### Inductive Definitions

**Languages**

**Grammars**

** Parsing**

---

### Inductive Definitions

What do these seemingly-unrelated topics have to do with each other?

---

### Inductive Definitions

**Languages**

**Grammars**

** Parsing**

---

### Inductive Definitions

**Languages**

**Grammars**

** Parsing**

---

### 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**: 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 elements, applied any finite number of times.

---

### Example of ID: Binary Trees

- $T$ is a binary tree.
- If $T_1$ and $T_2$ are binary trees, then so is:
  
  ![Binary Tree](image)

- **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 $\omega$

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

Notes

- $\omega$ is an infinite set.
- $\omega$'s members are all finite.
- $\omega$ 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.
  - Examples:
    - $0$ is $\{\}$, the empty set; $X'$ is the set $X$.
    - $0$ is $\{\}$, the empty set; $X'$ is the set $X \cup \{X\}$.
  - In the second example: $0$ is $\{\}$, $0'$ is $\{\}, \{\{\}\}$, $0''$ is $\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}$, ...
  - 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, ...

1-adic Numerals

- The only digit is 1.
- The empty string (denoted $\lambda$ so it is readable) stands for 0.
- 1X (1 followed by X) stands for $X'$.
- The numerals are:
  - $\lambda$, 1, 11, 111, 1111, 11111, ...
- Could also use lists: [ ], [1], [1, 1], [1, 1, 1], ...
2-adic Numerals

- The digits are 1 and 2.
- The empty string (denoted \( \lambda \) so it is visible) stands for 0.
- The numerals are:
  \( \lambda, 1, 2, 11, 12, 21, 22, 111, 112, \ldots \)
- 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.
- You know the rest.

Numerals vs. Numbers

- Numbers are abstract.
- Numerals are a concrete representation.

Strings over an alphabet \( \Sigma \)

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

Strings over an alphabet \( \Sigma \)

- Basis: \( \lambda \) is in \( \Sigma^* \).
- Inductive rule: If \( x \in \Sigma^* \) and \( \sigma \in \Sigma \), then \( x\sigma \) (\( x \) followed by \( \sigma \)) is in \( \Sigma^* \).
- Extremal clause.

Languages

- A language over \( \Sigma^* \) is any subset of \( \Sigma^* \).
- Examples, where \( \Sigma = \{a, b\} \)
  - \( (a, b)^* \) itself
  - \( \{} \) the empty language
  - \( \{ba, baba\} \) maybe your first language
  - \( \{\lambda, aa, aaaa, aaaaa, aaaaaa, \ldots\} \) the language of an even number of a's.
More Languages

- Examples, where $\Sigma = \{a, b\}$
  - $(\lambda, ab, ba, aabb, abab, baab, bbaa, aaabb, aababb, \ldots)$ 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.

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).

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.

Grammatical Definition

- $S$ is a symbol not in the alphabet of the language.
- $\rightarrow$ is a symbol meaning “can be rewritten as”.
- Grammar rules:
  - $S \rightarrow ab$
  - $S \rightarrow aSb$
  - $S \rightarrow SS$
- Application of rules is "non-deterministic".
- 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:
  - $S \rightarrow ab$
  - $S \rightarrow aSb$
  - $S \rightarrow SS$
- Example derivations of strings in the language:
  - $S \Rightarrow ab$
  - $S \Rightarrow aSb \Rightarrow aabb$
  - $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabb$
  - $S \Rightarrow SS \Rightarrow abS \Rightarrow abab$
  - $S \Rightarrow SS \Rightarrow SS \Rightarrow ababab$
  - $S \Rightarrow SS \Rightarrow aSbS \Rightarrow aabSb \Rightarrow aababaab$
**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:
  - \( A \rightarrow V \)
  - \( A \rightarrow V + A \)
  - \( V \rightarrow a \)
  - \( V \rightarrow b \)
  - \( V \rightarrow c \)

**Example Derivations**

- The productions are:
  - \( A \rightarrow V \)
  - \( A \rightarrow V + A \)
- Sample derivations:
  - \( A \Rightarrow V \Rightarrow a \)
  - \( A \Rightarrow V \Rightarrow c \)
  - \( A \Rightarrow V + A \Rightarrow c + A \Rightarrow c + V \Rightarrow c + a \)
  - \( A \Rightarrow V + A \Rightarrow c + A \Rightarrow c + V + A \Rightarrow c + b + A \Rightarrow c + b + V \Rightarrow c + b + a \)

**Shorthands on top of Shorthands**

- The productions are:
  - \( A \rightarrow V \)
  - \( A \rightarrow V + A \)
- Group by common left-hand sides
- Use \( | \) (read "or") to represent alternatives:
  - \( A \rightarrow V | V + A \)
  - \( V \rightarrow a | b | c \)
- 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 Meaning of String

Terminal string = red "fringe" of tree = "c + a + b"
Right Grouping (used so far) vs. Left Grouping Productions

Syntax Tree

\[
A \rightarrow A + V | V
\]

\[
V \rightarrow a | b | c
\]

A\[+\]V \[+\]V

\[
A\[+\]V
\]

\[
Vb
\]

\[
c
\]

A

Derivation Tree

Syntax Tree

A

\[
A \rightarrow V + A | V
\]

\[
V \rightarrow a | b | c
\]

A[V+A]

A[V+A]

\[
Vb
\]

\[
c
\]

A

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.
  - \[(a - b) - c \neq a - (b - c)\]
  - On computers, for floating point addition, associativity does not always hold.

Floating Point is Not Associative

- Try this:
  - \[\text{sumup}(m, n) = m > n ? 0 : 1./m + \text{sumup}(m+1, n)\; ;\]
  - \[\text{sumdown}(m, n) = m > n ? 0 : 1./n + \text{sumdown}(m, n-1)\; ;\]
  - \[\text{test}(n) = \text{sumup}(1, n) = = \text{sumdown}(1, n)\; ;\]
  - \[\text{map}(\text{test}, \text{range}(1, 100))\; ;\]
  - Grouping sensitivity is due to round-off error.

Precedence Issue

- To do so we must ensure that the derivation tree looks like this: and not this:

Example:
Grammar for Additive & Multiplicative Arithmetic Expressions

- The root is S.
- The terminals are \{a, b, c, +, *\}.
- The productions are:
  - \[S \rightarrow P + S \mid P\]
  - \[P \rightarrow V * P \mid V\]
  - \[V \rightarrow a \mid b \mid c\]
- 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:
  - \( S \rightarrow P + S \mid P \) \( S \) is a "sum"
  - \( P \rightarrow V \cdot P \mid V \) \( P \) is a "product"
  - \( V \rightarrow a \mid b \mid c \) \( V \) is a "variable"

Use of Syntactic Categories

- The use of syntactic categories will be seen when we start assigning meanings to expressions.

Example Derivations

- The productions are:
  - \( S \rightarrow P + S \mid P \)
  - \( P \rightarrow V \cdot P \mid V \)
  - \( V \rightarrow a \mid b \mid c \)
- Sample derivations (\( S \) is the syntactic category):
  - \( S \Rightarrow P \Rightarrow V \Rightarrow a \)
  - \( S \Rightarrow P + S \Rightarrow V \cdot S \Rightarrow a + S \Rightarrow a + P \Rightarrow a + V \Rightarrow a + b \)
  - \( S \Rightarrow P + S \Rightarrow V \cdot S \Rightarrow a + S \Rightarrow a + P \Rightarrow a + V \cdot P \Rightarrow a + b \cdot P \Rightarrow a + b \cdot c \)

Example Syntactic Categories

- The productions are:
  - \( S \rightarrow P + S \mid P \)
  - \( P \rightarrow V \cdot P \mid V \)
  - \( V \rightarrow a \mid b \mid c \)
- Sample sub-derivations:
  - Derivations from \( P \):
    - \( P \Rightarrow V \Rightarrow a \)
    - \( P \Rightarrow V \cdot P \Rightarrow a \cdot P \Rightarrow a \cdot V \Rightarrow a \cdot b \)
    - \( P \Rightarrow V \cdot P \Rightarrow a \cdot P \Rightarrow a \cdot V \cdot P \Rightarrow a \cdot b \cdot P \Rightarrow a \cdot b \cdot V \Rightarrow a \cdot b \cdot a \)
  - Observation: Derivations from \( P \) don't include any +

Two Main Language Problems

- Recognition problem:
  Is a given string in the language?
- Meaning problem:
  What is the meaning of a string if it is in the language?

Naïve Solution to the Recognition Problem

- To determine whether string \( x \) is in the language generated by a grammar:
  - Start with the start symbol.
  - Generate strings successively by applying productions.
  - Eventually either:
    - The string \( x \) is generated, or
    - The new strings being generated all exceed \( x \) in length.
  - So we can tell whether or not \( x \) is ever generated.
Parsing

- Parsing seeks to solve both problems:
  - Recognition
  - Meaning
- In addition, it tries to do recognition much more efficiently than the naïve solution.

Recursive Descent Parsing

- Simplest reasonably general form of parsing.
- Works for many, but not all grammars.
- Sometimes a grammar can be transformed to enable recursive descent.
- Recall that each auxiliary symbol in the grammar can be identified with a syntactic category, the set of strings that can be generated from that symbol (possibly with the help of other symbols). The meaning will derive from this idea.

Recursive Descent

- It's called "recursive" because in general grammar productions can "call" themselves or each other.
- It's called "descent" because parsing starts at the root of a "derivation tree" and proceeds toward the leaves.

Parse Methods

- For each auxiliary symbol in the grammar, construct a parse method
- Each parse method's responsibility is to recognize the longest string in the corresponding syntactic category in the remainder of the input, from the current point onward:

\[
S \rightarrow V + S | V \\
V \rightarrow a | b | c
\]

Example

- Consider the grammar with start symbol S:
  - \( S \rightarrow V + S | V \)
  - \( V \rightarrow a | b | c \)
  - The parse begins by trying to identify the entire input string as being in syntactic category S.
  - Clearly it must find a V to start.
    - To find a V, it checks to see whether the next symbol is one of these listed.
    - Having found a V, it checks to see if the next symbol is +.
      - If so, it recurses, trying to find another S.
      - If not, it stops.
    - After the top call to S returns, it checks to see whether there are any spurious remaining characters in the input.
      - If there are, the input is not accepted.
      - If not, the input is accepted.

Example: Success

- Suppose the input string is "a + b + c."
- Subscripts will indicate the particular instance of the method and the "argument" will indicate the unparsed remainder of the input.
  - The parser calls \( S_1(a + b + c) \).
  - \( S_1 \) calls \( V_1(a + b + c) \).
  - \( V_1 \) identifies a, returns success and unparsed input "b + c."
  - \( S_1 \) checks for + and finds it; therefore \( S_1 \) calls \( S_2(b + c) \).
  - \( S_2 \) calls \( V_2(b + c) \).
  - \( V_2 \) identifies b, returns success and unparsed input "c."
  - \( S_2 \) checks for + and finds it; therefore \( S_2 \) calls \( S_3(c) \).
  - \( S_3 \) calls \( V_3(c) \).
  - \( V_3 \) identifies c, returns success and unparsed input ""
  - \( S_3 \) checks for + and does not find it; therefore \( S_3 \) returns success with ""
  - \( S_2 \) returns success with ""
  - \( S_1 \) returns success with "". The string is accepted.
Example: Failure

- Suppose the input string is "a b + c".
- The parser calls $S("a b + c")$.
- $S$, calls $V("a b + c")$.
- $V$, identifies, returns success and unparsed input "b + c".
- $S$, checks for + and does not find it; therefore, $S$ returns success with "b + c".
- Since the top-level call to $S$ has returned, but there is residual input, the string is not accepted.

A rex version of parsing

- Each syntactic category will be a rex function.
- There is one argument:
  - the unparsed input, a list of characters.
- There are two results:
  - success or failure indicator
  - for failure: "failure"
  - for success: the Syntax Tree
  - the unparsed input.

A rex version of parsing (1)

```plaintext
// parse function for auxiliary S, rules S -> V | V + S
S(input) =>
  [result1, residue1] = V(input),                       // try V
  failed(result1) ?
    [FAILURE, residue1]                               // V failed
  : residue1 == [] ?
    [result1, residue1]                               // S -> V used
  : first(residue1) == '+' ?
    [result2, residue2] = S(rest(residue1)),           // try S -> V + S
    failed(result2) ?
      [FAILURE, residue2]                        // V +, but S failed
    : [mkTree('+', result1, result2), residue2]      // S -> V + S used
  : [result1, residue1];                                  // no more options
```

A rex version of parsing (2)

```plaintext
// parse function for auxiliary V, rules V -> a | b | c
V(chars) =>
  isVar(char) ? [mkTree(char), chars]                   // variable
  : [FAILURE, chars];                                   // not a variable

// auxiliary functions
FAILURE = "failure";
VARS = ['a', 'b', 'c'];
isVar(char) = member(char, VARS);
failed(result) = result == FAILURE;
mkTree(Var) = Var;
mkTree(Op, Tree1, Tree2) = [Op, Tree1, Tree2];
parse(string) = A(explode(string));
```

Test Cases

- `test(parse("a+b+c"), ["\[\'+', 'a', \[\'+', 'b', 'c'\]\], []]);`
- `test(parse("a+b+c+a"), ["\[\'+', 'a', \[\'+', 'b', \[\'+', 'c', 'a'\]\]\], []]);`
- `test(parse("ab+c"), ["a", \[\['b', '+', 'c'\]\]]);`
- `test(parse("a+b+"), [FAILURE, [\[\'
```

Operators + and * with * having higher precedence

- Rules:
  - $S \rightarrow P + S | P$
  - $P \rightarrow V * P | V$
  - $V \rightarrow a | b | c$
- Note that * is analogous to +.
  - $S$ is to $P$ and + as $P$ is to $V$ and *
- Therefore the same rule pattern applies to both.
Parsing Methods in Java

- In the Java version, we will "not need to" return the unparsed input as a value.
- We can side-effect the input stream to achieve a similar result, "using up" characters as we go.

Additive Grammar

\[
A \rightarrow V \mid V + A \\
V \rightarrow a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z
\]

Corresponding to the grammar above, there will be two parse methods:
- `A()`  
- `V()`

Each parses from the current point in the input.

Runnable Examples

- `parse/addRecursiveIVE/Additive.java`
- `parse/add/Additive.java`
- `parse/addMul/AddMul.java`
- `parse/simpleCalc/SimpleCalc.java`
V() method

```java
/** PARSE METHOD for V \(\rightarrow a|b|c|d|e|f|g|h|i|j|k|l| m|n|o|p
|q|r|s|t|u|v|w|x|y|z\)
*/
Object V()
{
    skipWhitespace();
    if( isVar(peek()) )
    {
        return makeString (nextChar ());
    }
    return failure;
}
```

makeString(char c)

```java
/** make a String from a char */
static String makeString (char c)
{
    return ( new StringBuffer(1).append(c)) .toString ();
}
```

isVar()

```java
/** predicate defining whether its argument is a variable */
boolean isVar(char c)
{
    switch( c )
    {
        case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g':
        case 'h': case 'i': case 'j': case 'k': case 'l': case 'm': case 'n':
        case 'o': case 'p': case 'q': case 'r': case 's': case 't': case 'u':
        case 'v': case 'w': case 'x': case 'y': case 'z':
            return true;
        default:
            return false;
    }
}

Do not use arithmetic on integer codes for this purpose.
```

Recursive A() method

```java
/** PARSE METHOD for A \(\rightarrow V \{ '+' V \}\)
*/
Object A()
{
    Object result;
    Object V1 = V();
    if( isFailure(V1) ) return failure;
    if( skipWhitespace() && nextCharIs('+') )
    {
        Object A2 = A();
        if( isFailure(A2) ) return failure;
        return Polylist.list('+', V1, A2);
    }
    else
    {
        return V1;
    }
}
```

Replacing some Recursion with Iteration

"Inverse McCarthy Transformation" for Grammars with left-grouping

- Recursion → Iteration
- Works in some cases, not all
- Use for convenience and readability

Both forms are "left grouping" in this example
**A() method, iterative version**

```java
/** PARSE METHOD for A -> V { '+' V } **/
Object A()
{
Object result;
Object V1 = V();
if( isFailure(V1) ) return failure;
result = V1;
while( skipWhiteSpace() && nextCharIs('+') )
{
Object V2 = V();
if( isFailure(V2) ) return failure;
result = Polylist.list('+', result, V2);
}
return result;
}
```

**The Additive/Multiplicative Grammar**

<table>
<thead>
<tr>
<th>Additive</th>
<th>Multiplicative</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow V { '+' V }$</td>
<td>$P \rightarrow V { '*' V }$</td>
</tr>
<tr>
<td>$V \rightarrow a</td>
<td>b</td>
</tr>
</tbody>
</table>

Remembering Precedence Rules

- Tighter-binding operators are introduce further away from the root of the grammar:

  - $+ \rightarrow A \rightarrow P \{ '+' P \}$
  - $* \rightarrow P \rightarrow V \{ '*' V \}$

  * binds more tightly than +

**Syntax Tree Applet**

- **Excercise: Include ^ (power)**

  - ^ binds the most tightly
  - * is next
  - + is the weakest

- Parentheses means "handle inside as a single unit"

  - Parallel level to a single variable
  - Sometimes called "primaries"
**Example: SimpleCalc**

- Parses numeric expressions with +, *, ()
- Computes the numeric answer
- Same grammar as SyntaxTree applet

```java
/**
 * SimpleCalc Parse method for A -> P { '+' P }
 */
Object A()
{
  Object result = P();                        // get first addend
  if (isFailure(result)) return failure;
  while (skipWhitespace() && nextCharIs('+'))
  {
    Object P2 = P();                          // get next addend
    if (isFailure(P2)) return failure;
    try
    {
      result = Arith.add(result, P2);        // accumulate result
    }
    catch (IllegalArgumentException e)
    {
      System.err.println("error: IllegalArgumentException caught");
    }
  }
  return result;
}
```

**Grammar for Unicalc**

- Example
  - 3.5 meters^2 / (watt hour)
- Operators
  - ^
  - /
  - juxtaposition (implied multiplication)
- Units (meter, second, etc.)
- Numbers (floating point allowed: 1.23e-45)
- Parentheses

**Result of Parsing Unicalc**

- A Unicalc quantity: polylist of 3 components:
  - numeric multiplier
  - numerator
  - denominator
- The parser may perform some "algebra":
  - ^ gets converted to multiplication
  - / and juxtaposition use Unicalc divide and multiply