コードのように台湾語を解析

Rubyによる白話字ローマ字の3段階解析

鄧慕凡 (Mu-Fan Teng)

RubyWorld Conference 2025

島根県立産業交流会館「くにびきメッセ」Nov. 7, 2025

自己紹介

鄧慕凡 (Mu-Fan Teng)

- 日本では竜堂 終と呼ばれています
- 5xRuby CO., LTD 創業者
- 台湾のRuby伝道師
- RubyConf Taiwan Chief Organizer
- 三度目のRubyWorld登壇（2015, 2023, 2025）

RubyCityMATSUE 縁結びの地との10年の物語

2015

🌸 **縁の始まり**
- 初めてRWCの講者として登壇
- RubyCity Matsueとの出会い

2023

💝 **縁結びの実現**
- 市長と市役所で会談
- 再びRWCの壇上へ

2024

🤝 **縁の深化**
- 上定市長の5xRuby訪問
- RubyCityとの絆が深まる

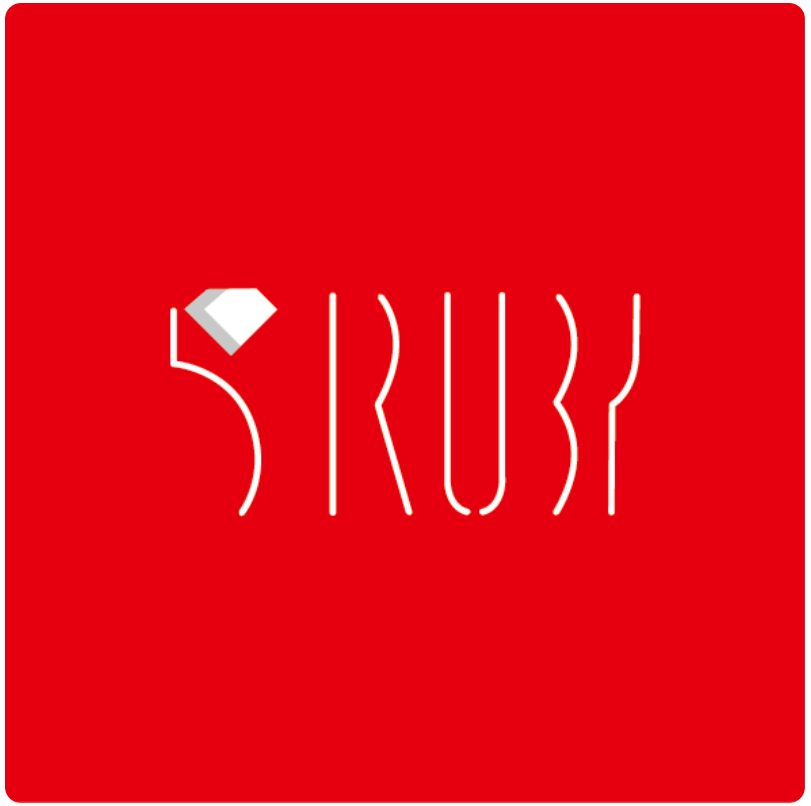
2025

💍 **縁結びの証**
- RubyConf Taiwan × COSCUP 2025 で覚書締結
- RubyCityとの正式な絆

5xRubyについて

「愛する技術で愛される製品を創る」

- **創業**: 2014年（台北）
- **専門**: Ruby/Railsを中心としたソフトウェア開発
- **実績**: スタートアップ向けシステム開発を中心に、政府機関との協業案件も手がける



5xRubyの事業

1. 委託開発サービス

- **台湾最大級のRuby開発会社**（2014年創業）
- クラウド・オンプレミス両対応のインフラ運用
- 日本・米国・シンガポールを含む国際展開
- スタートアップから上場企業まで長期パートナーシップ
- https://5xruby.com/en

2. SOSI製品

- セキュアリモートアクセス管理システム
- 踏み台サーバー（Bastion）機能
- ブラウザベース VDI ソリューション
- https://www.sosi.com.tw

アジェンダ

本日の内容

1. **無人入札の物語**
   - なぜ誰も手を出さなかったのか？
2. **台羅（POJ）とは？**
   - 台湾語のローマ字表記
3. **分詞アライメント処理の実装**
   - GSUB による実装
4. **Parser との出会い**
   - Parslet による再実装
5. **プロジェクトの成果**
   - Ruby の強みと実績
6. **まとめ**
   - 結論

スライド資料

https://rwc2025.ryudo.tw （日本語）

https://rwc2025.ryudo.tw/en （English）

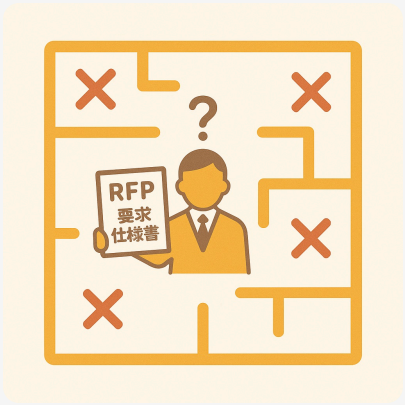
無人入札の物語

なぜ誰も手を出さなかったのか？

台湾政府案件の特殊性

技術の制約

- Microsoft製品への依存
- .NET/MS-SQL/Windows Server
- Ruby/Railsは落選しがち



プロセスの問題

- RFP（要求仕様書）の不備
- 担当者の専門知識不足
- 実務との乖離

隠れたコスト

- 膨大な文書作成業務
- セキュリティ監査・脆弱性診断
- 現地対応が必須の運用作業

8連敗からの学び

落選の理由（技術以外）

- Microsoft製品前提の仕様
- 既存システム、インフラとの「互換性要求」
- 評価基準の不透明さ
- 価格競争ではなく、技術スタックの制約

9回目：驚きの展開

- **競合：ゼロ**
- 「なぜ誰も入札しないのか？」
- 担当者も困惑：「本当に大丈夫ですか？」
- **一体何が起こったのか？**

落札後の真相

「分詞（文字分割）が煩雑すぎて
誰も手を出さない」

台羅（POJ）とは？

日本語との類似性から理解する

台羅（台湾閩南語ローマ字）とは？

前後文脈（漢字）	前後文脈（POJ）
去**日本**食壽司	khì **Jit-pún** tsiảh sú-sih
香港、澳門...、臺灣佮**日本**	Hiong-káng, Ò-mn̂g...Tâi-uân kah **Jit-pún**
的時，**日本**義工共臺灣人	ê sî, **Jit-pún** gī-kang kā Tâi-uân-lâng

台湾語のローマ字表記

- **正式名称**: 臺灣台語羅馬字拼音方案
- **略称**: 台羅 (Tâi-lô)
- **制定**: 2006年10月、台湾教育部公布
- **地位**: 台湾語の公式表記システム

中国語（北京語）ではない

- **台湾語**: 閩南語系の言語
- **特徴**:
  - 9つの声調
  - 独自の子音・母音体系
  - 鼻音化の表記
- **歴史**: 白話字 (POJ) をベースに IPA（国際音声記号）要素を取り入れて開発

日本語と台湾語の文字システム

日本語のシステム

漢字 → ひらがな

対応関係:

- 一組の語 → 一組の仮名

例:

```
生活      → せいかつ
新幹線    → しんかんせん
東京駅    → とうきょうえき
```

台湾語のシステム

漢字 → POJ

対応関係:

- 一組の語 → 一組の POJ

例:

```
紲落      → suà-lo̍h
新竹市    → Sin-tik-tshī
明仔載    → bîn-á-tsài
```

共通点: 一組の漢字 ↔ 一組の音標

→ だから「分詞アライメント処理」が必要！

実際の分詞アライメント処理例

入力データ（分詞前）：

- 漢字：`紲落來看新竹市明仔載二十六號的天氣`
- POJ：`suà-lo̍h lâi-khuànn Sin-tik-tshī bîn-á-tsài gī-tsap-lak hō ê thinn-khì`

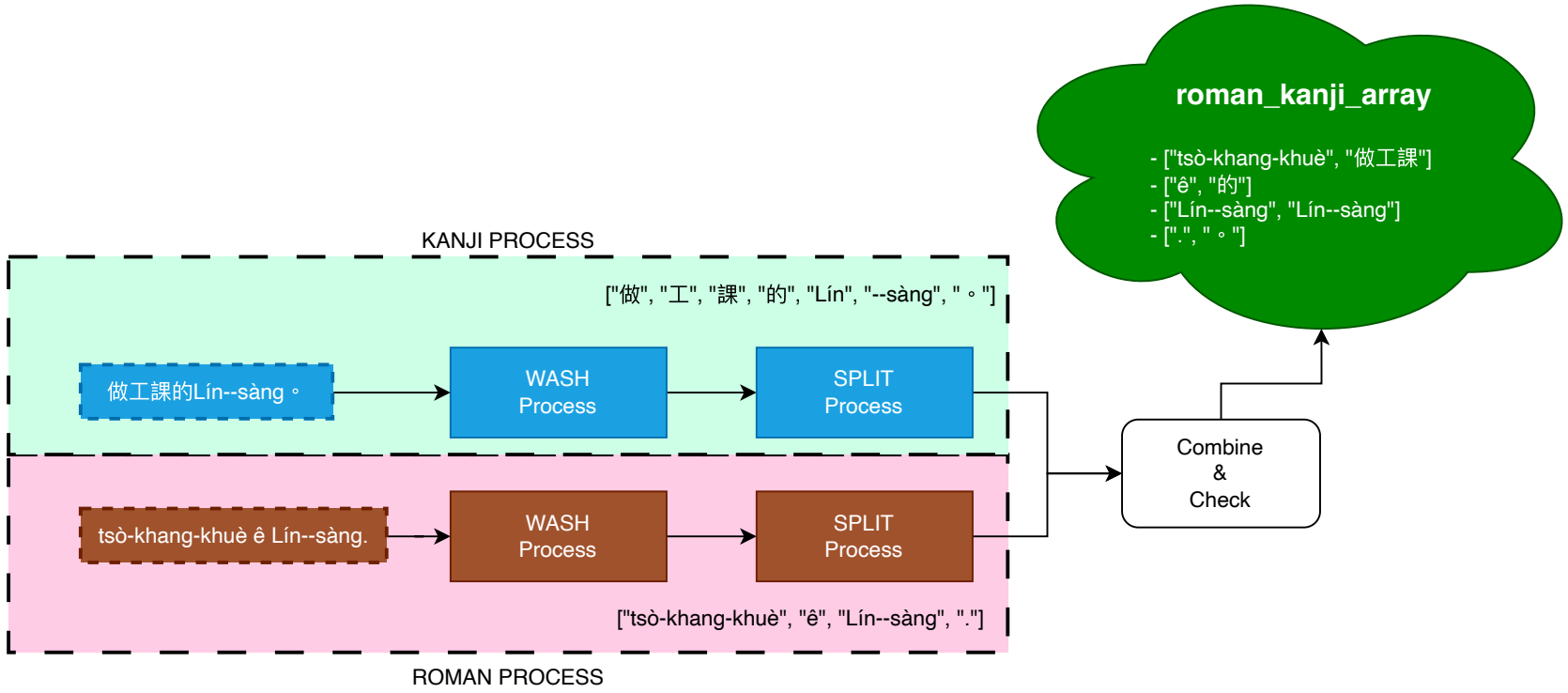
期待される出力（分詞アライメント処理後）：

漢字	POJ
紲落	suà-lo̍h
來看	lâi-khuànn
新竹市	Sin-tik-tshī
明仔載	bîn-á-tsài
二十六	gī-tsap-lak
號	hō
的	ê
天氣	thinn-khì

分詞アライメント処理の実装

3つのPhaseによる処理フロー

実装の全体フロー：3つのPhase



Phase 1: 正規化 (WASH)

washed_kanji - 漢字側

```
KANJI_GSUB_PATTERNS = {
  /(\w+)(--)(\w+)/ => '\1 \2\3',
  ')(' => ') (',
  /([^\,]),/ => '\1 ,',
  ....
}.freeze
def washed_kanji
  KANJI_GSUB_PATTERNS.reduce(kanji) do |ks, (mt, kp)|
    ks.gsub(mt, kp)
  end
end
```

処理内容:

- 記号の前後にスペース挿入
- ピリオド、カンマ、括弧などを分離
- POJ 文字（Lín--sàng）も適切に処理

実行例:

```
入力: 做工課的Lín--sàng。
出力: 做工課的Lín --sàng。
```

washed_roman - POJ側

```
ROMAN_GSUB_PATTERNS = {
  /''/ => "'",
  /(.)([_+=\:;"'~`“”\\」「\?!])(.)/ => '\1 \2 \3',
  /^([\(_+=\:;"'~`“”\\」「\?!])/ => '\1 ',
  /([\)_+=\:;"'~`“”\\」「\?!])$/ => ' \1',
  /(\.)([^\.])/ => '\1 \2',
  /([^\.])(\.)/ => '\1 \2',
}...
def washed_roman
  ROMAN_GSUB_PATTERNS.reduce(roman) do |rs, (mt, rp)|
    rs.gsub(mt, rp)
  end
end
```

処理内容:

- 65+ パターンで記号を正規化
- ✅ **ハイフンは保持** - 音節区切り
- ✅ **二重ハイフン（--）も保持** - 語間停頓

実行例:

```
入力: tsò-khang-khuè ê Lín--sàng.
出力: tsò-khang-khuè ê Lín--sàng .
```

Phase 2-1: splitted_kanji - 漢字の分割

実装コード

```ruby
# RXP_SPK - CJK文字と非CJK文字を識別
RXP_SPK = /[\p{Han}\p{Katakana}\p{Hiragana}\p{Hangul}\u3000-\u303F\uFF00-\uFFEF]|
  [^\p{Han}\p{Katakana}\p{Hiragana}\p{Hangul}\u3000-\u303F\uFF00-\uFFEF]+/x
ONE_KANJI_WORDS = {
  /(…) (。)/ => '\1\2', /(〻) (。)/ => '\1\2', /(--) ([^-])/ => '\1\2' }.freeze
def splitted_kanji
  combine_one_word(
    washed_kanji.scan(RXP_SPK).map do |spka|
      spka.split(/\s/)
    end.flatten.join(' ')
  ).split
end
# combine_one_word - 特殊組合せ処理
def combine_one_word(text)
  ONE_KANJI_WORDS.reduce(text) do |ks, (mt, kp)|
    ks.gsub(mt, kp)
  end
end
```

処理説明

1. RXP_SPK で文字スキャン

- CJK 文字（漢字、ひらがな等）
- 非CJK 文字（POJ、数字等）
- 一文字ずつ or 連続した非CJK文字をまとめて認識

2. combine_one_word で特殊処理

- ONE_KANJI_WORDS パターン適用
- 特定の記号組合せを結合

3. スペースで分割

4. Edge Case の処理:

- `Lín--sàng` が1つの token として認識される

実行例

入力: `做工課的Lín--sàng 。`
出力:
`["做", "工", "課", "的", "Lín--sàng", "。"]`

Phase 2-2: splitted_roman - POJの分割

実装コード

```ruby
def splitted_roman
  washed_roman
    .split(/\s/)
    .compact_blank
end
```

シンプル！わずか3行

処理説明

1. シンプル：スペースで分割

- Phase 1 で記号が既に分離済み
- スペースのみで分割可能

2. 重要な設計:

- ✅ **ハイフンでは分割しない**
- ✅ **二重ハイフン（--）も保持**
- 単語内の音節構造を維持

3. compact_blank で空白要素削除

実行例

入力: `tsò-khang-khuè ê Lín--sàng .`
出力:
`["tsò-khang-khuè", "ê", "Lín--sàng", "."]`

音節数:

- `tsò-khang-khuè` = 3音節
- `Lín--sàng` = 2音節（--は音節数に含まれない）

Phase 3: 対齊と検証

```ruby
def roman_kanji_array
  spk = splitted_kanji.dup
  splitted_roman.map do |rword|
    if rword == '--' || (SP_MIRRORS.key?(rword) &&
        #... Edge Case の処理
        [rword, spss]
      end
    end
  end
end

def set_arrays
  rka = roman_kanji_array.transpose
  assign_attributes(
    roman_array: rka[0],
    kanji_array: rka[1]
  )
  self.arrays_balanced = [
    roman_array.size.positive?,
    roman_array.size == kanji_array.size,
    kanji_array.join.size ==
      washed_kanji.delete(' ').size
  ].all?
end
```

処理説明

1. 音節数でマッチング

- ハイフン = 音節区切り
- `tsò-khang-khuè` (3音節) → 漢字3文字
- `Lín--sàng` (2音節) → Roman そのまま

2. Edge Case の処理:

- Roman が kanji 側にある場合、そのまま対応
- 二重ハイフン（--）は音節数に含まれない

3. 配列の組み合わせ： `transpose` で roman/kanji を分離

4. 平衡性検証（3条件）

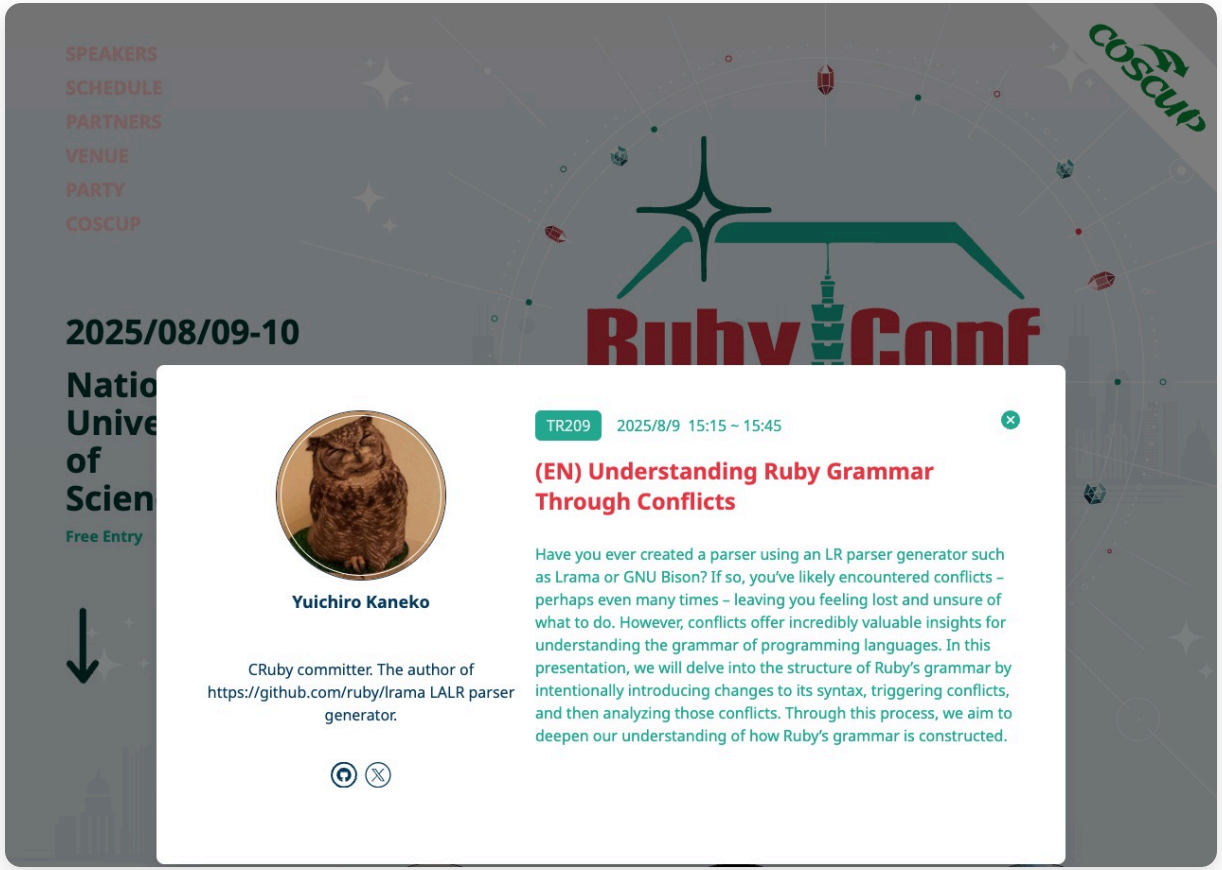
- ✅ 配列が空でない
- ✅ roman と kanji の要素数が一致
- ✅ kanji の総文字数が元の文字数と一致

kanji_array:	`["做工課", "的", "Lín--sàng", "。"]`
roman_array:	`["tsò-khang-khuè", "ê", "Lín--sàng", "."]`
roman_kanji_array	`[["tsò-khang-khuè", "做工課"], ["ê", "的"], ["Lín--sàng", "Lín--sàng"], [".", "。"]]`

Parserとの出会い

2024年の実装から2025年の気づきへ

金子さんのトークからの気づき



"Understanding Ruby Grammar Through Conflicts"

Parser の3段階処理

1. **Lexical Analysis** (字句解析)
2. **Syntax Analysis** (構文解析)
3. **Semantic Analysis** (意味解析)

💡 **「私がやっていたのは... Parser だったのか！」**

→ **「Parser として実装し直してみよう」**

Conference Driven Development

Parser の方式(ほうしき)で分詞(ぶんし)アライメント処理(しょり)を実装(じっそう)する

Parslet gem との出会(であ)い

Ruby で Parser を書くための DSL ライブラリ

なぜ Parslet？

- **PEG Parser**: Parsing Expression Grammar
- **Ruby DSL**: Ruby の文法で Parser を定義
- **明確な構造**: 3-phase 設計を自然に実現

```ruby
# Parslet の基本形
class MyParser < Parslet::Parser
  # Phase 1 & 2: 規則定義
  rule(:word) { match['a-z'].repeat(1) }
  rule(:sentence) { word >> space }

  root(:sentence)
end
```

Parslet の設計思想

Parslet は開発者に **3つの Phase** を意識させる設計：

Phase 1: Lexical Analysis

- `rule()` で Token 型を定義
- `match[]`, `str()` で文字パターン

Phase 2: Syntax Analysis

- `>>`, `|` で規則を組み合わせ
- 自動的に AST を構築

Phase 3: Semantic Analysis

- `Transform` クラスで変換
- AST → 最終データ構造

Parslet DSL 基礎語法

基本構文

`rule()` - 規則の定義

```
rule(:letter) { match['a-zA-Z'] }
rule(:digit) { match['0-9'] }
```

意味: 再利用可能な Parser 規則を定義

`match[]` - 文字クラス

```
match['a-z']           # a-z
match['a-zA-Z0-9']     # 英数字
match['\u0300-\u036F'] # 声調記号
```

意味: Regular Expressionの `[...]` と同じ

`str()` - 文字列マッチ

```
str('-')      # ハイフン
str('--')     # 二重ハイフン
str(' - ')    # スペース-ハイフン-スペース
```

意味: 文字列の完全一致

組み合わせ

`>>` - シーケンス

```
# A の後に B が続く
rule(:word) { letter >> letter }
```

意味: 順序を持つ連結（AND）

`|` - 選択

```
# A または B（順序が重要！）
rule(:token) do
  double_hyphen_word |  # 先に試す
  hyphenated_word       # 後で試す
end
```

重要: PEG は最初にマッチした選択肢を採用

`.repeat` - 繰り返し

```
match['a-z'].repeat      # 0回以上
match['a-z'].repeat(1)   # 1回以上
```

AST 構築

`.as(:symbol)` - 命名

```
# Token に型を付ける
rule(:word) {
  letter.repeat(1).as(:word)
}

# 出力される AST
{ word: "hello" }
```

意味: AST で識別するための名前

`root()` - 開始規則

```
# Parser の入口を指定
rule(:sentence) {
  token >> space?
}
root(:sentence)
```

意味: どの規則から解析を始めるか指定

Regexp → Parslet への変換（GSUB パターンから Parser 規則へ）

標点符号の処理

GSUB 方式

```ruby
# 65+ パターンの一部
ROMAN_GSUB_PATTERNS = {
  /,/ => ' , ',       # カンマ前後に空白
  /\./ => ' . ',      # ピリオド前後に空白
  /!/ => ' ! ',       # 感嘆符前後に空白
  /\?/ => ' ? ',      # 疑問符前後に空白
  # ... 他 60+ パターン
}

# 適用
text = "suà-lòh,lâi-khuànn"
ROMAN_GSUB_PATTERNS.each do |pattern, replacement|
  text = text.gsub(pattern, replacement)
end
# => "suà-lòh , lâi-khuànn"
```

特徴: 記号を空白で囲む → 後で split

Parslet 方式

```ruby
# 標点符号を Token として直接認識
rule(:punctuation) do
  str('...') | str('……') | str('……') |  # 複数文字優先
  match[',.:;()!?？！/~、ー…‥'] |          # 単一文字
  match["\"'\u201C\u201D\u2018\u2019"] |  # 引用符 （ いんようふ ）
  match['\u3000-\u303F']                  # CJK記号
end

# Token 規則
rule(:token) do
  hyphenated_word.as(:word) |
  punctuation.as(:punct)
end
```

入力: `"suà-lòh,lâi-khuànn"`

出力（AST）:

```
[
  { word: "suà-lòh" },
  { punct: "," },
  { word: "lâi-khuànn" }
]
```

特徴: Token として構造化 → split 不要

Regexp → Parslet への変換（連字符処理と音節数による漢字対応）

連字符の保持（Page 17）

GSUB 方式

```ruby
# Step 1: 記号正規化
text = "suà-lòh lâi-khuànn"
# ハイフンは保持（重要！）

# Step 2: スペース分割
tokens = text.split(/\s/)
# => ["suà-lòh", "lâi-khuànn"]

# Step 3: 音節数をカウント
syllables = "suà-lòh".split('-').size
# => 2

# Step 4: 漢字を音節数分取得
kanji_chars = ["紲", "落", "來", "看"]
combined = kanji_chars.shift(syllables).join
# => "紲落"
```

原理: ハイフン = 音節区切り

Parslet 方式

```ruby
# 連字符付き単語を1つの Token として認識
rule(:hyphenated_word) do
  syllable >>
  (single_hyphen >> syllable).repeat
end

# "suà-lòh" → { word: "suà-lòh" }
```

音節数の計算:

```ruby
# Phase 3: Transform
rule(word: simple(:w)) do
  syllables = w.to_s.split('-').size
  # => 2
end
```

漢字対応:

```ruby
# 音節数 = 漢字文字数
"suà-lòh".split('-').size  # => 2
"紲落".chars.size            # => 2
# ✅ 一致！
```

原理: Parser が音節構造を保持 → 自動対応

Ruby Parser との比較(ひかく)

Ruby Parser (Prism)

```
# 入力
"def foo(x); x + 1; end"
```

Phase 1: Lexical

```
[DEF][IDENTIFIER][LPAREN][IDENTIFIER]
[RPAREN][SEMICOLON][IDENTIFIER][PLUS]
[INTEGER][SEMICOLON][END]
```

Phase 2: Syntax

```
DefNode(
  name: :foo,
  parameters: ParametersNode(...),
  body: StatementsNode(...)
)
```

Phase 3: Semantic

- Type checking
- Scope analysis
- Code generation

台羅 Parser (RomanParserPure)

```
# 入力
"suà-lòh lâi-khuànn"
```

Phase 1: Lexical

```
[suà-lòh][lâi-khuànn]
```

Phase 2: Syntax

```
{
  sentence: [
    { word: "suà-lòh" },
    { word: "lâi-khuànn" }
  ]
}
```

Phase 3: Semantic

- AST transformation
- Array generation

```
["suà-lòh", "lâi-khuànn"]
```

漢字処理は POJ Parser に依存

一方向の依存関係：複雑な構造を先に解析

POJ Parser（複雑）

```ruby
# RomanParserPure - Parslet で実装
roman_array = [
  "suà-lòh",      # 2音節
  "lâi-khuànn",    # 2音節
  "Sin-tik-tshī"   # 3音節
]

# 音節数の計算
"suà-lòh".split('-').size  # => 2
"Sin-tik-tshī".split('-').size  # => 3
```

複雑な処理:

- ✅ ハイフンの意味解析（音節 vs 語間）
- ✅ 声調記号の認識（Unicode 結合文字）
- ✅ 二重ハイフン（--）の特殊処理
- ✅ 構文規則の定義と AST 構築

漢字処理（シンプル）

```ruby
# POJ の音節数に従うだけ
kanji = "紲落來看新竹市"

# 1. "suà-lòh" = 2音節
#    → 漢字2文字: "紲落"
# 2. "lâi-khuànn" = 2音節
#    → 漢字2文字: "來看"
# 3. "Sin-tik-tshī" = 3音節
#    → 漢字3文字: "新竹市"

kanji_array = ["紲落", "來看", "新竹市"]
```

シンプルな処理:

- ✅ 音節数 = 文字数の対応
- ✅ パターンマッチング（Edge Case 用）
- ✅ 独立した構文解析は不要

RomanParserPure の実装を試してみよう

GitHub で公開中：テストデータと検証スクリプト

https://github.com/ryudoawaru/rwc2025-slide

含まれているもの:

- 完全な RomanParserPure 実装
- 3000 件の実際のコーパスデータ

🧪 テスト結果

```
$ ruby test_parser.rb

================================================================================
Testing RomanParserPure with 3000 records
================================================================================
[████████████████████████████████████████████████████████████] 100.0% (3000/3000)

================================================================================
Final Results
================================================================================
Total records:    3000
Parse success:    3000 (100.0%)
Parse errors:     0 (0.0%)
================================================================================

🎉 PERFECT! 100% success rate achieved!
```

重要なポイント:

- ✅ **100% Parse 成功率 - 3000 件すべて正確に解析**
- ✅ **エラーなしで完全動作 - 実用性と理論の両立**

プロジェクトの成果

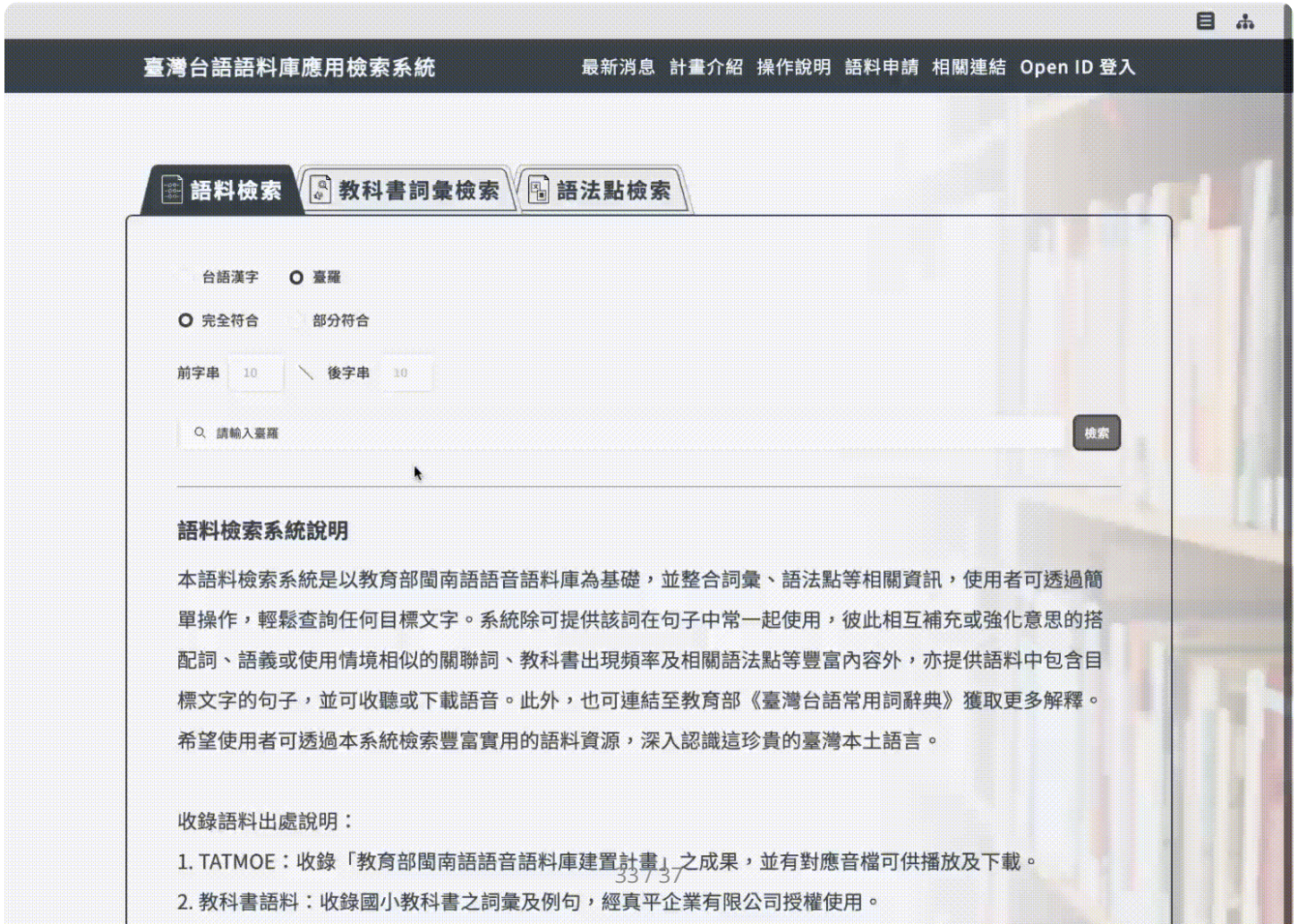
Rubyで実現した台湾語教育システム

台湾語コーパスシステム

- **公式名**: 臺灣台語語料庫 應用檢索系統
- **公開URL**: https://tggl.naer.edu.tw
- **委託元**: 教育部（文部省相当）・国家教育研究院

主要機能1: 語料検索

漢字・POJ・音声ファイルの統合検索システム



主要機能 2: 教科書語彙

台湾の教科書で使用される台湾語語彙のデータベース

主要機能 3: 語法点検索（文法ポイント検索）

台湾語の重要な文法パターンと例文の検索

結論

コンパイラ理論の普遍性

- プログラミング言語の Parser → 自然言語の処理
- Ruby の 3-phase 分析 → 台湾語の分詞アライメント

適切な道具を選び、原理を理解すれば、複雑な問題も解決できる

Conference から学び、新しい領域に挑戦する

- 既存の知識を新しい問題に応用
- エンジニアとしての成長の道

ご清聴ありがとうございました

📦 スライド & コード

https://github.com/ryudoawaru/rwc2025-slide

🏯 5xRuby

https://5xruby.com

🎪 ブース出展中

ぜひお立ち寄りください！