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 9 pages in this exam and 5 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	17	
2	24	
3	20	
4	15	
5	24	
TOTAL	**100**	

Problem 1: Miscellaneous [XYZ points]

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

   True or ***False***

4) **[XYZ 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) There are grammars that can be represented by NFAs but not by DFAs.

   True or ***False***

Problem 2: Earley Parser [XYZ 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	$\rightarrow$	int id (P) { B }
P	$\rightarrow$	id | P , P
E	$\rightarrow$	id | E (A)
A	$\rightarrow$	E | E , A
B	$\rightarrow$	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.

int	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
int	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
id	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
(	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
id	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
,	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
id	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
)	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
{	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
id	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E
}	Fun$\rightarrow$ int id (P) { B }	P$\rightarrow$ id | P , P	E$\rightarrow$ id | E (A)	A$\rightarrow$ E | E , A	B$\rightarrow$ E

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

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

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

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

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

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

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

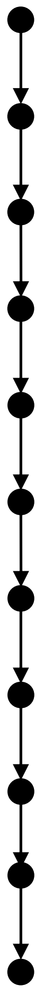
id Fun→ int id (P) { B } P→ id | P , P E→ id | E (A) A→ E | E , A B→ E

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

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

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

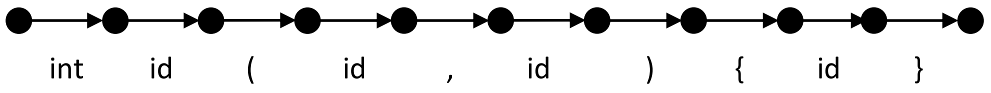
} Fun→ int id (P) { B } P→ id | P , P E→ id | E (A) A→ E | E , A B→ E



Part 2. [XYZ points] Is the grammar ambiguous? Circle one. **YES NO**

Part 3. [XYZ points] How many parse trees did the Earley parser discover? _______

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



Problem 3: Left-recursion Elimination [XYZ points]

This is a simplified grammar for regular expressions (we left out concatenation).

```
R -> R '|' R
   | R '*'
   | '(' R ')'
   | 0
   | 1
```

Part 1. [XYZ points] Write down the formula for eliminating left recursion. The formula should contain α and β:

Part 2. [XYZ points] What is α and β in the above grammar?

$\alpha =$

$\beta =$

Part 3. [XYZ points] Write a non-recursive CFG that recognizes the same language as the above grammar.

Problem 4: Representation Conversions [XYZ points]

Part 1. [XYZ 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.

ARITHMETIC CFG:

$E \to A \mid M \mid 0 \mid 1$
$A \to E + E$
$M \to M * M$

YOUR REGEX:

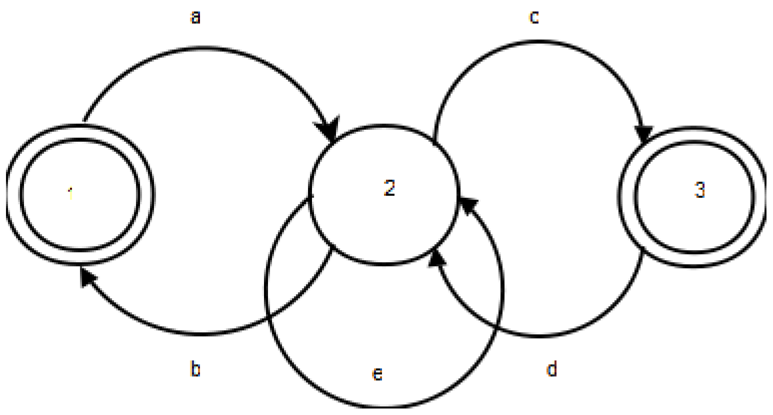
Part 2. [XYZ points] Same problem as above but for a different grammar:

Reverse Polish Notation CFG:

$E \to A E \mid M E \mid 0 \mid 1$
$A \to E E +$
$M \to E E *$

SOLUTION REGEX:

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



Problem 5: Grammars and Syntax Directed Translation [XYZ 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 [XYZ 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|}|.|
```

If the document contains a backslash followed by anything other than “emph”, report a lexical error. The *text* lexeme may contain escaped ‘}’ and ‘\’ characters. That is, it can contain pairs of characters ‘\}’ and ‘\\’.

Lexeme	**regular expression**	**token**
\emph{		**open**
}		**close**
text		**Text**
error		**-- *report error***

Part 2 [XYZ 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.

Part 3 [XYZ 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).