COMP 321 Homeworks 9 and 10: Parsing and evaluation implementation

---

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

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*
$\sim$, `hd`, `tl`	*Right*
Application	*Left*
$*$, /	*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.

PROBLEM 2. *Write a call-by-value evaluator for the language defined by the grammar in Problem 1. 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.