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

LAB 10 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.

PROBLEM 1: lexer.ml

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

(A)

~~~

Examine `lexer.ml'. The central function of interest is `lex\_string'

which is intended to process a string with an arithmetic expression in

it and produce a list of tokens of the `token' type defined at the top

of the file.

Study the given code carefully and answer the following questions on

how it works.

1. The code makes use of recursion. Describe how recursion is used to

process the entire string and to produce a list of the tokens that

are found.

**The base case of the recursion is when the pointer “pos” is >= the length of the string “len”. At this point, an empty list is returned. If the base case is not met, then patter mattern matching is used on every character of the string to determine it’s type. Whitespace is ignored.**

2. There is a loop associated with digits. Why is this needed while no

loop is needed for the other cases of tokens?

**It is possible for the integer to be greater than one digit (in this case, greater than one character). Thus, we must loop until we have found the last digit in the sequence, get the substring that holds the number, parse it to an integer, and then add it to the list of lexed tokens.**

3. Demonstrate some uses of `lex\_string' in a REPL to show that it

works.

**# lex\_string "5 + 5\*2";;**

**- : token list = [Int 5; Plus; Int 5; Times; Int 2]**

**# lex\_string "2\*(5 +(5\*2))";;**

**- : token list =**

**[Int 2; Times; OParen; Int 5; Plus; OParen; Int 5; Times; Int 2; CParen;**

**CParen]**

(B)

~~~

Extend the code in `lexer.ml' to account for the following new kinds

of tokens.

1. Subtraction with the Minus (-) sign - like Addition

2. Division with the Slash (/) - like multiplication

3. Identifiers of strings (abbreviated Ident) - like Int of integers.

To complete the extension, make the following three modifications.

1. Add in a variant to the `token' type for each of Minus, Slash,

Ident.

2. In `lex\_string', extend the `match/with' single character cases to

include one for - and / associated with Minus and Slash. These will

behave similarly to their Plus and Times counterparts.

3. Add in a case for character data which will become Ident cases and

be similar to Int. Make use of the provided `is\_letter' function to

help identify letters.

Once you have completed the modifications, uncomment the provided

examples at the end of the file to ensure the modifications compile

and work. Try out the `lex\_string' in a REPL.

**val actual1 : token list =**

**[Plus; Times; OParen; CParen; Int 123; Minus; Slash; Ident "abc"]**

**val expect1 : token list =**

**[Plus; Times; OParen; CParen; Int 123; Minus; Slash; Ident "abc"]**

**val ok1 : bool = true**

**val actual2 : token list =**

**[OParen; Ident "a"; Plus; Ident "b"; Plus; Ident "c"; CParen; Slash;**

**OParen; Int 9; Times; Ident "d"; Minus; Int 10; Slash; Int 4; CParen]**

**val expect2 : token list =**

**[OParen; Ident "a"; Plus; Ident "b"; Plus; Ident "c"; CParen; Slash;**

**OParen; Int 9; Times; Ident "d"; Minus; Int 10; Slash; Int 4; CParen]**

**val ok2 : bool = true**

PROBLEM 2: `parser.ml'

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

(A)

~~~

Examine `parser.ml'. This file contains a solution for Problem 1's

lexer as it is required for the parser (this is also useful if you get

stuck).

The central function of interest is `parse\_tokens' which converts a

list of `tokens' into a tree structure which will be referred to as a

Parse Tree or Abstract Syntax Tree (AST). This tree is comprised of

variants of the `expr' type defined midway through the file.

Describe the kinds of elements in the `expr' type and what kinds of

data they carry. Note how this creates a tree structure.

**The elements are tuples that carry two expressions (left and right values that can be any token variant type), and also a constant variant type that carries an integer value.**

(B)

~~~

The code for `parse\_tokens' uses recursion to perform the

parsing. Notice its structure of dividing the task into 4 mutually

recursive functions:

1. prec0 which handles Int and OParen/CParen tokens.

2. prec1 which handles Times tokens

3. prec2 which handles Plus tokens

4. parse which serves as an entry point for the beginning of parsing.

Answer the following questions about how each of these work.

1. Functions `prec2' starts by calling `prec1' and prec1 starts by

calling prec0. In both situations, a pair of `(lexpr, rest)' are

returned. What types are the `lexpr' and `rest' and how are they

used?

**lexpr is the left expression which can be any expr variant type, while rest is the tail part of the input list. lexpr is used as the left value for some expr variant type, and rest is pattern matched to find the next token. If the next token is an algebraic expression, then the right value “rexpr” is found and used for some expr variant type.**

2. `prec0' matches in one case the `OParen' token which corresponds to

`(' in the input. This is handled by calling the `parse'

function. Describe what effect this has in parsing. What happens if

there is no `CParen' (closing paraenthesis `)' ) in the input?

**Parse is called until whatever expression in the parenthesis is finished, at which point a closing parenthesis must be used to continue the recursion for the rest of the string. If Cparen is not matched, then a ParseError will occur, with the “unclosed parentheses” error.**

3. Try running `parse\_tokens' on the following inputs as follows and

report your results. Each of these will raise an error. How the

code reaches this error.

,----

| # parse\_tokens (lex\_string "5+10+");;

| # parse\_tokens (lex\_string "5+10+11 )");;

| # parse\_tokens (lex\_string "5\*10+11 9");;

| # parse\_tokens (lex\_string "5+10+11 + +");;

| # parse\_tokens (lex\_string "5\*(10+11");;

`----

**# parse\_tokens (lex\_string "5+10+");;**

**Exception: ParseError {msg = "expected an expression"; toks = []}.**

**# parse\_tokens (lex\_string "5+10+11 )");;**

**Exception: ParseError {msg = "tokens remain in stream"; toks = [CParen]}.**

**# parse\_tokens (lex\_string "5\*10+11 9");;**

**Exception: ParseError {msg = "tokens remain in stream"; toks = [Int 9]}.**

**# parse\_tokens (lex\_string "5+10+11 + +");;**

**Exception: ParseError {msg = "syntax error"; toks = [Plus]}.**

**# parse\_tokens (lex\_string "5\*(10+11");;**

**Exception:**

**ParseError {msg = "unclosed parentheses"; toks = [Int 10; Plus; Int 11]}.**

**Prec0 can raise 3 errors: if there is no integer after some algebraic expression, if there isn’t a closing parenthesis,and if there are 2 algebraic expressions in a row. A final function in parse\_tokens is also used to determine if there are tokens remaining the list that are not a part of the algebraic expression.**

(C)

~~~

Modify the parser to accommodate the new tokens - and / and string

identifiers. Add in the following variants type to the `expr' type.

- `Sub' which corresponds to the `Minus' token and is similar to `Add'

- `Div' which corresponds to the `Slash' token and is similar to `Mul'

- `Varname' which corresponds to the `Ident' token and is similar to

`Const'

Note that each of these has analogues already in the parser.

- Handle `Minus' tokens in the `prec2' function. It should create a

`Sub' expression similar to `Add'

- Handle `Slash' tokens in the `prec1' function. It should create a

`Div' expression similar to `Mul'.

- Handle `Ident' tokens in the `prec0' function. It should create a

`Varname' expression similar to `Int'.

When you complete the parser extensions, uncomment the code at the end

of the file which allows you to test it in a REPL.

**val lexed1 : token list =**

**[OParen; Ident "a"; Plus; Ident "b"; Plus; Ident "c"; CParen; Slash;**

**OParen; Int 9; Times; Ident "d"; Minus; Int 10; Slash; Int 4; CParen]**

**val parsed1 : expr =**

**Div (Add (Varname "a", Add (Varname "b", Varname "c")),**

**Sub (Mul (Const 9, Varname "d"), Div (Const 10, Const 4)))**

**val expect1 : expr =**

**Div (Add (Varname "a", Add (Varname "b", Varname "c")),**

**Sub (Mul (Const 9, Varname "d"), Div (Const 10, Const 4)))**

**val ok1 : bool = true**

**val lexed2 : token list =**

**[Int 1; Slash; Ident "weight"; Plus; Minus; Ident "hello"]**

**Exception: ParseError {msg = "syntax error"; toks = [Minus; Ident "hello"]}.**