CSCI 3155: Principles of Programming Languages
Spring 2014: April 22, 2014 **Quiz 6**

Name: _____

We will spend about 15 minutes individually and then 15 minutes with a partner on these questions before discussing as a class.

1. **Recursive Descent Parsing.** Consider the following language of arithmetic expressions:

$$\text{expressions} \quad \mathit{expr} ::= \mathit{num} \mid \mathit{expr}\ \text{`+'}\ \mathit{expr} \mid \mathit{expr}\ \text{`}*\text{'}\ \mathit{expr} \mid \text{`}-\text{'}\ \mathit{expr}$$

where *num* is a terminal for integer numbers. The corresponding Scala abstract syntax representation is given below:

```
sealed abstract class Expr
case class Num(i: Int) extends Expr
case class Plus(e1: Expr, e2: Expr) extends Expr
case class Times(e1: Expr, e2: Expr) extends Expr
case class Neg(e1: Expr) extends Expr
```

(a) [6 points] Refactor this grammar so that it is suitable for writing a recursive descent parser (i.e., should be unambiguous and without left recursion). We specify that all binary operators should be left associative, and the relative precedence of the operators from highest to lowest is as follows: '$-$', '$*$', '$+$'. To specify this desired precedence, your grammar will have 3 "layers" of non-terminals, which we call *expr*, *term*, and *factor*. The non-terminal *factor* should correspond to the smallest language. Your response should be in BNF (i.e., no sequence operators $\{\ldots\}$), though you may want to use EBNF with the sequence operator as an intermediate step. Hint: your answer should use 5 non-terminals (including *expr*, *term*, and *factor*).

Solution: In order to write a recursive descent parser, we need to eliminate left recursion. This requirement is made slightly trickier by wanting left associative binary operators. We begin by giving the following EBNF grammar:

$$\begin{aligned} \mathit{expr} &::= \mathit{term}\ \{\text{`+'}\ \mathit{term}\} \\ \mathit{term} &::= \mathit{factor}\ \{\text{`}*\text{'}\ \mathit{factor}\} \\ \mathit{factor} &::= \text{`}-\text{'}\ \mathit{factor} \mid \mathit{num} \end{aligned}$$

We then remove the sequence operators from the above by adding two non-terminals for the two uses of sequences:

$$\begin{aligned} \mathit{expr} &::= \mathit{term}\ \mathit{exprs} \\ \mathit{exprs} &::= \varepsilon \mid \text{`+'}\ \mathit{term}\ \mathit{exprs} \\ \mathit{term} &::= \mathit{factor}\ \mathit{terms} \\ \mathit{terms} &::= \varepsilon \mid \text{`}*\text{'}\ \mathit{factor}\ \mathit{terms} \\ \mathit{factor} &::= \text{`}-\text{'}\ \mathit{factor} \mid \mathit{num} \end{aligned}$$

(b) [6 points] Complete the following implementation of a parser for *factor*. You may assume a parser for *num* is given (`num : Input => ParseResult[Expr]`) and that the input element type is `Char`. Figure 1 at the end gives an except of `ParseResult` for your reference.

Solution:

```
def factor(next: Input): ParseResult[Expr] = (next.first, next.rest) match {
  case ('-', next) => factor(next) map Neg
  case _ => num(next)
}
```

2. [8 points] **Backtracking Search and Continuations.** Consider a language of Boolean formulæ:

formulæ	$f ::= x \mid b \mid f_1 \wedge f_2 \mid \cdots$
constants	$b ::=$ **true** | **false**
variables	x

We represent the abstract syntax of such formulæ using the Scala type `Formula` that includes a case `Var` for variables named by a `String`. The other cases are not relevant for this problem. An assignment (`Asn`) is an environment map from variables to Boolean values. Assume that we have two functions `eval` and `freeVars` that implement a big-step interpreter for formulæ and compute the free variables of formulæ, respectively:

```
sealed abstract class Formula
case class Var(x: String) extends Formula
...
type Asn = Map[String, Boolean]
def eval(a: Asn, f: Formula): Boolean = f match {
  case Var(x) = a(x)
  ...
}
def freeVars(f: Formula): Set[String] = ...
```

In this question, we will implement a SAT-solver that given a formulæ f tries to find a satisfying assignment a for the free variables of f (i.e., an a such that `eval(`a`,` f`)` returns **true**).

Our `sat` implementation will be brute force by simply trying all possible assignments to the free variables of f and stopping when we find the first satisfying assignment. If we have tried all possible assignments to the free variables, then `sat` should return `None`. We implement `sat` with a helper function `tryAll` to do this search, which should implement the following specification:

Name: _____

> Given formula f, list of unassigned free variables *fvs*, and partial assignment a (i.e., the free variables of f are $fvs \cup \mathrm{dom}(a)$), if f is satisfiable then `tryAll(`*fvs*`,` a`,` *fc*`)` returns `Some(`a'`)` where a' is a satisfying assignment for f that is an extension of a; otherwise, it returns *fc*`()`.

Hint: Observe from the specification that the `fc` parameter is a continuation that gets called when assignment `a` turns out not to be a satisfying assignment for formula `f`. Leveraging this callback parameter, you will want to create a new continuation for `fc` on a recursive invokation of `tryAll`.

Solution: As you may know, Propositional SAT is an NP-complete and Quantified SAT is PSPACE-complete. Thus, giving an efficient implementation is quite difficult, but our exponential, brute force algorithm can be written very compactly.

```
def sat(f: Formula): Option[Asn] = {
  def tryAll(fvs: List[String], a: Asn, fc: () => Option[Asn]): Option[Asn] = fvs match {
    case Nil => if (eval(a, f)) Some(a) else fc()
    case x :: xs =>
      tryAll(xs, a + (x -> true), { () => tryAll(xs, a + (x -> false), fc) })
  }
  tryAll(fvs, Map.empty, () => None)
}
```

You might notice that `fc` is just a thunk to use only when we have a complete assignment and the assignment is not satisfying (i.e., `fvs` is `Nil` and `eval(a,f)` is **false**). In essence, we have failed by guessing an assignment that is not satisfying—`fc` stands for "failure continuation." We could rewrite this function using call-by-name, though the thunk form perhaps makes it more explicit that the evaluation of the third argument to `tryAll` is suspended:

```
def sat(f: Formula): Option[Asn] = {
  def tryAll(fvs: VarSet, a: Asn, fc: => Option[Asn]): Option[Asn] = fvs match {
    case Nil => if (eval(a, f)) Some(a) else fc
    case x :: xs => tryAll(xs, a + (x -> true), tryAll(xs, a + (x -> false), fc))
  }
  tryAll(fvs, Map.empty, None)
}
```

Name: _____

```
sealed abstract class ParseResult[T] {
  def map[U](f: T => U): ParseResult[U]
}
case class Success[T](result: T, next: Input) extends ParseResult[T]
case class Failure(next: Input) extends ParseResult[Nothing]
```

Figure 1: Excerpt from the `ParseResult` type constructor for your reference.