

バイブルコーディングで作る 自作言語Cbインタプリタ！

AI駆動開発で実現する理想の言語 - 他言語のいいとこ取り

miyajima (@sl_0122)

2025/11/21



自己紹介



miyajima

Job: Software Developer

Twitter: @sl_0122

GitHub: @shadowlink0122

趣味

知識 0 からコンパイラ開発

ゲーム アニメ

家系ラーメン

💬 開発状況を垂れ流し中 : [osdev-jp](#)

目次

Cbとは

言語の概要とビジョン

Section 1: 実装した機能(インタプリタ)

C/C++ライクな基本構文 + モダン言語の機能

Section 2: バイブルコーディング

AI駆動開発による効率的な言語実装

Section 3: プロジェクトを通して学んだこと

成功と失敗から得た教訓

本日お話しすること / しないこと

✓ お話しすること

Cb言語の紹介

どんな言語か、どんな機能があるか

バイブルコーディングでの開発

AI駆動開発の実践と知見

✗ お話ししないこと

内部実装の詳細

インタプリタ/コンパイラのアルゴリズムや最適化手法

💡 実装に興味がある方はぜひ後で質問してください！

Cbとは？

読み方はシーフラット
C++をベースに、Rust/Go/TypeScript/Pythonの
優れた機能を統合したモダンなシステムプログラミング言語

🌀 設計コンセプト

- ▶ C/C++の親しみやすさ
- ▶ モダンな言語の書きやすさ
- ▶ 静的型付け + ジェネリクス
- ▶ インタープリタで即座に実行（現在）
- ▶ コンパイラでバイナリ生成（開発中）

🚀 主要機能

- ▶ 基本構文はいろんな言語のいいとこ取り
- ▶ Option/Result型によるエラーハンドリング
- ▶ interface/implとジェネリクス
- ▶ FFIで他言語と連携可能

各言語のいいとこ取り

Cbは既存言語の優れた機能を組み合わせ、より使いやすく強力な言語を目指しています

言語	取り入れた機能	Cbでの実装
C++	<ul style="list-style-type: none">テンプレート演算子オーバーロードポインタ/参照ゼロコスト抽象化	<ul style="list-style-type: none">→ ジェネリクス[T]→ メソッド定義での代替→ ポインタ(*T)と参照(&T)→ 最適化コンパイル
Rust	<ul style="list-style-type: none">所有権システムResult/Option型パターンマッチングトレイト	<ul style="list-style-type: none">→ 自動メモリ管理（計画中）→ Result[T,E] / Option[T]→ match式→ interface/impl
TypeScript	<ul style="list-style-type: none">型システム構造的型付けユニオン型async/await	<ul style="list-style-type: none">→ 明示的な型指定→ 構造体の型システム→ Union型（ 演算子）→ async/await（実装中）
Go	<ul style="list-style-type: none">defer文複数戻り値エラーハンドリングシンプルな構文	<ul style="list-style-type: none">→ defer（実装済み）→ 複数戻り値（構造体で実現）→ Result型でのエラー処理→ シンプルな文法設計
Python	<ul style="list-style-type: none">self キーワードリスト内包表記シンプルな構文	<ul style="list-style-type: none">→ メソッドでのself使用→ 配列操作（計画中）→ 読みやすい構文

🎯 Cbの独自性



Cbが目指すもの

低レベルから高レベルまで、1つの言語で

🔧 システムプログラミング

C++ **Rust** のように

- ▶ OS・カーネル開発
- ▶ デバイスドライバ
- ▶ 組み込みシステム
- ▶ メモリ直接制御
- ▶ ゼロコスト抽象化

🌐 アプリケーション開発

TypeScript **Go** のように

- ▶ Webフロントエンド
- ▶ バックエンドAPI
- ▶ GUI アプリケーション
- ▶ スクリプティング
- ▶ 高い生産性

↔
同じ言語で

究極の目標： OSからWebアプリまで、すべてCbで書ける世界

技術スタック

C++17実装のモダン言語処理系
インタープリタでとりあえず動かしたい
コンパイラで高速化したい

C++17 実装言語

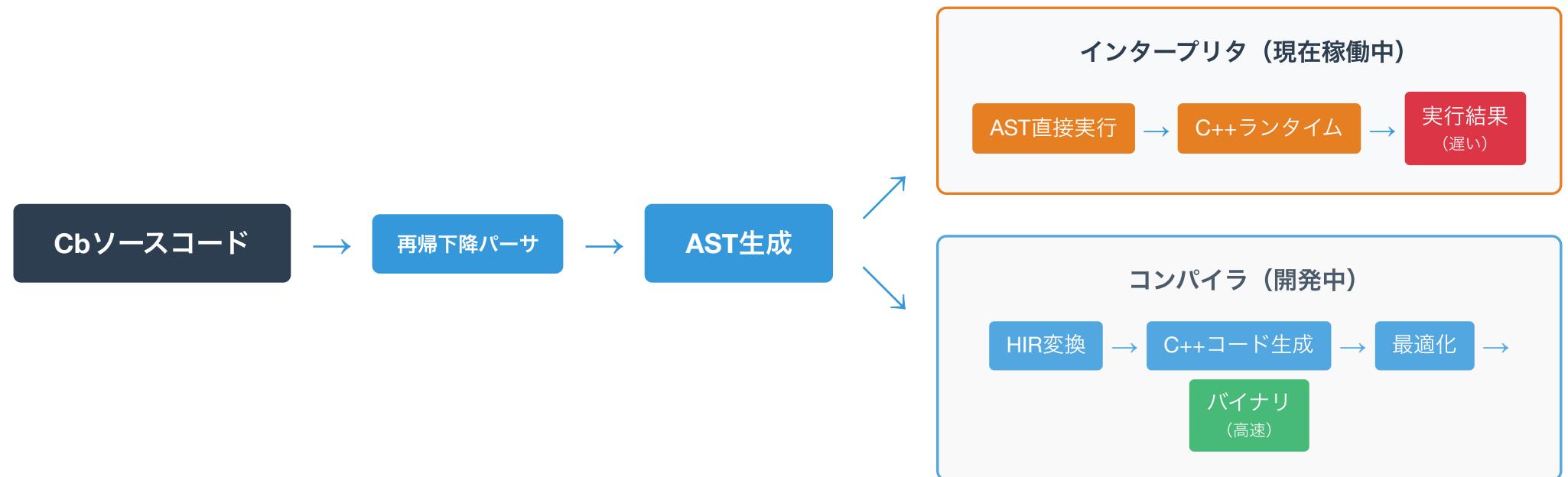
- モダンC++機能 (variant, optional)
- GitHub Copilot Pro+ & Claude Code活用
- 構造化束縛・テンプレート
- 高速コンパイル性能

Dual Mode 実行モデル

- インターパリタ (現在稼働)
- AST直接実行・即座に動作
- コンパイラ (開発中)
- 最適化バイナリ生成



実行アーキテクチャ



現状： インタープリタで開発中。最適化無しで低速だが、即座に実行可能

目標： コンパイラ完成でC++と同等の実行速度を実現

HIRベースのマルチターゲット戦略



⌚ HIRの役割

統一された中間表現

すべてのターゲットが共通のHIRから生成

最適化の一元化

HIRレベルで共通の最適化を適用

新ターゲット追加が容易

HIR→新言語の変換器を追加するだけ

クロスプラットフォーム

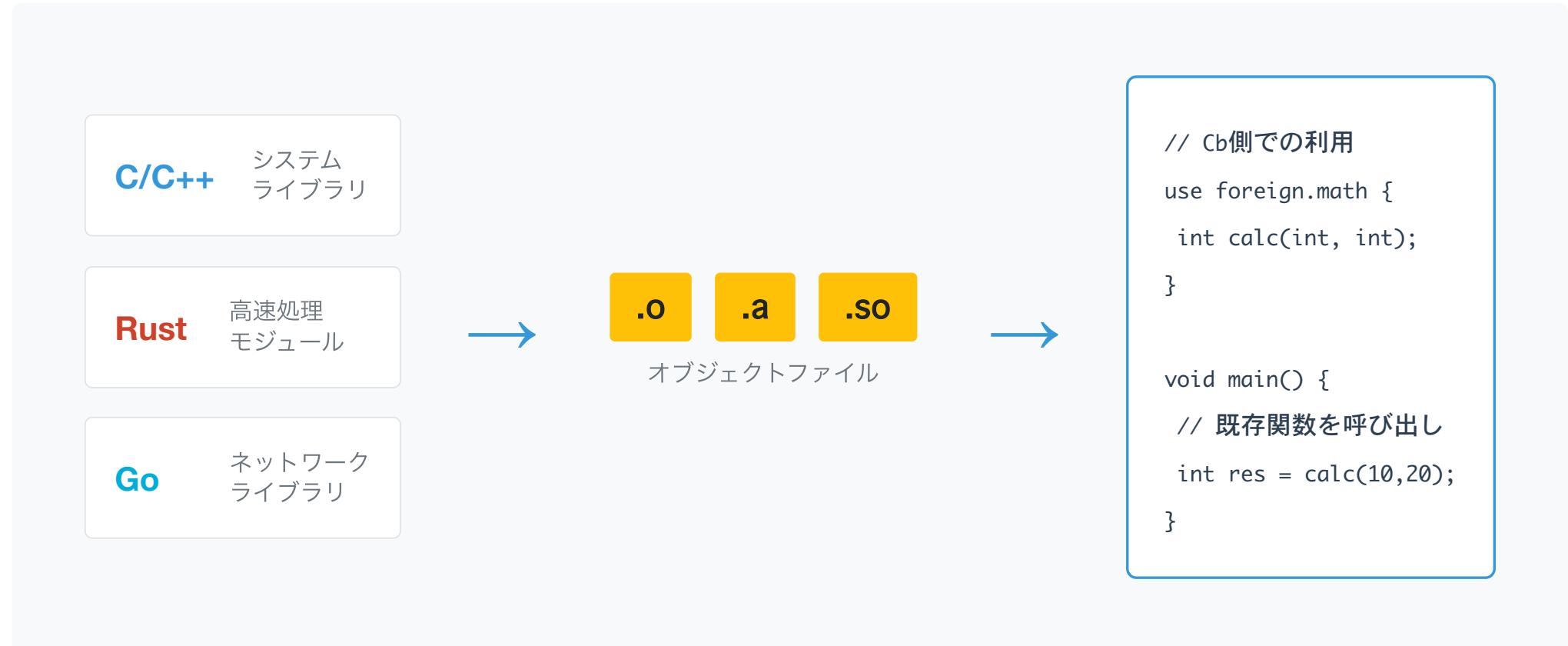
1つのコードから複数環境向けにビルド



FFI - 多言語連携機能

🔗 Foreign Function Interface

既存の資産を活用しながら、段階的にCbへ移行可能



💡 FFIのメリット

既存資産の活用

長年開発されたライブラリをそのまま利用

段階的な移行

必要な部分から徐々にCbで書き換え

言語の良いとこ取り

各言語の得意分野を組み合わせて利用

リスクの最小化

実績あるコードを維持しつつ新機能追加

パーサとテストシステム

再帰下降パーサ



手書き実装

柔軟なエラーメッセージとリカバリー



高速解析

パーサジェネレータ不要で直接的な実装



段階的パース

構文→型推論→意味解析の明確な分離

開発支援機能

実行モード

```
./cb run script.cb
```

```
./cb compile script.cb
```

- ▶ **run:** インタープリタ実行（主機能）
- ▶ **compile:** C++生成（開発中）
- ▶ デバッグモード対応

テストスイート

```
make test
```

```
./run_unified_tests.sh
```

- ▶ 850+のテストケース
- ▶ インタープリタ/コンパイラ両対応
- ▶ 仕様変更の影響検証



443/849 tests passing



52.2% coverage



早期バグ検出



開発ロードマップ

✓ Phase 1

基本構文

変数・関数・制御構造
型システム・配列
インターフェース
トレイト

📝 Phase 3

高度な機能

ジェネリクス
FFI
マクロシステム
(インタープリタ実装済)

🚧 Phase 2

コンパイラ化

HIR→C++変換
バイナリ生成

🔮 Phase 4

エコシステム

パッケージ管理
標準ライブラリ
IDE統合

現在: v0.13.1
インタプリタ終了

現在: v0.14.0
コンパイラ開始

次期: v1.0.0
エコシステム完成



ドキュメントとリリース管理

📚 ドキュメント

言語仕様書 部分メンテナンス

構文定義・型システム・標準API

実装優先度 アクティブ

機能優先順位・実装状況追跡

🚀 リリース管理

GitHubタグ

v0.13.1 v0.13.0 v0.12.0

リリースノート

✨ 新機能 🐞 バグ修正 📊 テスト状況

github.com/shadowlink0122/Cb | 500+ commits | v0.13.1



SECTION 1

実装した機能(インタプリタ)

C/C++ ライクな基本構文 + モダン言語
の機能

基本構文 - 変数と基本型

```
// Cbの基本構文はC++と同じ
int x = 42;
string name = "Cb Lang";
bool flag = true;
double pi = 3.14159;

// ポインタと参照
int* ptr = &x;
int& ref = x;

// 配列
int[10] numbers;
int[3][3] matrix;

// constとstatic
const int MAX = 100;
static int counter = 0;
```

- 💡 C++プログラマならすぐに書ける - 学習コストゼロで開始可能
- 💡 ポインタ(*T)と参照(&T) - 低レベル操作もサポート
- 💡 配列型 - 配列情報は型に付属する

プリミティブ型

整数型

- ▶ `tiny` - 8ビット符号付き整数
- ▶ `short` - 16ビット符号付き整数
- ▶ `int` - 32ビット符号付き整数
- ▶ `long` - 64ビット符号付き整数

浮動小数点型

- ▶ `float` - 32ビット浮動小数点数
- ▶ `double` - 64ビット浮動小数点数

その他の基本型

- ▶ `bool` - 真偽値 (true/false)
- ▶ `char` - 文字型
- ▶ `void` - 値なし型
- ▶ `string` - 文字列型

型の特徴

- ▶ C/C++互換のサイズ
- ▶ 明確なビット幅
- ▶ 効率的なメモリ使用

⌚ C/C++と同じプリミティブ型で学習コストゼロ

const修飾子とポインタ

🔒 constの位置による意味の違い

```
// 1. 値が固定(ポインタが指す値を変更不可)
const int* ptr1 = &value;
*ptr1 = 10;           // ✗ エラー: 値の変更不可
ptr1 = &other;       // ✓ OK: ポインタ自体は変更可能

// 2. アドレスが固定(ポインタ自体を変更不可)
int* const ptr2 = &value;
*ptr2 = 10;           // ✓ OK: 値の変更可能
ptr2 = &other;       // ✗ エラー: ポインタの変更不可

// 3. 両方固定(値もアドレスも変更不可)
const int* const ptr3 = &value;
*ptr3 = 10;           // ✗ エラー: 値の変更不可
ptr3 = &other;       // ✗ エラー: ポインタの変更不可

// 実用例: 読み取り専用配列の参照
void printArray(const int* const arr, int size) {
    // arrが指す配列の内容もarrのアドレスも変更不可
    // 安全に配列を読み取り専用で処理
    for (int i = 0; i < size; i++) {
        println(arr[i]);
    }
}
```

💡 メモリ安全性：constで意図しない変更を防止

基本構文 - 制御構造

⌚ 条件分岐とループ

```
// if-else文
if (x > 0) {
    println("Positive");
} else if (x < 0) {
    println("Negative");
} else {
    println("Zero");
}

// forループ(break/continue対応)
for (int i = 0; i < 100; i++) {
    if (i % 2 >== 0) continue; // 偶数をスキップ
    if (i > 10) break;        // 10を超えたなら終了
    println(i); // 1, 3, 5, 7, 9
}

// whileループ
int count = 0;
while (count < 5) {
    println("Count: ", count);
    count++;
}
```

 C言語ライクな親しみやすい構文

switch文 - 強化された条件分岐

多彩な条件指定

```
switch (num) {  
    // 単一条件  
    case(1){  
        println("1");  
    }  
    // OR条件(|)- いずれかにマッチ  
    case(2 | 3){  
        println("2 or 3");  
    }  
    // 範囲条件(...) - 範囲内にマッチ  
    case(4...6){  
        println("4 to 6");  
    }  
    // 複合条件も可能  
    case(7 | 10...12){  
        println("7 or 10 to 12");  
    }  
    // それ以外  
    else{  
        println("Other");  
    }  
}
```

⚡ 柔軟な条件指定で複雑な分岐を簡潔に表現

型システム - Union型（合併型）

⌚ TypeScriptライクなUnion型

```
// Union型の定義
typedef MixedType = int | string | bool;

MixedType value = 42;           // OK: int
value = "hello";               // OK: string
value = true;                  // OK: bool

// 型チェックとパターンマッチング
switch (value) {
  case (n){
    println("Number: ", n);
  }
  case (s){
    println("String: ", s);
  }
  case (b){
    println("Boolean: ", b);
  }
}
```

✨ 静的型付けと動的な柔軟性の両立

型システム - リテラル型

✨ 特定の値のみを許可する型

```
// リテラル型の定義
typedef DiceValue = 1 | 2 | 3 | 4 | 5 | 6;
typedef Color = "red" | "green" | "blue";

DiceValue dice = 2;           // OK
// dice = 7;           // Error: 7 is not valid

Color color = "red";         // OK
// color = "yellow";    // Error: not a valid color

// 関数の戻り値にも使える
Color getTrafficLight(int state) {
    if (state >= 0) return "red";
    if (state >= 1) return "green";
    return "blue"; // エラー状態
}
```

🛡️ コンパイル時の型安全性を強化

型システム - 構造体とリテラル初期化

構造体定義と初期化

```
// 構造体定義
struct Person {
    string name;
    int age;
    int height;
};

// 構造体リテラル(名前付き初期化)
Person p1 = {name: "Alice", age: 25, height: 165};

// 位置ベース初期化
Person p2 = {"Bob", 30, 180};

// 構造体のUnion型
typedef PersonData = Person | string | int;

// 配列のUnion型
typedef ArrayUnion = int[3] | bool[2];
typedef NumberArrays = int[3] | int[5];
```

 柔軟な初期化方法をサポート

構造体 - アクセス修飾子

🔒 private修飾子

```
struct Person {  
    private string ssn; // 外部から見えない  
    string name;  
    int age;  
}  
  
void example() {  
    Person p;  
    p.name = "Alice"; // OK  
    p.age = 30; // OK  
    // p.ssn = "123-45"; // エラー!  
}
```

- ▶ **カプセル化**
外部から直接アクセスできない
- ▶ **データ保護**
内部実装の隠蔽が可能

⚡ default修飾子

```
struct Config {  
    string host;  
    int port;  
    default string name = "default-server";  
}  
// ▲ default修飾子は1つだけ設定可能  
  
void example() {  
    Config c1;  
    // c1.name = "default-server" (自動設定)  
    c1.host = "localhost"; // 明示的に設定  
    c1.port = 8080; // 明示的に設定  
  
    Config c2 = {  
        host: "example.com",  
        port: 3000,  
        name: "custom-server" // 上書き可能  
    };  
}
```

- ▶ **1つのみ設定可能**
構造体に1つだけdefault指定
- ▶ **初期化の簡略化**
デフォルト値を自動設定
- ▶ **上書き可能**
必要に応じて明示的に設定

💡 **private**で安全性を確保、**default**で利便性を向上



Interface - トレイトシステム

💡 インターフェース定義

```
// インターフェース定義(Rustのtraitに相当)
interface Drawable {
    void draw();
    Point getPosition();
}

interface Clickable {
    void onClick();
    bool isClickable();
}

// 構造体定義
struct Button {
    string label;
    Point position;
    bool enabled;
};
```

► 振る舞いを定義する契約

Impl - インターフェースの実装

⚙️ 実装ブロック

```
// インターフェースの実装
impl Drawable for Button {
    void draw() {
        println("Drawing button: ", this.label);
        // 描画処理
    }

    Point getPosition() {
        return this.position;
    }
}

impl Clickable for Button {
    void onClick() {
        if (this.enabled) {
            println("Button clicked: ", this.label);
        }
    }

    bool isClickable() {
        return this.enabled;
    }
}
```

🚀 複数インターフェースの実装が可能

ポリモーフィズムの実現

🎨 インターフェース型として扱う

```
// インターフェース型として扱う
void renderUI(Drawable* items[], int count) {
    for (int i = 0; i < count; i++) {
        items[i]->draw();
    }
}

// 複数のインターフェースを要求
void handleInteraction(Drawable | Clickable widget) {
    widget.draw();
    if (widget.isClickable()) {
        widget.onClick();
    }
}

// 使用例
Button button = {
    label: "Submit",
    position: {x: 100, y: 200},
    enabled: true
};

// インターフェース型として扱える
Drawable* drawable = &button;
renderUI(drawable);
```

🔗 Rustのトレイトシステムの良さをCbに導入

コンストラクタ/デストラクタ & defer

🔨 コンストラクタ/デストラクタ

```
struct FileHandle {  
    private int fd;  
    private string path;  
  
    // コンストラクタ  
    constructor(string filepath) {  
        this.path = filepath;  
        this.fd = open(filepath);  
        println("File opened: {filepath}");  
    }  
  
    // デストラクタ  
    destructor() {  
        if (this.fd >= 0) {  
            close(this.fd);  
            println("File closed: {this.path}");  
        }  
    }  
  
    void example() {  
        FileHandle file("data.txt");  
        // 使用...  
    } // スコープを抜けると自動でデストラクタ呼び出し
```

⟳ defer - 遅延実行

```
void processFile(string path) {  
    int fd = open(path);  
    defer close(fd); // 関数終了時に自動実行  
  
    if (!isValid(fd)) {  
        return; // ここでもclose()が呼ばれる  
    }  
  
    string data = read(fd);  
    defer println("Processing complete");  
  
    processData(data);  
    // 関数終了時:  
    // 1. println("Processing complete")  
    // 2. close(fd) (逆順で実行)  
}  
  
// Go言語風のリソース管理  
void multipleDefer() {  
    defer println("3: Last");  
    defer println("2: Middle");  
    defer println("1: First");  
    // 実行順: 1 → 2 → 3 (LIFO)  
}
```

▶ LIFO順で実行

登録と逆順でクリーンアップ

▶ 例外安全

どんな終了経路でも実行

🛡 RAI + defer で確実なリソース管理を実現



Async/Await - 非同期関数の基本

⚡ 非同期関数の定義

```
// 非同期関数の定義
async int fetchData(string url) {
    println("Fetching: {url}");
    sleep(100); // ネットワーク遅延をシミュレート
    yield; // 明示的な協調ポイント
    println("Completed: {url}");
    return 42; // 取得したデータ数
}

// 並行実行の例
async void processMultipleTasks() {
    // 複数のタスクを同時に開始
    Future<int> task1 = fetchData("api/users");
    Future<int> task2 = fetchData("api/posts");
    Future<int> task3 = fetchData("api/comments");

    // すべての結果を待つ
    int users = await task1;
    int posts = await task2;
    int comments = await task3;

    println("Total: ", users + posts + comments);
}
```

⌚ 協調的マルチタスクで効率的な並行処理

イベントループと協調的動作

⌚ イベントループの仕組み

- ▶ 協調的マルチタスク
各タスクが自発的にCPUを譲る
- ▶ **yield** による制御
明示的な実行権の譲渡ポイント
- ▶ Auto-yield機能
async関数は1処理ごとに自動yield
- ▶ 軽量な並行処理
OSスレッドを使わない効率的な実装

⌚ sleep関数の実装

- ▶ 時間経過のみを監視
実際にスレッドをブロックしない
- ▶ イベントループで管理
タイムアウト時刻を記録して待機
- ▶ 他タスクに譲る
sleep中は他のタスクが実行可能

```
// sleepの動作イメージ
async void example() {
    println("Start");
    sleep(1000); // 1秒待機
    // ← ここで他のタスクが実行される
    println("After 1 second");
}
```

⌚ Auto-yieldにより、明示的なyield不要で協調的動作を実現

エラー伝播演算子

⚡ Result/Option型の伝播

```
// ?演算子 - エラーの早期リターン
Result<int, string> calculate() {
    int x = getValue()?;
    // Errなら即リターン
    int y = getAnother()?;
    // Errなら即リターン
    return Result<int, string>::Ok(x + y);
}

// try式 - Result/Optionのアンラップ
Result<int, string> processValue() {
    int value = try getValue(); // 成功時は値を取得
    return Result<int, string>::Ok(value * 2);
}

// checked演算 - オーバーフロー検出
Result<int, OverflowError> safeAdd(int a, int b) {
    int sum = checked(a + b); // オーバーフロー時はErr
    return Result<int, OverflowError>::Ok(sum);
}
```

🛡 Rustライクな安全なエラー処理機構

モジュールシステム - import

モジュールのインポート

```
// モジュールのインポート
import math;
import utils;
import io;

// 特定の関数・型のみインポート
import { sin, cos, sqrt } from math;
import { HashMap, Vector } from collections;

// 相対パスでのインポート
import ./local_module;
import ../parent_module;

// 条件付きコンパイル
#ifndef DEBUG
#define LOG(msg) println("[DEBUG] ", msg)
#else
#define LOG(msg) // 空のマクロ
#endif
```

📦 モダンなモジュールシステム

モジュールシステム - export

➡ モジュールのエクスポート

```
// 関数のエクスポート
export int add(int a, int b) {
    return a + b;
}

// 型のエクスポート
export struct Point {
    long x,
    long y
}

// 複数の要素をまとめてエクスポート
export {
    multiply,
    divide,
    Vector,
    Matrix
};

// デフォルトエクスポート
export default int main() {
    return 0;
}
```

📦 明示的なエクスポートで公開APIを制御

プリプロセッサ - 基本的なマクロ

🔧 マクロ定義と展開

```
// 単純なマクロ
#define PI 3.14159265359
#define MAX_SIZE 1024

// 関数マクロ
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

// 複雑なマクロ
#define FOR_EACH(item, array, size) \
    for (int _i = 0; _i < (size); _i++) { \
        auto item = (array)[_i];

#define END_FOR_EACH }

// 使用例
int[5] numbers = {1, 2, 3, 4, 5};
FOR_EACH(num, numbers, 5)
    println(num * 2);
END_FOR_EACH
```

⌚ 定数定義と簡単なコード生成



Enum (列挙型)



型安全な定数定義

```
// 値を指定したEnum
enum Status {
    OK = 200,
    NOT_FOUND = 404,
    ERROR = 500
}

// 自動採番されるEnum
enum Color { RED, GREEN, BLUE }

// 使用例
Status code = Status::OK;
if (code >= Status::NOT_FOUND) {
    println("Not found");
}

Color myColor = Color::RED;
switch (myColor) {
    case(Color::RED):  println("赤色"); break;
    case(Color::GREEN): println("緑色"); break;
    case(Color::BLUE):  println("青色"); break;
}
```

■ Cライクな列挙型で可読性向上

関数ポインタ

🔗 高階関数とコールバック

```
// 関数ポインタ型定義(戻り値の型のみ)
typedef int* Operation;

int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }

// 関数ポインタの使用
Operation op = &add;
int result = op(5, 3); // 8

// 関数を返す関数
Operation getOp(char symbol) {
    if (symbol >= '+') return &add;
    if (symbol >= '*') return &mul;
    return nullptr;
}

// 高階関数の例
void apply(Operation op, int x, int y) {
    println("Result: ", op(x, y));
}

apply(&add, 10, 20); // Result: 30
apply(&mul, 5, 7); // Result: 35
```

⚡ 動的な関数選択とコールバック機構

メモリ管理

動的メモリ割り当て

```
// 配列の動的割り当て
int[100]* arr = new int[100];
for (int i = 0; i < 100; i++) {
    arr[i] = i * 2;
}
delete[] arr;
```

```
// 構造体の動的割り当て
struct Data {
    int value;
    string name;
};

Data* data = new Data;
data->value = 42;
data->name = "example";
delete data;

// 手動メモリ管理(低レベル)
void* buffer = malloc(1024);
memcpy(buffer, sourceData, size);
free(buffer);
```

↗ C++ライクなメモリ管理機能

パターンマッチング

強力な分岐機構

```
// switch文の高度な機能
int processCode(int code) {
    switch (code) {
        case(1 | 2 | 3) {
            return code * 10; // OR条件
        }
        case(10...20) {
            return code * 5; // 範囲マッチング
        }
        else {
            return -1;
        }
    }
}

// match文(Option/Result用)
Result<int, string> result = divide(10, 2);
match (result) {
    Ok(value) => { println("成功: ", value); }
    Err(error) => { println("エラー: ", error); }
}
```

⌚ Rustライクな網羅的パターンマッチ

ジェネリクス



型パラメータによる汎用プログラミング

```
// ジェネリック構造体
struct Stack< T > {
    T[100] items;
    int top;

    void push(T item) {
        items[top++] = item;
    }

    T pop() {
        return items[--top];
    }
};

// 使用例
Stack<int> intStack;
intStack.push(42);
int value = intStack.pop();
```

⌚ C++スタイルのテンプレートシステム

ジェネリクス - Interface/Impl

⌚ ジェネリックなインターフェース実装

```
// ジェネリックインターフェース
interface Container<T> {
    void add(T item);
    T get(int index);
    int size();
};

// ジェネリック構造体
struct Box<T> {
    T[100] items;
    int count;
};

// ジェネリックなImpl
impl Container<T> for Box<T> {
    void add(T item) {
        this.items[this.count++] = item;
    }

    T get(int index) {
        return this.items[index];
    }

    int size() {
        return this.count;
    }
};

// 使用例
Box<int> intBox;
intBox.add(42);
intBox.add(100);
int val = intBox.get(0); // 42
```

⌚ **impl I<T> for S<T>** - 型パラメータを保持したまま実装

FFI（外部関数インターフェース）

🌐 C/C++ライブラリとの連携

```
// C/C++ライブラリの利用
use foreign.math {
    double sin(double x);
    double cos(double x);
    double sqrt(double x);
}

// 外部関数の呼び出し
double angle = 1.57;
double y = sin(angle);
double x = cos(angle);

// システムライブラリの利用
use foreign.c {
    void* malloc(int size);
    void free(void* ptr);
    int printf(string fmt, ...);
}

// C標準ライブラリの活用
void* buffer = malloc(1024);
free(buffer);
```

🔗 膨大なC/C++エコシステムへのアクセス

Option型 - Null安全性

組み込みOption型

```
// Option型でnullを安全に扱う
Option<int> findValue(string key) {
    if (map.contains(key)) {
        return Option<int>::Some(map[key]);
    }
    return Option<int>::None;
}

// パターンマッチングで処理
Option<int> result = findValue("age");
match (result) {
    Some(value) => {
        println("Found: ", value);
    }
    None => {
        println("Not found");
    }
}
```

🛡 Null参照エラーを型レベルで防止

Result型 - エラー処理

組み込みResult型

```
// Result型でエラーを明示的に扱う
Result<int, string> divide(int a, int b) {
    if (b >== 0) {
        return Result<int, string>::Err("Division by zero");
    }
    return Result<int, string>::Ok(a / b);
}

// 使用例
Result<int, string> res = divide(10, 2);
match (res) {
    Ok(value) => {
        println("Result: ", value);
    }
    Err(error) => {
        println("Error: ", error);
    }
}

// チェイン処理(計画中)
Result<int, string> calculate() {
    return divide(100, 5)
        .map(|x| x * 2)
        .and_then(|x| divide(x, 10));
}
```

✓ エラーを値として扱い、明示的に処理



Cb 開発環境サポート - VSCode拡張

🎨 シンタックスハイライト

- ▶ **TextMate文法ファイル**
VSCode用のシンタックス定義を実装
- ▶ **キーワードのハイライト**
async, match, impl等の強調表示
- ▶ **型の色分け**
Option, Result, Future等
- ▶ **コメントと文字列**
適切な色分けで可読性向上

📦 拡張機能 :

[.vscode/extensions/cb-language](#)

✨ サポート内容

- ▶ **ファイル認識**
.cb 拡張子を自動認識
- ▶ **括弧マッチング**
対応する括弧をハイライト
- ▶ **コメント切り替え**
Cmd/Ctrl + / でコメント化
- ▶ **インデント**
自動インデント機能

💡 今後の展開

LSP (Language Server Protocol) を実装し、補完、定義ジャンプ、エラー検出等の高度な機能を提供予定

🛠 快適な開発体験のために、基本的なエディタサポートを実装



セクション1まとめ：Cbの言語設計

🎯 設計思想

- ▶ **C++をベースに**
静的型付け、高速な実行、システムプログラミング
- ▶ **多言語からいいとこ取り**
モダン言語の優れた機能を統合
- ▶ **実用性重視**
開発者の生産性とコードの安全性を両立

⭐ 採用した機能

- ▶ **Rust風**
implブロック、Option/Result型、パターンマッチング
- ▶ **TypeScript風**
ユニオン型、リテラル型、型推論
- ▶ **Go風**
Interface、defer
- ▶ **Python風**
async/await、文字列補間

💡 「多言語のベストプラクティスを1つの言語に」

それぞれの言語の良いところを取り入れた、実用的で安全な言語を目指す

SECTION 2

バイブルコーディング

AI駆動開発による効率的な言語実装

AIによる開発の実態



Cbプロジェクトの開発体制

- ▶ 設計・ドキュメント作成
ほぼすべてAI
- ▶ コード実装
90%以上をAIが生成
- ▶ テスト作成
AIが自動生成
- ▶ リファクタリング
AIと人間の協働



使用しているAIツール

- ▶ GitHub Copilot Pro+
コード補完・生成
- ▶ Claude Code
複雑な実装・リファクタリング
- ▶ よく使うモデル
Claude Sonnet4.5

⚡ 人間1人 + AI = 大規模プロジェクト開発が可能

AIによる開発のメリット

⚡ 圧倒的な開発速度

- ▶ 数千行のコードを数分で生成
- ▶ ボイラープレートの自動化
- ▶ 単純作業からの解放

📚 膨大な知識の活用

- ▶ 最新のベストプラクティス
- ▶ 多言語・多技術スタックに対応
- ▶ 即座のドキュメント参照

- 💡 一貫性のあるコード - 統一されたスタイルとパターン
- 💡 アイデアの即座の検証 - プロトタイプを高速作成

人間にしかできないこと

◉ 言語設計の思想・ビジョン

「どういう言語にしたいか」という方向性

- ▶ 言語の哲学・コンセプト
- ▶ ターゲットユーザー像
- ▶ 機能の優先順位
- ▶ トレードオフの判断

⚖️ 重要な意思決定

- ▶ アーキテクチャの選択
- ▶ 技術スタックの決定
- ▶ パフォーマンスと可読性のバランス
- ▶ 実装方針の転換（例: Yacc/Lex → 手書きパーサ）

💡 AIは手段、方向性を決めるのは人間

AI開発の課題

⚠️ 膨大なドキュメント管理

- ▶ AIに任せるほど仕様書が増える
- ▶ ドキュメントの一貫性維持が困難
- ▶ 更新コストの増大

📦 ブラックボックス化

- ▶ 自分でも理解していない部分が多い
- ▶ コード生成のロジックが不透明
- ▶ デバッグ時の困難

- ⚠️ 1ファイル数千行の巨大ファイル生成 - 保守性の低下
- ⚠️ 人間側の深い理解が必要 - プロジェクト全体の熟知が必須

ドキュメント戦略のトレードオフ

✓ メリット

- ▶ プロンプトが簡潔に
「仕様書通りに実装して」で済む
- ▶ 一貫性の維持
全員が同じ仕様を参照
- ▶ AIの理解が深まる
コンテキストを共有しやすい

✗ デメリット

- ▶ 管理コストの増大
ドキュメント量が膨大に
- ▶ 更新の遅れ
実装とドキュメントの乖離
- ▶ 何が最新かわからない
バージョン管理の複雑化

⚠ 必要最小限のドキュメント + コードベースの真実



実際のドキュメント管理方法

📋 バージョン管理

IMPLEMENTATION_PRIORITY.md

現在の実装優先度と進捗状況

例: v0.14.0 統合テスト成功率 58% (493/849)

release_notes/v*.md

各バージョンのリリースノート

v0.10.0, v0.11.0, v0.12.0, v0.13.0...

docs/archive/releases/

詳細な実装計画と報告書のアーカイブ

✓ テストもドキュメント

tests/cases/

実行可能な仕様書として機能

849個のテストケースが言語仕様を表現

🤖 このスライド自体も

docs/presentation2/

✨ AIによって生成・管理

Gitで履歴を追跡、変更も容易

💡 ドキュメントもコードと同じ: バージョン管理 + 自動生成



定期的なリファクタリングの必要性

⚠️ よくある問題

- ▶ 1ファイル数千行のモンスターファイル
- ▶ 過度に複雑なロジック
- ▶ 重複コードの蔓延
- ▶ テストが困難な構造

✓ 解決策

1. 動くものができた時点でリファクタリング
「動く → 美しく」の2段階
2. メンテナンス可能な粒度に分割
1ファイル1000行以下を目安に
3. AIと協力してリファクタリング
「このファイルを分割して」

⌚ 動作確認 → リファクタリング → テスト のサイクル

AI開発を支える3つの武器



意図通りに実装できているか検証

- ▶ AIのコード品質を確認
- ▶ リファクタリングの安全性担保
- ▶ 回帰バグの早期発見



- ▶ **デバッグモード**
実行フローを可視化
- ▶ **サニタイザー**
メモリ・未定義動作を検出
- ▶ 詳細なログ出力、問題箇所の特定
が容易

💡 開発しているプロジェクトの状態がAIにも人間にもわかるツールは便利

テスト - AI開発の品質保証

🎯 テストの役割

- ▶ AIの実装が仕様通りか確認
- ▶ リファクタリング時の安全網
- ▶ 回帰バグの防止
- ▶ ドキュメントとしての機能

✨ cbプロジェクトでの実践

200個以上のテストケースを用意

- ▶ 基本構文テスト
- ▶ 型システムテスト
- ▶ HIR変換テスト
- ▶ コード生成テスト

🚀 テストがあるから大胆にリファクタリングできる

テストの実例



簡単なテストケース

```
// tests/cases/basic/test_arithmetic.cb
int main() {
    int a = 10;
    int b = 20;
    int sum = a + b;
    assert(sum >= 30); // 期待値チェック
    return 0;
}
```



テスト実行スクリプト

```
# tests/integration/run_unified_tests.sh
for test in tests/cases/**/*.cb; do
    echo "Testing: $test"
    ./cb compile $test -o /tmp/test_out
    if [ $? -eq 0 ]; then
        /tmp/test_out # 実行して結果確認
        echo "✓ PASS"
    else
        echo "✗ FAIL: Compilation error"
    fi
done
```

⚡ 200+ テストが数秒で完了

デバッグモード - 実行フローの可視化

🔍 デバッグモードとは

プログラムの実行過程を詳細にログ出力

- ▶ 関数呼び出しのトレース
- ▶ 変数の値の変化
- ▶ 条件分岐の判定結果
- ▶ エラーが発生した位置

💡 なぜ必要か

- ▶ AIが生成したコードの動作確認
- ▶ バグの原因を素早く特定
- ▶ 複雑なロジックの理解
- ▶ 本番環境での問題調査

❖ --debug などのオプションで詳細ログを有効化

デバッグモードの実例



コード内でのデバッグ出力

```
#ifdef DEBUG_MODE
    std::cerr << "[DEBUG] Parsing function: "
        << func_name << std::endl;
    std::cerr << "[DEBUG] Parameter count: "
        << params.size() << std::endl;
#endif
```



実行例

```
# 通常実行
$ ./cb compile test.cb
Compilation successful

# デバッグモード実行
$ ./cb --debug compile test.cb
[DEBUG] Parsing function: main
[DEBUG] Parameter count: 0
[DEBUG] Entering HIR conversion
[DEBUG] Converting expression: BinaryOp
[DEBUG] Left: IntLiteral(10)
[DEBUG] Right: IntLiteral(20)
Compilation successful
```

⌚ 問題箇所が一目瞭然

サニタイザー - メモリ安全性の番人

🛡 サニタイザーとは

実行時にメモリエラーや未定義動作を検出

- ▶ **AddressSanitizer (ASan)**
メモリリーク・バッファオーバーフロー
- ▶ **UndefinedBehaviorSanitizer (UBSan)**
未定義動作
- ▶ **ThreadSanitizer (TSan)**
データ競合

💡 AI開発での重要性

- ▶ AIが気づかないメモリエラー発見
- ▶ セグフォの原因を即座に特定
- ▶ 未定義動作の早期検出
- ▶ バグ修正時間の大幅短縮

✖ コンパイル時に `-fsanitize=address` で有効化

サニタイザーの実例

🔧 Makefileでの設定

```
# AddressSanitizer有効化
CXXFLAGS += -fsanitize=address
CXXFLAGS += -fno-omit-frame-pointer
LDFLAGS += -fsanitize=address

# UndefinedBehaviorSanitizer
CXXFLAGS += -fsanitize=undefined
```

🐛 検出例: メモリリーク

```
==12345==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 100 bytes in 1 object(s) allocated from:
 #0 in operator new(unsigned long)
 #1 in Parser::parseExpression() parser.cpp:234
 #2 in Parser::parseStatement() parser.cpp:156

SUMMARY: AddressSanitizer: 100 bytes leaked in 1 allocation(s)
```

⚡ 問題の行番号まで正確に教えてくれる

実例: Yacc/Lex → 手書きパーサへの移行

✖ 問題発生

- ▶ Yacc/Lexでは複雑な構文に対応困難
- ▶ エラーメッセージが不親切
- ▶ 拡張性に限界

⟳ 移行の決断 → ✓ 移行成功

手書き再帰下降パーサへ変更

- ▶ AIが既存テストを活用
- ▶ 200+のテストケースが安全網に
- ▶ テストが通るまでAIが修正
- ▶ 全テスト通過で完了確認

🚀 テストがあれば大胆な技術選択も可能



セクション2まとめ：AI開発の現実

🚀 AI開発の強み

- ▶ アイデア以外は爆速
実装・デバッグが圧倒的に速い
- ▶ デバッグ手法がAIにも有効
テスト、デバッグモード、Sanitizer
- ▶ 人間とAIの相乗効果
より効率的な開発サイクル

⚠️ ドキュメント戦略の落とし穴

- ▶ 短期的には有効
AIに仕様を理解させやすい
- ▶ 長期的な課題
 - ・AIがコンテキストを忘れる
 - ・実装と仕様の乖離が発生
- ▶ 膨大なドキュメント量
管理が困難に

💡 推奨アプローチ：人間用の必要最小限の仕様書 + リリースノート
全体像を把握しやすく、メンテナンスも容易

SECTION 3

プロジェクトを通して学んだこと

成功と失敗から得た教訓

良かったこと

✓ AI開発の強み

- ▶ 実装速度の向上

従来の数倍の速度で機能実装

- ▶ 学習コストの削減

未知の領域も素早くキャッチアップ

- ▶ コード品質の向上

多様なパターンの提案を受けられる

- ▶ ドキュメント整備

AIが仕様書や説明を生成してくれる

🎯 技術的成果

- ▶ 包括的な言語設計

多言語の良いとこ取りを実現

- ▶ モダンな機能

async/await、パターンマッチング等

- ▶ 実用的なツール群

テスト、デバッグ、Sanitizer完備

- ▶ 繙続的な改善

リファクタリングを繰り返せた

⚡️ AI駆動開発により、個人でも大規模プロジェクトに挑戦できた



課題・ダメだったこと

⚠️ AI開発の限界

- ▶ アーキテクチャ設計
大局的な設計判断は人間が必要
- ▶ 複雑なバグ
AIだけでは解決困難な問題も
- ▶ コンテキストの喪失
長期プロジェクトで情報が散逸
- ▶ 品質のブレ
生成されたコードの質にばらつき

🔧 技術的課題

- ▶ Yaccの限界
複雑な言語仕様には不向き
- ▶ デバッグの難しさ
生成されたコードの理解に時間
- ▶ パフォーマンス最適化
AIは最適解を出せない場合も
- ▶ ドキュメントの肥大化
管理が困難になりがち
- ▶ テストカバレッジ
網羅的なテストは人間が設計

💡 AIは強力なツールだが、人間の判断と管理が不可欠

💰 開発コストの現実

📊 開発期間とツール

▶ 開発期間

2024年7月～現在（3～4ヶ月）

▶ GitHub Copilot Pro+

\$39/月（約¥7,000）

開発当初から使用

起床～就寝まで使用でプレミアム
リクエスト1500超

▶ Claude (\$200プラン)

\$200/月（約¥30,000）

今月から使用開始

💡 コストパフォーマンス比較

✓ GitHub Copilot Pro+

¥7,000/月

- ▶ コード補完に特化
- ▶ コスパ最強
- ▶ 日常的な開発に最適
- ▶ 終日使用で1500リクエスト超

⚠ Claude \$200プラン

¥30,000/月

- ▶ 複雑な設計・リファクタリング
- ▶ 高コストだが高性能
- ▶ 特定タスクで威力発揮

💰 結論：コスパ的にはCopilotが圧勝。Claudeは必要な時だけ使うのが賢明



学んだこと・今後に活かせること

技術的学び

- ▶ コンパイラの仕組み
字句解析、構文解析、コード生成を体験
- ▶ 型システムの設計
Union型、Option/Result型の実装
- ▶ 非同期処理
イベントループとyieldの実装
- ▶ C++の深い理解
メモリ管理、テンプレート等

開発プロセスの学び

- ▶ AIとの協働方法
効果的なプロンプト、レビュー手法
- ▶ 段階的な開発
小さく作って改善を繰り返す
- ▶ ドキュメント戦略
最小限+リリースノートが有効
- ▶ テスト駆動開発
AIにもテストが重要

★ AI時代の開発：アイデアと方向性は人間、実装はAIと協力

今後の展望



短期目標 (v0.14.0～v0.16.0)

- ▶ パターンマッチングの完成
より複雑なパターンに対応
- ▶ **async/await**の更なる実装
非同期処理の実用化
- ▶ 標準ライブラリの拡充
ファイルI/O、ネットワーク等
- ▶ エラーメッセージの改善
分かりやすいエラー表示

★ 長期目標 (v1.0.0以降)

- ▶ **WebAssembly**対応
ブラウザでの実行を実現
- ▶ **LSP (Language Server)** 開発
VSCode等でのサポート
- ▶ パッケージマネージャ
ライブラリの管理システム
- ▶ コミュニティ形成
OSS化してユーザーを増やす



理想的の言語を目指して、継続的に進化させていきます



ご清聴ありがとうございました！

📱 開発状況を垂れ流し中：[osdev-jp](#)

🔗 GitHub：[shadowlink0122/Cb](#)

🐦 Twitter：[@sl_0122](#)

