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*

Answer: *Syntax* has to do with how programs are written, *semantics* has to do with what programs mean. (Week 1, Slide 36)

ii) *concrete syntax* and *abstract syntax*

Answer: *Concrete syntax* describes the particular sequences of bytes/characters/lines that must be entered to construct a program. *Abstract syntax* captures the essential structure of a program independent of written form. (Week 2, Slides 7–8)

iii) *interpreters* and *compilers*

Answer: *Interpreters* execute programs (interpreters "do"). *Compilers* translate programs (compilers "think about doing"). (Week 1, Slide 9)

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

Answer: *Front-end*: those parts of the compiler having most to do with the source language (such as source input, lexing, parsing, and static analysis). *Back-end*: those parts of the compiler having most to do with the target language (such as code generation, runtime organization, and optimization). (Week 1, Slide 51)

v) *lexemes* and *tokens*

Answer: *Lexemes* are specific sequences of characters that should be treated together as a single logical entity in source programs. *Tokens* are used to classify and distinguish between different sets of lexemes. (Week 2, Slides 14–15)

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)**

Answer: *Correctness* requires that the compiler translates valid source programs to valid target programs (1 point) in a way that preserves meaning (1 point; Week 1 Slide 7). *Other key properties* include: performance (in time/space) of compiled code; performance (in time/space) of compiler; good quality diagnostics; support for large projects; and convenient development environment. (up to 2 points; Week 1 Slides 7–8)

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:

Answer: Turns a stream of characters into a stream of tokens. Flags errors having to do with malformed tokens (e.g., unterminated comment, out of range literal) or illegal input character. (Week 1, Slide 18 + Week 2, Slide 48)

parsing:

Answer: Turns stream of tokens into abstract syntax tree. Flags syntax errors where input does match the grammar of the language. (Week 1, Slide 19)

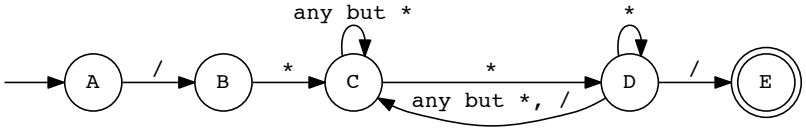
code generation:

Answer: Uses validated abstract syntax tree (output from static/semantic analysis) to produce equivalent target language code (either in abstract syntax form, or even plain text). If the compiler is correct, and if there are no resource problems (e.g., memory or disk), then no errors should occur during this process. (Week 1, Slide 21 + Sample solutions for HW1)

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)**.

Answer: (Week 2, Slide 47)



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)**

Answer: To start, write equations describing the languages that can be recognized from each DFA state: $A = /B$, $B = *C$, $C = (*D \mid uC)$, $D = (*D \mid /E \mid vC)$, and $E = \varepsilon$. (Here u and v are regexps that match any character except `*` and any character except `*` or `/`, respectively.) From $C = (*D \mid uC)$, we can conclude that $C = u^**D$. And then we can simplify the equation for D to $D = (*D \mid vu^**D \mid /) = ((*|vu^**)D \mid /)$. Hence $D = ((*|vu^**)^*/)$, and so $A = /* \; u^** \; (*|vu^**)^* \; */$. (Alternative, equivalent regexps or derivations, including some shortcuts, accepted; students have seen a similar regexp for this problem in HW2.)

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)**

Answer: Consider the examples: "`/* xxx */ yyy */`" and "`/* xxx */ yyy /* zzz */`". In both cases, the longest lexeme rule gives the wrong result because the comment should stop after the first "`*/`", making "`yyy`" visible, instead of matching the longest lexeme and consuming (at least) the second "`*/`".

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

Answer: This corresponds to a language of nested brackets, which is well-known as a language that requires a CFG and cannot be described by a regexp.

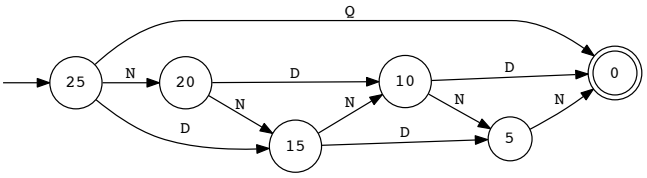
d) Outline an alternative strategy that could be used to implement support for bracketed comments in an implementation of the Novel language. **(2)**

Answer: This can be handled by using a counter to keep track of the nesting level (1 point). In more detail, the lexer would initialize the counter to 1 after the first "`/*`"; increment/decrement the counter on each subsequent "`/*`" or "`*/`"; and terminate the comment when the counter returns to zero (1 point).

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)**

Answer:



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)**

Answer: Writing an equation for each state in the DFA gives:

$$\begin{aligned} owes_{25} &= N\ owes_{20} \mid D\ owes_{15} \mid Q \\ owes_{20} &= N\ owes_{15} \mid D\ owes_{10} \\ owes_{15} &= N\ owes_{10} \mid D\ owes_{5} \\ owes_{10} &= N\ owes_{5} \mid D \\ owes_{5} &= N \end{aligned}$$

There are no cycles here (if $owes_n$ is defined in terms of $owes_m$, then $n > m$), so a regular expression for the full machine can now be obtained by repeated inlining in the definition for $owes_{25}$.

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)**

Answer: For (i): Add self loops on 20, 15, 10, and 5 for Q, and on 5 for D. For (ii): Add extra accept states for -20, -15, -10, -5 representing possible overpayments with corresponding transitions, such as $20 \longrightarrow -5$ on Q. (It's also ok if the student just suggests adding a single "overpay" state, or even just reuses the 0 accept state.)

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.

Answer: T (E.g. both left-most derivation and right-most derivation correspond to a single parse tree.)

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

Answer: F (Productions for different non-terminals can have a common look-ahead symbol. E.g. The grammar "`S -> a X; X -> a`" is LL(1).)

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.

Answer: F (A non-left-recursive CFG can be ambiguous, which cannot be implemented by any LL(k) parser.)

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

Answer: F (A shift moves a token from input to the stack. It's an unique single action. So there can't be shift/shift conflict.)

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.

Answer: T (Reading from right to left, a top-down parser at any step will recurse on the right-most non-terminal first, resulting in a right-most derivation. Hence it 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.

Answer: F (Bottom-up parsing is based on finding handles; it can handle both forms of recursion.)

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

Answer: T (In an LL(0) grammar, since there is no selection possible, any non-terminal can have only one (non-recursive) production. As a result, any such grammar can generate only one fixed string.)

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.

Answer: F (Consider the grammar "`E -> E + T |` ϵ`; T -> x`". It generates a string "`+ x`". Replacing "`E -> E + T`" with "`E -> T + E`" will make it impossible to generate the same string.)

6) **Ambiguity and Operator Precedence**:

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

$$E \to E + E \qquad E \to -E \qquad E \to x$$

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)**

Answer:

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

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

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)**

Answer:
$E \Rightarrow -E \Rightarrow -E + E \Rightarrow -E + x \Rightarrow -x + x$
$E \Rightarrow E + E \Rightarrow E + x \Rightarrow -E + x \Rightarrow -x + x$

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)**

Answer: $E \to E + T \qquad E \to T \qquad T \to -T \qquad T \to x$

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

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

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)**

Answer: Grammar (1) contains left-recursion, hence is not suitable for top-down parsing. Grammar (2) is. Both grammars are suitable for bottom-up parsing.

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

Answer: Two.

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

Answer:
1. $E \rightarrow (\ F \ )$
2. $F \rightarrow + \ L$
3. $F \rightarrow - \ L$
4. $L \rightarrow num \ M$
5. $M \rightarrow L$
6. $M \rightarrow \epsilon$

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

Answer:

	num	+	−	(	)	$
E				1		
F		2	3			
L	4					
M	5				6	

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)**

Answer:

	nullable?	*FIRST Set*	*FOLLOW Set*
S	no	a, c, d	\$
P	yes	a, c	a, d
Q	no	a, d	\$

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

Answer:

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

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

Answer:

	a	b	c	d	\$
S	1		1	1	
P	2,4		3	4	
Q	5			6	

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

Answer: It is not an LL(1) grammar, since the LL(1) parsing table is invalid.

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)**

Answer: $S \Rightarrow PQ \Rightarrow PaQ \Rightarrow Pad \Rightarrow abPad \Rightarrow ababPad \Rightarrow ababcad$.

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)**

Answer:

Stack	*Input Buffer*	*Action*
	$a\,b\,a\,b\,c\,a\,d$	s
a	$b\,a\,b\,c\,a\,d$	s
$a\,b$	$a\,b\,c\,a\,d$	s
$a\,b\,a$	$b\,c\,a\,d$	s
$a\,b\,a\,b$	$c\,a\,d$	s
$a\,b\,a\,b\,c$	$a\,d$	r3
$a\,b\,a\,b\,P$	$a\,d$	r2
$a\,b\,P$	$a\,d$	r2
P	$a\,d$	s
$P\,a$	d	s
$P\,a\,d$		r6
$P\,a\,Q$		r5
$P\,Q$		r1
S		a