

Implementing a Compiler

Kenjiro Taura

2024/06/16

Contents

The MinC (“Minimum C”) language	2
Overview of Inside a Compiler	5
Lexer and parser : source code \rightarrow AST	11
Intermediate Representation (IR)	39
Code generation	42

The MinC (“Minimum C”) language

MinC (“Minimum C”) spec overview

- all expressions have type `long` (64 bit integer)
 - no other integers, floating point numbers, pointers, or structs
 - everything is long \Rightarrow *type checks are unnecessary*
- no global variables or `typedef`
 - \Rightarrow a program = list of *function definitions*
- supported complex statements are `if`, `while`, and compound statement (`{ ... }`) only

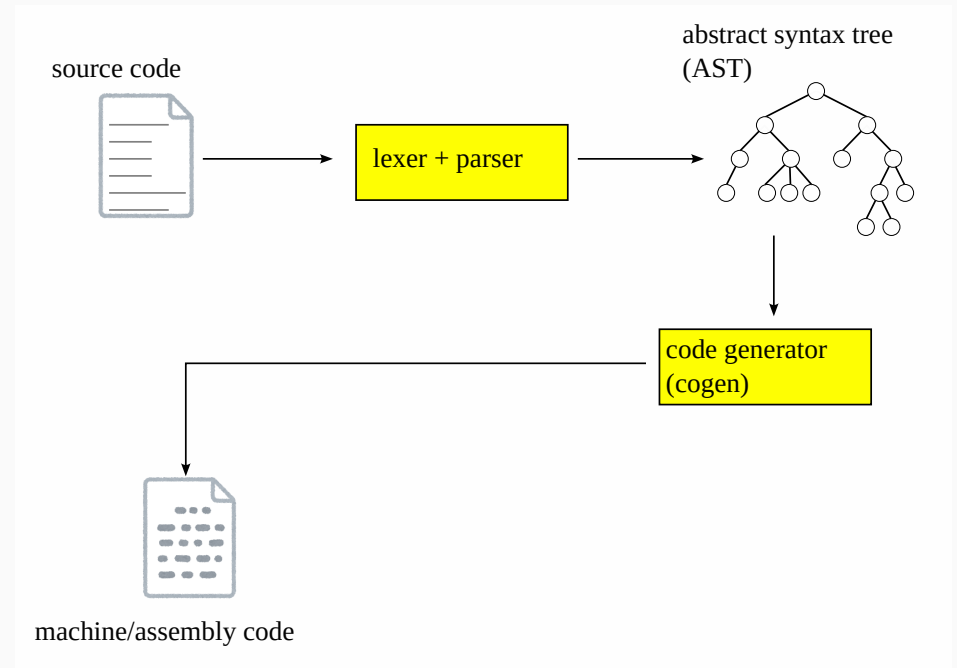
MinC (“Minimum C”) spec overview

- function calls follow the C convention \Rightarrow MinC code can call or be called by functions compiled by other compilers (e.g., gcc)

Overview of Inside a Compiler

Data structures

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



Data structures

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

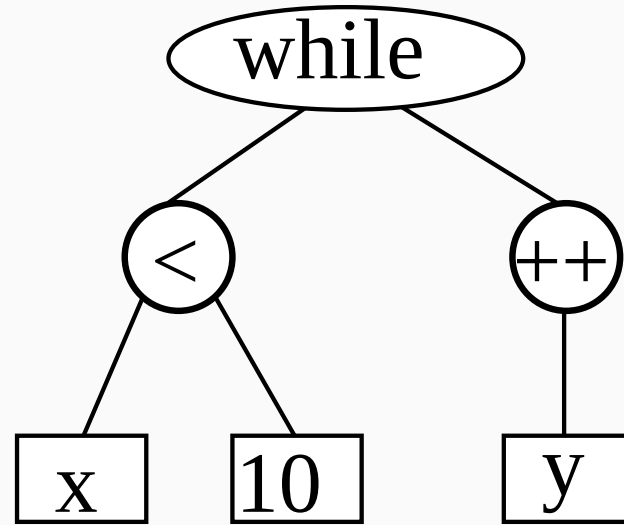
Typical compilation steps

1. **lexing and parsing:** source code (string) \rightarrow AST
2. IR generation: AST \rightarrow IR (*)
3. optimization: IR \rightarrow IR (*)
4. **code generation:** IR \rightarrow assembly

(*) : optional

Abstract Syntax Tree (AST)

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



Components of the baseline code

- `parser/`
 - `minc_grammar.y` ... grammar definition
 - `minc_to_xml.py` ... MinC \rightarrow XML converter
- `{go,jl,ml,rs}/minc/`
 - `minc_ast.??` ... abstract syntax tree (AST) definition
 - `minc_parse.??` ... XML \rightarrow AST
 - `minc_cogen.??` ... AST \rightarrow assembly
 - `main.??` or `minc.??` ... main driver

Your work

- files other than `minc_cogen.??` are given and need not be modified (unless you do something extra)
- `minc_cogen.??` is almost empty and your primary job is to complete it

Lexer and parser :
source code \rightarrow AST

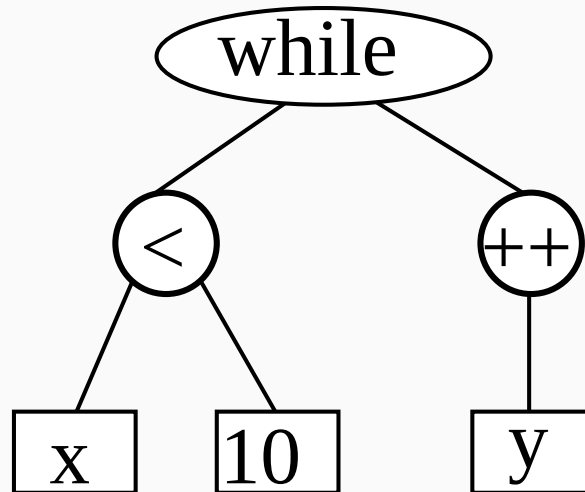
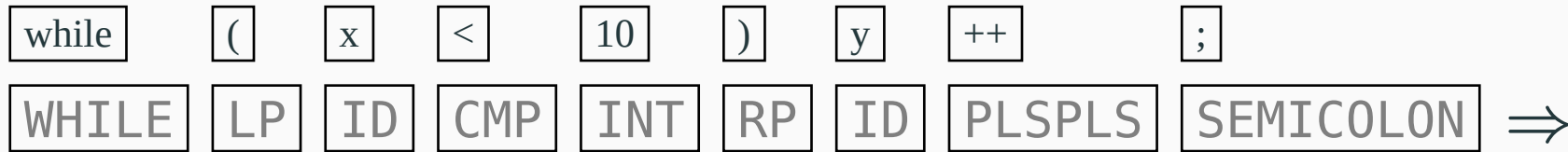
Lexer and parser

- **lexer**: string \rightarrow sequence of *tokens* (\approx words)
 - also called *lexical analyzer*, or *tokenizer*
 - `while (x < 10) y++; \Rightarrow`

while	(x	<	10)	y	++	;
WHILE	LP	ID	CMP	INT	RP	ID	PLSPLS	SEMICOLON

Lexer and parser

- **parser:** sequence of tokens \rightarrow AST



Specifying a grammar

- a grammar for *tokens*
 - specifies which character sequence constitutes a valid token
 - typically uses **Regular Expressions (RE)**
- a grammar for *the entire inputs*
 - specifies which token sequence constitutes a valid input
 - typically uses (a subset of) **Context Free Grammar (CFG)**
- note: there is an approach that uses a single grammar for both

Regular expression

- a regular expression is any expression that can be formed by:

ϵ	(empty string)
c	(a character)
$E E$	(concatenation)
$E \mid E$	(alternation)
E^*	(zero or more repetition)
(E)	(paren)

where E is a regular expression

- \mid , * , $($ and $)$ are literals

Regular expression

- expressions for convenience

$$\begin{aligned} E+ &\equiv E E^* \text{ (one or more repetition)} \\ E? &\equiv \varepsilon \mid E \text{ (optional)} \end{aligned}$$

Regular expression examples

- to build complex expressions, use symbols to represent regular expressions used in other regular expressions. e.g.,

nz	=	1 2 3 4 5 6 7 8 9	1, 2, ..., 9
digit	=	0 nz	0, 1, 2, ..., 9
non_neg	=	0 nz digit*	0, 12, 34
int	=	-? non_neg	0, -0, 12, -34
fraction	=	int (. digit*)?	-12.34
float	=	fraction (e int)	-12.34e-5

Regular expression examples

alpha = A | B | ... Z | a | b | ... z A, B, ..., Z, a, b, ..., z
alpha_ = alpha | _ A, B, ..., Z, a, b, ..., z, _
id = alpha_ (alpha_ | digit) * a, abc, a0_b1

Regular expression semantics (just for formality ...)

- a regular expression E represents *a set of strings*, written $\llbracket E \rrbracket$

$$\llbracket \varepsilon \rrbracket = \{ \text{“”} \}$$

$$\llbracket c \rrbracket = \{ c \}$$

$$\llbracket E_0 E_1 \rrbracket = \{ e_0 + e_1 \mid e_0 \in \llbracket E_0 \rrbracket, e_1 \in \llbracket E_1 \rrbracket \}$$

$$\llbracket E_0 \mid E_1 \rrbracket = \llbracket E_0 \rrbracket \cup \llbracket E_1 \rrbracket$$

$$\llbracket E^* \rrbracket = \{ \text{“”} \} \cup \{ e_0 + e_1 \mid e_0 \in \llbracket E \rrbracket, e_1 \in \llbracket E^* \rrbracket \}$$

$$\llbracket (E) \rrbracket = \llbracket E \rrbracket$$

- note: “+” represents string concatenation

Context Free Grammar (CFG)

- specified by a collection of *production rules*
- a production rule looks like

$$L \rightarrow R_0 R_1 \dots$$

where

- L : a symbol (*non-terminal*)
- R_i is either
 - a symbol defined by a production rule(s), or
 - a token name (a *terminal* symbol)

An example : expressions

$\text{expr} \rightarrow \text{int}$	12, 345, ...
$\text{expr} \rightarrow \text{id}$	f, x, i, is_prime, ...
$\text{expr} \rightarrow \text{unop expr}$	-x, exp, !a_greater_than_b
$\text{expr} \rightarrow \text{expr binop expr}$	x + y, a * x + b * y + 1, a & b, ...
$\text{expr} \rightarrow (\text{expr})$	3 * (a + 1)
$\text{expr} \rightarrow \text{funcall}$	

- blue symbols (int, id, unop, binop, (,)) are terminals (tokens)
- above rules overlook the fact that some operators (i.e., + and -) can be used as a unary operator and a binary operator

An example : function call

funcall	→	id (comma_exprs)	f(x, 2 * y, 1)
comma_exprs	→		
comma_exprs	→	expr	
comma_exprs	→	expr comma_expr_star	
comma_expr_star	→		
comma_expr_star	→	, expr comma_expr_star	

An example : statements

stmt \rightarrow ;
stmt \rightarrow continue ;
stmt \rightarrow break ;
stmt \rightarrow return ;
stmt \rightarrow { decl* stmt* }
stmt \rightarrow if (expr) stmt (else stmt)?
stmt \rightarrow while (expr) stmt
stmt \rightarrow expr ;

- as you have seen,
 - the same symbol L can appear multiple times in the lefthand side (i.e., *alternation*)
 - R_i can be L or any symbol defined earlier or later (i.e., definitions can be *recursive*)

A few shorthands

- we often use shorthands (`|`, `?`, `*`, `+`) that have similar meanings with those for RE
- they can be mechanically eliminated
- the above example using the shorthands:

`expr` \rightarrow `int` `|` `id` `|` `unop` `expr` `|` `expr` `binop` `expr` `|` `funcall`
`funcall` \rightarrow `id` `(` `comma_exprs` `)`
`comma_exprs` \rightarrow `|` `expr` `(` `,` `expr` `)``*`

CFG semantics (for formality)

- each symbol L represents a set of token sequences ($\llbracket L \rrbracket$)
- $\llbracket L \rrbracket$ is the set of token sequences that can result by, starting from L , repeatedly replacing a non-terminal symbol to the righthand side of its production rule, until it becomes a sequence of tokens (terminals)

expr \rightarrow funcall
 \rightarrow id (comma_exprs)
 \rightarrow id (expr comma_expr_star)
 \rightarrow id (id comma_expr_star)

CFG semantics (for formality)

- `id (id , expr comma_expr_star)`
- `id (id , expr + expr comma_expr_star)`
- `id (id , id + expr comma_expr_star)`
- `id (id , id + int comma_expr_star)`
- `id (id , id + int)`

∴ `id (id , id + int)` (e.g., `f(x, y + 1)`) ∈ $\llbracket \text{expr} \rrbracket$

An alternative semantics

- $\llbracket . \rrbracket$ is the minimal set of token sequences satisfying:

1. $\llbracket t \rrbracket = \{ t \}$ (t : terminal)

2. $L \rightarrow R_0 \dots R_{n-1}$ implies

$$\begin{aligned} r_0 \in \llbracket R_0 \rrbracket, \dots, r_{n-1} \in \llbracket R_{n-1} \rrbracket \\ \Rightarrow r_0 + \dots + r_{n-1} \in \llbracket L \rrbracket \end{aligned}$$

- “+” represents concatenation of token sequences

CFG is more expressive than RE

- as you might have noticed, RE is a special case of CFG
- all the constructs of RE can be straightforwardly expressed with CFG
- e.g., a CFG equivalent to RE “int = 0 | nz digit^{*}”

int \rightarrow 0

digit \rightarrow 0

int \rightarrow nz digits

digit \rightarrow nz

digits \rightarrow

nz \rightarrow 1 | ... | 9

digits \rightarrow digit digits

In general ...

- below, $C(e, L)$ is a function that converts regular expression e to an equivalent CFG s.t., $\llbracket L \rrbracket = \llbracket e \rrbracket$

$$C(\varepsilon, L) = \{L \rightarrow\}$$

$$C(c, L) = \{L \rightarrow c\}$$

$$C(E_0 E_1, L) = \{L \rightarrow R_0 R_1\} \cup C(E_0, R_0) \cup C(E_1, R_1)$$

$$C(E_0 | E_1, L) = \{L \rightarrow R_0, L \rightarrow R_1\} \cup C(E_0, R_0) \cup C(E_1, R_1)$$

$$C(E^*, L) = \{L \rightarrow \mid R L\} \cup C(E, R)$$

$$C((E)) = C(E, L)$$

- R, R_0 and R_1 are unique symbols that do not appear elsewhere

A CFG that cannot be expressed by RE

- intuitively, RE *can repeat* (E^*) *but cannot recurse*
- e.g., both “ $A \rightarrow \mid a A$ ” and “ $A \rightarrow \mid A a$ ” can be expressed by an RE (both are equivalent to a^*), but

$$A \rightarrow \mid a A b$$

cannot ($\llbracket A \rrbracket = \{\varepsilon, ab, aabb, aaabbb, \dots\} = \{a^n b^n \mid n \geq 0\}$)

- **the proof** is interesting but omitted

If $RE \subset CFG$, why use both (not just CFG)?

- parsing *general* CFG is expensive ($O(\text{length}^3)$)
- the primary reason is handling *alternatives* requires *backtrack*

$$A \rightarrow B_0 B_1 \dots \mid C_0 C_1 \dots \mid D_0 D_1 \dots$$

- practical parsers take either of the following two approaches
 1. allow only alternatives that can be determined with a *limited lookahead* (LL(1), LALR(1), etc.)
 2. allow backtrack with programmer-supplied *cut points*
(*Parsing Expression Grammar; PEG*)

CFG with a limited lookahead (LL(1), LALR(1), etc.)

- recall the syntax of statement

stmt \rightarrow ; | continue ; | break ; | return ; | { decl* stmt* }
| if (expr) | while (expr) stmt stmt (else stmt)?
| expr ;

- upon parsing a statement, which branch we should take can be determined just by its *first* token
- it is essential to have a separate tokenizer for this type of grammar* (looking ahead a token \neq looking ahead a character)

Parsing Expression Grammar (PEG)

- PEG allows unlimited lookahead (uses backtrack)
- in an alternative, it always tries branches in the written order (the order *does* matter!)
 - 1st branch,
 - if failed, 2nd branch,
 - if failed, 3rd branch, ...
- the programmer may insert a *cut point*
 - if a parser succeeds thus far, it tries no other branches

Lexer/parser generators

- based on the grammar, either:
 - write them by hand, or
 - use a **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, ...)

Lexer/parser generators

- some grammar frameworks (PEG) specify them in a single framework

Lexer/parser generators

- many programming languages have lexer/parser generators:
 - lex/yacc ([flex/bison](#)): C/C++
 - [ANTLR](#): C, C++, Java, Python, JavaScript, Go, ...
 - [ocamllex/menhir](#): OCaml
 - [tatsu](#): Python
 - etc.

In this exercise ...

- we use [tatsu](#), a parser generator tool based on PEG, to generate a Python program that converts C source into XML,
- which is then read by the respective XML library you have used before for your language
- see [grammar syntax](#) in tatsu
 - thanks to PEG, no need for separate definitions of tokens
- the MinC grammar in tatsu is given in `minc_grammar.y`

Intermediate Representation (IR)

Intermediate Representation (IR)

- a common representation of programs used by a compiler
- roughly \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. formulate optimizations as $\text{IR} \rightarrow \text{IR}$ transformations

Intermediate Representation (IR)

- **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

Code generation (minc_cogen) — basic structure

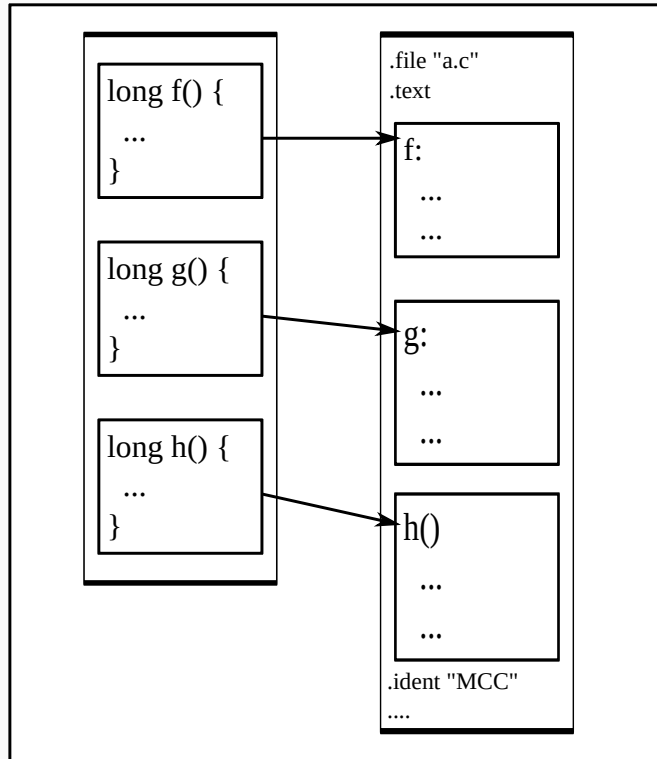
- takes an AST and returns machine code (a list of instructions)
- generate machine code for an AST \approx generate machine code of its components and properly arrange them
- program \rightarrow function definition \rightarrow statement \rightarrow expression

Code generation (minc_cogen) — basic structure

- code generator has lots of:
 - case analysis based on the type of the tree; use:
 - pattern matching (match à la OCaml and Rust), or
 - polymorphism (OCaml objects, Julia function, Go interface, Rust trait)
 - recursive calls to child trees

Compiling an entire file

- \approx concatenate compilation of individual function definitions

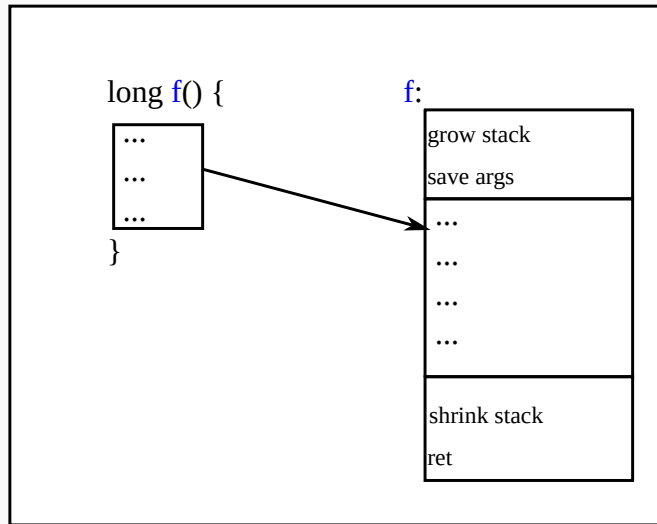


Pseudo code:

```
ast_to_asm_program (Program([d0, d1, ...])) ... =
    ...
    header
    + (ast_to_asm_def d0 ...)
    + (ast_to_asm_def d1 ...)
    + ...
    + trailer
```

Compiling a function definition

- \approx prologue (grow the stack, etc.) + code for the body (statement) + epilogue (shrink the stack, `ret`, etc.)

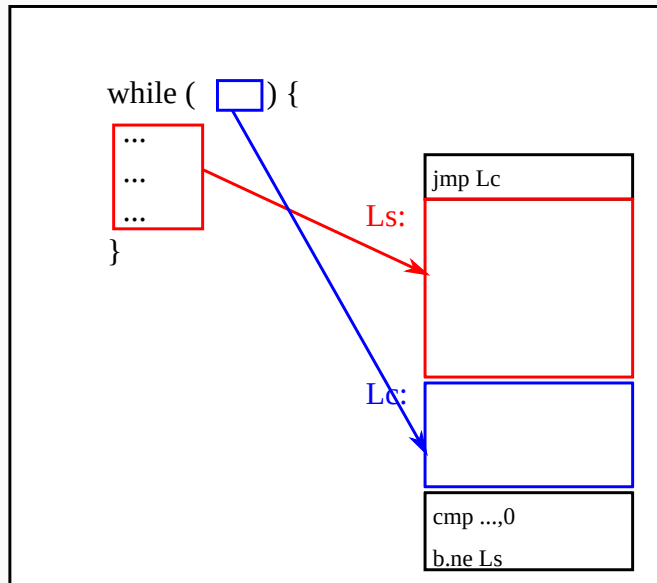


Pseudo code:

```
ast_to_asm_def (DefFun(f, params, ret_type, body)) =  
  (gen_prologue f ...)  
  + (ast_to_asm_stmt body ...)  
  + (gen_epilogue f ...)
```


Compiling a statement (while statement)

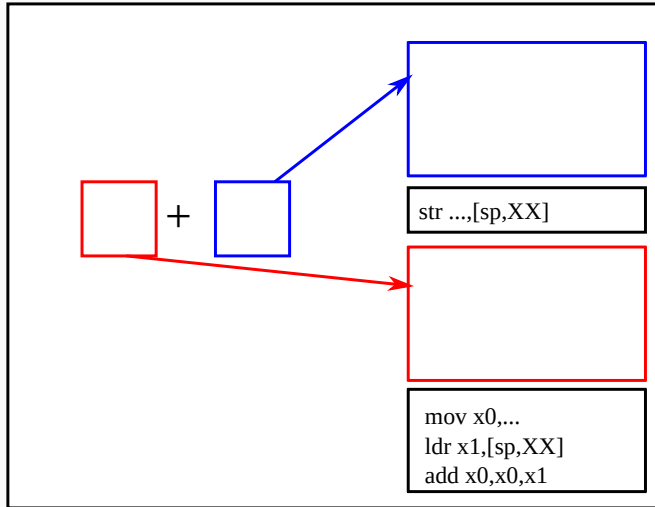
- \approx place code for the body, condition expression, compare, and a conditional branch



```
ast_to_asm_while_stmt (StmtWhile(cond, body)) ... =  
    cond_op,cond_insns = ast_to_asm_expr cond ... ;  
    body_insns = ast_to_asm_stmt body ... ;  
    ...  
    [ jmp Lc; Ls ]  
+ body_insns  
+ [ Lc ]  
+ cond_insns  
+ [ cmp cond_op,0; jne Ls ]
```

Compiling an expression (arithmetic)

- \approx code for the arguments; the arithmetic instruction



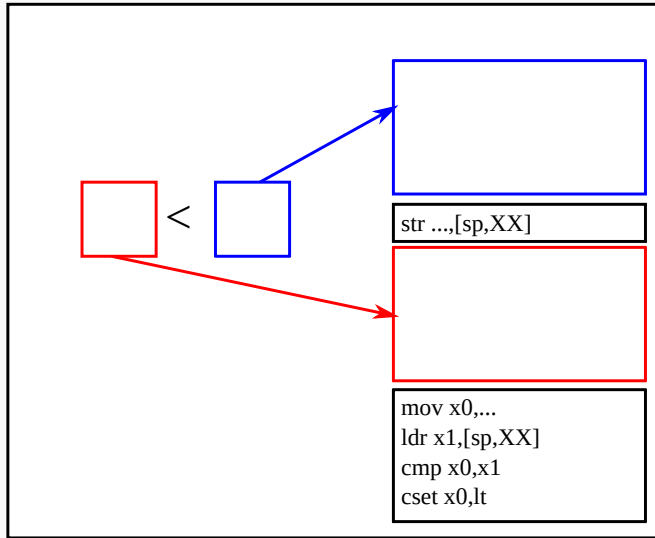
```
ast_to_asm_add_expr ExprOp("+", [e0; e1]) ... =  
  insns1,op1 = ast_to_asm_expr e1 ... ;  
  insns0,op0 = ast_to_asm_expr e0 ... ;  
  m = (* a slot on the stack *);  
  (insns1  
    + [ str op1,m ]  
    + insns0  
    + [ mov x0,op0;  
        ldr x1,m;  
        add x0,x0,x1 ],  
    x0)
```

Compiling an expression (comparison)

- $A < B$ is an expression that evaluates to:
 - 1 if $A < B$
 - 0 if $A \geq B$
- this can be done by `cmp` + conditional set (`cset`)

Compiling an expression (comparison)

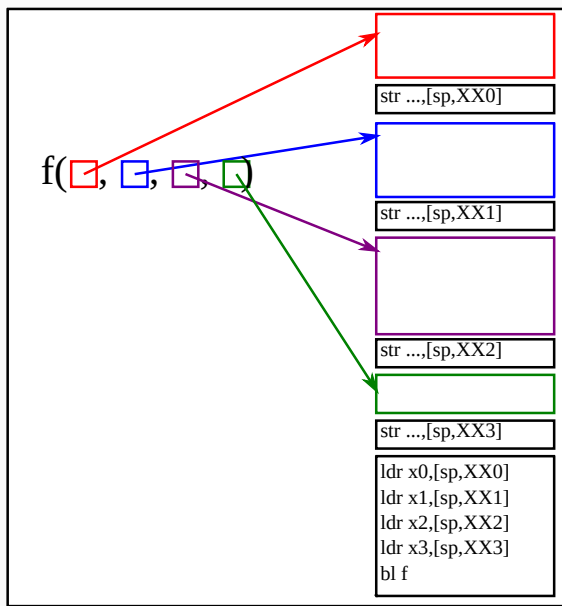
- \approx compile the arguments; compare; conditional set



```
ast_to_asm_cmp_expr (ExprOp("<", [e0; e1])) ... =  
  insns1,op1 = ast_to_asm_expr e1 ... ;  
  insns0,op0 = ast_to_asm_expr e0 ... ;  
  m1 = (* a slot on the stack *);  
  ...  
(insns1  
  + [ str op1,m1 ]  
  + insns0  
  + [ mov x0,op0;  
      ldr x1,m1;  
      cmp x0,x1;  
      cset x0,lt ],  
  x0)
```

Compiling an expression (function call)

- \approx compile all arguments; put them to positions specified by ABI; a `bl` instruction

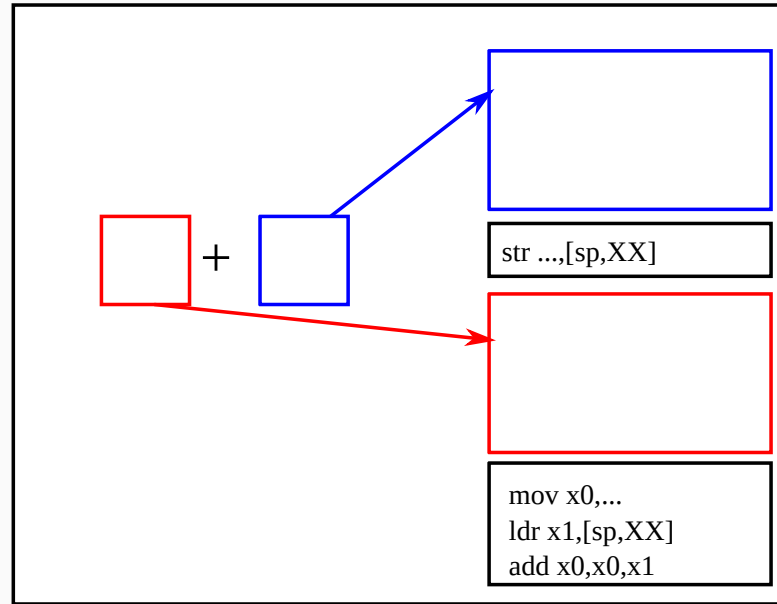


```
ast_to_asm_call_expr (ExprCall(f, [e0;e1;...])) ... =  
  [(i0,op0);(i1,op1);...], [m0;m1;...]  
  = ast_to_asm_exprs [e0;e1;...] ...;  
  ( (i0 + [str op0,m0])  
    + (i1 + [str op1,m1])  
    + ...  
    + [ldr x0,m0;  
      ldr x1,m1;  
      ...;  
      bl f],  
    x0)
```

A few left-out details

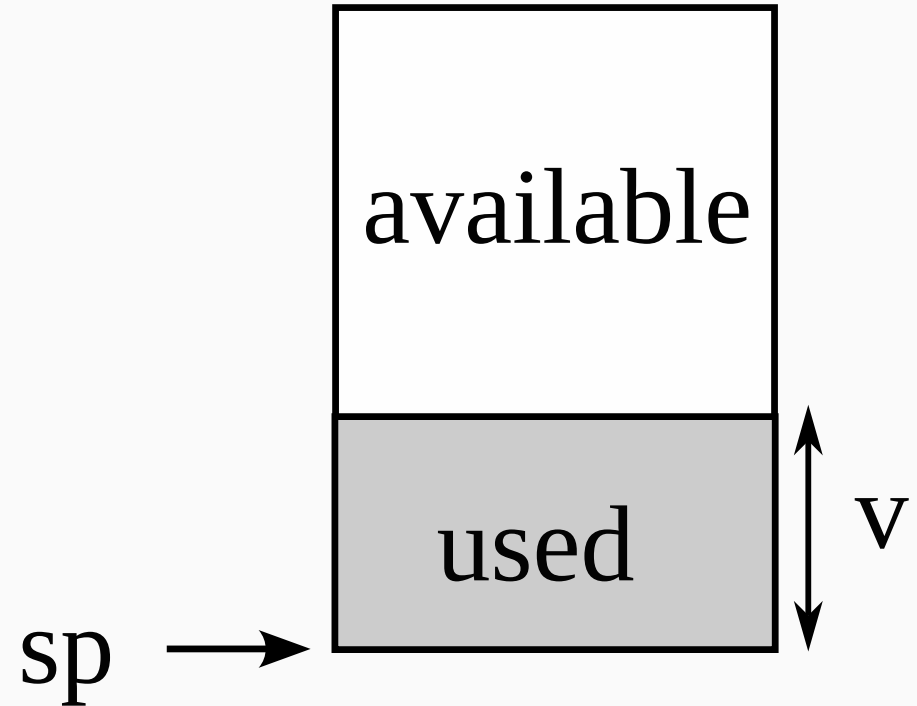
- how to determine locations to save values of *subexpressions* and *variables*
- that is, how to determine XX below:

A few left-out details



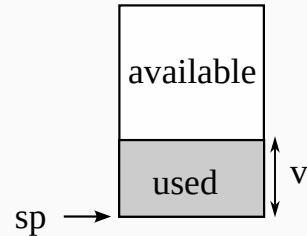
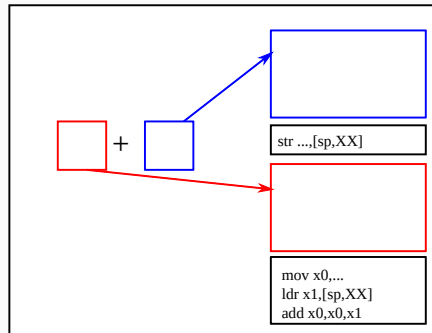
Determining where to save subexpressions

- `ast_to_asm_expr E` receives a value (v) pointing to the lowest end of free space in the current stack frame
- `ast_to_asm_expr E v ...` generates instructions that evaluate E using (destroying) only addresses at or above $SP+v$



Determining where to save subexpressions

- when evaluating $A + B$,
 - evaluate B , using $SP+v$ and higher; save the result at $SP+v$
 - evaluate A , using $v + 8$ and higher addresses



```
ast_to_asm_add_expr ExprOp("+", [e0; e1]) v ... =  
  insns1, op1 = ast_to_asm_expr e1 v ... ;  
  insns0, op0 = ast_to_asm_expr e0 (v+8) ... ;  
(  insns1  
  + [ str op1, [sp, v] ]  
  + insns0  
  + [ mov x0, op0;  
      ldr x1, [sp, v];  
      add x0, x0, x1],  
  x0)
```

Locations to hold variables

- example:

```
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

Locations to hold variables

- `ast_to_asm_stmt S v ...` generates instructions to execute `S`; they use (destroy) only addresses at or above $SP+v$
- \rightarrow hold `a`, `b`, and `c` at:
 - $a \mapsto SP+v$,
 - $b \mapsto SP+v+8$
 - $c \mapsto SP+v+16$

Environment: records where variables are held

- when a variable occurs in an expression, we need to get the location that holds the variable
 - e.g., to compile $x + 1$, we need to know where x is stored
- \rightarrow 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

```
ast_to_asm_expr (ExprId(x)) env v =  
  m = env_lookup x env;  
  ([ ldr x0,m ], x0)
```

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

ast_to_asm_stmt receives an environment too

```
ast_to_asm_stmt (StmtCompound(decls, stmts)) env v =  
  env', v' = env_extend decls env v;  
  ast_to_asm_stmts stmts env' v' ...
```

- `env_extend decls env v` :
 - assigns locations (`v`, `v+8`, `v+16`, ...) to variables declared in `decls`
 - registers them in `env`
 - returns the new environment `env'` and the new free space `v'`

Implementing environment

- an environment is a list of (*variable name*, *location*) pairs
- `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 $x \mapsto \text{loc}$ in addition to `env`
- `(env', v') = env_extend decls loc env`
 - can be easily built on `env_add` (left for you)