CSE P 501 19au Exam 11/21/19 Sample Solution

Question 1. (10 points, 1 each) Compiler phases. There are many possible errors that can occur in a program. For each of the following possible MiniJava errors, indicate when it would be detected, either at compile time or during execution, and, if the error can be detected by the compiler, indicate the earliest point in the compiler (scanner, parser, typecheck/semantics) where the error can definitely be detected. Assume that the compiler is a conventional one that generates native code for a single target machine (say, x86-64), and assume that the source language is the MiniJava for our project (if it helps, a copy of the MiniJava grammar is attached at the end of the exam for reference). Use the following abbreviations for the stages:

scan – scanner
parse – parser
sem – semantics/type check

run – runtime (i.e., when the compiled code is executed)
can't – can't always be done during either compilation or execution

__scan__ Standard MiniJava does not include a >> (right shift) operator
(Since there is no > operator in MiniJava, the > character cannot form part of a valid token. The scanner would detect that.)

__sem__ In the assignment statement `a=b;`, expression `b` has type `int` and `a` has type `boolean` and these types are incompatible for assignment.

__can't__ The program contains an infinite loop and will not terminate if it is executed.

__parse__ Standard MiniJava does not include a ++ "increment" operator.
('+' is a legal MiniJava operator, so the scanner would treat '++' as two adjacent tokens. The parser would reject the program when it tried to parse that.)

__sem__ If `x` has type `T` and the program contains a method call `x.f(17)`, there is no method `f` that has one integer parameter in class `T` or any of its superclasses.

__run__ In the method call `x.f(17)`, variable `x` is `null` (does not reference an object)

__sem__ Class `C` contains two definitions for method `f` that have different numbers of parameters (recall that MiniJava does not support method overloading).

__run__ The array reference `a[5]` is incorrect because `a` has too few elements.

__sem__ `System.out.println(x<y)` is not supported in MiniJava.
(MiniJava only supports println for `int` values.)

__parse__ `System.out.println()` is not supported in MiniJava.
(println requires an expression as an argument in MiniJava.)

Question 2. (12 points) Regular expressions. All reasonable file systems have directories and files, where directories can contain other directories and files, and there is a naming convention for referencing files.

We have a Unix/Linux-like operating system where file names are specified as follows (read carefully – this might be a *tiny* bit different from what you might assume). A name can start with an optional leading slash ('/'). Then there is a set of zero or more directory names separated by slashes. Directory names consist of one or more upper- or lower-case letters. Following the directory names, if any, is the required file name. File names include an identifier with one or more upper- or lower-case letters, followed by an optional one- to three-letter file extension. If the file extension is present, it is separated from the file name by a period ('.').

Examples of valid names: `/long/directory/path/with/a/file.txt`; `some/subDirectory/lib.a`; `/x.yz`; `simpleFileNameWithNoExtension`; `simpleFileNameWithExtension.cpp`

Examples of invalid names: `a/b//c` (adjacent '/' characters), `a/b/` (no file name after final '/'), `foo.docx` (more than three letters following '.'), `file1` (digit in file name) `file.` (no letters following '.')

(a) (6 points) Give a regular expression that generates strings representing file names 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.

name = **[a-zA-Z]+**

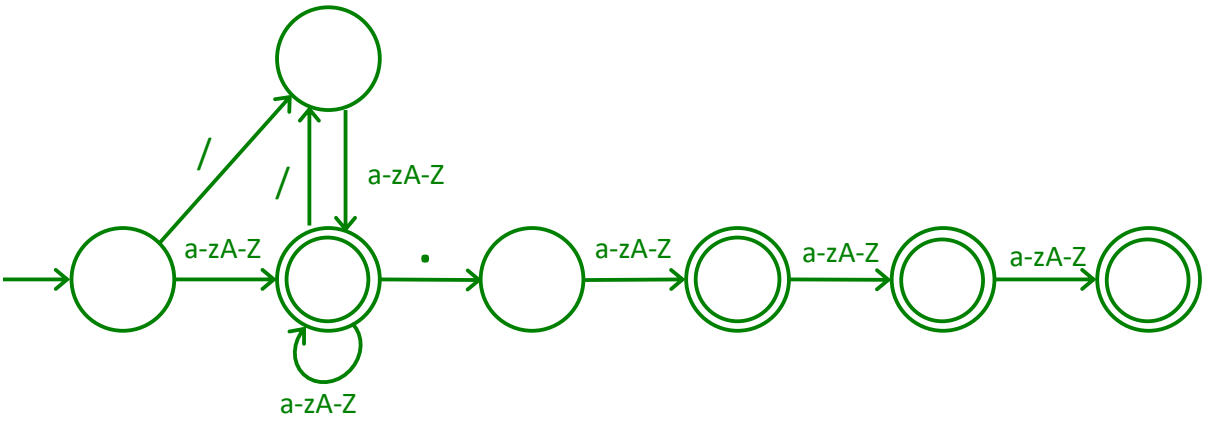
(/? (*name* /)*)? *name* (. [a-zA-Z] [a-zA-Z]? [a-zA-Z]?)?

There are some other regular expressions that generate the same set. Any correct answer received full credit, of course.

A couple of answers assumed that file extensions would consist of lower-case letters only `[a-z]`. Since the question didn't *explicitly* rule this out, we gave credit to those answers, however, given that everything else in the question used upper- and lower-case letters interchangeably, it would have been better to assume this applied to file extensions also – or at least to have asked about this during the exam instead of assuming otherwise.

(continued on next page)

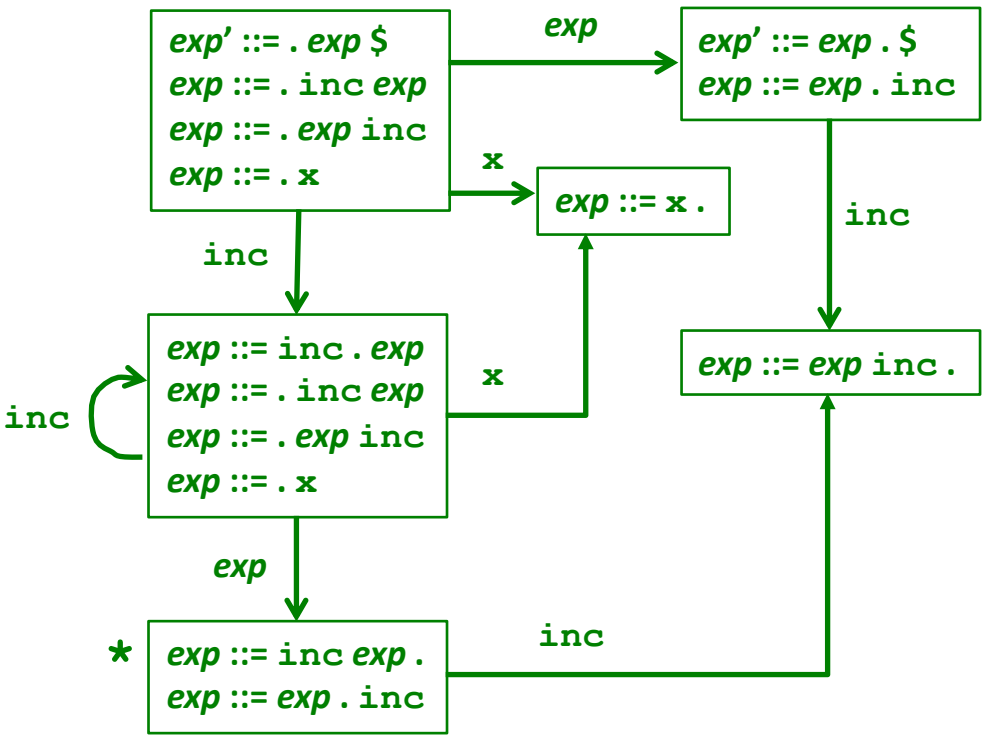
Question 2. (cont.) (b) (6 points) Draw a DFA that accepts file names as defined in the question and that are generated by your answer to part (a).



Question 3. (24 points) The you're-probably-not-surprised-to-see-it LR parsing question. In C, C++, Java, and other languages, ++ can be used as both a prefix and postfix increment operator. Consider the following grammar, which uses the terminal `inc` instead of ++ to avoid any possible confusion about whether it is a single terminal symbol (which it is), rather than a pair of separate + symbols (which it isn't).

0. *exp'* ::= *exp* $ ($ is end-of-file)
1. *exp* ::= `inc` *exp*
2. *exp* ::= *exp* `inc`
3. *exp* ::= `x`

(a) (14 points) Draw the LR(0) state machine for this grammar. You do not need to include the parse table with shift/reduce and goto actions, although you can write all or part of that later if you find it helpful to answer the rest of this question on the next page.



(continued on next page)

Question 3. (cont.) Grammar repeated for reference

0. *exp'* ::= *exp* $ ($ is end-of-file)
1. *exp* ::= `inc` *exp*
2. *exp* ::= *exp* `inc`
3. *exp* ::= `x`

(b) (4 points) Compute *nullable* and the FIRST and FOLLOW sets for the single nonterminal *exp* in the above grammar:

Symbol	nullable	FIRST	FOLLOW
exp	**no**	**`inc x`**	**`inc $`**

(c) (3 points) Is this grammar LR(0)? Why or why not?

No. There is a shift-reduce conflict in the state marked * when the next input is `inc`.

(d) (3 points) Is this grammar SLR? Why or why not?

No. Since `inc` is in FOLLOW(*exp*), we cannot delete the r1 (reduce using rule 1) action from the state that has the conflict when the input is `inc`.

Question 4. (12 points) (LL parsing/grammars) Consider the following grammar:

S ::= `b` | *M*
M ::= `a` | *M* `c c` | `c` *M* | `b b`

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 quick explanation of why it is not, and then 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 are not required to compute FIRST and FOLLOW information for the nonterminals in the grammar, but you might find some of that information useful in explaining your answer.

The grammar is not LL(1). FIRST(*M*) contains a, b, and c, which means that both rules for *s* have b in the FIRST set for their right-hand sides, and there are multiple FIRST conflicts in the right-hand sides for *M*.

There are many ways to create a grammar that generates the same language but is LL(1). Here are two possibilities:

(1) We start by applying the canonical fix for direct left recursion:

S ::= b | M
M ::= a N | b b N | c M N
N ::= c c N | ε

We need to perform left factoring because b is in the FIRST set for the right-hand side of both productions of *S*, but first we need to get the grammar in a form that can be factored by substituting *M* in the second *S* production:
S ::= b | a N | b b N | c M N
M ::= a N | b b N | c M N
N ::= c c N | ε

Now we can left factor:
S ::= b T | a N | c M N
T ::= ε | b N
M ::= a N | b b N | c M N
N ::= c c N | ε

There is still a conflict in N, because it is nullable and c appears in its FIRST and FOLLOW sets. To fix, notice that a single N non-terminal is equivalent to two N non-terminals next to each other because it can recurse an unlimited number of times. Therefore, since N already follows the occurrence of M in S ::= c M N, we can factor out the occurrence of N at the end of each production for M:

S ::= b T | a N | c M N
T ::= ε | b N
M ::= a | b b | c M
N ::= c c N | ε

This grammar generates the same language but is now LL(1).

(2). Note that the strings generated by M start with any number of "c"s, have either "a" or "bb" in the middle, and end with any even number of "c"s. Since the production M ::= M c c causes a conflict due to left recursion, we can rewrite M to generate these three segments using separate non-terminals and only right recursion:

S ::= b | M
M ::= c M | N
N ::= a O | b b O
O ::= c c O | ε

As before, we need to perform left factoring because b is in the FIRST set for the right-hand side of both productions of *S*, but first we need to get the grammar in a form that can be factored by substituting *M* in the second *S* production:
S ::= b | c M | N
M ::= c M | N
N ::= a O | b b O
O ::= c c O | ε

We substitute again, this time substituting N in the third production of S:
S ::= b | c M | a O | b b O
M ::= c M | N
N ::= a O | b b O
O ::= c c O | ε

Now we can left factor:
S ::= b T | c M | a O
T ::= ε | b O
M ::= c M | N
N ::= a O | b b O
O ::= c c O | ε

This grammar generates the same language but is now LL(1).

Question 5. (23 points) Compiler hacking: MiniJava++. We would like to add a new ++ pre-increment operator to MiniJava. (A copy of the MiniJava grammar is included at the end of the test for reference if needed.) The new ++ operator is a single token and is a *prefix* operator. We will add the following new rule to the MiniJava grammar:

Expression ::= "++" Expression

The new ++ operator has the same semantics as it does in C/C++/Java and similar languages. The Expression is evaluated, and it must be a l-value, i.e., an assignable location designating a variable, array element, or something similar. The ++ operator increments the contents of that location, and then the returned (computed) value of '++Expression' is the newly incremented value. Since we are extending MiniJava, the operand of ++ must designate a location containing an integer, since those are the only values in the language that can be incremented.

For example, suppose we have the following two statements in a MiniJava program:

```
x = 7;
System.out.println(++x);
```

The `System.out.println` statement should print 8, and the final value of `x` is 8.

Note: for this problem, we are only adding a prefix ++ operator, i.e., for expressions like `++x`. We are ***not*** also including a postfix `x++` operator.

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

Add a PLUSPLUS token to represent ++. (Of course, the new token could have another name, but we do need a separate new token.)

(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 ++ operator. 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`, and `Statement extends ASTNode`. Also remember that each AST node constructor has a `Location` parameter.)

```
public class Increment extends Exp {
  // add instance variables below
```

```
  public Exp e;  // operand of ++
```

```
  // constructor - add parameters and method body below

  public Increment( Exp e, Location pos ){
```

```
    super(pos);

    this.e = e;
```

```
  }
}
```

Note: the constructor parameter order could be reversed as long as it matches the AST node creation code in the CUP specification below.

(continued on next page)

Question 5. (cont.) (c) (5 points) Complete the CUP specification below to add an appropriate grammar rule and semantic actions for the new ++ operator, including creating a new Increment node (as defined in part (b) above). Your CUP grammar rule should use any new tokens or keywords defined in part (a) as appropriate. We have included the production for an Identifier expression (as found in the project starter code) as a reminder; your job is to add an equivalent case for the new increment (++) operator.

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

```
Expression ::= ...
        | IDENTIFIER:name
          {: RESULT = new IdentifierExp(name, namexleft); :}
        | /* add the CUP grammar rule and code for the ++
             operator in the box below */
```

```
PLUSPLUS Expression:e

{: RESULT = new Increment(e, exleft); :}
```

(d) (4 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that an expression using the ++ preincrement operator 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.

- **Verify that the expression argument of ++ designates a location (lvalue).**
- **Verify that the location that it designates has values of type `int`.**

(continued on next page)

Question 5. (cont.) (e) (6 points) Give the code shape for this new ++ preincrement operator, i.e., what code should be generated in the assembly language program to properly evaluate an expression with the new ++ operator.

You should show any instructions, labels, or other assembly language-level details needed, and also show where generated code to evaluate the subexpression that is the operand of the increment ++ operator would appear. Be sure you are clear about which parts of the code are computing or processing addresses of storage locations vs. the contents of those locations (i.e., lvalues vs. rvalues).

Your code does not need to be precisely correct x86-64 assembly code (i.e., you are not expected to have memorized instruction details), but it should be close enough so that your intent is clear and it basically equivalent to real x86-64 code.

Hints: you *definitely* won't need all the space on this page for your answer. But be careful that your code matches the semantics of the ++ operator precisely.

Visit node for Expression – the generated code needs to leave the lvalue location of the Expression in `%rax`

```
movq    %rax,%rdx          # copy expr address to second register
movq    0(%rdx),%rax       # load value
addq    $1,%rax            # increment value – result in %rax
movq    %rax,0(%rdx)       # store incremented value in original location
```

There are, of course, other code sequences that load the value, increment it, store it back, and leave the result in `%rax`. Any correct sequence received full credit. It also is not strictly required to leave the result in `%rax`, although that is the convention we used throughout our codeshape presentations.

Question 6. (14 points) x86-64. Translate the following C function into x86-64 assembly language. You must use the standard x86-64 C language conventions for function calls, register usage, and so forth. Note that this is just a C function, not a Java or MiniJava method, so there are no objects, no `this` pointer, and no vtables involved. You must translate the function exactly as given – you may not omit the recursive function call or other code, or rewrite the calculation into a loop or some other form. Assume that `int`s are 64-bit numbers, as in the MiniJava project.

There is a page at the end of the exam with rules for writing x86-64 code for this question along with some basic x86-64 assembly language reference information. Refer to it while answering this question, and feel free to remove that page from the exam for convenience.

```
// return the sum x+(x+1)+...+(y-1)+y
int sum(int x, int y) {
  if (x == y)
    return x;
  else
    return x + sum(x+1,y);
}
```

```
# argument registers: %rdi = x, %rsi = y
sum:    pushq   %rbp          # prologue - save old frame ptr
        movq    %rsp,%rbp     # set frame pointer
        subq    $16,%rsp      # aligned space for one variable
        cmpq    %rdi,%rsi     # compare x, y
        jne     else          # jump if not equal
        movq    %rdi,%rax     # result = x
        jmp     done
else:   movq    %rdi,0(%rsp)  # save x
        addq    $1,%rdi       # call sum(x+1,y)
        call    sum           # recursive result in %rax
        addq    0(%rsp),%rax  # add saved x to get result
done:   movq    %rbp,%rsp     # free stack frame
        popq    %rbp          # restore frame ptr
        ret                   # return
```

There are, of course, many other possible solutions. Any other correct solution that follows the correct calling and register conventions received full credit.

Question 7. (14 points) Value numbering. For the following block, (i) apply local value numbering (including version numbers for variables) to the statements in the block, then (ii) rewrite the code to eliminate redundant expressions using the value numbering information from part (i).

(i) Show the results of value numbering by writing value numbers and variable version numbers on the code below. Do not change the code (not yet – that's the next part). Recall that the notation v_i^n means that version i of variable v has value number n.

$a_1^1 = x_0^1$

$b_1^3 = x_0^1 + y_0^2$

$c_1^4 = a_1^1 + b_1^3$

$a_2^3 = a_1^1 + y_0^2$

$d_1^3 = b_1^3$

$b_2^5 = d_1^3 + a_2^3$

(ii) Now, rewrite the above code to show the results of eliminating redundant expressions using the value numbering information computed in part (i) above. Write your answer below.

(The redundant expression that is eliminated is the a+y addition that adds value numbers 1 and 2 to generate another copy of value number 3.)

$a_1^1 = x_0^1$

$b_1^3 = x_0^1 + y_0^2$

$c_1^4 = a_1^1 + b_1^3$

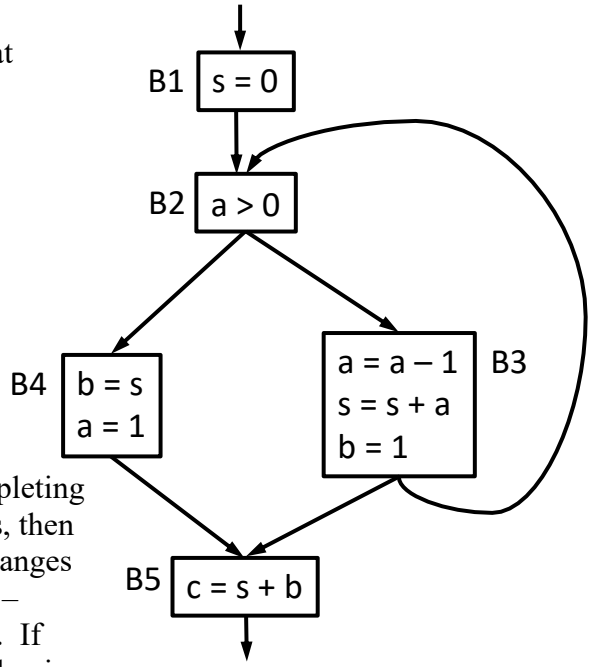
$a_2^3 = b_1^3$

$d_1^3 = b_1^3$

$b_2^5 = d_1^3 + a_2^3$

Question 8. (16 points) Dataflow Analysis – Liveness. In class we looked at the liveness dataflow problem. Recall that a variable *v* is *live* at program point *p* if there is any path from *p* to a use of *v* along which *v* is not defined. For a basic block *b* we define these sets:

- use[*b*] = variables used in *b* before any definition of that variable occurring in *b*.
- def[*b*] = variables defined in *b* and not killed later in *b*.
- in[*b*] = variables live on entry to *b*.
- out[*b*] = variables live on exit from *b*.



These sets are related by these equations:

- in[*b*] = use[*b*] $\cup$ (out[*b*] – def[*b*])
- out[*b*] = $\cup_{s \in succ[b]}$ in[*s*]

Calculate the live variables for each block in this CFG by completing the table below. You should first calculate the use and def sets, then iteratively calculate in and out sets until there are no further changes in the solution. Note that the blocks are listed in reverse order – information for the last block (5) is in the first row of the table. If you need additional space for your answer, continue on the following blank page, and write a note at the end of this page to indicate that you are doing that.

Block	use	def	out	in	out	in	out	in
5	**b, s**	**c**	**--**	**b, s**	**--**	**b, s**		
4	**s**	**a, b**	**b, s**	**s**	**b, s**	**s**		
3	**a, s**	**a, b, s**	**b, s**	**a, s**	**a, b, s**	**a, s**	**SAME**	
2	**a**	**--**	**a, s**	**a, s**	**a, s**	**a, s**		
1	**--**	**s**	**a, s**	**a**	**a, s**	**a**		

CSE P 501 19au Exam 11/21/19 Sample Solution

Reference information for use during the exam. Feel free to remove this page from the exam for convenience while answering questions.

The reverse side of this page contains the grammar for MiniJava.

Reference and ground rules for x86-64 code, (same as for the MiniJava project and other x86-64 code):

- You must use the Linux/gcc assembly language, and must follow the x86-64 function call, register, and stack frame conventions.
  - Argument registers: %rdi, %rsi, %rdx, %rcx, %r8, %r9 in that order
  - Called function must save and restore %rbx, %rbp, and %r12-%r15 if these are used in the function
  - Function result returned in %rax
  - %rsp must be aligned on a 16-byte boundary when a call instruction is executed
  - %rbp must be used as the base pointer (frame pointer) register for this exam, even though this is not strictly required by the x86-64 specification.
- Pointers, Booleans, and ints are 64 bits (8 bytes) each, as in MiniJava.
- Your x86-64 code must implement all of the statements in the original method. You may *not* rewrite the method into a different form that produces equivalent results (i.e., replacing a function call by the function body or rewriting the code to compute the result in a different way from the original). Other than that, you can use any reasonable x86-64 code that follows the standard function call and register conventions – you do not need to mimic the code produced by a MiniJava compiler.
- Please include *brief* comments in your code to help us understand what the code is supposed to be doing (which will help us assign partial credit if it doesn't do exactly what you intended.)