プログラミング言語 Wasabi の設計と実装

hotaka B1 環境情報学部

Index

→ もっと Wasabi言語自体をフォーカスした方が良い(ハマリポイントや設計、コンパイル時間、JavaScriptとの比較) → 今のWasabi言語でできる範囲内でベンチマークを取るのもあり

- 目的
- 背景
- WebAssembly について
- Wasabi 言語について
- まとめ

目的

- 「WebAssembly を素早くアプリに組み込める」をコンセプトにプログラミング言語「Wasabi」の設計と実装を行う。

背景

WebAssembly は主に Rust や C++ といった低水準のプログラミング言語からコンパイルしてアプリに組み込む。

これらの言語は付属のツールチェーンを用いてプロジェクトを作成しコンパイラのターゲットに WebAssembly を指定してからコンパイルを行う必要がある。

これらは規模が小さいアプリケーションや WebAssembly を部分的に利用したいアプリケーションの開発にとっては手順が多く学習コストも高い。

→ それらを解決するような言語を開発することでゲーム開発やクリエイティブコーディングなどにおける大量のオブジェクトの処理などを簡単に書けるようにしたい。

WebAssembly について

- Webブラウザで高速にプログラムを実行できるスタックマシンベースの仮想命令セット
- バイナリ形式でプログラムを実行する
- 中間表現としてS式を利用できる
- Webブラウザ以外の環境(OS上)でも動作するようになってきている
- 低水準のホスト言語(Rust や C++)からコンパイルして利用する
- DOM操作やブラウザのAPIを操作する際は JavaScript で定義した関数を呼び出して行う

対応ブラウザの例

- Google Chrome
- Firefox
- Safari

WebAssembly Text Format(WAT)

```
;; スタックマシンで処理される

i32.const 1
i32.const 2
i32.add
;; 3

;; JavaScript の関数を呼び出す

(import "js" "print" (func $print (param i32)))
i32.const 10
call $print
```

Wasabi 言語

- 開発言語: Rust
- 論理LOC: 2494行
- 実装済みの機能
  - WebAssembly Text Format へのコンパイル(S式)
  - 関数定義, JavaScript関数のインポート, 変数定義
  - if, while
  - 四則演算, 基本的な論理演算(and, or, not)
  - 型: i32, i64, f32, f64, bool

実行までの流れ : Wasabi のコード

`1 + 2` の結果を JavaScript の `print` 関数に渡すコード。

```
// JavaScript の関数をインポート
import js {
  fn alert(i32);
}

// `export` で関数を JavaScript 側に公開
export fn main() {
  alert(1 + 2);
}
```

実行までの流れ : Wasabi のコード

`1 + 2` の結果を JavaScript の `print` 関数に渡すコード。

```
// JavaScript の関数をインポート
import js {
  fn alert(i32);
}

// `export` で関数を JavaScript 側に公開
export fn main() {
  alert(1 + 2);
}
```

実行までの流れ : Wasabi のコード

`1 + 2` の結果を JavaScript の `print` 関数に渡すコード。

```
// JavaScript の関数をインポート
import js {
  fn alert(i32);
}

// `export` で関数を JavaScript 側に公開
export fn main() {
  alert(1 + 2);
}
```

実行までの流れ : WasabiからText Format(WAT)へのコンパイル

Wasabi言語のコンパイラCLIの `wasa` を利用する。

```
$ ./wasa main.was > main.wat
```

main.wat の内容

```
$ cat main.wat
(module
  (import
    "js"
    "print"
    (func
      $alert
      (param i32)
    )
  )
  (func
    $main
    (export "main")
    (call
      $alert
      (i32.add
        (i32.const 1)
        (i32.const 2)
      )
    )
  )
)
```

実行までの流れ : WAT からバイナリへの変換

WebAssembly が公式に提供しているツールチェインの `wabt` に付属している `wat2wasm` を使用する。

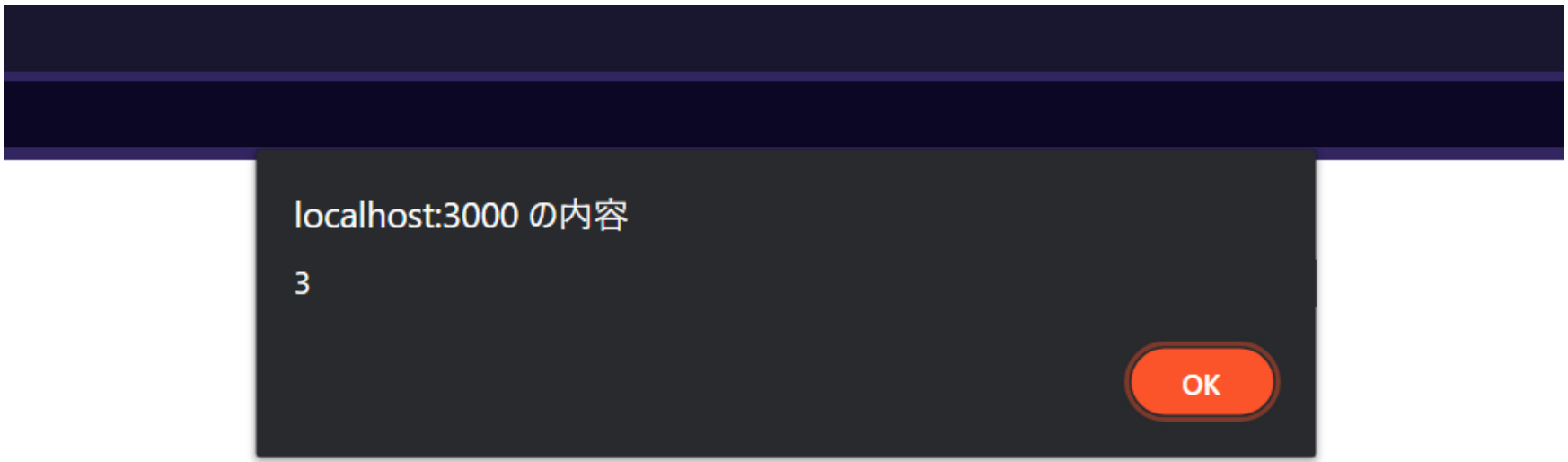
```
$ wat2wasm main.wat -O main.wasm
$ file main.wasm
main.wasm: WebAssembly (wasm) binary module version 0×1 (MVP)
```

実行までの流れ : JavaScript コードの準備

```
const imports = {
  js: {
      alert(value) {
          alert(value)
      }
  }
}
const { instance } = await WebAssembly.instantiateStreaming(fetch('/main.wasm'), imports)
instance.exports.main()
```

実行時のスクリーンショット

環境: Brave



構文紹介：`import <モジュール名> { <関数定義 ... > }`

JavaScript の `WebAssembly.instantiate` で渡された関数をインポートします。

```
// Wasabi
import js {
  fn alert(i32);
}
```

```
// JavaScript
WebAssembly.instantiate(source, {
  js: {
    alert(value) {
      alert(value)
    }
  }
})
```

構文紹介：`fn <関数名>(<引数...>): <返り値> { <本体> }`

```
// 返り値が無い場合は省略可能
fn main() {
  print(1 + 2);
}
```

```
fn add(a: i32, b: i32): i32 {
  // return a + b;
  a + b // ブロックの最後が式ならそのまま返り値になる。
}
```

構文紹介：`let <変数名>: <型> = 初期化`

ローカル変数を定義します。

```
fn main() {
  let year: i32 = 2003;
  let is_leap_year: bool = true;

  // 型を省略すると初期化式から自動的に型が決定されます。
  let radius = 10; // i32
  let pi = 3.14;   // f64
}
```

構文紹介：`while <式> { <処理> }`

```
fn main() {
  let i = 0;

  while i < 100 {
    print(i);
  };
}
```

今後の展望

- 配列の実装
- クラスや構造体の実装
- TypeScript の型定義ファイルの生成
- Language Server Protocol の対応