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/ll.html#calculating-nullable-first-and-follow

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

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

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:

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

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.

Solution

(a)

Nonterminal	Nullable?	First	Follow
Prog		if,while,skip,id,{	
Stmt		if,while,skip,id,{	else, ;, }, $
Stmts	✓	;	}, $

(b)

Nonterminal	$	if	bexp	then	else	while	do	skip	id	aexp	;	{	}
Prog		1				1		1	1			1	
Stmt		2				3		4	5			6	
Stmts	8										7		8

* 2. Consider now the following grammar:

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

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

(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).

Solution

(a) The Nullable, First and Follow maps can be tabulated as follows.

Nonterminal	Nullable?	First	Follow
Prog		if, while, skip, id	
Stmt		if, while, skip, id	;, else, $
Stmts	✓	;	;, else, $

(b) The parsing table is as follows:

Nonterminal	$	if	bexp	then	else	while	do	skip	id	aexp	;
Prog		1				1		1	1		
Stmt		2, 6				3, 6		4, 6	5, 6		
Stmts	8				8						7, 8

(c) The table shows a conflict between e.g. rules (3) and (6), i.e. when looking to generate the next portion of the input using nonterminal *Stmt* and the next letter of the input is

while, the rule to use is not determined uniquely. A string which would put the parser into this situation is e.g. while bexp do skip ; skip. Intuitively, it is not clear whether the second skip is part of the body of the loop (in which case, we should use rule 3) or after the loop has terminated (in which case, we should use rule 6).

* 3. Consider the following grammar for arithmetic expressions:

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

(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.

Solution

(a)

Nonterminal	Nullable?	First	Follow
E		num, id, (	+, *,)

(b)

Nonterminal	num	id	+	*	(	)
E	1,3,4	2,3,4			3,4,5	

(c) The parse table shows a conflict between e.g. rules 1, 3 and 4 when num is the next non-terminal. A string which will drive the parser to this conflict is num.

** 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.

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

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.

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

(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)?

Solution

(a) No they are not equivalent. In the original grammar, arithmetic expressions and Boolean expressions are quite separate, but in the LL(1) grammar, every arithmetic expression is considered as a Boolean expression. In other words, there is really just one syntactic category of "expressions" of various kinds. This is more common in real programming languages, and we rely on the type system to distinguish between Boolean expressions or arithmetic expressions. For example, in the LL(1) grammar, $\mathit{BExp} \rightarrow^* \mathsf{num}$, but $B\neg \rightarrow^* \mathsf{num}$ in the original grammar.

(b) This grammar is not LL(1). When deriving *BAtom* with the next letter of the input being (there is a conflict between rules (4) and (5). Intuitively, this is because we cannot tell just from looking at an opening parenthesis whether this is the opening parenthesis of an arithmetic expression – in which case we should use rule (4) – or of a Boolean expression – in which case we should use rule (5). This gives some intuition as to why we prefer to collapse arithmetic and Boolean expressions into a single category of expressions – in the LL(1) grammar for While, there is only one production for parenthesized expressions rather than separate ones for Boolean expressions and arithmetic expressions.