CS 4610 — Midterm 1

- **Write your name and UVa ID on the exam.** Pledge the exam before turning it in.
- There are 9 pages in this exam (including this one) and 6 questions, each with multiple parts. Some questions span multiple pages. All questions have some easy parts and some hard 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.
- The exam is closed book, but you may refer to your two pages of notes.
- Even vaguely looking at a cellphone or similar device (e.g., tablet computer) during this exam **is cheating**.
- 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. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. We might deduct points if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.
  - *Good Writing Example:* Python and Ruby have implemented some Smalltalk-inspired ideas with a more C-like syntax.
  - *Bad Writing Example:* Im in ur class, @cing ur t3stz!1!
- If you leave a non-extra-credit portion of the exam blank, **you will receive one-third of the points for that portion (rounded down) for not wasting our time.** If you randomly guess and throw likely words at us, we will be much less sanguine.

UVa ID: KEY

NAME (print): ANSWER KEY

UVa ID: (yes, again!) KEY

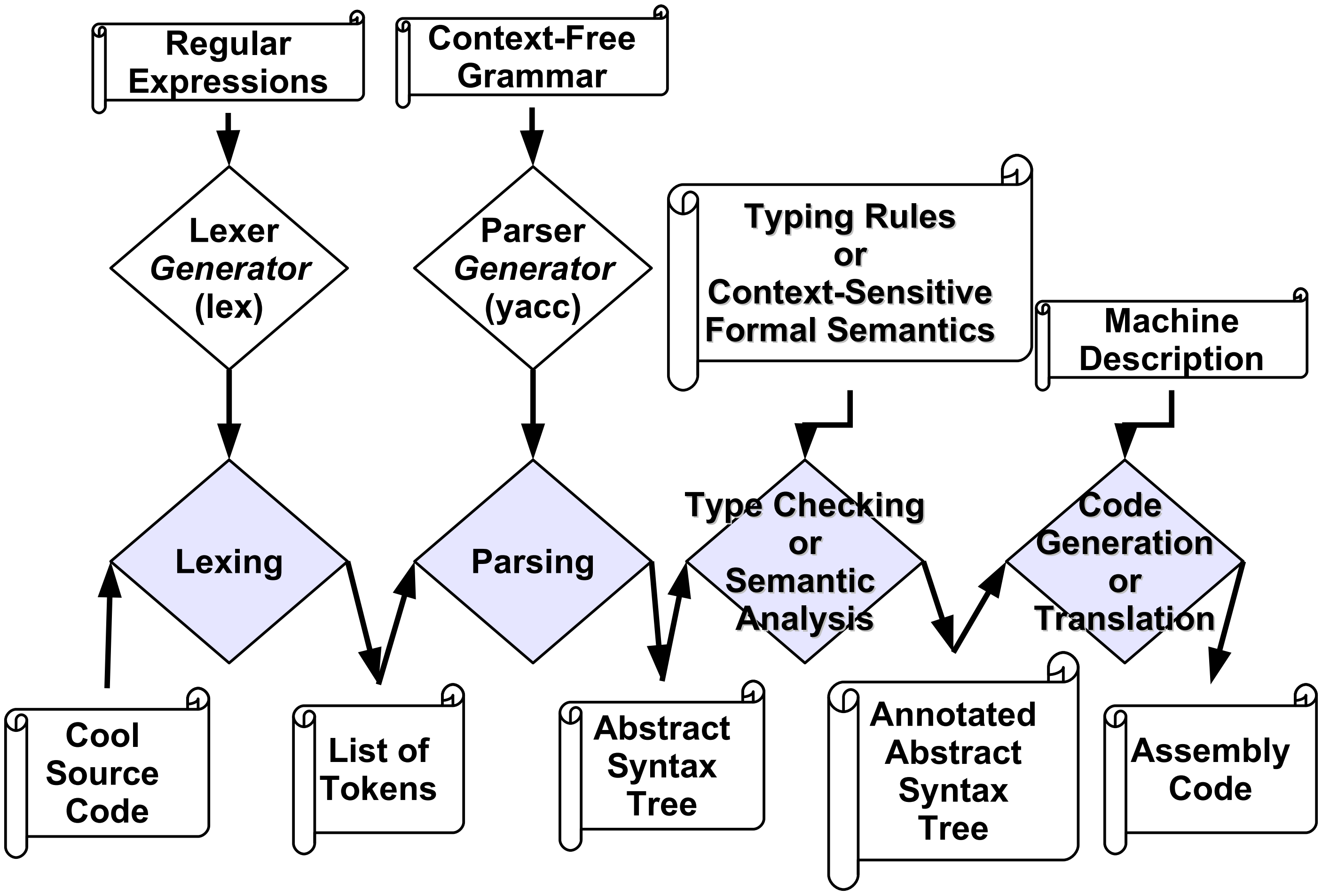
Problem	Max points	Points
1 — Compiler Stages	14	
2 — OCaml Functional Programming	13	
3 — Python Functional Programming	8	
4 — Regular Expressions	19	
5 — Ambiguity	14	
6 — Earley Parsing	32	
Extra Credit	0	
TOTAL	100	

Honor Pledge:

How do you think you did? ______________________________

1 Compiler Stages (14 points)

The following diagram shows the stages of a compiler. Label each of the eleven unlabeled diagram elements. Each unlabeled element is either a *generating tool* used in compiler construction, a *representation* of the subject program, a *stage* of the compiler, or a *formalism* used to guide or generate a stage of the compiler. Compiler stages are worth two points each, all other blanks are worth one point each.


Regular Expressions
Context-Free Grammar
Lexer *Generator* (lex)
Parser *Generator* (yacc)
Typing Rules or Context-Sensitive Formal Semantics
Machine Description
Lexing
Parsing
Type Checking or Semantic Analysis
Code Generation or Translation
Cool Source Code
List of Tokens
Abstract Syntax Tree
Annotated Abstract Syntax Tree
Assembly Code

2 OCaml Functional Programming (13 points)

Consider the following OCaml functions. The functions are correct and behave as specified; these are all direct copies of standard library functions.

```
let is_odd n = (n mod 2) = 1 (* returns true if the argument n is odd *)
(* map f [a1; ...; an] applies function f to a1, ..., an, and builds
   the list [f a1; ...; f an] with the results returned by f. *)
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> (f hd) :: (map f tl)
(* filter p l returns all the elements of the list l
   that satisfy the predicate p. *)
let rec filter p l = match l with
  | [] -> []
  | hd :: tl -> if p hd then hd :: (filter p tl) else (filter p tl)
(* fold_left f a [b1; ...; bn]] is f (... (f (f a b1) b2) ...) bn. *)
let rec fold_left f accu lst = match lst with
  | [] -> accu
  | hd :: tl -> fold_left f (f accu hd) tl
let mul x y = x * y (* Multiplication! *)
let add x y = x + y (* Addition! *)
```

Complete each of the following functions by filling in each blank with a *single* identifier, keyword or operator. You must write well-typed functional programs.

```
(* sum_list takes an integer list y as an argument and returns
   the arithmetic sum of all of the elements of y *)
let sum_list y = fold_left    ADD         0          Y
                           ---------- ---------- ----------
(* inner_sums [ [1;2;3] ; [] ; [7;8] ] returns [6; 0; 15].  *)
let inner_sums z =    MAP     sum_list     Z
                   ----------          ----------

(* prod_odds b takes a list of integers b as input and returns the
 * product of all odd integers present in b *)
let prod_odds b = FOLD_LEFT (fun x y -> MUL   X   Y  ) 1
                  ---------            ----- --- ---
  ( FILTER   IS_ODD  b)
   -------- --------
(* odds_last c permutes c so that the odd elements are at the end:
 * odds_last [1;2;3;4;5] = [4;2;1;3;5] *)
let odds_last c = List.fold_left (fun a e ->
  if is_odd    E     then    A     @ [   E    ]
             --------     --------   --------
                         else    E     ::    A     )   []     c
                              --------   --------   --------
```

3 Python Functional Programming (8 points)

Consider a function *generate* for enumerating "strings" (sequences of terminals and non-terminals) in a grammar. Only strings involving k derivation steps are returned. Example:

```
grammar = [ ("S", ["E"]) ,                # S -> E
            ("E", ["E", "+", "E"]) ,      # E -> E + E
            ("E", ["int"]) ]              # E -> int
for i in range(4):
  print i , generate(grammar, [["S"]], i)
```

Yields this output:

```
0 [['S']]                         # one string:  S
1 [['E']]                         # one string:  E
2 [['E', '+', 'E'], ['int']]      # two strings: E+E and int
3 [['E', '+', 'E', '+', 'E'], ['int', '+', 'E'],
   ['E', '+', 'E', '+', 'E'], ['E', '+', 'int']]
```

Complete the following recursive definition for *generate* by filling in each blank with a *single* identifier, keyword or operator.

```
def generate(grammar, strings_sofar, k):
  if k <= 0:
    return  STRINGS_SOFAR
           ---------------
  else:
    def flatten(lol):
      # flatten([ [1] , [2,3] ]) == [1,2,3]
      return [  ITEM     for sublist in lol for item in  SUBLIST   ]
             ----------                                 ----------
    def expansions_at_pos(string, pos):
      # expansions_at_pos(["E", "Z"], 0) returns
      # [ ["E", "+", "E", "Z"], [ "int", "Z" ] ]
      return [ string[0:pos] + rule[1] + string[pos+1:]
               for rule in grammar
               if  RULE  [  0   ] == string[ POS  ] ]
                  ------ ------             ------
    def all_expansions(string):
      return [ expansions_at_pos(string,i)
               for i in   RANGE    (  LEN      ( STRING   )) ]
                        ---------- ---------- ----------
    strings_now =  FLATTEN  ([ FLATTEN   (all_expansions(string))
                  ----------   ----------
                               for string in strings_sofar ])
    return generate(grammar, strings_now, k-1)
```

4 Regular Expressions and Automata (19 points)

For this question, the regular expressions are single character (a), epsilon (ϵ), concatenation ($r_1 r_2$), disjunction ($r_1|r_2$), Kleene star $r*$, plus $r+$ and option $r?$.

(a) (6 pts.) Write a regular expression (over the alphabet $\Sigma = \{a, b\}$) for the language of strings that have at least three occurrences of b and have an odd number of occurrences of b (but may contain other characters). Use at most 20 symbols in your answer (`strlen(answer) <= 20`).

$a^*ba^*(a^*ba^*ba^*)^+$

(b) (6 pts.) Draw a **DFA** that accepts the language from the above problem. Use at most 4 states in your answer.

```
                       b
                   /-------\
                   v       |
    +-+ b  +-+ b  +-+ b  ++==++
--> | |--->| |--->| |--->||  ||
    +-+    +-+    +-+    ++==++
    | ^    | ^    | ^     | ^
    | | a  | | a  | | a   | | a
    \-/    \-/    \-/     \-/
```

(c) (1 pt.) SOMETIMES. Given a countably infinite language L_1, there is an NFA n such that $L(n) = L_1$.

Examples: $L_1 = \{a^n | n > 0\}$ is countably infinite and is regular and has an NFA, but $L_1 = \{a^n b^n | n > 0\}$ is also countably infinite but is not regular so has no NFA.

(d) (1 pt.) ALWAYS. Given an NFA n, there is a DFA d such that $L(d) = \{st \mid s \in L(n) \wedge t \in L(n)\}$.

Put two copies of the NFA next to each other, add epsilon transitions from the accept state in the first one to the start state in the second one, and turn the accept state in the first one into a normal state.

(e) (1 pt.) SOMETIMES. Given an NFA n, there is a DFA d such that $L(d) = \{sss \mid s \in L(n)\}$.

Example: If $L(n) = \{a\}$ then $L(d) = \{aaa\}$ is regular and has a DFA. However, if n corresponds to the regular expression ab^* then $L(d) = \{ab^n ab^n ab^n | n \geq 0\}$ which is not regular (compare $a^n b^n$) and thus has no DFA.

The difference between this question and the previous one is that in the previous one, s and t could be *differnet* strings in $L(n)$, while in this one, it has to be the same s

three times in a row. That requires more memory than DFAs have: they can only remember the state they are currently in, not the path they took to get there.

(f) (1 pt.) SOMETIMES. Given a context-free grammar g, there exists an NFA n such that $L(g) = L(n)$.

Examples: If g just has $S \rightarrow \texttt{x}$ then that finite language is also regular and has an NFA. However, if g is $S \rightarrow (S) \mid \epsilon$ (i.e., if g encodes $(^n)^n$) then that language is not regular so it has no NFA.

(g) (1 pt.) NEVER. Given a language L_2 that is context-free but not regular, there exists an NFA n such that $L(n) = L_2$.

By definition L_2 is not regular, so no NFA or DFA can capture it!

(h) (1 pt.) ALWAYS. Given a finite language L_3, there exists a context-free grammar g such that $L_3 = L(g)$.

Suppose $L_3 = \{x_1, x_2, \ldots x_n\}$. The grammar g is $S \rightarrow x_1 \mid x_2 \mid \ldots \mid x_n$. That is, just make one rewrite rule for every string in the language. Since the language is finite, that works.

(i) (1 pt.) ALWAYS. Given an NFA n, there exists a regular expression r containing neither + nor ? such that $L(n) = L(r)$.

Convert the NFA to a regular exprsesion. Then rewrite each r^+ to be rr^* and rewrite each $r?$ to be $(r|\epsilon)$.

5 Ambiguity (15 points)

Consider the following grammar G_1.

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & \texttt{int} \\ E & \rightarrow & E\ +\ E \\ E & \rightarrow & !\ E \end{array}$$

(a) (4 pts.) Show that this grammar is ambiguous using the string "`! int + int`".

I believe everyone got full credit here, so I'll skip drawing the two parse trees. One begins with $E \rightarrow E + E$ and the other starts with $E \rightarrow !E$.

(b) (10 pts.) Rewrite the grammar to eliminate left recursion. That is, provide a grammar G_2 such that $L(G_1) = L(G_2)$ but G_2 admits no derivation $X \longrightarrow^* X\alpha$.

Many answers were possible. Here's one "textbook-style" example:

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & int\ F \\ E & \rightarrow & !\ E\ F \\ F & \rightarrow & \epsilon \\ F & \rightarrow & +\ E\ F \end{array}$$

Here's a very concise answer:

$$S \rightarrow int\ +\ S \mid !S \mid int$$

Good testcases to use to check your answer are `int` and `!!int + !!int + !!int`.

6 Earley Parsing (32 points)

(a) (28 pts.) Complete the Earley parsing chart (parsing table) on the next page.

(b) (2 pts.) When would we want to use Earley parsing instead of LL parsing? When would we want to use LL parsing instead of Earley parsing? Do not exceed four sentences.

Earley parsing can handle all context-free grammars, and should be used when the input grammar is not LL(k). For example, if the input grammar has left recursion, Earley is a good choice. By contrast, LL(1) can be more efficient (Earley is $\mathcal{O}(n^3)$ in the worst case), so if parsing speed and/or parsing size are key concerns, LL(1) might be used.

(c) (1 point if not blank.) What's your favorite thing about this class? (If you're in Compilers, answer for Compilers as well.)

(d) (1 point if not blank.) What's your least favorite thing about this class? (If you're in Compilers, answer for Compilers as well.)

(e) **Extra Credit (at most 2 points).** Name one video image Byron Boots used as an example in his seminar.

(f) **Extra Credit (at most 2 points).** Cultural literacy. Below are the English titles of ten important works of world literature. Each work is associated with one of the ten most common languages (by current number of native-language speakers; Ethnologue estimate). For each work, give the associated language. Be specific.

i. RUSSIAN. Crime and Punishment.
ii. GERMAN. Faust.
iii. CHINESE (or Mandarin, etc.). Journey to the West.
iv. ARABIC. Palace Walk.
v. ENGLISH. Sense and Sensibility.
vi. SPANISH. Don Quixote.
vii. PORTUGUESE. The Lusiads.
viii. HINDI. The Ocean of the Deeds of Rama.
ix. JAPANESE. Heiki Monogatari.
x. BENGALI. Where The Mind is Without Fear.

Grammar

S → id A M
A → = E | ε
M → and id A | ε
E → id | int

Input

id = int and id = id

id = int and id = id

chart [0]	chart [1]	chart [2]	chart [3]	chart [4]	chart [5]	chart [6]	chart [7]
S→•id A M ,0	S → id • A M ,0	A → = • E ,1	E → int • ,2	M → and • id A ,3	M → and id • A ,3	A → = • E ,5	E → id • ,6
	A → • = E ,1	E → • int ,2	A → E •E ,1		A → • = E ,5	E → • int ,6	A → = E • ,5
	A → • ,1	E → • id ,2	S → id A • M ,0		A → • ,5	E → • id ,6	M → and id A • ,3
	S → id A • M ,0		M → • and id A ,3		M → and id A • ,3		S → id A M • ,0
	M → • and id A ,1		M → • ,3		S → id A M • ,0		
	M → • ,1		S → id A M • ,0				
	S → id A M • ,0						