Recursive Descent Grammars
Outline

**Topics and Learning Objectives**

• Discuss grammar ambiguity, removing left recursions, and left-factoring grammars

**Assessments**

• Left recursion and left-factoring activity
Convert Grammar Into Source Code

1. Each rule defined in the grammar becomes a function/method.
2. References to that rule become a function call.
3. The body of a rule function follows the flow of the rule’s RHS.
4. Alternatives (a1 | a2 | ...) are handled with branches (if-then-else, switch statements, match expressions, etc.).
5. An optional grouping (...)* becomes a while loop that can loop zero or more times.
6. Epsilon transitions enable a function to return without doing anything.
7. Each reference to a terminal (token) requires you to check that the current token is correct, and then either report an error or advance to the next token.
This is a recognizer, not a parser. It does not output a tree. It just checks for valid input.
LL(k) Parsing

- Top-down approach
- Uses at most the next k tokens to select a production (look-ahead)
- Efficient, predictive parsing (no backtracking → linear time parsing)
- Implemented via recursive functions calls (ad-hoc, by hand)

- Or implemented using a table-driven approach (generated)
  - A stack is used in place of recursive function calls
  - Implement a push-down automaton
  - We’ll not be discussing this topic
Summary of LL Parsing

- Write Grammar
- Remove Ambiguity
- Remove Left Recursion
- Left Factor
- Implement the LL(k) Parser

Decide on an LL(k) parser

Top-Down Parser
Better Top-Down Parser
Recursive Descent Parser
Predictive Parser
Grammar Ambiguity

\[
\begin{align*}
expr & \rightarrow \ id \mid number \mid - \ expr \mid ( \ expr ) \\
& \mid expr \ op \ expr \\
expr & \rightarrow \ term \\
term & \rightarrow \ factor \mid term \ mult\_op \ factor \\
factor & \rightarrow \ id \mid number \mid - \ factor \mid ( \ expr ) \\
add\_op & \rightarrow \ + \mid - \\
mult\_op & \rightarrow \ * \mid / \\
\end{align*}
\]
Dangling Else in C

```c
int main() {
    int x;
    if (true) {
        if (false) {
            x = 5;
        }
        else {
            x = 4;  // dangling else
        }
    }
    else {
        x = 4;  // dangling else
    }
}
```
Dangling Else

What if `s1` is an `if`-statement?

We could parse this as either of the following:

C handles this problem by the following convention: an `else` always associates with the nearest `if`. 

C-ish Grammar:

```plaintext
statement : assignment ifStatement |;
assignment : 'let' Identifier '=' expression ';';
ifStatement : 'if' '(' expression ')' statement
           | 'if' '(' expression ')' statement 'else' statement;
```
statement
  : assignment | ifStatement | ...;

ifStatement
  : 'if' '(', expression ')', statement
  | 'if' '(', expression ')', statement 'else' statement;

Input: ‘if’ ‘(‘ expr₁ ‘)’ ‘if’ ‘(‘ expr₂ ‘)’ stmt₁ 'else' stmt₂
statement
  : assignment | ifStatement | ...;

assignment
  : 'let' Identifier '=%' expression ';';

ifStatement
  : matchedIf | unmatchedIf;

matchedIf
  : 'if' '(' expression ')' matchedIf 'else' matchedIf
    | statement;

unmatchedIf
  : 'if' '(' expression ')' statement
    | 'if' '(' expression ')' matchedIf 'else' unmatchedIf;
Grammar Ambiguity

• Grammar ambiguity can cause issues for the programmer using your language

• It can also cause problems for you while your writing your language (unexpected parse trees)

• Unfortunately, we cannot un-ambiguate a grammar automatically

• This problem is referred to as “undecidable”

• In practice though, it is possible to create unambiguous grammars by carefully examining your grammar rules
Summary of LL Parsing

1. Decide on an LL(k) parser
2. Write Grammar
3. Remove Ambiguity
4. Remove Left Recursion
5. Left Factor
6. Implement the LL(k) Parser
Left Recursion

expr : expr '+' term | term
term : ID | '(' expr ')'
Left Recursion

expr' : (Plus | Minus) term expr' | ε

FUNCTION ExprPrime()
    IF MatchInput(‘+’) || MatchInput(‘-’)
        ConsumeToken()
        Term()
        ExprPrime()
    ELSE
        return

FUNCTION Expr()
    TRY Expr()
    MatchInput(‘+’ || ‘-’)
        ConsumeToken()
        Term()
    ELSE
        Term()
Left Recursion

expr : expr '+' term | term
term : ID | ‘(‘ expr ‘)’
Left Recursion

\[
\text{expr} : \text{expr} \ ' + ' \ \text{term} \mid \text{term}
\]
\[
\text{term} : \text{ID} \mid \left( \left( ' \ \text{expr} \ ' \right) \right)
\]

Transformation Pattern

\[
A : \ A \ \alpha \mid \beta
\]
\[
A : \beta \ A'
\]
\[
A' : \alpha \ A' \mid \varepsilon
\]
Left Recursion

```
expr  :  expr \texttt{+} term | term
term  :  ID | \\
       \texttt{\textquotesingle\textquotesingle} expr \texttt{\textquotesingle\textquotesingle}
```

Transformation Pattern

```
A  :  A \alpha | \beta
A' : \beta A'
A' : \alpha A' | \epsilon
```
Left Recursion

expr : expr '+' term | term
term : ID | '(' expr ')'
Left Recursion

expr : expr ' +' term | term

term : ID | '(' expr ')'

Transformation Pattern

A : A α | β
A : β A'
A' : α A' | ε

What is α?
Left Recursion

expr : expr `+` term | term

term : ID | (`(` expr `)`)

Transformation Pattern

A : A α | β
A : β A'
A' : α A' | ε

What is β?
Left Recursion

expr : expr ‘+’ term | term
term : ID | ‘(‘ expr ‘)’

expr : term expr'
expr' : ‘+’ term expr'

Transformation Pattern
A : A α | β
A : β A'
A' : α A' | ε

What is A’?
Left Recursion

expr : expr ′+′ term | term

term : ID | ′(′ expr ′)′

expr : term expr′

expr′ : ′+′ term expr′ | ε

Transformation Pattern

A : A α | β
A′ : β A′
A′ : α A′ | ε
Left Recursion

\[
\begin{align*}
\text{expr} & : \text{expr} \ ' + ' \text{term} \mid \text{term} \\
\text{term} & : \text{ID} \mid \text{(' expr ')}
\end{align*}
\]

Copy non-left recursive rules without changing them.

Transformation Pattern

\[
\begin{align*}
A & : A \alpha \mid \beta \\
A & : \beta A' \\
A' & : \alpha A' \mid \epsilon
\end{align*}
\]
Left Recursion

General pattern for replacing left-recursion (removing more than one instance of left recursion)

\[ A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m \]

Is replaced with

\[ A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A' \]
\[ A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon \]
Indirect Left-Recursion

What about left recursion in the following grammar:

\[ S : Aa | b \]

\[ A : Ac | Sd | \epsilon \]

1. \( A \rightarrow A \text{, and} \)
2. \( S \rightarrow Ac \rightarrow Sd \)

---

### Left-Recursion Transformation

**Input Pattern:**

\[ A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m \]

**Output Pattern:**

\[ A \rightarrow \beta_1A' | \beta_2A' | \ldots | \beta_mA' \]

\[ A' \rightarrow \alpha_1A' | \alpha_2A' | \ldots | \alpha_nA' | \epsilon \]
Indirect Left-Recursion

What about left recursion in the following grammar:

\[
S : Aa | b \\
A : Ac | Sd | \varepsilon
\]

1. Label all non-terminals 1 through n
Indirect Left-Recursion

What about left recursion in the following grammar:

\[ S : Aa | b \]
\[ A : Ac | Sd | \varepsilon \]

1. Label all non-terminals 1 through \( n \)
2. Apply productions when the label on the RHS is less than LHS
Indirect Left-Recursion

What about left recursion in the following grammar:

\[ S : Aa | b \]
\[ A : Ac | S \]
\[ A : Ac | (Aa | b)d | \varepsilon \]

1. Label all non-terminals 1 through n
2. Apply productions when the label on the RHS is less than LHS
Indirect Left-Recursion

What about left recursion in the following grammar:

\[
\begin{align*}
S & : Aa \mid b \\
A & : Ac \mid (Aa \mid b)d \mid \epsilon \\
c & : Ac \mid Aa \\
d & : bd \mid \epsilon
\end{align*}
\]

1. Label all non-terminals 1 through n
2. Apply productions when the label on the RHS is less than LHS
Indirect Left-Recursion

What about left recursion in the following grammar:

\[
S : Aa \mid b \\
A : Ac \mid Aad \mid bd \mid \epsilon
\]

1. Label all non-terminals 1 through \( n \)
2. Apply productions when the label on the RHS is less than LHS
Indirect Left-Recursion

What about left recursion in the following grammar:

\[
S : Aa | b \\
A : Ac | Aad | bd | \epsilon
\]

1. Label all non-terminals 1 through \( n \)
2. Apply productions when the label on the RHS is less than LHS
3. Remove left recursions
Indirect Left-Recursion

What about left recursion in the following grammar:

\[
\begin{align*}
S & : Aa \mid b \\
A & : bdA' \mid A' \\
A' & : cA' \mid adA' \mid \varepsilon
\end{align*}
\]

1. Label all non-terminals 1 through n
2. Apply productions when the label on the RHS is less than LHS
3. Remove left recursions

**Left-Recursion Transformation**

Input Pattern:

\[
A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m
\]

Output Pattern:

\[
A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_m A' \\
A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' | \varepsilon
\]
Summary of LL Parsing

1. Write Grammar
2. Remove Ambiguity
3. Remove Left Recursion
4. Left Factor
5. Implement the LL(k) Parser

Steps:
- Decide on an LL(k) parser
- Top-Down Parser
- Better Top-Down Parser
- Recursive Descent Parser
- Predictive Parser
Left-Factoring (Remove Common Prefixes)

We’d like all decisions to be made based on the current token (no backtracking required)

\[ A : \text{some\_terminal} \quad B \quad | \quad \text{some\_other\_terminal} \quad C \]

For example, how do we choose which branch to take for \( E \)?

\[ E : \text{int} \quad E \quad | \quad \text{int} \quad T \]

This would require a backtracking process if we later find that the first alternative was wrong.
Why Remove Common Prefixes?

A : ‘@’ B | ‘$’ C

FUNCTION A()
  IF MatchInput(‘@’)
    ConsumeToken()
  B()
  ELSE IF MatchInput(‘$’)
    ConsumeToken()
  C()

E : ‘#’ E | ‘#’ T

FUNCTION E()
  MatchInput(‘#’)
  ConsumeToken()
  TRY
    E()
  ELSE
    T()
Left-Factoring (Remove Common Prefixes)

To eliminate backtracking or multiple look-ahead, we need to eliminate common prefixes.

What is the common prefix for the following grammar rule?

\[ A : \alpha \beta_1 \mid \alpha \beta_n \mid \gamma \]
Left-Factoring (Remove Common Prefixes)

To eliminate backtracking or multiple look-ahead, we need to eliminate common prefixes.

What is the common prefix for the following grammar rule?

\[ A : \alpha \beta_1 | \alpha \beta_n | \gamma \]
Left-Factoring (Remove Common Prefixes)

To eliminate backtracking or multiple look-ahead, we need to eliminate common prefixes.

What is the common prefix for the following grammar rule?

\[ A : \alpha \beta_1 \mid \alpha \beta_n \mid \gamma \]

Add a new rule that is all possible suffixes for the given common prefix.

\[ A : \alpha A' \mid \gamma \]
\[ A' : \beta_1 \mid \beta_n \]
Left-Factoring (Remove Common Prefixes)

1. \( \text{Expr} : \text{Term} + \text{Expr} | \text{Term} \)

2. \( \text{Term} : \text{int} \times \text{Term} | \text{int} / \text{Term} | '(' \text{Expr} ')' \)

This leads to right associativity
Ensuring Look-Ahead of 1

S : Ab | Bc
A : Df | CA
B : gA | e
C : dC | c
D : h | i

• Capitals are **non-terminals**
• Lowercase are **terminals**

• Is this valid input: `gdchfc`?

Any left recursion?
Ensuring Look-Ahead of 1

\[ S^1 : Ab^2 | Bc^3 \]
\[ A^2 : Df^5 | CA^4 \]
\[ B : gA^2 | e \]
\[ C : dC^4 | c \]
\[ D^5 : h | i \]

- Capitals are **non-terminals**
- Lowercase are **terminals**

- Is this valid input: \textit{gdchfc}?

Any left recursion? **Nope**
Ensuring Look-Ahead of 1

\[ S : \text{Ab} \mid \text{Bc} \]
\[ A : \text{Df} \mid \text{CA} \]
\[ B : \text{gA} \mid \text{e} \]
\[ C : \text{dC} \mid \text{c} \]
\[ D : \text{h} \mid \text{i} \]
Ensuring Look-Ahead of 1

S : Ab | Bc  S : (Df | CA)b | (gA | e)c
A : Df | CA  A : Df | CA
B : gA | e  B : gA | e
C : dC | c  C : dC | c
D : h | i  D : h | i
Ensuring Look-Ahead of 1

S : Ab | Bc | S : Dfb | CAb | gAc | ec
A : Df | CA | A : Df | CA
B : gA | e | B : gA | e
C : dC | c | C : dC | c
D : h | i | D : h | i
Ensuring Look-Ahead of 1

S : Ab | Bc  S : Dfb | CAb | gAc | ec
A : Df | CA  A : Df | CA
B : gA | e  B : gA | e
C : dC | c  C : dC | c
D : h | i  D : h | i
Ensuring Look-Ahead of 1

\[
S : \text{Ab} \mid \text{Bc} \quad \quad S : (h \mid i)fb \mid (dC \mid c)Ab \mid gAc \mid ec \\
A : \text{Df} \mid \text{CA} \quad \quad A : \text{Df} \mid \text{CA} \\
B : \text{gA} \mid e \quad \quad B : \text{gA} \mid e \\
C : dC \mid c \quad \quad C : dC \mid c \\
D : h \mid i \quad \quad D : h \mid i
\]
Ensuring Look-Ahead of 1

Now we can easily pick an alternative for the S rule based on the current input.
Ensuring Look-Ahead of 1

FUNCTION \( S() \)

MATCH

'\( h \)' -> ...
'\( i \)' -> ...
'\( d \)' -> ...
'\( c \)' -> ...
'\( g \)' -> ...
'\( e \)' -> ...

\( S : Ab \mid Bc \)
\( A : Df \mid CA \)
\( B : gA \mid e \)
\( C : dC \mid c \)
\( D : h \mid i \)
Ensuring Look-Ahead of 1

S : Ab | Bc
A : Df | CA
B : gA | e
C : dC | c
D : h | i

S : hfb | ifb | dcAb | cAb | gAc | ec
A : (h | i)f | (dC | c)A
B : gA | e
C : dC | c
D : h | i
Ensuring Look-Ahead of 1

S : Ab | Bc
A : Df | CA
B : gA | e
C : dC | c
D : h | i

S : hfb | ifb | dcAb | cAb | gAc | ec
A : hf | if | dCA | cA
B : gA | e
C : dC | c
D : h | i

Do we need all rules?
Ensuring Look-Ahead of 1

S : Ab | Bc
A : Df | CA
B : gA | e
C : dC | c
D : h | i

S : hfb | ifb | dcAb | cAb | gAc | ec
A : hf | if | dCA | cA
B : gA | e
C : dC | c
D : h | i

Is this valid input: gdchfc?
Summary of LL Parsing

- Decide on an LL(k) parser
  - Write Grammar
  - Remove Ambiguity
  - Remove Left Recursion
  - Left Factor
- Implement the LL(k) Parser
  - Top-Down Parser
  - Better Top-Down Parser
  - Recursive Descent Parser
  - Predictive Parser