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.

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

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

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

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

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

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

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

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

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

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

---

2. Consider the following grammar:
$$S \to SS+ \mid SS* \mid a$$
and the string $aa+a*$.

   - Give a leftmost derivation for the string.
   - Give a rightmost derivation for the string.
   - Give a parse tree for the string.
   - Is the grammar ambiguous or unambiguous? Justify your answer.
   - Describe the language generated by this grammar.

---

3. Calculate NULLABLE, FIRST and FOLLOW of the non-terminal symbols in the following grammar:

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

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.

5. Calculate Nullable, First and Follow for the following grammar:

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

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.

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

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

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

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

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

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

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.