CS 320
Computer Language Processing
Exercise Set 3

March 19, 2025

Exercise 1 If L is a regular language, then the set of prefixes of words in L is also a regular language. Given this fact, from a regular expression for L, we should be able to obtain a regular expression for the set of all prefixes of words in L as well.

We want to do this with a function prefixes that is recursive over the structure of the regular expression for L, i.e. of the form:

$$\begin{aligned}
\text{prefixes}(\epsilon) &= \epsilon \\
\text{prefixes}(a) &= a \mid \epsilon \\
\text{prefixes}(r \mid s) &= \text{prefixes}(r) \mid \text{prefixes}(s) \\
\text{prefixes}(r \cdot s) &= \dots \\
\text{prefixes}(r^*) &= \dots \\
\text{prefixes}(r^+) &= \dots
\end{aligned}$$

1. Complete the definition of prefixes above by filling in the missing cases.
2. Use this definition to find:
   (a) $\text{prefixes}(ab^*c)$
   (b) $\text{prefixes}((a \mid bc)^*)$

Solution The computation for $\text{prefixes}(\cdot)$ is similar to the computation of $\text{first}(\cdot)$ for grammars.

1. The missing cases:
   (a) $\text{prefixes}(r \cdot s) = \text{prefixes}(r) \mid r \cdot \text{prefixes}(s)$. Either we have read r partially, or we have read all of r, and a part of s.
   (b) $\text{prefixes}(r^*) = r * \cdot \text{prefixes}(r)$. We can consider $r^* = \epsilon \mid r \mid rr \mid \dots$, and apply the rules for union and concatenation. Intuitively, if the word has $n \geq 0$ instances of r, we can read $m < n$ instances of r, and then a prefix of the next instance of r.
   (c) $\text{prefixes}(r^+) = r^* \cdot \text{prefixes}(r)$. Same as previous. Why does the empty case still appear?

2. The prefix computations are:

   (a) $\text{prefixes}(ab^*c) = \epsilon \mid a \mid ab^*(b \mid c \mid \epsilon)$. Computation:

$$\begin{aligned}
\text{prefixes}(ab^*c) &= \text{prefixes}(a) \mid a \cdot \text{prefixes}(b^*c) && \text{[concatenation]} \\
&= (a \mid \epsilon) \mid a \cdot \text{prefixes}(b^*c) && [a] \\
&= (a \mid \epsilon) \mid a \cdot (\text{prefixes}(b^*) \mid b^*\, \text{prefixes}(c)) && \text{[concatenation]} \\
&= (a \mid \epsilon) \mid a \cdot (\text{prefixes}(b^*) \mid b^*(c \mid \epsilon)) && [c] \\
&= (a \mid \epsilon) \mid a \cdot (b^*\, \text{prefixes}(b) \mid b^*(c \mid \epsilon)) && \text{[star]} \\
&= (a \mid \epsilon) \mid a \cdot (b^*(b \mid \epsilon) \mid b^*(c \mid \epsilon)) && [b] \\
&= (a \mid \epsilon) \mid a \cdot (b^*(b \mid c \mid \epsilon)) && \text{[rewrite]} \\
&= \epsilon \mid a \mid a \cdot (b^*(b \mid c \mid \epsilon)) && \text{[rewrite]}
\end{aligned}$$

   (b) $\text{prefixes}((a \mid bc)^*) = (a \mid bc)^*(\epsilon \mid a \mid b \mid bc)$.

□

Exercise 2 Compute nullable, first, and follow for the non-terminals A and B in the following grammar:

$$\begin{aligned}
A &::= BAa \\
A &::= \\
B &::= bBc \\
B &::= AA
\end{aligned}$$

Remember to extend the language with an extra start production for the computation of follow.

Solution

1. nullable: we get the constraints

$$\begin{aligned}
\text{nullable}(A) &= \text{nullable}(BAa) \vee \text{nullable}(\epsilon) \\
\text{nullable}(B) &= \text{nullable}(bBc) \vee \text{nullable}(AA)
\end{aligned}$$

   We can solve these to get $\text{nullable}(A) = \text{nullable}(B) = \mathit{true}$.

2. first: we get the constraints (given that both A and B are nullable):

$$\begin{aligned}
\text{first}(A) &= \text{first}(BAa) \cup \text{first}(\epsilon) \\
&= \text{first}(B) \cup \text{first}(A) \cup \emptyset \\
&= \text{first}(B) \cup \text{first}(A) \\
\text{first}(B) &= \text{first}(bBc) \cup \text{first}(AA) \\
&= \{b\} \cup \text{first}(A) \cup \text{first}(A) \cup \emptyset \\
&= \{b\} \cup \text{first}(A)
\end{aligned}$$

   Starting from $\text{first}(A) = \text{first}(B) = \emptyset$, we iteratively compute the fixpoint to get $\text{first}(A) = \text{first}(B) = \{a, b\}$.

3. follow: we add a production $A' ::= A$ **EOF**, and get the constraints (in order of productions):

$$\{\mathbf{EOF}\} \subseteq \text{follow}(A)$$

$$\begin{aligned} \text{first}(A) &\subseteq \text{follow}(B) \\ \{a\} &\subseteq \text{follow}(A) \end{aligned}$$

$$\{c\} \subseteq \text{follow}(B)$$

$$\begin{aligned} \text{first}(A) &\subseteq \text{follow}(A) \\ \text{follow}(B) &\subseteq \text{follow}(A) \end{aligned}$$

Substituting the computed first sets, and computing a fixpoint, we get $\text{follow}(A) = \{a, b, c, \mathbf{EOF}\}$ and $\text{follow}(B) = \{a, b, c\}$.

□

Exercise 3 Given the following grammar for arithmetic expressions:

$$\begin{aligned} S &::= Exp\ \mathbf{EOF} \\ Exp &::= Term\ Add \\ Add &::= +\ Term\ Add \\ Add &::= -\ Term\ Add \\ Add &::= \\ Term &::= Factor\ Mul \\ Mul &::= *\ Factor\ Mul \\ Mul &::= /\ Factor\ Mul \\ Mul &::= \\ Factor &::= \mathbf{num} \\ Factor &::= (Exp) \end{aligned}$$

1. Compute nullable, first, follow for each of the non-terminals in the grammar.
2. Check if the grammar is LL(1). If not, modify the grammar to make it so.
3. Build the LL(1) parsing table for the grammar.
4. Using your parsing table, parse or attempt to parse (till error) the following strings, assuming that **num** matches any natural number:
   (a) $(3 + 4) * 5$ **EOF**
   (b) $2 + +$ **EOF**
   (c) 2 **EOF**
   (d) $2 * 3 + 4$ **EOF**
   (e) $2 + 3 * 4$ **EOF**

Solution

1. We can compute the nullable, first, and follow sets as:

   (a) nullable:

$$
\begin{aligned}
\text{nullable}(S) &= false \\
\text{nullable}(Exp) &= false \\
\text{nullable}(Add) &= true \\
\text{nullable}(Term) &= false \\
\text{nullable}(Mul) &= true \\
\text{nullable}(Factor) &= false
\end{aligned}
$$

   (b) first: we have constraints:

$$
\begin{aligned}
\text{first}(S) &= \text{first}(Exp) \\
\text{first}(Exp) &= \text{first}(Term) \\
\text{first}(Add) &= \{+\} \cup \{-\} \cup \emptyset \\
\text{first}(Term) &= \text{first}(Factor) \\
\text{first}(Mul) &= \{*\} \cup \{/\} \cup \emptyset \\
\text{first}(Factor) &= \{\textbf{num}\} \cup \{(\}
\end{aligned}
$$

   which can be solved to get:

$$
\begin{aligned}
\text{first}(S) &= \{\textbf{num}, (\} \\
\text{first}(Exp) &= \{\textbf{num}, (\} \\
\text{first}(Add) &= \{+, -\} \\
\text{first}(Term) &= \{\textbf{num}, (\} \\
\text{first}(Mul) &= \{*, /\} \\
\text{first}(Factor) &= \{\textbf{num}, (\}
\end{aligned}
$$

   (c) follow: we have constraints (for each rule, except empty/terminal rules):

$$
\begin{aligned}
\{\textbf{EOF}\} &\subseteq \text{follow}(Exp) \\
\\
\text{first}(Add) &\subseteq \text{follow}(Term) \\
\text{follow}(Exp) &\subseteq \text{follow}(Term) \\
\text{follow}(Exp) &\subseteq \text{follow}(Add) \\
\\
\text{first}(Add) &\subseteq \text{follow}(Term) \\
\text{follow}(Add) &\subseteq \text{follow}(Term)
\end{aligned}
$$

$$
\begin{aligned}
\text{first}(Add) &\subseteq \text{follow}(Term) \\
\text{follow}(Add) &\subseteq \text{follow}(Term) \\
\\
\text{first}(Mul) &\subseteq \text{follow}(Factor) \\
\text{follow}(Term) &\subseteq \text{follow}(Factor) \\
\text{follow}(Term) &\subseteq \text{follow}(Mul) \\
\\
\text{first}(Mul) &\subseteq \text{follow}(Factor) \\
\text{follow}(Mul) &\subseteq \text{follow}(Factor)
\end{aligned}
$$

$$\text{first}(Mul) \subseteq \text{follow}(Factor) \qquad \{)\} \subseteq \text{follow}(Exp)$$
$$\text{follow}(Mul) \subseteq \text{follow}(Factor)$$

The fixpoint can again be computed to get:

$$\begin{aligned}
\text{follow}(S) &= \{\} \\
\text{follow}(Exp) &= \{), \textbf{EOF}\} \\
\text{follow}(Add) &= \{), \textbf{EOF}\} \\
\text{follow}(Term) &= \{+, -,), \textbf{EOF}\} \\
\text{follow}(Mul) &= \{+, -,), \textbf{EOF}\} \\
\text{follow}(Factor) &= \{+, -, *, /,), \textbf{EOF}\}
\end{aligned}$$

2. The grammar is LL(1), there are no conflicts. Demonstrated by the parsing table below.

3. LL(1) parsing table:

	num	$+$	$-$	$*$	$/$	(	)	**EOF**
S	1					1		
Exp	1					1		
Add		1	2				3	3
$Term$	1					1		
Mul		3	3	1	2		3	3
$Factor$	1					2		

4. Parsing the strings:

   (a) $(3 + 4) * 5$ **EOF** ✓

   (b) $2 + +$ **EOF** — fails on the second $+$. The corresponding error cell in the parsing table is $(Term, +)$.

   (c) 2 **EOF** ✓

   (d) $2 * 3 + 4$ **EOF** ✓

   (e) $2 + 3 * 4$ **EOF** fails on the $*$. Error at $(Add, *)$.

   Example step-by-step LL(1) parsing state for $2 * 3 + 4$:

Lookahead	Stack	Next Rule
2	S	$S ::= Exp\ \mathbf{EOF}$
2	$Exp\ \mathbf{EOF}$	$Exp ::= Term\ Add$
2	$Term\ Add\ \mathbf{EOF}$	$Term ::= Factor\ Mul$
2	$Factor\ Mul\ Add\ \mathbf{EOF}$	$Factor ::= \mathbf{num}$
2	$\mathbf{num}\ Mul\ Add\ \mathbf{EOF}$	$match(\mathbf{num})$
$*$	$Mul\ Add\ \mathbf{EOF}$	$Mul ::= *\ Factor\ Mul$
$*$	$*\ Factor\ Mul\ Add\ \mathbf{EOF}$	$match(*)$
3	$Factor\ Mul\ Add\ \mathbf{EOF}$	$Factor ::= \mathbf{num}$
3	$\mathbf{num}\ Mul\ Add\ \mathbf{EOF}$	$match(\mathbf{num})$
$+$	$Mul\ Add\ \mathbf{EOF}$	$Mul ::=$
$+$	$Add\ \mathbf{EOF}$	$Add ::= +\ Term\ Add$
$+$	$+\ Term\ Add\ \mathbf{EOF}$	$match(+)$
4	$Term\ Add\ \mathbf{EOF}$	$Term ::= Factor\ Term*$
4	$Factor\ Mul\ Add\ \mathbf{EOF}$	$Factor ::= \mathbf{num}$
4	$\mathbf{num}\ Mul\ Add\ \mathbf{EOF}$	$match(\mathbf{num})$
EOF	$Mul\ Add\ \mathbf{EOF}$	$Mul ::=$
EOF	$Add\ \mathbf{EOF}$	$Add ::=$
EOF	$\mathbf{EOF}$	$match(\mathbf{EOF})$

□

Exercise 4 Argue that the following grammar is *not* LL(1). Produce an equivalent LL(1) grammar.

$$E ::= \mathbf{num} + E \mid \mathbf{num} - E \mid \mathbf{num}$$

Solution The language is clearly not LL(1), as on seeing a token **num**, we cannot decide whether to continue parsing it as $\mathbf{num} + E$, $\mathbf{num} - E$, or the end.

The notable problem is the common prefix between the rules. We can separate this out by introducing a new non-terminal T. This is a transformation known as *left factorization*.

$$E ::= \mathbf{num}\ T$$
$$T ::= +E \mid -E \mid \epsilon$$

□

Exercise 5 Consider the following grammar:

$$S ::= S(S) \mid S[S] \mid () \mid [\,]$$

Check whether the same transformation as the previous case can be applied to produce an LL(1) grammar. If not, argue why, and suggest a different transformation.

Solution Applying left factorization to the grammar, we get:

$$S ::= S\ T \mid S\ T \mid () \mid [\,]$$
$$T ::= (S) \mid [S]$$

This is not LL(1), as on reading a token "(", we cannot decide whether this is the final parentheses (base case) in the expression, or whether there is a T following it.

The problem is that this version of the grammar is left-recursive. A recursive-descent parser for this grammar would loop forever on the first rule. This is caused by the fact that our parsers are top-down, left to right. We can fix this by *moving* the recursion to the right. This is generally called *left recursion elimination*.

Transformed grammar steps (explanation below):

$$S ::= ()S' \mid [\,]S'$$
$$S' ::= (S)S' \mid [S]S' \mid \epsilon$$

To eliminate left-recursion in general, consider a non-terminal $A ::= A\alpha \mid \beta$, where β does not start with A (not left-recursive). We can remove the left recursion by introducing a new non-terminal, A', such that:

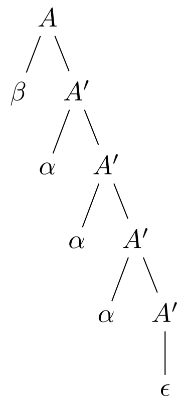
$$A ::= A' \mid \beta A'$$
$$A' ::= \alpha A' \mid \epsilon$$

i.e., for the left-recursive rule $A\alpha$, we instead attempt to parse an α followed by the rest. In exchange, the base case β now expects an A' to follow it. Note that β can be empty as well.

Intuitively, we are shifting the direction in which we look for instances of A. Consider a partial derivation starting from $\beta\alpha\alpha\alpha$. The original version of the grammar would complete the parsing as:

```
         A
        / \
       A   α
      / \
     A   α
    / \
   A   α
   |
   β
```

but with the new grammar, we parse it as:



There are two main pitfalls to remember with left-recursion elimination:

1. it may need to be applied several times till the grammar is unchanged, as the first transformation may introduce new (indirect) recursive rules (check $A ::= AA\alpha \mid \epsilon$).

2. it may require *inlining* some non-terminals, when the left recursion is *indirect*. For example, consider $A ::= B\alpha, B ::= A\beta$, where there is no immediate reduction to do, but inlining B, we get $A ::= A\beta\alpha$, where the elimination can be applied.

$\square$