教育用コンパイラXCCで x86コードを生成.

権藤克彦（東工大）

2009/3/25版

イントロダクション

概要（１）

- codegen-x86.c 中の関数 codegen()にコード生成部を実装してください.
  - codegen()はxcc.y 中で呼ばれます.
    - 構文解析・意味解析後に.
- コード生成のポイント
  - 式：後順序に抽象構文木を訪問.
    式の途中結果はスタックに積む.
  - 制御文：（条件）ジャンプ命令を使う.
  - 関数呼出し：引数をスタックに積んでcall命令.
  - 変数：
    - 引数や局所変数はベースポインタのオフセットで
      アクセス（例：8(%ebp)）
    - 大域変数はラベル名でアクセス（例：_x）

概要（2）：お約束

xcc.pptx 参照

- 記号表用の次の関数を使う.

```
struct Symbol *sym_lookup       (char *name);
struct Symbol *sym_lookup_label (char *name);
struct String *string_lookup    (char *data);
```

```
static void codegen_begin_block    (struct AST *ast);
static void codegen_end_block      (void);
static void codegen_begin_function (struct AST *ast);
static void codegen_end_function   (void);
```

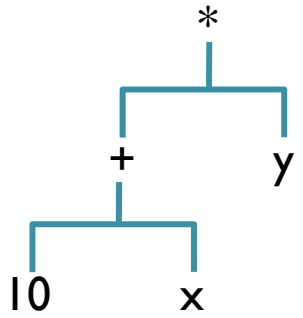
- アセンブリコード出力にはemit_code関数を使う.

```
static void emit_code (struct AST *ast, char *fmt, ...);
```

コード生成の概要：式（1）

- (10+x)*y のコード生成のイメージ.

式の計算結果は
スタックに積む



抽象構文木

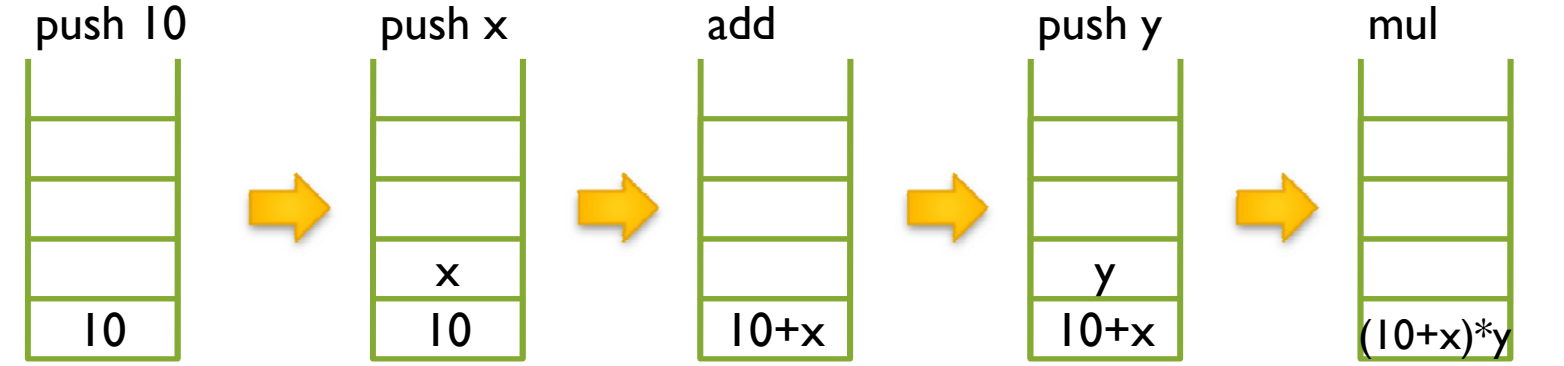
10 x + y *

後置記法に変換

```
push 10
push  x
add
push y
mul
```

これは実在しない
仮想コード

定数や変数はpush命令，
演算子は演算子命令に.



コード生成の概要：式（２）

- XCCのコード生成ではこうすると簡単.
  - x と y が大域変数の場合.

```
pushl   $10
pushl   _x
popl    %ecx
popl    %eax
addl    %ecx, %eax
pushl   %eax
pushl   _y
popl    %ecx
popl    %eax
imull   %ecx, %eax
pushl   %eax
```

定数10をスタックに積む.
変数xの値をスタックに積む.
スタックからポップしてxの値をレジスタ%ecxに格納.
スタックからポップして値10をレジスタ%eaxに格納.
%ecx と %eax の和を %eax に格納.
%eax の値（=10+x）をスタックに積む.
変数yの値をスタックに積む.
スタックからポップしてyの値をレジスタ%ecxに格納.
スタックからポップして10+xの値をレジスタ%eaxに格納.
%ecxと%eaxの積（=(10+x)*y）を%eaxに格納.
%eaxの値（=(10+x)*y）をスタックに積む.

x86では（レジスタを使わずに）スタック上の
2つのデータの和や積を計算できません.

コード生成の概要：式（３）

- GCCだとこうなる．
  - x と y が大域変数の場合．
  - すごくコンパクトになる．
    - 式の中間結果の格納場所として（スタックではなく）レジスタを積極的に使っているから．
    - ただし，レジスタ割り当て（register allocation）が複雑になる．→ 簡単のためスタックを使う手法を使おう．

```
movl    _x, %eax
addl    $10, %eax
imull   _y, %eax
```

変数xの値をレジスタ%eaxに格納．
定数10とレジスタ%eax（=x）の和を%eaxに格納．
変数yの値とレジスタ%eax（=10+x）の積を%eaxに格納．

コード生成の概要：制御文

- 「while (式) 文」は次のパターンでコード生成.

制御文はジャンプ命令を組み合わせる

コード生成の概要：関数呼出し

- 「式0 (式1, 式2, ..., 式n)」のコード生成.

多くの場合，
単なる関数名.

式0 (式1 , 式2 , ..., 式n)

関数呼び出しは
引数をスタックに積んで
call命令を実行する

式n のコンパイル結果	第n引数をスタックに積む.
⋮	
式2 のコンパイル結果	第2引数をスタックに積む.
式1 のコンパイル結果	第1引数をスタックに積む.
式0 のコンパイル結果	関数へのポインタをスタックに積む.

```
popl   %eax
call   *%eax
addl   $args_size, %esp
pushl  %eax
```

%eax = 関数へのポインタ
call命令で関数を呼び出す.
スタック上から引数を捨てる.
関数の返り値をスタックに積む.

args_size は引数のサイズの合計

コード生成の概要：変数

- int型の「変数x」のコード生成.

	左辺値	右辺値
大域変数	pushl $_x	pushl _x
引数	leal *offset*(%ebp), %eax pushl %eax	pushl *offset*(%ebp)
局所変数	leal *offset*(%ebp), %eax pushl %eax	pushl *offset*(%ebp)

左辺値は
メモリアドレスを
スタックに積む

右辺値は
メモリの中身を
スタックに積む

offset はベースポインタからの
相対オフセット（何バイト離れているか）

_xは大域変数xの
格納場所のラベル

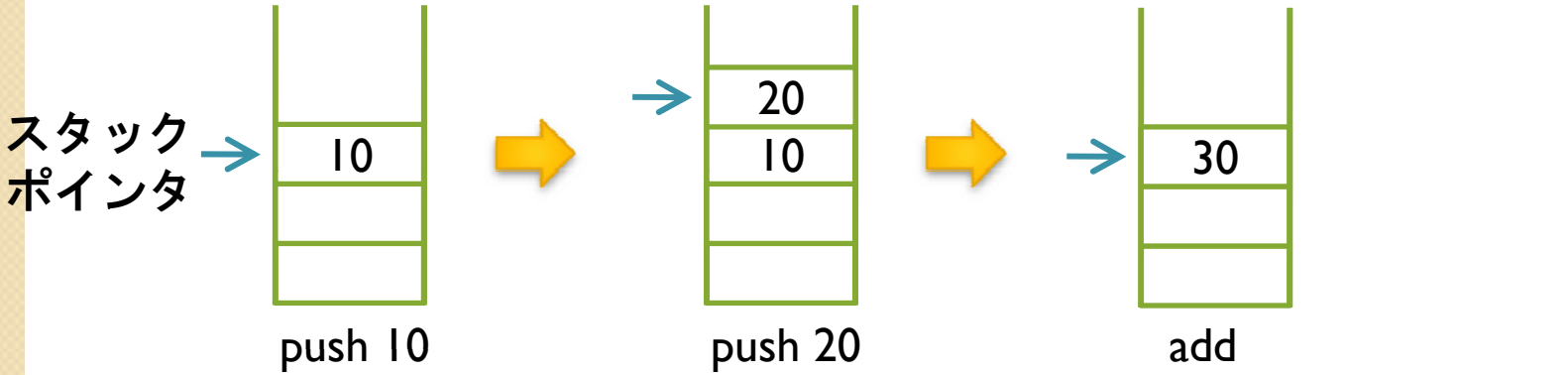
```
_x:
    .skip   4
```

式のコード生成

スタック機械

- スタック機械
  - 汎用レジスタが無い．→ 機械語命令にオペランドが無い．
  - 演算の対象や結果はスタック上に置く．

- 例：add は，push (pop () + pop ()) を計算する．
- 例：10+20の計算



最初のオペランドを
スタックに積む

次のオペランドを
スタックに積む

add命令はスタック上の2つの
データの和をスタックに積む

後置記法(postfix notation)

- 後置記法＝演算子をオペランドの後に書く記法.
  - 例：10+20の後置記法は 10 20 +.
    - 注：10+20は中置記法，+ 10 20は前置記法.
  - 例：(10+x)*y の後置記法は 10 x + y *.
- 別名，逆ポーランド記法.
- 後置記法の利点：
  - スタック機械の演算順序を直接表現する.
    - 式を左から右に読む.
    - オペランドならスタックにプッシュする.
    - 演算子（+や*）ならば，その演算子を実行する.

10 x + y *

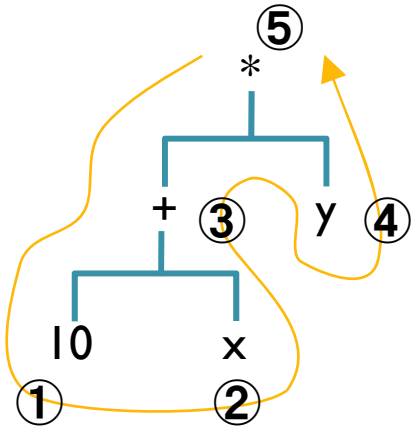
```
push 10
push  x
add
push y
mul
```

後順序(postorder)の木の訪問

- 抽象構文木から後置記法を得る方法.
  - 抽象構文木を**後順序**で訪問すれば良い.
- 後順序の訪問アルゴリズム（再帰的定義です）
  1. まず子ノードに後順序で（左から右に）訪問する.
  2. 次の自分のノードを処理する.

C言語で書くとこうなる.

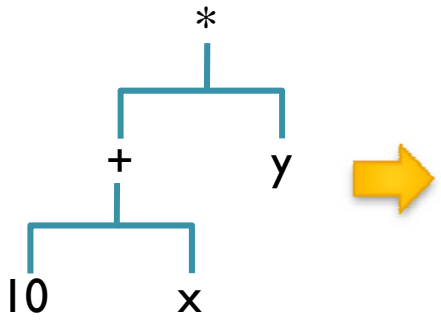
```
void
postorder (struct AST *ast)
{
  int i;
  for (i = 0; i < ast->num_child; i++) {
    postorder (ast->child [i]);
  }
  printf ("%d\n", ast->ast_type);
}
```



スタック機械：式のコード生成（１）

- スタック機械での式のコード生成：
  - 式の後置記法を左から右に読み，次の書き換えを行う．
    - 定数や変数→push命令．（例：push 10）．
    - 演算子→演算子命令．　（例：add）．

- 例：(10+x)*yのコード生成．



抽象構文木

10 x + y *

後置記法

```
push 10
push  x
add
push y
mul
```

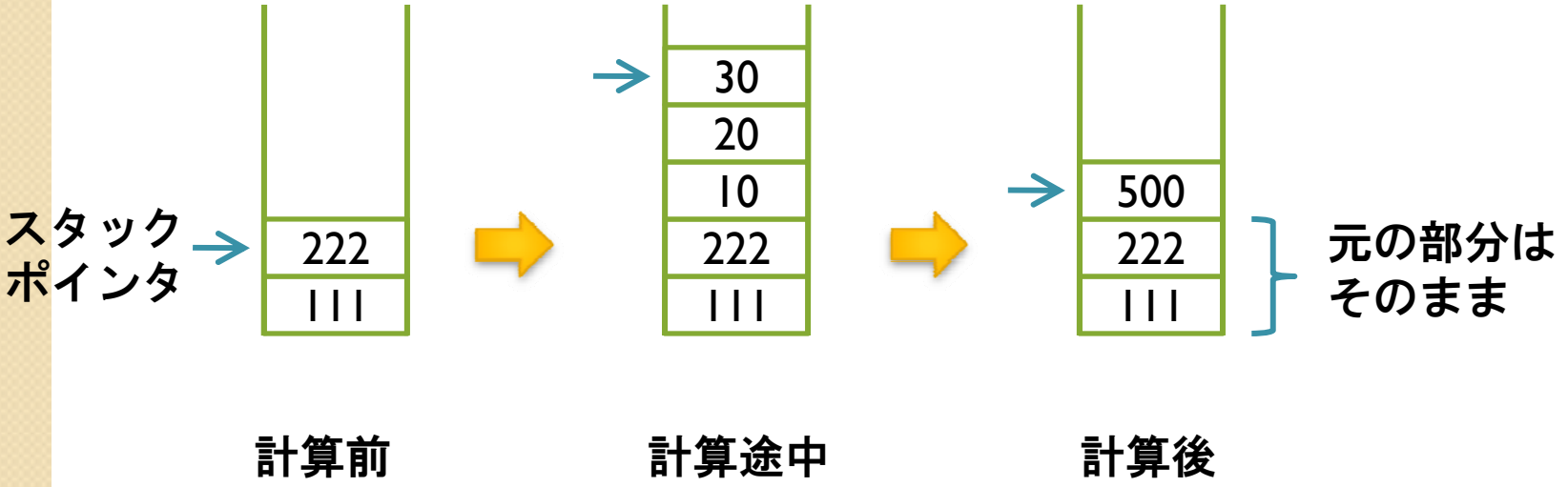
これは実在しない
仮想コード

スタック機械のコード

お約束：スタックの使い方

- 計算途中はいくらでもスタックを使ってよい.
- 計算終了後は元のスタックに計算結果を積んだ状態に必ずする.

- 例：10*(20+30)の計算



スタック機械：式のコード生成（2）

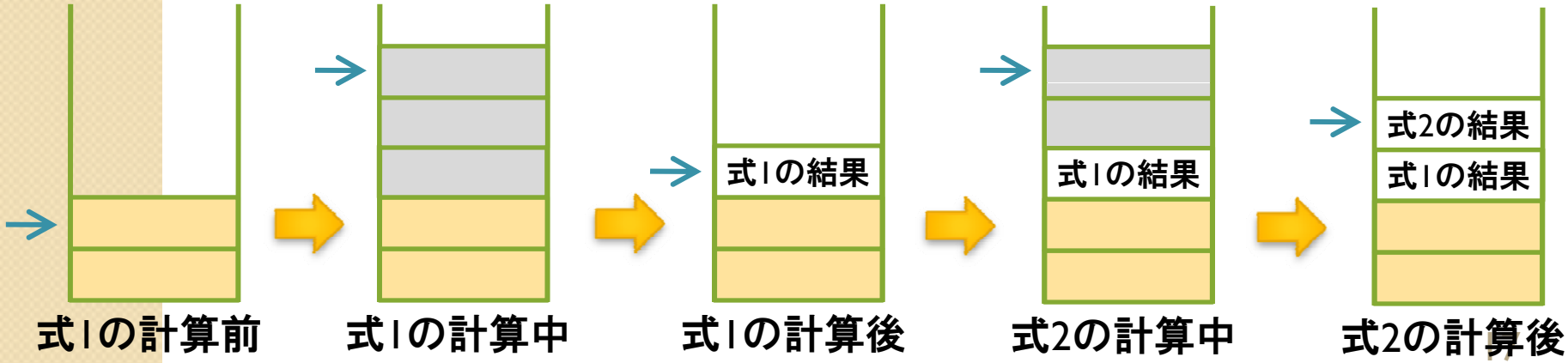
- 「式1+式2」のコード生成

  式1 のコンパイル結果

  式2 のコンパイル結果

  add

- 式1や式2がどんなに複雑でも，これでうまくいく．
  - 理由：式2の実行前には，元のスタックに式1の計算結果をプッシュした状態になる（する）から．式2も同じ．



x86での式のコード生成

- 演算子の実行直前に，スタックからレジスタに値を転送する．
  - この方法なら，決してレジスタ不足にはならない．
- 「式1+式2」のコード生成（int型同士の場合）

式1のコンパイル結果	式1の計算結果をスタックに積む
式2のコンパイル結果	式2の計算結果をスタックに積む
popl %ecx	式2の計算結果をレジスタ%ecxにポップ
popl %eax	式1の計算結果をレジスタ%eaxにポップ
addl %ecx, %eax	%ecx と %eax の和を%eax に格納
pushl %eax	%eax中の値をスタックにプッシュ

他にも方法はある．ここでは分かりやすさを優先．

ポインタ演算

- ＋と－は，オペランドがポインタ型のとき，意味が変わる．

```
int *p, *q, i;
p + i;
p - i;
p - q;
p + q;
```

p + i * sizeof (*p) を計算．
p - i * sizeof (*p) を計算．
(p - q) / sizeof (*p) を計算．
コンパイルエラー．

== のコード生成

- 「式1 == 式2」のコード生成.

式1 のコンパイル結果	式1の計算結果をスタックに積む
式2 のコンパイル結果	式2の計算結果をスタックに積む
popl %ecx	式2の計算結果をレジスタ%ecxにポップ
popl %eax	式1の計算結果をレジスタ%eaxにポップ
cmpl %ecx, %eax	%ecx と %eax を比較（%eflags が変化）
sete %al	等しいかの結果（0か1）を%alに位置を代入
movzbl %al, %eax	%al の値をゼロ拡張して %eax に転送.
pushl %eax	この式全体の結果（%eax）をスタックに積む

movzbl (Intel形式ではmovzx)は
バイトをダブルワードにゼロ
拡張して転送.

&& と || のコード生成

- 「左から右への評価」(left-to-right evaluation)を行う.
  - まず左オペランドを先に計算する.
  - 式全体の計算に必要な場合のみ，右オペランドを計算する.

- 例：(p != NULL) && (*p > 0)
  - (*p > 0)を計算するのは，(p != NULL)が真(1)の場合だけ.

  - 左オペランドの計算結果が偽(0)の場合は，右オペランドを計算せずに，式全体の値を偽(0)としなくてはいけない.

式のコード生成：定数

- 「定数」のコード生成.

  pushl $*value*　　*value* は定数値

式のコード生成：変数（1）

- int型の「変数x」のコード生成.

	左辺値	右辺値
大域変数	pushl $_x	pushl _x
引数	leal *offset*(%ebp), %eax pushl %eax	pushl *offset*(%ebp)
局所変数	leal *offset*(%ebp), %eax pushl %eax	pushl *offset*(%ebp)

左辺値は
メモリアドレスを
スタックに積む

右辺値は
メモリの中身を
スタックに積む

offset はベースポインタからの
相対オフセット（何バイト離れているか）

_xは大域変数xの
格納場所のラベル

```
_x:
    .skip   4
```

式のコード生成：変数（２）

- char型の「変数x」のコード生成（右辺値）.
  - int型に変換する.

```
movsbl  offset(%ebp), %eax
pushl   %eax
```

movsbl (Intel形式ではmovsx)はバイトをダブルワードに符号拡張して転送.

- 理由：char型の変数はint型に変換してプッシュする.
  - 言語仕様上，char型のデータは式中では int 型に暗黙に変換することになっているから.

詳細は「整数拡張(integer promotion)」を参照.

単項演算子のコード生成：&, ＊

- 「&式」の右辺値のコード生成.

  式 の左辺値のコード

- 「*式」の右辺値のコード生成.

  式 の右辺値のコード　計算結果はアドレス（のはず）

```
popl   %eax          アドレスをレジスタ%eaxに格納.
movl   0(%eax), %eax %eaxが指すメモリ中の値を%eaxに格納.
pushl  %eax          %eax値をプッシュ.
```

- 「*式」の左辺値のコード生成.

  式 の右辺値のコード　計算結果はアドレス（のはず）

代入演算子(=)のコード生成

- 「式1 = 式2」のコード生成（int型の場合）.

式2 の右辺値のコード	代入する値
式1 の左辺値のコード	代入先のアドレス

```
popl   %eax          アドレスを%eaxに代入
movl   0(%esp), %ecx スタックトップの値（＝代入する値）を%ecx に代入
movl   %ecx, 0(%eax) %ecx の値を%eaxが指すメモリ中に代入
```

- 代入する値をスタックトップに残してることに注意.
  - 代入式の値は，代入した値そのもの.
  - 例：printf ("%d\n", x = 999); /* 999 を表示 */
- x = y = z = 999 という式を評価するために必要.

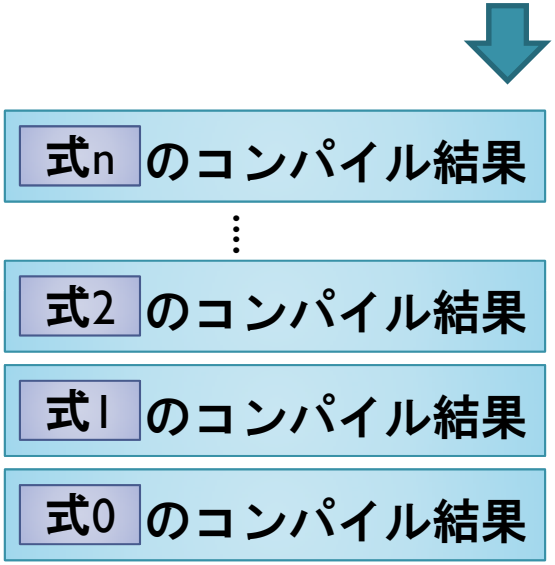
関数呼び出しのコード生成（1）

- 「式0 (式1, 式2, ..., 式n)」のコード生成.

多くの場合，
単なる関数名.

式0 (式1 , 式2 , ..., 式n)

関数呼び出しは
引数をスタックに積んで
call命令を実行する



第n引数をスタックに積む.

第2引数をスタックに積む.

第1引数をスタックに積む.

関数へのポインタをスタックに積む.

```
popl   %eax
call   *%eax
addl   $args_size, %esp
pushl  %eax
```

%eax = 関数へのポインタ
call命令で関数を呼び出す.
スタック上から引数を捨てる.
関数の返り値をスタックに積む.

args_size は引数のサイズの合計

関数呼び出しのコード生成（２）

- 「foo (10, 20)」のコード生成例.

```
pushl   $20          第2引数をスタックにプッシュ.
pushl   $10          第1引数をスタックにプッシュ.
pushl   $_foo        関数アドレスをスタックにプッシュ.
popl    %eax         関数アドレスを%eaxにポップ.
call    *%eax        関数をcall命令で呼び出す.
addl    $8, %esp     2つの実引数をスタック上から捨てる.
pushl   %eax         関数の返り値が%eaxに入っているので,
                     その返り値をスタックにプッシュ.
```

関数呼び出しのコード生成（３）

- 引数を評価する順序.
  - C言語仕様では未規定(unspecified).
  - コンパイラの都合の良い順序で評価してよい.
    - &&など，評価順序が決まっている演算子を除いて.

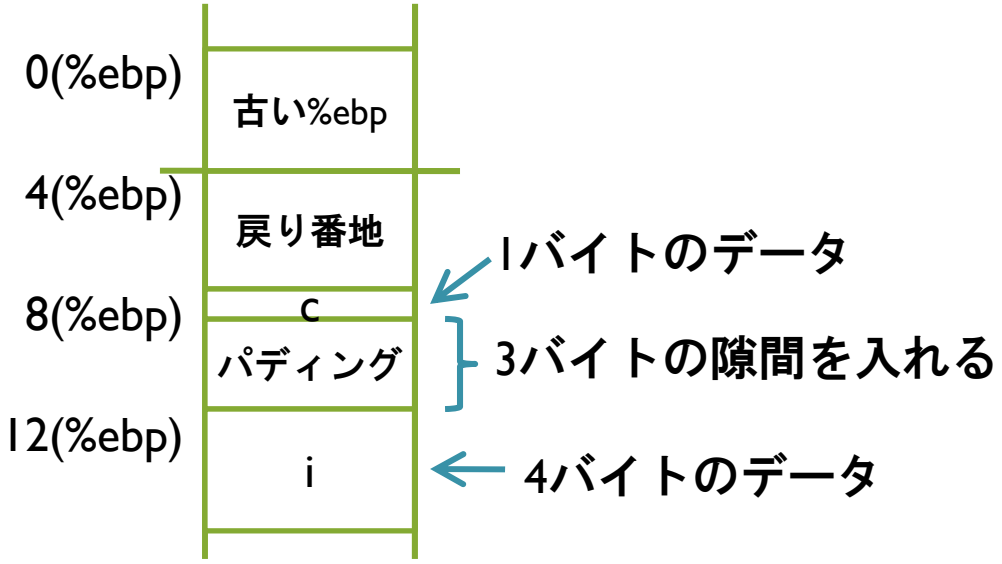
- 引数をスタックに積む順序・パディング.
  - ABI (application binary interface)が規定.
    - 例：http://www.caldera.com/developers/devspecs/abi386-4.pdf
    - 明文化されてないことも多い→ gcc出力を真似する.

  - ライブラリ関数とリンクしないのであれば，気にせず好きな方法で実装すればよい.

関数呼び出しのコード生成（４）

- お勧めの引数の積み方.
  - 逆順序に積む.
  - 引数は4バイト境界に置く.
    - つまり，4の倍数のアドレスに置く.
    - charの引数には3バイトのパディングが入る.

int foo (char c, int i); の場合

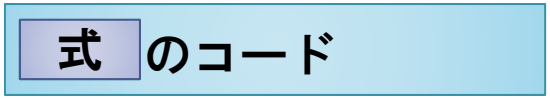


文のコード生成

式文のコード生成

- 式文＝式にセミコロンがついたもの.
  - 例：「foo (10, 20);」「x = 20;」「10;」
    - 最後の「10;」は文法上は正しいが，意味が無い.
    - 一般に式文は（値ではなく）副作用に意味がある.

- 「式;」のコード生成.

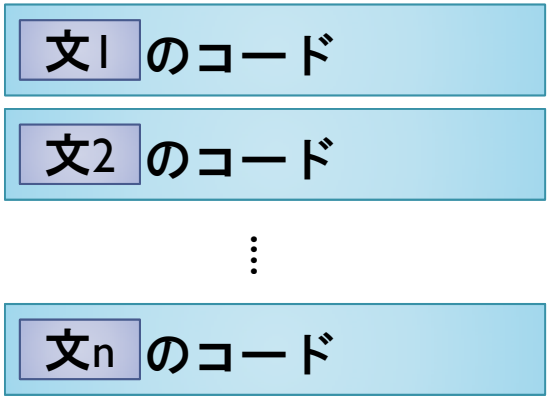



```
addl  $4, %esp
```

スタック上の式の値を捨てる.

ブロック文のコード生成

- 「{ 宣言リスト 文1 文2 … 文n}」のコード生成.



goto文とラベル文のコード生成

- 「ラベル：」のコード生成.

  label :

  label は他のラベルと
  重複しないラベル.

- 「goto ラベル;」のコード生成.

  jmp *label*

return文のコード生成

- 「return 式;」のコード生成.

式 のコード

```
popl  %eax
movl  %ebp, %esp
popl  %ebp
ret
```

関数の返り値を%eaxに格納
スタックフレームを廃棄

ret命令で戻り番地に制御を戻す.

または

式 のコード

```
popl  %eax
jmp   label
      ⋮
label :
movl  %ebp, %esp
popl  %ebp
ret
```

関数定義の最後の部分. p.40参照.

if 文のコード生成

- 「if (式) 文」のコード生成

if-else文のコード生成

- 「if (式) 文1 else 文2」のコード生成

式 のコード

```
    popl  %eax
    cmpl  $0, %eax
    je    L1
```

文1 のコード

```
    jmp   L2
L1 :
```

文2 のコード

```
L2 :
```

while文のコード生成

- 「while (式) 文」のコード生成.

L1:

式 のコード

```
popl  %eax
cmpl  $0, %eax
je    L2
```

文 のコード

```
jmp   L1
```

L2:

関数・その他のコード生成

関数定義のコード生成

- 関数定義「関数名 (引数) ブロック文」のコード生成

文字列定数のコード

p.42を参照

```
    .text
    .global   _funcname
    .def  _funcname; .scl 2; .type 32; .endef
_funcname :
    pushl  %ebp
    movl   %esp, %ebp
    subl   $local_size, %esp
```

以降をテキストセクションに配置
関数名を外部リンケージと宣言.
記憶クラスを大域，型を関数と定義.
関数名のラベル.
新しいスタックフレームを作成.

局所変数の領域を確保.

ブロック文 のコード

```
label :
    movl  %ebp, %esp
    popl  %ebp
    ret
```

return文のジャンプ先. p.35参照.
スタックフレームの破棄.

ret命令で戻り番地に制御を移す.

関数定義のコード生成（例）

- 関数定義「関数名 (引数) ブロック文」のコード生成

```
int add (int x, int y)
{
    int tmp;
    tmp = x + y;
    return tmp;
}
```

```
        .text
        .globl  _add
        .def    _add; .scl 2; .type 32; .endef
_add:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
```

ブロック文 のコード

```
L.XCC.RE.add:
        movl    %ebp, %esp
        popl    %ebp
        ret
```

文字列定数のコード生成

- 関数中の文字列定数をすべてセクション .rdata で定義しておく．（read-only なデータセクション）

```
    .section  .rdata,  "dr"
L1:
    .ascii    "aaaaa\0"
L2 :
    .ascii    "bbbbb\0"
    ....
```

文字列定数をヌル終端させるために
\0 をつける必要がある．
（.asciz 擬似命令を使う方法もある．）