CSCI 3155: Homework Assignment 2

Spring 2012: Due Monday, February 13, 2012

Like last time, find a partner. You will work on this assignment in pairs. However, note that each student needs to submit a write-up and are individually responsible for completing the assignment.

You are welcome to talk about these questions in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Recall the evaluation guideline from the course syllabus. Make sure that your file compiles and runs. A program that does not compile will not be graded.

Submission Instructions. To submit, upload to the moodle exactly two files named as follows:

- Homework2-YourIdentiKey.pdf with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)
- Homework2-YourIdentiKey.scala with your answers to the coding exercises

Replace YourIdentiKey with your IdentiKey. To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Getting Started. Download the code template Homework2.scala from the assignment page. Be sure to also write your name, your partner, and collaborators there.

1. Feedback. Complete the survey on the linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.

2. Grammars: Synthetic Examples.
   (a) Show that the following grammar is ambiguous:

   \[ A ::= A & A | V \]
   \[ V ::= a | b \]

   (b) Describe the language defined by the following grammar:

   \[ S ::= A | B | C \]
   \[ A ::= a A | a \]
   \[ B ::= b B | \epsilon \]
   \[ C ::= c C | c \]

   (from Sebesta, Chapter 3)
(c) Consider the following grammar:

\[
\begin{align*}
S & ::= A \ a \ B \ b \\
A & ::= A \ b \ | \ b \\
B & ::= a \ B \ | \ a
\end{align*}
\]

Which of the following sentences are in the language generated by this grammar? For the sentences that are described by this grammar, demonstrate that they are by giving derivations.

1. baab
2. bbbab
3. bbaaaaa
4. bbaab

(from Sebesta, Chapter 3)

(d) Consider the following grammar:

\[
\begin{align*}
S & ::= a \ S \ c \ B \ | \ A \ | \ b \\
A & ::= c \ A \ | \ c \\
B & ::= d \ | \ A
\end{align*}
\]

Which of the following sentences are in the language generated by this grammar? For the sentences that are described by this grammar, demonstrate that they are by giving parse trees.

1. abcd
2. acccbbd
3. acccbbc
4. acd
5. accc

(from Sebesta, Chapter 3)


Later in this class, we will use the PL-Detective tool that explores programming language concepts using a language called MYSTERY. The language MYSTERY is so named in that its semantics is configurable and thus determining them can be a mystery. MYSTERY does, however, have a fixed syntax that we explore a bit here.

(a) Consider the following two grammars for Expr. Note that the first grammar is part of Expr's grammar in MYSTERY. In both grammars, Operator and Operand are the same as in MYSTERY, though you do not need to know their productions for this question.

\[
\begin{align*}
Expr & ::= \ \text{Operand} \ | \ Expr \ \text{Operator} \ \text{Operand} \\
Expr & ::= \ \text{Operand} \ \text{ExprSuffix} \\
ExprSuffix & ::= \ \text{Operator} \ \text{Operand} \ \text{ExprSuffix} \ | \ \epsilon
\end{align*}
\]

i. Intuitively describe the expressions generated by the two grammars.
ii. Do these grammars generate the same or different expressions? Explain.

(b) Consider the following two grammars for Operand. The first grammar is part of Operand's grammar in MYSTERY. In both grammars, Expr is the same as in MYSTERY. Again, you do not need to look at the definition of Expr to answer this question.

\[
\text{Operand} ::= \text{Number} \mid \text{Id} \mid \text{Operand}[\text{Expr}]
\]

\[
\text{Operand} ::= \text{Number} \text{OperandSuffix} \mid \text{Id} \text{OperandSuffix}
\]

\[
\text{OperandSuffix} ::= [\text{Expr}] \text{OperandSuffix} \mid \varepsilon
\]

Note that we are using BNF (not EBNF), the square brackets ([...]) are terminals (used for array references).

i. Intuitively describe the expressions generated by the two grammars.

ii. Do these grammars generate the same or different expressions? Explain.

(c) A portion of the syntax for MYSTERY is given below.

\[
\text{Expr} ::= \text{Operand} \mid \text{Expr} \text{Operator} \text{Operand}
\]

\[
\text{Operand} ::= \text{Number} \mid \text{Id} \mid \text{Operand}[\text{Expr}]
\]

\[
\text{Operator} ::= + \mid > \mid \text{AND}
\]

Looking at the syntax of MYSTERY determine if the + operator is left or right associative. Also, determine if + has higher, lower, or the same precedence as >. Explain the reasoning behind your answer.

(d) Write a Scala expression to determine if – has higher precedence than << or vice versa. Make sure that you are checking for precedence in your expression and not for left or right associativity. Use parentheses to indicate the possible abstract syntax trees, and then show the evaluation of the possible expressions. Finally, explain how you arrived at the relative precedence of – and << based on the output that you saw in the Scala interpreter.

(e) Give a BNF grammar for floating point numbers that are made up of a fraction (e.g., 5.6 or 3.123 or -2.5) followed by an optional exponent (e.g., E10 or E-10). The exponent, if it exists, is the letter ‘E’ followed by an integer. For example, the following are floating point numbers: 3.5E3, 3.123E30, -2.5E2, -2.5E-2, and 3.5. The following are not examples of floating point numbers: 3.E3, E3, and 3.0E4.5.

More precisely, our floating point numbers must have a decimal point, do not have leading zeros, can have any number of trailing zeros, non-zero exponents (if it exists), must have non-zero fraction to have an exponent, and cannot have a ‘-’ in front of a zero number. The exponent cannot have leading zeros.

For this exercise, let us assume that the tokens are characters in the following alphabet \( \Sigma \):

\[
\Sigma \overset{\text{def}}{=} \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, E, -, .\}
\]

Your grammar should be completely defined (i.e., it should not count on a non-terminal that it does not itself define).

For this project, you will develop some functions for working with boolean formulae given by the following Scala case classes:

```scala
sealed abstract class Formula

case class B(bool: Boolean) extends Formula

case class Var(name: String) extends Formula

case class And(left: Formula, right: Formula) extends Formula

case class Or(left: Formula, right: Formula) extends Formula

case class Not(sub: Formula) extends Formula

case class Forall(binding: String, body: Formula) extends Formula

case class Exists(binding: String, body: Formula) extends Formula
```

Here, B and True represent Boolean values. Var(x) represents the Boolean variable with name x. The constructors And, Or, and Not represent Boolean conjunction, disjunction, and negation, respectively. For example, the mathematical formula

\[(a \lor b) \land c\]

would be represented by the Scala value

```
And(Or(Var("a"), Var("b")), Var("c"))
```

We will explain Forall and Exists in a moment.

To assign truth values to variables, we will a Scala `Map`:

```scala
type Assignment = Map[String, Boolean]
```

Here, if an Assignment maps a string s to a Boolean b, then that Assignment gives the variable represented by the string s the value b. For this homework, you do not need to worry about how to use Map in the Scala library, as we provide functions to manipulate assignments:

```scala
val empty: Assignment = Map[String, Boolean]

def lookup(a: Assignment, x: String): Option[Boolean]

def bind(a: Assignment, x: String, b: Boolean): Assignment
```

The lookup function tries to look up the binding for a variable: it returns Some of that binding if it exists or None if it does not. The bind function adds a binding to an assignment.

You may assume for purposes of this project that whenever you work with an assignment, all listed variables are distinct (i.e., you don't need to worry about shadowing names).

The last two kinds of formula, Forall and Exists, represent the similarly named quantifiers. The Boolean formula Forall(x, f) is true if f is true under all assignments to x (i.e., if f is true when x is assigned true and when x is assigned false). The Boolean formula Exists(x, f) is true if f is true either for x assigned true or x assigned false. For example, the formula

```
Forall("x", Or(Var "x", Var "y"))
```
is true under the assignment \(
\text{Map("y" -> true)} \) and false under \(
\text{Map("y" -> false)} \).

Implement the following functions that work with Boolean formulae. Do not use any Scala library functions for this assignment. You should manipulate Scala lists directly by pattern matching.

(a) **Set of Variables.** We will need a data structure to represent a set of variables. Here, we use Scala's List type and enforce that a variable set has no duplicates as a data structure invariant (as long as we only use \(\text{emptyVarSet, removeVar, and insertVar} \)).

```scala
type VarSet = List[String]

i. def removeVar(x: String, set: VarSet): VarSet. Removes the variable \( x \) from the set \( set \).

ii. def insertVar(x: String, set: VarSet): VarSet. Inserts the variable \( x \) into set \( set \) ensuring that there are no duplicates. The order of the variables does not matter.
```

(b) **Negation Normal Form.**

A formula is in negation normal form (NNF) if negation appears only “in front” of variables. For example, \( \text{Or(Not(Var "x"), B(true))} \) is in NNF, but \( \text{Not(And(Var "x", B(false))} \) is not.

- **def** toNNF(f: Formula): Formula. Given an arbitrary formula, this function returns an equivalent formula in NNF. To write this function, you can use the following equivalences:
  - \( \text{Not(B(true)) = B(false)} \)
  - \( \text{Not(B(false)) = B(true)} \)
  - \( \text{Not(Not(f)) = f} \)
  - \( \text{Not(And(f1, f2)) = Or(Not(f1), Not(f2))} \)
  - \( \text{Not(Or(f1, f2)) = And(Not(f1), Not(f2))} \)
  - \( \text{Not(Forall(x, f)) = Exists(x, Not(f))} \)
  - \( \text{Not(Exists(x, f)) = Forall(x, Not(f))} \)

Note that your implementation should be recursive.

(c) **Local Simplification.**

Here, write a few functions that only simplify the “top” node. They do not look for opportunities to simplify in the children of the node. They should not be recursive.

i. **def** simplifyNot(f: Formula): Formula. Eliminate constants from Not nodes using the following simplifications:
  - \( \text{Not(B(true)) = B(false)} \)
  - \( \text{Not(B(false)) = B(true)} \)

ii. **def** simplifyAnd(f: Formula): Formula. Eliminate constants from And nodes using the following simplifications:
  - \( \text{And(B(true), f2) = f2} \)
  - \( \text{And(f1, B(true)) = f1} \)
  - \( \text{And(B(false), f2) = B(false)} \)
  - \( \text{And(f1, B(false)) = B(false)} \)
iii. **def simplifyOr**(f: Formula): Formula. Perform the analogous constant elimination from Or nodes.

(d) **Global Simplification.**

Now write a recursive function **def simplifyM**(f: Formula): Formula that simplifies a formula as much as possible by applying the local simplifications in a bottom-up manner. You will want to call the local simplify functions that you defined previously.

(e) **Global Simplification with a Visitor.**

Now, we will write a recursive visitor function that will allow us to apply a any local simplification throughout a formula.

i. **def visit**(v: Formula => Formula, f: Formula): Formula. The visit(v, f) function takes a function argument v and a formula f. It applies the function v to each node in the formula f in a bottom-up fashion to build a new formula. For example, applying visit with a function v to 

\[
\text{And}(\text{B(true)}, \text{Var}("x"))
\]

will conceptually perform the following steps:

\[
\text{val } a = v(\text{B(true)}) \\
\text{val } b = v(\text{Var}("x")) \\
\text{val } c = v(\text{And}(a, b))
\]

Note how v is applied bottom-up (i.e., children before their parents).

ii. **def simplifyV**(f: Formula): Formula. Now write a global simplifier that computes the same results as simplifyM that you defined above but is not itself recursive. You will want to call visit with an appropriate local simplification function.

(f) **Evaluation and Satisfiability.**

Write the following functions that work with assignments:

i. **def eval**(a: Assignment, f: Formula): Boolean. This function evaluates the Boolean formula on the given variable assignment and returns the result as a Scala Boolean. For example, given the value

\[
\text{And}(\text{Or}(\text{Var}("x"), \text{Var}("y")), \text{Var}("z"))
\]

and the assignment

\[
\text{Map("x" -> true, "y" -> false, "z" -> true)}
\]

the eval function should return true. Your function can assume that all of the formula's free variables are specified in the assignment.

ii. **def freeVars**(f: Formula): VarSet. This function takes a formula and returns a list of the names of the free variables of the formula. The free variables are those that are not bound by a quantifier. For example, for

\[
\text{Exists("x", Or(Var("x"), Var("y")))}
\]

you should return the list List("y"). Make the computation of this set of free variables *tail recursive* (i.e., using a tail recursive helper function that does the work).
iii. **def** sat(f: Formula): Option[Assignment]. This function returns Some(a), where a is a satisfying assignment, if the formula is satisfiable, or None otherwise. For example,

```
sat (Or(Var("x"), Var("y")))
```

could return three things:

- Some(Map("x" -> true, "y" -> false)),
- Some(Map("x" -> false, "y" -> true)),
- Some(Map("x" -> true, "y" -> true))

whereas

```
sat (And(Var("x"), Not(Var("x")))
```

would return None. If more than one assignment is possible, you may return any assignment.

Do not worry about the efficiency of this function. You may simply try all possible assignments until you find one that is satisfying (e.g., using eval). You might find the freeVars function handy here.

(g) **Testing.**

You need to write at least one unit test for each of the above sections.

Name your most tricky test using the convention from before (e.g., testToNNF). Your tests should all take an argument of the type of the function being tested and return Boolean. For example, here's a test for toNNF:

```
def testToNNF(toNNF: Formula => Formula): Boolean =
  Or(B(true), Not(Var("x"))) == toNNF(Not(And(B(false), Var("x"))))
```

Note that since sat may return any possible assignment, your tests for sat will have to use it in combination with eval. You may receive **extra credit** if your test is particularly tricky and breaks the most number of implementations (and does not break your own). However, because we are not concerned with the efficiency of your sat implementation, do not make tests that take a long time to finish.