1. **Interpreters and Language Semantics** (18 points)

Appendix A contains the complete code for a variant of one of the SIMPLE imperative language interpreters we explored in class. This version includes a new "for-loop" construct. The new abstract syntax (on line 15) is `For(x, e, c)`, which is intended to have a semantics like the following c program:

```
for(x=0; x != e; x=x+1) {
  c;
}
```

Here, `x` is a variable, `e` is an expression, and `c` is the body of the loop. (Unlike more fully-featured for-loops, this version only allows for `x` to be incremented by 1 and requires the loop guard to test inequality with respect to the expression `e`.)

(a) (4 points) The OCaml definition `factorial_for` gives the abstract syntax tree for a SIMPLE program that uses a for loop. Translate it into c (-like) concrete syntax. (Since all of the types are integers, there is no need to mention type information.)

(b) (4 points) Consider the program `For("x", Imm 5, c)`. Does the semantics of `For` loops given by `interp_cmd` ensure that the command `c` will be executed exactly 5 times? Why or why not?

(c) (6 points) The `interp_cmd` function in Appendix A uses OCaml meta-level constructs to implement the loop semantics (see lines 49–59). An alternative is to "desugar" the `For` abstract syntax node into an equivalent command that does not mention `For` and then interpret that (as is done to handle `WhileNZ` on line 47). What code would replace lines 50–59 to desugar `For(x, e, c)`?

```
49      | For(x, e, c) ->
```

(d) (4 points) Suppose that instead of *interpreting* this language, we were *compiling* it to LLVM IR. The parser could desugar `For` immediately, so that we do not even need to include it as part of the abstract syntax. Describe one reason why the compiler might *not* want to do that.

2. **X86 and Calling Conventions** (16 points)

The following code computes the trace (i.e., the sum of the diagonal entries) of a square matrix, in C (left), and X86 assembly (right). (Recall that the c type `long` is a 64-bit integer value.)

```
long trace(unsigned long n,
           long m[n][n]) {
  long i;
  long result = 0;
  for (i = 0; i < n; i++) {
    result += m[i][i];
  }
  return result;
}
```

```
trace:
        movq  $0,     %rax
        movq  $0,     %rdx
        movq  %rsi,   %rcx
        movq  %rdi,   %r10
        imulq $8,     %r10
        addq  $8,     %r10
loop:
        cmpq  %rdi,   %rdx
        jl    body
        retq

body:
        addq  (%rcx), %rax
        addq  $1,     %rdx
        addq  %r10,   %rcx
        jmp   loop
```

(a) The parameter `n` is passed to `trace` in the following location (choose one)

☐ `-16(%rbp)` ☐ `%rdi` ☐ `%rsi` ☐ `%rax`

(b) ☐ True or ☐ False: The contents of the matrix `m` are laid out contiguously in memory

(c) The value of `i` is stored in the following location (choose one)

☐ `-16(%rbp)` ☐ `%rax` ☐ `%rdx` ☐ `%r10`

(d) Suppose that we call `trace` with `n=3`. When `trace` exits, the value stored in `rcx` is equal to which c-language expression? (choose one)

☐ `2` ☐ `((long*)m + 12)` ☐ `m[2][2]` ☐ `((long*)m + 96)`

(e) ☐ True or ☐ False: According to the cdecl standard, `%rbp` is a callee-save register.

(f) ☐ True or ☐ False: X86 code is structured as *basic blocks*, with labeled entry points, straight-line code, and terminator instructions.

(g) (4 points) Write a sequence of X86lite instructions that has the same effect as `pushq %rcx` *without* using `pushq`:

3. **LLVM IR** (20 points)

Consider the following C code (left), and corresponding LLVMlite (right):

```
struct C { int64_t x; int64_t y; };
struct A { int64_t head[10];
           struct C c;
           int64_t foot[10]; };
```

```
%C = { i64, i64 }
%A = { [10 x i64],
       %C,
       [10 x i64] }
```

(a) (2 points) How many bytes does an LLVM value of type `[5 x %C*]` occupy?

(b) (2 points) How many bytes does an LLVM value of type `[5 x %A]` occupy?

(c) (10 points) Consider the following c statement (where `a` is of type `A*`):

```
int64_t x = (*a).foot[3] + (*a).c.y;
```

Suppose that `%a` is the LLVM uid of type `%A*` corresponding to the value stored in `a`. Fill in the blanks of the following LLVM IR snippet so that it corresponds to the c statement above. (The comments indicate the what sort of missing information should be filled in.)

```
%x    = alloca ___________   ; LLVM type

%tmp1 = getelementptr %A* %a, _________________________________________ ; GEP path

%val1 = load i64, _____________________ ; LLVM type and uid

%tmp2 = getelementptr %A* %a, _________________________________________ ; GEP path

%val2 = load i64, _____________________ ; LLVM type and uid

%val3 = ____________ i64 %val1, %val2  ; LLVM binop instruction

store i64 %val3, ____________ %x ; LLVM type
```

(d) (4 points) In your LLVMlite-to-x86lite code generator, the stack layout assigns each uid a stack slot that is referenced by a constant (negative) offset from rbp. Could you instead reference stack slots by a constant (positive) offset from rsp? Why or why not?

(e) (2 points) In a LLVM control flow graph with N vertices, what is the maximum number of edges?

4. **Lexing** (16 points)

Appendix B has the ocamllex code for a simple lexer program based on the ones we have seen in class.

(a) (2 points each) For each of the following input strings provided on stdin, what output sequence would be printed by the lexer? If the lexer generates an error anywhere during its operation, write "Char X is unexpected." where X is the illegal character. (Unlike the real program, which prints one token per line, you can put the output on one line.) We have done the first one for you.

i. `"012 x"`
Output: `Int 12     Ident x`

ii. `"if0x"`
Output:

iii. `"0ifx"`
Output:

iv. `"if 001(*"`
Output:

v. `"(**"`
Output:

(b) (4 points) Suppose that you wanted to modify the lexer so that identifiers can (but do not have to) start with an underscore. Which line (or lines) of the lexer code would you modify and what change would you make?

(c) (4 points) Is it possible for the DFA corresponding to an NFA to have *fewer* states? If so, give an example. If not, briefly explain why.

5. **Parsing** (20 points)

(a) (3 points) Is it possible to construct a context free grammar that accepts exactly the set of well-formed LLVM IR programs? Why or why not?

Consider the following context free grammar with terminals $\{x, y, z, \$\}$ and non-terminals S', S and T. S is the starting nonterminal and $ is the end-of-input marker.

$$
\begin{array}{rcl}
S' & \longrightarrow & S\$ \\
S & \longrightarrow & xST \\
S & \longrightarrow & \epsilon \\
T & \longrightarrow & yT \\
T & \longrightarrow & yz
\end{array}
$$

(b) (3 points) Which of the following strings are accepted by this grammar? (Mark all such strings)

☐ ϵ ☐ xyz ☐ xzy ☐ $xyzyz$ ☐ $xxyzyz$

(c) (3 points) Is this grammar LL(1)? Why or why not?

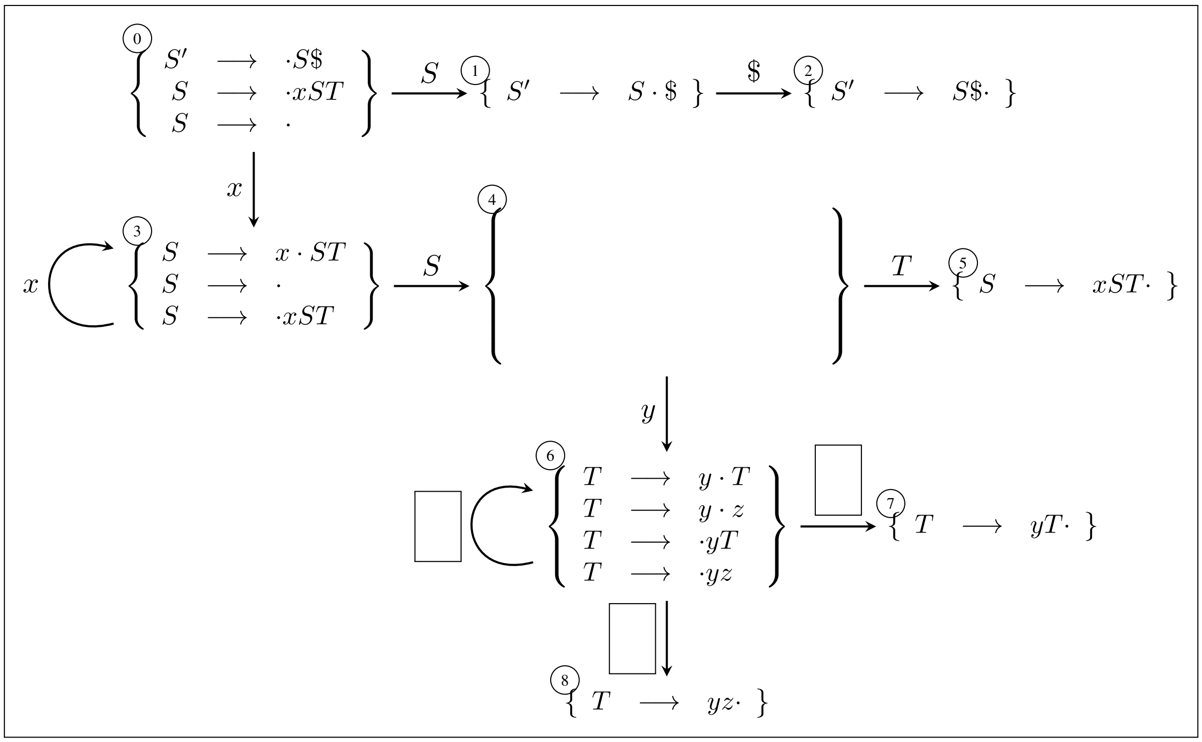
(d) (2 points) What is Follow(S), the follow set of nonterminal S, for this grammar?

Follow(S) =

For your reference, here is the same grammar again:

$$\begin{array}{rcl} S' & \longrightarrow & S\$ \\ S & \longrightarrow & xST \\ S & \longrightarrow & \epsilon \\ T & \longrightarrow & yT \\ T & \longrightarrow & yz \end{array}$$

(e) (6 points) Complete the DFA for the state space corresponding to the LR(0) parser for this grammar. Each state is numbered and consists of a set of LR(0) items. State ⓪ is the start state. You need to fill in the items for state ④ and the missing labels for the three edges originating at state ⑥. *Hint:* the items in state ④ do not indicate any reduce actions.



(b) (3 points) This grammar is *not* LR(0) but it *is* SLR(1). Briefly explain why.

Appendix A: Interpreter Code

```
type var = string

type exp =
  | Var of var
  | Imm of int
  | Add of exp * exp
  | Mul of exp * exp

type cmd =
  | Skip
  | Assn of var * exp
  | Seq of cmd * cmd
  | IfNZ of exp * cmd * cmd
  | WhileNZ of exp * cmd
  | For of var * exp * cmd      (* <------ This is the new construct *)

type state = (var * int) list

let set (s:state) (x:var) (v:int) =
  (x,v)::s

let rec get (s:state) (x:var) : int =
  begin match s with
    | [] -> 0   (* uninitialized variables are 0 *)
    | (y,v)::rest -> if x = y then v else get rest x
  end

let rec interp_exp (e:exp) (s:state) : int =
  begin match e with
    | Var x -> get s x
    | Imm v -> v
    | Add(e1, e2) -> (interp_exp e1 s) + (interp_exp e2 s)
    | Mul(e1, e2) -> (interp_exp e1 s) * (interp_exp e2 s)
  end
```

```
let rec interp_cmd (c:cmd) (s:state) : state =
  begin match c with
    | Skip -> s

    | Assn(x, e)   -> set s x (interp_exp e s)

    | Seq(c1, c2)  -> interp_cmd c2 (interp_cmd c1 s)

    | IfNZ(e, c1, c2) ->
      interp_cmd (if (interp_exp e s) <> 0 then c1 else c2) s

    | WhileNZ(e, c) ->
      interp_cmd (IfNZ(e, Seq(c, WhileNZ(e, c)), Skip)) s

    | For(x, e, c) ->
      let s0 = set s x 0 in
      let rec loop s =
        let e = interp_exp e s in
        let v = get s x in
        if v = e then s else
          let s' = interp_cmd c s in
          let v' = get s' x in
          loop (set s' x (v'+1))
      in
      loop s0
  end

(* The cmd [factorial_for] computes factorial of 6 using a for loop
   (and the available SIMPLE arithmetic instructions):  *)

let factorial_for : cmd =
  let x = "x" in
  let ans = "ans" in
  Seq(Assn(x, Imm 6),
      Seq(Assn(ans, Imm 1),
          For("y", Var x,
              Assn(ans, Mul(Var ans, (Add(Var x, Mul(Var "y", Imm(-1)))))))))
```

Appendix B: Lexer Code

```
{
open Lexing
type token =  | Int    of int64  | Ident  of string  | LPAREN
              | LPARENSTAR  | STARRPAREN  | IF
let print_token t =
  begin match t with
    | Int x   -> (Printf.printf "Int %Ld\n%!" x)
    | Ident s -> (Printf.printf "Ident  %s\n%!" s)
    | IF      -> (Printf.printf "IF\n%!")
    | LPAREN  -> (Printf.printf "LPAREN\n%!")
    | LPARENSTAR -> (Printf.printf "LPARENSTAR\n%!")
    | STARRPAREN -> (Printf.printf "STARRPAREN\n%!")
  end
let acc = ref []
let emit t = acc := t::(!acc)
exception Lex_error of char
}

let character = ['a'-'z''A'-'Z']
let digit = ['0'-'9']
let underscore = ['_']
let whitespace = [' ' '\t' '\n' '\r']
let identifier =  character (character|digit|underscore)*

rule lex = parse
  | "if"          { emit IF; lex lexbuf }
  | identifier    { emit (Ident (lexeme lexbuf)); lex lexbuf }
  | '('           { emit LPAREN; lex lexbuf }
  | "(*"          { emit LPARENSTAR; lex lexbuf }
  | "*)"          { emit STARRPAREN; lex lexbuf }
  | whitespace+   { lex lexbuf }
  | digit+        { emit (Int (Int64.of_string (lexeme lexbuf))); lex lexbuf }
  | eof           { List.rev (!acc) }
  | _ as c        { raise (Lex_error c) }

{
let _ =
  try
    List.iter print_token (lex (from_channel stdin))
  with
    | Lex_error c -> Printf.printf "Char %s is unexpected.\n" (Char.escaped c)
}
```