CSE 401/M501 21au Midterm Exam 11/5/21

Name _____ ID # _____

There are 5 questions worth a total of 100 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 books, closed notes, closed electronics. However, you may have one 5x8 notecard for reference with any hand-written information you wish on both sides. Please turn off all cell phones, personal electronics, alarm watches, and pagers, and return your tray tables and seat backs to their full upright, locked positions. Sound or video recording and the taking of photographs is prohibited.

If you have a question during the exam, please raise your hand and someone will come to help you.

There is an extra blank page at the end of the exam you can use if your answer(s) do not fit in the space provided. Please indicate on the original page(s) if your answer(s) is(are) continued on that last page.

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

Score _____

1 _____ / 18

2 _____ / 16

3 _____ / 32

4 _____ / 14

5 _____ / 20

Question 1. (18 points) We are designing a new programming language with the following conventions for method names. There are two types of method names: ordinary method names and private method names. Both sets of names can include combinations of lower-case letters and underscores only – no digits or upper-case letters. A name may not contain two or more consecutive underscore characters. An ordinary method name may not start with a leading underscore, but may contain a trailing underscore. A private method name *must* start with a single leading underscore and cannot end with an underscore. Examples of legal names:

Ordinary method names: `a_method` `another_method_` `simplename` `xy_`
Private method names: `_secret` `_private_method`

Fine print: You must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r*`|`*s* , *r*`*` , *r*`+` , *r*`?`, character classes like `[a-cxy]` and `[^aeiou]`, abbreviations *name=regexp*, and parenthesized regular expressions. No additional operations that might be found in the "regexp" packages in various Unix programs, scanner generators like JFlex, or programming language libraries are allowed.

(a) (8 points) Give a regular expression (possibly with subexpressions if it makes things easier) that generates all valid method names (ordinary and private) according to the above rules.

(b) (10 points) Draw a DFA that accepts all valid method names (ordinary and private) according to the above rules.

Question 2. (16 points) Ambiguity. Here is a small grammar for expressions involving assignment (=) and addition (+) and the terminal symbol `x`.

E ::= `x` = *E* | *E* + `x` | `x`

Is this grammar ambiguous? If so, give a proof that it is by showing two distinct parse trees or two different leftmost (or two different rightmost) derivations for some string generated by the grammar. If not, give an informal, but precise, argument why it is not ambiguous.

Note: whitespace in the grammar is only for readability and is not part of the grammar or the strings generated by it.

Question 3. (32 points) The "OMG it's *back*!!" LR parsing question. Here is a different grammar for the language from the previous problem involving assignment (=) and addition (+) and the terminal symbol `x`. The extra *S′* ::= *A* `$` rule needed to handle end-of-file in an LR parser has been added for you. As is usual, whitespace in the grammar is only for readability and is not part of the grammar or the strings generated by it.

0. *S′* ::= *A* `$` (`$` is end-of-file)
1. *A* ::= `x` = *A*
2. *A* ::= *E*
3. *E* ::= *E* + `x`
4. *E* ::= `x`

(a) (16 points) Draw the LR(0) state machine for this grammar. When you finish, you should number the states in the final diagram in whatever order you wish so you can use the state numbers to answer later parts of this question.

(b) (6 points) Compute *nullable* and the FIRST and FOLLOW sets for the nonterminals *A* and *E* in the above grammar:

Symbol	nullable	FIRST	FOLLOW
A			
E			

Question 3. (cont.) Grammar repeated from previous page for reference:

0. S' ::= A $ ($ is end-of-file)
1. A ::= x = A
2. A ::= E
3. E ::= E + x
4. E ::= x

(c) (5 points) Is this grammar LR(0)? Explain why or why not. Your answer should describe **all** of the problems that exist if the grammar is not LR(0) by identifying the relevant state number(s) in your diagram in part (a) and the specific issues in those state(s) (i.e., something like "state 47 has a shift-reduce conflict if the next input is x", but with, of course, state numbers and correct details from your diagram). If the grammar is LR(0) you should explain why. You do not need to write out the full LR(0) parse tables as part of your answer, just explain whether the state machine is LR(0) or not, and why.

(d) (5 points) Is this grammar SLR? Explain why or why not. As with your answer to the previous part of the question, refer to states by number in the LR diagram in your answer to part (a) and give specific explanations of why the grammar is SLR or why not.

Question 4. (14 points) LL parsing. Here is another look at the grammar from the previous question (the extra $S' ::= A\ \$$ production needed to handle end-of-file in the LR parser has been omitted since it is not needed here).

1. $A ::=$ x = A
2. $A ::= E$
3. $E ::= E$ + x
4. $E ::=$ x

Is this grammar, as written, suitable for constructing a top-down LL(1) predictive parser? If it is, your answer should give a technical explanation why it is. If not, your answer should give a technical explanation listing **all** of the problems with this particular grammar that prevent it from being suitable for a LL(1) predictive parser. You do not need to rewrite the grammar to fix the problems if there are any – just explain why it is or is not suitable for this use. (Hint: Explanations in terms of FIRST/FOLLOW and other properties needed by a LL(1) predictive parser may be helpful.)

Question 5. (20 points) Semantics. Suppose we have the following assignment statement in a MiniJava program:

```
p = x.f(x)<0;
```

(a) (10 points) Draw an abstract syntax tree (AST) for this statement at the bottom of this page. You should use appropriate names for AST nodes and have an appropriate level of abstraction and structural detail similar to the AST nodes in the MiniJava project AST classes, but don't worry about matching the exact names of classes or nodes found in the MiniJava code.

(b) (10 points) Annotate your AST by writing next to the appropriate nodes the checks or tests that should be done in the static semantics/type-checking phase of the compiler to ensure that this assignment statement does not contain errors. You do not need to specify an attribute grammar – just indicate the necessary tests. If a particular test applies to multiple nodes, you can write it once and indicate which nodes it applies to, as long as your meaning is clear and readable. You may assume that `int` is the only numeric type in the language.

Extra space for answers, if needed. Please be sure to label which question(s) are answered here, and be sure to put a note on the question page so the grader will know to look here.