Inductive Definitions
Languages, Grammars

Motivation: Parsing
- Parsing is the act of interpreting text as meaningful information.
- Example:
  - Programming language: Parsing interprets the language as an executable program.
  - Calculator: Parsing interprets the symbols to carry out the calculation being represented. $345 + 62.7*84.9$ doesn’t have a “magical” meaning; we have to give it one.

Grammars, and Induction
- Grammars provide a plan for parsing; they define the syntax of a language.
- Grammars are an instance of a more general concept: Inductive Definitions.
- rex rules are often inductive definitions; but grammars may be non-deterministic for a reason.

Inductive Definitions
- Inductive definitions are the main "constructive" way to define infinite sets.
- We will need infinite sets in much of what follows.

Elements of an inductive definition of a set $S$:
- Basis
- Induction rule(s)
- Extremal clause

Components of an inductive definition of a set $S$:
- Basis: Defines a few items to be in $S$.
- Induction rule(s): Introduce new items in $S$ based on existence of other, usually simpler, items.
- Extremal clause: Says that the only items in $S$ are those derivable by the previous two components, applied a finite number of times.
Example of ID: Binary Trees

- It is a binary tree.
- If $T_1$ and $T_2$ are binary trees, then so is:

  \[ \begin{array}{c}
    T_1 \\
    \text{Tree}
  \end{array} \quad \begin{array}{c}
    T_2 \\
    \text{Tree}
  \end{array} \]

- Extremal clause: The only binary trees are those constructible by a finite number of applications of the above rules.

Examples of Binary Trees

Example of ID: Natural Numbers

- Basis: $0$ is in $[]$.
- Induction: If $n$ is in $[]$, so is the successor of $n$ (variously denoted $n'$, $S(n)$, or $n+1$).
- Extremal: The only elements in $[]$ are those derivable by applications of the above rules.
- Examples: $0$, $0'$, $0''$, $0'''$, ... are all elements of $[]$.

Notes

- Set $[]$ is an infinite set.
- The members of $[]$ are all finite.
- Set $[]$ does not contain infinity ($\infty$) as an element.

Interpretations of Successor ($'$)

- What are $0'$, $0''$, $0'''$, ... really?
  - Strings of symbols, or
  - Things that can be constructed from sets, a more primitive concept.
  - Two variations:
    - $0'$ is $\emptyset$, the empty set; $X'$ is the set $X$.
    - $0'$ is $\emptyset$, the empty set; $X'$ is the set $X \cup \{X\}$.
  - In the second example: $0' = \emptyset$, $0'' = \{\emptyset\}$, $0''' = \{\emptyset, \{\emptyset\}\}$, ...
  - Advantage: $0'n$ is a set with $n$ distinct members.

Decimal Numerals

- We can agree by convention that
  - $1$ stands for $0'$,
  - $2$ stands for $0''$,
  - ...
  - $9$ stands for $0'''$.
  - Beyond that, give an algorithm for generating additional numerals:
    - $10$, $11$, $12$, $13$, ...
### Decimal Numbering Rule
- The successor of $x_0$ (concatenation) is $x_1$, the successor of $x_1$ is $x_2$, $\ldots$, and the successor of $x_8$ is $x_9$.
- The successor of $x_9$ is $y_0$ where $y$ is the successor of $x$.
- Example: $0, 1, \ldots, 9, 10, 11, \ldots, 19, 20, 21, \ldots, 99, 100, \ldots$

### 1-adic Numerals
- The only digit is 1.
- The empty string (denoted $\emptyset$ so it is readable) stands for 0.
- $\mathbf{IX}$ (I followed by X) stands for $X'$.
- The numerals are: $\emptyset, 1, 11, 111, 1111, \ldots$
- Could also use lists: $[ ], [1], [1, 1], [1, 1, 1], \ldots$

### 2-adic Numerals
- The digits are 1 and 2.
- The empty string (denoted $\emptyset$ so it is visible) stands for 0.
- The numerals are: $\emptyset, 1, 2, 11, 12, 21, 22, 111, 112, \ldots$
- Unlike binary numerals, there is no redundancy ($1, 01, 001, 0001, \ldots$ all mean the same thing in binary).

### Roman Numerals
- The digits are I, V, X, L, C, D, M.
- There is no string for 0.
- The successor of I is $s(I) = II$, $s(II) = III$, $s(III) = IV$, etc.

### Numerals vs. Numbers
- **Numbers** are abstract.
- **Numerals** are a concrete representation of numbers.

### Strings over an alphabet $\mathcal{S}$
- The set of all finite strings over an alphabet $\mathcal{S}$ is denoted $\mathcal{S}^*$.
- Example:
  - $(a, b)^* = \{ \emptyset, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots \}$
Strings over an alphabet $\Sigma$

- **Basis:** $\epsilon$, the empty string, is in $\Sigma^*$.
- **Inductive rule:** If $x \in \Sigma^*$ and $s \in \Sigma$, then $s \cdot x$ (followed by $x$) is in $\Sigma^*$.
- **Extremal clause.**

Languages

- A language over $\Sigma$ is any subset of $\Sigma^*$, the set of all strings over $\Sigma$.
- Examples, where $\Sigma = \{a, b\}$:
  - $(a, b)^*$ itself
  - $\emptyset$, the empty language
  - $(ba, bba)$, maybe your first language
  - $(\epsilon), aa, aaaa, aaaaa, \ldots)$ the language of an even number of $a$'s.

More Languages

- **Basis:** $ab$ is in $L$.
- **Inductive rules:**
  - If $x$ is in $L$, so is $axb$.
  - If $x_1$ and $x_2$ are in $L$, so is $x_1x_2$.

Non-Trivial Language Defined Inductively

- $L = \{ab, abab, aabb, aababb, \ldots\} \subseteq \Sigma^*$
- **Basis:** $ab$ is in $L$.
- **Inductive rules:**
  - If $x$ is in $L$, so is $axb$.
  - If $x_1$ and $x_2$ are in $L$, so is $x_1x_2$.

Languages

- There are lots of languages, some very weird.
- To be of computational interest, a language needs to be defined inductively.
- We need a way of telling whether a given string is in the language or not (called parsing the string).

Grammars: A Shorthand

- Spelling everything out with these inductive definitions is laborious.
- We need a shorthand, especially for more complex languages.
- The idea comes from linguistics and early work on computer languages.
Grammar Definition

- There is a "start symbol", or "root", say $S$, which is not in the alphabet of the language itself.
- $Æ$ is a symbol meaning "can be rewritten as".
- Grammar rules, for example:
  1. $S Æ ab$
  2. $S Æ aSb$
  3. $S Æ SS$
- Apply rules by "free choice".
- A sequence of applications is called a derivation.
- The strings in the language are those that don’t include $S$.

Using the Grammar Rules

- Grammar rules:
  1. $S Æ ab$
  2. $S Æ aSb$
  3. $S Æ SS$
- Example derivations of strings in the language:
  1. $S Æ ab$
  2. $S Æ aSb Æ aabb$
  3. $S Æ SS Æ abS Æ abab$
  4. $S Æ SS Æ aSbS Æ aabbS Æ aabbaSb Æ aabbaabb$

Generalizing Grammar Rules

- Instead of just $S$, allow multiple symbols, called auxiliaries, none of which are in the alphabet of the language.
- A distinguished auxiliary is called the root or "start symbol".
- The symbols in the alphabet of the language are called terminals.
- The rules are known as productions.

Example: Grammar for Additive Arithmetic Expressions

- The root is $A$.
- The terminals are $\{a, b, c, +\}$.
- The productions are:
  - $A Æ V$
  - $A Æ V + A$
  - $V Æ a$
  - $V Æ b$
  - $V Æ c$

Example Derivations

- The productions are:
  - $V Æ a$
  - $V Æ b$
  - $V Æ c$
  - Group by common left-hand sides
  - Use $|$ (read "or") to represent alternatives:
    - $A Æ V | V + A$
    - $V Æ a | b | c$
  - Note: $|$ "binds more loosely" than other symbols.
  - Same grammar, just a briefer notation.

Shorthands on top of Shorthands

- The productions are:
  - $V Æ a$
  - $V Æ b$
  - $V Æ c$
- Group by common left-hand sides
- Use $|$ (read "or") to represent alternatives:
  - $A Æ V | V + A$
  - $V Æ a | b | c$
  - Note: $|$ "binds more loosely" than other symbols.
  - Same grammar, just a briefer notation.
Derivation Tree Visualization

Terminal string = red “fringe” of tree = “c + a + b”

Syntax Tree (= Derivation Tree)
Shows Implied “Interpretation” of String

Right Grouping (used so far) vs. Left Grouping Productions

Does Grouping Matter?
- Mathematically, + is an associative operator:
  \((a + b) + c == a + (b + c)\)
- However:
  - There are non-associative operators, such as -, where it does matter.
    \((a - b) - c \neq a - (b - c)\)
  - Also, on computers, for floating point addition, associativity does not always hold.

Floating Point is Not Associative

- Try this:
  - `sumup(m, n) = m > n ? 0 : 1.0 + sumup(m+1, n);`
  - `sumdown(m, n) = m > n ? 0 : 1.0 + sumdown(m, n-1);`
  - `test(n) = sumup(1, n) == sumdown(1, n);`
  - `map(test, range(1, 100));`
- Sensitivity to grouping is due to round-off error.

Precedence Issue
(multiple operator symbols)

- How do we ensure that the syntax tree of
  \(a + b \times c\)
looks like this: and not this:
Multiple Auxiliaries

- We want " to "bind more tightly" than +.
- Use a different auxiliary symbol for each level of precedence.
- Arrange it so that expansions from the tighter binding auxiliary symbol can only be done after those of the looser binding auxiliary.

Precedence Issue

- We must ensure that the derivation tree for a+b*c looks like this: and not this:

![Derivation Tree Diagram]

Example: Grammar for Additive & Multiplicative Arithmetic Expressions

- The root is A.
- The terminals are \{a, b, c, +, *\}.
- The productions are:
  - A : M + A | M
  - M : V * M | V
  - V : a | b | c
- Intuitive rule: Operator "farther from the root" binds more tightly

Syntactic Categories

- The various auxiliary symbols typically represent syntactic categories: sets of sub-expressions having a certain type of meaning.
- Categories:
  - A : M + A | M A is a "sum"
  - M : V * M | V M is a "product"
  - V : a | b | c V is a "variable"

Example Derivations

- The productions are:
  - A : M + A | M
  - M : V * M | V
  - V : a | b | c
- Sample derivations (A is the syntactic category):
  1. A : M + V * a
  2. A : M + A | V + A | a + A | a + M | a + V | a + b
  3. A : M + A | V + M | a | V | a + V | M
  4. A : M + A | V + M | a | V | a + V | A
     a + A | a + V | a + V | a + V | a + V | a + V | a + V

Example Syntactic Categories

- The productions are:
  - A : M + A | M
  - M : V * M | V
  - V : a | b | c
- Sample sub-derivations, e.g. from M:
  1. M : V | V | a
  2. M : V * M | a | V | M | a | V | M | a | b | a | V | M | a | b | V | M | a | b | a
  3. M : V | V | M | a | V | M | a | V | M | a | b | V | M | a | b | V | M | a | b | a
- Observation: Derivations from M will never include any +'s.
<table>
<thead>
<tr>
<th>Exercise: Include ^ (power)</th>
<th>How to handle '()'</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ^ binds the most tightly</td>
<td>• Parentheses means</td>
</tr>
<tr>
<td>• * is next</td>
<td>“handle inside as a single unit”</td>
</tr>
<tr>
<td>• + is the weakest</td>
<td>• Parallel level to a single variable</td>
</tr>
<tr>
<td></td>
<td>• Sometimes called “primaries”</td>
</tr>
</tbody>
</table>