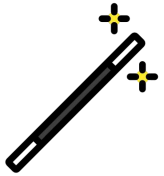
TreeP

Tree Processor ～“Lispの心”×“MLの型”×“見た目は普通”～

“マクロは欲しい。でもS式はちょっと…”
そんな欲張りに刺さるミニ言語



株式会社ネクストビート
テクノロジーエヴァンジェリスト
水島 宏太

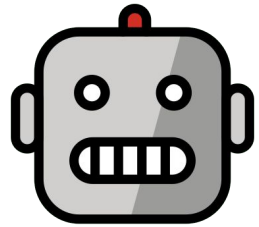
自己紹介

水島 宏太 / kmizu（みずしま こうた / けーみず）

- 株式会社ネクストビート所属
- GitHub: https://github.com/kmizu
- X: https://x.com/kmizu
- 趣味:
  - プログラミング言語作り ←今日はこれの話
  - 小説書き ←AIによる文体模倣とか
  - ゲーム（RPG、ノベルゲー中心）
  - ライトノベル（最近はあまり読んでないかも）

今日のゴール

TreePの「何がウリで、どう動いて、何が面白いか」を10分で掴む



- TreePのコンセプト（欲張りセット）
- 処理系パイプライン（学習しやすい王道）
- EASTとマクロ（型安全＆衛生的）
- 型推論・レコード・イテレータ（実用に寄せた機能）

※この言語、AIとペアプロで作った（3日くらい）

「普通に書ける」関数型言語に、ASTマクロを“簡単に”足したやつ

小さいメタ構文

XMLっぽい要素+属性モデル（EAST）

具象構文は普通

Lisp / XMLみたいな見た目にしない

衛生的マクロ

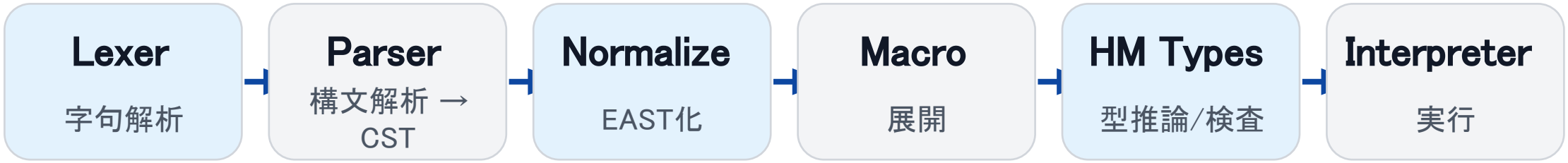
パターンマッチ＋置換＋gensym

HM型推論

Algorithm Wベースで型注釈を減らす

処理系パイプライン（学習しやすい王道）

表層構文 → EAST → マクロ → 型推論 → 実行



ポイント：EASTという中間表現を挟むと、
「マクロの書きやすさ」と「型検査」の両立がラクになる

全ノードが同じ形 → マクロ処理がシンプル

```
Element(kind, name?, attrs,
children, span?)
```

- kind: ″def″ / ″let″ / ″call″ など
- name: 識別子（省略可）
- attrs: 型情報などの属性
- children: 子要素（木）

例：関数定義 → EAST

```
def add(a: Int, b: Int) returns:
Int {
  return a + b
}
```

```
def name="add" a="Int" b="Int"
returns="Int" {
  return {
    call name="+" {
      var name="a"
      var name="b"
    }
  }
}
```

ユーザー定義マクロ（例：unless）

```
macro unless {
  pattern: unless($cond) { $body }
  expand: {
    if (!$cond) {
      $body
    }
  }
}
```

仕組み（ざっくり）

- パターン変数: $name でキャプチャ
- 展開テンプレ: $name を置換
- 衛生的: gensymで変数捕捉回避
- 型安全: EAST上で展開→型検査

動作確認済み“9個”の組み込みマクロ

assert　debug　log　trace　inc/dec　ifZero

ifPositive　until　when

（“when”はelse無しif。語感だけで勝ち）

マクロ呼び出しが、読みやすくなるやつ

Before（素直）

```
when(x > 0, () -> {
  println("positive")
})
```

After（気持ちいい）

```
when(x > 0) {
  println("positive")
}
```

内部変換（重要）

パーサが

name(args) { block }

を見つけたら…

```
name(args, () -> {
  block
})
```

に正規化する。マクロ側は「ただの関数呼び出し」扱いでOK。

※ただし：ブロックは内部的にラムダなので、returnの挙動に制限あり

型注釈、必要なときだけ“出社”

→ 関数の引数/戻り値を使用箇所から推論

```
def add(x, y) {
  return x + y
}
// x,y : Int  /  returns : Int と推論される

def main() returns: Int {
  let r = add(10, 20)
  println(r)
  return 0
```

サポート例

- List[A], Dict[K,V], Iter[T]
- タプル (A,B)
- 関数型 T1 -> T2（右結合）
- 拡張メソッドも推論に統合

Row多相レコードもある：{ x: T | ρ } で “フィールド不足OK” なまま扱える

“教育/実験用途”って言いながら、結構やる

- パターンマッチング
- イテレータ（Iter[T]）
- 拡張メソッド
- Row多相レコード
- 組み込みfor糖衣 → while+Iterへ展開

ソース構成も王道で読みやすい：
lexer / parser / east / macro / types / interpreter / cli

辞書＋イテレータの例

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

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

まとめ：TreePは“言語づくりの教材”にもなる

- **EASTでマクロを簡単に**
- **HM型推論で安全に**
- **ブロック引数で気持ちよく**

AIペアプロの学び（雑に言うと）

- “用語さえ合ってれば”処理系は案外進む
- 骨格はAI、仕上げは人間（or別AI）
- テストもAIに投げると速度が上がる

（人間の仕事：意図の言語化、名前付け、境界を切る）

Repo

github.com/kmizu/treep

次回予告：
「TreePで自作DSL」
やると楽しい

おしまい（型は裏切らん。たぶん）