Exam 1 - 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.

601.628 Compilers and Interpreters Fall 2020, Exam 1 - 628

Due: Wednesday, September 30 by 3:00pm EDT

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 1 - 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.

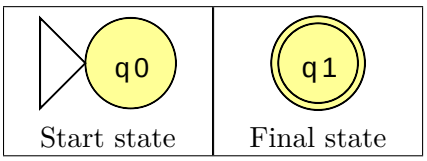
Important: Make sure the first page of your submitted document is the signed and dated cover sheet (which is the first page of this exam document.) You will not receive credit for the exam if your submission does not include the (signed and dated) cover sheet.

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

In all questions:

ϵ (epsilon) means the empty string of symbols.

For questions about finite automata, draw the start state and final state(s) like this:



When drawing transitions, make sure that the direction is clearly indicated.

[Questions begin on next page.]

A programming language has 5 kinds of tokens: identifiers, integer literals, left parenthesis, right parenthesis, and the **fn** keyword. An identifier is a sequence of 1 or more occurrences of the characters "a", "f", and "n". An integer literal is a sequence of one or more occurrences of the characters **0** and **1**. Left and right parentheses are the "(" and ")" symbols. The **fn** keyword has the lexeme "fn". Tokens may be separated by occurrences of the space ("␣") character.

Note that there is ambiguity between identifiers and the **fn** keyword. In any case where a token begins with "fn", and is not followed by another letter ("a", "f", or "n"), it is an occurrence of the **fn** keyword.

Some examples:

Lexeme	Kind of token
a	identifier
af	identifier
fn	**fn** keyword
fna	identifier
1	integer literal
10	integer literal
(	left parenthesis
)	right parenthesis
␣␣	whitespace (not really a token)

Question 1. [8 points] Specify a regular expression for each of the 5 kinds of tokens in the example programming language, as well as a regular expression matching whitespace separating tokens. (So, a total of six regular expressions.) Note that because parentheses are used for grouping in regular expressions, you should use \(and \) to denote literal left and right parenthesis characters.

Question 2. [8 points] Specify a *deterministic* finite automaton (DFA) for each of the 5 kinds of tokens, and also whitespace for separating tokens, in the example programming language. (So, a total of six finite automata.)

Question 3. [8 points] Construct a single *nondeterministic* finite automaton (NFA) from the deterministic finite automata (DFAs) you specified in Question 2. The NFA should have a single start state, should recognize the union of the languages recognized by the DFAs, and should retain the structure of the original DFAs. In other words, don't make any structural changes to the DFAs. (The slides for Lecture 4 show how to do construct a unified NFA from per-token DFAs.)

Question 4. [16 points] Convert the NFA in Question 3 to a DFA, using the algorithm demonstrated in class. **Important:** show the mapping for each reachable set of NFA states to a corresponding DFA state. Make sure that the transitions of the DFA are clearly labeled with a single input symbol. Do *not* minimize the DFA.

Question 5. [10 points] Assume that the DFA you constructed in Question 4 is used as the basis for a lexical analyzer. Briefly explain, in your own words, how the lexical analyzer can determine that the lexeme "fn" is the **fn** keyword and not an identifier.

[Questions continue on next page]

The syntax for example programming language discussed earlier is as follows. Nonterminals are in *italic*, terminals are **boldface**. The start symbol is *exp-list*.

exp-list → *exp*
exp-list → *exp exp-list*
exp → **integer-literal**
exp → **identifier**
exp → **(** *exp-list* **)**
exp → **(** **fn** **(** **identifier** **)** *exp* **)**

Question 6. [8 points] Show a derivation for the following input:

```
((fn (a) (fff a 101)) 110)
```

Note that following the lexical conventions described previously, "a" and "fff" are identifiers, "fn" is the **fn** keyword, and "101" and "110" are integer literals.

Question 7. [8 points] Draw the parse tree corresponding to the derivation in Question 6.

Question 8. [8 points] Show the FIRST and FOLLOW sets for the *exp* symbol.

Question 9. [16 points] Assume that you will implement a recursive descent parser for this programming language. Show pseudo-code for the parse function for the *exp* nonterminal symbol. You may assume that the lexical analyzer supports `peek` and `next` operations, and that there is an `expect` function which you can use to consume a specific kind of token.

Question 10. [10 points] Is the above grammar suitable for parsing using LL(1)? Briefly explain why or why not.