CIS 341 Midterm 27 February 2020

SOLUTIONS

1. Compiler Structure (4 points)

a. Which stage of a C compiler is responsible for generating an error message like:
"expected an expression"

☐ Lexer ☒ Parser ☐ Static Analyzer ☐ Code Generator ☐ Assembler

b. Which stage of a C compiler is responsible for generating an error message like:
"error: use of undeclared identifier 'z'"

☐ Lexer ☐ Parser ☒ Static Analyzer ☐ Code Generator ☐ Assembler

c. In which stage of a C compiler would regular expressions be most useful for the language implementor?

☒ Lexer ☐ Parser ☐ Static Analyzer ☐ Code Generator ☐ Assembler

d. Which stage of a C compiler is in charge of implementing the platform's calling conventions?

☐ Lexer ☐ Parser ☐ Static Analyzer ☒ Code Generator ☐ Assembler

2. Interpreters and Language Semantics (20 points)

We saw in the first homework how it is easy to implement an interpreter for a simple language of arithmetic expressions. Here we explore the ramifications of including"impure expressions," as are found in languages like C.

For example, in C you can write a function `foo`:

```
int foo() {
  int x = 0;
  int ans = x + (x = 1);
  return ans;
}
```

Here, the expression `x + (x = 1)` contains an assignment to `x`. In general, the meaning of an assignment expression `x =` exp is to compute the result r of evaluating exp, assign r to the variable `x` and yield r as the result of the whole assignment expression.

This is a significant change to the semantics of expressions, because it makes the order of evaluation observable: the result of exp_1 + exp_2 might depend on whether we evaluate exp_1 or exp_2 first. (C leaves this choice unspecified—the evaluation order is up to the compiler implementor, and might vary from one implementation to the next![1])

a. (1 point) What answer does `foo()` return if exp_1 + exp_2 evaluates exp_1 before exp_2? 1.

b. (1 point) What answer does `foo()` return if exp_1 + exp_2 evaluates exp_2 before exp_1? 2.

c. (8 points) We can model this situation by extending the interpreter from the Hellocaml project. Appendix A has the code for an expression datatype that includes a new constructor `VarAssn`. Complete the interpreter below so that expressions are evaluated left-to-right and the result is the value of the expression paired with the updated state.

```
let rec interpret_exp (s:state) (e:exp) : (int * state) =
  begin match e with
    | Var x -> (lookup s x, s)
    | Add(e1, e2) ->
       let (l, s1) = interpret_exp s e1 in
       let (r, s2) = interpret_exp s1 e2 in
       (l + r, s2)
    | Mul(e1, e2) ->
       let (l, s1) = interpret_exp s e1 in
       let (r, s2) = interpret_exp s1 e2 in
       (l * r, s2)
    | Lit i -> (i, s)
    | VarAssn(x, e) ->
       let (r, s1) = interpret_exp s e in
       (r, update s1 x r)
  end
```

[1] To be fair, most modern C implementations issue a warning if you use these features.

d. (6 points) This change to the meaning of expressions also affects which optimizations and code transformations are correct (i.e. do not change the meaning of the program). For example, transforming the expression `x + (x = 1)` to `(x = 1) + x` is *not* correct (so plus is not commutative).

Consider the following optimizer based on that from the homework. For each indicated case of the code, mark the box to say whether that line is OK (i.e. is a correct transformation) or BAD (i.e. may not be correct). Note that the `VarAssn` case will be treated in part **e**.

```
                let rec optimize (e:exp) : exp =
                  begin match e with
OK[X]  BAD[ ]       | Var _ -> e
OK[X]  BAD[ ]       | Lit _ -> e
                    | Add (e1, e2) ->
                      begin match (optimize e1, optimize e2) with
OK[X]  BAD[ ]           | (Lit x1, Lit x2) -> Lit (x1 + x2)
OK[X]  BAD[ ]           | (Lit 0, o2) -> o2
OK[X]  BAD[ ]           | (o1, Lit 0) -> o1
OK[X]  BAD[ ]           | (o1, o2) -> Add(o1, o2)
                      end
                    | Mul (e1, e2) ->
                      begin match (optimize e1, optimize e2) with
OK[X]  BAD[ ]           | (Lit x1, Lit x2) -> Lit (x1 * x2)
OK[ ]  BAD[X]           | (Lit 0, _) -> Lit 0
OK[ ]  BAD[X]           | (_, Lit 0) -> Lit 0
OK[X]  BAD[ ]           | (Lit 1, o2) -> o2
OK[X]  BAD[ ]           | (o1, Lit 1) -> o1
OK[X]  BAD[ ]           | (o1, o2) -> Mul(o1, o2)
                      end

                    | VarAssn(x,e1) -> (* handled in part e *)
                  end
```

e. (4 points) In the blank parts of the code below, implement one correct, non-trivial optimization that can be applied to the `VarAssn` expression. You'll need to fill in the pattern and the right-hand-side of the case. Your optimization should (at least in some circumstances) produce a smaller expression.

Answer: We provide two acceptable optimizations.

```
let rec optimize (e:exp) : exp =
    (* ... *)
    | VarAssn(x,e1) ->
      begin match (optimize e1) with

        | VarAssn(y,o1) ->
              if x = y then VarAssn(x, o1)
              else VarAssn(x,VarAssn(y,o1))

        | Var y ->
              if x = y then Var x
              else VarAssn(x, Var y)

        | o1 -> VarAssn(x, o1)
      end
  end
```

3. X86 and Calling Conventions (22 points)

Recall that according to the x86-64 calling conventions that we have been using, the first six arguments to a function are passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` and later arguments are passed on the stack in reverse order. The function result is returned in `%rax`.

Appendix B shows a C program that computes factorial, along with X86 code that might result from compiling the program.

a. (1 point) Which X86 code label corresponds to the program point between lines 4 and 5 of the C program (i.e. the start of the "then" branch)?

☐ `lbl_a` ☒ `lbl_b` ☐ `lbl_c`

b. (1 point) Which X86 code label corresponds to the program point between lines 6 and 7 of the C program (i.e. the start of the "else" branch)?

☒ `lbl_a` ☐ `lbl_b` ☐ `lbl_c`

c. (2 points) Suppose that we replace the `subq` instruction on line 13 of the X86 code with the following:
`leaq -1(%rcx), %rcx`
Would the program's behavior be changed? (Briefly explain why or why not.)

No: `subq $1, %rcx` subtracts 1 from the contents of `%rcx` and so does the `leaq` instruction with those operands .

d. (2 points) Suppose that the main code has the following instructions:

```
_main:
    ...
    movq    $3, %rsi
    movq    $5, %rdi
    callq   _factorial
```

Assuming that addresses occupy 8 bytes, how many total bytes of stack space will be used during this call to factorial? (Include the space used by recursive calls and any consumed by the calling conventions, including the saved return address.)

☐ 96 bytes (96 = 3 × 32)
☐ 120 bytes (120 = 3 × 40)
☐ 128 bytes (128 = 3 × 48)
☐ some other amount
☐ 160 bytes (160 = 5 × 32)
☐ 200 bytes (200 = 5 × 40)
☒ 240 bytes (240 = 5 × 48)

The X86 calling conventions that we have used so far assume that the caller will do some useful work after calling the function but before returning (i.e. to use the result of the called function in some interesting way.) In the case that the caller immediately returns the result of the call—i.e. the call is in *tail position*—a more efficient implementation is often possible.

For example, when the body of `f` ends in `return g(...);`, the unoptimized code would look like:

```
f:
    pushq   %rbp            ## save base pointer
    movq    %rsp, %rbp      ## set up local stack frame
    ...                     ## set up function args
    callq   g               ## (tail) call to g
    movq    %rax, %rax      ## move result of g into return register
    popq    %rbp            ## restore base pointer
    retq
```

Assuming that `g` follows the calling conventions too, this kind of tail call can often be optimized by using a `jmp` instruction, so the sequence above would become:

```
f:
    pushq   %rbp            ## save base pointer
    movq    %rsp, %rbp      ## set up local stack frame
    ...                     ## set up function args
    popq    %rbp            ## restore base pointer (early!)
    jmp     g               ## optimized tail call to g
```

Choose *one* answer:

e. (2 points) Under what circumstances is removing `movq op1, op1` (e.g. where `op1` is `%rax`) guaranteed to not change the behavior of the program?

- ☐ It is *always* correct to do this optimization.
- ☐ It is correct if the code segment of memory is read only (i.e. can't be written or executed).
- ☒ It is correct if the code segment of memory is execute only (it can't be read or written).
- ☐ It is correct if the code segment of memory is write only (it can't be read or executed).

f. (2 points) Why is it correct to replace the `callq g` and subsequent `retq` with `jmp g`?

- ☒ When `g` returns, `f`'s return address will still be on the top of the stack, so control will be passed back to `f`'s caller, as required.
- ☐ When `g` returns, `g`'s return address will still be on the top of the stack, so control will be passed back to `f`, as required.
- ☐ Since `f` does not return, neither will `g`.
- ☐ Since `callq` pushes a return address and `retq` pops one, it is always safe to remove `callq`/`retq` pairs like in this example.

g. (2 points) Why is it correct to move the `popq` instruction before the `jmp` to `g` ?

- ☐ `%rbp` is not used for function arguments and it is a *caller* save register.
- ☒ `%rbp` is not used for function arguments and it is a *callee* save register.
- ☐ The code in `g` will leave `%rsp` unchanged.
- ☐ The instruction `popq %rbp` is equivalent to `addq %rsp, $8`.

Now consider the following C program that implements a tail recursive version of factorial. It takes two inputs, `n` (the usual input), and `acc` (an accumulator). It returns $n! \times$ `acc`, so we can compute factorial of `n` by starting with an accumulator of `1`, that is by doing `facttail(n, 1)`.

```
int64_t facttail(int64_t n, int64_t acc) {
  if (n <= 1) {
    return acc;
  } else {
    return facttail(n - 1, n * acc);
  }
}
```

Note that the recursive use of `facttail` is itself a tail call, which means that it can be compiled via `jmp` as explained above. Moreover, any function that does not need to allocate stack space (and does not modify `%rbp`) doesn't need to set up the stack frame. With a bit of smart register usage, this means that the program above can be optimized to the following code, where we have omitted some of the operands. (Note that this is an efficient loop and much shorter than the non-tail version!)

h. (8 points) Fill in the blanks below with operands to complete the optimized implementation of `facttail`. It should comply with the X64 calling conventions. You will need to use the operands: `$1`, `%rsi`, `%rdi`, and `%rax` (perhaps multiple times each):

```
_facttail:                              ## @facttail
        cmpq    $1, %rdi
        j gt    then                    ##  Jump if "greater than"
        jmp     else
then:
        imulq   %rdi, %rsi
        subq    $1, %rdi
        jmp     _facttail
else:
        movq    %rsi, %rax
        retq
```

i. (2 points) Suppose that the main code has the following instructions.

```
_main:
    ...
    movq    $1, %rsi
    movq    $5, %rdi
    popq    %rbp
    jmp     _facttail
```

Assuming that addresses occupy 8 bytes, how many total bytes of stack space will be used during this call to `facttail`?

- ☒ 0 bytes
- ☐ 8 bytes
- ☐ 16 bytes
- ☐ 40 bytes ($40 = 8 \times 5$)
- ☐ some other amount

4. LLVM IR (20 points)

Consider the two LLVM IR types shown below.

```
%A = type { i64, %B* }
%B = type { %A, [3 x %A], %A }
```

Assume that (as in our LLVM lite) pointers are 8 bytes wide and that all data is 8-byte aligned (padding will not play a role in these questions).

a. (2 points) How many bytes will be allocated on the stack by the LLVM instruction `%v = alloca %A`?

16

b. (2 points) How many bytes will be allocated on the stack by the LLVM instruction `%w = alloca %B`? *Below,* `sizeof(%A)` *stands for your answer from part* ***a****.*

5× `sizeof(%A)`

c. (2 points) Assuming that uid `%a` contains a base pointer of type `%A*`, what is the type of uid `%ptr` calculated by the instruction:

```
%ptr = getelementptr %A, %A* %a, i32 0, i32 1
```

☐ `%A*` ☐ `%B*` ☐ `%A**` ☒ `%B**`

The types above were obtained by compiling the following C program.

```
1  struct B;  // forward declaration
2  struct A {int64_t x; struct B* y;};
3  struct B {struct A f, g[3], h; };
4
5  void foo() {
6    struct A a;
7    struct B b;
8    a.y = &b;
9    b.g[2] = a;
10   b.g[2].x = 341;
11 }
```

d. (2 points) Assuming that uid `%tmp` contains the base pointer of type `%A*`, which of the following `getelementptr` instructions would be used to calculate the address assigned to on line 8?

☐ `%ptr = getelementptr %A, %A* %tmp, i32 1`

☒ `%ptr = getelementptr %A, %A* %tmp, i32 0, i32 1`

☐ `%ptr = getelementptr %A, %A* %tmp, i32 0, i32 2`

☐ `%ptr = getelementptr %A, %A* %tmp, i32 1, i32 0`

e. (2 points) Assuming that uid `%tmp` contains the base pointer of type `%B*`, which of the following `getelementptr` instructions would be used to calculate the address assigned to on line 10?

☐ `%ptr = getelementptr %B, %B* %tmp, i32 1, i32 1, i32 2`

☐ `%ptr = getelementptr %B, %B* %tmp, i32 1, i32 2, i32 0`

☐ `%ptr = getelementptr %B, %B* %tmp, i32 0, i32 2, i32 1, i32 0`

☒ `%ptr = getelementptr %B, %B* %tmp, i32 0, i32 1, i32 2, i32 0`

Recall that the LLVM IR code for a function declaration is structured into a control flow graph (CFG), whose nodes consist of labeled basic blocks. There is an edge from one basic block to another if the *terminator* instruction of the first mentions label of the second. Appendix C contains the LLVM IR code for a version of the factorial function as shown in lecture.

e. (1 point)

True ☒ or False ☐

Rearranging the order in which a CFG's labeled blocks appear *does not* change the meaning of the LLVM code.

f. (3 points) The control-flow graphs of LLVM IR code are more structured than x86 assembly. Briefly describe a control-flow behavior expressible in an x86 program that *cannot* be represented using the LLVM (lite) IR used in this class.

In x86 you can do a computed jump, i.e. to jump to some offset of a label; that jump target need not be the entry point of a basic block. In x86 you can have "fallthrough" code that enters a basic block without a jump.

g. (3 points) The entry block of an LLVM IR function does *not* have its own label. Briefly explain why that design choice is justified.

This means that the only way for control to reach the entry block is via the `call` instruction. The function entry needs special treatment in the backend (i.e. to set up the stack frame and move in function arguments), so jumping to that code from elsewhere in the CFG would make compilation harder (you need to jump around the function preamble).

h. (3 points) Recall that most LLVM IR instructions are of the form `%uid = bop T %opnd1, %opnd2`, where `%uid` names the result and the operands can be literals or other, previously named, unique identifiers, but operands *do not* include nested arithmetic operations. Why is this a good structure for an intermediate representation? (Briefly explain.)

Translating to this form means that we have "named" the results of all intermediate computations, which has several benefits, including: it enforces a particular order of evaluation, intermediate results can be shared (e.g. by optimization), and this form is closer to the way that assembly code works, so it's easier to generate assembly from this form.

5. Lexing, Parsing, and Grammars (14 points)

a. (3 points) The context free grammar $S ::= \texttt{a}S \mid \varepsilon$, which generates the set of all strings consisting of only a's, is *un*ambiguous. Define an *ambiguous* grammar for the same language.

$S ::= \texttt{a} \mid SS \mid \varepsilon$ is one possible answer.

b. (2 points) Consider the following grammar where the $\diamond$ and $\star$ terminal symbols stand for infix binary operators (over integers).

$$\begin{array}{rcl} E & ::= & F \diamond E \mid F \\ F & ::= & F \star G \mid G \\ G & ::= & int \mid (E) \end{array}$$

- Which of $\diamond$ and $\star$ has *higher precedence*? ☐ $\diamond$ ☒ $\star$
- Which of $\diamond$ and $\star$ is *right associative*? ☒ $\diamond$ ☐ $\star$

c. (4 points) Consider the following grammar for the concrete syntax of a subset of OCaml expressions, where var represents variable names like x, y, z, etc.

$$E ::= var \mid \texttt{if } var \texttt{ then } E \mid \texttt{if } var \texttt{ then } E \texttt{ else } E$$

Show that this grammar is ambiguous by giving two different leftmost derivations for the string below. (This is known as the "dangling else" problem.) Underline the nonterminal being rewritten in each step.

"`if x then if y then y else z`"

Derivation 1:

$$\begin{array}{rl} \underline{E} \rightarrow & \texttt{if } x \texttt{ then } \underline{E} \\ \rightarrow & \texttt{if } x \texttt{ then if } y \texttt{ then } \underline{E} \texttt{ else } E \\ \rightarrow & \texttt{if } x \texttt{ then if } y \texttt{ then } y \texttt{ else } \underline{E} \\ \rightarrow & \texttt{if } x \texttt{ then if } y \texttt{ then } y \texttt{ else } z \end{array}$$

Derivation 2:

$$\begin{array}{rl} \underline{E} \rightarrow & \texttt{if } x \texttt{ then } \underline{E} \texttt{ else } E \\ \rightarrow & \texttt{if } x \texttt{ then if } y \texttt{ then } \underline{E} \texttt{ else } E \\ \rightarrow & \texttt{if } x \texttt{ then if } y \texttt{ then } y \texttt{ else } \underline{E} \\ \rightarrow & \texttt{if } x \texttt{ then if } y \texttt{ then } y \texttt{ else } z \end{array}$$

d. (5 points) Disambiguate the grammar above so that each `else` is associated with the closest `if` that does not already have an `else`. For example, for the string above, `else` z should be always associated with the `if` y, and never with `if` x.

Rewrite the grammar so that E_0 is the new start symbol and there are two additional nonterminals, E_1 and E_2. *Hint:* Think about how matched (i.e. with an `else`) and unmatched `if`'s can nest.

$$\begin{array}{rcl} E_0 & ::= & E_1 \mid E_2 \\ E_1 & ::= & \texttt{if } var \texttt{ then } E_0 \mid \texttt{if } var \texttt{ then } E_2 \texttt{ else } E_1 \\ E_2 & ::= & var \mid \texttt{if } var \texttt{ then } E_2 \texttt{ else } E_2 \end{array}$$

CIS341 Midterm 2020 Appendices

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

Appendix A: OCaml Code for Expressions

This code is a simple variant on the interpreters used in the first project. There is one new constructor for the `exp` type that represents "assignment expressions."

```
(* Variables are represented as strings *)
type var = string

(* Abstract syntax for expressions *)
type exp =
  | Var of var
  | Add of exp * exp
  | Mul of exp * exp
  | Lit of int
  (* New *)
  | VarAssn of var * exp

(* The type of states mapping each variable to an [int] *)
type state = var -> int

(* The initial state maps every variable to 0 *)
let init_state : state =
  fun x -> 0

(* Update an old state [s] to one that maps `x` to `v` but is otherwise
 * unchanged. *)
let update (s:state) (x:var) (v:int) : state =
  fun (y:var) ->
    if x = y then v else s y

(* Look up the value of a variable in a state: *)
let lookup (s:state) (x:var) : int = s x
```

Appendix B: X86 Factorial Code

The following C code defines a factorial function for 64-bit integers.

```c
#include<stdint.h>

int64_t factorial(int64_t n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * (factorial (n - 1));
  }
}
```

The following X86 assembly code is the result of compiling the above C program (without optimizations, and with some renaming of the block labels).

```
        .globl  _factorial              ## -- Begin function factorial
_factorial:                             ## @factorial
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $32, %rsp
        movq    %rdi, -16(%rbp)
        cmpq    $1, -16(%rbp)
        j gt    lbl_a
        jmp     lbl_b
lbl_a:
        movq    -16(%rbp), %rax
        movq    -16(%rbp), %rcx
        subq    $1, %rcx
        movq    %rcx, %rdi
        movq    %rax, -24(%rbp)
        callq   _factorial
        movq    -24(%rbp), %rcx
        imulq   %rax, %rcx
        movq    %rcx, -8(%rbp)
        jmp     lbl_c
lbl_b:
        movq    $1, -8(%rbp)
        jmp     lbl_c
lbl_c:
        movq    -8(%rbp), %rax
        addq    $32, %rsp
        popq    %rbp
        retq
```

Appendix C: LLVMLite IR

```
define i64 @factorial(i64 %n) {
  %1 = alloca i64
  %acc = alloca i64
  store i64 %n, i64* %1
  store i64 1, i64* %acc
  br label %loop

loop:
  %3 = load i64* %1
  %4 = icmp sgt i64 %3, 0
  br i1 %4, label %body, label %post

body:
  %6 = load i64* %acc
  %7 = load i64* %1
  %8 = mul nsw i64 %6, %7
  store i64 %8, i64* %acc
  %9 = load i64* %1
  %10 = sub nsw i64 %9, 1
  store i64 %10, i64* %1
  br label %loop

post:
  %12 = load i64* %acc
  ret i64 %12
}
```