CSE P 501 18sp Exam 5/24/18

Name ___________________________________

There are 7 questions worth a total of 110 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, no notes.

The exam will be scanned for grading. **Write on the front side of each page only.** The back sides will not be scanned. A blank page is provided at the end of the exam if you need extra space for answers. Please be sure to indicate on the original page of a question that your answers are continued on the back, and label the answers on this additional page if you use it.

Please wait to turn the page until everyone is told to begin.

Score ___________________

1 ________ / 12

2 ________ / 8

3 ________ / 20

4 ________ / 8

5 ________ / 26

6 ________ / 18

7 ________ / 18

Question 1. (12 points) Regular expressions. Our new summer intern is designing a programming language and has decided that the traditional syntax for comments is old and boring. The comments in this new programming language will be strings that start with the three characters `<##` and end with the two characters `#>`. Examples of comments: `<## comment #>` `<### xyzzy ###>` `<###>` `<#####>`. Examples that are not comments: `<##>` (need at least `<##` at the beginning and `#>` at the end), `<# # #>` (first three characters `<##` can't include spaces or other characters), etc.

(a) (6 points) Give a regular expression that generates strings representing comments as described above. (Hint: you may want to work on parts (a) and (b) at the same time.)

Ground rules (the fine print): You may only use the basic regular expression operations of concatenation, choice (|), and repetition (*) plus the derived operators ? and +, and simple character classes like `[abc0-9]` and `[^a-z]`. You may use abbreviations like `vowels = [aeiou]`. You may not use more complex operators found in various software tools that handle extended regular expressions and you should not use '\' or other escape characters. If you need to differentiate between terminal characters and regular expression operators, underline the terminal characters to distinguish them or do something equally simple and easy to read.

(b) (6 points) Draw a DFA that accepts comments as defined above.

Question 2. (8 points) Ambiguity. The syntax used to specify regular expressions can itself be defined by a context-free grammar. Here is one possible grammar for regular expressions with the operators concatenation, choice (`|`), Kleene star (`*`), and parenthesized subexpressions over the alphabet { `a` , `b` }.

R ::= *R R*	(concatenation)
R ::= *R* `|` *R*	(the `|` here is the literal regular expression choice operator)
R ::= *R* `*`	(Kleene star)
R ::= `(` *R* `)`	
R ::= `a`	
R ::= `b`	

Show that this grammar for specifying the syntax of regular expressions is ambiguous.

Question 3. (20 points) The you're-probably-not-surprised-to-see-it LR parsing question. Consider the following grammar.

0. *S'* ::= *S* $ ($ is end-of-file)
1. *S* ::= (*S*) *T*
2. *S* ::= x
3. *T* ::= x

(a) (12 points) Draw the LR(0) state machine for this grammar. (You do not need to include the table with shift/reduce and goto actions, although you can write that out later if you find it useful to answer other parts of the question.)

(continued next page)

Question 3. (cont.) Grammar repeated for reference

0. *S'* ::= *S* \$
1. *S* ::= (*S*) *T*
2. *S* ::= x
3. *T* ::= x

(b) (4 points) Compute FIRST, FOLLOW, and Nullable for each of the non-terminals in this grammar.

(c) (2 points) Is this grammar LR(0)? Why or why not?

(d) (2 points) Is this grammar SLR? Why or why not?

Question 4. (8 points) (LL parsing/grammars) Here is the grammar from the previous question again:

0. *S'* ::= *S* \$
1. *S* ::= (*S*) *T*
2. *S* ::= x
3. *T* ::= x

Is this a LL(1) grammar suitable for top-down predictive parsing? If yes, give a specific technical justification for your answer. If not, give a grammar that generates the same language and is LL(1) if that is possible. If no LL(1) grammar can generate the same language produced by the original grammar, give an explanation of why this is not possible.

Hint: You might want compute the FIRST/FOLLOW/nullable information for this grammar by answering that part of the previous question before you answer this question.

Question 5. (26 points) Compiler hacking. For this question we would like to add a new counting loop to MiniJava. (A copy of the MiniJava grammar is included at the end of the test for reference as needed.) For our new loop, we'll add the following rule to the MiniJava grammar:

Statement ::= "for" Identifier "from" Expression "to" Expression "do" Statement

The idea is that the Statement in the loop body is executed repeatedly with the Identifier assigned successive integer values starting with the value of the first Expression and increasing by 1 each time the loop repeats until the final iteration where the Identifier has the value of the second Expression. For example, the following code stores the value 1 + 2 + … + 10 in variable `sum`:

```
sum = 0;
for i from 1 to 10 do sum = sum + i;
```

The Identifier in the `for` statement must have been declared previously and must have type `int`. The two Expressions are only evaluated once, before the Identifier is assigned its initial value and before the loop body executes. The Expressions are not re-evaluated again as the loop executes. So, for example, the following code has exactly the same effect as the previous example:

```
sum = 0; i = 0;
for i from i+1 to i+10 do sum = sum + i;
```

In other words, the loop bounds `i+1` and `i+10` are evaluated before the loop begins execution and before the initial assignment to `i`, and are not reevaluated again. The Expressions are evaluated in order from left to right. If the value of the first Expression is greater than the value of the second Expression, then the body of the loop is not executed. The value in the Identifier is not defined after the loop terminates – it might be equal to the value of one of the Expressions, or it could have any other value depending on what the implementation does.

(a) (3 points) What new tokens and/or keywords would need to be added to the scanner and parser of our MiniJava compiler to add this new `for` statement to the original MiniJava grammar? Just list the tokens; you don't need to give JFlex or CUP specifications for them.

(continued on next page)

Question 5. (cont.) (b) (5 points) Complete the following new AST class to define an AST node type for the new `for` statement. You only need to define instance variables and the constructor. Assume that all appropriate package and import declarations are supplied, and don't worry about visitor code.

(Hint: recall that the AST package in MiniJava contains the following key classes: `ASTNode`, `Exp extends ASTNode`, `Statement extends ASTNode`, and `Identifier extends ASTNode`. Also remember that each AST node constructor has a `Location` parameter.)

```
public class For extends Statement {
  // add instance variables below



  // constructor - add parameters and method body below


  public For( ________________________________________ ){
    super(pos);



  }
}
```

(continued on next page)

Question 5. (cont.) (c) (5 points) Complete the CUP specification below to define a production for the new `for` statement with the associated semantic action(s) needed to parse a `for` statement and create an appropriate `For` node (as defined in part (b) above) into the AST. We have added the necessary additional code to the parser rule for Statement to get started.

Hint: recall that the `Location` of an item `foo` in a CUP grammar production can be referenced as `fooxleft`.

```
Statement ::= ...
            | ForStatement:s  {: RESULT = s; :}
           ...
            ;

ForStatement ::=
```

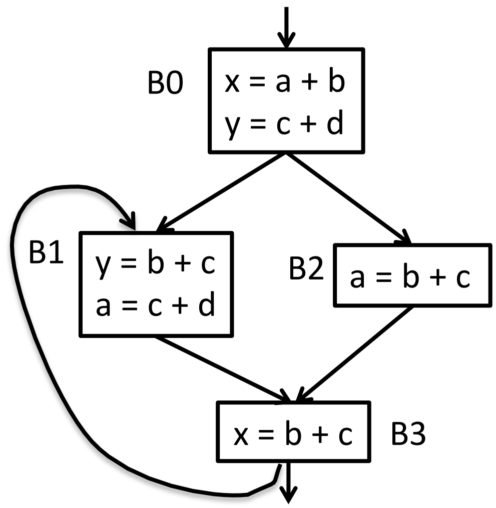
(d) (5 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that a `for` statement is legal. You do not need to give code for a visitor method or anything like that – just describe what language rules (if any) need to be checked.

(continued on next page)

Question 5. (cont.) (e) (8 points) Show the code shape for this new `for` statement, i.e., what code should be generated in the assembly language program to properly execute a `for` loop as specified on the previous pages. You should show instructions, labels, and other assembly language-level details that are needed for the new `for` loop statement itself, and also show where the generated code for the two Expressions and Statement that makes up the loop body would appear. Your code does not need to be precisely correct x86-64 assembly code (i.e., you were not expected to memorize instruction details), but it should be close enough so that your intent is clear and it basically equivalent to real x86-64 code.

Hint: you probably won't need all the space on this page for your answer. But be careful that your code matches the described operation of the `for` statement given previously.

The remaining questions concern the following control flow graph.



The rest of this page contains reference material and definitions that might be useful when answering some of the remaining questions.

You should **remove this page from the exam** and use it while answering the remaining questions. **Do not write on this page** – it will not be scanned for grading.

Reference Material

Every control flow graph has a unique **start node** s0.

Node *x* **dominates** node *y* if every path from s0 to *y* must go through x.

- A node *x* dominates itself.

A node *x* **strictly dominates** node *y* if *x* dominates *y* and $x \neq y$.

The **dominator set** of a node *y* is the set of all nodes *x* that dominate *y*.

An **immediate dominator** of a node *y*, idom(*y*), has the following properties:

- idom(*y*) strictly dominates *y* (i.e., dominates *y* but is different from *y*)
- idom(*y*) does not dominate any other strict dominator of *y*

A node might not have an immediate dominator. A node has at most one immediate dominator.

The **dominator tree** of a control flow graph is a tree where there is an edge from every node *x* to its immediate dominator idom(*x*).

The **dominance frontier** of a node *x* is the set of all nodes *y* such that

- *x* dominates a predecessor of *y*, but
- *x* does not strictly dominate *y*

Question 6. (18 points) Dataflow analysis – available expressions.

Recall from lecture that an expression *e* is *available* at a program point *p* if every path leading to point *p* contains a prior definition of expression *e* and *e* is not killed along a path from a prior definition by having one of its operands re-defined on that path.

We would like to compute the set of available expressions at the beginning of each basic block in the flowgraph shown on the previous page.

For each basic block *b* we define the following sets:

AVAIL(*b*) = the set of expressions available on entry to block *b*
NKILL(*b*) = the set of expressions *not killed* in *b* (i.e., all expressions defined somewhere in the flowgraph except for those killed in *b*)
DEF(*b*) = the set of all expressions defined in *b* and not subsequently killed in *b*

The dataflow equation relating these sets is

$$\text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$$

i.e., the expressions available on entry to block *b* are the intersection of the sets of expressions available on exit from all of its predecessor blocks *x* in the flow graph.

On the next page, calculate the DEF and NKILL sets for each block, then use that information to calculate the AVAIL sets for each block. You will only need to calculate the DEF and NKILL sets once for each block. You may need to re-calculate some of the AVAIL sets more than once as information about predecessor blocks change.

Hint: notice that there are only three expressions calculated in this flowgraph: a+b, b+c, and c+d. So all of the AVAIL, NKILL, and DEF sets for the different blocks will contain some, none, or all of those three expressions.

You should **remove this page from the exam** and use it while answering this question. **Do not write on this page** – it will not be scanned for grading.

Question 6. (cont.) (a) (8 points) For each of the blocks B0, B1, B2, and B3, write their DEF and NKILL sets in the table below.

Block	DEF	NKILL
B0		
B1		
B2		
B3		

(b) (10 points) Now, give the AVAIL sets showing the expressions available on entry to each block in the table below. If you need to update this information as you calculate the sets, be sure to cross out previous information so it is clear what your final answer is.

Block	AVAIL
B0	
B1	
B2	
B3	

Question 7. (18 points) Dominators and SSA. (a) (8 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the nodes that dominate it, the node that is its immediate dominator (if any), and the nodes that are in its dominance frontier (if any):

Node	Dominators	IDOM	Dominance Frontier
B0			
B1			
B2			
B3			

(b) (10 points) Now redraw the flowgraph in SSA (static single-assignment) form. You need to insert appropriate Φ-functions where they are required and, once that is done, add appropriate version numbers to all variables that are assigned in the flowgraph. You should not insert extra Φ-functions at the beginning of a block if they clearly would not be appropriate there, but we will not penalize a few extraneous Φ-functions if they are correct, but possibly not needed. You do not need to trace the steps of any particular algorithm to place the Φ-functions as long as you add them to the flowgraph in appropriate places.

Additional space for answers if needed. Please be sure to label your answers and indicate on the original question that your answers are continued here.