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.

Inductive Definitions

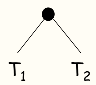
- Elements of an inductive definition of a set S.
  - Basis
  - Induction rule(s)
  - Extremal clause

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.

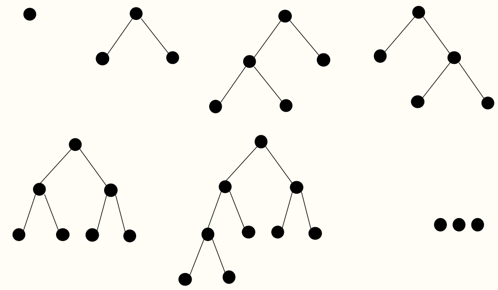
Example of ID: Binary Trees

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



- 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‴, 0⁗, … are all elements of □.

Notes

- Set □ is an **infinite** set.
- The members of □ are all **finite**.
- Set □ 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.
    - 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: 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 x0 (concatenation) is x1, the successor of x1 is x2, …, and the successor of x8 is x9.
- The successor of x9 is y0 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:
  □, 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.

Strings over an alphabet □

- The set of *all* finite strings over an alphabet □ is denoted □*.
- Example:
  - {a, b}* =
    {□, a, b, aa, ab, ba, bb, aaa, aab, aba, …}

Strings over an alphabet □

- Basis: □, the empty string, is in □*.
- Inductive rule: If x □ □* and □ □ □, then □ x (□ followed by x) is in □*.
- Extremal clause.

Languages

- A language over □ is any subset of □*, the set of all strings over □.
- Examples, where □ = {a, b}:
  - {a, b}* itself
  - { }, the empty language
  - {ba, baba}, maybe your first language
  - {□, aa, aaaa, aaaa, aaaaaa, …} the language of an even number of a's.

More Languages

- More Examples, where □ = {a, b}
  - {□, 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.
- 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.
- → is a symbol meaning "can be rewritten as".
- Grammar rules, for example:
  1. S → ab
  2. S → aSb
  3. S → 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 → ab
  2. S → aSb
  3. S → SS
- Example derivations of strings in the language:
  1. S → ab
  2. S → aSb → aabb
  3. S → aSb → aaSbb → aaabbb
  4. S → SS → abS → abab
  5. S → SS → SSS → ababab
  6. S → SS → aSbS → aabbS → aabbaSb → 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 → V
  - A → V + A
  - V → a
  - V → b
  - V → c

Example Derivations

- The productions are:
  - A → V
  - A → V + A
  - V → a
  - V → b
  - V → c
- Sample derivations:
  1. A → V → a
  2. A → V → c
  3. A → V + A → c + A → c + V → c + a
  4. A → V + A → c + A → c + V + A → c + b + A → c + b + V → c + b + a

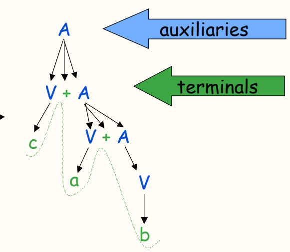
Shorthands on top of Shorthands

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

Derivation Tree Visualization

A → V | V + A
V → a | b | c

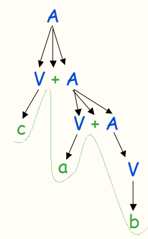
arrows indicate that a production is being applied



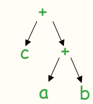
Terminal string = red "fringe" of tree = "c + a + b"

Syntax Tree (!= Derivation Tree) Shows Implied "Interpretation" of String

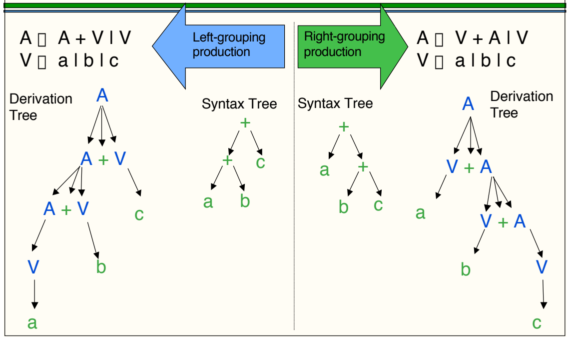
Derivation Tree



Syntax Tree



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



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

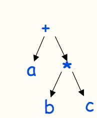
- Try this:
  - `sumup(m, n)   = m > n ? 0 : 1./m + sumup(m+1, n);`
  - `sumdown(m, n) = m > n ? 0 : 1./n + sumdown(m, n-1);`
  - `test(n) = sumup(1, n) == sumdown(1, n);`
  - `map(test, range(1, 100));`
  -
```
[1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1,
 1, 1, 1, 1, 1, 0, 0, 0, 0,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

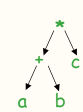
Precedence Issue (multiple operator symbols)

- How do we ensure that the syntax tree of

  a + b*c

  looks like this: and not this:



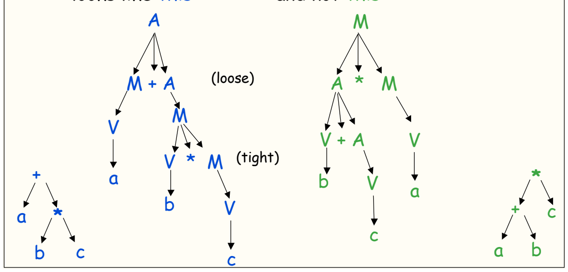


Multiple Auxiliaries

- 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:



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 + A → a + A → a + M → a + V*M → a + b * M → a + b * c
  4. A → M + A → V*M + A → a *M + A → a *V + A → a *b + A → a *b + M → a *b + V → a *b + c

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 → a
  2. M → V*M → a*M → a*V → a*b
  3. M → V*M → a*M → a*V*M → a*b*M → a*b*V → 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"