CMSC 330 Homework #5 Fall 2005

Using grammars for programming languages

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 like for example $-$`a` $+$ `b`, which according to the above would be (unlike in conventional arithmetic and languages you have seen) $-$(`a` $+$ `b`).

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. 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 which 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 following grammar:

<program>	::=	`(` `define` `f` `(` <formals> `)` <body> `)` `(` `f` <actuals> `)`
<formals>	::=	<formals> <id> $\mid \epsilon$
<actuals>	::=	<actuals> <expr> $\mid \epsilon$
<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 which 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 which 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`?

4. Write unambiguous grammars for the following languages:

a. $\{ w \mid w \in \{a, b\}^* \text{ and } w \text{ has an odd number of } a\text{'s and an odd number of } b\text{'s} \}$

b. $\{ w \mid w \in \{a, b\}^* \text{ and } w \text{ contains no substrings of the form } ab \}$

c. $\{ w \mid w \in \{a, b\}^* \text{ and every pair of } a\text{'s in } w \text{ is followed by at least one } b \}$