SOLUTIONS

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.)

```
x = 6;
ans = 1;
for(y=0; y != x; y = y+1) {
   ans = ans * (x + (y * -1));
}
```

(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?

Answer: No—there is no guarantee that the command `c` does not modify the value of the variable `"x"`. For instance if `c` is `x = x + (-2)` then the loop will not terminate.

(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) ->
50        interp_cmd
51          (Seq(Assn(x, Imm 0),
52                WhileNZ(Add(e, Mul(Var x, Imm(-1))),
53                Seq(c, Assn(x, Add(Var x, Imm(1)))))))
54        s
```

(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.

Answer: Keeping the structure of the `For` loop might make some program analyses and optimization easier. For example, if the loop variable is *not* modified by the body `c` of the loop and the loop guard is an `Imm n` value, then the frontend can replace the for loop with its unrolling `c ; c ; c ; ... c;` (n times), which might enable further optimizations.

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.)

```c
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)`

Answer: C semantics takes into account the size of the objects in the array, so offset 12 from a `long*` pointer is 8×12 bytes from base.

(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`:

```
subq $8, %rsp
movq %rcx, (%rsp)
```

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?

$5 \times 8 = 40$

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

$5 \times (8 \times (10 + 2 + 10)) = 880$

(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 i64
%tmp1 = getelementptr %A* %a, i32 0, i32 2, i64 3
%val1 = load i64, i64* %tmp1
%tmp2 = getelementptr %A* %a, i32 0, i32 1, i32 1
%val2 = load i64, i64* %tmp2
%val3 = add i64 %val1, %val2
store i64 %val3, i64* %x
```

(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?

Answer: No—stack slots cannot be referenced by a constant offset from `rsp`, because the offset from `rsp` can change dynamically when memory is allocated using the `alloca` instruction.

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

2N

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: `Ident if0x`

iii. `"0ifx"`

Output: `If 0    Ident ifx`

iv. `"if 001(*"`

Output: `IF    Int 1    LPARENSTAR`

v. `"(**"`

Output: `Char * is unexpected.`

(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?

Answer: There are at least two ways (but way 1 is simpler):

1. Change line 23 to be:

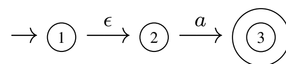
```
let identifier = (character|underscore) (character|digit|underscore)*
```

2. Change line 27 to be :

```
| '_'(char|digit|'_')* | identifier  emit (Ident (lexeme lexbuf)); lex lexbuf
```

(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.

Answer: It *is* possible. For instance, there might be "redundant" states reached by ϵ transitions. For example the following NFA is equivalent to a two-state DFA that starts at state ②. Both recognize just the language $\{a\}$.



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?

Answer: No—there are some well-formedness constraints on LLVM IR programs, such as typing, the SSA property, and the need for jump targets to be valid labels that are not context free.

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?

Answer: No—the terminal y appears in First set for both production rules of the nonterminal T.

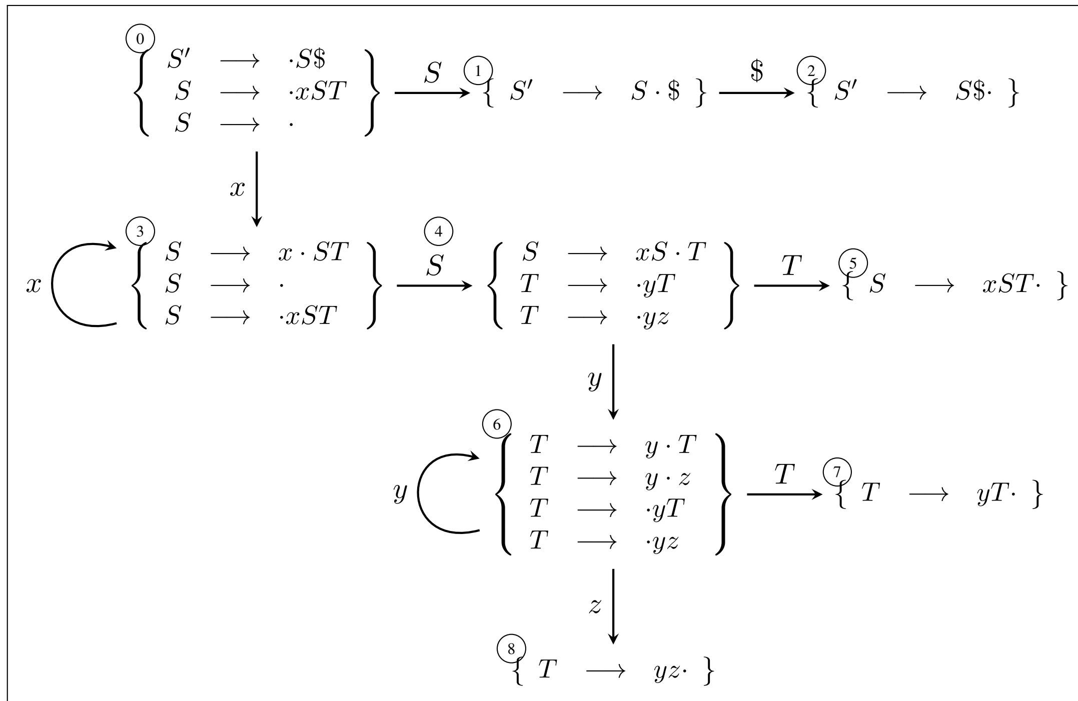
(d) (2 points) What is Follow(S), the follow set of nonterminal S, for this grammar?

Follow(S) = $\{\$, y\}$

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.

Answer: States ⓪ and ③ have shift-reduce conflicts, so this is not LR(0). However, because Follow(S) is $\{\$, y\}$ we can shift on lookahead x and reduce on \$ or y.

CIS341 Midterm 2022 Appendices

(Do not write answers in the appendices. They will not be graded)

Appendix A: Interpreter Code

```
1  type var = string
2
3  type exp =
4    | Var of var
5    | Imm of int
6    | Add of exp * exp
7    | Mul of exp * exp
8
9  type cmd =
10   | Skip
11   | Assn of var * exp
12   | Seq of cmd * cmd
13   | IfNZ of exp * cmd * cmd
14   | WhileNZ of exp * cmd
15   | For of var * exp * cmd      (* <----- This is the new construct *)
16
17 type state = (var * int) list
18
19 let set (s:state) (x:var) (v:int) =
20   (x,v)::s
21
22 let rec get (s:state) (x:var) : int =
23   begin match s with
24     | [] -> 0   (* uninitialized variables are 0 *)
25     | (y,v)::rest -> if x = y then v else get rest x
26   end
27
28 let rec interp_exp (e:exp) (s:state) : int =
29   begin match e with
30     | Var x -> get s x
31     | Imm v -> v
32     | Add(e1, e2) -> (interp_exp e1 s) + (interp_exp e2 s)
33     | Mul(e1, e2) -> (interp_exp e1 s) * (interp_exp e2 s)
34   end
```

```
35 let rec interp_cmd (c:cmd) (s:state) : state =
36   begin match c with
37     | Skip -> s
38
39     | Assn(x, e)   -> set s x (interp_exp e s)
40
41     | Seq(c1, c2)  -> interp_cmd c2 (interp_cmd c1 s)
42
43     | IfNZ(e, c1, c2) ->
44       interp_cmd (if (interp_exp e s) <> 0 then c1 else c2) s
45
46     | WhileNZ(e, c) ->
47       interp_cmd (IfNZ(e, Seq(c, WhileNZ(e, c)), Skip)) s
48
49     | For(x, e, c) ->
50       let s0 = set s x 0 in
51       let rec loop s =
52         let e = interp_exp e s in
53         let v = get s x in
54         if v = e then s else
55           let s' = interp_cmd c s in
56           let v' = get s' x in
57           loop (set s' x (v'+1))
58       in
59       loop s0
60   end
61
62 (* The cmd [factorial_for] computes factorial of 6 using a for loop
63    (and the available SIMPLE arithmetic instructions):  *)
64
65 let factorial_for : cmd =
66   let x = "x" in
67   let ans = "ans" in
68   Seq(Assn(x, Imm 6),
69       Seq(Assn(ans, Imm 1),
70           For("y", Var x,
71               Assn(ans, Mul(Var ans, (Add(Var x, Mul(Var "y", Imm(-1))))))))
```

Appendix B: Lexer Code

```
1  {
2  open Lexing
3  type token =  | Int    of int64  | Ident  of string  | LPAREN
4                | LPARENSTAR  | STARRPAREN  | IF
5  let print_token t =
6    begin match t with
7      | Int x   -> (Printf.printf "Int %Ld\n%!" x)
8      | Ident s -> (Printf.printf "Ident  %s\n%!" s)
9      | IF      -> (Printf.printf "IF\n%!")
10     | LPAREN  -> (Printf.printf "LPAREN\n%!")
11     | LPARENSTAR -> (Printf.printf "LPARENSTAR\n%!")
12     | STARRPAREN -> (Printf.printf "STARRPAREN\n%!")
13   end
14 let acc = ref []
15 let emit t = acc := t::(!acc)
16 exception Lex_error of char
17 }
18
19 let character = ['a'-'z''A'-'Z']
20 let digit = ['0'-'9']
21 let underscore = ['_']
22 let whitespace = [' ' '\t' '\n' '\r']
23 let identifier =  character (character|digit|underscore)*
24
25 rule lex = parse
26   | "if"           { emit IF; lex lexbuf }
27   | identifier     { emit (Ident (lexeme lexbuf)); lex lexbuf }
28   | '('            { emit LPAREN; lex lexbuf }
29   | "(*"           { emit LPARENSTAR; lex lexbuf }
30   | "*)"           { emit STARRPAREN; lex lexbuf }
31   | whitespace+    { lex lexbuf }
32   | digit+         { emit (Int (Int64.of_string (lexeme lexbuf))); lex lexbuf }
33   | eof            { List.rev (!acc) }
34   | _ as c         { raise (Lex_error c) }
35
36 {
37 let _ =
38   try
39     List.iter print_token (lex (from_channel stdin))
40   with
41     | Lex_error c -> Printf.printf "Char %s is unexpected.\n" (Char.escaped c)
42 }
```