Problems

1. (20 points) Given the translational semantics discussed in class, prove that the translation for arithmetic expressions is correct.

2. Given the program below and the semantic definition 'sem.pl' from the course website

```lisp
assign(i,1) seq
assign(z,0) seq
whiledo(le(i,n),
    assign(i,add(i,1)) seq
    assign(z,add(z,2)))
```

with \( \text{pre}(Q) = \text{lookup}(n, Q, vn) \) and \( \text{post}(T) = \text{lookup}(n, T, vn) \land \text{lookup}(z, T, 2 \ast vn) \).

(a) (10 points) Find the loop invariant.
(b) (10 points) Prove the partial correctness of this program for all values of \( n \).

3. Here you will investigate the semantics of the functional programming language Lisp using Prolog. Functional programming languages are side effect free. Lisp is an interesting language since everything in the language’s syntax is expressed as a list, hence the name Lisp as in List Processing. For example, the multiplication of two numbers is expressed as the list

\[
(* 2 4)
\]

This is called an \textit{s-expression}. An s-expression is a list where the first element of the list is either an operator or a function name. Programs are composed \textit{s-forms} which are also lists but only certain names are allowed to appear as the first elements of these s-form lists. Here is a simple program:

```lisp
(defun inc (i) (+ i 1))
(defvar x 0)
(setq x (inc x))
```

This program defines a function 'inc' with the argument list \( (i) \) and the function body \( (+ i 1) \). It then declares the variable \( x \) and initializes it to 0. Finally, it increments the variable \( x \) by calling function 'inc' and assigns the value to \( x \).

A quick tutorial of the language is

http://www.cs.sfu.ca/CC/310/pwfong/Lisp/1/tutorial1.html

Here we define a limited version of Lisp with the following syntax:
Unit ::= [ SFormList ] | []

SFormList ::= SForm | SForm, SFormList

SForm ::= [ defvar, x, SExp ]
  | [ setq, x, SExp ]
  | [ defun, f, [ FL ], SExp ]
  | [ defun, f, [ ], SExp ]

SExp ::= n
  | x
  | true
  | false
  | [ add, SExp, SExp, ... ]
  | [ sub, SExp, SExp ]
  | [ mult, SExp, SExp, ... ]
  | [ eq, SExp, SExp ]
  | [ le, SExp, SExp ]
  | [ neg, SExp ]
  | [ and, SExp, SExp, ... ]
  | [ or, SExp, SExp, ... ]
  | [ if, SExp, SExp, SExp ]
  | [ let, x, SExp, SExp ]
  | [ f ]
  | [ f, SExp, ... ]

FL ::= x, FL
  | x

Here we replaced the standard Lisp lists \((A B C \ldots)\) with Prolog lists \([A, B, C, \ldots]\) for easier processing. We see that a program is a list of s-forms and s-forms are composed of s-expressions. It is worthwhile to point out that operations such as \(+\) and \(*\) in Lisp have arbitrary arities greater than 2, that is, \((+ a b c d e)\)

will simply sum all the arguments \(a\) through \(e\). In the syntax defined above we modeled this by writing

\[[ add, SExp, SExp, \ldots ]\]

indicating that we need at least two arguments but there might be more.

The informal semantics is defined as follows:

defvar - declare a variable in the current environment and initialize it with the value of the SExp.

setq - assign the value of SExp to a declared variable.

defun - declare a function with a possibly empty formal argument list and a function body SExp.

n,true,false - constant.

x - a variable dereference within an s-expression.

add,mult,and,or - operators that need at least two operands whose semantics are the obvious interpretations, add adds the values of the operands, mult multiplies the values of the operands, and applies the logical and operation to all of its operands, and or applies the logical or operation to all its operands.

eq - returns true if its operands evaluate to equal values otherwise it returns false.
le - returns true if the first operand evaluates to a values less or equal to the value of the second operand, otherwise false.

neg - computes the logical not on the value of its operand.

if - if the first s-expression evaluates to true in the current state then if will evaluate the second s-expression in the current state otherwise it will evaluate the third s-expression in the current state.

let - creates a new local state by extending the current state with the declaration of the variable which is initialized with the first s-expression. The second s-expression is evaluated in this new local state and its value is returned as the return value for the let-expression.

f - is a function call with a possibly empty actual parameter list. If the parameter list is not empty then the call will create a new local state, bind the actual parameters to the formal parameters and then evaluate the function body. The value of the function body is returned as the value of the function call.

Please answer the following questions:

(a) (30 points) Create a semantic definition for this language.

(b) (10 points) Show that your semantic definition works by executing the following programs

1. [[defun, inc, [i], [add, i, i]], [defvar, x, 0], [setq, x, [inc, x]]]
2. [[defun, fact, [i], [if, [eq, i, 1], 1, [mult, i, [fact, [sub, i, 1]]]]], [defvar, x, [fact, 3]]]

(c) (10 points) Prove that [add, a, b] ~ [add, b, a]