EECS 483 Midterm March 12, 2024

SOLUTIONS

1. **Language Semantics and Correctness** (16 points)

   Appendix A contains a description of an arithmetic expression language with integers, addition, multiplication and division, and code for an interpreter.

   (a) (4 points) Which of the following rewrites are correct with respect to the language semantics given by `interpret_exp`? If the rewrite is incorrect, provide a concrete counter-example: a specific expression `e` for which `interpret_exp` has a different result when evaluated on the left and right expressions.

   i. rewrite `Add(e, e)` to `Mul(Lit 2, e)`

   Answer: Correct

   ii. rewrite `Mul(e, Lit 0)` to `Lit 0`

   Answer: Correct

   iii. rewrite `Mul(e1, e2)` to `Mul(e2, e1)`

   Answer: Correct

   (b) (4 points) Redo your analysis from part 1 but for a version of `interpret_exp` where the call to `divide_v1` is replaced by a call to `divide_v2`. If counter-example works as before, simply say so, no need to rewrite it.

   i. rewrite `Add(e, e)` to `Mul(Lit 2, e)`

   Answer: Correct

   ii. rewrite `Mul(e, Lit 0)` to `Lit 0`

   Answer: Incorrect, For `e: Div(1, 0)` the left will error and the right will not

   iii. rewrite `Mul(e1, e2)` to `Mul(e2, e1)`

   Answer: Correct

   (c) (4 points) Redo your analysis again but using `divide_v3`.

   i. rewrite `Add(e, e)` to `Mul(Lit 2, e)`

   Answer: Correct

   ii. rewrite `Mul(e, Lit 0)` to `Lit 0`

   Answer: Incorrect. For `e: Div(1, 0)` the left will error and the right will not

   iii. rewrite `Mul(e1, e2)` to `Mul(e2, e1)`

   Answer: Incorrect. For `e1: Div(1, 0), e2: Div(2, 0)`, the left will error with `DividedByZeroV3(1)` but the right will error with `DividedByZeroV3(2)`

   (d) (4 points) Redo your analysis again but using `divide_v4`.

i. rewrite `Add(e, e)` to `Mul(Lit 2, e)`

   Answer: Incorrect. For `e: Div(1, 0)`, the left will print `11`, but the right will print `1`

ii. rewrite `Mul(e, Lit 0)` to `Lit 0`

   Answer: Incorrect. For `e: Div(1, 0)`, the left will print `1`, but the right will print nothing

iii. rewrite `Mul(e1, e2)` to `Mul(e2, e1)`

   Answer: Incorrect. `e1: Div(1, 0), e2: Div(2, 0)`, the left will print `12` but the right will print `21`

2. **Calling Conventions and Assembly Code**(16 points)

   Appendix B contains a description of an unusual calling convention, the *CBPV calling convention*. For simplicity, we describe a version for functions of two arguments.

   (a) (2 points) Do the CBPV and AMD64 calling conventions use caller cleanup of arguments or callee cleanup? (Choose one)

   ☐ Both use caller cleanup ☒ AMD64 uses caller cleanup, CBPV uses callee cleanup
   ☐ Both use callee cleanup ☐ AMD64 uses callee cleanup, CBPV uses caller cleanup

   (b) (6 points) Give x86 assembly code for a function using the CBPV calling convention that returns the sum of its two arguments.

   Answer:

```
popq %rax
add %rsi, %rax
ret
```

   (c) (8 points) Give x86 assembly code that implements a call `x = f(1,2)` using the CBPV calling convention where the result is stored `x` is stored in `rax`. Assume the function you are calling from uses `rbp` and `rsp` to manage the stack frame as specified in HW3. Assume that the current values of all the registers except `rax` will be needed after the call. For full credit, use as little stack space as possible.

   Answer:

```
        pushq %rsi
        pushq %r8
        ...
        pushq %r14
        mov $1, %rsi
        leaq lbl(%rip), %rax
        pushq %rax
        pushq $2
        jmp f

lbl:
        popq %r14
        ...
        popq %r8
        popq %rsi
```

3. **LLVM** (20 points)

(a) (2 points) ☐ True or ☒ False: To correctly compile LLVM IR to assembly code, all LLVM temporaries must be stored on the stack.

Explanation: For many functions, all temporaries can be stored in registers.

(b) (4 points) The terminators in LLVMlite are direct `br`, conditional `br`, and return `ret`. All of these transfer control flow somewhere besides the next instruction. A function call, e.g., `call i64 @fac(i64 %2)` also transfers control flow. Why isn't a function call a terminator?

Answer: A function call is not a terminator because while control flow transfers to the callee, if the callee ever returns, control flow will resume at this point.

(c) A control-flow graph (CFG) is a *directed acyclic graph (DAG)* if it has no cycles, and is a *tree* if it is a DAG and additionally every node except the start node has exactly one predecessor.

- (4 points) Not every function in LLVM be can implemented using a CFG that is a DAG. Give an example of a CFG that is not a DAG for which there is no DAG that implements the same behavior.

  Answer: Any infinite or dynamically bounded loop

- (4 points) Can every CFG that is a DAG be implemented by a CFG that is a tree? Why or why not?

  Answer: Yes, because every node with multiple predecessors can be duplicated to make the DAG into a tree. However this will cause an exponential blowup in the size of the CFG in general so DAGs are still useful even without loops.

(d) (6 points) Appendix C contains a simple C function containing a switch statement. Hand-compile this to an LLVM function.

Answer:

```
define i64 @f(i64 %n) {
  %res1 = icmp eq i64 %n, 0
  br i1 %res1, label %blk_case0, label %blk_caseNot0

blk_case0:
  %tmp1 = call i64 @g(i64 0)
  br label %blk_end

blk_caseNot0:
  %res2 = icmp eq i64 %n, 1
  br i1 %res2, label %blk_case1, label %blk_caseDef

blk_case1:
  %tmp2 = call i64 @g(i64 1)
  br label %blk_caseDef

blk_caseDef:
  %tmp3 = call i64 @g(i64 2)
  ret i64 -1

blk_end:
  ret i64 0
}
```

4. **Regular Expressions, Finite Automata and Lexing** (20 points)

   (a) (2 points) ☐ True or ☒ False: An OCamllex program always outputs a sequence of tokens.

   Explanation: OCamllex programs generate code that matches regular expressions, but the right hand side of the match is a "semantic action" which can be arbitrary OCaml code which is not required to output a sequence of tokens.

   (b) (2 points) For an NFA with N states, what is the tightest bound we can give in general on the number of states in the corresponding minimal DFA?
   bound? **Answer:** The NFA to DFA construction makes a DFA with $O(2^n)$ states. Smaller DFAs

   may exist, but this is the tightest bound we can give in general

   (c) Let L be the set of strings over the alphabet $\{x, y, z\}$ where there is at least one y between any two xs.

      i. (4 points) Define a regular expression that matches precisely the strings in L.

      Answer: There are many equivalent answers. Here is one:

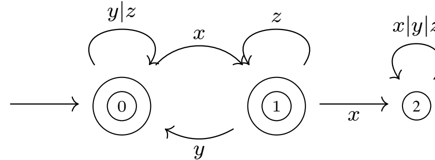
$$(y|z)^*(xz^*y(y|z)^*)^*(\varepsilon|x(y|z)^*)$$

      or using ? notation

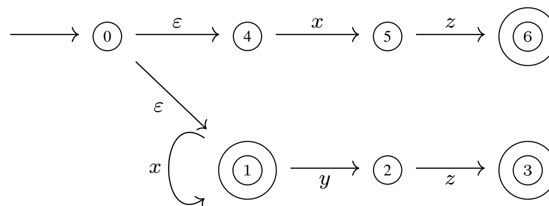
$$(y|z)^*(xz^*y(y|z)^*)^*(x(y|z)^*)?$$

      The English reading of this regex is that to generate a string in L, first generate some number of y, zs, then generate some number of xs that are followed by a mix of y, z that contains at least one y, and finally, optionally include a final x that is followed by final mix of y, zs.

      ii. (4 points) Define a DFA that accepts precisely the strings in L. Be sure to clearly mark which is the start node and which nodes are accepting.
      Answer:



   (d) Again we use the alphabet $\{x, y, z\}$. Consider the following NFA. We have labeled state 0 as the start state and marked accepting states $1, 3, 6$ by double circles.

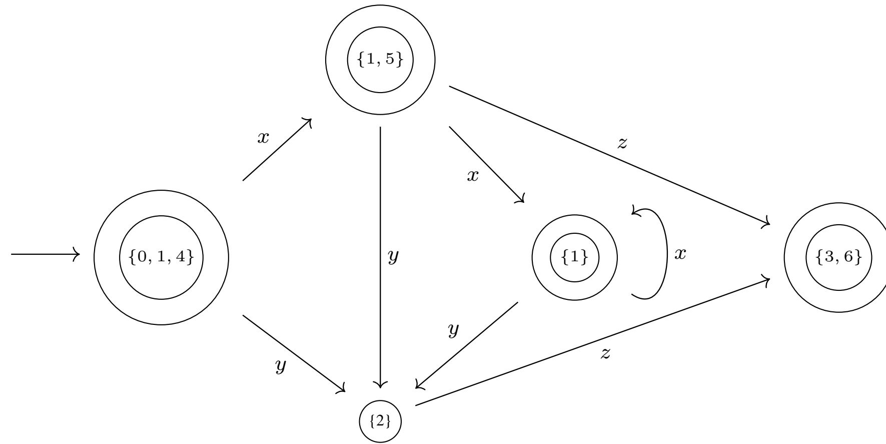


   - (4 points) Define a regular expression that matches the same language as that accepted by the NFA.

   Answer:

$$(xz)|(x^*(yz|\varepsilon))$$

- (4 points) Define a DFA that accepts the same strings as the NFA. For full credit, define the minimal DFA.
  Answer: Here we label the states by the subset of states in the NFA that they correspond to.



5. **Parsing** (20 points)

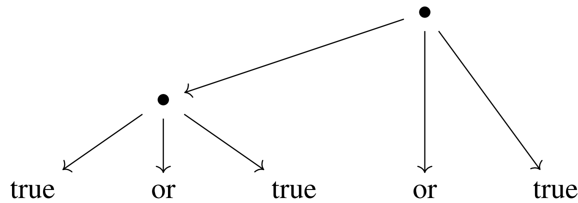
(a) (2 points) ☒ True or ☐ False: All LL(1) grammars are unambiguous.

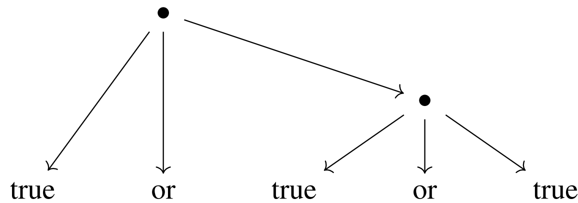
(b) (2 points) ☐ True or ☒ False: For an unambigious grammar, every parse tree has a unique derivation.

(c) (2 points) LR(1) parsers are a ☒ bottom-up or ☐ top-down parsing method?

(d) Consider the grammar in Appendix D

- (4 points) Show that the grammar is ambiguous by drawing two different parse trees for the same string. Two parses for "true or true or true"





- (4 points) In a typical programming language, what should the relative precedence relationship be between the binary operators `and`, `or` and the ternary operator `if_then_else_` .

`and` has higher precedence than `or` which has higher precedence than `if_then_else`

- (6 points) Define an LL(1) grammar that accepts the same strings as the original grammar in Appendix D, and with the precedence relationship you just described. Partial credit will be awarded for grammars that fit some but not all of the requirements of LL(1).

$$
\begin{array}{lcl}
B_0 & \longrightarrow & \texttt{if } B_0 \texttt{ then } B_0 \texttt{ else } B_0 \\
 & | & B_1 \\
B_1 & \longrightarrow & B_2\ B_1' \\
B_1' & \longrightarrow & \texttt{or } B_1 \\
 & | & \varepsilon \\
B_2 & \longrightarrow & B_3\ B_2' \\
B_2' & \longrightarrow & \texttt{and } B_2 \\
 & | & \varepsilon \\
B_3 & \longrightarrow & \texttt{true} \\
 & | & \texttt{false} \\
 & | & (\ B_0\ )
\end{array}
$$

EECS 483 Winter 2024 Midterm Appendices

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

Appendix A: Interpreter Code

```
1  type exp =
2   | Lit of int
3   | Mul of exp * exp
4   | Add of exp * exp
5   | Div of exp * exp
6
7  exception DividedByZeroV2
8  exception DividedByZeroV3 of int
9
10 let divide_v1 (m : int) (n : int) : int =
11   if n = 0
12   then 0
13   else m / n (* / in OCaml is integer division *)
14
15 let divide_v2 (m : int) (n : int) : int =
16   if n = 0
17   then raise DividedByZeroV2
18   else m / n
19
20 let divide_v3 (m : int) (n : int) : int =
21   if n = 0
22   then raise (DividedByZeroV3 m)
23   else m / n
24
25 let divide_v4 (m : int) (n : int) : int =
26   if n = 0
27   then (print_int(m); 0) (* print_int prints the input number to stdout *)
28   else m / n
29
30 let rec interpret_exp (e:exp) : int =
31   begin match e with
32     | Lit i -> i
33     | Add(e1, e2) -> (interpret_exp e1) + (interpret_exp e2)
34     | Mul(e1, e2) -> (interpret_exp e1) * (interpret_exp e2)
35     | Div(e1, e2) -> divide_v1 (interpret_exp e1) (interpret_exp e2)
36   end
```

Appendix B: CBPV Calling Convention

The CBPV calling convention for functions of two arguments works as follows:

1. All CBPV functions take two 64-bit arguments.
2. The first argument goes in `rsi`
3. The second argument is stored at `[%rsp]` and the return address is stored at `[%rsp + 8]`.
4. To return,
   (a) The return value should be in `rax`
   (b) The stack pointer `rsp` should be set to `16` more than its original value on entry to the function.
   (c) The instruction pointer `rip` should be set to the return address.
5. Registers `rsi`, `rax`, `rsp` and `r8-14` are caller-save and the remainder are callee-save.

Appendix C: Compiling Switch Statements

Assume in the following that `g` is of type `void g(int n)`.

```
int f(int n) {
  switch (n) {
  case 0:
    g(0);
    break;
  case 1:
    g(1);
  default:
    g(2)
    return -1;
  }
  return 0;
}
```

Appendix D: Grammar for Boolean Expressions

For clarity, terminal symbols are surrounded with a $\boxed{\text{box}}$.

$$
\begin{array}{rcl}
B & \longrightarrow & \boxed{\texttt{if}}\ B\ \boxed{\texttt{then}}\ B\ \boxed{\texttt{else}}\ B \\
 & | & B\ \boxed{\texttt{and}}\ B \\
 & | & B\ \boxed{\texttt{or}}\ B \\
 & | & \boxed{\texttt{true}} \\
 & | & \boxed{\texttt{false}} \\
 & | & \boxed{\texttt{(}}\ B\ \boxed{\texttt{)}}
\end{array}
$$