First Midterm Exam
CS164, Fall 2007
Oct 2, 2007

- Please read all instructions (including these) carefully.
- Write your name, login, and SID.
- No electronic devices are allowed, including cell phones used as watches.
- Silence your cell phones and place them in your bag.
- The exam is closed book, but you may refer to one (1) page of handwritten notes.
- Solutions will be graded on correctness and ***clarity***. Each problem has a relatively simple and straightforward solution. Partial solutions will be graded for partial credit.
- There are 8 pages in this exam and 4 questions, each with multiple parts. If you get stuck on a question move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. **Do not** use any additional scratch paper.

LOGIN: _____

NAME: _____

SID: _____

Problem	Max points	Points
1	20	
2	30	
3	25	
4	25	
TOTAL	**100**	

Problem 1: Miscellaneous [20 points]

1) **[4 points]** Circle pairs of regular expressions that are equivalent (in that they describe the same sets of strings):

a.	(ab)+	(ab)*ab
b.	ab*	(ab)*
c.	(a|b+)	(a|(b)+)
d.	a+*	a+

2) **[4 points]** Tokenize the following fragments of Java programs. Each fragment contains an error but tokenization is still possible. Indicate tokenization by drawing '|' characters between lexemes.

a. `|int| |j| |=| |a| |++| |b|;|`

b. `|int| |j| |=| |a|++|++|+|b|;|`

c. `|int| |$|foo| |(| |int| |a| |)| |{| |return| |1|;| |}|`

3) **[4 points]** CYK parser accepts arbitrary context-free grammars. This is because the CYK parser implicitly disambiguates these grammars.

True or ***False***

4) **[4 points]** A language is a set of strings. REGEX is the set of all languages that can be described with regular expressions and CFG is the set of all languages that can be described by context free grammars. Which relationship holds? Circle all applicable smileys. **Answer:** ☺A ☺B ☺C ☺D ☺E ☺F

A REGEX is a strict subset of CFG
B REGEX is a subset of CFG
C REGEX is equal to CFG
D REGEX is a superset of CFG
E REGEX is a strict superset of CFG
F None of the above

5) **[4 points]** There are languages that can be represented by NFAs but not by DFAs.

True or ***False***

Problem 2: Earley Parser [30 points]

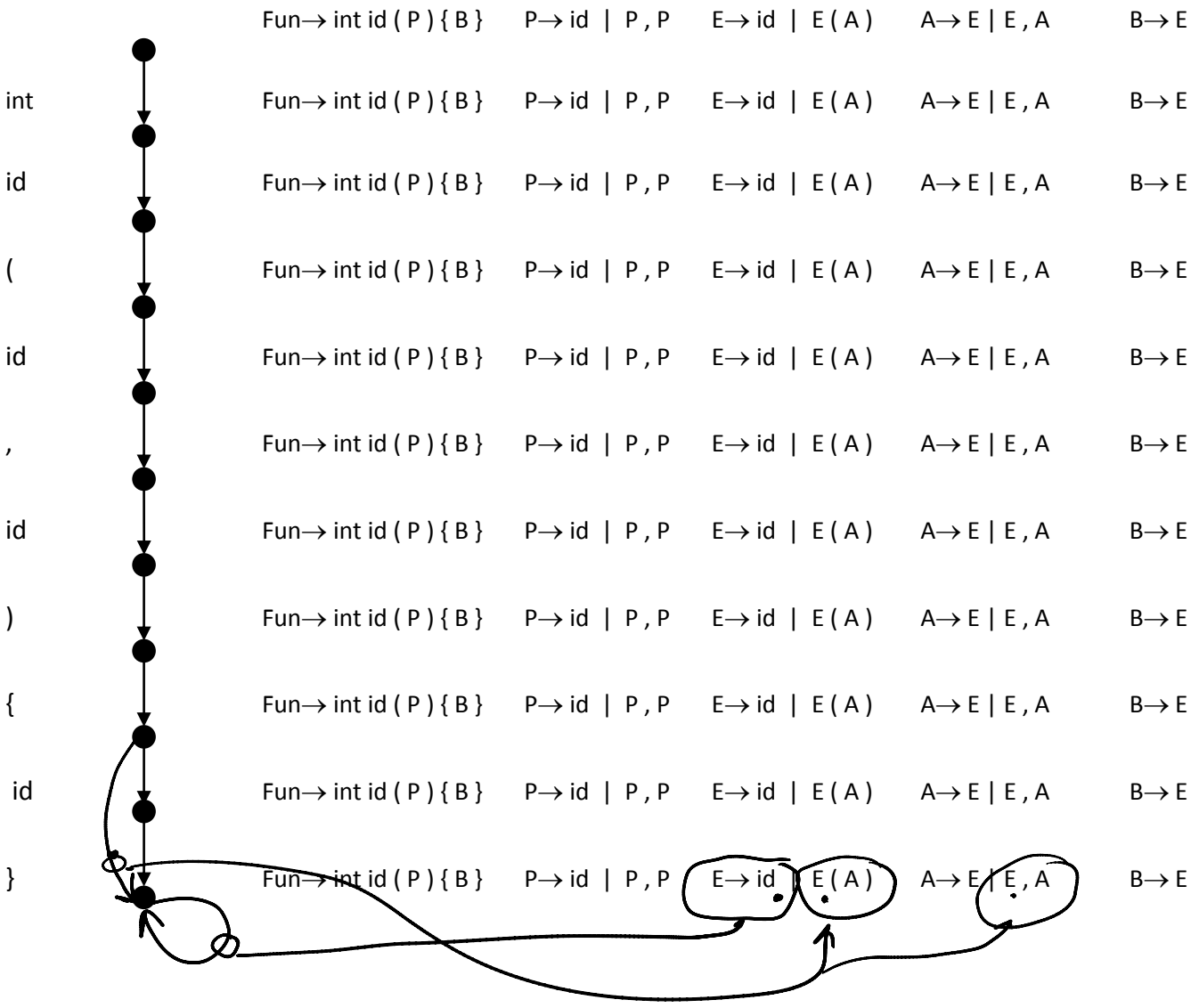
This is a grammar for a simple programming language with a single function declaration (F). The body (B) of the function contains an expression (E) that is either a variable or a function call. P is a list of formal parameters and A is a list actual arguments.

F → int id (P) { B }
P → id | P , P
E → id | E (A)
A → E | E , A
B → E

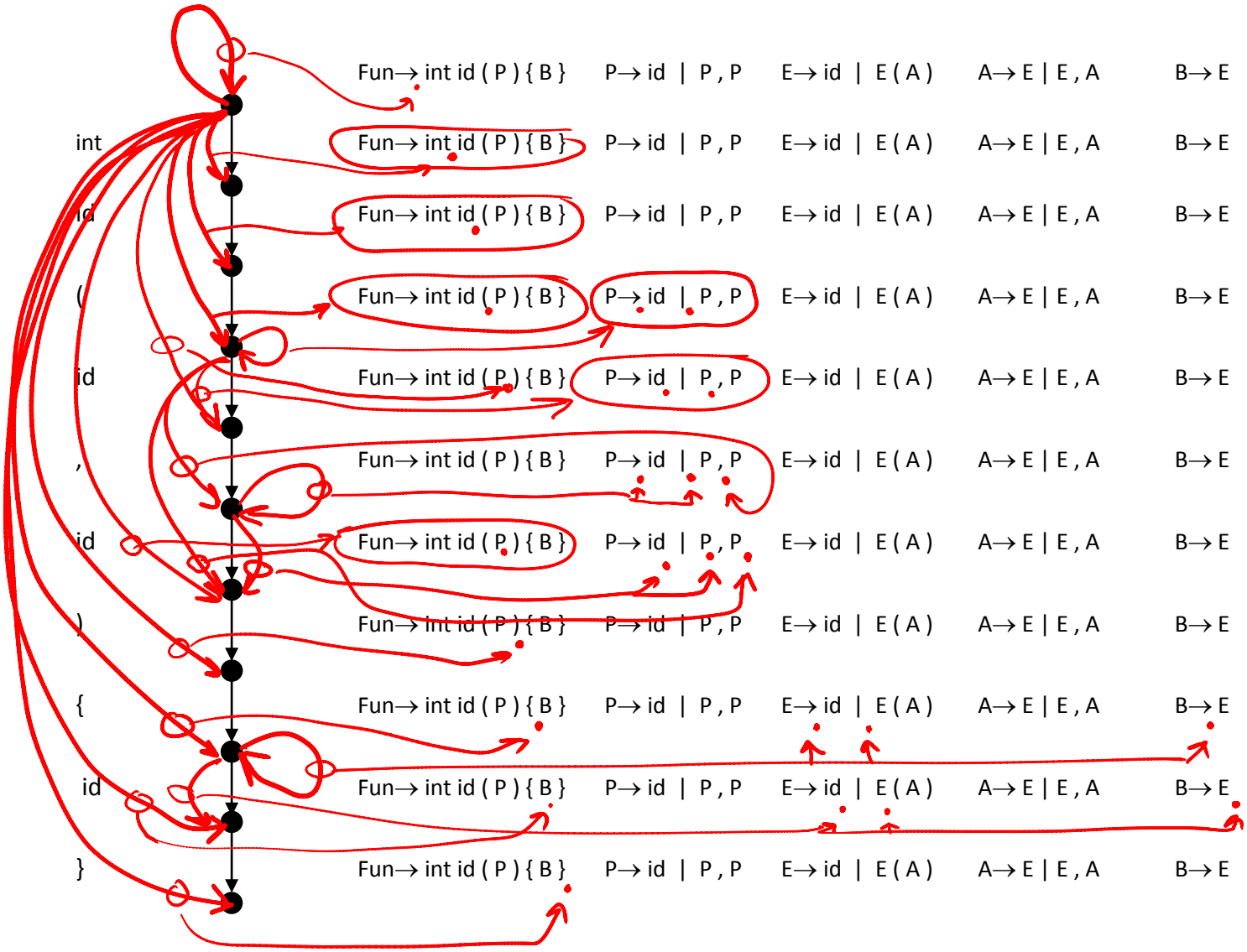
This is a string from the language described by our grammar:

int id (id , id) { id }

This is a **scratch area**. Solution goes on the next page. See below how to draw edges.



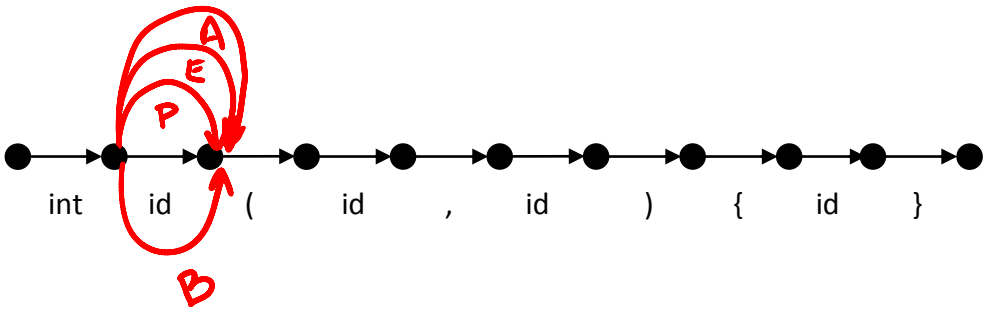
Part 1. [15 points] Show the edges added for this string by the Earley parser:



Part 2. [5 points] Is the grammar ambiguous? Circle one. **YES** **NO** *due to the PδP|P rule*

Part 3. [5 points] How many parse trees did the Earley parser discover? __ 1_____

Part 4. [5 points] Draw below three (3) edges that would be placed by the CYK parser but were **not** placed by the Earley parser. Label the edges in CYK style.



Problem 3: Representation Conversions [25 points]

Part 1. [8 points] Convert the following grammar to a regular expression, or concisely justify why this is not possible. **Note:** we do not care about the parse tree; the regular expression must represent the same set of strings as the grammar. **Hint:** write a few strings from the language and find a regular pattern, if possible.

ARITHMETIC CFG:

E → A | M | 0 | 1
A → E + E
M → E * E

YOUR REGEX:

('0' | '1') (('+' | '*') ('0' | '1'))*

Justification:

Part 2. [8 points] Same problem as above but for a different grammar: **Hint:** write a few strings from the language and find a regular pattern, if possible.

Reverse Polish Notation CFG:

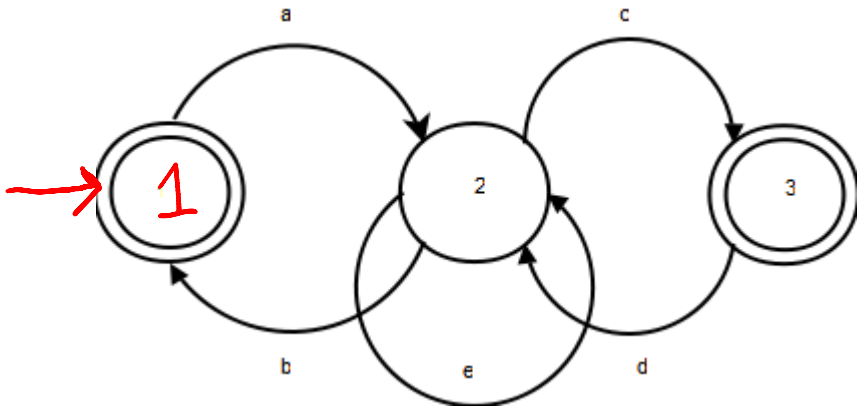
E → E E + | E E * | 0 | 1

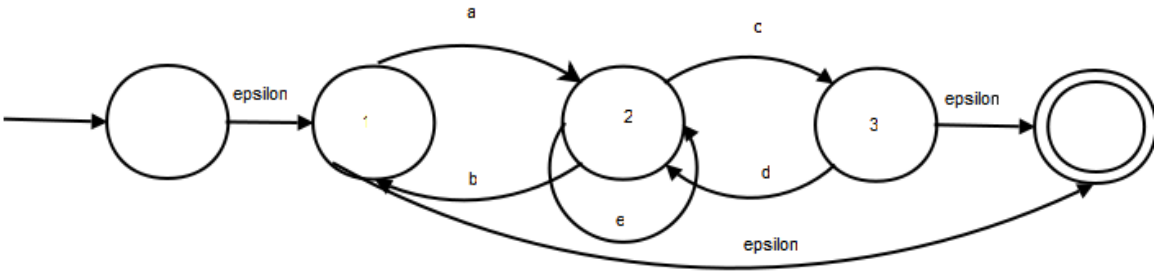
YOUR REGEX:

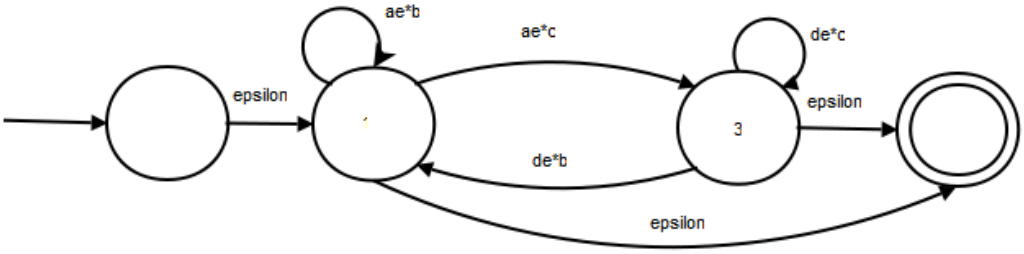
Justification: Due to a mistake on our part, we made this problem harder than originally intended. As a result, all of you got full credit. Some of you received an extra credit for making good observations about the wrong grammar.

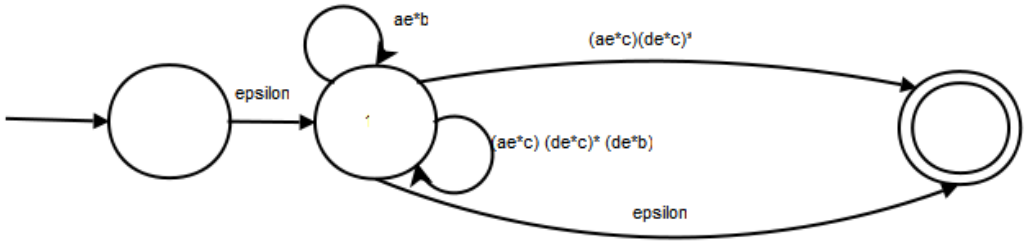
As some of you noticed, the grammar in the problem was not a correct RPM grammar. The correct RPN grammar is given above. The (correct) RPN grammar cannot be expressed as a regular language. Informally, it is because the grammar generates strings of the form 00+, 000++, 00+0+, etc, where the '+'s, viewed as binary operators, must always have zeros on the left; these zeros could be results of the evaluations. So there is a counting argument: one needs to keep track of the generated zeros in order to generate the right number of '+'s. Similar argument applied to the more complex grammar in the problem.

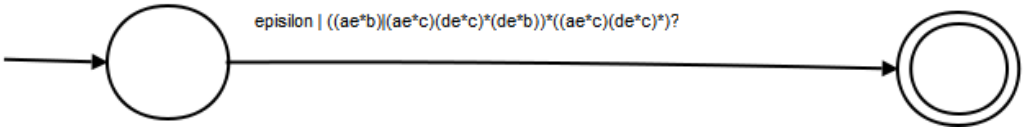
Part 3. [9 points] Convert the following automaton into a regular expression. Show each step: first eliminate node 2, then node 3.











Problem 4: Grammars and Syntax Directed Translation [25 points]

In this question we will design and implement (a tiny subset of) a language that will simplify development of HTML documents. Wikis already come with such a formatting language but we want something closer to a professional language like LaTeX.

We focus on a single aspect of the formatting language: adding *emphasis* to text by using an *italics* font. The tricky part is *nested* emphasis: *we want to emphasize text that is* already within *emphasized text.* In the previous sentence, the text "already within" is an emphasis nested within a bigger enclosing emphasis.

In HTML, the example sentence would need to be written as follows.

```
The only tricky part is <em>nested</em> emphasis: <em>we
want to emphasize text that is</em> already within <em>an
emphasized text</em>.
```

Note that HTML does not support nested emphasis. To support it, we had to turn off italics before the nested emphasis (before " already within"). In our language we want to make things clean and readable; we'll indicate beginning and end of emphasis fragments:

```
The only tricky part is \emph{nested} emphasis: \emph{we
want to emphasize text that is \emph{already within} an
emphasized text}.
```

Part 1 [7 points]: Write regular expressions for the lexemes in the language. You want to tokenize the input as indicated below with **|**.

```
|The only tricky part is |\emph{|nested|}| emphasis:
|\emph{|we want to emphasize text that is |\emph{|already
within|}| an emphasized text|}|.|
```

Backslashes may appear in the *text* lexeme when followed by by '}' or '\' characters. Backslashes may also be followed by "emph{"; this forms the *open* token. Anything else is a lexical error.

Syntax of regular expressions: you can use Python, including raw strings, but you can also use JavaScript regexes. Please underline your choice.

Lexeme	regular expression	Token
\emph{	**/\\emph{/**	**open**
}	**/}/**	**close**
Text	**/(\\[}\\]|[^\\}]|\n)*/**	**Text**

Part 2 [8 points]: Write a context-free grammar for parsing text with \emph directives. Make the grammar free of left recursion, as we'll use it to build a recursive descent parser. **Hint:** it is possible to write the grammar with three productions.

S δ *text* S | *open* S *close* S | □

A common mistake was that the grammar did not allow strings of the form
open text close ... open text close

Part 3 [10 points]: Write a recursive descent parser that translates text with \emph directives into HTML. When nested emphasis is deeper than two levels, emphasis fragments must alternate between italicized and non-italicized fonts.

You can use any programming language. If you use Python, for convenience, you can assume that it supports the ?: conditional operator and that assignments are expressions. Assume the availability of functions bool token(regex), int checkpoint(), and restore(int).

Answer: Here is one of many possible solutions. The tokens in italics are regular expressions defined in Part 1.

```
def S(em):
      var s
      if (s=token(text)):
            print s
            S(em)
      elif token(open):
            print em ? "<em>" : "</em>"
            S(not em)
            if token(close):
                  print em ? "</em>" : "<em>"
                  S(em)
            else:
                  report error and exit
      elif token (EOF):
            return # successfully parsed
      else:
            report error and exit
```

Start the translator by invoking `S(true).`