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

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 `lex\_parse\_eval.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: Lexer and Parser

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

(A)

~~~

In addition to arithmetic, the lexer/parser/evaluator understands two

additional features

1. `if/then/else' constructs for conditional execution

2. `let/in' expressions as `let x=1+2 in x\*7' for binding names to

values

Examine the first section of `lex\_parse\_eval.ml' which contains the

lexer. Explain what tokens exist for the keywords like `let' and `if'

and how the lexer creates these tokens versus variable name `Ident'

tokens.

**Once a character substring is created which represents a possible letter or word, the substring is matched with a possible keyword such as let, if, then, etc. If the substring matches one of these keywords, the associated value for that keyword is returned. If the substring is not one of those keywords, an identifier for that substring is instead returned.**

(B)

~~~

Examine the second section of `lex\_parse\_eval.ml' which contains the

parser. Examine the `expr' type which represents the tree-like

structure of parsed expressions. Describe the new entries in this

type that correspond to `if/then/else' and `let/in'

constructs. Describe their parts and whether expression trees will

always be binary trees.

**Both the let/in and the if/then/else entries are record types that hold corresponding data. The let/in record holds a string variable name, expression that will be bound to the variable name, and an expression that follows the “in” format of a let binding. The if/then/else record holds an expression that will be checked (bound to if\_expr), an expression that will be executed if if\_expr is true, and an expression that will be executed if if\_expr is false. Expression trees that utilize these entries will no longer be strictly binary, but have segments that incorporate three children instead of just a left and right child.**

(C)

~~~

The parser is somewhat more complex then previous versions but has

many of the same features in that it is comprised of a series of

mutually recursive functions. However some cosmetic differences are

immediately apparent.

- The names of the parsing elements do not mention their precedence

but instead name the kind of element they handle such as `parse\_add'

and `parse\_cond'. Some of these such as `parse\_add' handle more than

`Add' tokens but this should be easy to interpret.

- The parsing functions, starting with `parse\_expr' are shown in

source code "top-down" with lower-precedence `parse\_add' coming

before higher-precedence `parse\_mul'.

Examine the parsing functions carefully and answer the following

questions.

1. Is the parsing of `let/in' and `if/then/else' expressions higher or

lower precedence than adding and multiplying?

2. Functions like `parse\_add' make recursive calls to themselves to

try to parse more additions. Is this what `parse\_letin' and

`parse\_cond' do? Why or why not?

3. Functions like `parse\_add' first attempt to call higher-precedence

parsing functions like `parse\_mul'. They then use the results in

an addition. Is the same done by `parse\_letin' and `parse\_cond'?

Why or why not?

1. **The parse\_letin and parse\_cond expressions are of higher precedence than adding and multiplying.**
2. **Parse\_letin and parse\_cond do not make recursive to themselves, but rather calls to the top-level function parse\_expr because the record values that will be assigned to these variant types can be full expressions as well as simple identifiers.**
3. **The same is not done, since these expressions must pattern matched inputted tokens immediately. This is because if they were to first call the higher precedence function parse\_ident, a syntax error would occur.**

PROBLEM 2: Evaluator

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

(A)

~~~

Examine the third section of `lex\_parse\_eval.ml' which contains the

Evaluator. This portion defines types and functions relevant to

walking through an expression tree generated by the parser to evaluate

an answer.

The first few lines of the evaluator lay out a type `varval' for

results and create a `varmap\_t' type to map string names. Answer the

following questions about this section.

1. Describe the kinds of value that can result for evaluation or be

let-bound via `let/in' bindings.

2. How is OCaml's standard library used to easily derive functions for

adding and looking up variable bindings in the `varmap\_t'? What

standard functor is used?

3. Will the variable maps be mutable or immutable?

1. **The valval\_t type can only have two variants, an Integer and a Boolean value.**
2. **Map.make is used with a String module input to create a Varmap module that will have functions like add, remove, and find.**
3. **The variable maps will be immutable since Map.make is creates immutable maps**

(B)

~~~

Examine the `eval\_expr' function which is where most of the work of

the evaluator is performed. Answer the following questions.

1. What two arguments does `eval\_expr' take? What types are they as

inferred by looking through the rest of the code?

2. What action is taken when a `Varname' expression is found and what

error can result?

3. Inspect how the different arithmetic operators are

handled. Describe how the common task of evaluating the left/right

child expressions is handled while still performing appropriate

arithmetic operations.

4. Analyze the `Letin' case within `eval\_expr'. Describe how a new

binding is created in a `let/in' expression.

1. **Evl\_expr takes varmap, which is a variable map of type varmap\_t, and an expression tree of type expr.**
2. **The value associated with the variable name is searched in varmap using the Varmap.find\_opt function, and then matched. If the value is found, it is returned. Otherwise, a “No variable..” error is raised.**
3. **The arithmetic expressions are all evaluated the same, where the left and right sub-expressions are recursively evaluated, whereby another case of pattern matching is used when we have found left and right data. At this point, we once again use matching to evaluate the left and right data based on what the original expression was (Add, Sub, Mul, Div).**
4. **The var\_expr child of the Letin expression is evaluated and then added to a new map using it as the value, and the var\_name child of the Letin expression as the key. The top level eval\_expr function is then called using the new map and the in\_expr child of the Letin expression as the rest of the expression to be evaluated.**

(C)

~~~

The code for the `Cond' case is not complete so `if/then/else'

expressions will not evaluate yet. Fill in the gaps marked `EDIT ME'

to complete this code.

- Evaluate the `c.if\_expr' to determine if the test is true or false

- When b is true, evaluate c.then\_expr

- When b is false, evaluate c.else\_expr

Paste your code below as your solution.

**| Cond(c) -> begin**

**let test = eval\_expr varmap c.if\_expr in**

**match test with**

**| Bool b ->**

**if b = true then**

**eval\_expr varmap c.then\_expr**

**else**

**eval\_expr varmap c.else\_expr**

**| \_ ->**

**let msg = sprintf "Expected Bool for if <expr>, found '%s'"**

**(data\_string test) in**

**raise (EvalError{msg})**

**end**

Optional Extras

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

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.