CS 164 Programming Languages and Compilers **Spring 2019**

Discussion 2/18: LL Parsing and Earley's Algorithm

Instructor: Paul N. Hilfinger GSIs: Nikhil Athreya, Vivant Sakore

1. **LL Parsing Ambiguities.** An LL(k) grammar is a CFG used by a parser that scans input left-to-right ("L"), leftmost derivation ("L"), and uses k tokens of lookahead to predict the correct production. We've previously seen that a grammar is ambiguous if it has a parse tree that is not unique. A more formal definition of LL conflicts uses FIRST and FOLLOW sets.

   - **FIRST(A)**: the set of all terminals that could occur first in an expansion of the terminal or nonterminal A (include ϵ if A can expand to ϵ)
   - **FOLLOW(A)**: the set of all terminals that could follow an occurrence of the terminal or nonterminal A in a (partial) derivation

   There are two main types of LL(1) conflicts:

   - **FIRST/FIRST**: The FIRST sets of two different productions for same non-terminal intersect.
   - **FIRST/FOLLOW**: The FIRST set of a grammar rule contains an epsilon and the intersection with its FOLLOW set is not empty.

   Is the following grammar LL(1)? Justify your answer using FIRST and FOLLOW sets.

   - $S \to Xd \qquad X \to C \mid Ba \qquad C \to \epsilon \qquad B \to d$

   Answer:

   This is an instance of a FIRST/FOLLOW conflict. FIRST(`X`) contains the empty string and the intersection of FIRST(`X`) and FOLLOW(`X`) is not empty:
   FIRST(`S`) = FIRST(`X d`) = {`'d'`} FIRST(`X`) = FIRST(`C`)∪FIRST(`B a`) FIRST(`C`) = $\{\epsilon\}$

   FIRST(`B a`) = {`'d'`} FIRST(`B`) = {`'d'`}

   FOLLOW(`S`) = {} FOLLOW(`X`) = {`'d'`} FOLLOW(`C`) = {`'d'`} FOLLOW(`B`) = {`'a'`}

2. **Resolving Conflicts.**

   Consider the following grammar for numerical expressions with division, addition, and unary minus:

- $E \rightarrow Num \mid E/E \mid E + E \mid -E$

(a) Rewrite the grammar so that it is LL(1), so that '/' has higher precedence than '+', and so that '-' has highest precedence. '+' and '/' should be parsed in a right-associative way.

Answer:

```
expr : expr1 rest        expr1 : expr2 rest1        expr2 : '-' expr2
                                                           | NUM
rest : ϵ                 rest1 : ϵ
     | '+' expr                  | '/' expr1
```

(b) Compute the FIRST and FOLLOW sets for your re-written LL(1) grammar.

Answer:

$$\text{FIRST}(\texttt{expr1 rest}) = \text{FIRST}(\texttt{expr2 rest1}) = \{\texttt{'-'}, \texttt{NUM}\}$$

$$\text{FIRST}(\texttt{'+' expr}) = \{\ \texttt{'+'}\ \} \qquad \text{FIRST}(\texttt{'/' expr1}) = \{\ \texttt{'/'}\ \}$$
$$\text{FIRST}(\texttt{'-' expr2}) = \{\ \texttt{'-'}\ \} \qquad \text{FIRST}(\texttt{NUM}) = \{\ \texttt{NUM}\ \}$$
$$\text{FIRST}(\epsilon) = \{\ \epsilon\ \}$$

$$\text{FOLLOW}(\texttt{expr2}) = \{\ \texttt{'/'}, \texttt{'+'}, \dashv\ \}$$
$$\text{FOLLOW}(\texttt{expr1}) = \text{FOLLOW}(\texttt{rest1}) = \{\ \texttt{'+'}, \dashv\ \}$$
$$\text{FOLLOW}(\texttt{expr}) = \text{FOLLOW}(\texttt{rest}) = \{\ \dashv\ \}$$

(c) Draw the LL(1) parsing table for the grammar. You may need the following rules:

- For each production $X \rightarrow A_1 ... A_n$:
  - For each $1 \leq i \leq n$, and for each b in **First**(A_i): Set $T[X, b] = X \rightarrow A_1 ... A_n$. Stop when ϵ is not in **First**(A_i).
  - If $A_1 ... A_n \rightarrow^* \epsilon$, then for each b in **Follow**(X): Set $T[X, b] = \epsilon$.

Answer:

	$-$	NUM	$/$	$+$	$\dashv$
expr	expr → expr1 rest	expr → expr1 rest			
rest				rest → +expr	ϵ
expr1	expr1 → expr2 rest1	expr1 → expr2 rest1			
rest1			rest1 → / expr1	ϵ	ϵ
expr2	expr2 → - expr2	expr2 → NUM			

Here is an example trace, using the above table for parsing the string "6+4/4 ⊣".

$$
\begin{aligned}
\text{expr} &= 6 + 4/4 \dashv \\
\text{expr1 rest} &= 6 + 4/4 \dashv \\
\text{expr2 rest1 rest} &= 6 + 4/4 \dashv \\
\text{NUM rest1 rest} &= 6 + 4/4 \dashv \\
\text{rest1 rest} &= +4/4 \dashv \\
\text{rest} &= +4/4 \dashv \\
+ \text{ expr} &= +4/4 \dashv \\
\text{expr} &= 4/4 \dashv \\
\text{expr1 rest} &= 4/4 \dashv \\
\text{expr2 rest1 rest} &= 4/4 \dashv \\
\text{NUM rest1 rest} &= 4/4 \dashv \\
\text{rest1 rest} &= /4 \dashv \\
/ \text{ expr1 rest} &= /4 \dashv \\
\text{expr1 rest} &= 4 \dashv \\
\text{expr2 rest1 rest} &= 4 \dashv \\
\text{NUM rest1 rest} &= 4 \dashv \\
\text{rest1 rest} &= \dashv \\
\text{rest} &= \dashv \\
\text{EMPTY} &= \dashv
\end{aligned}
$$

3. **Earley's Algorithm.**

- Consider the following grammar:

$$
\begin{aligned}
P &\rightarrow E \dashv \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow ID
\end{aligned}
$$

Use Earley's algorithm to parse `ID + ID * ID` by filling out the chart below (you may not need all of the rows).

	ID		+	ID	*	ID
	0	1	2	3	4	5
a						
b						
c						
d						
e						
f						
g						
h						
i						
j						
k						
l						
m						

Answer: A solution has been posted in the `extra` directory.

- Draw the accepting parse tree(s) and identify whether there are ambiguities in the grammar.

  Answer: A solution has been posted in the `extra` directory.