組版は，全組版処理システムの泣き所である．簡潔なインターフェイスにすると機能が貧相になり，かといって高い自由度を保とうとすると簡素な表を組みたい場合でもかなり記述量が多くなってしまったりする．現状の SATYSFI の表組版機能は，後者に振り切れたような形式化が行なわれている．まず，表を構成するセルのための型としてcell 型が用意されている．これは以下で定義できるような代数的データ型だ：

```
type cell =
  | EmptyCell
  | NormalCell of paddings * inline-boxes
  | MultiCell  of int * int * paddings * inline-boxes
```

`EmptyCell` は，何も内容のないセルを表す値である．`NormalCell(`*pads*`,` *ib*`)` は，パディング *pads*，内容 *ib* の単純なセルを表す．`MultiCell(…)` については後述する．実際に表を組むのは次のようなプリミティヴだ：

`tabular` : (cell list) list → (length list → length list → graphics list) → inline-boxes

`tabular` *cellss grf* の形で使う．第 1 引数はセルの行が列方向に並んだリストであり，`tabular` はこれを適切に幅を調節して表に組み上げる．*grf* は罫線を指定するための函数であり，この函数にはページ分割処理時に i 行 j 列の表を組んだ時はセルの境界となるような絶対座標の x 値が $(j+1)$ 個並んだリストと絶対座標の y 値が $(i+1)$ 個並んだリストが渡されてきて，そしてその結果返されるグラフィックスを `tabular` は罫線として表に加えて描画する．

`MultiCell(`i`,` j`,` *pads*`,` *ib*`)` は，パディングが *pads* で内容が *ib* なのは単純なセルと同様だが，その位置から（その位置を含めて）下に i セル分，右に j セル分の領域を占めるセルになる．要するに，表計算ソフトなどで見られるセルの結合に相当する処理が実現できるのがこの `MultiCell(…)` であり，重なる部分のセルの内容は上書きされる[*71]．

例 9.1　あまりにもソースコードの未整備感が凄まじいが，現時点でも機能上表組版が可能であることを一応コード例とともに掲げる．`table` パッケージが提供する

71 現在の実装では，`MultiCell(…)` が上書きする部分のセルが `EmptyCell` ではなかった場合は実行時エラーになる．

`\tabular` コマンドを使って，以下のように表を組むことができる：

```
+p{ \tabular(fun t -> (
  let (c, l, r) = (t#c, t#l, t#r) in
  let m = t#m in
  let e = t#e in
  let cA = c {case A} in
  let cB = c {case B} in
  [ [l {Program}; m 1 2 {Answer}; e; m 1 2 {Time [ms]}; e ];
    [e          ; cA     ; cB     ; cA       ; cB         ];
    [l {foldn}  ; c {Yes}; c {Yes}; r  {${4}}; r {${16}}  ];
    [l {mc91}   ; c {No} ; c {-}  ; r {${24}}; r {-}      ];
    [l {rev}    ; c {Yes}; c {Yes}; r  {${4}}; r {${78}}  ];
    [l {max}    ; c {Yes}; c {Yes}; r  {${8}}; r {${69}}  ];
  ]
))(fun xs ys -> (
  let thin = stroke 0.5pt Color.black in
  let thick = stroke 1pt Color.black in
    match (ys, List.reverse ys) with
    | (y0 :: y1 :: y2 :: _, ylast :: _) ->
        ( match (xs, List.reverse xs) with
          | (x0 :: x1 :: _ :: x3 :: _, xlast :: _) ->
              let gsY1 =
                [y1; y2] |> List.map (fun y ->
                  thin (Gr.line (x0, y) (xlast, y))) in
              let gsY2 =
                [y0; ylast] |> List.map (fun y ->
                  thick (Gr.line (x0, y) (xlast, y))) in
              let gsX =
                [x1; x3] |> List.map (fun x ->
                  thin (Gr.line (x, y0) (x, ylast))) in
              List.concat [
                [ thin (Gr.line (x0, y1) (x1, y2)) ];
                gsX; gsY1; gsY2;
              ]
```

```
          | _ -> []
        )
    | _ -> []
)); }
```

Program	Answer		Time [ms]	
	case A	case B	case A	case B
foldn	Yes	Yes	4	16
mc91	No	–	24	–
rev	Yes	Yes	4	78
max	Yes	Yes	8	69

9.5. 画像

まだ幾分か簡素であるものの，SATYSFI は（ベクタでない，ピクセルが四方に並んだ）画像を取り扱う方法も備えている．画像のデータがもつ型として image 型が用意されており，以下のようなプリミティヴを使うことができる：

`load-image` : string → image

与えられたパスの画像を読み込む．現在は色空間が CYMK でない JPEG のみ対応している．ファイルが存在しない場合は実行時エラーを出して終了する．

`load-pdf-image` : string → int → image

第 1 引数に与えられたパスの PDF を読み込み，第 2 引数に与えられたページ番号の内容を画像として返す．ページ番号は最初のページを 1 とする．与えられたファイルが存在しなかったり，与えられたページ番号のページが存在しない場合は実行時エラーを出して終了する．

`use-image-by-width` : image → length → inline-boxes

第 1 引数に与えられた画像を第 2 引数に与えられた長さを幅とする寸法でインラインボックス列として返す．

Chapter 10

数式組版

理工系の文書をしたためる者にとって，数式組版機能は当然のものとして組版処理システムに備わっていてほしいものだ．TEX/LATEX が 30 年の長きに亘って使われ続けてきた大きな要因のひとつが，誰もが納得するような品質の数式組版機能を古くから提供している数少ない組版処理システムだったからであることにおそらく異論はないだろう．SATYSFI も，実はそこそこの品質の数式組版機能を有している：

```
+p{ \eqn(
  let int1 = ${\int_0^a} in
  let int2 = ${\int_0^\sqrt{a^2 - x^2}} in
  let sqrt2 = ${\sqrt{x^2 + r^2}} in
    ${ \frac{#int1 x \ordd x #int2 r #sqrt2 \ordd r}{
         #int1 \ordd x #int2 r #sqrt2 \ordd r} = \frac{2a}{5} }
); }
```

$$\frac{\displaystyle\int_0^a x\,\mathrm{d}x \int_0^{\sqrt{a^2-x^2}} r\sqrt{x^2+r^2}\,\mathrm{d}r}{\displaystyle\int_0^a \mathrm{d}x \int_0^{\sqrt{a^2-x^2}} r\sqrt{x^2+r^2}\,\mathrm{d}r} = \frac{2a}{5}$$

本章では，SATYSFI に備わっている数式組版機能について，詳細を追って見ていくことにしよう．なお，繰り返しになるが，SATYSFI での数式の記述はコードの見た目上 LATEX にそっくりなものの，裏でやっていることは両者で全然違う．LATEX は字句解析（トークン化）に相当する処理を組版処理と同時進行で行ない，かつ構文木も一切構築せずパターンマッチでトークン列を書き換えていくのに対し，SATYSFI は静的に構文解析して数式を構文木の形で保持している．

10.1. 数式の構文と扱い

数式は基本的にプログラム中で `${ … }` という形で書かれ，math 型をもつ．数式中では大小ラテン文字と数字のほか，以下の記号類が通常の文字として直接書ける：

```
+ - * / : = < > ~ . , ? `
```

数式として書けるのは ${[me]^*} という形であり，*me* は以下のような構文定義に基づく：

$$
\begin{aligned}
me &::= ma\big(['']^* \mid _mg\hat{}mg \mid ['']^*\hat{}mg \mid ['']^*\hat{}mg_mg \mid ['']^*_mg\big) \\
ma &::= \big(ch \mid mcapp \mid \#x\big) \\
mg &::= \Big(ma \mid \{[me]^*\} \mid \{|\big[[me]^*|\big]^*\}\Big) \\
mcapp &::= mcmd\Big[\big(\{[me]^*\} \mid !(expr) \mid !\{It\} \mid !\langle Bt\rangle\big)\Big]^*
\end{aligned}
$$

ただしここで *ch* は数式中で文字として使えるもの 1 文字，*mcmd* は数式コマンド（インラインコマンドと同一範疇のトークン），*expr* は式，*It* はインラインテキスト，*Bt* はブロックテキストを，*x* は変数をそれぞれ動く．`#x` は変数 *x* の内容である数式をそこに埋め込む機能の記法だが，セミコロンは末尾につけないことに注意．また，*mcapp* は数式コマンドの適用だが，引数の種類に式引数とインライン引数の差異のようなものはなく，すべて一律に扱われ，こちらもいかなる場合も（引数がない場合も）末尾にセミコロンをつけない．なお，数式中に数式コマンドの引数としてインラインテキストが出現しうるし，インラインテキストの中でも当然式引数または文中数式として数式が登場するので，`{ … }` の括弧が入り乱れて少しややこしいかもしれないが，気持ちとしては“`${}` の中では `{ … }` が数式扱い，`!{ … }` の中では再び `{ … }` がインラインテキスト扱いに戻る”という構造になっている．数式コマンドの定義は，次のような構文で行なう：

```
let-math mcmd x_1 … x_n = expr
```

インラインコマンドやブロックコマンドと違い，第 0 引数はとらない．各変数 x_i に τ_i がつく下で式 *expr* に math 型がつくとき，この数式コマンド *mcmd* が定義されて以下のような型をもつと定められている：

$$[\tau_1;\ \tau_2;\ \cdots;\ \tau_n]\ \text{math-cmd}$$

例えば型の導出規則 $\Gamma \vdash e : \tau$ を `\tj{\Gamma}{e}{\tau}` という具合に書くためのコマンド `\tj` : [math; math; math] math-cmd は以下のようにして定義できる：

```
let-math \tj mG me mty = ${#mG \vdash #me : #mty}
```

math 型の値はインラインテキストに近いもので，単に数式の“構造”を保持しているだけである[*72]．数式を“実際に組み上げる”のは次のプリミティヴが担う：

`embed-math` : context → math → inline-boxes

これを使った例を挙げてみる．別行立て数式を記述するコマンド `\eqn` の実装の根幹は，次のようになっている：

```
let-inline ctx \eqn math =
  inline-fil ++ embed-block-breakable ctx
    (line-break true true ctx
      (inline-fil ++ (embed-math ctx math) ++ inline-fil))
```

`embed-block-breakable` については第 6 章で説明したとおりである．`embed-math` が使う数式フォントの情報はテキスト処理文脈が保持しており，数式フォントの変更には以下のプリミティヴを用いる．

`set-math-font` : string → context → context

数式フォントには拡大率とベースライン調整率の概念はなく，フォント名だけを与える．

10.2. 文中数式の扱い

第 2 章でも説明したように，数式 `${ … }` はインラインテキスト中にいきなり書いても使えるのであった．これがどのような仕組みで実現されているのかというと，実は文中数式は実行時に何らかの [math] inline-cmd 型のコマンド `\math` が補われており，単に `${ … }` と書いたものは `\math($${ … });` と扱われるのである．この文中数式用コマンド

72 ただし“演算子がどのように結合しているか”というような情報は持っていない．1 つの“グループ” `{ … }` 内で文字は単に一列に並んでいるだけで，“構造”として保持されているのはあくまで指数・添字・分数といったものである．SATYSFI での数式の定式化はいわば“指数・添字・分数などが正体の隠されたコンストラクタで，構築のみができて分解ができない代数的データ型”といった具合だ．

`\math` はテキスト処理文脈に保持されており，インライン変換時に補われる．文中数式用コマンドを変えたい場合は

`set-math-command` : [math] inline-cmd → context → context

というプリミティヴを使い，*ctx* `|> set-math-command (command \math)` のようにコマンドを渡す．`(command …)` は“コマンドそのもの”を受け渡しするための特殊な構文だが，現在のところ `set-math-command` と第 12 章で述べる `get-initial-context` くらいでしか用いる必要がない．

10.3. 数式を構築するプリミティヴ

この節では，数式コマンドの定義に使える，数式を構築するためのプリミティヴを紹介する．用例は `math` パッケージの実装を見ると豊富にあるので参照されたい．

`math-frac` : math → math → math

第 1 引数を分子，第 2 引数を分母とする分数を返す．

`math-lower`, `math-upper` : math → math → math

`math-lower` m_1 m_2 で数式 m_1 の下に添字の大きさで数式 m_2 を組む．`math-upper` は上．

`math-radical` : math option → math → math

第 2 引数に渡された数式に根号をつける．第 1 引数は根号の左肩に乗せる指数がある場合は `Some( … )` で与え，必要ない場合は `None` で与えるつもりで用意したものだが，現在の実装では `None` の場合しか対応しておらず，`Some( … )` を渡すと実行時エラーになる．また根号のグリフの形も（フォントのデータを用いるのではなく）決まった形しか出力できない．将来的には `math-paren` のように根号をグラフィックスで指定できるようにする想定である．

`math-color` : color → math → math

`math-color` *color m* で数式 *m* に文字色を *color* にする指定をつけて返す．

ところで，数式を構成する“アトム”たちには，スペーシングのために属性が与えられる．それが以下で定義されるような代数的データ型，math-class 型である：

```
type math-class = MathOrd | MathBin | MathRel | MathOp | MathPunct
                | MathOpen | MathClose | MathPrefix | MathInner
```

各コンストラクタはそれぞれ通常，2 項演算子，関係子，作用子，パンクチュエーション，開き括弧類，閉じ括弧類，接頭類，部分式という属性に対応する．属性に基づくスペーシングは基本的に TEX による処理と同様になるようにしているので，どのようにスペーシングが行なわれるかについては [4] や [14] を参照されたい．ただし，新しい属性がひとつだけあり，`MathPrefix` すなわち接頭類アトムである．これは $\mathrm{d}x$ や ∂x といった，手前に通常アトムや閉じ括弧類アトムなどがきた場合に自動でスペースを挿入するための属性である．ともあれ，この math-class 型の関わるプリミティヴを見てみる：

`get-left-math-class`, `get-right-math-class` : context → math → math-class option

第 1 引数のテキスト処理文脈に基づき，第 2 引数の数式の左端と右端がそれぞれどのような属性の “アトム” になっているか返す．数式が空の場合は `None` が返る．

`math-group` : math-class → math-class → math → math

`math-group` cls_1 cls_2 m で数式 m 内は通常どおりのスペーシングで組むが，m 全体を左側のスペーシングに関しては属性 cls_1 を，右側のスペーシングに関しては属性 cls_2 をもっているかのように扱わせる．

`math-pull-in-scripts` : math-class → math-class → (math option → math option → math) → math

添字と指数がつけられたときに，“それらを内側に取り込んで使える” 数式要素を作成する機能である．`math-pull-in-scripts` cls_1 cls_2 k の形で使う．左側のスペーシングに関しては属性 cls_1 を，右側のスペーシングに関しては属性 cls_2 をもっているかのように扱わせる．実際の “中身の数式” は，使われた箇所で

- 添字も指数もつかなかった場合は k `None None`
- 添字 m_1 だけついた場合は k `(Some(`m_1`)) None`
- 指数 m_2 だけついた場合は k `None (Some(`m_2`))`
- 添字 m_1 も指数 m_2 もついた場合は k `(Some(`m_1`)) (Some(`m_2`))`

を評価した結果の数式とする．

`text-in-math` : math-class → (context → inline-boxes) → math

`text-in-math` *cls k* の形で使い，属性 *cls* の“アトム”として *k* が返すインラインボックス列を数式中に埋め込む．*k* に渡されるテキスト処理文脈は“数式を組む際に `embed-math` に渡されたテキスト処理文脈”である[*73]．

例 10.1 “指数と添字がつけられたときにそれぞれ上下につけかえる”ような `\lim` は以下のようにして定義できる．

```
let-math \lim =
  let m-lim = math-char MathOp `lim` in
    math-pull-in-scripts MathOp MathOp (fun moptS moptT -> (
      match (moptS, moptT) with
      | (None    , None    ) -> m-lim
      | (Some(mS), None    ) -> math-lower m-lim mS
      | (None    , Some(mT)) -> math-upper m-lim mT
      | (Some(mS), Some(mT)) -> math-upper (math-lower m-lim mS) mT
    ))
```

実際に以下のように添字が下につけかえられる．

```
+p{ ${\lim_{n \to \infty} a_n} }
```

$$\lim_{n\to\infty} a_n$$

10.4. 文字の扱いと文字クラス

SATYSFI の数式で使う文字には**文字クラス**という概念が伴う．これはイタリックやボールドローマン，スクリプト体，黒板太字といった分類であり，SATYSFI が数式を組む際には実際に数式中に書かれている文字をどの文字クラスに基づいてグリフに変換するかを“文

73 正確には保持している文字色が `math-color` で指定された文字色に更新されていたりするので少し違う．

字クラスの指定”に基づいて行なう．文字クラスは次のように定義される代数的データ型，math-char-class 型によって扱われる：

```
type math-char-class =
  | MathItalic | MathBoldItalic | MathRoman | MathBoldRoman | MathScript
  | MathBoldScript | MathFraktur | MathBoldFraktur | MathDoubleStruck
```

`math-char`, `math-big-char` : math-class → string → math

`math-char` *cls s* で文字列 *s* を属性 *cls* の“アトム”にしたものを返す．これは“使われた場所”での文字クラス指定によらず常に同一の文字列を返す．`math-big-char` は“大型演算子”向けで，OpenType の MATH テーブルに基づいて大きいグリフを選択する．

`math-char-class` : math-char-class → math → math

`math-char-class` *cls m* で，数式 *m* を文字クラス *cls* で組むことを指定する．

`math-variant-char` : math-class → ⟨italic : string; bold-italic : string;
roman : string; bold-roman : string; script : string; bold-script : string;
fraktur : string; bold-fraktur : string; double-struck : string⟩ → math

各文字クラスが指定されている下でどのような文字列に変換するか個別に指定した文字の“アトム”を返す．

例 10.1　LaTeX での同名コマンドに対応する `\mathrm` と `\bm` は

```
let-math \mathrm m = math-char-class MathRoman m
let-math \bm m = math-char-class BoldItalic m
```

で定義できる．実際に使用すると以下のとおり：

```
+p{ ${R, \mathrm{R}, \bm{R}} }
```

$R, \mathrm{R}, \boldsymbol{R}$

例 10.2　文字クラスに基づいて自動で変換文字列を切り替えるギリシア大文字は，次のように定義できる：

```
let single cp = string-unexplode [cp]

let greek-uppercase cp cpb cpr cprb =
  let s  = single cp  in let sb  = single cpb  in
  let sr = single cpr in let srb = single cprb in
    math-variant-char MathOrd (|
      italic        = s;  bold-italic   = sb;
      roman         = sr; bold-roman    = srb;
      script        = s;  bold-script   = sb;
      fraktur       = s;  bold-fraktur  = sb;
      double-struck = s; |)

let-math \Gamma = greek-uppercase 0x1D6E4 0x1D71E 0x0393 0x1D6AA
```

最後の行の 4 つの十六進数リテラルは，それぞれイタリック，ボールドイタリック，ローマン，ボールドローマンの数式用ガンマが割り当てられている Unicode コードポイントに対応する．

10.5. 括弧類

SATYSFI に於ける数式の，縦幅可変の括弧の扱いはかなり大胆な方針を採っている．括弧はグラフィックスで定義されるのである．以下で登場する paren 型は length → length → length → length → color → inline-boxes ∗ (length → length) という型のシノニムである．paren 型の函数は “括弧の片側の描き方” に相当するもので，内容の高さ，内容の深さ，軸の高さ，文字サイズ，文字色をこの順に受け取り，片側の括弧に相当するブロックボックス列と，添字・指数をカーニングするための函数との組を返す．カーニング用の函数はベースラインを 0 とし上を正方向とする “y 座標を受け取り，その高さに添字や指数がつくとき，どれだけカーニングを施せばよいかの長さを返す（正の長さを返すと添字・指数が括弧により近づく）．これは，例えば丸括弧の場合，括弧の形に沿うようなカーニング用の函数を右側の括弧に指定することで，$\left(\frac{A}{\sqrt{1+x^2}}+b\right)^2$ ではなく $\left(\frac{A}{\sqrt{1+x^2}}+b\right)^2$ のように自動で調整された位置に添字・指数をつけてくれる方が数式把握上の誤解をきたしにくく望ましいだろう，という動機から設けられている機能である．

`math-paren` : paren → paren → math → math

`math-paren` *paren*$_1$ *paren*$_2$ *m* で *paren*$_1$ を左括弧，*paren*$_2$ を右括弧として数式 *m* を囲んだものを返す．両 *paren*$_i$ に渡される第 1 引数と第 2 引数は *m* を組み上げたときの高さと深さである．

`math-paren-with-middle` : paren → paren → paren → math list → math

`math-paren-with-middle` *paren*$_1$ *paren*$_2$ *paren*$_3$ *ms* の形で使う．*paren*$_1$ を左括弧，*paren*$_2$ を右括弧として使うのは `math-paren` と同様だが，第 3 引数の括弧 *paren*$_3$ で第 4 引数の数式のリストを区切って括弧内に配置する点が異なっている．$\left\langle \mathrm{e}^{ik'x} \middle| \frac{A}{a} \middle| \mathrm{e}^{ikx} \right\rangle$ など，“区切り用の縦棒が左右の括弧とともに伸びる” ような数式の構成要素を組むのに有用である．

“連続的に大きさの変わる” 括弧のグラフィックスを定義するのは非常に骨の折れる作業だが，多くの入力に対して良いグリフを出力できる括弧の定義を与えられた時の満足はひとしおである．`math` パッケージには著者がそんな風に地道に定義した paren 型のデータがいくつもあり，例えば丸括弧の左右はそれぞれ `Math.paren-left` および `Math.paren-right` で使うことができる．なお，`math` コマンドの括弧は，入れ子になると自動で少しずつ大きくなる機能がついている．尤も，不必要に外側の括弧を大きくすることは数式組版上不適切な処理と判断されうるので [14]，利用時には留意されたい：

```
+p{ ${\paren{a + \paren{b + \paren{c + \paren{d + e}}}}} }
```

$$\Big(a + \Big(b + \Big(c + (d + e)\Big)\Big)\Big)$$

例 10.1　左半開区間のコマンドは，次のようにして定義できる：

```
let-math \opcls m1 m2 =
  math-paren Math.paren-left Math.sqbracket-right ${#m1, #m2}
```

```
+p{ ${\opcls{0}{1}, \opcls{\frac{1}{2^n}}{1}} }
```

$$(0, 1], \left(\frac{1}{2^n}, 1\right]$$

Chapter 11
相互参照

相互参照 (*cross reference*) とは，章節や図表の番号といった“内容そのものに関係なく割り当てられ，かつ文書中の他の場所から指し示すときに使われる”情報を自動的に取り扱う機能である．第 1 章でも述べたように，相互参照はマークアップ方式の組版処理システムが（WYSIWYG 方式のそれに比べて）簡潔に実現することを得意としやすい機能だ．

SATYSFI での相互参照の実現方法は LaTeX でのそれと基本的に同様であり，“1 回目の処理で相互参照用の ID とその内容を紐づけて外部ファイルに記録しておき，2 回目の処理で補完する”方式をとっている．ここで相互参照の ID とそれに紐づけられた（番号などの）内容を記録しておく外部ファイルが第 2 章で登場し即座に無視されていた *foo*`.satysfi-aux`，すなわち**補助ファイル** (*auxiliary file*) なのである．なお，上で述べた“1 回目で記録して 2 回目で補完する”をより正確に言えば“毎回組版処理が始まるときに（既に存在すれば）補助ファイルを読み込み，相互参照の ID と内容の紐づけが不動点に達するまで組版処理を繰り返す”である[*74]．SATYSFI を使用している際に，型検査を通った文書の処理でしばしば

```
needs another trial for solving cross references...
evaluating texts (2nd trial) ...
```

などとログを出して最初からもう一度組版処理を始め，やがて

```
all cross references were solved.
```

74 相互参照の紐づけがいつまでも不動点に到達しないような意地悪な記述をすることも（残念ながら）できる．この意地悪な挙動を型システムなどの言語機能によって排除することは原理的に難しい．SATYSFI は，相互参照の紐づけが不動点に達していなくても，あらかじめ定められた打ち切り回数に到達したら

```
could not reach to fixpoint when resolving cross references.
```

とログを表示して終了する．打ち切り回数は 4 回である（今のところこれを変更する方法は提供していない）．

などと報告して（正常に PDF を出力して）終了するのに気づいた方も多いだろうが，これは相互参照の紐づけが不動点に達するまで組版処理を繰り返しているのである．

実現方法からもわかるように，相互参照はかなり原始的な定式化によって成り立っている．これはひとえに相互参照を必ず “1 回の処理” で整合させるのが本質的に困難なことに起因する．“1 回の処理” で整合させる本質的な困難さを感じ取れる大雑把な例を以下に挙げてみよう：　相互参照で或る章節のページ番号を（「p ページの第 $m.n$ 節を見よ」の p として）挿入したいとする．そのページ番号がいくつなのかはページ分割をするまでわからないから，ページ番号としてどのような内容を本文中に挿入したらよいのか事前には決められない．`inline-graphics` と同様の要領で内容をあらかじめ未定にしておいてページ分割後に内容を決定してはどうかと考えるかもしれないが，ページ番号の内容がグリフ列として版面上どれだけの幅を占めるかさえ事前にはわからない．このように，内容がページ分割結果に依存する処理は（当然ながらどこでページ分割するかも内容に依存しているという逆向きの依存もあるので）“1 回の処理” で整合させることが一般にきわめて困難なのである．

仕組みが原始的なだけあって，相互参照機能を実現しているプリミティヴは非常に単純だ．以下の 2 種類のみである：

`register-cross-reference` : string → string → unit

`register-cross-reference` *key value* で相互参照の ID *key* から内容 *value* への紐づけを登録する．既に *key* から何らかの内容への紐づけが登録されていた場合は上書きされ，また古い内容と新しい内容が文字列として異なっていた場合は相互参照が不動点に達していないという理由で SATYSFI 処理系に再処理を促す．*key* からの紐づけが元々登録されていなかった場合も同様に再処理を促す．

`get-cross-reference` : string → string option

`get-cross-reference` *key* という形で使い，既に *key* に紐づけられた何らかの内容 *value* が登録されている場合は `Some(`*value*`)` が返される．*key* に紐づけられた内容が登録されていない場合は `None` を返す．

なお，登録されていない相互参照 ID *key* が `get-cross-reference` によって取得を試みられ，そのまま相互参照が不動点に達して終了した場合，SATYSFI は

```
some cross references were not solved:
```

という文言とともにそのような *key* を列挙して報告する．

Chapter 12
ページ分割処理

ページ分割は，おそらく SATYSFI の機構の中でも最も理解が難しい部分だろう．処理の概要自体はブロックボックス列になった文書の全内容を適切な長さで切り分けて各ページの内容にするというだけなのだが，まともな文書をつくるにはページ分割に伴って種々の手続き的処理を行なう必要があり，したがってクラスファイルを開発するには避けて通れない道である．まず，ページ分割処理に相当するプリミティヴは

`page-break` : page $\to \tau_2 \to \tau_3 \to$ block-boxes $\to$ document

である．τ_2 と τ_3 はやや複雑なため後述するが，これらの型がつく第 2・第 3 引数を置いておけば `page-break` のインターフェイスは単純極まりないもので，第 1 引数は紙面のサイズ，第 4 引数はページ分割対象となるブロックボックス列である．page 型は

```
type page =
  | A0Paper | A1Paper | A2Paper | A3Paper | A4Paper | A5Paper
  | USLetter | USLegal
  | UserDefinedPaper of length * length
```

で定義されるようなデータ型で，`A4Paper` などのコンストラクタがそれぞれ何を表すのかは想像のとおりである[*75]．要するに，ひとまず最も大雑把に言えば，`page-break` とは第 4 引数に与えられたブロックボックス列を第 1 引数に与えられた紙面の大きさに基づいてページ分割を行なうプリミティヴ，ということになる．ただし，これだけでは説明とし

---

75 正直なところ紙面のサイズ指定は length $*$ length 型で十分で，`A4Paper` などに相当する値は

`Paper.a4` : length $*$ length

のようにパッケージレヴェルで与えればよかったのだが，過去の阿呆な著者はこんな形式化をしてしまった．おそらく最も近い将来の非互換な変更でこの page 型は姿を消すだろう．

てあまりにも語弊が大きいので，以降順次詳細に説明する．

まず注目すべきは `page-break` の返り値の型 document だ．この型は `page-break` 以外のプリミティヴには返り値の型として一切現れず，したがって document 型の値を構築することは本質的に `page-break` によってしかできないようになっている．SATYSFI は文書ファイルにつく型が document 型であることを型検査時に要請するので，クラスファイルの奥底には必ず `page-break` があって[*76]，そこでユーザが与えた種々の情報をもとに組み上げられたブロックボックス列にページ分割処理を施すための記述が為されている．`stdjareport` クラスに基づいて文書を作成するときに書いていた

```
@require: stdjareport

document (|
  （文書情報）
|) '<
  （内容）
>
```

の `document` は実は単なる函数であり，上の記述は `document` という函数の適用である．`document` 函数は

$$\forall\alpha :: \langle\!|\,\text{title} : \text{inline-text};\ \text{author} : \text{inline-text}\,|\!\rangle.\ \alpha \to \text{block-text} \to \text{document}$$

という型をもつので，タイトルおよび著者を記述したレコードを第 1 引数に，文書内容のブロックテキストを第 2 引数に受け取り，最終的に document 型の値を返すということになっているのである．

さて，後回しにしていた τ_2 の話である．これは以下のような型だ：

$$\langle\!|\,\text{page-number} : \text{int}\,|\!\rangle \to \langle\!|\,\text{text-height} : \text{length};\ \text{text-origin} : \text{point}\,|\!\rangle$$

ここから，ひとまず `page-break` に与える第 2 引数はレコードを受け取ってレコードを返す函数であることがわかる．この函数は，ページ分割処理の最中に $(n-1)$ ページまでの内容が確定し，今まさに残りのブロックボックス列から n ページ目の内容を切り分ける箇所を探し始めるというときに `(| page-number = n; |)` というレコードが渡されることで呼

76 ユーザに直接 `page-break` を書かせるような病的なクラスファイルなら `page-break` を含まないかもしれないが，そんなクラスファイルをつくるような必要はほぼない．

び出され，`(| text-height = h; text-origin = (x, y); |)` という形の値を返すことで

- h：　n ページ目の内容のテキストの領域の段落方向の長さ
- (x, y)：　実際に切り分けられた内容を配置するときの左上の座標

という情報をページ分割処理を担う機構に教える．ただし，グラフィックスの座標系とは異なり，ここでの (x, y) の y は紙面の左上を原点とし下向きを $+y$ 方向とする座標系に基づく値なので注意されたい．どのページも同じ地点から同じ長さだけ本文が記述されているような文書を出力したいなら n によらず同一の h と (x, y) を返すような函数にすればよいし，ページの左右で（綴じしろなどの都合で）余白を変えたい場合は n の偶奇で異なる値を返すような函数にすればよいだろう．

同様に，`page-break` の第 3 引数につく型である τ_3 についても述べると，以下のようなものである：

$$(\!|\text{page-number} : \text{int}|\!) \to (\!|\text{header-content} : \text{block-boxes};\ \text{header-origin} : \text{point};\ \text{footer-content} : \text{block-boxes};\ \text{footer-origin} : \text{point}|\!)$$

やはりレコードを受け取ってレコードを返す函数の型である．`page-break` の第 3 引数に指定された函数は，第 n ページの内容が確定し，後述するようなページ内の“フック処理”がすべて終了したときに `(| page-number = n; |)` というレコードが渡されることで呼び出され，

```
(|
  header-content = bb1; header-origin = (x1, y1);
  footer-content = bb2; footer-origin = (x2, y2);
 |)
```

という形のレコードを返すことで

- (x_1, y_1) を左上としてブロックボックス列 bb_1 をヘッダとして描く
- (x_2, y_2) を左上としてブロックボックス列 bb_2 をフッタとして描く

という情報を与える．上で説明した第 2 引数と同じく，ここで返す座標も左上を原点とし下向きを $+y$ 方向とする座標系であるので注意されたい．この座標系を用いるのは `page-break` の第 2・第 3 引数のみである．

以上がプリミティヴ `page-break` の仕様なのだが，ほんの少しつけ加えるとすれば，それは“実際に行分割が行なわれるタイミング”についてである．実は“`page-break` 自体がページ分割処理をしている”わけではなく，文書全体が document 型の値として返された時，初めてページ分割処理が起動する．言うなれば，document 型の値とは最終的なページ分割に関わる情報を実際にページ分割を行なわずに“凍結”したものである．したがって，最終的に返される document 型の値以外に document 型の値をつくるような処理を書いたとしても，それが実際にページ分割されることはない．そもそも document 型の値を加工するようなプリミティヴは一切ないので，PDF に出力する内容を返す以外で `page-break` によって document 型の値をつくる意味はなく，意味があるとすればそれは返り値を使わずにページ分割時の“副作用”だけ起こすことを意図したものであろうが，それもできないようになっているのである[*77]．

12.1. ページ分割に関連するプリミティヴ

ページ分割に関する処理を行なうためのプリミティヴがいくつかあるので，ここで紹介する．

`clear-page` : block-boxes

強制的に改ページする．ページ分割される対象ではない箇所に挿入されている場合は `block-nil` と等価である．

`add-footnote` : block-boxes → inline-boxes

`add-footnote` *bb* でブロックボックス列 *bb* を脚註として挿入する．返り値は inline-boxes 型で，この返り値のインラインボックス列を挿入した場所に関連づけられてページ分割処理時に脚註の挿入されるページが決まる．インラインボックス列としての効能自体は `inline-nil` と同等である．

`hook-page-break` : $\left(\left(\!\left|\text{page-number} : \text{int}\right|\!\right) \to \text{point} \to \text{unit}\right) \to \text{inline-boxes}$

`hook-page-break` f の形で使う．返り値はインラインボックス列であり，組版上の効能は `inline-nil` と同等なのだが，ページ分割処理の最中にこれが挿入されている箇所がどのページに含まれるか確定したときに“フック函数”である f が

77 できないようになっている理由は特にないので，必要があれば将来の非互換な変更で仕様を変え，“`page-break` 自体がページ分割処理をする”ようにしてもよいかもしれない．

起動される．同一ページ内に複数の“フック函数”が仕込まれている場合，ページ内容上で手前にあるものから順に起動される．この“フック”函数の潜んだインラインボックス列がページ分割の結果 n ページ目に含まれることになり，その参照点が (x, y) であったとき，f には第 1 引数として `(| page-number = n; |)` が，第 2 引数として `(x, y)` が渡されることで起動される．そしてそれらの引数の値を用いてこの f の“函数の中身”でページ番号に基づく相互参照の紐づけを新たに登録したりといった様々な“副作用”が起こせる，というわけである．

第 9 章で説明した「グラフィックスの評価はページ分割時に行なわれる」という話は，実はこの `hook-page-break` と同様である．`inline-graphics` の第 4 引数など，参照点を受け取ってグラフィックスを返す類いの函数は，`hook-page-break` による“フック函数”と同じタイミングで評価が起動するのである．

12.2. クラスファイルの基礎

さて，ページ分割処理について把握したところでクラスファイルを創ろうと軽く思い立ったとき，ふと今まで特に意識していなかったことに気づくかもしれない：“最も外側の”テキスト処理文脈はどこから来るのだろう？ これはごく単純で，次のようなプリミティヴによって受け取れる：

`get-initial-context` : length → [math] inline-cmd → context

デフォルトのテキスト処理文脈を返す．第 1 引数は段落幅を，第 2 引数はインラインテキスト中に現れた数式に適用するコマンドを `set-math-command` と同じ要領で与える．

段落幅を指定できるのはこの `get-initial-context` のみである．“まともな”クラスファイルの中にはこの `get-initial-context` が必ずどこかにあり，これが返したテキスト処理文脈を適宜加工して“最も外側の”テキスト処理文脈としているのである．

例 12.1 “多少なりともまともに使える，実質的に最小のクラスファイル”は以下のように定義できる：

```
let-inline ctx \math m =
  script-guard Latin (embed-math ctx m)

let-block ctx +p it =
  line-break true true ctx
```

```
    (read-inline ctx it ++ inline-fil)

let document bt =
  let ctx =
    get-initial-context 440pt (command \math)
      |> set-dominant-narrow-script Latin
  in
  let bb = read-block ctx bt in
  page-break A4Paper
    (fun _ -> (|
      text-origin = (80pt, 100pt);
      text-height = 630pt;
    |))
    (fun _ -> (|
      header-origin  = (0pt, 0pt);
      header-content = block-nil;
      footer-origin  = (0pt, 0pt);
      footer-content = block-nil;
    |))
    bb
```

このクラスファイルの下では，ユーザが新たにコマンドを定義しない限り，`+p` を用いてひたすら段落が並べられたブロックテキストを `document` の引数に渡した文書しか記述することができない．

Chapter 13
Future Work

SATYSFI は現時点でも充分実用に耐える程度に機能を備えているが，今後さらに拡張していきたいと考えている点がいくつもある．本章ではこれについて簡潔に述べることとしよう．端的に言ってしまえば開発に協力してくださるかもしれない方への喧伝である．

13.1. 機能の拡張

13.1.1. テキストモード

テキストモード (*text mode*) は，SATYSFI で書かれた文書を HTML や LaTeX といったテキストファイルへとトランスパイルするためのモードである．SATYSFI の前身である，拙作の Macrodown という言語が担っていた処理の，形式化を改めた再来ともいえる．テキストモードでは，PDF を出力する通常のモードである PDF モード (*PDF mode*) には備わっていた inline-boxes などの型や `read-inline` などのプリミティヴはなく，インラインテキストは `read-inline` の代わりに用意されたプリミティヴ `stringify-inline` により文字列化される．一応すでに `--text-mode` というオプションで起動して試用できるが，数式には対応していない．将来的には数式も MathML へトランスパイルしたり実際に（PDF モードと同様に）組版処理を施して SVG として出力したりする機能を備えつけることを想定している．

13.1.2. 文字組版機能の拡張

縦書き，LTR/RTL 混植　縦書きは少なくとも和文組版では根強い需要があるし，ラテン文字とアラビア文字など LTR な文字体系と RTL な文字体系の混植も多言語組版では避けることのできない道だ．特に縦書きはプリミティヴの設計など言語のインターフェイスを大幅に拡張する必要が見込まれる．

2 段組み　論文組版で特に需要がある．インターフェイスの設計や，脚註や浮動する図表な

どと共存させた処理の実装がやや難しい．

OpenType フィーチャの ON/OFF 機能　OpenType フォントの GSUB や GPOS といったテーブルには発展的な文字組版を実現するための種々のデータが格納されており，現在の SATYSFI は OpenType フィーチャのうち合字処理のために liga テーブルを，カーニング処理のために kern テーブルを， ダイアクリティカルマークの処理のために mark テーブルと mkmk テーブルを，このほか数式フォントのために ssty テーブルなどを読み出すのだが，パッケージレヴェルでこれらのフィーチャの有効・無効を切り替えるプリミティヴを提供していないので，例えば jp90 (JIS90 Forms) などのフィーチャを有効にして漢字字形を切り替えるなどの高度な調整はまだできない．将来的にこれを可能にしたいと考えている．

合字やカーニング処理の言語ごとの切り替え　合字処理やカーニング処理は現在ラテン文字のみに対応しているが，これを他の文字体系に拡張し，かつ言語ごとに切り替えられるようにしたい．実装自体は単純だが，インターフェイスをどう簡潔にするかが多少非自明である．

ウィドウ，オーファン対応　段落の最後の行だけ分割されて次のページの先頭に組まれて見栄えが悪いものを**ウィドウ** (*widow*)，段落の最後の行で行長に対してかなり短いため見栄えが悪いものを**オーファン** (*orphan*) とそれぞれ呼ぶ[*78]．これらを防ぐ（何らかのペナルティを与えて出現を抑制する）機構が現在の SATYSFI には備わっていない．

数式の alignment tab　LaTeX の align 環境などでは数式の揃え位置を決定するのに alignment tab という概念が備わっており， デフォルトでは & という文字がこれの記述を担う．SATYSFI にはこれが備わっていないため，`\align` コマンドなどでは揃える場所で切った数式のリストを渡すというぎこちない仕組みを採ってしまっている．SATYSFI にも alignment tab に相当する機能を追加し，これをより自然なインターフェイスにしたい．

ページ分割できる表組版　現在の SATYSFI はインラインボックス列としての表しか組めないが，ブロックボックス列版でかつ途中でページ分割できる表を組む機能を追加したいと考えている．

行長が変わる段落　途中で行長が変わるような段落の組版は，途中で図版を回り込むよう

---

78 何をウィドウやオーファンと呼ぶかには多少揺れがあるようだが，「短すぎる段落最終行」「直後で改ページされた段落の最初の行」「直前で改ページされた段落の最後の行」のいずれかではある．

にテキストを組んだりするのに便利なのだが，計算量上効率的な実現がかなり難しい．TEX では `\parshape` というコマンドにより実現されているが，これは「何行目をどれだけの長さにする」という指定方法を採っており，行分割後の各行に含まれる内容の高さと深さによっては段落全体が所望の形にはならず縦に伸びたりしてしまう．何らかの方法で段落を（矩形とは限らない，途中で幅が変わるような）図形に収めるように行分割する処理をアルゴリズムとして確立したい．

13.1.3. 言語機能やシステム上の拡張

シグネチャ，多重モジュール　シグネチャに別名をつけたり，モジュールを多重化できるようにモジュールシステムを拡張したい．

型クラスまたはファンクタ　現状の $\mathrm{SATYSF_I}$ はいわゆるオーヴァーロードができないため，記述が複雑になってしまうことがよくある．アドホック多相を型安全に実現するため，Haskell 等で根強い支持を確立している**型クラス** (*typeclass*) を導入したいと考えている．或いは ML 系言語で盛んに使われている**ファンクタ** (*functor*)[*79]の仕組みを導入してもよいかもしれない．

パッケージマネージャ　現代的な計算機言語では，多くのパッケージをインストールしたり用いたりするのにパッケージマネージャは必須である．

レコードに関する型システムの拡張　現在の $\mathrm{SATYSF_I}$ は [23] に基づくレコード計算の型システムを備えているが，より柔軟な型システムへと拡張できないかと思案中である．場合によっては既に OCaml などに取り込まれている**列多相** (*row polymorphism*) [29] や，或いはさらに強力な拡張 [24] に切り替えることも検討するかもしれない．

末尾再帰最適化　計算上高いパフォーマンスを求められる用途ではないため，$\mathrm{SATYSF_I}$ は特に組版処理前に顕著な最適化をしない．実際（行分割処理など内部的な処理ではなく）パッケージレヴェルで記述された処理はほとんどが入力に対して線型で済むような処理である．しかし，実用上 PEG 構文解析器など複雑な処理を要する場面もあり，或る程度基本的な最適化を行なう必要性を感じつつある．その最たるものが末尾再帰最適化である．

CFF 系 OpenType フォントの部分埋め込み　OpenType フォントには，大別して

---

79 ここでいうファンクタとは“モジュールをモジュールに写す函数のようなもの”であり，Haskell などでファンクタと呼ばれているものとは全く関係がないことに注意．

- `glyf` や `loca` などのテーブルによってグリフ情報が格納されている TrueType アウトライン系のもの
- `CFF` テーブルなどによってそれらの情報が格納されている CFF アウトライン系のもの

の 2 種類がある．SATYSFI は `otfm` [3] という著者が大幅に拡張した OpenType フォントデコーダによって自前でフォントをデコードして使っている．部分埋め込みも自前で行なっているのだが，TrueType 系の OpenType フォントには対応しているものの CFF 系の OpenType フォントはまだ部分埋め込みできない．そのため，現状では

- CFF アウトラインをもつ非 TrueType Collection の OpenType フォントは全部埋め込みされ，出力される PDF の容量はかなり大きくなる
- CFF アウトラインをもつ TrueType Collection の OpenType フォントは（PDF の仕様上全部埋め込みできないため）使用できない

という状況である．`CFF` テーブルの部分埋め込みを実装し，これを改善したい．

画像読み込みの GIF，PNG 等対応　プリミティヴ `load-image` は（色空間が CMYK 以外の）JPEG のみに対応しているが，これを色空間が CMYK の JPEG や GIF，PNG 等に拡張したい．

タグつき PDF の出力　PDF 1.7 には，PDF ファイルに“マークアップ情報”を埋め込む機能がある．これの出力に関するプリミティヴを実装したい．

相対パスの扱い，プリプロセッサ　現在の SATYSFI の実装に於いて，`load-image` などのパスを文字列として受け取るプリミティヴは，相対パスが渡されたときには SATYSFI が起動されたカレントディレクトリからの相対パスとして解釈される．これは文書を構成する（システムにインストールされたパッケージ以外の）ファイルが複数のディレクトリに跨がる場合には特に著しく不便な性質だ．これを改善するため，相対パスを静的に絶対パスへと変換できるようなプリプロセスを何らかの形で導入したい．さらにいえば，長期的な目標ではあるが，より一般性の高い，OCaml でいう PPX に相当するようなプリプロセッサ機能を導入したいとも考えている．

テキスト処理文脈が保持する情報を拡張する機能　現状ではテキスト処理文脈が保持できる情報は事前に決められており，ユーザやパッケージ開発者が増設することはできない．これを可能にする言語機能を提供したい．具体的には

```
context-entry entry-name : type = default
```

といった構文でテキスト処理文脈が新たにもつ“エントリー”を宣言でき，実際にその情報に `ctx##`*entry-name* でアクセスしたり，`update-entry` という新しい予約語の字句を使った構文 *ctx* `|> update-entry ##`*entry-name v* で更新したりできるようにしたい．

13.2. 既存機能の破壊的な変更

テキスト処理　ひとまずそれらしく和欧混植ができるようにすることを目標として実装したため，文字組版に関してかなりアドホックな点が多い．現在いくらか見受けられる奇妙な形式化を改め，かつ多言語組版への対応を志向し，より一般性の高いテキスト処理を追求したい．

プリミティヴのインターフェイス変更　`line-break` が段落の上下でページ分割してよいかを第 1 引数と第 2 引数で受け取るという形式をとっていることや，段落の上下のマージンを `set-paragraph-margin` を介して設定する方式になっていることなど，現状の SATYSFI にはエレガントでもなければ実用上便利でもないようなインターフェイスをもつプリミティヴがいくらかある．互換性を損なうためタイミングには注意を要するが，可能であればこれらの機構はよりスマートな形式化に改めたい．

graphic と graphic list　グラフィックスに関する型の現状の形式化は，graphics list 型と graphics 型の“立場的な違い”がほとんどない．これらを graphics 型に統一し，

`concat-graphics` : graphics list → graphics

のようなプリミティヴを追加することを検討したい．

Chapter 14
言語仕様

この章は，対象をより細かい粒度で理解しようとせずにはいられない，物好きな読者向けに用意した．全貌を形式的に記述するには SATYSF$_{\mathrm{I}}$ は大きすぎるので，根幹となる部分だけを抜き出したサブセットを定義し，それについて操作的意味論や型システムをフォーマルに記述することを試みた．

14.1. テキストの扱いの形式的記述

14.1.1. 構文

この節では，SATYSF$_{\mathrm{I}}$ に於けるテキストやボックス列の扱いを形式的に記述した，ごく小さなサブセットの言語を与える．項 e，値 v，インラインテキスト要素 it，ブロックテキスト要素 bt は以下の BNF でそれぞれ定義される：

$$
\begin{aligned}
e &::= v \mid x \mid e\ e \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \mathbf{read^{I}}\ e\ e \mid \mathbf{read^{B}}\ e\ e \\
v &::= c \mid \boldsymbol{\lambda} x.\ e \mid \boldsymbol{\lambda}^{\mathbf{I}} x.\ e \mid \boldsymbol{\lambda}^{\mathbf{B}} x.\ e \mid ctx \mid \left\{[it]^{*}\right\} \mid \left\langle [bt]^{*} \right\rangle \mid \left\{\!\!\left\{ [ib]^{*} \right\}\!\!\right\} \mid \left\langle\!\!\left\langle [bb]^{*} \right\rangle\!\!\right\rangle \\
it &::= ch \mid e\ [e]^{*} \\
bt &::= e\ [e]^{*}
\end{aligned}
$$

ここでメタな記法 $[M]^{*}$ は 0 回以上の有限個の M の連接を表し，x は変数を，c は定数やプリミティヴを，ctx はテキスト処理文脈を，ch は Unicode コードポイントなどテキストの要素としての文字をそれぞれ動くとする．ib は形式的には定義を与えないが，インラインボックス列の要素，すなわちどんなサイズのどんなフォントで組まれるかといった情報を含む“既に組めるようになっているテキスト”の要素にあたる．イメージとしては inline の

i とか n などひとつひとつが ib に該当する[*80]. bb も同様に，形式的定義は与えないが“すでに組版処理をするための情報がすべて入っている，段落中の 1 行を表すデータ”におおよそ相当する．

インラインテキスト要素およびブロックテキスト要素の $e_0\ e_1\ \cdots\ e_n$ はコマンドの適用である．実際の SATYSF$_{\mathrm{I}}$ ではコマンドにあたる e_0 の部分は簡単のためコマンド名しか構文的に受けつけないようになっているが，ここでは定式化の都合上一般の項が書けるものとする．$\left(\mathbf{read^I}\ e_1\ e_2\right)$ は `read-inline` e_1 e_2 に相当し，同様に $\left(\mathbf{read^B}\ e_1\ e_2\right)$ も `read-block` e_1 e_2 に相当する．簡単のため部分適用できないようになっているが，

$$\mathbf{let}\ r\ x\ y = \mathbf{read^I}\ x\ y$$

などと定義して r を使えば部分適用できるので気にしなくてよい[*81]. $\left(\boldsymbol{\lambda}^{\mathbf{I}} x.\ e\right)$ は“インラインコマンド抽象”で，x はテキスト処理文脈を受け取るための変数である．計算上の振舞いは通常のラムダ抽象 $(\boldsymbol{\lambda} x.\ e)$ と全く同様の直観で理解できるものだが，型つけの都合上区別されている．$\left(\boldsymbol{\lambda}^{\mathbf{I}} x.\ e\right)$ 自体を直接書く方法は SATYSF$_{\mathrm{I}}$ の処理系からは提供されておらず，“インラインコマンド抽象”は必ず `let-inline` によってコマンド名に束縛する必要がある[*82]. `let-inline` によるコマンド定義の構文：

```
let-inline x cmd y1 ··· yn = e1 in e2
```

は以下の糖衣構文である：

$$\mathbf{let}\ cmd = \boldsymbol{\lambda}^{\mathbf{I}} x.\ \boldsymbol{\lambda} y_1.\ \ldots\ \boldsymbol{\lambda} y_n.\ e_1\ \mathbf{in}\ e_2$$

---

80 実際にはカーニングや合字の処理，さらにはハイフネーション辞書に基づくソフトハイフンの挿入などの処理があるため，“1 文字 1 文字を別個にインラインテキストからインラインボックス列に変換している”わけではないが，ひとまずの近似的な理解としてはこれで十分である．

81 実際のプリミティヴの `read-inline` も，正確にはこの r に対応する．

82 不必要な複雑化を避ける意図がある．一応コマンド *cmd* に束縛されている“インラインコマンド抽象”を `(command` *cmd*`)` で取り出すことはできるものの，`get-initial-context` や `set-math-command` などのプリミティヴに引数として渡す以外の本質的な用途はない．

“ブロックコマンド抽象” についても同様に，

```
let-block x cmd y1 ··· yn = e1 in e2
```

は以下の糖衣構文である：

$$\textbf{let}\ cmd = \boldsymbol{\lambda}^{\mathbf{B}} x.\ \boldsymbol{\lambda} y_1.\ \ldots\ \boldsymbol{\lambda} y_n.\ e_1\ \textbf{in}\ e_2$$

最初に掲げた BNF を見てもしかすると意外に感じたかもしれないが，インラインテキスト $\{It\}$ とブロックテキスト $\langle Bt \rangle$ はそれ自体（単に項なのではなく）値である．すなわち，プログラムの評価中に $\{It\}$ という形が出てきたら，（たとえ中にインラインコマンドが残っている状態でも）それ以上評価されずに函数やコマンドに渡されたりする．これは SATYSF$_{\mathrm{I}}$ 言語を理解する上で非常に重要な点である．詳しくは次節の操作的意味論の際に説明する．

14.1.2. 操作的意味論

操作的意味論はいわゆる big step semantics で定式化する．まず，項 e が値 v へと評価されるという 2 項関係 $e \downarrow v$ は次のように定義される：

$$\frac{}{v \downarrow v}\text{(E-Val)} \qquad \frac{e_1 \downarrow \boldsymbol{\lambda} x.\ e \qquad e_2 \downarrow v_2 \qquad [v_2/x]e \downarrow v}{e_1\ e_2 \downarrow v}\text{(E-App)}$$

$$\frac{e_1 \downarrow v_1 \qquad [v_1/x]e_2 \downarrow v_2}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \downarrow v_2}\text{(E-LetIn)}$$

$$\frac{e_1 \downarrow ctx \qquad e_2 \downarrow \{It\} \qquad ctx \vdash^{\mathbf{I}} It \downarrow Ib}{\textbf{read}^{\mathbf{I}}\ e_1\ e_2 \downarrow \{\!|Ib|\!\}}\text{(E-ReadI)}$$

$$\frac{e_1 \downarrow ctx \qquad e_2 \downarrow \langle Bt \rangle \qquad ctx \vdash^{\mathbf{B}} Bt \downarrow Bb}{\textbf{read}^{\mathbf{B}}\ e_1\ e_2 \downarrow \langle\!|Bb|\!\rangle}\text{(E-ReadB)}$$

インラインテキストの評価は以下のように定められる．すなわち，テキスト処理文脈 ctx の下でインラインテキスト It がインラインボックス列 Ib へ評価されるという 3 項関係 $ctx \vdash^{\mathbf{I}} It \downarrow Ib$ は以下のように定義される：

$$\frac{}{ctx \vdash^{\mathbf{I}} \varepsilon \downarrow \varepsilon}\text{(EI-Empty)} \qquad \frac{ib := \mathrm{Typeface}(ctx, ch) \qquad ctx \vdash^{\mathbf{I}} It \downarrow Ib}{ctx \vdash^{\mathbf{I}} ch \cdot It \downarrow ib \cdot Ib}\text{(EI-Char)}$$

$$\frac{e_0 \downarrow \boldsymbol{\lambda}^{\mathbf{I}} x.\ e \qquad \big([ctx/x]e\big)\ e_1\ \cdots\ e_n \downarrow \{\!|Ib'|\!\} \qquad ctx \vdash^{\mathbf{I}} It \downarrow Ib}{ctx \vdash^{\mathbf{I}} \big(e_0\ e_1\ \cdots\ e_n\big) \cdot It \downarrow Ib' \cdot Ib}\text{(EI-Command)}$$

ここで Typeface(ctx, ch) はテキスト処理文脈 ctx（の文字サイズやフォントといった情報）に基づいて文字 ch を“組める状態にしたもの”を表す．ブロックテキストについても，テキスト処理文脈 ctx の下でブロックテキスト Bt がブロックボックス列 Bb へ評価されるという 3 項関係 $ctx \vdash^{\mathbf{B}} Bt \downarrow Bb$ が同様に以下のように定義される：

$$\frac{}{ctx \vdash^{\mathbf{B}} \varepsilon \downarrow \varepsilon}\text{(EB-Empty)}$$

$$\frac{e_0 \downarrow \boldsymbol{\lambda}^{\mathbf{B}} x.\ e \qquad \big([ctx/x]e\big)\ e_1\ \cdots\ e_n \downarrow \langle\!|Bb'|\!\rangle \qquad ctx \vdash^{\mathbf{B}} Bt \downarrow Bb}{ctx \vdash^{\mathbf{B}} \big(e_0\ e_1\ \cdots\ e_n\big) \cdot Bt \downarrow Bb' \cdot Bb}\text{(EB-Command)}$$

注目すべき評価規則は (E-ReadI) だ．この規則では，まず第 1 引数のテキスト処理文脈が評価され，続いて第 2 引数として渡された項 e_2 が評価されインラインテキスト $\{It\}$ となる．前節で述べたように，一般にこのインラインテキストの中にはコマンドの適用が含まれている．これらのコマンド適用の評価を“起動する”のがまさに $\big(\mathbf{read}^{\mathbf{I}}\ \text{- -}\big)$ の仕事なのである．(E-ReadB) についても，同様のことが言える．

実際にはテキスト処理文脈 ctx の保持する情報を更新するプリミティヴや，テキスト処理文脈と $\{\!|Ib|\!\}$ の形の項（つまりインラインボックス列）を受け取って適切に行分割し $\langle\!|Bb|\!\rangle$ の形の項（つまりブロックボックス列）を返すようなプリミティヴが備わっていて，これによって“意味のある”処理が記述できるわけだが，核としては上記のような体系が $\mathrm{SATySF_I}$ の基礎をなしている．

14.1.3. 型システム

前節で定義した計算体系上に型システムを与える．実際の $\mathrm{SATySF_I}$ の型システムはそれなりに巨大で，SML# と同様のレコード [23] やラベルなしのオプション引数を扱えるようになっているが，一旦それらを除いて，テキストとボックス列に関わる根幹となる部分だけを抜き出して扱うことにする．基本型 b，単相型 τ，多相型 σ は以下の BNF でそれぞれ定義される：

$$\begin{aligned}
b &::= \text{context} \mid \text{inline-text} \mid \text{block-text} \mid \text{inline-boxes} \mid \text{block-boxes} \mid \cdots \\
\tau &::= \alpha \mid b \mid \tau \to \tau \mid \big[[\tau]^*\big]\ \text{inline-cmd} \mid \big[[\tau]^*\big]\ \text{block-cmd} \\
\sigma &::= \tau \mid \forall \alpha.\ \sigma
\end{aligned}$$

ただし，α は型変数を動くとする．型環境 Γ の下で e に型 τ がつくという型判定の 3 項関係 $\Gamma \vdash e : \tau$ は以下の規則で与えられる：

$$\frac{\Delta(c) \equiv \tau}{\Gamma \vdash c : \tau}\text{(T-Const)} \qquad \frac{\Gamma(x) \geqslant \tau}{\Gamma \vdash x : \tau}\text{(T-Var)} \qquad \frac{\Gamma\big[x \mapsto \tau'\big] \vdash e : \tau}{\Gamma \vdash \boldsymbol{\lambda} x.\ e : \tau' \to \tau}\text{(T-Lam)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma\big[x \mapsto \mathrm{Gen}\big(\Gamma, \tau_1\big)\big] \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\text{(T-LetIn)}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\text{(T-App)} \qquad \frac{}{\Gamma \vdash ctx : \text{context}}\text{(T-Context)}$$

$$\frac{\Gamma \vdash e_1 : \text{context} \qquad \Gamma \vdash e_2 : \text{inline-text}}{\Gamma \vdash \mathbf{read^I}\ e_1\ e_2 : \text{inline-boxes}}\text{(T-ReadI)}$$

$$\frac{\Gamma \vdash e_1 : \text{context} \qquad \Gamma \vdash e_2 : \text{block-text}}{\Gamma \vdash \mathbf{read^B}\ e_1\ e_2 : \text{block-boxes}}\text{(T-ReadB)}$$

$$\frac{}{\Gamma \vdash \{\!|Ib|\!\} : \text{inline-boxes}}\text{(T-BoxI)} \qquad \frac{}{\Gamma \vdash \langle\!|Bb|\!\rangle : \text{block-boxes}}\text{(T-BoxB)}$$

$$\frac{\Gamma \vdash^{\mathbf{I}} It}{\Gamma \vdash \{It\} : \text{inline-text}}\text{(T-TextI)} \qquad \frac{\Gamma \vdash^{\mathbf{B}} Bt}{\Gamma \vdash \langle Bt \rangle : \text{block-text}}\text{(T-TextB)}$$

$$\frac{\Gamma[x \mapsto \text{context}] \vdash e : \tau_1 \to \cdots \to \tau_n \to \text{inline-boxes}}{\Gamma \vdash \boldsymbol{\lambda}^{\mathbf{I}} x.\ e : \big[\tau_1; \cdots; \tau_n\big]\ \text{inline-cmd}}\text{(T-LamI)}$$

$$\frac{\Gamma[x \mapsto \text{context}] \vdash e : \tau_1 \to \cdots \to \tau_n \to \text{block-boxes}}{\Gamma \vdash \boldsymbol{\lambda}^{\mathbf{B}} x.\ e : \big[\tau_1; \cdots; \tau_n\big]\ \text{block-cmd}}\text{(T-LamB)}$$

ただし Δ は定数をその型に写す写像，$\sigma \geqslant \tau$ は単相型 τ が多相型 σ のインスタンスであること，$\mathrm{Gen}\big(\Gamma, \tau\big)$ は単相型 τ の型環境 Γ に基づいた閉包である多相型をそれぞれ表す．インラインテキストおよびブロックテキストに対する型つけ規則は以下で与えられる：

$$\frac{}{\Gamma \vdash^{\mathbf{I}} \varepsilon}\text{(TI-Empty)} \qquad \frac{\Gamma \vdash^{\mathbf{I}} It}{\Gamma \vdash^{\mathbf{I}} ch \cdot It}\text{(TI-Char)} \qquad \frac{}{\Gamma \vdash^{\mathbf{B}} \varepsilon}\text{(TB-Empty)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : \big[\tau_1; \cdots; \tau_n\big]\ \text{inline-cmd} \\ \Gamma \vdash e_i : \tau_i \quad (\text{for each } i) \qquad \Gamma \vdash^{\mathbf{I}} It\end{array}}{\Gamma \vdash^{\mathbf{I}} \big(e_0\ e_1\ \cdots\ e_n\big) \cdot It}\text{(TI-Command)}$$

$$\frac{\Gamma \vdash e_0 : [\tau_1; \cdots; \tau_n] \text{ block-cmd} \quad \Gamma \vdash e_i : \tau_i \ \ (\text{for each } i) \qquad \Gamma \vdash^{\mathbf{B}} Bt}{\Gamma \vdash^{\mathbf{B}} (e_0\ e_1\ \cdots\ e_n) \cdot Bt} \text{(TB-Command)}$$

14.1.4. 考察

インラインテキストやブロックテキストが値であり，$(\mathbf{read^I}\ \text{- -})$ や $(\mathbf{read^B}\ \text{- -})$ によってはじめてその中身の評価が起動するという形式化は，幾分不自然に感じられたかもしれない．どうしてこんな一見回りくどい方式を採っているのだろう？　実は，これは自動組版処理で行なわれる“本質的に手続き的な処理”に起因する．次のような例を考えると納得がいくかもしれない：　仮に SATYSF$_{\mathrm{I}}$ がテキストの評価も通常のプログラムの式と同じように“値呼び戦略風の”評価順序で評価されるような何らかの操作的意味論をもつ言語だったとしよう．この仮定の下で，以下のテキストを評価することを考える：

```
+section{外側}<
  +subsection{内側}<
    （略）
  >
  （略）
>
```

このとき，`+subsection` の適用は，`+section` の適用よりも先に評価されることになる．これは章節番号を自動で挿入する処理とは非常に相性が悪い．このような操作的意味論の下でも章節番号を正確に自動で入れる方法は依然としてあるかもしれないが，少なくとも自然に記述することはかなり困難だろう．結局このような例から「テキストに対する評価は“遅延評価風の”評価順序にした方が筋が良いだろう」という判断をし，本章で紹介したような言語が SATYSF$_{\mathrm{I}}$ の核をなすことになったのである．

参考文献

[1] Adobe Systems. *PostScript Language Reference Manual*. Addison Wesley Longman, 1985.

[2] 浅井 健一 . プログラミングの基礎 . サイエンス社 , 2007.

[3] Daniel Bünzli and Takashi Suwa. *An extended version of Otfm*. `https://github.com/gfngfn/otfm`, 2017.

[4] Victor Eijkhout. *TEX by Topic*. Addison–Wesley, 1991.

[5] Matthias Ettrich. *LyX – The Document Processor*. `https://www.lyx.org`, 1995.

[6] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

[7] Masaki Hara. *satysfi-cross-windows*. `https://github.com/qnighy/satysfi-cross-windows`, 2018.

[8] Masaki Hara. *SATYSFI 構文メモ (2018/02/23)*. `https://gist.github.com/qnighy/951d1ee5a7ee7c8f95bb323e881eaf8a`, 2018.

[9] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, pages 29–60, 1969.

[10] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE, 2008.

[11] 五十嵐 淳 . プログラミング *in OCaml*. 技術評論社 , 2007.

[12] 五十嵐 淳 . プログラミング言語の基礎概念 . サイエンス社 , 2011.

[13] Hiromi Ishii. *Steamer: presentation in SATYSFI*. `https://github.com/konn/satysfi-steamer`, 2018.

[14] 木枝 祐介 . 数式組版 . ラムダノート , 2018.

[15] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, pages 505–519, 2007.

[16] Donald E. Knuth. *The TEXbook*. Addison–Wisley, 1986.

[17] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software–Practice and Experience*, 11, pages 1119–1184, 1981.

[18] 黒木 裕介 . 組版におけるオペレーションズ・リサーチ . オペレーションズ・リサーチ , 60(9), pages 536–542, 2015.

[19] Franklin M. Liang. *Word Hy-phen-a-tion by Com-put-er*. Ph. D. thesis, Stanford University, 1983.

[20] miettal. *SATYSFI* が *Web* で動くやつを作った — 暇な女子高専生のブログ . http://miettal.hatenablog.com/entry/2018/02/15/133623, 2018.

[21] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science (JCSS)*, 17, pages 348–374, 1978.

[22] mjambon. *Yojson*. https://github.com/mjambon/yojson, 2010.

[23] Atsushi Ohori. A polymorphic record calculus and its compilation. In *ACM Transactions on Programming Languages and Systems*, 17(6), pages 844–895, 1995.

[24] Edward Osinski. *A Polymorphic Type System and Compilation Scheme for Record Concatenation*. Ph. D. thesis, New York University, 2006.

[25] pandaman64. *SATYSFI Playground*. http://satysfi-playground.tech/, 2018.

[26] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[27] Benjamin C. Pierce（住井 英二郎 監訳，遠藤 侑介 訳，酒井 政裕 訳，今井 敬吾 訳，黒木 裕介 訳，今井 宜洋 訳，才川 隆文 訳，今井 健男 訳）. 型システム入門 . オーム社 , 2013.

[28] John Plaice and Yannis Haralanbous. *Draft documentation for the Ω system*. http://www.bakoma-tex.com/doc/omega/base/doc-18.pdf, 1999.

[29] Didier Rémy. Type inference for records in a natural extension of ML. *Theoretical Aspects of*

Object-Oriented Programming. Types, Semantics and Language Design, pages 67–95, 1993.

[30] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[31] Unicode Consortium. *Unicode Standard Annex #11: East Asian Width (Unicode 10.0.0)*. `http://unicode.org/reports/tr11/`, 2017.

[32] Unicode Consortium. *Unicode Standard Annex #14: Unicode Line Breaking Algorithm (Unicode 10.0.0)*. `http://unicode.org/reports/tr14/`, 2017.

[33] Unicode Consortium. *Unicode Standard Annex #24: Unicode Script Property (Unicode 10.0.0)*. `http://unicode.org/reports/tr24/`, 2017.

[34] W3C 日本語組版タスクフォース . *W3C* 技術ノート 日本語組版処理の要件 . 東京電機大学出版局 , 2012.

The SATYSFIbook （Web 公開版 第 1 版）

発行日	2018 年 12 月 5 日
発行	Takashi SUWA
連絡先	gfngfn.bd@gmail.com