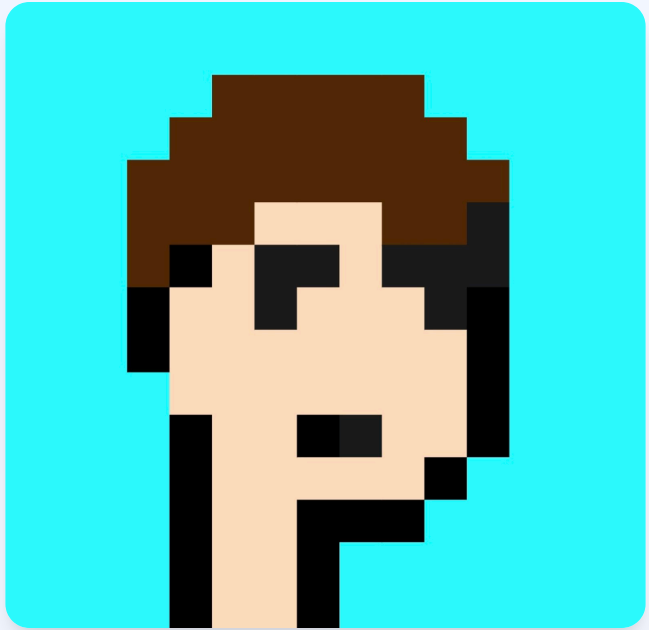
fast lean, ast-grep

Terminal Night #1 / fujitani sora

自己紹介

- **fujitani sora** / @_fs0414
- 👤 2001（24）
- 🏢 株式会社トリドリ・software engineer
- 🎤 技育CAMPの公式メンター
- 💪 TSKaigiの運営
- 💻 Terminal Use : NeoVim, WebTerm, ClaudeCode,
- 📱 Photo, Post 👌



最近のTips

CodeFormatterに凝っています。
Prettierのバグを直したり、自作のRust製Ruby Code Formatterを公開したり
https://github.com/fs0414/rfmt

アジェンダ

1. 従来の検索手法
2. ast-grepとは何か
3. 基本的な使い方
4. 実践例1: フォーマット非依存検索
5. 実践例2: API使用パターン検出
6. 実践例3: リファクタリング支援
7. 高度な活用法
8. まとめと次のステップ

コードベースから特定のパターンを検索したい

ex: Prettierのコードから、isNode関数の呼び出しを全て見つけたい

- 文字列で検索する
- 正規表現で検索する

🤔

文字列ベースの検索

```
$ grep "isNode" src/
```

- フォーマットに依存
- コメントや文字列リテラルも検出
- コードの構造は考慮しない

```
// すべてマッチする
isNode(node, ["type"])           // ← 検索対象
// isNode is deprecated          // ← コメント
const text = "isNode function"   // ← 文字列リテラル
```

正規表現検索

```
$ grep -E "isNode\(.*\[.*\]\)" src/
```

- 改行を跨ぐパターンには対応しづらい
  grep -Pzo などのoptionを使えば可能ではある
- ネストした構造の表現が複雑
- パターンが長くなる傾向

```
// マッチする
isNode(node, ["type1", "type2"])

// マッチしない
isNode(node, [
  "type1",
  "type2"
])
```

👀 コードをASTに変換しての検索

AST (Abstract Syntax Tree) = 抽象構文木

コードをASTにParse
→ ASTから「関数呼び出し」のNodeみを検索可能

```
// コードをASTとして解析
isNode(node, ["type"])
```

```
# AST表現（簡略化）
CallExpression:
  callee:
    name: "isNode"
  arguments:
    - Identifier: "node"
    - ArrayExpression:
        - "type"
```

検索方式の比較表

方式	速度	精度	フォーマット非依存	構造理解
grep	高	低	非対応	非対応
正規表現	中	中	部分的	非対応
ASTを利用	低	高	対応	対応

ast-grep : ASTベースの検索ツール

ast-grepの特徴

- コードの構造検索・置換を行う
- 20種類以上のプログラミング言語に対応

> *構文解析機能を備えたgrep/sedのようなものと考えてください。*
> *AST（抽象構文木）に基づいてコードを検索・修正するためのパターンを記述でき、数千ものファイルに対してインタラクティブに操作を行うことが可能です。*

https://ast-grep.github.io/

ast-grepが理解するコード構造

```
// 検索対象コード
isNode(node, ["type1", "type2"])
```

```
# ast-grepのパターンマッチング
pattern: isNode($NODE, $TYPES)

# マッチする箇所
$NODE  → node
$TYPES → ["type1", "type2"]
```

ast-grepの基本 - メタ変数

メタ変数	説明	使用例
$VAR	単一ノードにマッチ	$VAR.method()
$$$	0個以上のノードにマッチ	func($$$)
$$MULTI	名前付き複数ノード	func($$ARGS)

例:

```
// パターン: console.$METHOD($$$)
console.log("hello")        // マッチ
console.error("error", e)   // マッチ
console.warn()              // マッチ
```

実践例1 - フォーマット非依存検索

ref : github.com/prettier/prettier

Parse結果としての構文木は同一
WhiteSpaceなどの情報はParserの字句解析時点で除去される

```
// パターン1: 1行
isNode(node, ["sequence", "mapping"])

// パターン2: 複数行・インデント
isNode(node, [
  "documentHead",
  "documentBody",
  "flowMapping",
  "flowSequence",
])

// パターン3: スペースなし
isNode(node,["type"])
```

フォーマット非依存検索 - grepの場合

```
$ grep "isNode.*\[" src/language-yaml/
```

- Defaultで改行を跨ぐパターンを検出できない
- 複雑な正規表現は認知負荷が高い

```
✅ isNode(node, ["sequence", "mapping"])
✅ isNode(node,["type"])
❌ isNode(node, [\n     // 複数行は検出できない
```

フォーマット非依存検索 - ast-grepの場合

```
$ ast-grep --lang js --pattern 'isNode($NODE, [$$$])' src/language-yaml/
```

すべてのフォーマットを検出

結果:

```
✅ isNode(node, ["sequence", "mapping"])
✅ isNode(node, [
     "documentHead",
     "documentBody",
     ...
   ])
✅ isNode(node,["type"])
```

実際の検出結果

```
// src/language-yaml/print/misc.js:32:
    !isNode(node, [
      "documentHead",
      "documentBody",
      "flowMapping",
      "flowSequence",
    ])

// src/language-yaml/printer-yaml.js:83:
    if (isNode(node, ["sequence", "mapping"]) &&  ... )

src/language-yaml/printer-yaml.js:115:
    if ( ... && !isNode(node, ["document", "documentHead"]))
```

検出件数: 10箇所以上

実践例2 - 空配列チェックパターンの検出

配列の存在と要素チェック

目的: 空でない配列をチェックする様々なパターンを検出

```
// prettierで見られるパターン
// パターン1: 長さチェック
if (array && array.length > 0)

// パターン2: 論理AND
if (array && array.length)

// パターン3: ユーティリティ関数
if (isNonEmptyArray(array))

// パターン4: Optional chaining
if (array?.length > 0)
```

空配列チェックパターンの検出

```
# パターン1: array.length > 0 を検出
$ ast-grep --pattern 'if ($ARRAY && $ARRAY.length > 0)' src/

# パターン2: isNonEmptyArray関数の使用箇所を検出
$ ast-grep --pattern 'isNonEmptyArray($ARG)' src/
```

検出結果:

```
✅ src/language-js/print.js:89    if (node.decorators && node.decorators.length > 0)
✅ src/language-js/utils.js:234   if (comments && comments.length > 0)
✅ src/common/util.js:45          if (isNonEmptyArray(node.properties))
✅ src/language-html/print.js:156 if (isNonEmptyArray(node.children))
✅ src/document/doc-printer.js:78 if (parts && parts.length > 0)

検出件数: 50箇所以上（統一の余地あり）
```

YAMLルールで高度な検索

```
$ ast-grep scan --rule debug-rule.yml src/
```

- メッセージをカスタマイズ
- 重要度を設定
- 複数ルールを管理しやすい

```yaml
# debug-rule.yml
id: find-debug-console
language: js
rule:
  pattern: if ($DEBUG) console.$METHOD($$$)
message: Debug console statement found
severity: warning
```

実践例3 - リファクタリング支援

配列の最後の要素へのアクセス

prettierコードベースには両方が混在している

```
// 古いスタイル（ES5）
arr[arr.length - 1]
items[items.length - 1]

// 新しいスタイル（ES2022）
arr.at(-1)
items.at(-1)
```

リファクタリング候補の検出

実行:

```
$ ast-grep scan --rule modernize-array.yml src/language-yaml/utils.js
```

```
id: modernize-array-access
language: js
rule:
  any:
    - pattern: $ARR[$ARR.length - 1]
    - pattern: $ARR.slice(-1)[0]
message: Consider using modern array.at(-1) syntax
```

検出結果とリファクタリング

utils.js での検出結果

```
help[modernize-array-access]:
     ┌─ src/language-yaml/utils.js:190:7
     │
190  │ lines[lines.length - 1] = [...lines.at(-1), ...words];
     │ ^^^^^^^^^^^^^^^^^^^^^^^^
     │

help[modernize-array-access]:
     ┌─ src/language-yaml/utils.js:250:7
     │
250  │ lines[lines.length - 1] = [...lines.at(-1), ...words];
     │ ^^^^^^^^^^^^^^^^^^^^^^^^
     │

help[modernize-array-access]:
     ┌─ src/language-yaml/utils.js:261:9
     │
261  │ words[words.length - 1] += " " + word;
     │ ^^^^^^^^^^^^^^^^^^^^^^^^
     │
```

3箇所のリファクタリング候補を発見

自動置換の実行

基本の置換コマンド:

```
$ ast-grep --lang js \
  --pattern '$ARR[$ARR.length - 1]' \
  --rewrite '$ARR.at(-1)' \
  src/language-yaml/utils.js
```

重要なオプション:

- --pattern: 検索パターン
- --rewrite: 置換パターン
- --update-all: プレビューではなく実際に更新
- --interactive: 対話的に確認しながら置換

実際の置換結果（–update-all適用後）:

```
// before
const lastLine = lines[lines.length - 1];
return arr[arr.length - 1] || defaultValue;
const last = words[words.length - 1];
```

```
// after
const lastLine = lines.at(-1);
return arr.at(-1) || defaultValue;
const last = words.at(-1);
```

高度な使用例 - 条件の組み合わせ

- all : すべての条件を満たす
- any : いずれかの条件を満たす
- not : 条件を満たさない
- inside : 特定のスコープ内

```yaml
id: complex-pattern
language: js
rule:
  all:
    - pattern: if ($COND) console.$METHOD($$$)
    - not:
        pattern: if (process.env.DEBUG) console.log($$$)
  any:
    - inside:
        pattern: function $FUNC($$$) { $$$ }
message: Non-production console statement found
```

高度な使用例 - スコープ検索

- 特定のswitch caseの処理を検索
- 関数内の特定パターンのみ検出
- クラスメソッド内の処理を検索

```yaml
id: find-in-switch
language: js
rule:
  pattern: |
    switch ($EXPR) {
      $$$
      case "root": { $$$BODY }
      $$$
    }
```

メタ変数の高度な使用法

パターン:

```
$ARR[$ARR.length - 1]
```

重要: $ARR が2回出現 = **同じ変数**である必要がある

マッチする:

```
lines[lines.length - 1]  // ✅ lines が2回
words[words.length - 1]  // ✅ words が2回
```

マッチしない:

```
lines[words.length - 1]  // ❌ 異なる変数
```

セキュリティチェックの例

Node.jsでの典型的な脆弱性:

```
// コマンドインジェクション
exec('cat ' + userInput, cb);  // 危険

// eval使用
eval(getUserInput());  // 危険

// パストラバーサル
fs.readFile('./uploads/' + file, cb);  // 危険
```

検出ルール例:

```
id: detect-command-injection
language: js
rule:
  pattern: exec($STR + $VAR, $$$)
message: Potential command injection
severity: error
```

総じて好きなところ

- 検索パターンをymlで管理, 共有できる
  - 情報が永続化して読み取り可能であることはAgenticCodgingにとって重要だと思っている
- 構文木で一致するコードの一括置換が可能である
- 特定言語に依存しない汎用ツールである
  - 構文解析可能であれば
  - 個別のCustomLintを理解するよりも汎用的な仕組みかな？と思う
- 使い手次第で色々できる拡張性
- astに関する知識の隠蔽, toolとしてのinterfaceが優れているなと思う

install

インストール:

```
# macOS (Homebrew)
brew install ast-grep

# bun

bun install @ast-grep/cli

# cargo
cargo install ast-grep
```

動作確認:

```
ast-grep --version
```

Reference

- 公式ドキュメント: https://ast-grep.github.io/
- **Playground**: https://ast-grep.github.io/playground.html
- パターン構文ガイド: https://ast-grep.github.io/guide/pattern-syntax.html
- **GitHub**: https://github.com/ast-grep/ast-grep

see you later