\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

LAB 11 QUESTIONS

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

- Name: Moti Begna

- NetID: begna002

Answer the questions below according to the lab specification. Write

your answers directly in this text file and submit it to complete the

lab.

Files `buggy\_lpe.ml' and `lpe\_main.ml'

======================================

This lab deals with a lexer, parser, evaluator system for a small

language that includes arithmetic, `let/in' expressions, and

`if/then/else' expressions. `lex\_parse\_eval.ml' is primarily

responsible for this and is divided into 4 sections that handle a

simple arithmetic language with some more programmatic elements. The 4

sections are:

1. Lexer: which converts a character string into a list of tokens.

2. Parser: which converts a list of tokens into an expression tree,

often referred to as a Parse Tree or Abstract Syntax Tree (AST).

3. Evaluator: which analyzes the expression tree and computes a

numeric result.

4. To-string functions: which are used to convert token lists and

parse trees to strings that can be printed.

The functions in `lex\_parse\_eval.ml' are used in the file

`lpe\_main.ml' which takes an expression from the command line and

performs lexing, parsing, and evaluation on it. Here are some

examples though the examples for `if/then/else' won't work until the

lab is completed.

,----

| > ocamlc lex\_parse\_eval.ml lpe\_main.ml

|

| > ./a.out '1'

| Tokens:

| [Int(1)]

|

| Parse Tree:

| IConst(1)

|

| Result:

| Int(1)

|

| > ./a.out 'true'

| Tokens:

| [Bool(true)]

|

| Parse Tree:

| BConst(true)

|

| Result:

| Bool(true)

|

| > ./a.out '1+2'

| Tokens:

| [Int(1); Plus; Int(2)]

|

| Parse Tree:

| Add

| IConst(1)

| IConst(2)

|

| Result:

| Int(3)

|

| > ./a.out '1+2\*3'

| Tokens:

| [Int(1); Plus; Int(2); Times; Int(3)]

|

| Parse Tree:

| Add

| IConst(1)

| Mul

| IConst(2)

| IConst(3)

|

| Result:

| Int(7)

|

| > ./a.out 'if false then 1+2\*3 else 4\*5' # WON'T WORK UNTIL LAB IS COMPLETED

| Tokens:

| [If; Bool(false); Then; Int(1); Plus; Int(2); Times; Int(3); Else ; Int(4);

| Times; Int(5)]

|

| Parse Tree:

| Cond

| .if\_expr:

| BConst(false)

| .then\_expr:

| Mul

| IConst(4)

| IConst(5)

| .else\_expr:

| Add

| IConst(1)

| Mul

| IConst(2)

| IConst(3)

|

| Result:

| Int(20)

|

| > ./a.out 'let x=5 in let y=2 in x\*y'

| Tokens:

| [Let; Ident(x); Equal; Int(5); In; Let; Ident(y); Equal; Int(2); In;

| Ident(x); Times; Ident(y)]

|

| Parse Tree:

| Letin( x )

| .var\_expr:

| IConst(5)

| .in\_expr:

| Letin( y )

| .var\_expr:

| IConst(2)

| .in\_expr:

| Mul

| Varname(x)

| Varname(y)

|

| Result:

| Int(10)

`----

PROBLEM 1: Debugger Basics

==========================

(A)

~~~

Compile the two source files together to create an executable. Make

sure that debug information is turned on via the `-g' switch as in

,----

| > ocamlc -g buggy\_lpe.ml lpe\_main.ml

| File "buggy\_lpe.ml", line 249, characters 5-42:

| Warning 10: this expression should have type unit.

`----

Note that the warning shown is likely to be shown and will be

addressed later in the lab.

Run the program as shown which should generate an exception.

,----

| ./a.out 'let x=5 in x+2'

`----

Show the results of this exception in the terminal which should be a

bit perplexing.

While it might be possible to diagnose the problem simply by examining

the source code carefully, the remainder of this problem will

demonstrate how to use the debugger to gather information that can

lead to a fix.

**Tokens:**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**Fatal error: exception Buggy\_lpe.ParseError("syntax error", \_)**

(B)

~~~

Start the debugger with the compiled program as in:

,----

| > ocamldebug a.out

| OCaml Debugger version 4.xx

|

| (ocd)

`----

Since the `lpe\_main.ml' program must take command line arguments, set

them using the `set arguments <args>' command as in

,----

| (ocd) set arguments 'let x=5 in x+2'

`----

Run the program with the `run' command and copy the results below.

**Tokens:**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**Time: 2375**

**Program end.**

**Uncaught exception: Buggy\_lpe.ParseError ("syntax error", \_)**

(C)

~~~

A major strength of OCaml's debugger is it enables so-called

"time-traveling" allowing both forward and backward steps. A standard

debugger such as is widely available in C and Java allows only forward

steps, not backwards. The technique that enables ocamldebug this is

not conceptually not difficult: it simply saves program state

occasionally as a 'checkpoint' and when one requests backwards

movement, the program position is noted and the program is restarted

from the latest checkpoint to the desired moment in time.

As an end user, this means on hitting an uncaught excepton, one can

simply back up a step to see what is happening using the `backstep'

command. Moving forward is a matter of issuing a `step' command. Do

some backsteps and steps after hitting the exception from the last

part. Paste the results below.

**(ocd) backstep**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**Time: 2374 - pc: 132852 - module Buggy\_lpe**

**178 | \_ -> raise (ParseError {msg="syntax error"; toks=toks})<|a|>**

**(ocd) backstep**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**Time: 2373 - pc: 132828 - module Buggy\_lpe**

**178 | \_ -> <|b|>raise (ParseError {msg="syntax error"; toks=toks})**

**(ocd) step**

**Time: 2374 - pc: 132852 - module Buggy\_lpe**

**178 | \_ -> raise (ParseError {msg="syntax error"; toks=toks})<|a|>**

(E)

~~~

Next set some \*breakpoints\* at functions of interest which will stop

the debugger from executing when reached. If an uncaught exception has

been hit, all of the modules associated with program are already

loaded one can use the name of the function to set the breakpoint as

in:

,----

| (ocd) break Buggy\_lpe.parse\_muldiv

| Breakpoint 1 at 145252: file buggy\_lpe.ml, line 137, characters 3-348

`----

Follow this with the `reverse' command which will run the program in

"reverse" until the break point is hit. After hitting it, run the

`list' and `backtrace' commands and paste the results below.

**(ocd) break Buggy\_lpe.parse\_muldiv**

**Breakpoint 1 at 132092: file buggy\_lpe.ml, line 137, characters 3-348**

**(ocd) reverse**

**Tokens:**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**Time: 2366 - pc: 132092 - module Buggy\_lpe**

**Breakpoint: 1**

**137 <|b|>let rec iter lexpr toks =**

**(ocd) list**

**127 | Minus :: rest -> (\* found - \*)**

**128 let (rexpr,rest) = parse\_letin rest in (\* consume a higher-prec expression \*)**

**129 iter (Sub(lexpr,rexpr)) rest (\* create a Sub tree and iterate again \*)**

**130 | \_ -> (lexpr, toks)**

**131 in**

**132 let (lexpr, rest) = parse\_muldiv toks in (\* create the initial left expression \*)**

**133 iter lexpr rest (\* start iterating \*)**

**134**

**135 (\* parse multiplication and division, same principle as parse\_addsub \*)**

**136 and parse\_muldiv toks =**

**137 <|b|>let rec iter lexpr toks =**

**138 match toks with**

**139 | Times :: rest ->**

**140 let (rexpr,rest) = parse\_ident rest in**

**141 iter (Mul(lexpr,rexpr)) rest**

**142 | Slash :: rest ->**

**143 let (rexpr,rest) = parse\_ident rest in**

**144 iter (Div(lexpr,rexpr)) rest**

**145 | \_ -> (lexpr, toks)**

**146 in**

**147 let (lexpr, rest) = parse\_ident toks in**

**(ocd) backtrace**

**Backtrace:**

**#0 Buggy\_lpe buggy\_lpe.ml:137:3**

**#1 Buggy\_lpe buggy\_lpe.ml:132:40**

**#2 Buggy\_lpe buggy\_lpe.ml:115:32**

**#3 Lpe\_main lpe\_main.ml:30:39**

Breakpoints from the beginning of Programs

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

When running from the beginning of a program, not all modules are

loaded so functions may not be identifiable by name. In this case,

breakpoints at specific source lines like the below where 136 is the

source line corresponding to the `parse\_muldiv' function.

,----

| (ocd) break @ Buggy\_lpe 136

| Loading program... done.

| Breakpoint 1 at 145252: file buggy\_lpe.ml, line 137, characters 3-348

`----

PROBLEM 2: Debugging the Program

================================

(A)

~~~

After positioning the debugger as indicated above, print the `toks'

variable then use the `step' command to step through the execution of

`parse\_muldiv'.

- What token is at the front of the token list?

- Does the control flow seem to make sense for parsing a the tokens at

the beginning of the token list?

- After examining the source code, is the function call sequence

appropriate or should changes be made?

- Describe the bug that is in the source code and how to fix it.

**The let token is at the front of the token list. Changes need to be made to the source code, specifically in lines 147. Here, parse\_letin should be called, but instead parse\_ident is being called which is not the next higher precedent function.**

(B)

~~~

After identifying the bug in the parser, make a change to fix it. Quit

the debugger (via `quit'), recompile, re-run the program on the

problem input to show some progress has been made. Paste your results

below and state how the results differ.

**Tokens:**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**Letin( x )**

**.var\_expr:**

**IConst(5)**

**.in\_expr:**

**Add**

**Varname(x)**

**IConst(2)**

**Result:**

**Fatal error: exception Buggy\_lpe.EvalError("No variable 'x' bound")**

**This time, there is no syntax error and the parse tree is allowed to print. However, there is an evaluation error of “No variable ‘x’ bound.**

(C)

~~~

Start the debugger again, set the arguments, and run to the uncaught

exception. Backstep and then use `list' to show the source position.

The trouble at this point should be apparent: something is wrong with

the `varmap'. Unfortunately, the debugger is not equipped to print

standard Maps as evidenced by failed attempt to print:

,----

| > ocamldebug a.out

|

| (ocd) set arguments 'let x=5 in x+2'

| (ocd) run

| Uncaught exception: Buggy\_lpe.EvalError "No variable 'x' bound"

|

| (ocd) backstep

|

| (ocd) print varmap

| varmap: varval\_t Varmap.t = <abstr>

`----

This output is hidden behind an abstraction barrier. One can install

"printers" which will display data for maps and other more customized

types but this is beyond the scope of the lab.

Rather, turn your attention to the warning that has been issued on

every compilation. Examine the associated code and describe what is

being done wrong.

**When pattern matching to let in from lines 247-250, we must add the variable name and data to our varmap. However, since this function is immutable, we have to catch the return and set it to a variable. This is not being done in the code, and varmap is instead being treated as a mutable data type.**

(D)

~~~

Fix the evaluation error, recompile, and show that the trouble code

now evaluates correctly. Show your corrected lines of code in

`buggy\_lpe.ml' and the results of running on the input.

**Tokens:**

**[Let; Ident(x); Equal; Int(5); In; Ident(x); Plus; Int(2)]**

**Parse Tree:**

**Letin( x )**

**.var\_expr:**

**IConst(5)**

**.in\_expr:**

**Add**

**Varname(x)**

**IConst(2)**

**Result:**

**Int(7)**

Optional Extras

===============

IDENTICAL TO LAB 11

Currently the lexer/parser/evaluator does not handle numeric

comparisons to produce boolean results such as

,----

| 5 < 2 -> Bool false

| if 1+2 > 0 then 8 else 4 -> Int 8

`----

This will be a required part of the final assignment interpreter so it

would be an excellent exercise to extend the system to handle these

new expression types.

- Extend the lexer to include < and >. The = sign is already part of

the lexer.

- Extend the expression type to include comparison expressions for

Less, Greater, Equal with constituent left/right expressions (like

arithmetic).

- Extend the parser functions with a new function to parse

comparisons. This should occur at a lower precedence than

arithmetic.

- Extend the evaluator to include evaluation cases for

comparisons. These should check that their left/right expressions

are integers, do the appropriate comparison on the numbers, and

return a Bool. You may wish to model them after the arithmetic

evaluation code.