

Name: _____

CSCI-4430 Programming Languages

Midterm Exam

This exam consists of 9 sections with questions worth a total of 62 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. The exam must be completed in the regular class time period 12:00–1:50 pm. 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.** You may leave after turning in the exam.

Part I

1 A Grammar for Simple Arithmetic Expressions

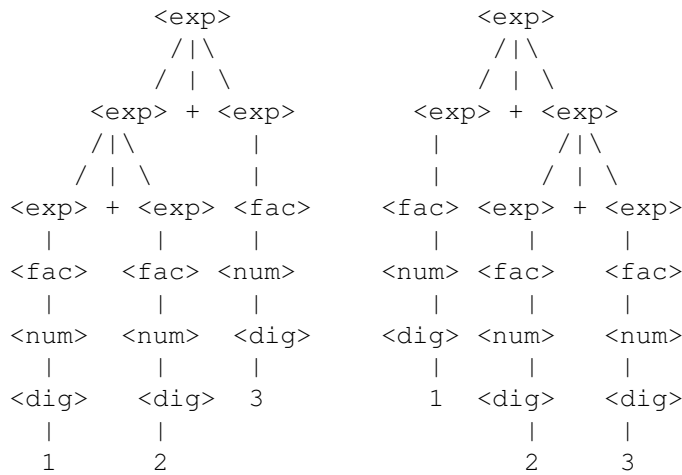
The following is a grammar for simple arithmetic expressions that involve only natural numbers, addition, and multiplication.

$$\begin{aligned}\langle \text{digit} \rangle & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \langle \text{number} \rangle & ::= \{ \langle \text{digit} \rangle \}^+ \\ \langle \text{expression} \rangle & ::= \langle \text{factor} \rangle \mid \langle \text{expression} \rangle + \langle \text{expression} \rangle \\ \langle \text{factor} \rangle & ::= \langle \text{number} \rangle \mid \langle \text{factor} \rangle * \langle \text{factor} \rangle\end{aligned}$$

1. What are the tokens of this language?

2 pts

2. With this grammar, the expression 1+2+3 has two parse trees:



The existence of two parse trees for the same string shows that the grammar is (circle one)

2 pts

- (a) ambiguous
- (b) context-free
- (c) context-sensitive
- (d) left-associative
- (e) right-associative

2 Evaluating Simple Arithmetic Expressions

For *evaluation* of expressions by an interpreter (or for code-generation by a compiler), parse trees are not the most convenient representation, because they still contain information about the original string representation that is unnecessary for computation. The following Oz program is a little expression evaluator that assumes its input is in the form of an *abstract syntax tree* constructed from Oz tuples.

```

fun {Eval X}
  case X
  of int(N) then N
  [] add(X Y) then {Eval X}+{Eval Y}
  [] mul(X Y) then {Eval X}*{Eval Y}
  end
end

```

1. For the first parse tree shown in part 2 of the preceding question, draw the corresponding abstract syntax tree.

4 pts

2. Is the `Eval` function tail-recursive? Explain.

2 pts

3 Some

Here is a definition of a function that, given a list `Xs` and a function `P`, returns `true` if `{P X}` returns `true` for some element `X` of `Xs` and otherwise it returns `false`.

```
fun {Some Xs P}
  case Xs of
    nil then false
  [] X|Xr then
    if {P X} then true else {Some Xr P} end
    %-----
  end
end
```

1. Rewrite the underlined `if` expression using one of Oz's boolean operators instead.

2 pts

2. One of the functions we wrote as a homework exercise was the `Member` function. `{Member Y Xs}` returns `true` if `Y` occurs in list `Xs` and otherwise it returns `false`. Here, instead of writing an independent recursive definition of `Member`, show how to implement it by passing an appropriate function to `Some`.

4 pts

3. We know that Oz function definitions and function calls are linguistic abstractions; they are defined by translation into kernel language procedure definitions and procedure calls. Translate the definition of `Some` into a procedure definition in kernel language syntax. The first part of the translation is given; fill in the rest. (Hint: Be sure your translation reproduces the semantics of the original when there is an error in calling `Some`, with, say, a number instead of a list.)

6 pts

```

Some = proc {$ Xs P B}
      case Xs of
        nil then B=false
      else case Xs of X|Xr then

```

4 An Abstract Machine Execution

The following sequence of abstract machine states is the beginning of an execution of an Oz (kernel language) statement, expressed in a format similar to one we've used in several lab exercises. Continue the execution to termination.

10 pts

<pre> local Y in local X in X=1 local X in X=2 end Y=X end end </pre>
Env: {}
Store: {}

⇒

<pre> local X in X=1 local X in X=2 end Y=X end </pre>
Env: {Y → y}
Store: {y}

⇒

<pre> X=1 local X in X=2 end Y=X </pre>
Env: $\{Y \rightarrow y, X \rightarrow x\}$
Store: $\{y, x\}$

\Rightarrow

<pre> X=1 </pre>
Env: $\{Y \rightarrow y, X \rightarrow x\}$
<pre> local X in X=2 end Y=X </pre>
Env: $\{Y \rightarrow y, X \rightarrow x\}$
Store: $\{y, x\}$

\Rightarrow

Name: _____

CSCI-4430 Programming Languages

Midterm Exam, Part II

5 Arity

What does `{Arity r(b:1 a:2 3:7)}` return?

2 pts

6 Free and Bound Identifiers

List for each of the following Oz statements the free and bound variable identifiers. (Don't forget: an identifier can be both free and bound in the same statement, because one occurrence of it is free and another occurrence is bound.)

6 pts

1. `{P X Y} local X in {X P Y} end`

Free: _____ Bound: _____

2. `local X in local Y in {X Y Z} end end`

Free: _____ Bound: _____

3. `case X of f(Y) then {P Y} else {Q Y} end`

Free: _____ Bound: _____

7 Producing Power Functions

Consider the following variant of the `Pow`-function we've seen in lectures and labs.

```
fun {PowFactory X}
  fun {PowAcc N A}
    if N==0 then A
    else {PowAcc N-1 X*A}
    end
  end
  fun {Pow N}
    {PowAcc N 1}
  end
in
  Pow
end
```

1. Write an expression using `PowFactory` to compute 3 to the 5-th power.

4 pts

2. Give the external references for `POWACC` **2 pts**
3. Give the external references for `POW`. **2 pts**

8 Semantics of exceptions

When an exception is raised in an Oz program by the (kernel language) semantic statement

$$(\text{raise } \langle x \rangle \text{ end}, E)$$

semantic statements are popped off the semantic stack looking for a catch statement.

1. No computation is involved in processing each semantic statement that is popped off, other than checking whether or not it is a catch. In C++, on the other hand, the corresponding way of raising an exception (with a `throw` statement) must process each popped stack frame to call the destructors of all objects referenced in the stack frame. Explain why this is necessary in a C++ program but not in an Oz program. **4 pts**

2. Assuming

$$(\text{catch } \langle y \rangle \text{ then } \langle s \rangle \text{ end}, E_c)$$

is the `catch` statement found, which of the following is the semantic statement pushed on the semantic stack as the last step of processing the `raise` statement? (Hint: Only one of these really makes any sense, if you remember how environments are used in the abstract machine.)

- (a) $(\langle s \rangle, E + \{\langle y \rangle \rightarrow E_c(\langle x \rangle)\})$
- (b) $(\langle s \rangle, E + \{\langle y \rangle \rightarrow E(\langle x \rangle)\})$
- (c) $(\langle s \rangle, E_c + \{\langle y \rangle \rightarrow E(\langle x \rangle)\})$
- (d) $(\langle s \rangle, E_c + \{\langle y \rangle \rightarrow E_c(\langle x \rangle)\})$

4 pts

9 A Translation Attempt

The semantics of the try-finally statement in Oz

```
try <s>1 finally <s>2 end
```

is defined by translating it into a kernel language try-catch:

```
try  
  <s>1  
catch X then  
  <s>2  
  raise X end  
end  
<s>2
```

If <s>2 is a large statement (perhaps composed of dozens of smaller statements), it's a problem that this translation duplicates it. Consider the following alternative translation that uses a boolean variable to keep track of whether an exception has occurred in order to avoid duplicating <s>2:

```
local ExceptionOccurred SavedException in  
  ExceptionOccurred=true  
  try  
    <s>1  
    ExceptionOccurred = false  
  catch X then  
    SavedException=X  
  end  
  <s>2  
  if ExceptionOccurred then  
    raise SavedException end  
  end  
end
```

However, this statement does **not** have the same semantics as the first translation. Why not? Make your answer precise by describing a case where the two statements would execute differently.

6 pts