CMSC 330 Using grammars for programming languages Fall 2012

1. Grammars can be used to express concepts like associativity and precedence of operators. For each of the following languages, give unambiguous grammars capturing the appropriate associativity and precedence.

   a. Propositional calculus formulas having operators unary $\sim$ (negation) and binary $\rightarrow$ (implication), operands `p` and `q`, and parentheses. The operators should both be right associative and have their usual precedence, which is that $\sim$ has higher precedence than $\rightarrow$.

   b. C expressions with identifiers `a` and `b` and operators `->`, `*`, and `++`. `->` is a left associative binary operator, `*` is a right associative unary prefix operator, and `++` is a left associative unary postfix operator. The prefix `*` operator has a lower precedence than the postfix `++` operator, but higher than the binary `->` operator. For example, the expression `*a++` increments `a` before dereferencing it.

   c. Operators in APL are right associative and have equal precedence, unless altered by parentheses. The operators $+$, $-$, *, / can appear as both monadic (unary) and dyadic (binary) operators. Assignment ($\leftarrow$) is also treated as a binary operator that returns the value assigned as its result. Write an unambiguous context free grammar for APL expressions containing the operands `a`, `b`, and `c`.

   Since the operators all have the same precedence, make sure your grammar gives the correct interpretation to an expression such as for example $-\mathtt{a}+\mathtt{b}$, which according to the above would be (unlike in conventional arithmetic and languages you have seen) $-(\mathtt{a}+\mathtt{b})$.

   d. In C the binary bitwise and shift operators `«`, `»`, `&`, `^`, and `|` are all left–associative. The unary prefix operator `~` is right–associative. Their precedence is as follows:

`~`	highest prededence
`«` and `»`	next highest precedence (these have the same precedence)
`&`	next highest precedence
`^`	next highest precedence
`|`	lowest precedence

   Parentheses may also be used in expressions using these operators, and may be thought of as operators having the highest precedence.

   Write an unambiguous context–free grammar that generates all valid C expressions using these operators, assuming the only valid operand is a variable named `n`.

   e. Repeat the previous problem now assuming that all of the binary operators mentioned are right–associative instead of left–associative.

   f. Repeat part (d) now assuming that the right operand of the shift operators `«` and `»` can only be either the constant `0xF` or `0x1`. The left operand of the shift operators, and the operands of the other operators, may be either `n` or any valid subexpression formed using any operators.

   g. Write an unambiguous context–free grammar that generates certain array element references in C or Java. Array element references may be single dimensional or multidimensional. The only valid array name is `a`. An array name is followed by one or more subscripts, which are square braces (`[]`) surrounding either the variable `n`, or another array reference. (Note that parentheses are not allowed in array element references in this problem.) Some valid array references would be:

```
a[n]
a[n][n]
a[a[n]]
a[n][a[n]][a[a[n][n]a[n]][n]
```

If an array is multidimensional your grammar must give the interpretation that the subscripts are grouped from **left to right**.

h. `bc` is a UNIX calculator utility; you can run `bc` and type in expressions to be evaluated, including assignments to variables, function definitions, etc. Write an unambiguous context–free grammar generating a small subset of valid `bc` expressions using the following operators, which are listed in order of **decreasing precedence**:

`++`	the postfix unary increment operator is **nonassociative**
`^`	the binary exponentation operator is right–associative
`=`	the assignment operator is right–associative
`<`, `>`, and `==`	these three comparison operators have the same precedence and are left–associative

The only valid variable names are `x`, `y` and `z`. Parentheses may also be used in expressions using these operators, and may be thought of as operators having the highest precedence.

Saying that the `++` operator is nonassociative means that multiple occurrences of this operator cannot be applied in sequence. (It can only be applied to an lvalue, but it returns an rvalue, so it cannot be applied to its result.) For example, `x++` is valid but `x++++` is not.

Note that the left operand of any assignment must just be a **single variable name**. The right operand of an assignment, and the operands of the other operators (other than `++` as mentioned above), may be either `x` or `y`, or any valid subexpression formed using any operators. Since the assignment operator is associative it may be applied in sequence, but only a variable name may appear on the left side of an assignment. For example, the expression `x = y = x++ « x` is valid, but `x++ = y` and `x = y ^ x = x` are not.

Lastly note that the operators' precedence means that an expression like `x = y < z` should have the interpretation `(x = y) < z` (`y` is assigned to `x`, and the result of the assignment is compared to `z`), not `x = (y < z)` as you might expect (which is what the expression would mean in C or Java, for example).

2. In lecture an ambiguous grammar for `if` statments was shown, in which it was impossible to conclude which of two nested `if` statments an `else` belonged to (the "dangling else" problem). Consider the two grammars below (written using BNF notation). Does each grammar cure the dangling else problem by eliminating the ambiguity of which `if` statement an `else` part is associated with? Assume in both cases that the only valid boolean expression consists of the single boolean variable `b`, and the only valid statement consists of the terminal string `skip` (obviously we could extend the grammars with more realistic productions, but it would just make the problem longer and not really be germane).

a.
```
<stmtlist>   ::=  <stmtlist> ; <stmt> | <stmt>
<stmt>       ::=  <ifstmt> | skip
<ifstmt>     ::=  if b then <stmtlist> <elsepart> end
<elsepart>   ::=  else <stmtlist> | ϵ
```

b.
```
<stmt>       ::=  <ustmt> | <cstmt>
<ustmt>      ::=  skip | begin <stmtlist> end
<cstmt>      ::=  if b then <ustmt> else <stmt> | if b then <ustmt>
<stmtlist>   ::=  <stmtlist> ; <stmt> | <stmt>
```

Note that <ustmt> represents an unconditionally–executed statement, while <cstmt> represents a conditionally–executed statement.

3. Context-free grammars cannot express all the syntactic restrictions normally placed on programming languages; for instance one example is that all function calls must have the same number of actual

parameters as the number of formal parameters in the function's definition. Consider a Scheme–like language that permits a definition of a single function `f` followed by a single call to it (each with an arbitrary number of parameters) defined by the grammar below. (Note that every expression in Scheme is surrounded by parentheses.)

```
<program>   ::=  ( define f ( <formals> ) <body> ) ( f <actuals> )
<formals>   ::=  <formals> <id>  |  ϵ
<actuals>   ::=  <actuals> <expr>  |  ϵ
<body>      ::=  ...
<id>        ::=  ...
<expr>      ::=  ...
```

The expression `( define f ( <formals> ) <body> )` is the definition of a function `f` with parameter list <formals> and body <body>, while the following expression `( f <actuals> )` is a following call to `f` with actual parameters <actuals>. Productions for <body>, <id> and <expr> aren't given; you should assume that they derive the body of a function (a list of expressions), an identifier, and an expression respectively. As above, this grammar generates small programs that consist of only a single definition of one function, followed by a single call to it.

The problem with this grammar is that it derives programs (consisting of a function definition followed by a call) where the call to the function `f` has a different number of arguments than the definition of the function `f` has parameters. For example, you can verify that the following can be derived from the start symbol <program>; this is a case where `f` is defined to have two formal parameters but is then called with only one actual parameter: `( define f ( <id> <d> ) <body> ) ( f <id> )`.

a. Develop another grammar for this language that enforces the restriction that a call to `f` must have the same number of arguments as there are formal parameters in its definition. As above, you can treat <body>, <id> and <expr> as nonterminals.

b. Determine whether it is possible to generalize the approach in part (a) to a language in which arbitrarily many calls to a procedure or function can occur. In other words, can your grammar from part (a) be adjusted to allow multiple calls to `f` following the definition of `f`, where each call has the same number of actual parameters as the number of formal parameters in the definition of `f`?