

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

🌸 縁の始まり

- 初めてRWCの講者として登壇
- RubyCity Matsueとの出会い

2015



👉 縁の深化

- 上定市長の5xRuby訪問
- RubyCityとの絆が深まる

2024



2023

💖 縁結びの実現

- 市長と市役所で会談
- 再びRWCの壇上へ



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

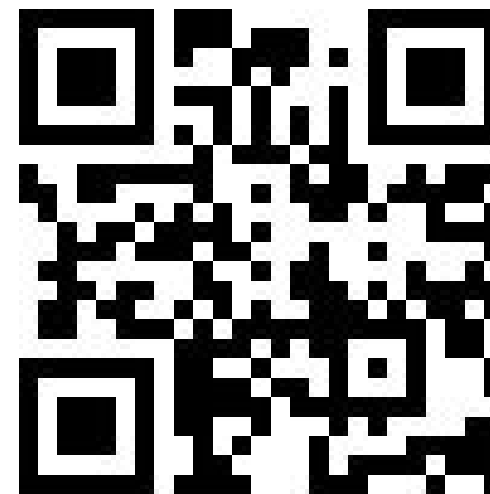
6. まとめ

- 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

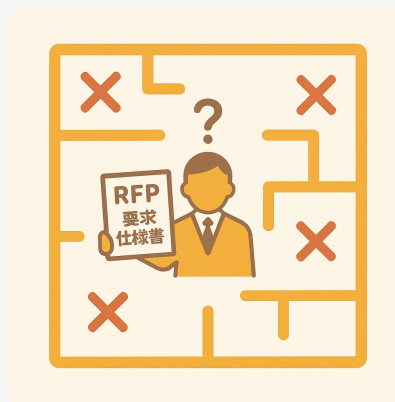
なぜ誰も手を出さなかったのか？

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

入札記録

第1回	×
第2回	×
第3回	×
第4回	×
第5回	×
第6回	?
第9回	

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 tsiàh sú-sih
香港、澳門...、臺灣恰日本	Hiong-káng, Ò-m̄ng...Tâi-uân kah Jit-pún
的時，日本義工共臺灣人	ê sī, 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 (Tài-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:

繼落	→	suà-lòh
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： 繼落來看新竹市明仔載二十六號的天氣
- POJ： suà-lòh 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）：

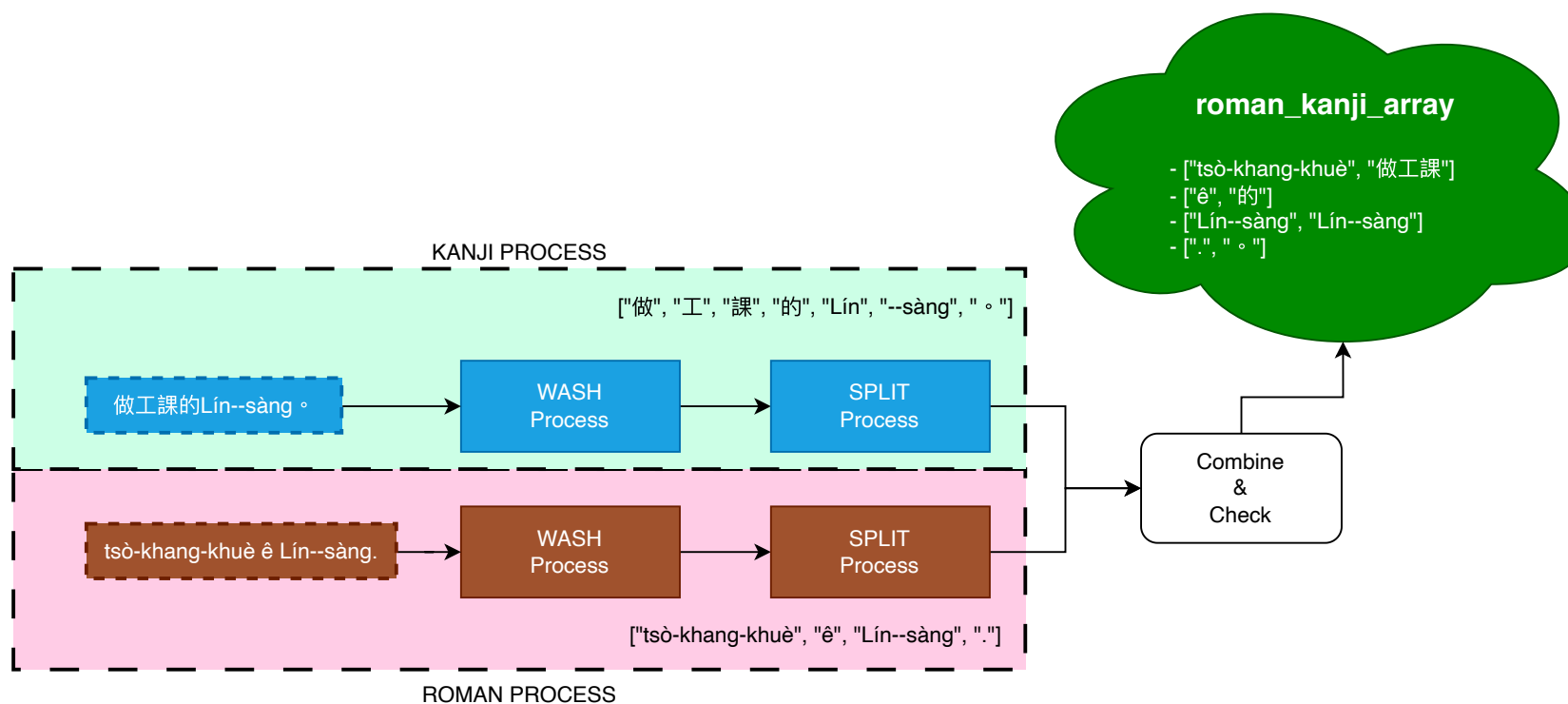
漢字	POJ
繼落	suà-lòh
來看	lâi-khuànn
新竹市	Sin-tik-tshī
明仔載	bîn-á-tsài
二十六	gī-tsap-lak
號	hō
的	ê
天氣	thinn-khì



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

```
# 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', /(『) (。)/ => '\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

```
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

```
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

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

The screenshot shows the RubyConf 2025 event page. On the left, there's a navigation menu with links: SPEAKERS, SCHEDULE, PARTNERS, VENUE, PARTY, and COSCUP. Below this, the dates 2025/08/09-10 are displayed. The main content area features a talk by Yuichiro Kaneko, titled "(EN) Understanding Ruby Grammar Through Conflicts". The talk is scheduled for 2025/8/9 at 15:15 ~ 15:45. The speaker's profile includes a photo of an owl and a bio: "CRuby committer. The author of https://github.com/ruby/lrama LALR parser generator." The talk description reads: "Have you ever created a parser using an LR parser generator such as Lrama or GNU Bison? If so, you've likely encountered conflicts – perhaps even many times – leaving you feeling lost and unsure of what to do. However, conflicts offer incredibly valuable insights for understanding the grammar of programming languages. In this presentation, we will delve into the structure of Ruby's grammar by intentionally introducing changes to its syntax, triggering conflicts, and then analyzing those conflicts. Through this process, we aim to deepen our understanding of how Ruby's grammar is constructed."

"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(しより)を実装(じっそう)する

Parslet gem との出会い

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

```
# 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枝を採用

.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

```
# 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

```
# 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

Regex → 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

```
# 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 = ["継", "落", "来", "看"]
combined = kanji_chars.shift(syllables).join
# => "継落"
```

原理: ハイフン = 音節区切り

Parslet 方式

```
# 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:

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

漢字対応:

```
# 音節数 = 漢字文字数
"suà-lòh".split('-').size # => 2
"継落".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)

```
# 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
"sua-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)

```
# POJ の音節数に従うだけ
kanji = "繼落來看新竹市"

# 1. "suà-lòh" = 2音節
#   → 漢字2文字: "繼落"
# 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 = ["繼落", "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

台灣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



臺灣台語語料庫應用檢索系統

檢索系統 系統說明 語料申請 相關連結 Open ID 登入

臺灣台語語料庫 應用檢索系統

Tâi-uân Tâi-gí Gí-liâu-khòo Ìng-iōng Kiám-sik Hē-thóng



語料檢索

查詢詞彙在語料中的情境，且有搭配詞等資訊，有助於了解語意及用法。



教科書詞彙檢索

查詢教科書出現的詞彙，了解詞彙的詞類及用法，有助於教材編輯參考。



語法點檢索

查詢台語重要的句型及語法點，並附說明及例句，有助於系統性學習。

最新消息

 語料申請已經正式開放囉！

2024-06-19 ▾

主要機能1: 語料檢索

漢字・POJ・音声ファイルの統合検索システム

臺灣台語語料庫應用檢索系統

最新消息 計畫介紹 操作說明 語料申請 相關連結 Open ID 登入

語料檢索 教科書詞彙檢索 語法點檢索

台語漢字 ☒ 臺羅

☒ 完全符合 ☐ 部分符合

前字串 10 後字串 10

語料檢索系統說明

本語料檢索系統是以教育部閩南語語音語料庫為基礎，並整合詞彙、語法點等相關資訊，使用者可透過簡單操作，輕鬆查詢任何目標文字。系統除可提供該詞在句子中常一起使用，彼此相互補充或強化意思的搭配詞、語義或使用情境相似的關聯詞、教科書出現頻率及相關語法點等豐富內容外，亦提供語料中包含目標文字的句子，並可收聽或下載語音。此外，也可連結至教育部《臺灣台語常用詞辭典》獲取更多解釋。希望使用者可透過本系統檢索豐富實用的語料資源，深入認識這珍貴的臺灣本土語言。

收錄語料出處說明：

1. TATMOE：收錄「教育部閩南語語音語料庫建置計畫」之成果，並有對應音檔可供播放及下載。

2. 教科書語料：收錄國小教科書之詞彙及例句，經真平企業有限公司授權使用。

© 2025 5xRuby CO., LTD

5xRUBY

主要機能 2: 教科書語彙

台湾の教科書で使用す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

臺灣台語語料庫應用檢索系統

檢索系統系統說明語料申請相關連結Open ID 登入

語料檢索教科書詞彙檢語法點檢

台語漢字

臺羅

輸入台語字詞查語法點

檢索

1. 選擇查詢「台語漢字」或「臺羅」，輸入查詢字詞，可輸入語法詞（如「共」、「予」等語法功能詞，或賓語、補語、動詞等語法角色詞），點選「檢索」。

2. 查詢結果包含語法說明與語料庫例句，點選「看更多」可查看更多語例與用法。互有關聯之語法點成組顯示，可點開或收合。

3. 可不輸入查詢字詞直接瀏覽全部語法點。

顯示筆數20筆

編碼	語法點	gí-huat-tiám	語法說明	例句
001-01	是	sī	即「是」。否定用「毋是」。	<div><div>這是真危險的代誌。</div><div>我攞是班裡的第一名。</div><div>看更多</div></div>
002-01	是……的	sī ... + --ê	強調「是……的」中間的成分。	<div><div>口味是一粒一的。</div><div>心肝頭實在是足凝的。</div><div>看更多</div></div>
003-01	……的……	... ê ...	表示所有格、修飾語，「的」後面的成分可以省略。	<div><div>我的車幹來猛去，</div><div>伊的英語名是啥物？</div><div>看更多</div></div>
004-01	……的	... --ê	置於名詞、形容詞、動詞之後，省略其後的名詞，指稱具備該特徵的名詞。	<div><div>阿爸畫的到底是啥物？</div><div>關豬的當做掠龍的。</div><div>看更多</div></div>

© 2025 5xRuby CO., LTD

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](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！