CSCI 742 - Compiler Construction

Lecture 16

Implementing Recursive-Descent Parsers

Instructor: Hossein Hojjat

March 1, 2017

Recap: Conversion to LL(1)

- Consider the following grammar for additive expressions:

$$S \to E$$
$$E \to E + F \mid \texttt{num}$$
$$F \to (E) \mid \texttt{num}$$

- Grammar has left-recursion so is not LL(1)
- Grammar is LL(1) after removing left-recursion:

$$S \to E$$
$$E \to \texttt{num}\ E'$$
$$E' \to +FE' \mid \epsilon$$
$$F \to (E) \mid \texttt{num}$$

Predictive Parser Implementation

$$S \to E$$
$$E \to \texttt{num}\ E'$$
$$E' \to +FE' \mid \epsilon$$
$$F \to (E) \mid \texttt{num}$$

- Predictive parsing table:

	num	+	(	)	$
S	$\to E$				
E	$\to \texttt{num}\ E'$				
E'		$\to +FE'$		$\to \epsilon$	$\to \epsilon$
F	$\to \texttt{num}$		$\to (E)$		

- Implement a recursive descent parser using the mutually recursive procedures `parse_S`, `parse_E`, `parse_E'` and `parse_F`

Construct Parsing Tables

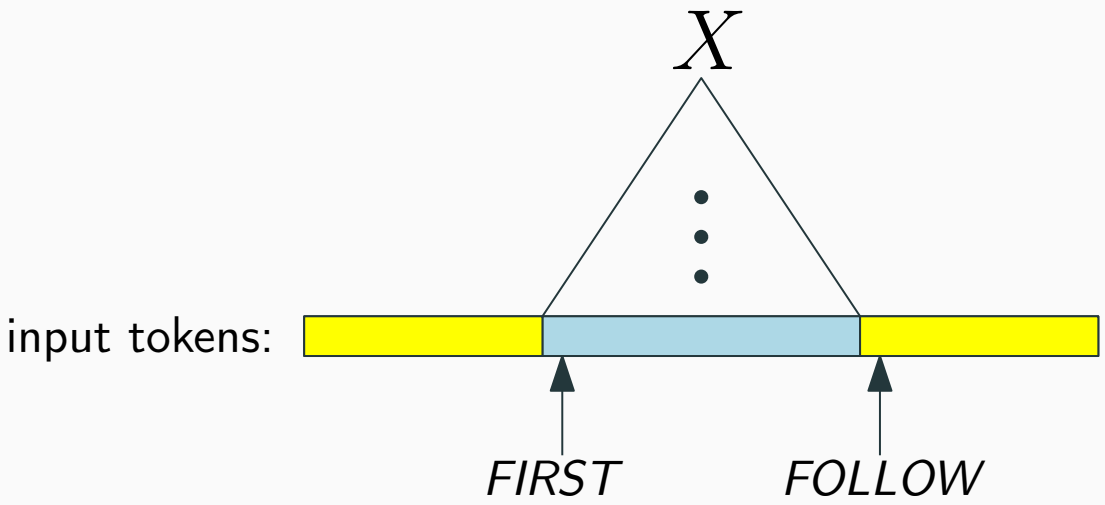
- Need an algorithm to generate predictive parse tables from an LL(1) grammar

$$S \to E$$
$$E \to \texttt{num}\ E'$$
$$E' \to +FE' \mid \epsilon$$
$$F \to (E) \mid \texttt{num}$$

	num	+	(	)	$
S	$\to E$				
E	$\to \texttt{num}\ E'$				
E'		$\to +FE'$		$\to \epsilon$	$\to \epsilon$
F	$\to \texttt{num}$		$\to (E)$		

Prerequisite Definitions

- Parsing table determines for each non-terminal and each look-ahead symbol what one production to use
- *FIRST*(γ) for arbitrary string of terminals and non-terminals γ is: set of symbols that might begin the fully expanded version of γ
- *FOLLOW*(X) for a non-terminal X is:
  set of symbols that might follow the derivation of X in the input stream



Parse Table Entries

- Consider a production $X \to \gamma$
- Add $\to \gamma$ to the X row for each symbol in *FIRST*(γ)
- If γ can derive ϵ (γ is nullable), add $\to \gamma$ for each symbol in *FOLLOW*(X)
- Grammar is LL(1) if no conflicting entries

$S \to E$
$E \to \texttt{num}\ E'$
$E' \to +FE' \mid \epsilon$
$F \to (E) \mid \texttt{num}$

	num	+	(	)	$
S	$\to E$				
E	$\to \texttt{num}\ E'$				
E'		$\to +FE'$		$\to \epsilon$	$\to \epsilon$
F	$\to \texttt{num}$		$\to (E)$		

Computing nullable

- X is nullable if it can derive the empty string
- If it derives ϵ directly ($X \to \epsilon$)
- If it has a production $X \to Y_1 Y_2 \cdots Y_n$ where all RHS symbols (Y_1, Y_2, $\cdots$, Y_n) are nullable
- **Algorithm:** assume all non-terminals non-nullable, apply rules repeatedly until no change in status

Computing *FIRST*

Definition. $\mathit{FIRST}(\gamma) = \{a \mid \gamma \Rightarrow^* a\beta\} \cup \{\epsilon \mid \gamma \Rightarrow^* \epsilon\}$

1. $\mathit{FIRST}(a) = \{a\}$

2. For all productions $X \to Y_1 \cdots Y_n$

- Add $\mathit{FIRST}(Y_1) - \{\epsilon\}$ to $\mathit{FIRST}(X)$. Stop if $\epsilon \notin \mathit{FIRST}(Y_1)$
- Add $\mathit{FIRST}(Y_2) - \{\epsilon\}$ to $\mathit{FIRST}(X)$. Stop if $\epsilon \notin \mathit{FIRST}(Y_2)$
- $\cdots$
- Add $\mathit{FIRST}(Y_n) - \{\epsilon\}$ to $\mathit{FIRST}(X)$. Stop if $\epsilon \notin \mathit{FIRST}(Y_n)$
- Add ϵ to $\mathit{FIRST}(X)$

Exercise

Question.

- Compute the *FIRST* sets for non-terminals

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \texttt{num}
\end{aligned}
$$

Exercise

Question.

- Compute the *FIRST* sets for non-terminals

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \texttt{num}
\end{aligned}
$$

Answer.

$$
\begin{aligned}
\mathit{FIRST}(E) &= \{\ (\ ,\ \texttt{num}\ \} & \mathit{FIRST}(E') &= \{+\ ,\ \epsilon\} \\
\mathit{FIRST}(T) &= \{\ (\ ,\ \texttt{num}\ \} & \mathit{FIRST}(T') &= \{*\ ,\ \epsilon\}
\end{aligned}
$$

$$
\mathit{FIRST}(F) = \{\ (\ ,\ \texttt{num}\}
$$

Computing *FOLLOW*

Definition. $\mathit{FOLLOW}(X) = \{a \mid S \Rightarrow^* \beta X a \gamma\}$

1. Compute the *FIRST* sets for all non-terminals first
2. Add \$ to $\mathit{FOLLOW}(S)$ (S is the start non-terminal)
3. For all productions $Z \to \cdots X Y_1 \cdots Y_n$
   - Add $\mathit{FIRST}(Y_1) - \{\epsilon\}$ to $\mathit{FOLLOW}(X)$. Stop if $\epsilon \notin \mathit{FIRST}(Y_1)$
   - Add $\mathit{FIRST}(Y_2) - \{\epsilon\}$ to $\mathit{FOLLOW}(X)$. Stop if $\epsilon \notin \mathit{FIRST}(Y_2)$
   - $\cdots$
   - Add $\mathit{FIRST}(Y_n) - \{\epsilon\}$ to $\mathit{FOLLOW}(X)$. Stop if $\epsilon \notin \mathit{FIRST}(Y_n)$
   - Add $\mathit{FOLLOW}(Z)$ to $\mathit{FOLLOW}(X)$

Exercise

Question.

- Compute the *FOLLOW* sets for non-terminals

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \texttt{num}
\end{aligned}
$$

Exercise

Question.

- Compute the *FOLLOW* sets for non-terminals

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \texttt{num}
\end{aligned}
$$

Answer.

$$\mathit{FOLLOW}(E) = \{\ )\ ,\ \$\} \qquad \mathit{FOLLOW}(E') = \{\ )\ ,\ \$\}$$
$$\mathit{FOLLOW}(T) = \{\ )\ ,\ +\ ,\$\} \qquad \mathit{FOLLOW}(T') = \{\ )\ ,\ +\ ,\$\}$$
$$\mathit{FOLLOW}(F) = \{\ )\ ,\ *\ ,\ +\ ,\$\}$$

Ambiguous Grammars

- Construction of predictive parse table for ambiguous grammar results in conflicts
  - but converse does not hold

$$S \to S + S \mid S * S \mid \mathtt{num}$$

- $\mathit{FIRST}(S + S) = \mathit{FIRST}(S * S) = \mathit{FIRST}(\mathtt{num}) = \{\mathtt{num}\}$

	num	+	$
S	$\to \mathtt{num}, \to S + S, \to S * S$		

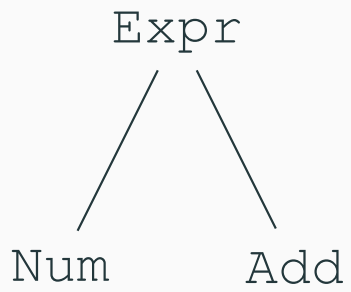
Completing the Parser

- Now we know how to construct a recursive-descent parser for an LL(1) grammar
- LL(k) generalizes this to k lookahead tokens
- LL(k) parser generators can be used to automate the process
  - (e.g. ANTLR)
- Can we use recursive descent to build an abstract syntax tree too?

Creating the AST

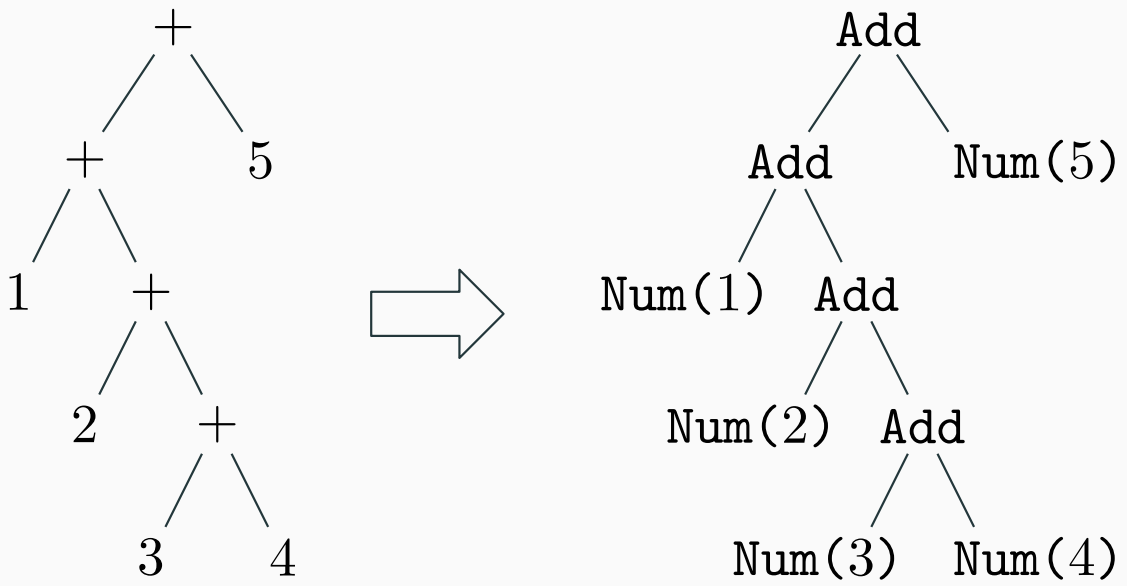
$$
\begin{aligned}
S &\rightarrow ES' \\
S' &\rightarrow \epsilon \mid +S \\
E &\rightarrow \text{num} \mid (S)
\end{aligned}
$$

```
abstract class Expr { }
class Add extends Expr {
  Expr left, right;
  Add(Expr L, Expr R) {
    left = L; right = R;
  }
}
class Num extends Expr {
  int value;
  Num (int v) { value = v);
}
```



AST Representation

input: `(1 + 2 + (3 + 4)) + 5`



How can we generate the AST during recursive descent parsing?

Creating the AST

- Just add code to each parsing procedure to create the appropriate nodes
- Works because parse tree and call tree have same shape
- `parse_S`, `parse_S'`, `parse_E` all return an `Expr`:

```
void parse_E()        ⇒        Expr parse_E()
void parse_S()        ⇒        Expr parse_S()
void parse_S'()       ⇒        Expr parse_S'()
```

AST Creation: `parse_E`

$$
\begin{aligned}
S &\to ES' \\
S' &\to \epsilon \mid +S \\
E &\to \text{num} \mid (S)
\end{aligned}
$$

```
Expr parse_E() {
    switch(token) {
      case num: // E → num
        Expr result = Num (token.value);
        token = input.read();
        return result;
      case '(': // E → ( S )
        token = input.read();
        Expr result = parse_S();
        if (token != ')') throw new ParseError();
        token = input.read(); return result;
      default: throw new ParseError();
    }
```

AST Creation: `parse_E`

$$
\begin{aligned}
S &\to ES' \\
S' &\to \epsilon \mid +S \\
E &\to \text{num} \mid (S)
\end{aligned}
$$

```
Expr parse_S() {
  switch (token) {
    case num:
    case '(':
      Expr left = parse_E();
      Expr right = parse_S'();
      if (right == null) return left;
        else return new Add(left, right);
    default: throw new ParseError();
  }
}
```