

Name: _____

CSCI-4430 Programming Languages

Final Exam, Part I

This exam consists of 12 sections with questions worth a total of 91 points (the raw scores will be rescaled to a 100 point scale). It is closed book: no notes, printed material, computers, calculators, or communicating devices may be used. You have 3 hours in which to complete the exam, but you might need much less time (if you are really on top of this stuff). I suggest that you read over the entire exam before beginning to write your answers.

The exam is split into **two separately stapled parts** (so that they can be graded in parallel). **Be sure your name is filled in on the first page of both parts.** Write your answers on the exam pages. If you need scratch paper, pick some up from the stack on the table in the front of the room. If you use extra sheets for your answers, be sure it's clear which question they go with and staple them to the back of that part of the exam (using the stapler on the table in the front of the room).

Please don't ask for hints or even for clarifications. Understanding the questions is part of the exam. However, if you believe there is an outright **error** in the statement of a question, do bring it to my (or the proctor's) attention.

If you finish early, don't turn in your exam until you have checked over your answers, **making sure in each case that the question you answered was the question that was asked.**

GOOD LUCK! (And have a good winter break.)

1 Thread Expressions

Oz provides thread expressions as syntactic sugar for its kernel language thread statement. The following code is the kernel language version of a single thread expression.

```
local R1 R2 in
  thread local A in A=N-1 {Fib A R1} end end
  thread local B in B=N-2 {Fib B R2} end end
  {Number. '+' R1 R2 R}
end
```

Write that thread expression (in as simple a form as you can).

2 pts

2 Producer/Consumer Stream Programming

The following code is the same as the Sieve of Eratosthenes example (the original, unoptimized version) discussed in the textbook and one of the labs, except that the `Sieve` function is expressed as a procedure instead of a function.

```

fun {Generate N Limit}
  if N<Limit then
    N|{Generate N+1 Limit}
  else nil end
end
proc {Sieve Xs R}
  case Xs
  of nil then R=nil
  [] X|Xr then Ys in
    {Delay 1}
    thread Ys={Filter Xr fun {$ Y} Y mod X \= 0 end} end
    local Z in      %%
      {Sieve Ys Z} %%% version 1
      R=X|Z         %
    end            %%
  end
end
% Main program
local Xs Ys in
  thread Xs={Generate 2 10000} end
  thread Ys={Sieve Xs} end
  {Browse Ys}
end

```

1. Rewrite the function call of `Sieve` in the main program as a procedure call: 1 pt

2. (Circle one) true or false: given that `Sieve` is now defined as a procedure, it is necessary to change all function calls of `Sieve` (such as the one in the main program) into procedure calls. 1 pt

3. The local statement containing the recursive call of `Sieve` could also be written

```

local Z in      %%
  R=X|Z          %%% version 2
  {Sieve Ys Z} %
end            %%

```

Which version would be produced by Oz in translating the expression

```
X|{Sieve Ys}
```

into kernel language, (circle one) version 1 or version 2?

1 pt

4. Which version of `Sieve`'s definition is iterative, the one with the code in (circle one) version 1 or version 2? Explain.

2 pts

5. One version will begin showing the primes immediately, while the other version will not show any output in the Browser (except for an underscore) until the computation is done—which, because of the `Delay` call, will be more than 10 seconds after computation begins. Which version will begin showing them immediately, (circle one) version 1 or version 2?

1 pt

6. Suppose you remove the use of a thread statement when creating a new filter: replace

```
thread Ys={Filter Xr fun {$ Y} Y mod X \= 0 end} end
```

with

```
Ys={Filter Xr fun {$ Y} Y mod X \= 0 end}
```

Circle one: true or false: the program will still produce exactly the same list of primes as before.

1 pt

3 Lazy Stream Programming

1. Useful concepts in lazy stream programming include lazy producers, lazy consumers, and lazy _____ (the lazy function `Distinct` in the Hamming numbers concurrency problem is an example).
2. In order to request elements of a lazy stream, develop a function `{Request Xs N}` that requests the first `N` elements of the stream `Xs`.

2 pts

4 pts

4 Concurrent Waiting

1. For waiting on a single variable Oz provides the `wait` procedure: `{wait x}` waits until variable `x` is bound. By “waits” we mean it suspends execution of the _____ in which the `wait` call occurs. **1 pt**
(Fill in the blank with something besides “statement.”)

2. Write a procedure `waitAll` that takes a list of statements, each packaged in a nullary procedure, executes each statement in its own thread, and waits until all have terminated. For example,

```
{waitAll [proc {$} {Delay 6000} {Browse a} {Delay 2000} end
         proc {$} {Delay 2000} {Browse b} end]
{Browse c}
```

should display `b` after about 2 seconds, `a` after about 6 seconds and `c` after about 8 seconds. **6 pts**

Hint: Include in each thread created, as its last statement, a binding of a variable to a value. Then apply `wait` to each of the variables, so you set up waiting until all variables used in the threads are bound.

If you have trouble programming this for a list of statements, for half credit write a special case procedure `waitThree` such that `{waitThree S1 S2 S3}` runs `S1`, `S2`, and `S3` each in its own thread and waits until all three have terminated.

3. We’ve also seen a function called `waitTwo`, but it has a different meaning from a special case of `waitAll`. Describe the semantics of the function call

```
{waitTwo X Y}
```

where `X` and `Y` are variables, as defined in the textbook. **3 pts**

4. Describe a programming situation where you would need `WaitTwo`. (You may use a situation that occurred in one of the homework problems.)

2 pts

5 Agents With State

Using

```

fun {NewAgent Process InitState}
  Port Stream
in
  Port={NewPort Stream}
  thread Dummy in
    Dummy={FoldL Stream Process InitState}
  end
  Port
end

```

we created a bank account agent abstraction by writing a `BankProcess` function that we could plug into `NewAgent` to get a bank account agent able to handle withdrawals, deposits, and balance requests. The state of the agent is the account's balance, initially 0.

```

fun {BankProcess S M}
  case M
  of withdraw(N) then S-N
  [] deposit(N) then S+N
  [] balance(B) then B=S S
  end
end

```

For example, we can write

```

BA={NewAgent BankProcess 0}
{Send BA deposit(100)}
{Send BA deposit(100)}
{Send BA withdraw(255)}
local B in {Send BA balance(B)} {Wait B} {Show B} end

```

The balance shown would be negative, because this code doesn't prevent an overdraft (a withdrawal of more than the current balance). Bob Threadman attempts to solve this problem by programming

```

proc {SafeWithdraw BA W}
  B in {Send BA balance(B)}
  if W =< B then {Send BA withdraw(W)}
    {Show `You withdrew `#W#` dollars, balance is now `#B-W`}
  end

```

```

    else {Show `Sorry, no overdrafts are allowed!`}
    end
end

```

and providing only this function to bank customers (along with methods `balance(N)` and `deposit(N)`): they are allowed to do `{SafeWithdraw BA Amount}` but not `{Send BA withdraw(Amount)}`. (We are assuming that some means of authorization to use the account is applied but we're not dealing with it here. The only thing to note is that multiple customers, such as spouses, could have authorized access to the same account, and they could be accessing it concurrently.)

1. Bob's solution is not good enough! Describe a scenario in which an overdraft could still occur.

3 pts

2. Bob now recognizes that to do the withdrawal operation safely a message `safewithdraw(W B Ok)` needs to be added to `BankProcess` that does several operations together, atomically: check the balance for sufficient funds, make the withdrawal if possible, return the (unchanged or new) balance, and indicate whether the transaction was completed or not. He can then code the user-level procedure `SafeWithdraw` as follows:

```

proc {SafeWithdraw BA W}
  B Ok in {Send BA safewithdraw(W B Ok)}
  if Ok then
    {Show `You withdrew `#W#` dollars, balance is now `#B`}
  else {Show `Sorry, no overdrafts are allowed!`}
  end
end

```

Help Bob out by writing the code for handling the `safewithdraw(W B Ok)` message. Remember that the `BankProcess` function must always return the account balance.

3 pts

6 Erlang Semantics

Suppose an Erlang process executes the following receive expression

```

receive
  {public_key_request, From, FromName} ->
    io:format("Bob: I received a request from ~s for my public key,~n"
              ++ "  which I'll now grant.~n~n", [FromName]),
    From ! {granted, Public, self(), "Bob"};
    Logger ! {public_key_request_granted, self(), From};
  {secret_key_request, From, FromName} ->
    io:format("Bob: I received a request from ~s for my secret key.~n"
              ++ "  The nerve!~n~n", [FromName]),
    From ! {no_way, self(), "Bob"};
    Logger ! {secret_key_request_declined, self(), From}
end

```

1. Suppose also that the process's mailbox contains both a `secret_key_request` message (sent to the mailbox first) and a `public_key_request` message (sent to the mailbox second). According to the semantics of Erlang's `receive` expression—which we know well from having modeled it in Oz—which message will be accepted by that receive expression? Circle one:

2 pts

- (a) the `secret_key_request`
- (b) the `public_key_request`

2. The message to the `Logger` process (in either case) is sent (circle one):

2 pts

- (a) immediately without waiting for the send of the message back to `From` to complete
- (b) only after the send of the message back to `From` has completed.

(Again, the answer can be deduced from the way we modeled the semantics of Erlang's message handling in Oz.)

7 ADT Implementation Choices

In one of the labs we wrote an implementation called `MyArray` of an Array ADT in terms of a tuple of cells, with the following interface:

- `A={MyArray.new L H I}` returns a new array with indices from `L` to `H`, inclusive, all initialized to `I`.
- `{MyArray.put A I X}` puts in `A` the mapping of `I` to `X`.
- `X={MyArray.get A I}` returns from `A` the mapping of `I`.
- `L={MyArray.low A}` returns the lower index bound of `A`.
- `H={MyArray.high A}` returns the higher index bound of `A`.

1. Among the choices one has in implementing an ADT are whether to make it stateful or declarative, and whether to make it unbundled or bundled. In this case, these choices are already determined by the problem description. It is

- (a) Circle one: stateful vs. declarative. Explain why this is determined by the interface.

3 pts

- (b) Circle one: bundled vs. unbundled. Explain why this is determined by the interface.

3 pts

2. We discussed one other dimension of choices that one has in implementing an ADT. It is

_____ vs. _____

3 pts

Name: _____

CSCI-4430 Programming Languages

Final Exam, Part II

8 Parameter Passing Methods

Below are five example programs that model five different parameter passing methods.

1. Fill in each blank after “Call by” with the name of the parameter passing method that is being modeled in the example code (that follows the blank). Hint: the names of four of the five methods are: value, value-result, reference, name. 5 pts

2. Also fill in each blank after “displays:” with the value that would be displayed in the Browser. 5 pts

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Call by _____

proc {Sqr D}
  A={NewCell D}
in
  A:=@A+1
  {Browse @A*@A} % displays:_____
end
{Sqr 8}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Call by _____

proc {Sqr A}
  B={A}
in
  B:=@B*@B
end
local
  C={NewCell 0}
in
  C:=8
  {Sqr fun {$} C end}
  {Browse @C} % displays:_____
end
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Call by _____

proc {Sqr A}
  D={NewCell @A}
in
  D:=@D*@D
  A:=@D
end
local
  C={NewCell 0}
in
  C:=8
  {Sqr C}
  {Browse @C} % displays:_____
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Call by _____

proc {Sqr A}
  A:=@A*@A
end
local
  C={NewCell 0}
in
  C:=8
  {Sqr C}
  {Browse @C} % displays:_____
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Call by _____

proc {Sqr A}
  {A}:=@{A}*@{A}
end
local
  C={NewCell 0}
in
  C:=8
  {Sqr fun {$} C end}
  {Browse @C} % displays:_____
end

```

3. C++ provides direct language support for two of these five parameter passing methods. (The other three have to modeled using a combination of other language features.) They are

- (a) Call by _____.
- (b) Call by _____.

1 pt

1 pt

10 Object Oriented Programming

Consider the following code defining a `Log` class for keeping a log of bank transactions, an `Account` class defining transfer and balance methods, and a `LoggedAccount` class that inherits from `Account` and extends the meaning of `Account`'s `transfer` method to include adding the transaction to the log.

```

class Log
  attr logged
  meth init
    logged:=nil
  end
  meth addentry(E)
    logged:=E|@logged
  end
  meth report(L)
    L={Reverse @logged}
  end
end
class Account
  attr balance:0
  meth transfer(Amt)
    balance:=@balance+Amt
  end
  meth getBal(Bal)
    Bal=@balance
  end
  meth batchTransfer(AmtList)
    for A in AmtList do
      {self transfer(A)}
    end
  end
end
class LoggedAccount from Account
  meth init skip end
  meth transfer(Amt)
    {LogObj addentry(transfer(Amt))}
    Account,transfer(Amt)
  end
end

```

Here is a sample use of these classes:

```

LogObj={New Log init}
LA={New LoggedAccount init}
{LA transfer(50)}
{LA transfer(60)}
local X in {LA getBal(X)} {Show X} end
local L in {LogObj report(L)} {Show L} end
{LA batchTransfer([70 80 90])}
local X in {LA getBal(X)} {Show X} end
local L in {LogObj report(L)} {Show L} end

```

1. In the statement

```
{LogObj addentry(transfer(Amt))}
```

it may appear that the new `transfer` method is calling itself recursively, but that's not the case. Name the syntactic unit that `transfer(Amt)` must parse as in this context: _____.

1 pts

2. Explain why it could not be parsed as a method call instead:

2 pts

3. There are two other lines of the code where the old or new `transfer` method *is* called. One is a static method call and the other is a dynamic method call. Identify them and show which is which by writing "static call" or "dynamic call" in the margin next to the line.

4 pts

4. Suppose you decide you want only individual amount transfers to be logged, but not batch transfers. Show how you could modify the code to make this change in logging behavior. You should change only one line of code. (Indicate the line you are changing by drawing an arrow to it labeled "changed for question 10-4:" followed by the changed line of code.)

2 pts

11 Relational Programming in Oz

Suppose we are given

```

proc {Parent Father Son}
  choice Father=abe Son=bob
  [] Father=abe Son=carl
  [] Father=bob Son=fred
  [] Father=fred Son=george
  [] Father=carl Son=howard
end
end
proc {Ancestor X Y}
  choice {Parent X Y}
  [] Z in {Parent X Z} {Ancestor Z Y}
end
end

```

1. Find all descendants of Bob. That is, write code that uses `Solve` and a for loop to browse all solutions `X` such that

4 pts

```
{Ancestor bob X}
```

-
2. Suppose we wish (unconventionally) to redefine the ancestor relation to be reflexive. E.g., we consider the descendants of George to include George himself, and more generally for any person X we consider the descendants of X to include X . Rewrite the `Ancestor` procedure accordingly.

4 pts

12 Relational Programming in Prolog

Which of the following descriptions apply to the Prolog language? Circle T for true, F for false.

8 pts

1. T F Includes lists among its built-in data types.
2. T F Does not allow new variables to appear in the subgoals of a rule.
3. T F Uses the closed world assumption.
4. T F Implements Horn clause logic.
5. T F Implements a complete logic.
6. T F Implements a non-monotonic logic.
7. T F Is both in theory and in practice purely declarative in the (broad) sense of not requiring procedural information in users' programs.
8. T F Uses a breadth-first search strategy.