

PROGRAMMING LANGUAGES AND COMPUTATION

Week 3: LL(1) Grammars

For this problem sheet you will need to consult the algorithms given for calculating Nullable, First and Follow found in the notes: <https://uob-coms20007.github.io/reference/syntax/11.html#calculating-nullable-first-and-follow>

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

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

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:

while bexp do id \leftarrow 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:

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

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

- Construct the Nullable, First and Follow maps for this grammar.
- Construct the parsing table for this grammar.
- 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:

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

- Construct the nullable, first and follow maps for this grammar.
- Construct the parse table for this grammar.
- 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.

$$\begin{array}{ll}
 B & \longrightarrow \text{true} \mid \text{false} \mid A < A \mid A = A \mid !B \mid B \&\& B \mid B \parallel B \mid (B) \\
 A & \longrightarrow \text{id} \mid \text{num} \mid A + A \mid A - A \mid A * A \mid (A)
 \end{array}$$

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.

$$\begin{aligned}
 BExp &\longrightarrow BFac BExps \\
 BExps &\longrightarrow \parallel BFac BExps \\
 &\quad | \epsilon \\
 BFac &\longrightarrow BNeg BFacs \\
 BFacs &\longrightarrow \&\& BNeg BFacs \\
 &\quad | \epsilon \\
 BNeg &\longrightarrow ! BNeg \\
 &\quad | BRel \\
 BRel &\longrightarrow AExp BRels \\
 BRels &\longrightarrow < AExp \\
 &\quad | = AExp \\
 &\quad | \epsilon \\
 AExp &\longrightarrow AFac AExps \\
 AExps &\longrightarrow + AFac AExps \\
 &\quad | - AFac AExps \\
 &\quad | \epsilon \\
 AFac &\longrightarrow Atom AFacs \\
 AFacs &\longrightarrow * Atom AFacs \\
 &\quad | \epsilon \\
 Atom &\longrightarrow \text{num} \\
 &\quad | \text{true} \\
 &\quad | \text{false} \\
 &\quad | \text{id} \\
 &\quad | (BExp)
 \end{aligned}$$

- (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:

$$BExp \longrightarrow BAtom BExps \quad (1)$$

$$BExps \longrightarrow \&\& BAtom BExps \quad (2)$$

$$BExps \mid \epsilon \quad (3)$$

$$BAtom \longrightarrow AExp < AExp \quad (4)$$

$$BAtom \mid (BExp) \quad (5)$$

$$AExp \longrightarrow AAtom AExps \quad (6)$$

$$AExps \longrightarrow + AAtom AExps \quad (7)$$

$$AExps \mid \epsilon \quad (8)$$

$$AAtom \longrightarrow \text{num} \quad (9)$$

$$AAtom \mid (AExp) \quad (10)$$

Is this grammar LL(1)?