Recursive definition of sets and structural induction

Instructor: Marc Moreno Maza
Tower of Hanoi game

• Rules of the game:
  – Start with all disks on the first peg.
  – At any step, can move a disk to another peg, as long as it is not placed on top of a smaller disk.
  – Goal: move the whole tower onto the second peg.

• Question: how many steps are needed to move the tower of 8 disks? How about n disks?
Tower of Hanoi game

• Rules of the game:
  – Start with all disks on the first peg.
  – At any step, can move a disk to another peg, as long as it is not placed on top of a smaller disk.
  – Goal: move the whole tower onto the second peg.
• Question: how many steps are needed to move the tower of 8 disks? How about n disks?

• Let us call the number of moves needed to transfer n disks $H(n)$.
  – Names of pegs do not matter: from any peg $i$ to any peg $j \neq i$ would take the same number of steps.
• Basis: only one disk can be transferred in one step.
  – So $H(1) = 1$
• Recursive step:
  – suppose we have $n-1$ disks. To transfer them all to peg 2, need $H(n-1)$ number of steps.
  – To transfer the remaining disk to peg 3, 1 step.
  – To transfer $n-1$ disks from peg 2 to peg 3 need $H(n-1)$ steps again.
  – So $H(n) = 2H(n-1) + 1$ (recurrence).
• Closed form: $H(n) = 2^n - 1$. 
Recurrence relations

- **Recurrence**: an equation that defines an \( n^{th} \) element in a sequence in terms of one or more of previous terms.
  - \( H(n) = 2H(n-1)+1 \)
  - \( F(n) = F(n-1)+F(n-2) \)
  - \( T(n) = aT(n-1) \)

- **A closed form** of a recurrence relation is an expression that defines an \( n^{th} \) element in a sequence in terms of \( n \) directly.
  - Often use recurrence relations and their closed forms to describe performance of (especially recursive) algorithms.
Recursive definitions of sets

• So far, we talked about recursive definitions of sequences. We can also give recursive definitions of sets.
  – E.g: recursive definition of a set \( S = \{0, 1\}^* \)
    • **Basis:** empty string is in \( S \).
    • **Recursive step:** if \( w \in S \), then \( w0 \in S \) and \( w1 \in S \)
      – Here, \( w0 \) means string \( w \) with 0 appended at the end; same for \( w1 \)
Recursive definitions of sets

• Recursive definition of a set \( S = \{0, 1\}^* \)
  – Alternatively:
    • **Basis**: empty string, 0 and 1 are in \( S \).
    • **Recursive step**: if \( s \) and \( t \) are in \( S \), then \( st \in S \)
      – here, \( st \) is concatenation: symbols of \( s \) followed by symbols of \( t \)
      – If \( s = 101 \) and \( t = 0011 \), then \( st = 1010011 \)
  – Additionally, need a **restriction condition**: the set \( S \) contains only elements produced from basis using recursive step rule.
Trees

- In computer science, a **tree** is an undirected graph without cycles
  - **Undirected**: all edges go both ways, no arrows.
  - **Cycle**: sequence of edges going back to the same point.
Trees

• **Recursive definition of trees:**
  
  – **Base:** A single vertex $v$ is a tree.

  – **Recursion:**
    
    • Let $T$ be a tree, and $v$ a new vertex.
    
    • Then a new tree consist of $T$, $v$, and an edge (connection) between some vertex of $T$ and $v$.

  – **Restriction:**
    
    • Anything that cannot be constructed with this rule from this base is not a tree.
Arithmetic expressions

• Suppose you are writing a piece of code that takes an arithmetic expression and, say evaluates it.
  – “5*3-1”, “40-(x+1)*7”, etc

• How to describe a valid arithmetic expression?
Arithmetic expressions

• *How to describe a valid arithmetic expression?*
• Define a set of all valid arithmetic expressions *recursively.*
  
  – **Base:** A number or a variable is a valid arithmetic expression.  
    • 5, 100, x, a
  
  – **Recursion:**  
    • If A and B are valid arithmetic expressions, then so are \((A), A + B, A - B, A * B, A / B.\)
      
      – Constructing 40-(x+1)*7: first construct 40, x, 1, 7. Then (x+1). Then (x+1)*7, finally 40-(x+1)*7
      
      – Caveat: how do we know the order of evaluation? On that later.
  
  – **Restriction:** nothing else is a valid arithmetic expression.
Formulas

• What is a well-formed propositional logic formula?
  
  \[(p \lor \neg q) \land r \rightarrow (\neg p \rightarrow r)\]

  – **Base**: a propositional variable \(p, q, r \ldots\)
    • Or a constant \(TRUE, FALSE\)

  – **Recursion**:  
    • If \(F\) and \(G\) are propositional formulas, so are \((F), \neg F, F \land G, F \lor G, F \rightarrow G, F \leftrightarrow G.\)

  – And nothing else.
Formulas

• What is a well-formed predicate logic formula?
  – $\exists x \in D \ \forall y \in \mathbb{Z} \ P((x, y) \lor Q(x, z)) \land x = y$
  – **Base**: a predicate with free variables
    • $P(x), \ x=y, \ ...$
  – **Recursion**:
    • If $F$ and $G$ are predicate logic formulas, so are $(F)$, $\neg F$, $F \land G$, $F \lor G$, $F \rightarrow G$, $F \leftrightarrow G$
    • If $F$ is a predicate logic formula with a free variable $x$, then $\exists x \in D \ F$ and $\forall x \in D \ F$ are predicate logic formulas.
  – **And nothing else**.
    • So $\exists x \in \text{People} \ Likes(x, y \land x)$, $Likes(y \neq x)$ is not a well-formed predicate logic formula!
Grammars

• A context-free grammar consists of
  – A set $V$ of variables (using capital letters)
    • Including a start variable $S$.
  – A set $\Sigma$ of terminals (disjoint from $V$; alphabet)
  – A set $R$ of rules, where each rule consists of a variable from $V$ and a string of variables and terminals.
    • If $A \rightarrow w$ is a rule, we say variable $A$ yields string $w$.
      – This is not the same “$\rightarrow$” as implication, a different use of the same symbol.
    • We use shortcut “$|$” when the same variable might yield several possible strings: $A \rightarrow w_1 | w_2 | \ldots | w_k$
    • Can use $A$ again within the rule: Recursion!
      – Different occurrences of the same variable can be interpreted as different strings.
  • When left with just terminals, a string is derived.
Grammars

• A general recursive definition for these is called a grammar.
  – In particular, here we have “context-free” grammars, where symbols have the same meaning wherever they are.

• A language generated by a grammar consists of all strings of terminals that can be derived from the start variable by applying the rules.
  – All strings are derived by repeatedly applying the grammar rules to each variable until there are no variables left (just the terminals).
Examples of grammars

• Example: **language** \{1, 00\} consisting of two strings 1 and 00
  \[ S \rightarrow 1 | 00 \]
  • Variables: S. Terminals: 1 and 00.

• Example: **strings** over \{0, 1\} with all 0s before all 1s.
  \[ S \rightarrow 0S | S1 | \_ \]
  • Variables: S. Terminals: 0 and 1.
Examples of grammars

• Example: **propositional formulas.**

  1. $F \rightarrow F \lor F$
  2. $F \rightarrow F \land F$
  3. $F \rightarrow \neg F$
  4. $F \rightarrow (F)$
  5. $F \rightarrow p \mid q \mid r \mid TRUE \mid FALSE$

• Here, the only variable is $F$ (it is a start variable), and terminals are
  $\lor, \land, \neg, (, ), p, q, r, TRUE, FALSE$

• To obtain $(p \lor \neg q) \land r$, first apply rule 2, then rule 1, then rule 5 to get $p$, then rule 3, then rule 5 to get $q$, then rule 5 to get $r$. 
Examples of grammars

• Example: arithmetic expressions
  
  \[ \begin{align*}
  EXPR & \rightarrow EXPR + EXPR \mid EXPR - EXPR \mid EXPR \times \\
  & \quad \mid EXPR / EXPR \mid (EXPR) \mid NUMBER \mid -NUMBER \\
  NUMBER & \rightarrow 0DIGITS \mid \ldots \mid 9DIGITS \\
  DIGITS & \rightarrow \_ \mid NUMBER
  \end{align*} \]

• Here, \_ stands for empty string.

  Variables: EXPR, NUMBER, DIGITS (S is starting).

  Terminals: +, -, *, /, 0, ..., 9.

• We used separate NUMBER to avoid multiple “-”.

• And separate DIGITS to have an empty string to finish writing a number, but to avoid an empty number.
Encoding order of precedence

• Easier to specify in which order to process parts of the formula.
  – Better grammar for arithmetic expressions (for simplicity, with x,y,z instead of numbers):
    1.  $EXPR \rightarrow EXPR + TERM \mid EXPR - TERM \mid TERM$
    2.  $TERM \rightarrow TERM \times FACTOR \mid TERM / FACTOR \mid FACTOR$
    3.  $FACTOR \rightarrow (EXPR) \mid x \mid y \mid z$
  – Here, variables are EXPR, TERM and FACTOR (with EXPR a starting variable).
  – Now can encode precedence.
    • And put parentheses more sensibly.
Parse trees.

Visualization of derivations:

parse trees.

1. \( EXPR \rightarrow EXPR + TERM \mid EXPR - TERM \mid TERM \)

2. \( TERM \rightarrow TERM \ast \)
   \( FACTOR \mid TERM \div \)
   \( FACTOR \mid FACTOR \)

3. \( FACTOR \rightarrow (EXPR) \mid x \mid y \mid z \)

• String \((x+y)\ast z\)
Parse trees.

• Visualization of derivations: parse trees.
  – Simpler example:
    • $S \rightarrow 0S \mid S1 \mid -$ 
    • String 001
Puzzle

• Do the following two English sentences have the same parse trees?
  
  – Time flies like an arrow.
  
  – Fruit flies like an apple.
Structural induction

• Let \( S \subseteq U \) be a recursively defined set, and \( F(x) \) is a property (of \( x \in U \)).

• Then
  – if all \( x \) in the base of \( S \) have the property,
  – and applying the recursion rules preserves the property,
  – then all elements in \( S \) have the property.
Multiples of 3

• Let’s define a set $S$ of numbers as follows.
  – **Base:** $3 \in S$
  – **Recursion:** if $x, y \in S$, then $x + y \in S$

• **Claim:** all numbers in $S$ are divisible by 3
  – That is, $\forall x \in S \exists z \in \mathbb{N} \ x = 3z$. 
Multiples of 3

• Proof (by structural induction).
  – Base case: 3 is divisible by 3 (y=1).
  – Recursion: Let $x, y \in S$. Then $\exists z, u \in \mathbb{N} \ x = 3z \land y = 3u$.
    • Then $x + y = 3z + 3u = 3(z + u)$.
    • Therefore, $x + y$ is divisible by 3.
  – As there are no other elements in $S$ except for those constructed from 3 by the recursion rule, all elements in $S$ are divisible by 3.
Binary trees

• **Rooted trees** are trees with a special vertex designated as a root.
  
  – Rooted trees are **binary** if every vertex has **at most three edges**: one going towards the root, and two going away from the root. **Full** if every vertex has either 2 or 0 edges going away from the root.
Binary trees

• **Recursive definition of full binary trees:**
  
  – **Base:** A single vertex \( v \) is a full binary tree with that vertex as a root.

  – **Recursion:**
    
    • Let \( T_1, \ T_2 \) be full binary trees with roots \( r_1, r_2 \), respectively. Let \( v \) be a new vertex.
    
    • A new full binary tree with root \( v \) is formed by connecting \( r_1 \) and \( r_2 \) to \( v \).

  – **Restriction:**
    
    • Anything that cannot be constructed with this rule from this base is not a full binary tree.
Height of a full binary tree

- The **height** of a rooted tree, $h(T)$, is the maximum number of edges to get from any vertex to the root.
  - Height of a tree with a single vertex is 0.

- Claim: Let $n(T)$ be the number of vertices in a full binary tree $T$. Then $n(T) \leq 2^{h(T)+1} - 1$
Height of a full binary tree

- Proof (by structural induction)
  - **Base case**: a tree with a single vertex has \( n(T) = 1 \) and \( h(T) = 0 \). So \( 2^{h(T)+1} − 1 = 1 \geq 1 \)
  - **Recursion**: Suppose \( T \) was built by attaching \( T_1, T_2 \) to a new root vertex \( v \).
    - Number of vertices in \( T \) is \( n(T) = n(T_1) + n(T_2) + 1 \)
    - Every vertex in \( T_1 \) or \( T_2 \) now has one extra step to get to the new root in \( T \). So \( h(T) = 1 + \max(h(T_1), h(T_2)) \)
    - By the induction hypothesis, \( n(T_1) \leq 2^{h(T_1)+1} − 1 \) and \( n(T_2) \leq 2^{h(T_2)+1} − 1 \)
    - \( n(T) = n(T_1) + n(T_2) + 1 \)
      \[ \leq 1 + (2^{h(T_1)+1} − 1) + (2^{h(T_2)+1} − 1) \]
      \[ \leq 2 \cdot \max(2^{h(T_1)+1}, 2^{h(T_2)+1}) − 1 \]
      \[ \leq 2 \cdot \max(h(T_1), h(T_2)) + 1 − 1 \]
      \[ = 2 \cdot 2^{h(T)} − 1 = 2^{h(T)+1} − 1 \]
    - Therefore, the number of vertices of any binary tree \( T \) is less than \( 2^{h(T)+1} − 1 \)
Height of a full binary tree

- Claim: Let \( n(T) \) be the number of vertices in a full binary tree \( T \). Then \( n(T) \leq 2^{h(T)+1} - 1 \)

- Alternatively, height of a binary tree is at least \( \log_2 n(T) \)
  - If you have a recursive program that calls itself twice (e.g., within if ... then ... else ...)
  - Then if this code executes \( n \) times (maybe on \( n \) different cases)
  - Then the program runs in time at least \( \log_2 n \), even when cases are checked in parallel.