SOLUTIONS
1. **Interpreters and Language Semantics** (22 points)

We saw in the first homework how it is easy to implement an interpreter for a simple language of arithmetic expressions. Interpreted languages like Python, Javascript, Perl, and Ruby use similar techniques but provide many more language features. Alas, these dynamically typed languages are also prone to programming “Wat!?“s—simple programs that violate programmer expectations. For example, in Ruby it is possible for the code $x * y$ to produce the empty string but $y * x$ to fail with an exception, violating commutativity! (Wat!?) In this problem we will explore how strange behaviors like this can easily arise in interpreted languages.

**Step 1: Adding Boolean Values**  We first extend the expression language with boolean constants, an equality test, which creates boolean values, and a conditional expression, which lets us use them. This code is shown in Appendix A. The big change, compared with the expression language of the homework, is the new value datatype, which indicates that there are two types of values produced by our interpreter: integers and booleans, tagged using the OCaml constructors Int and Bool, respectively.

The next two questions pertain to the value and exp datatypes.

**a.** (1 point) Which of the following OCaml terms is an appropriate abstract syntax representation of the object-language expression $(0 * x) == False$?

- ☐ Mul (Eq (Int 0, Var "x"), Bool false)
- ☐ Mul (Eq (Imm (Int 0), Var "x"'), Imm (Bool false))
- ☐ Eq (Mul (Int 0, Var "x"), Bool false)
- ☒ Eq (Mul (Imm (Int 0), Var "x"), Imm (Bool false))

**b.** (1 point) We use the concrete syntax `if e1 then e2 else e3` for the abstract syntax `If(e1, e2, e3)`. Which of the following concrete syntax strings corresponds to

- ☒ `if x == y then x else (y == x)`
- ☐ `if x == y then (y == x) else y`
- ☒ `if x == (y == x) then x else y`
- ☐ `if (x == y) == x then x else y`
c. (8 points) Recall that we use a context to give meaning to the variables that might be mentioned in an expression. For each context below, indicate which result that the interpreter shown in Appendix A produces for \text{interpret ctxt e}, where \text{e} is a suitable encoding of the object-language concrete syntax for the expression:

\[
\text{if } (x == 0) \text{ then } y \text{ else } x
\]

Note that the interpreter uses \text{eqv} to “lift” OCaml’s notion of equality to work on object-language values.

\begin{enumerate}
\item \textbf{i.} \text{let ctxt} = ["x", Int 0]
  \begin{itemize}
  \item Int 0
  \item Bool true
  \item "variable x not defined"
  \item Int 1
  \item Bool false
  \item "variable y not defined"
  \end{itemize}
\item \textbf{ii.} \text{let ctxt} = ["x", Int 1]
  \begin{itemize}
  \item Int 0
  \item Bool true
  \item "variable x not defined"
  \item Int 1
  \item Bool false
  \item "variable y not defined"
  \end{itemize}
\item \textbf{iii.} \text{let ctxt} = ["x", Int 0]; ["y", Bool true]
  \begin{itemize}
  \item Int 0
  \item Bool true
  \item "variable x not defined"
  \item Int 1
  \item Bool false
  \item "variable y not defined"
  \end{itemize}
\item \textbf{iv.} \text{let ctxt} = ["x", Bool false]; ["y", Bool true]
  \begin{itemize}
  \item Int 0
  \item Bool true
  \item "variable x not defined"
  \item Int 1
  \item Bool false
  \item "variable y not defined"
  \end{itemize}
\end{enumerate}

d. (6 points) As implementors of this language, we might like to make it have better performance. In the space below, write code that replaces the case for \text{Mul} so that it “short circuits” the computation of \(e_1 \times e_2\): when \(e_1\) evaluates to zero the optimized interpreter doesn’t evaluate \(e_2\) at all and simply returns zero. (If \(e_2\) is expensive, this can be a significant win.) Your solution should still use \text{mulv} if \(e_1\) does not evaluate to 0.

\begin{verbatim}
let rec interpret (c:ctxt) (e:expr) : value =
begin match e with
| ... (* other cases remain the same *)
| Mul(e1, e2) ->
let v1 = interpret c e1 in
if v1 = Int 0 then Int 0 else
let v2 = interpret c e2 in
mulv v1 v2
end
\end{verbatim}
e. (2 points) The optimization proposed in part e. changes the computed outcome of some programs. Write a term (using abstract syntax) that will yield a different outcome under this “optimized” interpreter. Assume that both versions of the interpreter still use the definition of \textsc{mulv} from the Appendix. Describe the outcome under the original and the “optimized” semantics:

\begin{verbatim}
let example : exp = Mul(Imm(Int 0), Imm(Bool(true)))
\end{verbatim}

Original outcome: failure Optimized outcome: 0

f. (4 points) One place where adding support for additional primitive datatypes is a bit tricky is deciding how to handle the basic operators like equality and multiplication. Because the language is not statically typed, we have to determine what to do with an expression like \( x \times y \) when \( x \) and \( y \) are arbitrary values (i.e. including booleans), not just when \( x \) and \( y \) denote integers.

The function \textsc{mulv}, given in the Appendix, is one possible implementation of such a multiplication function: it succeeds, producing a tagged \textsc{Int} when both of its arguments are \textsc{Int} values, and fails with a dynamic type error otherwise. It is tempting (and many languages succumb to such temptations) to “define” multiplication for non-integer values in an ad-hoc way.

Suppose we change \textsc{mulv} to the following, which overloads \( * \) to mean “logical and” for booleans; it also silently converts \textsc{True} to 1 and \textsc{False} to 0 when multiplying mixed integers and booleans.

\begin{verbatim}
let mulv (v1:value) (v2:value) : value =
  match v1, v2 with
  | Int x , Int y -> Int (x*y)
  | Bool a, Int y -> if a then Int y else Int 0
  | Int x , Bool b -> if b then Int x else Int 0
  | Bool a, Bool b -> Bool (a && b)
\end{verbatim}

This definition might seem convenient, but it interacts poorly with the equality operator \( == \) as implemented by \textsc{eqv}. To see why, consider the following two mathematical expressions, which are equivalent under the ordinary rules of arithmetic:

\[
\begin{align*}
\text{if } x = 1 \text{ then } (\text{if } x \times y = y \text{ then } 1 \text{ else } 2) \text{ else } 3 \\
\text{if } x = 1 \text{ then } 1 \text{ else } 3
\end{align*}
\]

A direct transliteration of the above math into our programming language gives us these two expressions:

\begin{verbatim}
if x == 1 then (if x * y == y then 1 else 2) else 3
if x == 1 then 1 else 3
\end{verbatim}

Prove that, with the overloaded definition of \textsc{mulv} above, these two programs have different semantics by giving values for \( x \) and \( y \) on which the two programs are interpreted differently. (Wat?!)

\[
x = \text{Int 1} \quad \quad y = \text{Bool true}
\]
2. X86 and Calling Conventions (20 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. In this problem we will explore how this calling convention can be used to implement the C programming-language feature of “varargs”—functions that accept a variable number of arguments.

The following C program calls the function add_em_up, which accepts one or more arguments.

```c
extern int64_t add_em_up ( int64_t arg_count ,...)

int main ( void ) {
    printf ("%llu\n", add_em_up (3, 5, 5, 6)); /* prints 16. */
    printf ("%llu\n", add_em_up (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)); /* prints 55. */
    return 0;
}
```

The first argument to add_em_up is an integer that specifies the number of arguments that follow. The “...” in the function prototype represents this statically-unknown number of arguments. main uses the C standard library printf function (which is itself a vararg function!) to print the resulting of calling add_em_up twice, once with 4 arguments and once with 11 arguments. (Note: the printf format %llu marks a placeholder for a 64-bit unsigned integer.)

X86 code corresponding to this main function can be found in Appendix B, along with a (hand-written) implementation of add_em_up.

Suppose we run the x86 program starting from main.

a. (3 points) Consider the first call to add_em_up. What 64-bit integer value will be contained in the register %r10 when execution reaches line 12 of add_em_up? (This line is marked with [A].)

- [ ] -1
- [X] -2
- [ ] 3
- [ ] 5
- [ ] 6

b. (3 points) Consider the second call to add_em_up. According to this code, what value does the memory location pointed to by %r11 contain once execution has reached the end of line 10 in add_em_up? (This line is marked with [B].)

- [ ] 10 (a 64-bit integer)
- [ ] 6 (a 64-bit integer)
- [X] a return address (which points to the instruction at line 26 of main)
- [ ] the saved %rbp value that was pushed to the stack on line 4 of add_em_up

c. (3 points) Consider line 17 of add_em_up (marked with [C]). What do the parentheses around (%r11) mean?

- [ ] Nothing, they’re not significant.
- [X] They indicate an “indirect” operand whose meaning is the value pointed to by %r11 when it is treated as a memory address.
- [ ] They indicate an “indirect” operand whose meaning is the value of the address stored in %r11
- [ ] They mark %r11 as a callee-save register.
d. (3 points) Suppose that we changed line 6 of `main` to be `movq $6, %rdi` (while leaving the rest of the code untouched). What effect would this have when we run the program? The program would...

- crash before printing any output.
- print 16 but crash before printing second sum.
- print a “garbage” value (which looks like a large random number) and then print 55
- print 16 and then print a “garbage” output

e. (3 points) Suppose that we instead changed line 14 of `main` to be `movq $6, %rdi` (while leaving the rest of the code untouched). What effect would this have when we run the program? The program would...

- print 16 but crash before printing second sum.
- print 16 but then print a a “garbage” value (which looks like a large random number)
- print 16 and then print 21
- print 16 and then print 25

e. (3 points) Suppose that we instead changed line 14 of `main` to be `movq $6, %rdi` (while leaving the rest of the code untouched). What effect would this have when we run the program? The program would...

- print 16 but crash before printing second sum.
- print 16 but then print a a “garbage” value (which looks like a large random number)
- print 16 and then print 21
- print 16 and then print 25

f. (5 points) The code for `add_em_up` that is given in the Appendix was hand written in x86 assembly, but that isn’t very easy to use. The C programming language supports vararg functions by offering the programmer several macros: `va_list`, `va_start`, `va_arg`, and `va_end` whose use is shown in the C implementation of `add_em_up` shown below:

```c
int64_t add_em_up (int64_t count, ...)
{
    va_list ap;
    int64_t i, sum = 0;
    va_start (ap, count); /* Initialize the argument list. */
    for (i = 0; i < count; i++) {
        sum += va_arg (ap, int64_t); /* Get the next argument value. */
    }
    va_end (ap); /* Clean up. */
    return sum;
}
```

As shown above, `va_list` is a kind of iterator for the vararg function arguments: `va_start` initializes the iterator and `va_arg` takes the expected type of the “next” argument (here `int64_t`), returns the result, and advances the iterator. It is not possible to access the varargs in a different order (i.e. to access a later vararg before an earlier one). Based on your understanding of the C calling conventions, explain why not:

**Answer:** C calling conventions require call-by-copy, so the amount of space taken up by the list of variable arguments on the stack depends on their number and types (which determine their size). To compute the address of some arbitrary vararg, one would have to know the sizes (or types) of all the preceding arguments.
3. **LLVM IR** (20 points)

This problem examines the design of the LLVM IR. For your reference, Appendix C contains an excerpt of the LLVM Lite IR we used in homework three.

**a.** (5 points) Recall that the `Bitcast` operation, as specified in the project description, casts a value from one pointer type to another pointer type. Suppose we relax this restriction to allow `Bitcast` to also convert between `i64` values and pointer values. Does this change affect how `Bitcast` is compiled to x86 assembly? (Assume that we target the 64-bit subset of x86 as in the homework.) Briefly explain.

*Answer:* No, this doesn’t affect the compiler. `Bitcast` compiles to a `movq` instruction, and since `i64` and pointers are both 64-bits, they can be interchanged freely.

**b.** (5 points) Suppose that, in addition to making `Bitcast` more flexible as described in part a, we also remove the `call` instruction from the LLVM IR. Do these changes affect the expressiveness of the IR? That is, can we simulate the semantics of the `call` instruction using only the remaining LLVM constructs? If `call` cannot be simulated, briefly explain why not. If `call` can be simulated, explain which other LLVM instruction(s) can be used instead.

*Answer:* This *does* change the expressiveness of the IR. There is no way to simulate a function call using the other operations of the language because we cannot access the program counter using only LLVM features. Relaxing `Bitcast` does not give us access to the program counter, since the only way to reliably construct pointers is via `Alloca`. (Note: `Alloca`’s semantics don’t say anything about the existence of a stack.) However, any answer that says one could simulate the existence of a program counter and `Call` by carefully “guessing” the location of the initial stack pointer and emulating the entire call stack via LLVM instructions will get partial credit.)
c. (5 points) Suppose that, in addition to making Bitcast more flexible as described in part a, we also remove the GEP (getelementptr) instruction from the LLVM IR (but leave Call as usual). Do these changes affect the expressiveness of the IR? That is, can we simulate the semantics of the GEP instruction using only the remaining LLVM constructs? If GEP cannot be simulated, briefly explain why not. If GEP can be simulated, explain which other LLVM instruction(s) can be used instead.

*Answer:* This does not change the expressiveness of the IR. We can use a combination of the relaxed Bitcast and Add to compute the required pointer offsets: Bitcast a pointer to an integer, use Add to simulate the offsets computed by the “path” of the GEP operation, and then use Bitcast to convert the integer.

d. (5 points) Consider the following LLVM program in which parts of the GEP path parameters have been omitted:

```c
%T1 = type {%T2*, [ 4 x i64 ] }
%T2 = type {%T1*, i64, i64}

@G1 = global %T1 {%T2* @G2 , [4 x i64] [i64 42, i64 43, i64 44, i64 45]}
@G2 = global %T2 {%T1* @G1, i64 101, i64 341}

define i64 @foo (i32 %x) {
    % ptr1 = getelementptr %T2, %T2* @G2, i32 0, i32 2
    %v1 = load i64, i64* % ptr1
    % ptr2 = getelementptr %T2, %T2* @G2, i32 0, i32 0
    %v2 = load %T1*, %T1** % ptr2
    % ptr3 = getelementptr %T1, %T1* %v2, i32 0, i32 1, i32 %x
    %v3 = load i64, i64* % ptr3
    % ans = add i64 %v1, %v3
    ret i64 % ans
}
```

Fill in the blanks above to complete the GEP paths so that the calls:

- %a = i64 call foo(i32 0)
- %b = i64 call foo(i32 2)

result in %a containing 383 and %b containing 385.
4. Lexing, Parsing, and Grammars (18 points)

This problem considers some issues with lexing and parsing OCaml expressions. Our grammar will be
built from the following tokens, according to the lexer specification shown below (where we’ve omitted
the standard definitions of whitespace and character):

```plaintext
type token = ARR | LPAREN | RPAREN | FUN | VAR of string

rule token = parse
  | whitespace+ { token lexbuf } (* skip whitespace *)
  | 'fun'       { FUN }
  | character+ { VAR (lexeme lexbuf) }
  | "->"       { ARR }
  | '('        { LPAREN }
  | ')'        { RPAREN }
  | _ as c     { failwith "unexpected char" }
```

a. (2 points) According to the lexer definition above, how many tokens will be produced when tokenizing
the string “fun x -> z x”?

**Answer:** 5

Consider the following grammar for the concrete syntax of a subset of OCaml expressions, where var is
the token for variable names like x, y, z, etc.

```plaintext
E ::= var
    | E E
    | fun var -> E
    | ( E )
```

b. (4 points) Demonstrate that this grammar is ambiguous by giving two distinct, leftmost derivations
for the string “fun x -> z x”

**Derivation 1:**

1. \[ E \rightarrow \text{fun} \; x \rightarrow \text{E} \]
2. \[ \rightarrow \text{fun} \; x \rightarrow \text{E} \; \text{E} \]
3. \[ \rightarrow \text{fun} \; x \rightarrow z \; \text{E} \]
4. \[ \rightarrow \text{fun} \; x \rightarrow z \; x \]

**Derivation 2:**

1. \[ E \rightarrow \text{E} \; \text{E} \]
2. \[ \rightarrow \text{fun} \; x \rightarrow \text{E} \; \text{E} \]
3. \[ \rightarrow \text{fun} \; x \rightarrow z \; \text{E} \]
4. \[ \rightarrow \text{fun} \; x \rightarrow z \; x \]
c. (8 points) Disambiguate the grammar above so that it follows OCaml’s usual parsing rules. Recall that OCaml uses whitespace to stand for function application. Such applications should be left associative and should have higher precedence than any function binder \texttt{fun} \: x \: \rightarrow \: E.

Rewrite the grammar so that \( E_0 \) is the new start symbol. Introduce as many new nonterminals \( E_1, E_2, E_3, \) etc., as needed.

\[
\begin{align*}
E_0 & \ ::= \ \texttt{fun} \ \text{var} \ \rightarrow \ E_0 \\
& \quad \mid \ E_1 \\
E_1 & \ ::= \ E_1 \ E_2 \\
& \quad \mid \ E_2 \\
E_2 & \ ::= \ \text{var} \\
& \quad \mid \ (E_0)
\end{align*}
\]

d. (4 points) The Lisp and Scheme programming languages are infamous (and sometimes derided) for requiring the use of many parentheses. However, that design does have a significant benefit: the entire language grammar can be written as follows, where the \texttt{atom} token includes variables and literal constants. Here \( () \) is pronounced “nil” and \( (S \ . \ S) \) is a “cons cell” (this is actually where OCaml gets the name for its list constructors).

\[
\begin{align*}
S & \ ::= \ \text{atom} \\
& \quad \mid \ () \\
& \quad \mid \ (S \ . \ S)
\end{align*}
\]

This grammar is not quite in LL(1) form. Write an equivalent grammar that is:

\[
\begin{align*}
S & \ ::= \ \text{atom} \\
& \quad \mid \ (S') \\
S' & \ ::= \ () \\
& \quad \mid \ S \ . \ S
\end{align*}
\]
Appendix A: Interpreter Code for Ints and Booleans

type value = (* the possible dynamic values *)
| Int of int
| Bool of bool

type exp =
| Var of string (* string representing an object-language variable *)
| Imm of value (* an immediate value *)
| Mul of exp * exp (* product of two expressions *)
| Eq of exp * exp (* equality of two expressions *)
| If of exp * exp * exp (* conditional choice *)

(* Evaluation contexts: associate variables with their dynamic values *)
type ctxt = (string * value) list

let rec lookup (x: string) (c: ctxt) : value =
begin match c
| [] -> failwith @@ "var " ˆ x ˆ " undefined"
| (y,v)::rest -> if x = y then v else lookup x rest
end

(* Multiply two dynamic values: fails if non-Ints are provided *)

let mulv (v1: value) (v2: value) : value =
multiply v1, v2 with
| Int x, Int y -> Int (x*y)
| _, _ -> failwith "expected int but got bool"

(* Compares two dynamic values for equality *)
let eqv (v1: value) (v2: value) : value =
Bool (v1 = v2)

let rec interpret (c: ctxt) (e: exp) : value =
begin match e
| Var x -> lookup x c
| Imm v -> v
| Mul(e1, e2) ->
let v1 = interpret c e1 in
let v2 = interpret c e2 in
mulv v1 v2
| Eq(e1, e2) ->
let v1 = interpret c e1 in
let v2 = interpret c e2 in
eqv v1 v2
| If(e, e1, e2) ->
match interpret c e with
| Bool true -> interpret c e1
| Bool false -> interpret c e2
| Int _ -> failwith "expected bool but got int"
end
Appendix B: X86 and Calling Conventions

x86 code for main

```assembly
.text
.globl _main
_main: ## @main
    pushq %rbp
    movq %rsp, %rbp
    movq $3, %rdi
    movq $5, %rsi
    movq $5, %rdx
    movq $6, %rcx
    callq _add_em_up
    leaq L_.str(%rip), %rdi
    movq %rax, %rsi
    callq _printf
    movq $10, %rdi
    movq $1, %rsi
    movq $2, %rdx
    movq $3, %rcx
    movq $4, %r8
    movq $5, %r9
    pushq $10
    pushq $9
    pushq $8
    pushq $7
    pushq $6
    callq _add_em_up
    addq $40, %rsp
    leaq L_.str(%rip), %rdi
    movq %rax, %rsi
    callq _printf
    popq %rbp
    retq
.data
.L_.str: ## @.str
.asciz "%llu\n"
```
x86 code for add_em_up

The following x86 code implements the `add_em_up` function. It accepts a variable number of arguments, each one a 64-bit integer value. The first one is the count of the number of remaining arguments. This function returns the sum of those arguments.

```assembly
.text
.globl _add_em_up
.add_em_up:            ## @add_em_up
    pushq %rbp
    movq %rsp , %rbp
    movq $0, %rax  ## initialize the accumulator
    movq %rdi , %r10  ## %rdi has the arg_count
    subq $5, %r10  ## %r10 is the loop counter
    movq %rbp , %r11  ## %r11 points into the stack
    addq $8, %r11  ## [B]
    ENTRY:
    cmpq $0, %r10  ## exit the loop? [A]
    jle ARG5
    LOOP:
    addq $8, %r11  ## get arg's stack address
    addq (%r11), %rax  ## add the arg to the accumulator [C]
    subq $1, %r10  ## decrement counter
    jmp ENTRY
    ARG5:
    cmpq $4, %rdi  ## is there a 5th vararg?
    jle ARG4
    addq %r9, %rax
    ARG4:
    cmpq $3, %rdi  ## is there a 4th vararg?
    jle ARG3
    addq %r8, %rax
    ARG3:
    cmpq $2, %rdi  ## is there a 3rd vararg?
    jle ARG2
    addq %rcx, %rax
    ARG2:
    cmpq $1, %rdi  ## is there a 2nd vararg?
    jle ARG1
    addq %rdx, %rax
    ARG1:
    cmpq $0, %rdi  ## is there a 1st vararg?
    jle ARG0
    addq %rsi, %rax
ARG0:
    popq %rbp
    retq
```
Appendix C: LLVM Lite IR

This code is a direct excerpt of the OCaml abstract syntax for the LLVM Lite IR taken from Homework 3.

(* LLVM types *)

```ocaml
type ty =
| Void | I1 | I8 | I64 | Ptr of ty | Struct of ty list | Array of int * ty
| Fun of ty list * ty
| Namedt of tid
```

(* Function type: argument types and return type *)

```ocaml
type fty = ty list * ty
```

(* Syntactic Values *)

```ocaml
type operand =
| Null
| Const of int64
| Gid of gid
| Id of uid
```

(* Binary i64 Operations *)

```ocaml
type bop =
| Add | Sub | Mul | Shl | Lshr | Ashr | And | Or | Xor
```

(* Comparison Operators *)

```ocaml
type cnd =
| Eq | Ne | Slt | Sle | Sgt | Sge
```

(* Instructions *)

```ocaml
type insn =
| Binop of bop * ty * operand * operand
| Alloca of ty
| Load of ty * operand
| Store of ty * operand * operand
| Icmp of cnd * ty * operand * operand
| Call of ty * operand * (ty * operand) list
| Bitcast of ty * operand * ty
| Gep of ty * operand * operand list
```

```ocaml
type terminator =
| Ret of ty * operand option
| Br of lbl
| Cbr of operand * lbl * lbl
```

(* Basic Blocks *)

```ocaml
type block = { insns : (uid * insn) list; term : (uid * terminator) }
```

(* Control Flow Graphs: entry and labeled blocks *)

```ocaml
type cfg = block * (lbl * block) list
```