Unit 7
Recursive definitions

Computer Science 1002

Introduction to Logic for Computer Scientists

1

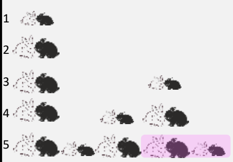
Puzzle: rabbits on an island

- A ship leaves a pair of baby rabbits on an island (with a lot of food).
- After a pair of rabbits reaches 2 months of age, they produce another pair of rabbits, and keep producing a pair every month thereafter.
- Which in turn start reproducing every month when reaching 2 months of age…
  - So every pair starts reproducing at 2 months, and creates a new pair every month from then on.
- How many pairs of rabbits will be on the island in n months, assuming no rabbits die?

2

Fibonacci sequence

0
1
2
3
4
5



- 0^{th} month: 0 pair $F_0 = 0$
- 1^{st} month: 1 pair $F_1 = 1$
- 2^{nd} month: 1 pairs $F_2 = 1$
- 3^{rd} month: 2 pairs $F_3 = 2$
- 4^{th} month: 3 pairs $F_4 = 3$
- 5^{th} month: 5 pairs $F_5 = 5$
- …
- n^{th} month: $F_n = F_{n-1} + F_{n-2}$

To compute how many pairs of rabbits will be there at n^{th} month, add the number of pairs of rabbits at month $n-1$ and the number of pairs of rabbits at month $n-2$

- Number of adult rabbit pairs = number of rabbit pairs one month ago.
  - All rabbits become adults in one month, and have their own babies in two months.
- + Number of baby rabbit pairs = number of rabbit pairs already born two month ago
  - Each of which was old enough to give birth to a pair of baby rabbits.

3

Sequences

- A *sequence* of elements from some set S is a function $f: \mathbb{N} \to S$
  - We usually start natural numbers with 0, but some books start with 1.
  - Elements (terms) of the sequence are written as $a_0, a_1, a_2, \dots, a_n$
    - or using another letter with subscripts such as $F_0, F_1, \dots$ or $s_0, s_1, \dots$, etc.
    - a_0 is called an *initial term* of the sequence.
    - For every i , i^{th} term of a sequence is a_i, where $a_i = f(i)$
  - Fibonacci sequence from the rabbit puzzle:
    - 0,1,1,2,3,5,8,13…
      - Sometimes these are also called Fibonacci numbers
    - $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5,$ etc

4

Arithmetic and geometric progressions

- Arithmetic progression:
  - A sequence of the form $a, a+d, a+2d, a+3d, \dots$ for some numbers a, d. Here, the initial term $= a$
    - Example: $a = 2, d = 3$. Then $s_0 = 2, s_1 = 5, s_2 = 8, s_3 = 11, s_4 = 14 \dots$
- Geometric progression:
  - A sequence of the form $a, ar, ar^2, ar^3, \dots$ for some numbers a, r
    - Here, initial term is also a
    - Example: $a = 2, r = 3$. Then $s_0 = 2, s_1 = 6, s_2 = 18, s_3 = 54, s_4 = 162 \dots$

5

Recurrences

- A sequence is often described by saying how to compute the next element from the previous ones
  - Fibonacci sequence: $F_n = F_{n-1} + F_{n-2}$
- This kind of description, where a_n is expressed as a formula dependent on values of previous elements in the sequence is called a *recurrence*.
  - Sometimes use recurrences to define functions directly, too.
- A *recursive definition* of a sequence consists of a recurrence together with the values of the initial term (sometimes first few terms, called *basis* or *initial condition*).

6

Recurrences

- A *recursive definition* of a sequence consists of a recurrence together with the values of the initial term (sometimes first few terms, called *basis* or *initial condition*).

Fibonacci sequence:

– Basis: $F_0 = 0,\ F_1 = 1$

– Recurrence: $F_n = F_{n-1} + F_{n-2}$

A sequence of powersets:

– Basis: $A_0 = \emptyset$

– Recurrence: $A_{n+1} = \mathcal{P}(A_n)$

Arithmetic progression:

– Basis: $s_0 = a$

– Recurrence: $s_n = s_{n-1} + d$

Geometric progression:

– Basis: $s_0 = a$

– Recurrence: $s_n = s_{n-1} * r$

7

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?

8

9

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?

10

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.
- Let us call the number of moves needed to transfer n disks H(n).
  - 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
- Recursion: to transfer n disks from peg 1 to peg 2.
  - To transfer n-1 disks from peg 1 to peg 3 need $H(n-1)$ steps.
  - To transfer the remaining disk to peg 2, need 1 step.
  - To transfer n-1 disks from peg 3 to peg 2 need H(n-1) steps again.
  - So H(n) = 2H(n-1)+1 (recurrence).

But how many steps does it really take?

11

Closed form of a recurrence

- Tower of Hanoi for $n = 8$
  - $H(8) = 2H(7) + 1 = 2(\,(2H(6) + 1) + 1 = 4H(6) + 2 + 1 = 4H(6) + 3$
    $= 4(2H(5) + 1) + 3 = 8H(5) + 7 = 16H(4) + 15 = 32H(3) + 31$
    $= 64H(2) + 63 = 128H(1) + 127 = 128 * 1 + 127 = 255$
- 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.
  - A closed form of the Tower of Hanoi recurrence is $H(n) = \Sigma_{i=0}^{n-1} 2^i = 2^n - 1$

12

Solving recurrences

- Solving a recurrence: finding a closed form.
  - Solving the recurrence H(n)=2H(n-1)+1

  H(n) $= 2 \cdot H(n-1) + 1 = 2(2H(n-2)+1)+1 = 2^2 H(n-2) + 2 + 1$
  $= 2^3 H(n-3) + 2^2 + 2 + 1 = 2^4 H(n-4) + 2^3 + 2^2 + 2 + 1 \ldots$

- Closed form of the Tower of Hanoi recurrence: $H(n) = \Sigma_{i=0}^{n-1} 2^i = 2^n - 1$
  - Proof by induction (using recursive definition of $H(n)$).
    - Base case: H(1)=1. Induction hypothesis: $H(k) = 2^k - 1$.
    - Induction step: $H(k+1) = 2H(k) + 1 = 2(2^k - 1) + 1 = 2^{k+1} - 1$
  - Or by noticing that a binary number 111…1 (n-1 1s) plus 1 gives a binary number 10000…0 (1 followed by n-1 0s)

13

14

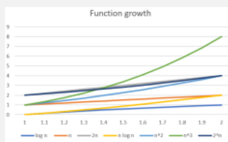
Function growth

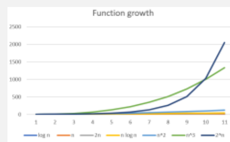
- In Tower of Hanoi, adding one more disk doubles the number of steps. $H(n) = 2^n - 1$.
  - We say that the function H(n) **grows exponentially**
- What does it mean that a function "grows" at a certain rate?
  - Is $f(n) = 100n$ larger than $g(n) = n^2$?
    - Only when $n < 100$. For the remaining infinitely many values of n, $f(n) \leq g(n)$
    - So $g(n)$ grows faster than $f(n)$
  - To compare functions, check which becomes larger as n increases (to infinity).
- Often think of functions as describing running time of an algorithm.
  - For programs, performance on larger inputs matters more.
  - Constant factors don't matter that much.

15

Comparing growth rate of functions

- How to estimate the rate of growth of a function?
  - Plotting a graph?





- Not quite conclusive:
  - How do you know what they will do past the part on the graph?

16

Big-O notation

- We say that $f(n)$ grows at most as fast as $g(n)$ if
  - There is a value n_0 such that after n_0, $f(n)$ is always at most as large as $g(n)$
  - Except if two functions differ by only a constant factor, consider them having the same growth rate.
    - So more correctly, there is a value n_0 and a constant c such that for all n after n_0, $f(n)$ is always at most as large as $c \cdot g(n)$
- Denote set of all functions growing at most as fast as $g(n)$ by $\boldsymbol{O(g(n))}$
  - **Big-Oh** of $g(n)$.
    - Some books say "f is in O(g)", others "f is O(g)", both are OK.
  - g(n) is an **asymptotic upper bound** for f(n).
  - When both $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$, write $f(n) \in \Theta(g(n))$
    - f(n) is in **big-Theta** of g(n)).

17

Big-O notation

- More generally, for real-valued functions $f(x)$ and $g(x)$,

  $$f(x) \in O(g(x)) \text{ iff}$$
  $$\exists x_0 \in \mathbb{R}^{\geq 0}\ \exists c \in \mathbb{R}^{>0}\ \forall x \geq x_0\ \ |f(x)| \leq c \cdot |g(x)|$$

- That is, from some point x_0 on, each $|f(x)|$ is less than |g(x)| (up to a fixed constant factor).
  - When functions describe program running times, they give the number of steps a program takes on an input of size n, making them $\mathbb{N} \to \mathbb{N}$
    - so use n for x and ignore | |.
  - You will see a lot of big-O notation in COMP 2002
    - and possibly some in COMP 1000 and COMP 1001

18

Big-O notation examples

$f(n) \in O(g(n))$ iff
$\exists\, n_0 \in \mathbb{N}\ \exists c \in \mathbb{R}^{>0}\ \forall n \geq n_0\ f(n) \leq c \cdot g(n)$

- $f(n) = n^2, g(n) = 2^n$.
  - Take c=1, $n_0 = 4$.
  - For every $n \geq n_0, f(n) \leq g(n)$
    - Proof by induction.
  - So $\mathrm{n}^2 \in O(2^n)$
- $f(n) = n^2 + 100n, g(n) = 10n^2$.
  - Here, $f(n) \in O(g(n))$ and also $g(n) \in O(f(n))$
    - So $f(n) \in \Theta(g(n))$
    - $f(n) \in O(g(n))$: c = 20 and/or $n_0 = 100$ work.
    - $g(n) \in O(f(n))$: Take c=10, $n_0 = 1$.
  - Can ignore constants and look only at the leading term in a sum.
- $f(n) = n^2, g(n) = 10n$.
  - Can take $c = 1, n_0 = 10$
  - Or take arbitrary c
  - No matter what c is, when $n > c \cdot 10,\ n^2 \geq c \cdot 10n$
  - So $n^2 \notin O(10n)$.

19

Common computational complexity classes

- As we can ignore constants and only consider leading (fastest growing) term in a sum, common classes in Computer Science are:
  - Logarithmic: $O(\log n)$, where $\log n$ is usually $\log_2 n$
    - The base of the log does not matter, as base change multiplies by a constant.
  - Linear: $O(n)$
  - $O(n \log n)$: No established name, but quite common in Computer Science
  - Quadratic: $O(n^2)$
  - Cubic: $O(n^3)$
  - Polynomial: $O(n^k)$ for some $k \in \mathbb{N}$
  - Exponential: $O(2^n)$
    - Note: $2^{2n} \notin O(2^n)$, since $2^{2n} = (2^n)^2 = 2^n \cdot 2^n$
    - So a constant can only be ignored in front of the whole expression, not inside!

20

Master theorem

- Solving recurrences in general might be tricky.
  - When the recurrence is of the form T(n)=a T(n/b)+f(n), there is a general method to estimate the growth rate of a function defined by the recurrence
  - This is called the Master Theorem for recurrences.

21

O-notation and computational complexity

- If there is an algorithm that for every input x solves the problem in at most $t(|x|)$ steps for some $t: \mathbb{N} \to \mathbb{N}$ such that $t(n) \in O(f(n))$, then the problem is *solvable in time* $O(f(n))$.
  - If there are several ways to solve the problem, pick the algorithm with slowest growing t(n)

  - Comparing number of steps rather than actual running times to avoid having to compare different implementations running on different hardware.
  - Our algorithm for the Tower of Hanoi with n disks took $2^n - 1$ steps, so complexity of Tower of Hanoi is in $O(2^n)$

22

O-notation and computational complexity

Sometimes (rarely!) we can also prove that it is not possible to solve the problem faster than $c \cdot f(n)$: then, we say that the problem has complexity $\Theta(f(n))$

- For Tower of Hanoi, any algorithm needs $\geq 2^n - 1$ steps.
- So the complexity of the Tower of Hanoi is $\Theta(2^n)$

Proving that there is no faster way to solve a problem is harder!

- To put an *upper bound* on complexity of a problem, enough to find **one** algorithm solving it in $t(n) \in O(f(n))$ steps.
- To prove a *lower bound* $f(n)$ on complexity of a problem, have to show that **every algorithm** solving it correctly has $f(n) \in O(t(n))$
  - Notation: $t(n) \in \Omega(f(n))$, pronounced "big-omega"

23

24

Closed form of some sequences

- Arithmetic progression: $a, a+d, a+2d, a+3d, \dots, a+nd, \dots$
  - **Closed form:** s_n = a $+ nd$
- Geometric progression: $a, ar, ar^2, ar^3, \dots, ar^n, \dots$
  - **Closed form:** $s_n = a \cdot r^n$
- Fibonacci sequence: 1,1,2,3,5,8,13, …
  - $F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$
  - **Closed form:** $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$
    - Where φ *("phi")* is the "golden ratio": a ratio such that $\frac{a+b}{a} = \frac{a}{b}$
    - $\varphi = \frac{1+\sqrt{5}}{2}$

25

More examples of recursive definitions

There is much more that can be defined with recursive definitions rather than just sequences and functions.

In the following recursive definitions, we will call the recursive step "recursion" rather than "recurrence", as we are not defining n^{th} element of a sequence (or function value at n)

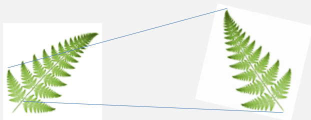
Recursive definition of a sum (where $n \geq m$)
- Basis: $\sum_{i=m}^{m} f(i) = f(m)$.
- Recursion: $\sum_{i=m}^{n+1} f(i) = (\sum_{i=m}^{n} f(i)) + f(n+1)$

Recursive definition of a product (where $n \geq m$)
- Basis: $\prod_{i=m}^{m} f(i) = f(m)$.
- Recursion: $\prod_{i=m}^{n+1} f(i) = (\prod_{i=m}^{n+1} f(i)) * f(n+1)$

26

Fractals

- Can use recursive definitions to define fractals
  - And draw them
  - And prove their properties.
- A fractal is a self-similar object: a part looks like the whole.



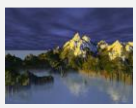
27

Fractals in nature

- A fern leaf

- Romanesco broccoli

- Mountains



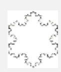
- Stock market

28

29

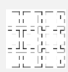
Mathematical fractals

- Koch curve and snowflake



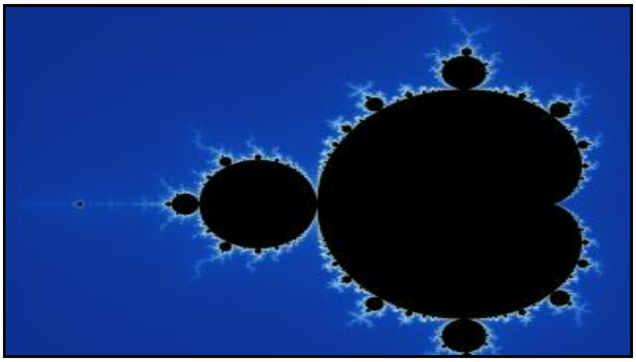
- Sierpinski triangle, pyramid, carpet

- Hilbert space-filling curve



- Mandelbrot set

30

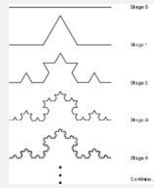


31

Koch curve

Recursive definition of Koch curve:

- *Basis:* an interval
- *Recursion:* Replace the inner third of each interval with two intervals of the same length sticking out in a triangle
  - That is, make a equilateral triangle on top of the middle third, then remove the middle third leaving the remaining two sides of the triangle.

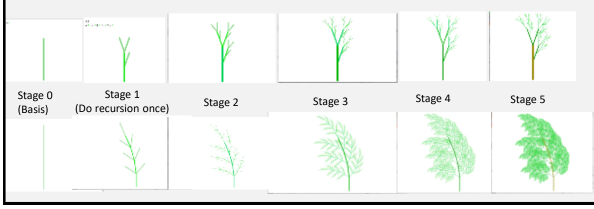


Is it a line? Is it a plane? It is a fractal! Its dimension is $\log_3 4$

32

Playing with fractals

- Fractal Grower by Joel Castellanos:
- http://www.cs.unm.edu/~joel/PaperFoldingFractal/paper.html



33

34

Recursive definitions of sets

- So far, we talked about recursive definitions of sequences, functions, formulas and fractals. We can, in general, recursively define sets.
  - Recursive definition of a set S=$\{0,1\}^*$
    - Basis: empty string λ is in S.
    - Recursion: 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
      - If $w = 011$, then $w0 = 0110$, and $w1 = 0111$
  - Alternatively:
    - Basis: empty string λ, 0 and 1 are in S.
    - Recursion: 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
  - We always assume that the set S contains only elements produced from basis using recursion rule.

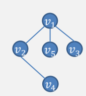
35

Trees

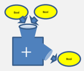
- In computer science, a **tree** is an undirected graph without cycles
  - **Undirected**: all edges go both ways, no arrows.
  - **Cycle**: sequence of edges going back to the same point.

Undirected cycle (not a tree)

- Recursive definition of trees:
  - Basis: A single vertex v is a tree.
  - Recursion:
    - Let T be a tree, and v a new vertex not in T.
    - Then a new tree consist of T, v, and an edge (connection) between some vertex of T and v.



36



Arithmetic expressions

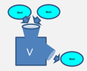
Suppose you are writing a piece of code that takes an arithmetic expression (“5*3-1”, “40-(x+1)*7”, etc), checks that it is well-formed (input is correct), and evaluates it.

How to describe a well-formed arithmetic expression? Define a set of all well-formed arithmetic expressions recursively:

- Basis: A number or a variable is a well-formed arithmetic expression.
  - 5, 100, x, a
- Recursion: If A and B are well-formed arithmetic expressions then so are (A), $A + B, A - B,\ A * B, A$ / B.

40-(x+1)*7 is well-formed: first build 40, x, 1, 7. Then x+1. Then (x+1). Then (x+1)*7, finally 40-(x+1)*7
- Caveat: how do we know the order of evaluation? On that later.

37

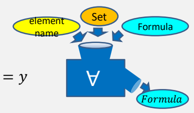


Formulas

- What is a well-formed propositional logic formula?
  - $(p \vee \neg q) \wedge r \rightarrow (\neg p \rightarrow r)$
  - Basis: a propositional variable p, q, r ...
    - Or a constant $TRUE, FALSE$
  - Recursion: if F and G are propositional formulas, so are $(F),\ \neg F,$ $F \wedge G, F \vee G, F \rightarrow G, F \leftrightarrow G.$
  - And nothing else is a well-formed propositional logic formula.

38

Formulas



What is a well-formed predicate logic formula?

$$\exists x \in D\ \forall y \in \mathbb{Z}\ (P(x,y) \vee Q(x,z)) \wedge x = y$$

- Basis: a predicate with inputs from its domain
  - P(x), x=y, Even(5),...
- Recursion:
  - If F and G are predicate logic formulas, so are $(F),\ \neg F,\ F \wedge G, F \vee G, F \rightarrow G, F \leftrightarrow G.$
  - If F is a predicate logic formula with a free variable x, and D is the domain of x, then $\exists x \in D\ F$ and $\forall x \in D\ F$ are predicate logic formulas.
- And nothing else.
  - So $\exists x \in People\ \ Likes(x, y \wedge x),\ Likes(y \neq x)$ is not a well-formed predicate logic formula!

39

Puzzle

- Are the following English sentences built the same way?
  - Time flies like an arrow.

  - Fruit flies like an apple.

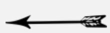
40

41

Puzzle

- Are the following English sentences built the same way?

  - Time flies like an arrow.



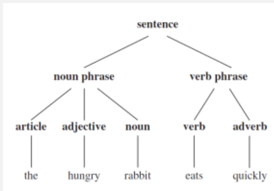
  - Fruit flies like an apple.

42

Grammars

- Remember that sets of strings are called **languages**.
- A type of recursive definition of a language is called a **grammar**.

- Different natural languages also have different grammars!
  - English: Subject/Verb/Object
  - Japanese: Subject/Object/Verb
  - Gaelic: Verb/Subject/Object
  - Russian: order does not matter



43

Context-free grammars

A special kind of grammars is **context-free grammars**, where symbols have the same meaning wherever they are.

A context-free grammar consists of

- A set V of **variables** (using capital letters), including a **start variable** S.
  - Note: these are variables, not sets.
- A set Σ of **terminals** (disjoint from V; alphabet of the language.)
- A set R of **rules**, where each rule consists of a variable from V, followed by $\rightarrow$, followed by a string of variables and terminals.
  - This is not the same "$\rightarrow$" as implication, a different use of the same symbol.
  - If several rules start with the same non-terminal such as $A \rightarrow w_1, A \rightarrow w_2$, etc, we use a shortcut "|" (means "or") to write them as one rule: $A \rightarrow w_1 | \, w_2 | \dots | w_k$

44

Context-free grammars

A context-free grammar: a set V of **variables**, including a **start variable** S, a set Σ (Sigma) of **terminals** (alphabet) and a set R of **rules** of the form $A \rightarrow w$, where $A \in V, w \in (V \cup \Sigma)^*$

- If $A \rightarrow w$ is a rule, we say variable A **yields** string w.
  - Can use A within the rule, as many times as we want: recursion!
  - Different occurrences of the same variable can produce different strings.
- A **derivation** is a sequence of strings each of which is obtained from the previous by applying some rule to its substring.
  - If $w = \ell Ar$ and there is a rule $A \rightarrow u$, then ℓur is **directly derived** from w, written $w \Rightarrow \ell ur$.
  - w_n is **derived from** w_0 if there is a sequence $w_0, w_1, \dots, w_n$ where $\forall i \; w_i \Rightarrow w_{i+1}$
- **A language generated by a grammar** consists of all strings of terminals that can be derived from the start variable S.

45

Language L of all strings over {0,1} with all 0s before all 1s.

Strings in L: λ, 0, 1, 11, 001, 00111, 011111, 00000000,…

Strings not in L: 10, 1110, 010101, 00100000…

$S \rightarrow \lambda$
$S \rightarrow 0S$
$S \rightarrow S1$

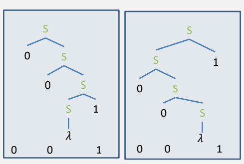
Recursive definition:
- Basis: $\lambda \in L$
- Recursion: if $w \in L$, then $0w \in L$ and $w1 \in L$

- Grammar for L consists of 3 rules: $S \rightarrow 0S,\ S \rightarrow S1,\ S \rightarrow \lambda$
  - Shorter description of the grammar: $S \rightarrow 0S \mid S1 \mid \lambda$
  - Variables: S. Terminals: 0 and 1. As before, λ is the empty string.
  - Derivation of a string 001: $S \Rightarrow 0S \Rightarrow 0S1 \Rightarrow 00S1 \Rightarrow 00\lambda1 = 001$
    - Alternative derivation of 001: $S \Rightarrow S1 \Rightarrow 0S1 \Rightarrow 00S1 \Rightarrow 00\lambda1 = 001$

46

Parse trees:

- Parse trees: visualizing derivations
  - Similar to syntax trees,
  - except all internal nodes are variables,
  - and all nodes on the bottom are terminals.



Grammar: $S \rightarrow 0S \mid S1 \mid \lambda$
- Derivation of a string 001:
  $S \Rightarrow 0S \Rightarrow 00S \Rightarrow 00S1 \Rightarrow 00\lambda1 = 001$
- Alternative derivation of 001:
  $S \Rightarrow S1 \Rightarrow 0S1 \Rightarrow 00S1 \Rightarrow 00\lambda1 = 001$

47

Examples of context-free grammars

- L contains two strings 1 and 00
  $S \rightarrow \ 1 \mid 00$
  - Variables: S.
  - Terminals: 1 and 00 (or 1 and 0).

- L is strings over $\{a, b, c, d\}$ with even number of "a"s
  $S \rightarrow SaSaS \mid SbS \mid ScS | SdS | \lambda$
  - Variables: S. Terminals: a,b,c,d.

- Natural numbers, start symbol N
  $\mathrm{N} \rightarrow \ 0 \mid M$
  $\mathrm{M} \rightarrow 1D|2D|3D|4D$
  $M \rightarrow 5D|6D|7D|8D|9D$
  $D \rightarrow M|0D|\lambda$

- Rational numbers: same as natural numbers plus a new start symbol R, new variable P and new rules
  $R \rightarrow P|\text{-}P$
  $\mathrm{P} \rightarrow N|N/M|N.N$

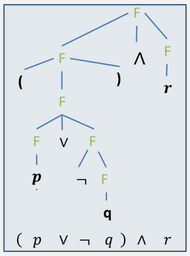
48

Propositional formulas

- Recursive definition:
  - Basis: a propositional variable p, q, r or a constant $TRUE, FALSE$
  - Recursion: if F and G are propositional formulas, then so are $(F),\ \neg F,\ F \wedge G, F \vee G, F \rightarrow G, F \leftrightarrow G$.
- Grammar: $F \rightarrow\ F \vee F | F \wedge F | \neg F | (F) | F \rightarrow G | F \leftrightarrow G$
  $F \rightarrow p \mid q \mid r \mid TRUE \mid FALSE$

Here, the only variable is F (it is a start variable), and the set of terminals is $\{\vee, \wedge, \neg, \rightarrow, \leftrightarrow, (,), p, q, r, TRUE, FALSE\}$

Deriving $(p \vee \neg q) \wedge r$: $F \Rightarrow F \wedge F \Rightarrow (F) \wedge F \Rightarrow (F) \wedge r \Rightarrow$
$\Rightarrow (F \vee F) \wedge r \Rightarrow (p \vee F) \wedge r \Rightarrow (p \vee \neg F) \wedge r \Rightarrow (p \vee \neg q) \wedge r$



49

Context-free grammars for arithmetic expressions

$EXPR \rightarrow EXPR + EXPR \mid EXPR - EXPR \mid EXPR * EXPR$
$EXPR \rightarrow EXPR\ / EXPR \mid (EXPR) \mid 0 \mid NUMBER \mid$-NUMBER
NUMBER $\rightarrow 1DIGITS \mid \ldots \mid 9DIGITS$
$DIGITS \rightarrow \lambda \mid NUMBER \mid 0DIGITS$

Grammar for natural numbers

- Variables: EXPR, NUMBER, DIGITS (EXPR is starting).
- Terminals: +,-,*, /, 0,…,9,(,).

- Problem: this definition of arithmetic expressions (and the previous definition of propositional formulas) do not have any information about order of operations.

50

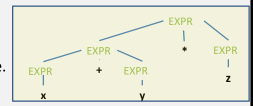
Encoding order of precedence

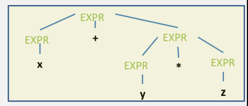
- Easier to specify in which order to process parts of the formula.
  - Better grammar for arithmetic expressions (for simplicity, with only x,y,z instead of numbers):
    $EXPR \rightarrow EXPR + TERM \mid EXPR - TERM \mid TERM$
    $TERM \rightarrow TERM * FACTOR \mid TERM\ / FACTOR \mid FACTOR$
    $FACTOR \rightarrow (EXPR)$ | x | y | z
  - Here, variables are EXPR, TERM and FACTOR (with EXPR a starting variable).
  - Now can encode precedence.

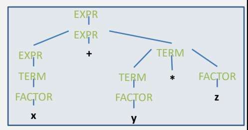
51

Ambiguous grammars

- A context-free grammar is **ambiguous** if there is some string with more than one possible parse tree.
  - Grammar $S \rightarrow 0S | S1 | \lambda$ was ambiguous since 001 had two parse trees.
- First grammar for arithmetic expressions (and grammar for propositional formulas) were ambiguous: $x + y * z$ has two parse rees.
  $EXPR \rightarrow EXPR + EXPR | EXPR - EXPR$
  $EXPR \rightarrow EXPR * EXPR \mid EXPR\ / EXPR$
  $EXPR \rightarrow (EXPR)$|x|y|z
- Second grammar for arithmetic expressions is not ambiguous
  $EXPR \rightarrow EXPR + TERM \mid EXPR - TERM \mid TERM$
  $TERM \rightarrow TERM * FACTOR \mid TERM\ / FACTOR \mid FACTOR$
  $FACTOR \rightarrow (EXPR)$ | x | y | z







52

53

Recursive definitions of sets

- So far, we talked about recursive definitions of sequences, functions, formulas and fractals. We can, in general, recursively define sets.
  - Recursive definition of a set $S=\{0,1\}^*$
    - Basis: empty string λ is in S.
    - Recursion: 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
      - If $w = 011$, then $w0 = 0110$, and $w1 = 0111$
  - Alternatively:
    - Basis: empty string λ, 0 and 1 are in S.
    - Recursion: 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
  - We always assume that the set S contains only elements produced from basis using recursion rule.

54

Structural induction

- Let $S \subseteq U$ be a recursively defined set
- Let F(x) be a predicate with domain U
  - Think of $F(x)$ as some property that elements of U may have.
- Then
  - if $F(x)$ is true for all x in the basis of S,
  - and applying the recursion rules preserves F.
  - then all elements in S have the property F.

55

Let's define a set S of numbers as follows.
- Basis: $3 \in S$
- Recursion: if $x, y \in S$, then $x + y \in S$

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

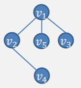
Proof (by structural induction).
- **Base case:** 3 is divisible by 3 (z=1).
- **Recursive step:**
  - Let $x, y \in S$. Then $\exists z, u \in \mathbb{N}\ x = 3z \wedge y = 3u$. **(inductive hypothesis)**
  - Then $x + y = 3z + 3u = 3(z + u)$. **(induction step)**
  - Therefore, $x + y$ is divisible 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.

56

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.
- 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.

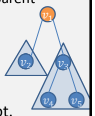
Undirected cycle (not a tree)



57

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.
    - For a vertex in a rooted tree, its neighbour towards the root is called "parent"
    - And its neighbours away from the root are called "children"
  - **Full** if every vertex has either 2 or 0 edges going away from the root.

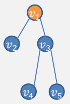


- Recursive definition of full binary trees:
  - Basis: A single vertex v is a full binary tree with that vertex as a root.
  - Recursion:
    - Let T_1, T_2 be full binary trees with roots r_1, r_2, respectively. Let v be a new vertex.
    - A new full binary tree with root v is formed by connecting r_1 and r_2 to v.

58

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 $n(T) \leq 2^{h(T)+1} - 1$

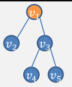


Height 2

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

59

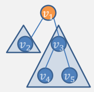
Height of a full binary tree



- Claim: Let $n(T)$ be the number of vertices in a full binary tree T.
  Then $n(T) \leq 2^{h(T)+1} - 1$, where $h(T)$ is the height of T.
- 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 $\mathrm{n(T)} = n(T_1) + n(T_2) + 1$
    - 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 $n(T_2) \leq 2^{h(T_2)+1} - 1$
    - $\mathrm{n(T)} = \cdots$ *(see next page)*

60

- Claim: Let $n(T)$ be the number of vertices in a full binary tree T.
  Then $n(T) \leq 2^{h(T)+1} - 1$, where $h(T)$ is the height of T.
- Proof (by structural induction)
  - Base case: holds.
  - Recursion: Suppose T was built by attaching T_1, T_2 to a new root vertex v.
    - Number of vertices in T is $\mathrm{n(T)} = n(T_1) + n(T_2) + 1$
    - 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 $n(T_2) \leq 2^{h(T_2)+1} - 1$

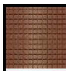


- $\mathrm{n(T)} = n(T_1) + n(T_2) + 1$

$$\leq 1 + (2^{h(T_1)+1} - 1) + (2^{h(T_2)+1} - 1) \quad \text{(by ind. hyp)}$$
$$\leq 2 \cdot \max(2^{h(T_1)+1}, 2^{h(T_2)+1}) - 1$$
$$\leq 2 \cdot 2^{\max(h(T_1), h(T_2))+1} - 1$$
$$= 2 \cdot 2^{h(T)} - 1 = 2^{h(T)+1} - 1$$

Therefore, the number of vertices of any binary tree T is $\leq 2^{h(T)+1} - 1$

61




Puzzle: chocolate squares

- Suppose you have a piece of chocolate like this:

- How many squares are in it?
  - of all sizes, from single to the whole thing

62