コンパイラ

湯淺太一

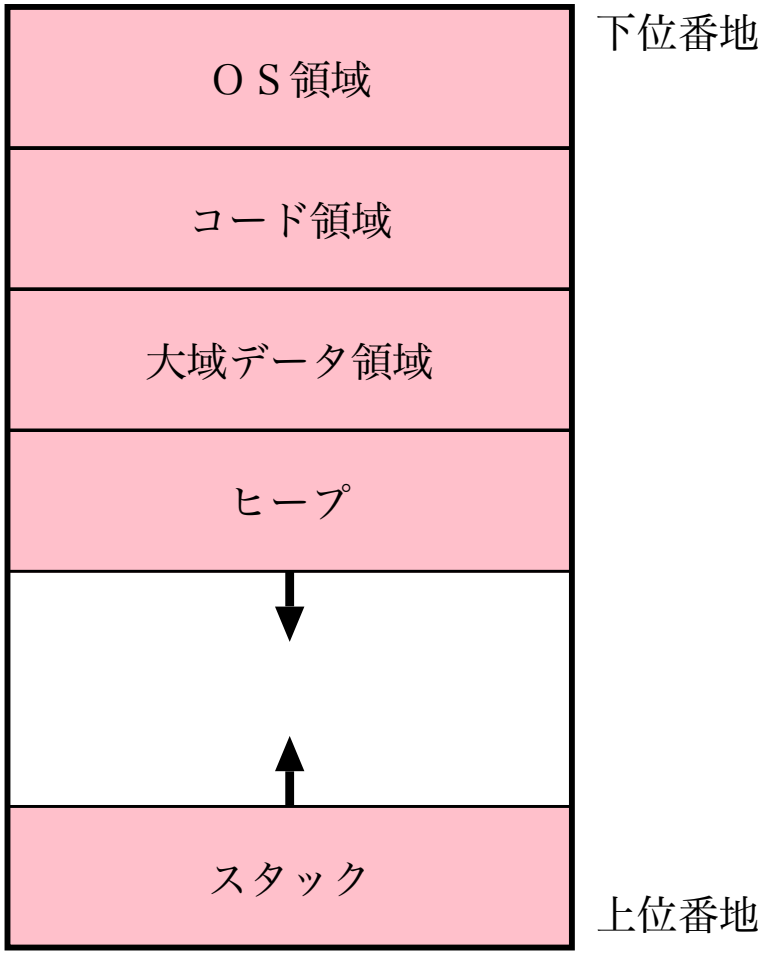
第６章

コ ー ド 生 成

実行環境のモデル

メモリ領域

- OS領域（場所・サイズは固定）
- ユーザ領域
  - コード領域（ロード時に確定）
  - データ領域
    - 大域データ領域（ロード時に確定）
    - スタック（実行時に拡大・縮小）
    - ヒープ（実行時に拡大）



CPUとレジスタ

中央処理装置 (Central Processing Unit, CPU):
　コード領域から機械語命令を一つずつ取り出し実行

算術論理装置 (Arithmetic Logic Unit, ALU):
　算術演算や比較演算を実行

Pentium のレジスタ

1. **汎用レジスタ** (general-purpose register):
   オペランドとして使用できる32ビット長レジスタ
   ベースポインタ`ebp`，スタックポインタ`esp`，`eax`，`ebx`，`ecx`，`edx`など
2. **条件フラグ** (condition flag):
   比較命令の結果を格納する1ビット長レジスタ
   ゼロフラグ`zf`，符号フラグ`sf`など
3. **命令ポインタ** (プログラムカウンタ，instruction pointer):
   次の命令の場所を指す32ビット長レジスタ`eip`
   CPUは次の動作を繰り返す:
   1. `eip`の指す位置から命令を一つ取り出す
   2. 次の命令を指すように`eip`を更新する
   3. 取り出した命令を実行する

アセンブリ命令

ラベル宣言

〈ラベル〉:

アセンブリ命令

〈命令名〉〈オペランド$_1$〉，. . . ，〈オペランド$_n$〉

まとめて

〈ラベル〉:〈命令名〉〈オペランド$_1$〉，. . .

オペランド:

- 汎用レジスタ
- メモリ番地
  - ラベル
  - 相対番地: $n[R]$
- 整数定数

算術命令:

```
add   eax,ebx       ; eaxにebxの値を足す
sub   esp,4         ; espから4を引く
imul  ebx,_x        ; ebxにxの値を掛ける
```

移動命令:

```
mov   eax,4         ; eaxに整数4を格納する
mov   ebp,esp       ; espの値をebpに格納する
```

無条件ジャンプ命令:

```
jmp   ラベル
```

比較命令:

```
cmp   x,y
```

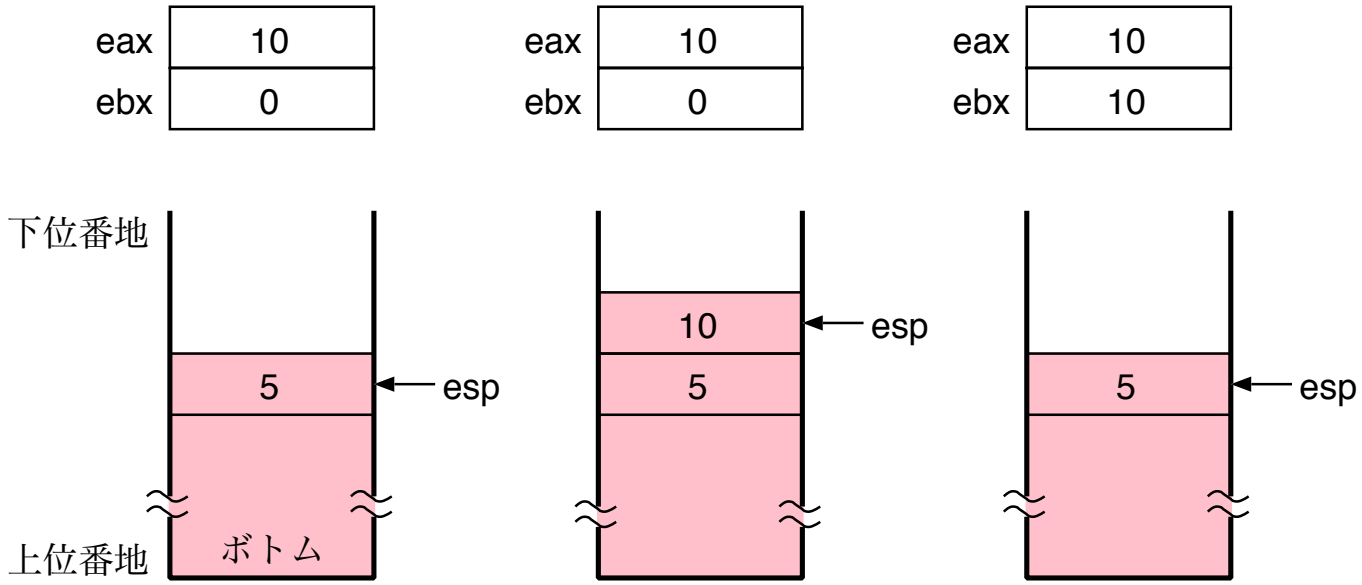
条件付きジャンプ命令:

```
j…    ラベル
```

命令	条件	意味
jg	$x > y$	Jump if Greater
jge	$x \geq y$	Jump if Greater or Equal
je	$x = y$	Jump if Equal
jne	$x \neq y$	Jump if Not Equal
jl	$x < y$	Jump if Less
jle	$x \leq y$	Jump if Less or Equal

スタック操作命令:

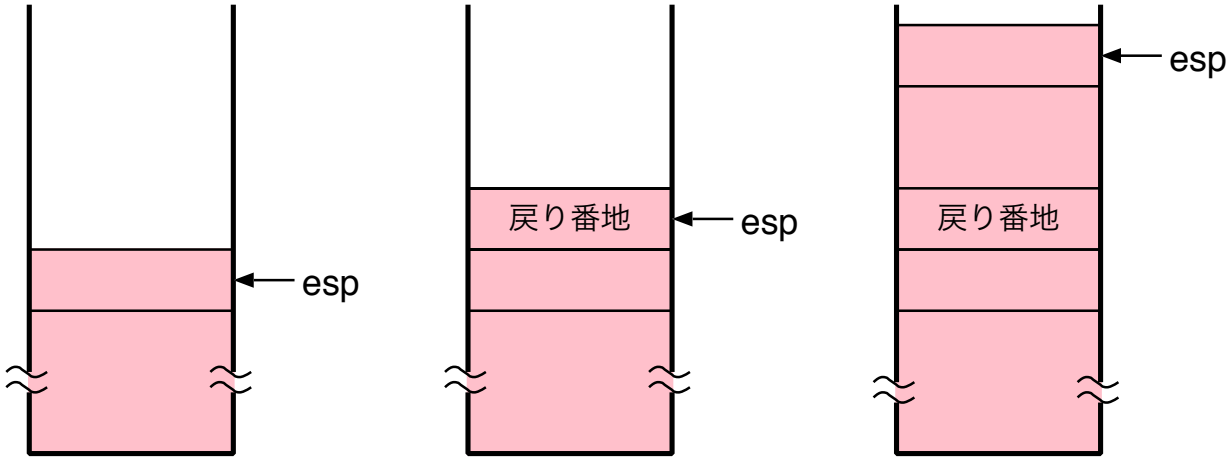
```
push eax
pop  ebx
```



(a) 実行前　(b) push eax 実行後　(c) pop ebx 実行後

関数呼出し命令: `call` ラベル

リターン命令: `ret`



(a) 呼出し直前
(e)リターン直後

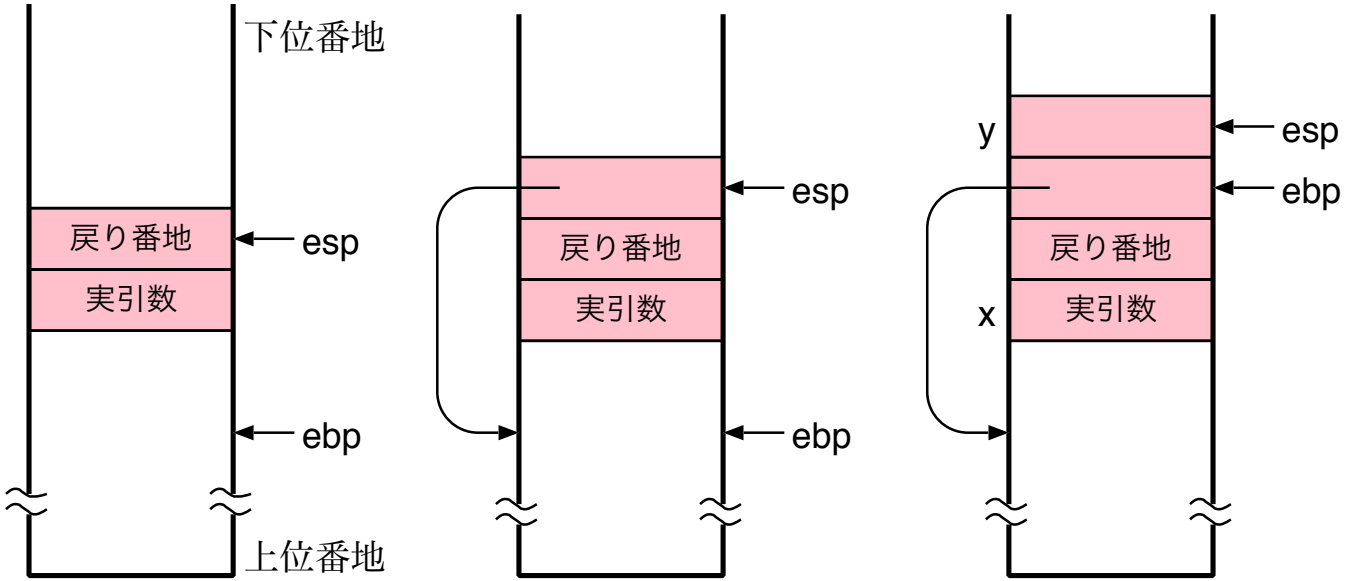
(b) 呼出し直後
(d) リターン直前

(c) 関数実行中

関数呼出し

```
int foo(int x) {
    int y = x*x;
    return y+2;
}
```

```
 1  _foo: push  ebp
 2        mov   ebp,esp
 3        sub   esp,4
...       ...
 9        mov   esp,ebp
10        pop   ebp
11        ret
```



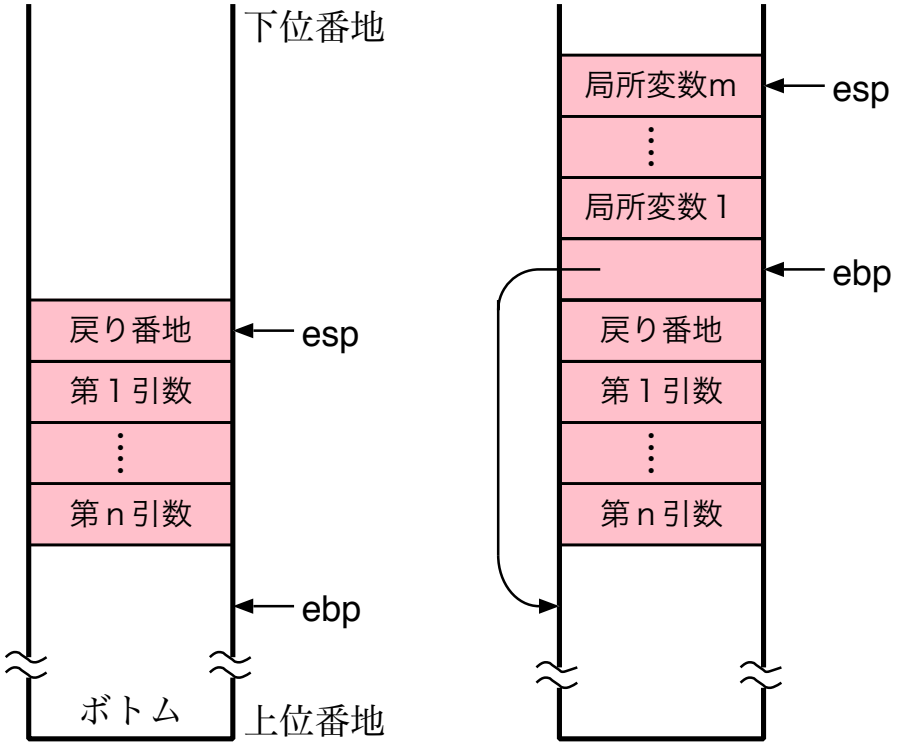
(a) 呼出し直後　(b) ebp の値の保存　(c) 本体実行中

fooの呼出し

```
push  5
call  _foo
add   esp,4
```

一般の関数 f のコード

```
_f:   push ebp
      mov  ebp,esp
      sub  esp, Nlocal
      本体の実行
Lret: mov  esp,ebp
      pop  ebp
      ret
```



(a) 呼出し直後　　(b) 本体実行中

関数呼出し式 $f(e_1, e_2, \ldots, e_n)$ のコード

e_nを計算し，結果をプッシュする

...

e_2を計算し，結果をプッシュする

e_1を計算し，結果をプッシュする

```
call _f
add  esp,n × 4
```

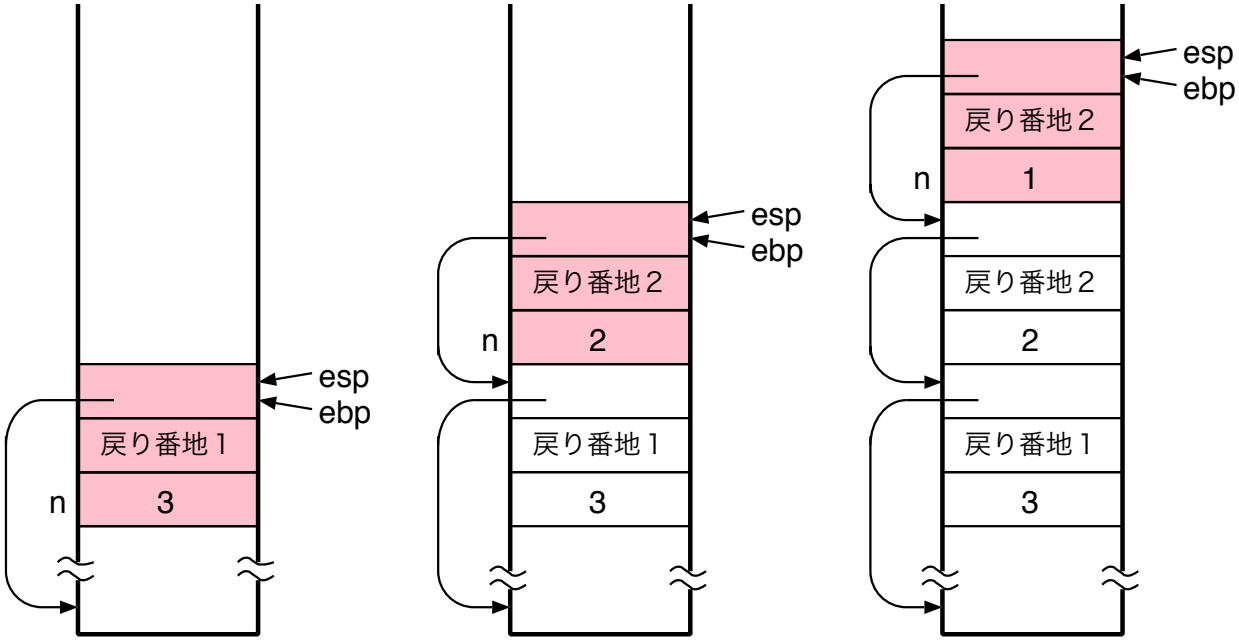
例: f(1,g(2,3),4)のコード

```
push 4              ; fへの第3引数
push 3              ; gへの第2引数
push 2              ; gへの第1引数
call _g
add  esp,8
push eax            ; fへの第2引数
push 1              ; fへの第1引数
call _f
add  esp,12
```

例: 階乗を求める関数factのコードとその実行

```
int fact(int n) {
    if (n == 1) return 1;
    else return n * fact(n-1);
}
```

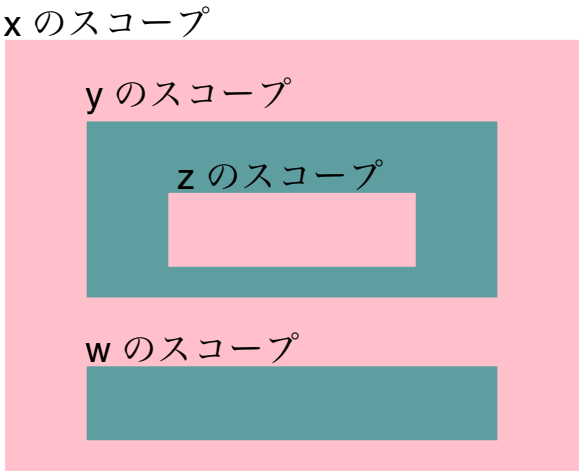
```
_fact: push ebp
       mov  ebp,esp
       cmp  8[ebp],1      ; n = 1かどうか
       jne  L2            ; n ≠ 1なら L2へ
       mov  eax,1         ; n = 1なら戻り値は1
       jmp  L1
  L2:  mov  eax,8[ebp]    ; n − 1の計算
       sub  eax,1
       push eax
       call _fact         ; 再帰呼出しで(n − 1)!の計算
       add  esp,4
       imul eax,8[ebp]    ; n × (n − 1)!の計算
  L1:  pop  ebp
       ret
```



(a) fact(3) 実行中　(b) fact(2) 実行中　(c) fact(1) 実行中

局所変数等の割当て

1. 局所変数領域のサイズ $Nlocal$ をできるだけ小さく
   → 同じ位置に複数の局所変数を割り当てる
2. スコープが重なる局所変数は，同じ位置に割り当てない



順序	動作	相対番地	last_alloc	max_alloc
0	初期化		0	0
1	x の割当て	−4	1	1
2	y の割当て	−8	2	2
3	z の割当て	−12	3	3
4	z の解放		2	3
5	y の解放		1	3
6	w の割当て	−8	2	3
7	w の解放		1	3
8	x の解放		0	3

$Nlocal = 3 \times 4 = 12$ バイト使用

レジスタの退避

Pentiumの汎用レジスタは，
espとebpの他に6個だけ

1. 呼出し後保存レジスタ (callee-saved register)
   呼び出す側が使用中かもしれない汎用レジスタ.
   呼び出される関数は，使用前に退避
2. 呼出し前保存レジスタ (caller-saved register)
   上記以外の汎用レジスタ.
   呼び出される関数は，退避せずに使用してよい
   別の関数を呼び出すときは，退避しておく

少なくともeaxレジスタは呼出し前保存
→ 全レジスタが呼出し前保存と仮定

文のコード生成

複合文のコード

$\{d_1 \ \cdots \ d_n \ \ s_1 \ \cdots \ s_m\}$

d_1の初期化
...
d_nの初期化
s_1の実行
...
s_mの実行

宣言 `int x = 10;` の初期化

`mov` n`[ebp],10`

if文のコード

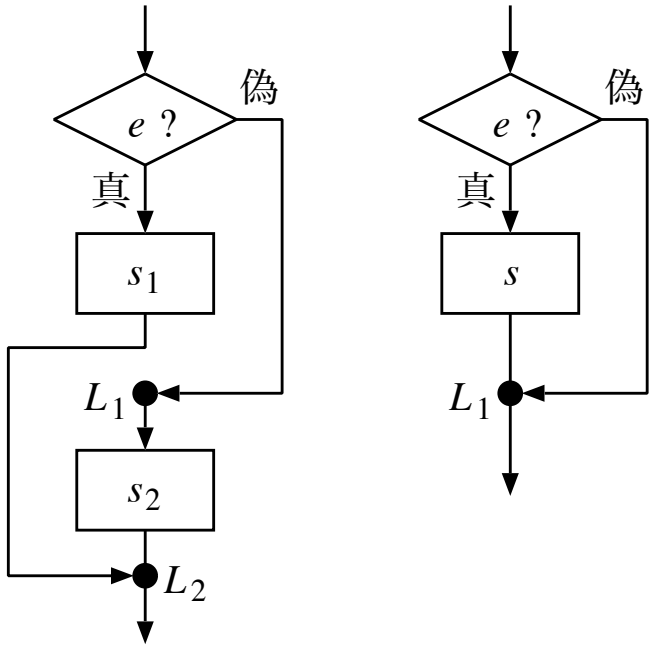
`if` (*e*) s_1 `else` s_2

*e*を計算し，結果が偽ならばL_1へジャンプ
s_1の実行

`jmp` L_2

L_1: s_2の実行

L_2:



`if` (*e*) *s*

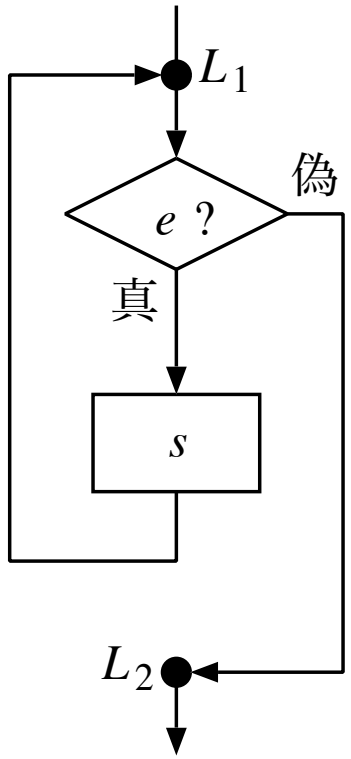
*e*を計算し，結果が偽ならばL_1へジャンプ
*s*の実行

L_1:

while文のコード

`while` (*e*) *s*

L_1: *e*を計算し，結果が偽ならばL_2へジャンプ
*s*の実行
`jmp` L_1
L_2:



break文のコード

`jmp` L_2

continue文のコード

`jmp` L_1

式文と代入式のコード

式文 e;

eの計算（結果は破棄）

変数vへの代入式v `=` e'

e'の計算（Rに結果）

```
mov   loc(v),R
```

例: `x = y = z+1;`

```
mov   R,loc(z)
add   R,1
mov   loc(y),R
mov   loc(x),R
```

ラベルとジャンプ命令の抑制

```
if (e1) {
  if (e2) s1 else s2
}
```

	e_1を計算し，結果が偽ならL_1へジャンプ
	e_2を計算し，結果が偽ならL_2へジャンプ
	s_1の実行
	jmp L_3
L_2:	s_2の実行
L_3:	
L_1:	

L_1とL_3は融合可能

抑制の方法

1. まったく参照されないラベルを抑制する
2. 直後に来るラベル（もしあれば）を再利用する
3. 直後が無条件ジャンプなら，そのラベルへジャンプする

算術式のコード生成

演算式$e_1 \circ e_2$:

1. e_1の値をあるレジスタRにロードし，
2. Rを第1オペランド，e_2の値を第2オペランドとして，
   ‘$\circ$’に対応する命令$inst$を実行する．

例: `x+y`

```
mov   R,loc(x)
add   R,loc(y)
```

例: `a*b+y`

```
mov   R,loc(a)
imul  R,loc(b)
add   R,loc(y)
```

例: `a*b+x*y`

```
mov   R1,loc(a)
imul  R1,loc(b)
mov   R2,loc(x)
imul  R2,loc(y)
add   R1,R2
```

例: R_1だけを使って`a*b+x*y`を計算

```
mov  R1,loc(a)
imul R1,loc(b)
mov  temp,R1
mov  R1,loc(x)
imul R1,loc(y)
add  R1,temp
```

可換演算と非可換演算

例: R_1だけを使って`a*b-x*y`を計算

```
mov  R1,loc(a)
imul R1,loc(b)
mov  temp,R1
mov  R1,loc(x)
imul R1,loc(y)
sub  R1,temp
```

ではなく

```
mov  R1,loc(x)
imul R1,loc(y)
mov  temp,R1
mov  R1,loc(a)
imul R1,loc(b)
sub  R1,temp
```

利用可能なレジスタ数が N,
e_1 と e_2 の計算に必要なレジスタ数も N のとき,
e_1 - e_2 は

e_1 の計算（R_1 に結果）

```
mov   temp,R1
```

e_2 の計算（R_2 に結果）

```
mov   R3,temp
sub   R3,R2
```

よりも

e_2 の計算（R_2 に結果）

```
mov   temp,R2
```

e_1 の計算（R_1 に結果）

```
sub   R1,temp
```

右辺の計算を先に！

算術式のコード生成アルゴリズム

N: 式の計算に利用できるレジスタ数（$N \geq 2$）

$\rho(e)$: eの計算に必要なレジスタ数

$N \geq \rho(e) \geq 0$
$\rho(e) = 0 \leftrightarrow e$は変数か定数

アルゴリズム:

算術式$e_1 \circ e_2$に対して,
まず$\rho(e_1)$と$\rho(e_2)$を求め
各e_iのコード生成開始時点で,
$\rho(e_i)$個以上のレジスタを空ける

RSL (Right-Save-Left) 型

$\rho(e_1) = \rho(e_2) = N$のとき

e_2の計算（R_2に結果）

`mov` $temp, R_2$

e_1の計算（R_1に結果）

$inst \quad R_1, temp$

RL (Right-Left) 型

$\rho(e_1) < N$のとき

e_2の計算（R_2に結果）
e_1の計算（R_1に結果）

$inst \quad R_1, R_2$

R (Right) 型

e_1が変数/定数vで，演算が可換のとき

e_2の計算（R_2に結果）

$inst \quad R_2, \mathrm{loc}(v)$

LR (Left-Right) 型

$\rho(e_2) < N$のとき

e_1の計算（R_1に結果）
e_2の計算（R_2に結果）

$inst \quad R_1, R_2$

L (Left) 型

e_2が変数/定数vのとき

e_1の計算（R_1に結果）

$inst \quad R_1, \mathrm{loc}(\mathrm{v})$

	$\rho(e_2) = N$	その他	$\rho(e_2) = 0$
$\rho(e_1) = N$	RSL	LR	L
$N > \rho(e_1) > 0$	RL	RL/LR	L
$\rho(e_1) = 0$（非可換）	RL	RL	L
$\rho(e_1) = 0$（可換）	R	R	L

RSL型コード生成ルーチン

```
reg emit_RSL_code(char *inst, tree e1, tree e2) {
    reg R1, R2 = emit_expr(e2);
    loc temp = allocate_temp();
    emit("mov", temp, R2);
    release_register(R2);
    R1 = emit_expr(e1);
    emit(inst, R1, temp);
    release_temp(temp);
    return R1;
}
```

RL型コード生成ルーチン

```
reg emit_RL_code(char *inst, tree e1, tree e2) {
    reg R2 = emit_expr(e2);
    reg R1 = emit_expr(e1);
    emit(inst, R1, R2);
    release_register(R2);
    return R1;
}
```

使用レジスタ数の計算

算術式 $e_1 \circ e_2$ の使用レジスタ数

$\rho(e_2)$の値	N	その他	0
$\rho(e_1) = N$	N	N	N
その他	N	RL型: $\max(\rho(e_1)+1, \rho(e_2))$ LR型: $\max(\rho(e_1), \rho(e_2)+1)$	$\rho(e_1)$
$\rho(e_1) = 0$（非可換）	N	$\max(2, \rho(e_2))$	1
$\rho(e_1) = 0$（可換）	N	$\rho(e_2)$	1

$\rho(e_1) = N$ または $\rho(e_2) = N$ なら $\rho(e_1 \circ e_2) = N$

RL型なら $\rho(e_1 \circ e_2) = \max(\rho(e_1)+1, \rho(e_2))$

e_2の計算（R_2に結果）	$\rho(e_2)$個使用
e_1の計算（R_1に結果）	$\rho(e_1)+1$個使用
$inst \quad R_1, R_2$	

例: `a*b-x*y`

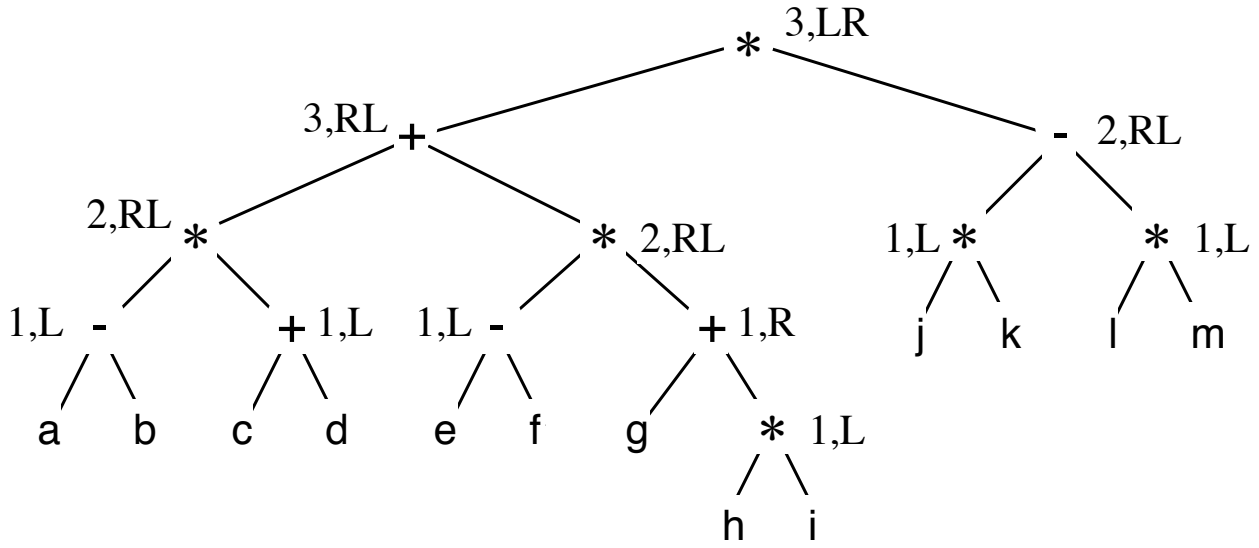
$\rho($`a`$) = \rho($`b`$) = 0$ より $\rho($`a*b`$) = 1$，同様に $\rho($`x*y`$) = 1$

したがって，`a*b-x*y`はRL型

```
mov   R,loc(x)
imul  R,loc(y)
mov   R',loc(a)
imul  R',loc(b)
sub   R',R
```

例: $N = 3$のとき,

```
((a-b)*(c+d)+(e-f)*(g+h*i))*(j*k-l*m)
```




```
mov  eax,h      mov  eax,c      mov  eax,l
imul eax,i      add  eax,d      imul eax,m
add  eax,g      mov  ecx,a      mov  ebx,j
mov  ebx,e      sub  ecx,b      imul ebx,k
sub  ebx,f      imul ecx,eax    sub  ebx,eax
imul ebx,eax    add  ecx,ebx    imul ecx,ebx
```

関数呼出し式の使用レジスタ数

f(`(`e_1`,` $\ldots$`,` e_n`)`)

各e_iの値をスタックにプッシュ

e_iの計算（Rに結果）

`push` R

$\rho(e_i)$個のレジスタを使用

すべての実引数をスタックにプッシュ

使用レジスタ数は$\max(m, \rho(e_1), \ldots, \rho(e_n))$

fの使用レジスタ数は，分からない．$\rightarrow$最大の N 個と仮定

$$\rho(f(e_1, \ldots, e_n)) = \max(N, \rho(e_1), \ldots, \rho(e_n)) = N$$

例: `f()-x*y`

ρ(`f()`) $= N$，ρ(`x*y`) $= 1$だからLR型

```
call _f
mov  R,loc(x)
imul R,loc(y)
sub  eax,R
```

条件ジャンプのコード生成

「eを計算し，結果が偽ならLへジャンプ」

```
eの計算（Rに結果）
cmp   R,0          ; 0と比較
je    L            ; 等しければジャンプ
```

「真なら」の場合は，`je`を`jne`で置き換える．

例: `if (f()) x = 10;`のコード

```
      call _f
      cmp  eax,0
      je   L
      mov  loc(x),10
L:
```

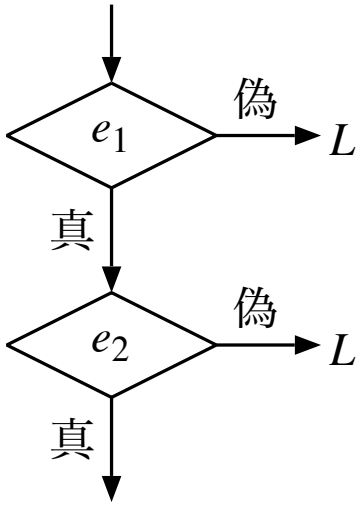
「e_1 >= e_2が真ならLへ」（e_1 >= e_2がRL型のとき）

```
e2の計算（R2に結果）
e1の計算（R1に結果）
cmp   R1,R2
jge   L
```

「e_1 && e_2が偽ならLへ」

e_1が偽ならLへ

e_2が偽ならLへ

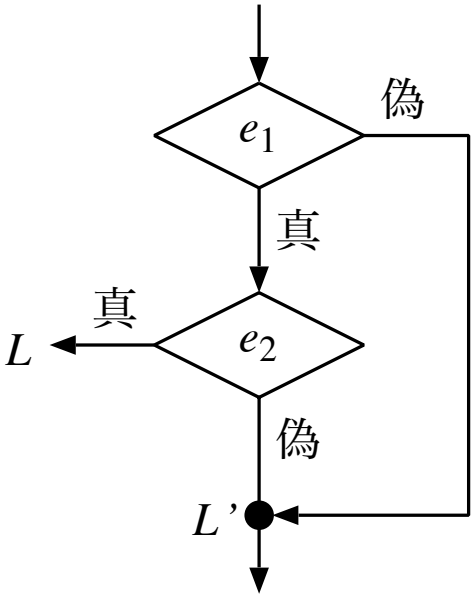


「e_1 && e_2が真ならLへ」

e_1が偽ならL'へ

e_2が真ならLへ

L':



「e_1 || e_2が真ならLへ」

e_1が真ならLへ

e_2が真ならLへ

「e_1 || e_2が偽ならLへ」

e_1が真ならL'へ

e_2が偽ならLへ

L':

「!eが真ならLへ」

eが偽ならLへ

「!eが偽ならLへ」

eが真ならLへ

例: 「(a && b) || !(c || d)が真ならLへ」

(a && b) || !(c || d)が真ならLへ

a && bが真ならLへ

aが偽ならL1へ

```
cmp loc(a),0
je L1
```

bが真ならLへ

```
cmp loc(b),0
jne L
```

L1:

!(c || d)が真ならLへ

c || dが偽ならLへ

cが真ならL2へ

```
cmp loc(c),0
jne L2
```

dが偽ならLへ

```
cmp loc(d),0
je L
```

L2:

戻り値の計算コード

`return` e`;`

eの計算（Rに結果）

`mov` `eax,`R

`jmp` $Lret$

モード指定で移動命令を削除

eaxモード: 結果をできるだけ`eax`に

no-eaxモード: 結果をできるだけ`eax`以外に

freeモード: どのレジスタでもよい

例: `return a*b-x*y;`のコード

`a*b-x*y`全体はeaxモード，`a*b-x*y`はRL型

`x*y`の計算（R_2に結果）	no-eaxモード
`a*b`の計算（R_1に結果）	eaxモード

$inst$ R_1`,`R_2

生成結果（移動命令削除）

```
mov   ebx,loc(x)
imul  ebx,loc(y)
mov   eax,loc(a)
imul  eax,loc(b)
sub   eax,ebx
jmp   Lret
```

関数コードの生成例

```
_fact: push ebp
       mov  ebp,esp
       本体 { if ··· }の実行
         if文if(n==1) ··· else ···の実行
           n==1が偽ならL2へ
             cmp  8[ebp],1
             jne  L2
           return 1;の実行
              mov  eax,1
            jmp  L1
       L2:  return n*fact(n-1);の実行
            n*fact(n-1)の計算
              fact(n-1)の計算
                n-1を計算し，結果をプッシュ
                  mov  eax,8[ebp]
                  sub  eax,1
                  push eax
                call _fact
                add  esp,4
              imul eax,8[ebp]
       L1: pop  ebp
         ret
```

その他のトピック

1. 局所関数
   - 定義を包含する他の局所関数・最上位関数のフレーム参照
   - 静的リンクまたはディスプレイを使用
2. ３オペランド命令と１オペランド命令
   - ２オペランド命令と同様に考察
3. 呼出し後保存レジスタ
   - 呼出し前保存レジスタを優先的に割り当てる
   - 関数呼出しを含む式は，呼出し後保存レジスタを使って高速化