

Name (PRINTED): _____		
University ID #: _____		
Circle your TA's name:	Guilherme	Nir
Circle your discussion time:	12:00	1:00

CMSC 330

Exam #2

Fall 2006

**Do not open this exam until you are told. Read these instructions:**

1. This is a closed book exam. **No notes or other aids are allowed.**
2. **You must turn in your exam immediately when time is called at the end.**
3. This exam contains 7 pages, including this one. **Make sure you have all the pages.** Each question's point value is next to its number. **Write your name on the top of all pages before starting the exam.**
4. In order to be eligible for as much partial credit as possible, show all of your work for each problem, and **clearly indicate** your answers. Credit **cannot** be given for illegible answers.
5. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.
6. If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.
7. If you need scratch paper during the exam, please raise your hand. Scratch paper must be turned in with your exam, with your name and ID number written on it. Scratch paper **will not** be graded.
8. Small syntax errors will be ignored in any code you have to write on this exam, as long as the concepts are correct.
9. The Campus Senate has adopted a policy asking students to include the following handwritten statement on each examination and assignment in every course: "*I pledge on my honor that I have not given or received any unauthorized assistance on this examination.*" Therefore, **just before turning in your exam**, you are requested to write this pledge **in full** and **sign it** below:

---



---

Good luck!

1	2	3	4	Total

1. [15 pts.] **Short Answer.**

- a. [5 pts.] List the *free variables* in the following C function. In one to two sentences, define what the free variables of a scope are in general.

```
void foo(int a, int b) {  
    int c;  
  
    c = a + b;  
    d = c * 2;  
    return d;  
}
```

**Answer:** The only free variable in this function body is `d`. The free variables in a scope are those that are not bound by some definition or declaration.

- b. [10 pts.] The following OCaml function is not tail recursive. Briefly explain why not, and transform it into a tail recursive function.

```
(* returns 0 + 1 + ... + n *)  
let rec sum n =  
    if n = 0 then 0  
    else n + (sum (n-1))
```

**Answer:** The function is not tail recursive because after the recursive call to `sum`, more work happens in the function (namely, the result of the recursive call is added to `n`).

Here is a tail recursive version:

```
let sum n =  
    let rec sum_helper acc m =  
        if m = 0 then acc  
        else sum_helper (acc+m) (m-1)  
    in  
    sum_helper 0 n
```

2. [25 points] **Parameter passing.** Suppose you are developing a new, statically-scoped language that has the following fairly obvious syntax:

```
integer n;
integer i;
integer array A[5];
function f(integer array x, integer y, integer z)
  begin
    n = z-2;
    z = y-1;
    x[y] = x[z]+1;
    print(x[1], x[2], x[3], x[4], y, z);
  end;
```

```
n = 5;
for (i = 0; i < n; i++)
  A[i] = i;
end
call f(A, n, n)
print(A[1], A[2], A[3], A[4], n);
```

Fill out the following table showing what would be printed if the language used either call-by-value or call-by-reference for all parameters. Under call-by-value, assume the *entire* array is passed by value (this is different than C, which would pass a pointer to the array).

	x[1]	x[2]	x[3]	x[4]	y	z
Call-by-value	1	2	3	4	5	4
	A[1]	A[2]	A[3]	A[4]	n	
Call-by-value	1	2	3	4	3	
	x[1]	x[2]	x[3]	x[4]	y	z
Call-by-reference	1	3	3	4	2	2
	A[1]	A[2]	A[3]	A[4]	n	
Call-by-reference	1	3	3	4	2	

3. [25 pts.] **Grammars.**

Consider the following two grammars for expressions, where  $i$  stands for any integer.

$$\begin{array}{ll} E ::= E + T \mid T & E ::= E * T \mid T \\ T ::= T * P \mid P & T ::= T + P \mid P \\ P ::= (E) \mid i & P ::= (E) \mid i \end{array}$$

(G1)

(G2)

- a. [5 points] For *both* grammars G1 and G2, give the leftmost derivations for the string  $2+3*4$

**Answer:**

$$(G1) E \Rightarrow E+T \Rightarrow T+T \Rightarrow P+T \Rightarrow 2+T \Rightarrow 2+T*P \Rightarrow 2+P*P \Rightarrow 2+3*P \Rightarrow 2+3*4$$

$$(G2) E \Rightarrow E*T \Rightarrow T+P*T \Rightarrow P+P*T \Rightarrow 2+P*T \Rightarrow 2+3*T \Rightarrow 2+3*P \Rightarrow 2+3*4$$

- b. [10 points] Do both G1 and G2 accept the same set of strings? Explain why or why not.

**Answer:** They generate the same strings.

The two E productions in G1 generate the strings  $T+T+\dots+T$  and the two T productions generate the strings  $P*P*\dots*P$  so these four productions generate the strings  $P?P?P?P\dots?P$  where ? can either be + or \*.

Similarly in G2 the E productions generate  $T*T*\dots*T$  and the T productions generate  $P+P+\dots+P$ . So these four also generate the strings  $P?P?P?P\dots?P$  where ? can either be + or \*.

This is just the regular expression:  $(P(+|*)) * P$  which is the same in both grammars. The P productions are the same in both grammars, so the resulting strings generated by G1 and G2 are the same.

Name: \_\_\_\_\_

c. [10 points] Give a context-free grammar for the set  $\{a^n b^m c^n | m, n \geq 0\}$ .

**Answer:** Here is a grammar for the set:

$$\begin{aligned} S &::= a S c \mid T \\ T &::= b T \mid \varepsilon \end{aligned}$$

4. [35 pts.] **OCaml.** Consider the following OCaml type describing a grammar:

```

type word =
  Term of char           (* a terminal symbol *)
| Nonterm of char       (* a non-terminal symbol *)
type prod = char * (word list) (* a production *)
type grammar = prod list (* a grammar *)

```

We use `char` to name both terminals and non-terminals. For example, the OCaml code below defines `g1` to be the grammar G1 from problem 3:

```

let (p1 : prod) = ('E', [Nonterm 'E'; Term '+'; Nonterm 'T'])
let (p2 : prod) = ('E', [Nonterm 'T'])
let (p3 : prod) = ('T', [Nonterm 'T'; Term '*'; Nonterm 'P'])
let (p4 : prod) = ('T', [Nonterm 'P'])
let (p5 : prod) = ('P', [Term '('; Nonterm 'E'; Term ')'])
let (p6 : prod) = ('P', [Term 'i'])
let g1 = [p1; p2; p3; p4; p5; p6]

```

In the following problems, you may use any standard OCaml library functions you like.

- a. [5 pts.] Write a function `start : grammar -> char` that returns the start symbol of the grammar. As in our formal notation, we assume that the left-hand side of the first grammar production in the list is the start symbol. Your function may do anything if the grammar has no productions.

**Answer:**

```

let start ((c, _)::_) = c

```

- b. [15 pts.] Write a function `terms : grammar -> char list` that returns a list containing the terminal symbols used in strings produced by the grammar. You may list the terminals in any order, and duplicates are OK.

**Answer:**

```

let rec terms (g:grammar) =
  let add_term acc = function
    Term c -> c::acc
  | _ -> acc
  in
  match g with
  [] -> []
| (c, wl)::t -> (List.fold_left add_term [] wl) @ (terms t)

```

- c. [15 pts.] Write a function `left_reduce` : `prod -> word list -> word list` that applies the production to the left-most non-terminal matching the production's left-hand side, if any, returning the new `word list`. For example, the production is  $N \rightarrow a B c$ , and the `word list` is `a N b N`, then this function should return `a a B c b N`.

**Answer:**

```
let rec left_reduce (c, wl) = function
  [] -> []
| (Term d)::r -> (Term d)::(left_reduce (c, wl) r)
| (Nonterm d)::r ->
  if c = d then
    wl @ r
  else
    (Nonterm d)::(left_reduce (c, wl) r)
```