プログラミング言語 10
言語処理系

田浦

言語処理系実装の形態

- インタプリタ: プログラムを解釈実行 (プログラムと入力から出力を直接計算)
- トランスレータ: プログラムを別の言語 (例: C) に翻訳
  - 例: OpenMP (C の並列拡張) を C (+ Pthreads) に翻訳
- コンパイラ: プログラムを機械語に翻訳

なぜ(今も)言語処理系を学ぶか

- 新ハードウェア用の処理系
  - GPU 用の C/C++言語 (CUDA, OpenACC, OpenMP)
  - プロセッサの新しい命令セット (e.g., SIMD) への対応
  - 量子コンピュータ, 量子アニーラ
- 新汎用言語
  - Scala, Julia, Go, Rust, etc.
- 言語の拡張
  - 並列処理用拡張 (例: OpenMP, CUDA, OpenACC, Cilk)
  - ベクトル命令用拡張
  - 型システム拡張 (例: PyPy, TypeScript)
- 目的に特化した言語
  - 統計パッケージ (R, MatLab, etc.)
  - データ処理 (SQL, NoSQL, SPARQL, etc.)
  - 機械学習
  - 制約解消系, 定理証明系 (Coq, Isabelle, etc.)
  - アプリケーション用マクロ言語 (Visual Basic (MS Office 用), Emacs Lisp (Emacs), Javascript (ウェブブラウザ), etc.)

高水準言語 vs. 機械語

	高水準言語 (e.g., C)	機械語
制御構造	for, while, if, ...	≈ go to だけ
式	任意の入れ子	≈ C = A op B だけ
局所変数の数	いくらでも	≈ レジスタ数まで
局所変数の寿命	関数実行中	≈ 関数呼び出しまで

- これらのギャップを埋めるのがコンパイラ
- https://www.felixcloutier.com/x86/index.html

コード生成 — 人間コンパイラ内観

- 例: 以下 (ちなみに $\sqrt{c}$ を求めるニュートン法) をどう機械語にするか

```
1  double sq(double c, long n) {
2    double x = c;
3    for (long i = 0; i < n; i++) {
4      x = x / 2 + c / (x + x);
5    }
6    return x;
7  }
```

ステップ1 — 制御構造を goto だけに

```
1  double sq(double c, long n) {
2    double x = c;
3    for (long i = 0; i < n; i++) {
4      x = x / 2 + c / (x + x);
5    }
6    return x;
7  }
```

⇒

```
1  double sq(double c, long n) {
2    double x = c;
3    long i = 0;
4    if (i >= n) goto Lend;
5  Lstart:
6    x = x / 2 + c / (2 * x);
7    i++;
8    if (i < n) goto Lstart;
9  Lend:
10   return x;
11 }
```

ステップ2 — 全部を C = A op B に

```
1  double sq(double c, long n) {
2    double x = c;
3    long i = 0;
4    if (i >= n) goto Lend;
5  Lstart:
6    x = x / 2 + c / (2 * x);
7    i++;
8    if (i < n) goto Lstart;
9  Lend:
10   return x;
11 }
```

⇒

```
1  double sq3(double c, long n) {
2    double x = c;
3    long i = 0;
4    if (!(i < n)) goto Lend;
5  Lstart:
6    double t0 = 2;
7    double t1 = x / t0;
8    double t2 = t0 * x;
9    double t3 = c / t2;
10   x = t1 + t3;
11   i = i + 1;
12   if (i < n) goto Lstart;
13 Lend:
14   return x;
15 }
```

ステップ3 — 変数に機械語レベルでの変数 (レジスタまたはメモリ) を割り当て

- 注: 浮動小数点数の定数は命令中には書けない

```
1  /* c : xmm0, n : rdi */
2  double sq3(double c, long n) {
3    double x = c;        /* x : xmm1 */
4    long i = 0;          /* i : rsi */
5    if (!(i < n)) goto Lend;
6   Lstart:
7    double t0 = 2;       /* t0 : xmm2 */
8    double t1 = x / t0; /* t1 : xmm3 */
9    double t2 = t0 * x; /* t2 : xmm4 */
10   double t3 = c / t2; /* t3 : xmm5 */
11   x = t1 + t3;
12   i = i + 1;
13   if (i < n) goto Lstart;
14  Lend:
15   return x;
16 }
```

ステップ4 — 命令に変換

```
1  /* c : xmm0, n : rdi */
2  double sq3(double c, long n) {
3   # double x = c;        /*x:xmm1*/
4   movasd %xmm0,%xmm1
5   # long i = 0;          /*i:rsi*/
6   movq $0,%rsi
7  .Lstart:
8   # if (!(i < n)) goto Lend;
9   cmpq %rdi,%rsi  # n - i
10  jle .Lend
11  # double t0 = 2;       /*t0:xmm2*/
12  movasd .L2(%rip),%xmm2
13  # double t1 = x / t0; /*t1:xmm3*/
14  movasd %xmm1,%xmm3
15  divq %xmm2,%xmm3
16  # double t2 = t0 * x; /*t2:xmm4*/
17  movasd %xmm0,%xmm4
18  mulsd xmm2,%xmm4
```

```
1   # double t3 = c/t2; /*t3:xmm5*/
2   movasd %xmm0,%xmm5
3   divsd %xmm4,%xmm5
4   # x = t1 + t3;
5   movasd %xmm3,%xmm1
6   addsd %xmm5,%xmm1
7   # i = i + 1;
8   addq $1,%rsi
9   # if (i < n) goto Lstart;
10  cmpq %rdi,%rsi  # n - i
11  jl .Lstart
12 .Lend:
13  # return x;
14  movq %xmm1,%xmm0
15  ret
16 }
```

コード生成 — 一般的には難しいところ

- 気軽に各途中結果にレジスタを割り当てたが ...

```
1   double x = c;          /* x : xmm1 */
2   long i = 0;            /* i : rsi */
3  Lstart:
4   if (!(i < n)) goto Lend;
5   double t0 = 2;         /* t0 : xmm2 */
6   double t1 = x / t0;    /* t1 : xmm3 */
7   double t2 = t0 * x;    /* t2 : xmm4 */
8   double t3 = c / t2;    /* t3 : xmm5 */
```

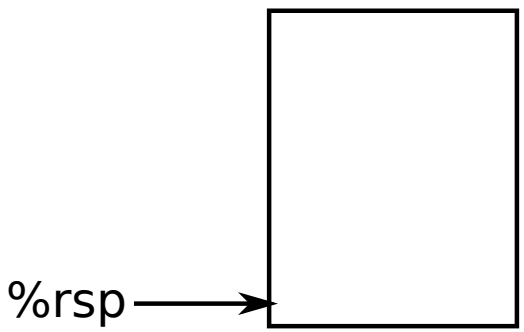
- レジスタは足りなくなるかも知れない
- 多くのレジスタは関数呼び出しをまたがると破壊される
- オペランドレジスタが限定されている命令もある (e.g., 整数割り算の被除数は %rax, %rdx ≡ %rax, %rdx は割り算をまたがると破壊される)
- → 一般にはメモリ (スタック領域) も使う必要がある

コード生成 — 超単純版

- 途中結果は一般にはメモリ (スタック領域) も使う必要がある
  ⇒ 「常に」スタック領域を使うのが単純

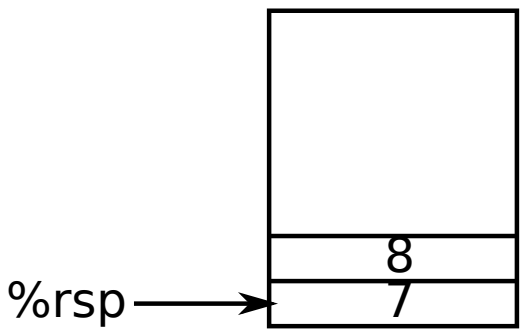
関数呼び出しに伴うスタックの動き

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- f実行中



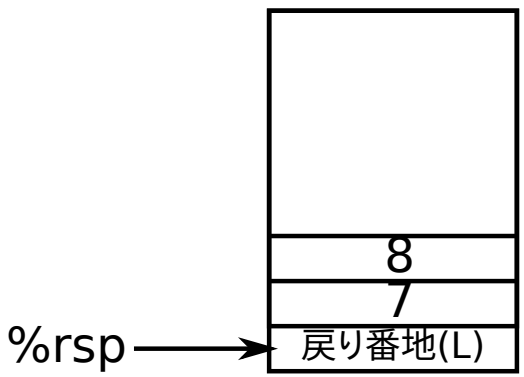
関数呼び出しに伴うスタックの動き

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- `call f` 実行直前



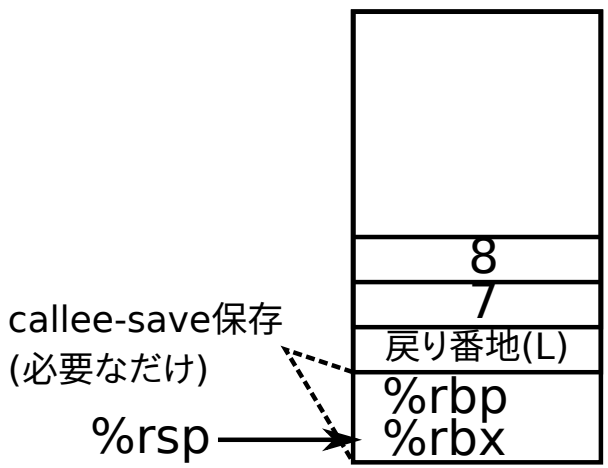
関数呼び出しに伴うスタックの動き

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- `call f` 実行直後 (g 開始)



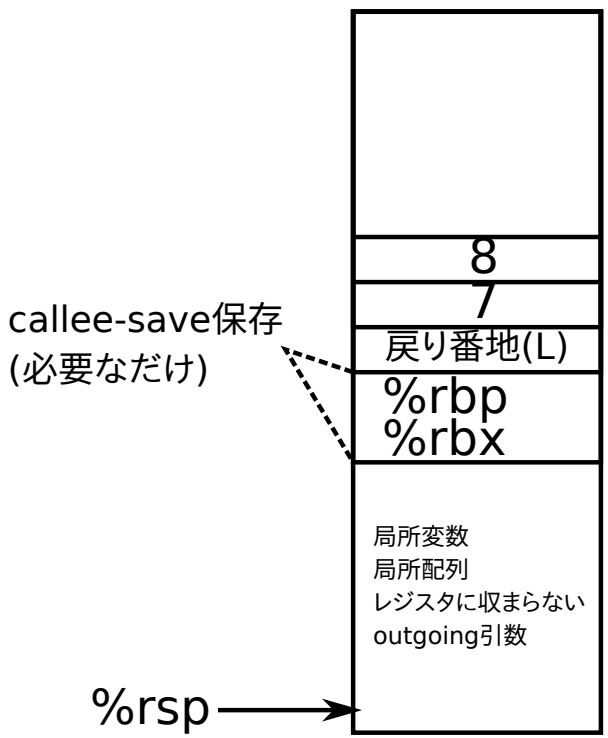
関数呼び出しに伴うスタックの動き

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- g が使う callee-save レジスタ保存



関数呼び出しに伴うスタックの動き

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- g 実行中



超単純版でのコード生成例

- 意味を感じられるよう, 関数呼び出しを含む例

```
1  double integ(long n) {
2    double x = 0;
3    double dx = 1 / (double)n;
4    double s = 0;
5    for (long i = 0; i < n; i++) {
6      s += f(x);
7      x += dx;
8    }
9    return s * dx;
10 }
```

“goto” 化と “C = A op B” 化

```
1  double integ3(long n) {     /*  n :  0(%rsp) */
2    double x = 0;             /*  x :  8(%rsp) */
3    double t0 = 1;            /* t0 : 16(%rsp) */
4    double t1 = (double)n;    /* t1 : 24(%rsp) */
5    double dx = t0 / t1;      /* dx : 32(%rsp) */
6    double s = 0;             /*  s : 40(%rsp) */
7    long i = 0;               /*  i : 48(%rsp) */
8    if (!(i < n)) goto Lend;
9   Lstart:
10   double t2 = f(x);         /* t2 : 56(%rsp) */
11   s += t2;
12   x += dx;
13   i += 1;
14   if (i < n) goto Lstart;
15   Lend:
16   double t3 = s * dx;       /* t3 : 64(%rsp) */
17   return t3;
18 }
```

機械語

```
1  double integ3(long n) {
2    /* n : 0(%rsp) */
3    movq %rdi,0(%rsp)
4    # double x = 0;
5    /* x : 8(%rsp)*/
6    movsd .L0(%rip),%xmm0
7    movsd %xmm0,8(%rsp)
8    # double t0 = 1;
9    /* t0 : 16(%rsp)*/
10   movq $1,16(%rsp)
11   # double t1 = (double)n;
12   /* t1 : 24(%rsp)*/
13   cvtsi2sdq 0(%rsp),%xmm0
14   movsd %xmm0,24(%rsp)
15   # double dx = t0 / t1;
16   /* dx : 32(%rsp) */
17   movsd 16(%rsp),%xmm0
18   divsd 24(%rsp),%xmm0
19   movsd %xmm0,32(%rsp)
20   # double s = 0;
21   /* s : 40(%rsp) */
22   movsd .L0(%rip),%xmm0
23   movsd %xmm0,40(%rsp)
```

```
1    # long i = 0;
2    /* i : 48(%rsp) */
3    movq $0,48(%rsp)
4    # if (!(i < n)) goto Lend;
5    movq 0(%rsp),%rdi
6    cmpq 48(%rsp),%rdi # n - i
7    jle .Lend
8  .Lstart:
9    # double t2 = f(x);
10   /* t2 : 56(%rsp) */
11   movq 8(%rsp),%rdi
12   call f
13   movq %rax,56(%rsp)
14   # s += t2;
15   movq 40(%rsp),%xmm0
16   addsd 56(%rsp),%xmm0
17   movq %xmm0,40(%rsp)
18   # x += dx;
19   movsd 8(%rsp),%xmm0
20   addsd 32(%rsp),%xmm0
21   movsd %xmm0,8(%rsp)
```

機械語

```
1    # i += 1;
2    movq 48(%rsp),%rdi
3    addq $1,%rdi
4    movq %rdi,48(%rsp)
5    # if (i < n) goto Lstart;
6    movq 0(%rsp),%rdi
7    cmpq 48(%rsp),%rdi # n - i
8    jg .Lstart
9  .Lend:
10   movsd 40(%rsp),%xmm0
11   addsd 32(%rsp),%xmm0
12   addsd %xmm0,64(%rsp)
13   # return t3;
14   addsd 64(%rsp),%xmm0
15   ret
16 }
```

コード生成 — 演習での前提

- 型は long (8 バイト整数) のみ
  - したがって typedef なども無し
  - int もなし, 浮動小数点数もポインタもなし
  - 全部 long だから静的な型検査もいらない
- 大域変数もなし ⇒
  - プログラム = 関数定義のリスト
- ややこしい文は if, while, 複合文 ({ ... }) のみ
- 変数宣言は複合文の先頭で, 初期化の式もなし
- 以上は字句の定義 (cc_lex.mll), 文法の定義 (cc_parse.mly) に反映されている

プログラムの構成

- cc_ast.ml — 構文木定義
- cc_parse.mly — 文法定義
- cc_lex.mll — 字句定義
- cc_cogen.ml — 構文木からコード生成
- cc.ml — メイン

構文木定義 — 関数定義

- C の関数定義の例

```
1 long f (long x, long y) {
2   return x + y;
3 }
```

- ⇒ 関数定義の構文木の定義

```
1 type definition =
2   FUN_DEF of (type_expr * string * (type_expr * string) list * stmt)
```

構文木の定義 — 文

- if 文

```
1 if (x < y) { x++; return x; }  else return y;
```

⇒

```
1 STMT_IF of (expr * stmt * stmt)
```

- 複合文

```
1 { long r; if (x < y) r = 10; else r = 20;  }
```

⇒

```
1 STMT_COMPOUND of ((type_expr * string) list * stmt list)
```

構文木の定義 — 文

- while 文

```
while (i < n) { foo(i); i++; }
```

⇒

```
STMT_WHILE of (expr * stmt)
```

- ⇒ (諸々まとめた) 文の構文木の定義

```
type stmt =
  STMT_EMPTY
| STMT_CONTINUE
| STMT_BREAK
| STMT_RETURN of expr  (* e.g., return 123; *)
| STMT_EXPR of expr    (* e.g., f(x); *)
| STMT_COMPOUND of ((type_expr * string) list * stmt list)
| STMT_IF of (expr * stmt * stmt)
| STMT_WHILE of (expr * stmt)
```

構文木の定義 — 式

- 2 項演算

```
x + y + 1
```

⇒

```
EXPR_BIN_OP of bin_op * expr * expr
```

注: 代入 (a = b) も 2 項演算の一種 (C の代入は, 文ではなく式)

- 関数呼び出し

```
... f(x + 1, y + 2, z + 3) ...
```

⇒

```
EXPR_CALL of (string * expr list)
```

構文木の定義 — 式

- ⇒ (諸々まとめた) 式の構文木の定義

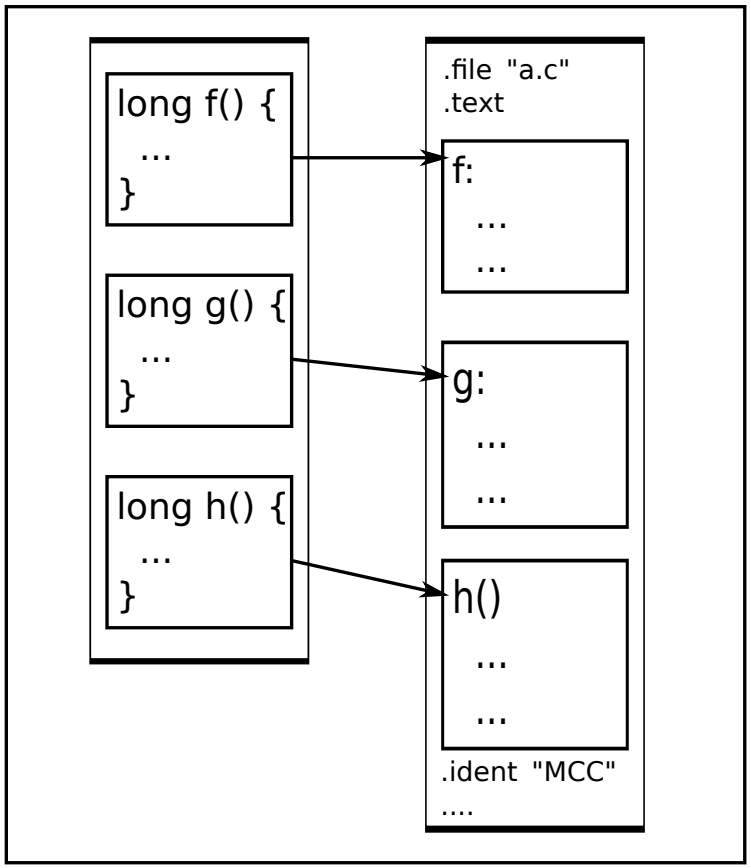
```
1 type expr =
2   EXPR_NUM of int    (* e.g., 3 *)
3 | EXPR_VAR of string (* e.g., x *)
4 | EXPR_BIN_OP of bin_op * expr * expr
5 | EXPR_UN_OP of un_op * expr (* e.g., -f(x) *)
6 | EXPR_CALL of (string * expr list)
```

コード生成 — 基本スタイル

- ある構文木に対する機械語 ≈ その構成要素に対する機械語を適切に並べたもの

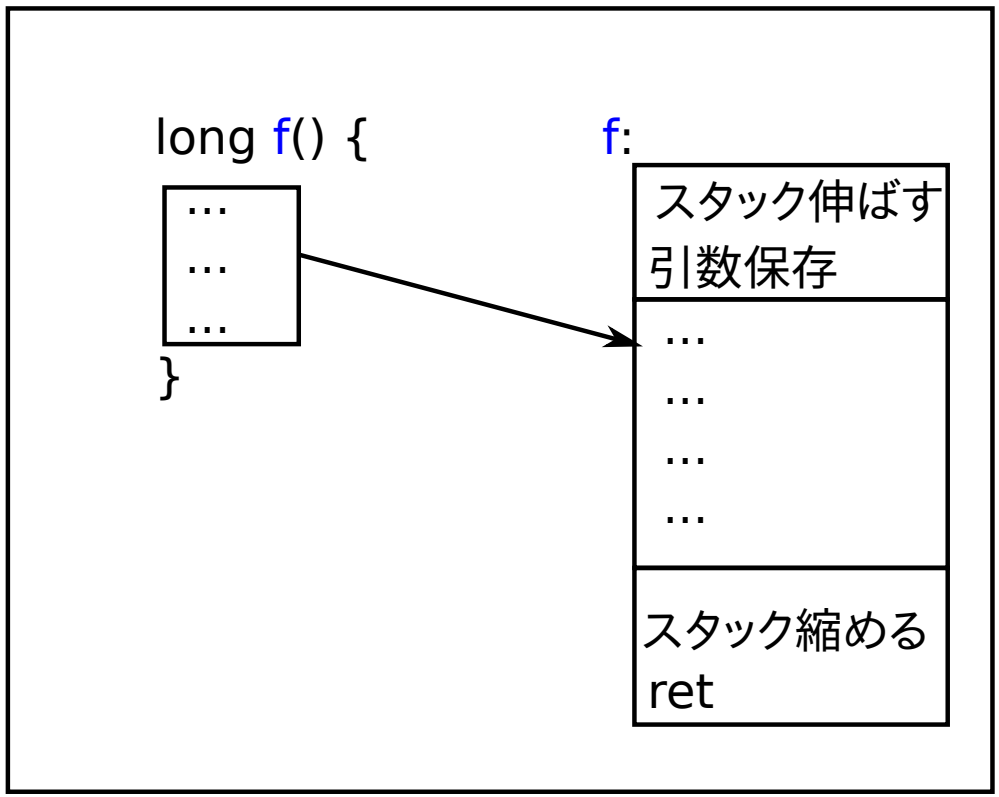
ファイル全体のコンパイル

- (要約) 関数毎にコンパイルしたものを連結



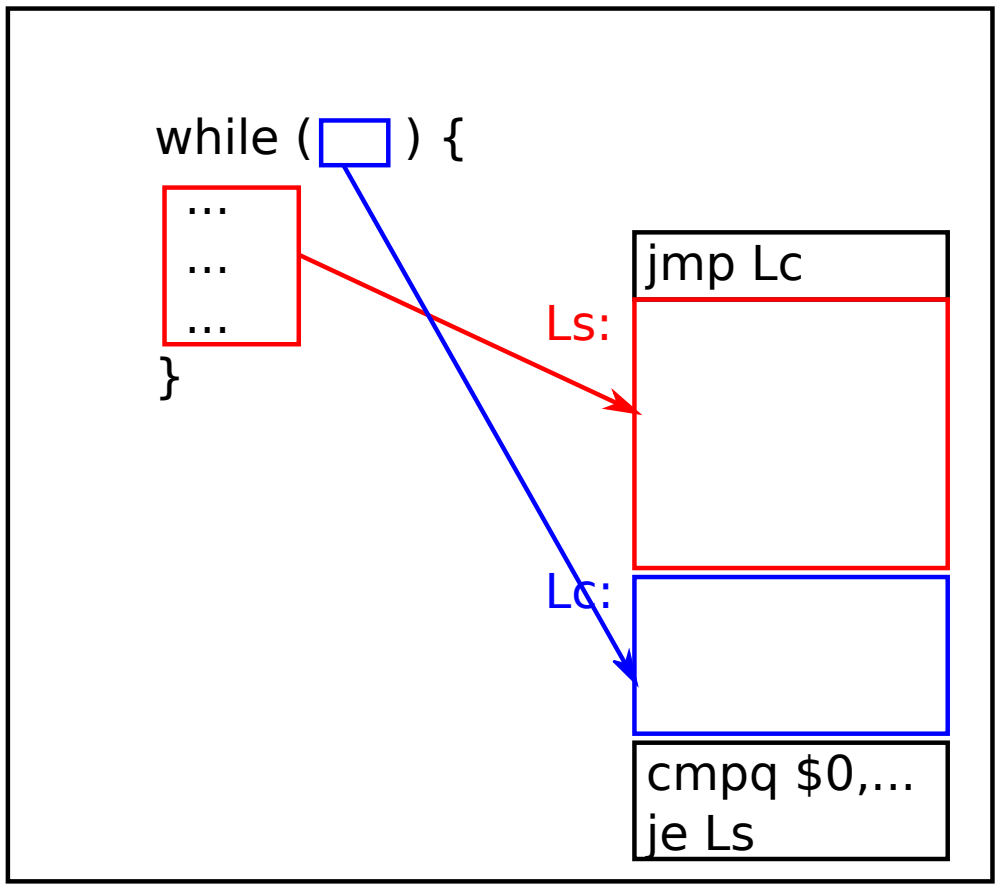
関数定義のコンパイル

- (要約) 文をコンパイルしたものの前後に, プロローグ (スタックを伸ばす, etc.), エピローグ (スタックを縮める) をつける



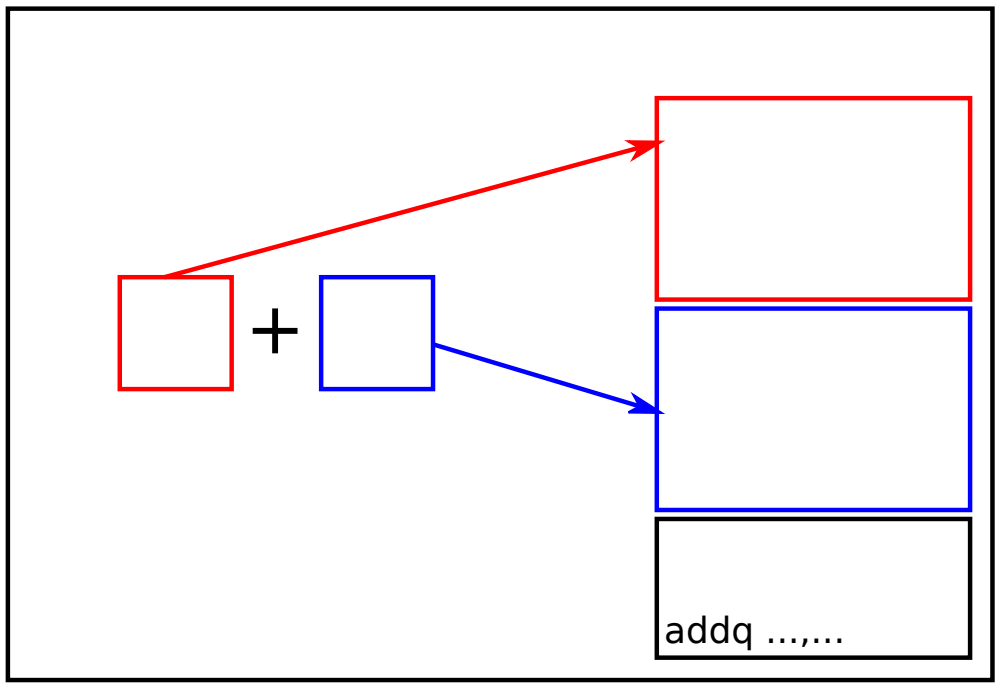
文のコンパイル (while 文を例に)

- (要約) 条件式, 本体をコンパイルしたものを以下のように配置. ループの継続判定コードをつける



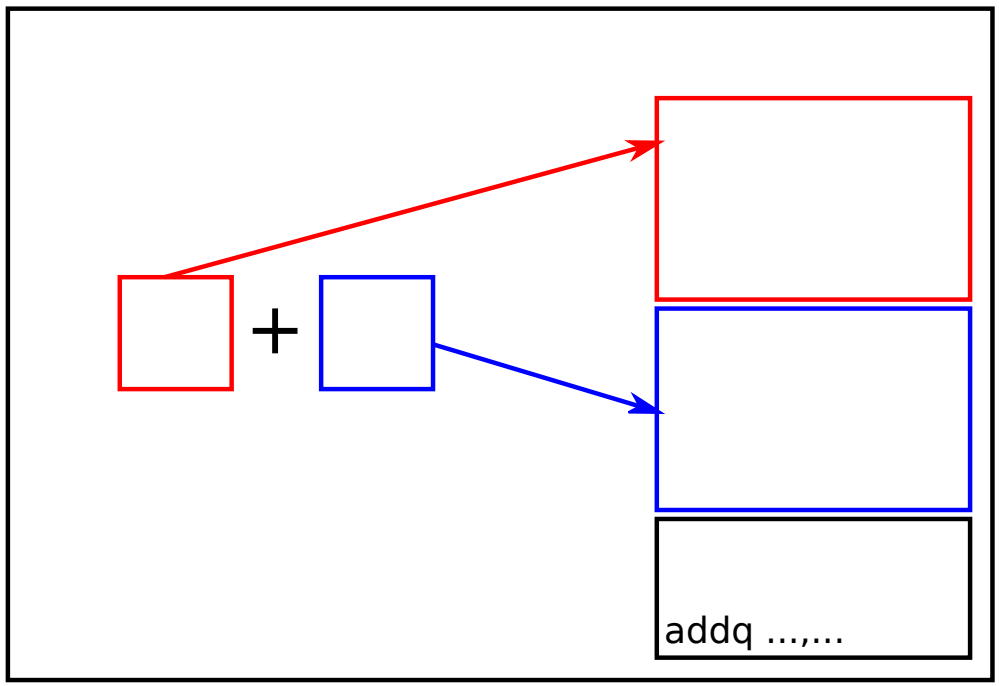
式のコンパイル(足し算を例に)

- (要約) + の引数をそれぞれコンパイルして足し算命令をつける



式のコンパイル (比較演算の場合)

- (要約) < の引数をそれぞれコンパイルして比較すればよいが ...



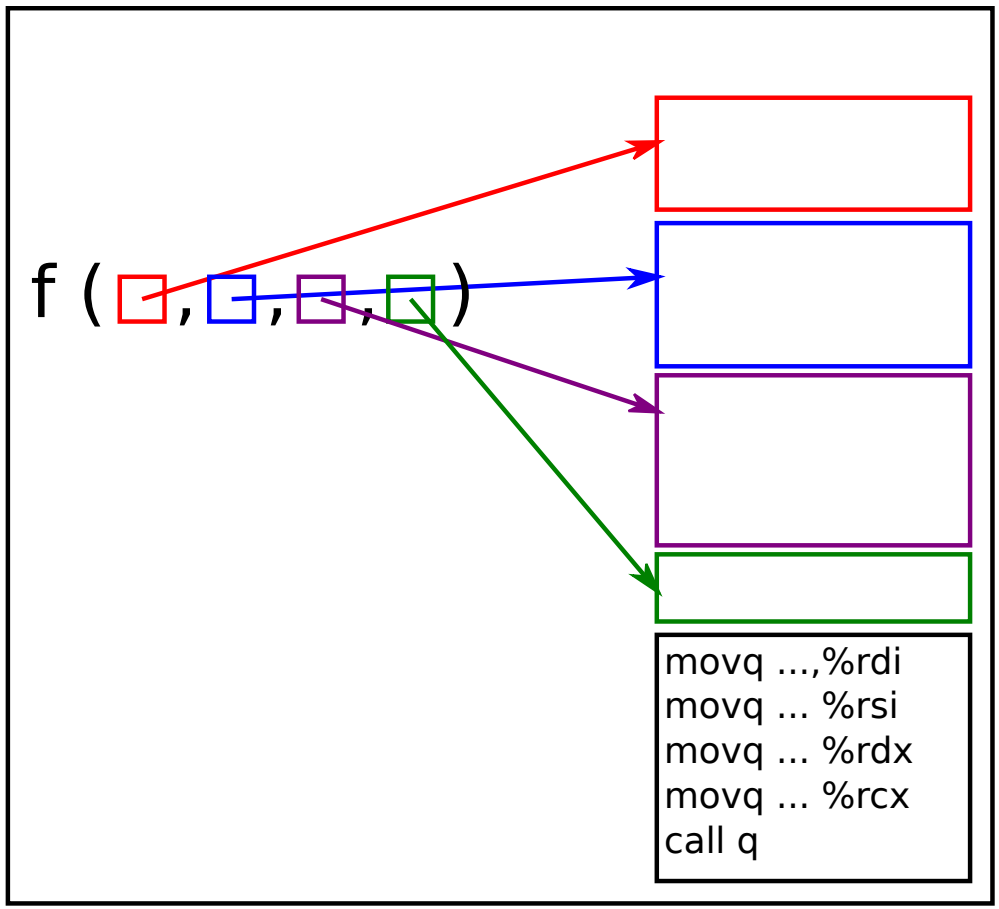
式のコンパイル (比較演算の場合)

- A < B は,
  - A < B ならば 1
  - A >= B ならば 0

  という値を持つ式
- `z = x < y;`, `(x < y) + z` のような式も許される (if や while の条件部分に来るとは限らない) ことに注意
- アセンブリ言語でこれを生成する命令は?
  1. 方法 1: 条件分岐
  2. 方法 2: 条件 move 命令

式のコンパイル (関数呼び出し)

- (要約) 引数をそれぞれコンパイル; 引数を所定の位置に並べる; call 命令

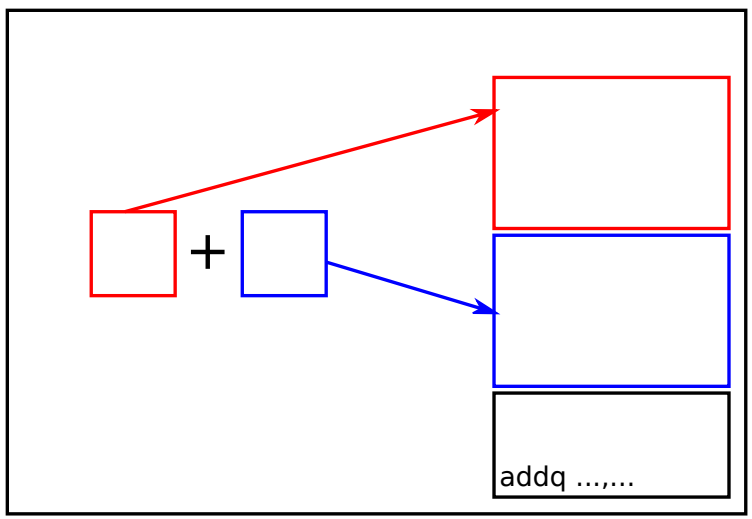


OCaml プログラムでは

- 上述したような, コンパイルの方針は OCaml でパターンマッチ (match 式) と再帰呼び出しを使うと非常に見通しよく書ける

部分式間の値の受け渡し

- (要約) A + B のような式のコンパイルでは, A や B を評価した結果が格納された位置をもとに, 足し算命令を発行する必要がある
- A や B が定数の場合のように, 直接命令のオペランドに出来る場合もある
- $\Rightarrow$ 式のコンパイル結果は, (命令列, 結果を示すオペランド) の組



変数の格納位置

- `x + y` のような式をコンパイルする時, `x` や `y` をコンパイルする必要がある
- `x` をコンパイルするには, `x` がどこに格納されているかを知る必要がある
- $\Rightarrow$ 「環境」が必要
- 環境 = 「変数名 (文字列) $\mapsto$ 変数の位置」のマッピング

中間言語 (IR)

- 原理的には「構文木, 環境」から直接アセンブリ言語を出すことも可能だが, 色々な理由で, アセンブリ言語と似ているが少し違う「中間言語 (Intermediate Representation; IR)」を通すことが普通
- IR vs. アセンブリ

	IR	機械語
制御構造	$\approx$ go to だけ	$\approx$ go to だけ
式	$\approx$ C = A op B だけ	$\approx$ C = A op B だけ
局所変数の数	いくらでも	$\approx$ レジスタ数まで
局所変数の寿命	関数実行中	$\approx$ 関数呼び出しまで

中間言語 (IR) の存在理由

- 入力 → IR の変換を易しくする (任意個の, 寿命が関数呼び出しをまたがる変数を利用可能にする)
- 複数の入力言語の実装を容易にする — C, C++, Java, etc. で入力言語 → IR 以外は共通
- 複数プロセッサへの実装を容易にする — Intel, ARM, etc. で, IR → 機械語 以外は共通
- 最適化 — 「IR → IR の変換」または「IR → 機械語の変換」で種々の最適化を表現