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 page-sides 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

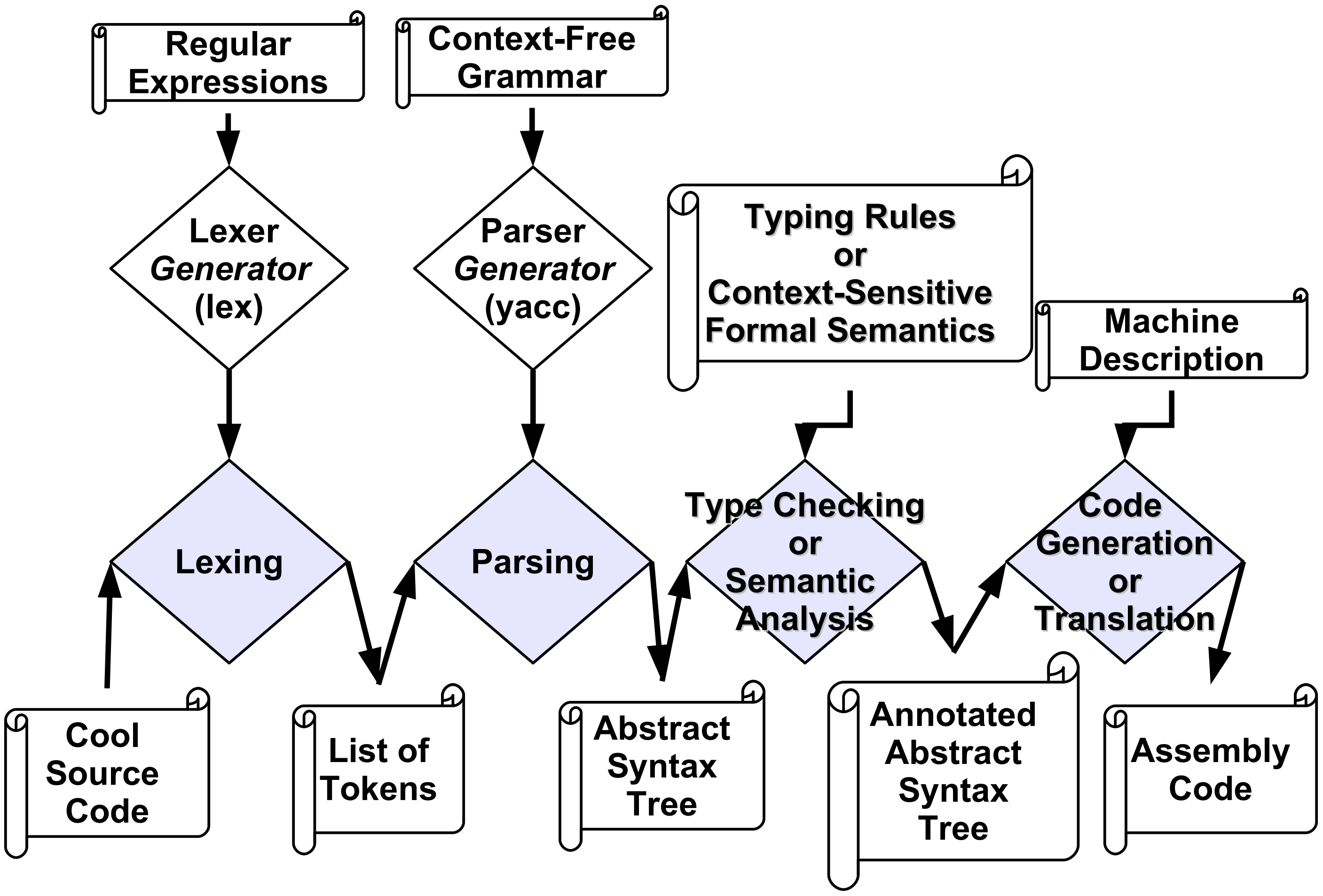
Problem	Max points	Points
1 — Compiler Stages	14	
2 — OCaml Functional Programming	13	
3 — Python Functional Programming	8	
4 — Regular Expressions	19	
5 — Ambiguity	14	
6 — Earley Parsing	32	
Extra Credit	0	
TOTAL	100	

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_even n = (n mod 2) = 0 (* returns true if the argument n is even *)
(* 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_lens takes an string list y as an argument and returns
   the arithmetic sum of all of the lengths of elements of y *)
let sum_lens y = fold_left  ADD        0 ( MAP         LENGTH    y)
                           ----------    ---------- ----------

(* inner_sums [ ["ant";"bat"] ; [] ; ["oh";"no"] ] returns [6; 0; 4]. *)
let inner_sums z =  MAP        sum_lens  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     (fun x y -> MUL      X         Y       ) 1
                  ---------             --------- --------- ---------
  ( FILTER  (fun a ->  NOT        IS_EVEN   a) b)
   ---------          ---------  ----------
(* odds_last c permutes c so that the odd elements are at the front:
 * odds_last [1;2;3;4;5] = [4;2;1;3;5] *)
let odds_last c = List.fold_left (fun a e ->
  if is_even  E       then  E        ::  A
             -------       ---------    ---------
                    else  A        @ [ E      ])  []       c
                          ---------    ---------   ---------
```

3 Python Functional Programming (8 points)

Consider a function *nfa_accepts* for determining if a string is in the language of an NFA. For simplicity we do not consider epsilon transitions. For example, consider an NFA accepting the regular language denoted by the regular expression $c \mid a(aa) * b$ below:

```
edges = [ ("q0", "a", "q1") ,  # in state q0, on a, goto q1
          ("q0", "c", "q2") ,  # in state q0, on c, goto q2
          ("q1", "b", "q2") ,  # in state q1, on b, goto q2
          ("q1", "a", "q0") ]  # in state q1, on a, goto q0
final = [ "q2" ]
start = "q0"

for s in [ "a" , "ab", "c", "aab", "aaab", ]:
  print s, ":", nfa_accepts(start, edges, final, s)

print [x*x for x in range(10) if x > 5] # list comprehension hint
```

Yields this output:

```
a : False
ab : True
c : True
aab : False
aaab : True

[36, 49, 64, 81]
```

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

```
def nfa_accepts(state, edges, final, string):
  if len( STRING    ) == 0:
         ----------
    return  STATE      in  FINAL
           ----------     ----------
  else:
    destinations = [  DEST    for (start,symb,dest) in  EDGES
                     -------                           ----------
                     if  START   == state and symb ==  STRING   [0] ]
                        --------                       ----------
    return True in [ nfa_accepts( DEST   , edges, final,  STRING [1:])
                                 -------                 --------
                     for dest in destinations ]
```

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 two occurrences of a and have an even number of occurrences of a (but may contain other characters). Use at most 20 symbols in your answer (`strlen(answer) <= 20`).

$(b*\ a\ b*\ a\ b*)+$

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

```
      b           b            b
     /-\         /-\          /-\
     | v         | v          | |
     ---         ---          | v
    /   \   a   /   \   a    /====\
--->|   |------>|   |------>||  ||
    \___/       \___/       \====/
                  ^     a       |
                  \_____________/
```

(c) (1 pt.) ALWAYS. Given a finite language L_1, there exists a context-free grammar g such that $L_1 = L(g)$.

If the language $L_1 = \{a, b, c\}$, then g is $S \to a|b|c$.

(d) (1 pt.) ALWAYS. Given an NFA n, there is a finite or countably infinite language L_2 such that $L(n) = L_2$.

Every string is finite. A language is a set of strings. An NFA can be finite (e.g., accepting $a?$) or infinite (e.g., accepting $a*$).

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

Some context-free grammars, such as $S \to a$, have corresponding NFAs (because those context free grammars actually describe regular languages).

Conversely, some context free grammars, such as $S \to aSb \mid \epsilon$ (yielding $a^n b^n$) do not have associated regular languages and thus do not have equivalent NFAs.

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

Make two copies of n. Connect the accepting state of the first copy to the start state of the second copy, and make that first accepting state non-accepting. Then convert the result to a DFA.

Note that s and t do not have to be the same string — you just have to make it through n twice.

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

Convert the NFA n to a regular expression. Then:

If you see $r*$, rewrite it as $(r+ \mid \epsilon)$.

If you see $r?$, rewrite it as $(r \mid \epsilon)$.

(h) (1 pt.) SOMETIMES. Given a countably infinite language L_3, there is an NFA n such that $L(n) = L_3$.

Some countably infinite languages, such as $\{a^n \mid n \geq 0\}$, are regular (regular expression $a*$) and thus have associated NFAs.

Some countably infinite languages, such as $\{a^n b^n \mid n \geq 0\}$, are not regular, and thus do not have any associated NFA.

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

This one was super tricky! The insight is that s has to be the same every time.

This works sometimes: consider d with $L(d) = \{a, b\}$. Then there is definitely an NFA that accepts $aaaa \mid bbbb$.

However, consider a DFA d that accepts the regular expression $(ab)*$. Then we'd need an NFA n that accepts $\{a^n b^n a^n b^n a^n b^n a^n b^n \mid n \geq 0\}$. But that's just a harder version of $a^n b^n$. So it's not regular, so there's no such NFA.

5 Ambiguity (14 points)

Consider the following grammar G_1.

$$\begin{array}{rcl} S & \rightarrow & E \\ E & \rightarrow & \texttt{true} \mid \texttt{false} \\ E & \rightarrow & E \texttt{ or } E \mid E \texttt{ and } E \\ E & \rightarrow & \texttt{not } E \end{array}$$

(a) (4 pts.) Show that this grammar is ambiguous using the string "`not false or true`".

I believe everyone got full credit here, so I'll skip drawing the trees. One has $E \rightarrow \texttt{not}E$ first and one has $E \rightarrow E\texttt{or}E$ first.

(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 many correct answers were possible. Some examples:

$$\begin{array}{rcl} S & \rightarrow & E \\ E & \rightarrow & \texttt{true or } E \\ E & \rightarrow & \texttt{false or } E \\ E & \rightarrow & \texttt{true and } E \\ E & \rightarrow & \texttt{false and } E \\ E & \rightarrow & \texttt{not } E \\ E & \rightarrow & \texttt{true} \\ E & \rightarrow & \texttt{false} \end{array}$$

Or:

$$\begin{array}{rcl} S & \rightarrow & A\ B \\ A & \rightarrow & \texttt{not } A \\ A & \rightarrow & \texttt{true} \\ A & \rightarrow & \texttt{false} \\ B & \rightarrow & \texttt{or } S \\ B & \rightarrow & \texttt{and } S \\ B & \rightarrow & \epsilon \end{array}$$

6 Earley Parsing (32 points)

(a) (27 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 supports all CFGs (including grammars with left recursion and other ambiguous grammars) and is simple to understand and verify. However, it can be slow (cubic time, worst case).

LL parsing supports only a subset of CFGs (e.g., LL(1) parsing supports only LL(1) grammars: those where you can make an LL(1) parsing table with no duplicate entries) but can be much faster in practice.

(c) **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. ____________________. The Ocean of the Deeds of Rama.

ii. ____________________. Where the Mind is Without Fear.

iii. ____________________. Journey to the West.

iv. ____________________. The Ingenious Hidalgo Don Quixote of La Mancha.

v. ____________________. The Tale of the Heike.

vi. ____________________. Faust.

vii. ____________________. Palace Walk.

viii. ____________________. The Lusiads.

ix. ____________________. Sense and Sensibility.

x. ____________________. Crime and Punishment.

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						