Compiler Design and Construction
Syntax Analysis

Slides modified from Louden Book and Dr. Scherger
The Role of the Parser

- The following figure shows the position of the parser in a compiler:
- Basically it asks the lexical analyzer for a token whenever it needs one and builds a parse tree which is fed to the rest of the front end.
- In practice, the activities of the rest of the front end are usually included in the parser so it produces intermediate code instead of a parse tree.
The Role of the Parser

- There are universal parsing methods that will parse any grammar but they are too inefficient to use in compilers.

- Almost all programming languages have such simple grammars that an efficient top-down or bottom-up parser can parse a source program with a single left-to-right scan of the input.

- Another role of the parser is to detect syntax errors in the source, report each error accurately and recover from it so other syntax errors can be found.
For some examples of common syntax errors consider the following Pascal program:

(1) program prmax(input, output);
(2) var
(3)   x, y : integer;
(4) function max(i:integer; j:integer) : integer;
(5)   { return maximum of integers i and j }
(6) begin
(7)   if i > j then max := i
(8)   else max := j
(9) end;
(10) begin
(11)   readln (x,y);
(12)   writeln (max(x,y))
(13) end.
Errors in punctuation are common.

program prmax(input, output);
var
  x, y : integer;

function max(i:integer; j:integer) : integer;
{ return maximum of integers i and j }
begin
  if i > j then max := i
  else max := j
end;

begin
  readln (x,y);
  writeln (max(x,y))
end.
Errors in punctuation are common. For example:

- using a comma instead of a semicolon in the argument list of a function declaration (line 4);
- leaving out a mandatory semicolon at the end of a line (line 4);
- or using an extraneous semicolon before an else (line 7).

```
(1) program prmax(input, output);
(2) var
(3)   x, y : integer;
(4)   function max(i:integer, j:integer) : integer;
(5)   { return maximum of integers i and j }
(6)   begin
(7)     if i > j then max := i ;
(8)     else max := j
(9)   end;
(10) begin
(11)   readln (x,y);
(12)   writeln (max(x,y))
(13) end.
```
Syntax Error Handling

- **Operator errors often occur:**
  - For example, using `=` instead of `:=` (line 7 or 8).

```plaintext
(1) program prmax(input, output);
(2) var
(3)   x, y : integer;
(4) function max(i:integer; j:integer) : integer;
(5)   { return maximum of integers i and j }
(6)   begin
(7)     if i > j then max := i
(8)     else max := j
(9)   end;
(10) begin
(11)   readln (x,y);
(12)   writeln (max(x,y))
(13) end.
```
Keywords may be misspelled: `writeln` instead of `writeln` (line 12).

```plaintext
program prmax(input, output);
var
  x, y : integer;

function max(i:integer; j:integer) : integer;
{ return maximum of integers i and j }
begin
  if i > j then max := i
  else max := j
end;

begin
  readln (x,y);
  writeln (max(x,y))
end.
```
A **begin** or **end** may be missing (line 9). Usually difficult to repair.

```plaintext
program prmax(input, output);
var
  x, y : integer;

function max(i:integer; j:integer) : integer;
{ return maximum of integers i and j }
begin
  if i > j then max := i
  else max := j
end;

begin
  readln (x,y);
  writeln (max(x,y))
end.
```
Error Reporting

- A common technique is to print the offending line with a pointer to the position of the error.

- The parser might add a diagnostic message like
  
  "semicolon missing at this position"

  if it knows what the likely error is.
Error Recovery

- The parser should try to recover from an error quickly so subsequent errors can be reported.
  - If the parser doesn't recover correctly it may report spurious errors.

- Panic-mode recovery:
  - Discard input tokens until a synchronizing token (like ; or end) is found.
  - Simple but may skip a considerable amount of input before checking for errors again.
  - Will not generate an infinite loop.

- Phrase-level recovery:
  - Replace the prefix of the remaining input with some string to allow the parser to continue.
  - Examples:
    - Replace a comma with a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
    - Must be careful not to get into an infinite loop.
Error Recovery Strategies

- Recovery with error productions:
  - Augment the grammar with productions to handle common errors.
  - Example:

```plaintext
parameter_list --> identifier_list : type
| parameter_list ; identifier_list : type
| parameter_list , {error; writeln("comma should be a semicolon")} identifier_list : type
```
Error Recovery Strategies

- Recovery with global corrections:
  - Find the minimum number of changes to correct the erroneous input stream.
  - Too costly in time and space to implement.
  - Currently only of theoretical interest.
Context Free Grammars (Again!)

- Context-free grammars are defined previously:
  - They are a convenient way of describing the syntax of programming languages.

- A string of terminals (tokens) is a sentence in the source language of a compiler if and only if it can be parsed using the grammar defining the syntax of that language.

- A string of vocabulary symbols (terminal and nonterminal) that can be derived from $S$ (in zero 0 or more steps) is a sentential form
Derivations

- One of the simple compilers presented describes parsing as the construction of a parse tree whose root is the start symbol and whose leaves are the tokens in the input stream.

- Parsing can also be described as a re-writing process:
  - Each production in the grammar is a **re-writing rule** that says that an appearance of the nonterminal on the left-side can be replaced by the string of symbols on the right-side.
  - An input string of tokens is a sentence in the source language if and only if it can be derived from the start symbol by applying some sequence of re-writing rules.
Derivations: Top Down Parsing

• To introduce top-down parsing we consider the following context-free grammar:

\[
\begin{align*}
\text{expr} & \rightarrow \text{term } \text{rest} \\
\text{rest} & \rightarrow + \text{term } \text{rest} \mid - \text{term } \text{rest} \mid \varepsilon \\
\text{term} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

• and show the construction of the parse tree for the input string: \(9 - 5 + 2\).
Derivations: Top Down Parsing

- **Initialization**: The root of the parse tree must be the starting symbol of the grammar, $expr$. 

expr
Derivations: Top Down Parsing

- **Step 1**: The only production for `expr` is `expr --> term rest` so the root node must have a `term` node and a `rest` node as children.
Derivations: Top Down Parsing

- **Step 2:** The first token in the input is 9 and the only production in the grammar containing a 9 is:
  - `term --> 9` so 9 must be a leaf with the `term` node as a parent.

![Syntax Tree Diagram]

- `expr`
  - `term`
    - 9
  - `rest`
Derivations: Top Down Parsing

- **Step 3:** The next token in the input is the minus-sign and the only production in the grammar containing a minus-sign is:
  - `rest --> - term rest`. The `rest` node must have a minus-sign leaf, a `term` node and a `rest` node as children.

```
expr
  term
    9
  rest
    term
    rest
```
Derivations: Top Down Parsing

- **Step 4:** The next token in the input is 5 and the only production in the grammar containing a 5 is:
  - $term \rightarrow 5$ so 5 must be a leaf with a $term$ node as a parent.
Derivations: Top Down Parsing

- **Step 5:** The next token in the input is the plus-sign and the only production in the grammar containing a plus-sign is:
  - `$rest \rightarrow + \ term \ rest$`.
  - A `$rest$` node must have a plus-sign leaf, a `$term$` node and a `$rest$` node as children.
Derivations: Top Down Parsing

- **Step 6:** The next token in the input is 2 and the only production in the grammar containing a 2 is: \( \text{term} \rightarrow 2 \) so 2 must be a leaf with a term node as a parent.

```
expr
  /\  \
term  rest
   / \    \
  9   -   term  rest
     /\    /\
    5  5  term  rest
         /\  \
        2   \
```
Derivations: Top Down Parsing

- **Step 7:** The whole input has been absorbed but the parse tree still has a *rest* node with no children.
- The *rest* $\rightarrow \varepsilon$ production must now be used to give the *rest* node the empty string as a child.
Derivations: What We Just Did...

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations
- Two derivations are of interest
  - *Leftmost derivation* — replace leftmost NT at each step
  - *Rightmost derivation* — replace rightmost NT at each step

- These are the two systematic derivations
  - *(We don’t care about randomly-ordered derivations!)*

- An example shown soon will show the two types of derivations.
  - Interestingly, they turn out to be different
The example constructed the parse tree in seven steps where each step used a production in the grammar.

Below is a table to show what occurs when the production of each step is used as a re-writing rule on a symbol string:

- The initial symbol string just contains the start symbol, `expr`.

<table>
<thead>
<tr>
<th>Steps</th>
<th>Productions (Re-writing Rules)</th>
<th>Symbol Strings (Sentinel Forms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>expr -&gt; term rest</code></td>
<td><code>term rest</code></td>
</tr>
<tr>
<td>2</td>
<td><code>term -&gt; 9</code></td>
<td><code>9 rest</code></td>
</tr>
<tr>
<td>3</td>
<td><code>rest -&gt; - term rest</code></td>
<td><code>9 - term rest</code></td>
</tr>
<tr>
<td>4</td>
<td><code>term -&gt; 5</code></td>
<td><code>9 - 5 rest</code></td>
</tr>
<tr>
<td>5</td>
<td><code>rest -&gt; + term rest</code></td>
<td><code>9 - 5 + term rest</code></td>
</tr>
<tr>
<td>6</td>
<td><code>term -&gt; 2</code></td>
<td><code>9 - 5 + 2 rest</code></td>
</tr>
<tr>
<td>7</td>
<td><code>rest -&gt; ε</code></td>
<td><code>9 - 5 + 2</code></td>
</tr>
</tbody>
</table>
Derivations

- Only the last symbol string in the derivation is a *sentence* in the language:
  - Earlier symbol strings are not sentences because they contain nonterminals as well as terminals so they are merely *sentential forms*.
A derivation is usually shown as a sequence of the sentential forms separated by double-line arrows, $\Rightarrow$.

The first sentential form in the sequence is the start symbol of the grammar and the last sentential form is a sentence in the language.

For example, the foregoing derivation for $9 - 5 + 2$ is usually written:

\[
\begin{align*}
\text{expr} & \Rightarrow \\
\text{term rest} & \Rightarrow \\
9 & \Rightarrow \\
9 - \text{term rest} & \Rightarrow \\
9 - 5 & \Rightarrow \\
9 - 5 + \text{term rest} & \Rightarrow \\
9 - 5 + 2 & \Rightarrow \\
9 - 5 + 2 & \\
\text{Or} \\
\text{expr} & \Rightarrow \text{term rest} \Rightarrow 9 \Rightarrow 9 - \\
\text{term rest} & \Rightarrow 9 - 5 \Rightarrow 9 - 5 + \\
\text{term rest} & \Rightarrow 9 - 5 + 2 \Rightarrow 9 - 5 + 2
\end{align*}
\]
Derivations

- The double-line arrow, \( \Rightarrow \), is read as "derives in one step".

- The symbol, \( \Rightarrow^* \), is read as "derives in zero or more steps".

- Thus, \( expr \Rightarrow^* 9-5+2 \) because:

\[
expr \Rightarrow term \ rest \Rightarrow 9 \ rest \Rightarrow 9- \ term \ rest \Rightarrow 9-5 \ rest \Rightarrow 9-5+ \ term \ rest \Rightarrow 9-5+2 \ rest \Rightarrow 9-5+2
\]
Derivations

• Each step of a derivation replaces a single nonterminal in the sentential form with a string of symbols on the right side of some production for that nonterminal.

• When there are two or more nonterminals in the sentential form which nonterminal gets replaced?

• It doesn't matter, the parse tree can be built several different ways.

• Having chosen the nonterminal to be replaced which of its re-writing rules should be applied?

• This does matter, in an ambiguous grammar choosing the wrong re-writing rule (production) will construct a different parse tree for the same token stream.

• If the grammar is unambiguous then the correct re-writing rule must be selected at each step or the parse tree can't be built.
The foregoing derivation of \( 9-5+2 \) is called a \textit{leftmost} derivation because each step replaces the leftmost nonterminal in the sentential form.

Each step of a \textit{rightmost} derivation step replaces the rightmost nonterminal in the sentential form:

<table>
<thead>
<tr>
<th>Steps</th>
<th>Productions (Re-writing Rules)</th>
<th>Symbol Strings (Sentinel Forms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>expr -&gt; term rest</td>
<td>term rest</td>
</tr>
<tr>
<td>1</td>
<td>rest -&gt; - term rest</td>
<td>term - term rest</td>
</tr>
<tr>
<td>2</td>
<td>rest -&gt; + term rest</td>
<td>term - term + term rest</td>
</tr>
<tr>
<td>3</td>
<td>rest -&gt; e</td>
<td>term - term + term</td>
</tr>
<tr>
<td>4</td>
<td>term -&gt; 2</td>
<td>term - term + 2</td>
</tr>
<tr>
<td>5</td>
<td>term -&gt; 5</td>
<td>term - 5 + 2</td>
</tr>
<tr>
<td>6</td>
<td>term -&gt; 9</td>
<td>9 - 5 + 2</td>
</tr>
</tbody>
</table>
Derivations

- Note that both derivations of \(9-5+2\) used the same seven re-writing rules but in a different order.

- Why does the parsing process described previously construct the parse tree using the leftmost derivation?

- Both derivations build the parse tree top-down but the leftmost derivation builds the left-side of the tree first and the rightmost derivation builds the right-side first.

- The parsing process chooses the leftmost derivation because it reads the input token string from left-to-right.
Derivations

- A bottom-up parser performs a derivation in reverse order:
  - Starting with the sentence and ending with the start symbol of the grammar.

- Each step in a bottom-up parser performs a production of the grammar in reverse:
  - *Reducing* the sentential form by finding a string of symbols in the form that correspond to the right-side of some production and replacing that string with the nonterminal of that production.
Derivations

- What kind of derivation should a bottom-up parser perform in reverse order?

- Note that the last step of a leftmost derivation builds the rightmost corner of the parse tree while the last step of a rightmost derivation builds the leftmost corner.

- A bottom-up parser reads the input tokens from left-to-right so it performs a rightmost derivation in reverse order.
Derivations

• Parsers are classified by the order they read the input tokens and by the kind of derivations they perform.

• A top-down parser that reads the input tokens from left-to-right and performs a leftmost derivation is an LL-parser.

• A bottom-up parser that reads the input tokens from left-to-right and performs a rightmost derivation is an LR-parser.
Another Example: The Two Derivations for $x - 2 \times y$

In both cases, $Expr \Rightarrow^* id - num \times id$

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>$Expr$</td>
</tr>
<tr>
<td>1</td>
<td>$Expr \ Op \ Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$&lt;id, x&gt; \ Op \ Expr$</td>
</tr>
<tr>
<td>5</td>
<td>$&lt;id, x&gt; - Expr$</td>
</tr>
<tr>
<td>1</td>
<td>$&lt;id, x&gt; - Expr \ Op \ Expr$</td>
</tr>
<tr>
<td>2</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; \ Op \ Expr$</td>
</tr>
<tr>
<td>6</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; \times \ Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; \times &lt;id, y&gt;$</td>
</tr>
</tbody>
</table>

*Leftmost derivation*

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>$Expr$</td>
</tr>
<tr>
<td>1</td>
<td>$Expr \ Op \ Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$Expr \ Op \ &lt;id, y&gt;$</td>
</tr>
<tr>
<td>5</td>
<td>$Expr \times &lt;id, y&gt;$</td>
</tr>
<tr>
<td>1</td>
<td>$Expr \ Op \ Expr \times &lt;id, y&gt;$</td>
</tr>
<tr>
<td>2</td>
<td>$Expr \ Op &lt;num, 2&gt; \times &lt;id, y&gt;$</td>
</tr>
<tr>
<td>6</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; \times &lt;id, y&gt;$</td>
</tr>
</tbody>
</table>

*Rightmost derivation*
Derivations and Parse Trees

Leftmost derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Expr Op Expr</td>
</tr>
<tr>
<td>3</td>
<td>&lt;id,x&gt; Op Expr</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id,x&gt; - Expr</td>
</tr>
<tr>
<td>1</td>
<td>&lt;id,x&gt; - Expr Op Expr</td>
</tr>
<tr>
<td>2</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; Op Expr</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * Expr</td>
</tr>
<tr>
<td>3</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
</tbody>
</table>

This evaluates as \( x - (2 * y) \)
Derivations and Parse Trees

Rightmost derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>$Expr$</td>
</tr>
<tr>
<td>1</td>
<td>$Expr\ Op\ Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$Expr\ Op\ &lt;id,y&gt;$</td>
</tr>
<tr>
<td>6</td>
<td>$Expr\ *\ &lt;id,y&gt;$</td>
</tr>
<tr>
<td>1</td>
<td>$Expr\ Op\ Expr\ *\ &lt;id,y&gt;$</td>
</tr>
<tr>
<td>2</td>
<td>$Expr\ Op\ &lt;num,2&gt;\ *\ &lt;id,y&gt;$</td>
</tr>
<tr>
<td>5</td>
<td>$Expr\ -\ &lt;num,2&gt;\ *\ &lt;id,y&gt;$</td>
</tr>
<tr>
<td>3</td>
<td>$&lt;id,x&gt;\ -\ &lt;num,2&gt;\ *\ &lt;id,y&gt;$</td>
</tr>
</tbody>
</table>

This evaluates as $(x - 2) * y$
Derivations and Precedence

These two derivations point out a problem with the grammar: It has no notion of precedence, or implied order of evaluation.

To add precedence:
- Create a non-terminal for each level of precedence
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions:
- Multiplication and division, first
- Subtraction and addition, next
Derivations and Precedence

Adding the standard algebraic precedence produces:

<table>
<thead>
<tr>
<th>Level</th>
<th>Grammar Non-Term</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr</td>
<td>Expr + Term</td>
</tr>
<tr>
<td>3</td>
<td>Expr - Term</td>
<td>Term</td>
</tr>
<tr>
<td>4</td>
<td>Term</td>
<td>Term * Factor</td>
</tr>
<tr>
<td>5</td>
<td>Term</td>
<td>Term / Factor</td>
</tr>
<tr>
<td>6</td>
<td>Factor</td>
<td>number</td>
</tr>
<tr>
<td>7</td>
<td>Factor</td>
<td>id</td>
</tr>
<tr>
<td>8</td>
<td>Factor</td>
<td>number</td>
</tr>
<tr>
<td>9</td>
<td>Factor</td>
<td>id</td>
</tr>
</tbody>
</table>

This grammar is slightly larger
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

Let's see how it parses \( x - 2 * y \)
Derivations and Precedence

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
</tr>
<tr>
<td>3</td>
<td>Expr → Term</td>
</tr>
<tr>
<td>5</td>
<td>Expr → Term * Factor</td>
</tr>
<tr>
<td>9</td>
<td>Expr → Term * &lt;id,y&gt;</td>
</tr>
<tr>
<td>7</td>
<td>Expr → Factor* &lt;id,y&gt;</td>
</tr>
<tr>
<td>8</td>
<td>Expr → &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>4</td>
<td>Term → &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>7</td>
<td>Factor → &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; → &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
</tbody>
</table>

The rightmost derivation

This produces $x - (2 * y)$, along with an appropriate parse tree. Both the leftmost and rightmost derivations give the same expression, because the grammar directly encodes the desired precedence.

Its parse tree
Ambiguous Grammars

Our original expression grammar had other problems

- This grammar allows multiple leftmost derivations for $x - 2 * y$
- Hard to automate derivation if $> 1$ choice
- The grammar is *ambiguous*
Two Leftmost Derivations for $x - 2 * y$

The Difference:
- Different productions chosen on the second step

<table>
<thead>
<tr>
<th>Rule</th>
<th>Original choice</th>
<th>New choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$Expr$</td>
<td>$Expr$</td>
</tr>
<tr>
<td>1</td>
<td>$Expr , Op , Expr$</td>
<td>$Expr , Op , Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$\langle id, x \rangle , Op , Expr$</td>
<td>$\langle id, x \rangle , Op , Expr$</td>
</tr>
<tr>
<td>5</td>
<td>$\langle id, x \rangle , - , Expr$</td>
<td>$\langle id, x \rangle , - , Expr$</td>
</tr>
<tr>
<td>1</td>
<td>$\langle id, x \rangle , - , Expr , Op , Expr$</td>
<td>$\langle id, x \rangle , Op , Expr , Op , Expr$</td>
</tr>
<tr>
<td>2</td>
<td>$\langle id, x \rangle , - , \langle num, 2 \rangle , Op , Expr$</td>
<td>$\langle id, x \rangle , - , \langle num, 2 \rangle , Op , Expr$</td>
</tr>
<tr>
<td>6</td>
<td>$\langle id, x \rangle , - , \langle num, 2 \rangle , * , Expr$</td>
<td>$\langle id, x \rangle , - , \langle num, 2 \rangle , * , Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$\langle id, x \rangle , - , \langle num, 2 \rangle , * , \langle id, y \rangle$</td>
<td>$\langle id, x \rangle , - , \langle num, 2 \rangle , * , \langle id, y \rangle$</td>
</tr>
</tbody>
</table>

Both derivations succeed in producing $x - 2 * y$
Writing a Grammar

- Context-free grammars can describe a larger class of languages than regular expressions.

- Most of the syntax of a programming language can be described with a context-free grammar but there are still certain constraints that can't be so described
  - (such as not using a variable before it's declared.)

- Those constraints are checked by the semantic analyzer.
Every construct that can be described a regular expression can also be described by a grammar.

For example, the regular expression and the grammar shown at the right describe the same language:

- the set of strings of a's and b's ending in \texttt{abb}.

\[
(a \mid b) \ast a \ b \ b
\]

\[
\begin{align*}
A_0 & \rightarrow a \ A_0 \mid b \ A_0 \mid a \ A_1 \\
A_1 & \rightarrow b \ A_2 \\
A_2 & \rightarrow b \ A_3 \\
A_3 & \rightarrow \varepsilon
\end{align*}
\]
Then why do we describe a lexical analyzer in terms of regular expressions when we could've used a grammar instead?

Here are four possible reasons:

1. Lexical analysis doesn't need a notation as powerful as a grammar;
2. Regular expressions are easier to understand;
3. More efficient lexical analyzers can be implemented from regular expressions;
4. Separating lexical analysis from nonlexical analysis splits the front end of a compiler into two manageable-size parts.

\[(a \mid b)^* a b b\]

\[
\begin{align*}
A_0 & \rightarrow a A_0 \mid b A_0 \mid a A_1 \\
A_1 & \rightarrow b A_2 \\
A_2 & \rightarrow b A_3 \\
A_3 & \rightarrow \varepsilon
\end{align*}
\]
A grammar $G$ generates a language $L$ if and only if:

1. every string generated by $G$ is in $L$; and
2. every string in $L$ can indeed be generated by $G$.

Consider the grammar:

$S \rightarrow e \mid (S)S$

generates all strings of balanced parentheses.

Such a derivation must be of the form

$S \rightarrow (S)S \rightarrow^* (x)S \rightarrow^* (x)y$

Verifying the Language Generated by a Grammar
Ambiguity

• Most programming languages allow both `if-then` and `if-then-else` conditional statements.

• For example, the productions for a statement are:

```
stmt --> if expr then stmt
    | if expr then stmt else stmt
    | other
```

where `other` stands for all other statements.
Ambiguity

- Any such language has a "dangling-else" ambiguity:
  - if $E_1$ then if $E_2$ then $S_1$ else $S_2$

- where $E_1$ and $E_2$ are logical expressions and $S_1$ and $S_2$ are statements.

- If $E_1$ is false should $S_2$ be executed or not?
  - It depends on which parse tree is used.
  - languages with the "dangling-else" ambiguity resolve the problem by matching each `else` with the closest previous unmatched `then`. 
Ambiguity

Definitions

- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is *ambiguous*.
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is *ambiguous*.
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar.

Classic example — the *if-then-else* problem

\[
Stmt \rightarrow \text{if } Expr \text{ then } Stmt \\
| \text{if } Expr \text{ then } Stmt \text{ else } Stmt \\
| \text{... other stmts ...}
\]

*This ambiguity is entirely grammatical in nature*
This sentential form has two derivations

\[
\text{if } \text{Expr}_1 \text{ then if } \text{Expr}_2 \text{ then Stmt}_1 \text{ else Stmt}_2
\]
Ambiguity

- Removing the ambiguity
  - Must rewrite the grammar to avoid generating the problem
  - Match each `else` to innermost unmatched `if` *(common sense rule)*

1. `Stmt → WithElse`
2.   | `NoElse`
3. `WithElse → if Expr then WithElse else WithElse`
4.   | `OtherStmt`
5. `NoElse → if Expr then Stmt`
6.   | `if Expr then WithElse else NoElse`

Intuition: a `NoElse` always has no else on its last cascaded `else if` statement

- With this grammar, the example has only one derivation
# Ambiguity

\[ \text{if } \textit{Expr}_1 \text{ then if } \textit{Expr}_2 \text{ then } \textit{Stmt}_1 \text{ else } \textit{Stmt}_2 \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \textit{Stmt} )</td>
</tr>
<tr>
<td>2</td>
<td>( \textit{NoElse} )</td>
</tr>
<tr>
<td>5</td>
<td>( \text{if } \textit{Expr} \text{ then } \textit{Stmt} )</td>
</tr>
<tr>
<td>6</td>
<td>( \text{if } \textit{E}_1 \text{ then } \textit{Stmt} )</td>
</tr>
<tr>
<td>1</td>
<td>( \text{if } \textit{E}_1 \text{ then } \textit{WithElse} )</td>
</tr>
<tr>
<td>3</td>
<td>( \text{if } \textit{E}_1 \text{ then if } \textit{Expr} \text{ then } \textit{WithElse} \text{ else } \textit{WithElse} )</td>
</tr>
<tr>
<td>7</td>
<td>( \text{if } \textit{E}_1 \text{ then if } \textit{E}_2 \text{ then } \textit{WithElse} \text{ else } \textit{WithElse} )</td>
</tr>
<tr>
<td>4</td>
<td>( \text{if } \textit{E}_1 \text{ then if } \textit{E}_2 \text{ then } \textit{S}_1 \text{ else } \textit{WithElse} )</td>
</tr>
<tr>
<td>4</td>
<td>( \text{if } \textit{E}_1 \text{ then if } \textit{E}_2 \text{ then } \textit{S}_1 \text{ else } \textit{S}_2 )</td>
</tr>
</tbody>
</table>

This binds the \textit{else} controlling \( \textit{S}_2 \) to the inner if
Deeper Ambiguity

Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

\[ a = f(17) \]

In many Algol-like languages, \( f \) could be either a function or a subscripted variable

Disambiguating this one requires context

- Need values of declarations
- Really an issue of type, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
  - Step outside grammar rather than use a more complex grammar
Ambiguity - the Final Word

Ambiguity arises from two distinct sources

- Confusion in the context-free syntax
  - (if-then-else)
- Confusion that requires context to resolve
  - (overloading)

Resolving ambiguity

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity takes cooperation
  - Knowledge of declarations, types, …
  - Accept a superset of $L(G)$ & check it by other means†
  - This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar

- Parsing techniques that “do the right thing”
- i.e., always select the same derivation
Eliminating Left Recursion

- A grammar is **left recursive** if it contains a nonterminal \( A \) such that there is a chain of one or more derivations

\[
A \Rightarrow \ldots \Rightarrow A\ Z
\]

- where \( Z \) is a (possibly empty) string of symbols.

- Top-down parsing methods can’t handle left recursion so a method of eliminating it is needed.

- The following algorithm changes all left recursion into immediate left recursion and then eliminates it.
Eliminating Left Recursion

- Input: Grammar G with no cycles or ε-productions
- Output: An equivalent grammar with no left recursion
- Method: Apply the following algorithm to G. Note that the resulting non-left-recursive grammar may have ε-productions.

Arrange the non-terminals in some order \( A_1, A_2, \ldots, A_n \)

for \( i := 1 \) to \( n \) do begin
  for \( j := 1 \) to \( i-1 \) do begin
    replace each production of the form \( A_i \rightarrow A_j \gamma \) by the productions
    \( A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \ldots | \delta_k \gamma \)
    where \( A_j \rightarrow \delta_1 | \delta_2 | \ldots | \delta_k \) are all the current \( A_j \)-productions
  end
  eliminate the immediate left recursion among the \( A_i \)-productions
end
Immediate Left Recursion:

Immediate left recursion occurs where the grammar has a production for a nonterminal that begins with the same nonterminal:

Here is a general method for eliminating it:
Eliminating Left Recursion

Let $A$ be a nonterminal that has $m$ productions beginning with the same nonterminal, $A$, and $n$ other productions:

$$A \rightarrow A \alpha_1 | A \alpha_2 | \ldots | A \alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$$

where each $\alpha$ and $\beta$ is a string of grammar symbols and no $\beta$ begins with $A$.

To eliminate the immediate left recursion a new nonterminal, $A'$, is added to the grammar with the productions:

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A'$$

and the productions for nonterminal $A$ are changed to:

$$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$$
Eliminating Left Recursion: Example: id_list

- As an example consider the productions for *id_list* in the grammar for the coding projects:

  \[
  \text{id}_\text{list} \rightarrow \text{ID} \\
  \quad \mid \text{id}_\text{list} \text{COMMA ID}
  \]

  \[
  \text{id}_\text{list} \rightarrow \text{COMMA ID} \text{id}_\text{list}\_\text{rest}
  \mid \varepsilon
  \]

- In this example, there is only one \( \alpha \), COMMA ID, and only one \( \beta \), ID.

- To eliminate the immediate left recursion, a new nonterminal, *id_list_rest*, is added to the grammar, and the productions for *id_list* and *id_list_rest* are:

  \[
  \text{id}_\text{list} \rightarrow \text{ID} \text{id}_\text{list}\_\text{rest}
  \]

  \[
  \text{id}_\text{list}\_\text{rest} \rightarrow \text{COMMA ID} \text{id}_\text{list}\_\text{rest} \mid \varepsilon
  \]
Eliminating Left Recursion: Another Example: declarations

- As another example consider the productions for *declarations* in the grammar for the coding projects:

\[
\text{declarations} \rightarrow \text{declarations} \text{VARTOK declaration SEMICOL} \mid \varepsilon
\]

- There is only one \(\alpha\), VARTOK declaration SEMICOL, and the only \(\beta\), is the empty string, \(\varepsilon\).

- A new nonterminal, *declarations_rest*, is added to the grammar, and the productions for *declarations* and *declarations_rest* are:

\[
\text{declarations} \rightarrow \text{declarations}_\text{rest}
\]

\[
\text{declarations}_\text{rest} \rightarrow \text{VARTOK declaration SEMICOL declarations}_\text{rest} \mid \varepsilon
\]
This example illustrates what occurs when the only $\beta$ is the empty string.

*declarations* now has only one production:

\[
declarations \rightarrow declarations\_rest
\]

and this is the only production with *declarations\_rest* on the right-side.

We might as well change the name of *declarations\_rest* to *declarations* and change the grammar to read:

\[
declarations \rightarrow \text{VARTOK declaration SEMICOL declarations} | \varepsilon
\]
Left Factoring

- Left factoring is useful for producing a grammar suitable for a predictive parser. As an example consider the productions for `statement` in the `grammar for the coding projects`:

```
statement --> variable ASSIGNOP expr
    | procedure_call
    | block
    | IFTOK expr THENTOK statement ELSETOK statement
    | WHILETOK expr DOTOK statement
```
Left Factoring

- Three of the productions for statement begin with the nonterminals: variable, procedure_call, and block.
- The productions for these three nonterminals are:

  variable --> ID
       | ID LBRK expr RBRK

  procedure_call --> ID
       | ID LPAR expr_list RPAR

  block --> BEGINTOK opt_statements ENDTOK
In the productions for *statement* we replace nonterminals: *variable, procedure_call*, and *block*, by the right-sides of their productions to obtain:

```
statement --> ID ASSIGNOP expr
  | ID LBRK expr RBRK ASSIGNOP expr
  | ID
  | ID LPAR expr_list RPAR
  | BEGINTOK opt_statements ENDTOK
  | IFTOK expr THEN TOK statement ELSE TOK statement
  | WHI L E T OK expr DOTOK statement
```
Now every production for `statement` begins with a terminal but four of the productions begin with the same terminal, `ID`, so we add a new nonterminal, `statement_rest`, to the grammar and left factor `ID` out of those four productions to obtain:

```
statement --> ID statement_rest
  | BEGINTOK opt_statements ENDTOK
  | IFTOK expr THENTOK statement ELSETOK statement
  | WHILETOK expr DOTOK
```

```
statement statement_rest --> ASSIGNOP expr
  | LBRK expr RBRK ASSIGNOP expr
  | LPAR expr_list RPAR
  | ε
```
Left Factoring

• Note that the alternative productions for statement start with different terminals so a predictive parser will have no trouble selecting the correct production.

• The same is true for the alternative productions for statement_rest.

• In this example, nonterminals variable and procedure_call no longer appear on the right-side of any production in the project grammar so they can be deleted (along with their productions.)

• Nonterminal block still appears on the right-sides of productions for program and subroutine so it must be kept in the grammar.
Non-Context Free Language Constructs

- Programming languages insist that variables be declared before being used but there is no way of incorporating this constraint in a grammar.

- Another constraint that can't be enforced in a grammar is that the number and types of arguments in a function call agree with the number and types of the formal parameters in the definition of the function.

- Checks for these kinds of constraints are performed in the semantic analyzer.