CS 321 Languages and Compiler Design I — Midterm Fall 2013

NAME:

Instructions:

- Do not turn over this page until the test begins.
- Time allowed is 100 minutes (ONE hour and FORTY minutes). You may leave early if you wish, but please do so as quietly as possible to avoid disturbing other students.
- We **do not** expect that you will be able to complete answers to all of the questions in the available time. Our best guess is that you will only have enough time for around 2/3 of the questions, but it could be more or less. As such, we encourage you to answer as many questions (in full or in part) as you are able, and to be strategic in deciding which questions you choose to answer. Do not be discouraged if you do not finish everything; well account for this in the way that the exam is graded.
- You may only bring basic writing tools (pen, pencil, ruler, ...) in to this test with you. No other materials, including textbooks, calculators, notes, computers, phones, tablets, teleportation devices, etc. may be used.
- Do all your work on these sheets. You may request additional paper for rough work during the test by raising your hand. All of the paper that you use should be clearly labeled with your name and should be turned in at the end of the test.
- Try to keep your answers as clear, as concise, and as legible as possible. None of these problems requires very long answers, but you should aim to make good use of the space provided.
- The points for each section are shown in parentheses on the right-hand side. Note that some questions are worth more than others. Credit will be given for partial answers, and for correct working or explanation, even if the final answer is incorrect. If you make any assumptions about the interpretation of a question, be sure to document that in your solution.
- Read the questions carefully. You may ask for clarification at any time during the exam. If you feel that you need to make additional assumptions in order to answer a question, be sure to explain that as part of your answer.
- Discussion of this exam with colleagues or classmates is not allowed until all solutions have been turned in.
- Breaches of academic integrity will be treated very seriously.

1) **Fundamental Concepts**:

Comment briefly on the distinctions between each of the following pairs of terms in the context of programming languages: **(10)**

i) *syntax* and *semantics*

ii) *concrete syntax* and *abstract syntax*

iii) *interpreters* and *compilers*

iv) the *front-end* and the *back-end* of a compiler

v) *lexemes* and *tokens*

2) **Compiler Goals and Structure**:

a) Explain what is meant by *compiler correctness*, and identify other key properties or features of compilers that are likely to be important to their users. **(4)**

b) Explain the role of the following three processes in a typical compiler. In each case, your answer should mention the kind of information that is manipulated, and the kinds of error conditions that might be detected. **(6)**

lexical analysis:

parsing:

code generation:

Continues on next page ...

3) **Lexical Analysis**:

a) A *bracketed comment* in a language like C, C++, or Java begins with the two character sequence "/*" and ends with the two character sequence "*/". Draw a diagram of a *deterministic* finite automaton (DFA) that will recognize all valid comments of this form. **(4)**.

b) Using the DFA from (a) as a guide, and showing the steps in the method that you use, construct a regular expression for matching bracketed comments. **(6)**

c) Bracketed comments in C, C++, and Java do not nest, meaning that each such comment terminates at the *first* occurrence of the "`*/`" sequence after the opening "`/*`". For example, an input of the form "`x + /* y + /* z + */ t */`" will be treated as equivalent to "`x + t */`" (which, of course, will result in a syntax error). Suppose that we want to build a compiler for a new programming language, Novel, that supports *nested* comments of this form. In particular, this means that each opening "`/*`" should be paired with a corresponding closing "`*/`". With this approach, the example input described previously will be treated as equivalent to "`x +`". Explain briefly why neither of the following two techniques can be used directly to implement this feature. [*Note:* be sure to include appropriate examples, or references to well-known results, to justify your answers.]

i) *Maximal munch* (also sometimes known as the *longest lexeme* rule), which would correctly match the first "`/*`" with the second "`*/`" in the example above. **(3)**

ii) Using a tool like `jflex` to build a lexer with a regular expression that can match a bracketed comment. **(3)**

d) Outline an alternative strategy that could be used to implement support for bracketed comments in an implementation of the Novel language. **(2)**

Continues on next page ...

4) **Regular Expressions**:

a) A vending machine has been designed to deliver chocolate treats to a purchaser who deposits 25 cents into the machine using a combination of quarters (25 cents each, abbreviated Q), dimes (10 cents each, abbreviated D), and nickels (5 cents each, abbreviated N). Draw a minimal DFA (omitting stuck states and any transitions to such states) to describe all of the different sequences of coins that can be used to complete a purchase. [*Hint:* Each state in your DFA will likely correspond to an amount of money that must be paid to complete the transaction.] **(3)**

b) Construct a regular expression over the alphabet {Q, D, N} to describe all, and only, the sequences of coins that your machine can recognize. [*Hint:* To simplify the task, you are encouraged to introduce simple names as abbreviations for more complex regular expressions, just as you might do using a tool like `jflex`.] **(3)**

c) There are at least two different ways for a vending machine to respond if a purchaser inserts a coin that is too large: (i) Return the coin immediately to the purchaser and await payment (in smaller coins); or (ii) accept overpayment and complete the transaction, providing the purchaser with appropriate change. Explain how your DFA can be modified to account for the sequences of coins corresponding to completed transactions in each of these two approaches. [*Note:* It is enough just to describe the extra transitions and/or states that you would add; you do not need to draw the modified DFAs, or to add any new symbols corresponding to rejecting coins or providing change.] **(4)**

5) **Grammars and Parsing, True or False?**

Select true/false for each of the following statements. Include a brief comment to justify each of your answers. **(8)**

T F Multiple derivations can correspond to a single parse tree.

T F A grammar cannot be LL(1) if any two productions have a common look-ahead symbol.

T F If a context-free grammar does not have left-recursion, it can always be implemented by an LL(k) parser with a fixed k.

T F There are three types of conflict in a bottom-up parser: reduce/reduce, shift/reduce, and shift/shift.

T F Using the same naming convention as LL parser and LR parser, a top-down parser that reads input from right to left should be called an RR parser.

T F Right-recursive grammars cannot be implemented by bottom-up parsers, just like left-recursive grammars cannot be implemented by top-down parsers.

T F Regular expressions are more powerful than LL(0) grammars. (*Hint:* Think what kind of language an LL(0) grammar can generate.)

T F In any expression grammar, replacing a production of the form `E -> E` $\oplus$ `T` by `E -> T` $\oplus$ `E`, where `T` is a non-terminal and $\oplus$ is a binary operator, would not affect the set of expressions the grammar can generate.

Continues on next page ...

6) **Ambiguity and Operator Precedence**:

Consider the following grammar G_1, in which $+$, $-$, and x are terminals:

$$E \rightarrow E + E \qquad E \rightarrow -E \qquad E \rightarrow x$$

a) Show that this grammar is ambiguous by finding a sentence with two distinct parse trees. Draw the two trees. [*Hint:* There is a sentence of this nature with length ≤ 5.] **(4)**

b) An attempt to fix the ambiguity problem in G_1 resulted in the following new grammar G_2:

$$E \rightarrow E + T \qquad E \rightarrow T \qquad T \rightarrow -E \qquad T \rightarrow x$$

By finding two distinct right-most derivations for the same sentence, show that this grammar is still ambiguous. [*Note:* You may draw parse trees to help, but the answer must be presented in the form of derivations.] **(4)**

c) Assume the operator $+$ is left-associative, and the operator $-$ has a higher precedence order than $+$. Use this information to transform the grammar G_1 into an equivalent but unambiguous grammar G_3. **(2)**

7) **Grammars for Top-Down Parsing**:

Lisp's arithmetic operations can take any number of operands greater or equal to one. It uses a prefix notation, and has a left-to-right associativity. For instance, `(- 10 2 3)` evaluates as `((10-2)-3) = 5`. For the single-operand cases, the semantics is given as: `(+ n) = (+ 0 n)` and `(- n) = (- 0 n)`, so `(- 5)` evaluates to `-5`.

a) Consider the following two proposed grammars for Lisp's arithmetic operations. In both grammars E and L are nonterminals, other symbols are terminals.

(1)
$$\begin{aligned} E &\rightarrow (+\,L\,) \\ E &\rightarrow (-\,L\,) \\ L &\rightarrow L\ num \\ L &\rightarrow num \end{aligned}$$

(2)
$$\begin{aligned} E &\rightarrow (+\,L\,) \\ E &\rightarrow (-\,L\,) \\ L &\rightarrow num\ L \\ L &\rightarrow num \end{aligned}$$

Which of the two grammars is/are suitable for top-down parsing? Which of the two grammars is/are suitable for bottom-up parsing? **(2)**

b) How many lookaheads are needed for the grammar that is suitable for top-down parsing? **(2)**

c) Transform the top-down grammar into an LL(1) grammar. Express the grammar in (non-extended) BNF. **(2)**

d) Construct an LL(1) parsing table for this grammar. You may use whatever method you see fit. **(4)**

Continues on next page ...

8) **LL(1) Parse Table Construction**:

In the following grammar, S is the start symbol, a, b, c, and d are terminals; and \$ is the end-of-file marker. (The augmented production is introduced to help computing the follow set for S.)

0. $S_0 \rightarrow S\,\$$ (augmented production)
1. $S \rightarrow P\,Q$
2. $P \rightarrow a\,b\,P$
3. $P \rightarrow c$
4. $P \rightarrow \epsilon$
5. $Q \rightarrow a\,Q$
6. $Q \rightarrow d$

a) Compute the nullable predicate, and the FIRST and FOLLOW sets for the nonterminals in this grammar, putting your answers in the table. **(4)**

	nullable?	*FIRST Set*	*FOLLOW Set*
S			
P			
Q			

b) Compute the PREDICT sets for the productions, putting your answers in the table. **(4)**

	PREDICT Set
1. $S \rightarrow P\,Q$	
2. $P \rightarrow a\,b\,P$	
3. $P \rightarrow c$	
4. $P \rightarrow \epsilon$	
5. $Q \rightarrow a\,Q$	
6. $Q \rightarrow d$	

c) Construct an LL(1) parsing table for this grammar. **(3)**

d) Is this grammar LL(1)? Why or why not? **(1)**

9) **Bottom-up Parsing**:

Consider the following grammar (the same grammar was used in Question 8, but the two questions can be answered independently):

0. $S_0 \rightarrow S\,\$$ (augmented production)
1. $S \rightarrow P\,Q$
2. $P \rightarrow a\,b\,P$
3. $P \rightarrow c$
4. $P \rightarrow \epsilon$
5. $Q \rightarrow a\,Q$
6. $Q \rightarrow d$

a) Find a right-most derivation for the input sentence $a\,b\,a\,b\,c\,a\,d$. **(2)**

b) Using the stack & input-buffer model, trace the operation of a shift/reduce parser for this grammar on the input. For each reduce action, indicate which production is used. **(6)**

Stack	*Input Buffer*	*Action*

[This page may be used for rough working and notes.]