Inductive Definitions
Languages
Grammars
Parsing

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

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

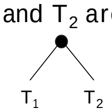
- Elements of an inductive definition of a set S.
  - Basis
  - Induction rule(s)
  - Extremal clause

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 numer of times.

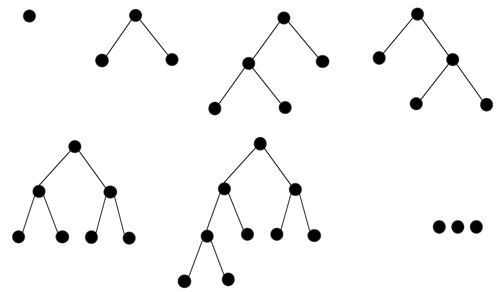
Example of ID: Binary Trees

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



- 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 ω

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

Notes

- ω is an infinite set.
- ω’s members are all finite.
- ω does *not* contain infinity (∞) 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}, **or**
      - 0 is { }, the empty set; X’ is the set X $\cup$ {X}.
    - In the second example: 0 is {}, 0’ is {{}}, 0’’ is {{}, {{}}}, 0’’’ is {{}, {{}}, {{}, {{}}}}, …
    - Advantage: $0^{n\text{ 's}}$ 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 λ so it is readable) stands for 0.
- 1X (1 followed by X) stands for X’.
- The numerals are:
  λ, 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 λ so it is visible) stands for 0.
- The numerals are:
  λ, 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.
- You know the rest.

Numerals vs. Numbers

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

Strings over an alphabet Σ

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

Strings over an alphabet Σ

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

Languages

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

More Languages

- Examples, where $\Sigma = \{a, b\}$
  - $\{\lambda$, ab, ba, aabb, abab, baab, bbaa, aaabbb, aababb, …} 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, …} the language in which the number of a's is *not* equal to the number of b's.
  - {ab, abab, aabb, aababb, …} 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, …}
- 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:
  - $S \Rightarrow ab$
  - $S \Rightarrow aSb \Rightarrow aabb$
  - $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$
  - $S \Rightarrow SS \Rightarrow abS \Rightarrow abab$
  - $S \Rightarrow SS \Rightarrow SSS \Rightarrow ababab$
  - $S \Rightarrow SS \Rightarrow aSbS \Rightarrow aabbS \Rightarrow aabbaSb \Rightarrow aabbaabb$

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

Example Derivations

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

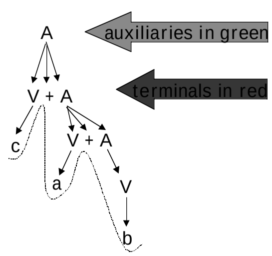
Shorthands on top of Shorthands

- The productions are:
  - $A \rightarrow V$
  - $A \rightarrow V + A$
  - $V \rightarrow a$
  - $V \rightarrow b$
  - $V \rightarrow c$
- 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

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

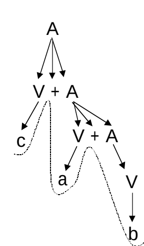
arrows indicate that a production is being applied



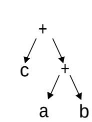
Terminal string = red "fringe" of tree = "c + a + b"

Syntax Tree (!= Derivation Tree) Shows Implied Meaning of String

Derivation Tree



Syntax Tree



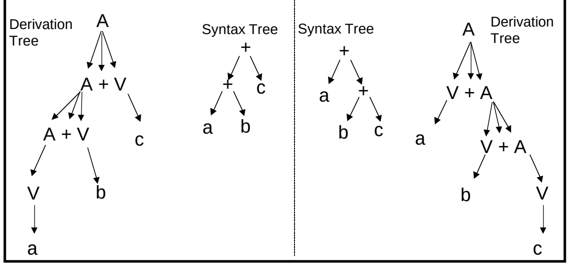
Right Grouping (used so far) vs. Left Grouping Productions

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

Left-grouping production

Right-grouping production

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



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)
  - On computers, for floating point addition, associativity does not always hold.

Floating Point is Not Associative

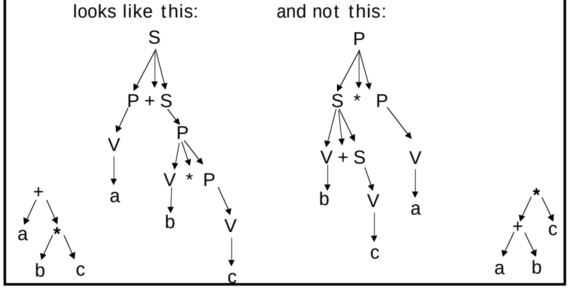
- Try this:
  - `sumup(m, n)   = m > n ? 0 : 1./m + sumup(m+1, n);`
  - `sumdown(m, n) = m > n ? 0 : 1./n + sumdown(m, n-1);`
  - `test(n) = sumup(1, n) == sumdown(1, n);`
  - `map(test, range(1, 100));`
  - ```
    [1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1,
     1, 1, 1, 1, 1, 0, 0,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

Precedence Issue
(multiple operator symbols)

- How do we ensure that the syntax tree of
  a + b*c
  looks like this: and not this:

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:
  - $S \rightarrow P + S \ | \ P$ S is a "sum"
  - $P \rightarrow V * P \ | \ V$ P is a "product"
  - $V \rightarrow a \ | \ b \ | \ c$ 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:
  - $S \rightarrow P + S \ | \ P$
  - $P \rightarrow V * P \ | \ V$
  - $V \rightarrow a \ | \ b \ | \ c$
- Sample derivations (S is the syntactic category):
  - $S \Rightarrow P \Rightarrow V \Rightarrow a$
  - $S \Rightarrow P + S \Rightarrow V + S \Rightarrow a + S \Rightarrow a + P \Rightarrow a + V \Rightarrow a + b$
  - $S \Rightarrow P + S \Rightarrow V + S \Rightarrow a + S \Rightarrow a + P \Rightarrow a + V{*}P \Rightarrow a + b * P \Rightarrow a + b * c$
  - $S \Rightarrow P + S \Rightarrow V{*}P + S \Rightarrow a{*}P + S \Rightarrow a{*}V + S \Rightarrow a{*}b + S \Rightarrow a{*}b + P \Rightarrow a{*}b + V \Rightarrow a{*}b + c$

Example Syntactic Categories

- The productions are:
  - $S \rightarrow P + S \ | \ P$
  - $P \rightarrow V * P \ | \ V$
  - $V \rightarrow a \ | \ b \ | \ c$
- Sample sub-derivations:
  - **Derivations from P**:
    - $P \Rightarrow V \Rightarrow a$
    - $P \Rightarrow V{*}P \Rightarrow a{*}P \Rightarrow a{*}V \Rightarrow a{*}b$
    - $P \Rightarrow V{*}P \Rightarrow a{*}P \Rightarrow a{*}V{*}P \Rightarrow a{*}b{*}P \Rightarrow a{*}b{*}V \Rightarrow a{*}b{*}a$
  - Observation: Derivations from P don't include any +'s.

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 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:

$$\underbrace{a\ +}_{\text{passed}}\ \underbrace{b\ *\ c}_{\text{remaining}}$$

Example

- Consider the grammar with start symbol S:
  - S → V + S | V
  - V → 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 those 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

S → V + S | V
V → a | b | c

- 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

$S \rightarrow V + S \mid V$
$V \rightarrow a \mid b \mid c$

- Suppose the input string is "a b + c".
- 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 does not find it; therefore S_1 returns success, with "b + c".
- **Since the top-level call to S_1 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)

```
// 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)

```
// parse function for auxiliary V, rules V -> a | b | c

V([]) => [FAILURE, []];                                        // no input

V([c | chars]) => isVar(c) ? [mkTree(c), chars];               // variable

V([c | chars]) => [FAILURE, [c | 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, []]);

test(parse("+a"),           [FAILURE, ['+', 'a']]);
```

Operators + and * with * having higher precedence

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

rex parsing for +, * (S)

```
// function for auxiliary S, rules S -> V | V + S

S(input) =>
    [result1, residue1] = P(input),                        // try P

    failed(result1) ?
        [FAILURE, residue1]                                // P failed

  : residue1 == [] ?
        [result1, residue1]                                // S -> P used

  : first(residue1) == '+' ?

      ([result2, residue2] = S(rest(residue1)),            // try S -> P + S

           failed(result2) ?
               [FAILURE, residue2]                         // P +, but S failed

         : [mkTree('+', result1, result2), residue2]       // S -> P + S used
      )

  : [result1, residue1];                                   // no more options
```

rex parsing for +, * (P)

```
// function for auxiliary P, rules P -> V | V + P

P(input) =>
    [result1, residue1] = V(input),                        // try V

    failed(result1) ?
        [FAILURE, residue1]                                // V failed

  : residue1 == [] ?
        [result1, residue1]                                // P -> V used

  : first(residue1) == '*' ?

      ([result2, residue2] = P(rest(residue1)),            // try P -> V + P

           failed(result2) ?
               [FAILURE, residue2]                         // V +, but P failed

         : [mkTree('*', result1, result2), residue2]       // P -> V + P used
      )

  : [result1, residue1];                                   // no more options
```

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.

Parsing Methods in Java

```
/**
 * ParseFromString is the base class for parsing from a String,
 * such as a single input line.
 */

class ParseFromString
{
ParseFromString(String input) // constructor

char nextChar()

boolean nextCharIs(char c)

char peek()

boolean skipWhitespace()
}
```

various utility methods

Additive Grammar

A → V | V + A

V → 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

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/addRecursive/Additive.java

parse/add/Additive.java

parse/addMult/AddMult.java

parse/simpleCalc/SimpleCalc.java

V() method

```
/**
 * PARSE METHOD for V → 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)

```
/**
 * make a String from a char
 */

static String makeString(char c)
  {
  return (new StringBuffer(1).append(c)).toString();
  }
```

isVar()

```
/**
 * 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

```
/**
 * PARSE METHOD for A -> 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

{ } is a meta-symbol meaning **"0 or more of what's inside"**

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

Recursive Form

A → V | A + V
V → a | b | c

Iterative Form

A → V **{** + V **}**
V → a | b | c

both forms are "left grouping" in this example

A() method, iterative version

```
/** 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

Additive

```
A -> V { '+' V }

V -> 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
```

Additive and Multiplicative

```
A -> P  { '+' P }
P -> V  { '*' V }
V -> 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
```

Construct methods by analogy.

Remembering Precedence Rules

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

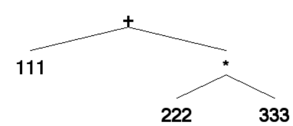
root → $A \rightarrow P$ { '+' P }

$P \rightarrow V$ { '*' V } ← * binds more tightly than +

Syntax Tree Applet

Input numeric expression for syntax analysis:

111 + 222 * 333



http://www.cs.hmc.edu/courses/current/examples/java/parse/syntaxTree/SyntaxTree.html

Excercise: 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"

Example: SimpleCalc

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

```
/**
 * 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