Assignment 2
Syntax, Semantics and Static Type Checking

Submission instructions:

1. Each task has clear instruction indicating in which file you answers and code must be written.
2. Submit an archive file named ***id1_id2.zip*** (which contains three files task2.rkt, task5.rkt and ex2.pdf) where *id1* and *id2* are the IDs of the students responsible for the submission (or ***id1.zip*** if there is one student in the group).
3. A contract must be included for every implemented procedure.
4. Make sure the .rkt file is in the correct format (see the announcement about "WXME format" in the course web page).
5. Use exact procedure and file names, as your code will be tested automatically.

Task 1: Applicative and Normal Order

With the following expression, one can identify in which way the interpreter he works with evaluates expressions (normal/applicative)

```
(define normal?
  (lambda()
    (let ((e (display 'not-)))
      (display 'normal))))
```

1. Evaluate the expression `(normal?)` according to the ***normal*** and to the ***applicative*** evaluation algorithm. Remember that `display` is a primitive procedure that returns the value `Void`. Show all steps of the two evaluations.

2. Is it possible to write a similar procedure such that if the interpreter is applicative, the evaluation of the procedure displays “applicative”, otherwise it displays “not-applicative”? If it is possible write a procedure that does that, otherwise write an explanation why it’s not possible.

3. Is there a Scheme expression (in the sublanguage we have learned so far) which will terminate (and won’t return error) under both applicative order and normal order evaluations and return different results? If there is, give an example. Otherwise, explain.

Write your answers in *ex2.pdf*.

Task 2: Higher order functions

Part 1

Write the function `derive` that receives a function as a parameter and returns a function which given a number computes the derivative of the original function in that point.

The definition of the derivative of a function is $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$. Use the definition with h=0.001 to calculate the derivative. Complete the code and contract in *task2.rkt* file. (A similar procedure is discussed in the course book in Chapter 1.)

Part 2

One way to compute the square root of a number is to use Newton's method of successive approximations. Given a number x, this method uses an initial guess, denoted by y, and by averaging y with $\frac{x}{y}$ we get a guess closer to the square root. Continuing the iterations we get closer and closer guesses until we get a good approximation to the square root. This code finds an approximate square root:

```
; Type: Number -> Number
; Purpose: compute the square root of a number using Newton's method
; Signature: sqrt(x)
; Precondition: x >= 0
; Test: (sqrt 4.0) ~ 2.0
(define sqrt
  (lambda (x)
    (letrec ((iter (lambda (guess)
                     (if (good-enough? guess x)
                         guess
                         (iter (improve guess x)))))
             (good-enough? (lambda (guess x)
                             (< (abs (- (* guess guess) x)) 0.001)))
             (improve (lambda (guess x)
                        (/ (+ guess (/ x guess)) 2))))
      (iter 1.0))))
```

This example is a specific instance of the more general Newton method to find roots of a function. Write the function `root` that receives a function f and returns one of its **roots**, i.e., returns a value x such that $(f\ x) \approx 0.0$. Read the description of the general Newton method here (http://en.wikipedia.org/wiki/Newton%27s_method) to understand how to improve the guess at each step. Use the initial value to start the iteration to be 1.0 and use the function `good-enough?` provided in the file as your stopping criteria. Complete the code and contract in *task2.rkt* file.

Task 3: Type checking and inference

Part 1- Axiomatic type inference

a. The application rule for **let** we saw in class is given below:

```
For every: type environment _Tenv,
  variables _v1, _v2, ..., _vn, n>=0,
  expressions _e1, _e2, ..., _en,
  expressions _b1, _b2, ..., _bm, m>=1, and
  type expressions _S1, ...,_Sn, _U1, ...,_Um :
If _Tenv |- _e1:_S1,
   _Tenv |- _e2:_S2,
   _Tenv |- _e3:_S3,
   ...,
   _Tenv |- _en:_Sn,
   _Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _b1:_U1,
   _Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _b2:_U2,
   ...,
   _Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _bm:_Um
Then _Tenv |- (let ((_v1 _e1) (_v2 _e2) ... (_vn _en))
                _b1 _b2 ... _bm):_Um
```

Add the application rule for **letrec**.

Remember that **letrec** has the same syntax as **let**, but the values of expressions `e1…en` bound to variables `v1…vn` are evaluated in the scope of `v1…vn`.

Write your answer in ex2.pdf.

b. Derive the type for the following expression using the type-derivation algorithm:

```
(letrec ((foo (lambda (f)
                  (goo foo f)))
         (goo (lambda (foo f)
                  (lambda()(foo f)))))
   (foo (lambda (x) x)))
```

For each step indicate:
- which Rule is used,
- the base typing-statements,
- type-substitution applied,
- and whether Monotonicity was used.

Write your answer in ex2.pdf.

Part 2- Type inference using type constraints

1) In Section 2.3.3.5 of the Course Book, we introduce rules to generate type equations for different types of Abstract Syntax expressions: Numbers, Booleans, Symbols, Primitive Procedures, Application and Lambda expressions. Add rules to generate type equations for *Define* and *Letrec* expressions. Explain your rules.

2) Determine if the following expressions are well-typed and what is their type with the type constraints system. Each well-typed expression can be used in the expressions that follow it. Use the equations you wrote in the previous part. Show all steps and write your answers in *ex2.pdf*.

```
a) (define f
      (lambda (n)
         (letrec ((g (lambda (n)
                          (if (< n 0.001)
                              n
                              (g (/ n 2))))))
             g)))
b) ((f 3) 4)
```

Part 3

Provide answers in ex2.pdf:

a. Consider the type inference algorithm discussed in class (Book Section 2.3.3). Give three reasons that explain why the type derivation of a Scheme expression can fail. Give an example of each case.

b. Why do we need a rule for "define" when we know its type in advance? (See Book Section 2.3.3.2)

Task 4: Procedure Currying and Partial Evaluation

Consider the following 4 definitions of the functions compose and compose-n (similar functions were discussed in Practical Session 2).

```
; Type: [[T1 -> T2] * [T2 -> T3] -> [T1 -> T3]]
; Purpose: Compute the composition of 2 functions
; Signature: compose(f1, f2)
; Test: ((compose add1 add1) 1) -> 3
(define compose
  (lambda (f1 f2)
    (lambda (x) (f1 (f2 x)))))

; Type: [Number * [T -> T] -> [T -> T]]
; Purpose: Compose a function f n times.
; Signature: c1-n(n, f)
; Precondition: n > 0
; Test: ((c1-n 10 add1) 1) -> 11
(define c1-n
  (lambda (n f)
    (lambda (x)
      (if (= n 1)
          (f x)
          ((c1-n (- n 1) f) (f x))))))
```

```
; Type: [Number * [T -> T] -> [T -> T]]
; Purpose: Compose a function f n times.
; Signature: c2-n(n, f)
; Precondition: n > 0
; Test: ((c2-n 10 add1) 1) -> 11
(define c2-n
  (lambda (n f)
    (if (= n 1)
        f
        (compose f (c2-n (- n 1) f)))))

; Type: [Number -> [[T -> T] -> [T -> T]]]
; Purpose: Curried version of compose-n
; Pre-condition: n > 0
; Signature: comp1-n(n)(f)
; Test: (((comp1-n 10) add1) 1) -> 11
(define comp1-n
  (lambda (n)
    (lambda (f)
      (cond ((= n 1) f)
            ((even? n) ((comp1-n (/ n 2)) (compose f f)))
            (else (compose f ((comp1-n (- n 1)) f)))))))

; Type: [Number -> [[T -> T] -> [T -> T]]]
; Purpose: Curried version of compose-n
; Pre-condition: n > 0
; Signature: comp2-n(n)(f)
; Test: (((comp2-n 10) add1) 1) -> 11
(define comp2-n
  (lambda (n)
    (cond ((= n 1) (lambda (f) f))
          ((even? n)
           (let ((cn/2 (comp2-n (/ n 2))))
             (lambda (f) (cn/2 (compose f f)))))
          (else
           (let ((cn-1 (comp2-n (- n 1))))
             (lambda (f) (compose f (cn-1 f))))))))
```

For all of the questions below, you may want to add calls to "display" in the body of some functions to explore their behavior. Write you answers in *ex2.pdf*.

1. How many function applications are computed when evaluating (c1-n 3 add1)?
   How many when evaluating ((c1-n 3 add1) 1)? Explain by showing the main steps of the applicative-eval steps of this expression.
2. How many function applications are computed when evaluating (c2-n 10 add1)?
   How many when evaluating ((c2-n 3 add1) 1)?
3. Is there an advantage of c2-n over c1-n in terms of computational complexity?
4. Which functions are invoked and in which order when executing ((comp1-n 3) add1)?
   In general how many function evaluations are computed when evaluating ((comp1-n n) f)?
5. Which functions are invoked and in which order when executing ((comp2-n 3) add1)?
   In general, how many function evaluations are computed when evaluating ((comp2-n n) f)?
   Explain by showing the main steps of applicative-eval steps of this expression.
6. Write the type and describe in words what the function computed by (comp2-n n) does.

Task 5: Abstract and Concrete Syntax, list processing.

In this question, we will write an interpreter for a language that allows computations over polynomials of a single variable. This assignment's purpose is to experiment with concrete and abstract syntax and to practice list processing.

To describe a polynomial object, we use the following Scheme-like syntax, with a 'p' symbol used as a tag to easily recognize such expressions:

```
'(' 'p' '+' ('(' '*' <number> '(' '**' 'x' <number> ')' ')')* <number> ')'
```

Example: `(p + (* 2 (** x 3)) (* 1 (** x 2)) (* 4 (** x 1)) 5)` describes the polynomial $2x^3 + x^2 + 4x + 5$.

Note: To make the code simpler, our interpreter accepts only polynomial expressions where the variables are ordered by the exponent from high to low and no two terms with the same exponent appear in the same polynomial. You can assume the input for polynomials is valid and you will not have to test for valid concrete syntax of polynomials.
For example, these expressions are **not valid**:

- `(p + (* 2 (** x 1)) (* 2 (** x 2)) 1) ;; because of order`
- `(p + (* 2 (** x 1)) (* 1 (** x 1)) 1) ;; because of repetition`

Now that we have a way to describe polynomials, we define operators to perform actions on them:

<u>Plus</u>: adding two polynomials together.

Syntax: `'(' '+' <poly-exp> <poly-exp> ')'`
Example:
```
(+   (p + (* 2 (** x 1)) 2)
     (p + (* 1 (** x 2)) 2))
```
Denotes: $(2x + 1) + (x^2 + 2)$

<u>Times</u>: multiplying two polynomials together.

Syntax: `'(' '*' <poly-exp> <poly-exp> ')'`
Example:
```
(*   (p + (* 1 (** x 3)) 1)
     (p + (* 1 (** x 2)) 2))
```
Denotes: $(x^3 + 1)(x^2 + 2)$

<u>Define</u>: assigning a value to a variable.

Syntax: `'(' 'def' <symbol> <poly-exp> ')'`
Example: `(def p1 (p + (* 3 (** x 1)) 4))`
Denotes: $p1 = (3x + 4)$

<u>Apply</u>: calculating the value of a polynomial given the value of the variable.

Syntax: `'(' <poly-exp> <number-exp> ')'`
Example: `((p + (* 3 (** x 1)) 4) 7)`
Denotes: given $f(x) = (3x + 4), compute\ f(7)$

To summarize, the following BNF describes the concrete syntax of our polynomial calculator:

```
<exp> :: <def> | <number-exp> | <symbol> | <poly-exp>
<poly-exp> :: <poly> | <plus> | <times> | <symbol>
<number-exp> :: <number> | <apply>
<poly> :: '(' 'p' '+' ( '(' '*' <number> '(' '**' 'x' <number> ')' ')' )* <number> ')'
<def> :: '(' 'def' <symbol> <poly-exp> ')'
<plus> :: '(' '+' <poly-exp> <poly-exp> ')'
<times> :: '(' '*' <poly-exp> <poly-exp> ')'
<apply> :: '(' <poly-exp> <number-exp> ')'
<number> :: a Scheme number
<symbol> :: a Scheme symbol
```

> **NOTE**: the concrete syntax of this language relies on Scheme list values. To manipulate lists, Scheme supports the quote operator over lists. This is a **different usage** of the quote syntax that we know and that is a value constructor for symbols.
> When reading the expression: `'(an embedded (list))` – the Scheme interpreter returns a constant list value `(an embedded (list))` – and does not evaluate this list as a regular form (which would have been the case if quote was absent).
> The `(read)` primitive Scheme function (Type: [Empty -> Value]) prompts the user for a value and returns it as a number, boolean, symbol or list value.

The abstract syntax of the language is presented below:

```
<Poly>:
      Components:  Degree: <Number>
                   Coeffs: List<Number>
                         with length(Coeffs) = Degree + 1 and Coeffs corresponding
                         to the degrees in descending order, from n to 0.
<Plus>:
      Components:  P1: <Poly-exp>
                   P2: <Poly-exp>
<Times>:
      Components:  P1: <Poly-exp>
                   P2: <Poly-exp>
<Apply>:
      Components:  P: <Poly-exp>
                   X: <Number-exp>
<Define>:
      Components:  Var: <Symbol>
                   Val: <Poly-exp>
<Poly-exp>:
      Kinds: <Poly>, <Times>, <Plus>, <Symbol>
<Number-exp>:
      Kinds: <Number>, <Apply>
```

The code provided in task5.rkt provides the skeleton of a complete interpreter for this polynomial calculator. It is organized in the following sections:

- **Abstract syntax**: functions that define constructors, membership predicates and accessors for the abstract syntax types of the language;
- **Parser**: the parser transforms the concrete syntax defined by the BNF of the language into abstract syntax expressions.
- **Primitives**: functions to perform operations over polynomial values (addition, multiplication, application).
- **Environment**: functions to represent the global environment required to implement the “define” construct of the language;
- **Interpreter**: the interpreter implements the computation rules over the abstract syntax.

- **Read-Eval-Print-Loop**: The REPL reads concrete syntax expressions, parses them, evaluates the abstract syntax expressions, evaluates them with the interpreter, and loops again.

Part 1

Complete the code of the primitive functions `poly-plus` and `poly-apply`.
The code of the primitive function `poly-times` is provided as an example in task5.rkt.
`poly-plus` adds two polynomials together and returns a new polynomial.
`poly-apply` calculates the value of the polynomial given a number which is the value of the variable.
For example, the value of $2x^2 + 1$ applied on $x = 2$ is 9.
Use Horner's method to calculate the value efficiently. Here is a pseudo-code (in Python) of Horner's implementation:

```
def horner(x, *polynomial):
    """A function that implements the Horner Scheme for evaluating a
    polynomial of coefficients *polynomial in x."""
    result = 0
    for coefficient in polynomial:
        result = result * x + coefficient
    return result
```

You can read here (http://en.wikipedia.org/wiki/Horner%27s_method) to understand how this works.

Complete the code and contracts of the 2 functions poly-plus and poly-apply in *task5.rkt*.

What is Horner's method efficiency when computing (p x) for a polynomial p of degree n (count multiplication operations)? How does it compare with the "naïve" evaluation method of (p x)? Answer in *ex2.pdf*

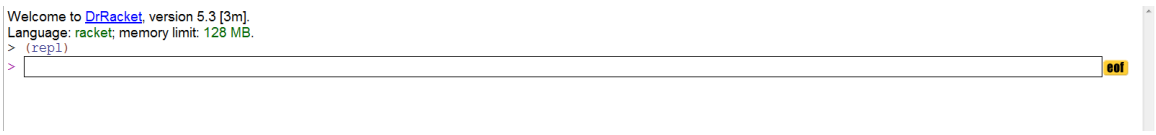
Part 2

Complete the contracts of the abstract syntax functions that implement types: <Times>, <Define>, <Apply> and <Poly-exp>. Answer in task5.rkt.

Complete the code of the parser to convert all legal expression types to abstract syntax types.
Complete the code and contract of the function parse in *task5.rkt*.

After part 1 and part 2 you have a working interpreter that can accept expressions in the language and evaluate them.

Use the function `repl` to check your work. `repl` is the **R**ead-**E**val-**P**rint **L**oop that reads an expression from the user, parses the expression, evaluates it, and prints the result back to the user.
Once you apply `repl` (notice it has not parameters), you will see a box where you can type the expressions for evaluation. This is a picture of the interaction panel of DrRacket.



Experiment with `repl` to see if you interpreter works as it should.

You can use the following code to test you work:

```
>(parse '(p + 1))
'(poly 0 (1))
>(parse '(+ (p + (* 2 (** x 1)) 3) (p + (* 4 (** x 2)) (* 2 (** x 1)) 3)))
'(plus (poly 1 (2 3)) (poly 2 (4 2 3)))
>(Ipoly '(apply (poly 2 (1 2 3)) 2) empty-env)
11
```

These examples are minimal. Provide more extensive tests before submitting your work.

Part 3

We call the concrete syntax we used so far to describe polynomials "scheme like" as it is similar to Scheme's syntax and easy to parse using Scheme's primitive list functions. In order to write polynomials that are easier to understand for humans, we now introduce a more "natural" syntax. These are its rules:

- The expression starts with the `p` tag (as in the original concrete syntax).
- A term corresponding to variable x with coefficient a and exponent b will be written as $a\ x\ b$, where a and b must both appear, meaning that $x\ b$ or $a\ x$ are invalid expressions and $b > 0$.
- After the variable expression, the '+' symbol must appear.
- The expression always ends with a number (which can be zero).

Examples:
`(p 2 x 4 + 3 x 2 + 2)` valid expression.
`(p 2 x 1 + 3 x 2 + 2)` invalid expression,
`(p 2 x + 3 x 2 + 2)` invalid expression.
`(p 2 x 2 + 1 x 1)` invalid expression.

1. Write the new BNF rule for the "natural" form. Write the answer in *ex2.pdf*
2. Add support in you interpreter for these expressions. Decide how to add this expression to the language and make the necessary changes to the abstract and concrete syntax (if needed). Add the necessary code in *task5.rkt*.

Note: Your implementation should support either form of concrete syntax – Scheme-like or natural.

Note: the same restrictions defined for the Scheme form apply to the natural form.

Hint:
- How the parser distinguishes polynomial expressions in the Scheme-like syntax from those in Natural syntax?
- Is the natural form really that different from the scheme form?

Part 4

We want to add a new action to the language; so that we can derive a polynomial.
To do so, we want to add the expression `(der poly)` to the language.
Unlike the `derive` function we saw in Task 2, `der` will return a polynomial that represents the exact derivative of the polynomial. This exact computation is possible because we know how to symbolically derive polynomials.

Decide how to add this expression to the language and make the necessary changes to the abstract and concrete syntax (if needed).
Example of derive expressions:

```
>(der (p 2 x 2 + 2 x 1 + 1))      ; Derive (2x^2 + 2x + 1)
(poly 1 (4 2))                    ; returns (4x + 2)
```

Write your code with contracts in *ex5.rkt*.

Part 5

Questions: (write answers in *ex2.pdf*)

1. What type of evaluation strategy is implemented by the `Ipoly`? Is it applicative order or normal order? Explain.
2. What are the possible types of values computed by this language? Explain.
3. Assume we want to support writing polynomials in an even more natural form. For example:
   (p 2 x + x 2 – 3 x)
   The key differences are that the terms can appear in any order, that terms with coefficient 0 do not appear, that multiple terms with the same degree can appear in a single polynomial expression and that terms with a negative coefficient appear without the + delimiter.
   Can you describe this syntax using BNF? Explain.
   Does this syntax require a different abstract syntax for polynomial expressions? Explain.

Good luck, Have fun!