Programming Language (10)
Making a compiler

田浦

Contents

Contents

From high-level programming languages to machine code

- there are *no structured control flows* (for, while, if, etc.); everything must be done by (conditional) jump instructions (≈ "goto" statement)
- an instruction can perform *only a single operation*, so nested expressions (e.g., `a * x + b * y + c * z`) must be broken down into a series of instructions
- a register ≈ a variable, but
  - you have *only a fixed number of them*, so some values may have to be spilled on memory (esp. at function calls)
  - function parameters and return values are on predetermined registers (*calling convention* or *Application Binary Interface*)

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;
}
```

Step 1 — make all controls "goto"s

```
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  }
```

$\Rightarrow$

```
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  }
```

Step 2 — flatten all nested expressions to "C = A op B"

```
 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 | }
```

$\Rightarrow$

```
 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 | }
```

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

- note: cannot write floating point constants in instructions

```
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  }
```

Step 4 — convert them to machine instructions

```
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 }
```

Things are more complex in general ...

- we've liberally assigned registers to intermediate results, but ...

```
1    double x = c;        /* x : xmm1 */
2    Lstart:
3    if (!(i < n)) goto Lend;
4    double t0 = 2;        /* t0 : xmm2 */
5    double t1 = x / t0; /* t1 : xmm3 */
6    double t2 = t0 * x; /* t2 : xmm4 */
7    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` $\equiv$ `rax` and `rdx` are destroyed across an integer division)
- $\rightarrow$ you must use memory ("stack" region) as well

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`
- a floating point number return value : `xmm0`
- `rsp` : points the end of the stack upon function entry, which holds the return address
- *callee-save* registers: `rbx, rbp, r12, r13, r14, r15, rsp` (preserved across function calls $\rightarrow$ 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`
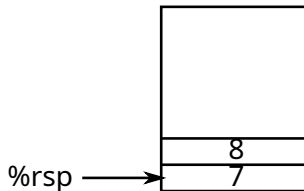
ABI (How function call works)

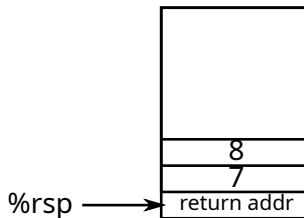- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- during f

%rsp
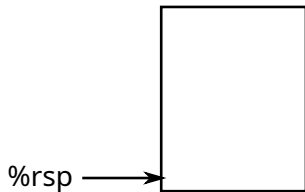
ABI (How function call works)

- 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
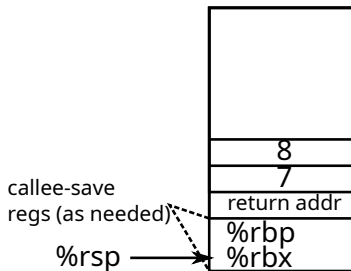
ABI (How function call works)

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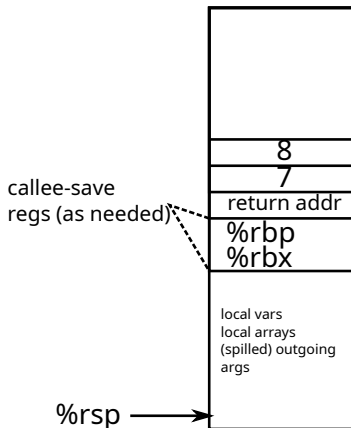- right after "`call g`" (when g started)

ABI (How function call works)

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

ABI (How function call works)

- `long f() { ... g(1,2,3,4,5,6,7,8); ... }`
- extend stack as needed to execute g

A simplest general strategy for code generation

- in general,
  - there may be too many intermediate results to hold on registers
  - values used after a function call must be saved on memory (or callee-save registers)
  - ⇒ *"always"* using memory (stack) is the simplest strategy
- a register is used only "temporarily" to apply an instruction

%rsp ⟶ □
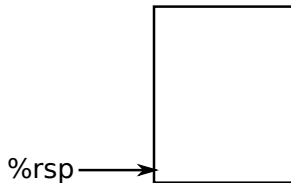
A simple code generation including function calls

```
 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  }
```

converting to "goto"s and "C = A op B"s

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

allocate memory slot for intermediate values

```
 1  double integ(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 / Machine code

```
1  double integ(long n) {
2    subq $72,%rsp
3    /* n : 0(%rsp) */
4    movq %rdi,0(%rsp)
5    # double x = 0;
6    /* x : 8(%rsp)*/
7    movsd .L0(%rip),%xmm0
8    movsd %xmm0,8(%rsp)
9    # double t0 = 1;
10   /* t0 : 16(%rsp)*/
11   movsd .L1(%rip), %xmm0
12   movsd %xmm0,16(%rsp)
13   # double t1 = (double)n;
14   /* t1 : 24(%rsp)*/
15   cvtsi2sdq 0(%rsp),%xmm0
16   movsd %xmm0,24(%rsp)
17   # double dx = t0 / t1;
18   /* dx : 32(%rsp) */
19   movsd 16(%rsp),%xmm0
20   divsd 24(%rsp),%xmm0
21   movsd %xmm0,32(%rsp)
22   # double s = 0;
23   /* s : 40(%rsp) */
```

```
1    movsd .L0(%rip),%xmm0
2    movsd %xmm0,40(%rsp)
3    # long i = 0;
4    /* i : 48(%rsp) */
5    movq $0,48(%rsp)
6    # if (!(i < n)) goto Lend;
7    movq 0(%rsp),%rdi
8    cmpq 48(%rsp),%rdi # n - i
9    jle .Lend
10  .Lstart:
11   # double t2 = f(x);
12   /* t2 : 56(%rsp) */
13   movq 8(%rsp),%rdi
14   call f
15   movq %rax,56(%rsp)
16   # s += t2;
17   movq 40(%rsp),%xmm0
18   addsd 56(%rsp),%xmm0
19   movq %xmm0,40(%rsp)
20   # x += dx;
21   movsd 8(%rsp),%xmm0
22   addsd 32(%rsp),%xmm0
23   movsd %xmm0,8(%rsp)
```

機械語 / Machine code

```
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 }
```

Contents

MinC ("Minimum C") spec overview

- this will be your final report if you choose option A
- all expressions have type `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
- function calls with C conventions, so you can call or be called by C functions compiled by ordinary compilers (e.g., gcc)
- supported complex statements are `if, while` and compound statement (`{ ... }`) only

# Structure of compilers

- Abstract Syntax Tree (AST) : data structure representing the program
- Intermediate Representatin (IR) : common representation portable across multiple source/target languages
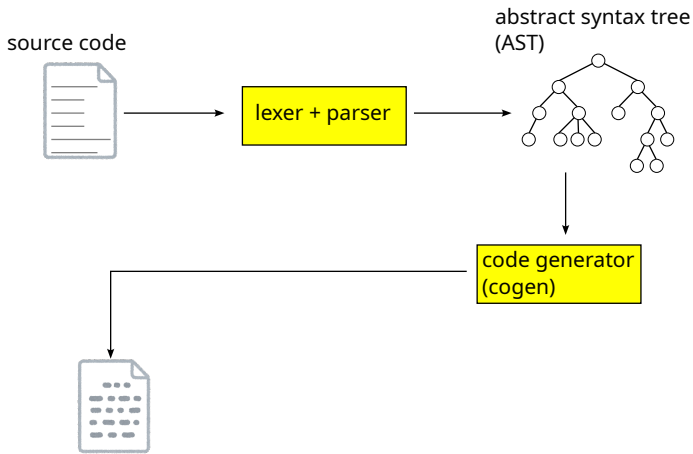
Structure of compilers

- Abstract Syntax Tree (AST) : data structure representing the program
- Intermediate Representatin (IR) : common representation portable across multiple source/target languages
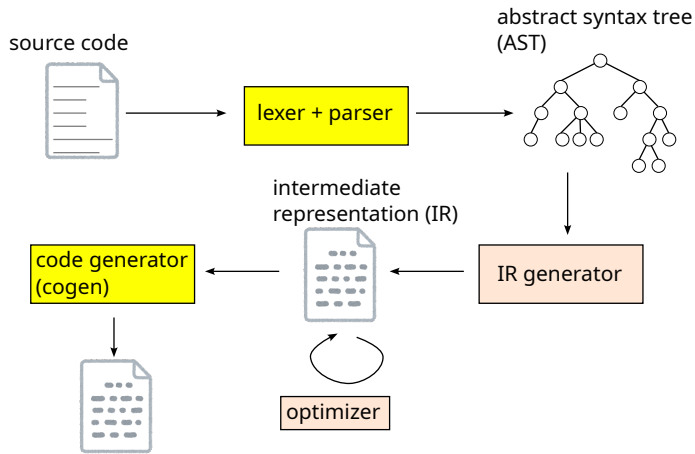
Structure of the program

- `parser/`
  - `minc_grammar.y` — grammar definition
  - `minc_to_xml.py` — minC → XML converter
- `{ml,jl,go,rs}/minc/`
  - `minc_ast.??` — abstract syntax tree (AST) definition
  - `minc_parse.??` — XML → AST
  - `minc_cogen.??` — AST → assembly
  - `main.??` or `minc.??` — main driver
- the exact location depends on the language
- your work will be mostly done in `minc_cogen.??`
- other parts are given

Abstract Syntax Tree (AST)

- naturally represent a program
  - ▸ the whole program
  - ▸ function definition
  - ▸ statement
  - ▸ expression
  - ▸ etc.
- also called *parse tree*
- see `minc_ast.??`

```
1  while (x < 10)
2    y++;
```

$\Rightarrow$
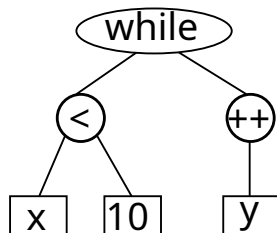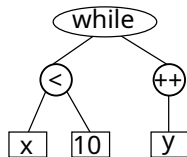
Lexer and parser

- lexer (lexical analyzer, tokenizer) : string → sequence of "tokens" (words)

  ```
  while (x < 10) y++;
  ```
  → | while | (| x | < | 10 |) | y | ++ | ; |

- parser : sequence of tokens ⇒ AST

| while | (| x | < | 10 |) | y | ++ | ; | ⇒

Implementing lexer and parser

- first write a *grammar*, typically in the Backus-Naur form (BNF)
- e.g., (part of C grammar)
  - *statement = while-statement | if-statement | . . .*
  - *while-statement =* `'while'` `'('` *expr* `')'` *statement*
  - *expr = number | expr* `'+'` *expr | . . .*
  - *number = digit+ | digit+ . digit* | . . .*
  - *digit =* [0-9]
  - . . .
- based on the grammar, either
  - write them by hand, or
  - use a *lexser/parser generators*

Lexer/parser generators

- *lexer generator* generates a lexer from a syntax of *tokens* (variables, numbers, . . .)
- *parser generator* generates a parser from a syntax of higher-level constructs (expressions, statements, . . .)
- some grammar frameworks (PEG) specify them in a single framework

Lexer/parser generators

- many programming languages have lexer/parser generators for them
  - ▸ lex/yacc (flex/bison) : C/C++
  - ▸ ANTLR : C, C++, Java, Python, JavaScript, Go, . . .
  - ▸ ocamllex/menhir : OCaml
  - ▸ tatsu : Python
  - ▸ etc.
- this exercise uses tatsu, to generate a Python program that converts C source into XML
  - ▸ the grammar is in `minc_grammar.y`
- each language reads the XML by the respective XML library you have used before

Intermediate Representation (IR)

- a common representation of programs internally used by a compiler
    - hopefully independent from the source language (C, C++, Rust, Go, Julia, etc.)
    - hopefully independent from the target language (x86, ARM, PowerPC, etc.)
- it generally looks like "an assembly with infinite registers (variables)"
- a compiler performs optimizations as IR $\rightarrow$ IR transformations
- *note:* in the exercise you could design your IR, but it is not necessary (it is possible to directly go from AST $\rightarrow$ asm)

Code generation (`minc_cogen`) — basic structure

- takes an AST and returns machine code (a list of instructions)
- generate machine code for an AST $\approx$ generate machine code of its components and properly arrange them
- the program (`program`) $\rightarrow$ function definition (`definition`) $\rightarrow$ statement (`stmt`) $\rightarrow$ expression (`expr`)
- code generator has lots of
  - ▶ case analysis based on the type of the tree; use
    - ⋆ pattern matching (OCaml match and Rust match) or
    - ⋆ polymorphism (OCaml objects, Julia function, Go interface, Rust trait)
  - ▶ recursive calls to child trees

Compiling an entire file

- ≈ concatenate compilation of individual function definitions
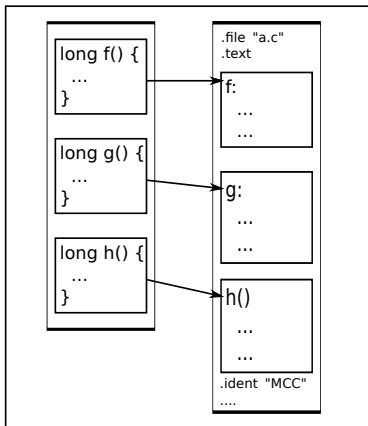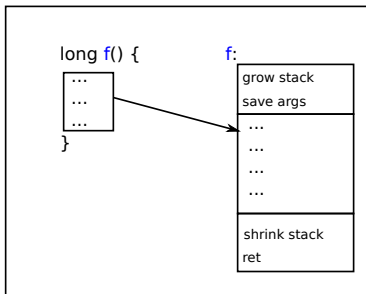

In OCaml, it will look like ...

```
1  let ast_to_insns_program defs ... =
2    List.concat (List.map (fun def ->
           ast_to_insns_def def ...) defs)
```

Compiling a function definition

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


```
1  let ast_to_insns_def def ... =
2    match def with
3      DefFun(f, params, ret_type, body) ->
4        (gen_prologue def)
5        @ (ast_to_insns_stmt body ...)
6        @ (gen_epilogue def)
```

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
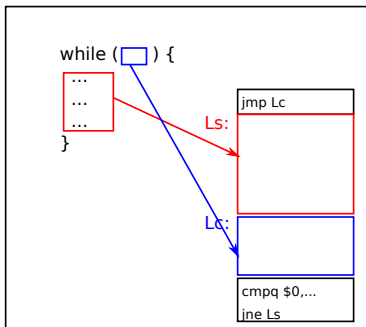

```
1   let rec ast_to_insns_stmt stmt ...  =
2     match stmt with
3       ...
4     | StmtWhile(cond, body) ->
5       let cond_op,cond_insns =
            ast_to_insns_expr cond ... in
6       let body_insns = ast_to_insns_stmt body
            ... in
7       let ... in
8       [ jmp Lc;
9         Ls ]
10      @ body_insns
11      [ Lc ]
12      @ cond_insns @
13      [ cmpq $0,cond_op;
14        jne Ls ]
```

Compiling an expression (arithmetic)

- ≈ compile the arguments; an arithmetic instruction
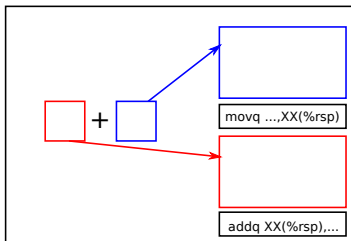

```
1   let rec ast_to_insns_expr expr ... =
2     match expr with
3       ...
4     | ExprOp("+", [e0; e1]) ->
5       let insns1,op1 = ast_insns_expr e1 ... in
6       let insns0,op0 = ast_insns_expr e0 ... in
7       let m = a slot on the stack in
8       ((insns1
9       @ [ movq op1,m ]
10      @ insns0
11      @ [ addq m,op0 ]), (* op0 = op0 + m *)
12      op0)
13    | ...
```

- 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"

Compiling an expression (comparison)

- $A < B$ is an expression that evaluates to
  - ▸ 1 if $A < B$
  - ▸ 0 if $A \geq B$
- 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.,

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

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

# Compiling an expression (comparison)

- ≈ compile the arguments; compare; conditional set
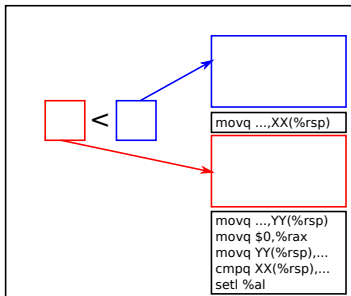

```
1   let rec ast_to_insns_expr expr ... =
2     match expr with
3       ...
4     | ExprOp("<", [e0; e1]) ->
5       let insns1,op1 = ast_to_insns_expr e1 ...
              in
6       let insns0,op0 = ast_to_insns_expr e0 ...
              in
7       let m0 = a slot on the stack in
8       let m1 = a slot on the stack in
9         ...
10      ((insns1
11        @ [ movq op1,m1 ]
12        @ insns0
13        @ [ movq op0,m0;
14            movq $0,%rax;
15            movq m0,op0;
16            cmpq m1,op0;
17            setl rax ]
18      op0)
19    | ...
```

Compiling an expression (function call)

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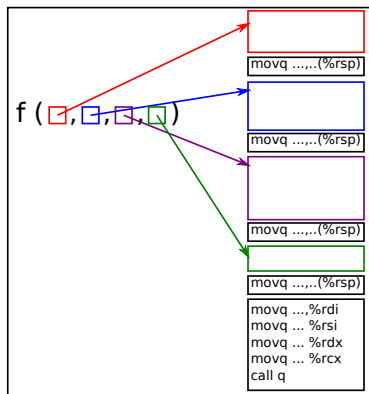

```
1  let rec ast_to_insns_expr expr ... =
2    match expr with
3      ...
4      | ExprCall(f, args) ->
5        let insns,arg_vars =
              ast_to_insns_exprs args env
              var_idx in
6        ((insns @ (make_call f arg_vars)),
              rax)
```

Details we have been leaving out

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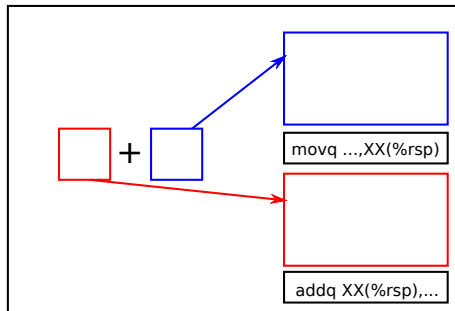
Determining where to save subexpressions

- `ast_to_insns_expr` receives a value (v) pointing to the lowest end of free space
  `ast_to_insns_expr` E v ... generates instructions that evaluate E using (destroying) only addresses above v(`%rsp`)
- $\rightarrow$ when evaluating $\boxed{A} + \boxed{B}$, save $\boxed{B}$ at v(`%rsp`)
- let $\boxed{A}$ use $v + 8$ and higher addresses
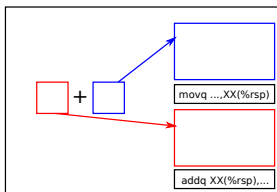
```
1   let rec ast_to_insns_expr expr v =
2    match expr with
3     ...
4     | ExprOp("+", e0, e1) ->
5     let insns1,op1 = ast_to_insns_expr e1 v .. in
6     let insns0,op0 = ast_to_insns_expr e0 (v + 8) .. in
7     let m = v(%rsp) in
8     ((insns1
9     @ [ movq op1,m ]
10    @ insns0
11    @ [ addq m,op0 ]), (* op0 = op0 + m *)
12    op0)
13    | ...
```

Locations to hold variables

- ex:

```
1  if (...) {
2    long a, b, c;
3    ...
4  }
```

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

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

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 $\mapsto$ location" *(environment)* and pass it to `ast_to_insns_stmt` and `ast_to_insns_expr`
- when new variables are declared at the beginning of a compound statement ({ ... }), add new mappings to it

`ast_to_insns_expr` receives an environment

```
1   let rec ast_to_insns_expr expr env v =
2     match expr with
3       ...
4     | ExprId(x) ->
5         let loc = env_lookup x env in
6         ([ movq loc,... ], ...)
7     | ...
```

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

`ast_to_insns_stmt` receives an environment too

```
1  let rec ast_to_insns_stmt expr env v =
2    match expr with
3      ...
4    | StmtCompound(decls, stmts) ->
5        let env',v' = env_extend decls env v in
6        cogen_stmts stmts env' v' ...
7    | ...
```

- 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'*

Implementing environment

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

  returns the location paired with x in environment env
- 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`)
- an environment can be easily implemented with a list of (variable name, location)'s and is left for your exercise