CSE 401/M501 25au Midterm Exam 10/31/25

Name ______________________________________ UW netid ____________@uw.edu

There are 7 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, the taking of photographs, and time travel are prohibited.

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

There is an extra sheet with two blank pages 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.

After the blank pages with extra space for answers is a copy of the MiniJava grammar. You should remove that page from the exam and use it for reference as needed.

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

Score _________________

1 _______ / 20

2 _______ / 10

3 _______ / 36

4 _______ / 8

5 _______ / 10

6 _______ / 14

7 _______ / 2

CSE 401/M501 25au Midterm Exam 10/31/25

Question 1. (20 points) 6-7 Regular expression (and DFA). For this problem, we would like to generate and recognize strings consisting of one or more decimal digit characters with the following properties:

- The string must contain at least one digit 6 and at least one digit 7 character
- The string must not end with the digit 7

Examples of strings that are legal sequences of characters according to these rules: 76 (the smallest possible answer), 0123456789, 676, 007060, 9876, 135798642, etc.

Examples of strings that are not legal sequences of characters according to these rules: 123 (no 6 or 7), 02468 (missing 7), 767 (ends in 7), 007 (ends in 7, missing 6), 13579 (missing 6), 6 (missing 7), 7 (missing 6 and ends with 7).

(a) (10 points) Give a regular expression that generates all strings that are legal strings of digits according to the above rules. You may assume the alphabet is restricted to the decimal digit characters (0-9) if you wish.

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. As long as the solutions are correct, the regular expressions and DFA do not need to be simplified or minimized.

(b) (10 points) Draw a DFA that accepts all valid strings of digits described above and generated by the regular expression from part (a). As in part (a), you may assume the alphabet is restricted to the decimal digit characters (0-9).

Question 2. (10 points) Scanners. A scanner for MiniJava should be able to process any text input, even if it contains something other than a legal MiniJava program. It should report errors if it encounters input characters that do not form proper MiniJava tokens.

What happens if we use a scanner for MiniJava to process the following input? (This is a tiny bash shell script that prints out all of the misspelled words in its argument file. The \` characters in the file are backquote punctuation characters.)

```
wrongWords=`aspell list<$1`
echo ${wrongWords}
```

Below, list in order the tokens that would be returned by a scanner for MiniJava as it reads this input. If there is a *lexical* error in the input, indicate where that error is encountered by writing a *short* note explaining the error in between the valid tokens that appear before and after the error(s) (something *very* brief like "invalid %" would be fine). The token list should include any tokens found after any error(s) in the input, i.e., scanning should continue after discovering an error. You may use any reasonable token names (e.g., LPAREN, ID(x), etc.) as long as your meaning is clear.

A copy of the MiniJava grammar is attached as the last page of the exam. You should remove it from the exam and use it for reference while you answer this question. You should assume that the scanner processes MiniJava syntax as defined in that grammar, with no extensions or changes to the language. Also recall that the MiniJava project defines an <IDENTIFIER> as a sequence of letters, digits, and underscores, starting with a letter; uppercase letters are distinguished (different) from lowercase; and an <INTEGER_LITERAL> is a sequence of decimal digits not starting with 0, or the number 0 by itself, denoting a decimal integer value.

Question 3. (36 points) Trick or Treat! 👻🎃🧹🦇 Here it is – the usual haunted grammar question. Consider the following grammar. The nonterminal *B* is the start symbol of the grammar, and the extra *B'* ::= *B* $ rule needed to handle end-of-file in an LR parser has been added for you.

0. *B'* ::= *B* `$` ($ is EOF)
1. *B* ::= `b` *A*
2. *A* ::= `a` *A* `t`
3. *A* ::= `a t`

(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 that you can use the state numbers in later parts of this question. The state numbers should be successive integers starting with 0, 1, 2, 3, ….

(continued on next page)

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

0. B' ::= B `$` (`$` is EOF)
1. B ::= `b` A
2. A ::= `a` A `t`
3. A ::= `a t`

(b) (8 points) Write the LR(0) parser table for the LR parser DFA shown in your answer to part (a). To save time, an empty table is provided below. However, it probably has more rows than you need. Use only as many rows as needed and leave the rest blank.

State #	`a`	`b`	`t`	`$`	A	B
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

(continued on next page)

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

0. B' ::= B `$` (`$` is EOF)
1. B ::= `b` A
2. A ::= `a` A `t`
3. A ::= `a t`

(c) (3 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 answers to parts (a) and (b) and the specific issues in those state(s) (i.e., something like "state 67 has a shift-reduce conflict if the next input is `foo`", but with, of course, correct state numbers and details from your parser). If the grammar is LR(0), you should give a technical explanation why it is (this can be brief).

(d) (6 points) Complete the following table showing the FIRST and FOLLOW sets and nullable for each of the nonterminals in this grammar. You should include $ (the end-of-file marker) in the FOLLOW set for any nonterminal where it is appropriate.

	FIRST	FOLLOW	nullable
A			
B			

(e) (3 points) Is this grammar SLR? Give a brief technical explanation why or why not.

Question 4. (8 points) Top-down parsing. Take another look at the grammar from the previous problem, but omitting the *B'* ::= *B* $ rule that was added for LR parsing:

1. *B* ::= `b` *A*
2. *A* ::= `a` *A* `t`
3. *A* ::= `a t`

(Recall that *B* is the start symbol for this grammar.)

Is this grammar suitable for constructing a top-down LL(1) predictive parser? If so, explain why. If not, explain why not, and, if possible, construct a different grammar that generates the same language that is suitable for a top-down LL(1) predictive parser, or explain why this can't be done. (You might find the FIRST/FOLLOW/Nullable information from the previous problem useful in answering this question.)

Question 5. (10 points) Grammars and ambiguity. Suppose we have the following traditional unambiguous grammar for expressions involving identifiers and the `*` and + operators.

expr ::= *expr* + *term* | *term*

term ::= *term* * *factor* | *factor*

factor ::= *id* | `(` *expr* `)`

id ::= `a` | `b` | `c` | `w` | `x` | `y` | `z`

In this grammar, `*` has higher precedence than +, and both `*` and + operators are left-associative, so `w+x*y+z` means `(w+(x*y))+z`.

To make our language more useful for bit manipulation problems, we would like to add a left-shift operator << . As in Java/C/C++, the new << operator has lower precedence than the + and `*` operators. Like those operators, << is also left-associative. So if we write `x<<y<<z` it means `(x<<y)<<z` and the expression `w<<x+y*z<<z` means `(w<<(x+(y*z))<<z`.

Below, give a modified version of the above grammar that includes the new << operator so that it has lower precedence than both + and `*` and is left associative. In the resulting grammar, `*` should still have higher precedence than +, and those operators should remain left associative. It also should still be possible to surround any expression or subexpression with parentheses like `(x<<y)` as in the original grammar. The resulting grammar should be unambiguous like the original grammar, and should be able to generate the same strings as the original grammar plus expressions involving the new << operator with appropriate precedence and associativity. To save time, you do not need to copy the grammar rule for *id* – assume that it will be the same in the new grammar, or you can write in your changes on the original grammar above instead of copying it.

Question 6. (14 points) Semantics. Many programming languages include a `**` operator for exponentiation. The meaning of `x**y` is to compute x^y. The `**` operator is right-associative, so `x**y**z` means `x**(y**z)`. The new `**` operator has higher precedence than the other arithmetic operators like `+-*`. Suppose we add a `**` exponentiation operator to MiniJava to provide integer exponentiation for integer values (only) without making any other changes to the language, and suppose we encounter the following assignment statement in a MiniJava program that uses this extension:

```
distSquared = (x2 - x1) ** 2 + (y2 - y1) ** 2;
```

(a) (7 points) At the bottom of this page, draw an abstract syntax tree (AST) for this assignment statement. You should use appropriate names or symbols for the 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 or details of classes or nodes found in the MiniJava starter code.

(b) (7 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 statement does not contain errors. If a particular check or 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.

Question 7. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

(b) (1 points) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

Yes, yes, it *must* be included!!!

No opinion / don't care

None of the above. My answer is __________________________________.

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.

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.