Exam 3 - 628 Cover Sheet

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device. I also affirm that I have completed the exam according to the restrictions listed in the exam document.

Signed: _____

Print name: _____

Date: _____

Please submit a signed and dated copy of this cover sheet as the first page of your exam submission.

You will not receive credit for the exam unless your submission includes this (signed and dated) cover sheet.

601.428 Compilers and Interpreters Fall 2020, Exam 3 - 628

Due: Wednesday, November 18 by 3:00pm Baltimore time (UTC-05:00)

The permitted resources for this exam are:

- The textbook(s)
- Materials posted directly on the course website (e.g., slides)

Do *not* use any resources other than the ones explicitly noted above.

You may *not* write program(s) or use automated calculation devices or programs. You will need to do required calculations by hand. In other words, this is a "pencil and paper" exam, but you can (and should) type your answers. (Neatly handwritten answers are also fine.)

Do *not* discuss the exam with anyone else: your answers must be your own answers.

You will submit your answers to Gradescope as "Exam 3 - 628" in PDF format. When you upload to Gradescope, you will need to select the page of your submitted document corresponding to your answer to each question. You may use software (word processor, LaTeX, etc.) to prepare your answers.

Important: Make sure the first page of your submitted document is the signed and dated cover sheet (which is the first page of this exam document.) You will not receive credit for the exam if your submission does not include the (signed and dated) cover sheet.

Important: Show your work, and justify your answers. "Bare" answers (without supporting work or justification) may not receive full credit.

[Questions begin on next page.]

Question 1. [63 points] Consider the following context-free grammar for *prefix expressions*:

expr → ***** *expr expr*
expr → **+** *expr expr*
expr → **int-literal**
expr → **lv** [**int-literal**]

Assume that *****, **+**, **int-literal**, **lv**, [, and] are terminal symbols. Also, assume that

- lexemes for **int-literal** symbols consist of a sequence of digit characters
- the lexeme for a **lv** symbol is always "**lv**"
- lexemes for ***** and **+** are always "*****" and "**+**", respectively
- lexemes for [and] are always "[" and "]", respectively

(a) [9 points] Show a derivation and parse tree for the input

```
* + lv [ 0 ] 3 + lv [ 1 ] 10
```

(b) [45 points] Show an attribute grammar that translates prefix expressions (as specified by the grammar rules above) into a sequence of x86-64 assembly language instructions that carry out the computation described by the expression. Assume that subexpressions generated by the *expr* → **lv** [**int-literal**] rule are local variables, where the **int-literal** indicates the identity of the local variable. I.e., `lv [ 0 ]` and `lv [ 1 ]` are different local variables because `0` and `1` are different integer values.

Your attribute grammar can make reasonable assumptions about how storage is allocated for local variables; for example, that the `%rbp` register points to the storage block allocated for local variables. (State your assumptions about how storage is allocated.) You may assume that the integer identifying the local variable can be used to compute the offset of the local variable's storage.

You may also make reasonable assumptions about where computed values are stored. (Hint: the generated assembly language instructions may use the stack. The stack `push` and `pop` instructions could be very useful.)

Keep in mind that an attribute grammar is a functional computation: there is no mutable global state.

(c) [9 points] Show the attribute values for each parse node in the parse tree you specified in part (a). Also, explain how the attribute values are computed (i.e., what is the evaluation order.)

Question 2. [27 points]

Consider the following x86-64 function (called `partition`):

```
01:     .globl partition
02: partition:
03:     pushq %r12                    /* lowidx */
04:     pushq %r13                    /* highidx */
05:     subq $8, %rsp
06:     movq $0, %r12
07:     movq %rsi, %r13
08:     decq %r13
09: .LpartitionLoop:
10:     cmpq %r12, %r13
11:     jb .LpartitionDone
12:     cmpl (%rdi,%r12,4), %edx      /* pivot_val > arr[lowidx]? */
13:     jbe .LpartitionCheckHighidx   /* if not, check arr[highidx] */
14:     incq %r12                     /* lowidx++ */
15:     jmp .LpartitionLoop           /* continue loop */
16: .LpartitionCheckHighidx:
17:     cmpl (%rdi,%r13,4), %edx      /* pivot_val <= arr[highidx]? */
18:     ja .LpartitionSwapElts        /* if not, need to swap elements */
19:     decq %r13                     /* highidx-- */
20:     jmp .LpartitionLoop           /* continue loop */
21: .LpartitionSwapElts:
22:     /* swap elements */
23:     movl (%rdi,%r12,4), %r10d
24:     movl (%rdi,%r13,4), %r11d
25:     movl %r10d, (%rdi,%r13,4)
26:     movl %r11d, (%rdi,%r12,4)
27:     incq %r12                     /* lowidx++ */
28:     decq %r13                     /* highidx-- */
29:     jmp .LpartitionLoop           /* continue loop */
30: .LpartitionDone:
31:     movl %r13d, %eax              /* return value */
32:     addq $8, %rsp
33:     popq %r13
34:     popq %r12
35:     ret
```

Draw a control flow graph for this function. For each basic block in the control flow graph, indicate which instructions are in the basic block. (You can specify a range of line numbers to specify which instructions are in each basic block.) Show control flow edges. Indicate which basic blocks are the entry point (where execution starts) and exit point (where execution finishes.)

Question 3. [10 points] In the representation of basic blocks in a control flow graph, should a procedure call instruction always appear at the end of the block? Or, could it be placed at a position prior to the last instruction in a block? Briefly argue one of the following positions:

1. A procedure call should always be placed at the end of a basic block
2. A procedure call could appear anywhere in a basic block

Provide a justification for your position.