# Midterm practice and solution

## Exam Facts

**Tues Feb 13 1– 2:30pm Location TBA** (we'll tell you as soon as we know)
(note the early start time, location is likely to not be usual lecture room)

There is no alternate exam, everyone is expected to take the exam at the scheduled time. Local SITN students will come to campus for the exam, since that ensures you will be able to ask questions and receive any clarifications that might come up during the exam. SITN folks outside the Bay Area will take the exam at their local site. As soon as we know where room scheduling assigns us, we will put a link on the web page to a campus map with information about parking options for those of you coming from off-campus.

## Format

The midterm will be an in-class 90-minute written exam. The exam is closed-book/closed-note, but you can bring one 81/2" x 11" sheet of paper filled on both sides with whatever facts you think will be helpful**.** Your sheet is supposed to lessen the anxiety of the closed-book exam (i.e. don't have to memorize all details, do have something to refer for the complicated topics) without requiring that the questions be as involved or difficult as an open-book setting usually warrants. Writing up your sheet is a good means of studying for the exam, too. You are encouraged to prepare your sheets in groups, share ideas about what to include, and so on.

## Material

The midterm will cover material from all lectures prior to the midterm. This primarily means the first two compilation phases: lexical and syntax analysis. Material similar to the homework will be emphasized, although you are responsible for all topics presented in lecture and in the handouts. The sorts of things that you should be prepared to demonstrate knowledge of:

- Overview of a compiler— terminology, general task breakdown
- Regular expressions— writing and understanding regular expressions, equivalence between regex/NFA,/DFA, conversion among forms (Thompson's algorithm, subset construction)
- Lexical analysis— loop and switch implementations, scanner generators, using lex
- Grammars— Chomsky's hierarchy, parse trees, derivations, ambiguity, writing simple grammars
- Top-down parsing— LL(1) grammars, grammar conditioning (removing ambiguity, left-recursion, left-factoring), first and follow set computation, top-down recursive descent implementation, table-driven predictive parsing
- Shift-reduce parsing— building LR(0) and LR(1) configurating sets, parse tables, tracing parser operation, shift/reduce and reduce/reduce conflicts, differences between LR, SLR, and LALR
- Comparisons between parsing techniques— advantages/disadvantages: grammar restrictions, parser code maintenance, future flexibility, space/time efficiency, error-handling, etc.
- Syntax-directed translation —attribute grammars, inherited and synthesized attributes, use of yacc attributes and actions, simple error-handling

The rest of this handout is the midterm I gave last quarter. The exam had a fairly high median (mid 80's) so perhaps was a bit on the short/easy side. Also fall quarter's midterm was scheduled earlier in the quarter (before pp2), so yacc was not covered, but is fair game this quarter. Solutions to all problems are given at the end of this handout, but we encourage you to not look at them until you have worked through the problems yourself.

**1) a)** Indicate true or false for the following statements. (I usually score T/F as +N for a correct answer, and –N for a wrong answer, leaving blank is zero. This is to discourage random guessing.)

_____ If a grammar has a non-terminal `X` where `First(X)` and `Follow(X)` are not disjoint, there must be a conflict in the LL(1) parse table for this grammar.

_____ Every LL(1) grammar is LALR(1).

_____ If the items `[X –> x•z, y]` and `[Y –> •z, y/x]` are found in the same LR(1) state, there will be a reduce/reduce conflict somewhere in the parse table.

_____ The nested comments supported by Decaf form a regular language.

_____ Consider a grammar with no e-productions and no single productions (i.e. no productions with a single symbol on the right-hand side). For an input of n tokens, a bottom-up parser can make a maximum of n/2 reductions.

**b)** The language APL takes an unusual approach to resolving arithmetic expression ambiguity: all operators have the same precedence, and all associate right-to-left. Thus `8*5-3–1 = 24`. Parentheses can be used for explicit grouping. Although at odds with standard mathematical usage, it does make parsing easy. Write an unambiguous grammar for APL arithmetic expressions using the four binary operators (+ - * /), integer operands, and parentheses.

**2)** In C, if you need to embed a double-quote character inside a string constant, you can escape the double-quote by preceding it with a backslash. Decaf did not allow this, but we would like to add this feature now.

**a)** Build a DFA that recognizes valid Decaf string constants with escaped quote characters. As before, string constants do not allow newlines or unescaped double-quotes, but any other character is allowed within the enclosing pair of quotes (including other characters preceded by backslash which are just treated as two normal characters). This is a **DFA**. All transitions not explicitly shown in your automata are assumed to lead to the sinkhole.

**b)** Write a regular expression that recognizes these string constants. You may use any of the regular expression constructs supported by lex/flex. Because double-quote and backslash have special meaning and require escape sequences, feel free to use `q` for double-quote and `b` for backslash to avoid confusion

**3)** Re-write the following grammar as necessary to make it LL(1).

`S –> <S> | <>S | <S>S | <>`

**4)** Compute the first and follow sets for the non-terminals in the following grammar:

S –> ABf
A –> Ce | BbS
B –> dAg | ∂
C –> c | aA

| | First | Follow |
|---|---|---|
| S | | |
| A | | |
| B | | |
| C | | |

**5)** Given the following grammar with its first and follow sets, fill in the LL(1) parse table. If there are conflicts, just write both entries into the table.

S –> Xy | YZ
X –> (X) | x
Y –> yYz | ∂
Z –> zXz | Y

| | First | Follow |
|---|---|---|
| S | { x y z ( ∂ } | { $ } |
| X | { x ( } | { y z ) } |
| Y | { y ∂ } | { y z $ } |
| Z | { y z ∂ } | {$ } |

| | x | y | z | ( | ) | $ |
|---|---|---|---|---|---|---|
| S | | | | | | |
| X | | | | | | |
| Y | | | | | | |
| Z | | | | | | |

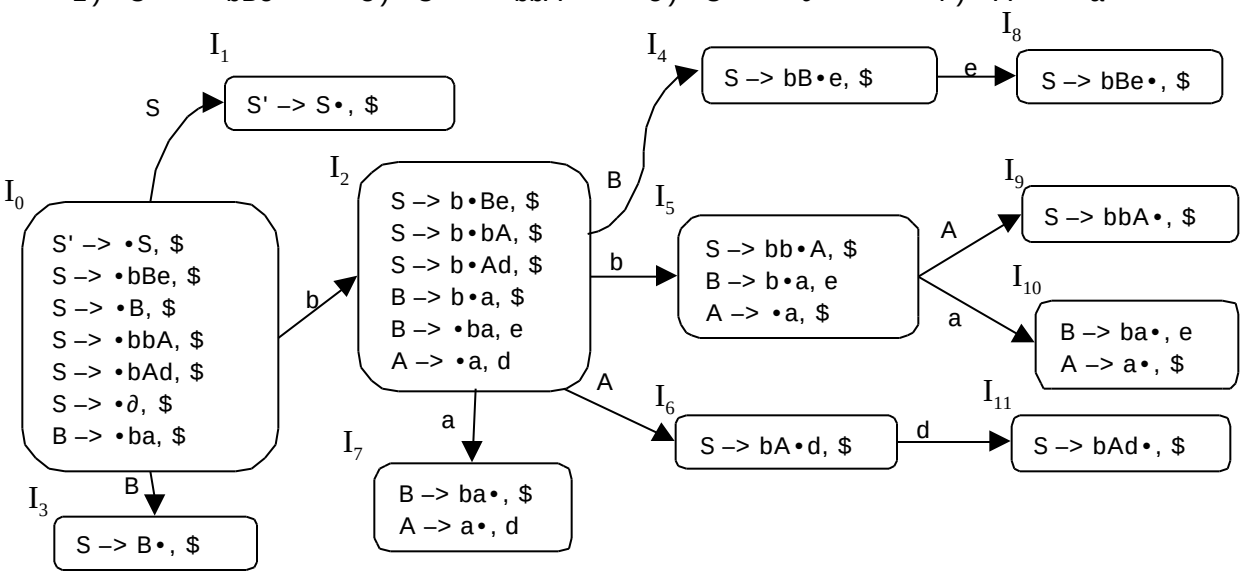
**6)** Given the following grammar:

S' –> S
S –> aAB
A –> Ab | ∂
B –> c | cAd

Draw a goto-graph of exactly those states of an LR(1) parser that are pushed on the parse stack during a parse of the input abc (this will be a subset of all configurating sets). If you accidentally construct irrelevant states, cross them out. Do not add or change any productions in the grammar.

**7)** You are given the following numbered productions and complete LR(1) goto-graph:

0) S' –> S  1) S –> bBe  2) S –> B  3) S –> bbA  4) S –> bAd  5) S –> ∂  6) B –> ba  7) A –> a




**a)** Fill in the first 8 rows of the LR(1) parse table:

| State | Action | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|
| | a | b | d | e | $ | S | A | B |
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |

**b)** Circle the smallest class of grammars this falls into: SLR(1) LALR(1) LR(1)

**c)** Are the LR(1) and LALR(1) parse tables the same? Briefly explain.

**d)** Are the LALR(1) and SLR(1) parse tables the same? Briefly explain.

## Solutions

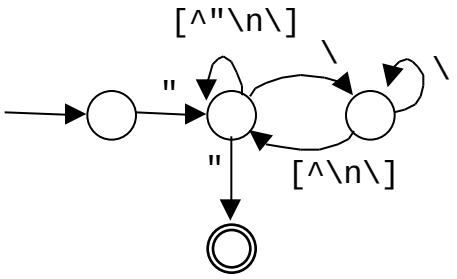
**1) a)**

F F T F F

**b)**

```
E   –>  T op E | T
op  –>  + | - | * | /
T   –>  int | (E)
```

**2) a)** The loop on backslash on the far right state was optional (i.e. we accepted with or without as full credit, they would differ on how they treated the sequence \\" inside a string)




**b)** `q([^qb\n]|b+.)*q` or using escapes `\"([^"\n\\]|\\+.)*\"`

**3)**

```
S  –>  <A
A  –>  >B | S>B
B  –>  S | ∂
```

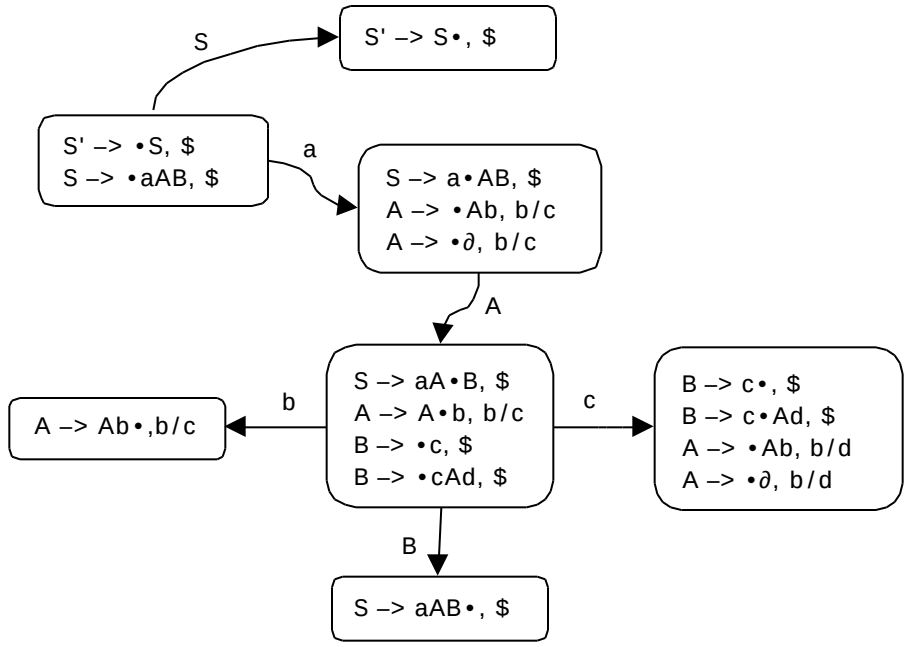
**4)**

| | First | Follow |
|---|---|---|
| S | { a b c d } | { d e f g $ } |
| A | { a b c d } | { d e f g } |
| B | { d ∂ } | { b f } |
| C | { a c } | { e } |

**5)**

| | x | y | z | ( | ) | $ |
|---|---|---|---|---|---|---|
| S | S –> Xy | S –> YZ | S –> YZ | S –> Xy | | S–> YZ |
| X | X –> x | | | X –> (X) | | |
| Y | | Y –> yYz,<br>Y –> ∂ | Y –> ∂ | | | Y –> ∂ |
| Z | | Z –> Y | Z –> zXz | | | Z –> Y |

**6)**




**7) a)**

| State | Action | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|
| | a | b | d | e | $ | S | A | B |
| 0 | | s2 | | | r5 | 1 | | 3 |
| 1 | | | | | accept | | | |
| 2 | s7 | s5 | | | | | 6 | 4 |
| 3 | | | | | r2 | | | |
| 4 | | | | s8 | | | | |
| 5 | s10 | | | | | | 9 | |
| 6 | | | s11 | | | | | |
| 7 | | | r7 | | r6 | | | |

**b)** LR(1).

**c)** No, the LALR(1) parses merges states 7 and 10 and introduce a reduce/reduce conflict on input $. It will have one fewer row and a conflict in that row.

**d)** Yes, the merged lookaheads end up recreating the full follow set for every non-terminal.