"普通"のWebアプリでWasmを活用する

松村祐貴(niboshi)

Wasm、やっていってますか?

- かつての私の認識
  - Wasmって速度全振りの特殊なシチュエーションで使うものっぽい?🤔
  - ウチみたいな普通のWebアプリだと関係ないよね😅
- 最近の私の認識
  - 意外と使いどころがあるかも！😊

アジェンダ

- Web Assembly(Wasm)とは
    - 3つの柱
    - 既存技術との比較
    - 活用例
- 株式会社HelpfeelでWasm活用にtryしている背景
    - どのような課題があるのか
    - なぜWasmなのか
    - Wasmをどう利用するのか
- やってみてどうだったか

自己紹介

- 松村祐貴
- Helpfeelプロダクトマネージャー
- GoとTypeScriptとビールとスタバが好き
- 以前は大手SIerで、リサーチャーやITコンサルタントをしていました。

インターネットのすがた

- niboshi
- Twitter: @mpppk
- GitHub: @mpppk

Web Assembly(Wasm)とは?

Web Assembly(Wasm)とは?

- ネイティブに近いパフォーマンスで動作する、
コンパクトなバイナリ形式の低レベルなアセンブリー風言語[1]
- S 式ベースのテキスト表現(.wat)も存在する[2]が、基本的にはC/C++/Rust などの他言語からのコンパイルが想定されている

```
(module
 (func $i (import "imports" "imported_func") (param i32))
 (func (export "exported_func")
   i32.const 42
   call $i
 )
)
```

Wasm外部から”imported_func”をインポート

Wasm外部へ”exported_func”をエクスポート

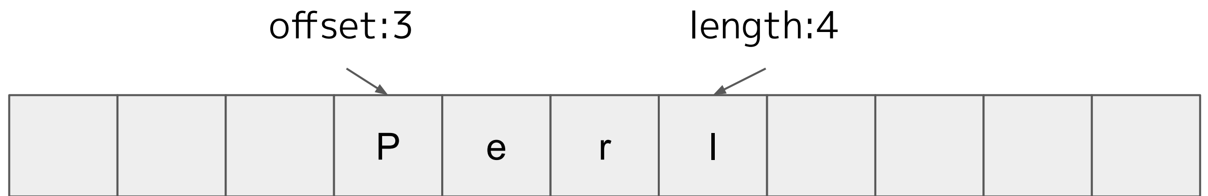
“imported_func”に42を引数として渡す

“imported_func”を呼び出す

S式ベースのテキスト表現(wat)でWasmを記述した例

Web Assembly(Wasm)とは? (2)

- Wasmは線形メモリと呼ばれる非常に単純なメモリモデルを持つ
- Wasmで利用できるプリミティブ型は、i32, i64, f32, f64の4つ[1]
- 外部から与えられた文字列を扱うだけでも以下の処理が必要
  - 外部からWasmの線型メモリに、文字列のバイト列を書き込む
  - Wasm側で線型メモリ上のoffsetとlengthを受け取り、当該文字列のバイト列を読み込む



- そのため、現実的にはグルーコード生成など周辺ツールのサポートが必要
  - wasm-packやwasmer-packなど(後述)

Wasm、3本の柱

from [The Three Pillars of WebAssembly]

Portability

デバイス、言語、プラットフォーム非依存

Security

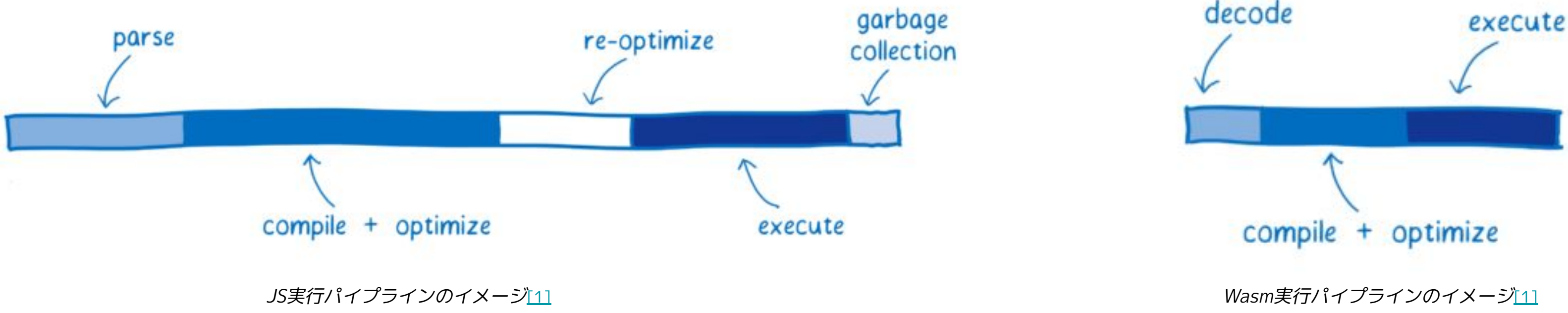
サンドボックス化された環境

Performance

ネイティブコードに近い速度で動作

Performance

- なぜWasmは速いのか[1]
  - バイナリフォーマットのため、サイズが小さい→ダウンロード時間の短縮
  - パースが不要(代わりにデコードは必要だが)
  - 事前に最適化済み



JS実行パイプラインのイメージ[1]　*Wasm実行パイプラインのイメージ*[1]

- ただし、Wasm呼び出しのオーバーヘッドが存在するため、適当に実装するとむしろ遅くなる

Security

- Wasmはサンドボックス化されている
    - ファイルシステムやネットワークは触れないので、別途WASIの実装が必要
    - ブラウザのDOMやlocalStorageも触れない
- CFI(Control Flow Integrity)と呼ばれる実行時の安全性確保の仕組みがある
    - なんかすごそうだけどよく分かってないので省略します、誰か教えてください

Portability

- Hardware-independent
  - デスクトップ、モバイル、組み込みなどデバイスに依存せず実行できる
- Language-independent
  - 特定の言語に依存しない
- Platform-independent
  - ブラウザはもちろん独立/他言語に組み込まれたVMとしてなど、Wasmの処理系がさまざまな場所で実行できる

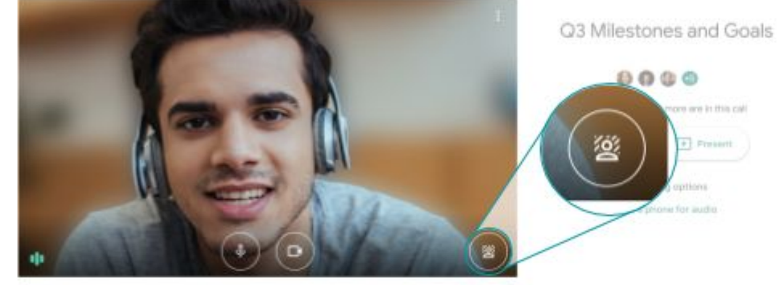
既存技術との比較

	Hardware-independent	Language-Independent	Platform-Independent
Wasm	✅	✅	✅
JVM	🔼iOS向けの実用的なJVMは存在しないが、理屈上はサポート可能	❌	❌Java Applet…ウッ頭が
NaCl	✅LLVM IRを配ってクライアントでコン パイルするパワフルなアプローチ	❌C or C++	✅
Docker	❌モバイルや組み込みでは動かない	✅	❌Webでは動かない

Wasm活用例(not only for web!)

WEB以外でのWasm活用も進んでいる

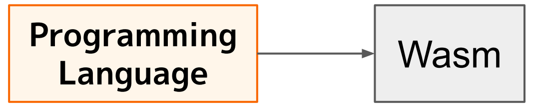
- Google Meet
  - CPU heavyな処理(ぼかし処理)の高速化
- SWC
  - ユーザの手元で動くプラグインとしての利用
- Envoy
  - サーバで動くプラグインとして利用
- Fastly / Vercel
  - (エッジ)コンピューティングの実行基盤
- FFMPEG.wasm
  - 既存資産のポーティング



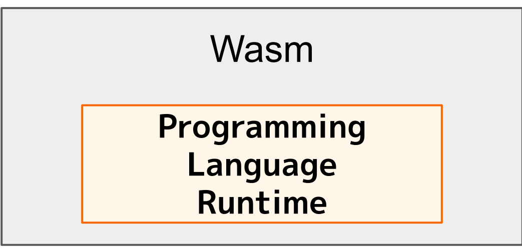
from https://workspaceupdates.googleblog.com/2020/09/blur-your-background-in-google-meet.html

Wasmとプログラミング言語の関わり方

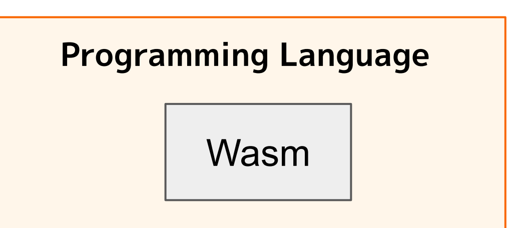
- プログラミング言語をWasmへコンパイルする
  - C++, Rustなど



- プログラミング言語のランタイムをWasmに変換してWasm+WASI VM上で動作させる
  - Python, Rubyなど

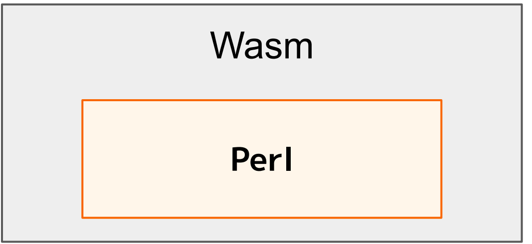


- Wasmをプログラミング言語から呼び出す
  - (当然)ブラウザやNode.jsはネイティブ対応
  - wasmtimeやwasmerなどのruntimeは各言語向けのSDKを提供している

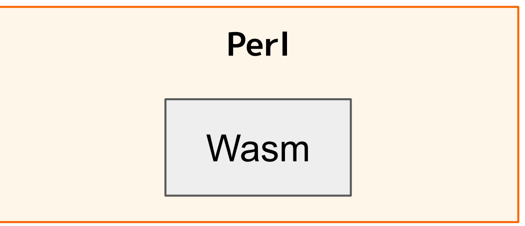


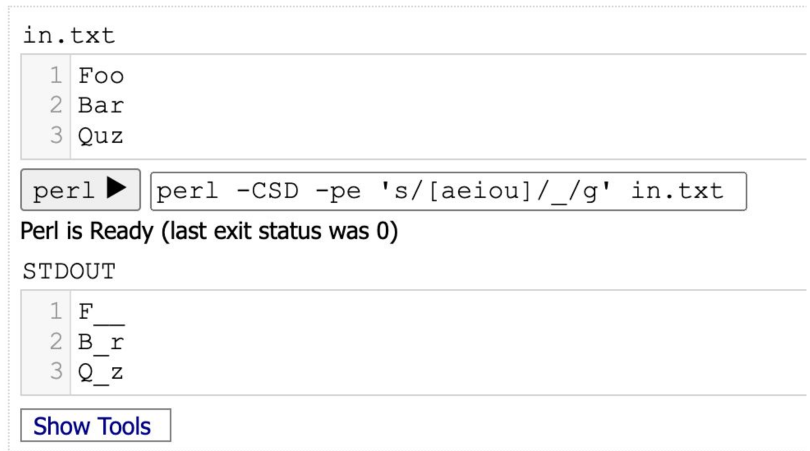
PerlとWasm

- https://webperl.zero-g.net/
  - Wasm上でperlを動かす



- https://github.com/perlwasm/Wasm
  - Wasmでperlのモジュールを実装できる





ここまでのまとめ

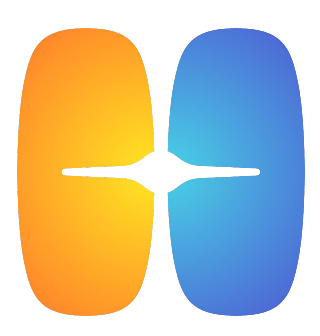
- WasmにはPerformance，Security，Portabilityの3つの観点がある
- 既存の技術とはPortabilityの観点で異なる
- Web以外でも活用が進んでいる
- プログラミング言語をWasmに変換するだけでなく、プログラミング言語からWasmを呼び出すこともできる

これらを踏まえて、普通のWebアプリはWasmとどう関わっていくべきか？

株式会社HelpfeelでのWasm活用へのtry (構想編)

株式会社Helpfeelの紹介

情報からナレッジを作り、磨き上げ、届けるところまで、All-in-Oneで行うことができるSaaSプラットフォーム



Helpfeel

知識を届ける
エンタープライズサーチ

Scrapbox

知識を磨き上げる
アイディエーションツール

Gyazo

情報を知識にする
メディアキャプチャー

Helpfeelの紹介

どんな質問でも答えられる本当に役立つFAQシステム

PayPayフリマ

ヘルプページ

キーワードで調べる

例: 発送方法

よくある質問

［出品者］購入者が受取評価してくれない

［出品者］受取評価されたのに売上金に反映されない

［出品者］サイズオーバーで返送された・再送したい

伊予銀行
金融機関コード:0174

検索 店舗ATM メニュー

よくあるご質問

例: ポイント

よくある質問

いよぎんインターネットEB操作中に、タイムアウトエラーのメッセージが表示された

いよぎんIDの登録確認メールのURLをタップしたらエラーになりました

よく見られている言葉

ALBUS

聞きたいことはなんですか？

例: クレジットカード

よくある質問

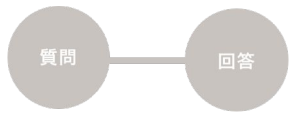
招待コードはどこで入力できますか？

注文後の変更・キャンセルはできますか？

機種変更をしてメールアドレスを変更したいのですが、パスワードを忘れてしまいました

既存FAQの課題感とHelpfeelのアプローチ

従来のFAQ



01 言い換えや曖昧な検索

「故障」だとヒットするけど、
「動かない」だとヒットしない

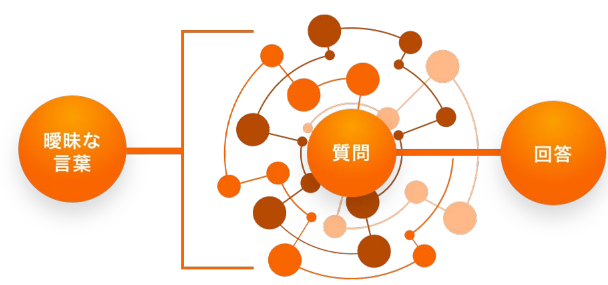
故障	動かない
返品交換の手続き 返品交換はこちら	No Result...

02 タイプミス

ちょっとタイプミスしただけで
結果が出てこない

ショッピング	シッピング
ショッピングの方法	No Result...

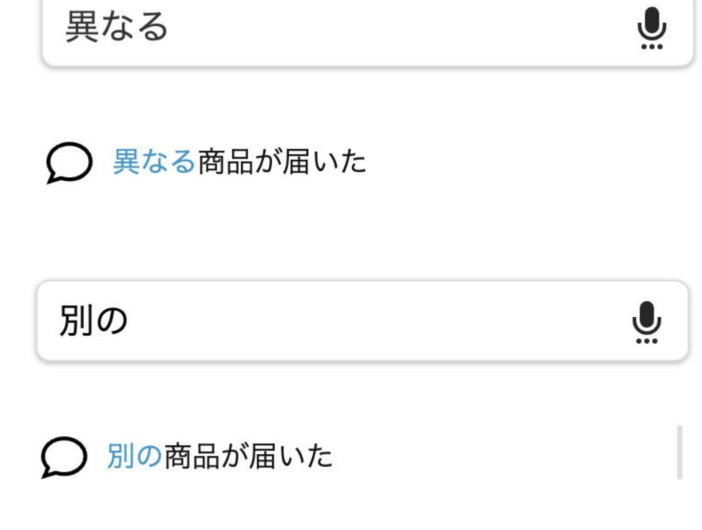
Helpfeel



- Typo tolerance
- 動詞の活用系の吸収
- 送り仮名の表記揺れ吸収
- 業界別の頻出単語辞書の活用
- **回答と質問が1対多のデータ構造**
- etc.

Helpfeel記法

- HelpfeelではFAQ記事の管理をScrapboxで行います
- **Helpfeel記法**を書くことにより、質問の半自動生成ができます
  - (異なる|別の)商品が(届いた|来た)
    - 異なる商品が届いた
    - 別の商品が届いた
    - 異なる商品が来た
    - 別の商品が来た
    - の4パターンに(論理的に)展開される！
- 変数宣言と参照もできます
  - item: (商品|品物)
  - {item}が届かない
    - 商品が届かない
    - 品物が届かない
    - の2パターンに(論理的に)展開される！



実際は他にも色々な記法がありますが今日は省略します

現状のHelpfeel記法パーサの課題感

FAQデータ前処理バッチ
- Helpfeel記法のパース
- 質問生成処理
- 形態素解析
- etc.

アプリケーションサーバ
- Viewの返却
- API提供

エンドユーザ向け画面

異なる

異なる商品が届いた

処理結果を保存

処理結果を取得

MongoDB

社内向け
制作支援ツール

Helpfeel記法のLintingがしたい！
→Node.js or ブラウザ上での Helpfeel 記法のパースが必要

PythonとJavaScriptそれぞれでHelpfeel記法パーサをメンテしたくない…
→共通の実装を各言語から利用できないか?

Polyglotな環境で処理を共通化するときの選択肢

- APIサーバ
  - サーバのメンテしたくない
  - 大量に呼び出すので通信のオーバーヘッドも気になる
- プロセス間RPC
  - エラーハンドリングとか難しそう?
  - 具体的なRPCのプロトコルも決めなきゃ
- Wasm
  - Portability(Language-independent & Platform-independent)を活用できる！

ここまでのまとめ

- HelpfeelはJavaScriptとPythonで構成されるPolyglotなシステム
- 多言語に跨って利用できるライブラリ(独自記法パーサ)を実装したい
- WasmのPortabilityで実現できるか考えてみるぞ

株式会社HelpfeelでのWasm活用へのtry (実践編)

- Wasm外部とのコミュニケーション
- テスト戦略
- Packaging
- Helpfeel社での具体的な活用例

Helpfeelで作っているもの: helpfeel-parser

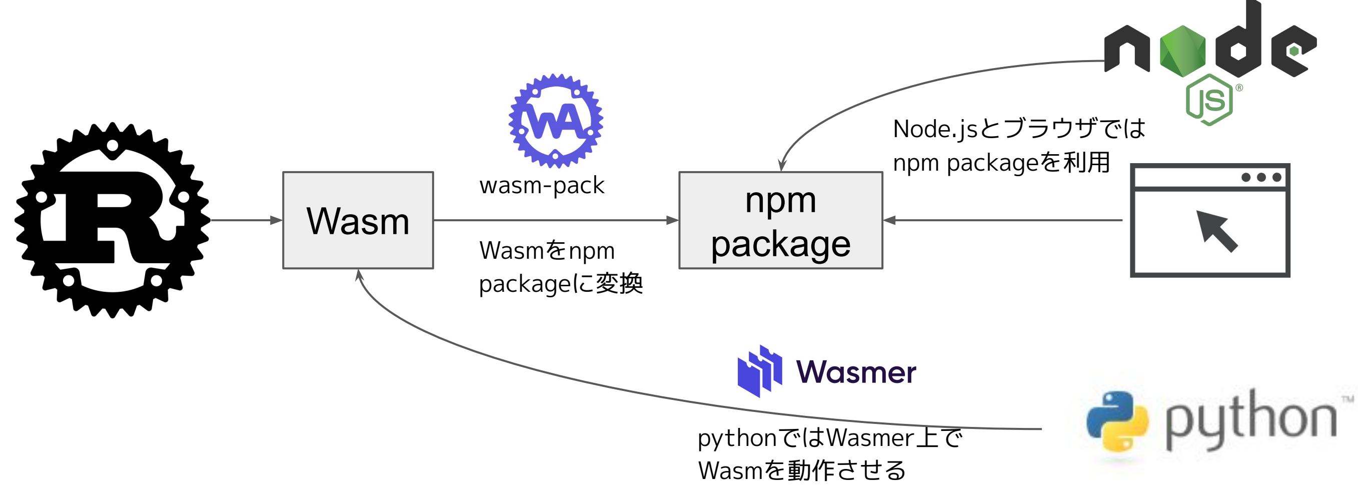
- (異なる|別の)商品が(届いた|来た)のような記法をパースするライブラリ
- パーサはRustで実装し、Wasmに変換
- JavaScriptからは普通のnpm packageとして扱うことができる

```
import { HFResult, parseNotation, Stmt } from "@nota/helpfeel-parser";

const parsed = parseNotation("(異なる|別の)商品が(届いた|来た)");
const expected: HFResult<Stmt[], never> = {
 data: [
   {
     type: "group",
     content: [
       [ { type: "text", content: "異なる" } ],
       [ { type: "text", content: "別の" } ]
     ],
   },
   { type: "text", content: "商品が" },
   {
     type: "group",
     content: [
       [ { type: "text", content: "届いた" } ],
       [ { type: "text", content: "来た" } ]
     ],
   }
 ],
 result: "ok"
}
expect(parsed).toEqual(expected);
```

Wasm活用時の検討事項と今回のスタック

- 利用する言語: Rust
- テスト戦略: Rust/Wasm/JSに対してそれぞれテスト
- Wasm外部とのコミュニケーション: wasm-bindgen
- Wasmで実装したライブラリの配布: wasm-pack



Wasm外部とのコミュニケーション

Wasm外部とのコミュニケーション

- おさらい: HelpfeelはJavaScriptとPythonでできているので、それぞれからWasmを呼び出したい
- 課題
  - Wasmで利用できる型は、i32, i64, f32, f64の4つ
    - そのままでは文字列すら扱えない！
  - PythonにはそもそもWasmのランタイムが無い
- 解決策
  - JavaScript: wasm-bindgen
    - Wasm-JS間のデータのやり取りや透過的なAPIの相互呼び出しをいい感じにしてくれる
  - Python: Wasmer
    - さまざまな言語に対応しているWasmのランタイム

Wasm-JS間のコミュニケーション - wasm-bindgen

- Rust側で実装した`parse_notation`メソッドをJS側から`parseNotation`として呼び出す例を考える

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(typescript_type = "HFResult<Stmt[], HFNotationError>")]
    pub type JsNotationResult;
}
```

Rust側とTS側での型定義の紐付け

JS側にexportする際のメソッド名を指定

```
#[wasm_bindgen(js_name = parseNotation)]
pub fn parse_notation_for_web(notation: &str) -> JsNotationResult {
    panic::set_hook(Box::new(console_error_panic_hook::hook));

    let res = parse_notation(notation);
    JsNotationResult::from(JsValue::from_serde(&HFResult::new(res)).unwrap())
```

Wasmの制約を気にせずString型を受け取れる

Rust側のpanicをconsole.errorに変換

pure Rustで書かれたコア実装

JSに値を返すためにシリアライズ

```
import { parseNotation } from "@nota/helpfeel-parser";
const parsed = parseNotation("(異なる|別の)商品が(届いた|来た)");
```

JS側からはWasmを意識せず透過的に利用できる
(packagingには後述のwasm-packを利用)

Wasm-Python間のコミュニケーション - ランタイムの比較

- 2023/3時点で、メンテナンスが活発かつPythonのbindingが存在するランタイム
    - Wasmtime
        - Bytecode Allianceによる実装
        - かつてはリファレンス実装扱いだったが、2022年9月にv1.0に到達し、Production Readyとなった
    - Wasmer
        - Wasmer, Inc.による実装
        - ランタイムだけでなく、パッケージマネージャのWAPMやグルーコード生成ライブラリのwasmer-packなど、エコシステムが充実している
    - あるベンチマークによると、パフォーマンスはどちらもほぼ変わらないらしい[1]
- helpfeel-parserではWasmerを選択した
    - 特に深い理由はなく、当時はWasmtimeがv1になってなかっただけ
    - 今から開発するならwasm-packではなくwasmer-packに寄せてもいいかもしれない(未検証)

Wasm-Python間のコミュニケーション - Wasmer

- Node.jsやブラウザのWebAssemblyモジュールに似たAPIを提供する

```
// simple.rs
#[no_mangle]
pub extern fn sum(x: i32, y: i32) -> i32 {
    x + y
}
```

```
// simple.py
from wasmer import Store, Module, Instance
import os

__dir__ = os.path.dirname(os.path.realpath(__file__))
module = Module(Store(), open(__dir__ + '/simple.wasm', 'rb').read())
instance = Instance(module)
result = instance.exports.sum(1, 2)
print(result) # 3!
```

Wasmを利用したライブラリのPackaging

wasm-packによるnpm packageの自動生成

```
$ wasm-pack build
    --target nodejs
    --out-dir pkg-node
...
$ tree pkg-node
pkg-node
├── helpfeel_parser.d.ts
├── helpfeel_parser.js
├── helpfeel_parser_bg.wasm
├── helpfeel_parser_bg.wasm.d.ts
└── package.json
```

- wasm-pack
  - wasm-bindgenで実装したRustコードをnpm packageにパッケージングするツール
  - targetとして、bundler(webpack)向けやNode.js向けなどが選択できる

wasm-packによるnpm packageの自動生成

```
$ wasm-pack build
    --target nodejs
    --out-dir pkg-node
...
$ tree pkg-node
pkg-node
├── helpfeel_parser.d.ts
├── helpfeel_parser.js
├── helpfeel_parser_bg.wasm
├── helpfeel_parser_bg.wasm.d.ts
└── package.json
```

```
{
 "name": "@nota/helpfeel-parser",
 "description": "parser for helpfeel notation",
 "version": "0.1.2",
 "license": "TBD",
 "repository": {
   "type": "git",
   "url": "https://github.com/nota/helpfeel-parser/"
 },
 "files": [
   "helpfeel_parser_bg.wasm",
   "helpfeel_parser.js",
   "helpfeel_parser.d.ts"
 ],
 "main": "helpfeel_parser.js",
 "types": "helpfeel_parser.d.ts"
}
```

wasm-packによるnpm packageの自動生成

```
$ wasm-pack build
    --target nodejs
    --out-dir pkg-node
...
$ tree pkg-node
pkg-node
├── helpfeel_parser.d.ts
├── helpfeel_parser.js
├── helpfeel_parser_bg.wasm
├── helpfeel_parser_bg.wasm.d.ts
└── package.json
```

```
export const memory: WebAssembly.Memory;
export function parseNotation(a: number, b: number): number;
```

今回実装したメソッドの型定義
Wasmからexportされているメソッド自体は引数としてnumberしか受け取らない
(Wasmで扱える型の制約)

```
export function __wbindgen_malloc(a: number): number;
export function __wbindgen_realloc(
                                    a: number,
                                    b: number,
                                    c: number
                                    ): number;
export function __wbindgen_free(a: number, b: number): void;
```

Wasm上のLinear Memoryを操作するヘルパーメソッド

- Wasmから直接exportされているメソッドの型定義

wasm-packによるnpm packageの自動生成

```
$ wasm-pack build
    --target nodejs
    --out-dir pkg-node
...
$ tree pkg-node
pkg-node
├── helpfeel_parser.d.ts
├── helpfeel_parser.js
├── helpfeel_parser_bg.wasm
├── helpfeel_parser_bg.wasm.d.ts
└── package.json
```

```
// （一部抜粋）
const path = require('path').join(__dirname, 'helpfeel_parser_bg.wasm');
const bytes = require('fs').readFileSync(path);

const wasmModule = new WebAssembly.Module(bytes);
const wasmInstance = new WebAssembly.Instance(wasmModule, imports);
wasm = wasmInstance.exports;

/**
 * @param {string} notation
 * @returns {HFResult<Stmt[], HFNotationError>}
 */
module.exports.parseNotation = function(notation) {
    const ptr0 = passStringToWasm0(notation, wasm.__wbindgen_malloc, wasm.__wbindg
    const len0 = WASM_VECTOR_LEN;
    const ret = wasm.parseNotation(ptr0, len0);
    return takeObject(ret);
};
```

Wasmをファイルシステムから読み込んでインスタンス化

文字列をいい感じにWasmに渡すためのヘルパーメソッド

- Wasmを意識せずに利用するための、Wasmモジュールのインスタンス化や文字列をWasm上のメモリに格納する処理が生成される

wasm-packによるnpm packageの自動生成

```
$ wasm-pack build
    --target nodejs
    --out-dir pkg-node
...
$ tree pkg-node
pkg-node
├── helpfeel_parser.d.ts
├── helpfeel_parser.js
├── helpfeel_parser_bg.wasm
├── helpfeel_parser_bg.wasm.d.ts
└── package.json
```

```rust
// types.rs
#[derive(Debug, Serialize, PartialEq, Eq,
TypescriptDefinition)]
#[serde(tag = "r", content = "data", rename_all = "camelCase")]
pub enum HFResult<T, E> {
    Ok(T),
    Err(E),
}
```

TypescriptDefinitionによりTypeScriptの型が自動生成される

```typescript
// helpfeel_parser.d.ts
export type HFResult<T, E> =
| { r: "ok"; data: T }
| { r: "err"; data: E };
```

- typescript-definitions crateを利用することで、Rustのコードから TypeScriptの型定義を自動生成できる

パッケージの配布

- GitHub Packagesでnpm packageを配布
  - GitHubのPersonal Access Token(PAT)により、社内メンバーだけが利用できるprivateなnpm packageとして配布することができる
  - 最近実装されたfine-grained PATにより、きめ細やかなアクセス制御ができるようになる…はずだが、2023/03/18現在、GitHub PackagesはClassic PATしか対応していない🥲
- GitHub Actionsを利用してgit tagのpushをトリガーにGitHub Packagesへnpm publishしている

```yaml
"on": {push: {tags: ["v*.*.*"] } }
jobs:
 release:
   steps:
     - uses: actions/checkout@v2
     - uses: ncipollo/release-action@v1
 publish:
   needs: [ release ]
   steps:
     - uses: actions/checkout@v3
     - uses: actions/setup-node@v3
     - run: wasm-pack build --scope nota --target nodejs --out-dir pkg
     - run: cd pkg && npm publish
```

テスト

- Wasm(Rust)側でのテスト
- wasm-packを活用したテスト
- JS側でのテスト
- Python側でのテスト(ちゃんとできてないので省略)

テスト - Wasm(Rust)

- Pure Rustで実装されたコアロジックは普通にRustのテストを書けばOK

```
let res = parse_notation("(異なる|別の)商品が(届いた|来た)").unwrap();
assert_eq!( res, vec![
    Group(vec![
        vec![Text("異なる".to_string())],
        vec![Text("別の".to_string())]
    ]),
    Text("商品が".to_string()),
    Group(vec![
        vec![Text("届いた".to_string())],
        vec![Text("来た".to_string())]
    ])
]);
```

parse結果が期待したASTの構造になっているかをテスト

テスト - wasm-pack+headless browser

- wasm-packのtestコマンドを利用すると、
  ビルドしたWasmに対して前述のRustで実装したテストを実行できる
  - フラグで環境を指定できる

```
$ wasm-pack test --node // Node.js環境でテストを実行
```

  - 環境としてヘッドレスブラウザを指定することもできる！

```
$ wasm-pack test --headless --chrome —-firefox
```

テスト - npm pakcage

- wasm-packによって生成されたnpm packageに対してテストを書けば、JS向けのグルーコードを含めたend to endなテストを実行できる
- 今回はjestのテストだけを含むnpmプロジェクトを作成した

```
import { HFResult, parseNotation, Stmt } from "@nota/helpfeel-parser";

const parsed = parseNotation("(異なる|別の)商品が(届いた|来た)");
const expected: HFResult<Stmt[], never> = {
 data: [
   {
     type: "group",
     content: [
       [ { type: "text", content: "異なる" } ],
       [ { type: "text", content: "別の" } ]
     ],
   },
   { type: "text", content: "商品が" },
   {
     type: "group",
     content: [
       [ { type: "text", content: "届いた" } ],
       [ { type: "text", content: "来た" } ]
     ],
   }
 ],
 result: "ok"
}
expect(parsed).toEqual(expected);
```

Wasm packageの具体的な活用例と運用

helpfeel-parserの構成

yarn workspaceによるmonorepo構成

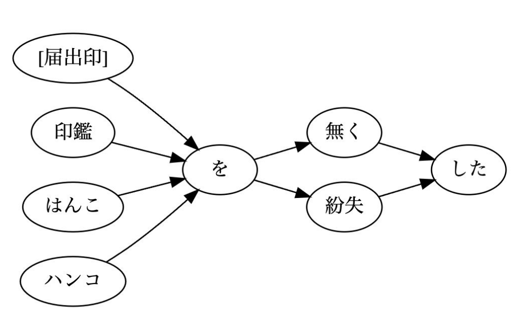
- helpfeel-parser
  - パース処理を行うメインパッケージ
- helpfeel-parser-utils
  - parserが返す構文木をグラフ構造(DAG)に変換
  - グラフ構造の探索を行うメソッドなども提供
- react-helpfeel-parser
  - utilsで生成したDAGをGraphvizで可視化するReact Component
- playground
  - helpfeel-parser+αを試せるWebアプリ
- tests
  - parserをテストするためだけのnpmプロジェクト

Input Glossary

are: ([届出印]|印鑑|{hanko})
hanko: (はんこ|ハンコ)
lost: (無く|紛失)

Input Notation

{are}を{lost}した



Playgroundの利用イメージ

wasm-packで生成したWasm packageの依存関係

- wasm-packは以下のターゲットに対応している
  - ES Modules
  - Node.js
  - Webpack
- Wasm packageを利用する場合、どれに依存すべきか?
  - ES Modules以外のターゲットでは、Universalな運用ができない
  - ES Modulesターゲットにすると、ES Moduleとしての運用が必要になる(当たり前)
- helpfeel-parserでは
  - 極力Wasm package自体には依存せず、パース後のオブジェクトを受け取る
  - 直接の依存が必要な場合はES Modulesターゲットに依存する

tests
import type
helpfeel-parser-utils
import
import
helpfeel-parser
Wasm
import type
react-helpfeel-parser
import
import
Playground
import

Wasm、実際どうですか

やってみてここがうれしい

- 多言語間での実装の共通化が実現できそう
- JSとの繋ぎ込みはwasm-bindgenやwasm-packが
  ほとんどやってくれるので簡単
- (これはRustの話だが、)構文木の表現に代数的データ型が便利

やってみてここがたいへん

- wasmerのpython bindingをそのまま使うのは、APIがprimitive過ぎて大変
  - wasmerio/wasmer-packでこの辺が解決するはずなので試したい
- ホスティングサービスによってWasmの対応状況が異なる
- wasm-packでUniversal packageを作成するにはES Modulesにするしかない
- Rustの学習コスト…

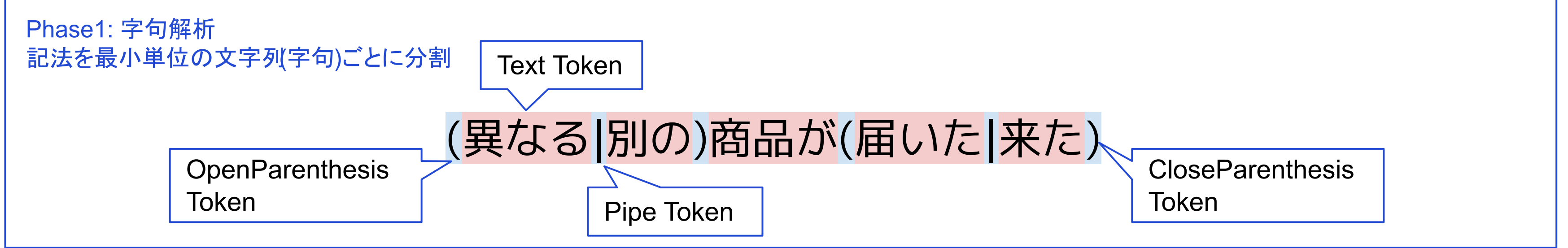
まとめ

- HelpfeelではWasmのPortabilityを活用したパーサの実装に挑戦している
- wasm-bindgenやWasmerなどのエコシステムにより、ある程度の実現可能性が確認できた
- しかしRustもWasm周りのエコシステムも分からない事がたくさんあり、大変エキサイティングです
    - 今すぐ入社して助けてください！

おわり

Appendix

パーサの実装



Phase2: 構文解析
分割した文字列(字句)から木構造(AST)を生成

パーサの実装 - lexer

- 字句解析(lexer): logos

```rust
#[derive(Logos)]
enum Token {
    #[token("(")]
    OpenParenthesis,
    #[token(")")]
    CloseParenthesis,
    #[token("|")]
    Pipe,
    #[regex(r"[^()|]+")]
    Text(&'a str),
}
// => tokenを定義しておくと、Logosがlexerを自動生成してくれる
```

独特の雰囲気を放つLogosのマスコットキャラクター

パーサの実装 - parser generator

- 構文解析(parser generator): lalrpop
  - Helpfeel記法のパース定義はやや複雑になるので、代わりにカッコで囲まれた数値をパースする例を紹介します

```
pub Term: i32 = {
    <n:Num> => n,
    "(" <t:Term> ")" => t,
};

Num: i32 = <s:r"[0-9]+"> => i32::from_str(s).unwrap();
```

```
(  (  1  2  3  )  )
|  |  |     |  |  |
|  |  +-Num-+  |  |
|  |     |     |  |
|  |   Term    |  |
|  |     |     |  |
|  +---Term----+  |
|        |        |
+------Term-------+
```

生成したパーサによるパースのイメージ

  - lalrpopのパーサ定義を書いておくと、ビルド時にパーサが自動生成される

wasm-packで生成したWasm packageの依存関係

tests

import type

helpfeel-parser-utils

import

import

外部リポジトリ

import

package

publish

helpfeel-parser

Wasm

import type

react-helpfeel-parser

import

import

Playground

import