Programming Languages and Computation

Week 3: LL(1) Grammars

* 1. Consider the following grammar, which describes the structure of statements in the While programming language.

$$
\begin{array}{rcll}
\mathit{Prog} & \longrightarrow & \mathit{Stmt}\ \mathit{Stmts}\ \$ & (1) \\
\mathit{Stmt} & \longrightarrow & \mathsf{if}\ \mathsf{bexp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt} & (2) \\
 & | & \mathsf{while}\ \mathsf{bexp}\ \mathsf{do}\ \mathit{Stmt} & (3) \\
 & | & \mathsf{skip} & (4) \\
 & | & \mathsf{id} \leftarrow \mathit{aexp} & (5) \\
 & | & \{\mathit{Stmt}\ \mathit{Stmts}\} & (6) \\
\mathit{Stmts} & \longrightarrow & ;\ \mathit{Stmt}\ \mathit{Stmts} & (7) \\
 & | & \epsilon & (8)
\end{array}
$$

In it expressions appear only as two terminal symbols bexp and aexp because the structure of expressions is not important to the exercises. In total, the terminal symbols are: if, then, else, while, do, skip, id, bexp, aexp, the left and right braces, the end-of-input marker and the semicolon. The rules are numbered to make constructing the parse tables easier. The language of this grammar (start symbol *Prog*) includes strings such as:

$$\mathsf{while}\ \mathsf{bexp}\ \mathsf{do}\ \mathsf{id} \leftarrow \mathsf{aexp}$$

i.e. strings that show the control structure of the program without specifying the particular expressions involved.

(a) By following the algorithms given in the reference notes, fill out the table. In the second column, row X (i.e. with nonterminal X), insert Nullable(X). In the third column, row X, insert First(X); and in the fourth Follow(X).

Nonterminal	Nullable?	First	Follow
Prog			
Stmt			
Stmts			

(b) Construct the parse table for the grammar.

* 2. Consider now the following grammar:

$$
\begin{array}{rcll}
\mathit{Prog} & \longrightarrow & \mathit{Stmt}\ \mathit{Stmts}\ \$ & (1) \\[4pt]
\mathit{Stmt} & \longrightarrow & \textsf{if bexp then } \mathit{Stmt} \textsf{ else } \mathit{Stmt} & (2) \\
 & | & \textsf{while bexp do } \mathit{Stmt} & (3) \\
 & | & \textsf{skip} & (4) \\
 & | & \textsf{id} \leftarrow \mathit{aexp} & (5) \\
 & | & \mathit{Stmt}\ \mathit{Stmts} & (6) \\[4pt]
\mathit{Stmts} & \longrightarrow & ;\ \mathit{Stmt}\ \mathit{Stmts} & (7) \\
 & | & \epsilon & (8)
\end{array}
$$

This grammar is the same as the previous one, except that braces have been removed in line 6.

(a) Construct the Nullable, First and Follow maps for this grammar.

(b) Construct the parsing table for this grammar.

(c) Give an example of a string in the language of this grammar that could not be handled by a recursive descent parser (i.e. a string in the language where some step in the derivation of the string is not uniquely determined by a combination of the leftmost non-terminal and the next letter of the input).

* 3. Consider the following grammar for arithmetic expressions:

$$
\begin{array}{rcll}
E & \longrightarrow & \textsf{num} & (1) \\
 & | & \textsf{id} & (2) \\
 & | & E+E & (3) \\
 & | & E*E & (4) \\
 & | & (E) & (5)
\end{array}
$$

(a) Construct the nullable, first and follow maps for this grammar.

(b) Construct the parse table for this grammar.

(c) Give an example of a string in the language of this grammar that could not be handled by a recursive descent parser.

** 4. Consider the following two grammars for arithmetic and Boolean expressions. The first has the same structure as the grammar we gave when we introduced the While language. Nonterminal B derives Boolean expressions and nonterminal A derives arithmetic expressions.

$$
\begin{array}{rcl}
B & \longrightarrow & \textsf{true} \mid \textsf{false} \mid A<A \mid A=A \mid !B \mid B \,\&\&\, B \mid B \,||\, B \mid (B) \\
A & \longrightarrow & \textsf{id} \mid \textsf{num} \mid A+A \mid A-A \mid A*A \mid (A)
\end{array}
$$

The second grammar has the same structure as the LL(1) description given in the practical sheet. Nonterminal *BExp* derives Boolean expressions and nonterminal *AExp* derives arithmetic expressions.

$$
\begin{array}{rcl}
\mathit{BExp} & \longrightarrow & \mathit{BFac}\ \mathit{BExps} \\[1em]
\mathit{BExps} & \longrightarrow & \|\ \mathit{BFac}\ \mathit{BExps} \\
 & | & \epsilon \\[1em]
\mathit{BFac} & \longrightarrow & \mathit{BNeg}\ \mathit{BFacs} \\[1em]
\mathit{BFacs} & \longrightarrow & \&\&\ \mathit{BNeg}\ \mathit{BFacs} \\
 & | & \epsilon \\[1em]
\mathit{BNeg} & \longrightarrow & !\ \mathit{BNeg} \\
 & | & \mathit{BRel} \\[1em]
\mathit{BRel} & \longrightarrow & \mathit{AExp}\ \mathit{BRels} \\[1em]
\mathit{BRels} & \longrightarrow & <\mathit{AExp} \\
 & | & =\mathit{AExp} \\
 & | & \epsilon \\[1em]
\mathit{AExp} & \longrightarrow & \mathit{AFac}\ \mathit{AExps} \\[1em]
\mathit{AExps} & \longrightarrow & +\ \mathit{AFac}\ \mathit{AExps} \\
 & | & -\ \mathit{AFac}\ \mathit{AExps} \\
 & | & \epsilon \\[1em]
\mathit{AFac} & \longrightarrow & \mathit{Atom}\ \mathit{AFacs} \\[1em]
\mathit{AFacs} & \longrightarrow & *\ \mathit{Atom}\ \mathit{AFacs} \\
 & | & \epsilon \\[1em]
\mathit{Atom} & \longrightarrow & \mathsf{num} \\
 & | & \mathsf{true} \\
 & | & \mathsf{false} \\
 & | & \mathsf{id} \\
 & | & (\ \mathit{BExp}\ )
\end{array}
$$

(a) Are these two grammars equivalent (do *B* and *BExp* derive the same strings, *A* and *AExp* derive the same strings?)

(b) Consider the following grammar:

$$
\begin{array}{rcll}
\mathit{BExp} & \longrightarrow & \mathit{BAtom}\ \mathit{BExps} & (1) \\
\mathit{BExps} & \longrightarrow & \&\&\ \mathit{BAtomBExps} & (2) \\
\mathit{BExps} & | & \epsilon & (3) \\
\mathit{BAtom} & \longrightarrow & \mathit{AExp} < \mathit{AExp} & (4) \\
\mathit{BAtom} & | & (\ \mathit{BExp}\ ) & (5) \\
\mathit{AExp} & \longrightarrow & \mathit{AAtom}\ \mathit{AExps} & (6) \\
\mathit{AExps} & \longrightarrow & +\ \mathit{AAtom}\ \mathit{AExps} & (7) \\
\mathit{AExps} & | & \epsilon & (8) \\
\mathit{AAtom} & \longrightarrow & \mathsf{num} & (9) \\
\mathit{AAtom} & | & (\ \mathit{AExp}\ ) & (10)
\end{array}
$$

Is this grammar LL(1)?