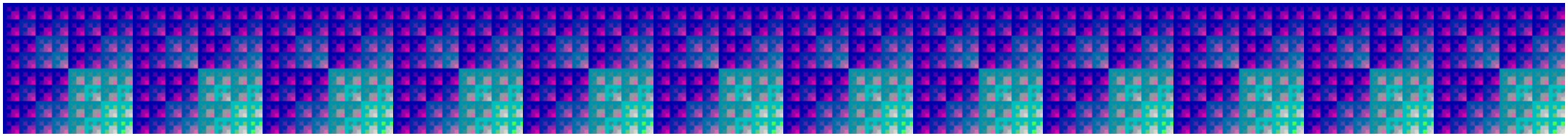
コンパイラ演習 第5回



2005/11/10

大山 恵弘　　佐藤 秀明

今回の内容

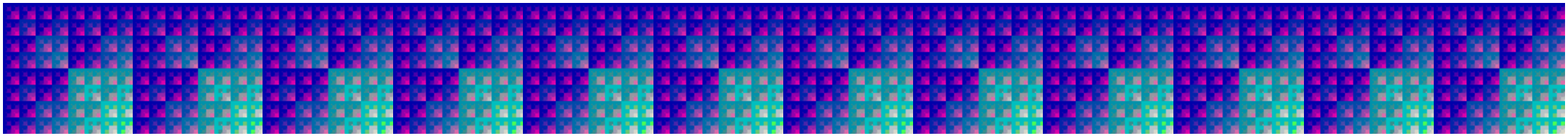
- レジスタ割り当て
  - Interference
  - Coalescing (レジスタ合わせ)
- 生きている変数のsave/restore
  - レジスタ溢れ(spilling)
  - 関数呼び出し
  - 条件分岐

相互に依存していて厄介！

講義方針

- そこそこの速さ&そこそこの易しさ
  - MinCaml方式
    - regAlloc.target-latespill.mlをもとに説明
- 各自アルゴリズムを工夫してみてください

レジスタ割り当て



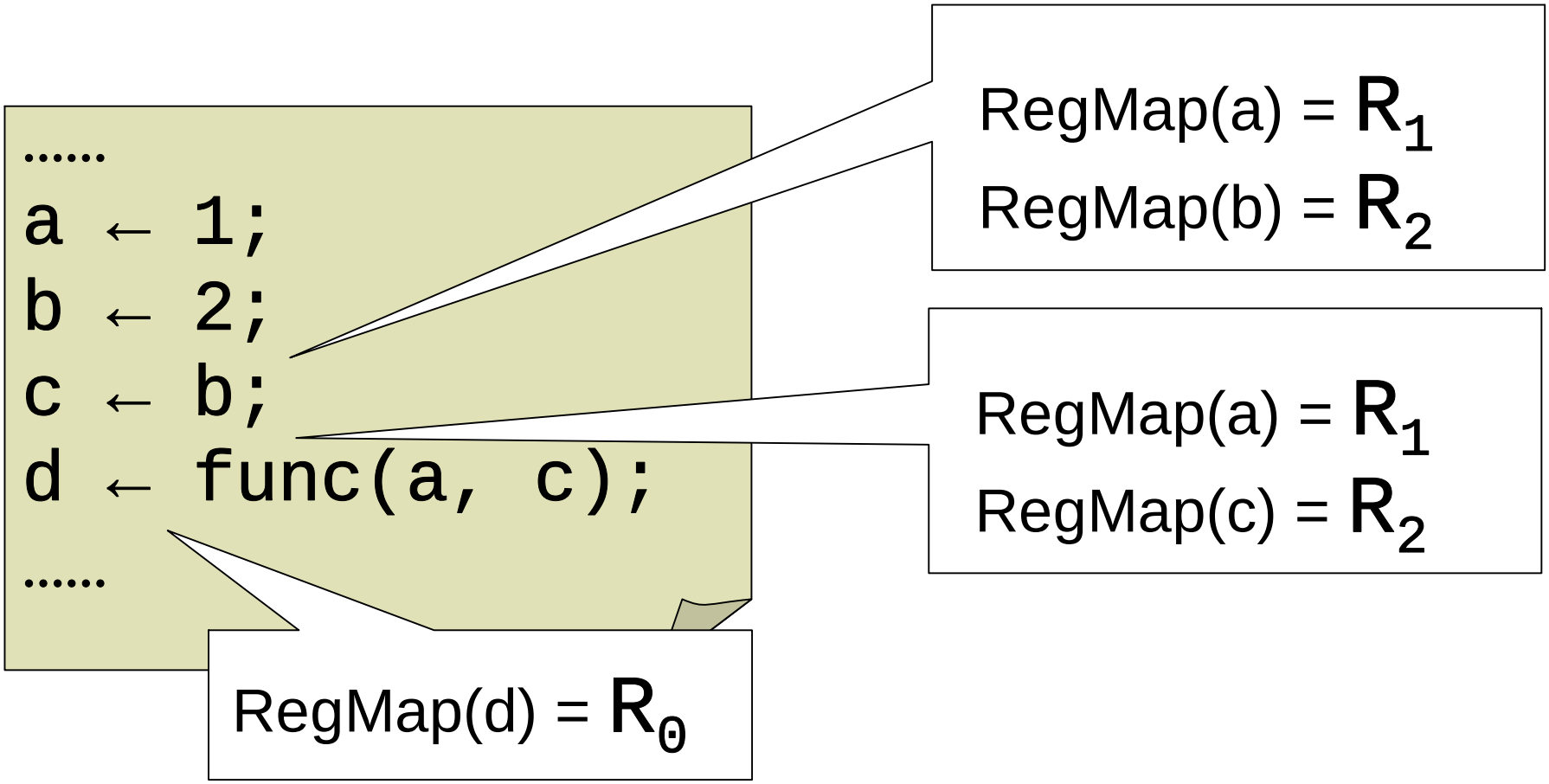
方針

1. Interference
   - 生きている変数を上書きしない
2. Coalescing(targeting)
   - 無駄なmovはできるだけ減らす

レジスタ割り当ての表現構造

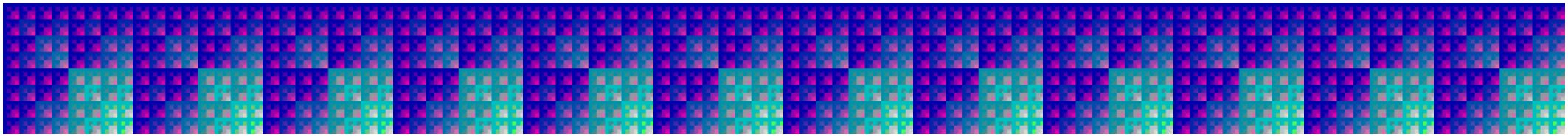
- 変数からレジスタへの写像として実装
  - regenv(x) = R_n
- 処理の進行とともに写像を逐次更新

レジスタ割り当て写像の操作



生きている変数の

save/restore



save命令、restore命令

- save(x): xの値をスタックに退避
  - xが他の値で上書きされるおそれのある時
- restore(x): xの値をスタックから復帰
  - いま使いたい変数がレジスタにない時

命令挿入の方針

- レジスタ溢れ
  - レジスタが足りないときに適当な変数をsave
- 関数呼び出し
  - 関数を呼び出す前に、生きている変数をsave
- 条件分岐
  - 各分岐先で異なるレジスタに割り当てられた変数は、合流後は改めてrestoreしてから使用

レジスタ溢れの例(1/2)

(汎用レジスタはR_0 、R_1、R_2の3つだけとする)

```
let rec f a b =
  let x = –a in let y = –b in x – y – a – b
```

↓

```
let rec f R1 R2 =
  let x = -R1 in let y = -R2 in x - y - R1 – R2
```

↓

```
let rec f R1 R2 =
  let R0 = -R1 in let y = -R2 in R0 - y - R1 –
   R2
```

yに割り当てるレジスタは?

レジスタ溢れの例(2/2)

- 変数bを一時的にスタックへ退避

```
let rec f a b =
      let x = –a in let y = –b in x – y – a – b
                              ↓
let rec f a b =
      let x = –a in save(b); let y = –b in
      x – y – a – (restore(b); b)
```

関数呼び出しの例

```
let x = ... in
  let y = ... in
  let z = f x y in
  if z ≤ 0 then x − 1 else y − 2
```

↓

```
let x = ... in
  let y = ... in
  save(x); save(y); let z = f x y in
  if z ≤ 0  then restore(x); x − 1
   else restore(y); y − 2
```

条件分岐の例(1/2)

```
let a = if f () then x - y else y − x in a + x + y
```

↓

```
let a = save(x); save(y); if f ()
  then (restore(x); x) - (restore(y); y)
    (* regenv = { x→R1, y→R2 } *)
  else (restore(y); y) - (restore(x); x)
    (* regenv = { x→R2, y→R1 } *)
in a + x + y
```

合流後のレジスタ割り当ては?

条件分岐の例(2/2)

- 合流後にx、yを改めてrestore

```
let a = if f () then x - y else y − x in a + x + y
```

↓

```
let a = save(x); save(y); if f ()
  then (restore(x); x) - (restore(y); y)
    (* regenv = { x→R1, y→R2 } *)
  else (restore(y); y) - (restore(x); x)
    (* regenv = { x→R2, y→R1 } *)
in a + (restore(x); x) + (restore(y); y)
```

regAlloc.target-earlyspill.ml(1/3)

- regAlloc.mlと同様
- どうせ後でsaveする変数は定義直後にsave
  - ややこしいので参考程度に

regAlloc.target-earlyspill.ml(2/3)

- 変数xの退避が必要になったら、
  1. 現在の位置にForget命令を挿入
     - xのレジスタ割り当てを削除
  2. 式をToSpillコンストラクタに入れて返す
- 退避の必要な変数がなかったら、
  1. 式をレジスタ割り当て
  2. NoSpillコンストラクタに入れて返す

regAlloc.target-earlyspill.ml(3/3)

- ToSpillコンストラクタを受け取ったら、
  1. 退避する変数xの定義までさかのぼる
  2. 定義の直後にsave(x)を挿入
  3. 式のレジスタ割り当てをやり直す
- NoSpillコンストラクタを受け取ったら、
  - 式をそのまま返す

spillが多すぎるときは...

- ヒープポインタを（専用レジスタではなく）汎用レジスタにとる
  - 関数の引数や返値として付け加える
    - (x, h) ← CallCls(y, h, z_1, ..., z_n)
    - (x, h) ← CallDir(L_f, h, z_1, ..., z_n)
    - return(x, h)
- ヒープポインタをメモリ（固定）におく
  - MakeCls等が稀ならば得
- リターンアドレスを（専用レジスタではなく）汎用レジスタにとる
  - 関数からのreturnに「戻り先」として付け加える
    - return x to r

共通課題(1/2)

- 例にならい、次式へsave/restoreを挿入してみよ。どのように入れるのが「より良い」だろうか。

```
let x = ... in
let y = (if x ≤ 0 then f 1 else 2) in
let z = (if y ≤ 3 then x − 4 else g 5) in
x − y − z
```

共通課題(2/2)

- 適度に簡単な関数（フィボナッチ数列、アッカーマン関数、最大公約数など）に対し、例にならって手でレジスタ割り当てを行ってみよ。
  - 良いレジスタ割り当てをした結果と悪いレジスタ割り当てをした結果の両方を提出
    - レジスタ移動回数に差を作る
    - 少ないレジスタ数を仮定し、spill回数に差を作る
    - etc

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 (11/24) の午後1時
- Subject: report 5 <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