Homework 7  
CSE 4102 Homework 7, Spring 2016  
William Dickson  
05/01/2016  
Section: 001  
Instructor: Jeffrey A. Meunier

# Introduction:

# In this assignment I wrote a Prolog program to evaluate constructs that resemble those from a functional language. This evaluator was written in a recursive manner: there is an eval predicate that evaluates expressions and yields their values. For most expressions, the value of the expression depends on the values of the sub-expressions, which are evaluated recursively.

# Output:

liam@Liams-MacBook-Air:[{...}/CSE4102/hw7]$ gprolog

GNU Prolog 1.4.4 (64 bits)

Compiled Apr 9 2016, 13:47:50 with clang

By Daniel Diaz

Copyright (C) 1999-2013 Daniel Diaz

| ?- [eval].

compiling {...}/CSE4102/hw7/eval.pl for byte code...

{...}/CSE4102/hw7/eval.pl compiled, 88 lines read - 8400 bytes written, 8 ms

(2 ms) yes

| ?- eval(+(4,-1), Result).

Result = 3

yes

| ?- eval(fun(x, +(20, x)), Foo), eval(app(Foo, 10), Result).

Foo = clos(x,20+x,[])

Result = 30

yes

| ?- eval(disp(+(10, 20)), Result).

10+20

Result = 10+20

yes

| ?- eval(fun(x, +(20, x)), Foo).

Foo = clos(x,20+x,[])

yes

| ?- eval(if(true, 1, 0), Result).

Result = 1

(1 ms) yes

| ?- eval(if(false, 1, 0), Result).

Result = 0

yes

| ?- eval(def(x, 100), [], \_, EnvX), eval(x, EnvX, Result, Env).

Env = [x=100]

EnvX = [x=100]

Result = 100

yes

| ?- eval(let([[x,2],[y,5]], +(x,y)), [], R, Env).

Env = []

R = 7

yes

| ?- eval(seq([+(1,1), +(1,5)]), [], Result, Env).

Env = []

Result = 6

yes

| ?-

# Source Code:

/\* Creates a new environment which is stored

as an empty list \*/

env\_new([]).

/\* Binds a value to a name in the environment and

return this in the last parameter \*/

env\_bind(Env, Name, Value, Env\_) :- Env\_ = [Name=Value|Env].

/\* Looks for a value for a given name in the environment and

returns the value or throws a name\_not\_bound exception \*/

env\_lookup([], Name, \_) :- throw(name\_not\_bound(Name)).

env\_lookup([Name=Value|\_], Name, Value) :- !.

env\_lookup([\_|Env], Name, Value) :- env\_lookup(Env, Name, Value).

% evaluator predicates

% +(X, Y) returns the sum of X and Y

eval(+(X, Y), Env, Return, ReturnEnv) :- !,

eval(X, Env, EvalX, EnvX),

eval(Y, EnvX, EvalY, ReturnEnv),

Return is EvalX + EvalY.

% app(Fun, Arg) applys arguments to a function

eval(app(clos(Param, Body, ClosEnv), Arg), Env, Return, Env) :- !,

eval(Arg, ClosEnv, ArgVal, ArgEnv),

env\_bind(ArgEnv, Param, ArgVal, BindEnv),

eval(Body, BindEnv, Return, \_).

eval(app(Fun, \_), \_, \_, \_) :- throw(not\_an\_abstraction(Fun)).

/\* disp(Expr) evaluates the expression and prints the evaulated

value, the returned values are the unmodified input values \*/

eval(disp(Expr), Env, Expr, Env) :- !,

print(Expr).

% fun(Param, Body) creates a closure from a function

eval(fun(Param, Body), Env, clos(Param, Body, Env), Env) :- !.

% if(Cond, Conseq, Alt) evaluates an if, then, else

eval(if(Cond, Conseq, Alt), Env, Return, ReturnEnv) :- !,

eval(Cond, Env, Boolean, CondEnv),

(Boolean -> eval(Conseq, CondEnv, Return, ReturnEnv) ;

eval(Alt, CondEnv, Return, ReturnEnv)).

/\* def(Name, Expr) defines a new binding for a name to an expression

(that must be evaluated before binding) \*/

eval(def(Name, Expr), Env, Return, ReturnEnv) :- !,

eval(Expr, Env, Return, ExprEnv),

env\_bind(ExprEnv, Name, Return, ReturnEnv).

/\* let(Bindings, Body) adds bindings to an environment, evaluates the

body with the environment, then returns the result with the original

environment \*/

eval(let(Bindings, Body), Env, Return, Env) :- !,

eval\_let(Bindings, Env, LetEnv), % eval\_let helper function below

eval(Body, LetEnv, Return, \_).

/\* seq(Exprs) evaluates a list of expressions and returns the final

return value as well as the modified environment. \*/

eval(seq([]), Env, nothing, Env) :- !.

eval(seq([Expr|[]]), Env, Return, ReturnEnv) :- !,

eval(Expr, Env, Return, ReturnEnv).

eval(seq([Expr|Tail]), Env, Return, ReturnEnv) :- !,

eval(Expr, Env, \_, ExprEnv),

eval(seq(Tail), ExprEnv, Return, ReturnEnv).

% special cases for built-in control predicates (from documentation)

eval(true, Env, true, Env) :- !.

eval(false, Env, fail, Env) :- !.

eval(repeat, Env, repeat, Env) :- !.

eval(fail, Env, fail, Env) :- !.

/\* evaluate atoms and return the bound value to the environment if it

is successful \*/

eval(X, Env, V, Env) :- atom(X),

!,

env\_lookup(Env, X, V).

% catch-all

eval(X, Env, X, Env).

% alias for simplicity

eval(Expr, Result) :- eval(Expr, [], Result, \_).

/\* let helper adds bindings to the environment until it is out of bindings

to add \*/

eval\_let([], Env, Env) :- !.

eval\_let([[Name|Expr]|Tail], Env, ReturnEnv) :- !,

eval(Expr, Env, ExprVal, EvalEnv),

env\_bind(EvalEnv, Name, ExprVal, BindEnv),

eval\_let(Tail, BindEnv, ReturnEnv).