Goal of Lecture

- Character Stream
- Token Stream
- Parse Tree
- Abstract syntax tree
- Modified intermediate form
- Modified target language

This includes context-free grammar.

- Scanner (lexical analysis)
- Parser (syntax analysis)
- Semantic analysis & intermediate code gen.
- Machine-independent optimization (optional)
- Symbol Table
- Machine-specific optimization (optional)
Parsing

• The main task of parsing is to identify the syntax.
Review: Regular Expression Rules

• A RE consist of:
  • A character (e.g., “0”, “1”, …)
  • The empty string (i.e., “ε”)
  • Two REs next to each other (e.g., “non_negative_digit digit”) to denote concatenation.
  • Two REs separated by “|” next to each other (e.g., “non_negative_digit | digit”) to denote one RE or the other.
  • An RE followed by “*” (called the Kleene star) to denote zero or more iterations of the RE.
  • Parentheses (in order to avoid ambiguity).
A RE is NEVER defined in terms of itself! Thus, REs cannot define recursive statements.
Review: Regular Expression Rules

• A RE consist of:
  
  • A character (e.g., “0”, “1”, ...)
  
  • The empty string (i.e., “ε”)
  
  • Two REs next to each other (e.g., “non_negative_digit digit”) to denote concatenation.
  
  • Two REs separated by “|” next to each other (e.g., “non_negative_digit | digit”) to denote one RE or the other.
  
  • An RE followed by “*” (called the Kleene star) to denote zero or more iterations of the RE.
  
  • Parentheses (in order to avoid ambiguity).

For example, REs cannot define arithmetic expressions with parentheses.
Context-Free Grammars

- Context-Free Grammars (CFGs) are similar to REs except that they can handle recursion.

**Arithmetic expression with parentheses**

\[expr \rightarrow id | number | - expr | (expr) | expr \ op \ expr\]

\[op \rightarrow + | - | * | /\]
Context-Free Grammars

- Context-Free Grammars (CFGs) are similar to REs except that they can handle recursion.

Arithmetic expression with parentheses

\[
\text{expr} \rightarrow \text{id} | \text{number} | - \text{expr} | (\text{expr}) | \text{expr} \text{ op} \text{ expr}
\]

\[
\text{op} \rightarrow + | - | * | /
\]
Context-Free Grammars (CFGs) are similar to REs except that they can handle recursion.

Arithmetic expression with parentheses

\[ expr \rightarrow id | number | - expr | ( expr ) | expr \ op \ expr \]

\[ op \rightarrow + | - | * | / \]
Context-Free Grammars (CFGs) are similar to REs except that they can handle recursion.

\[
expr \rightarrow \text{id} | \text{number} | - expr | (expr) | expr \text{ op } expr
\]

\[
\text{op} \rightarrow + | - | * | /
\]

Non-terminals are symbols that are defined by the CFG and can appear on both the left and right side of "→".

Arithmetic expression with parentheses
Context-Free Grammars (CFGs) are similar to REs except that they can handle recursion.

**Arithmetic expression with parentheses**

\[ expr \rightarrow id | number | - expr | ( expr ) | expr \ op \ expr \]

\[ op \rightarrow + | - | * | / \]
Context-Free Grammars (CFGs) are similar to REs except that they can handle recursion.

Terminals are the strings that define the grammar and can only appear on the right side of "→".

Arithmetic expression with parentheses

```
expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /
```
Context-Free Grammars (CFGs) are similar to REs except that they can handle recursion.

\[
expr \rightarrow \text{id} | \text{number} | - \ expr | ( \ expr ) | expr \ op \ expr
\]

\[
\text{op} \rightarrow + | - | * | /
\]

Arithmetic expression with parentheses

In the book non-terminals are written in typewriter font, others write it in “normal” font.
• Technically, the Kleene star (*) and parentheses are not allowed under the CFG rules, called Backus-Naur Form (BNF).

• However, for convenience, we will use Extended BNF that includes the Kleene star, parentheses, and the Kleene Plus (+), which stands for “one or more iterations.”
EBNF Example.

• The Kleene star and parentheses can be written as follows

\[
id_list \rightarrow id (, \ id)^*
\]

\[
id_list \rightarrow id
\]

\[
id_list \rightarrow id_list, \ id
\]
Derivation

A derivation is “a series of replacement operations that derive a string of terminals from the start symbol.”

\[
slope * x + intercept \\
\Rightarrow expr \ op \ expr + id \\
\Rightarrow expr \ op \ id + id \\
\Rightarrow expr \ op \ id + id \\
\Rightarrow id \ * \ id + id \\
\Rightarrow (slope) \ * \ (x) + (intercept)
\]
A derivation is "a series of replacement operations that derive a string of terminals from the start symbol."

\[ \Rightarrow \text{denotes "derived from"} \]

\[ \text{slope} \times x + \text{intercept} \]

\[ \Rightarrow \text{expr op expr} + \text{id} \]
\[ \Rightarrow \text{expr op id} + \text{id} \]
\[ \Rightarrow \text{expr} \times \text{id} + \text{id} \]
\[ \Rightarrow \text{id} \times \text{id} + \text{id} \]
\[ \Rightarrow \text{expr} + \text{id} \]
\[ \Rightarrow \text{id} \times \text{id} + \text{id} \]
\[ \Rightarrow \text{expr op id} + \text{id} \]
\[ \Rightarrow \text{expr op expr} + \text{id} \]

\[ \text{(slope)} \times (x) + \text{(intercept)} \]
A derivation is "a series of replacement operations that derive a string of terminals from the start symbol." This derivation replaces the rightmost non-terminal. Derivations with this behavior are called (surprisingly) rightmost or canonical derivation.
Parse Tree

- A parse tree is the graphical representation of the derivation.
A parse tree is the graphical representation of the derivation.

This parse tree constructs the formula \((\text{slope} \times x) + \text{intercept}\).
Parse Tree

- A parse tree is the graphical representation of the derivation.
A parse tree is the graphical representation of the derivation.

Parse Tree

Let's try deriving “2*a*b+c”
Parse Tree (Ambiguous)

- This grammar is ambiguous and can construct the following parse tree.
• This grammar is ambiguous and can construct the following parse tree.

This is a leftmost derivation.
This grammar is ambiguous and can construct the following parse tree.

This parse tree constructs the formula \( \text{slope} \times (x + \text{intercept}) \) which is not equal to \( \text{slope} \times x + \text{intercept} \).
Parse Tree (Ambiguous)

- This grammar is ambiguous and can construct the following parse tree.

```
expr
  | expr
  | op
  |   | expr
  |   |   | id (slope)
  |   |   | *
  |   | expr
  |   |   | id (x)
  |   |   | +
  |   | expr
  |   |   | id (intercept)
```

Let's try deriving “2*a*b+c”
Disambiguating grammar

• The problem with our original grammar was that we did not fully express the grammatical structure (i.e., associativity and precedence).

• To create an unambiguous grammar, we need to fully specify the grammar.

\[
\begin{align*}
expr & \rightarrow \ term \mid expr\ add\_op\ term \\
term & \rightarrow \ factor \mid term\ mult\_op\ factor \\
factor & \rightarrow \ id \mid number \mid -\ factor \mid (\ expr ) \\
add\_op & \rightarrow + \mid - \\
mult\_op & \rightarrow * \mid /
\end{align*}
\]
Disambiguating grammar

- The problem with our original grammar was that we did not fully express the grammatical structure (i.e., associativity and precedence).
- To create an unambiguous grammar, we need to fully specify the grammar.

```
expr → term | expr add_op term

term → factor | term mult_op factor

factor → id | number | - factor | (expr)

add_op → + | -

mult_op → * | /
```
3 + 4 * 5

```
expr
  /\   
expr  add_op
   /\       
term  +      expr
   /\       /
factor term     mul_op
  /\       /
number(3) factor number(5)
```
Disambiguating grammar

• The problem with our original grammar was that it did not fully express the grammatical structure (i.e., associativity and precedence).

• To create an unambiguous grammar, we need to fully specify the grammar.

\[
\begin{align*}
\text{expr} \rightarrow & \quad \text{term} \mid \text{expr add_op term} \\
\text{term} \rightarrow & \quad \text{factor} \mid \text{term mult_op factor} \\
\text{factor} \rightarrow & \quad \text{id} \mid \text{number} \mid - \text{factor} \mid (\text{expr}) \\
\text{add_op} \rightarrow & \quad + \mid - \\
\text{mult_op} \rightarrow & \quad * \mid / 
\end{align*}
\]

Let's try deriving “3*4+5*6+7”
$3 \times 4 + 5 \times 6 + 7$
Disambiguating grammar

• The problem with our original grammar was that it did not fully express the grammatical structure (i.e., associativity and precedence).

• To create an unambiguous grammar, we need to fully specify the grammar.

```
expr → term | expr add_op term

term → factor | term mult_op factor

factor → id | number | - factor | ( expr )

add_op → + | -

mult_op → * | /
```
How can you derive these trees by examining one character at a time?
In order to derive these trees, the first character that we need to examine is the math symbol.
Thus, to parse these “sentences,” we first need to search through them to find the math symbols . . . then we need to sort out the multiplication from the addition. . . ugh. . .
Java Spec

• Available on-line

• Examples
LL and LR Derivation

• CFGs can be parsed in $O(n^3)$ time, where $n$ is length of the program.

• This is too long for most code; however, there are two types of grammars that can be parsed in linear time, i.e., $O(n)$.

  • **LL**: “Left-to-right, Left-most derivation”
  • **LR**: “Left-to-right, Right-most derivation”
Top-down

- **LL parsers** are top-down, i.e., they identify the non-terminals first and terminals second.
  - **LL grammars** are grammars that can be parsed by an LL parser.
Top-Down

1. \( id_list \)

2. \( id_list \)
   - \( id (A) \)
   - \( id_list_tail \)

3. \( id_list \)
   - \( id (A) \)
   - \( id_list_tail \)
   - \( , \)
   - \( id (B) \)
   - \( id_list_tail \)

Production Rules:
- \( id_list \to id id_list_tail \)
- \( id_list_tail \to , id id_list_tail \)
- \( id_list_tail \to ; \)
- \( A, B, C; \)
1. id_list

2. id (A)  id_list_tail

3. id (A)  id_list_tail
   ,  id (B)  id_list_tail

LL parsers are sometimes called predictive because they “predict” the next state.
Top-Down

1. \textit{id\_list}

2. \textit{id\_list}
   \begin{itemize}
   \item \textit{id (A)}
   \item \textit{id\_list\_tail}
   \end{itemize}

3. \textit{id\_list}
   \begin{itemize}
   \item \textit{id (A)}
   \item \textit{id\_list\_tail}
   \item \textit{, id (B)}
   \item \textit{id\_list\_tail}
   \end{itemize}

For example, after \textit{id(A)} is “discovered”, the next state is “predicted as \textit{id\_list\_tail}.”
Top-Down

1. **id**

2. **id_list**
   - id (A)
   - id_list_tail

3. **id_list**
   - id (A)
   - id_list_tail
     - ,
     - id (B)
     - id_list_tail

Notice that tokens are placed in the tree from the left-most to right-most.

Top-Down

```
 id_list → id id_list_tail
 id_list_tail → , id id_list_tail
 id_list_tail → ;
 A, B, C;
```
Bottom-up

- **LR parsers** are bottom-up, i.e., they discover the terminals first and non-terminals second.
  - **LR grammars** are grammars that can be parsed by an LR parser.
  - All **LL grammars are LR grammars** but not vice versa.
Bottom-up

1. id (A)

2. id (A),

6. id (A), id (B), id (C);

7. id (A), id (B), id (C) id_list_tail;
Bottom-up

8.  id (A), id (B), id (C), id_list_tail → id_list_tail, id id_list_tail
A, B, C;

id_list → id id_list_tail
id_list_tail → ;
LR parsers are sometimes called **shift** because they “shift” the states.
Notice that tokens are added to the tree from the right-most to the left-most.
Bottom-up

Sometimes you see LL and LR parsers written as LL(n) and LR(n) to denote the parser needs to look ahead n tokens.
The problem with this grammar is that it can require an \textit{arbitrarily large} number of terminals to be “shifted” before placing them into the tree.
A better bottom-up grammar

This grammar limits the number of “suspended” non-terminals.

\[
\begin{align*}
  \text{id\_list} & \rightarrow \text{id\_list\_prefix} ; \\
  \text{id\_list\_prefix} & \rightarrow \text{id\_list\_prefix}, \text{id} \\
  \text{id\_list\_prefix} & \rightarrow \text{id}
\end{align*}
\]
A better bottom-up grammar

Let's try parsing “A, B, C;”

\[
\text{id\_list} \rightarrow \text{id\_list\_prefix} ; \\
\text{id\_list\_prefix} \rightarrow \text{id\_list\_prefix}, \text{id} \\
\text{id\_list\_prefix} \rightarrow \text{id}
\]
A better bottom-up grammar

However, it **cannot** be parsed by LL (top-down) parser. Since when the parser discovers an “id” it does not “know” the number of “id_list_prefixs”

```
id_list → id_list_prefix ;

id_list_prefix → id_list_prefix, id

id_list_prefix → id
```
A better bottom-up grammar

Both of these are valid break downs, but we don’t know which one. Therefore, is **NOT** a valid LL grammar, but it is a valid LR grammar.
This grammar (for the calculator) unlike the previous calculator grammar is an LL grammar, because when an “id” is encountered we know exactly where it belongs.
Let's try “c := 2*A+B”
Let’s try “2*A+B”

```latex
\text{expr} \rightarrow \text{term} | \text{expr add_op term}

\text{term} \rightarrow \text{factor} | \text{term mult_op factor}

\text{factor} \rightarrow \text{id} | \text{number} | - \text{factor} | (\text{expr})

\text{add_op} \rightarrow + | -

\text{mult_op} \rightarrow * | /
```
Let’s try “$c := 2A + B$”

$$expr \rightarrow id | number | - expr | (expr) | expr \ op \ expr$$

$$op \rightarrow + | - | * | /$$
Recursive Descent & LL Parse Table

• There are two ways to code a parser for LL grammars:
  • **Recursive Descent**, which is a recursive program with case statements that correspond to each one-to-one to nonterminals.
  • **LL Parse Table**, which consists of an iterative driver program and a table that contains all of the nonterminals.
program → stmt_list $$

stmt_list → stmt stmt_list | ε

stmt → id:= expr | read id | write expr

expr → term term_tail

term_tail → add_op term term_tail | ε

term → factor factor_tail

factor_tail → mult_op factor factor_tail | ε

factor → (expr) | id | literal

add_op → + | -

mult_op → * | /
Recursive Descent

```
procedure program()
  case in_tok of
    id, read, write, $$:
      stmt_list()
      match($$
    else
      return error
  
procedure stmt()
  case in_tok of
    id: match(id); match(:=); expr()
    read: match(read); match(id)
    write: match(write); expr()
    else
      return error

procedure stmt_list()
  case in_tok of
    id, read, write:
      stmt(); stmt_list();
    $$:
      skip
    else
      return error

procedure match(expec)
  if in_tok = expec
    consume in_tok
  else
    return error
```
Recursive Descent

procedure program()
    case in_tok of
        id, read, write:
            stmt_list()
            match($$)
        $$:
            skip
        else
            return error
    return error

procedure stmt_list()
    case in_tok of
        id, read, write:
            stmt(); stmt_list();
        $$:
            skip
        else
            return error

procedure match(expec)
    if in_tok = expec
        consume in_tok
        return error
    else
        return error

• **in_tok** is a global variable that is the current token
• **consume** changes in_tok to the next token.
Recursive Descent

procedure program()
  case in_tok of
    id, read, write, $$:
      stmt_list()
      match($$)
    else
      return error
  end

procedure stmt_list()
  case in_tok of
    id, read, write:
      stmt(); stmt_list();
    $$:
      skip
    else
      return error
  end

procedure stmt()
  case in_tok of
    id:
      match(id); match(=); expr()
    read:
      match(read); match(id)
    write:
      match(write); expr()
    else
      return error
  end

procedure match(expec)
  if in_tok = expec
    consume in_tok
  else
    return error
The question is how do we label the case statements?
First, Follow, and Predict

• Three functions allow us to label the branches
  • \textbf{FIRST}(a): The terminals (and \( \varepsilon \)) that can be the first tokens of the non-terminal symbol \( a \).
  • \textbf{FOLLOW}(A): The terminals that can follow the terminal or non-terminal symbol \( A \)
  • \textbf{PREDICT}(A \rightarrow a): The terminals that can be the first tokens as a result of the production \( A \rightarrow a \)
- FIRST\( (program) \) = \{id, read, write, \$$\} \\
- FOLLOW\( (program) \) = \{\varepsilon\} \\
- PREDICT\( (program \rightarrow stmt\_list \$$) = \{id, read, write, \$$\} \\
- FOLLOW\( (id) \) = \{+, -, *, /, ), :=, id, read, write, \$$\}
\[
\begin{align*}
\text{program} & \rightarrow \text{stmt\_list} \mathbin{\$}\$ \\
\text{stmt\_list} & \rightarrow \text{stmt} \text{stmt\_list} \mid \epsilon \\
\text{stmt} & \rightarrow \text{id:= expr} \mid \text{read id} \mid \text{write expr} \\
\text{expr} & \rightarrow \text{term term\_tail} \\
\text{term\_tail} & \rightarrow \text{add\_op term term\_tail} \mid \epsilon \\
\text{term} & \rightarrow \text{factor factor\_tail} \\
\text{factor\_tail} & \rightarrow \text{mult\_op factor factor\_tail} \mid \epsilon \\
\text{factor} & \rightarrow (\text{expr}) \mid \text{id} \mid \text{literal} \\
\text{add\_op} & \rightarrow + \mid - \\
\text{mult\_op} & \rightarrow \ast \mid / \\
\text{FIRST}(\text{factor\_tail}) & = \{ \ast, \; /, \; \epsilon \} \\
\text{FOLLOW}(\text{factor\_tail}) & = \{ +, -, ), \text{id}, \text{read}, \text{write}, \mathbin{\$}\$ \} \\
\text{PREDICT}(\text{factor\_tail} \rightarrow m\_op \text{ factor factor\_tail}) & = \{ \ast, \; / \} \\
\text{PREDICT}(\text{factor\_tail} \rightarrow \epsilon) & = \{ +, -, ), \text{id}, \text{read}, \text{write}, \mathbin{\$}\$ \}
\end{align*}
\]
Since \textit{factor\_tail} can be “transformed” into an empty statement \textbf{PREDICT(factor\_tail $\rightarrow \varepsilon$)} equals \textbf{FOLLOW(factor\_tail)}.
These are all of the PREDICT() values from every production.

```
procedure program()
    case in_tok of
        id, read, write, $$:
            stmt_list()
            match($$)
        else
            return error
    end

procedure stmt_list()
    case in_tok of
        id, read, write:
            stmt(); stmt_list();
        $$:
            skip
        else
            return error
    end

procedure stmt()
    case in_tok of
        id:
            match(id); match(=); expr()
        read:
            match(read); match(id)
        write:
            match(write); expr()
        else
            return error
    end

procedure match(expec)
    if in_tok = expec
        consume in_tok
    else
        return error
```
Constructing FIRST, FOLLOW, and PREDICT

• To construct the FIRST, FOLLOW, and PREDICT tables we iterate through the grammar building on knowledge.
  • First, we define all of the “obvious” FIRST and FOLLOW values
    • For example, $\in\text{FOLLOW (stmt\_list)}$ and $\{id,\, \text{read, write}\} \in \text{FIRST(stmt)}$
  • Next, we build on this,
    • For example, $\{id, \text{read, write}\} \in \text{FIRST(stmt\_list)}$ since $\text{stmt\_list}$ can begin with $\text{stmt}$ and $\{id, \text{read, write}\} \in \text{FIRST(stmt)}$
  • We then continue on until we get no more knowledge.
Let’s try making tables for this grammar

\[
\begin{align*}
expr & \rightarrow \ term \mid expr \ add\_op \ term \\
term & \rightarrow \ factor \mid term \ mult\_op \ factor \\
factor & \rightarrow \ id \mid \text{number} \mid - factor \mid ( expr ) \\
add\_op & \rightarrow \ + \mid - \\
mult\_op & \rightarrow \ * \mid / 
\end{align*}
\]
Table-Driven

p_stack: stack of symbols;
p_stack.push(st_symbol);

loop
  exp_sym := p_stack.pop
  if exp_sym = terminal
    match(exp_sym);
    if exp_sym=$$ return
  else
    if table[exp_sym,in_tok].action = error
      return error
    else
      prediction := table[exp_sym,in_tok].prod;
      foreach sym in prod_table[prediction]
        p_stack.push(sym)
<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Input Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>program</code></td>
<td>read A read B</td>
</tr>
<tr>
<td><code>stmt_list</code></td>
<td>read A read B</td>
</tr>
<tr>
<td><code>stmt stmt_list</code></td>
<td>read A read B</td>
</tr>
<tr>
<td><code>read id stmt_list</code></td>
<td>read A read B</td>
</tr>
<tr>
<td><code>id stmt_list</code></td>
<td>A read B</td>
</tr>
</tbody>
</table>
Writing an LL(1) Grammar--Left Recursion

• **Left recursion** is where the leftmost symbol is a recursive non-terminal symbol.

  \[
  \text{id\_list} \rightarrow \text{id\_list\_prefix};
  \]
  \[
  \text{id\_list\_prefix} \rightarrow \text{id\_list\_prefix}, \text{id}
  \]
  \[
  \text{id\_list\_prefix} \rightarrow \text{id}
  \]

• This can cause a grammar **not to be LL(1)**.

• It is **desirable for LR grammars**.
Writing an LL(1) Grammar-- Eliminating Left Recursion

• To eliminate left recursion replace it with **right recursion**.

```plaintext
id_list → id_list_prefix;

id_list_prefix → id_list_prefix, id

id_list_prefix → id

id_list → id id_list_tail

id_list_tail → , id id_list_tail

id_list_tail → ;
```
Writing an LL(1) Grammar--Common Prefix

- **Common prefixes** occur when there is more than one prefix for a given nonterminal.

\[
\text{stmt} \rightarrow \text{id} := \text{expr} \\
\text{stmt} \rightarrow \text{id} (\text{arguments})
\]

- Again, this causes a grammar **not to be LL(1)**.
Writing an LL(1) Grammar--Left factoring

- **Common prefixes** To get rid of common prefixes we use a technique called **left factoring**.

```
stmt → id := expr
stmt → id (arguments)
stmt → id stmt_list_tail
stmt_list_tail → := expr | (arguments)
```
Dangling else

• Even if left recursion and common prefixes don’t exist a language may not be LL(1).

• In Pascal, there is the problem that an else statement in if-then-else statements is optional. Because we don’t know which if to match else to.

```plaintext
if AAA then
  if BBB then
    CCC
else
  DDD
```
Dangling else

- In Pascal there is **NO LL(1)** parser that can handle this problem.
- Even though a proper LR(1) parser can handle this, it may not handle it in a **method the programmer desires**.

```pascal
if AAA then
  if BBB then
    CCC
  else
    DDD
```
Dangling else

• Thus, to write this code correctly (based on indentation) “begin” and “end” statements must be added.

```plaintext
if AAA then
  if BBB then
    CCC
else
  DDD
if AAA then
begin
  if BBB then
    CCC
end
else
  DDD
```