Announcements
A Few Announcements

- Course website
  

- There were no readings for today.

- Attendance (only for a few initial classes).

- Has everyone brought their laptops? We may not use it today, but it would be good to bring it with you.
  
  - In case someone does not have a laptop, please contact me. They can check one from the college for the semester.
Review
• This class will teach you
  – some common principles underlying most languages
  – some new ways of thinking about programs (new paradigms)

• A programming language must be
  – as easy as possible for people to
    • learn
    • read
    • write
  – as easy as possible for a compiler to translate into efficient machine code or an interpreter to execute
Formal Languages
Defining a Language

- To define a computer language we need to say
  - How do we tell if a file of characters is a legal (grammatical) program in this language? (syntax)
  - How do we define what a program in this language means? (semantics)
- Semantics: several approaches but in practice we just use English to explain the meaning
- Syntax: defined by a formal grammar
S → NP VP
NP → Name | Det Noun
VP → Verb | Verb NP
Name → john | mary
Det → a | the
Det → some | every
Noun → boy | girl
Verb → runs | likes
A grammar, $G$, is a quadruple $<T,N,P,S>$, where

- $T$ is a set of **terminal** symbols (e.g., john, mary, a, the, some, every, boy, girl, runs, likes);
- $N$ is a set of **nonterminal** symbols (e.g., S, NP, VP, Name, Det, Noun, Verb);
- $P$ is a set of **productions**, or rewrite rules (e.g., Det $\rightarrow$ a the);
- $S$ is a special **start** symbol.

The **language** of $G$, $L(G)$, is the set of all terminal sequences that can be produced by applying the rewrite rules, repeatedly, starting with $S$. 
• For Programming Languages:
  Stmt → Identifier := Digit
  Identifier → Letter | Identifier Letter | Identifier Digit
  Letter → a | b | c | ... | x | y | z
  Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

• Backus-Naur Form (BNF):
  <stmt> ::= <id> := <digit>
  <id> ::= <letter> | <id> <letter> | <id> <digit>
  <letter> ::= a | b | c | ... | x | y | z
  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Types of Grammars

• **Context Free Grammars:**
  - Every production has a single nonterminal on the left-hand side: \( A \rightarrow \ldots \)
  - Disallowed: \( X A \rightarrow X a \)

• **Regular Grammars:**
  - Productions take the form: \( A \rightarrow c \), or are all either left-linear: \( A \rightarrow B a \), or right-linear: \( A \rightarrow a B \)
  - Disallowed: \( S \rightarrow a S b \)
  - Cannot generate the language \( \{ a^n b^n | n = 1,2,3, \ldots \} \)
Types of Grammar

- **Context Free Grammars (CFGs)** are used to specify the overall structure of a programming language:
  - if/then/else, ...
  - brackets: ( ), { }, begin/end, ...

- **Regular Grammars (RGs)** are used to specify the structure of tokens:
  - identifiers, numbers, keywords, ...

- **Note**: The recognition problem for CFGs and RGs requires a different computational model (more on this later).
Q: Generate \( x2 := 0 \) in this grammar (call it \( G \))?

\[
\begin{align*}
<\text{stmt}> & \rightarrow <\text{id}> := <\text{digit}> \\
& \rightarrow <\text{id}> <\text{digit}> := <\text{digit}> \\
& \rightarrow <\text{letter}> <\text{digit}> := <\text{digit}> \\
& \rightarrow x <\text{digit}> := <\text{digit}> \\
& \rightarrow x 2 := <\text{digit}> \\
& \rightarrow x 2 := 0
\end{align*}
\]

Yes! This is a leftmost or canonical derivation in \( G \).
Q: Recognize \( x2 := 0 \) as a terminal sequence in \( L(G) \)?

\[
\begin{align*}
x 2 := 0 & \rightarrow <\text{letter}> 2 := 0 \\
& \rightarrow <\text{id}> 2 := 0 \\
& \rightarrow <\text{id}> <\text{digit}> := 0 \\
& \rightarrow <\text{id}> := 0 \\
& \rightarrow <\text{id}> := <\text{digit}> \\
& \rightarrow <\text{stmt}>
\end{align*}
\]

Yes! This is a parse of the sentence \( x2 := 0 \) in \( G \).
Each internal node is a nonterminal; its children are drawn from the right-hand side of one of the productions for that nonterminal.
Gammas are not Unique

- Consider a grammar $G'$:
  
  ```
  <stmt> ::= <ident> ::= <digit>
  <ident> ::= <letter> | <ident> <char>
  <char> ::= <letter> | <digit>
  <letter> ::= a | b | c | ... | x | y | z
  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  ```

- The grammar $G'$ generates the same language as $G$, but it has different parse trees.
Gammars are not Unique

Parse Tree for G

Parse Tree for G’
Ambiguity

\[ S \rightarrow NP \ VP \]
\[ NP \rightarrow \text{Name} | \text{Det} \ \text{Noun} | NP \ PP \]
\[ PP \rightarrow \text{Prep} \ NP \]
\[ VP \rightarrow \text{Verb} | \text{Verb} \ NP \]
\[ \text{Name} \rightarrow \text{john} | \text{mary} \]
\[ \text{Det} \rightarrow a | \text{the} | \text{some} | \text{every} \]
\[ \text{Prep} \rightarrow on | with | under | ... \]
\[ \text{Noun} \rightarrow \text{man} | \text{hill} | \text{telescope} | ... \]
\[ \text{Verb} \rightarrow \text{saw} | \text{runs} | \text{likes} | ... \]
Ambiguity

... a man on a hill with a telescope

NP
  /  
NP  PP
   /   
NP  PP
  /    
Det Noun Prep NP
  /    
Det Noun
Ambiguity
Here is a simplified grammar for Pascal:

\[
<\text{stmt}> ::= <\text{if-stmt}> \mid <\text{assign}> \mid ...
\]

\[
<\text{if-stmt}> ::= \text{if} <\text{expr}> \text{ then } <\text{stmt}> \mid \\
\text{if} <\text{expr}> \text{ then } <\text{stmt}> \text{ else } <\text{stmt}>
\]

\[
<\text{assign}> ::= <\text{id}> ::= <\text{digit}>
\]

\[
<\text{expr}> ::= <\text{id}> = 0
\]

\[
<\text{id}> ::= a \mid b \mid c \mid ... \mid x \mid y \mid z
\]

\[
<\text{digit}> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

How are compound “if” statements parsed using this grammar?
if \( x = 0 \) then if \( y = 0 \) then \( z := 1 \) else \( w := 2 \)

Parse Tree 1

```plaintext
<stmt>
  <if-stmt>
    if <expr> then <stmt>
    <id> = 0
    x
    <if-stmt>
      if <expr> then <stmt> else <stmt>
      <id> = 0
      <assign>
        y
        <id> := <digit>
        z
        1
      <assign>
        <id> := <digit>
        w
        2
  </stmt>
</if-stmt>
```
if \( x = 0 \) then if \( y = 0 \) then \( z := 1 \) else \( w := 2 \)

Parse Tree 2

Q: which tree is correct?
How to Fix Dangling Else?

- **Algol60**: use block structure
  
  ```
  if x = 0 then begin if y = 0 then z := 1 end else w := 2
  ```

- **Algol68**: use statement begin/end markers
  
  ```
  if x = 0 then if y = 0 then z := 1 fi else w := 2 fi
  ```

- **Pascal**: change the grammar of “if” statement to disallow the second parse tree, i.e., *always associate an “else” with the closest “if”*. 
Here is a revised grammar for Pascal:

\[
<\text{stmt}> ::= <\text{stmt1}> \mid <\text{stmt2}>
\]
\[
<\text{stmt1}> ::= \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt1}> \ \text{else} \ <\text{stmt1}> \mid \\
\quad <\text{assign}> \mid ...
\]
\[
<\text{stmt2}> ::= \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt}> \mid \\
\quad \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt1}> \ \text{else} \ <\text{stmt2}>
\]
\[
<\text{assign}> ::= <\text{id}> := <\text{digit}>
\]
\[
<\text{expr}> ::= <\text{id}> = 0
\]
\[
<\text{id}> ::= a \mid b \mid c \mid ... \mid x \mid y \mid z
\]
\[
<\text{digit}> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]
if $x = 0$ then if $y = 0$ then $z := 1$ else $w := 2$

In the new grammar there is only one parse tree!
Here is a grammar for arithmetic expressions:

\[
<\text{expr}> ::= <\text{expr}> + <\text{expr}> \mid <\text{expr}> - <\text{expr}> \mid <\text{expr}> \times <\text{expr}> \mid <\text{expr}> / <\text{expr}> \mid <\var> \mid <\text{num}>
\]

\[
<\var> ::= a \mid b \mid c \mid \ldots \mid x \mid y \mid z
\]

\[
<\text{num}> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

Using this grammar, how would we parse: \( x + 3 \times y \) ?
Two Parse Trees

```
(expr)  (expr)
  |    |    |
  +   +   *   *
  |    |    |    |
(var) (var) (num) (num)
  |    |    |    |
x    y    3    y
```

```
(expr)  (expr)
  |    |    |
  *   +   *
  |    |    |
(var) (var) (num) (num)
  |    |    |    |
x    y    x    3
```
Modify the grammar to add precedence:

\[
\begin{align*}
<\text{expr}> &::= <\text{expr}> + <\text{expr}> \mid <\text{expr}> - <\text{expr}> \mid <\text{term}> \\
<\text{term}> &::= <\text{term}> \ast <\text{term}> \mid <\text{term}> / <\text{term}> \mid <\text{factor}> \\
<\text{factor}> &::= <\text{var}> \mid <\text{num}> \mid ( <\text{expr}> ) \\
<\text{var}> &::= a \mid b \mid c \mid \ldots \mid x \mid y \mid z \\
<\text{num}> &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

Using this grammar, how would we parse: \( x + 3 \ast y \)?
Using this grammar, how would we parse: \( 7 - 4 - 2 \)?
Only One Parse Tree

But there are two parse trees for the second example:

```
<e>
  /   
<e> - 2
  /   
7 - 4
```

```
<e>
  /   
7 - <e>
  /   
4 - 2
```

```
<expr >
  /   
<term>
  /   
<factor >
  /   
<var >
   x
```

```
<expr >
  /   
<term>
  /   
<factor >
  /   
<var >
   x
```

```
<term>
  /   
<factor >
  /   
<num>
   3
```

```
<term>
  /   
<factor >
  /   
<var >
   y
```
Modify the grammar to add associativity:

\[ \begin{align*}
\text{<expr>} & ::= \text{<expr>} + \text{<term>} \mid \text{<expr>} - \text{<term>} \mid \\
& \quad \text{<term>}
\end{align*} \]

\[ \begin{align*}
\text{<term>} & ::= \text{<term>} \ast \text{<factor>} \mid \text{<term>} / \text{<factor>} \mid \\
& \quad \text{<factor>}
\end{align*} \]

\[ \begin{align*}
\text{<factor>} & ::= \text{<var>} \mid \text{<num>} \mid (\text{<expr>})
\end{align*} \]

\[ \begin{align*}
\text{<var>} & ::= \text{a} \mid \text{b} \mid \text{c} \mid \ldots \mid \text{x} \mid \text{y} \mid \text{z}
\end{align*} \]

\[ \begin{align*}
\text{<num>} & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*} \]

Using this grammar, how would we parse: 7 - 4 - 2?
Only One Parse Tree

```
<expr>  
  /    
<expr>  -  <term>  
    |    
  <factor>  
    |    
<term>  
  |    
<factor>  
    |    
<num>  
  |    
2
```

```
<expr>  
  /    
<expr>  -  <term>  
    |    
<term>  
  |    
<factor>  
    |    
<num>  
  |    
7
```
Concrete vs. Abstract Syntax

- **Concrete Syntax**
  - `<expr>`
  - `<expr> + <expr>`
  - `<term>`
  - `<term>`
  - `<factor>`
  - `<var>`
  - `x`
  - `<term> * <term>`
  - `<factor>`
  - `<factor>`
  - `<num>`
  - `<var>`
  - `3`
  - `y`

- **Abstract Syntax**
  - `+`
  - `x`
  - `*`
  - `3`
  - `y`

---

**Pradhan**  
**Lecture 01: Programming Languages (CMPU-235)**
Write nonterminals as in BNF. (Variant: Write them with initial capital letters, or using a different font.)

Use additional *metasymbols*, as shortcuts:

- {...} means repeat the enclosed text zero or more times
- […] means the enclosed text is optional
- (...) is used for grouping, usually with the alternation symbol, e.g., (... | ...).

If { }, [ ], or ( ) are used as terminal symbols in the language being defined, then they must be quoted. (Variant: They must be underlined.)
Examples:

<expr> ::= <term> { ( + | - ) <term> }
<term> ::= <factor> { ( * | / ) <factor> }
<factor> ::= <var> | <num> | ‘(’ <expr> ‘)’

<if-stmt> ::= if <expr> then <stmt> [ else <stmt> ]

<identifier> ::= <letter> { ( <letter> | <digit> ) }