Exercises on Compilers

Jordi Cortadella

February 7, 2022

Parsing

1. Define unambiguous grammars for the following languages:

   a) The set of all strings of a's and b's that are palindromes.

   Solution:

   $$S \to \varepsilon \mid a \mid b \mid aSa \mid bSb$$

   b) Strings that match the pattern a^*b^* and have more a's than b's.

   Solution:

   $$\begin{array}{rcl} S & \to & AX \\ A & \to & a \mid aA \\ X & \to & aXb \mid \varepsilon \end{array}$$

   c) Strings with balanced parenthesis and square braces. Example:

   ```
   ( [ [ ] ( ( ) [ ( ) ] [ ] ) ] )
   ```

   Solution:

   $$S \to S\,(\,S\,) \mid S\,[\,S\,] \mid \varepsilon$$

   d) The set of all strings of a's and b's such that every a is immediately followed by at least one b.

   Solution:

   $$S \to bS \mid abS \mid \varepsilon$$

   e) The set of all strings of a's and b's with an equal number of a's and b's.

   Solution: S generates the language that has an equal number of a's and b's. A generates the language than has one more a than b. B generates the language than has one more b than a.

   $$\begin{array}{rcl} S & \to & aB \mid bA \mid \varepsilon \\ A & \to & aS \mid bAA \\ B & \to & bS \mid aBB \end{array}$$

f) The set of all strings of a's and b's with a different number of a's and b's.

Solution:

S: different number of a's and b's
X: more a's than b's
Y: more b's than a's
E: equal number of a's and b's
A: one more a than b
B: one more b than a

$$
\begin{array}{rcl}
S & \to & X \mid Y \\
E & \to & aB \mid bA \mid \varepsilon \\
A & \to & aE \mid bAA \\
B & \to & bE \mid aBB \\
X & \to & AX \mid A \\
Y & \to & BY \mid B
\end{array}
$$

g) Blocks of statements in Pascal, where the semicolons *separate* the statements, e.g.,

```
( statement ;  ( statement ; statement ) ; statement )
```

Solution:

Let I be a statement.

$$
\begin{array}{rcl}
BP & \to & (\ LI\ ) \\
LI & \to & I\,;LI \ \mid\ I \\
I & \to & \texttt{statement} \ \mid\ BP
\end{array}
$$

h) Blocks of statements in C, where the semicolons *terminate* the statements, e.g.,

```
{ statement;  { statement; statement; }  statement; }
```

Solution:

Let I be a statement.

$$
\begin{array}{rcl}
BC & \to & \{\ LI\ \} \\
LI & \to & I\ \ LI \ \mid\ \varepsilon \\
I & \to & \texttt{statement}\,; \ \mid\ BC
\end{array}
$$

2. Consider the following grammar:

$$S \rightarrow SS+ \ \mid SS* \ \mid a$$

and the string $aa{+}a*$.

- Give a leftmost derivation for the string.

Solution:

$$S \rightarrow SS* \rightarrow SS+S* \rightarrow aS+S* \rightarrow aa+S* \rightarrow aa+a*$$

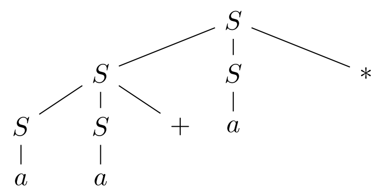
- Give a rightmost derivation for the string.

Solution:

$$S \rightarrow SS* \rightarrow Sa* \rightarrow SS+a* \rightarrow Sa+a* \rightarrow aa+a*$$

- Give a parse tree for the string.

Solution:



- Is the grammar ambiguous or unambiguous? Justify your answer.

Solution: The grammar is unambiguous since only one parse tree can be generated for each string. The three productions of S end with a different terminal symbol (a, $+$ or $*$). It is easy to see that by parsing symbols from right to left, it is only possible to choose one of the productions.

- Describe the language generated by this grammar.

Solution: This grammar generates arithmetic expressions in inverse Polish notation, where $+$ and $*$ are the operators and a represents the operands.

3. Calculate Nullable, First and Follow of the non-terminal symbols in the following grammar:

$$\begin{array}{lcl} A & \rightarrow & B \mid a \\ B & \rightarrow & b \mid \varepsilon \\ C & \rightarrow & c \mid ABC \end{array}$$

Solution:

	Nullable	**First**	**Follow**
A	Yes	$\{a, b\}$	$\{a, b, c\}$
B	Yes	$\{b\}$	$\{a, b, c\}$
C	No	$\{a, b, c\}$	$\emptyset$

4. Consider the following grammar:

$$\begin{array}{rcl} S & \rightarrow & cABc \\ A & \rightarrow & aAa \mid c \\ B & \rightarrow & bBb \mid c \end{array}$$

- Calculate First and Follow for the non-terminal symbols.
- Construct the LL(1) parsing table and check whether it is an LL(1) grammar.

Solution:

	First	**Follow**
S	$\{c\}$	$\{\$\}$
A	$\{a, c\}$	$\{a, b, c\}$
B	$\{b, c\}$	$\{b, c\}$

LL(1) parsing table:

	a	b	c
S			$S \rightarrow cABc$
A	$A \rightarrow aAa$		$A \rightarrow c$
B		$B \rightarrow bBb$	$B \rightarrow c$

It is an LL(1) grammar since there are no conflicts.

5. Calculate NULLABLE, FIRST and FOLLOW for the following grammar:

$$\begin{array}{lcl} S & \rightarrow & uBDz \\ B & \rightarrow & Bv \mid w \\ D & \rightarrow & EF \\ E & \rightarrow & y \mid \varepsilon \\ F & \rightarrow & x \mid \varepsilon \end{array}$$

Construct the LL(1) parsing table and give evidence that this grammar is not LL(1). Modify the grammar as little as possible to make an LL(1) grammar that accepts the same language.

Solution:

	Nullable	**First**	**Follow**
S	no	$\{u\}$	$\{\$\}$
B	no	$\{w\}$	$\{v, x, y, z\}$
D	yes	$\{x, y\}$	$\{z\}$
E	yes	$\{y\}$	$\{x, z\}$
F	yes	$\{x\}$	$\{z\}$

LL(1) parsing table:

	u	v	w	x	y	z
S	$S \rightarrow uBDz$					
B			$B \rightarrow Bv \mid w$			
D				$D \rightarrow EF$	$D \rightarrow EF$	$D \rightarrow EF$
E				$E \rightarrow \varepsilon$	$E \rightarrow y$	$E \rightarrow \varepsilon$
F				$F \rightarrow x$		$F \rightarrow \varepsilon$

It is not an LL(1) grammar since there is a conflict in cell $\langle B, w \rangle$. This is caused by the left recursion of the production rule of B.
It can be easily realized that the language of B is wv^*. The same language can be generated using and additional symbol (B') and right recursion, e.g., $B \rightarrow wB'$ and $B' \rightarrow vB' \mid \varepsilon$.
With the new rule, we have that

$$\textbf{Nullable}(B') = \mathit{yes} \qquad \textbf{First}(B') = \{v\} \qquad \textbf{Follow}(B) = \textbf{Follow}(B') = \{x, y, z\}.$$

The other definitions of Nullable, First and Follow remain the same.
With this transformation, the LL(1) parsing table would be as follows:

	u	v	w	x	y	z
S	$S \rightarrow uBDz$					
B			$B \rightarrow wB'$			
B'		$B' \rightarrow vB'$		$B' \rightarrow \varepsilon$	$B' \rightarrow \varepsilon$	$B' \rightarrow \varepsilon$
D				$D \rightarrow EF$	$D \rightarrow EF$	$D \rightarrow EF$
E				$E \rightarrow \varepsilon$	$E \rightarrow y$	$E \rightarrow \varepsilon$
F				$F \rightarrow x$		$F \rightarrow \varepsilon$

6. Design a table-driven top-down parser for the following grammar:

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow T + E \mid T \\
T &\rightarrow \texttt{num} * T \mid \texttt{num}
\end{aligned}
$$

Solution: First of all, the grammar is not LL(1) since E (and also T) have productions with common prefixes. We need to transform the grammar:

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow T E' \\
E' &\rightarrow + E \mid \varepsilon \\
T &\rightarrow \texttt{num}\ T' \\
T' &\rightarrow * T \mid \varepsilon
\end{aligned}
$$

	Nullable	First	Follow
S	no	{num}	{\$}
E	no	{num}	{\$}
E'	yes	{+}	{\$}
T	no	{num}	{+, \$}
T'	yes	{∗}	{+, \$}

LL(1) parsing table:

	num	+	∗	\$
S	$S \rightarrow E$			
E	$E \rightarrow TE'$			
E'		$E' \rightarrow + E$		$E \rightarrow \varepsilon$
T	$T \rightarrow \texttt{num}\ T'$			
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow * T$	$T' \rightarrow \varepsilon$

It is an LL(1) grammar since there are no conflicts.

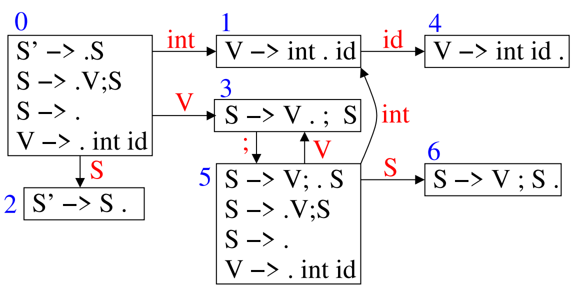
7. Design an LR(1) parser for the following grammar:

$$
\begin{aligned}
S' &\rightarrow S && (1) \\
S &\rightarrow V\ ;\ S \mid \varepsilon && (2)\ (3) \\
V &\rightarrow \texttt{int id} && (4)
\end{aligned}
$$

Solution:

	Nullable	First	Follow
S'	yes	{int}	{\$}
S	yes	{int}	{\$}
V	no	{int}	{;}

Automaton:



LR(1) table:

	int	id	;	\$	S	V
0	s1			r3	2	3
1		s4				
2				acc		
3			s5			
4			r4			
5	s1			r3	6	3
6				r2		

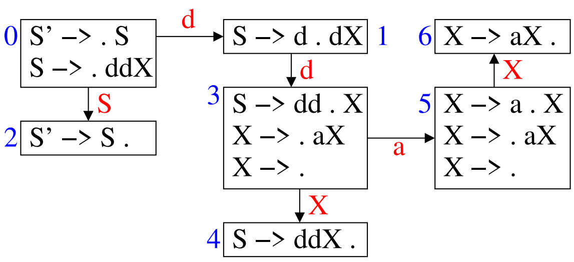
8. Design an LR(1) parser for the following grammar:

$$
\begin{array}{lcll}
S' & \rightarrow & S & \quad (1) \\
S & \rightarrow & ddX & \quad (2) \\
X & \rightarrow & aX \mid \varepsilon & \quad (3)\ (4)
\end{array}
$$

Solution:

	Nullable	**First**	**Follow**
S'	no	$\{d\}$	$\{\$\}$
S	no	$\{d\}$	$\{\$\}$
X	yes	$\{a\}$	$\{\$\}$

Automaton:



LR(1) table:

	a	d	\$	S	X
0		s1		2	
1		s3			
2			acc		
3	s5		r4		4
4			r2		
5	s5		r4		6
6			r3		

9. Consider the following EBFN grammar, where the tokens are the symbols within quotes (e.g., `’if’`), `ID` and `INTEGER`.

```
program : statement+ ;

statement :
      ’if’ paren_expr statement (’else’ statement)?
    | ’while’ paren_expr statement
    | ’do’ statement ’while’ paren_expr ’;’
    | ’{’ statement* ’}’
    | expr ’;’
    | ’;’
;

paren_expr: ’(’ expr ’)’ ;
expr : test | ID ’=’ expr ;
test: sum (’<’ sum)? ;
sum : term (’+’ term | ’-’ term)* ;
term : ID | INTEGER | paren_expr;
```

Calculate `First` and `Follow` for each non-terminal symbol and desgin a recursive-descent parser (ANTLR style). Assume that `EOF` is the last token of any program, i.e., `EOF` $\in$ FOLLOW(`program`). For the code of the parse, you can use the variable `Token` to represent the current token, the function `nexttoken()` to read the next token and the function `match(T)`, with the following definition:

```
match(T) {
    if (Token == T) nexttoken();
    else SyntaxError();
}
```

In the code you can also use expressions like

```
if (Token in First(expr)) {...}
```

You can do your own code optimizations to avoid redundant checks in the parser.

Note: if you would detect some conflict in one of the functions, describe the conflict and generate the code for the remaining functions as if this conflict would not exist.

Solution:

	First
program	’if’ ’while’ ’do’ ’{’ ’;’ ID INTEGER ’(’
statement	’if’ ’while’ ’do’ ’{’ ’;’ ID INTEGER ’(’
paren_expr	’(’
expr	ID INTEGER ’(’
test	ID INTEGER ’(’
sum	ID INTEGER ’(’
term	ID INTEGER ’(’

	Follow
program	EOF
statement	’if’ ’while’ ’do’ ’{’ ’;’ ID INTEGER ’(’ ’else’ ’}’ EOF
paren_expr	’if’ ’while’ ’do’ ’{’ ’;’ ID INTEGER ’(’ ’)’ ’+’ ’-’ ’<’
expr	’)’ ’;’
test	’)’ ’;’
sum	’<’ ’)’ ’;’
term	’+’ ’-’ ’<’ ’)’ ’;’

```
void program() {
    do {
        statement();
    } while (Token in First(statement));
}

void statement() {
   if (Token == 'if') {
       nexttoken(); paren_expr(); statement();
       if (Token == 'else') {
           // Greedy option since 'else' is also in Follow(statement)
           nexttoken(); statement();
       }
   } else if (Token == 'while') {
       nexttoken(); paren_expr(); statement();
   } else if (Token == 'do') {
       nexttoken(); statement(); match('hwile'); paren_expr(); match(';');
   } else if (Token == '{') {
       nexttoken();
       while (Token in First(statement)) statement();
       match('}');
   } else if (Token in First(expr)) {
       expr(); match(';');
   } else if (Token == ';') nexttoken();
   else SyntaxError();
}

void paren_expr() {
    match('('); expr(); match(')');
}

void expr() {
    // Here is a conflict since ID in First(test)
    if (Token in First(test)) test();
    else if (Token == ID) {
        nexttoken(); match('='); expr();
    } else SyntaxError();
}

void test() {
    sum();
    if (Token == '<') {
        nexttoken(); sum();
    }
}

void sum() {
    term();
    while (Token == '+' or Token == '-') {
        nexttoken(); term();  // Code optimization has been applied here
    }
}

void term() {
    if (Token == ID) nexttoken();
    else if (Token == INTEGER) nexttoken();
    else if (Token == '(')) {
      nexttoken(); paren_expr();
    } else SyntaxError();
}
```