プログラミング言語(?)を自作した話

Mobile Act OSAKA #12

自己紹介

- Twitter: @penguin_sharp
- GitHub: MeilCli
- Web: https://meilcli.net
- Skill: C#, Kotlin, Android, Azure Pipelines, GitHub Actions
- Work: Fenrir Inc.
  - Android Application Engineer
  - 発言は個人の見解であり所属する組織の公式見解ではありません

なぜ言語を自作することになったのか

```yaml
jobs:
  carthage:
    runs-on: macOS-latest
    steps:
    - uses: actions/checkout@v1
    - uses: MeilCli/carthage-update-check-action@master
      id: outdated
    - uses: 8398a7/action-slack@v2
      if: steps.outdated.outputs.has_carthage_update != 'false'
      with:
        status: ${{ job.status }}
        text: ${{ steps.outdated.outputs.carthage_update_text }}
        author_name: GitHub Actions
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}
```

自作Action

Slackに通知する
Action

A. GitHub ActionsのAction間でデータのやり取りを簡単にできるようにしたかったから

```yaml
jobs:
  carthage:
    runs-on: macOS-latest
    steps:
    - uses: actions/checkout@v1
    - uses: MeilCli/carthage-update-check-action@master
      id: outdated
    - uses: 8398a7/action-slack@v2
      if: steps.outdated.outputs.has_carthage_update != 'false'
      with:
        status: ${{ job.status }}
        text: ${{ steps.outdated.outputs.carthage_update_text }}
        author_name: GitHub Actions
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}
```

OutputのIDを指定

自作Actionの出力
(テキスト)

A. GitHub ActionsのAction間でデータのやり取りを簡単にできるようにしたかったから

```yaml
jobs:
  carthage:
    runs-on: macOS-latest
    steps:
    - uses: actions/checkout@v1
    - uses: MeilCli/carthage-update-check-action@master
      id: outdated
    - uses: 8398a7/action-slack@v2
      if: steps.outdated.outputs.has_carthage_update != 'false'
      with:
        status: ${{ job.status }}
        text: ${{ steps.outdated.outputs.carthage_update_text }}
        author_name: GitHub Actions
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}
```

Json形式での出力も対応してる

そのまま通知されるので人間が読める形式である必要性

A. GitHub ActionsのAction間でデータのやり取りを簡単にできるようにしたかったから

```yaml
jobs:
  carthage:
    runs-on: macOS-latest
    steps:
    - uses: actions/checkout@v1
    - uses: MeilCli/carthage-update-check-action@master
      id: outdated
    - uses: 8398a7/action-slack@v2
      if: steps.outdated.outputs.has_carthage_update != 'false'
      with:
        status: ${{ job.status }}
        text: ${{ steps.outdated.outputs.carthage_update_text }}
        author_name: GitHub Actions
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}
```

Actionの間にJson to Textな変換ができるActionがあれば便利そう

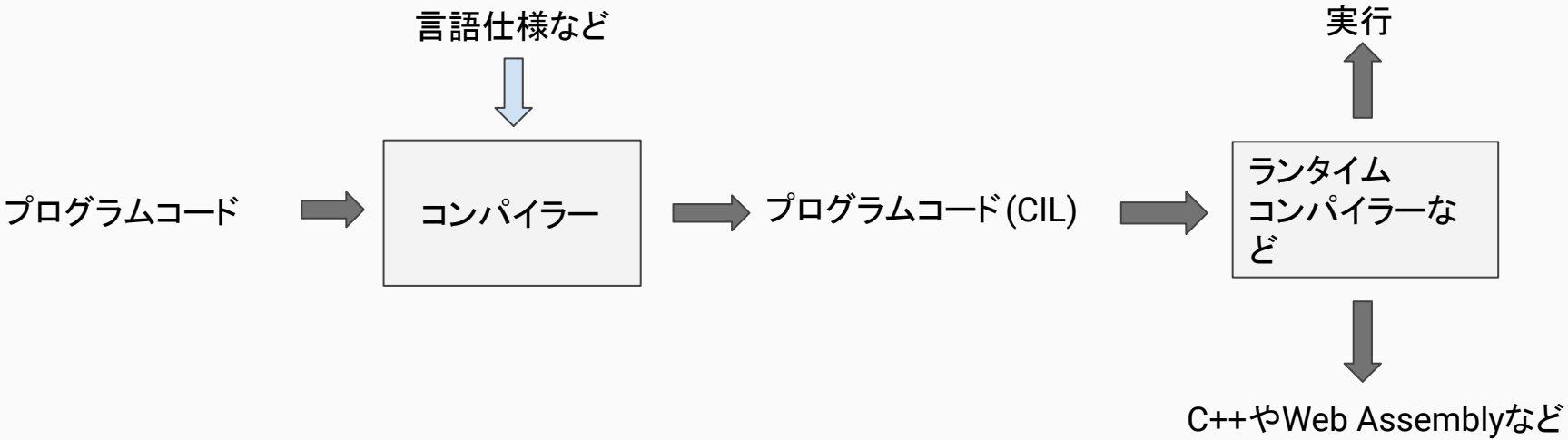
A. GitHub ActionsのAction間でデータのやり取りを簡単にできるようにしたかったから

言語の要件

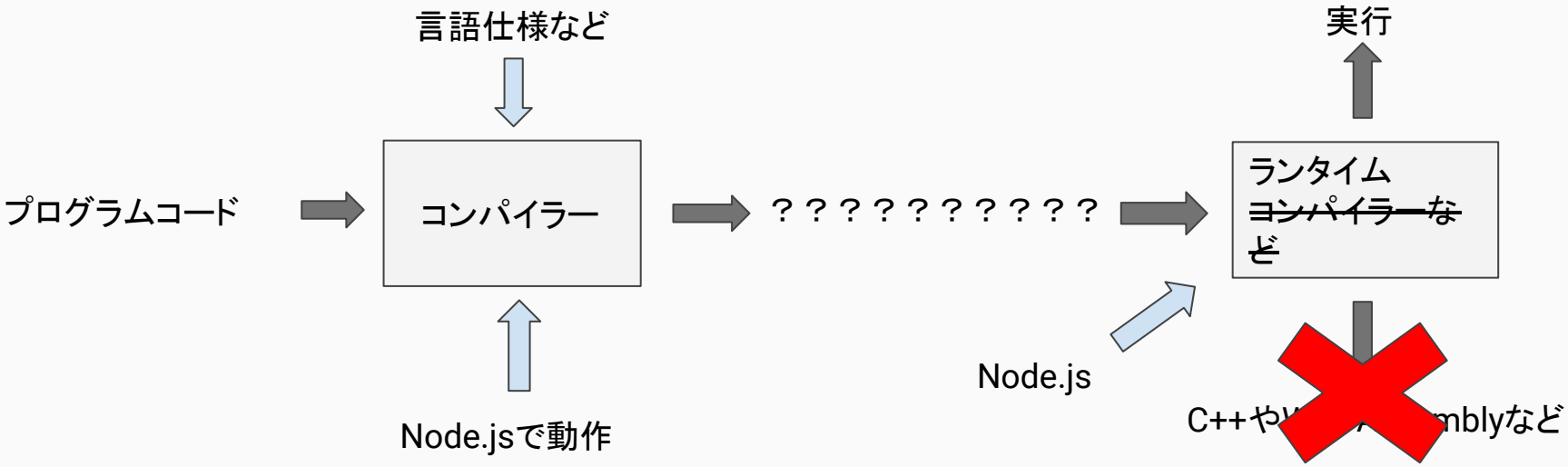
- GitHub Actionsで動かすのでNode.js上で動作
- JSON Object to TextやJSON Array to TextなどなどTextへの柔軟な変換性
- TypeScriptで作成
  - JavaScript使いたくないので…
- ついでに
  - 実装と言語仕様は切り離して他言語での実装もできるように
  - 言語仕様に対するテストデータを実装と切り離して用意
    - 各実装でのテストが楽になりそう

どういう感じの実装にするか

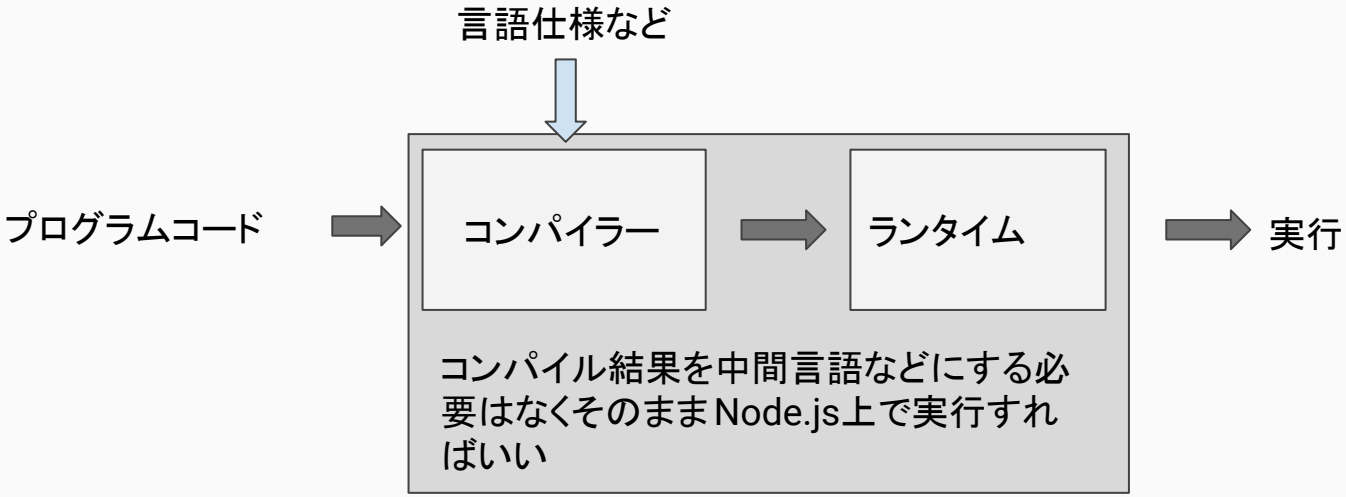
C#の場合 ※図はちょっとてきとーに作ってます



作るもの


言語仕様など
プログラムコード
コンパイラー
Node.jsで動作
？？？？？？？？？？？
実行
ランタイム
~~コンパイラーなど~~
Node.js
C++や
mblyなど

作るもの



インタプリタっぽい (?)

コンパイラーどうやって作るか

コンパイラーのしくみ

Wikipediaによるとコンパイラーは以下の部分からなることが多いらしい

- ソースコードを読み込み、トークンに分解する字句解析部
- トークン列をもとにプログラムの構文木を構築する構文解析部
- 構文木からオブジェクトコードを生成するコード生成部

https://ja.wikipedia.org/wiki/%E3%82%B3%E3%83%B3%E3%83%91%E3%82%A4%E3%83%A9#%E3%81%97%E3%81%8F%E3%81%BF%E3%81%A8%E8%A8%AD%E8%A8%88

なるほど🤔

みようみまねでコンパイラーを作ってみる

コンパイルから実行まで4つのフェーズに分ける

- 字句解析(Lexer): コードをTokenに変換
- パーサー(AST Parser): Tokenから構文木に変換(実行不能形式)
- 構文解析(Semantic Analyzer): 構文木を実行可能な構文木に変換
- コンパイラー(Compiler): 実行可能な構文木から実行用のオブジェクト生成
    - コンパイラーの中でコンパイラーって名前が出てきたややこしいですがいい名前が思いつかなかっただけです

Lexer

1. あらかじめ決めた言語仕様から意味上の区別となるトークンを決定する
2. プログラムコードを1文字ずつ読み込み、トークンの種類を抽出する
3. ついでにエスケープシークエンスを行っておく

https://github.com/MeilCli/Jfol.TS/blob/master/src/lexers/lexer.ts

```
private analyzeRawToken(source: string): RawToken[] {
    const tokens: RawToken[] = [];

    let text = "";

    const addRawTextIfNeeded = () => {
        if (text.length != 0) {
            tokens.push({ rawText: text, type: "Text" });
        }
        text = "";
    };

    for (const c of source) {
        switch (c) {
            case "$":
                addRawTextIfNeeded();
                tokens.push({ rawText: c, type: "Dollar" });
                break;
            case "(":
                addRawTextIfNeeded();
                tokens.push({ rawText: c, type: "LeftParenthesis" });
                break;
            case ")":
                addRawTextIfNeeded();
                tokens.push({ rawText: c, type: "RightParenthesis" });
                break;
            case "[":
```

AST Parser

1. トークンを1つずつ見ていき、特定の意味のあるトークンがきたら特定の分解を行うという感じにする
2. このときツリー状になるように分解する

https://github.com/MeilCli/Jfol.TS/blob/master/src/ast/parser.ts

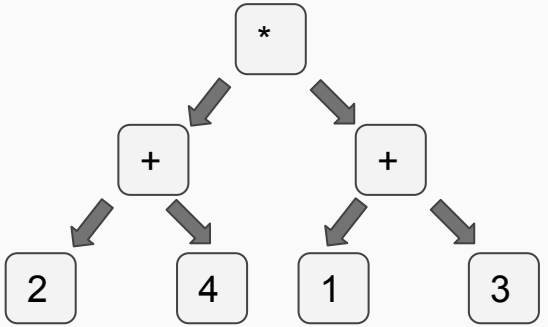
```
private childParseToken(tokens: NumberedToken[], parseLiteralAndOperator = false): Node[] {
    const nodes: Node[] = [];

    for (let i = 0; i < tokens.length; i++) {
        const currentToken = tokens[i];
        const context = new Context(tokens, i);

        switch (currentToken.type) {
            case "Dollar":
                nodes.push(this.parseField(context));
                break;
            case "DoubleDollar":
                nodes.push(this.parseFunction(context));
                break;
            case "LeftParenthesis":
            case "RightParenthesis":
                if (parseLiteralAndOperator) {
                    nodes.push(this.parseExpression(context));
                    break;
                }
                this.throwError(currentToken, `syntax error, please escape ${currentToken.type}`);
                break;
            case "LeftSquareBracket":
            case "RightSquareBracket":
```

Semantic Analyzer

- AST Parserでパースしたまでは実行しにくい形になっている
  - 式を単純に分解している
- 演算子の優先度を加味した構文木に変換する
  - 逆ポーランド記法への変換を使う

中置記法: (2 + 4) * (1 + 3)
後置記法: 2 4 + 1 3 + *
前置記法: * + 2 4 + 1 3

これを構文木で↓のように表す



Compiler

- Semantic Analyzerで実行可能な構文木になっているので組み込みの関数や演算子などと紐付ける
- 紐づくNodeは以下の感じ
  - リテラル
  - 関数
  - 演算子
  - フィールド(JSON)
- 紐付けができたら実行するだけ
  - 細かい仕組みを説明すると長くなるので割愛

肝心の構文について

あまり深くは考えずに作成(実験的構文としてそのうち破壊的変更するかも)

- *$フィールド名*でJSONのフィールドを参照
  - 配列の場合は*$フィールド名[配列のループ時に実行するボディ]*
- *$$関数名(引数)[関数ボディ]*で関数を実行
- フィールドや関数以外の文字はそのまま出力

こういう感じ

JSON

```
{
   "array": [
      {
         "package": {
            "name": "pack1"
         }
      },
      {
         "package": {
            "name": "pack2"
         }
      }
   ]
}
```

自作言語

```
Packages Total: $$(array.length)
$array[$$index: $(package.name)$$separator[\n]]
```

結果

```
Packages Total: 2
0: pack1
1: pack2
```

おわりに

- 言語仕様: https://github.com/MeilCli/Jfol
- 実装: https://github.com/MeilCli/Jfol.TS
- テストデータ: https://github.com/MeilCli/Jfol.Test
- Playground: https://github.com/MeilCli/Jfol.Playground

なお、npmへ公開した直後に他のことにモチベが出たため
本題のGitHub ActionsのAction作成まではできてないです😅