Implementing a Compiler

Kenjiro Taura

2024/06/23

Contents

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
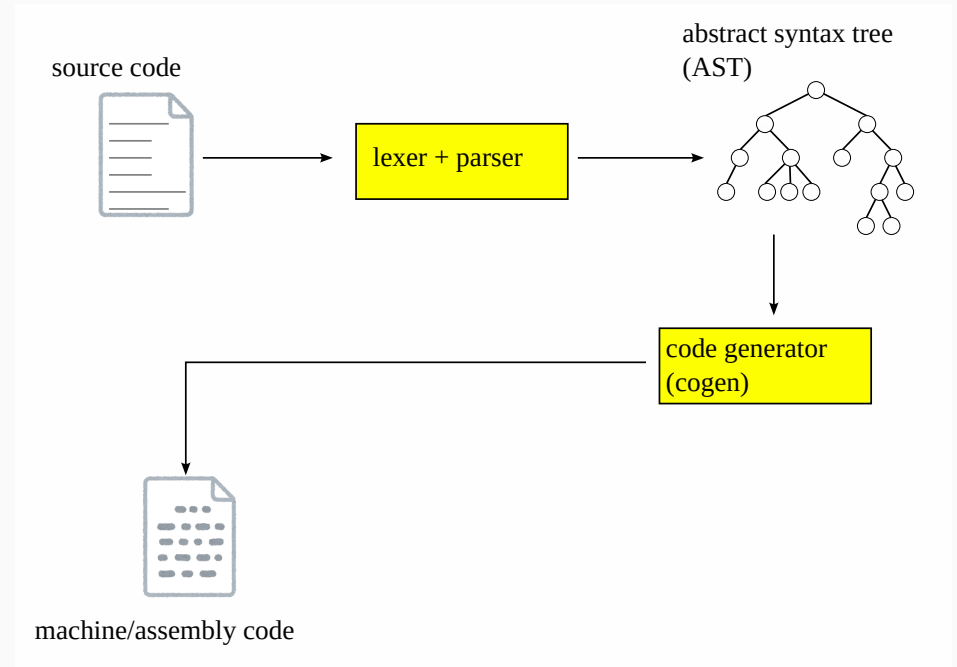
MinC ("Minimum C") spec overview

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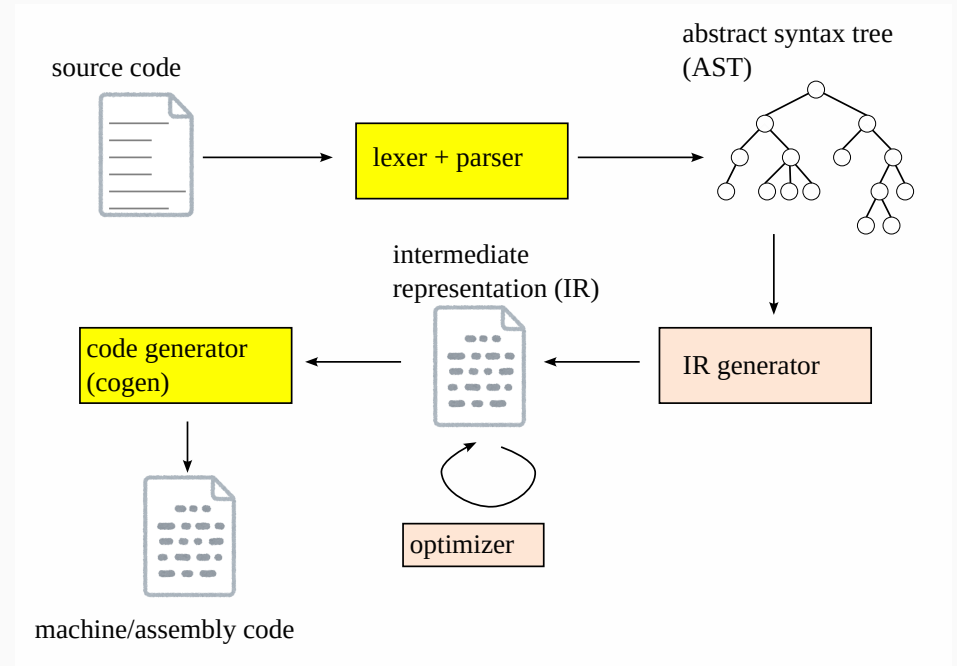
Overview of Inside a Compiler

Data structures

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

source code

lexer + parser

abstract syntax tree (AST)

code generator (cogen)

machine/assembly code

Data structures

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


source code

lexer + parser

abstract syntax tree (AST)

intermediate representation (IR)

code generator (cogen)

IR generator

optimizer

machine/assembly code

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**
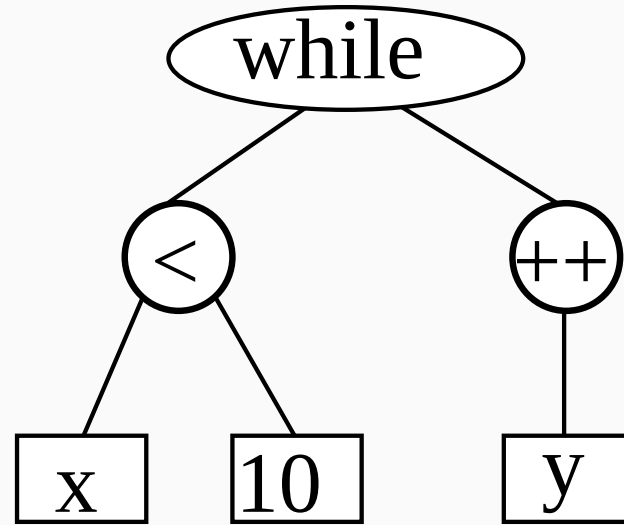
Components of the baseline code

- `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

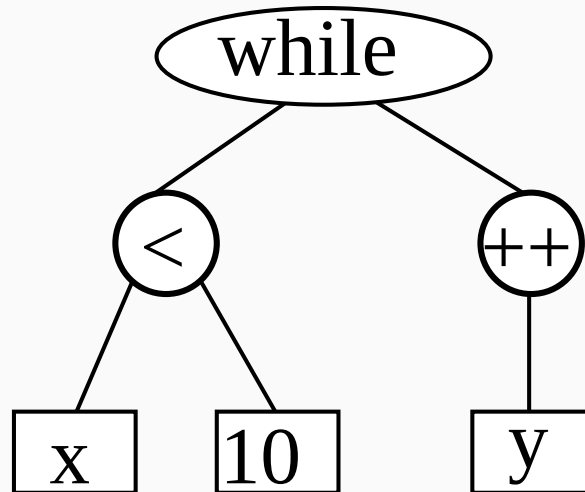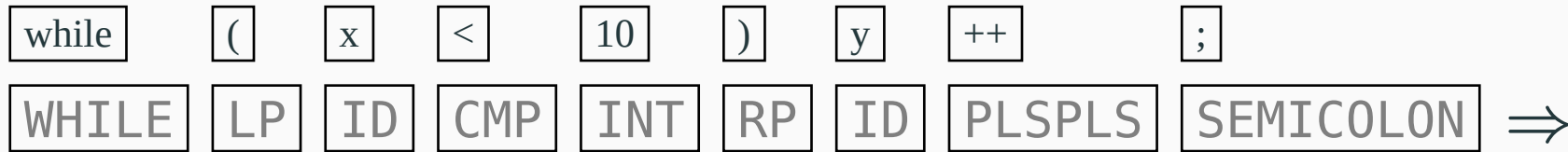# Lexer and parser : source code → AST

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

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

| WHILE | LP | ID | CMP | INT | RP | ID | PLSPLS | SEMICOLON | $\Rightarrow$

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:

$$\varepsilon \quad \text{(empty string)}$$
$$c \quad \text{(a character)}$$
$$E\ E \quad \text{(concatenation)}$$
$$E\ |\ E \quad \text{(alternation)}$$
$$E* \quad \text{(zero or more repetition)}$$
$$(E) \quad \text{(paren)}$$

where E is a regular expression

- |, *, (and) are literals

- expressions for convenience

$$E\text{+} \equiv E \ E\text{*} \ \text{(one or more repetition)}$$
$$E\text{?} \equiv \varepsilon \mid E \ \ \text{(optional)}$$

Regular expression examples

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

$$
\begin{aligned}
\text{nz} &= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \qquad &&1, 2, \ldots, 9\\
\text{digit} &= 0 \mid \text{nz} &&0, 1, 2, \ldots, 9\\
\text{non\_neg} &= 0 \mid \text{nz digit*} &&0, 12, 34\\
\text{int} &= \text{-? non\_neg} &&0, -0, 12, -34\\
\text{fraction} &= \text{int (. digit*)?} &&-12.34\\
\text{float} &= \text{fraction (e int)} &&-12.34e\text{-}5
\end{aligned}
$$

Regular expression examples

$$alpha \quad = \quad A \mid B \mid \ldots Z \mid a \mid b \mid \ldots z \qquad \text{A, B, ..., Z, a, b, ..., z}$$

$$alpha\_ \quad = \quad alpha \mid \_ \qquad \text{A, B, ..., Z, a, b, ..., z, \_}$$

$$id \qquad = \quad alpha\_ \; (\; alpha\_ \mid digit \;)* \qquad \text{a, abc, a0\_b1}$$

Regular expression semantics (just for formality …)

- a regular expression E represents *a set of strings*, written $[\![E]\!]$

$$
\begin{aligned}
[\![\varepsilon]\!] &= \{\ \text{""}\ \} \\
[\![c]\!] &= \{\ c\ \} \\
[\![E_0\ E_1]\!] &= \{\ e_0 + e_1 \mid e_0 \in [\![E_0]\!], e_1 \in [\![E_1]\!]\ \} \\
[\![E_0 \mid E_1]\!] &= [\![E_0]\!] \cup [\![E_1]\!] \\
[\![E^*]\!] &= \{\ \text{""}\ \} \cup \{e_0 + e_1 \mid e_0 \in [\![E]\!], e_1 \in [\![E^*]\!]\ \} \\
[\![(E)]\!] &= [\![E]\!]
\end{aligned}
$$

- note: "+" represents string concatenation

Context Free Grammar (CFG)

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

$$L \rightarrow R_0 \ R_1 \ ...$$

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

$$
\begin{array}{lll}
\text{expr} & \rightarrow & \text{int} \qquad\qquad\qquad\quad \text{12, 345, …} \\
\text{expr} & \rightarrow & \text{id} \qquad\qquad\qquad\quad \text{f, x, i, is\_prime, …} \\
\text{expr} & \rightarrow & \text{unop expr} \qquad\qquad \text{-x, exp, !a\_greater\_than\_b} \\
\text{expr} & \rightarrow & \text{expr binop expr} \quad \text{x + y, a * x + b * y + 1, a \& b, …} \\
\text{expr} & \rightarrow & \text{(expr)} \qquad\qquad\; \text{3 * (a + 1)} \\
\text{expr} & \rightarrow & \text{funcall}
\end{array}
$$

- 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

funcall	$\rightarrow$	id (comma_exprs)	f(x, 2 * y, 1)
comma_exprs	$\rightarrow$		
comma_exprs	$\rightarrow$	expr	
comma_exprs	$\rightarrow$	expr comma_expr_star	
comma_expr_star	$\rightarrow$		
comma_expr_star	$\rightarrow$	, expr comma_expr_star	

$$\begin{aligned}
\text{stmt} &\rightarrow\ ; \\
\text{stmt} &\rightarrow\ \text{continue}\ ; \\
\text{stmt} &\rightarrow\ \text{break}\ ; \\
\text{stmt} &\rightarrow\ \text{return}\ ; \\
\text{stmt} &\rightarrow\ \{\ \text{decl* stmt*}\ \} \\
\text{stmt} &\rightarrow\ \text{if (expr) stmt (else stmt)?} \\
\text{stmt} &\rightarrow\ \text{while (expr) stmt} \\
\text{stmt} &\rightarrow\ \text{expr}\ ;
\end{aligned}$$

Notes

- 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 ($[\![L]\!]$)
- $[\![L]\!]$ 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)

$$
\begin{aligned}
\text{expr} \;\to\; &\text{funcall} \\
\to\; &\text{id (comma\_exprs)} \\
\to\; &\text{id (expr comma\_expr\_star)} \\
\to\; &\text{id (id comma\_expr\_star)}
\end{aligned}
$$

$\rightarrow$ id (id , expr comma_expr_star)

$\rightarrow$ id (id , expr + expr comma_expr_star)

$\rightarrow$ id (id , id + expr comma_expr_star)

$\rightarrow$ id (id , id + int comma_expr_star)

$\rightarrow$ id (id , id + int)

$\therefore$ id (id , id + int) (e.g., f(x, y + 1)) $\in$ $[\![$expr$]\!]$

An alternative semantics

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

1. $\llbracket\, t \,\rrbracket = \{\, t \,\}\ (t : \text{terminal})$
2. $L \rightarrow R_0 \ldots R_{n-1}$ implies

$$r_0 \in \llbracket R_0 \rrbracket, \ldots, r_{n-1} \in \llbracket R_{n-1} \rrbracket$$
$$\Rightarrow r_0 + \ldots + r_{n-1} \in \llbracket L \rrbracket$$

- "+" 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*"

$$\text{int} \rightarrow 0$$
$$\text{int} \rightarrow \text{nz digits}$$
$$\text{digits} \rightarrow$$
$$\text{digits} \rightarrow \text{digit digits}$$

$$\text{digit} \rightarrow 0$$
$$\text{digit} \rightarrow \text{nz}$$
$$\text{nz} \rightarrow 1 \mid ... \mid 9$$

In general …

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

$$C(\varepsilon, L) = \{L \to\}$$
$$C(c, L) = \{L \to c\}$$
$$C(E_0\ E_1, L) = \{L \to R_0\ R_1\} \cup C(E_0, R_0) \cup C(E_1, R_1)$$
$$C(E_0 | E_1, L) = \{L \to R_0, L \to R_1\} \cup C(E_0, R_0) \cup C(E_1, R_1)$$
$$C(E*, L) = \{L \to |\ R\ L\} \cup C(E, R)$$
$$C((E))] = C(E, L)$$

- 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 "$A \to \mid a\ A$" and "$A \to \mid A\ a$" *can* be expressed by an RE (both are equivalent to a^*), but

$$A \to \mid a\ A\ b$$

cannot ($[\![A]\!] = \{\varepsilon, ab, aabb, aaabbb, ...\} = \{a^n b^n \mid n \geq 0\}$)
- 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*

$$A \to B_0\ B_1\ ... \mid C_0\ C_1\ ... \mid D_0\ D_1\ ...$$

- 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   →   ; | 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, …)

- 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 IR $\rightarrow$ 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

- 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 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

- ≈ concatenate compilation of individual function definitions
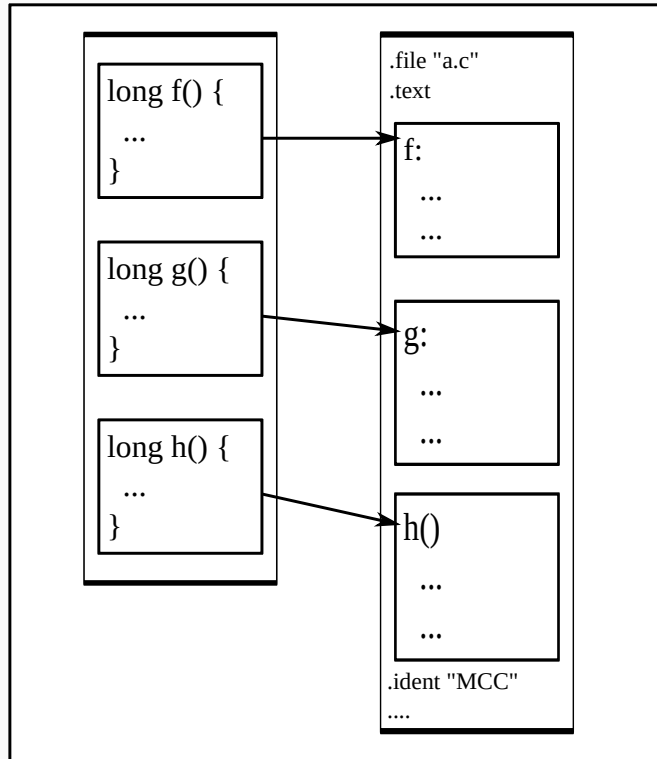
Pseudo code:

```
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.)
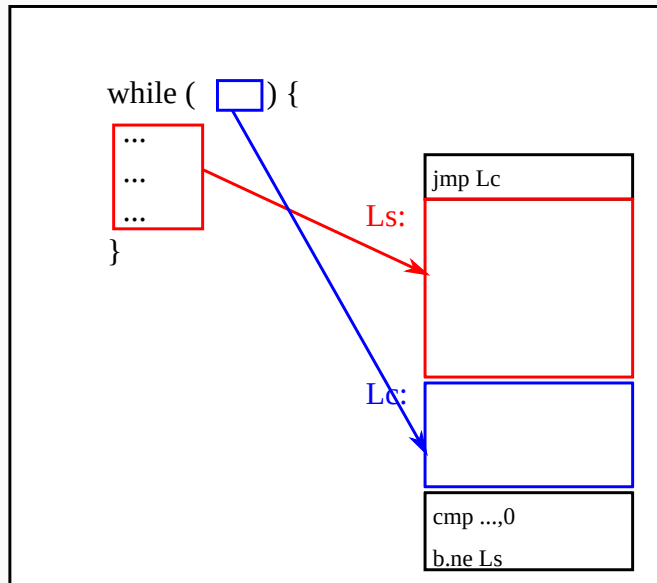
Pseudo code:

```
long f() {              f:
   ...                    grow stack
   ...                    save args
   ...                    ...
}                         ...
                         ...
                         ...

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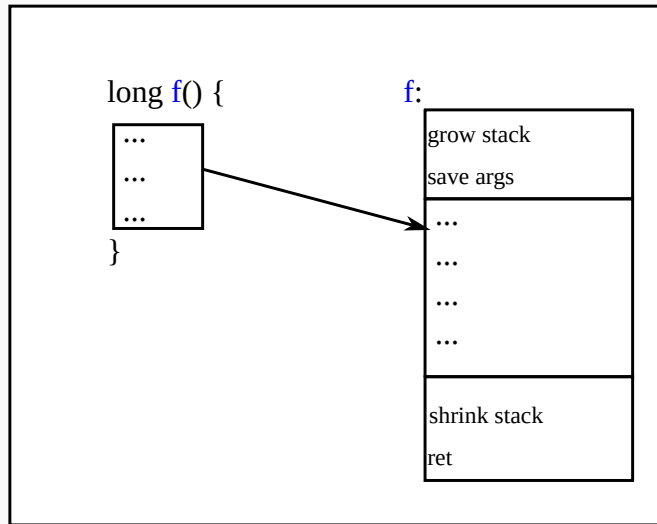
```
ast_to_asm_def (DefFun(f, params, ret_type, body)) =
      (gen_prologue f ...)
   + (ast_to_asm_stmt body ...)
   + (gen_epilogue f ...)
```

- ≈ 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)
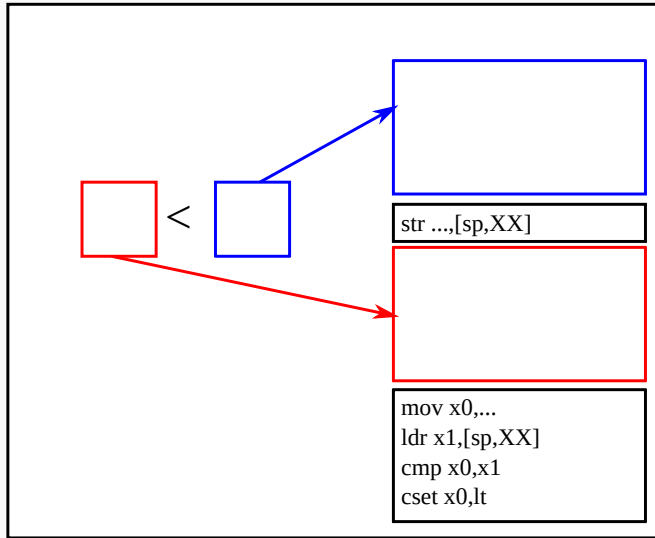
- ≈ code for the arguments; the arithmetic instruction
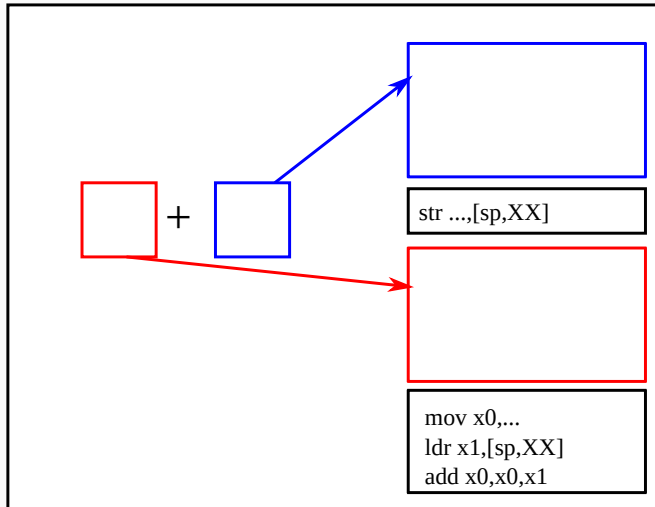

```
ast_to_asm_add_expr ExprOp("+", [e0; e1]) ... =
   insns1,op1 = ast_to_asm_expr e1 ... ;
   insns0,op0 = ast_to_asm_expr e0 ... ;
   m = (* a slot on the stack *);
   (insns1
    + [ str op1,m ]
    + insns0
    + [ mov x0,op0;
        ldr x1,m;
        add x0,x0,x1 ],
   x0)
```

Compiling an expression (comparison)

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

Compiling an expression (comparison)

- ≈ 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)

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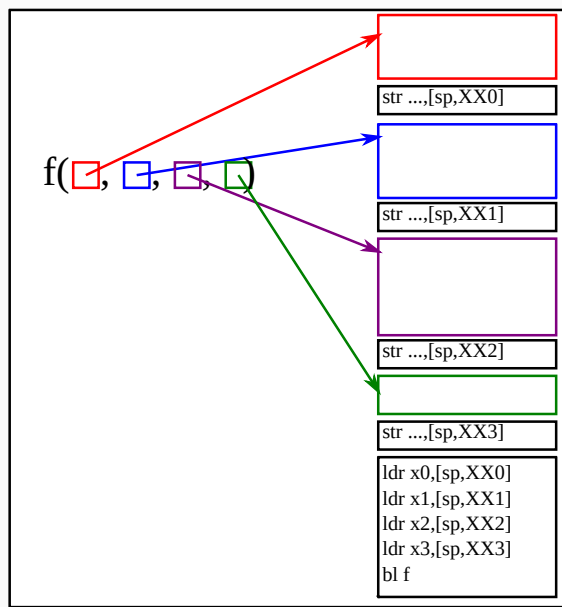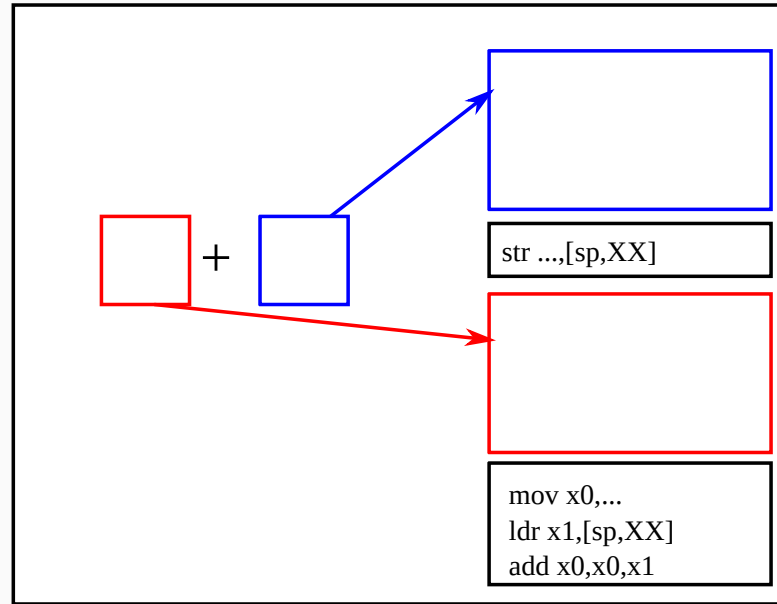


```
ast_to_asm_call_expr (ExprCall(f, [e0;e1;...])) ... =
    [(i0,op0);(i1,op1);..],[m0;m1;...]
        = ast_to_asm_exprs [e0;e1;...] ...;
    (   (i0 + [str op0,m0])
     + (i1 + [str op1,m1])
     + ...
     + [ldr x0,m0;
        ldr x1,m1;
        ...;
        bl f],
    x0)
```

A few left-out details

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

str ...,[sp,XX]

mov x0,...
ldr x1,[sp,XX]
add x0,x0,x1

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

available

used

v

sp →
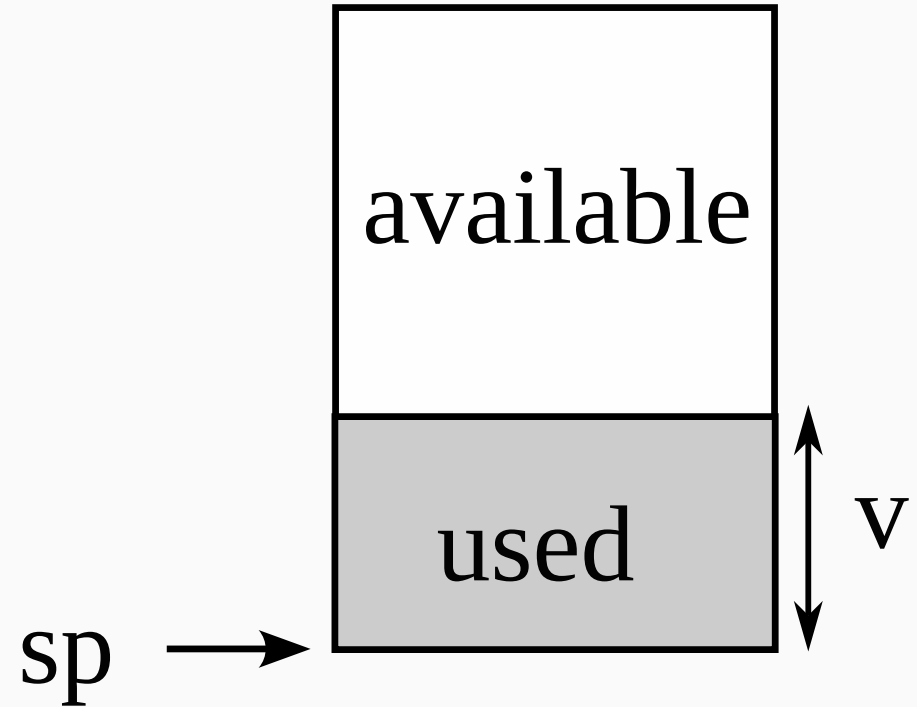
Determining where to save subexpressions

- when evaluating $A + B$,
  1. evaluate B, using SP+v and higher; save the result at SP+v
  2. evaluate A, using $v + 8$ and higher addresses
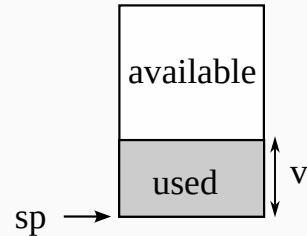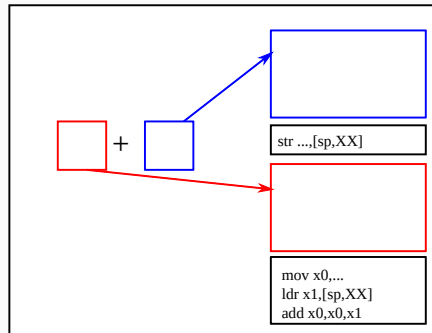
```
ast_to_asm_add_expr ExprOp("+",[e0;e1]) v ... =
    insns1,op1 = ast_to_asm_expr e1  v     ... ;
    insns0,op0 = ast_to_asm_expr e0 (v+8) ... ;
    (  insns1
     + [ str op1,[sp,v] ]
     + insns0
     + [ mov x0,op0;
         ldr x1,[sp,v];
         add x0,x0,x1],
     x0)
```

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
- → 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 (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 $x \mapsto$ `loc` in addition to `env`
- `(env', v') = env_extend decls loc env`
  - ▸ can be easily built on `env_add` (left for you)