SecHack365

Cynops

学習駆動コース 坂井ゼミ
恵祖茂 寿歩

A C Compiler Targeting Brainfuck

Cynops

- **C言語**を**Brainfuck**へ変換するコンパイラ
- **再帰関数**に対応
- **構造体**に対応
- **ポインタ**に対応

エスケープシーケンスによる画面描画

```
  a b c d e f g h
1|*|*|*|*|*|*|*|*|
2|*|*|O|*|*|O|*|*|
3|*|O|*|*|O|*|*|*|
4|*|*|O|*|O|*|*|*|
5|*|O|O|*|*|*|*|*|
6|*|O|O|*|O|*|*|*|
7|*|*|*|*|*|O|*|*|
8|*|*|*|*|*|*|O|*|

Current turn: *(Black)
Game over.
Black (*) : 51
White (O) : 13
Black wins!
```

ボット搭載オセロ

C言語

- **抽象度の高い**言語
- **変数・関数**が使える

Brainfuck

- **8命令のみ**のシンプルな言語
- **無限長テープ**＋**ヘッド**の計算モデル

命令	動作の説明
>	ポインタを右のセルへ移動
<	ポインタを左のセルへ移動
+	現在のセルの値を 1 増加
-	現在のセルの値を 1 減少
.	現在のセルの値を文字として出力
,	1 文字入力し，現在のセルに格納
[	現在のセルが 0 なら対応する] の次へジャンプ
]	現在のセルが 0 でなければ対応する [へ戻る

```c
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

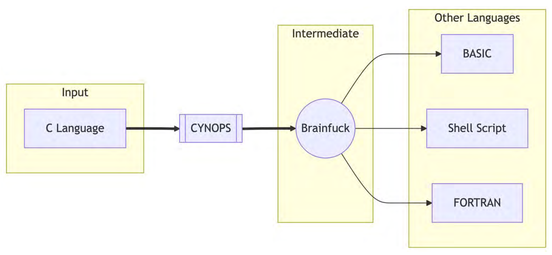
C言語でのHello world

```
++++++++
[
    >++++
    [
        >++>+++>+++>+<<<<-
    ]
    >+>+>->>+[<]<-
]
>>.>---.+++++++..+++.>>.<-.<.+
++.------.--------.>>+.>++.
```

BrainfuckでのHello world

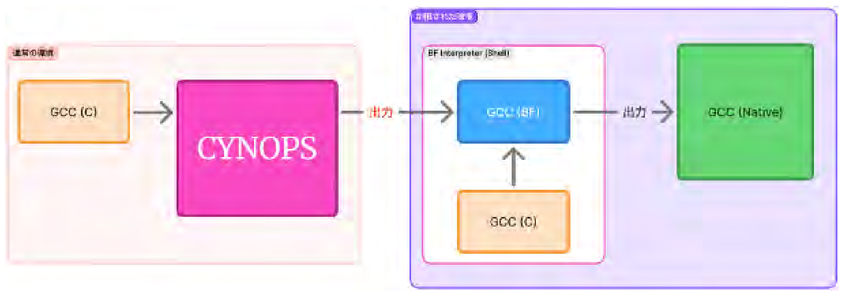
Brainfuckを経由して!

仕様が簡潔
→ **処理系の実装**が容易
→ **多言語変換**が容易
Brainfuck が動けば C も動く



実用性の観点からの可能性

最初にコンパイラを BF 上でコンパイルして
ネイティブ化すれば、その後のコンパイルは
ネイティブ環境で行えるため、高速化？




```
go(){
((t[i]+=8));while((t[i]&65535));do
((++i,t[i]+=4,t[i+1]+=t[i]*2,t[i+2]+=t[i]*3,t[i+3]+=t[i]*3,t[i+4]
+=t[i],t[i]=0,++i,t[i]+=1,++i,t[i]+=1,++i,t[i]-=1,i+=2,t[i]+=1
));while((t[i]&65535));do
((--i));done
((--i,t[i]-=1));done
((i+=2));putbyte
((++i,t[i]-=3));putbyte
((t[i]+=7));putbyte
putbyte
((t[i]+=3));putbyte
((i+=2));putbyte
((--i,t[i]-=1));putbyte
((--i));putbyte
((t[i]+=3));putbyte
((t[i]-=6));putbyte
((t[i]-=8));putbyte
((i+=2,t[i]+=1));putbyte
((++i,t[i]+=2));putbyte
}
```

Brainfuck 経由でshell scriptに変換

世界最先端 ・唯一無二

- 関数ポインタおよび構造体に対応した C→BF コンパイラは **Cynops のみ**
- 関数スコープを跨ぐポインタ操作や多次元配列を扱えるような実装も稀

全てサポート

- 再帰関数
- 関数ポインタ
- 構造体
- 関数を跨ぐポインタ操作
- 多次元配列

追加機能・独自機能

- 配列は値として扱う

```c
int a[3] = {1, 2, 3};
int b[3] = a;
```

```c
int add(int a, int b) { return (a + b); }
int sub(int a, int b) { return (a - b); }
int mul(int a, int b) { return (a * b); }
int div(int a, int b) { return (a / b); }
int mod(int a, int b) { return (a % b); }

int apply(int a, int b, int (*fn)(int, int)) { return (*fn)(a, b); }

void main(void) {
    int a = 5, b = 5;
    int (*fns[])(int, int) = {&add, &sub, &mul, &div, &mod};

    int i = 0;
    for (i = 0; i < 5; i++) {
        print_int(apply(a, b, fns[i]));
    }

    return;
}
```

コンパイル可能なプログラムの例

- 故に配列をそのまま代入・返却できる

```c
// fn(char [6]) -> char [6]
char(fn(char arg[6]))[6] {
  return arg;
}
```

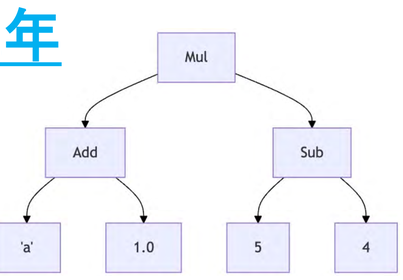
- ポインタ演算は禁止,参照のみ

```c
int a[3] = {1, 2, 3};
int* p = &a[2];  // OK
p = p + 1;  // NG
```

SecHack365での1年

6月
- C言語の構文規則を分析
- 再帰下降**パーサ実装**

```
('a' + 1.0) * (5 - 4)
```



7月
- **スコープ管理**
  - シャドーイング等
- **型**情報の**パース**

```c
void (*complex(int, void (*)(int)))(int);

int sum(int a, int b) {
    int x = a + b;
    {
        int x = 2;
        int y = x + 1;
    }
    int z = x;
    return z;
}
```

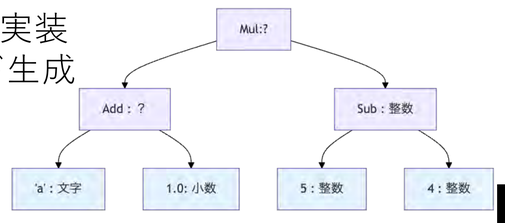
```
Scope tree
└─ : Scope[0] (2 symbols)
   ├─ complex: fn(int, *fn(int) -> void) -> *fn(int) -> void
   ├─ sum: fn(int, int) -> int
   └─ : Scope[0.0] (2 symbols)
      ├─ a: int
      ├─ b: int
      └─ : Scope[0.0.0] (2 symbols)
         ├─ z: int
         ├─ x: int
         └─ : Scope[0.0.0.0] (2 symbols)
            ├─ x: int
            └─ y: int
```

型の完全一致を要求する設計を採用した。Brainfuck対応には役立ったが、その分、非常に書きにくくなった。

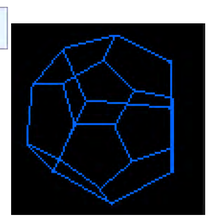
成立しないにもかかわらず、型のパースは想像以上に困難だった。

```c
void putchar(char);
void printf(char (*s)[0]);

void main(void) {
    int arr[5] = {1, 2, 3, 4, 5};
    int* p = arr;  // 禁止,expected `*int`, found `*[int; 5]`

    char ezomo[6] = "ezomo\0";
    printf((char (*)[0]) & ezomo);  // 面倒で危険なキャストが必要
    putchar((char)((int)'0' + 4));  // 扱が一致しないと演算ができない
    return;
}
```

8月
- **型一致チェック**の実装
- LLVM IR向けコード生成
- **型推論**の一部導入



9月
- LLVM IR向けコンパイラの実装を終了
- エスケープシーケンスによる画面描画



10月
- 既存の**C→BFコンパイラ**の分析
- コンパイル手法の検討・試行錯誤

単一のスタックのみで管理され、絶対アドレスが存在しない環境において、コンパイル時に確定させたスタック高を用いて変数アクセスを実現している。
再帰呼び出し等の動的な状況下でも基準位置を自動で継承させることで、一貫したメモリ操作を実現しています。

11月
- Goto及びStackを用いた関数の作成
- スタック深度を内部管理し、**安定したポインタ**を実現

このような基本的な機能すらないため、まずCで関数を作成してコンパイルし、演算を作成した関数の呼び出しに差し替える形で実装する。

12月
- 加減乗除・modの実装
- 入出力機能の追加
- デモプログラムの作成
- 固定小数点演算の一部対応

型の厳密一致を要求する設計を採用したことで、コンパイル時に型に応じた演算関数の切り替えが可能となり、固定小数点演算への対応を比較的容易に実現できました。

```c
void print_double(double);
void putchar(char);

void main(void) {
    print_double(3.141 * 2.718), putchar('\n');
    print_double(3.141 + 2.718), putchar('\n');
    print_double(3.141 - 2.718), putchar('\n');

    return;
}
```

```
❯ ./run.sh test.c
    Finished `dev` profi
    Running `target/deb
8.5372
5.8590
0.4230
```

1月
- 第11回 山陰 数学と基礎論研究集会で発表

2月
- 32bit対応への挑戦

Stack Frame Layout (Function Call)

Higher Address (Top of Stack)

Arguments (n…1) [呼び出し先への引数伝達]
Old Global Address [基準ポインタの退避]
Return Point [戻り先ラベル ID]
Return Value Area [Alloc (Return Type Size)]

Lower Address (Bottom of Frame)

謝辞

本プロジェクトの遂行にあたり、SecHack365運営事務局ならびにトレーナーの皆様に心より感謝申し上げます。特に、一年間にわたり熱血なご指導と示唆に富む助言を惜しみなく与えてくださった坂井弘亮トレーナーに、ここに記して深く感謝の意を表します。
応募書類の添削から日々の壁打ちまで一年を通してご支援くださった米子工業高等専門学校の内田雅人先生に、心より御礼申し上げます。
そして何よりも、本活動を日々支え続けてくれた家族に、最も深い感謝をここに表します。

NICT National Institute of Information and Communications Technology

GitHub: https://github.com/ezomo
Twitter(X): https://twitter.com/Ezomo_Daniel
Others : http://daniel.ezomo.jp

Cynops