TreeP: Codex CLIで作るミニ言語

言語作り × コーディングAIの実験録

自作プログラミング言語について語る集い by 2025年11月21日（木）

水島 宏太 (@kmizu)

自己紹介

- 水島 宏太（@kmizu）
  - Scalaおじさん & 構文解析おじさん & 言語おじさん
- ふだんやってること
  - 業務では Web / Scala / 生成AI まわり
  - プライベートでは、趣味でプログラミング言語を量産
- きょうの立ち位置
  - 「言語実装を AI にどこまで任せられるか？」を
    TreeP というミニ言語で試してみた人

今日お話しすること

1. TreeP ってどんな言語か
2. パイプライン構成（EAST・マクロ・HM 型推論）
3. Codex CLI を使った実装フロー
4. AI と人間の「役割分担」についての所感
5. まとめと今後の展望

なぜまた新しい言語なのか

- 世の中には試したい言語機能のアイデアが多すぎる
  - マクロで遊びたい
  - 型システムをちょっといじりたい
  - レコードやイテレータを試したい
- でも、1つの言語に全部入れるとグチャグチャになる
- なので
  - 機能ごとにミニ言語を切り出して実験する
  - 今回は「TreeP」で
    - EAST＋衛生的マクロ＋HM 型推論＋Row レコード
    - そして「AI でここまで書かせた」事例

TreeP とは

- Tree Processing Language = TreeP
  - Scala 3 製のミニ言語処理系
  - 拡張子は `.treep`
- シンプルな処理パイプライン
  - 表層構文 → EAST 正規化 → マクロ展開
    → Hindley–Milner 型推論 → インタプリタ実行
- 想定用途
  - 言語実装の教材 / 実験用プレイグラウンド
  - 「AI に手伝わせて言語を作る」ためのネタ

TreeP の設計ゴール

- 学びやすい
  - 処理パイプラインがはっきり見える
  - 各ステージを個別に追える
- 触りやすい
  - リスト・辞書・イテレータなど
    よくあるデータ型をコンパクトに提供
- 型で遊べる
  - HM 型推論に、拡張メソッドや Row 多相レコードを追加
- 実装しやすい
  - Scala 3 で素直な AST / EAST 構成
  - Codex CLI に投げやすい粒度のタスクに分割

パイプライン全体像

```
Source (.treep)
  ↓  構文解析
Concrete AST
  ↓  EAST 正規化
EAST (Expression-AST)
  ↓  マクロ展開（組み込み＋ユーザー定義）
Macro-Expanded EAST
  ↓  Hindley–Milner 型推論 (Algorithm W)
Typed EAST
  ↓  インタプリタで実行
結果（値）を出力
```

最初のプログラム

```
def main() returns: Int {
  println("hello from treep!")
  return 0
}
```

実行

```
sbt "run run samples/hello.treep"
# => hello from treep!
```

- ふつうの命令型+関数型っぽい構文 - `return` あり
  - 戻り値の型注釈あり

辞書とイテレータの例

```
const m = { "a": 1, "b": 2 }

def sumDict() returns: Int {
  let s = 0
  for (pair in: m) {
    println(pair)
    s = s + snd(pair)
  }
  println("total:")
  println(s)
  return s
}
```

- `m` はキー文字列 → 値 `Int` の辞書
- `for (pair in: m)` でイテレータ経由で回す
- `snd(pair)` で値だけ取り出して合計

衛生的マクロ（ざっくり）

- TreeP のマクロは
  - EAST 上で動く「構造化マクロ」
    - 変数捕捉を避けるために `gensym` 的な操作をする
- ポイント
  - 「文字列置換」ではなく AST レベルでパターンマッチ
  - 生成された一時変数がユーザー変数と衝突しない
- ねらい
  - 「糖衣構文」はマクロに追い出す
  - コア言語はできるだけ小さくシンプルに保つ

組み込みマクロ: for ループ

表層構文:

```
for (pair in: m) {
  println(pair)
}
```

EASTへの展開イメージ:

```
let it = iter(m)
while (hasNext(it)) {
  let pair = next(it)
  println(pair)
}
```

- ユーザーから見ると「ただの for」
- 中身はイテレータ API に展開されるマクロ

EASTがあるおかげで、変形を型安全に書きやすい

Hindley-Milner 型推論

- ベースは Algorithm W
  - 型変数・関数型・タプル・リスト・辞書・イテレータなど
- 拡張メソッドも型推論に統合
  - `xs.map(f)` みたいな記法を素直に扱いたい
    -例（イメージ）：

```
def map(xs, f) {
  // f: A -> B
  // xs: List[A]
  // 戻り値: List[B]
}
```

型注を全部書かなくてもコンパイラ側で推論してくれる

Row 多相レコード

- 型レベルではこんなイメージ:

```
{ name: String | ρ }
```

- 「最低限 `name: String` を持っていれば良い」
  - それ以外のフィールドは何でも良い
- レコード拡張や「辞書っぽいもの」を型安全に扱うための下地

データモデルとメソッド解決

- 主な組み込みデータ型
  - `Int`, `Bool`, `String`
  - `List[T], Dict[K, V], Iter[T]`
- メソッド解決規則（ざっくり）

1. ビルトインメソッド
2. レコードの関数フィールド
3. トップレベル関数（レシーバを第1引数に取る形）

拡張メソッドっぽいものを簡単な規則で実現したかった

CLI コマンド

```
# 依存解決とビルド
sbt compile

# サンプルを生成（samples/hello.treep）
sbt "run new"

# サンプルを実行
sbt "run run samples/hello.treep"

# カレント配下の .treep をまとめてビルド
sbt "run build"

# テスト
sbt test
```

ここから本題: Codex CLI の話

- ここまで：TreeP のざっくり紹介
- ここから：どうやって実装したか
  - 手で書いたところ
  - Codex CLI に丸投げしたところ
- ゴール
- 「言語実装 × コーディングAI」の現実的な落としどころを共有する

Codex CLI とは（ざっくり）

- OpenAI 系のコーディングモデルをローカルのリポジトリに対して CLI から叩けるツール
- できることのイメージ
  - 「このファイルにこの関数を追加して」
  - 「この差分を説明して」
  - 「この仕様から実装を書いて」
- 今回の使い方
  - TreeP の処理系を「小さなタスク」に分解してひたすら投げるペアプロ相手

従来スタイルと言語実装

- ふだんの自分の言語実装フロー
  - 仕様をざっと妄想
  - パーサを書き
  - AST / 型環境 / インタプリタを手書き
  - 必要に応じてリファクタ
- 悩ましいポイント
  - AST 定義やインタプリタ実装のボイラープレート
  - テストケースを量産する手間
  - ここを Codex CLI にどこまで押し付けられるか？

→ TreeP で実験

役割分担の設計

- 人間（水島）がやったこと
  - パイプライン構成の設計
  - コア言語のざっくり仕様策定
  - EASTがどんな「感じ」かを伝える
  - テストケースの仕様を伝える
  - ミスってたら修正指示
- Codex CLI（LLM）に任せたこと
  - それ以外ほぼ全部

Codex CLIでの具体的なフロー

1. README に近い設計メモを書く
2. そのメモをペーストして `src/main/scala/.../Ast.scala` のひな型をCodex CLIに生成させる
3. 同様にパーサ / EAST / 型推論器のファイルを生成
4. 生成されたコードを人間が読む
   - 型エラー・設計の齟齬があれば修正してもらう
5. 動くようになったら「こんな感じのテストを書いて」と追加依頼

うまいくいったかどうか

- 予想以上に
  - ぶっちゃけ自分より言語や型システムに詳しい（！）
- 雑に投げてもとちゃんとしたコードが返ってくる
- 人間の役割は適当に突っつくことだけ

心地よかったパターン

- REPL 的な対話
  - エラーを貼り付けて「このエラーを直して」とだけ頼む
  - これだけで大抵なんとかなる
  - 袋小路になったときだけ雑に指示をする
- 小さな差分で回す
  - めんどうなときはでかい指示を一発やることも

言語作り×LLM の面白さ

- 言語設計との相性がよかった点
  - 文法・型システムなど
  - 「文章で語れるもの」が多い
- AST のようなツリー構造はLLM にとっても扱いやすい
- 他方「どんな言語にしたいか」という気持ちはまだ人間の仕事
  - EASTのアイデアとか拡張メソッドいれたいとか
- 所感: 言語を作るという行為自体がLLM時代の良い思考トレーニングに……なる？

TreeP の今後

- 想定している拡張ネタ
  - マクロ周りの強化
    - パターンマッチマクロ
    - 検証用マクロ
  - Row レコードまわりの実験
  - 簡単なコード整形（フォーマッタ）
- AI 活用面
- TreeP のサンプルコード生成をLLM にやらせる
- TreeP 自体をLLMに解説させる教材化

まとめ

- TreeP
  - EAST 正規化 → 衛生的マクロ → HM 型推論 → インタプリタというシンプルなパイプラインを持つミニ言語
- Codex CLI
  - 言語処理系の「退屈な部分」をかなり肩代わりしてくれる
    - 「難しい部分」もだいぶ肩代わりしてくれる
  - 唯一美学は人間の領域（？）
- 「AIに言語処理系を書かせる/修正する」がもっと当たり前になりそう