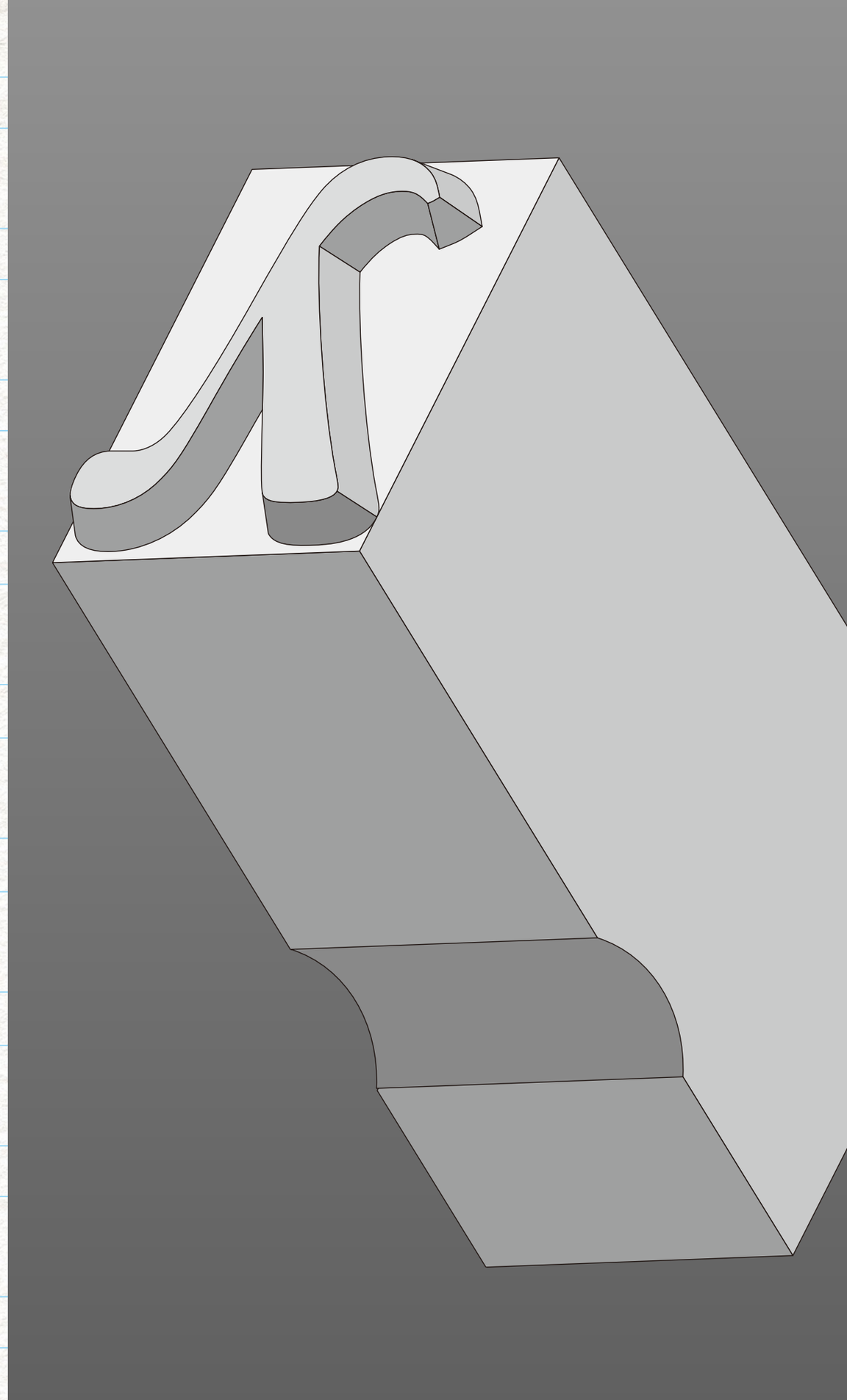
静的型つき函数型組版処理システム SATySFi の紹介

2022年3月7日

諏訪　敬之 (gfngfn)



概要

SATySFi（サティスファイ）とは

- 講演者が 2017 年度 IPA 未踏事業の1プロジェクトとして開発した
  比較的新しい**組版処理システム**
- **静的型つき**のいわゆる**函数型**のプログラミング言語で
  パッケージや文書本体が書ける
  - 一般的なプログラムと同じ要領で読み書きできるので
    カスタマイズ性が高い
    - cf. TEX/LATEX： 複雑な意味論ゆえにパッケージ作成には鍛錬を要する上,
      熟練者でも他人の書いた実装は読んで改変するのが難しい
  - 誤った記述をしたときも，型検査によって**迅速**（典型的には 0.1 秒ほど）
    かつ原因の解りやすいエラー報告を出してくれる傾向にある
    - cf. TEX/LATEX： 些細なミスでも十数秒待ってから解りにくいエラーが出がち

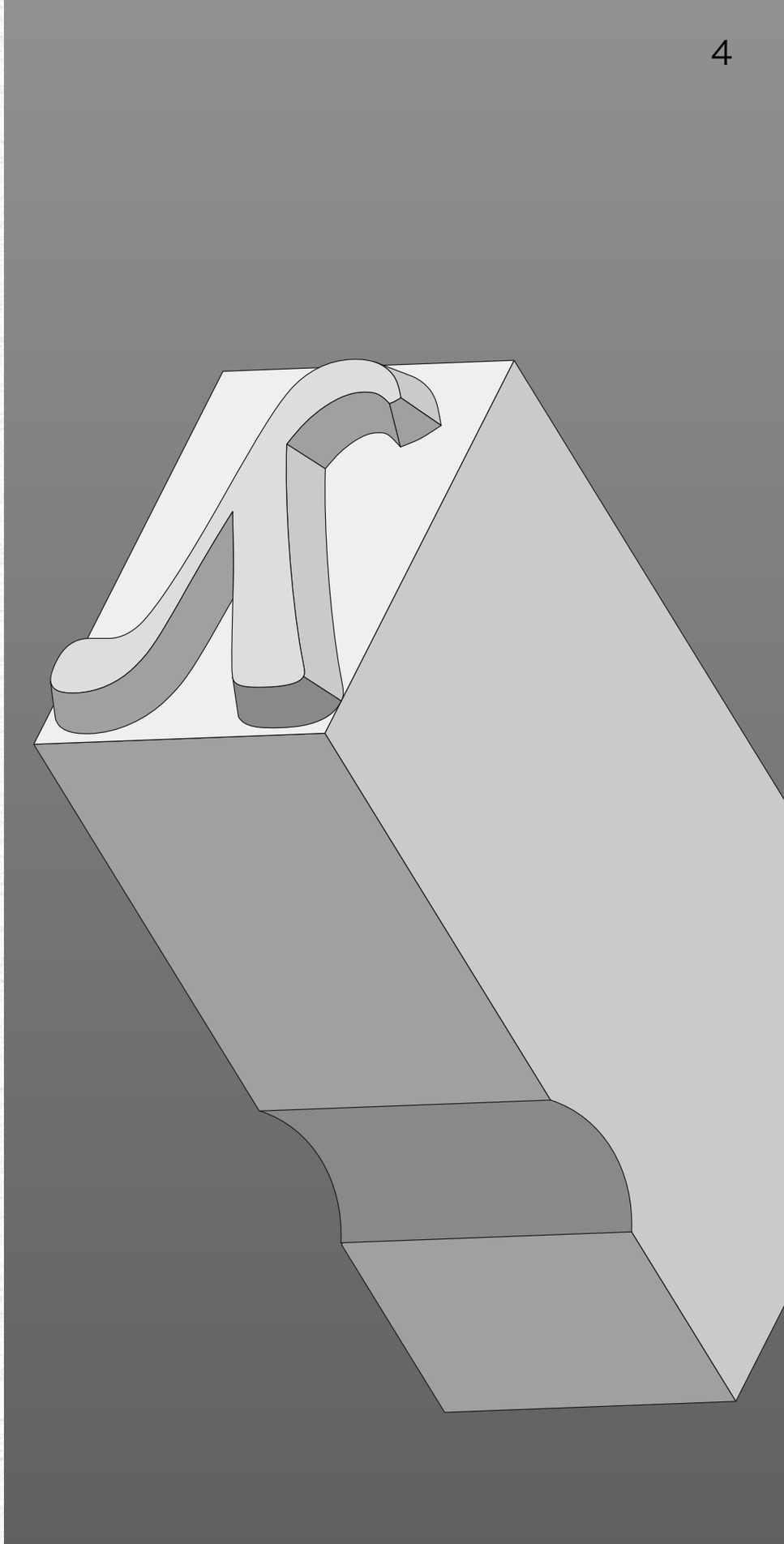
本講演の内容

SATγSF| の紹介

- 開発動機・目的意識の共有
- PPL に参加される方々の関心と照らし合わせ,
  組版処理・文書作成といった用途に要請されて
  SATγSF| にどんな domain-specific な意味論や型システムが搭載されているかを概説
- その他，有志の方々による SATγSF| パッケージや周辺ツールの紹介

目次

- **開発動機・目的意識**
- 動作デモ
- 文字組版の定式化とその型つけ
- 文書のマークアップ方法の定式化
- 行分割処理の仕組み
- 多段階計算ベースのマクロ機構による DSL
- 有志の方々による周辺ツール・パッケージ
- まとめ



文書作成ソフトウェアの形態

大別して 2 種類：

マークアップ言語＋処理系

TEX・LATEX

[Knuth 1978] [Lamport 1985]

troff

[Osanna 1973]

SATySFi

前処理用途

RE:VIEW

[青木 et al. 2002]

[Gruber 2004]

WYSIWYG エディタ

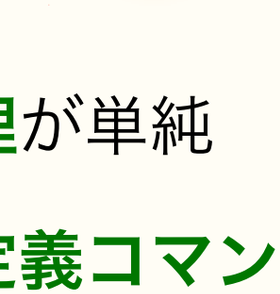
[Microsoft 1983] [Adobe 1991]

[Quark 1987]

マークアップ言語方式の性質

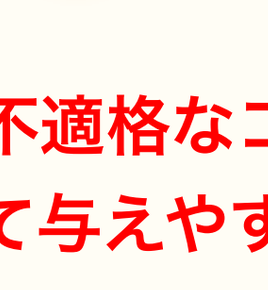
WYSIWYG エディタ方式と比べて，傾向として：

- **差分管理**が単純
- **ユーザ定義コマンド**により：
  - **複雑な自動処理**が実現可能
  - 文書の体裁が
    後から柔軟に変更可能



この特徴を活かしたい人が想定ユーザ

- **ユーザが不適格なコードを入力として与えやすい**



どのようにエラーが報告されるかが執筆効率に影響

既存システムの弱点

例： TEX/LATEX

- **エラー報告が不親切で遅い**傾向にある

```
! Undefined control sequence.
```

```
! Missing $ inserted.
```

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

  - これゆえに，少し凝った処理を定義しようとすると**デバッグが困難**
  - 処理系の実装が怠惰なのではなく，TEX のもつ操作的意味論の都合上
    不親切なエラーにならざるを得ないという側面が大きい

手の込んだ自動処理が定義できつつ，“明らかなミス”には親切なエラーが出せるような構文・意味論の言語にしたい

最初の案： 型つきの wrapper をつくる

静的型つきで，TEX/LATEX コードを出力するような
言語を創ればよいのではないか？

2015 年頃に **Macrodown** という ML 系言語として実装し，
それなりに使えるものにはなった

顕在化した弱点：

- ページ数や紙面上の幅といった組版上の概念に依存する処理は wrapper 側で実装しにくく，**依然として LATEX 側での実装が必要**
  - 特にクラスファイルの実装などは wrapper 側では到底完結しないはず

“TEX言語”の操作的意味論の概説

wrapper 言語が与えにくかったりエラー報告が不親切だったりする
要因は，主に TEX のもつ操作的意味論の複雑怪奇さにある：

- 字句解析に近いことはするが，**抽象構文木の概念はない**
- 制御綴（\foo）の展開は後続トークン列に対するパターンマッチング
  - “括弧 { … } が釣り合いパターンに合致する最短のトークン列” を引数とみなす
- 破壊的に変更可能な変数の機構が多用される

- 字句解析は “lazy” で，前方で評価相当の処理が行なわれていても
  後方はまだトークン化されていない文字列であり，
  前方の評価結果が後方の字句解析方法を変更しうる
- 以上のような複雑な機構が組版処理と密結合で，避けにくい

静的に特定の性質を保証するという試みはあまり望みが持てない

小さいサブセットでもあまり強い保証はできない様子 [Erdweg & Ostermann 2010]

目的意識

LaTeX バックエンドを wrap した言語で済ませずに,
自前で組版処理まで行なえる言語と処理系を設計・実装したい

主要な要件：

- **本質的に組版処理に関わるような複雑な自動処理ができること**
  - 他人の書いた自動処理の実装も大きな支障なく読んで改変しやすいことも含む
- **明らかなエラーについては静的に**（＝組版処理を始めるより前に）**検知し, その原因をなるべく特定しやすい形で報告できること**

蛇足：もっと雑な問題意識

(* 敢えてやや挑戦的に書くなら：

- TEX/LATEX はもちろん或る面では素晴らしいソフトウェアだが,
  これだけ言語設計の知見が溜まっている現在でも
  扱いにくい意味論をもつ TEX にずっと依存し続けるのは
  工学的に健全な状態とは言いにくいのではないか？
  - cf. 汎用の計算機言語では様々な言語設計が日夜提案されている
- 日々プログラムの性質を静的に保証することをひたすら熱心に
  考えているのに，いざ論文を書く際の道具ではその結果を享受せず
  不親切なエラーばかり見ているのは，紺屋の白袴ではないか？

*)

成果物

新しい組版処理システム

SATySFi

https://github.com/gfngfn/SATySFi

Static **A**nalysis-based **TY**pesetting **S**ystem
for **F**unctional **I**mplementation

（かなり強引なこじつけ）

特徴

構文が大別して 2 層に分かれている

“マークアップ層”

```
+section{\SATySFi;の原稿例}<
  +p{
    PPLでは\emph{2回目の発表}です.
    お世話になります.
  }
>
```

- LaTeX 風
- インライン { … } と
  ブロック < … > の区別がある

“プログラム層”

```
let-inline ctx \emph it =
  let ctx =
    ctx |> set-font
      Latin italic-font
  in
  read-inline ctx it
```

- OCaml 風

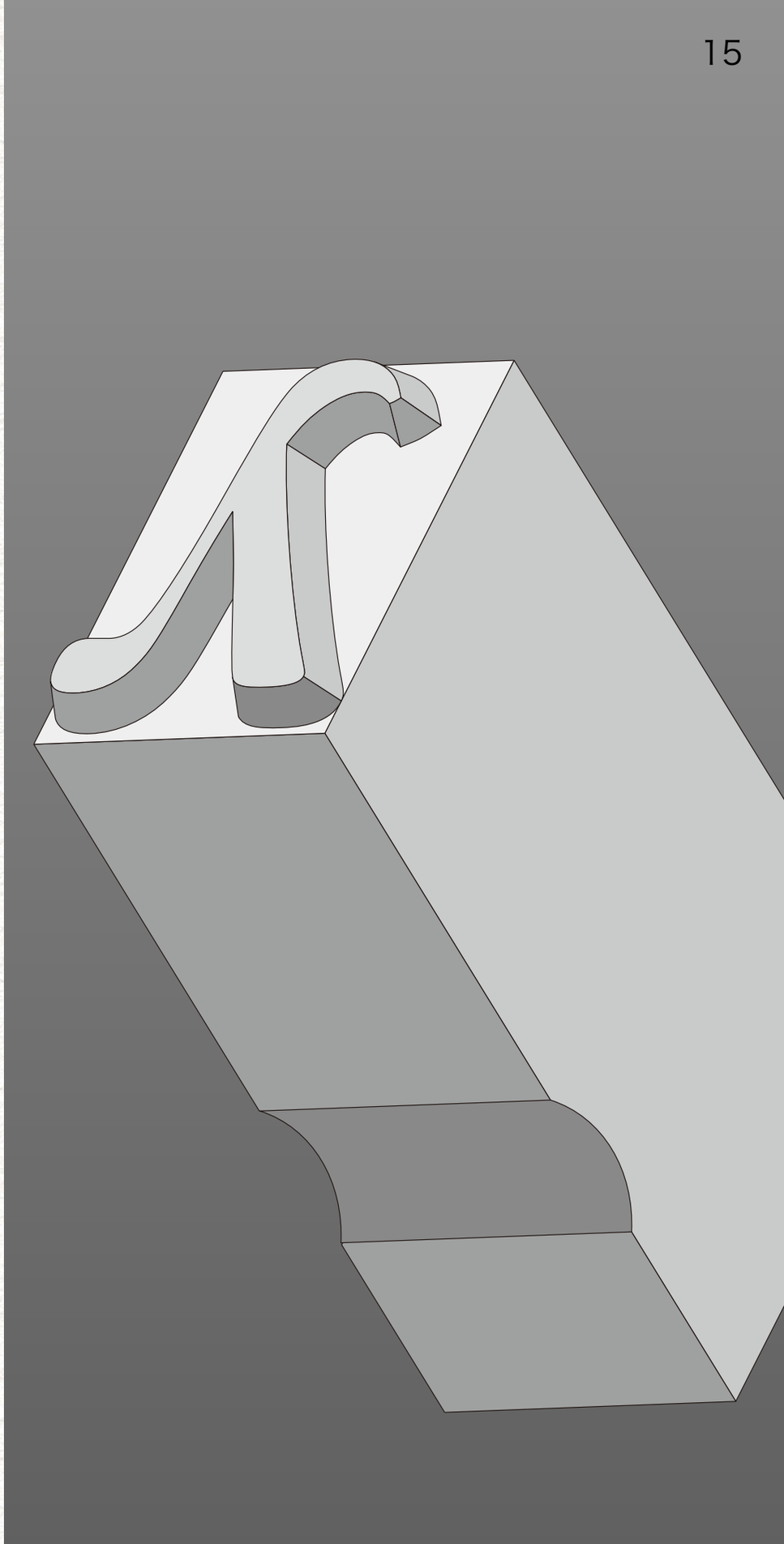
特徴

“プログラム層” ではコマンド定義が OCaml 風の言語で書ける

- “グローバルな状態” をあまり気にせずに処理が書ける
  - ※ 止むを得ず mutable reference の機能はあり，採番などに使用
- **型システム**を用いた静的な（＝組版処理に先立った）エラー報告能力
  - 通常の let 多相＋α 程度
    - レコード計算 [Ohori 1995][Rémy 1993][Gaster & Jones 1996]
    - オプション引数
  - 依存型や篩型は無し
  - **組版処理やドキュメント処理に関わる多数の基本型と組込み函数**
    - これが今回紹介する主な対象

目次

- 開発動機・目的意識
- **動作デモ**
- 文字組版の定式化とその型つけ
- 文書のマークアップ方法の定式化
- 行分割処理の仕組み
- 多段階計算ベースのマクロ機構による DSL
- 有志の方々による周辺ツール・パッケージ
- まとめ



動作デモ１： 正常系

（デモ映像： https://drive.google.com/file/d/1muaWGgyAGfIYJNyKfKvvAzN8vur1sTG3/view?usp=sharing）

動作デモ２： 異常系

以下のような「いろは歌を幅 5cm の枠で囲って組む」出力をしようとして
やりがちな不適格なコードを書いてしまったときのエラー報告を
LaTeX と SATySFi で比較

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，酔いもせず。

LaTeX の場合

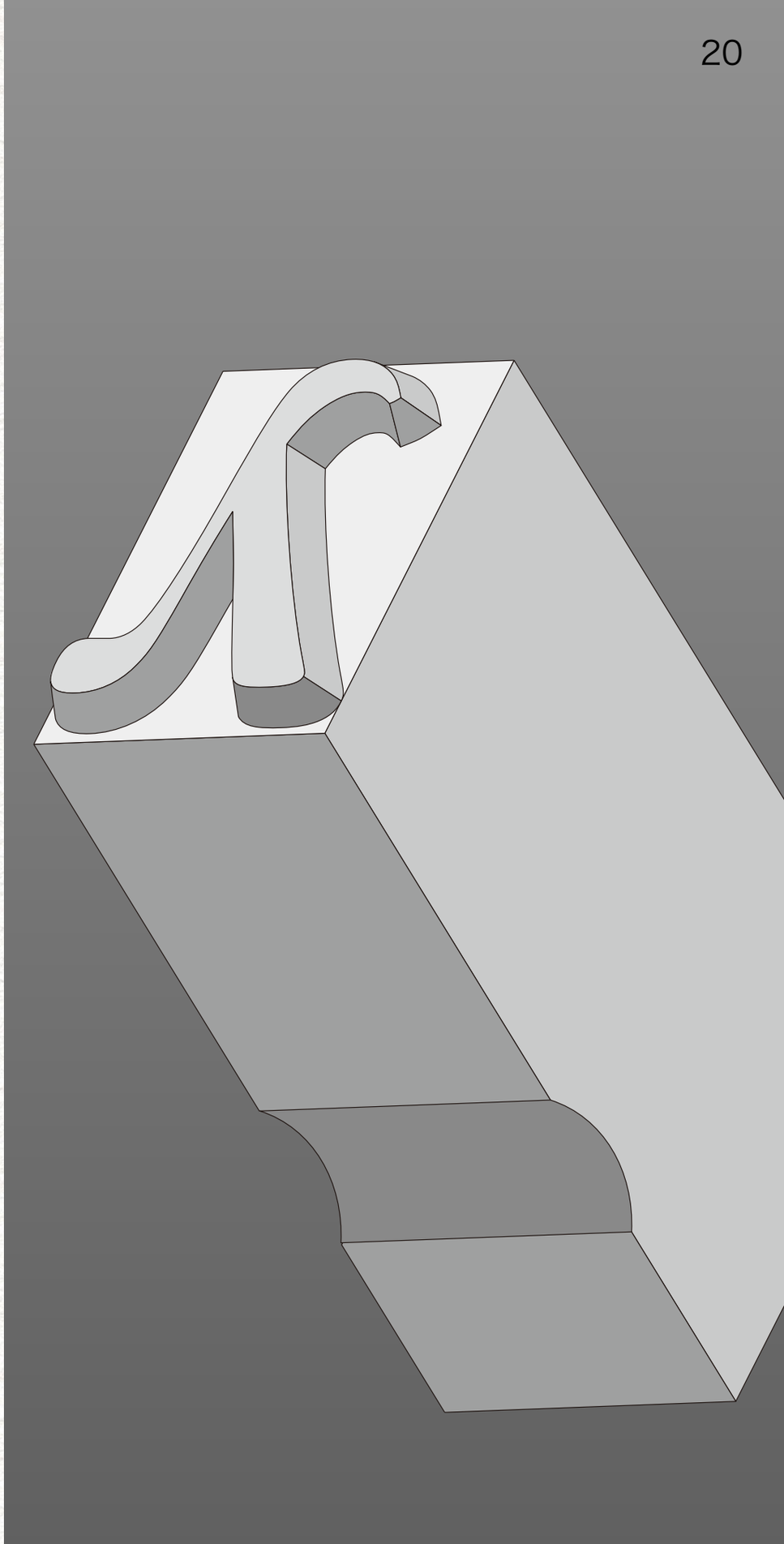
（デモ映像： https://drive.google.com/file/d/1r14gHwCUxe3DQjoxkGHDWWkwtHWqMwaY/view?usp=sharing）

SATySFi の場合

（デモ映像： https://drive.google.com/file/d/1DyKB0V2xIKz6SMYNGdEWnupPAxXkWCeE/view?usp=sharing）

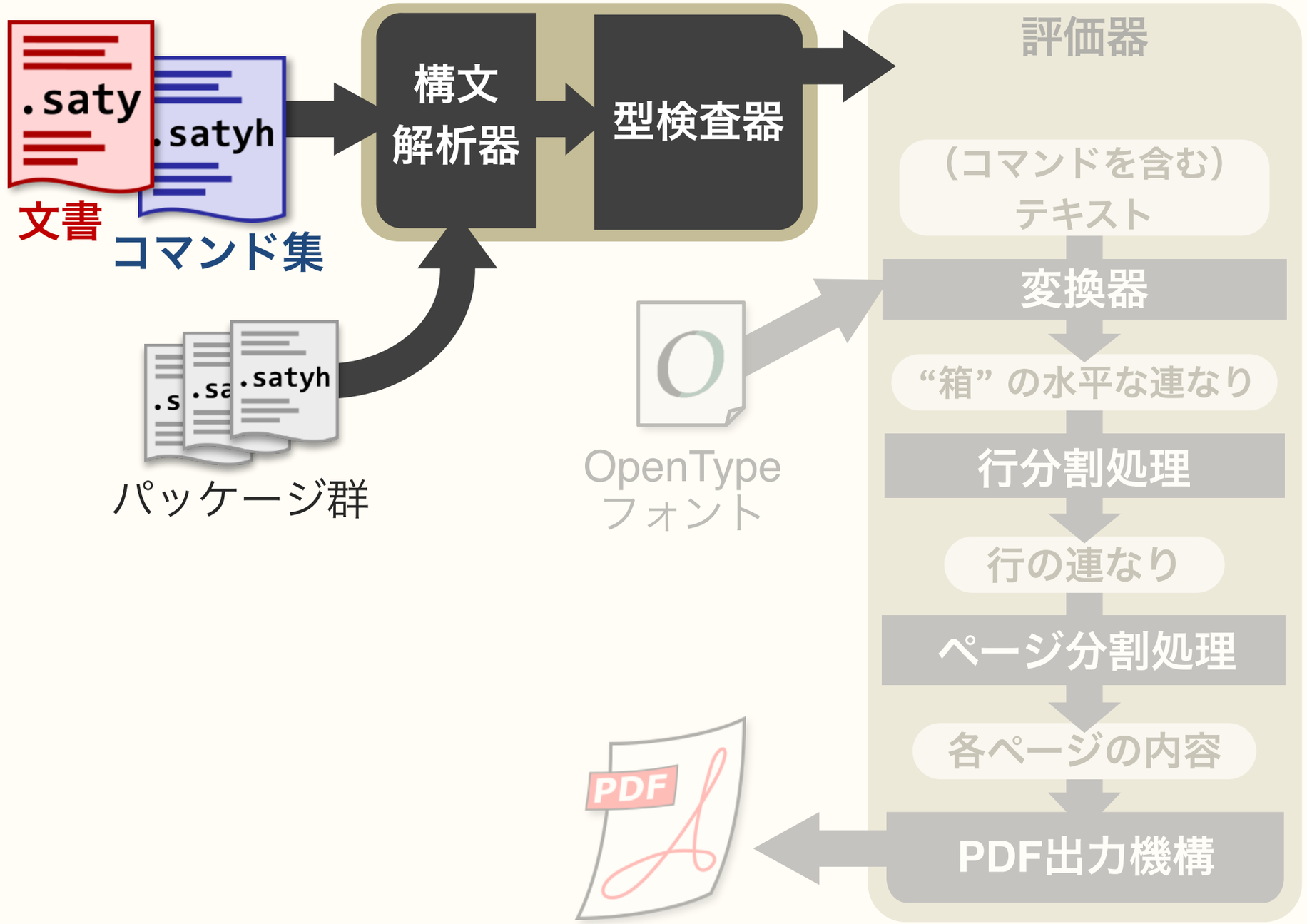
目次

- 開発動機・目的意識
- 動作デモ
- **文字組版の定式化とその型つけ**
- 文書のマークアップ方法の定式化
- 行分割処理の仕組み
- 多段階計算ベースのマクロ機構による DSL
- 有志の方々による周辺ツール・パッケージ
- まとめ



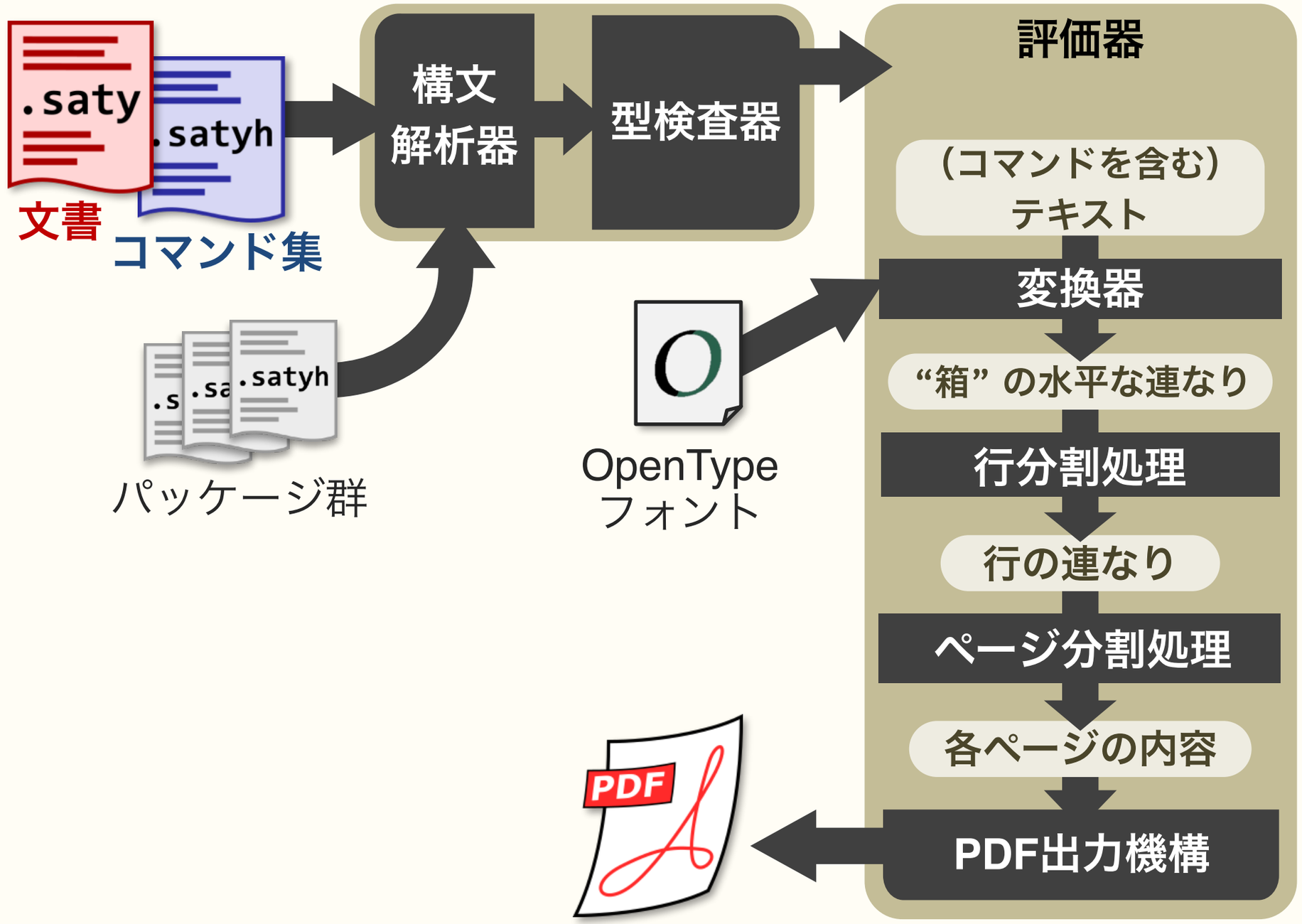
処理系の構成の模式図

フロントエンド（検査機構）＋ バックエンド（インタプリタ）

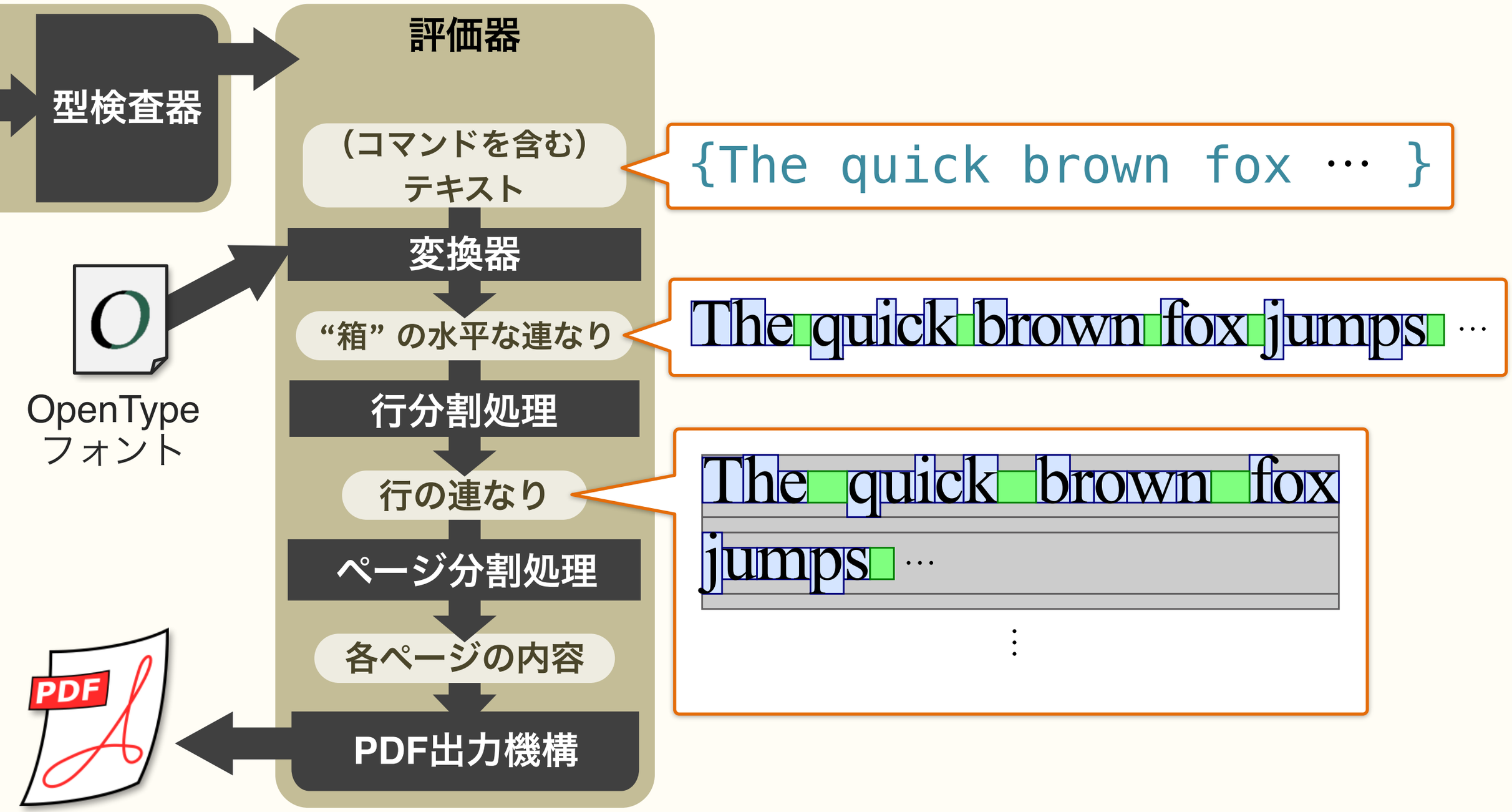


処理系の構成の模式図

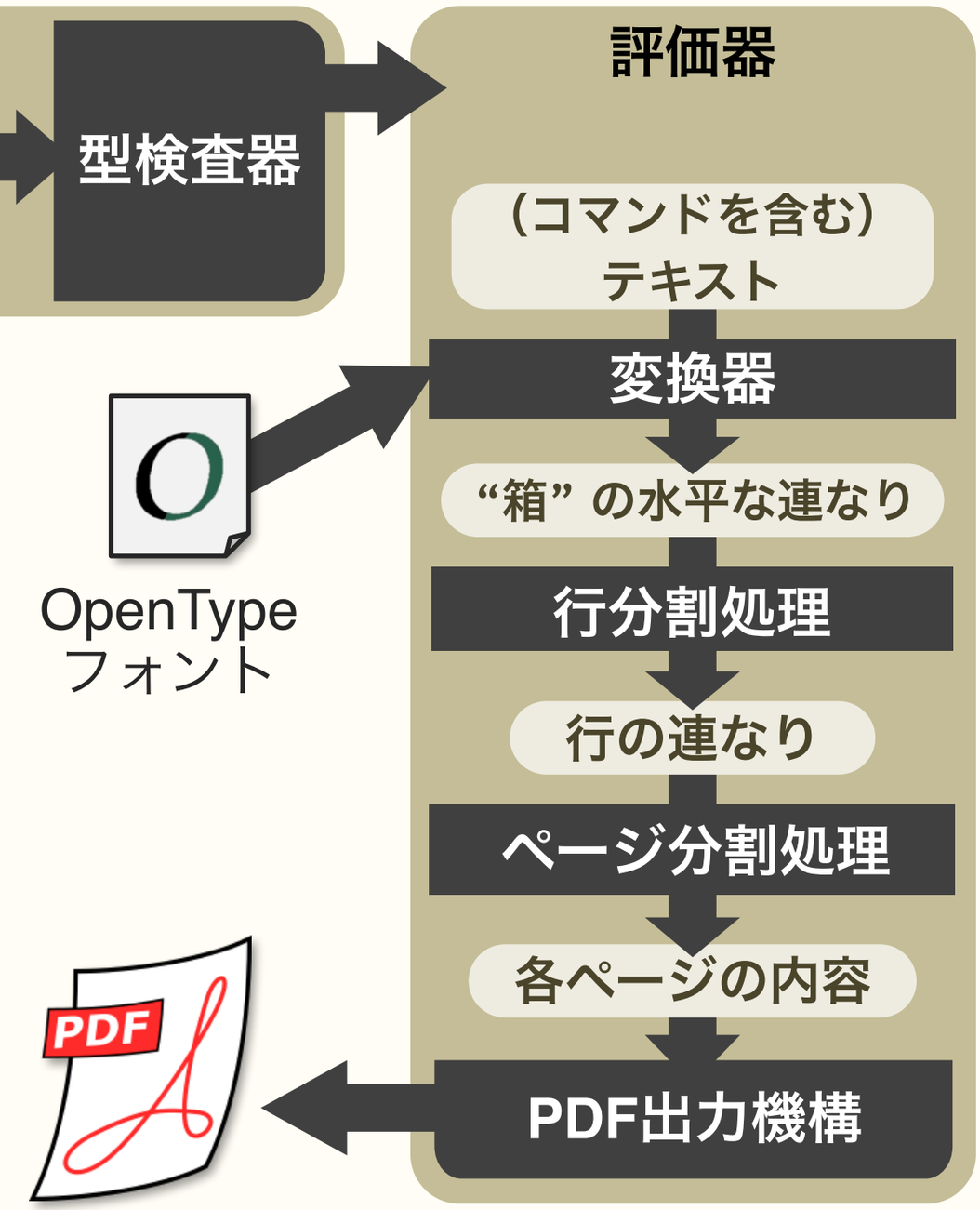
フロントエンド（検査機構）＋ バックエンド（インタプリタ）



処理系の構成の模式図


評価器
型検査器
(コマンドを含む)
テキスト
{The quick brown fox … }
変換器
OpenType
フォント
“箱”の水平な連なり
The quick brown fox jumps …
行分割処理
行の連なり
The quick brown fox
jumps …
ページ分割処理
各ページの内容
PDF出力機構
PDF

処理系の構成の模式図

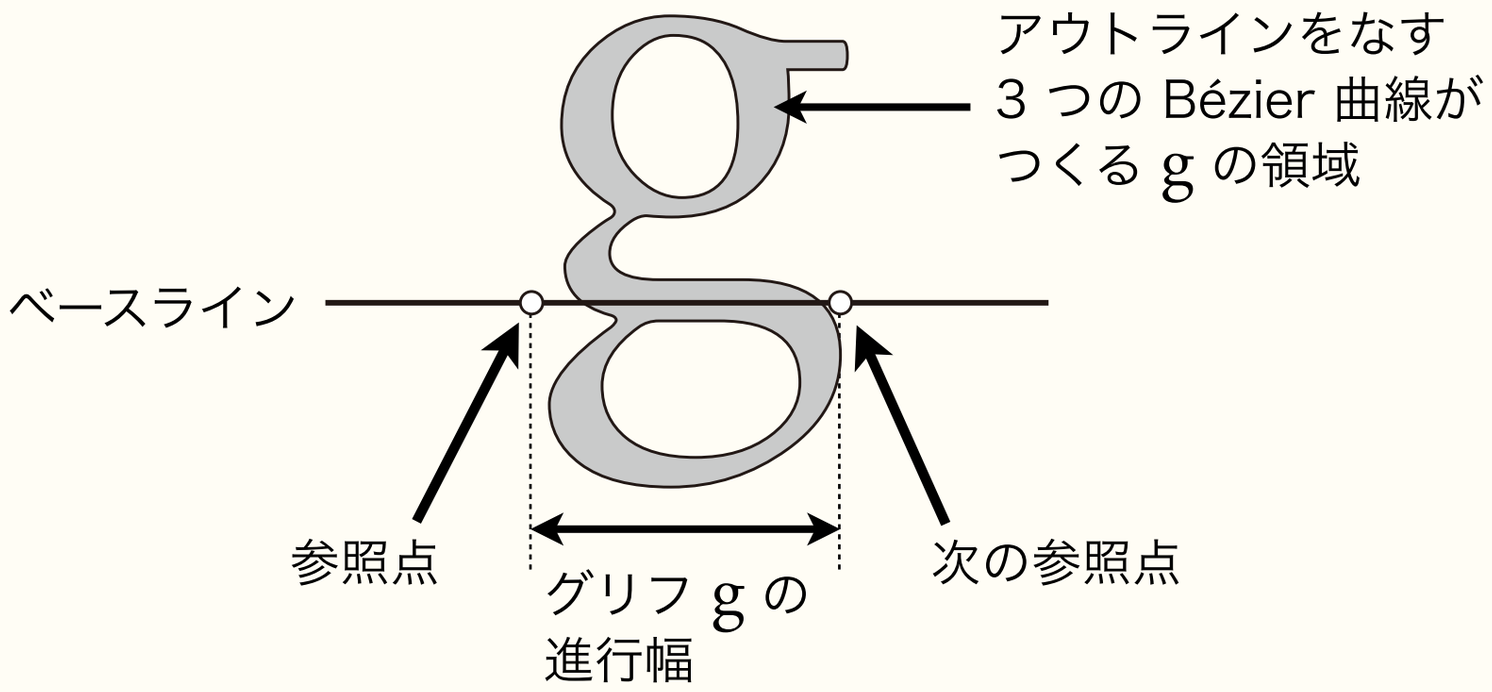


自動処理の実装では
このあたりの組版処理に
介入できてほしい

自動処理の定義が型検査に通れば,
評価時にここで
データの不整合が起きないような
型システムを用意したい

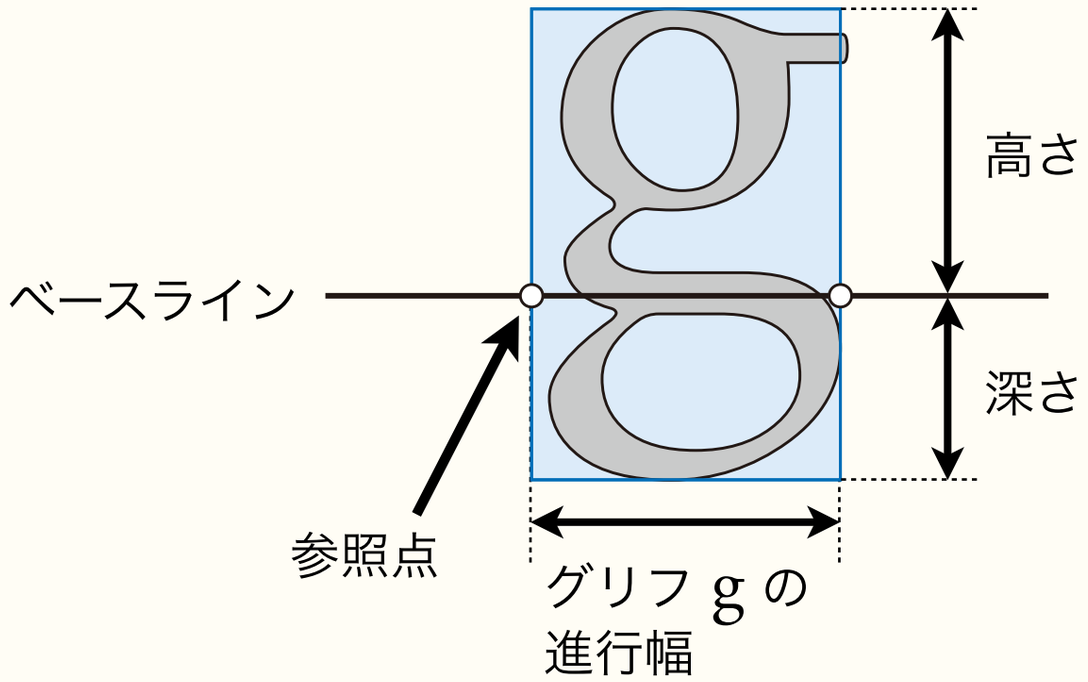
一般の組版処理に於ける文字の扱い方

- **グリフ** (*glyph*)： 図形としての文字を扱う単位
- グリフは**ベースライン** (*baseline*) を基準にして並ぶ
- 各グリフは以下のデータをもつ：
  - **アウトライン**： **参照点** (*reference point*) からの相対座標で記録された閉曲線
  - **進行幅** (*advance width*)： その文字の配置によって参照点をどれだけ進めるか



一般の組版処理に於ける文字の扱い方

- **グリフ** (*glyph*)： 図形としての文字を扱う単位
- グリフは**ベースライン** (*baseline*) を基準にして並ぶ
- 各グリフは以下のデータをもつ：
  - **アウトライン**： **参照点** (*reference point*) からの相対座標で記録された閉曲線
  - **進行幅** (*advance width*)： その文字の配置によって参照点をどれだけ進めるか



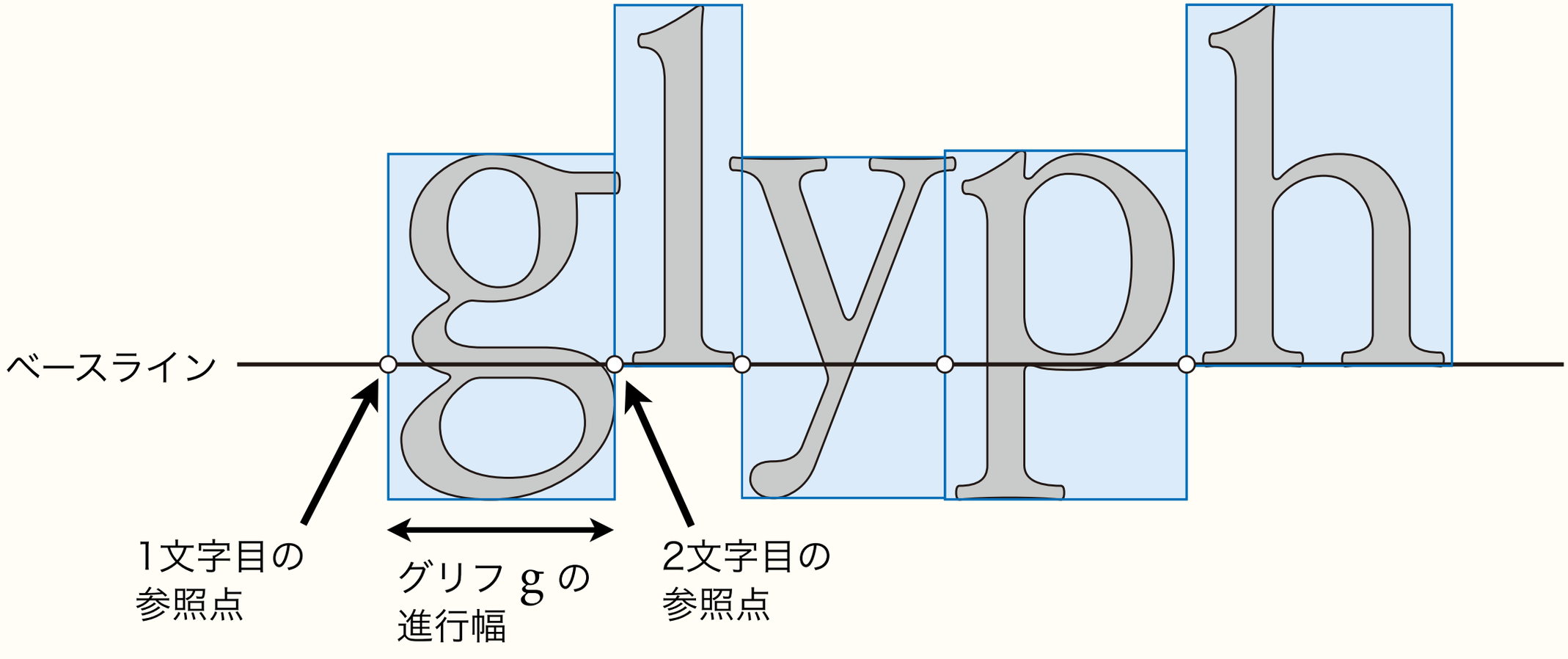
さらに以下で高さと深さを定義し，
グリフは仮想的な “箱” をもつとみなす

- **高さ**： max(0, 閉曲線の最高到達座標)
- **深さ**： max(0, −(閉曲線の最低到達座標))

※ これは普遍的な定式化とまでは言えないが，
少なくとも TEX の組版処理 [Knuth 1978] では
実質的にこれに相当する方法を採っており，
SATYSFI でも踏襲

一般の組版処理に於ける文字の扱い方

- **グリフ** (*glyph*)： 図形としての文字を扱う単位
- グリフは**ベースライン** (*baseline*) を基準にして並ぶ
- 各グリフは以下のデータをもつ：
  - **アウトライン**： **参照点** (*reference point*) からの相対座標で記録された閉曲線
  - **進行幅** (*advance width*)： その文字の配置によって参照点をどれだけ進めるか

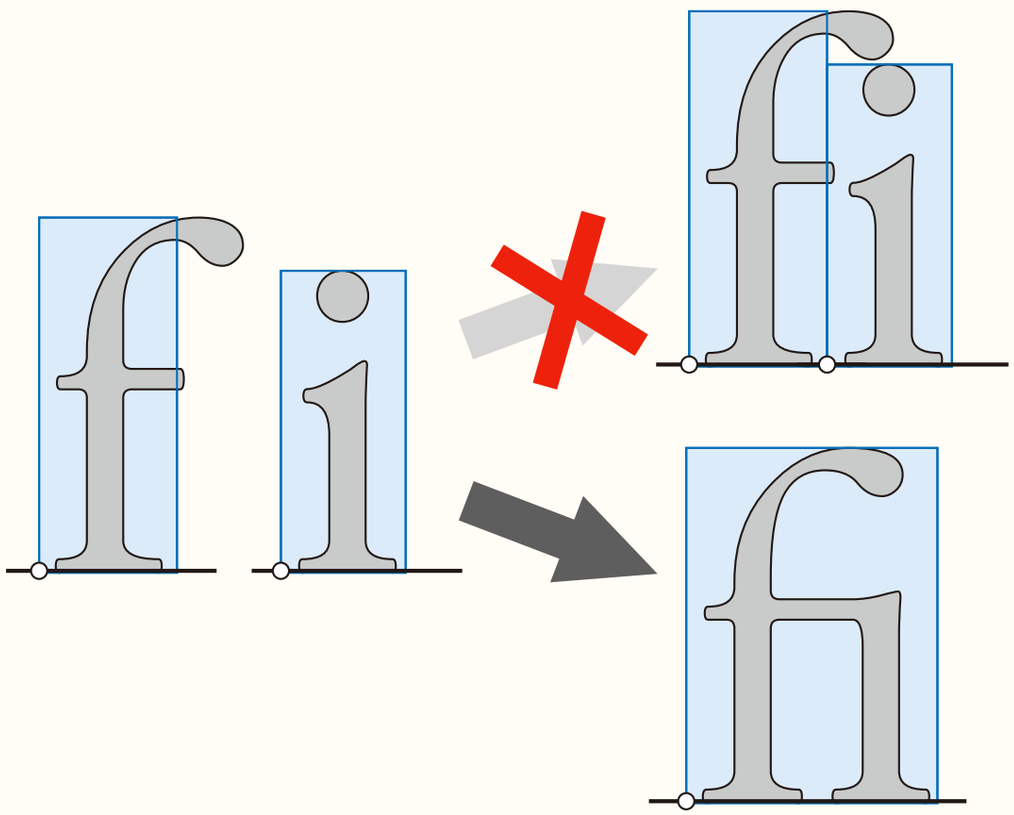


組版処理に於ける文字の定式化

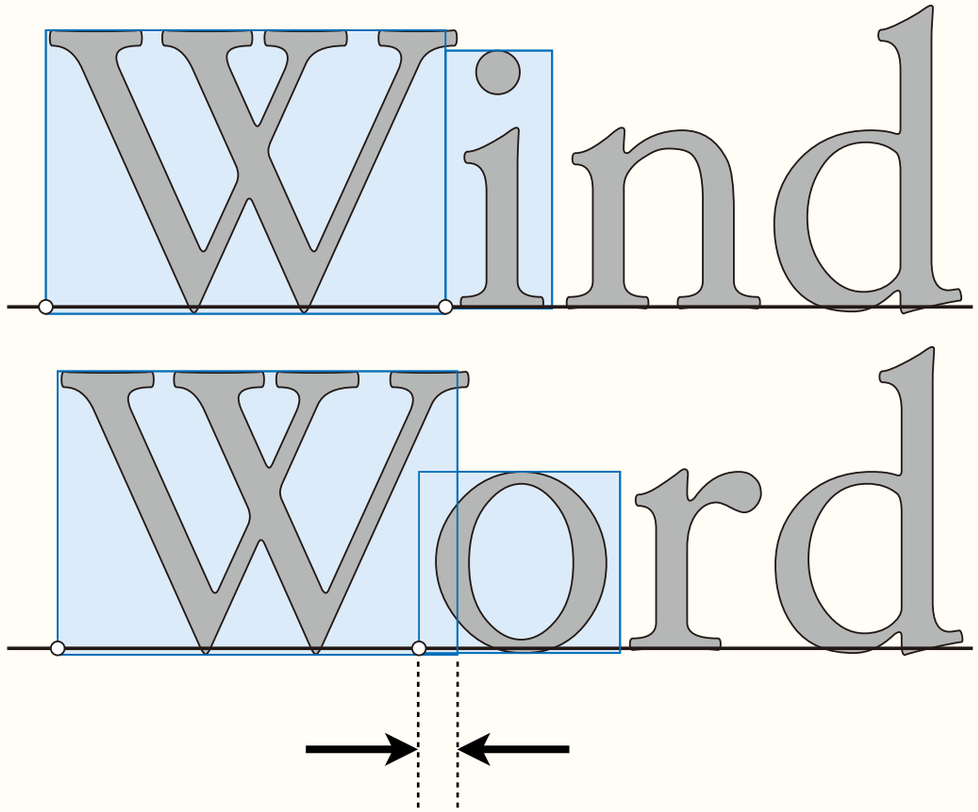
アウトラインと進行幅だけでは支障をきたす場合もあるので,
特別な組み合わせに対してのみ適用される処理がある

- フォントファイル内にそのためのデータが格納されている

合字 (*ligature*)：
相性が悪い形状のグリフが並ぶときは
合体して 1 グリフにする



カーニング (*kerning*)：
進行幅に従うと字間が不自然になる
組み合わせでは調整を行なう



実際に組まれた合字・カーニングの例（デモ再掲）

プポーションなど，欧文組版に必要な高度な調整機能がいくつも備わっている．実際に以下の $\mathrm{SATYSFI}$ によって組まれた欧文を見てみよう：

The quick brown fox jumps over the lazy dog.

¿But aren't Kafka's Schloß and Æsop's Œuvres often naïve vis-à-vis the dæmonic phœnix's official rôle in fluffy soufflés?

The MIT License: Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the

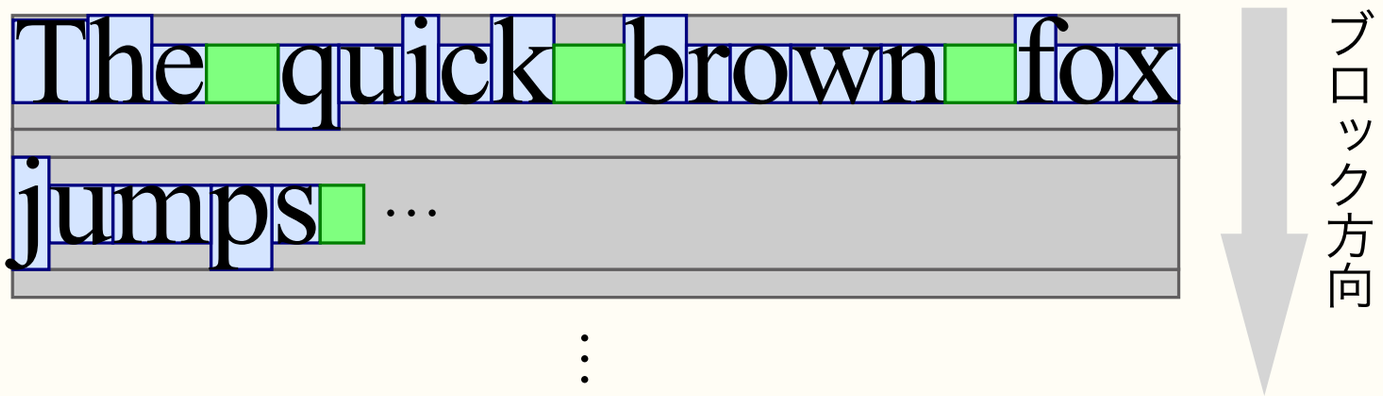
“箱”の連なりとしての行や段落

グリフの “箱” は水平方向に並んで “行” をなす

- この “行” を**インラインボックス列**と呼ぶ

“行” は**行分割処理** (*line breaking*) によって切り分けられて “段落” になる

- 空白などは適切に伸縮し，行長が揃うように整形
- “段落” も横に細長い “箱” が鉛直方向に連なったもので，
  ブロックボックス列と呼ぶ



操作対象としてのボックス列の定式化

インラインボックス列をプログラムで操作する対象として追加：

値　$v ::= \lambda x\,.\,e \mid \cdots \mid \boldsymbol{ib}$

インラインボックス列　$\boldsymbol{ib} ::= [ib]^*$

0 個以上の有限個の列

$ib ::=$

$\mid$ e $\mid \cdots \mid$ あ $\mid \cdots$　グリフ

$\mid$ ▯　空白

⋮

インラインボックス列には列全体で **inline-boxes** という基本型をつける：

$$\frac{}{\Gamma \vdash \boldsymbol{ib} : \textbf{inline-boxes}}$$

以下の組込み函数で連結できる：

$$\textbf{(++)} : \textbf{inline-boxes} \to \textbf{inline-boxes} \to \textbf{inline-boxes}$$

操作対象としてのボックス列の定式化

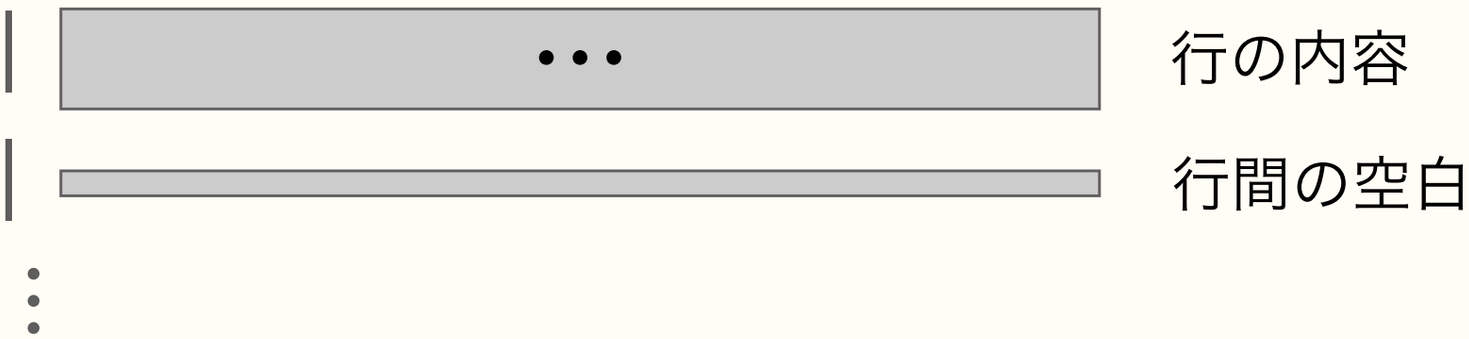
ブロックボックス列も操作対象として追加：

値　$v ::= \lambda x\,.\,e \mid \cdots \mid \boldsymbol{bb}$

ブロックボックス列　$\boldsymbol{bb} ::= [bb]^*$

0 個以上の有限個の列,
直観としては縦に連なっている

$bb ::=$



ブロックボックス列には列全体で **block-boxes** という基本型をつける：

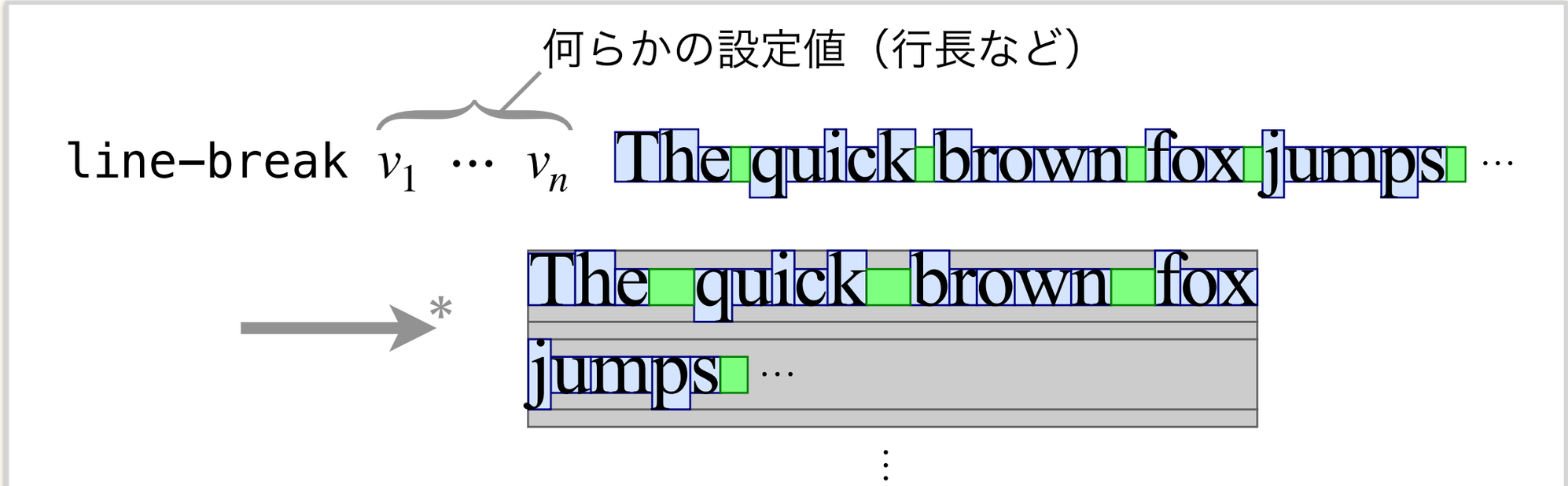
$$\frac{}{\Gamma \vdash bb : \textbf{block-boxes}}$$

やはり組込み函数で連結できる：

$$\textbf{(+++)} : \textbf{block-boxes} \to \textbf{block-boxes} \to \textbf{block-boxes}$$

組込み函数による行分割処理の提供

以下のような組込み函数 **`line-break`** として
行分割処理を提供してユーザが使用できるとよさそう：



`line-break` の型は以下の形：

$$\tau_1 \to \cdots \to \tau_n \to \textbf{inline-boxes} \to \textbf{block-boxes}$$

ここまでのまとめ

- グリフのなす “箱” が水平に並んだもの： **インラインボックス列 *ib***
- 細長い “行の箱” が縦に並んだもの： **ブロックボックス列 *bb***

$$v ::= \cdots \mid \mathit{ib} \mid \mathit{bb}$$

- ***ib*** の例：

- ***bb*** の例：

- 基本型の追加：

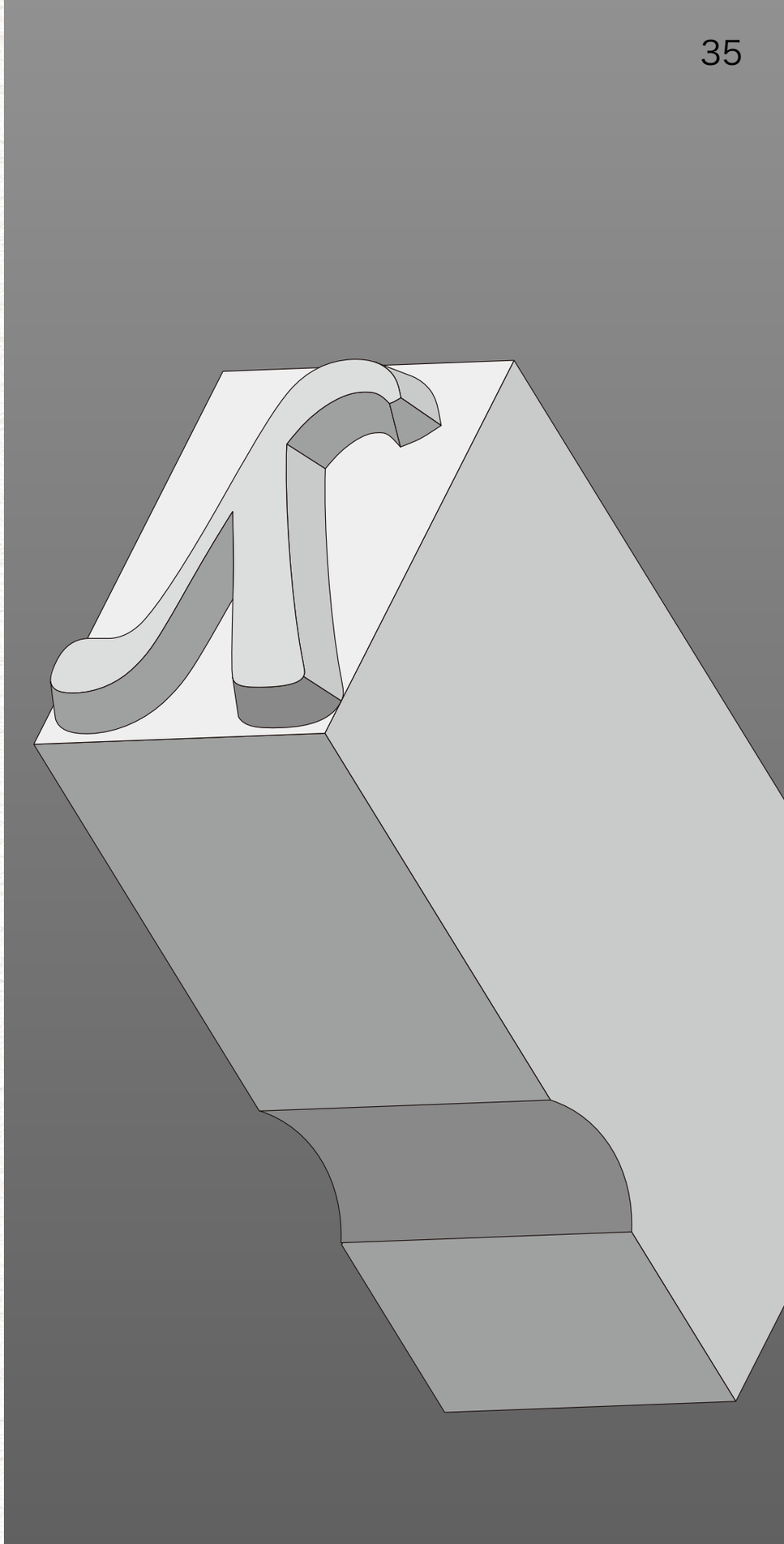
$$\tau ::= \cdots \mid \textbf{inline-boxes} \mid \textbf{block-boxes}$$

- **行分割処理**は以下の組込み函数で提供：

$$\texttt{line-break} : \tau_1 \to \cdots \to \tau_n \to \textbf{inline-boxes} \to \textbf{block-boxes}$$

目次

- 開発動機・目的意識
- 動作デモ
- 文字組版の定式化とその型つけ
- **文書のマークアップ方法の定式化**
- 行分割処理の仕組み
- 多段階計算ベースのマクロ機構による DSL
- 有志の方々による周辺ツール・パッケージ
- まとめ



文書のマークアップ方法の定式化

ここまででボックス列の処理方法の大枠は定式化できたが,
当然ながらエンドユーザがグリフを直接与えたりはしないので,
何らかのマークアップが施されたテキスト形式の原稿から
ボックス列へと変換して最終的に文書を組めるような仕組みが必要

どんな形式で記述する言語にするか？

目的意識からして以下を満たしていることが肝要：

- 何らかの方法で静的に妥当性を検査できる
- ユーザ自身がマークアップに使うコマンドを新規に定義できる

ADT で原稿を記述するには問題あり

XML に近い要領で，原稿を ADT（代数的データ型）で扱う？

クラスファイルが ADT を定義することでマークアップ方法を規定すると，
クラスファイルを使うエンドユーザ側ではマークアップの定義を拡張しにくい

- いわゆる **Expression problem**
- Open data types [Löh & Hinze 2006] があれば拡張できるが，
  データを使う側の函数の複雑化が依然避けにくい
  - OCaml でいう `type` t `+=` C `of` τ

Expression problem と無縁になるような定式化はできないか？

いわゆる Tagless-final 的な定式化をするとよさそう

現状の原稿記述方法

- `{ … }`： **インラインテキスト**
- `< … >`： **ブロックテキスト**
- `\foo`： **インラインコマンド**
  - インラインテキスト中のマークアップ
- `+foo`： **ブロックコマンド**
  - ブロックテキスト中のマークアップ

原稿の例（再掲）：

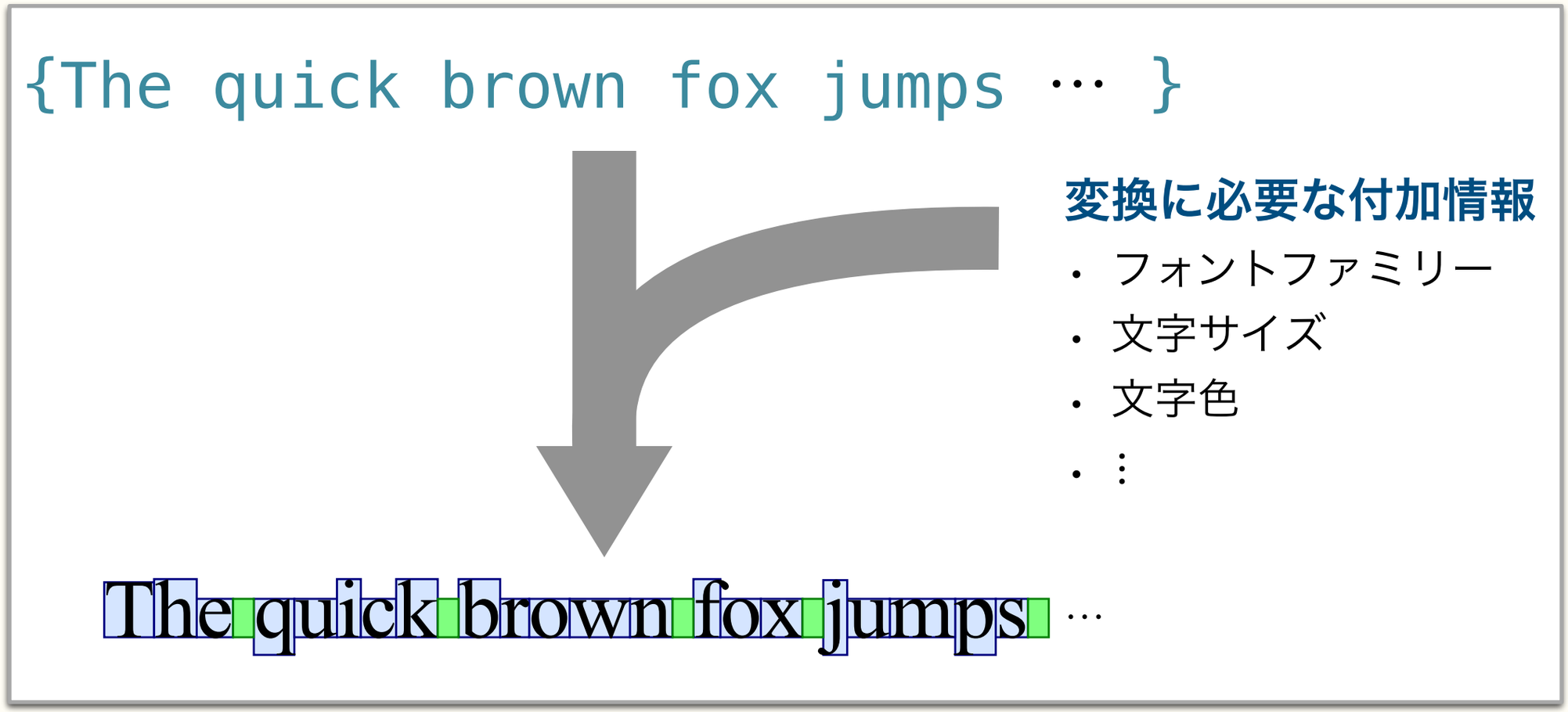
```
+section{\SATySFi;の原稿例}<
  +p{
    PPLでは\emph{2回目の発表}です.
    お世話になります.
  }
>
```

インラインテキスト $\{\ it\ \}$ の形式：

$$
\begin{aligned}
it &::= [it]^* \\
it &::= char && \text{文字（= Unicode コードポイント）} \\
&\quad \mid \texttt{\textbackslash foo}\,[e]^* && \text{コマンド適用}
\end{aligned}
$$

現状の原稿処理方法の定式化

まず，コマンド適用がなく文字列だけの場合の変換は簡単：



付加情報はそれらを全部集めたレコードのような値として
プログラム上で扱うことにし，この値を**テキスト処理文脈**と呼ぶ

テキスト処理文脈の定式化

長さ定数（例： 12pt）

$$v ::= \cdots \mid Ctx$$

$$Ctx ::= \{\text{font-size} = \ell, \ldots\}$$

$$\tau ::= \cdots \mid \textbf{context}$$

$$\frac{}{\Gamma \vdash Ctx : \textbf{context}}$$

安易に更新できてほしくない内容もあるので,
レコードであることはユーザからは見えず,
各項目の “getter” と “setter” が組込み函数として定義されている：

長さの型

- **`get-font-size`** : **context** → **length**
  - 格納されている文字サイズを取り出す
- **`set-font-size`** : **length** → **context** → **context**
  - (*Ctx* `|> set-font-size` ℓ) で文字サイズを ℓ に更新した *Ctx* を返す

ボックス列への変換の組込み函数

以下のような組込み函数 **read-inline** として提供：

```
read-inline Ctx {The quick brown fox jumps ⋯ }
```

→* The quick brown fox jumps ⋯

型は read-inline : **context** → **inline-text** → **inline-boxes**

インラインテキストにつく基本型

コマンド適用を含む場合に一般化

コマンドを適切に定義すれば以下の要領で変換できるようにしたい：

`read-inline` *Ctx* `{The \emph{quick brown} fox …}`

→* The *quick brown* fox …

`{The \emph{quick brown} fox …}`

この適用の部分を「テキスト処理文脈を受け取るとインラインボックス列になるもの」と捉える

`\emph{quick brown}` : **context → inline-boxes** とみなせる

`\emph` : **inline-text → context → inline-boxes** とみなせる

コマンド定義

`\emph` : **inline-text → context → inline-boxes** は以下のように定義できそう：

```
let \emph (it : inline-text) (ctx : context) =
  let ctx = ctx |> set-italic-font in
  read-inline ctx it
```

フォントをイタリック体に更新する，**context → context** 型の補助函数

実際の SATYSFI では以下で `\emph` を定義する：

テキスト処理文脈を受け取る変数が
構文上で手前に来ただけ

```
let-inline ctx \emph it =
  let ctx = ctx |> set-italic-font in
  read-inline ctx it
```

コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox …}
```

→* The *quick brown* fox …

という評価の途中経過を見てみる

Ctx →

```
The \emph{quick brown} fox …
```

コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox …}
```

$\longrightarrow^{*}$ The *quick brown* fox …

という評価の途中経過を見てみる

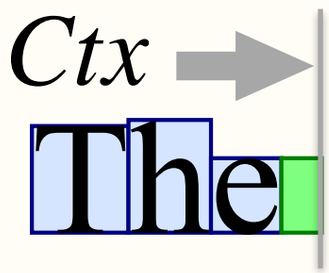
Ctx

```
The \emph{quick brown} fox …
```

コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox …}
```

という評価の途中経過を見てみる




```
\emph{quick brown} fox …
```

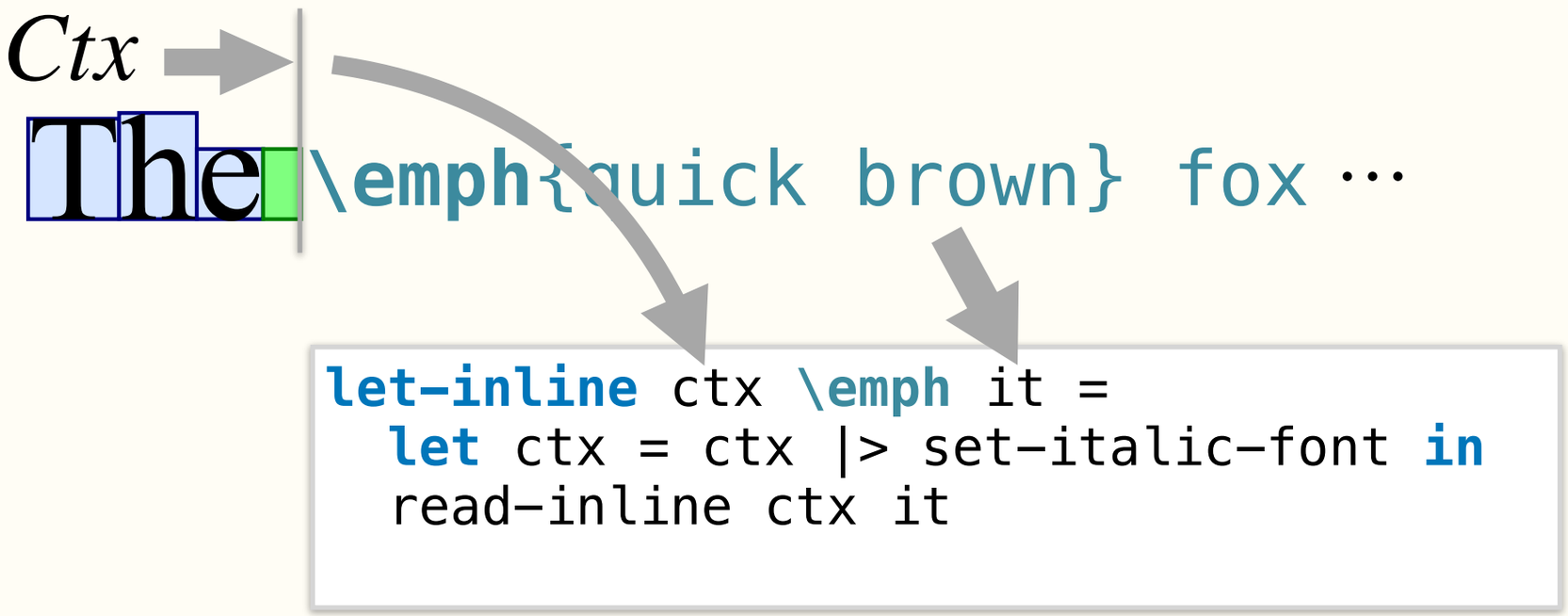
```
let-inline ctx \emph it =
  let ctx = ctx |> set-italic-font in
  read-inline ctx it
```

コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox … }
```

$\longrightarrow^{*}$ The *quick brown* fox …

という評価の途中経過を見てみる



Ctx のフォントをイタリックに更新した文脈 Ctx' がつくられ,

入れ子で `read-inline` Ctx' `{quick brown}` が評価される

コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox …}
```

$\longrightarrow^{*}$ The *quick brown* fox …

という評価の途中経過を見てみる

Ctx →

The | `\emph{quick brown} fox …`

Ctx′ →

| `quick brown`

コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox …}
```

$\longrightarrow^{*}$ The *quick brown* fox …

という評価の途中経過を見てみる

Ctx →

The \emph{quick brown} fox …

Ctx′ →

quick brown

コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox …}
```

$\longrightarrow^{*}$ The *quick brown* fox …

という評価の途中経過を見てみる

Ctx →

The `\emph{quick brown} fox …`

Ctx′ →

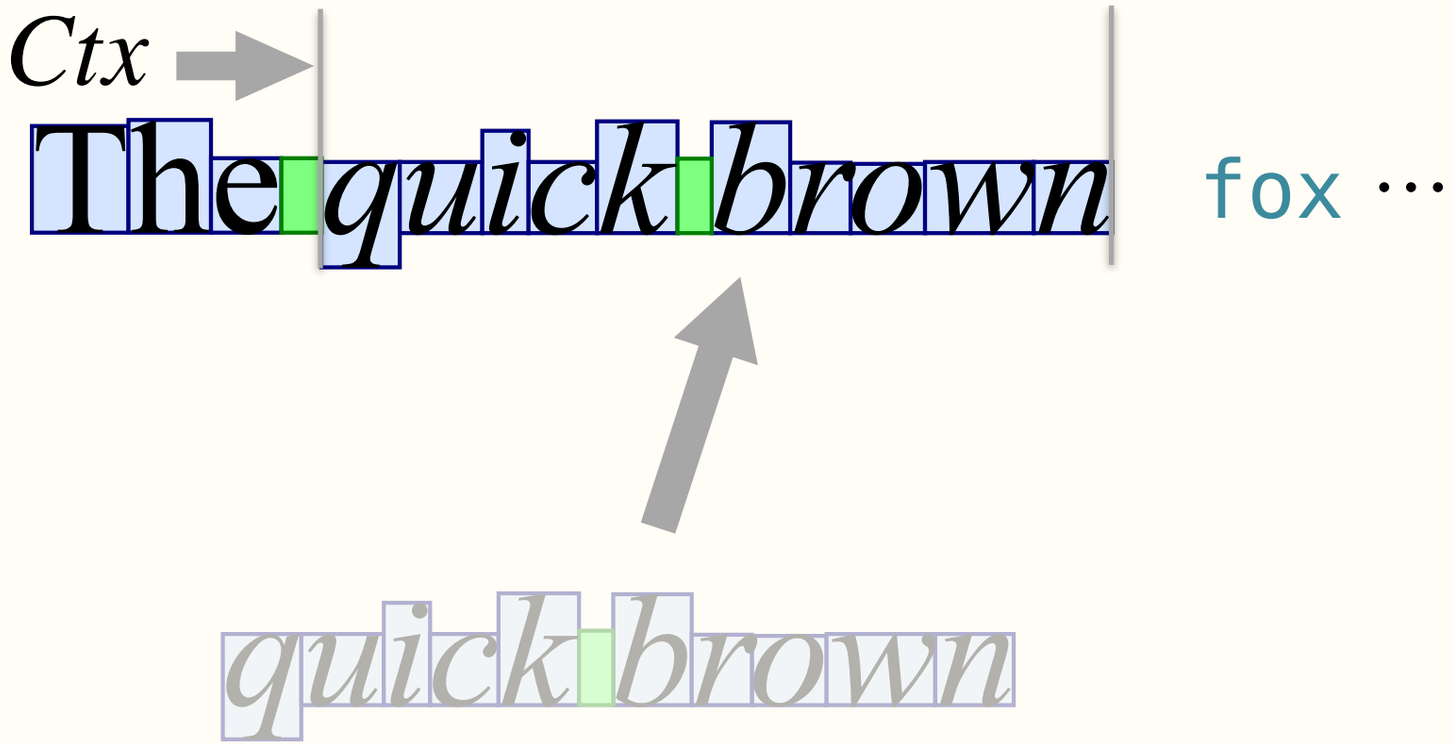
quick brown

コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox … }
```

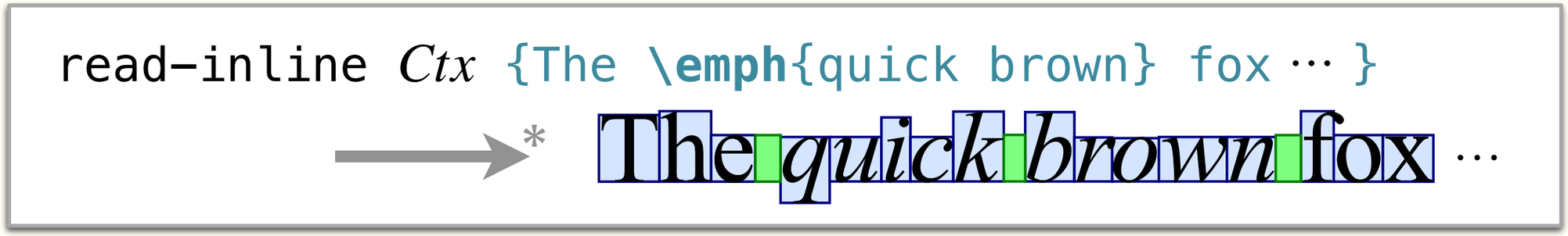
→* The *quick brown* fox …

という評価の途中経過を見てみる

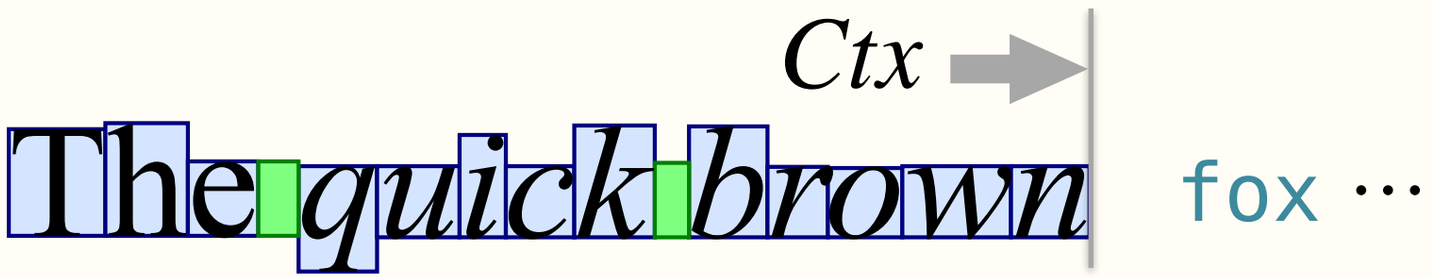


戻り値がコマンド適用のあった箇所に挿入される

コマンド適用の評価のイメージ



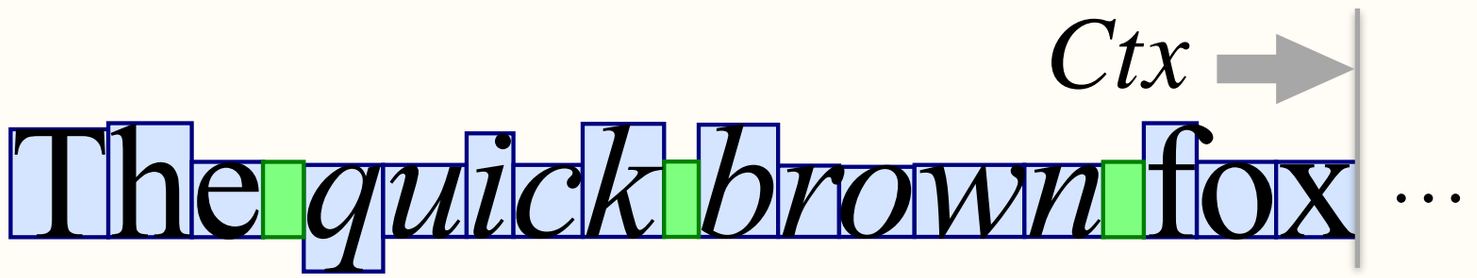
という評価の途中経過を見てみる



コマンド適用の評価のイメージ

```
read-inline Ctx {The \emph{quick brown} fox …}
```

という評価の途中経過を見てみる



コマンドの型

`\emph` は実際の SATySFi では [**inline-text**] **inline-cmd** 型がつけられる

- 前述の通り意味論的には **inline-text** → **context** → **inline-boxes** 型の函数

```
let-inline ctx \emph it =
  let ctx = ctx |> set-italic-font in
  read-inline ctx it
```

より一般に，$\tau_1, \ldots, \tau_n$ 型の引数をとる n 引数のコマンドは
$[\tau_1; \ldots; \tau_n]$ **inline-cmd** 型がつけられる

- 意味論的には $\tau_1 \to \cdots \to \tau_n \to$ **context** → **inline-boxes** 型の函数
- この型つけにより，**原稿を書く際はテキスト処理文脈を意識する必要がなくなる**

ブロックテキストの場合

ブロックテキスト $< bt >$ の要素はコマンド適用のみ：

$$bt \ ::= \ [bt]^* \qquad bt \ ::= \ \texttt{+foo}[e]^*$$

ブロックコマンドも同様に $\tau_1 \to \cdots \to \tau_n \to$ **context** $\to$ **block-boxes** 型の関数とみなせて，$[\tau_1; \ldots; \tau_n]$ **block-cmd** 型をつけて扱う

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

段落を組むコマンド **+p** は以下の形で定義でき，[**inline-text**] **block-cmd** 型

```
let-block ctx +p it = …
```

（**+p** の実装方法は次節で後述）

ここまでのまとめ

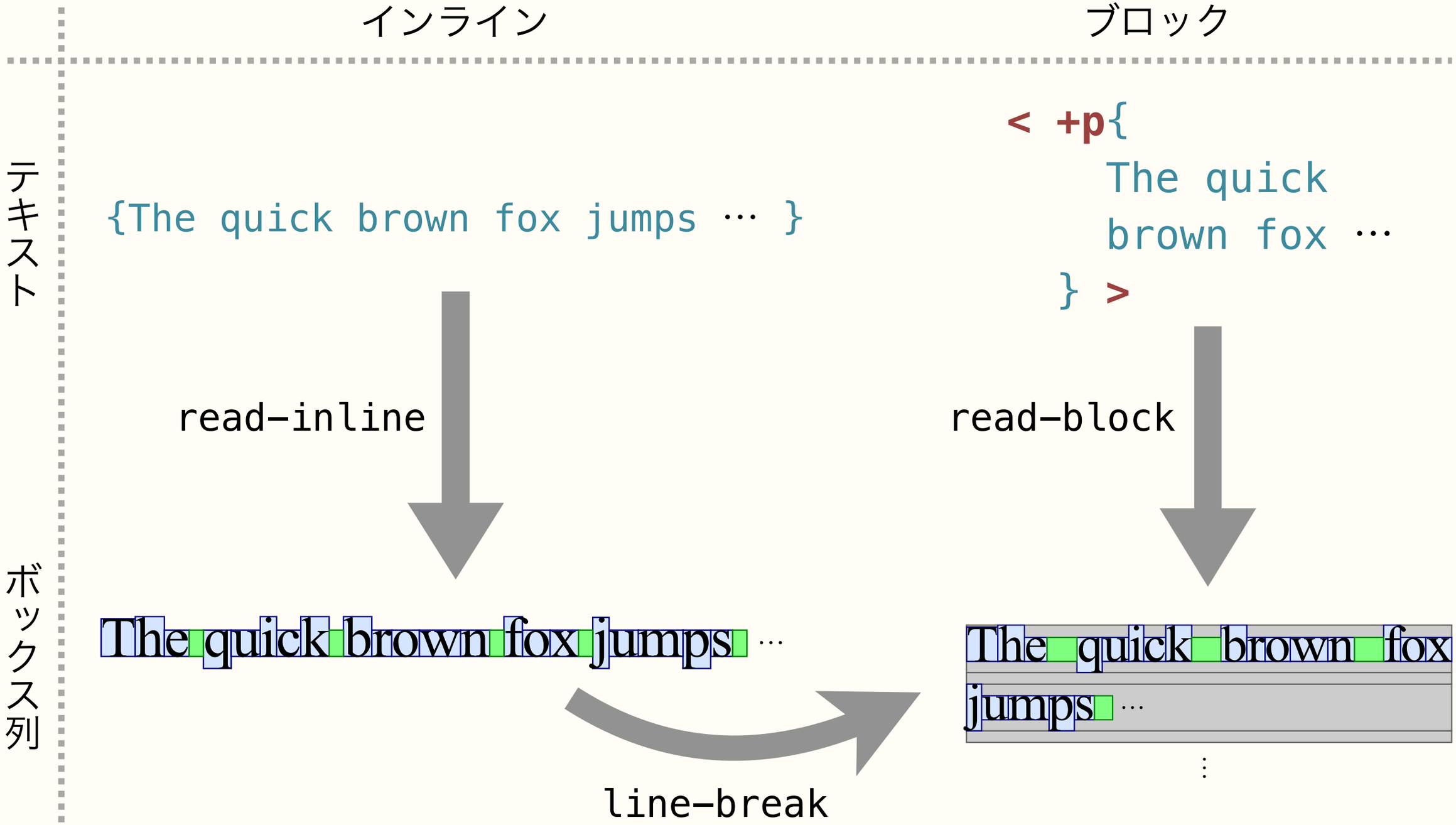
- **インラインテキスト**，**ブロックテキスト**，**テキスト処理文脈**を追加：

$$
\begin{array}{rcl}
v & ::= & \cdots \mid \boldsymbol{ib} \mid \boldsymbol{bb} \mid \texttt{\{}\ it\ \texttt{\}} \mid \texttt{<}\ bt\ \texttt{>} \mid Ctx \\
\boldsymbol{it} & ::= & [it]^* \qquad it ::= char \mid \texttt{\textbackslash foo}[e]^* \\
\boldsymbol{bt} & ::= & [bt]^* \qquad bt ::= \texttt{+foo}[e]^*
\end{array}
$$

$$
\begin{array}{rcl}
\tau & ::= & \cdots \mid \textbf{inline-boxes} \mid \textbf{block-boxes} \mid \textbf{inline-text} \mid \textbf{block-text} \\
& & \mid \textbf{context} \mid \textbf{length} \mid \left[[\tau]^*\right]\ \textbf{inline-cmd} \mid \left[[\tau]^*\right]\ \textbf{block-cmd}
\end{array}
$$

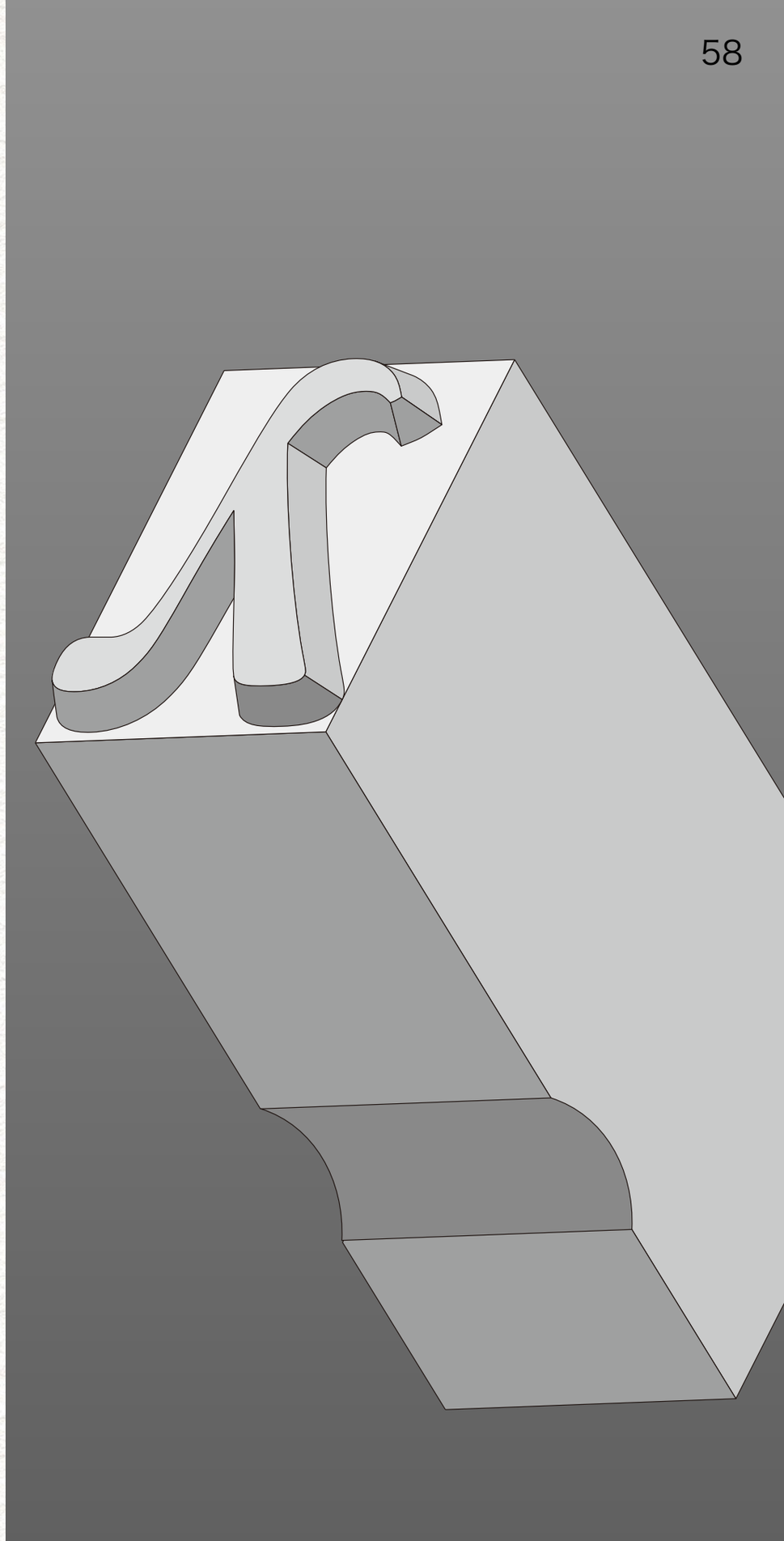
- 組込み函数
  - `line-break` : $\tau_1 \to \cdots \to \tau_n \to$ **inline-text** $\to$ **inline-boxes**
  - **`read-inline`** : **context** $\to$ **inline-text** $\to$ **inline-boxes**
  - **`read-block`** : **context** $\to$ **inline-text** $\to$ **inline-boxes**
  - テキスト処理文脈の getter と setter
    - **`get-font-size`** : **context** $\to$ **length**

ここまでのまとめ

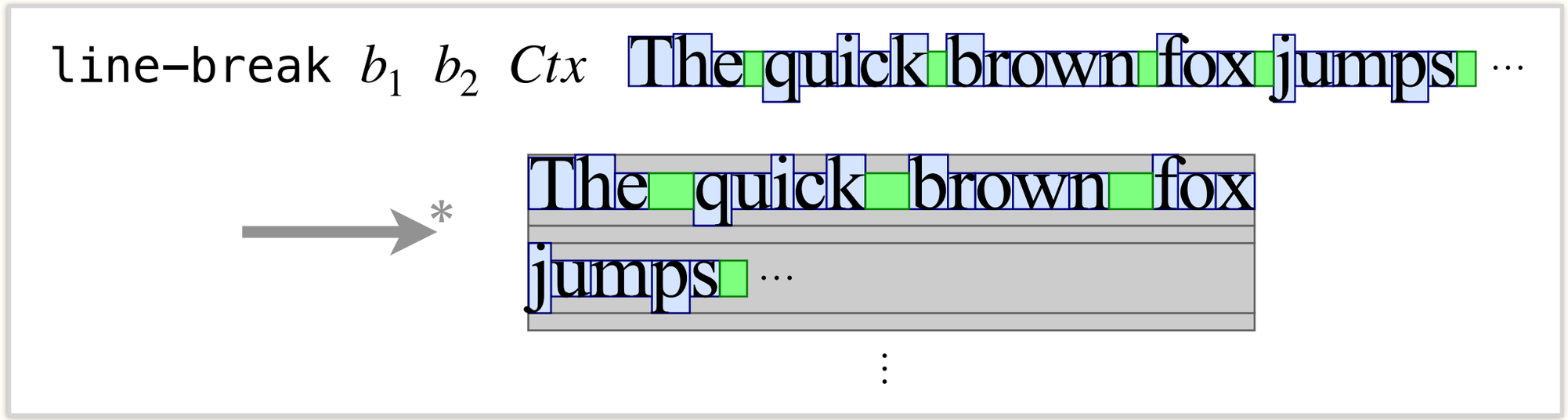

インライン
ブロック
テキスト
ボックス列
{The quick brown fox jumps … }
< +p{
The quick
brown fox …
} >
read-inline
read-block
The quick brown fox jumps …
The quick brown fox
jumps …
line-break

目次

- 開発動機・目的意識
- 動作デモ
- 文字組版の定式化とその型つけ
- 文書のマークアップ方法の定式化
- **行分割処理の仕組み**
- 多段階計算ベースのマクロ機構による DSL
- 有志の方々による周辺ツール・パッケージ
- まとめ



実際の行分割処理の組込み函数



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

- b_1, b_2： ２ つの真偽値
  - 行分割処理自体と関わる設定値ではないので今回は触れない
- Ctx： テキスト処理文脈
  - 行長など，行分割処理に必要な設定値はここから取り出して使う

行分割処理の仕組み

行分割処理問題

- **入力**： インラインボックス列 $\boldsymbol{ib}$, 所望の行長 ℓ
  - 他のパラメータは瑣末なので今回は捨象する
- **出力**： $\boldsymbol{ib}$ を空白を適切に伸縮するなどして各行の長さが ℓ になるように切り分けてできるブロックボックス列のうち**最も見映えの良いもの**
  - 見映えの良さは何らかの評価方法で適切に定義する必要あり

※ “前から順に詰めていき，溢れたらそこで行分割” という貪欲法では一般には最良の結果にはならない

SATYSFI の行分割処理では，TEX のために提案された **Knuth–Plass アルゴリズム** [Knuth & Plass 1981] の定式化を踏襲

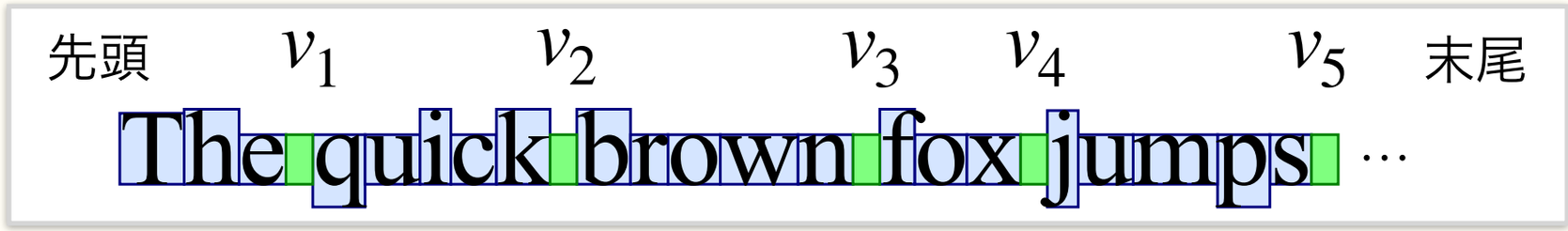
- 多少の独自拡張あり

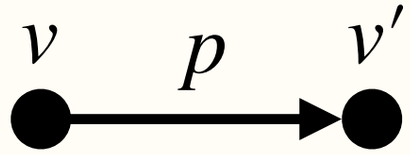
Knuth–Plass アルゴリズム [Knuth & Plass 1981]

行分割処理問題を**重みつき有向非巡回グラフ上の最短経路問題に帰着する**

内容 ***ib*** と行長指定 ℓ から $G = (V, E; d)$ を構成

- 頂点集合 $V :=$ (***ib*** 中の行分割可能な箇所全体) $\uplus$ {先頭, 末尾}
  - 模式図：



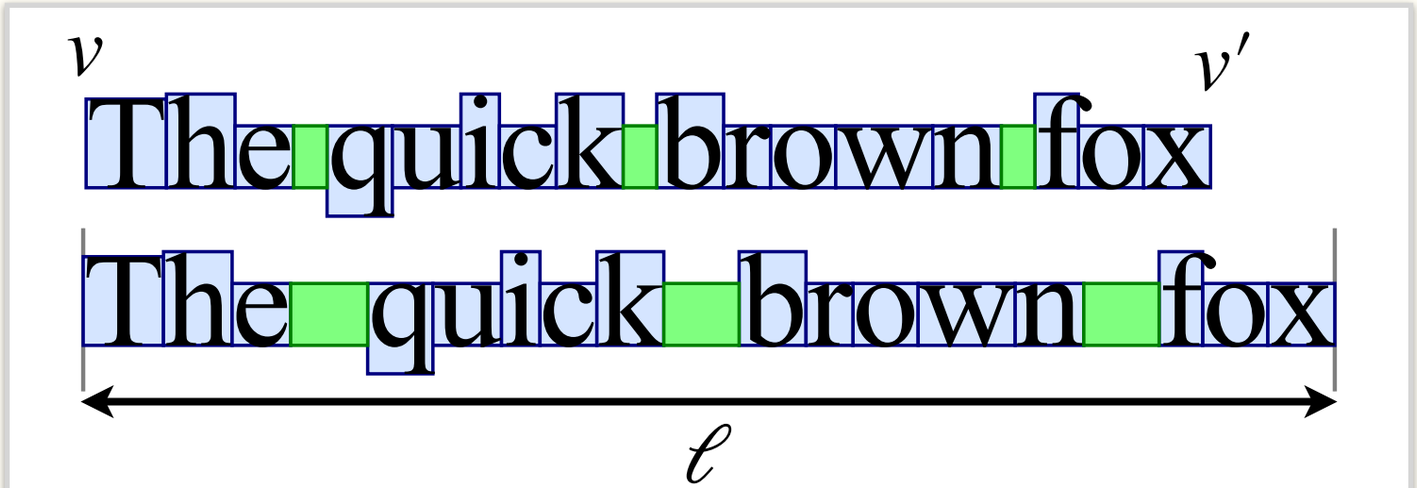
- 辺 

（***ib*** 内で v は v' よりも手前の点）
  - “v で行分割してその次に v' で行分割したとき，v と v' の間の内容を長さが ℓ になるように 1 行として組むと，その見映えの評価は p”
    - p は大きいほど見映えが悪いことを表すペナルティ値（決定方法は次頁で説明）

G の「先頭」から「末尾」へ至る最短経路を求めると，それが通る各頂点がペナルティ総和を最小化する（＝最も見映えの良くなる）切り分け箇所

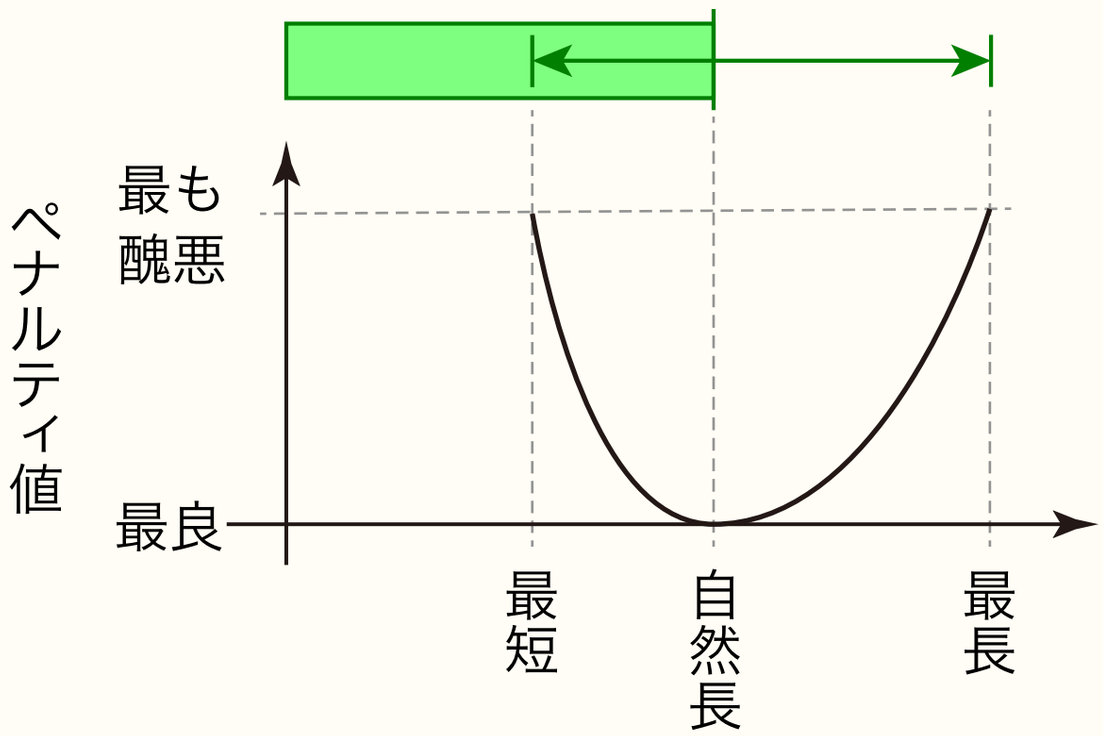
1 行の見映えの評価方法 [Knuth & Plass 1981]

行分割候補箇所 v とその次の候補箇所 v' を決めたら，空白を伸縮して
v と v' の間の内容が長さ ℓ になるようにし，その見映えを評価



見映え評価のために，空白は
(自然長, 伸びる許容量, 縮む許容量)
という 3 つのパラメータを持っている

- バネのようなイメージ
- 自然長に近いほど見映えが良い
  ことを表す，下に凸な評価函数をもつ



一般化された行分割可能箇所の指定方法

実際の行分割可能箇所は単語間の空白だけとは限らず,
もう少し一般化された仕組みになっている

$$ib ::= [ib]^*$$

$$ib ::= \text{e} \mid \cdots \mid \text{あ} \mid \cdots$$ グリフ

$$\mid \ \ $$ 伸縮する空白

$$\mid \left\{ \begin{matrix} ib \\ ib \ / \ ib \end{matrix} \right\}_p$$ discretionary break

$$\vdots$$

$$\left\{ \begin{matrix} ib_0 \\ ib_1 \ / \ ib_2 \end{matrix} \right\}_p \quad : \quad$$ “ここで行分割するかしないか選べる.
行分割しないなら ib_0 と等価であり,
一方ここで行分割するなら手前の行末に ib_1 を,
後方の行頭に ib_2 をそれぞれ結合し，かつ
分割そのものによりペナルティ p が生じる”

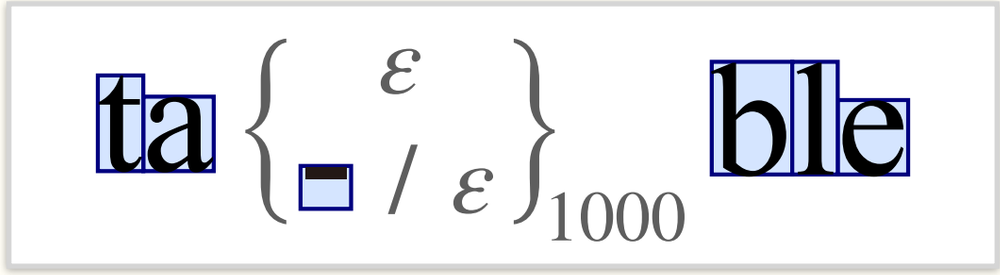
単語間の空白は $\left\{ \begin{matrix} \ \ \\ \varepsilon \ / \ \varepsilon \end{matrix} \right\}_{100}$ と表現される

ε ： 空列

Discretionary break による種々の行分割の表現

discretionary break の仕組みにより様々な行分割方法が表現できる：

- **ハイフネーション**による単語中での行分割：



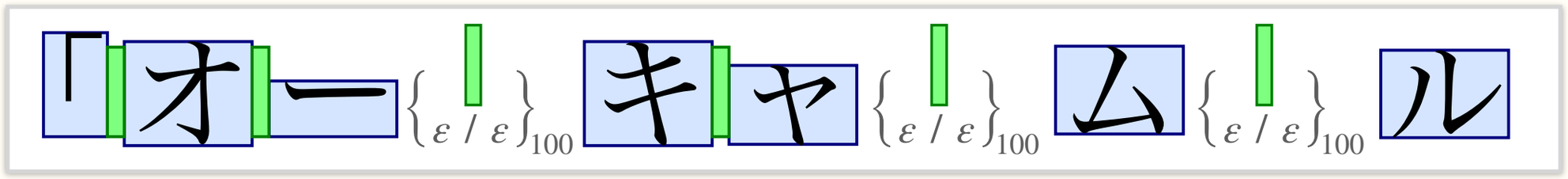
  - **単語中の一部箇所は行分割してよく，分割時は手前の行末にハイフンが入る**
    - 例：“table” は “ta-ble” と分割してよいので上図のように扱う
    - 英語だと分割してよい箇所は単語ごとに決まっているが，愚直に表引きせずとも **Liang–Knuth アルゴリズム** [Liang 1983] という推定方法が使える
  - **ハイフネーションは単語間での分割よりも好ましくないということを反映し，ペナルティを高めに設定**
  - 実際は単語中で分割するかしないかで合字にするかやカーニングするかが変わりうるのでもう少し複雑な仕組み

Discretionary break による種々の行分割の表現

- 和文組版
  - 以下のような規則におおよそ準拠
    - 『**日本語組版処理の要件**』，通称 **JLreq** [W3C 日本語組版タスクフォース 2012]
    - **Unicode Line Breaking Algorithm** [Unicode Consortium 2017]
  - 大抵の文字間は行分割してよい

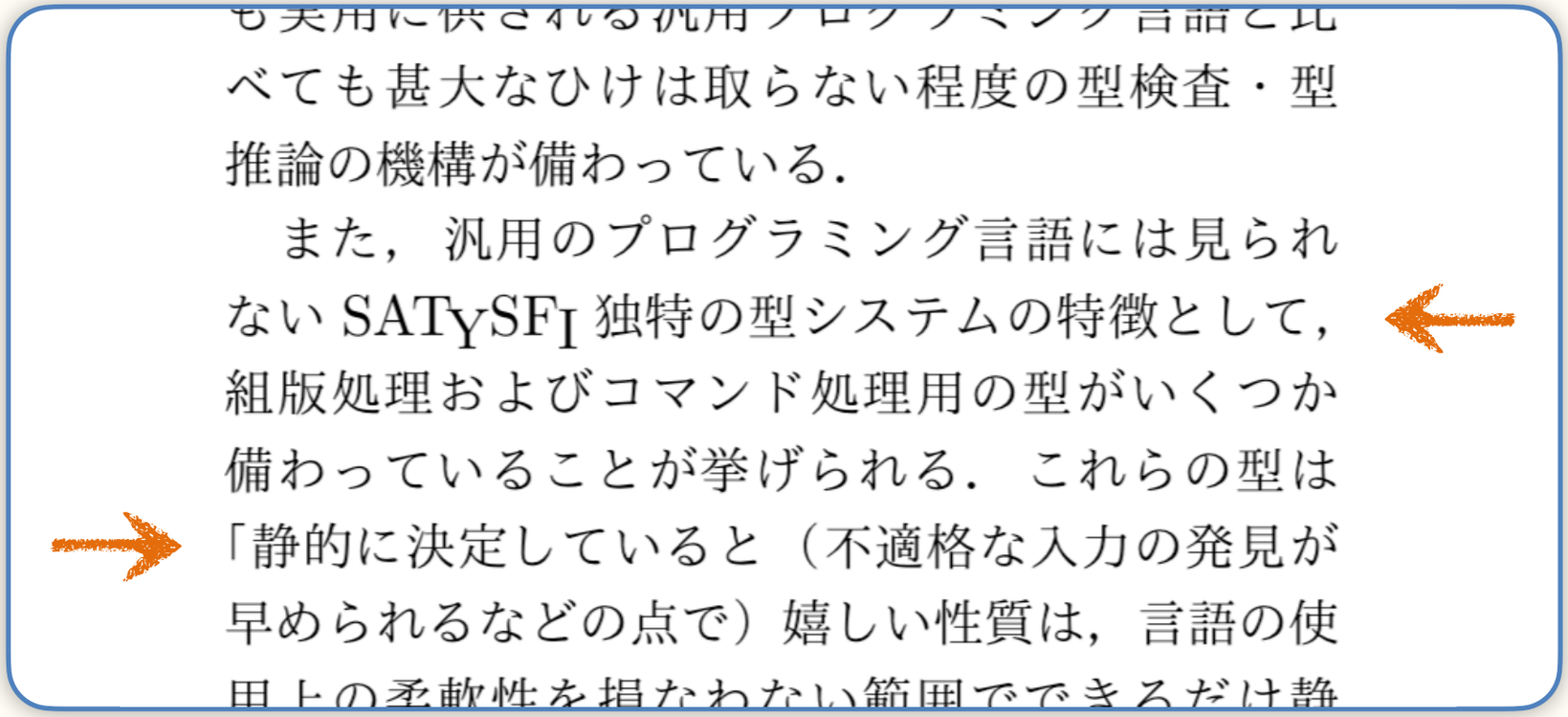
    ➡ 原則として**各文字間に自然長 0 の空白をもつ discretionary break が入る**

  - ただし，行頭/行末禁則はある
    - 行頭不可の文字： 閉じ括弧，句読点，「ゃ」，「ー」など
    - 行末不可の文字： 開き括弧など

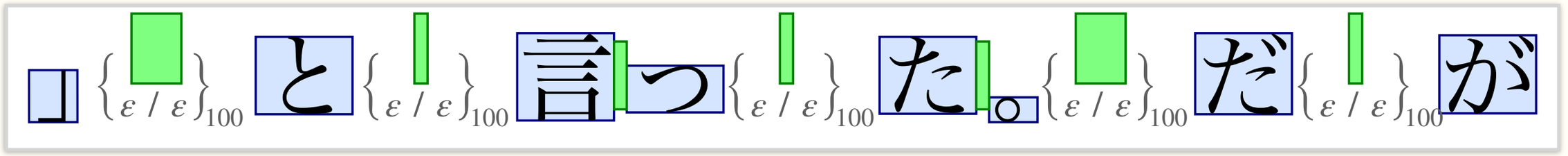


Discretionary break による種々の行分割の表現

– 行頭の開き括弧の直前や，行末の閉じ括弧や句読点の直後は，
行中とは異なり二分アキ（＝いわゆる半角幅の空白）を入れない



➡ 該当箇所は自然長が二分アキの空白をもつ discretionary break になる



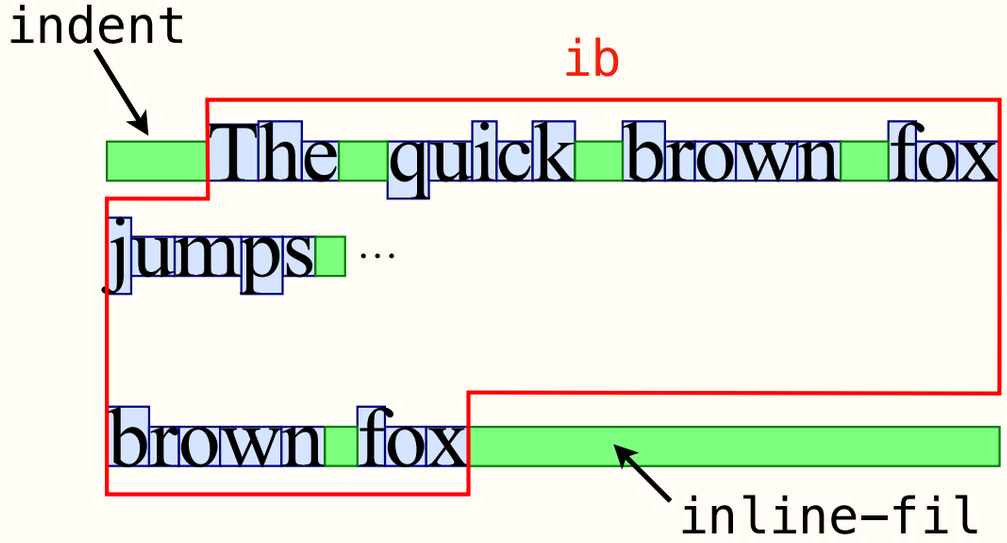
行分割処理の利用例

段落を組むコマンド **+p** : [**inline-text**] **block-cmd** は
組込み函数 `line-break` を用いて実装：

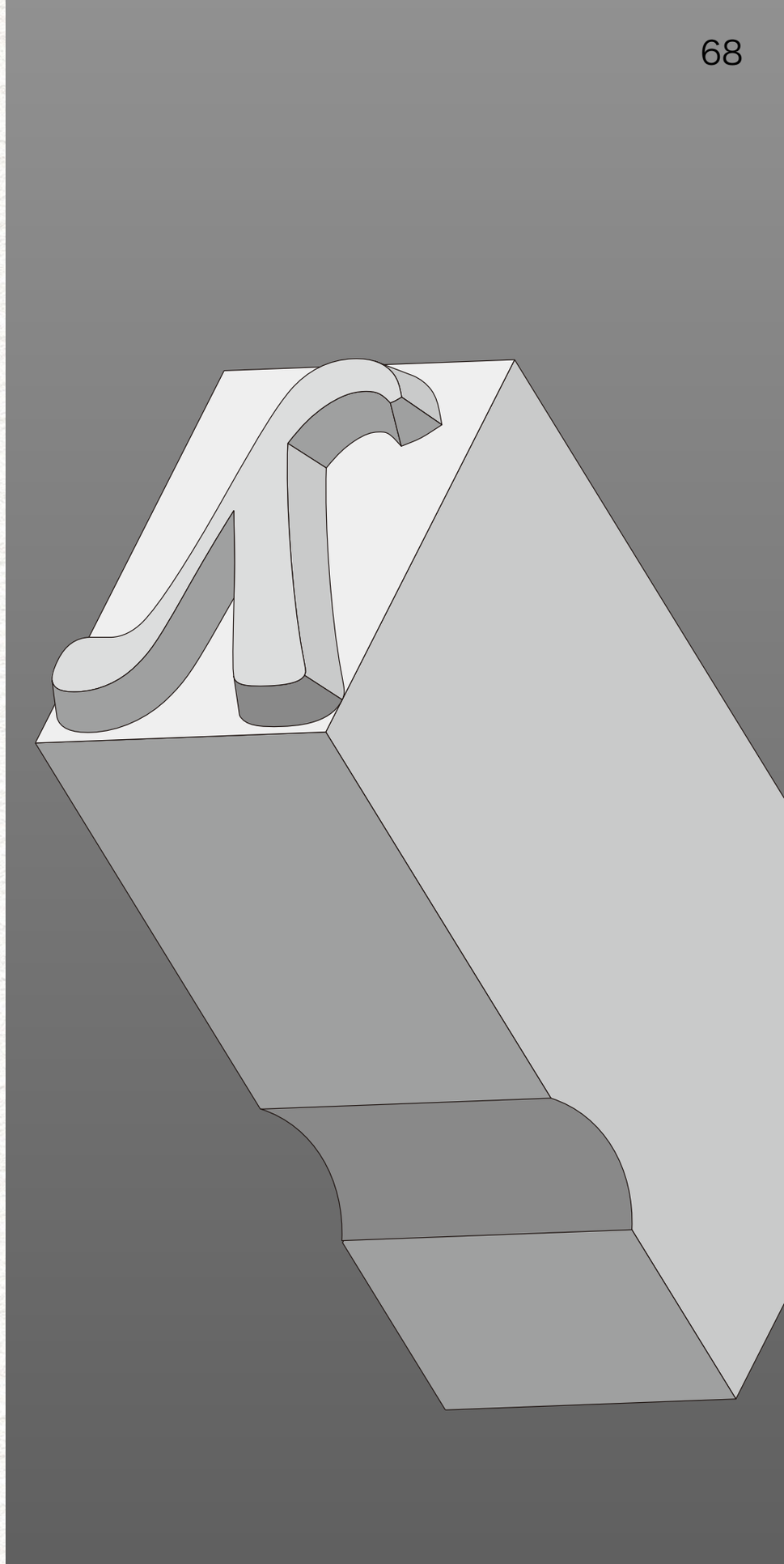
```
let-block ctx +p it =
  let ib = read-inline ctx it in
  let indent = inline-skip (get-font-size ctx) in
  line-break true true ctx (indent ++ ib ++ inline-fil)
```

他の組込み函数：

- **inline-skip** : **length → inline-boxes**
  - 指定した長さの伸縮できない空白を返す
  - ここでは段落冒頭の字下げに使用
- **inline-fil** : **inline-boxes**
  - 自然長が 0 で，かつペナルティなくどこまでも伸びられる特殊な空白
  - ここでは「段落最終行は行末まで到達しなくてよい」ことの実現に使用

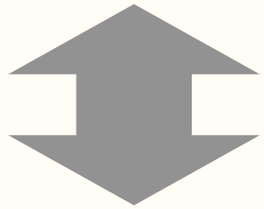


目次



文書記述上の更なる問題意識

- 静的に型がつくおかげで迅速で多くの場合親切なエラー報告が行なわれ，或る程度使いやすいものになった
  - とりわけパッケージの実装にはなかなか威力を発揮
    - TEX/LATEX と SATYSFI 両方で熱心にパッケージを書かれる方に（TEX と比べて）「プログラミング言語としては明らかに楽」という感想を貰えた



- 一方で，**型つけの都合上どうしても原稿の記述が煩雑になってしまう**場面が出てくる
  - 特に文書作成の用途では（汎用の計算機言語以上に）複雑なデータも簡潔に記述できることを望まれがち

記述複雑化に対するナイーヴな回避策

言語内 DSL の導入

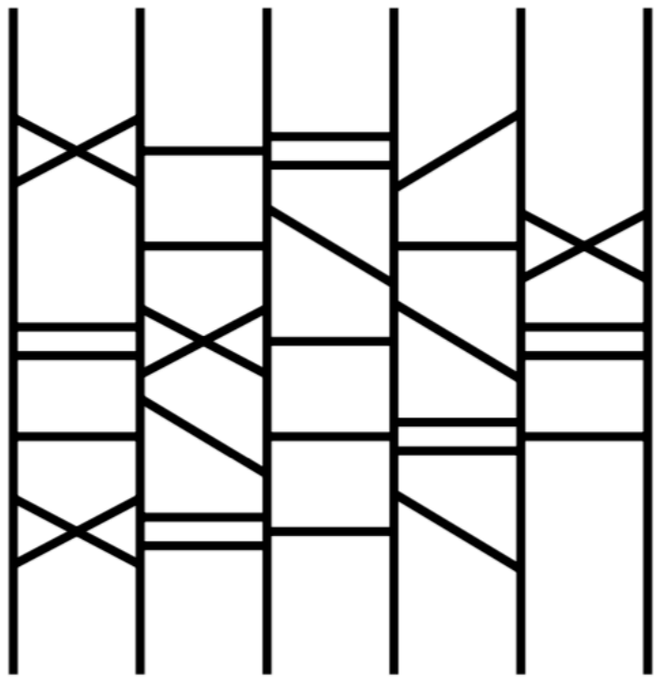
- 文字列を受け取って構文解析するようにコマンドを定義する
- 簡単な例： あみだくじの記述をアスキーアートでできるようにする

DSL方式

```
\amidakuji(```
   |X|-|=|/| |
   | |-|\|-|X|
   |=|X|-|\|=|
   |-|\|-|=|-|
   |X|=|-|\| |
```);
```

通常の実装

```
\amidakuji([
   [Cross; Line; … ];
   [Empty; Line; … ];
   [Double; Cross; … ];
   …
]);
```


```

ナイーヴな回避策の問題点とその解決策

DSL の解釈は動的に（＝組版処理開始後に）**行なわれるため,**
エラーがあった場合に発見が遅い

- DSL を使っただけで動的にエラーが出る世界に逆戻り

DSL の解釈は組版処理に依存することなく可能なので,
言語本体にマクロ機構を導入して前処理できるようにすればよい

しかし安易にマクロ機構を入れると今度はエラーが不親切になりやすいので,
できるだけ原因のわかりやすいエラーが出る仕組みにしたい

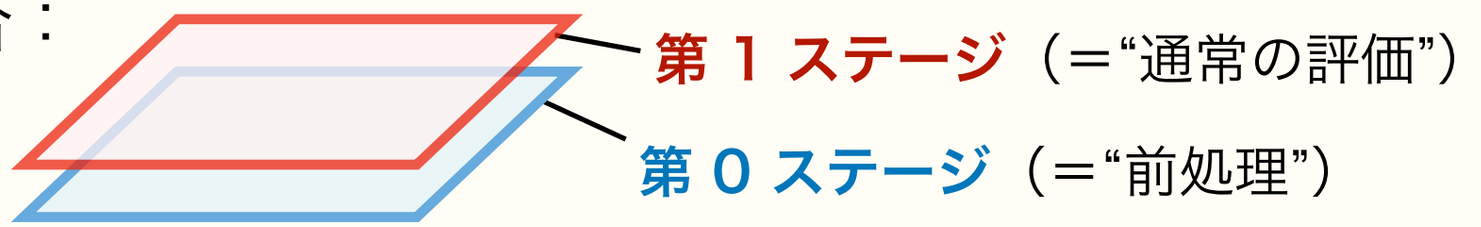
多段階計算 (*multi-stage programming*) に基づくマクロ機構を導入しよう

多段階計算とは

- 動的にコードを生成して更にそれを実行するような機構を備えた,
  計算が複数の**ステージ** (*stage*) からなる体系
  ＋**そのような操作を安全に行なうための型システム**

- 例： **MetaML** [Taha & Sheard 1997], **λ○** [Davies 1996],
  MetaOCaml [Kiselyov 2014], 他多数

- 特に 2 ステージの場合：



- 詳細は明日！：

11:05-12:15 セッション16 多段計算・プログラム合成 (座長：佐藤 重幸（東京大学）)

Synbit: Synthesizing Bidirectional Programs using Unidirectional Sketches [C2]
Masaomi Yamaguchi(1), Kazutaka Matsuda(1), Cristina David(2), Meng Wang(2)((1)Graduate School of Informatic

複数ステージの値が同一ストラクチャのメンバとして共存できる多段階計算のためのモジュールシステム [C1]
諏訪 敬之(1)((1)（民間企業）)

Stage-Aware Equality Types for a Dependently-Typed Multi-Stage Calculus [C1]
勝田 峻太朗(1), 五十嵐 淳(1)((1)京都大学)

マクロ機構の言語設計

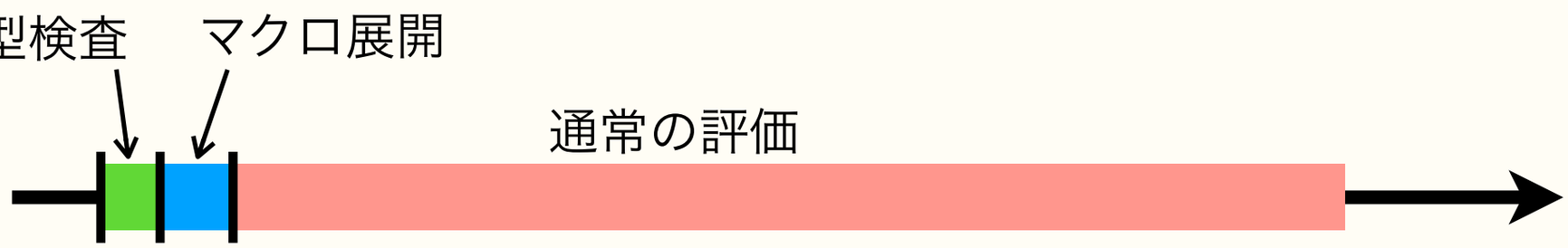
2 ステージの MetaML [Taha & Sheard 1997] をベースにした
MacroML [Ganz, Sabry & Taha 2001] に近いマクロ機構の定式化を採用

- マクロの使用例：

```
\repeat@~(3){\foo{A}}
% 通常の評価が始まるタイミングでは，最初から
% \foo{A}\foo{A}\foo{A} と書いてあったのと同じ扱いになる
```

  - **~(…)** はマクロ展開中に使われる第 0 ステージの値の引数
  - ~ なしの引数は展開後のコードの一部となる

- マクロの定義自体が展開より前に型検査され，型がつけば
  展開結果は必ず well-typed
- 典型的にはマクロ展開（＝コード生成）は一瞬で終わるので，
  マクロ展開中に panic すれば “事実上静的に” エラーが報告される



マクロ機構による DSL 処理

第 0 ステージの引数として DSL を記述する：

```
\amidakuji@~(@```
   |X|-|=|/| |
   | |-|\|-|X|
   |=|X|-|\|=|
   |-|\|-|=|-|
   |X|=|-|\| |
```~);
```

展開

```
\amidakuji([
   [Cross; Line; … ];
   [Empty; Line; … ];
   [Double; Cross; … ];
   …
]);
```

- マクロ展開時に DSL を処理し，構文エラーがあれば panic する
  - エラーは “事実上静的に” に出る
  - DSL コード中のどこが原因でエラーが起きたかをマクロ内から
    わかりやすく報告できるように，先頭位置がどのファイルの何行目か
    自動で補われる文字列リテラル `@`…` を使う
- 正常に展開された場合，結果のテキストは評価時に通常どおり処理
```

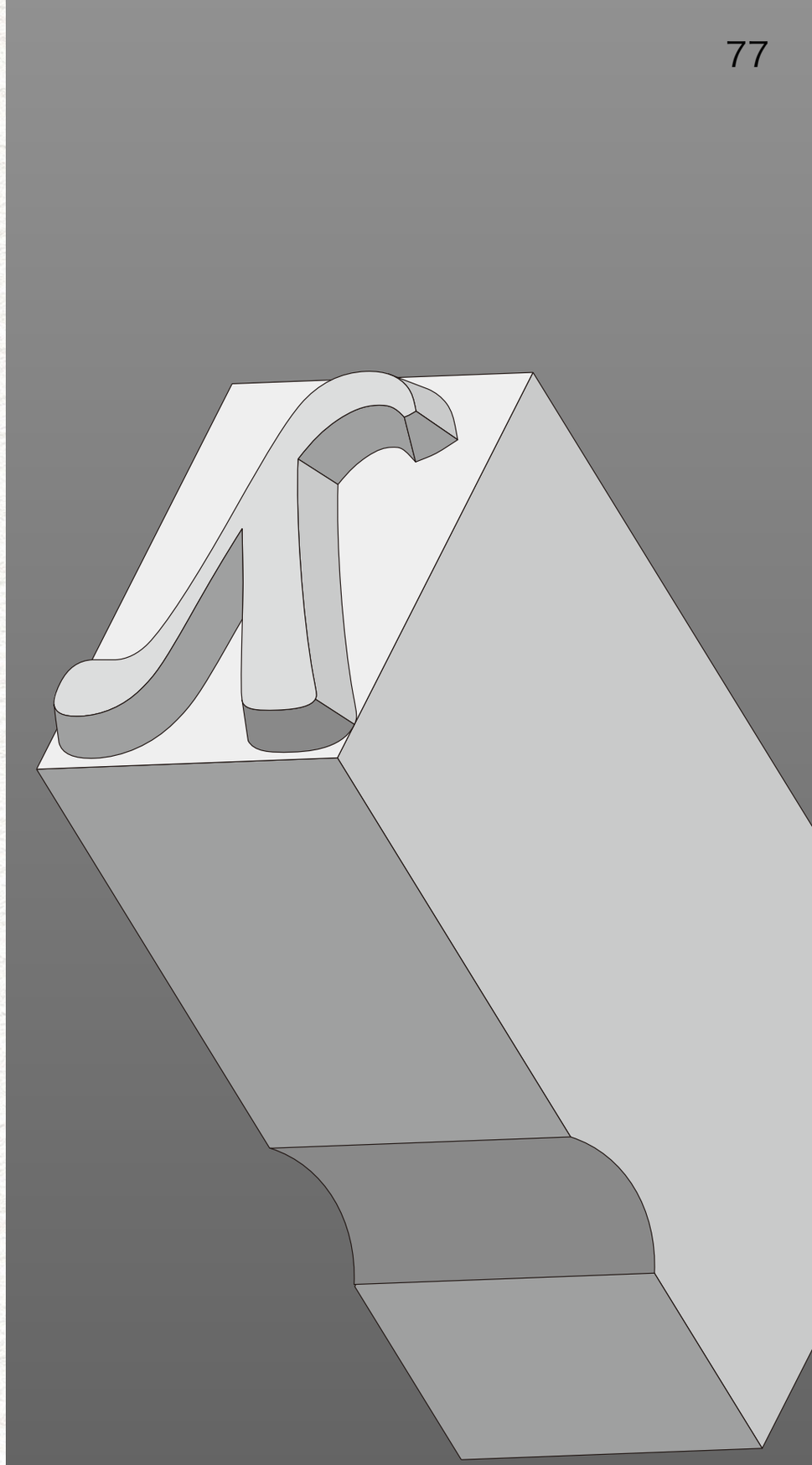
マクロ機構の動作デモ１： 正常系

（デモ映像： https://drive.google.com/file/d/1l9HeT6Hn_lNn0ptVobVDYwW6T9A8xKvZ/view?usp=sharing）

マクロ機構の動作デモ２： 異常系

（デモ映像： https://drive.google.com/file/d/1pbTOSWiflEkTghS3TzlerG5NCcb2iIoI/view?usp=sharing）

目次



有志の方々によるツール・パッケージ

ありがたいことに有志の方々によって
種々の周辺ツールやパッケージの開発が進行中

- SATYSFI 用パッケージマネージャ **Satyrographos** [Sakamoto 2018]
  - github.com/na4zagin3/satyrographos
  - OCaml のパッケージマネージャである OPAM と併用する設計
  - フォントや設定ファイルも管理可能

- 登録されているパッケージ一覧が見られるページ [Matsunaga 2020]
  - satyrographos-packages.netlify.app
  - 現在約 50 パッケージ

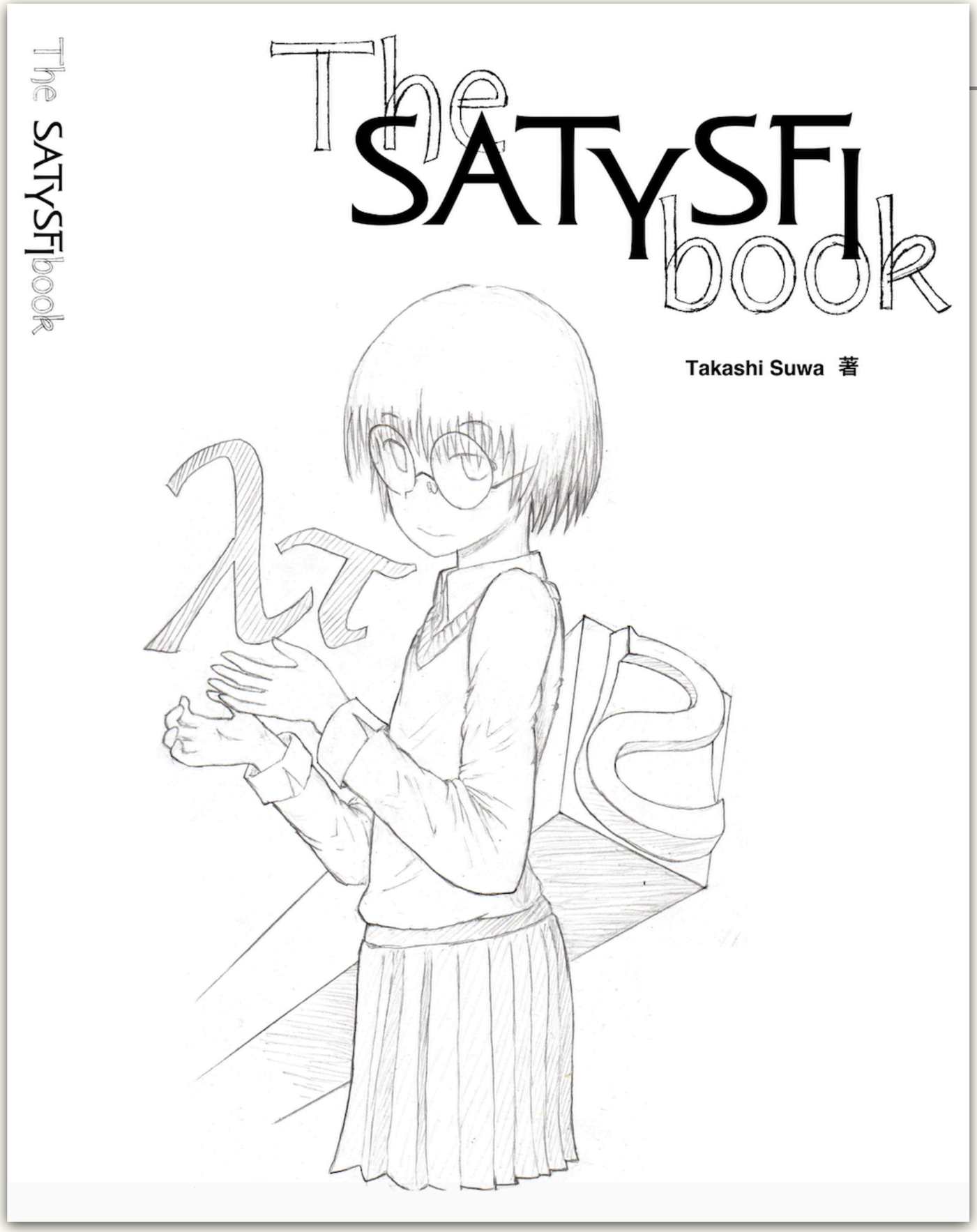
- **satysfi-base** [Nishiwaki, Murase & Kaneko 2019]
  - github.com/nyuichi/satysfi-base
  - 標準ライブラリの大幅拡充
  - パーサコンビネータなども用意されている

有志の方々によるツール・パッケージ

- **easytable** [monaqa 2020] github.com/monaqa/satysfi-easytable
  - 表組みを簡潔に実現するためのパッケージ
- **matrixcd** [Abe 2020] github.com/abenori/satysfi-matrixcd
  - LaTeX の tikz-cd のように可換図式を描くためのパッケージ
- **make-latex** [Kaneko 2020] github.com/puripuri2100/SATySFi-make-latex
  - **テキスト出力モード**により，SATySFi ソースを LaTeX ソースに変換して出力するためのパッケージ
    - 言語側で扱えないことにより数式は未対応
- **Steamer** [Ishii 2018], **SLyDIFi** [monaqa 2019]
  - github.com/konn/satysfi-steamer
  - github.com/monaqa/slydifi
  - いずれもスライド作成用のクラスファイル

他にも数多くのツールやパッケージが公開されています

宣伝



B5 判 / 160 ページ

無償公開中：

https://booth.pm/ja/items/1127224

satysfi book　検索

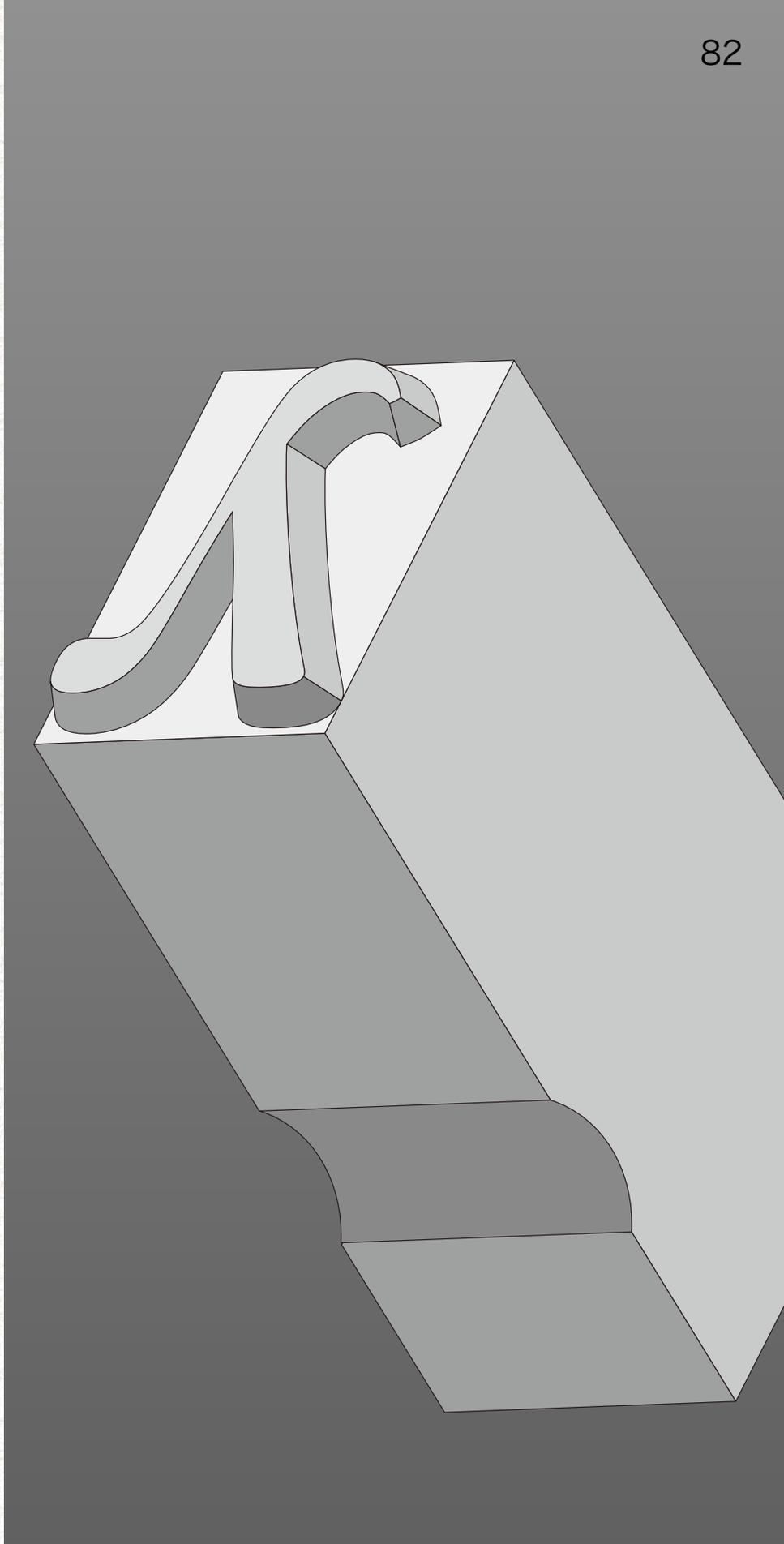
謝辞

以下の方々に深く感謝申し上げます：

- 開発を支援頂いた皆様
  - 2017 年度 IPA 未踏 IT 人材発掘・育成事業
  - 株式会社ドワンゴ（アルバイトとして）
  - The SATγSFIbook をご購入頂いた皆様
- 熱心に Pull Request や助言を下さった開発者の皆様
  - Daiki Matsunaga (`matsud224`) さん，Daichi Oohashi (`leque`) さん，
    Noriaki Sakamoto (`na4zagin3`) さん，Takuma Ishikawa (`nekketsuuu`) さん，
    `elpinal` さん，Masaki Hara (`qnighy`) さん，`sakas--` さん，他多数名
- パッケージ・周辺ツール開発者の皆様
- **SATγSFI Slack**・**SATγSFIConf** 等，コミュニティを運営して下さる皆様
  - Naoki Kaneko (`puripuri2100`) さん，`monaqa` さん，他多数名

目次

- 開発動機・目的意識
- 動作デモ
- 文字組版の定式化とその型つけ
- 文書のマークアップ方法の定式化
- 行分割処理の仕組み
- 多段階計算ベースのマクロ機構による DSL
- 有志の方々による周辺ツール・パッケージ
- **まとめ**



まとめ

函数型の言語設計をベースとする,
静的型つきの組版処理システム SATγSF| について紹介しました

- 意味論上の主な特徴：
  - ボックス列によるグリフなどの取り扱い
  - テキストと文脈をボックス列に変換する函数としてのコマンド
  - 種々の組込み函数で制御できる行分割処理
  - 言語内 DSL を使用してもエラー報告を迅速かつ明快に保つための,
    多段階計算に基づくマクロ機構
- ありがたいことに有志の方々により周辺ツールやパッケージも
  精力的に開発されている

参考文献

- Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proc. of LICS'96*, pp. 184–195, 1996.
- Sebastian Erdweg and Klaus Ostermann. Featherweight TeX and parser correctness. In *SLE 2010*. *LNCS*, vol. 6563, pp. 397–416, 2011.
- Steve Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *Proc. of ICFP'01*, pages 74–85, 2001.
- Benedict Gaster and Mark Jones. A polymorphic type system for extensible records and variants, 1996.
- Oleg Kiselyov. The design and implementation of BER MetaOCaml. In *Functional and Logic Programming*. *FLOPS 2014. Lecture Notes in Computer Science*, vol 8475, 2014.
- Donald Knuth and Michael Plass. Breaking paragraphs into lines. *Software–Practice and Experience*, 11, pages 1119–1184, 1981.
- Franklin Liang. *Word Hy-phen-a-tion by Com-put-er*. Ph. D. thesis, Stanford University, 1983.
- Andres Löh and Ralf Hinze. Open data types and open functions. In *Proc. of PPDP'06*, pp. 133–144, 2006.
- Atsushi Ohori. A polymorphic record calculus and its compilation. In *ACM Transactions on Programming Languages and Systems*, 17(6), pages 844–895, 1995.
- 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.
- Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248 (1-2), pp. 211–242, 2000.
- Unicode Consortium. *Unicode Standard Annex #14: Unicode Line Breaking Algorithm (Unicode 10.0.0)*. http://unicode.org/reports/tr14/, 2017.
- 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.