# コンパイラ基盤bittnの設計と評価
## - プログラミング言語を簡単につくる -

二ノ方 理仁（芝中学校　1年）

https://github.com/bittn/bittn

## 序論

### 背景

**プログラミング言語・ドメイン固有言語（Domain Specific Language：DSL）**の利点

- 汎用プログラミング言語の仕様に左右されず簡潔に書ける.
- 構文の自由度が高い.

既存の**コンパイラ基盤**（LLVM）は便利だが，機能が幅広く膨大

- 簡易なDSLの実装に時間と労力がかかる.
- 実装したい機能を探す作業が難しい.

→ DSL向けコンパイラ基盤の実現

- DSL実装にかかる時間と労力を削減できる.
- 言語のデザインに集中でき，多様なDSLが生まれる.

### 目的

**DSLの作成・テスト・考察を省力化し，効率的に繰り返すためのコンパイラ基盤の制作**

### ドメイン固有言語とは

特定の目的のために作られた比較的小規模なプログラミング言語のこと．用途に合わせて作られているため，効率的に記述できる.

## 方法

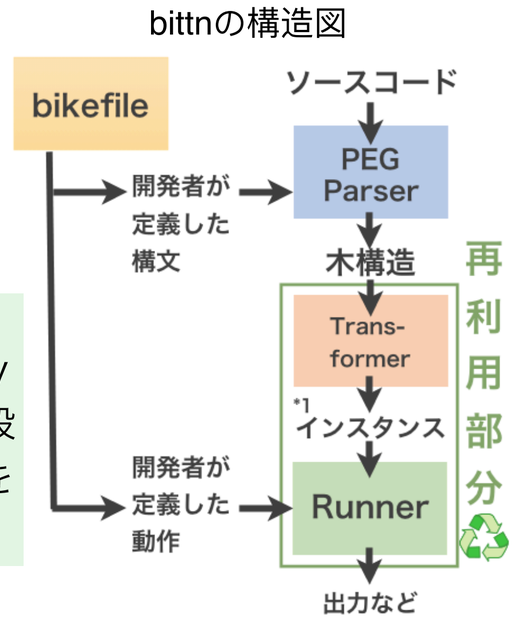
本研究では，効率的にDSLを作成するコンパイラ基盤bittnを制作する.

1. バックエンドは簡潔な構造を採用し，**中間コードを再利用**する
2. パーサの記法は**PEG**(Parsing Expression Grammar)を採用する
3. 言語開発者が**記述するファイルを一元化**する

### 再利用する構造

- Transformerが木構造から**インスタンス**を作成
- Runnerがインスタンスのcall関数を呼び出し

bittnの構造図




### インスタンスとは（*1）

インスタンスとは，bittnの記述言語Rubyが扱うメモリ上の領域である．クラスが設計図だとすると，インスタンスは設計図を元に作られた実体である.

### PEGパーサ

パーサのライブラリとして，Rubyの内部DSL Parsletを利用している.

### PEGパーサの利点

PEGはプログラミング言語の文法表記法の一つである．PEGは，処理の順序が明確で曖昧さがないという特徴がある．結果，複数の解釈があることによるバグが起きにくい.

### bikefileの作成

開発者はbikefileだけに記述すればよく，変更・追加が容易である.

- bikefileには構文・基本設定・動作を定義
- 動作をクラスへ記述
- Transformerがinit関数を呼び出し
- Runnerがcall関数を呼び出し
- 右図の例では，init関数で引数を@datasに格納，call関数で与えられている@datasをprint

テスト言語Reiwaのbikefile（一部）

```
1  class BittnTestLangParser < Parslet::Parser
2    idens = ["print"]                          ↑構文を定義
3    中略
4    rule(:code) { }
5    rule(:line) { }
6    rule(:func) {
7      idens.map{|f| str(f)}.inject(:|).as(:idens) >> param
8    }
9    中略
10 end
11 class Lang end                               ←基本設定を定義
12 中略
13 class FuncNode
14   def call()
15     idens = Marshal.load(@data[0][0]).exec
16     param = Marshal.load(@data[0][1]).call
17     PrintNode.new(param).call
18   end# (中略)
19 end
20 class PrintNode                              ←func内で呼ばれる
21   def call()                                    PrintNodeを定義
22     print(@data)
23   end# (中略)
24 end
```

## 評価

### 対象

本研究で製作したコンパイラ基盤bittnを用いてDSL Reiwaを作成した.
比較するDSLとして，既存のプラットフォームやツールを用いず，フルスクラッチでHeiseiを作成した.

### 内容

#### 評価１

動く言語ができるまでの作業量を比較する.
言語の仕様はReiwaとHeisei共に，print関数，変数を実装する.
書いたコードの行数を調べる.

#### 評価２

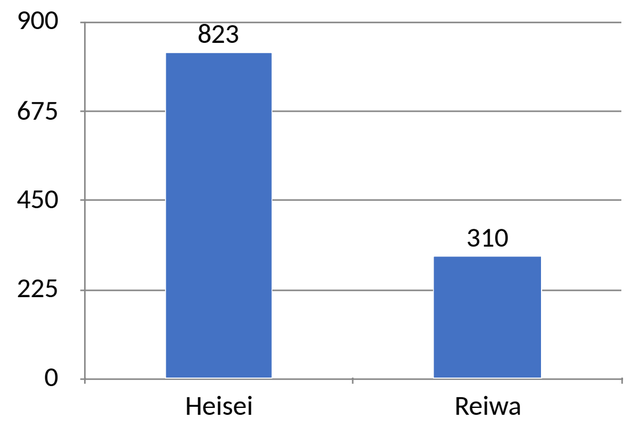
開発を進めた時，効率が保たれるか比較する.
評価1で比較したReiwaとHeiseiに，date関数を追加して実装する.
書いたコードの行数を調べる.

## 結果

### 評価１の結果

| 言語 | Heisei 1 | Reiwa 1 |
|---|---|---|
| 行数 | 823 | 310 |

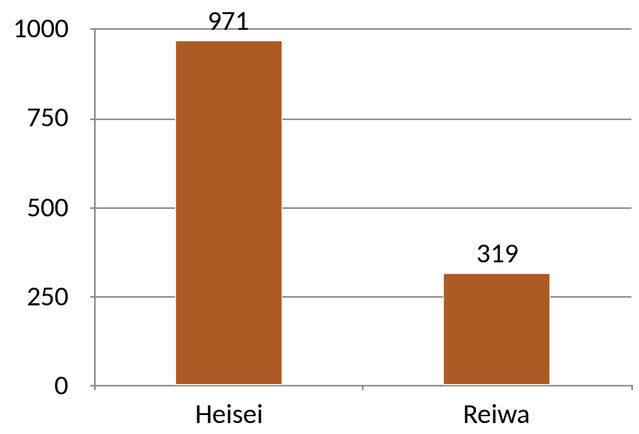
Reiwa 1 は，Heisei 1 と比べ，**62%**少ない行数で書くことができた.




### 評価２の結果

| 言語 | Heisei 2 | Reiwa 2 |
|---|---|---|
| 行数 | 971 | 319 |

date関数を追加するのに必要な行数は，Heisei２が148行，Reiwa２は9行だった.
関数を追加した後の行数は，ReiwaがHeiseiに比べて**67%**少なかった.




## 考察

評価１の結果から，bittnを用いると，DSLをより少ない作業量で作成できることがわかった．これは，Heiseiに記述した処理を，Reiwaはbittnの**バックエンド**で行なっているからである.
また，評価２では，関数を追加する時の行数もReiwaは少なかった．このように，少ない作業量で機能を追加できることから，テスト・考察を繰り返す作業も**省力化**できると考えられる.

## 結論

### 結論

**コンパイラ基盤を用いて中間コードの再利用を行うこと**で，DSL作成の作業量を減らすことができた．また，**バグが起きにくい構造**を選ぶこと，開発者が**記述する場所を一元化**することも，作業効率を上げる効果があると考えられる.

### 課題

現在使用しているParsletライブラリを自作のパーサに置き換え，バックエンド構造をわかりやすくしたい．また，bikefileを内部DSLに変更し，Rubyを知らない人でも書けるようにしたい．LLVM・Parsletを使用した場合と比較した評価実験も行いたい.