Fall, 2004 Name: _____

Com S 541 — Programming Languages 1

Test on Logic Programming, λProlog, and Operational Semantics

Special Directions for this Test

This test has 5 questions and pages numbered 1 through 13.
This test is open book and notes.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Unless we specifically request, your λProlog code does not have to run "backwards"; that is, you do not have to test relations where the last argument is given and other arguments are sought. In the examples you may see "no (more) solutions" or "yes" printed; your program does not have to have exactly this output; that is, it could have more solutions than our solution for some queries, or fewer.
You may use cut (`!`) and `not` in your code.
You may use the following module in your code without writing it into your test.

```
sig rtc.
  type rtc (A -> A -> o) -> (A -> A -> o).
end

%%% reflexive, transitive closure of a relation.

module rtc.

  rtc P X X.
  rtc P X Z :- (P X Y), (rtc P Y Z).
```

When you write λProlog code on this test, you may use anything built-in to λProlog without writing it in your test. You are encouraged to define relations not specifically asked for if they are useful to your programming; these should be written into your test if they are not built-in.

1. (5 points) Suppose we give λProlog the following query (which follows the prompt `?-`).

   ```
   ?- 1 + 4 = 5.
   ```

   Circle the correct answer and give a brief explanation.

   (a) The query succeeds (prints "yes").
   (b) The query fails (prints "no (more) solutions").
   (c) The query loops forever.
   (d) The query produces an error message.

2. Consider the following λProlog module.

```
sig q2.
  type memb    A -> (list A) -> o.
  type join    (list A) -> (list A) -> (list A) -> o.

  type member  A -> (list A) -> o.
  type conjoin (list A) -> (list A) -> (list A) -> o.
end


module q2.
  memb X (X::L).
  memb X (Y::L) :- memb X L.

  join nil K K.
  join (X::L) K M :- memb X K, !, join L K M.
  join (X::L) K (X::M) :- join L K M.

  member X (X::L) :- !.
  member X (Y::L) :- member X L.

  conjoin nil K K.
  conjoin (X::L) K M :- member X K, conjoin L K M.
  conjoin (X::L) K (X::M) :- conjoin L K M.
end
```

(a) (5 points) Fill in the two blanks (___) in the following transcript from running λProlog on this module. (You can write one or more lines in the second blank.)

```
[q2] ?- join (1::nil) (1::2::nil) M.

The answer substitution:

M = ____________________

More solutions (y/n)? y

________________________
```

(b) (5 points) Fill in the two blanks (___) in the following transcript from running λProlog on this same module, but using `conjoin` instead of `join`. (You can write one or more lines in the second blank.)

```
[q2] ?- conjoin (1::nil) (1::2::nil) M.

The answer substitution:

M = ____________________

More solutions (y/n)? y

________________________
```

3. (20 points) This problem is to find the free variables in expressions drawn from a grammar. The grammar is a simplified version of Java, called "Classic Java" (Flatt, *et al.*, 1999). For this problem we will only be concerned with the expressions. (Later problems will use other parts.)

```
sig classicjava_syntax.
  % nonterminals
  kind program type.
  kind declaration type.
  kind fielddecl type.
  kind methoddecl type.
  kind expression type.
  kind superclassname type.

  % programs (declarations, "main" expression)
  type prog (list declaration) -> expression -> program.

  % top-level declarations (class name, superclass name, fields, methods)
  type class string -> string -> (list fielddecl) -> (list methoddecl)
             -> declaration.

  % field declarations (name)
  type field string -> fielddecl.

  % method declarations (name, formal parameter name, body)
  type meth string -> string -> expression -> methoddecl.

  % expressions
  type new     string -> expression.   % instance creation (class name)
  type var     string -> expression.   % variable reference (variable name)
  type this    expression.             % this (the self object)
  type null    expression.             % null reference expression
  % method call (target expression, method name, actual argument expression)
  type mcall   expression -> string -> expression -> expression.
  % field dereference (target expression, field name)
  type dot     expression -> string -> expression.
  % field assignment (target, field name, right hand expression)
  type fassign expression -> string -> expression -> expression.
  % sequential composition
  type semi    expression -> expression -> expression.
end

module classicjava_syntax.  end  % see the signature file.
```

In this grammar, variable references are only found in abstract syntax trees of the form `var` *s*, where *s* is a string. Your problem is to specify, in λProlog, the relation `free` whose signature is given by

```
sig classicjava_free.
  accum_sig classicjava_syntax.
  type free expression -> (list string) -> o.
end
```

and is such that `free E L` holds if `L` contains just the strings that are the names of all the free variable references found in `E`. The order of the strings in `L` does not matter, and duplicates are permitted. (Hint: You can use the module `q2` from problem 2 above if you wish.)

Examples are contained in the test module shown below.

```
sig free_tests.
  accum_sig test_helpers, classicjava_free.
end

module free_tests.
  accumulate test_helpers, classicjava_free.
  import set_equals.

  local testcase int -> expression -> (list string) -> o.

  %% adapters from test_helpers.
  check_exists 1 J :- testcase J _ _.
  check 1 J :- testcase J E L, free E L', set_equals L L'.

  %          expression     expected answer
  testcase 1 (var "y")      ("y" :: nil).
  testcase 2 (new "Object") nil.
  testcase 3 this           nil.
  testcase 4 null           nil.
  testcase 5 (mcall (var "o") "m" (var "y")) % expression
             ("o" :: "y" :: nil).            % answer
  testcase 6 (dot (mcall (var "o") "m" (mcall this "h" (var "y"))) "f")
             ("o" :: "y" :: nil).
  testcase 7 (fassign this "f" (mcall (dot (var "o") "g")
                                      "p" (dot (var "x") "h")))
             ("o" :: "x" :: nil).
  testcase 8 (semi (fassign (var "y") "f" (var "z")) (dot (var "y") "f"))
             ("z" :: "y" :: nil).
end
```

Please write your answer below

```
module classicjava_free.
```

4. (30 points) In this problem you will program the following three auxiliary functions for the grammar in problem 3.

```
sig classicjava_aux.
  accum_sig classicjava_syntax, q2, rtc.
  type direct_subclass (list declaration) -> string -> string -> o.
  type subtypeof (list declaration) -> string -> string -> o.
  type fieldsof (list declaration) -> string -> (list string) -> o.
end
```

   - The query `direct_subclass CT Sub Sup` succeeds if, in the list of declarations `CT`, `Sub` is a direct subclass of `Sup`. That is, there must be a declaration in `CT` of the form `class Sub Sup` ....
   - The query `subtypeof CT Sub Sup` succeeds if, in the list of declarations `CT`, `Sub` is a direct or indirect subclass of `Sup`. A class is considered a subtype of itself, even if it is not in `CT`.
   - The query `fieldsof CT C Fs` succeeds if, in the list of declarations `CT`, `C` and its supertypes declare all the fields in `Fs`, or if `C` is the string `"Object"` and `Fs` is `nil`. This last clause means that the class Object has no fields.

Assume all classes in a program have distinct names. Examples are contained in the test module shown below. (There is space following this for your answer.)

```
sig aux_tests.
  accum_sig test_helpers, classicjava_aux.
end

module aux_tests.
  accumulate test_helpers, classicjava_aux.
  import set_equals.

  local direct_subclass_test int -> string -> string -> o.
  local not_direct_subclass_test int -> string -> string -> o.
  local subtypeof_test int ->  string -> string -> o.
  local not_subtypeof_test int ->  string -> string -> o.
  local fieldsof_test int -> string -> (list string) -> o.
  local example (list declaration) -> o.

  %% adapters from test_helpers.
  check_exists 1 J :- direct_subclass_test J _ _.
  check_exists 2 J :- not_direct_subclass_test J _ _.
  check_exists 3 J :- subtypeof_test J _ _.
  check_exists 4 J :- not_subtypeof_test J _ _.
  check_exists 5 J :- fieldsof_test J _ _.
  check 1 J :- direct_subclass_test J Sub Sup, example CT,
               direct_subclass CT Sub Sup.
  check 2 J :- not_direct_subclass_test J Sub Sup, example CT,
               not (direct_subclass CT Sub Sup).
  check 3 J :- subtypeof_test J Sub Sup, example CT,
               subtypeof CT Sub Sup.
  check 4 J :- not_subtypeof_test J Sub Sup, example CT,
               not (subtypeof CT Sub Sup).
  check 5 J :- fieldsof_test J C Fs, example CT, fieldsof CT C Fs',
               set_equals Fs Fs'.

  % example declarations for use in testing
```

```
example (
  (class "ColoredStack" "Stack" ((field "color") :: nil) nil) ::
  (class "Stack" "Object" ((field "elems") :: (field "size") :: nil) nil) ::
  (class "CachedStack" "Stack" ((field "cache") :: nil) nil) ::
  (class "E" "D" ((field "fe1") :: (field "fe2") :: nil) nil) ::
  (class "B" "A" ((field "fb1") :: (field "fb2") :: nil) nil) ::
  (class "C" "B" ((field "fc1") :: (field "fc2") :: nil) nil) ::
  (class "D" "C" ((field "fd1") :: (field "fd2") :: nil) nil) ::
  (class "A" "Object" ((field "fa1") :: (field "fa2") :: nil) nil) ::
  nil).

% direct_subclass examples that succeed
direct_subclass_test 1 "Stack" "Object".
direct_subclass_test 2 "ColoredStack" "Stack".
direct_subclass_test 3 "CachedStack" "Stack".
direct_subclass_test 4 "B" "A".
direct_subclass_test 5 "C" "B".
direct_subclass_test 6 "D" "C".
direct_subclass_test 7 "E" "D".

% direct_subclass examples that should fail (so their negation succeeds).
not_direct_subclass_test 1 "Object" "Object".
not_direct_subclass_test 2 "CachedStack" "ColoredStack".
not_direct_subclass_test 3 "ColoredStack" "CachedStack".
not_direct_subclass_test 4 "Stack" "CachedStack".
not_direct_subclass_test 5 "Stack" "A".
not_direct_subclass_test 6 "A" "B".
not_direct_subclass_test 7 "C" "A".
not_direct_subclass_test 8 "D" "A".
not_direct_subclass_test 9 "E" "A".

% subtypeof examples that succeed
subtypeof_test J X Y :- J < 8, !, direct_subclass_test J X Y.
subtypeof_test 8 "Object" "Object".
subtypeof_test 9 "ColoredStack" "Object".
subtypeof_test 10 "A" "A".
subtypeof_test 11 "C" "A".
subtypeof_test 12 "D" "A".
subtypeof_test 13 "E" "A".

% subtypeof examples that should fail (so their negation succeeds).
not_subtypeof_test 1 "CachedStack" "ColoredStack".
not_subtypeof_test 2 "ColoredStack" "CachedStack".
not_subtypeof_test 3 "A" "B".
not_subtypeof_test 4 "A" "C".
not_subtypeof_test 5 "A" "D".
not_subtypeof_test 6 "A" "E".

%               class name     expected answer
fieldsof_test 1 "Object"       nil.
fieldsof_test 2 "ColoredStack" ("color"::"elems"::"size"::nil).
fieldsof_test 3 "Stack"        ("elems"::"size"::nil).
fieldsof_test 4 "CachedStack"  ("cache"::"elems"::"size"::nil).
fieldsof_test 5 "E"            ("fe1"::"fe2"::"fd1"::"fd2"::
```

```
                                  "fc1"::"fc2"::"fb1"::"fb2"::
                                  "fa1"::"fa2"::nil).
  fieldsof_test 6 "C"            ("fc1"::"fc2"::"fb1"::"fb2"::
                                  "fa1"::"fa2"::nil).
  fieldsof_test 7 "A"            ("fa1"::"fa2"::nil).
end
```

Please write your answer below. In your answer, you may use the module `rtc` described in class without writing it into your solution. (Hint, hint.)

```
module classicjava_aux.
```

The following problems are about the operational semantics of Classic Java (see the grammar in problem 3). This page contains some background information, for later reference.

Assume a module that correctly implements the following signature is defined.

```
sig classicjava_aux2.
  accum_sig classicjava_aux.

  %% ‘‘methodBody CT C M F Body’’ succeeds if F the formal parmeter name
  %% and Body is the body of method M in class C, as declared in CT.
  type methodBody (list declaration) -> string -> string
                  -> string -> expression -> o.

  %% ‘‘substFor As Fs Body Body2’’ succeeds if Body2 results from
  %% substituting the As for the corresponding Fs, in Body.
  %% It is assumed that As and Fs have the same length, and that the As
  %% do not occur in the Fs.
  type substFor (list expression) -> (list expression)
                -> expression -> expression -> o.
end
```

The domains of (expressible) values and storable values are defined by the following module's signature.

```
sig domains.
  accum_sig finite_function.

  kind value    type.  % expressible values
  kind storable type.  % storable values

  %% constructors for values
  % references (address)
  type ref     int -> value.
  % null references ()
  type nullref value.

  %% constructors for storable values
  % objects (class name, fields map)
  type object  string -> (finite_fun string value) -> storable.
end
```

We add one piece of syntax as well.

```
sig classicjava_added_syntax.
  accum_sig classicjava_syntax.
  % Syntax that is not user-visible but used in the semantics
  type loc int -> expression.
end
```

Finally, we assume the modules `store` and `finite_function`, as in the homework.

The module `classicjava` below defines the operational semantics of our simplified version of Classic Java. Following this are some questions. You may want to come back to the semantics after reading the questions.

```
sig classicjava.
  accum_sig classicjava_aux2, classicjava_added_syntax, store, domains.

  % configurations
  kind configuration type.

  type config expression -> (store storable) -> configuration.

  % input, reduction, and output relations
  type inputP   program -> configuration -> o.
  type reduces (list declaration)
                -> configuration -> configuration -> o.
  type outputP  configuration -> value -> store storable -> o.
  type outputE  configuration -> value -> store storable -> o.

  % meaning functions
  type meaning program -> value -> store storable -> o.
  type meaningE (list declaration)
                -> expression -> store storable
                -> value -> store storable -> o.
end

module classicjava.

  accumulate classicjava_aux2, classicjava_added_syntax, store, domains.

  %% Programs

  %% ``meaning Prog Val Store'' succeeds if Val and Store are
  %% the final value and store that result from evaluating Prog
  %% in the initial store.
  meaning (prog Decls Body) Val Store :-
    inputP (prog Decls Body) Config,
    rtc (reduces Decls) Config Config',
    outputP Config' Val Store.

  %% ``inputP C Config'' succeeds if Config is the configuration that
  %% contains C and the intial store.
  inputP (prog Decls Body) (config Body S0) :-
      initial_store S0.

  %% ``outputC Config S'' succeeds if Config is terminal and contains store S.
  outputP (config (loc L) S) (ref L) S.
  outputP (config null S)    nullref S.

  %% ``newObjectValueFor CT C O'' succeeds if O is an initial object value,
  %% with all null fields, for an object of class C (determined by CT).
  local newObjectValueFor (list declaration) -> string -> storable -> o.
  newObjectValueFor CT C (object C FMap) :-
    fieldsof CT C Fs, nullForAll Fs FMap.
```

```
%% ‘‘nullForAll Fs FMap’’ succeeds if FMap is a finite function
%% that maps each element of Fs to nullref.
local nullForAll (list string) -> (finite_fun string value) -> o.
nullForAll nil empty_f_fun.
nullForAll (F::Fs) (f_fun_extend F nullref FMap) :- nullForAll Fs FMap.

%% Expressions

%% ‘‘meaningE CT E S V S2’’ succeeds if,
%% with the list of declarations CT,
%% evaluating expression E in store S,
%% produces the value V and final store S2.
meaningE CT E S V S’ :-
  rtc (reduces CT) (config E S) (config E’ S’),
  outputE (config E’ S’) V S’.

%% ‘‘outputE ConfigE V’’ succeeds if V is the final value of
%% the expression configuration ConfigE.
outputE (config (loc L) S) (ref L) S.
outputE (config null S)    nullref S.

%% ‘‘value2exp V E’’ succeeds if the value V is equivalent to expression E.
local value2exp value -> expression -> o.
value2exp (ref L) (loc L).
value2exp nullref null.

%% ‘‘isTerminal E’’ succeeds if the expression E is terminal.
local isTerminal expression -> o.
isTerminal (loc _).
isTerminal null.

%% ‘‘reduces CT Config1 Config2’’ succeeds if there is a one step
%% (expression) reduction from configuration Config1,
%% which results in Config2.

%% [new]
    ((newObjectValueFor CT C O), (alloc O S L S’))
=> % ------------------------------------------------
    (reduces CT (config (new C) S) (config (loc L) S’)).

%% var expressions are substituted out (they would be terminal/errors)

%% null and loc are terminal

%% [method-call]
     ((access S L0 (object C FMap)),
      (methodBody CT C M Formal Body),
      (substFor ((loc L0)::(loc L1)::nil) (this::(var Formal)::nil)
                Body Body’))
=> % ----------------------------------------------------
     (reduces CT (config (mcall (loc L0) M (loc L1)) S)
                 (config Body’ S)).
```

```
%% [method-reduce-target]
     (reduces CT (config E0 S) (config E0' S'))
=> % ------------------------------------------------
     (reduces CT (config (mcall E0 M (E1)) S)
                 (config (mcall E0' M (E1)) S')).

%% [method-reduce-argument]
     (reduces CT (config E1 S) (config E1' S'))
=> % ------------------------------------------------
     (reduces CT (config (mcall (loc L0) M (E1)) S)
                 (config (mcall (loc L0) M (E1')) S')).

%% [dot]
     ((access S L (object C FMap)),
      (f_fun_apply FMap F V),
      (value2exp V E))
=> % ------------------------------------------------
     (reduces CT (config (dot (loc L) F) S)
                 (config E S)).

%% [dot-reduce-target]
     (reduces CT (config E S) (config E' S'))
=> % ------------------------------------------------
     (reduces CT (config (dot E F) S)
                 (config (dot E' F) S')).

%% [fassign]
     ((access S L0 (object C FMap)),
      (value2exp V1 (loc L1)),
      (update S L0 (object C (f_fun_extend F V1 FMap)) S'))
=> % ------------------------------------------------
     (reduces CT (config (fassign (loc L0) F (loc L1)) S)
                 (config (loc L1) S')).

%% [fassign-reduce-target]
     (reduces CT (config E0 S) (config E0' S'))
=> % ------------------------------------------------
     (reduces CT (config (fassign E0 F E1) S)
                 (config (fassign E0' F E1) S')).

%% [fassign-reduce-rhs]
     (reduces CT (config E1 S) (config E1' S'))
=> % ------------------------------------------------
     (reduces CT (config (fassign (loc L0) F E1) S)
                 (config (fassign (loc L0) F E1') S')).

%% [semi-reduce-left]
     (reduces CT (config E1 S) (config E1' S'))
=> % ------------------------------------------------
     (reduces CT (config (semi E1 E2) S)
                 (config (semi E1' E2) S')).
```

```
  %% [semi-done-left]
       (isTerminal E1)
  => % ------------------------------------------------
       (reduces CT (config (semi E1 E2) S)
                   (config E2 S)).
end
```

5. Answer each of the following and give a brief explanation.

   (a) (10 points) In the above semantics, what is the parameter passing mechanism used to call methods? Is it call by value or normal order evaluation? That is are arguments always evaluated before methods are called?

   (b) (5 points) What happens when the target of a method call is `null`? That is, what does the semantics say about a call such as `mcall null "m" (`E`)`?

   (c) (10 points) Are subexpressions of an expression evaluated in any particular order? If so, in what order are the subexpressions of an expression evaluated?

   (d) (10 points) Does this semantics support recursive methods? If so briefly explain how they are processed, and if not, explain why a recursive method would not work in the semantics.