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

```
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;
}
```

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;
}
```

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;
}
```

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
}
```

Things are more complex in general . . .

- we've liberally assigned 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 */
```

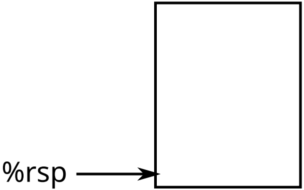
- registers are finite (may run out)
- some registers are destroyed (i.e., values on them are lost) across a function call
- some instructions demand operands to be on specific registers (e.g., dividend of integer division must be on `rax` and `rdx` $\equiv$ `rax` and `rdx` are destroyed across an integer 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 → 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`

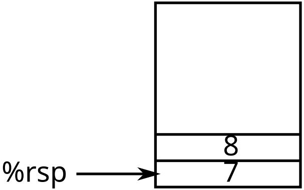
ABI (How function call works)

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



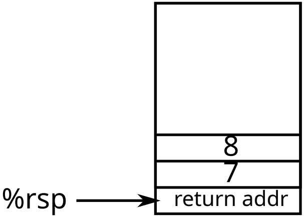
ABI (How function call works)

- `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`



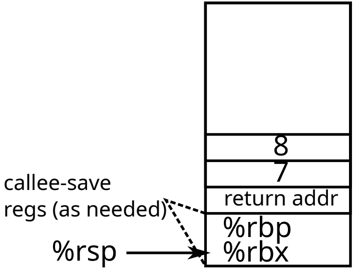
ABI (How function call works)

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



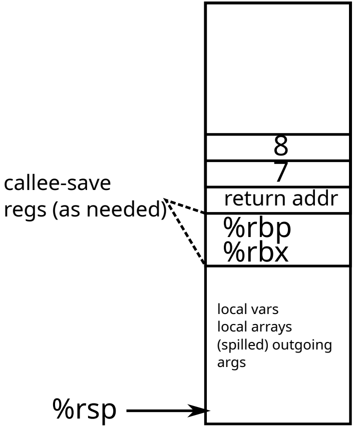
ABI (How function call works)

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



ABI (How function call works)

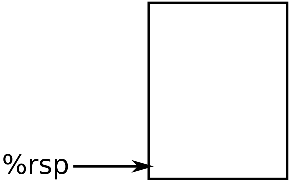
- `long f() { ...  g(1,2,3,4,5,6,7,8); ...  }`
- extend stack as needed to execute `g`



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



A code generation based on the simple strategy

```
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;
}
```

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

```
double integ(long n) {
  double x = 0;
  double t0 = 1;
  double t1 = (double)n;
  double dx = t0 / t1;
  double s = 0;
  long i = 0;
  if (!(i < n)) goto Lend;
 Lstart:
  double t2 = f(x);
  s += t2;
  x += dx;
  i += 1;
  if (i < n) goto Lstart;
  Lend:
  double t3 = s * dx;
  return t3;
}
```

allocate memory slot for intermediate values

```
double integ(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 integ(long n) {
  subq $72,%rsp
  /* 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)*/
  movsd .L1(%rip), %xmm0
  movsd %xmm0,16(%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
}
```

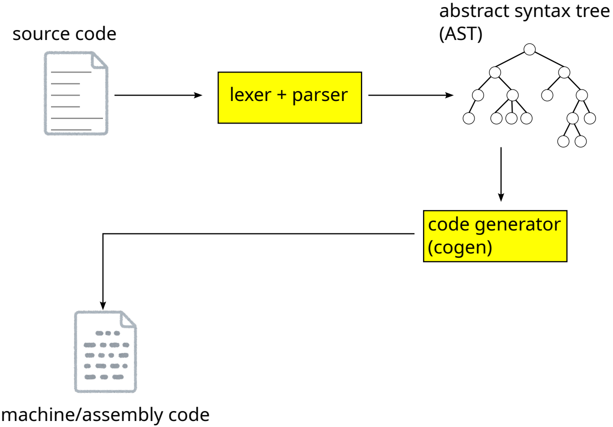
Contents

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

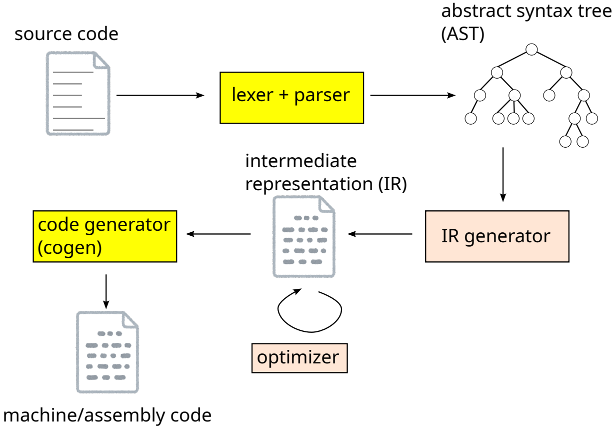
Structure of compilers

- Abstract Syntax Tree (AST) : data structure representing the program



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

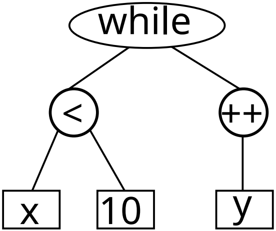
- `parser/`
  - `minc_grammar.y` — grammar definition
  - `minc_to_xml.py` — minC → XML converter
- `{go,jl,ml,rs}/minc/`
  - `minc_ast.??` — abstract syntax tree (AST) definition
  - `minc_parse.??` — XML → AST
  - `minc_cogen.??` — AST → assembly
  - `main.??` or `minc.??` — main driver
- the exact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

Abstract Syntax Tree (AST)

- a data structure that naturally represents a program
  - the whole program,
  - function definition,
  - statement,
  - expression,
  - ...
- also called *parse tree*
- see `minc_ast.??`

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

$\Rightarrow$



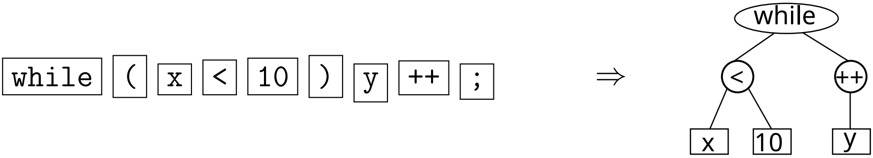
Lexer and parser

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

  "`while (x < 10) y++;`"

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

- parser : sequence of tokens ⇒ AST



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 the definition of *tokens* (variables, numbers, ...)
- *parser generator* generates a parser from the definition 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)

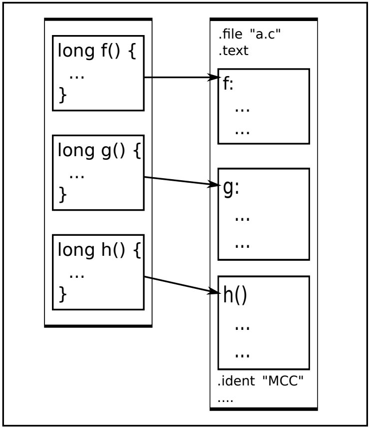
- a common representation of programs used by a compiler
- generally $\approx$ "an assembly with unlimited variables"
- purposes
  1. achieve portability
     - ★ hopefully independent from the source language (C, C++, Rust, Go, Julia, etc.)
     - ★ hopefully independent from the target language (x86, ARM, PowerPC, etc.)
  2. perform 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 ≈ generate machine code of its components and properly arrange them
- the program (`program`) → function definition (`definition`) → statement (`stmt`) → 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

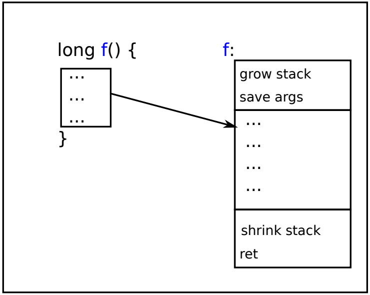


In OCaml, it will look like ...

```
let ast_to_asm_program (Program(defs)) ... =
  ...
  header
  @ List.concat (List.map (fun def ->
          ast_to_asm_def def ...) defs)
  @ trailer
```

Compiling a function definition

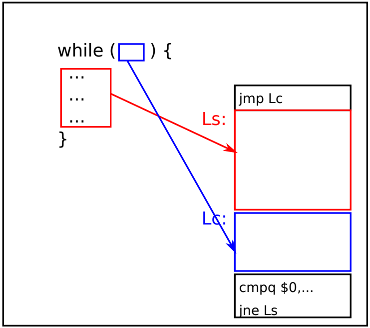
- ≈ compile the body (statement); put prologue (grow the stack, etc.) and epilogue (shrink the stack, ret, etc.)




```
let ast_to_asm_def def ... =
  match def with
    DefFun(f, params, ret_type, body) ->
      (gen_prologue def)
      @ (ast_to_asm_stmt body ...)
      @ (gen_epilogue def)
```

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

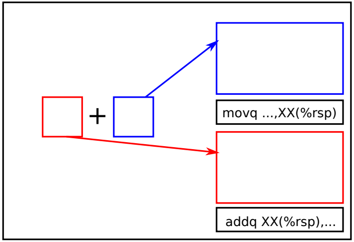
- ≈ place compilation of the condition expression and the body as follows. add a conditional to determine if the loop continues




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

Compiling an expression (arithmetic)

- ≈ compile the arguments; an arithmetic instruction




```
let rec ast_to_asm_expr expr ... =
  match expr with
    ...
  | ExprOp("+", [e0; e1]) ->
    let insns1,op1 = ast_to_asm_expr e1 ... in
    let insns0,op0 = ast_to_asm_expr e0 ... in
    let m = a slot on the stack in
    ((insns1
    @ [ movq op1,m ]
    @ insns0
    @ [ addq m,op0 ]), (* op0 = op0 + 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"

Compiling an expression (comparison)

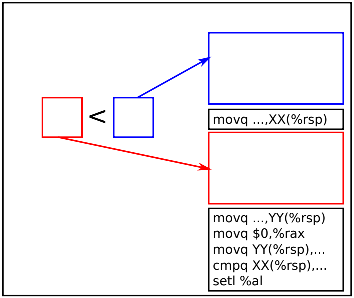
- A `<` B is an expression that evaluates to
  - 1 if A `<` B
  - 0 if A `>=` B
- no single instruction exactly does this
- note that they can appear anywhere expression can
  - `z = x < y`, `(x < y) + z`, and `f(x < 1)` are allowed (they do not necessarily appear in condition expression of `if` or `while`)
- 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` ≤ 0 (less-than-or-equal)

Compiling an expression (comparison)

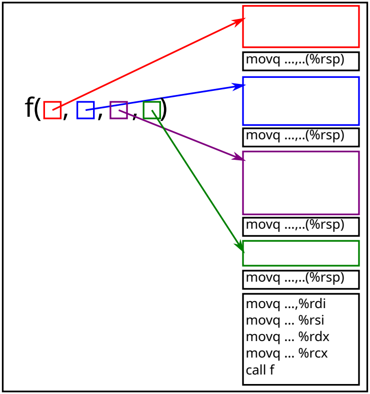
- ≈ compile the arguments; compare; conditional set




```
let rec ast_to_asm_expr expr ... =
  match expr with
    ...
  | ExprOp("<", [e0; e1]) ->
    let insns1,op1 = ast_to_asm_expr e1 ... in
    let insns0,op0 = ast_to_asm_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;
          setl rax ]
    op0)
  | ...
```

Compiling an expression (function call)

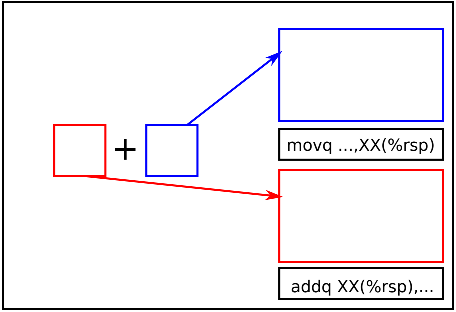
- ≈ compile all arguments; put them to positions specified by ABI; a `call` instruction




```
let rec ast_to_asm_expr expr ... =
  match expr with
    ...
  | ExprCall(f, args) ->
     let insns,arg_vars =
          ast_to_asm_exprs args env
          var_idx in
     ((insns @ (make_call f arg_vars)),
          rax)
```

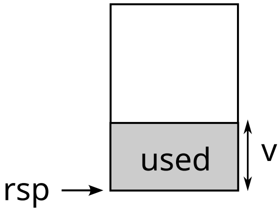
Details we have been leaving out

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



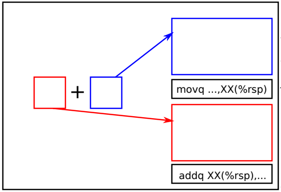
Determining where to save subexpressions

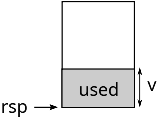
- `ast_to_asm_expr` E receives a value (v) pointing to the lowest end of free space
- `ast_to_asm_expr` E v … generates instructions that evaluate E using (destroying) only addresses at or above v`(%rsp)`



Determining where to save subexpressions

- $\rightarrow$ when evaluating $A + B$, save B at v(%rsp)
- let A use $v + 8$ and higher addresses






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

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
- $\rightarrow$ `ast_to_asm_stmt` also takes v pointing to the beginning of the free space
- `ast_to_asm_stmt` S v $\ldots$ generates instructions to execute S; they use (destroy) only addresses at or 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_asm_stmt` and `ast_to_asm_expr`
- when new variables are declared at the beginning of a compound statement ({ ... }), add new mappings to it

`ast_to_asm_expr` receives an environment

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

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

`ast_to_asm_stmt` receives an environment too

```
let rec ast_to_asm_stmt stmt env v =
  match stmt with
    ...
  | StmtCompound(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'*

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*)
- *env', v'* = `env_extend` *decls loc env*

  can be easily built on `env_add` (left for you)