Inductive Definitions
Languages,
Grammars,
Parsing
Motivation: Parsing

- Parsing is the act of turning text into meaningful information.

- Example:
  - **Programming language**: Parsing makes the language into an executable machine language 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

- 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

- is a binary tree.

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

  ![Binary Tree Diagram]

- 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‴, 0⁴″, ... 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.
  
  Two variations:
  - $0$ 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 $\{\} \cup \{\}$, $0''$ is $\{\}$, $\{\} \cup \{\}$, $\{\} \cup \{\} \cup \{\}$, ... 

  Advantage: $0^n$'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, ...
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 \( \lambda \) so it is readable) stands for 0.
- \( 1X \) (1 followed by X) stands for \( X' \).
- The numerals are:
  \[ \lambda, 1, 11, 111, 1111, 1111, \ldots \]

- 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, ... 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 $\Sigma$

- 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 \( \sigma x \) (\( \sigma \) followed by \( x \)) is in \( \Sigma^* \).

- **Extremal clause.**
Languages

- A language over $\Sigma$ is any subset of $\Sigma^*$.

- Examples, where $\Sigma = \{a, b\}$
  - $\{a, b\}^*$ itself
  - $\{\} \quad$ the empty language
  - $\{ba, baba\} \quad$ maybe your first language
  - $\{\lambda, aa, aaaa, aaaa, aaaaaa, \ldots\} \quad$ the language of an even number of $a$'s.
More Languages

- **Examples, where** $\Sigma = \{a, b\}$
  - $\{\lambda, ab, ba, aabb, abab, baab, bbba, aaabbb, aababbb, \ldots\}$ 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.
  - $\{ab, abab, aabb, aababbb, \ldots\}$ 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.

- The idea comes from linguistics and early work on computer languages.
Grammatical Definition

- There is a “start symbol”, or “root”, say S, not in the alphabet of the language itself.
- → is a symbol meaning “can be rewritten as”.
- Grammar rules:
  - $S \rightarrow ab$
  - $S \rightarrow aSb$
  - $S \rightarrow SS$
- Application of rules is 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:
  ∗ S → ab
  ∗ S → aSb
  ∗ S → SS

∗ Example derivations of strings in the language:
  ∗ S ⇒ ab
  ∗ S ⇒ aSb ⇒ aabb
  ∗ S ⇒ aSb ⇒ aaSbb ⇒ aaabbb
  ∗ S ⇒ SS ⇒ abS ⇒ abab
  ∗ S ⇒ SS ⇒ SSS ⇒ ababab
  ∗ S ⇒ SS ⇒ aSbS ⇒ aabSb ⇒ aabbaabbb
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:
  - $A \Rightarrow V \Rightarrow a$
  - $A \Rightarrow V \Rightarrow c$
  - $A \Rightarrow V + A \Rightarrow c + A \Rightarrow c + V \Rightarrow c + a$
  - $A \Rightarrow V + A \Rightarrow c + A \Rightarrow c + V + A \Rightarrow c + b + A \Rightarrow c + b + V \Rightarrow c + b + a$
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:

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

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

Same grammar, just a briefer notation.
Derivation Tree Visualization

\[ A \rightarrow V \mid V + A \]
\[ V \rightarrow a \mid b \mid c \]

Terminals in red
Auxiliaries in green

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

A → A + V | V
V → a | b | c

Left-grouping production

A + V
A + V
V
V
a
b
c

Derivation Tree

Syntax Tree

Right-grouping production

A → V + A | V
V → a | b | c

A + V
V + A
V + A
V + A
b
V
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.
    \[ (a - b) - c \neq a - (b - c) \]
  - On computers, for floating point addition, associativity does not always hold.
Floating Point is Not Associative

Try this:

- \( \text{sumup}(m, n) = \begin{cases} 0 : & m > n \\ 1./m + \text{sumup}(m+1, n) : & \text{otherwise} \end{cases} \)
- \( \text{sumdown}(m, n) = \begin{cases} 0 : & m > n \\ 1./n + \text{sumdown}(m, n-1) : & \text{otherwise} \end{cases} \)
- \( \text{test}(n) = \text{sumup}(1, n) == \text{sumdown}(1, n) \)
- \( \text{map}(\text{test}, \text{range}(1, 100)) \)

- [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

- Grouping sensitivity is due to round-off error.
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:

\[
\begin{align*}
&\text{(loose)} \\
&\text{(tight)}
\end{align*}
\]
Example:
Grammar for Additive & Multiplicative
Arithmetic Expressions

- The root is \( A \).
- The terminals are \{a, b, c, +, \*\}.
- The productions are:
  - \( A \rightarrow M + A \mid M \)
  - \( M \rightarrow V \* M \mid V \)
  - \( V \rightarrow a \mid b \mid c \)
- 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:**
- \( A \rightarrow M + A \mid M \)  
  \( S \) is a “sum”
- \( M \rightarrow V \ast M \mid V \)  
  \( P \) is a “product”
- \( V \rightarrow a \mid b \mid c \)  
  \( V \) is a “variable”
Example Derivations

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

- Sample derivations ($A$ is the syntactic category):
  - $A \Rightarrow M \Rightarrow V \Rightarrow a$
  - $A \Rightarrow M + A \Rightarrow V + A \Rightarrow a + A \Rightarrow a + M \Rightarrow a + V \Rightarrow a + b$
  - $A \Rightarrow M + A \Rightarrow V \cdot M + A \Rightarrow a \cdot M + A \Rightarrow a \cdot V \Rightarrow a \cdot b \cdot c$
  - $A \Rightarrow M + A \Rightarrow V \cdot M + A \Rightarrow a \cdot M + A \Rightarrow a \cdot V + A \Rightarrow a \cdot b + A \Rightarrow a \cdot b + M \Rightarrow a \cdot b + V \Rightarrow a \cdot b + c$
Example Syntactic Categories

- The productions are:
  - \( A \rightarrow M + A \mid M \)
  - \( M \rightarrow V \ast M \mid V \)
  - \( V \rightarrow a \mid b \mid c \)

- Sample sub-derivations:
  - **Derivations from \( M \):**
    - \( M \Rightarrow V \Rightarrow a \)
    - \( M \Rightarrow V \ast M \Rightarrow a \ast M \Rightarrow a \ast V \Rightarrow a \ast b \)
    - \( M \Rightarrow V \ast M \Rightarrow a \ast M \Rightarrow a \ast V \ast M \Rightarrow a \ast b \ast M \Rightarrow a \ast b \ast V \Rightarrow a \ast b \ast a \)
  - **Observation:** Derivations from \( M \) don’t 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”
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:

\[
\underbrace{a + b \ast c}_{\text{passed}} \underbrace{\text{remaining}}_{\text{remaining}}
\]
Example

- Consider the grammar with start symbol $S$:
  - $S \rightarrow V + S \mid V$
  - $V \rightarrow a \mid b \mid c$
- 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

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

- 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

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 success: the Syntax Tree
    - for failure: FAILURE (some special value, not a syntax tree)
  - the unparsed input.
A rex version of parsing (1)

// parse function for auxiliary A, rules A -> V | V + A

A(input) =
    Vresult = V(input),                            // try for V
    [tree1, residue1] = Vresult,                  // use A -> V
    residue1 == [] ? Vresult                      // failure
    : failed(tree1) ? Vresult                     // failure
    : first(residue1) == '+' ?

    ( [tree2, residue2] = A(rest(residue1)),      // try A -> V + A
      failed(tree2) ?
      Vresult                                    // use A -> V only
      : [mkTree('+', tree1, tree2), residue2]     // use A -> V + A
    )

    : Vresult;                                    // use A -> V
```javascript
// Test cases

test(A(explode("a")), ['a', []]);
test(A(explode("a+b")), [['+', 'a', 'b'], []]);
test(A(explode("a+b+c")), [['+', 'a', ['+', 'b', 'c']], []]);
test(A(explode("a+b+c+a")), [['+', 'a', ['+', 'b', 'c']], ['+'], [ ]]);
test(A(explode("")), [FAILURE, []]);
test(A(explode("+")), [FAILURE, ['+'], [ ]]);
test(A(explode("ab")), ['a', ['b']]);
test(A(explode("a+b+")), [['+', 'a', 'b'], ['+'], [ ]]);
test(A(explode("a+b+c+")), [['+', 'a', ['+', 'b', 'c']], ['+'], [ ]]);
test(A(explode("ab+c")), ['a', ['b', '+', 'c']]);
test(A(explode("a+b+")), [['+', 'a', 'b'], ['+'], [ ]]);
```
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));
Operators + and *
with * having higher precedence

Rules:
- \( A \rightarrow M + A \mid M \)
- \( M \rightarrow V * M \mid V \)
- \( V \rightarrow a \mid b \mid c \)

Note that * is analogous to +.
- \( A \) is to \( M \) and + as
  - \( M \) is to \( V \) and *

Therefore the same rule pattern applies to both.
A(input) =
    Mresult = M(input),    // try for M
    [tree1, residue1] = Mresult,
    residue1 == [] ? Mresult    // use A -> M

    : failed(tree1) ? Mresult    // failure

    : first(residue1) == '+' ?

        ( [tree2, residue2] = A(rest(residue1)),    // try A -> M + A
           failed(tree2) ?
             Mresult    // use A -> M only
               : [mkTree('+', tree1, tree2), residue2]    // use A -> M + A

    )

    : Mresult;    // use A -> M
rex parsing for +, * (M)

\[
M(\text{input}) = \\
\quad V\text{result} = V(\text{input}), \\
\quad [\text{tree1}, \text{residue1}] = V\text{result}, \\
\quad \text{residue1} == [] ? V\text{result} \\
\quad : \text{failed}(\text{tree1}) ? V\text{result} \\
\quad : \text{first}(\text{residue1}) == '*' ? \\
\quad \\
\quad \quad ( [\text{tree2}, \text{residue2}] = M(\text{rest}(\text{residue1})), \\
\quad \quad \quad \text{failed}(\text{tree2}) ? \\
\quad \quad \quad \quad V\text{result} \\
\quad \quad \quad : [\text{mkTree}('\ast', \text{tree1}, \text{tree2}), \text{residue2}] ) \\
\quad : V\text{result}; \\
\quad : V\text{result};
\]

// try for V
// use M -> V
// failure
// try M -> V * M
// use M -> V only
// use M -> V + M
// use M -> V
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.

We can store the input stream in the parse object, rather than pass it as an argument.
/**
* 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()
}
Additive Grammar

\[ A \rightarrow V \mid V + A \]

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

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
/**
 * PARSE METHOD for V → abcdeffghijklmnopqrstuvwxyz
 */

Object V()
{
    skipWhitespace();

    if( isVar(peek()) )
        {
            return makeString(nextChar());
        }
    return failure;
}
/**
 * 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 OpenList.list('+', V1, A2);
    }
    else
    {
        return V1;
    }
}

Replacing some Recursion with Iteration
“Inverse McCarthy Transformation”
for Grammars with \texttt{left}-grouping

\{ \} is a meta-symbol meaning “0 or more of what’s inside”

- **Recursion \rightarrow Iteration**
- **Works in some cases, not all**
- **Use for convenience and readability**

\begin{align*}
\text{Recursive Form} & \quad A \rightarrow V \mid A + V \\
\quad & \quad V \rightarrow a \mid b \mid c
\end{align*}

\begin{align*}
\text{Iterative Form} & \quad A \rightarrow V \{ + V \} \\
\quad & \quad V \rightarrow a \mid b \mid c
\end{align*}

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

\[ A \rightarrow P \{ \text{`+` } P \} \]
\[ P \rightarrow V \{ \text{`*` } V \} \]

* 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
Example: SimpleCalc

- Parses numeric expressions with +, *, ()
- Computes the numeric answer
- Same grammar as SyntaxTree applet
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:
  Object with 3 components:
    - numeric multiplier
    - numerator
    - denominator

- The parser may perform some “algebra“:
  - ^ gets converted to multiplication
  - / and juxtaposition use Unicalc divide and multiply