CSE 401/M501 19au Midterm Exam 11/1/19 Sample Solution

Question 1. (16 points) A traditional regular expression question.

A terminal command in Linux consists of a single command name followed by zero or more options. For this problem, a command name is a sequence of one or more lower-case letters. An option consists of a single dash followed by a single lower-case letter, or two dashes followed by one or more lower-case letters. A single space (only) is used to separate command names from any following options and to separate options from one another. Examples: **`gcc␣-v␣-g`** and **`gcc␣--version␣-g`**. In these examples, the ␣ character (a horizontal bracket) is used to represent a single space. You should do the same in your answers.

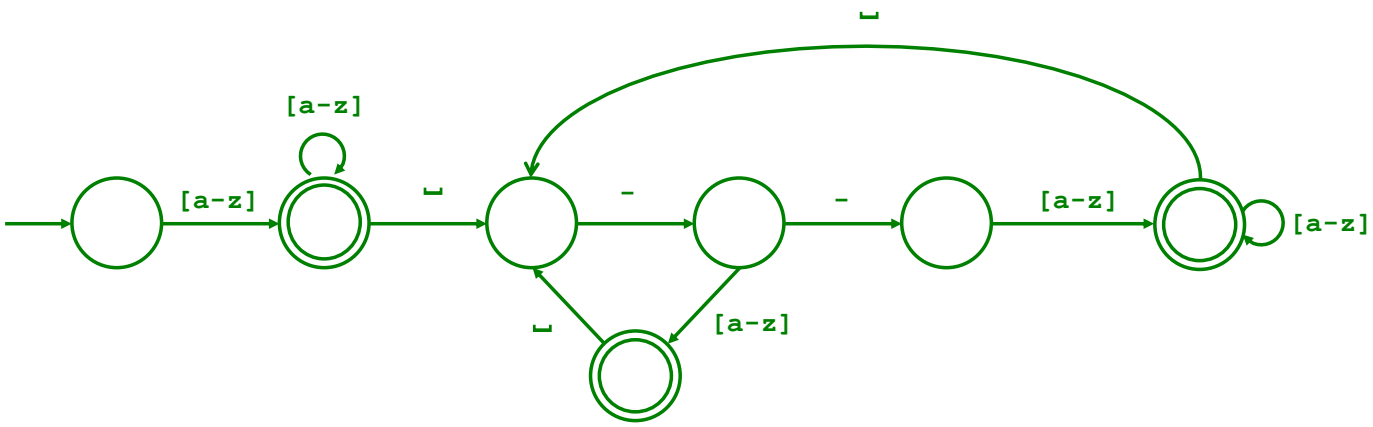
As with homework problems, you must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r* | *s* , *r** , *r*+ , *r*?, character classes like `[a-cxy]` and `[^aeiou]`, abbreviations *name=regexp*, and parenthesized regular expressions. No additional operations that might be found in the "regexp" packages in various Unix programs, scanner generators like JFlex, or language libraries are allowed. Use the ␣ character to represent a single space, as in the examples above.

(a) (8 points) Give a regular expression (possibly with subexpressions if it makes things clearer) that generates all valid Linux commands according to the above rules.

letter = [a-z]

letter+ (␣ - letter | ␣ -- letter+) *

(b) (8 points) Draw a DFA that accepts all valid Linux commands according to the above rules.



Question 2. (8 points). A not so traditional regular expression question.

We would like to write a regular expression for the following language: all strings consisting of **a**'s, **b**'s, and **c**'s, including the empty string, that do not contain the substring **abc**.

After staying up all night to watch the 2019 compilers world championships, your instructor came up with the following regular expression to generate this language. Unfortunately, it appears not to be correct.

(**b** | **c** | **a**[^**b**] | **ab**[^**c**])*

(a) (4 points) Write two strings that are generated by this regular expression but that are ***not*** part of the specified language (i.e., strings that contain the substring **abc**, or are equal to that string, which should not be generated by a correct solution).

There are many possibilities. Here are two simple ones:

aabc

ababc

(b) (4 points) Write two strings that are ***not*** generated by this regular expression but should be (i.e., strings that are part of this language but are not generated by the regular expression).

Again, there are many possible answers. Here are a few:

a

ab

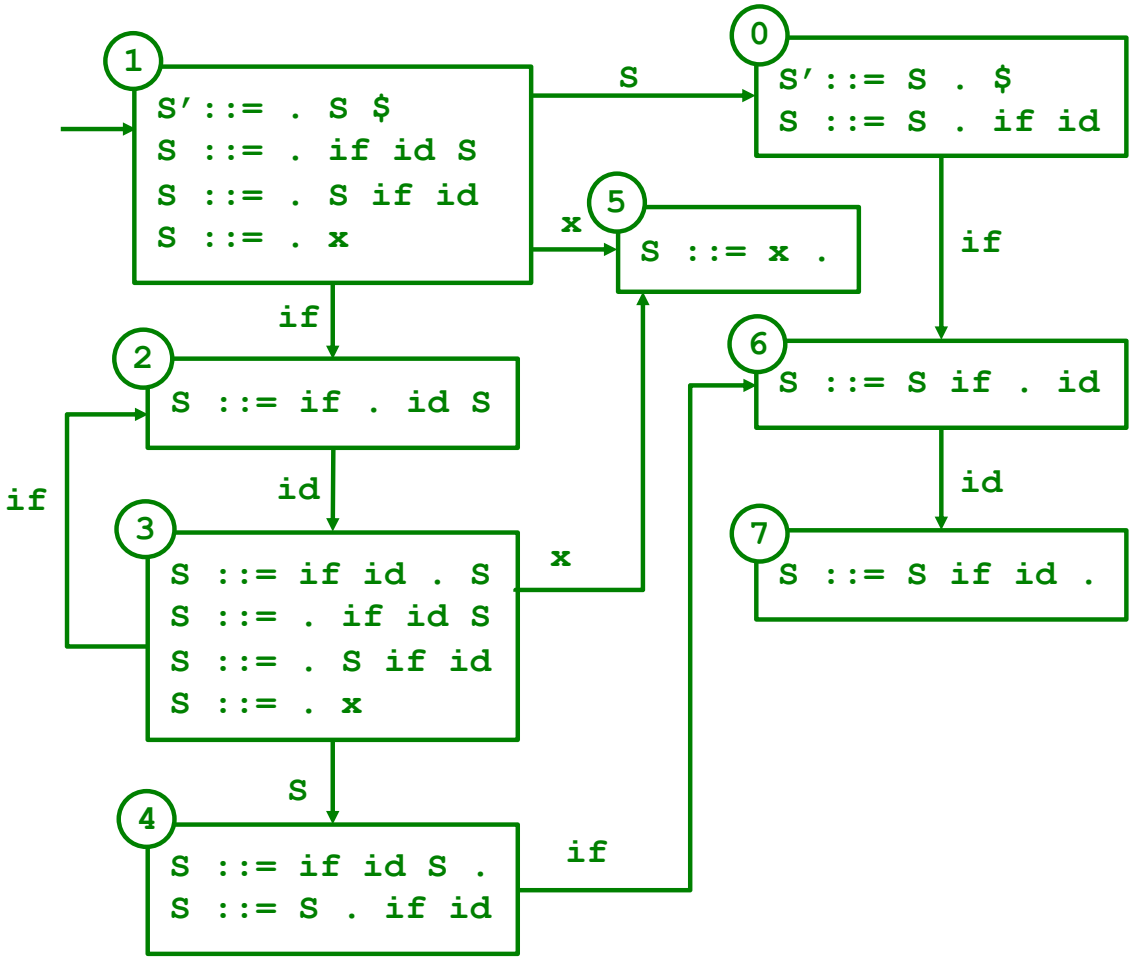
ba

bab

Question 3. (32 points) The traditional LR parsing question. In most programming languages the conditional statement is "if *condition statement*". But in English we can write "if it's cold, wear a hat" or "wear a hat if it's cold". This question uses a very simplified grammar for conditional statements that includes the second pattern. The non-terminal *S* stands for *statement*, and all other things in the grammar rules are terminal symbols. The extra *S'*::=*S*$ rule to handle end-of-file has already been added for you.

0. *S'* ::= *S* `$` ($ represents end-of-file)
1. *S* ::= `if id` *S*
2. *S* ::= *S* `if id`
3. *S* ::= `x`

(a) (14 points) Draw the LR(0) state machine for this grammar.



(b) (4 points) Compute *nullable* and the FIRST and FOLLOW sets for the single nonterminal *S* in the above grammar:

Symbol	nullable	FIRST	FOLLOW
S	No	`if x`	`if $`

(continued on next page)

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

0. *S′* ::= *S* `$` ($ represents end-of-file)
1. *S* ::= `if id` *S*
2. *S* ::= *S* `if id`
3. *S* ::= `x`

(c) (10 points) Write the LR(0) parse table for this grammar based on the LR(0) state machine in your answer to part (a).

State	Shift/Reduce Actions				GOTO
	`if`	`id`	`x`	$	*S*
0	s6			acc	
1	s2		s5		g0
2		s3			
3	s2		s5		g4
4	r1, s6	r1	r1	r1	
5	r3	r3	r3	r3	
6		s7			
7	r2	r2	r2	r2	

(d) (2 points) Is this grammar LR(0)? Explain why or why not.

No. There is a shift-reduce conflict in state 4 if the next symbol is `if`.

(e) (2 points) Is this grammar SLR? Explain why or why not.

No. The follow set of *S* includes `if`, so the SLR construction would not remove the r1 action from state 4 when the next symbol is `if`. The shift-reduce conflict in that state remains.

Question 4. (12 points) Ambiguity: a question where the "wrong" answer is the right one this time!

When we started looking at ambiguity, we looked at a simple expression grammar like this one (using only addition and multiplication operators and identifiers for simplicity).

exp ::= *exp* + *exp* | *exp* * *exp* | id

We observed that this grammar was ambiguous both because it did not enforce left associativity (grouping of subexpressions from left to right) and it didn't handle operator precedence properly (multiplication should have higher precedence than addition).

Write a grammar for arithmetic expressions using +, *, and `id` that correctly enforces precedence between operators (* has higher precedence than +), **but** that **still** has ambiguous associativity. The resulting grammar should generate the same language (set of strings) as the original grammar above but be ambiguous because of the associativity only, not the operator precedence.

Here is one possibility:

exp* ::= *exp* + *exp* | *term

***term* ::= *term* * *term* | `id`**

Question 5. (16 points, 4 each) LL grammars. For each of the following grammars, indicate whether the grammar satisfies the LL(1) condition (meaning it is suitable for a top-down predictive parser). If the grammar does not satisfy the LL(1) condition give a brief, technical description of why not. Your reason must mention a specific terminal symbol or symbols in the grammar that causes the problem in terms of any relevant FIRST or FOLLOW sets. (You do not need to compute complete FIRST / FOLLOW / nullable information for any non-terminal in the grammar, but you should include whatever information about these you find useful in your explanations.)

Whitespace in the grammar rules is only for clarity and is not part of the grammar.

(a) S ::= W `f` | `b` | W
W ::= ε | S `b`

Not LL(1) for many reasons. Both *S* and *W* are nullable, and the FOLLOW sets for both *S* and *W* contain `b`. Combined with the other information in the productions, `b` is in FIRST[*W*] as well as in FIRST[*S*]. That means `b` could be the first terminal symbol generated by every right-hand side of every production.

Also, `f` is in FIRST[*S*], since *W* is nullable, which also means it is in FIRST[*W*] because of the *W* ::= *S*`b` production. It is also in FOLLOW[*W*]. So `f` could be the first terminal in the right-hand side of either *W* production, and it can also begin the right-hand sides of the first and third *S* productions.

(b) S ::= `a` X `b`
X ::= `c` X | `a` X | ε

Is LL(1)

(c) S ::= `a` Y | Y `a`
Y ::= `b` | `c` | `a`

Not LL(1) because `a` appears in the FIRST sets of the right-hand sides of both *S* productions. Terminal `a` is the first symbol in FIRST[`a`*Y*], and it also is in FIRST[*Y*], so it is in FIRST[*Y*`a`].

(d) S ::= Z `b` | `c`
Z ::= `d` | S `a`

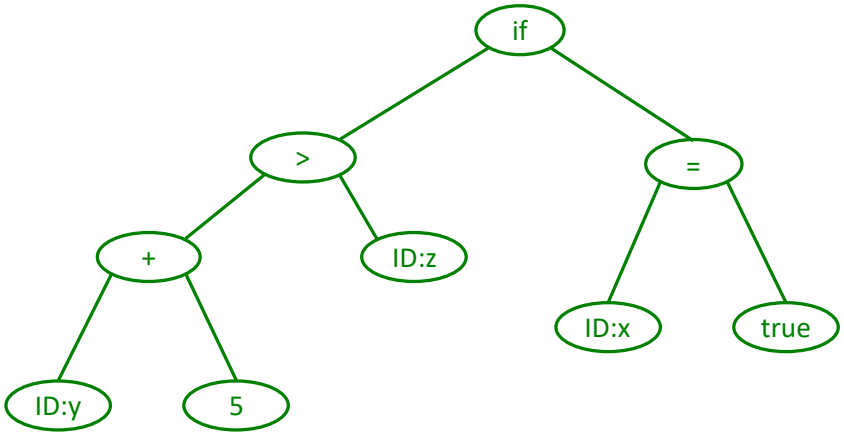
Not LL(1). FIRST[*Z*] contains `d` because of the first Z production, and it is in FIRST[*S*] because of the first S production, so the FIRST sets for the right-hand sides of both *Z* productions contain `d`. Also, `c` is in FIRST[Z] because it is in FIRST[*S*], so the FIRST sets for both productions for *S* contain `c`.

Question 6. (16 points) Semantics. In Ruby, the "conditional statement with `if` at the end" is written as "*statement* `if` *condition*", but its meaning is exactly the same as if it had been written "`if` *condition* `then` *statement*". Here is a Ruby conditional statement that stores `true` in `x` if the condition `y+5>z` is true:

```
x = true if y+5 > z;
```

(a) (7 points) Draw an abstract syntax tree (AST) for this statement in the blank space at the bottom of the 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 AST classes, but don't worry about matching the exact names of classes or node types found in the MiniJava code.

(b) (9 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 any errors. You do not need to specify an attribute grammar – just show 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.



Semantic checks needed, identified by node(s):

- **All identifier (ID:) nodes: verify that the identifier is declared and in scope.**
- **+: verify both operands have type int. Type of the + node is int.**
- **>: verify that both operands have type int. Type of the > node is Boolean**
- **= (assignment): verify that the left-hand side operand (ID:x) designates an assignable location (lvalue); verify that the type of the expression (true) is assignment-compatible with the type of ID:x, which should be Boolean here**
- **if: verify that the type of the left subtree is Boolean.**

Note: even though the condition appears after the statement in this "backwards" `if` statement, the condition should probably be the left child of the `if` node, since the same AST `if` node would be used for both forms of the `if` statement. When grading the problem there was no deduction if the tree was drawn the other way.