COMP 321 Homework Final Exam: Implementation

For this exam, you will implement a parser, evaluator, and type-inference engine for a simple expression-based programming language. A program has the form

```
val x1 = e1 ;
val x2 = e2 ;
...
val xN = eN ;
```

where the `xk` are identifiers and the `ek` are expressions. Expression `ek` can include identifiers `xi` for any $i < k$. So an example program might be

```
val f = fn x => fn y => 2*x + y ;
val g = fn x => x/3 ;
val x = f 7 (g 6) ;
```

A program evaluates to the value of the last expression, so this program evaluates to 16.

Formally, the grammar of programs is given as follows:

p ::= s | s p

s ::= val x = e ;

e ::= x | n | true | false | [] |
$\sim e$ | e + e | e − e | e * e | e / e
e < e | e <= e | e > e | e >= e | e = e | e <> e
not e | e andalso e | e orelse e
e :: e | hd e | tl e
if e then e else e fi
fn x => e | e e
(e)

In this grammar, x ranges over identifiers and n over natural number literals. p is the start symbol, so this grammar defines programs, which are semi-colon delimited sequences of statements, each of which is of the form val $x = e$; (even if the program consists of a single statement, it must end with a semi-colon).

Operator precedence and associativity is given in the following table; operators in the same row have equal precedence, and earlier rows have higher precedence than later rows:

Operators	Associativity
Application	Left
∼, hd, tl, not	Right
*, /	Left
+, −	Left
::	Right
<, <=, >, >=, =, <>	Left
andalso, orelse	Left
Abstraction	Right

Expressions may be parenthesized with (and) to control operator precedence. The conditional expression is considered a special form rather than an operator. Expressions of the form *e e* are application expressions; associativity is to the left, so *e1 e2 e3* represents (*e1 e2*)(*e3*). Abstraction is thought of as a unary operator; `fn` x => e is the operator λx applied to the argument e. Because the binary relations will be implemented as (curried) binary operators of type $\alpha \to \alpha \to$ `bool` (for appropriate α), expressions such as $x < y < z$ are legal in this grammar, but will raise a type error when evaluated (or type-inference is run). This is essentially the precedence ordering for the same operations in SML. The precdence of application means that, e.g., (`fn` x => $2{*}x$) 3 + 4 evaluates to 10.

This final project consists of three problems:

PROBLEM 1. *Write a parser for the language given by the above grammar.*

PROBLEM 2. *Write a call-by-value evaluator for the language defined by the above grammar. You will need to write evaluators for semi-colon-terminated expressions and for programs. The inference rules for evaluating expressions are the obvious ones (see the* Lambda-calculus *handout for examples and ask if you have questions). An assignment statement evaluates to the value its right-hand side evaluates to, and a sequence of assignments evaluates to the last assignment statement.*

Of course, one has a sequence of assignments so that one can use identifiers defined in earlier statements in the expressions of later statements. This sounds like the program evaluator needs to maintain an environment, but in our case we can do something much simpler, which will make evaluation of programs reduce easily to evaluation of expressions. The trick is to do a pre-processing step on the AST that has the effect of treating each assignment but the last as a desugared `let`*-expression, the body of which is the (recursively processed) remaining statements. The last statement is replaced by its right-hand side. So, for example,*

```
val x1 = e1 ;
val x2 = e2 ;
val x3 = e3 ;
```

would be processed into an AST equivalent to parsing the expression

```
(fn x1 => (fn x2 => e3) e2) e1
```

Of course, your expression evaluator will still have to maintain environments as part of the notion of closure.

PROBLEM 3. *Write a type-inference module for this programming language. The types are given by the following grammar:*

$$\rho, \sigma, \tau ::= v \mid \texttt{int} \mid \texttt{bool} \mid \sigma \to \tau \mid [\sigma]$$

where v ranges over type variables. The type $[\sigma]$ represents what in ML would be σ `list`*.*