Name:_____ Student ID:_____

1. (5 minutes). The English sentence "Colorless green ideas sleep furiously." is valid syntactically, but is complete nonsense semantically. Write a Scheme expressio that has the same property.

2. Consider the following two predicates:

```
c_keyword(auto).
c_keyword(break).
```

```
c_keyword_x(auto) :- !.
c_keyword_x(break) :- !.
```

```
c_keyword(case).                    c_keyword_x(case) :- !.
...                                 ...
```

2a (5 minutes). Give a sample top-level call to c_keyword/1 whose behavior would change if it used c_keyword_x/1 instead, and explain why the behaviors differ.

2b (5 minutes). A disadvantage of the above approach is that we have to list all the C keywords twice. Rewrite the predicates so that the C keywords need to appear only once in the Prolog source code. Do not change the externally-visible behavior of the predicates.

3 (15 minutes). Consider the following DCG rule, taken from the solution to Homework 1:

```
access_to_object(C, C2) -->
    [access],
    general_access_modifier(C1, [*|C2]),
    nonarray_type(C, C1).
```

Translate this rule to an ordinary Prolog rule, without using DCG notation. Give an example call to the Prolog predicate that you've defined. Your call should be something that might arise during the successful evaluation of ada2c_type/2.

4. The following deliberately-obscure C program has a single runtime error, but otherwise it is a valid program.

```
int main (void)
{
  struct main { int main; } main;

  {
    {
      enum { main = 0 };
      if (main)
        return main;
    }
    main.main = 1;
    goto main;
  }

  {
  main:
    {
      struct main main = main;
      return &main.main - &main.main;
    }
  }
}
```

4a (5 minutes): How many different names are in this program? For each name, identify where it is defined, and where it is used.

4b (10 minutes): Given that this is a valid C program (except for the runtime error), describe the scope rules for the C features used in this example, explaining in general how a C compiler can disambiguate multiple names that have the same identifier.

4c (5 minutes): What is the runtime error in the above program?

5. This problem asks you to write some Scheme procedures. Your solutions can assume that the arguments to each procedure have the proper form: i.e., procedures need not check for errors.

Consider the following part of the Homework 2 solution:

```
(define (mapbt f obj1 . objs)
  (let mapit ((obj1 obj1) (objs objs))
    (cond ((pair? obj1)
           (cons (mapit (car obj1) (map car objs))
                 (mapit (cdr obj1) (map cdr objs))))
          ((null? obj1)
           '())
          (else
           (apply f obj1 objs)))))
```

5a (5 minutes). Write a simpler function map1bt that acts just like mapbt, except it accepts exactly two arguments (a function and a binary tree) instead of accepting two or more arguments.

5b (15 minutes). Write an implementation of map1bt in Ocaml. Use an idiomatic translation, not a literal one. What type does your Ocaml function have?

5c (15 minutes). Suppose we have the opposite problem: we have a binary tree of procedures, and want to apply each of them to an object, yielding a binary tree of results, one for each procedure. That is, we want to define a procedure btmap such that, for example:

```
(btmap
  (cons (cons list real?)
        (cons (cons '() integer?)
              (cons (cons sin cos) -)))
  0.5)
```

yields:

```
(((0.5) . #t)
 (() . #f)
 (0.479425538604203 . 0.8775825618903728) . -0.5)
```

btmap always takes two arguments: the first is a binary tree of procedures (i.e., containing only pairs, procedures, and the empty list, and without any cycles), such that each procedure accepts a single argument; and the second is an arbitrary object, which will be passed to each of the procedures.

Write an implementation of btmap in Scheme. You may use any of the standard Scheme builtin procedures, and you may also use the map1bt procedure as specified above. Your implementation should be as short and elegant as possible.

5d (5 minutes). Suppose you had the further problem of generalizing btmap so that it could handle ingrown binary trees of procedures. How would you go about this?

5e (10 minutes). Suppose you have two binary trees: the first one "tp" is a tree of procedures (like the argument to btmap), and the second one "to" is a tree of objects (like the argument to mapbt1) each of which is neither a

pair nor the empty list. Write a Scheme procedure (maptrees tp to) that applies each of the procedures to each of the objects, and produces a tree containing all of the results. The top part of the tree should have the same structure as the tree of procedures, and each subtree under the top part should have the same structure as the tree of objects. For example:

```
(maptrees (list - (cons '() +) sin) '(1 () (2 . 3)))
```

should return

```
((-1 () (-2 . -3))
 (() 1 () (2 . 3))
 (0.8414709848078965 ()
  (0.9092974268256817 . 0.1411200080598672)))
```

Your solution can assume the existence of solutions for the maplbt and btmap functions described above.

::::::::::::::
midlans.txt
::::::::::::::

1. (:-)

2a. c_keyword(X), X=break
This will succeed with c_keyword, but will fail with c_keyword_x because it succeeds first with X=auto, and fails when backtracked into

2b. c_keyword_x(X) :- c_keyword(X), !.

3. access_to_object(C, C2, [access|L], L2) :-
general_access_modifier(C1, [*|C2], L, L1),
nonarray_type(C, C1, L1, L2).

accesss_to_object(C, C2, [access,all,id(float)], L2)

4a. 7 names: main function, struct main, main member (used three times), outer "main" variable (used once), enum value (used twice), label (used once), inner "main" variable (used three times). Exact uses left as an exercise for the reader.

4b. Identifiers declared inside begin-end braces are not visible outside, except for labels, whose scope are the containing function. There are multiple name spaces: ordinary names, labels, struct tags, and member names. Syntactic context determines which space a name belongs to, e.g., a name after "goto" or before ":" must be a label. Names in an inner scope shadow names from the same name space declared in an outer scope.

4c. In "struct main main = main;" the variable "main" is used without first being initialized.

5a.
```
(define (maplbt f obj)
  (let mapit ((obj obj))
    (cond ((pair? obj)
           (cons (mapit (car obj))
                 (mapit (cdr obj))))
          ((null? obj)
           '())
          (else
           (f obj)))))
```

5b. type 'a bt = Cons of 'a bt * 'a bt | Leaf of 'a option

```
      let map1bt f obj =
        let rec mapit = function
          | Cons (a, d) -> Cons (mapit a, mapit d)
          | Leaf None -> Leaf None
          | Leaf Some obj -> f obj
        in mapit obj

5c.   (define (btmap procs obj)
         (map1bt (lambda (proc) (proc obj)) procs))

5d.   (define (btmapi procs obj)
         (mapibt (lambda (proc) (proc obj)) procs))
      where mapibt is the solution to Homework 2.

5e.   (define (maptrees tp to)
         (map1bt (lambda (f) (map1bt f to)) tp))

      Here's another answer, which is not quite as nice:

      (define (maptrees tp to)
         (btmap (map1bt (lambda (f) (lambda (bt) (map1bt f bt))) tp) to))
```

::::::::::::::
mid2.txt
::::::::::::::

UCLA Computer Science 131 (Spring 2004) Section 2 midterm
100 minutes total, open book, open notes

Name:_____ Student ID:_____

1 (5 minutes). Of the languages C++, Ocaml, Prolog, and Scheme, which is the most likely to have implementations where programs execute quickly? List the languages in rough order of expected runtime efficiency, and briefly justify your ordering.

2 (5 minutes). Is a DCG rule a Horn clause? Explain why or why not.

3 (5 minutes). Does the following C function have well-defined behavior regardless of the value of x? If so, explain the behavior; if not, explain why not.

```
int f (int x) { return -x == ~x + 1; }
```

4 (10 minutes). In C and C++, parentheses are required around the test expression of an if-statement (e.g., "if (x == 3) x++;"), but not around the returned-value expression of an return statement (e.g., "return x++;"). Why are the parentheses necessary in the former but not the latter? Give an example of what would go wrong if parentheses were not required around the former.

5 (10 minutes). An identifier is said to have "discontiguous scope" if the locations where it is visible are not all adjacent to each other in the text of the program. Give an example of discontiguous scope in Scheme; or, if it's not possible, explain why not. Similarly, give an example in Prolog (or explain why you can't).

6 (10 minutes). Translate the following Scheme function to Ocaml. Style

counts here, so please use an idiomatic rather than a literal translation.

```
(define (all-pairs? list)
  (or (null? list)
      (and (pair? (car list))
           (all-pairs? (cdr list)))))
```

7 (15 minutes). Consider the following section of code in the sample solution for Homework 1:

```
id([id(X) | C], C) --> [id(X)], {\+ c_keyword(X)}.

c_keyword(auto).
c_keyword(break).
c_keyword(case).
...
```

Suppose the grammar rule was rewritten as follows:

```
id([id(X) | C], C) --> {\+ c_keyword(X)}, [id(X)].
```

What would go wrong, and why? Give an example use of the Prolog "id" predicate that illustrates the problem.

8. Consider the following code, taken from the sample solution for Homework 2:

```
(define (mapbt f obj1 . objs)
  (let mapit ((obj1 obj1) (objs objs))
    (cond ((pair? obj1)
           (cons (mapit (car obj1) (map car objs))
                 (mapit (cdr obj1) (map cdr objs))))
          ((null? obj1) '())
          (#t (apply f obj1 objs)))))
```

8a (5 minutes). Is mapbt tail-recursive? If so, why? If not, why not?

8b (5 minutes). Suppose id is the identity function (lambda (x) x). If bt is a binary tree, what is the difference (in the caller's point of view) between (id bt) and (mapbt id bt)? Give an example use that illustrates the difference.

8c (10 minutes). Suppose we remove the line "((null? obj1) '())" from the definition of mapbt. How will this change the behavior of mapbt? Give an example use that illustrates the change in behavior.

9 (20 minutes). For each of the following Scheme top-level expressions, give the simplest possible expression that has the same external behavior whenever the original expression has well-defined behavior. Your simplifications may use side effects, and may use any of the predefined Scheme procedures or special forms. For the purpose of this question, an expression is "simpler" if it has fewer tokens, where tokens are as described in R5RS section 7.1.1. If the expression cannot be simplified, say so.

```
(not not)
(lambda (x) (if x x (not x)))
(cons cons '())
(lambda (x) (cons 1 (cons 2 (cons x '()))))
((lambda (x) (x x)) symbol?)
(let ((y (lambda (y) y))) (let ((y (lambda (x) y))) (y 10)))
```

```
(lambda (x) (if (car x) (cdr x) (car x)))
(lambda (x) (call/cc (lambda (k) (x k))))
(lambda (f x) (let g ((x x))
                (if (null? x) x (cons (f (car x)) (g (cdr x))))))
(letrec ((p (lambda (f) (f p)))) (p (lambda (x) x)))
```

::::::::::::::
mid2ans.txt
::::::::::::::

1. C++ should be fastest since it descends from C and has a great deal of compile-time information about its objects, and (if the program is well designed) does not need a garbage collector. Ocaml should be next, since it has compile-time type checking. Scheme requires run-time checking. Prolog requires support for unification and backtracking, which is perhaps the slowest computation model here.

2. DCG rules are compiled into Horn clauses.

3. The behavior is not well-defined for two reasons. First, C does not specify whether negative numbers are implemented using twos-complement, ones-complement, or signed-magnitude; for example, only on twos-complement machines is ~x == 1-x. Second, arithmetic overflow can occur in either the negation or the addition, and the resulting behavior is undefined.

4. Parentheses are needed to avoid syntactic ambiguity. For example, the following are both valid C and C++ statements with different semantics:

```
if (f (g ())) /* empty statement */; else break;
if (f) (g ()); else break;
```

   but if you remove the parentheses in question, they'd both look the same:

```
if f (g ()); else break;
```

5. (let ((x 1)) (+ (let ((x x)) x) x))
   The outer 'x' has discontiguous scope because the inner 'x' shadows it. This isn't possible in Prolog, where a logical variable always has scope that extends to the entire containing clause.

6.
```
type 'a bt = Pair of 'a bt * 'a bt | Nonpair of 'a | Null
let rec all_pairs = function
  | [] -> true
  | Null :: _ -> false
  | Nonpair _ :: _ -> false
  | Pair _ :: tail -> all_pairs tail
```

7. \+ c_keyword(X) would be invoked before X is instantiated, c_keyword(X) would succeed with X bound to, say "auto", and therefore \+ c_keyword(X) would always fail. Hence this grammar rule would always fail. Here's an example use:
   id([id(foo)], R, C1, []).

8a. mapbt is not tail recursive because its result is passed to cons before mapbt returns.

8b. (eq? bt (id bt)), but this is not true for (mapbt id bt) since the latter returns a copy of bt.

8c. It will cause f to be invoked on the empty list, and will return whatever f returns. For example, (mapbt (lambda (x) 1) '(a)) returns (1) without the change, but with the change it returns (1 . 1).

9.
```
#f
(lambda (x) (or x #t))
(list cons)
(lambda (x) (list 1 2 x))
#f
(lambda (y) y)
(lambda (x) (and (car x) (cdr x)))
call/cc
map
(letrec ((p (lambda (f) (f p)))) p)
```