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.
    - $345 + 62.7 \times 84.9$
    - 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.
Inductive Definitions

- Elements of an inductive definition of a set S.
  - Basis
  - Induction rule(s)
  - Extremal clause
**Inductive Definitions**

- **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

- is a binary tree.

- If $T_1$ and $T_2$ are binary trees, then so is:

```
    *
   /\   /
  /   \ /   \
 T_1   T_2
```

- 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 \( \mathbb{N} \).
- **Induction**: If \( n \) is in \( \mathbb{N} \), so is the successor of \( n \) (variously denoted \( n' \), \( S(n) \), or \( n+1 \)).
- **Extremal**: The only elements in \( \mathbb{N} \) are those derivable by applications of the above rules.
- **Examples**: 0, 0', 0'', 0''', \ldots are all elements of \( \mathbb{N} \).
Set $\mathbb{I}$ is an infinite set.

The members of $\mathbb{I}$ are all finite.

Set $\mathbb{I}$ 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 {}, the empty set; X' is the set {X}, or
      - 0 is {}, the empty set; X' is the set X \{X\}.
    - In the second example: 0 is {}, 0' is {{}}, 0'' is {{}, {{}}}, 0''' is {{}, {}, {{}, {{}}}}, ...
    - Advantage: On's is a set with n distinct members.
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 $x_2$, …, 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, …, 9, 10, 11, …, 19, 20, 21, …, 99, 100, …
1-adic Numerals

- The only digit is 1.
- The empty string (denoted so it is readable) stands for 0.
- 1X (1 followed by X) stands for X’.
- The numerals are:
  - , 1, 11, 111, 1111, 1111, ...
- Could also use lists: [], [1], [1, 1], [1, 1, 1], ...
2-adic Numerals

- The digits are 1 and 2.
- The empty string (denoted so it is visible) stands for 0.
- The numerals are:

  0, 1, 2, 11, 12, 21, 22, 111, 112, ...

- Unlike binary numerals, there is no redundancy (1, 01, 001, 0001, ... 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.
The set of all finite strings over an alphabet $\Sigma$ is denoted $\Sigma^*$. 

Example:

$$\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\}$$
Strings over an alphabet

- **Basis:** $\lambda$, the empty string, is in $S^*$. 

- **Inductive rule:** If $x \in S^*$ and $s \in S$, then $s \cdot x$ (s followed by x) is in $S^*$. 

- **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
  - $\{\} \cup \{ba, baba\}$, maybe your first language
  - $\{\emptyset, aa, aaaa, aaaa, aaaaaa, \ldots\}$ the language of an even number of $a$’s.
More Languages

- More Examples, where \( \Sigma = \{a, b\} \)
  - \( \{\emptyset, ab, ba, aabb, abab, baab, bbaa, aaabbb, aababb, \ldots\} \) the language in which the number of a’s equals the number of b’s.
  - \( \{a, b, aa, bb, aab, aba, baa, abb, bab, bba, \ldots\} \) the language in which the number of a’s is not equal to the number of b’s.
  - \( \{ab, abab, aabb, aababb, \ldots\} \) a language you might recognize.
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).
Non-Trivial Language Defined Inductively

- \( L = \{ab, abab, aabb, aababb, \ldots\} \)
- **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 \( x_1x_2 \).
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 \textit{not} in the alphabet of the language itself.
- $\Rightarrow$ is a symbol meaning “can be rewritten as”.

Grammar rules, for example:

1. $S \Rightarrow ab$
2. $S \Rightarrow aSb$
3. $S \Rightarrow SS$

- Apply rules by “free choice”.
- A sequence of applications is called a \textit{derivation}.
- The strings in the language are those that don’t include S.
Using the Grammar Rules

Grammar rules:
1. $S \rightarrow ab$
2. $S \rightarrow aSb$
3. $S \rightarrow SS$

Example derivations of strings in the language:
1. $S \rightarrow ab$
2. $S \rightarrow aSb \rightarrow aabb$
3. $S \rightarrow aSb \rightarrow aaSbb \rightarrow aaabbb$
4. $S \rightarrow SS \rightarrow abS \rightarrow abab$
5. $S \rightarrow SS \rightarrow SSS \rightarrow ababab$
6. $S \rightarrow SS \rightarrow aSbS \rightarrow aabbS \rightarrow aabbaSb \rightarrow aabbaabb$
Instead of just $S$, allow multiple symbols, called \textit{auxiliaries}, none of which are in the alphabet of the language.

A distinguished auxiliary is called the \textit{root} or “\textit{start symbol}”.

The symbols in the alphabet of the language are called \textit{terminals}.

The rules are known as \textit{productions}.
Example:
Grammar for Additive Arithmetic Expressions

- The root is A.
- The terminals are \{a, b, c, +\}.
- The productions are:
  - \(A \rightarrow V\)
  - \(A \rightarrow V + A\)
  - \(V \rightarrow a\)
  - \(V \rightarrow b\)
  - \(V \rightarrow c\)
Example Derivations

The productions are:

- $A \rightarrow V$
- $A \rightarrow V + A$
- $V \rightarrow a$
- $V \rightarrow b$
- $V \rightarrow c$

Sample derivations:

1. $A \rightarrow V \rightarrow a$
2. $A \rightarrow V \rightarrow c$
3. $A \rightarrow V + A \rightarrow c + A \rightarrow c + V \rightarrow c + a$
4. $A \rightarrow V + A \rightarrow c + A \rightarrow c + V + A \rightarrow c + b + A \rightarrow c + b + V \rightarrow c + b + a$
The productions are:
- $A \rightarrow V$
- $A \rightarrow V + A$
- $V \rightarrow a$
- $V \rightarrow b$
- $V \rightarrow c$

*Group* by common left-hand sides

Use $|$ (read “or”) to represent alternatives:
- $A \rightarrow V \mid V + A$
- $V \rightarrow a \mid b \mid c$

Note: $|$ “binds more loosely” than other symbols.

Same grammar, just a briefer notation.
Derivation Tree Visualization

$$A \rightarrow V | V + A$$
$$V \rightarrow a | b | c$$

Arrows indicate that a production is being applied.

Terminal string = red "fringe" of tree = “c + a + b”
Syntax Tree (=! Derivation Tree)
Shows Implied "Interpretation" of String

Derivation Tree

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

Right-grouping production

Left-grouping production

Derivation Tree

Syntax Tree

Syntax Tree

Derivation Tree

A → A + V | V
V → a | b | c

A → A + V | V
V → a | b | c

A → V + A | V
V → a | b | c

A → V + A | V
V → a | b | c
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.
    \[(a - b) - c != a - (b - c)\]
  - 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./m + sumup(m+1, n);`
  - `sumdown(m, n) = m > n ? 0 : 1./n + sumdown(m, n-1);`
  - `test(n) = sumup(1, n) == sumdown(1, n);`
  - `map(test, range(1, 100));`
  - `[1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`

- **Sensitivity to grouping is due to round-off error.**
How do we ensure that the syntax tree of \(a + b\times c\) looks like this: \(\begin{array}{c}
+ \\
\downarrow \\
a \\
\downarrow \\
b \\
\downarrow \\
c
\end{array}\) and not this: \(\begin{array}{c}
\times \\
\downarrow \\
a \\
\downarrow \\
b \\
\downarrow \\
c
\end{array}\)
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.
We must ensure that the **derivation** tree for \( a+b\times c \) looks like this: and not this:

- **Precedence Issue**

- **We must ensure that the derivation tree for** \( a+b\times c \) **looks like this:**

- **and not this:**

- (loose)

- (tight)
Example:
Grammar for Additive & Multiplicative Arithmetic Expressions

- The root is $A$.
- The terminals are $\{a, b, c, +, *\}$.
- The productions are:
  - $A \rightarrow M + A | M$
  - $M \rightarrow V * M | V$
  - $V \rightarrow a | b | c$
- Intuitive rule: Operator “farther from the root” binds more tightly
The various auxiliary symbols typically represent **syntactic categories**: sets of sub-expressions having a certain type of meaning.

**Categories:**
- \( A \rightarrow M + A \mid M \) \( A \) is a “sum”
- \( M \rightarrow V * M \mid V \) \( M \) is a “product”
- \( V \rightarrow a \mid b \mid c \) \( V \) is a “variable”
Example Derivations

- The productions are:
  - $A \rightarrow M + A \mid M$
  - $M \rightarrow V^*M \mid V$
  - $V \rightarrow a \mid b \mid c$

- Sample derivations ($A$ is the syntactic category):
  1. $A \rightarrow M \rightarrow V \rightarrow a$
  2. $A \rightarrow M + A \rightarrow V + A \rightarrow a + A \rightarrow a + M \rightarrow a + V \rightarrow a + b$
  3. $A \rightarrow M + A \rightarrow V + A \rightarrow a + A \rightarrow a + M \rightarrow a + V^*M \\
     \rightarrow a + b^* M \rightarrow a + b^* c$
  4. $A \rightarrow M + A \rightarrow V^*M + A \rightarrow a^*M + A \rightarrow a^*V + A \\
     \rightarrow a^*b + A \rightarrow a^*b + M \rightarrow a^*b + V \rightarrow a^*b + c$
Example Syntactic Categories

- The productions are:
  - $A \rightarrow M + A \mid M$
  - $M \rightarrow V^*M \mid V$
  - $V \rightarrow a \mid b \mid c$

- Sample sub-derivations, e.g. from $M$:
  1. $M \rightarrow V \rightarrow a$
  2. $M \rightarrow V^*M \rightarrow a^*M \rightarrow a^*V \rightarrow a^*b$
  3. $M \rightarrow V^*M \rightarrow a^*M \rightarrow a^*V^*M \rightarrow a^*b^*M \rightarrow a^*b^*V \rightarrow a^*b^*a$

- Observation: Derivations from $M$ will never include any $+$'s.
Exercise: Include ^ (power)

- ^ binds the most tightly
- * is next
- + is the weakest
How to handle ‘(’     ’)’

- Parentheses means
  “handle inside as a single unit”

- Parallel level to a single variable
  • Sometimes called “primaries”