Syntax Analysis (Parsing)
Syntax Analysis (Parsing)

\[ x = (2+3) \times 9; \quad \text{// mismatched parentheses} \]

\[ \text{if } x>y \ x = 2; \quad \text{// missing parentheses} \]

\[ \text{while } (x==3) \ do \ f1(); \quad \text{// invalid keyword do} \]
What is Syntax Analysis?

After lexical analysis (scanning), we have a series of tokens. In syntax analysis (or parsing), we want to interpret what those tokens mean.

Goal: Recover the structure described by that series of tokens.

Goal: Report errors if those tokens do not properly encode a structure.
What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.
What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.
- **Goal**: Recover the structure described by that series of tokens.
- **Goal**: Report errors if those tokens do not properly encode a structure.
Formal Languages

- An alphabet is a set $\sum$ of symbols that act as letters.
- A language over $\sum$ is a set of strings made from symbols in $\sum$. 
An **alphabet** is a set $\sum$ of symbols that act as letters.

A **language** over $\sum$ is a set of strings made from symbols in $\sum$.

When scanning, our alphabet was ASCII or Unicode characters. We produced tokens.
Formal Languages

- An **alphabet** is a set $\Sigma$ of symbols that act as letters.
- A **language** over $\Sigma$ is a set of strings made from symbols in $\Sigma$.
- When scanning, our alphabet was ASCII or Unicode characters. We produced tokens.
- When parsing, our alphabet is the set of tokens produced by the scanner.
When scanning, we used regular expressions to define each token.

Unfortunately, regular expressions are (usually) too weak to define programming languages.
Regular Expressions

- When scanning, we used regular expressions to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
- Cannot define a regular expression matching all expressions with properly balanced parentheses.
- Cannot define a regular expression matching all functions with properly nested block structure.
When scanning, we used regular expressions to define each token. Unfortunately, regular expressions are (usually) too weak to define programming languages. Cannot define a regular expression matching all expressions with properly balanced parentheses. Cannot define a regular expression matching all functions with properly nested block structure. We need a more powerful formalism.
A context-free grammar (or CFG) is a formalism for defining languages.

Can define the context-free languages, a strict superset of the regular languages.

Unlike regular grammars, the right hand-side of the production rules are unrestricted.
One possible CFG for describing all legal arithmetic expressions using addition, subtraction, multiplication, and division:

\[
E \rightarrow \mathtt{int} \mid E \; \text{Op} \; E \mid (E) \\
\text{Op} \rightarrow + \mid - \mid * \mid /
\]
One possible CFG for describing all legal arithmetic expressions using addition, subtraction, multiplication, and division

\[
\begin{align*}
E & \rightarrow \text{int} \\
E & \rightarrow E \; \text{Op} \; E \\
E & \rightarrow (E) \\
\text{Op} & \rightarrow + \\
\text{Op} & \rightarrow - \\
\text{Op} & \rightarrow * \\
\text{Op} & \rightarrow / \\
E & \Rightarrow E \; \text{Op} \; E \\
E & \Rightarrow E \; \text{Op} \; (E) \\
E & \Rightarrow E \; \text{Op} \; (E \; \text{Op} \; E) \\
E & \Rightarrow E \; \text{Op} \; (E \; \text{Op} \; E) \\
E & \Rightarrow \text{int} \; \text{Op} \; (E \; \text{Op} \; E) \\
E & \Rightarrow \text{int} \; \text{Op} \; (\text{int} \; \text{Op} \; E) \\
E & \Rightarrow \text{int} \; \text{Op} \; (\text{int} \; \text{Op} \; \text{int}) \\
E & \Rightarrow \text{int} \; \text{Op} \; (\text{int} \; + \; \text{int})
\end{align*}
\]
Formally, a context-free grammar (as is the regular grammar) is a collection of four objects:

- A set of nonterminal symbols (or variables),
- A set of terminal symbols,
- A set of production rules saying how each nonterminal can be converted by a string of terminals and nonterminals, and
- A start symbol that begins the derivation.
A context-free grammar is said to be *ambiguous* if there is more than one derivation for a particular string.
A context-free grammar is said to be ambiguous if there is more than one derivation for a particular string.

Consider:

1. $S \rightarrow ASB$
2. $S \rightarrow \epsilon$
3. $A \rightarrow a$
4. $B \rightarrow b$
Ambiguity

Consider:

1. $\text{Expr} \rightarrow \text{Expr} + \text{Expr}$
2. $\text{Expr} \rightarrow \text{Expr} \ast \text{Expr}$
3. $\text{Expr} \rightarrow ( \text{Expr} )$
4. $\text{Expr} \rightarrow \text{var}$
5. $\text{Expr} \rightarrow \text{const}$

There are two different derivation trees for the string $\text{var}+\text{var} \ast \text{var}$

![Derivation Trees]

Janyl Jumadinova

Compiler Development (CMPSC 401)
Ambiguity

We need unambiguous grammars for parsing
- The derivation determines the shape of the parse tree/abstract syntax tree, which in turn determines meaning.
Ambiguity

- We need unambiguous grammars for parsing
  - The derivation determines the shape of the parse tree/abstract syntax tree, which in turn determines meaning.
- If a grammar can be made unambiguous at all, it is usually made unambiguous through layering.
Ambiguity

- We need unambiguous grammars for parsing
  - The derivation determines the shape of the parse tree/abstract syntax tree, which in turn determines meaning.
- If a grammar can be made unambiguous at all, it is usually made unambiguous through **layering**.
  - Have exactly one way to build each piece of the string.
  - Have exactly one way of combining those pieces back together.
Resolving Ambiguity

- *With grammar:* If you can re-design the language, can avoid the problem entirely, e.g., create an `end` to match closest `if`
Resolving Ambiguity

- **With grammar:** If you can re-design the language, can avoid the problem entirely, e.g., create an `end` to match closest `if`.

- **With tools:** Most parser tools can cope with ambiguous grammars.
  - Typically one can specify operator precedence and associativity.
  - Allows simpler, ambiguous grammar with fewer nonterminals as basis for generated parser, without creating problems.
Precedence Declaration

- If we leave the world of pure CFGs, we can often resolve ambiguities through precedence declarations
  - e.g. multiplication has higher precedence than addition, but lower precedence than exponentiation.
Precedence Declaration

- If we leave the world of pure CFGs, we can often resolve ambiguities through precedence declarations - e.g. multiplication has higher precedence than addition, but lower precedence than exponentiation.
- Allows for unambiguous parsing of ambiguous grammars.