# プログラミング言語 10
# 言語処理系 / Implementing Programming Languages

田浦

# 目次 / Contents

# 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

# Various forms of language implementation

- interpreter: interprets and executes programs (takes a program and an input; and computes the output)
- translator: translates programs into another language (e.g., C)
  - e.g. translate OpenMP (parallel extension to C) to C (+ Pthreads)
- compiler: translates programs into a machine (assembly) code

# 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.)

# Why do you want to build a language, today?

- new hardware
  - C/C++ for GPUs (CUDA, OpenACC, OpenMP)
  - new instruction set (e.g., SIMD) of the processor
  - quantum computers, quantum annealers
- new general purpose languages
  - Scala, Julia, Go, Rust, etc.
- new extension
  - parallel processing (ex: OpenMP, CUDA, OpenACC, Cilk)
  - vector/SIMD processing
  - type system extension for safety (ex: PyPy, TypeScript)
- new special purpose (domain specific) languages
  - statistics (R, MatLab, etc.)
  - data processing (SQL, NoSQL, SPARQL, etc.)
  - deep learning
  - constraint solving, proof assistance (Coq, Isabelle, etc.)
  - macro (Visual Basic (MS Office), Emacs Lisp (Emacs), Javascript (web browser), etc.)

# Contents

1. コンパイラの仕事 / What do compilers do basically?

2. より一般の場合 / More general cases

3. 最小限の C からのコード先生器/ Implementing a minimum C compiler

4. 中間言語/ Intermediate Representation

# 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`
- `https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start`

# High level language vs. machine code

| | high-level (e.g., C) | machine |
|---|---|---|
| control | for, while, if, . . . | ≈ jump (“go to”) only |
| expression | arbitrary nest | ≈ C = A op B only |
| the number of local variables | arbitrary | ≈ only a fixed number of registers |
| lifetime of a local variable | during the function call that defined it | ≈ some registers live only up to the next function call |

- compiler’s main job is to fill those gaps
- `https://www.felixcloutier.com/x86/index.html`
- `https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start`

# 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

```
double sq(double c, long n) {
  double x = c;
  for (long i = 0; i < n; i++) {
    x = x / 2 + c / (x + x);
  }
  return x;
}
```

# Code generation by hand — introspecting "human compiler"

- ex: how to convert the following (which finds $\sqrt{c}$ by the Newton method) into machine language

```
double sq(double c, long n) {
  double x = c;
  for (long i = 0; i < n; i++) {
    x = x / 2 + c / (x + x);
  }
  return x;
}
```

# ステップ 1 — 制御構造を goto だけに

```c
double sq(double c, long n) {
  double x = c;
  for (long i = 0; i < n; i++) {
    x = x / 2 + c / (x + x);
  }
  return x;
}
```

⇒

```c
double sq(double c, long n) {
  double x = c;
  long i = 0;
  if (i >= n) goto Lend;
Lstart:
  x = x / 2 + c / (2 * x);
  i++;
  if (i < n) goto Lstart;
Lend:
  return x;
}
```

# Step 1 — make all controls “goto”s

```
double sq(double c, long n) {
  double x = c;
  for (long i = 0; i < n; i++) {
    x = x / 2 + c / (x + x);
  }
  return x;
}
```

⇒

```
double sq(double c, long n) {
  double x = c;
  long i = 0;
  if (i >= n) goto Lend;
Lstart:
  x = x / 2 + c / (2 * x);
  i++;
  if (i < n) goto Lstart;
Lend:
  return x;
}
```

# ステップ 2 — 式を C = A op B に

```
double sq(double c, long n) {
  double x = c;
  long i = 0;
  if (i >= n) goto Lend;
Lstart:
  x = x / 2 + c / (2 * x);
  i++;
  if (i < n) goto Lstart;
Lend:
  return x;
}
```

⇒

```
double sq3(double c, long n) {
  double x = c;
  long i = 0;
  if (!(i < n)) goto Lend;
Lstart:
  double t0 = 2;
  double t1 = x / t0;
  double t2 = t0 * x;
  double t3 = c / t2;
  x = t1 + t3;
  i = i + 1;
  if (i < n) goto Lstart;
 Lend:
  return x;
}
```

# Step 2 — flatten all nested expressions to “C = A op B”

```
double sq(double c, long n) {
  double x = c;
  long i = 0;
  if (i >= n) goto Lend;
Lstart:
  x = x / 2 + c / (2 * x);
  i++;
  if (i < n) goto Lstart;
Lend:
  return x;
}
```

⇒

```
double sq3(double c, long n) {
  double x = c;
  long i = 0;
  if (!(i < n)) goto Lend;
Lstart:
  double t0 = 2;
  double t1 = x / t0;
  double t2 = t0 * x;
  double t3 = c / t2;
  x = t1 + t3;
  i = i + 1;
  if (i < n) goto Lstart;
 Lend:
  return x;
}
```

# 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/ cannot embed floating point constants in instructions

```
/* c : xmm0, n : rdi */
double sq3(double c, long n) {
  double x = c;        /* x : xmm1 */
  long i = 0;          /* i : rsi */
  if (!(i < n)) goto Lend;
 Lstart:
  double t0 = 2;       /* t0 : xmm2 */
  double t1 = x / t0; /* t1 : xmm3 */
  double t2 = t0 * x; /* t2 : xmm4 */
  double t3 = c / t2; /* t3 : xmm5 */
  x = t1 + t3;
  i = i + 1;
  if (i < n) goto Lstart;
 Lend:
  return x;
}
```

# Step 3 —assign "machine variables" (registers or memory) to variables

- note: cannot write floating point constants in instructions

```
/* c : xmm0, n : rdi */
double sq3(double c, long n) {
  double x = c;       /* x : xmm1 */
  long i = 0;         /* i : rsi */
  if (!(i < n)) goto Lend;
 Lstart:
  double t0 = 2;      /* t0 : xmm2 */
  double t1 = x / t0; /* t1 : xmm3 */
  double t2 = t0 * x; /* t2 : xmm4 */
  double t3 = c / t2; /* t3 : xmm5 */
  x = t1 + t3;
  i = i + 1;
  if (i < n) goto Lstart;
 Lend:
  return x;
}
```

# ステップ4 — 命令に変換

```
/* c : xmm0, n : rdi */
double sq3(double c, long n) {
 # double x = c;       /*x:xmm1*/
 movasd %xmm0,%xmm1
 # long i = 0;         /*i:rsi*/
 movq $0,%rsi
.Lstart:
 # if (!(i < n)) goto Lend;
 cmpq %rdi,%rsi  # n - i
 jle .Lend
 # double t0 = 2;      /*t0:xmm2*/
 movasd .L2(%rip),%xmm2
 # double t1 = x / t0; /*t1:xmm3*/
 movasd %xmm1,%xmm3
 divq %xmm2,%xmm3
 # double t2 = t0 * x; /*t2:xmm4*/
 movasd %xmm0,%xmm4
 mulsd xmm2,%xmm4
```

```
 # double t3 = c/t2; /*t3:xmm5*/
 movasd %xmm0,%xmm5
 divsd %xmm4,%xmm5
 # x = t1 + t3;
 movasd %xmm3,%xmm1
 addsd %xmm5,%xmm1
 # i = i + 1;
 addq $1,%rsi
 # if (i < n) goto Lstart;
 cmpq %rdi,%rsi  # n - i
 jl .Lstart
.Lend:
 # return x;
 movq %xmm1,%xmm0
 ret
}
```

# Step 4 — convert them to machine instructions

```
/* c : xmm0, n : rdi */
double sq3(double c, long n) {
 # double x = c;         /*x:xmm1*/
 movasd %xmm0,%xmm1
 # long i = 0;           /*i:rsi*/
 movq $0,%rsi
.Lstart:
 # if (!(i < n)) goto Lend;
 cmpq %rdi,%rsi  # n - i
 jle .Lend
 # double t0 = 2;        /*t0:xmm2*/
 movasd .L2(%rip),%xmm2
 # double t1 = x / t0; /*t1:xmm3*/
 movasd %xmm1,%xmm3
 divq %xmm2,%xmm3
 # double t2 = t0 * x; /*t2:xmm4*/
 movasd %xmm0,%xmm4
 mulsd xmm2,%xmm4
```

```
 # double t3 = c/t2; /*t3:xmm5*/
 movasd %xmm0,%xmm5
 divsd %xmm4,%xmm5
 # x = t1 + t3;
 movasd %xmm3,%xmm1
 addsd %xmm5,%xmm1
 # i = i + 1;
 addq $1,%rsi
 # if (i < n) goto Lstart;
 cmpq %rdi,%rsi  # n - i
 jl .Lstart
.Lend:
 # return x;
 movq %xmm1,%xmm0
 ret
}
```

# Contents

# 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 . . .

```
  double x = c;       /* x : xmm1 */
Lstart:
  if (!(i < n)) goto Lend;
  double t0 = 2;      /* t0 : xmm2 */
  double t1 = x / t0; /* t1 : xmm3 */
  double t2 = t0 * x; /* t2 : xmm4 */
  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

# Things are more complex in general . . .

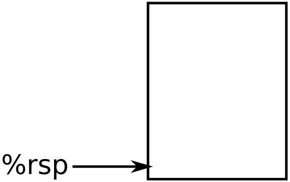
- we’ve liberally assign registers to intermediate results, but . . .

```
double x = c;       /* x : xmm1 */
Lstart:
if (!(i < n)) goto Lend;
double t0 = 2;      /* t0 : xmm2 */
double t1 = x / t0; /* t1 : xmm3 */
double t2 = t0 * x; /* t2 : xmm4 */
double t3 = c / t2; /* t3 : xmm5 */
```

- registers are finite (may run out)
- some registers are destroyed (i.e., values on them are lost) across a function call
- some instructions demand operands to be on specific registers (e.g., dividend of integer division must be on `rax` and `rdx` ≡ `rax` and `rdx` are destroyed across an integer division)
- → you must use memory (“stack” region) as well

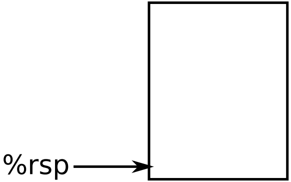
# 最も単純だが一般的な (コンパイラによる) コード生成の作戦

- 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



# A simplest general strategy for code generation by a compiler

- in general, memory (stack) must be used to hold intermediate results ⇒ simply, *"always"* use stack
- a register is used only "temporarily" (to read an operand from memory, which is immediately used by an instruction)



# レジスタ使用慣例 (ABI)

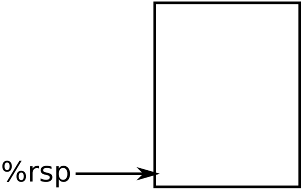
- 整数/ポインタの第1-6引数: rdi, rsi, rdx, rcx, r8, r9
- 浮動小数点数の引数は, xmm0, xmm1, . . .
- 整数/ポインタの返り値：`rax`
- rsp：関数先頭でスタックの端を指し, そこには戻り番地が格納されている
- callee-save レジスタ: rbx, rbp, r12, r13, r14, r15 (関数呼び出しをまたがって保存 → 呼び出された関数がそれらを使う場合は保存してから使う)
- その他のレジスタは caller-save (関数呼び出しをまたがったら壊れると仮定してコードを生成)
- `https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start` “general-purpose registers” を参照

# Register usage conventions (ABI)

- the first six integer/pointers arguments : `rdi, rsi, rdx, rcx, r8, r9`
- floating point number arguments : `xmm0, xmm1, ...`
- an integer/pointer return value : `rax`
- `rsp` : points the end of the stack upon function entry, which holds the return address
- callee-save registers: `rbx, rbp, r12, r13, r14, r15` (preserved across function calls → a function must save them before using (setting a value to) them)
- other registers are caller-save (a function must assume they are destroyed across function calls)
- see “general-purpose” registers in `https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start`

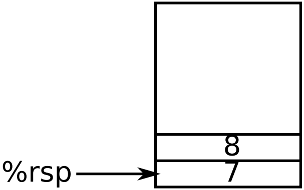
# 関数呼び出し時の動き

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



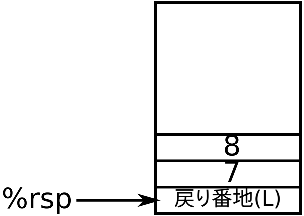
# 関数呼び出し時の動き

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- `call g` 実行直前 rdi=1, rsi=2, rdx=3, rcx=4, r8=5, r9=6



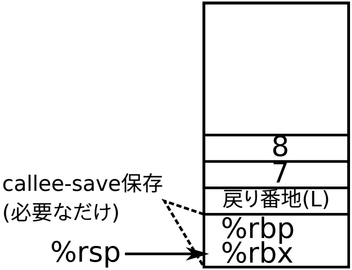
# 関数呼び出し時の動き

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



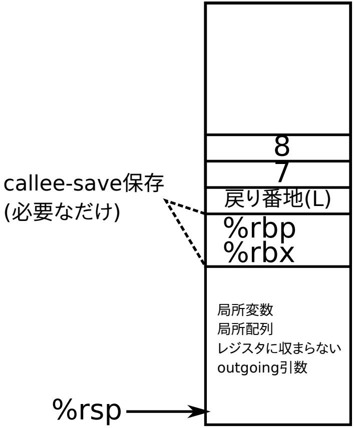
# 関数呼び出し時の動き

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



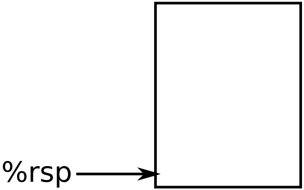
# 関数呼び出し時の動き

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



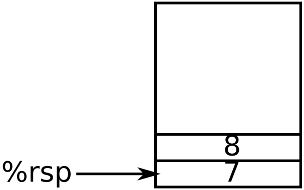
# What happens upon function calls

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



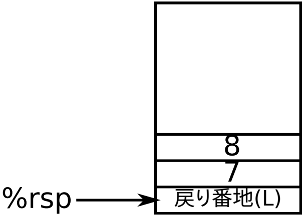
# What happens upon function calls

- `long f() { ...  g(1,2,3,4,5,6,7,8); ... }`
- right before "`call g`" `rdi=1, rsi=2, rdx=3, rcx=4, r8=5, r9=6`



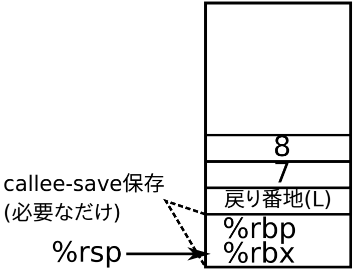
# What happens upon function calls

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- right after "`call g`" (when `g` started)



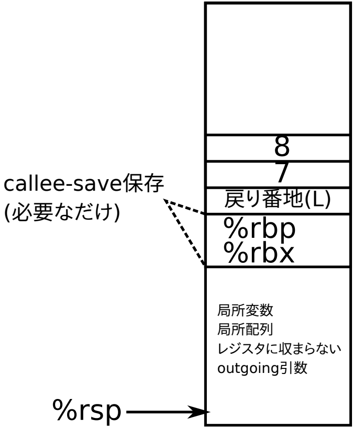
# What happens upon function calls

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- save callee-save registers `g` uses



# What happens upon function calls

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



# 関数呼び出しを含むコード生成例

```
double integ(long n) {
  double x = 0;
  double dx = 1 / (double)n;
  double s = 0;
  for (long i = 0; i < n; i++) {
    s += f(x);
    x += dx;
  }
  return s * dx;
}
```

# Code generation including function calls

```
double integ(long n) {
  double x = 0;
  double dx = 1 / (double)n;
  double s = 0;
  for (long i = 0; i < n; i++) {
    s += f(x);
    x += dx;
  }
  return s * dx;
}
```

# “goto” 化と “C = A op B” 化

```
double integ3(long n) {    /*  n :  0(%rsp) */
  double x = 0;            /*  x :  8(%rsp) */
  double t0 = 1;           /* t0 : 16(%rsp) */
  double t1 = (double)n;   /* t1 : 24(%rsp) */
  double dx = t0 / t1;     /* dx : 32(%rsp) */
  double s = 0;            /*  s : 40(%rsp) */
  long i = 0;              /*  i : 48(%rsp) */
  if (!(i < n)) goto Lend;
 Lstart:
  double t2 = f(x);        /* t2 : 56(%rsp) */
  s += t2;
  x += dx;
  i += 1;
  if (i < n) goto Lstart;
  Lend:
  double t3 = s * dx;      /* t3 : 64(%rsp) */
  return t3;
}
```

# converting to “goto”s and “C = A op B”s

```
double integ3(long n) {    /*  n :  0(%rsp) */
  double x = 0;            /*  x :  8(%rsp) */
  double t0 = 1;           /* t0 : 16(%rsp) */
  double t1 = (double)n;   /* t1 : 24(%rsp) */
  double dx = t0 / t1;     /* dx : 32(%rsp) */
  double s = 0;            /*  s : 40(%rsp) */
  long i = 0;              /*  i : 48(%rsp) */
  if (!(i < n)) goto Lend;
 Lstart:
  double t2 = f(x);        /* t2 : 56(%rsp) */
  s += t2;
  x += dx;
  i += 1;
  if (i < n) goto Lstart;
  Lend:
  double t3 = s * dx;      /* t3 : 64(%rsp) */
  return t3;
}
```

# 機械語 / Machine code

```
double integ3(long n) {
  /* n : 0(%rsp) */
  movq %rdi,0(%rsp)
  # double x = 0;
  /* x : 8(%rsp)*/
  movsd .L0(%rip),%xmm0
  movsd %xmm0,8(%rsp)
  # double t0 = 1;
  /* t0 : 16(%rsp)*/
  movq $1,16(%rsp)
  # double t1 = (double)n;
  /* t1 : 24(%rsp)*/
  cvtsi2sdq 0(%rsp),%xmm0
  movsd %xmm0,24(%rsp)
  # double dx = t0 / t1;
  /* dx : 32(%rsp) */
  movsd 16(%rsp),%xmm0
  divsd 24(%rsp),%xmm0
  movsd %xmm0,32(%rsp)
  # double s = 0;
  /* s : 40(%rsp) */
  movsd .L0(%rip),%xmm0
  movsd %xmm0,40(%rsp)
```

```
  # long i = 0;
  /* i : 48(%rsp) */
  movq $0,48(%rsp)
  # if (!(i < n)) goto Lend;
  movq 0(%rsp),%rdi
  cmpq 48(%rsp),%rdi # n - i
  jle .Lend
.Lstart:
  # double t2 = f(x);
  /* t2 : 56(%rsp) */
  movq 8(%rsp),%rdi
  call f
  movq %rax,56(%rsp)
  # s += t2;
  movq 40(%rsp),%xmm0
  addsd 56(%rsp),%xmm0
  movq %xmm0,40(%rsp)
  # x += dx;
  movsd 8(%rsp),%xmm0
  addsd 32(%rsp),%xmm0
  movsd %xmm0,8(%rsp)
```

# 機械語 / Machine code

```
  # i += 1;
  movq 48(%rsp),%rdi
  addq $1,%rdi
  movq %rdi,48(%rsp)
  # if (i < n) goto Lstart;
  movq 0(%rsp),%rdi
  cmpq 48(%rsp),%rdi # n - i
  jg .Lstart
.Lend:
  movsd 40(%rsp),%xmm0
  addsd 32(%rsp),%xmm0
  addsd %xmm0,64(%rsp)
  # return t3;
  addsd 64(%rsp),%xmm0
  ret
}
```

# Contents

# コード生成器 — 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

# Code generator — scope of the exercise

- all data types are `long (8 byte integers)`
    - no `typedef`s
    - no `int`s, floating point numbers, or pointers
    - everything is long, so type checks are unnecessary
- no global variables ⇒
    - a program = list of function definitions
- supported complex statements are `if`, `while` and compound statement (`{ ... }`) only
- all variable definitions must come at the beginning of a block and initializes (`long` $x$ `=` *expr*) are not supported
- they are expressed in token definitions (`cc_lex.mll`), and grammar definitions (`cc_parse.mly`)

# 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

演習のほとんどの部分は, `cc_cogen.ml` で行われるだろう

# Structure of the program

- `cc_ast.ml` — abstract syntax tree (AST) definition
- `cc_parse.mly` — grammar definition
- `cc_lex.mll` — lexer definition
- `cc_cogen.ml` — code generation from AST
- `cc.ml` — main driver

your work in this exercise will be mostly done in `cc_cogen.ml`

# 構文木定義 (cc_ast.ml) — 関数定義

- C の関数定義の例

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

- ⇒ 関数定義の構文木の定義

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

# AST (`cc_ast.ml`) — function definition

- an example C function definition

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

- ⇒ AST definition for function definition

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

# 構文木の定義 — 文

- if 文

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

⇒

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

- 複合文

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

⇒

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

# AST — statements

- `if` statement

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

⇒

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

- blocks

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

⇒

```
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)
```

# AST — statements

- while 文

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

⇒

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

- ⇒ (putting them together) AST definition for statements

```
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)
```

# AST — expressions

- binary operations

```
  x + y + 1
```

⇒

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

Note: assignment (`a = b`) is a kind of binary operation (C's assignment is not a statement but an expression)

- function call

```
  ... 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

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

# AST — expressions

- ⇒ (putting them together) AST definitions for expressions

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

# コード生成 (`cc_cogen.ml`) — 基本スタイル

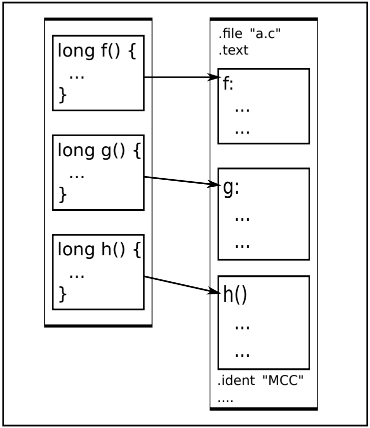
- 構文木 (AST) を受け取り, 対応する機械語 (「命令」のリスト) を返す
- ある構文木に対する機械語の生成 ≈ その構成要素に対する機械語を適切に並べる
- プログラム全体 (program) → 関数定義 (definition) → 文 (stmt) → 式 (expr)
- コード生成器のプログラムの見た目は, 多数の (1) 構文木に対するパターンマッチ (match) と (2) 子ノードに対する再帰呼出し

# Code generation (`cc_cogen.ml`) — basic structure

- takes a parse tree (AST) and returns machine code (a list of instructions)
- generating machine code for an AST ≈ arrange machine code for its components
- the program (`program`) → function definition (`definition`) → statement (`stmt`) → expression (`expr`)
- code generator has lots of (1) pattern matching (match) against AST and (2) recursive calls to child trees

# ファイル全体のコンパイル

- ≈ 関数毎にコンパイルしたものを連結



long f() {
…
}

long g() {
…
}

long h() {
…
}

.file "a.c"
.text

f:
…
…

g:
…
…

h()
…
…

.ident "MCC"
….

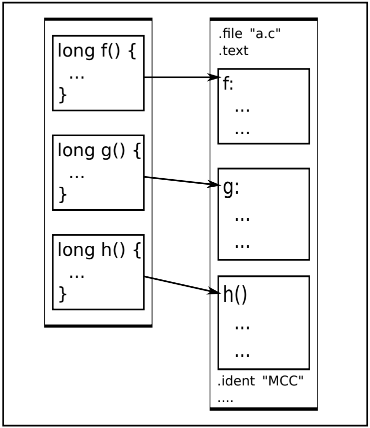
コード概形

```
let cogen_program defs ... =
  (gen_header ...)
  @ List.concat (List.map (fun def ->
          cogen_def def ...) defs)
  (gen_trailer ...)
```

注: あくまで説明用の概形であって, 上記通りの形でなくてよい (こだわってはいけない)

# Compiling an entire file

- ≈ concatenate compilation of individual function definitions



long f() {
…
}

long g() {
…
}

long h() {
…
}

.file "a.c"
.text

f:
…
…

g:
…
…

h()
…
…

.ident "MCC"
....

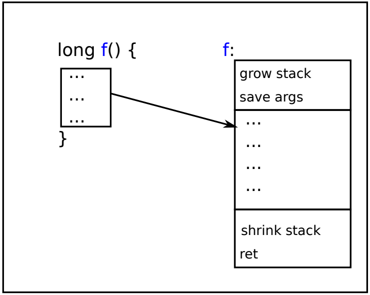
It will look like . . .

```
let cogen_program defs ... =
  (gen_header ...)
  @ List.concat (List.map (fun def ->
          cogen_def def ...) defs)
  (gen_trailer ...)
```

Note: the above is an outline for the illustration purpose you do not have to (should not) stick to

# 関数定義のコンパイル

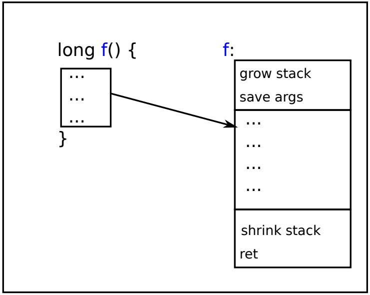
- ≈ 文をコンパイルしたものの前後に, プロローグ (スタックを伸ばす, etc.), エピローグ (スタックを縮める, ret, etc.) をつける



```
let cogen_def def ... =
  match def with FUN_DEF(ret_type, f, params, body) =
     (gen_prologue def)
   @ (cogen_stmt body ...)
   @ (gen_epilogue def)
```

# Compiling a function definition

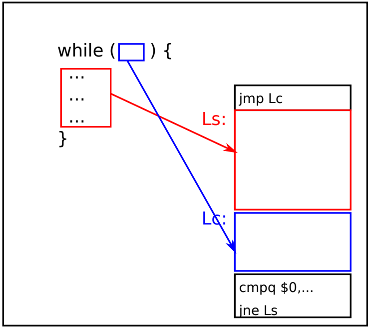
- ≈ compile the body (statement); put prologue (grow the stack, etc.) and epilogue (shrink the stack, ret, etc.)



```
let cogen_def def ... =
  match def with FUN_DEF(ret_type, f, params
          , body) =
     (gen_prologue def)
    @ (cogen_stmt body ...)
    @ (gen_epilogue def)
```

# 文のコンパイル(例: while 文)

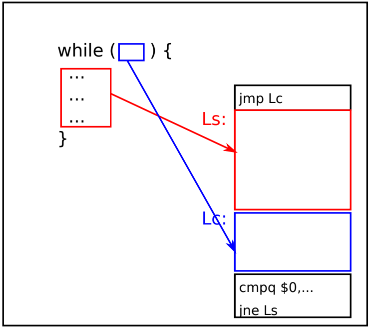
- ≈ 条件式, 本体をコンパイルしたものを以下のように配置. ループの継続判定コードをつける



```
let rec cogen_stmt stmt ...  =
 match stmt with
  ...
 | Cc_ast.STMT_WHILE(cond, body) ->
  let cond_op,cond_insns = cogen_expr cond
     ... in
  let body_insns = cogen_stmt stmt ... in
  let ... in
  [ jmp Lc;
   Ls ]
  @ body_insns
  [ Lc ]
  @ cond_insns @
  [ cmpq $0,cond_op;
   jne Ls ]
```

# Compiling a statement (e.g., `while` statement)

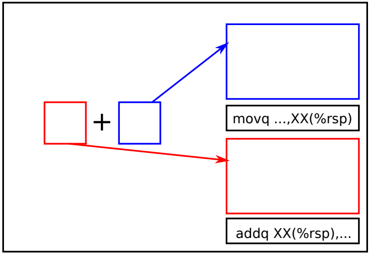
- ≈ place compilation of the condition expression and the body as follows. add a conditional to determine if the loop continues



```
let rec cogen_stmt stmt ...  =
  match stmt with
    ...
   | Cc_ast.STMT_WHILE(cond, body) ->
     let cond_op,cond_insns = cogen_expr cond
            ... in
     let body_insns = cogen_stmt stmt ... in
     let ... in
     [ jmp Lc;
       Ls ]
     @ body_insns
     [ Lc ]
     @ cond_insns @
     [ cmpq $0,cond_op;
       jne Ls ]
```

# 式のコンパイル(算術演算)

- ≈ 引数をそれぞれコンパイル; 演算命令

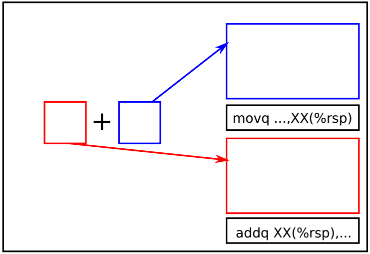


```
let rec cogen_expr expr ... =
  match expr with
    ...
  | Cc_ast.EXPR_BIN_OP(op, e0, e1) ->
    let insns1,op1 = cogen_expr e1 ... in
    let insns0,op0 = cogen_expr e0 ... in
    let m = スタック上のスロット in
    ((insns1
    @ [ movq op1,m ]
    @ insns0
    @ [ op m,op0 ]), (* op0 = op0 op m *)
    op0)
  | ...
```

- 注: `movq XX(%rsp),...` は第一オペランドの結果をスタックに格納し, 第二オペランドの評価中に壊されることがないようにしている
- 最も単純な方式 = 「すべての中間結果をスタックに格納する」に沿った方式

# Compiling an expression (arithmetic)

- ≈ compile the arguments; an arithmetic instruction



```
let rec cogen_expr expr ... =
  match expr with
    ...
  | Cc_ast.EXPR_BIN_OP(op, e0, e1) ->
    let insns1,op1 = cogen_expr e1 ... in
    let insns0,op0 = cogen_expr e0 ... in
    let m = a slot on the stack in
    ((insns1
    @ [ movq op1,m ]
    @ insns0
    @ [ op m,op0 ]), (* op0 = op0 op m *)
    op0)
  | ...
```

- Remark: `movq XX(%rsp),...` saves the first operand, ensuring it won't be destroyed during the evaluation of the second
- remember we are following the simplest strategy = "save all intermediate results on the stack"

# 式のコンパイル (比較演算)

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

  という値を持つ式
- 式が許される任意の場所に現れうることに注意
  - `z = x < y`, `(x < y) + z`, `f(x < 1)` 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)
- アセンブリ言語でこれを生成する命令は?
  1. 条件分岐
  2. 条件つき set 命令. 例:

```
movq $0,%rax
cmpq %rdi,%rsi
setle %al
```

で, `%rsi - %rdi` $\leq 0$ (less-than-or-equal) ならば, `%al` (`%rax` の下 8 bit) に 1 がセットされる

# Compiling an expression (comparison)

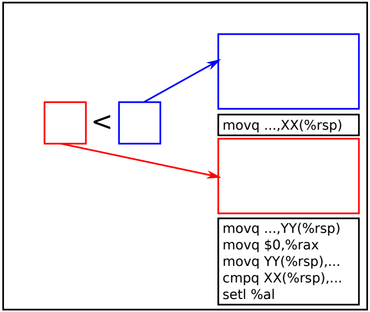
- $A$ `<` $B$ is an expression that evaluates to
  - 1 if $A$ `<` $B$
  - 0 if $A$ `>=` $B$
- no single instruction exactly does this
- note that they can appear anywhere expression can
  - `z = x < y`, `(x < y) + z`, and `f(x < 1)` are allowed (they do not necessarily appear in condition expression of `if` or `while`)
- how to do it in assembly code?
  1. conditional branch
  2. *conditional set instruction*. e.g.,

```
movq $0,%rax
cmpq %rdi,%rsi
setle %al
```

will set `%al` (the lowest 8 bits of `%rax`) to 1 when `%rsi - %rdi` $\leq 0$ (less-than-or-equal)

# 式のコンパイル (比較演算)

- ≈ < の引数をそれぞれコンパイル; 比較; 条件付き set



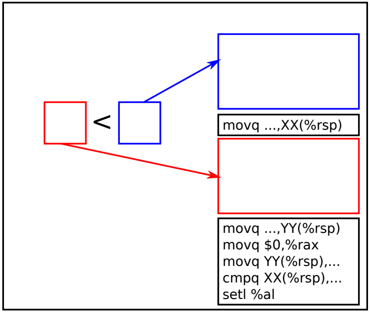
```
movq ...,XX(%rsp)
```

```
movq ...,YY(%rsp)
movq $0,%rax
movq YY(%rsp),...
cmpq XX(%rsp),...
setl %al
```

```
let rec cogen_expr expr ... =
  match expr with
    ...
  | Cc_ast.EXPR_CMP_OP(op, e0, e1) ->
    let insns1,op1 = cogen_expr e1 ... in
    let insns0,op0 = cogen_expr e0 ... in
    let m0 = スタック上のスロット in
    let m1 = スタック上のスロット in
      ...
    ((insns1
      @ [ movq op1,m1 ]
      @ insns0
      @ [ movq op0,m0;
          movq $0,%rax;
          movq m0,op0;
          cmpq m1,op0;
          setop rax ]
    op0)
  | ...
```

# Compiling an expression (comparison)

- ≈ compile the arguments; compare; conditional set



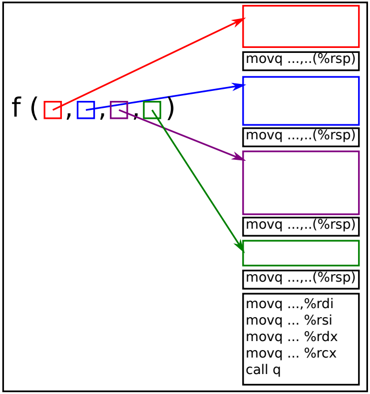
```
movq ...,XX(%rsp)
```

```
movq ...,YY(%rsp)
movq $0,%rax
movq YY(%rsp),...
cmpq XX(%rsp),...
setl %al
```

```
let rec cogen_expr expr ... =
  match expr with
    ...
   | Cc_ast.EXPR_CMP_OP(op, e0, e1) ->
    let insns1,op1 = cogen_expr e1 ... in
    let insns0,op0 = cogen_expr e0 ... in
    let m0 = a slot on the stack in
    let m1 = a slot on the stack in
      ...
    ((insns1
      @ [ movq op1,m1 ]
      @ insns0
      @ [ movq op0,m0;
          movq $0,%rax;
          movq m0,op0;
          cmpq m1,op0;
          setop rax ]
    op0)
   | ...
```

# 式のコンパイル (関数呼び出し)

- ≈ 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



f (□,□,□,□)

movq ...,..(%rsp)

movq ...,..(%rsp)

movq ...,..(%rsp)

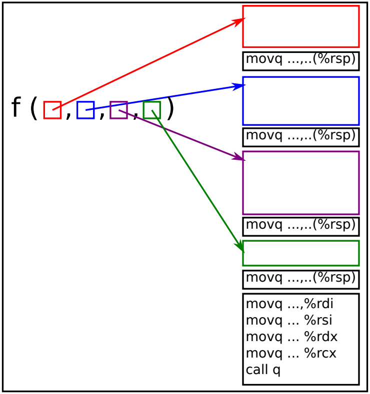
movq ...,..(%rsp)

movq ...,%rdi
movq ... %rsi
movq ... %rdx
movq ... %rcx
call q

```
let rec cogen_expr expr ... =
  match expr with
    ...
  | Cc_ast.EXPR_CALL(f, args) ->
     let insns,arg_vars = cogen_exprs args
          env var_idx in
     ((insns @ (make_call f arg_vars)), rax)
```

# Compiling an expression (function call)

- ≈ compile all arguments; put them to positions specified by ABI; a `call` instruction



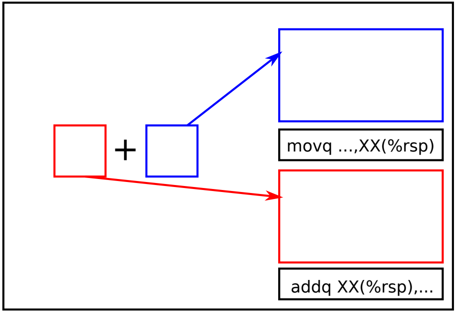
f (□,□,□,□)

```
movq ...,..(%rsp)
movq ...,..(%rsp)
movq ...,..(%rsp)
movq ...,..(%rsp)
movq ...,%rdi
movq ... %rsi
movq ... %rdx
movq ... %rcx
call q
```

```
let rec cogen_expr expr ... =
  match expr with
    ...
  | Cc_ast.EXPR_CALL(f, args) ->
     let insns,arg_vars = cogen_exprs
         args env var_idx in
     ((insns @ (make_call f arg_vars)),
         rax)
```

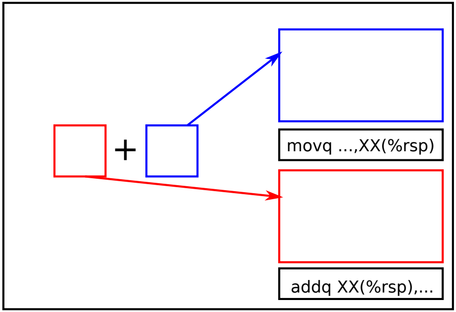
# 見過ごしてきた詳細

- 「部分式の値」や「変数の値」を保存しておく場所の決め方
- 以下の XX の決め方



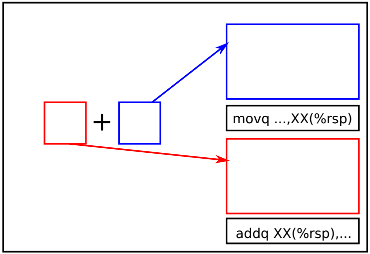
# Details we have been leaving out

- how to determine locations to save values of *subexpressions* and *variables*
- that is, how to determine XX below



# 「部分式の値」の格納場所の決め方

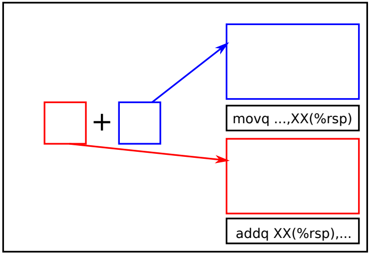
- cogen_expr に, 空き領域の先頭を示す引数 ($v$) を渡す
  意味: (cogen_expr $E$ $v$ ...) は, $E$ を評価する命令列を生成; それは (%rsp + $v$) 以上のアドレスのみを使う (破壊する)
- → $\boxed{A}$ + $\boxed{B}$ の命令列中で, $\boxed{B}$ の結果は $v$(%rsp) に保存
- $\boxed{A}$ は $v + 8$ 以降を使う



```
let rec cogen_expr expr v =
  match expr with
    ...
  | Cc_ast.EXPR_BIN_OP(op, e0, e1) ->
    let insns1,op1 = cogen_expr e1 v ... in
    let insns0,op0 = cogen_expr e0 (v + 8)
          ... in
    let m = v(%rsp) in
    ((insns1
    @ [ movq op1,m ]
    @ insns0
    @ [ op m,op0 ]), (* op0 = op0 op m *)
    op0)
  | ...
```

# Determining where to save subexpressions

- `cogen_expr` receives a value ($v$) pointing to the free space spec: `cogen_expr` $E$ $v$ … generates instructions that evaluate $E$ using (destroying) only addresses above (`%rsp + `$v$)
- → when evaluating $\boxed{A} + \boxed{B}$, save $\boxed{B}$ at $v$(`%rsp`)
- let $\boxed{A}$ use $v + 8$ and higher addresses



```
let rec cogen_expr expr v =
  match expr with
    ...
  | Cc_ast.EXPR_BIN_OP(op, e0, e1) ->
    let insns1,op1 = cogen_expr e1 v ... in
    let insns0,op0 = cogen_expr e0 (v + 8)
          ... in
    let m = v(%rsp) in
    ((insns1
    @ [ movq op1,m ]
    @ insns0
    @ [ op m,op0 ]), (* op0 = op0 op m *)
    op0)
  | ...
```

# 「変数の値」の格納場所

- 例:

```
if (...) {
  long a, b, c;
  ...
}
```

- 変数 `a, b, c` をスタック上に格納する必要がある
- 部分式の保存場所とほぼ同じ問題
- → `cogen_stmt` にも, 空き領域の先頭を示す引数 $v$ を渡す

  意味: `cogen_stmt` $S$ $v$ $\ldots$ は, $S$ を実行する命令列を生成; それは (`%rsp + ` $v$) 以上のアドレスのみを使う (破壊する)
- → 例えば `a` $\mapsto v$(`%rsp`), `b` $\mapsto v + 8$(`%rsp`), `c` $\mapsto v + 16$(`%rsp`) に格納

# Locations to hold variables

- ex:

```
if (...) {
  long a, b, c;
  ...
}
```

- we need to hold `a, b, c` on the stack
- the problem is almost identical to saving values of subexpressions
- → `cogen_stmt` also takes $v$ pointing to the beginning of the free space

  spec: `cogen_stmt` $S$ $v$ …generates instructions to execute $S$; they use (destroy) only addresses above (`%rsp +` $v$)
- → e.g., hold `a` $\mapsto v$`(%rsp)`, `b` $\mapsto v + 8$`(%rsp)`, `c` $\mapsto v + 16$`(%rsp)`

# 環境: 変数の格納場所の情報

- 「変数の値の格納場所」は, 式に変数が出現した際にそれを取り出せる必要がある
  - 例: `x + 1` をコンパイルするには, `x` の格納場所を知る必要
- 「変数 $\mapsto$ 格納場所」の写像を管理するデータ構造 (環境) を作り, `cogen_stmt`, `cogen_expr` はそれを受け取るようにする
- 複合文 ({ . . . }) の先頭で変数宣言が行われた時に, 環境に新たな写像を追加
- 環境は, 連想リスト (`List.assoc`) を用いて簡単に作れる

# Environment: records where variables are held

- when a variable occurs in an expression, we need to get the location that holds the variable
  - ex: to compile `x + 1`, we need to know where `x` is held
- make a data structure that holds a mapping "variable ↦ location" *(environment)* and pass it to `cogen_stmt` and `cogen_expr`
- when new variables are declared at the beginning of a compound statement ($\{ \dots \}$), add new mappings to it
- an environment can be easily built with association list (`List.assoc`)

# cogen_expr は環境を受け取る

```
let rec cogen_expr expr env v =
  match expr with
    ...
  | Cc_ast.EXPR_VAR(x) ->
     let loc = env_lookup x env in
     ([ movq loc,... ], ...)
  | ...
```

- `env_lookup` $x$ $env$ は, 環境 $env$ 中から $x$ の格納位置を探す

# `cogen_expr` receives an environment

```
let rec cogen_expr expr env v =
  match expr with
    ...
  | Cc_ast.EXPR_VAR(x) ->
     let loc = env_lookup x env in
     ([ movq loc,... ], ...)
  | ...
```

- `env_lookup` *x env* searches environment *env* for *x* and returns its location

# cogen_stmt も環境を受け取る

```
let rec cogen_stmt expr env v =
  match expr with
    ...
  | Cc_ast.STMT_COMPOUND(decls, stmts) ->
     let env’,v’ = env_extend decls env v in
     cogen_stmts stmts env’ v’ ...
  | ...
```

- `env_extend` *decls env v* は,
  - 変数宣言 *decls* で宣言された変数に格納場所を割り当て $(v, v + 8, v + 16, \ldots)$,
  - それらを環境 *env* に登録
  - 新しい環境 *env'* と空き領域 *v'* を返す

# `cogen_stmt` receives an environment too

```
let rec cogen_stmt expr env v =
  match expr with
    ...
  | Cc_ast.STMT_COMPOUND(decls, stmts) ->
     let env',v' = env_extend decls env v in
     cogen_stmts stmts env' v' ...
  | ...
```

- `env_extend` *decls env v*
  - assign locations ($v$, $v + 8$, $v + 16$, $\ldots$) to variables declared in *decls*
  - register them in *env*
  - return the new environment *env'* and the new free space *v'*

# 環境の実装

- 環境は (変数名, 格納場所) のリスト
- *loc* `=` `env_lookup` *x* *env*

  環境 *env* において, 変数 *x* の格納場所 *loc* を返す (cf. `List.assoc`)
- *env'* `=` `env_add` *x* *loc* *env*

  *env* に, $x \mapsto loc$ が追加された環境 *env'* を返す (`(`*x* `,` *loc*`)::`*env*)
- これを元に `env_extend` *decls* *env* *v* を作るのは演習問題

# Implementing environment

- an environment is a list of (variable name, location)
- *loc* `=` `env_lookup` $x$ *env*

  returns $x$'s location in environment *env* (cf. `List.assoc`)
- *env'* `=` `env_add` $x$ *loc* *env*

  returns a new environment *env'* which has a new mapping $x \mapsto$ *loc* in addition to *env* (`(`$x$`,` *loc*`)::`*env*)
- implementing `env_extend` *decls* *env* $v$ based on this is your exercise

# Contents

# 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 |

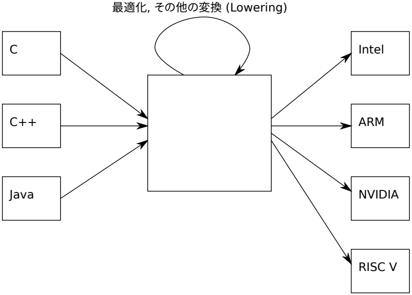
# Intermediate representation (IR)

- while it is in theory possible to directly emit assembly code from a pair (AST, environment), most compilers first emit "Intermediate Representation; IR" which are similar to but different from the assembly code
- IR vs. assembly

| | IR | assembly |
|---|---|---|
| control | ≈ go to | ≈ go to |
| expression | ≈ C = A op B only | ≈ C = A op B only |
| the number of local variables | arbitrary | ≈ only a fixed number of registers |
| lifetime of a local variable | during the function call that defined it | ≈ some registers live only up to the next function call |

# 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



# Why IR?

- simplify program → IR conversion (allow arbitrary number of local variables, live across function calls)
- make it easy to implement multiple input languages — C, C++, Java, etc. can share everything but input → IR
- make it easy to target multiple processors — Intel, ARM, etc. can share everything but IR → machine code
- optimization — represent various optimizations as “IR → IR” or “IR → machine code” transformation