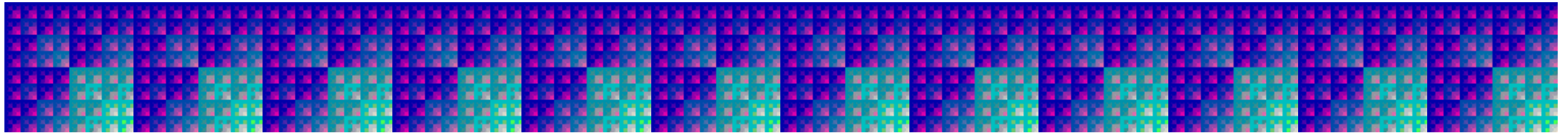


コンパイラ演習 第6回



2005/11/17

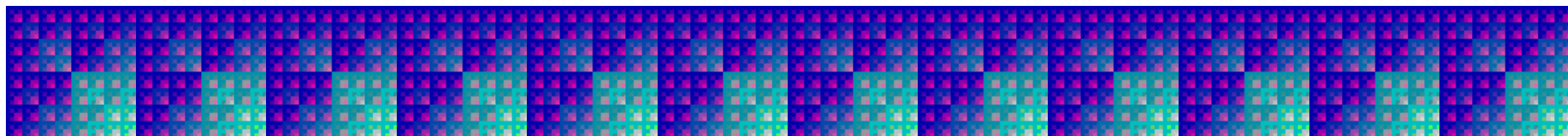
大山 恵弘

佐藤 秀明

今回の内容

- 実マシンコード生成
 - アセンブリ生成(emit.ml)
 - スタブ、ライブラリとのリンク
- 末尾呼び出し最適化
 - 関数呼び出しからの効率的なリターン(emit.ml)
 - [参考]CPS変換
- 種々の簡単な拡張
 - MinCamlにない機能

実マシンコード生成



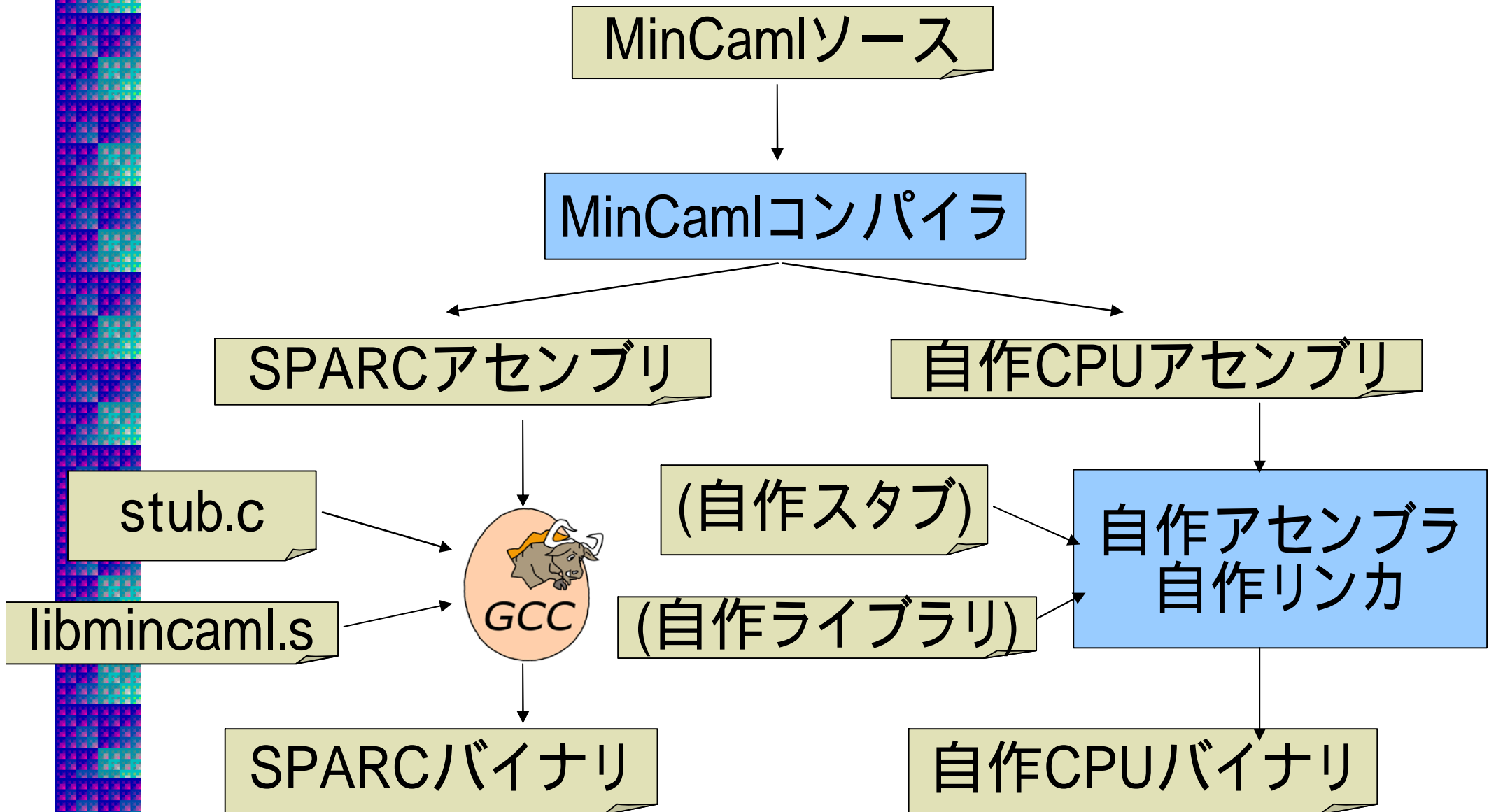
アセンブリ生成

- save/restoreをストア/ロードとして明示化
 - スタックの状態を追跡(stackmap, stackset)
 - if文の合流後は両方のスタックの積集合をとる
- 関数呼び出し規約の明示化
 - 引数を正しいレジスタにセット(shuffle関数)
 - リターンアドレスのsave/restore
 - スタックポインタの管理
- 条件分岐の明示化
 - 分岐用・合流用のラベルを導入
- 他は単純

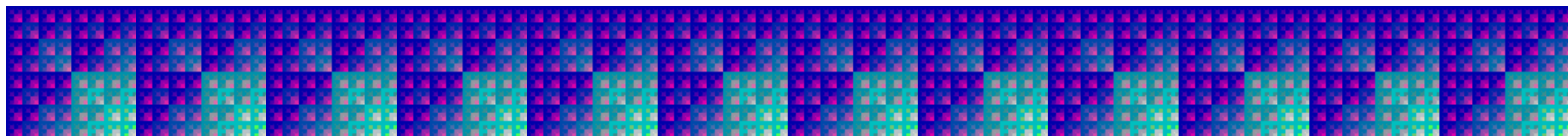
外部ファイル

- スタブ(stub.c)
 - ヒープとスタックを確保
 - その後MinCamlのエントリポイントをcall
- ライブラリ(libmincaml.s)
 - 外部関数を定義
 - 入出力
 - 配列操作
 - 数値計算
- 自作CPU向けの外部ファイルも必要かも
 - アーキテクチャ次第

ビルドの流れ



末尾呼び出し最適化



末尾呼び出し

```
let rec fact x r =  
  if x <= 1 then r  
  else fact (x - 1) (r * x)
```

- 関数の最後の処理がcall

単純にコンパイルすると...

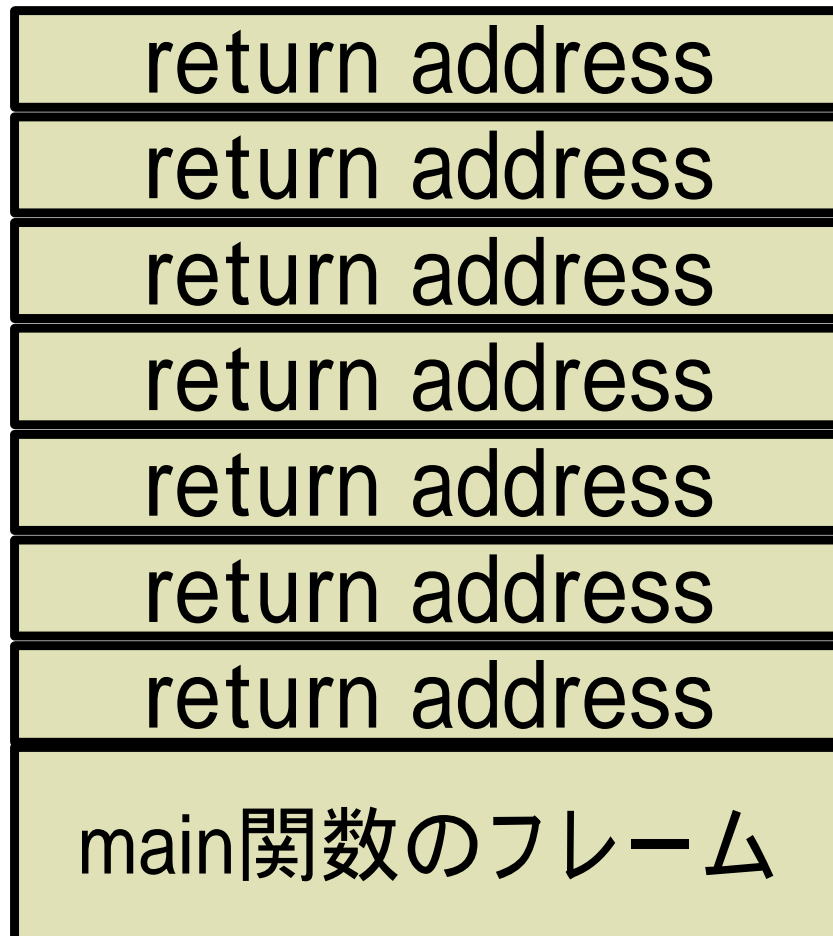
```
let rec fact x r =  
  if x <= 1 then r  
  else fact (x - 1) (r * x)
```

```
save(Rret)  
add Rsp, 4, Rsp  
call fact  
nop  
sub Rsp, 4, Rsp  
restore(Rret)  
retl  
nop
```

} 返り番地だけから
なるフレームを構成

} 返り番地をpopして
直ちにリターン

その結果

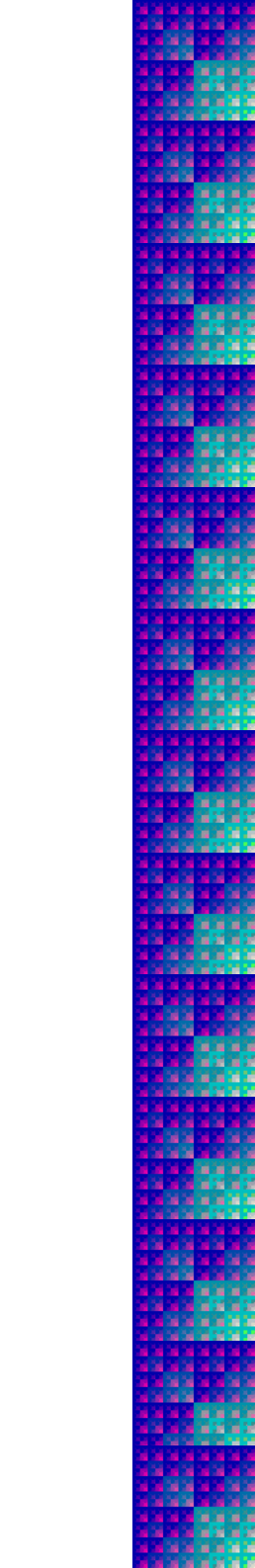


無駄なフレームが
積みあがる！



何がいけなかったのか？ どうすればよいのか？

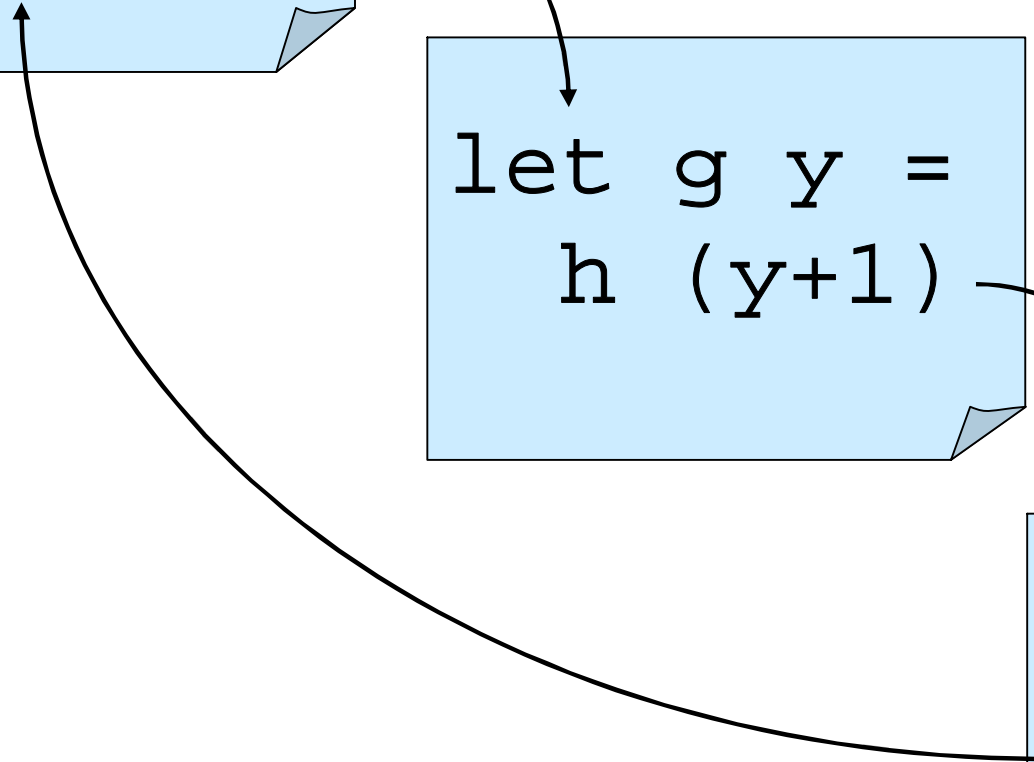
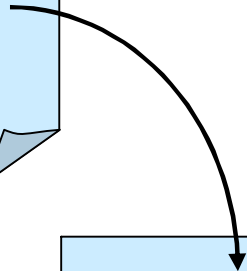
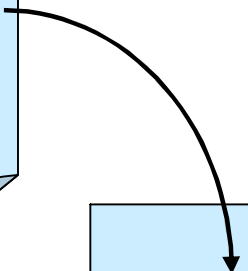
- 問題の元凶: 末尾呼び出し直後の地点に律義にリターン
 - 時間的に無駄
 - 何もしない地点へわざわざリターン
 - 空間的にも無駄
 - リターンするために、返り番地を余分に退避
- することがないなら、呼び出し元に直接返ればよい!
 - コンパイラが末尾呼び出しを認識



```
let f x =  
  ...  
  (g 8) - 3  
  ...
```

```
let g y =  
  h (y+1)
```

```
let h z =  
  z * 2
```



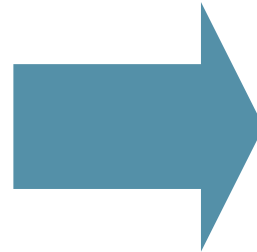
末尾呼び出し最適化

- 末尾呼び出し時の無駄なジャンプ・退避を除去
- cf. 末尾再帰
 - 関数の最後の処理が自分自身の再帰呼び出し
 - 末尾呼び出し最適化により、ループに変換

「ifの直後にreturn」する場合

- 合流する必要がない

```
cmp x, y
bg L1
nop
... (then節) ...
b L2
nop
L1: ... (else節) ...
L2: retl
nop
```

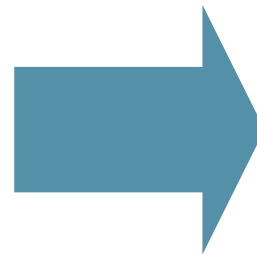


```
cmp x, y
bg L1
nop
... (then節) ...
retl
nop
L1: ... (else節) ...
retl
nop
```

「callの直後にreturn」する場合

- callをただのgotoにできる

```
save(Rret)  
add Rsp, n, Rsp  
call Lf  
nop  
sub Rsp, n, Rsp  
restore(Rret)  
retl  
nop
```



```
b Lf  
nop
```

実装

- 変換中の式が末尾かどうかを管理
 - Tail: 末尾
 - 末尾呼び出し最適化を実行、または
 - 結果を返り値用のレジスタにセットしてリターン
 - NonTail(r): 末尾でない
 - 結果をレジスタrにセット

[参考]CPS変換

- すべてのcallやifをtailにしてしまう!
 - nontailな関数適用 / 条件分岐の継続を生成
 - 「その後に行うこと」をクロージャで表現
 - すべての関数定義 / 関数適用に
仮引数 / 実引数として継続を追加
 - 関数の戻り値は継続に渡す
- 変換およびA正規化の完了したK正規形
に対して行くと簡単

CPS変換のイメージ

```
let rec f x =  
  let a = p 100 in  
  let b = q 200 in  
  x + a + b + r 300
```

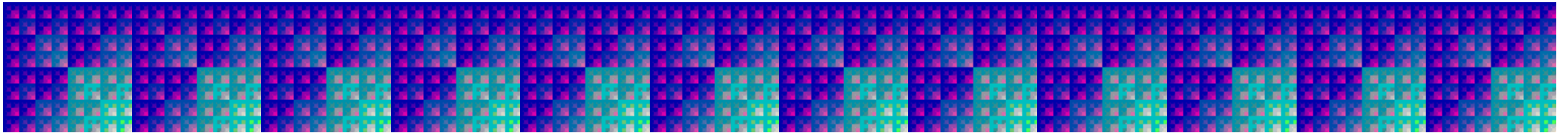
この部分を実行する関数（クロージャ）を新たに導入。pの引数として渡す

pは実行が終わったら、引数にもらった関数（クロージャ）を呼び出すことにより「復帰」する

CPS変換の利点/欠点

- 利点:以降の処理が容易
 - 関数呼び出し時のsave/restoreが不要
 - 「リターンアドレス」の概念が不要
 - スタックも不要
- 欠点:クロージャが頻繁に生成/適用される
 - 効率的なヒープ管理の必要性
 - inter-proceduralなレジスタ割り当て
 - エスケープ解析
 - generational garbage collection

種々の簡単な拡張



レコード、Variant

- レコード: フィールドがアルファベット順に並んだtupleとみなす

$\{ \text{foo} = 3; \text{bar} = 7 \}$
 $= \{ \text{bar} = 7; \text{foo} = 3 \}$
 $(7, 3)$

- Variant: コンストラクタを整数で表し、それを第1要素とするtupleにする

type α list = Nil | Cons of $\alpha * \alpha$ list として

Nil (0)
Cons(x, y) (1, x, y)

抽象、部分適用

- 抽象: let recに置換

$\text{fun } x \quad M \quad \text{let rec } f \ x = M \text{ in } f$

(fはfreshな変数名)

- 部分適用: let recと完全適用に置換
たとえば $\text{let rec } f \ x \ y = x - y$ なら

$f \ 3 \quad \text{let rec } g \ y = f \ 3 \ y \text{ in } g$

(gはfreshな変数名)

- 関数の型情報が必要

共通課題(1/3)

- 以下のプログラムはどのようなアセンブリにコンパイルされるか、末尾呼び出し最適化をしない場合とする場合、それぞれについて説明せよ。
 - ヒント: 末尾呼び出し最適化をする場合、手続き型言語でループを用いて書いたgcdと同じアセンブリになる(はず)

```
let rec gcd m n =  
  if m <= 0 then n else  
  if m <= n then gcd m (n - m) else  
  gcd n (m - n) in  
print_int (gcd 21600 337500)
```

共通課題(2/3)

- 以下のプログラムを手動でCPS変換せよ。
 - K正規化はしてもしなくてもよい
 - 余裕があれば、CPS変換した場合としなかった場合でどのようなアセンブリにコンパイルされるか、両者を比較してみよう

```
let rec ack x y =  
  if x <= 0 then y - -1 else  
  if y <= 0 then ack (x - 1) 1 else  
  ack (x - 1) (ack x (y - 1)) in  
print_int (ack 3 10)
```


共通課題(3/3)

- 「種々の簡単な拡張」(の一部)を用いるMLプログラムを書け。それを既存のMLコンパイラがどうコンパイルするか調べ、解説せよ。
 - 同程度以上に複雑な他のプリミティブについて調べてもよい

課題の提出先と締め切り

- 提出先: `compiler-enshu@yl.is.s.u-tokyo.ac.jp`
- 締め切り: 2週間後 (12/1) の午後1時
- Subject: report 6 <学籍番号> <アカウント>
- 本文にも氏名と学籍番号を明記のこと

課題の提出についての注意

- プログラムだけでなく、説明・考察・感想なども書くこと
- 基本的にはメールの本文に解答を記述
- 多くのソースを送る必要がある課題では、ソースをtarファイルなどに固めてメールに添付のこと

今後の予定

- 次回からは応用編です

11/24: 休講

12/1: 休講

12/8: Garbage Collection

12/15: オブジェクト

12/22: Polymorphism

1/12: 例外処理

1/19: エスケープ解析

1/26: リージョン推論?