

# コンパイラ実験 第1回

江口 慎悟  
押川 広樹  
塚田 武志

# 実験の大目標

## 計算機システムを作る

- 計算機システムが（ハードもソフトも）  
black box でないことを「実感」

# 全体の進め方

- 班（3～5人？）に分かれて進む
    - うち一人がコンパイラ係
  - 具体的な制作物
    - 自作の CPU の設計・実装
    - 自作 CPU 向けのコンパイラ
    - ...
- レイトレーシングプログラムを  
正しく動作させることが条件

# コンパイラ係の目標

自作 CPU 向け  
コンパイラの実装

# コンパイラ実験の進め方

- 毎回講義を 15分～45分 ほど

- 対象言語：OCaml もどき
- 記述言語：OCaml

- あとは各自で

- 講義資料はサポートサイトから

<http://www.kb.is.s.u-tokyo.ac.jp/~tsukada/cgi-bin/m/>

# 担当者 と 問い合わせ先

- 江口 慎悟
- 押川 広樹
- 塚田 武志

■ [compiler-jikken@kb.is.s.u-tokyo.ac.jp](mailto:compiler-jikken@kb.is.s.u-tokyo.ac.jp)

# お手本とするコンパイラ

## ○ MinCaml

- 作者： 住井 英二郎
- 対象言語 OCaml もどき
- 記述言語： OCaml
- 対象CPU： SPARC, PowerPC, x86 など

<http://esumii.github.io/min-caml/>

# 講義の内容（予定）

## ○ 基礎編

- コンパイラの概要
- K正規化、 $\alpha$ 変換、最適化
- クロージャ変換
- 仮想マシンコード割り当て
- レジスタ割り当て
- アセンブラ作成、末尾呼び出し最適化

## ○ 応用編

- 全6回



## 2 種類の課題

- 共通課題

- 全員（コンパイラ係を含む）が解く

- コンパイラ係向け課題

- コンパイラ係が解く
- ほかの人も問いてもよい

# 単位取得条件（コ係を除く）

○ 以下の両方ともを満たすこと

- 共通課題を **6つ以上** 提出
  - 余裕を持って出すのがおすすめ
- プロセッサ担当者の出す条件をクリア

※ 余帰納的な定義とする

# 単位取得条件（コ係）

- 自作 CPU（or シミュレータ）上で  
レイトレプログラムを完動させること
- 共通課題を **6つ以上** 提出
- コンパイラ係向け課題を **1つ以上** 提出
  - **口頭諮問あり**（例年は2月末～3月頭だった）
- プロセッサ担当の出す課題をクリア

# コンパイラの概要

# コンパイラとは

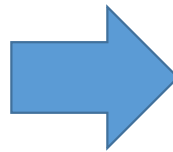
- ある言語のプログラムを別の言語に翻訳するシステム
- gcc
  - C（等）をアセンブリに
- javac
  - Java を JVM bytecode に
- ocamlc
  - OCaml を OCaml VM bytecode に

# コンパイルの例

## アセンブリ

### ML

```
let rec gcd m n =  
  if m = 0 then n else  
  if m <= n then  
    gcd m (n - m)  
  else  
    gcd n (m - n)
```



```
.section ".text"  
gcd.7:  
    cmp        %i2, 0  
    bne        be_else.17  
    nop  
    mov        %i3, %i2  
    retl  
    nop  
be_else.17:  
    cmp        %i2, %i3  
    bg         ble_else.18  
    nop  
    sub        %i3, %i2, %i3  
    b          gcd.7  
ble_else.18:  
    sub        %i2, %i3, %o5  
    mov        %i3, %i2  
    mov        %o5, %i3  
    b          gcd.7  
    nop
```

# コンパイラの難しさ

## 。言語間のギャップ

### ML

- 変数
- 関数・クロージャ
- 式・文

### アセンブリ

- レジスタ・メモリ
- ジャンプ・分岐
- CPU 命令

# 解決策：ちよつとずつ変換

## ML

- 変数
- 関数・クロージャ
- 式・文

クロージャ  
変換



## 中間言語 1

- 変数
- ジャンプ・分岐
- 式・文

仮想マシンコード生成



## 中間言語 2

- 変数
- ジャンプ・分岐
- CPU 命令

レジスタ  
割り当て



## アセンブリ

- レジスタ・メモリ
- ジャンプ・分岐
- CPU 命令





# MinCaml での実現

```
(* main.ml *)
let lexbuf outchan l =
  Id.counter := 0;
  Typing.extenv := M.empty;
  Emit.f outchan
    (RegAlloc.f
      (Simm.f
        (Virtual.f
          (Closure.f
            (iter !limit
              (Alpha.f
                (KNormal.f
                  (Typing.f
                    (Parser.exp Lexer.token 1))))))))))
```

字句解析

# MinCaml での実現

```
(* main.ml *)
let lexbuf outchan 1 =
  Id.counter := 0;
  Typing.extenv := M.empty;
  Emit.f outchan
    (RegAlloc.f
      (Simm.f
        (Virtual.f
          (Closure.f
            (iter !limit
              (Alpha.f
                (KNormal.f
                  (Typing.f
                    (Parser.exp Lexer.token 1))))))))))
```

構文解析

# MinCaml での実現

```
(* main.ml *)
let lexbuf outchan l =
  Id.counter := 0;
  Typing.extenv := M.empty;
  Emit.f outchan
    (RegAlloc.f
      (Simm.f
        (Virtual.f
          (Closure.f
            (iter !limit
              (Alpha.f
                (KNormal.f
                  (Typing.f
                    (Parser.exp Lexer.token 1))))))))))
```



型検査

# MinCaml での実現

```
(* main.ml *)
let lexbuf outchan l =
  Id.counter := 0;
  Typing.extenv := M.empty;
  Emit.f outchan
    (RegAlloc.f
      (Simm.f
        (Virtual.f
          (Closure.f
            (iter !limit
              (Alpha.f
                (KNormal.f
                  (Typing.f
                    (Parser.exp Lexer.token 1))))))))))
```



K 正規化



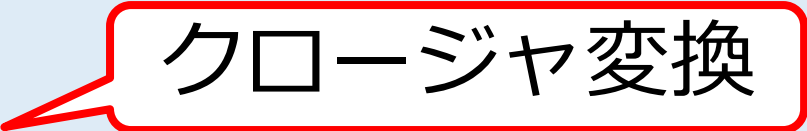
# MinCaml での実現

```
(* main.ml *)  
let lexbuf outchan 1 =  
  Id.counter := 0;  
  Typing.extenv := M.empty;  
  Emit.f outchan  
    (RegAlloc.f  
      (Simm.f  
        (Virtual.f  
          (Closure.f  
            (iter !limit  
              (Alpha.f  
                (KNormal.f  
                  (Typing.f  
                    (Parser.exp Lexer.token 1))))))))))
```

最適化

# MinCaml での実現

```
(* main.ml *)
let lexbuf outchan l =
  Id.counter := 0;
  Typing.extenv := M.empty;
  Emit.f outchan
    (RegAlloc.f
      (Simm.f
        (Virtual.f
          (Closure.f
            (iter !limit
              (Alpha.f
                (KNormal.f
                  (Typing.f
                    (Parser.exp Lexer.token 1)))))))))))
```




クローージャ変換



# MinCaml での実現

```
(* main.ml *)
let lexbuf outchan 1 =
  Id.counter := 0;
  Typing.extenv := M.empty;
  Emit.f outchan
    (RegAlloc.f
      (Simm.f
        (Virtual.f
          (Closure.f
            (iter !limit
              (Alpha.f
                (KNormal.f
                  (Typing.f
                    (Parser.exp Lexer.token 1)))))))))))
```



仮想マシンコード生成

# MinCaml での実現

```
(* main.ml *)
let lexbuf outchan 1 =
  Id.counter := 0;
  Typing.extenv := M.empty;
  Emit.f outchan
    (RegAlloc.f
     (Simm.f
      (Virtual.f
       (Closure.f
        (iter !limit
         (Alpha.f
          (KNormal.f
           (Typing.f
            (Parser.exp Lexer.token 1))))))))))
```

# MinCaml での実現

```
(* main.ml *)  
let lexbuf outchan 1 =  
  Id.counter := 0;  
  Typing.extenv := M.empty;  
  Emit.f outchan  
    (RegAlloc.f  
      (Simm.f  
        (Virtual.f  
          (Closure.f  
            (iter !limit  
              (Alpha.f  
                (KNormal.f  
                  (Typing.f  
                    (Parser.exp Lexer.token 1))))))))))
```

アセンブリ生成

# 字句解析・構文解析

- 字句解析

- 文字列をトークン列に変換

- 構文解析

- トークン列を抽象構文器に変換

- MinCaml は ocamllex、ocamlyacc を利用

- 関数論理型プログラミング実験 第5回 参照

# MinCaml の型推論

- 基本的には ML の型推論

- 関数論理型プログラミング実験 第8回 参照

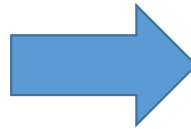
- ただし

- 型変数の実装に参照を利用
- 未定義の変数は外部変数扱い
- 型多相なし

# K正規化

- 計算途中の結果も全て変数に代入
  - アセンブリに近い
  - 評価順序を明確に

1 + 2 - 3 + 4 - 5

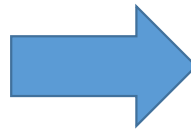


```
let a = 1 + 2 in
let b = a - 3 in
let c = b + 4 in
let d = c - 5 in
d
```

# α変換

- 全ての変数を異なる名前に
  - その後の処理（例：最適化）が楽に

```
let x = 1 in  
let x = x + 1 in  
  x
```



```
let x0 = 1 in  
let x1 = x0 + 1 in  
  x1
```

# 最適化

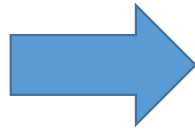
- MinCaml では以下の 3 つを繰り返す
  - インライン展開
  - 定数たたみこみ
  - 不要定義削除
- 他にも最適化の手法は様々。  
各自で色々と試みてみよう



# クロージャ変換

- 局所関数定義を大域関数定義に  
(以下はイメージ)

```
let f a =  
  let g b =  
    a + b in  
  g 3
```



```
let g a b = a + b in  
let f a = g a 3
```

(実は、上は lambda lifting)

# 仮想マシンコード生成

- よりアセンブリに近い形式に変換
  - 組やクロージャ・配列の操作を  
メモリロード・ストア命令の組み合わせに変換
  - (MinCaml における) 実際のアセンブリとの差異
    - レジスタが無限個
    - if や関数呼び出し

# レジスタ割り当て

- 仮想マシンコードのレジスタに  
実際のレジスタを割り当て
  - MinCaml はそれほど複雑なことはしていない
    - そのこのこの速さ & そこそこの易しさ

# アセンブリ生成

## ○ 実際のアセンブリを出力

- MinCaml では ここで関数呼び出しや if 式を実際の命令に変換
  - 加えて末尾呼び出しの最適化も
    - ループ最適化をしたい場合はもっと前のフェイズでやるべし

レポート課題について

# 締切の原則

- 共通課題
  - 二週間後の午後一時（JST）
- コンパイラ係向けの課題
  - 2019/2/28 23:59（JST）

# 課題の提出先

- 提出はサポートサイト経由

<http://www.kb.is.s.u-tokyo.ac.jp/~tsukada/cgi-bin/m/>

- 「コンパイラ実験2018」が見えない人は私まで

- 質問は下記まで

[compiler-jikken@kb.is.s.u-tokyo.ac.jp](mailto:compiler-jikken@kb.is.s.u-tokyo.ac.jp)

# レポート

- プログラムだけでなく  
説明・実行例・考察なども書くこと
  - ネガティブな例 (e.g. 型付け不能な例) も大切
  - 十分でなければ 0 点もありうる
- レポートはコードと独立した .txt で
  - レポートを読めば「回答者が理解している」と採点者が「信じられる」ように



# コードの提出

- 一式を tar や zip 等でまとめること
  - 展開したときに
    - <学籍番号の下6桁>  
というディレクトリが一つ生成されるように
  - 単体でビルドできるように

# 作成するコード

- レポートにはソースコードを引用すること
  - レポートの理解に必要な分だけ
- ソースコードでは オリジナルの MinCaml からの変更点が明らかになるように
  - レポート本文に明記し コードにもコメントで明示
- 関数・変数には適切な名前を
- 理解に十分なコメントをつけること

# レポート課題 1

締切：2018/10/10 13:00(JST)

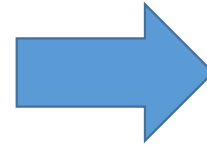
# 共通課題

3 問中 1 問以上を解答のこと

# 問 1

- MinCaml を改造し Syntax.t や Normal.t 等の中間結果を出力できるようにせよ
  - きっとデバッグに役立つ

```
let rec fib n =  
  if n <= 1 then n else  
  fib (n-1) + fib (n-2)  
in  
  print_int (fib 30)
```



```
...  
IF  
  LE  
    VAR n  
    INT 1  
  INT 1  
  ADD  
  ...
```

## 問 2

- MinCaml を改造し、  
パースエラー時や型エラー時に  
どの箇所に問題があったか示すようにせよ

# 参考：パーサが処理中の位置の取得

- パーサが処理している位置は、  
    `Parsing.symbol_start`  
    `Parsing.rhs_start`  
    などで得ることができる
- 行番号を正しく得るには  
    改行を `lexer` に教える必要がある
  - 行の終わりで `Lexing.new_line` を呼び出す
    - `min-caml` は改行を単に読み飛ばしているため、  
    行番号を正しく得るには `lexer.mll` を書き換える必要がある

# 参考

OCaml リファレンスの Parsing の項目

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/Parsing.html>

OCaml リファレンスの Lexing の項目

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/Lexing.html>

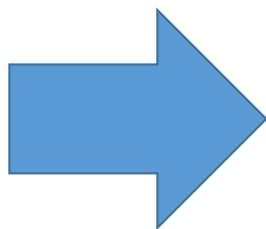


# 問 3

- MinCaml を改造し、  
出力されたアセンブリと入力コードの対応が  
分かるようにせよ
  - どの命令が 入力コードのどの式に対応しているか

# 例

```
1 let rec gcd m n =  
2   if m = 0 then n else  
3   if m <= n then  
4     gcd m (n - m)  
5   else  
6     gcd n (m - n)
```



```
.section ".text"  
gcd.7:  
    cmp        %i2, 0           ! 2  
    bne        be_else.17      ! 2  
    nop  
    mov        %i3, %i2        ! 2  
    retl  
    nop  
be_else.17:  
    cmp        %i2, %i3        ! 3  
    bg         ble_else.18      ! 3  
    nop  
    sub        %i3, %i2, %i3    ! 4  
    b          gcd.7           ! 4  
ble_else.18:  
    sub        %i2, %i3, %o5    ! 6  
    mov        %i3, %i2        ! 6  
    mov        %o5, %i3        ! 6  
    b          gcd.7           ! 6  
    nop
```

# 謝辞

本講義では、今までの講義を担当された以下の方々によって書かれた資料の多くを再利用させていただいております。ここに感謝いたします

住井英二郎先生 大山恵弘先生 前田俊行先生  
松田一孝先生 佐藤秀明氏 山下諒蔵氏 佐藤春旗氏  
小酒井隆広氏 潮田資秀氏 鈴木友博氏 渡邊祐貴氏  
秋山茂樹氏 池尻拓郎氏 野瀬貴史氏 中村晃一氏  
由水輝氏