General Computer Science 320101/2 — 2008/9

Michael Kohlhase
School of Engineering & Computer Science
Jacobs University
m.kohlhase@jacobs-university.de
office: Room 62, Research 1, phone: x3140

 1 JACOBS UNIVERSITY

1 Preface

[1] EdNote(1)

This document contains the course notes for the course General Computer Science I & II held at Jacobs University Bremen[1] in the academic years 2003-2008. The document mixes the slides presented in class with comments of the instructor to give students a more complete background reference.

This document is made available for the students of this course only. It is still a draft, and will develop over the course of the course. It will be developed further in coming academic years.

This document is also an experiment in knowledge representation. Under the hood, it uses the sTeX package, a TeX/LaTeX extension for semantic markup. Eventually, this will enable to export the contents into eLearning platforms like *Connexions* (see `http://cnx.rice.edu`) or *ActiveMath* (see `http://www.activemath.org`).

Comments and extensions are always welcome, please send them to the author.

[1] EdNote: extend this into a real preface

[1] International University Bremen until Fall 2006

Acknowledgments: Some of the material in this course is based on course notes prepared by Andreas Birk, who held the course 320101/2 "General Computer Science" at IUB in the years 2001-03. Parts of his course and the current course materials were based on the book "Hardware Design" (in German; Keller/Paul; Teubner Leibzig 1995). The section on Search algorithms is based on materials obtained from Bernhard Beckert (Uni Koblenz), which in turn are based on Russel&Norwig's lecture slides that go with the book "Artificial Intelligence: A Modern Approach" (Second Edition) by Stuart Russell and Peter Norvig.

The presentation of the Programming language Standard ML, which serves as the primary programming tool of this course is in part based on the course notes of Gert Smolka's excellent course "Programming" at Saarland University, which will appear as a book (in German) soon.

The preparation of the course notes has been greatly helped by Ioan Sucan, who has done much of the initial editing needed for semantic preloading. Herbert Jäger and Normen Müller have given advice on the contents, and Immanuel Normann has contributed many of the problems and organized them consistently.

The following students have submitted corrections and suggestions to this and earlier versions of the notes: Saksham Raj Gautam, Anton Kirilov, Philipp Meerkamp, Paul Ngana, Darko Pesikan, Stojanco Stamkov, Nikolaus Rath, Evans Bekoe, Marek Laska, Moritz Beber, Andrei Aiordachioaie, Magdalena Golden, Andrei Eugeniu Ioniţă, Semir Elezović, Dimitar Asenov, Alen Stojanov, Felix Schlesinger, Ştefan Anca, Dante Stroe, Irina Calciu, Nemanja Ivanovski, Abdulaziz Kivaza, Anca Dragan, Razvan Turtoi, Catalin Duta, Andrei Dragan, Dimitar Misev, Vladislav Perelman, Milen Paskov, Kestutis Cesnavicius, Mohammad Faisal, Janis Beckert, Karolis Uziela, Josip Djolonga, Flavia Grosan, Aleksandar Siljanovski.

Contents

2 Welcome and Administrativa

Happy new year! and Welcome Back!

▷ I hope you have recovered over the last 6 weeks (slept a lot)

▷ I hope that those of you who had problems last semester have caught up on the material (We will need much of it this year)

▷ I hope that you are eager to learn more about Computer Science (I certainly am!)

 2 JACOBS UNIVERSITY

Your Evaluations

▷ First: thanks for filling out the forms (to all 55 of you!)

Evaluations are a good tool for optimizing teaching/learning

▷ Second: I have read all of them, and I will take action on some of them.

- ▷ *Change the instructor next year!* (not your call)
- ▷ *nice course. SML rulez! I really learned recursion* (thanks)
- ▷ *To improve this course, I would remove its "ML part"* (let me explain,...)
- ▷ *He doesnnt' care about teaching. He simply comes unprepared to the lectures* (have you ever attended?)
- ▷ *the slides tell simple things in very complicated ways* (this is a problem)
- ▷ *The problem is with the workload, it is too much* (I agree, but we want to give you a chance to become Computer Scientists)
- ▷ *More examples should be provided,* (will try to this; e.g. worked problems)

 3 JACOBS UNIVERSITY

3 Recap from General CS I

Recap from GenCSI: Discrete Math and SML

- ▷ MathTalk (Rigorous communication about sets, relations,functions)
- ▷ unary natural numbers. (we have to start with something)
  - ▷ Axiomatic foundation, in particular induction (Peano Axioms)
  - ▷ constructors s, o, defined functions like $+$
- ▷ Abstract Data Types (ADT) (generalize natural numbers)
  - ▷ sorts, constructors, (defined) parameters, variables, terms, substitutions
  - ▷ define parameters by (sets of) recursive equations (rules)
  - ▷ abstract interpretation, termination,
- ▷ Programming in SML (ADT on real machines)
  - ▷ strong types, recursive functions, higher-order syntax, exceptions, ...
  - ▷ basic data types/algorithms: numbers, lists, strings,

  JACOBS UNIVERSITY

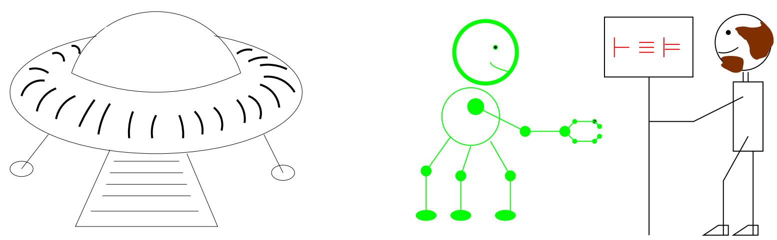
Recap from GenCSI: Formal Languages and Boolean Algebra

- ▷ Formal Languages and Codes (models of "real" programming languages)
  - ▷ string codes, prefix codes, uniform length codes
  - ▷ formal language for unary arithmetics (onion architecture)
  - ▷ syntax and semantics (...by mapping to something we understand)
- ▷ Boolean Algebra (special syntax, semantics, ...)
  - ▷ Boolean functions vs. expressions (syntax vs. semantics again)
  - ▷ Normal forms (Boolean polynomials, clauses, CNF, DNF)
- ▷ Complexity analysis (what does it cost in the limit?)
  - ▷ Landau Notations (aka. "big-O") (function classes)
  - ▷ upper/lower bounds on costs for Boolean functions (all exponential)
- ▷ Constructing Minimal Polynomials (simpler than general minimal expressions)
  - ▷ Prime implicants, Quine McCluskey (you really liked that...)

  JACOBS UNIVERSITY

Recap from GenCSI: Properties of Calculi

- Correctness: (provable implies valid)
  - $\mathcal{H} \vdash \mathbf{B}$ implies $\mathcal{H} \models \mathbf{B}$ (equivalent: $\vdash \mathbf{A}$ implies $\models \mathbf{B}$)
- Completeness: (valid implies provable)
  - $\mathcal{H} \models \mathbf{B}$ implies $\mathcal{H} \vdash \mathbf{B}$ (equivalent: $\models \mathbf{A}$ implies $\vdash \mathbf{B}$)
- Goal: $\vdash \mathbf{A}$ iff $\models \mathbf{A}$ (provability and validity coincide)
  - To TRUTH through PROOF (CALCULEMUS [Leibniz ~1680])



Test Calculi: Tableaux and Model Generation

- Idea: instead of showing $\emptyset \vdash Th$, show $\neg Th \vdash$ trouble (use $\bot$ for trouble)

Tableau Refutation (Validity) $\models P \wedge Q \Rightarrow Q \wedge P$

$$\begin{array}{c} P \wedge Q \Rightarrow Q \wedge P^{\mathsf{F}} \\ P \wedge Q^{\mathsf{T}} \\ Q \wedge P^{\mathsf{F}} \\ P^{\mathsf{T}} \\ Q^{\mathsf{T}} \\ \begin{array}{c|c} P^{\mathsf{F}} & Q^{\mathsf{F}} \\ \bot & \bot \end{array} \end{array}$$

No Model

Model generation (Satisfiability) $\models P \wedge (Q \vee \neg R) \wedge \neg Q$

$$\begin{array}{c} P \wedge (Q \vee \neg R) \wedge \neg Q^{\mathsf{T}} \\ P \wedge (Q \vee \neg R)^{\mathsf{T}} \\ \neg Q^{\mathsf{T}} \\ Q^{\mathsf{F}} \\ P^{\mathsf{T}} \\ Q \vee \neg R^{\mathsf{T}} \\ \begin{array}{c|c} Q^{\mathsf{T}} & \neg R^{\mathsf{T}} \\ \bot & R^{\mathsf{F}} \end{array} \end{array}$$

Model $\{P^{\mathsf{T}}, Q^{\mathsf{F}}, R^{\mathsf{F}}\}$

Variable Assignment: $\varphi := \{P \mapsto \mathsf{T}, Q \mapsto \mathsf{F}, R \mapsto \mathsf{F}\}$

- Algorithm: Fully expand all possible tableaux, (no rule can be applied)
  - Satisfiable, iff there are open branches (correspond to models)

Tableau calculi develop a formula in a tree-shaped arrangement that represents a case analysis on when a formula can be made true (or false). Therefore the formulae are decorated with exponents that hold the intended truth value.

On the left we have a refutation tableau that analyzes a negated formula (it is decorated with the intended truth value F). Both branches contain an elementary contradiction $\bot$.

On the right we have a model generation tableau, which analyzes a positive formula (it is decorated with the intended truth value T. This tableau uses the same rules as the refutation tableau, but makes a case analysis of when this formula can be satisfied. In this case we have a closed branch and an open one, which corresponds a model).

Now that we have seen the examples, we can write down the tableau rules formally.

4 Resolution for Propositional Logic

The next calculus is a test calculus based on the conjunctive normal form. In contrast to the tableau method, it does not compute the normal form as it goes along, but has a pre-processing step that does this and a single inference rule that maintains the normal form. The goal of this calculus is to derive the (the empty disjunction), which is unsatisfiable.

Another Test Calculus: Resolution

- Definition 4.1: A clause is a disjunction of literals. We will use $\square$ for the empty disjunction (no disjuncts) and call it the empty clause.
- Definition 4.2: (Resolution Calculus)

  The resolution calculus operates a clause sets via a single inference rule:

$$\frac{P^{\mathsf{T}} \vee \mathbf{A} \quad P^{\mathsf{F}} \vee \mathbf{B}}{\mathbf{A} \vee \mathbf{B}}$$

  This rule allows to add the clause below the line to a clause set which contains the two clauses above.
- Definition 4.3: (Resolution Refutation)

  Let S be a clause set, and $(\mathcal{D}\colon S \vdash_{\mathcal{R}} T)$ a $\mathcal{R}$ derivation then we call $\mathcal{D}$ resolution refutation, iff $\square \in T$.

A calculus for CNF Transformation

- Definition 4.4: (Transformation into Conjunctive Normal Form)

  The CNF transformation calculus $\mathcal{CNF}$ consists of the following four inference rules on clause sets.

$$\frac{\mathbf{C} \vee (\mathbf{A} \vee \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}} \qquad \frac{\mathbf{C} \vee (\mathbf{A} \vee \mathbf{B})^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{F}}} \qquad \frac{\mathbf{C} \vee \neg\mathbf{A}^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}}} \qquad \frac{\mathbf{C} \vee \neg\mathbf{A}^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}}$$

- Definition 4.5: We write $\mathit{CNF}(\mathbf{A})$ for the set of all clauses derivable from $\mathbf{A}^{\mathsf{F}}$ via the rules above.
- Definition 4.6: (Resolution Proof)

  We call a resolution refutation $(\mathcal{P}\colon \mathit{CNF}(\mathbf{A}) \vdash_{\mathcal{R}} T)$ a resolution sproof for $\mathbf{A} \in \mathit{wff}_o(\mathcal{V}_o)$.

Note: Note that the **C**-terms in the definition of the resolution calculus are necesary, since we assumed that the assumptions of the inference rule must match full formulae. The **C**-terms are used with the convention that they are optional. So that we can also simplify $(\mathbf{A} \vee \mathbf{B})^{\mathsf{T}}$ to $\mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}$.

The background behind this notation is that $\mathbf{A}$ and $T \vee \mathbf{A}$ are equivalent for any $\mathbf{A}$. That allows us to interpret the **C**-terms in the assumptions as T and thus leave them out.

The resolution calculus as we have formulated it here is quite frugal; we have left out rules for the connectives $\vee$, $\Rightarrow$, and $\Leftrightarrow$, relying on the fact that formulae containing these connectives can be translated into ones without before CNF transformation. The advantage of having a calculus with few inference rules is that we can prove meta-properties like soundness and completeness with less effort (these proofs usually require one case per inference rule). On the other hand, adding specialized inference rules makes proofs shorter and more readable.

Fortunately, there is a way to have your cake and eat it. Derived inference rules have the property that they are formally redundant, since they do not change the expressive power of the calculus. Therefore we can leave them out when proving meta-properties, but include them when actually using the calculus.

Derived Rules of Inference

- Definition 4.7: Let $\mathcal{C}$ be a calculus, a rule of inference $\frac{\mathbf{A}_1 \quad \ldots \quad \mathbf{A}_n}{\mathbf{C}}$ is called a derived inference rule in $\mathcal{C}$, iff there is a $\mathcal{C}$-proof of $\mathbf{A}_1, \ldots, \mathbf{A}_n \vdash \mathbf{C}$.

- Example 4.8:

$$\cfrac{\cfrac{\cfrac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee (\neg\mathbf{A} \vee \mathbf{B})^{\mathsf{T}}}}{\mathbf{C} \vee \neg\mathbf{A}^{\mathsf{T}} \vee \mathbf{B}^{\mathsf{T}}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}} \quad \mapsto \quad \frac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}}$$

- Others:

$$\frac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{T}}} \quad \frac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{F}}} \qquad \frac{\mathbf{C} \vee \mathbf{A} \wedge \mathbf{B}^{\mathsf{T}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{T}}; \mathbf{C} \vee \mathbf{B}^{\mathsf{T}}} \quad \frac{\mathbf{C} \vee \mathbf{A} \wedge \mathbf{B}^{\mathsf{F}}}{\mathbf{C} \vee \mathbf{A}^{\mathsf{F}} \vee \mathbf{B}^{\mathsf{F}}}$$

 10 JACOBS UNIVERSITY

With these derived rules, theorem proving becomes quite efficient. To get a better understanding of the calculus, we look at an example: we prove an axiom of the Hilbert Calculus we have studied above.

Example: Proving Axiom S

▷ Example 4.9: Clause Normal Form transformation

$$\cfrac{\cfrac{\cfrac{(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R^{\mathsf{F}}}{P \Rightarrow Q \Rightarrow R^{\mathsf{T}}; (P \Rightarrow Q) \Rightarrow P \Rightarrow R^{\mathsf{F}}}}{P^{\mathsf{F}} \vee (Q \Rightarrow R)^{\mathsf{T}}; P \Rightarrow Q^{\mathsf{T}}; P \Rightarrow R^{\mathsf{F}}}}{P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}; P^{\mathsf{F}} \vee Q^{\mathsf{T}}; P^{\mathsf{T}}; R^{\mathsf{F}}}$$

$$CNF = \{P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}\,,\ P^{\mathsf{F}} \vee Q^{\mathsf{T}}\,,\ P^{\mathsf{T}}\,,\ R^{\mathsf{F}}\}$$

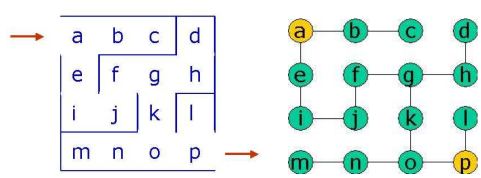
▷ Example 4.10: Resolution Proof

1	$P^{\mathsf{F}} \vee Q^{\mathsf{F}} \vee R^{\mathsf{T}}$	initial
2	$P^{\mathsf{F}} \vee Q^{\mathsf{T}}$	initial
3	P^{T}	initial
4	R^{F}	initial
5	$P^{\mathsf{F}} \vee Q^{\mathsf{F}}$	resolve 1.3 with 4.1
6	Q^{F}	resolve 5.1 with 3.1
7	P^{F}	resolve 2.2 with 6.1
8	$\square$	resolve 7.1 with 3.1

5 Graphs and Trees

Some more Discrete Math: Graphs and Trees

▷ Remember our Maze Example from the Intro? (long time ago)



$$\left\langle \left\{ \begin{array}{l} \langle a,e\rangle, \langle e,i\rangle, \langle i,j\rangle, \\ \langle f,j\rangle, \langle f,g\rangle, \langle g,h\rangle, \\ \langle d,h\rangle, \langle g,k\rangle, \langle a,b\rangle \\ \langle m,n\rangle, \langle n,o\rangle, \langle b,c\rangle \\ \langle k,o\rangle, \langle o,p\rangle, \langle l,p\rangle \end{array} \right\}, a, p \right\rangle$$

▷ We represented the maze as a graph for clarity.

▷ Now, we are interested in circuits, which we will also represent as graphs.

▷ Let us look at the theory of graphs first (so we know what we are doing)

Graphs and trees are fundamental data structures for computer science, they will pop up in many disguises in almost all areas of CS. We have already seen various forms of trees: formula trees, tableaux, ... We will now look at their mathematical treatment, so that we are equipped to talk and think about combinatory circuits.

We will first introduce the formal definitions of graphs (trees will turn out to be special graphs), and then fortify our intuition using some examples.

Basic Definitions: Graphs

▷ Definition 5.1: An undirected graph is a pair $\langle V, E\rangle$ such that

- ▷ V is a set of so-called vertices (or nodes) (draw as circles)
- ▷ $E \subseteq \{\{v, v'\} \mid v, v' \in V, v \neq v'\}$ is the set of its undirected edges (draw as lines)

▷ Definition 5.2: A directed graph (also called digraph) is a pair $\langle V, E\rangle$ such that

▷
- ▷ V is a set of vertexes
- ▷ $E \subseteq V \times V$ is the set of its directed edges

▷ Definition 5.3: Given a graph $G = \langle V, E\rangle$. The in-degree $indeg(v)$ and the out-degree $outdeg(v)$ of a vertex $v \in V$ are defined as

- ▷ $indeg(v) = \#\{w \mid \langle w, v\rangle \in E\}$
- ▷ $outdeg(v) = \#\{w \mid \langle v, w\rangle \in E\}$

Note: For an undirected graph, $indeg(v) = outdeg(v)$ for all nodes v.

 13 JACOBS UNIVERSITY

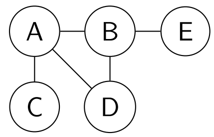
We will mostly concentrate on directed graphs in the following, since they are most important for the applications we have in mind. Many of the notions can be defined for undirected graphs with a little imagination. For instance the definitions for *indeg* and *outdeg* are the obvious variants: $indeg(v) = \#\{w \mid \{w, v\} \in E\}$ and $outdeg(v) = \#\{w \mid \{v, w\} \in E\}$

In the following if we do not specify that a graph is undirected, it will be assumed to be directed.

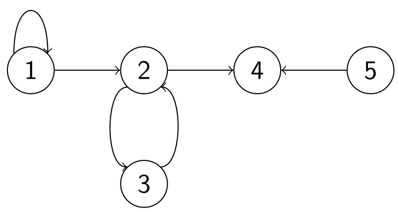
This is a very abstract yet elementary definition. We only need very basic concepts like sets and ordered pairs to understand them. The main difference between directed and undirected graphs can be visualized in the graphic representations below:

Examples

- ▷ Example 5.4: An undirected graph $G_1 = \langle V_1, E_1 \rangle$, where $V_1 = \{A, B, C, D, E\}$ $E_1 = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{B, E\}\}$



- ▷ Example 5.5: A directed graph $G_2 = \langle V_2, E_2 \rangle$, where $V_2 = \{1, 2, 3, 4, 5\}$ $E_2 = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle, \langle 2, 4 \rangle, \langle 5, 4 \rangle\}$



In a directed graph, the edges (shown as the connections between the circular nodes) have a direction (mathematically they are ordered pairs), whereas the edges in an undirected graph do not (mathematically, they are represented as a set of two elements, in which there is no natural order).

Note furthermore that the two diagrams are not graphs in the strict sense: they are only pictures of graphs. This is similar to the famous painting by René Magritte that you have surely seen before.

The Graph Diagrams are not Graphs

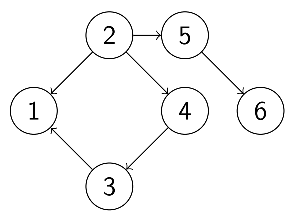
They are pictures of graphs (of course!)

If we think about it for a while, we see that directed graphs are nothing new to us. We have defined a directed graph to be a set of pairs over a base set (of nodes). These objects we have seen in the beginning of this course and called them relations. So directed graphs are special relations. We will now introduce some nomenclature based on this intuition.

Directed Graphs

- ▷ Idea: Directed Graphs are nothing else than relations
- ▷ Definition 5.6: Let $G = \langle V, E \rangle$ be a directed graph, then we call a node $v \in V$
  - ▷ initial, iff there is no $w \in V$ such that $\langle w, v \rangle \in E$. (no predecessor)
  - ▷ terminal, iff there is no $w \in V$ such that $\langle v, w \rangle \in E$. (no successor)

  In a graph G, node v is also called a source (sink) of G, iff it is initial (terminal) in G.
- ▷ Example 5.7: The node 2 is initial, and the nodes 1 and 6 are terminal in



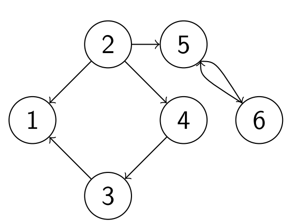
For mathematically defined objects it is always very important to know when two representations are equal. We have already seen this for sets, where $\{a, b\}$ and $\{b, a, b\}$ represent the same set: the set with the elements a and b. In the case of graphs, the condition is a little more involved: we have to find a bijection of nodes that respects the edges.

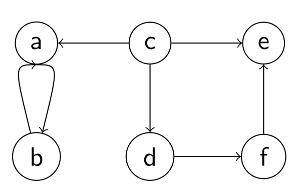
Graph Isomorphisms

- ▷ Definition 5.8: A graph isomorphism between two graphs $G = \langle V, E \rangle$ and $G' = \langle V', E' \rangle$ is a bijective function $\psi \colon V \to V'$ with

directed graphs	undirected graphs
$\langle a, b \rangle \in E \Leftrightarrow \langle \psi(a), \psi(b) \rangle \in E'$	$\{a, b\} \in E \Leftrightarrow \{\psi(a), \psi(b)\} \in E'$

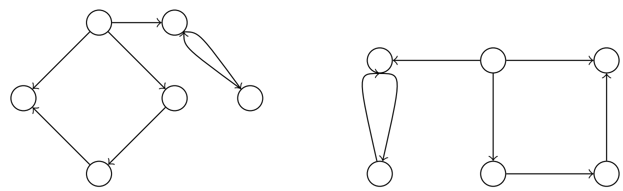
- ▷ Definition 5.9: Two graphs G and G' are equivalent iff there is a graph-isomorphism ψ between G and G'.
- ▷ Example 5.10: G_1 and G_2 are equivalent as there exists a graph isomorphism $\psi := \{a \mapsto 5, b \mapsto 6, c \mapsto 2, d \mapsto 4, e \mapsto 1, f \mapsto 3\}$ between them.





Note that we have only marked the circular nodes in the diagrams with the names of the elements that represent the nodes for convenience, the only thing that matters for graphs is which nodes are connected to which. Indeed that is just what the definition of graph equivalence via the existence of an isomorphism says: two graphs are equivalent, iff they have the same number of nodes and the same edge connection pattern. The objects that are used to represent them are purely coincidental, they can be changed by an isomorphism at will. Furthermore, as we have seen in the example, the shape of the diagram is purely an artifact of the presentation; It does not matter at all.

So the following two diagrams stand for the same graph, (it is just much more difficult to state the graph isomorphism)

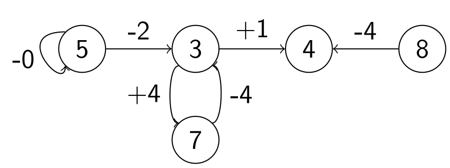


Note that directed and undirected graphs are totally different mathematical objects. It is easy to think that an undirected edge $\{a, b\}$ is the same as a pair $\langle a, b\rangle, \langle b, a\rangle$ of directed edges in both directions, but a priory these two have nothing to do with each other. They are certainly not equivalent via the graph equivalent defined above; we only have graph equivalence between directed graphs and also between undirected graphs, but not between graphs of differing classes.

Now that we understand graphs, we can add more structure. We do this by defining a labeling function from nodes and edges.

Labeled Graphs

- ▷ Definition 5.11: A labeled graph G is a triple $\langle V, E, f\rangle$ where $\langle V, E\rangle$ is a graph and $f: V \cup E \to R$ is a partial function into a set R of labels.
- ▷ Notation 5.12:write labels next to their vertex or edge. If the actual name of a vertex does not matter, its label can be written into it.
- ▷ Example 5.13: $G = \langle V, E, f\rangle$ with $V = \{A, B, C, D, E\}$, where
  - ▷ $E = \{\langle A, A\rangle, \langle A, B\rangle, \langle B, C\rangle, \langle C, B\rangle, \langle B, D\rangle, \langle E, D\rangle\}$
  - ▷ $f: V \cup E \to \{+, -, \emptyset\} \times \{1, \ldots, 9\}$ with
    - ▷ $f(A) = 5, f(B) = 3, f(C) = 7, f(D) = 4, f(E) = 8,$
    - ▷ $f(\langle A, A\rangle) = -0, f(\langle A, B\rangle) = -2, f(\langle B, C\rangle) = +4,$
    - ▷ $f(\langle C, B\rangle) = -4, f(\langle B, D\rangle) = +1, f(\langle E, D\rangle) = +4$



Note that in this diagram, the markings in the nodes do denote something: this time the labels given by the labeling function f, not the objects used to construct the graph. This is somewhat confusing, but traditional.

Now we come to a very important concept for graphs. A path is intuitively a sequence of nodes that can be traversed by following directed edges in the right direction or undirected edges.

Paths in Graphs

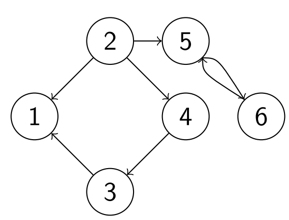
- ▷ Definition 5.14: Given a directed graph $G = \langle V, E\rangle$, then we call a vector $p = \langle v_0, \ldots, v_n\rangle \in V^{n+1}$ a path in G iff $\langle v_{i-1}, v_i\rangle \in E$ for all $1 \leq i \leq n$, $n > 0$.
  - ▷ v_0 is called the start of p (write $start(p)$)
  - ▷ v_n is called the end of p (write $end(p)$)
  - ▷ n is called the length of p (write $len(p)$)

  Note: Not all v_i-s in a path are necessarily different.
- ▷ Notation 5.15:For a graph $G = \langle V, E\rangle$ and a path $p = \langle v_0, \ldots, v_n\rangle \in V^{n+1}$, write
  - ▷ $v \in p$, iff $v \in V$ is a vertex on the path ($\exists i.v_i = v$)
  - ▷ $e \in p$, iff $e = \langle v, v'\rangle \in E$ is an edge on the path ($\exists i.v_i = v \wedge v_{i+1} = v'$)
- ▷ Notation 5.16:We write $\Pi(G)$ for the set of all paths in a graph G.

An important special case of a path is one that starts and ends in the same node. We call it a cycle. The problem with cyclic graphs is that they contain paths of infinite length, even if they have only a finite number of nodes.

Cycles in Graphs

- ▷ Definition 5.17: Given a graph $G = \langle V, E\rangle$, then
  - ▷ a path p is called cyclic (or a cycle) iff $start(p) = end(p)$.
  - ▷ a cycle $\langle v_0, \ldots, v_n\rangle$ is called simple, iff $v_i \neq v_j$ for $1 \leq i, j \leq n$ with $i \neq j$.
  - ▷ graph G is called acyclic iff there is no cyclic path in G.
- ▷ Example 5.18: $\langle 2, 4, 3\rangle$ and $\langle 2, 5, 6, 5, 6, 5\rangle$ are paths in



  $\langle 2, 4, 3, 1, 2\rangle$ is not a path (no edge from vertex 1 to vertex 2)

  The graph is not acyclic ($\langle 5, 6, 5\rangle$ is a cycle)

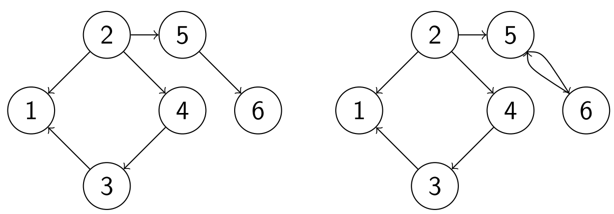
Of course, speaking about cycles is only meaningful in directed graphs, since undirected graphs can only be acyclic, iff they do not have edges at all. We will sometimes use the abbreviation DAG for directed acyclic graph.

Graph Depth

▷ Definition 5.19: Let $G := \langle V, E\rangle$ be a digraph, then the depth $dp(v)$ of a vertex $v \in V$ is defined to be 0, if v is a source of G and $sup\{len(p) \mid indeg(start(p)) = 0, end(p) = v\}$ otherwise, i.e. the length of the longest path from a source of G to v. (⚠ can be infinite)

▷ Definition 5.20: Given a digraph $G = \langle V, E\rangle$. The depth ($dp(G)$) of G is defined as $sup\{len(p) \mid p \in \Pi(G)\}$, i.e. the maximal path length in G.

▷ Example 5.21: The vertex 6 has depth two in the left grpahs and infine depth in the right one.



The left graph has depth three (cf. node 1), the right one has infinite depth (cf. nodes 5 and 6)

We now come to a very important special class of graphs, called trees.

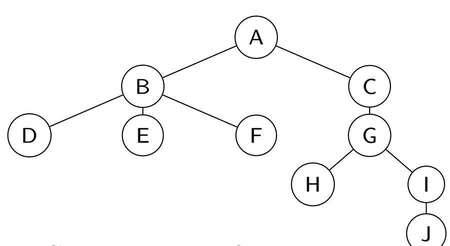
Trees

▷ Definition 5.22: A tree is a directed acyclic graph $G = \langle V, E\rangle$ such that

- ▷ There is exactly one initial node $v_r \in V$ (called the root)
- ▷ All nodes but the root have in-degree 1.

We call v the parent of w, iff $\langle v, w\rangle \in E$ (w is a child of v). We call a node v a leaf of G, iff it is terminal, i.e. if it does not have children.

▷ Example 5.23: A tree with root A and leaves D, E, F, H, and J.



F is a child of B and G is the parent of H and I.

▷ Lemma 5.24: *For any node $v \in V$ except the root v_r, there is exactly one path $p \in \Pi(G)$ with $start(p) = v_r$ and $end(p) = v$.* (proof by induction on the number of nodes)

In Computer Science trees are traditionally drawn upside-down with their root at the top, and the leaves at the bottom. The only reason for this is that (like in nature) trees grow from the root upwards and if we draw a tree it is convenient to start at the top of the page downwards, since we do not have to know the height of the picture in advance.

Let us now look at a prominent example of a tree: the parse tree of a Boolean expression. Intuitively, this is the tree given by the brackets in a Boolean expression. Whenever we have an expression of the form $\mathbf{A} \circ \mathbf{B}$, then we make a tree with root $\circ$ and two subtrees, which are constructed from $\mathbf{A}$ and $\mathbf{B}$ in the same manner.

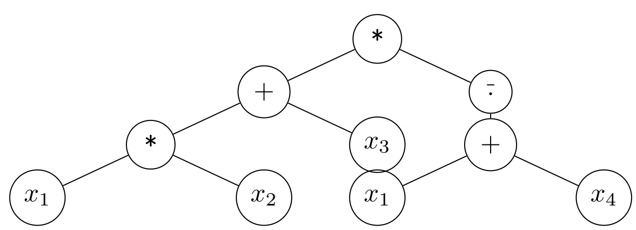
This allows us to view Boolean expressions as trees and apply all the mathematics (nomenclature and results) we will develop for them.

The Parse-Tree of a Boolean Expression

▷ Definition 5.25: The parse-tree P_e of a Boolean expression e is a labeled tree $P_e = \langle V_e, E_e, f_e \rangle$, which is recursively defined as

- ▹ if $e = \overline{e'}$ then $V_e := V_{e'} \cup \{v\}$, $E_e := E_{e'} \cup \{\langle v, {v'}_r \rangle\}$, and $f_e := f_{e'} \cup \{v \mapsto -\}$, where $P_{e'} = \langle V_{e'}, E_{e'} \rangle$ is the parse-tree of e', ${v'}_r$ is the root of $P_{e'}$, and v is an object not in $V_{e'}$.
- ▹ if $e = e_1 \circ e_2$ with $\circ \in \{*, +\}$ then $V_e := V_{e_1} \cup V_{e_2} \cup \{v\}$, $E_e := E_{e_1} \cup E_{e_2} \cup \{\langle v, v_1^r \rangle, \langle v, v_2^r \rangle\}$, and $f_e := f_{e_1} \cup f_{e_2} \cup \{v \mapsto \circ\}$, where the $P_{e_i} = \langle V_{e_i}, E_{e_i} \rangle$ are the parse-trees of e_i and v_i^r is the root of P_{e_i} and v is an object not in $V_{e_1} \cup V_{e_2}$.
- ▹ if $e \in V \cup C$ then, $V_e = \{e\}$ and $E_e = \emptyset$.

▷ Example 5.26: the parse tree of $(((x_1 * x_2) + x_3) * \overline{(x_1 + x_4)})$ is



©: Michael Kohlhase 23

6 Introduction to Combinatorial Circuits

We will now come to another model of computation: combinatorial circuits (also called combinational circuits). These are models of logic circuits (physical objects made of transistors (or cathode tubes) and wires, parts of integrated circuits, etc), which abstract from the inner structure for the switching elements (called gates) and the geometric configuration of the connections. Thus, combinatorial circuits allow us to concentrate on the functional properties of these circuits, without getting bogged down with e.g. configuration- or geometric considerations. These can be added to the models, but are not part of the discussion of this course.

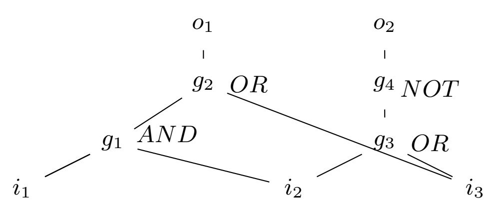
Combinatorial Circuits as Graphs

▷ **Definition 6.1:** A combinatorial circuit is a labeled acyclic graph $G = \langle V, E, f_g \rangle$ with label set $\{OR, AND, NOT\}$, such that

- ▷ $indeg(v) = 2$ and $outdeg(v) = 1$ for all nodes $v \in {f_g}^{-1}\{AND, OR\}$
- ▷ $indeg(v) = outdeg(v) = 1$ for all nodes $v \in {f_g}^{-1}\{NOT\}$

We call the set $I(G)$ ($O(G)$) of initial (terminal) nodes in G the input (output) vertexes, and the set $F(G) := (V \backslash (I(G) \cup O(G)))$ the set of gates.

▷ **Example 6.2:** The following graph $G_{cir1} = \langle V, E \rangle$ is a combinatorial circuit



▷ **Definition 6.3:** Add two special input nodes $0, 1$ to a combinatorial circuit G to form a combinatorial circuit with constants. (will use this from now on)

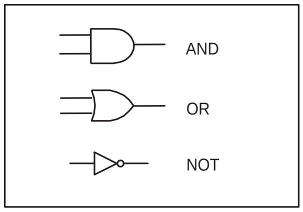
 24 JACOBS UNIVERSITY

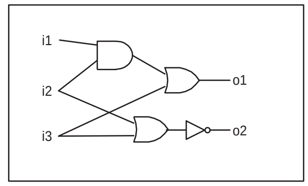
So combinatorial circuits are simply a class of specialized labeled directed graphs. As such, they inherit the nomenclature and equality conditions we introduced for graphs. The motivation for the restrictions is simple, we want to model computing devices based on gates, i.e. simple computational devices that behave like logical connectives: the AND gate has two input edges and one output edge; the the output edge has value 1, iff the two input edges do too.

Since combinatorial circuits are a primary tool for understanding logic circuits, they have their own traditional visual display format. Gates are drawn with special node shapes and edges are traditionally drawn on a rectangular grid, using bifurcating edges instead of multiple lines with blobs distinguishing bifurcations from edge crossings. This graph design is motivated by readability considerations (combinatorial circuits can become rather large in practice) and the layout of early printed circuits.

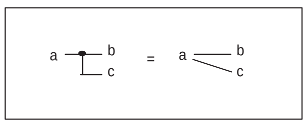
Using Special Symbols to Draw Combinatorial Circuits

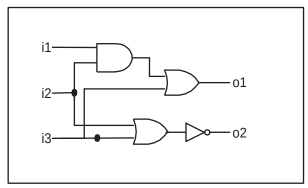
▷ The symbols for the logic gates AND, OR, and NOT.





▷ Junction Symbols as shorthands for several edges





 25 JACOBS UNIVERSITY

In particular, the diagram on the lower right is a visualization for the combinatory circuit G_{circ1} from the last slide.

To view combinatorial circuits as models of computation, we will have to make a connection between the gate structure and their input-output behavior more explicit. We will use a tool for this we have studied in detail before: Boolean expressions. The first thing we will do is to annotate all the edges in a combinatorial circuit with Boolean expressions that correspond to the values on the edges (as a function of the input values of the circuit).

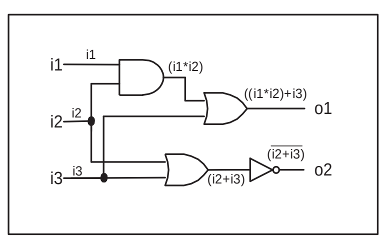
Computing with Combinatorial Circuits

▷ Combinatorial Circuits and parse trees for Boolean expressions look similar

▷ Idea: Let's annotate edges in combinatorial circuit with Boolean Expressions!

▷ Definition 6.4: Given a combinatorial circuit $G = \langle V, E, f_g\rangle$ and an edge $e = \langle v, w\rangle \in E$, the expression label $f_L((e))$ is defined as

$f_L(\langle v, w\rangle)$	if
v	$v \in I(G)$
$\overline{f_L(\langle u, v\rangle)}$	$f_g(v) = NOT$
$(f_L\langle u, v\rangle * f_L\langle u', v\rangle)$	$f_g(v) = AND$
$(f_L\langle u, v\rangle + f_L\langle u', v\rangle)$	$f_g(v) = OR$



 26 JACOBS UNIVERSITY

Armed with the expression label of edges we can now make the computational behavior of combinatory circuits explicit. The intuition is that a combinatorial circuit computes a certain Boolean function, if we interpret the input vertices as obtaining as values the corresponding arguments and passing them on to gates via the edges in the circuit. The gates then compute the result from their input edges and pass the result on to the next gate or an output vertex via their output edge.

Computing with Combinatorial Circuits

- Definition 6.5: A combinatorial circuit $G = \langle V, E, f_g\rangle$ with input vertices $i_1, \ldots, i_n$ and output vertices $o_1, \ldots, o_m$ computes an n-ary Boolean function

$$f\colon \{0,1\}^n \to \{0,1\}^m; \langle i_1, \ldots, i_n\rangle \mapsto \langle f_{e_1}(i_1, \ldots, i_n), \ldots, f_{e_m}(i_1, \ldots, i_n)\rangle$$

where $e_i = f_L(\langle v, o_i\rangle)$.

- Example 6.6: The circuit example on the last slide defines the Boolean function $f\colon \{0,1\}^3 \to \{0,1\}^2; \langle i_1, i_2, i_3\rangle \mapsto \langle f_{((i_1 * i_2)+i_3)}, f_{\overline{(i_2 * i_3)}}\rangle$
- Definition 6.7: The cost $C(G)$ of a circuit G is the number of gates in G.
- Problem: For a given boolean function f, find combinational circuits of minimal cost and depth that compute f.

 27 JACOBS UNIVERSITY

Note: The opposite problem, i.e., the conversion of a Boolean function into a combinatorial circuit, can be solved by determining the related expressions and their parse-trees. Note that there is a canonical graph-isomorphism between the parse-tree of an expression e and a combinatorial circuit that has an output that computes f_e.

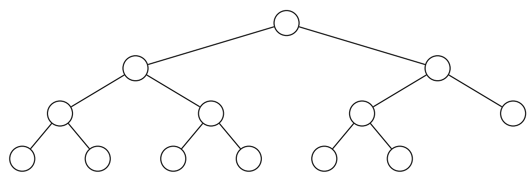
6.1 Preparing some Theory

The main properties of combinatory circuits we are interested in studying will be the the number of gates and the depth of a circuit. The number of gates is of practical importance, since it is a measure of the cost that is needed for producing the circuit in the physical world. The depth is interesting, since it is an approximation for the speed with which a combinatory circuit can compute: while in most physical realizations, signals can travel through wires at at (almost) the speed of light, gates have finite computation times.

Therefore we look at special configurations for combinatory circuits that have good depth and cost. These will become important, when we build actual combinatorial circuits with given input/output behavior.

Balanced Binary Trees

- Definition 6.8: A binary tree is a tree where all nodes have out-degree 2 or 0.
- Definition 6.9: A binary tree G is called balanced iff the depth of all leaves differs by at most by 1. (fully balanced, iff depth difference 0)
- Constructing a binary tree $G_{bbt} = \langle V, E\rangle$ with n leaves
  - step 1: select a $u \in V$ as root, $(V_1 := \{u\}, E_1 := \emptyset)$
  - step 2: select $v, w \in V$ not yet in V, $(V_i = V_{i-1} \cup \{v, w\})$
  - step 3: add two edges $\langle u, v\rangle$ and $\langle u, w\rangle$ where u is the leftmost of the shallowest nodes with $outdeg(u) = 0$, $(E_i := E_{i-1} \cup \{\langle u, v\rangle, \langle u, w\rangle\})$
  - repeat steps 2 and 3 until $i = n$ $(V = V_n, E = E_n)$
- Example 6.10: 7 leaves



We will now establish a few properties of these balanced binary trees that show that they are good building blocks for combinatory circuits.

Size Lemma for Balanced Trees

- Lemma 6.11: *Let $G = \langle V, E\rangle$ be a balanced binary tree of depth $n > i$, then the set $V_i := \{v \in V \mid dp(v) = i\}$ of vertexes at depth i has cardinality 2^i.*
- Proof: via induction over the depth i.

P.1 We have to consider two cases

P.1.1 $i = 0$: then $V_i = \{v_r\}$, where v_r is the root, so $\#V_0 = \{v_r\} = 1 = 2^0$.

P.1.2 $i > 0$: then V_{i-1} contains 2^{i-1} vertexes (IH)

P.1.2.2 By the definition of a binary tree, each $v \in V_{i-1}$ is a leaf or has two children that are at depth i.

P.1.2.3 as G is balanced and $dp(G) = n > i$, V_{i-1} cannot contain leaves

P.1.2.4 thus $\#V_i = 2 \cdot \#V_{i-1} = 2 \cdot 2^{i-1} = 2^i$

- Corollary 6.12: *A fully balanced tree of depth d has $2^{d+1} - 1$ nodes.*
- Proof:

  Let $G := \langle V, E\rangle$ be a fully balanced tree, $\#V = \sum_{i=1}^{d} 2^i = 2^{d+1} - 1$. □

This shows that balanced binary trees grow in breadth very quickly, a consequence of this is that they are very shallow (and this compute very fast), which is the essence of the next result.

Depth Lemma for Balanced Trees

▷ Lemma 6.13: *Let $G = \langle V, E\rangle$ be a balanced binary tree, then $dp(G) = \lfloor \log_2 (\#V)\rfloor$.*

▷ Proof: by calculation

P.1 Let $V' := (V\backslash W)$, where W is the set of nodes at level $d = dp(G)$

P.2 By the size lemma, $\#V' = 2^{d-1+1} - 1 = 2^d - 1$

P.3 then $\#V = 2^d - 1 + k$, where $k = \#W$, $1 \leq k \leq 2^d$

P.4 so $\#V = c \cdot 2^d$ where $c \in \mathbb{R}$ and $1 \leq c < 2$, or $0 \leq \log_2 c < 1$

P.5 thus $\log_2 \#V = \log_2 c \cdot 2^d = \log_2 c + d$ and

P.6 hence $d = \log_2 \#V - \log_2 c = \lfloor \log_2 (\#V)\rfloor$. □

Leaves of Binary Trees

▷ Lemma 6.14: *Any binary tree with m leaves has $2m - 1$ vertexes.*

▷ Proof: by induction on m.

P.1 We have two cases $m = 1$: then $V = \{v_r\}$ and $\#V = 1 = 2 \cdot 1 - 1$.

P.1.2 $m > 1$:

P.1.2.1 then any binary tree G with $m - 1$ leaves has $2m - 3$ vertexes (IH)

P.1.2.2 To get m leaves, add 2 children to some leaf of G. (add two to get one more)

P.1.2.3 Thus $\#V = 2m - 3 + 2 = 2m - 1$. □

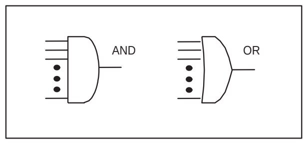
□

In particular, the size of a binary tree is independent of the its form if we fix the number of leaves. So we can optimimze the depth of a binary tree by taking a balanced one without a size penalty. This will become important for building fast combinatory circuits.

We now use the results on balanced binary trees to build generalized gates as building blocks for combinational circuits.

n-ary Gates as Subgraphs

- ▷ Idea: Identify (and abbreviate) frequently occurring subgraphs
- ▷ Definition 6.15: $(AND x_1, \ldots, x_n) := \prod_{i=1}^{n} x_i$ and $(OR x_1, \ldots, x_n) := \sum_{i=1}^{n} x_i$
- ▷ Note: These can be realized as balanced binary trees G_n
- ▷ Corollary 6.16: $C(G_n) = n - 1$ *and* $dp(G_n) = \lfloor \log_2(n) \rfloor$.
- ▷ Notation 6.17:



Using these building blocks, we can establish a worst-case result for the depth of a combinatory circuit computing a given Boolean function.

Worst Case Depth Theorem for Combinatorial Circuits

- ▷ Theorem 6.18: *The worst case depth $dp(G)$ of a combinatorial circuit G which realizes an $k \times n$-dimensional boolean function is bounded by $dp(G) \leq n + \lceil \log_2(n) \rceil + 1$.*
- ▷ Proof: The main trick behind this bound is that AND and OR are associative and that the according gates can be arranged in a balanced binary tree.

  P.1 Function f corresponding to the output o_j of the circuit G can be transformed in DNF

  P.2 each monomial consists of at most n literals

  P.3 the possible negation of inputs for some literals can be done in depth 1

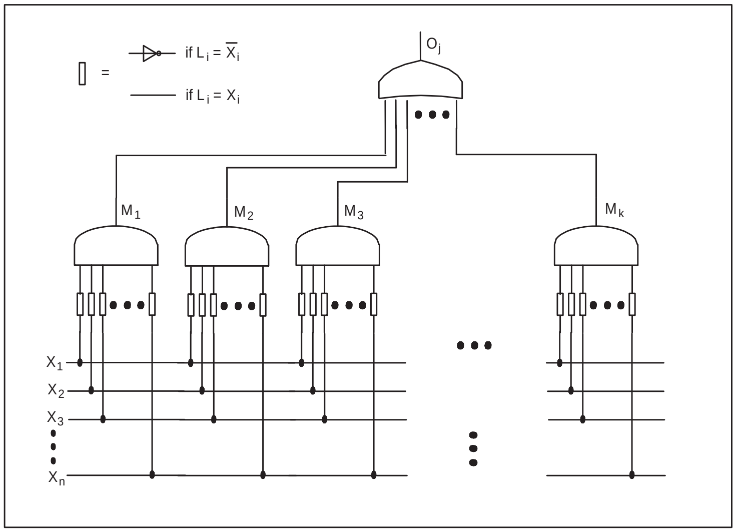
  P.4 for each monomial the ANDs in the related circuit can be arranged in a balanced binary tree of depth $\lceil \log_2(n) \rceil$

  P.5 there are at most 2^n monomials which can be ORed together in a balanced binary tree of depth $\lceil \log_2(2^n) \rceil = n$. □

Of course depth result is related to the first worst-case complexity result for Boolean expressions[2]; it uses the same idea: to use the disjunctive normal form of the Boolean function. However, instead of using a Boolean expression, we become more concrete here and use a combinatorial circuit. EdNote(2)

[2] EdNote: how to do assertion references?

An example of a DNF circuit



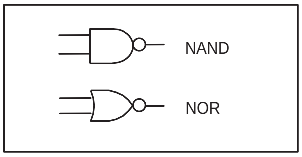
In the circuit diagram above, we have of course drawn a very particular case (as an example for possible others.) One thing that might be confusing is that it looks as if the lower n-ary conjunction operators look as if they have edges to all the input variables, which a DNF does not have in general.

Of course, by now, we know how to do better in practice. Instead of the DNF, we can always compute the minimal polynomial for a given Boolean function using the Quine-McCluskey algorithm and derive a combinatorial circuit from this. While this does not give us any theoretical mileage (there are Boolean functions where the DNF is already the minimal polynomial), but will greatly improve the cost in practice.

Until now, we have somewhat arbitrarily concentrated on combinational circuits with AND, OR, and NOT gates. The reason for this was that we had already developed a theory of Boolean expressions with the connectives $\vee$, $\wedge$, and $\neg$ that we can use. In practical circuits often other gates are used, since they are simpler to manufacture and more uniform. In particular, it is sufficient to use only one type of gate as we will see now.

Other Logical Connectives and Gates

- Are the gates AND, OR, and NOT ideal?
- Idea: Combine NOT with the binary ones to $NAND$, NOR (enough?)

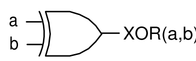


$NAND$	0	1
0	1	1
1	1	0

NOR	0	1
0	1	0
1	0	0

- Corresponding logical conectives are written as $\uparrow$ ($NAND$) and $\downarrow$ (NOR).
- We will also need .

XOR	0	1
0	0	1
1	1	0

- The gate is written as 

, the logical connective as $\oplus$.

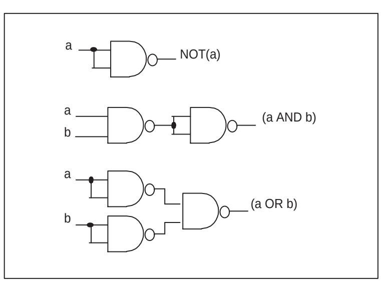
The Universality of $NAND$ and NOR

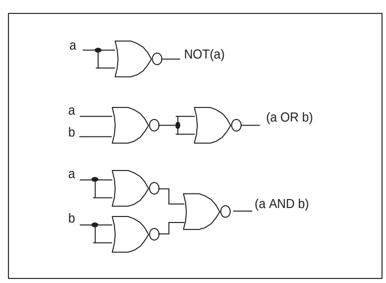
- Theorem 6.19: *$NAND$ and NOR are universal; i.e. any Boolean function can be expressed in terms of them.*
- Proof: express AND, OR, and NOT via $NAND$ and NOR respectively

$NOT(a)$	$NAND(a,a)$	$NOR(a,a)$
$AND(a,b)$	$NAND(NAND(a,b),NAND(a,b))$	$NOR(NOR(a,a),NOR(b,b))$
$OR(a,b)$	$NAND(NAND(a,a),NAND(b,b))$	$NOR(NOR(a,b),NOR(a,b))$

□

- here are the corresponding diagrams for the combinational circuits.





Of course, a simple substitution along these lines will blow up the cost of the circuits by a factor of up to three and double the depth, which would be prohibitive. To get around this, we would have to develop a theory of Boolean expressions and complexity using the $NAND$ and NOR connectives, along with suitable replacements for the Quine-McCluskey algorithm. This would give cost and depth results comparable to the ones developed here. This is beyond the scope of this course.

7 Basic Arithmetics with Combinational Circuits

We have seen that combinational circuits are good models for implementing Boolean functions: they allow us to make predictions about properties like costs and depths (computation speed), while abstracting from other properties like geometrical realization, etc.

We will now extend the analysis to circuits that can compute with numbers, i.e. that implement the basic arithmetical operations (addition, multiplication, subtraction, and division on integers). To be able to do this, we need to interpret sequences of bits as integers. So before we jump into arithmetical circuits, we will have a look at number representations.

Positional Number Systems

- Problem: For realistic arithmetics we need better number representations than the unary natural numbers ($|n_{unary}| \in \Theta(n)$ [number of /])
- Recap: the unary number system
  - build up numbers from /es (start with ' ' and add /)
  - addition $\oplus$ as concatenation ($\odot, \oplus 1n, exp, \ldots$ defined from that)

  Idea: build a clever code on the unary numbers
- interpret sequences of /es as strings: ϵ stands for the number 0
- Definition 7.1: A positional number system $\mathcal{N}$ is a triple $\mathcal{N} = \langle D_b, \varphi_b, \psi_b \rangle$ with
  - D_b is a finite alphabet of b so-called digits.($b := \#D_b$ base or radix of $\mathcal{N}$)
  - $\varphi_b \colon D_b \to \{\epsilon, /, \ldots, /^{[b-1]}\}$ is bijective (first b unary numbers)
  - $\psi_b \colon {D_b}^+ \to \{/\}^*; \langle n_k, \ldots, n_1 \rangle \mapsto \bigoplus_{i=1}^{k} (\varphi_b(n_i) \odot exp(/^{[b]}, /^{[i-1]}))$ (extends φ_b to string code)

 37 JACOBS UNIVERSITY

In the unary number system, it was rather simple to do arithmetics, the most important operation (addition) was very simple, it was just concatenation. From this we can implement the other operations by simple recursive procedures, e.g. in SML or as abstract procedures in abstract data types. To make the arguments more transparent, we will use special symbols for the arithmetic operations on unary natural numbers: $\oplus$ (addition), $\odot$ (multiplication), $\bigoplus_1^n ()$ (sum over n numbers), and $\bigodot_1^n ()$ (product over n numbers).

The problem with the unary number system is that it uses enormous amounts of space, when writing down large numbers. Using the Landau notation we introduced earlier, we see that for writing down a number n in unary representation we need n slashes. So if $|n_{unary}|$ is the "cost of representing n in unary representation", we get $|n_{unary}| \in \Theta(n)$. Of course that will never do for practical chips. We obviously need a better encoding.

If we look at the unary number system from a greater distance (now that we know more CS, we can interpret the representations as strings), we see that we are not using a very important feature of strings here: position. As we only have one letter in our alphabet (/), we cannot, so we should use a larger alphabet. The main idea behind a positional number system $\mathcal{N} = \langle D_b, \varphi_b, \psi_b \rangle$ is that we encode numbers as strings of digits (characters in the alphabet D_b), such that the position matters, and to give these encoding a meaning by mapping them into the unary natural numbers via a mapping ψ_b. This is the the same process we did for the logics; we are now doing it for number systems. However, here, we also want to ensure that the meaning mapping ψ_b is a bijection, since we want to define the arithmetics on the encodings by reference to The arithmetical operators on the unary natural numbers.

We can look at this as a bootstrapping process, where the unary natural numbers constitute the seed system we build up everything from.

Just like we did for string codes earlier, we build up the meaning mapping ψ_b on characters from D_b first. To have a chance to make ψ bijective, we insist that the "character code" φ_b is is a bijection from D_b and the first b unary natural numbers. Now we extend φ_b from a character code to a string code, however unlike earlier, we do not use simple concatenation to induce the string code, but a much more complicated function based on the arithmetic operations on unary natural numbers. We will see later[3] that this give us a bijection between ${D_b}^+$ and the unary natural numbers. EdNote(3)

Commonly Used Positional Number Systems

▷ Example 7.2: The following positional number systems are in common use.

name	set	base	digits	example
unary	$\mathbb{N}_1$	1	/	1
binary	$\mathbb{N}_2$	2	0,1	0101000111_2
octal	$\mathbb{N}_8$	8	0,1,...,7	63027_8
decimal	$\mathbb{N}_{10}$	10	0,1,...,9	162098_{10} or 162098
hexadecimal	$\mathbb{N}_{16}$	16	0,1,...,9,A,...,F	$FF3A12_{16}$

▷ Notation 7.3:attach the base of $\mathcal{N}$ to every number from $\mathcal{N}$.(default: decimal)

Trick: Group triples or quadruples of binary digits into recognizable chunks (add leading zeros as needed)

▷ ▷ $110001101011100_2 = \underbrace{0110_2}_{6_{16}}\underbrace{0011_2}_{3_{16}}\underbrace{0101_2}_{5_{16}}\underbrace{1100_2}_{C_{16}} = 635C_{16}$

▷ $110001101011100_2 = \underbrace{110_2}_{6_8}\underbrace{001_2}_{1_8}\underbrace{101_2}_{5_8}\underbrace{011_2}_{3_8}\underbrace{100_2}_{4_8} = 61534_8$

▷ $F3A_{16} = \underbrace{F_{16}}_{1111_2}\underbrace{3_{16}}_{0011_2}\underbrace{A_{16}}_{1010_2} = 111100111010_2$, $4721_8 = \underbrace{4_8}_{100_2}\underbrace{7_8}_{111_2}\underbrace{2_8}_{010_2}\underbrace{1_8}_{001_2} = 100111010001_2$

We have all seen positional number systems: our decimal system is one (for the base 10). Other systems that important for us are the binary system (it is the smallest non-degenerate one) and the octal- (base 8) and hexadecimal- (base 16) systems. These come from the fact that binary numbers are very hard for humans to scan. Therefore it became customary to group three or four digits together and introduce we (compound) digits for them. The octal system is mostly relevant for historic reasons, the hexadecimal system is in widespread use as syntactic sugar for binary numbers, which form the basis for circuits, since binary digits can be represented physically by current/no current.

Now that we have defined positional number systems, we want to define the arithmetic operations on the these number representations. We do this by using an old trick in math. If we have an operation $f_T : T \to T$ on a set T and a well-behaved mapping ψ from a set S into T, then we can "pull-back" the operation on f_T to S by defining the operation $f_S : S \to S$ by $f_S(s) := \psi^{-1}(f_T(\psi(s)))$ according to the following diagram.

[3] EdNote: reference

$$\begin{tikzcd} S \arrow[r, bend left, "\psi"] & T \arrow[l, "\psi^{-1}"] \\ S \arrow[u, "f_S = \psi^{-1} \circ f_T \circ \psi"'] \arrow[r, "\psi"] & T \arrow[u, "f_T"'] \end{tikzcd}$$

Obviously, this construction can be done in any case, where ψ is bijective (and thus has an inverse function). For defining the arithmetic operations on the positional number representations, we do the same construction, but for binary functions (after we have established that ψ is indeed a bijection).

The fact that ψ_b is a bijection a posteriori justifies our notation, where we have only indicated the base of the positional number system. Indeed any two positional number systems are isomorphic: they have bijections ψ_b into the unary natural numbers, and therefore there is a bijection between them.

Arithmetics for PNS

- ▷ Lemma 7.4: *Let $\mathcal{N} := \langle D_b, \varphi_b, \psi_b \rangle$ be a PNS, then ψ_b is bijective.*
- ▷ Proof: construct ${\psi_b}^{-1}$ by successive division modulo the base of $\mathcal{N}$. □

  Idea: use this to define arithmetics on $\mathcal{N}$.
- ⊳ Definition 7.5: Let $\mathcal{N} := \langle D_b, \varphi_b, \psi_b \rangle$ be a PNS of base b, then we define a binary function $+_b\colon \mathbb{N}_b \times \mathbb{N}_b \to \mathbb{N}_b$ by $(x +_b y) := {\psi_b}^{-1}(\psi_b(x) \oplus \psi_b(y))$.
- ▷ Note: The addition rules (carry chain addition) generalize from the decimal system to general PNS
- ▷ Idea: Do the same for other arithmetic operations. (works like a charm)
- ▷ Future: Concentrate on binary arithmetics. (implement into circuits)

 39 JACOBS UNIVERSITY

The next step is now to implement the induced arithmetical operations into combinational circuits, starting with addition. Before we can do this, we have to specify which (Boolean) function we really want to implement. For convenience, we will use the usual decimal (base 10) representations of numbers and their operations to argue about these circuits. So we need conversion functions from decimal numbers to binary numbers to get back and forth. Fortunately, these are easy to come by, since we use the bijections ψ from both systems into the unary natural numbers, which we can compose to get the transformations.

Arithmetic Circuits for Binary Numbers

- Idea: Use combinational circuits to do basic arithmetics.
- Definition 7.6: Given the (abstract) number $a \in \mathbb{N}$, $B(a)$ denotes from now on the binary representation of a.

  For the opposite case, i.e., the natural number represented by a binary string $a = \langle a_{n-1}, \ldots, a_0 \rangle \in \mathbb{B}^n$, the notation $\langle\langle a \rangle\rangle$ is used, i.e.,

$$\langle\langle a \rangle\rangle = \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

- Definition 7.7: An n-bit adder is a circuit computing the function $f_{+2}^n \colon \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^{n+1}$ with

$$f_{+2}^n(a; b) := B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle)$$

 40 JACOBS UNIVERSITY

If we look at the definition again, we see that we are again using a pull-back construction. These will pop up all over the place, since they make life quite easy and safe.

Before we actually get a combinational circuit for an n-bit adder, we will build a very useful circuit as a building block: the half adder (so-called, since it will take two to build a full adder).

The Half-Adder

- There are different ways to implement an adder. All of them build upon two basic components, the half-adder and the full-adder.

  Definition 7.8: A half adder is a circuit HA implementing the function f_{HA} in the truth table on the right.

$$f_{HA} \colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}^2; \langle a, b \rangle \mapsto \langle c, s \rangle$$

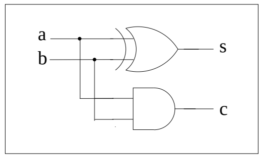
  s is called the sum bit and c the carry bit.

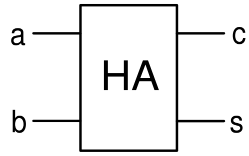
a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Note: The carry can be computed by a simple AND, i.e., $c = AND(a, b)$, and the sum bit by a XOR function.

 41 JACOBS UNIVERSITY

Building and Evaluating the Half-Adder

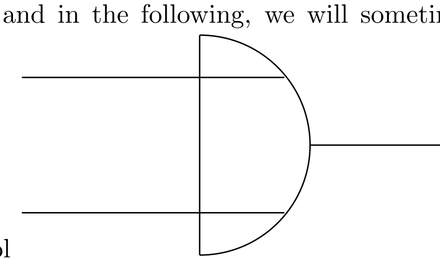




- ▷ So, the half-adder corresponds to the Boolean function $f_{HA} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}^2; \langle a, b \rangle \mapsto \langle (a \oplus b), (a \wedge b) \rangle$
- ▷ Note: $HAa, b = B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle)$, i.e., it is indeed an adder.
- ▷ We count XOR as one gate, so $C(HA) = 2$ and $dp(HA) = 1$.

Now that we have the half adder as a building block it is rather simple to arrive at a full adder circuit.

, in the diagram for the full adder, and in the following, we will sometimes use a variant gate symbol for the OR gate: The symbol



. It has the same outline as an AND gate, but the input lines go all the way through.

The Full Adder

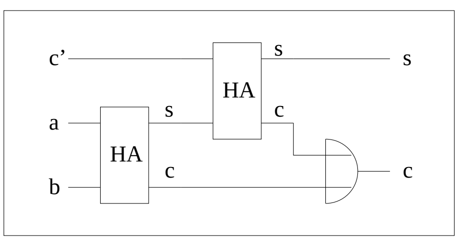
▷ Definition 7.9: The 1-bit full adder is a circuit FA^1 that implements the function $f^1_{FA}\colon \mathbb{B}\times\mathbb{B}\times\mathbb{B}\to\mathbb{B}^2$ with $f^1_{FA}(a,b,c') = B(\langle\!\langle a\rangle\!\rangle + \langle\!\langle b\rangle\!\rangle + \langle\!\langle c'\rangle\!\rangle)$

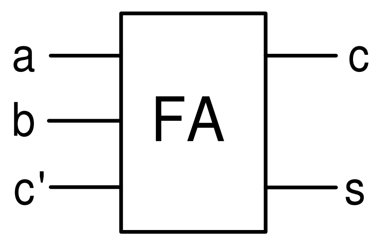
▷ The result of the full-adder is also denoted with $\langle c,s\rangle$, i.e., a carry and a sum bit. .

▷ the easiest way to implement a full adder is to use two half adders and an OR gate.

▷ Lemma 7.10: (Cost and Depth)
$C(FA^1) = 2C(HA)+1 = 5$ *and* $dp(FA^1) = 2dp(HA)+1 = 3$

a	b	c'	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





Note: Note that in the right hand graphics, we use another notation for the OR gate.[4] EdNote(4)

Of course adding single digits is a rather simple task, and hardly worth the effort, if this is all we can do. What we are really after, are circuits that will add n-bit binary natural numbers, so that we arrive at computer chips that can add long numbers for us.

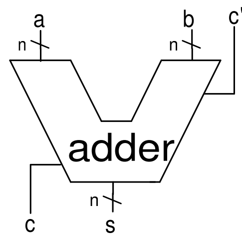
[4] EdNote: Todo: introduce this earlier, or change the graphics here (or both)

Full n-bit Adder

▷ Definition 7.11: An n-bit full adder ($n > 1$) is a circuit that corresponds to $f_{FA}^n \colon \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^n; \langle a, b, c' \rangle = B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + \langle\langle c' \rangle\rangle)$

▷ Notation 7.12:We will draw the n-bit full adder with the following symbol in circuit diagrams.

Note that we are abbreviating n-bit input and output edges with a single one that has a slash and the number n next to it.



▷ There are various implementations of the full n-bit adder, we will look at two of them

 44

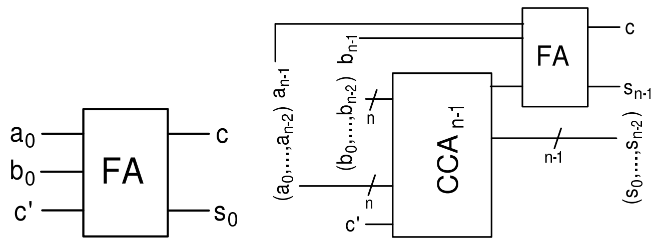
This implementation follows the intuition behind elementary school addition (only for binary numbers): we write the numbers below each other in a tabulated fashion, and from the least significant digit, we follow the process of

- adding the two digits with carry from the previous column
- recording the sum bit as the result, and
- passing the carry bit on to the next column

until one of the numbers ends.

The Carry Chain Adder

▷ The inductively designed circuit of the carry chain adder.



- ▷ $n = 1$: the CCA^1 consists of a full adder
- ▷ $n > 1$: the CCA^n consists of an $(n-1)$-bit carry chain adder CCA^{n-1} and a full adder that sums up the carry of CCA^{n-1}, the last bit of a and the last bit of b

▷ Definition 7.13: An n-bit carry chain adder CCA^n is inductively defined as

- ▷ $f^1_{CCA}(a_0, b_0, c) = f^1_{FA}(a_0, b_0, c)$
- ▷ $f^n_{CCA}(\langle a_{n-1}, \ldots, a_0\rangle, \langle b_{n-1}, \ldots, b_0\rangle, c') = \langle c, s_{n-1}, \ldots, s_0\rangle$ with
  - ▷ $\langle c, s_{n-1}\rangle = f^{n-1}_{FA}(a_{n-1}, b_{n-1}, c_{n-1})$
  - ▷ $\langle c_{n-1}, s_{n-2}, \ldots, s_0\rangle = f^{n-1}_{CCA}(\langle a_{n-2}, \ldots, a_0\rangle, \langle b_{n-2}, \ldots, b_0\rangle, c')$

The Carry Chain Adder

▷ (Cost)
$C(CCA^n) = C(CCA^{n-1}) + C(FA^1) = C(CCA^{n-1}) + 5 = 5n = O(n)$

▷ Lemma 7.14: (Depth)
$dp(CCA^n) = dp(CCA^{n-1}) + dp(FA^1) = dp(CCA^{n-1}) + 3 = 3n = O(n)$

▷ The carry chain adder is simple, but cost and depth are high. (depth is critical (speed))

▷ Question: Can we do better?

▷ Problem: the carry ripples up the chain (upper parts wait for carries from lower part)

A consequence of using the carry chain adder is that if we go from a 32-bit architecture to a 64-bit architecture, the speed of additions in the chips would not increase, but decrease (by 50%). Of course, we can carry out 64-bit additions now, a task that would have needed a special routine at the software level (these typically involve at least 4 32-bit additions so there is a speedup for such additions), but most addition problems in practice involve small (under 32-bit) numbers, so we will have an overall performance loss (not what we really want for all that cost).

If we want to do better in terms of depth of an n-bit adder, we have to break the dependency on the carry, let us look at a decimal addition example to get the idea. Consider the following snapshot of an carry chain addition

first summand		3	4	7	9	8	3	4	7	9	2
second summand		$2_?$	$5_?$	$1_?$	$8_?$	$1_?$	$7_?$	8_1	7_1	2_0	1_0
partial sum	?	?	?	?	?	?	?	?	5	1	3

We have already computed the first three partial sums. Carry chain addition would simply go on and ripple the carry information through until the left end is reached (after all what can we do? we need the carry information to carry out left partial sums). Now, if we only knew what the carry would be e.g. at column 5, then we could start a partial summation chain there as well.

The central idea in the so-called "*conditional sum adder*" we will pursue now, is to trade time for space, and just compute both cases (with and without carry), and then later choose which one was the correct one, and discard the other. We can visualize this in the following schema.

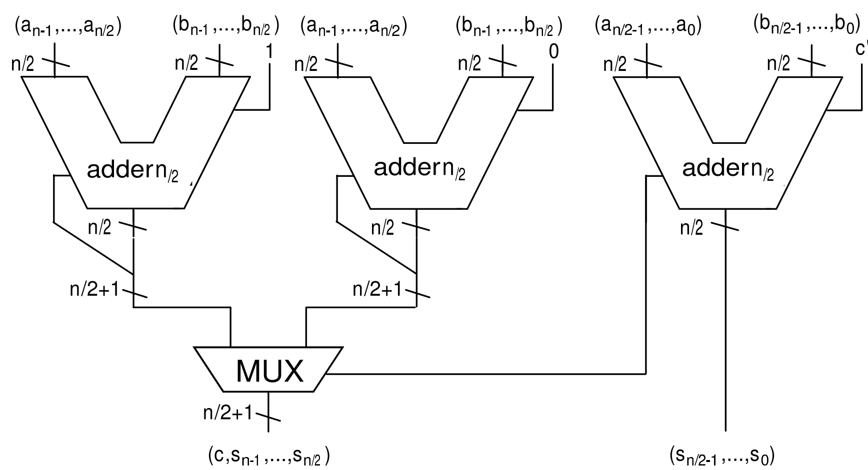
first summand		3	4	7	9	8	3	4	7	9	2
second summand		$2_?$	5_0	1_1	$8_?$	$1_?$	$7_?$	8_1	7_1	2_0	1_0
lower sum							?	?	5	1	3
upper sum. with carry	?	?	?	9	8	0					
upper sum. no carry	?	?	?	9	7	9					

Here we start at column 10 to compute the lower sum, and at column 6 to compute two upper sums, one with carry, and one without. Once we have fully computed the lower sum, we will know about the carry in column 6, so we can simply choose which upper sum was the correct one and combine lower and upper sum to the result.

Obviously, if we can compute the three sums in parallel, then we are done in only five steps not ten as above. Of course, this idea can be iterated: the upper and lower sums need not be computed by carry chain addition, but can be computed by conditional sum adders as well.

The Conditional Sum Adder

- ▷ Idea: pre-compute both possible upper sums (e.g. upper half) for carries 0 and 1, then choose (via MUX) the right one according to lower sum.
- ▷ the inductive definition of the circuit of a conditional sum adder (CSA).



 47

The Conditional Sum Adder

▷ Definition 7.15: An n-bit conditional sum adder CSA^n is recursively defined as

▷ $f_{CSA}^n(\langle a_{n-1}, \ldots, a_0\rangle, \langle b_{n-1}, \ldots, b_0\rangle, c') = \langle c, s_{n-1}, \ldots, s_0\rangle$ where

▷ $\langle c_{n/2}, s_{n/2-1}, \ldots, s_0\rangle = f_{CSA}^{n/2}(\langle a_{n/2-1}, \ldots, a_0\rangle, \langle b_{n/2-1}, \ldots, b_0\rangle, c')$

▷ $\langle c, s_{n-1}, \ldots, s_{n/2}\rangle = \begin{cases} f_{CSA}^{n/2}(\langle a_{n-1}, \ldots, a_{n/2}\rangle, \langle b_{n-1}, \ldots, b_{n/2}\rangle, 0) \text{ iff } c_{n/2} = 0 \\ f_{CSA}^{n/2}(\langle a_{n-1}, \ldots, a_{n/2}\rangle, \langle b_{n-1}, \ldots, b_{n/2}\rangle, 1) \text{ iff } c_{n/2} = 1 \end{cases}$

▷ $f_{CSA}^1(a_0, b_0, c) = f_{FA}^1(a_0, b_0, c)$

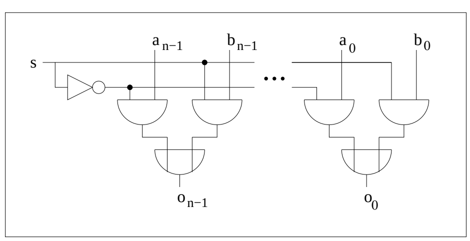
The only circuit that we still have to look at is the one that chooses the correct upper sums. Fortunately, this is a rather simple design that makes use of the classical trick that "if C, then A, else B" can be expressed as "(C and A) or ($\neg C$ and B)".

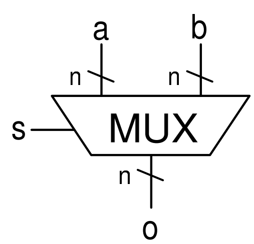
The Multiplexer

▷ Definition 7.16: An n-bit multiplexer MUX^n is a circuit which implements the function $f_{MUX}^n \colon \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B}^n$ with

$$f(a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0, s) = \begin{cases} a_{n-1}, \ldots, a_0 \; if \; s = 0 \\ b_{n-1}, \ldots, b_0 \; if \; s = 1 \end{cases}$$

▷ Idea: A multiplexer chooses between two n-bit input vectors A and B depending on the value of the control bit s.





▷ Cost and depth: $C(MUX^n) = 3n + 1$ and $dp(MUX^n) = 3$.

Now that we have completely implemented the conditional lookahead adder circuit, we can analyze it for its cost and depth (to see whether we have really made things better with this design). Analyzing the depth is rather simple, we only have to solve the recursive equation that combines the recursive call of the adder with the multiplexer. Conveniently, the 1-bit full adder has the same depth as the multiplexer.

The Depth of CSA

- (Obviously)
  $dp(CSA^n) = dp(CSA^{n/2}) + dp(MUX^{n/2+1})$
- solve the recursive equation:

$$\begin{aligned} dp(CSA^n) &= dp(CSA^{n/2}) + dp(MUX^{n/2+1}) \\ &= dp(CSA^{n/2}) + 3 \\ &= dp(CSA^{n/4}) + 3 + 3 \\ &= dp(CSA^{n/8}) + 3 + 3 + 3 \\ & \quad \ldots \\ &= dp(CSA^{n2^{-i}}) + 3i \\ &= dp(CSA^1) + 3\log_2 n \\ &= 3\log_2 n + 3 \end{aligned}$$

 50

The analysis for the cost is much more complex, we also have to solve a recursive equation, but a more difficult one. Instead of just guessing the correct closed form, we will use the opportunity to show a more general technique: using Master's theorem for recursive equations. There are many similar theorems which can be used in situations like these, going into them or proving Master's theorem would be beyond the scope of the course.

The Cost of CSA

- (Obviously)
  $C(CSA^n) = 3C(CSA^{n/2}) + C(MUX^{n/2+1})$.
- Problem: How to solve this recursive equation?
- Solution: Guess a closed formula, prove by induction. (if we are lucky)
- Solution2: Use a general tool for solving recursive equations.
- Theorem 7.17: (Master's Theorem for Recursive Equations)
  Given the recursively defined function $f\colon \mathbb{N} \to \mathbb{R}$, such that $f(1) = c \in \mathbb{R}$ and $f(b^k) = af(b^{k-1}) + g(b^k)$ for some $1 \leq a \in \mathbb{R}$, $k \in \mathbb{N}$, and $g\colon \mathbb{N} \to \mathbb{R}$, then $f(b^k) = ca^k + \sum_{i=0}^{k-1} a^i g(b^{k-i})$
- We have $C(CSA^n) = 3C(CSA^{n/2}) + C(MUX^{n/2+1}) = 3C(CSA^{n/2}) + 3(n/2+1) + 1 = 3C(CSA^{n/2}) + \frac{3}{2}n + 4$
- So, $C(CSA^n)$ is a function that can be handled via Master's theorem with $a = 3$, $b = 2$, $n = b^k$, $g(n) = 3/2n + 4$, and $c = C(f^1_{CSA}) = C(FA^1) = 5$

 51

The Cost of CSA

- thus $C(CSA^n) = 5 \cdot 3^{\log_2 n} + \sum_{i=0}^{\log_2 n-1} (3^i \cdot \frac{3}{2}n \cdot 2^{-i} + 4)$
- Note: $a^{\log_2 n} = 2^{\log_2 a^{\log_2 n}} = 2^{\log_2 a \cdot \log_2 n} = 2^{\log_2 n^{\log_2 a}} = n^{\log_2 a}$

$$\begin{aligned}
C(CSA^n) &= 5 \cdot 3^{\log_2 n} + \sum_{i=0}^{\log_2 n-1} (3^i \cdot \tfrac{3}{2}n \cdot 2^{-i} + 4) \\
&= 5n^{\log_2 3} + \sum_{i=1}^{\log_2 n} n\tfrac{3}{2}^i n + 4 \\
&= 5n^{\log_2 3} + n \cdot \sum_{i=1}^{\log_2 n} \tfrac{3}{2}^i + 4\log_2 n \\
&= 5n^{\log_2 3} + 2n \cdot (\tfrac{3}{2}^{\log_2 n+1} - 1) + 4\log_2 n \\
&= 5n^{\log_2 3} + 3n \cdot n^{\log_2 \frac{3}{2}} - 2n + 4\log_2 n \\
&= 8n^{\log_2 3} - 2n + 4\log_2 n \in O(n^{\log_2 3})
\end{aligned}$$

The Cost of CSA

- Theorem 7.18: *The cost and the depth of the conditional sum adder are in the following complexity classes:*

$$C(CSA^n) \in O(n^{\log_2 3}) \qquad dp(CSA^n) \in O(\log_2 n)$$

- Compare with: $C(CCA^n) \in O(n) \qquad dp(CCA^n) \in O(n)$
- So, the conditional sum adder has a smaller depth than the carry chain adder. This smaller depth is paid with higher cost.
- There is another adder that combines the small cost of the carry chain adder with the low depth of the conditional sum adder. This has a cost $C(CLA^n) \in O(n)$ and a depth of $dp(CLA^n) \in O(\log_2 n)$.

Instead of perfecting the n-bit adder further (and there are lots of designs and optimizations out there, since this has high commercial relevance), we will extend the range of arithmetic operations. The next thing we come to is subtraction.

8 Arithmetics for Two's Complement Numbers

This of course presents us with a problem directly: the n-bit binary natural numbers, we have used for representing numbers are closed under addition, but not under subtraction: If we have two n-bit binary numbers $B(n)$, and $B(m)$, then $B(n+m)$ is an $n+1$-bit binary natural number. If we count the most significant bit separately as the carry bit, then we have a n-bit result. For subtraction this is not the case: $B(n-m)$ is only a n-bit binary natural number, if $m \geq n$ (whatever we do with the carry). So we have to think about representing negative binary natural numbers first. It turns out that the solution using sign bits that immediately comes to mind is not the best one.

Negative Numbers and Subtraction

- Note: So far we have completely ignored the existence of negative numbers.
- Problem: Subtraction is a partial operation without them.
- Question: Can we extend the binary number systems for negative numbers?
- Simple Solution: Use a
- Definition 8.1: ($(n+1)$-bit signed binary number system)

$$\langle\langle a_n, a_{n-1}, \ldots, a_0 \rangle\rangle^- := \begin{cases} \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \text{ if } a_n = 0 \\ -\langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \text{ if } a_n = 1 \end{cases}$$

- Note: We need to fix string length to identify the sign bit. (leading zeroes)
- Example 8.2: In the 8-bit signed binary number system
  - 10011001 represents -25 ($\langle\langle 10011001 \rangle\rangle^- = -(2^4 + 2^3 + 2^0)$)
  - 00101100 corresponds to a positive number: 44

Here we did the naive solution, just as in the decimal system, we just added a sign bit, which specifies the polarity of the number representation. The first consequence of this that we have to keep in mind is that we have to fix the width of the representation: Unlike the representation for binary natural numbers which can be arbitrarily extended to the left, we have to know which bit is the sign bit. This is not a big problem in the world of combinational circuits, since we have a fixed width of input/output edges anyway.

Problems of Sign-Bit Systems

- Generally: An n-bit signed binary number system allows to represent the integers from $-2^{n-1}+1$ to $+2^{n-1}-1$.
- $2^{n-1}-1$ positive numbers, $2^{n-1}-1$ negative numbers, and the zero
- Thus we represent $\#\{\langle\!\langle s\rangle\!\rangle^{-} \mid s \in \mathbb{B}^n\} = 2\cdot(2^{n-1}-1)+1 = 2^n-1$ numbers all in all
- One number must be represented twice (But there are 2^n strings of length n.)
- $10\ldots0$ and $00\ldots0$ both represent the zero as $-1\cdot 0 = 1\cdot 0$.

signed binary				$\mathbb{Z}$
0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0
1	0	0	0	-0
1	0	0	1	-1
1	0	1	0	-2
1	0	1	1	-3
1	1	0	0	-4
1	1	0	1	-5
1	1	1	0	-6
1	1	1	1	-7

- We could build arithmetic circuits using this, but there is a more elegant way!

  JACOBS UNIVERSITY

All of these problems could be dealt with in principle, but together they form a nuisance, that at least prompts us to look for something more elegant. The so-called two's complement representation also uses a sign bit, but arranges the lower part of the table in the last slide in the opposite order, freeing the negative representation of the zero. The technical trick here is to use the sign bit (we still have to take into account the width n of the representation) not as a mirror, but to translate the positive representation by subtracting 2^n.

The Two's Complement Number System

- Definition 8.3: Given the binary string $a = \langle a_n, \ldots, a_0\rangle \in \mathbb{B}^{n+1}$, where $n > 1$. The integer represented by a in the $(n+1)$-bit two's complement, written as $\langle\!\langle a\rangle\!\rangle_n^{2s}$, is defined as

$$\begin{aligned}\langle\!\langle a\rangle\!\rangle_n^{2s} &= -a_n \cdot 2^n + \langle\!\langle a[n-1,0]\rangle\!\rangle \\ &= -a_n \cdot 2^n + \sum_{i=0}^{n-1} a_i \cdot 2^i\end{aligned}$$

- Notation 8.4:Write $B_n^{2s}(z)$ for the binary string that represents z in the two's complement number system, i.e., $\langle\!\langle B_n^{2s}(z)\rangle\!\rangle_n^{2s} = z$.

2's compl.				integer
0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0
1	1	1	1	-1
1	1	1	0	-2
1	1	0	1	-3
1	1	0	0	-4
1	0	1	1	-5
1	0	1	0	-6
1	0	0	1	-7
1	0	0	0	-8

  JACOBS UNIVERSITY

We will see that this representation has much better properties than the naive sign-bit representation we experimented with above. The first set of properties are quite trivial, they just formalize the intuition of moving the representation down, rather than mirroring it.

Properties of Two's Complement Numbers (TCN)

Let $b = \langle b_n, \ldots, b_0 \rangle$ be a number in the $n+1$-bit two's complement system, then

- ▷ Positive numbers and the zero have a sign bit 0, i.e., $b_n = 0 \Leftrightarrow \langle\!\langle b \rangle\!\rangle_n^{2s} \geq 0$.
- ▷ Negative numbers have a sign bit 1, i.e., $b_n = 1 \Leftrightarrow \langle\!\langle b \rangle\!\rangle_n^{2s} < 0$.
- ▷ For positive numbers, the two's complement representation corresponds to the normal binary number representation, i.e., $b_n = 0 \Leftrightarrow \langle\!\langle b \rangle\!\rangle_n^{2s} = \langle\!\langle b \rangle\!\rangle$
- ▷ There is a unique representation of the number zero in the n-bit two's complement system, namely $0^{n+1} = \langle 0, \ldots, 0 \rangle$.
- ▷ This number system has an asymmetric range $\mathcal{R}_{n+1}^{2s} := \{-2^n, \ldots, 2^{n-1}\}$.

 57 JACOBS UNIVERSITY

The next property is so central for what we want to do, it is upgraded to a theorem. It says that the mirroring operation (passing from a number to it's negative sibling) can be achieved by two very simple operations: flipping all the zeros and ones, and incrementing.

The Structure Theorem for TCN

- ▷ Theorem 8.5: *Let $a \in \mathbb{B}^{n+1}$ be a binary string, then $-\langle\!\langle a \rangle\!\rangle_n^{2s} = \langle\!\langle \overline{a} \rangle\!\rangle_n^{2s} + 1$.*
- ▷ Proof: by calculation using the definitions

$$\begin{aligned}
\langle\!\langle \overline{a_n}, \overline{a_{n-1}}, \ldots, \overline{a_0} \rangle\!\rangle_n^{2s} &= -\overline{a_n} \cdot 2^n + \langle\!\langle \overline{a_{n-1}}, \ldots, \overline{a_0} \rangle\!\rangle \\
&= \overline{a_n} \cdot (-2^n) + \sum_{i=0}^{n-1} \overline{a_i} \cdot 2^i \\
&= (1 - a_n) \cdot (-2^n) + \sum_{i=0}^{n-1} (1 - a_i) \cdot 2^i \\
&= (1 - a_n) \cdot (-2^n) + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} a_i \cdot 2^i \\
&= -2^n + a_n \cdot 2^n + 2^n - 1 - \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle \\
&= (-2^n + 2^n) + a_n \cdot 2^n - \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle - 1 \\
&= -(a_n \cdot (-2^n) + \langle\!\langle a_{n-1}, \ldots, a_0 \rangle\!\rangle) - 1 \\
&= -\langle\!\langle a \rangle\!\rangle_n^{2s} - 1
\end{aligned}$$

□

 58 JACOBS UNIVERSITY

A first simple application of the TCN structure theorem is that we can use our existing conversion routines (for binary natural numbers) to do TCN conversion (for integers).

Application: Converting from and to TCN?

- to convert an integer $-z \in \mathbb{Z}$ with $z \in \mathbb{N}$ into an n-bit TCN
  - generate the n-bit binary number representation $B(z) = \langle b_{n-1}, \ldots, b_0 \rangle$
  - complement it to $\overline{B(z)}$, i.e., the bitwise negation $\overline{b_i}$ of $B(z)$
  - increment (add 1) $\overline{B(z)}$, i.e. compute $B(\langle\langle \overline{B(z)} \rangle\rangle + 1)$
- to convert a negative n-bit TCN $b = \langle b_{n-1}, \ldots, b_0 \rangle$, into an integer
  - decrement b, (compute $B(\langle\langle b \rangle\rangle - 1)$)
  - complement it to $\overline{B(\langle\langle b \rangle\rangle - 1)}$
  - compute the decimal representation and negate it to $-\langle\langle \overline{B(\langle\langle b \rangle\rangle - 1)} \rangle\rangle$

Subtraction and Two's Complement Numbers

- Idea: With negative numbers use our adders directly
- Definition 8.6:An n-bit subtracter is a circuit that implements the function $f_{SUB}^n \colon \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^n$ such that

$$f_{SUB}^n(a, b, b') = B_n^{2s}(\langle\langle a \rangle\rangle_n^{2s} - \langle\langle b \rangle\rangle_n^{2s} - b')$$

  for all $a, b \in \mathbb{B}^n$ and $b' \in \mathbb{B}$. The bit b' is the so-called input borrow bit.
- Note: We have $\langle\langle a \rangle\rangle_n^{2s} - \langle\langle b \rangle\rangle_n^{2s} = \langle\langle a \rangle\rangle_n^{2s} + (-\langle\langle b \rangle\rangle_n^{2s}) = \langle\langle a \rangle\rangle_n^{2s} + \langle\langle \overline{b} \rangle\rangle_n^{2s} + 1$
- Idea: Can we implement an n-bit subtracter as $f_{SUB}^n(a, b, b') = f_{FA}^n(a, \overline{b}, \overline{b'})$?
- not immediately: We have to make sure that the full adder plays nice with twos complement numbers

In addition to the unique representation of the zero, the two's complement system has an additional important property. It is namely possible to use the adder circuits introduced previously without any modification to add integers in two's complement representation.

Addition of TCN

- Idea: use the adders without modification for TCN arithmetic
- Definition 8.7: An n-bit two's complement adder ($n > 1$) is a circuit that corresponds to the function $f_{TCA}^n \colon \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^n$, such that $f_{TCA}^n(a, b, c') = B_n^{2s}(\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c')$ for all $a, b \in \mathbb{B}^n$ and $c' \in \mathbb{B}$.
- Theorem 8.8: $f_{TCA}^n = f_{FA}^n$ *(first prove some Lemmas)*

It is not obvious that the same circuits can be used for the addition of binary and two's complement numbers. So, it has to be shown that the above function $TCAcircFNn$ and the full adder function

f_{FA}^n from definition?? are identical. To prove this fact, we first need the following lemma stating that a $(n+1)$-bit two's complement number can be generated from a n-bit two's complement number without changing its value by duplicating the sign-bit:

TCN Sign Bit Duplication Lemma

▷ Idea: An $(n+1)$-bit TCN can be generated from a n-bit TCN without changing its value by duplicating the sign-bit

▷ Lemma 8.9: *Let $a = \langle a_n, \ldots, a_0 \rangle \in \mathbb{B}^{n+1}$ be a binary string, then* $\langle\langle a_n, a_n, a_{n-1}, \ldots, a_0 \rangle\rangle_n^{2s} = \langle\langle a \rangle\rangle_n^{2s}$.

▷ Proof: by calculation

$$\begin{aligned}
\langle\langle a_n, a_n, a_{n-1}, \ldots, a_0 \rangle\rangle_n^{2s} &= -a_n \cdot 2^{n+1} + \langle\langle a_n, a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= -a_n \cdot 2^{n+1} + a_n \cdot 2^n + \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= a_n \cdot (-2^{n+1} + 2^n) + \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= a_n \cdot (-2 \cdot 2^n + 2^n) + \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= -a_n \cdot 2^n + \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle \\
&= \langle\langle a \rangle\rangle_n^{2s}
\end{aligned}$$

□

We will now come to a major structural result for two's complement numbers. It will serve two purposes for us:

1. It will show that the same circuits that produce the sum of binary numbers also produce proper sums of two's complement numbers.
2. It states concrete conditions when a valid result is produced, namely when the last two carry-bits are identical.

The TCN Main Theorem

▷ Let $a = \langle a_{n-1}, \ldots, a_0 \rangle, b = \langle b_{n-1}, \ldots, b_0 \rangle \in \mathbb{B}^n$ and $c \in \mathbb{B}$.

▷ Definition 8.10:We call $ic_k(a, b, c)$, the k-th intermediate carry of an addition of a and b,

$$\langle\langle ic_k(a, b, c), s_{k-1}, \ldots, s_0 \rangle\rangle = \langle\langle a_{k-1}, \ldots, a_0 \rangle\rangle + \langle\langle b_{k-1}, \ldots, b_0 \rangle\rangle + c$$

for some $s_i \in \mathbb{B}$.

▷ Theorem 8.11:

1. $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c \in \mathcal{R}_n^{2s}$, *iff* $ic_{n+1}(a, b, c) = ic_n(a, b, c)$.
2. *If* $ic_{n+1}(a, b, c) = ic_n(a, b, c)$, *then* $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c = \langle\langle s \rangle\rangle_n^{2s}$, *where* $\langle\langle ic_{n+1}(a, b, c), s_n, \ldots, s_0 \rangle\rangle = \langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + c$.

Proof of the TCN Main Theorem

Proof: Let us consider the sign-bits a_n and b_n separately from the value-bits $a' = \langle a_{n-1}, \ldots, a_0 \rangle$ and $b' = \langle b_{n-1}, \ldots, b_0 \rangle$.

P.1 Then
$$\begin{aligned} \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c &= \langle\langle a_{n-1}, \ldots, a_0 \rangle\rangle + \langle\langle b_{n-1}, \ldots, b_0 \rangle\rangle + c \\ &= \langle\langle ic_n(a,b,c), s_{n-1} \ldots, s_0 \rangle\rangle \end{aligned}$$
and $a_n + b_n + ic_n(a,b,c) = \langle\langle ic_{n+1}(a,b,c), s_n \rangle\rangle$.

P.2 We have to consider three cases

P.2.1 $a_n = b_n = 0$:

P.2.1.1 a and b are both positive, so $ic_{n+1}(a,b,c) = 0$ and furthermore
$$\begin{aligned} ic_n(a,b,c) = 0 &\Leftrightarrow \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \leq 2^n - 1 \\ &\Leftrightarrow \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c \leq 2^n - 1 \end{aligned}$$

P.2.1.2 Hence,
$$\begin{aligned} \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c &= \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \\ &= \langle\langle s_{n-1}, \ldots, s_0 \rangle\rangle \\ &= \langle\langle 0, s_{n-1}, \ldots, s_0 \rangle\rangle = \langle\langle s \rangle\rangle_n^{2s} \end{aligned}$$ □

□

Proof of the TCN Main Theorem

Proof: cont'd

P.1 Case 2:

P.1.1 $a_n = b_n = 1$:

P.1.1.1 a and b are both negative, so $ic_{n+1}(a,b,c) = 1$ and furthermore $ic_n(a,b,c) = 1$, iff $\langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \geq 2^n$, which is the case, iff $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c = -2^{n+1} + \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \geq -2^n$

P.1.1.2 Hence,
$$\begin{aligned} \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c &= -2^n + \langle\langle a' \rangle\rangle + -2^n + \langle\langle b' \rangle\rangle + c \\ &= -2^{n+1} + \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \\ &= -2^{n+1} + \langle\langle 1, s_{n-1}, \ldots, s_0 \rangle\rangle \\ &= -2^n + \langle\langle s_{n-1}, \ldots, s_0 \rangle\rangle \\ &= \langle\langle s \rangle\rangle_n^{2s} \end{aligned}$$ □

□

Proof of the TCN Main Theorem Proof: cont'd

P.1 Case 3:

P.1.1 $a_n \neq b_n$:

P.1.1.1 Without loss of generality assume that $a_n = 0$ and $b_n = 1$. (then $ic_{n+1}(a,b,c) = ic_n(a,b,c)$)

P.1.1.2 Hence, the sum of a and b is in the admissible range as

$$\langle\langle a\rangle\rangle_n^{2s} + \langle\langle b\rangle\rangle_n^{2s} + c = \langle\langle a'\rangle\rangle + \langle\langle b'\rangle\rangle + c - 2^n$$

and $0 \leq \langle\langle a'\rangle\rangle + \langle\langle b'\rangle\rangle + c \leq 2^{n+1} - 1$

P.1.1.3 So we have

$$\begin{aligned}\langle\langle a\rangle\rangle_n^{2s} + \langle\langle b\rangle\rangle_n^{2s} + c &= -2^n + \langle\langle a'\rangle\rangle + \langle\langle b'\rangle\rangle + c \\ &= -2^n + \langle\langle ic_n(a,b,c), s_{n-1}, \ldots, s_0\rangle\rangle \\ &= -(1 - ic_n(a,b,c)) \cdot 2^n + \langle\langle s_{n-1}, \ldots, s_0\rangle\rangle \\ &= \langle\langle \overline{ic_n(a,b,c)}, s_{n-1}, \ldots, s_0\rangle\rangle^{2s}\end{aligned}$$

P.1.1.4 Furthermore, we can conclude that $\langle\langle \overline{ic_n(a,b,c)}, s_{n-1}, \ldots, s_0\rangle\rangle^{2s} = \langle\langle s\rangle\rangle_n^{2s}$ as $s_n = a_n \oplus b_n \oplus ic_n(a,b,c) \oplus = 1 \oplus ic_n(a,b,c) \oplus = \overline{ic_n(a,b,c)}$. □

P.2 Thus we have considered all the cases and completed the proof. □

The Main Theorem for TCN again

- ▷ Given two $(n+1)$-bit two's complement numbers a and b. The above theorem tells us that the result s of an $(n+1)$-bit adder is the proper sum in two's complement representation iff the last two carries are identical.
- ▷ If not, a and b were too large or too small.

The most important application of the main TCN theorem is that we can build a combinatorial circuit that can add and subtract (depending on a control bit). This is actually the first instance of a concrete programmable computation device we have seen up to date (we interpret the control bit as a program, which changes the behavior of the device). The fact that this is so simple, it only runs two programs should not deter us; we will come up with more complex things later.

Building an Add/Subtract Unit

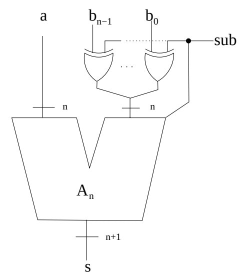
▷ Idea: Build a Combinational Circuit that can add and subtract (sub $= 1 \rightsquigarrow$ subtract)

▷ If sub $= 0$, then the circuit acts like an adder ($a \oplus 0 \oplus = a$)

▷ If sub $= 1$, let $S := \langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle \overline{b_{n-1}}, \ldots, \overline{b_0} \rangle\!\rangle_n^{2s} + 1$ ($a \oplus 0 \oplus = 1 - a$)

▷ For $S \in \mathcal{R}_{n-1}^{2s}$ the TCN theorem guarantees

$$\begin{aligned} & \langle\!\langle s_{n-1}, \ldots, s_0 \rangle\!\rangle_n^{2s} \\ = \; & \langle\!\langle a \rangle\!\rangle_n^{2s} + \langle\!\langle \overline{b_{n-1}}, \ldots, \overline{b_0} \rangle\!\rangle_n^{2s} + 1 \\ = \; & \langle\!\langle a \rangle\!\rangle_n^{2s} - \langle\!\langle b \rangle\!\rangle_n^{2s} - 1 + 1 \end{aligned}$$



▷ Summary: We have built a combinational circuit that can perform 2 arithmetic operations depending on a control bit.

▷ Idea: Extend this to a arithmetic logic unit (ALU) with more operations (+, -, *, /, n-AND, n-OR,...)

In fact extended variants of the very simple Add/Subtract unit are at the heart of any computer. These are called arithmetic logic units.

9 Sequential Logic Circuits and Memory Elements

So far we have considered combinatorial logic, i.e. circuits for which the output depends only on the inputs. In many instances it is desirable to have the next output depend on the current output.

Sequential Logic Circuits

▷ In combinational circuits, outputs only depend on inputs (no state)

▷ We have disregarded all timing issues (except for favoring shallow circuits)

▷ Definition 9.1: Circuits that remember their current output or state are often called sequential logic circuits.

▷ Example 9.2: A *counter*, where the next number to be output is determined by the current number stored.

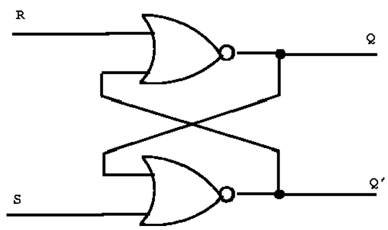
▷ Sequential logic circuits need some ability to store the current state

Clearly, sequential logic requires the ability to store the current state. In other words, *memory* is required by sequential logic circuits. We will investigate basic circuits that have the ability to store bits of data. We will start with the simplest possible memory element, and develop more elaborate versions from it.

The circuit we are about to introduce is the simplest circuit that can keep a state, and thus act as a (precursor to) a storage element. Note that we are leaving the realm of acyclic graphs here. Indeed storage elements cannot be realized with combinational circuits as defined above.

RS Flip-Flop

▷ **Definition** 9.3: A RS-flipflop (or RS-latch)is constructed by feeding the outputs of two NOR gates back to the other NOR gates input. The inputs R and S are referred to as the Reset and Set inputs, respectively.



R	S	Q	Q'	Comment
0	1	1	0	Set
1	0	0	1	Reset
0	0	Q	Q'	Hold state
1	1	?	?	Avoid

▷ Note: the output Q' is simply the inverse of Q. (supplied for convenience)

▷ Note: An RS flipflop can also be constructed from $NAND$ gates.

To understand the operation of the RS-flipflop we first reminde ourselves of the truth table of the NOR gate on the right: If one of the inputs is 1, then the output is 0, irrespective of the other. To understand the RS-flipflop, we will go through the input combinations summarized in the table above in detail. Consider the following scenarios:

↓	0	1
0	1	0
1	0	0

$S = 1$ **and** $R = 0$ The output of the bottom NOR gate is 0, and thus $Q' = 0$ irrespective of the other input. So both inputs to the top NOR gate are 0, thus, $Q = 1$. Hence, the input combination $S = 1$ and $R = 0$ leads to the flipflop being *set* to $Q = 1$.

$S = 0$ **and** $R = 1$ The argument for this situation is symmetric to the one above, so the outputs become $Q = 0$ and $Q' = 1$. We say that the flipflop is *reset*.

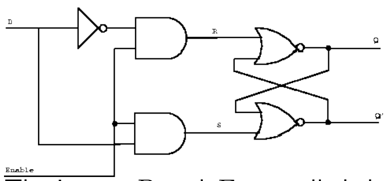
$S = 0$ **and** $R = 0$ Assume the flipflop is set ($Q = 0$ and $Q' = 1$), then the output of the top NOR gate remains at $Q = 1$ and the bottom NOR gate stays at $Q' = 0$. Similarly, when the flipflop is in a reset state ($Q = 1$ and $Q' = 0$), it will remain there with this input combination. Therefore, with inputs $S = 0$ and $R = 0$, the flipflop remains in its state.

$S = 1$ **and** $R = 1$ This input combination will be avoided, we have all the functionality (*set*, *reset*, and *hold*) we want from a memory element.

An RS-flipflop is rarely used in actual sequential logic. However, it is the fundamental building block for the very useful D-flipflop.

The D-Flipflop: the simplest memory device

- Recap: A RS-flipflop can store a state (set Q to 1 or reset Q to 0)
- Problem: We would like to have a single data input and avoid $R = S$ states.
- Idea: Add interface logic to do just this
- Definition 9.4: A D-Flipflop is an RS-flipflop with interface logic as below.



E	D	R	S	Q	Comment
1	1	0	1	1	set Q to 1
1	0	1	0	0	reset Q to 0
0	D	0	0	Q	hold Q

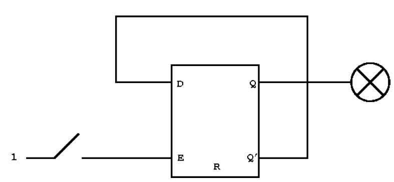
The inputs D and E are called the data and enable inputs.

- When $E = 1$ the value of D determines the value of the output Q, when E returns to 0, the most recent input D is "remembered."

Sequential logic circuits are constructed from memory elements and combinatorial logic gates. The introduction of the memory elements allows these circuits to remember their state. We will illustrate this through a simple example.

Example: On/Off Switch

- Problem: Pushing a button toggles a LED between on and off. (first push switches the LED on, second push off,. . .)
- Idea: Use a D-flipflop (to remember whether the LED is currently on or off) connect its Q' ouput to its D input (next state is inverse of current state)



In the on/off circuit, the external inputs (buttons) were connected to the E input. Definition 9.5: Such circuits are often called asynchronous as they keep track of events that occur at arbitrary instants of time, synchronous circuits in contrast operate on a periodic basis and the Enable input is connected to a common clock signal.

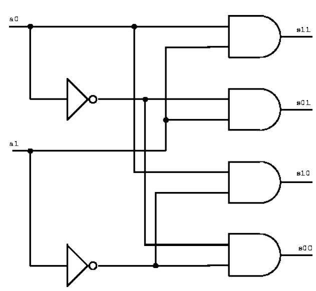
Random Access Memory Chips

- ▷ *Random access memory* (RAM) is used for storing a large number of bits.
- ▷ RAM is made up of storage elements similar to the D-flipflops we discussed.
- ▷ Principally, each storage element has a unique number or address represented in binary form.
- ▷ When the address of the storage element is provided to the RAM chip, the corresponding memory element can be written to or read from.
- ▷ We will consider the following questions:
  - ▷ What is the physical structure of RAM chips?
  - ▷ How are addresses used to select a particular storage element?
  - ▷ What do individual storage elements look like?
  - ▷ How is reading and writing distinguished?

 73 JACOBS UNIVERSITY

Address Decoder Logic

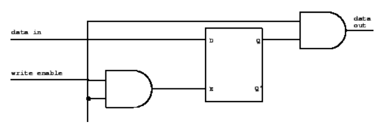
- ▷ Idea: Need a circuit that activates the row/column given the binary address:
  - ▷ At any time, only 1 output line is “on” and all others are off.
  - ▷ The line that is “on” specifies the desired column or row.
- ▷ Definition 9.6: The n-bit address decoder ADL^n has a n inputs and 2^n outputs. $f^m_{ADL}(a) = \langle b_1, \ldots, b_{2^n} \rangle$, where $b_i = 1$, iff $i = \langle\langle a \rangle\rangle$.
- ▷ Example 9.7: (Address decoder logic for 2-bit addresses)



 74 JACOBS UNIVERSITY

Storage Elements

- ▷ Idea (Input): Use a D-flipflop connect its E input to the ADL output. Connect the D-input to the common RAM data input line. (input only if addressed)
- ▷ Idea (Output): Connect the flipflop output to common RAM output line. But first AND with ADL output (output only if addressed)
- ▷ Problem: The read process should leave the value of the gate unchanged.
- ▷ Idea: Introduce a "write enable" signal (protect data during read) AND it with the ADL output and connect it to the flipflop's E input.
- ▷ Definition 9.8: A Storage Element is given by the foolowing diagram



 ©: Michael Kohlhase 75 JACOBS UNIVERSITY

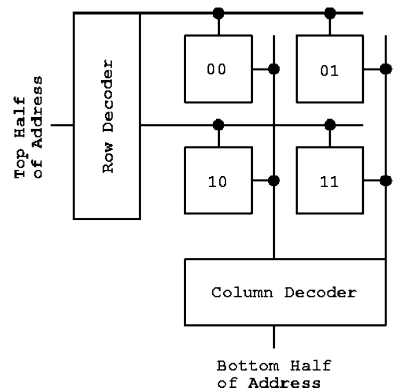
Remarks

- ▷ The storage elements are often simplified to reduce the number of transistors.
- ▷ For example, with care one can replace the flipflop by a capacitor.
- ▷ Also, with large memory chips it is not feasible to connect the data input and output and write enable lines directly to all storage elements.
- ▷ Also, with care one can use the same line for data input and data output.
- ▷ Today, multi-gigabyte RAM chips are on the market.
- ▷ The capacity of RAM chips doubles approximately every year.

 ©: Michael Kohlhase 76 JACOBS UNIVERSITY

Layout of Memory Chips

- ▷ To take advantage of the two-dimensional nature of the chip, storage elements are arranged on a square grid. (columns and rows of storage elements)
- ▷ For example, a 1 Megabit RAM chip has of 1024 rows and 1024 columns.
- ▷ idenfity storage element can be identified by its row and column "coordinates". (AND them for addressing)
- ▷ Hence, to select a particular storage location the address information must be translated into row and column specification.
- ▷ The address information is divided into two halfs; the top half is used to select the row and the bottom half is used to select the column.



 77 JACOBS UNIVERSITY

10 How to build a Computer (in Principle)

In this part of the course, we will learn how to use the very simple computational devices we built in the last section and extend them to fully programmable devices using the so-called "von Neumann Architecture". For this, we need random access memory (RAM).

For our purposes, . They can be written to, (after which they store the n values at their n input edges), and they can be queried: then their output edges have the n values that were stored in the memory cell. Querying a memory cell does not change the value stored in it.

Our notion of time is similarly simple, in our analysis we assume a series of discrete clock ticks that synchronize all events in the circuit. We will only observe the circuits on each clock tick and assume that all computational devices introduced for the register machine complete computation before the next tick. Real circuits, also have a clock that synchronizes events (the clock frequency (currently around 3 GHz for desktop CPUs) is a common approximation measure of processor performance), but the assumption of elementary computations taking only one click is wrong in production systems.

How to Build a Computer (REMA; the Register Machine)

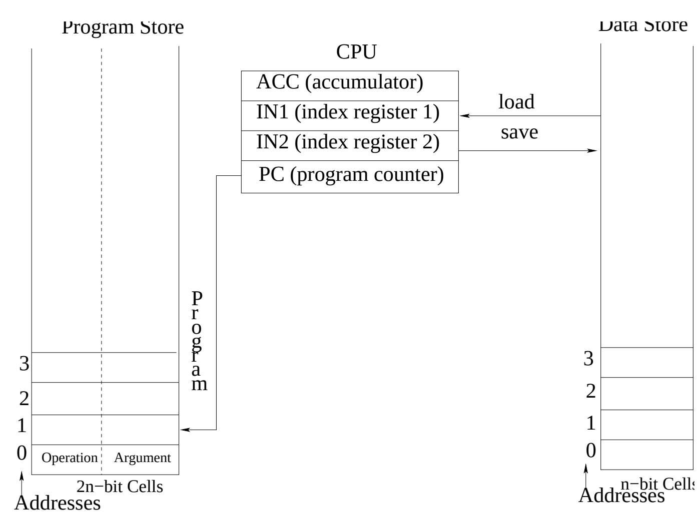
- ▷ Take an n-bit arithmetic logic unit (ALU)
- ▷ add registers: few (named) n-bit memory cells near the ALU
  - ▷ program counter (PC) (points to current command in program store)
  - ▷ accumulator (ACC) (the a input and output of the ALU)
- ▷ add RAM: lots of random access memory (elsewhere)
  - ▷ program store: $2n$-bit memory cells (addressed by $P\colon \mathbb{N} \to \mathbb{B}^{2n}$)
  - ▷ data store: n-bit memory cells (words addressed by $D\colon \mathbb{N} \to \mathbb{B}^{n}$)
- ▷ add a memory management unit(MMU) (move values between RAM and registers)
- ▷ program it in assembler language (lowest level of programming)

We have three kinds of memory areas in the REMA register machine: The registers (our architecture has two, which is the minimal number, real architectures have more for convenience) are just simple n-bit memory cells.

The programstore is a sequence of up to 2^n memory $2n$-bit memory cells, which can be accessed (written to and queried) randomly i.e. by referencing their position in the sequence; we do not have to access them by some fixed regime, e.g. one after the other, in sequence (hence the name random access memory: RAM). We address the Program store by a function $P\colon \mathbb{N} \to \mathbb{B}^{2n}$. The data store is also RAM, but a sequence or n-bit cells, which is addressed by the function $D\colon \mathbb{N} \to \mathbb{B}^{n}$.

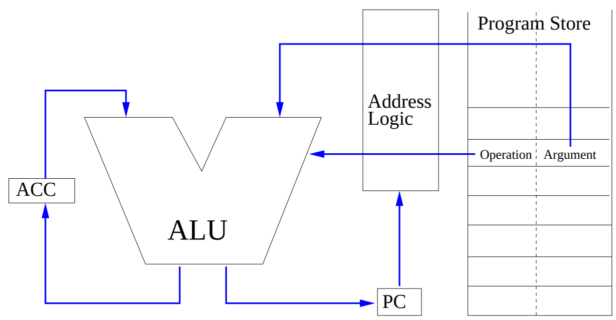
The value of the program counter is interpreted as a binary number that addresses a $2n$-bit cell in the program store. The accumulator is the register that contains one of the inputs to the ALU before the operation (the other is given as the argument of the program instruction); the result of the ALU is stored in the accumulator after the instruction is carried out.

Memory Plan of a Register Machine



The ALU and the MMU are control circuits, they have a set of n-bit inputs, and n-bit outputs, and an n-bit control input. The prototypical ALU, we have already seen, applies arithmetic or logical operator to its regular inputs according to the value of the control input. The MMU is very similar, it moves n-bit values between the RAM and the registers according to the value at the control input. We say that the MMU moves the (n-bit) value from a register R to a memory cell C, iff after the move both have the same value: that of R. This is usually implemented as a query operation on R and a write operation to C. Both the ALU and the MMU could in principle encode 2^n operators (or commands), in practice, they have fewer, since they share the command space.

Circuit Overview over the CPU



In this architecture (called the register machine architecture), programs are sequences of $2n$-bit numbers. The first n-bit part encodes the instruction, the second one the argument of the instruction. The program counter addresses the current instruction (operation + argument).

We will now instantiate this general register machine with a concrete (hypothetical) realization, which is sufficient for general programming, in principle. In particular, we will need to identify a set of program operations. We will come up with 18 operations, so we need to set $n \geq 5$. It is possible to do programming with $n = 4$ designs, but we are interested in the general principles more than optimization.

The main idea of programming at the circuit level is to map the operator code (an n-bit binary number) of the current instruction to the control input of the ALU and the MMU, which will then perform the action encoded in the operator.

Since it is very tedious to look at the binary operator codes (even it we present them as hexadecimal numbers). Therefore it has become customary to use a mnemonic encoding of these in simple word tokens, which are simpler to read, the so-called assembler language.

Assembler Language

- Idea: Store program instructions as n-bit values in program store, map these to control inputs of ALU, MMU.
- Definition 10.1: assembler language as mnemonic encoding of n-bit binary codes.

instruction	effect	PC	comment
`LOAD` i	$ACC := D(i)$	$PC := PC+1$	load data
`STORE` i	$D(i) := ACC$	$PC := PC+1$	store data
`ADD` i	$ACC := ACC + D(i)$	$PC := PC+1$	add to ACC
`SUB` i	$ACC := ACC - D(i)$	$PC := PC+1$	subtract from ACC
`LOADI` i	$ACC := i$	$PC := PC+1$	load number
`ADDI` i	$ACC := ACC + i$	$PC := PC+1$	add number
`SUBI` i	$ACC := ACC - i$	$PC := PC+1$	subtract number

Definition 10.2: The meaning of the program instructions are specified in their ability to change the state of the memory of the register machine. So to understand them, we have to trace the state of the memory over time (looking at a snapshot after each clock tick; this is what we do in the comment fields in the tables on the next slide). We speak of an imperative programming language, if this is the case.

Example 10.3: This is in contrast to the programming language SML that we have looked at before. There we are not interested in the state of memory. In fact state is something that we want to avoid in such functional programming languages for conceptual clarity; we relegated all things that need state into special constructs: effects.

To be able to trace the memory state over time, we also have to think about the initial state of the register machine (e.g. after we have turned on the power). We assume the state of the registers and the data store to be arbitrary (who knows what the machine has dreamt). More interestingly, we assume the state of the program store to be given externally. For the moment, we may assume (as was the case with the first computers) that the program store is just implemented as a large array of binary switches; one for each bit in the program store. Programming a computer at that time was done by flipping the switches ($2n$) for each instructions. Nowadays, parts of the initial program of a computer (those that run, when the power is turned on and bootstrap the operating system) is still given in special memory (called the firmware) that keeps its state even when power is shut off. This is conceptually very similar to a bank of switches.

Example Programs

▷ Example 10.4: Exchange the values of cells 0 and 1 in the data store

P	instruction	comment
0	`LOAD` 0	$ACC\colon = D(0) = x$
1	`STORE` 2	$D(2)\colon = ACC = x$
2	`LOAD` 1	$ACC\colon = D(1) = y$
3	`STORE` 0	$D(0)\colon = ACC = y$
4	`LOAD` 2	$ACC\colon = D(2) = x$
5	`STORE` 1	$D(1)\colon = ACC = x$

▷ Example 10.5: Let $D(1) = a$, $D(2) = b$, and $D(3) = c$, store $a + b + c$ in data cell 4

P	instruction	comment
0	`LOAD` 1	$ACC\colon = D(1) = a$
1	`ADD` 2	$ACC\colon = ACC + D(2) = a + b$
2	`ADD` 3	$ACC\colon = ACC + D(3) = a + b + c$
3	`STORE` 4	$D(4)\colon = ACC = a + b + c$

▷ use `LOADI` i, `ADDI` i, `SUBI` i to set/increment/decrement ACC (impossible otherwise)

So far, the problems we have been able to solve are quite simple. They had in common that we had to know the addresses of the memory cells we wanted to operate on at programming time, which is not very realistic. To alleviate this restriction, we will now introduce a new set of instructions, which allow to calculate with addresses.

Index Registers

▷ Problem: Given $D(0) = x$ and $D(1) = y$, how do we store y into cell x of the data store? (impossible, as we have only absolute addressing)

▷ Idea: introduce more registers and register instructions ($IN1$, $IN2$ suffice)

instruction	effect	PC	comment
`LOADIN`j i	$ACC\colon = D(INj + i)$	$PC\colon = PC + 1$	relative load
`STOREIN`j i	$D(INj + i)\colon = ACC$	$PC\colon = PC + 1$	relative store
`MOVE` S T	$T\colon = S$	$PC\colon = PC + 1$	move register S (source) to register T (target)

▷ Problem Solution:

P	instruction	comment
0	`LOAD` 0	$ACC\colon = D(0) = x$
1	`MOVE` ACC $IN1$	$IN1\colon = ACC = x$
2	`LOAD` 1	$ACC\colon = D(1) = y$
3	`STOREIN1` 0	$D(x) = D(IN1 + 0)\colon = ACC = y$

Note that the `LOADIN` are not binary instructions, but that this is just a short notation for unary instructions `LOADIN` 1 and `LOADIN` 2 (and similarly for `MOVE` S T).

Note furthermore, that the addition logic in `LOADIN` j is simply for convenience (most assembler languages have it, since working with address offsets is commonplace). We could have always imitated this by a simpler relative load command and an `ADD` instruction.

A very important ability we have to add to the language is a set of instructions that allow us to re-use program fragments multiple times. If we look at the instructions we have seen so far, then we see that they all increment the program counter. As a consequence, program execution is a linear walk through the program instructions: every instruction is executed exactly once. The set of problems we can solve with this is extremely limited. Therefore we add a new kind of instruction. Jump instructions directly manipulate the program counter by adding the argument to it (note that this partially invalidates the circuit overview slide above[5], but we will not worry about this). EdNote(5)

Another very important ability is to be able to change the program execution under certain conditions. In our simple language, we will only make jump instructions conditional (this is sufficient, since we can always jump the respective instruction sequence that we wanted to make conditional). For convenience, we give ourselves a set of comparison relations (two would have sufficed, e.g. $=$ and $<$) that we can use to test.

Jump Instructions

- ▷ Problem: Until now, we can only write linear programs (A program with n steps executes n instructions)
- ▷ Idea: Need instructions that manipulate the PC directly
- ▷ Let $\mathcal{R} \in \{<,=,>,\leq,\neq,\geq\}$ be a comparison relation

instruction	effect	PC	comment
`JUMP` i		$PC: = PC+i$	jump forward i steps
`JUMP`$_{\mathcal{R}}$ i		$PC: = \begin{cases} PC+i \text{ if } \mathcal{R}(ACC,0) \\ PC+1 \text{ if else} \end{cases}$	conditional jump

- ▷ Two more:

instruction	effect	PC	comment
`NOP` i		$PC: = PC+1$	no operation
`STOP` i			stop computation

The final addition to the language are the `NOP` (no operation) and `STOP` operations. Both do not look at their argument (we have to supply one though, so we fit our instruction format). the `NOP` instruction is sometimes convenient, if we keep jump offsets rational, and the `STOP` instruction terminates the program run (e.g. to give the user a chance to look at the results.)

[5]EdNote: reference

Example Program

▷ Example 10.6: Let $D(0) = n$, $D(1) = a$, and $D(2) = b$, copy the values of cells $a, \ldots, a + n - 1$ to cells $b, \ldots, b + n - 1$, while $a, b \geq 3$ and $|a - b| \geq n$.

P	instruction	comment	P	instruction	comment
0	`LOAD` 1	$ACC\colon = a$	10	`MOVE` ACC $IN1$	$IN1\colon = IN1 + 1$
1	`MOVE` ACC $IN1$	$IN1\colon = a$	11	`MOVE` $IN2$ ACC	
2	`LOAD` 2	$ACC\colon = b$	12	`ADDI` 1	
3	`MOVE` ACC $IN2$	$IN2\colon = b$	13	`MOVE` ACC $IN2$	$IN2\colon = IN2 + 1$
4	`LOAD` 0	$ACC\colon = n$	14	`LOAD` 0	
5	$\texttt{JUMP}_=$ 13	if $n = 0$ then stop	15	`SUBI` 1	
6	`LOADIN1` 0	$ACC\colon = D(IN1)$	16	`STORE` 0	$D(0)\colon = D(0) - 1$
7	`STOREIN2` 0	$D(IN2)\colon = ACC$	17	`JUMP` -12	goto step 5
8	`MOVE` $IN1$ ACC		18	`STOP` 0	Stop
9	`ADDI` 1				

▷ Lemma 10.7: *We have $D(0) = n-(i-1)$, $IN1 = a+i-1$, and $IN2 = b+i-1$ for all $1 \leq i \leq n+1$.* (*the program does what we want*)

▷ proof by induction on n.

▷ Definition 10.8: The induction hypotheses are called loop invariants.

11 How to build a SML-Compiler (in Principle)

In this part of the course, we will build a compiler for a simple functional programming language. A compiler is a program that examines a program in a high-level programming language and transforms it into a program in a language that can be interpreted by an existing computation engine, in our case, the register machine we discussed above.

We have seen that our register machine runs programs written in assembler, a simple machine language expressed in two-word instructions. Machine languages should be designed such that on the processors that can be built machine language programs can execute efficiently. On the other hand machine languages should be built, so that programs in a variety of high-level programming languages can be transformed automatically (i.e. compiled) into efficient machine programs. We have seen that our assembler language `ASM` is a serviceable, if frugal approximation of the first goal for very simple processors. We will now show that it also satisfies the second goal by exhibiting a compiler for a simple SML-like language.

In the last 20 years, the machine languages for state-of-the art processors have hardly changed. This stability was a precondition for the enormous increase of computing power we have witnessed during this time. At the same time, high-level programming languages have developed considerably, and with them, their needs for features in machine-languages. This leads to a significant mismatch, which has been bridged by the concept of a *virtual machine*.

virtualmachine is a simple machine-language program that interprets a slightly higher-level program — the "bytecode" — and simulates it on the existing processor. Byte code is still considered a machine language, just that it is realized via software on a real computer, instead of running directly on the machine. This allows to keep the compilers simple while only paying a small price in efficiency.

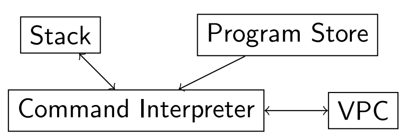
In our compiler, we will take this approach, we will first build a simple virtual machine (an `ASM` program) and then build a compiler that translates functional programs into byte code.

Virtual Machines

- ▷ Question: How to run high-level programming languages (like SML) on REMA?
- ▷ Answer: By providing a compiler, i.e. an ASM program that reads SML programs (as data) and transforms them into ASM programs.
- ▷ But: ASM is optimized for building simple, efficient processors, not as a translation target!
- ▷ Idea: Build an ASM program VM that interprets a better translation target language (interpret REMA+VM as a "virtual machine")
- ▷ Definition 11.1: An ASM program VM is called a virtual machine for a language $\mathcal{L}(\mathtt{VM})$, iff VM inputs a $\mathcal{L}(\mathtt{VM})$ program (as data) and runs it on REMA.
- ▷ Plan: Instead of building a compiler for SML to ASM, build a virtual machine VM for REMA and a compiler from SML to $\mathcal{L}(\mathtt{VM})$. (simpler and more transparent)

A Virtual Machine for Functional Programming

- ▷ Idea: We will build a stack-based virtual machine; this will have four components



- ▷ The stack is a memory segment operated as a "last-in-first-out" LIFO sequence
- ▷ The program store is a memory segment interpreted as a sequence of instructions
- ▷ The command interpreter is a ASM program that interprets commands from the program store and operates on the stack.
- ▷ The virtual program counter (VPC) is a register that acts as a the pointer to the current instruction in the program store.
- ▷ The virtual machine starts with the empty stack and VPC at the beginning of the program.

A Stack-Based VM language (Arithmetic Commands)

▷ Definition 11.2: VM Arithmetic Commands act on the stack

instruction	effect	VPC
`con` i	pushes i onto stack	$VPC\colon = VPC + 2$
`add`	pop x, pop y, push $x + y$	$VPC\colon = VPC + 1$
`sub`	pop x, pop y, push $x - y$	$VPC\colon = VPC + 1$
`mul`	pop x, pop y, push $x \cdot y$	$VPC\colon = VPC + 1$
`leq`	pop x, pop y, if $x \leq y$ push 1, else push 0	$VPC\colon = VPC + 1$

▷ Example 11.3: The $\mathcal{L}(\mathtt{VM})$ program "`con` 4 `con` 7 `add`" pushes $7 + 4 = 11$ to the stack.

▷ Example 11.4: Note the order of the arguments: the program "`con` 4 `con` 7 `sub`" first pushes 4, and then 7, then pops x and then y (so $x = 7$ and $y = 4$) and finally pushes $x - y = 7 - 4 = 3$.

▷ Stack-based operations work very well with the recursive structure of arithmetic expressions: we can compute the value of the expression $4 \cdot 3 - 7 \cdot 2$ with

`con` 2 `con` 7 `mul`	$7 \cdot 2$
`con` 3 `con` 4 `mul`	$4 \cdot 3$
`sub`	$4 \cdot 3 - 7 \cdot 2$

 88 JACOBS UNIVERSITY

Note: A feature that we will see time and again is that every (syntactically well-formed) expression leaves only the result value on the stack. In the present case, the computation never touches the part of the stack that was present before computing the expression. This is plausible, since the computation of the value of an expression is purely functional, it should not have an effect on the state of the virtual machine VM (other than leaving the result of course).

A Stack-Based VM language (Control)

▷ Definition 11.5: Control operators

instruction	effect	VPC
jp i		$VPC := VPC + i$
cjp i	pop x	if $x = 0$, then $VPC := VPC + i$ else $VPC := VPC + 2$
halt		—

▷ cjp is a "jump on false"-type expression. (if the condition is false, we jump else we continue)

▷ Example 11.6: For conditional expressions we use the conditional jump expressions: We can express "if $1 \leq 2$ then $4 - 3$ else $7 \cdot 5$" by the program

con 2 con 1 leq cjp 9	if $1 \leq 2$
con 3 con 4 sub jp 7	then $4 - 3$
con 5 con 7 mul	else $7 \cdot 5$
halt	

In the example, we first push 2, and then 1 to the stack. Then leq pops (so $x = 1$), pops again (making $y = 2$) and computes $x \leq y$ (which comes out as true), so it pushes 1, then it continues (it would jump to the else case on false).

Note: Again, the only effect of the conditional statement is to leave the result on the stack. It does not touch the contents of the stack at and below the original stack pointer.

A Stack-Based VM language (Imperative Variables)

▷ Definition 11.7: Imperative access to variables: Let $\mathcal{S}(i)$ be the number at stack position i.

instruction	effect	VPC
peek i	push $\mathcal{S}(i)$	$VPC := VPC + 2$
poke i	pop x $\mathcal{S}(i) := x$	$VPC := VPC + 2$

▷ Example 11.8: The program "con 5 con 7 peek 0 peek 1 add poke 1 mul halt" computes $5 \cdot (7 + 5) = 60$.

Of course the last example is somewhat contrived, this is certainly not the best way to compute $5 \cdot (7 + 5) = 60$, but it does the trick.

Extended Example: A `while` Loop

▷ Example 11.9: Consider the following program that computes $(12)!$ and the corresponding $\mathcal{L}(\texttt{VM})$ program:

`var n := 12; var a := 1;`	con 12 con 1
`while 2 <= n do (`	peek 0 con 2 leq cjp 18
`  a := a * n;`	peek 0 peek 1 mul poke 1
`  n := n - 1;`	con 1 peek 0 sub poke 0
`)`	jp -21
`return a;`	peek 1 halt

▷ Note that variable declarations only push the values to the stack, (memory allocation)

▷ they are referenced by `peek`ing the respective stack position

▷ they are assigned by `poke`ing the stack position (must remember that)

We see that again, only the result of the computation is left on the stack. In fact, the code snippet consists of two variable declarations (which extend the stack) and one **`while`** statement, which does not, and the **`return`** statement, which extends the stack again. In this case, we see that even though the **`while`** statement does not extend the stack it does change the stack below by the variable assignments (implemented as `poke` in $\mathcal{L}(\texttt{VM})$). We will use the example above as guiding intuition for a compiler from a simple imperative language to $\mathcal{L}(\texttt{VM})$ byte code below. But first we build a virtual machine for $\mathcal{L}(\texttt{VM})$.

We will now build a virtual machine for $\mathcal{L}(\texttt{VM})$ along the specification above.

A Virtual Machine for $\mathcal{L}(\texttt{VM})$

▷ We need to build a concrete `ASM` program that acts as a virtual machine for $\mathcal{L}(\texttt{VM})$.

▷ Choose a concrete register machine size: e.g. 32-bit words (like in a PC)

▷ Choose memory layout in the data store

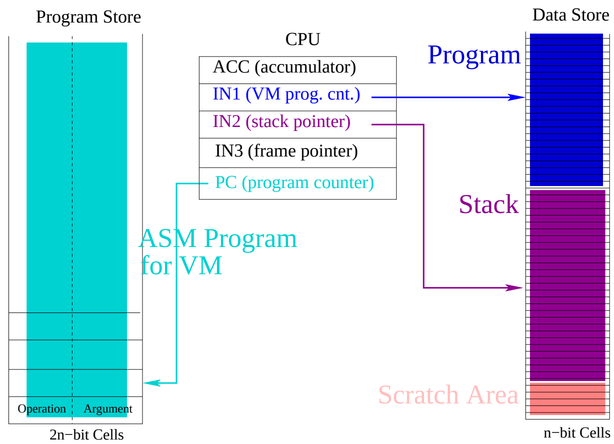
- ▷ the `VM` stack: $D(8)$ to $D(2^{24}-1)$, and (need the first 8 cells for `VM` data)
- ▷ the $\mathcal{L}(\texttt{VM})$ program store: $D(2^{24})$ to $D(2^{32}-1)$
- ▷ We represent the virtual program counter VPC by the index register $IN1$ and the stack pointer by the index register $IN2$ (with offset 8).
- ▷ We will use $D(0)$ as an argument store.

▷ choose a numerical representation for the $\mathcal{L}(\texttt{VM})$ instructions: (have lots of space)

`halt` $\mapsto 0$, `add` $\mapsto 1$, `sub` $\mapsto 2$, ...

Recall that the virtual machine `VM` is a `ASM` program, so it will reside in the `REMA` program store. This is the program executed by the register machine. So both the `VM` stack and the $\mathcal{L}(\texttt{VM})$ program have to be stored in the `REMA` data store (therefore we treat $\mathcal{L}(\texttt{VM})$ programs as sequences of words and have to do counting acrobatics for instructions of differing length). We somewhat arbitrarily fix a boundary in the data store of REMA at cell number $2^{24} - 1$. We will also need a little piece of scratch-pad memory, which we locate at cells 0-7 for convenience (then we can simply address with absolute numbers as addresses).

Memory Layout for the Virtual Machine



Extending REMA and ASM

- Give ourselves another register $IN3$ (and LOADIN 3, STOREIN 3, MOVE $*$ $IN3$, MOVE $IN3$ $*$)
- We will use a syntactic variant of ASM for transparency
  - JUMP and JUMP$_{\mathcal{R}}$ with labels(compute relative jump distances automatically)
  - inc R for MOVE R ACC, ADDI 1, MOVE ACC R (dec R similar)
  - note that inc R and dec R overwrite the current ACC (take care of it)
- All additions can be eliminated by substitution.

With these extensions, it is quite simple to write the ASM code that implements the virtual machine VM. The first part is a simple jump table, a piece of code that does nothing else than distributing the program flow according to the (numerical) instruction head. We assume that this program segment is located at the beginning of the program store, so that the REMA program counter points to the first instruction. This initializes the VM program counter and its stack pointer to the first cells of their memory segments. We assume that the $\mathcal{L}$(VM) program is already loaded in its proper location, since we have not discussed input and output for REMA.

Starting VM: the Jump Table

label	instruction	effect	comment
	LOADI 2^{24}	$ACC := 2^{24}$	load VM start address
	MOVE ACC $IN1$	$VPC := ACC$	set VPC
	LOADI 7	$ACC := 7$	load top of stack address
	MOVE ACC $IN2$	$SP := ACC$	set SP
$\langle jt \rangle$	LOADIN1 0	$ACC := D(IN1)$	load instruction
	$\texttt{JUMP}_=$ ⟨halt⟩		goto ⟨halt⟩
	SUBI 1		next instruction code
	$\texttt{JUMP}_=$ ⟨add⟩		goto ⟨add⟩
	SUBI 1		next instruction code
	$\texttt{JUMP}_=$ ⟨sub⟩		goto ⟨sub⟩
	⋮	⋮	⋮
⟨halt⟩	STOP 0		stop
	⋮	⋮	⋮

Now it only remains to present the ASM programs for the individual $\mathcal{L}$(VM) instructions. We will start with the arithmetical operations. The code for con is absolutely straightforward: we increment the VM program counter to point to the argument, read it, and store it to the cell the (suitably incremented) VM stack pointer points to. Once procedure has been executed we increment the VM program counter again, so that it points to the next $\mathcal{L}$(VM) instruction, and jump back to the beginning of the jump table.

For the add instruction we have to use the scratch pad area, since we have to pop two values from the stack (and we can only keep one in the accumulator). We just cache the first value in cell 0 of the program store.

Implementing Arithmetic Operators

label	instruction	effect	comment
⟨con⟩	inc $IN1$	$VPC := VPC + 1$	point to arg
	inc $IN2$	$SP := SP + 1$	prepare push
	LOADIN1 0	$ACC := D(VPC)$	read arg
	STOREIN2 0	$D(SP) := ACC$	store for push
	inc $IN1$	$VPC := VPC + 1$	point to next
	JUMP $\langle jt \rangle$		jump back
⟨add⟩	LOADIN2 0	$ACC := D(SP)$	read arg 1
	STORE 0	$D(0) := ACC$	cache it
	dec $IN2$	$SP := SP - 1$	pop
	LOADIN2 0	$ACC := D(SP)$	read arg 2
	ADD 0	$ACC := ACC + D(0)$	add cached arg 1
	STOREIN2 0	$D(SP) := ACC$	store it
	inc $IN1$	$VPC := VPC + 1$	point to next
	JUMP $\langle jt \rangle$		jump back

▷ sub, mul, and leq similar to add.

For example, mul could be implemented as follows:

label	instruction	effect	comment
⟨mul⟩	dec $IN2$	$SP: = SP - 1$	
	LOADI 0		
	STORE 1	$D(1): = 0$	initialize result
	LOADIN2 1	$ACC: = D(SP+1)$	read arg 1
	STORE 0	$D(0): = ACC$	initialize counter to arg 1
⟨*loop*⟩	$\texttt{JUMP}_{=}$ ⟨*end*⟩		if counter=0, we are finished
	LOADIN2 0	$ACC: = D(SP)$	read arg 2
	ADD 1	$ACC: = ACC + D(1)$	current sum increased by arg 2
	STORE 1	$D(1): = ACC$	cache result
	LOAD 0		
	SUBI 1		
	STORE 0	$D(0): = D(0) - 1$	decrease counter by 1
	JUMP *loop*		repeat addition
⟨*end*⟩	LOAD 1		load result
	STOREIN2 0		push it on stack
	inc $IN1$		
	JUMP ⟨*jt*⟩		back to jump table

Note that `mul` is the only instruction whose corresponding piece of code is not of the unit complexity. For the jump instructions, we do exactly what we would expect, we load the jump distance, add it to the register $IN1$, which we use to represent the `VM` program counter VPC. Incidentally, we can use the code for `jp` for the conditional jump `cjp`.

Control Instructions

label	instruction	effect	comment
⟨jp⟩	MOVE $IN1$ ACC	$ACC: = VPC$	
	STORE 0	$D(0): = ACC$	cache VPC
	LOADIN1 1	$ACC: = D(VPC+1)$	load i
	ADD 0	$ACC: = ACC + D(0)$	compute new VPC value
	MOVE ACC $IN1$	$IN1: = ACC$	update VPC
	JUMP ⟨*jt*⟩		jump back
⟨cjp⟩	dec $IN2$	$SP: = SP - 1$	update for pop
	LOADIN2 1	$ACC: = D(SP+1)$	pop value to ACC
	$\texttt{JUMP}_{=}$ ⟨jp⟩		perform jump if $ACC = 0$
	MOVE $IN1$ ACC		otherwise, go on
	ADDI 2		
	MOVE ACC $IN1$	$VPC: = VPC + 2$	point to next
	JUMP ⟨*jt*⟩		jump back

Imperative Stack Operations: peek

label	instruction	effect	comment
⟨peek⟩	MOVE $IN1$ ACC	$ACC: = IN1$	
	STORE 0	$D(0): = ACC$	cache VPC
	LOADIN1 1	$ACC: = D(VPC+1)$	load i
	MOVE ACC $IN1$	$IN1: = ACC$	
	inc $IN2$		prepare push
	LOADIN1 8	$ACC: = D(IN1+8)$	load $\mathcal{S}(i)$
	STOREIN2 0		push $\mathcal{S}(i)$
	LOAD 0	$ACC: = D(0)$	load old VPC
	ADDI 2		compute new value
	MOVE ACC $IN1$		update VPC
	JUMP ⟨*jt*⟩		jump back

Imperative Stack Operations: poke

label	instruction	effect	comment
⟨poke⟩	MOVE $IN1$ ACC	$ACC:=IN1$	
	STORE 0	$D(0):=ACC$	cache VPC
	LOADIN1 1	$ACC:=D(VPC+1)$	load i
	MOVE ACC $IN1$	$IN1:=ACC$	
	LOADIN2 0	$ACC:=\mathcal{S}(i)$	pop to ACC
	STOREIN1 8	$D(IN1+8):=ACC$	store in $\mathcal{S}(i)$
	dec $IN2$	$IN2:=IN2-1$	
	LOAD 0	$ACC:=D(0)$	get old VPC
	ADD 2	$ACC:=ACC+2$	add 2
	MOVE ACC $IN1$		update VPC
	JUMP ⟨jt⟩		jump back

We will now build a compiler for a simple imperative language to warm up to the task of building one for a functional one. We will write this compiler in SML, since we are most familiar with this. The first step is to define the language we want to talk about.

A very simple Imperative Programming Language

- Plan: Only consider the bare-bones core of a language. (we are only interested in principles)
  - We will call this language SW (Simple While Language)
  - no types: all values have type `int`, use 0 for `false` all other numbers for `true`.
  - only worry about abstract syntax (we do not want to build a parser) We will realize this as an SML data type.

The following slide presents the SML data types for SW programs.

Abstract Syntax of SW

```
type id  = string                 (* identifier          *)

datatype exp =                    (* expression          *)
   Con     of int                 (* constant            *)
 | Var     of id                  (* variable            *)
 | Add     of exp* exp            (* addition            *)
 | Sub     of exp * exp           (* subtraction         *)
 | Mul     of exp * exp           (* multiplication      *)
 | Leq     of exp * exp           (* less or equal test  *)

datatype sta =                    (* statement           *)
   Assign of id  * exp            (* assignment          *)
 | If     of exp * sta * sta      (* conditional         *)
 | While  of exp * sta            (* while loop          *)
 | Seq    of sta list             (* sequentialization   *)

type declaration = id * exp

type program  = declaration list * sta * exp
```


A SW program (see the next slide for an example) first declares a set of variables (type `declaration`), executes a statement (type `sta`), and finally returns an expression (type `exp`). Expressions of SW can read the values of variables, but cannot change them. The statements of SW can read and change the values of variables, but do not return values (as usual in imperative languages). Note that SW follows common practice in imperative languages and models the conditional as a statement.

Concrete vs. Abstract Syntax of a SW Program

```
var n:= 12; var a:= 1;
  while 2<=n do
  a:= a*n;
  n:= n-1
end
return a
```

```
([ ("n", Con 12), ("a", Con 1 ) ],
 While(Leq(Con 2, Var"n"),
 Seq [Assign("a", Mul(Var"a", Var"n")),
      Assign("n", Sub(Var"n", Con 1))]
 ),
 Var"a")
```

 102 JACOBS UNIVERSITY

As expected, the program is represented as a triple: the first component is a list of declarations, the second is a statement, and the third is an expression (in this case, the value of a single variable). We will use this example as the guiding intuition for building a compiler.

Before we can come to the implementation of the compiler, we will need an infrastructure for environments.

Needed Infrastructure: Environments

- ▷ Need a structure to keep track of the values of declared identifiers. (take shadowing into account)
- ▷ Definition 11.10: An environment is a finite partial function from keys (identifiers) to values.
- ▷ We will need the following operations on environments:
  - ▷ creation of an empty environment ($\leadsto$ the empty function)
  - ▷ insertion of a key/value pair $\langle k, v\rangle$ into an environment φ: ($\leadsto \varphi, [v/k]$)
  - ▷ lookup of the value v for a key k in φ ($\leadsto \varphi(k)$)
- ▷ Realization in SML by a structure with the following signature

```
type 'a env   (* a is the value type *)
exception Unbound of id   (* Unbound *)
val empty  : 'a env
val insert : id * 'a * 'a env -> 'a env (* id is the key type *)
val lookup : id * 'a env -> 'a
```

 103 JACOBS UNIVERSITY

We will also need an SML type for $\mathcal{L}$(VM) programs. Fortunately, this is very simple.

An SML Data Type for $\mathcal{L}(\mathtt{VM})$ Programs

```
type index = int
type noi   = int                   (* number of instructions *)

datatype instruction =
     con       of int
   | add | sub | mul     (* addition, subtraction, multiplication *)
   | leq                 (* less or equal test      *)
   | jp    of noi        (* unconditional jump      *)
   | cjp   of noi        (* conditional jump        *)
   | peek  of index      (* push value from stack   *)
   | poke  of index      (* update value in stack   *)
   | halt                (* halt machine            *)

type code = instruction list

fun wlen (xs:code) = foldl (fn (x,y) => wln(x)+y) 0 xs
fun wln(con _)=2 | wln(add)=1 | wln(sub)=1 | wln(mul)=1 | wln(leq)=1
  | wln(jp _)=2  | wln(cjp _)=2
  | wln(peek _)=2 | wln(poke _)=2 | wln(halt)=1
```

 104 JACOBS UNIVERSITY

The next slide has the main SML function for compiling SW programs. Its argument is a SW program (type program) and its result is an expression of type code, i.e. a list of $\mathcal{L}(\mathtt{VM})$ instructions. From there, we only need to apply a simple conversion (which we omit) to numbers to obtain $\mathcal{L}(\mathtt{VM})$ byte code.

Compiling SW programs

- ▷ SML function from SW programs (type program) to $\mathcal{L}(\mathtt{VM})$ programs (type code).
- ▷ uses three auxiliary functions for compiling declarations (compileD), statements (compileS), and expressions (compileE).
- ▷ these use an environment to relate variable names with their stack index.
- ▷ the initial environment is created by the declarations. (therefore compileD has an environment as return value)

```
type env = index env
fun compile ((ds,s,e) : program) : code =
  let
    val (cds, env) = compileD(ds, empty, ~1)
  in
    cds @ compileS(s,env) @ compileE(e,env) @ [halt]
  end
```

 105 JACOBS UNIVERSITY

The next slide has the function for compiling SW expressions. It is realized as a case statement over the structure of the expression.

Compiling SW Expressions

- constants are pushed to the stack.
- variables are looked up in the stack by the index determined by the environment (and pushed to the stack).
- arguments to arithmetic operations are pushed to the stack in reverse order.

```
fun compileE (e:exp, env:env) : code =
  case e of
    Con i       => [con i]
  | Var i       => [peek (lookup(i,env))]
  | Add(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [add]
  | Sub(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [sub]
  | Mul(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [mul]
  | Leq(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [leq]
```


Compiling SW statements is only slightly more complicated: the constituent statements and expressions are compiled first, and then the resulting code fragments are combined by $\mathcal{L}$(VM) control instructions (as the fragments already exist, the relative jump distances can just be looked up). For a sequence of statements, we just map `compileS` over it using the respective environment.

Compiling SW Statements

```
fun compileS (s:sta, env:env) : code =
    case s of
      Assign(i,e) => compileE(e, env) @ [poke (lookup(i,env))]
    | If(e,s1,s2) =>
       let
         val ce  = compileE(e, env)
         val cs1 = compileS(s1, env)
         val cs2 = compileS(s2, env)
       in
         ce @ [cjp (wlen cs1 + 4)] @ cs1 @ [jp (wlen cs2 + 2)] @ cs2
       end
    | While(e, s) =>
        let
         val ce = compileE(e, env)
         val cs = compileS(s, env)
        in
         ce @ [cjp (wlen cs + 4)] @ cs @ [jp (~(wlen cs + wlen ce + 2))]
        end
    | Seq ss      => foldr (fn (s,c) => compileS(s,env) @ c) nil ss
```


As we anticipated above, the `compileD` function is more complex than the other two. It gives $\mathcal{L}$(VM) program fragment and an environment as a value and takes a stack index as an additional argument. For every declaration, it extends the environment by the key/value pair k/v, where k is the variable name and v is the next stack index (it is incremented for every declaration). Then the expression of the declaration is compiled and prepended to the value of the recursive call.

Compiling SW Declarations

```
fun compileD (ds: declaration list, env:env, sa:index): code*env =
    case ds of
      nil         => (nil,env)
    | (i,e)::dr => let
                       val env'        = insert(i, sa+1, env)
                       val (cdr,env'') = compileD(dr, env', sa+1)
                     in
                       (compileE(e,env) @ cdr, env'')
                     end
```


This completes the compiler for SW (except for the byte code generator which is trivial and an implementation of environments, which is available elsewhere). So, together with the virtual machine for $\mathcal{L}$(VM) we discussed above, we can run SW programs on the register machine REMA.

If we now use the REMA simulator from exercise[6], then we can run SW programs on our computers outright. EdNote(6)

One thing that distinguishes SW from real programming languages is that it does not support procedure declarations. This does not make the language less expressive in principle, but makes structured programming much harder. The reason we did not introduce this is that our virtual machine does not have a good infrastructure that supports this. Therefore we will extend $\mathcal{L}$(VM) with new operations next.

Note that the compiler we have seen above produces $\mathcal{L}$(VM) programs that have what is often called "memory leaks". Variables that we declare in our SW program are not cleaned up before the program halts. In the current implementation we will not fix this (We would need an instruction for our VM that will "pop" a variable without storing it anywhere or that will simply decrease virtual stack pointer by a given value.), but we will get a better understanding for this when we talk about the static procedures next.

Compiling the Extended Example: A while Loop

▷ Example 11.11: Consider the following program that computes (12)! and the corresponding $\mathcal{L}$(VM) program:

`var n := 12; var a := 1;`	con 12 con 1
`while 2 <= n do (`	peek 0 con 2 leq cjp 18
`  a := a * n;`	peek 0 peek 1 mul poke 1
`  n := n - 1;`	con 1 peek 0 sub poke 0
`)`	jp −21
`return a;`	peek 1 halt

▷ Note that variable declarations only push the values to the stack, (memory allocation)

▷ they are referenced by peeking the respective stack position

▷ they are assigned by pokeing the stack position (must remember that)

Definition 11.12: In general, we need an environment and an instruction sequence to represent a procedure, but in many cases, we can get by with an instruction sequence alone. We speak of

[6]EdNote: include the exercises into the course materials and reference the right one here

static procedures in this case.

Example 11.13: Some programming languages like C or Pascal are designed so that all procedures can be represented as static procedures. SML and Java do not restrict themselves in this way.

We will now extend the virtual machine by four instructions that allow to represent static procedures with arbitrary numbers of arguments. We will explain the meaning of these extensions via an example: the procedure on the next slide, which computes 10^2.

Adding (Static) Procedures

- ▷ We have a full compiler for a very simple imperative programming language
- ▷ Problem: No support for subroutines/procedures. (no support for structured programming)
- ▷ Extensions to the Virtual Machine

```
type index = int
type noi   = int          (* number of instructions *)
type noa = int            (* number of arguments    *)
type ca    = int          (* code address           *)

datatype instruction =
   ...
   | proc    of noa*noi   (* begin of procedure code   *)
   | arg     of index     (* push value from frame     *)
   | call    of ca        (* call procedure            *)
   | return               (* return from procedure call *)
```


Translation of a Static Procedure

```
[proc 2 26,                            (*   fun exp(x,n) =        *)
 con 0, arg 2, leq, cjp 5,             (*     if n<=0             *)
 con 1, return,                        (*     then 1              *)
 con 1, arg 2, sub, arg 1,             (*     else x*exp(x,n-1)   *)
 call 0, arg 1, mul,
 return,                               (*  in                     *)
 con 2, con 10, call 0,                (*   exp(10,2)             *)
 halt]                                 (*  end                    *)
```

`proc` a l contains information about the number a of arguments and the length l of the procedure in the number of words needed to store it, together with the length of `proc` a l itself (3).

`arg` i pushes the i^{th} argument from the current frame to the stack.

`call` p pushes the current program address (opens a new frame), and jumps to the program address p

`return` takes the current frame from the stack, jumps to previous program address.

Static Procedures (Simulation)

```
proc 2 26,
 [con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

empty stack

- ▷ proc jumps over the body of the procedure declaration (with the help of its second argument.)

 ©: Michael Kohlhase 112

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, jp 13,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10
2

- ▷ We push the arguments onto the stack

 ©: Michael Kohlhase 113

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

32	0
10	-1
2	-2

- ▷ call pushes the return address (of the call statement in the $\mathcal{L}(\mathtt{VM})$ program)
- ▷ then it jumps to the first body instruction.

 ©: Michael Kohlhase 114

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq , cjp 5,
 con 1, return ,
 con 1, arg 2, sub , arg 1,
 call 0, arg 1, mul ,
 return ,
 con 2, con 10, call 0,
 halt]
```

2	
0	
32	0
10	-1
2	-2

▷ arg i pushes the i^{th} argument onto the stack

 115

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq , cjp 5,
 con 1, return ,
 con 1, arg 2, sub , arg 1,
 call 0, arg 1, mul ,
 return ,
 con 2, con 10, call 0,
 halt]
```

0	
32	0
10	-1
2	-2

▷ Comparison turns out false, so we push 0.

 116

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq , cjp 5,
 con 1, return ,
 con 1, arg 2, sub , arg 1,
 call 0, arg 1, mul ,
 return ,
 con 2, con 10, call 0,
 halt]
```

32	0
10	-1
2	-2

▷ cjp pops the truth value and jumps (on false).

 117

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

2	
1	
32	0
10	-1
2	-2

- ▷ we first push 1
- ▷ then we push the second argument (from the call frame position -2)

 118

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

1	
32	0
10	-1
2	-2

- ▷ we subtract

 119

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10	
1	
32	0
10	-1
2	-2

- ▷ then we push the second argument (from the call frame position -1)

 120

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

22	0
10	-1
1	-2
32	
10	
2	

▷ `call` jumps to the first body instruction,

▷ and pushes the return address (22 this time) onto the stack.

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

1	
0	
22	0
10	-1
1	-2
32	
10	
2	

▷ we augment the stack

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

22	0
10	-1
1	-2
32	
10	
2	

▷ we compare the top two, and jump ahead (on false)

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

1	
1	
22	0
10	-1
1	-2
32	
10	
2	

▷ we augment the stack again

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10	
0	
22	0
10	-1
1	-2
32	
10	
2	

▷ subtract and push the first argument

 125 JACOBS UNIVERSITY

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

22	0
10	-1
0	-2
22	
10	
1	
32	
10	
2	

▷ `call` pushes the return address and moves the current frame up

 126 JACOBS UNIVERSITY

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

0	
0	
22	0
10	-1
0	-2
22	
10	
1	
32	
10	
2	

▷ we augment the stack again,

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

22	0
10	-1
0	-2
22	
10	
1	
32	
10	
2	

▷ `leq` compares the top two numbers, `cjp` pops the result and does not jump.

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

1	
22	0
10	-1
0	-2
22	
10	
1	
32	
10	
2	

▷ we push the result value 1

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

1	
22	0
10	-1
1	-2
32	
10	
2	

▷ `return` interprets the top of the stack as the result,

▷ it jumps to the return address memorized right below the top of the stack,

▷ deletes the current frame

▷ and puts the result back on top of the remaining stack.

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10	
1	
22	0
10	-1
1	-2
32	
10	
2	

▷ `arg` pushes the first argument from the (new) current frame

 131

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10	
22	0
10	-1
1	-2
32	
10	
2	

▷ `mul` multiplies, pops the arguments and pushes the result.

 132

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

10	
32	0
10	-1
2	-2

- ▷ `return` interprets the top of the stack as the result,
- ▷ it jumps to the return address,
- ▷ deletes the current frame
- ▷ and puts the result back on top of the remaining stack.

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

100	
32	0
10	-1
2	-2

- ▷ we push argument 1 (in this case 10), multiply the top two numbers, and push the result to the stack

Static Procedures (Simulation)

```
[proc 2 26,
 con 0, arg 2, leq, cjp 5,
 con 1, return,
 con 1, arg 2, sub, arg 1,
 call 0, arg 1, mul,
 return,
 con 2, con 10, call 0,
 halt]
```

100

- ▷ `return` interprets the top of the stack as the result,
- ▷ it jumps to the return address (32 this time),
- ▷ deletes the current frame
- ▷ and puts the result back on top of the remaining stack (which is empty here).

Static Procedures (Simulation)

```
[proc 2 26,
con 0, arg 2, leq, cjp 5,
con 1, return,
con 1, arg 2, sub, arg 1,
call 0, arg 1, mul,
return,
con 2, con 10, call 0,
halt]
```

100

- ▷ we are finally done; the result is on the top of the stack. Note that the stack below has not changed.

What have we seen?

- ▷ The four new VM commands allow us to model static procedures.

  `proc` a l contains information about the number a of arguments and the length l of the procedure

  `arg` i pushes the i^{th} argument from the current frame to the stack. (Note that arguments are stored in reverse order on the stack)

  `call` p pushes the current program address (opens a new frame), and jumps to the program address p

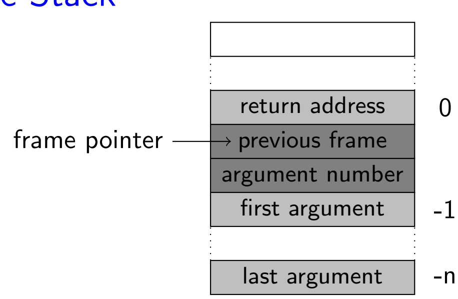
  `return` takes the current frame from the stack, jumps to previous program address. (which is cached in the frame)

- ▷ `call` and `return` jointly have the effect of replacing the arguments by the result of the procedure.

We will now extend our implementation of the virtual machine by the new instructions.

Realizing Call Frames on the Stack

- ▷ Problem: How do we know what the current frame is? (after all, `return` has to pop it)
- ▷ Idea: Maintain , and cache information about the previous frame and the number of arguments in the frame.



- ▷ Add two internal cells to the frame, that are hidden to the outside. The upper one is called the anchor cell.
- ▷ In the anchor cell we store the stack address of the anchor cell of the previous frame.
- ▷ The frame pointer points to the anchor cell of the uppermost frame.

Realizing proc

▷ proc a l jumps over the procedure with the help of the length l of the procedure.

label	instruction	effect	comment
⟨proc⟩	MOVE $IN1$ ACC	$ACC := VPC$	
	STORE 0	$D(0) := ACC$	cache VPC
	LOADIN1 2	$ACC := D(VPC+2)$	load length
	ADD 0	$ACC := ACC + D(0)$	compute new VPC value
	MOVE ACC $IN1$	$IN1 := ACC$	update VPC
	JUMP ⟨jt⟩		jump back

Realizing arg

▷ arg i pushes the i^{th} argument from the current frame to the stack.

▷ use the register $IN3$ for the frame pointer. (extend for first frame)

label	instruction	effect	comment
⟨arg⟩	LOADIN1 1	$ACC := D(VPC+1)$	load i
	STORE 0	$D(0) := ACC$	cache i
	MOVE $IN3$ ACC		
	STORE 1	$D(1) := FP$	cache FP
	SUBI 1		
	SUB 0	$ACC := FP - 1 - i$	load argument position
	MOVE ACC $IN3$	$FP := ACC$	move it to FP
	inc $IN2$	$SP := SP + 1$	prepare push
	LOADIN3 0	$ACC := D(FP)$	load arg i
	STOREIN2 0	$D(SP) := ACC$	push arg i
	LOAD 1	$ACC := D(1)$	load FP
	MOVE ACC $IN3$	$FP := ACC$	recover FP
	MOVE $IN1$ ACC		
	ADDI 2		
	MOVE ACC $IN1$	$VPC := VPC + 2$	next instruction
	JUMP ⟨jt⟩		jump back

Realizing call

▷ call p pushes the current program address, and jumps to the program address p (pushes the internal cells first!)

label	instruction	effect	comment
⟨call⟩	MOVE $IN1$ ACC		
	STORE 0	$D(0) := IN1$	cache current VPC
	inc $IN2$	$SP := SP + 1$	prepare push for later
	LOADIN1 1	$ACC := D(VPC + 1)$	load argument
	ADDI $2^{24} + 3$	$ACC := ACC + 2^{24} + 3$	add displacement and skip proc a l
	MOVE ACC $IN1$	$VPC := ACC$	point to the first instruction
	LOADIN1 -2	$ACC := D(VPC - 2)$	stealing a from proc a l
	STOREIN2 0	$D(SP) := ACC$	push the number of arguments
	inc $IN2$	$SP := SP + 1$	prepare push
	MOVE $IN3$ ACC	$ACC := IN3$	load FP
	STOREIN2 0	$D(SP) := ACC$	create anchor cell
	MOVE $IN2$ $IN3$	$FP := SP$	update FP
	inc $IN2$	$SP := SP + 1$	prepare push
	LOAD 0	$ACC := D(0)$	load VPC
	ADDI 2	$ACC := ACC + 2$	point to next instruction
	STOREIN2 0	$D(SP) := ACC$	push the return address
	JUMP ⟨jt⟩		jump back

 141 JACOBS UNIVERSITY

Note that with these instructions we have maintained the linear quality. Thus the virtual machine is still linear in the speed of the underlying register machine REMA.

Realizing return

▷ return takes the current frame from the stack, jumps to previous program address. (which is cached in the frame)

label	instruction	effect	comment
⟨return⟩	LOADIN2 0	$ACC := D(SP)$	load top value
	STORE 0	$D(0) := ACC$	cache it
	LOADIN2 -1	$ACC := D(SP - 1)$	load return address
	MOVE ACC $IN1$	$IN1 := ACC$	set VPC to it
	LOADIN3 -1	$ACC := D(FP - 1)$	load the number n of arguments
	STORE 1	$D(1) := D(FP - 1)$	cache it
	MOVE $IN3$ ACC	$ACC := FP$	$ACC = FP$
	SUBI 1	$ACC := ACC - 1$	$ACC = FP - 1$
	SUB 1	$ACC := ACC - D(1)$	$ACC = FP - 1 - n$
	MOVE ACC $IN2$	$IN2 := ACC$	$SP = ACC$
	LOADIN3 0	$ACC := D(FP)$	load anchor value
	MOVE ACC $IN3$	$IN3 := ACC$	point to previous frame
	LOAD 0	$ACC := D(0)$	load cached return value
	STOREIN2 0	$D(IN2) := ACC$	pop return value
	JUMP ⟨jt⟩		jump back

 142 JACOBS UNIVERSITY

Note that all the realizations of the $\mathcal{L}$(VM) instructions are linear code segments in the assembler code, so they can be executed in linear time. Thus the virtual machine language is only a constant factor slower than the clock speed of REMA. This is a characteristic of most virtual machines.

We now have the prerequisites to model procedures calls in a programming language. Instead of adding them to a imperative programming language, we will study them in the context of a functional programming language. For this we choose a minimal core of the functional programming language SML, which we will call μML. For this language, static procedures as we have seen them above are enough.

μML, a very simple Functional Programming Language

- Plan: Only consider the bare-bones core of a language (we only interested in principles)
  - We will call this language μML (micro ML)
  - no types: all values have type `int`, use 0 for `false` all other numbers for `true`.
  - only worry about abstract syntax (we do not want to build a parser) We will realize this as an SML data type.

Abstract Syntax of μML

```
type id = string                          (* identifier           *)

datatype exp =                            (* expression           *)
       Con  of int                          (* constant             *)
     | Id   of id                           (* argument             *)
     | Add  of exp * exp                    (* addition             *)
     | Sub  of exp * exp                    (* subtraction          *)
     | Mul  of exp * exp                    (* multiplication       *)
     | Leq  of exp * exp                    (* less or equal test   *)
     | App  of id  * exp list               (* application          *)
     | If   of exp * exp * exp              (* conditional          *)

type declaration = id * id list * exp

type program = declaration list * exp
```


Concrete vs. Abstract Syntax of μML

- A μML program first declares procedures, then evaluates expression for the return value.

```
let                          ([
  fun exp(x,n) =                ("exp", ["x", "n"],
    if n<=0                       If(Leq(Id"n", Con 0),
    then 1                             Con 1,
    else x*exp(x,n-1)                  Mul(Id"x", App("exp", [Id"x", Sub(Id"n", Con 1)]))))
in                              ],
  exp(2,10)                     App("exp", [Con 2, Con 10])
end                          )
```


The next step is to build a compiler for μML into programs in the extended $\mathcal{L}(\mathtt{VM})$. Just as above, we will write this compiler in SML.

Compiling μML Expressions

```
exception Error of string
datatype idType = Arg of index | Proc of ca
type env = idType env

fun compileE (e:exp, env:env, tail:code) : code =
  case e of
    Con i        => [con i] @ tail
  | Id i         => [arg((lookupA(i,env)))] @ tail
  | Add(e1,e2)   => compileEs([e1,e2], env) @ [add] @ tail
  | Sub(e1,e2)   => compileEs([e1,e2], env) @ [sub] @ tail
  | Mul(e1,e2)   => compileEs([e1,e2], env) @ [mul] @ tail
  | Leq(e1,e2)   => compileEs([e1,e2], env) @ [leq] @ tail
  | If(e1,e2,e3) => let
                      val c1 = compileE(e1,env,nil)
                      val c2 = compileE(e2,env,tail)
                      val c3 = compileE(e3,env,tail)
                    in if null tail
                      then c1 @ [cjp (4+wlen c2)] @ c2
                              @ [jp (2+wlen c3)] @ c3
                      else c1 @ [cjp (2+wlen c2)] @ c2 @ c3
                    end
  | App(i, es)   => compileEs(es,env) @ [call (lookupP(i,env))] @ tail
```


Compiling μML Expressions (Continued)

```
  and            (* mutual recursion with compileE *)
fun compileEs (es : exp list, env:env) : code =
  foldl (fn (e,c) => compileE(e, env, nil) @ c) nil es

fun lookupA (i,env) =
  case lookup(i,env) of
    Arg i => i
  | _     => raise Error("Argument expected: " ^ i)

fun lookupP (i,env) =
  case lookup(i,env) of
    Proc ca => ca
  | _       => raise Error("Procedure expected: " ^ i)
```


Compiling μML Expressions (Continued)

```
fun insertArgs' (i, (env, ai)) = (insert(i,Arg ai,env), ai+1)

fun insertArgs (is, env) = (foldl insertArgs' (env,1) is)

fun compileD (ds: declaration list, env:env, ca:ca) : code*env =
  case ds of
    nil          => (nil,env)
  | (i,is,e)::dr =>
      let
        val env'        = insert(i, Proc(ca+1), env)
        val env''       = insertArgs(is, env')
        val ce          = compileE(e, env'', [return])
        val cd          = [proc (length is, 3+wlen ce)] @ ce
                                 (* 3+wlen ce = wlen cd *)
        val (cdr,env'') = compileD(dr, env', ca + wlen cd)
      in
        (cd @ cdr, env'')
      end
```


Compiling μML

```
fun compile ((ds,e) : program) : code =
  let
    val (cds,env) = compileD(ds, empty, ~1)
  in
    cds @ compileE(e,env,nil) @ [halt]
  end
  handle
  Unbound i => raise Error("Unbound identifier: " ^ i)
```

 149 JACOBS UNIVERSITY

Where To Go Now?

- ▷ We have completed a μML compiler, which generates $\mathcal{L}(\mathtt{VM})$ code from μML programs.
- ▷ μML is minimal, but Turing-Complete (has conditionals and procedures)

 150 JACOBS UNIVERSITY

12 A theoretical View on Computation

Now that we have seen a couple of models of computation, computing machines, programs, ..., we should pause a moment and see what we have achieved.

What have we achieved

- ▷ what have we done? We have sketched
  - ▷ a concrete machine model (combinatory circuits)
  - ▷ a concrete algorithm model (assembler programs)
- ▷ Evaluation: (is this good?)
  - ▷ how does it compare with SML on a laptop?
  - ▷ Can we compute all (string/numerical) functions in this model?
  - ▷ Can we always prove that our programs do the right thing?
- ▷ Towards Theoretical Computer Science (as a tool to answer these)
  - ▷ look at a much simpler (but less concrete) machine model (Turing Machine)
  - ▷ show that TM can [encode/be encoded in] SML, assembler, Java,...
  - ▷ Conjecture: [Church/Turing] (unprovable, but accepted)
    All non-trivial machine models and programming languages are equivalent

 151 JACOBS UNIVERSITY

The idea we are going to pursue here is a very fundamental one for Computer Science: The Turing Machine. The main idea here is that we want to explore what the "simplest" (whatever that may mean) computing machine could be. The answer is quite surprising, we do not need wires, electricity, silicon, etc; we only need a very simple machine that can write and read to a tape following a simple set of rules.

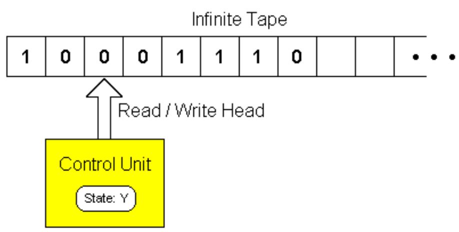
Of course such machines can be built (and have been), but this is not the important aspect here. Turing machines are mainly used for thought experiments, where we simulate them in our heads.

Note that the physical realization of the machine as a box with a (paper) tape is immaterial, it is inspired by the technology at the time of its inception (in the late 1940ties; the age of ticker-tape communincation).

Turing Machines

▷ Idea: Simulate a machine by a person executing a well-defined procedure!

- ▷ Setup: Person changes the contents of an infinite amount of ordered paper sheets that can contain one of a finite set of symbols.
- ▷ Memory: The person needs to remember one of a finite set of states
- ▷ Procedure: "If your state is 42 and the symbol you see is a '0' then replace this with a '1', remember the state 17, and go to the following sheet."



 152 JACOBS UNIVERSITY

More Precisely: Turing machine

▷ Definition 12.1: A Turing Machine consists of

- ▷ An infinite tape which is divided into cells, one next to the other (each cell contains a symbol from a finite alphabet $\mathcal{L}$ with $\#\mathcal{L} \geq 2$ and $0 \in \mathcal{L}$)
- ▷ A head that can read/write symbols on the tape and move left/right.
- ▷ A state register that stores the state of the Turing machine. (finite set of states, register initialized with a special start state)
- ▷ An action table (or transition function) that tells the machine what symbol to write, how to move the head and what its new state will be, given the symbol it has just read on the tape and the state it is currently in. (If no entry applicable the machine will halt)

Note: every part of the machine is finite, but it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.

▷

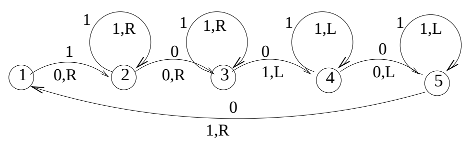
 153 JACOBS UNIVERSITY

Turning Machine Example 12.2: with Alphabet $\{0,1\}$

- ▷ Given: a series of 1s on the tape (with head initially on the leftmost)
- ▷ Computation: doubles the 1's with a 0 in between, i.e., "111" becomes "1110111".
- ▷ The set of states is $\{s_1, s_2, s_3, s_4, s_5\}$ (s_1 start state)
- ▷ actions:

Old	Read	Wr.	Mv.	New	Old	Read	Wr.	Mv.	New
s_1	1	0	R	s_2	s_4	1	1	L	s_4
s_2	1	1	R	s_2	s_4	0	0	L	s_5
s_2	0	0	R	s_3	s_5	1	1	L	s_5
s_3	1	1	R	s_3	s_5	0	1	R	s_1
s_3	0	1	L	s_4					

- ▷ state machine:



Example Computation

- ▷ $\mathcal{T}$ starts out in s_1, replaces the first 1 with a 0, then
- ▷ uses s_2 to move to the right, skipping over 1's and the first 0 encountered.
- ▷ s_3 then skips over the next sequence of 1's (initially there are none) and replaces the first 0 it finds with a 1.
- ▷ s_4 moves back left, skipping over 1's until it finds a 0 and switches to s_5.

Step	State	Tape	Step	State	Tape
1	s_1	$\boxed{1}$1	9	s_2	10$\boxed{0}$1
2	s_2	0$\boxed{1}$	10	s_3	100$\boxed{1}$
3	s_2	01$\boxed{0}$	11	s_3	1001$\boxed{0}$
4	s_3	010$\boxed{0}$	12	s_4	100$\boxed{1}$1
5	s_4	01$\boxed{0}$1	13	s_4	10$\boxed{0}$11
6	s_5	0$\boxed{1}$01	14	s_5	1$\boxed{0}$011
7	s_5	$\boxed{0}$101	15	s_1	11$\boxed{0}$11
8	s_1	1$\boxed{1}$01		— halt —	

- ▷ s_5 then moves to the left, skipping over 1's until it finds the 0 that was originally written by s_1.
- ▷ It replaces that 0 with a 1, moves one position to the right and enters s1 again for another round of the loop.
- ▷ This continues until s_1 finds a 0 (this is the 0 right in the middle between the two strings of 1's) at which time the machine halts

What can Turing Machines compute?

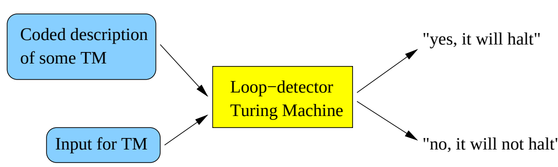
- Empirically: anything any other program can also compute
  - Memory is not a problem (tape is infinite)
  - Efficiency is not a problem (purely theoretical question)
  - Data representation is not a problem (we can use binary, or whatever symbols we like)
- All attempts to characterize computation have turned out to be equivalent
  - primitive recursive functions ([Gödel, Kleene])
  - lambda calculus ([Church])
  - Post production systems ([Post])
  - Turing machines ([Turing])
  - Random-access machine
- conjecture 12.3: ([Church/Turing])

  (unprovable, but accepted)

  Anything that can be computed at all, can be computed by a Turing Machine

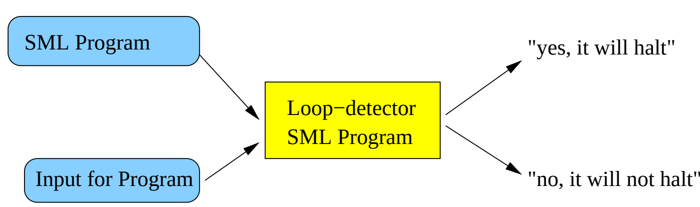
Is there anything that cannot be computed by a TM

- Theorem 12.4: *No Turing machine can infallibly tell if another Turing machine will get stuck in an infinite loop on some given input.*



- Proof:

P.1 let's do the argument with SML instead of a TM
assume that there is a loop detector program written in SML



□

Testing the Loop Detector Program Proof:

P.1 The general shape of the Loop detector program

```
fun will_halt(program,data) =
    ... lots of complicated code ...
    if ( ... more code ...) then true else false;
will_halt : (int -> int) -> int -> bool
```

test programs	behave exactly as we anticipated

`fun halter (n) = 1;`
`halter : int -> int`
`fun looper (n) = looper(n+1);`
`looper : int -> int`
	`will_halt(halter,1);`
	`val true : bool`
	`will_halt(looper,1);`
	`val false : bool`

P.2 Consider the following Program

```
function turing (prog) = if will_halt(prog,prog) then looper(1) else 1;
```

P.3 Yeah, so what? what happens, if we feed the `turing` function to itself? □

What happens indeed? Proof:

P.1
```
function turing (prog) = if will\_halt(prog,prog) then looper(1) else 1;
```

the `turing` function uses `will_halt` to analyze the function given to it.

▷ If the function halts when fed itself as data, the `turing` function goes into an infinite loop.

▷ If the function goes into an infinite loop when fed itself as data, the `turing` function immediately halts.

P.2 But if the function happens to be the `turing` function itself, then

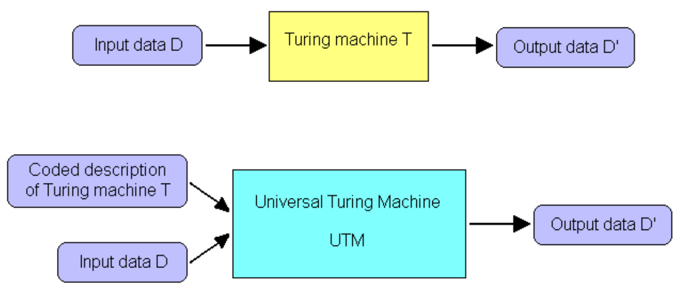
▷ the `turing` function goes into an infinite loop if the `turing` function halts (when fed itself as input)

▷ the `turing` function halts if the `turing` function goes into an infinite loop (when fed itself as input)

P.3 This is a blatant logical contradiction! (Thus there cannot be a `will_halt` function) □

Universal Turing machines

- Note: A Turing machine computes a fixed partial string function.
- In that sense it behaves like a computer with a fixed program.
- Idea: we can encode the action table of any Turing machine in a string.
  - try to construct a Turing machine that expects on its tape
  - a string describing an action table followed by
  - a string describing the input tape, and then
  - computes the tape that the encoded Turing machine would have computed.
- Theorem 12.5: *such a Turing machine is indeed possible (e.g. with 2 states, 18 symbols)*
- Definition 12.6: call it a universal Turing machine. (it can simulate any TM)



  - UTM accepts a coded description of a Turing machine and simulates the behavior of the machine on the input data.
  - The coded description acts as a program that the UTM executes, the UTM's own internal program is fixed.
- The existence of the UTM is what makes computers fundamentally different from other machines such as telephones, CD players, VCRs, refrigerators, toaster-ovens, or cars.

 160 JACOBS UNIVERSITY

13 Problem Solving

In this section, we will look at a class of algorithms called search algorithms. These are algorithms that help in quite general situations, where there is a precisely described problem, that needs to be solved.

Before we come to the algorithms, we need to get a grip on the problems themselves, and the problem solving process.

The first step is to classify the problem solving process by the amount of knowledge we have available. It makes a difference, whether we know all the factors involved in the problem before we actually are in the situation. In this case, we can solve the problem in the abstract, i.e. make a plan before we actually enter the situation (i.e. offline), and then when the problem arises, only execute the plan. If we do not have complete knowledge, then we can only make partial plans, and have to be in the situation to obtain new knowledge (e.g. by observing the effects of our actions or the actions of others). As this is much more difficult we will restrict ourselves to solving.

Problem solving

- Problem: Find algorithms that help solving problems in general
- Idea: If we can describe/represent problems in a standardized way, we may have a chance to find general algorithms.

  We will use the following two concepts to describe problems

  States A set of possible situations in in our problem domain

  Actions A set of possible actions that get us from one state to another.

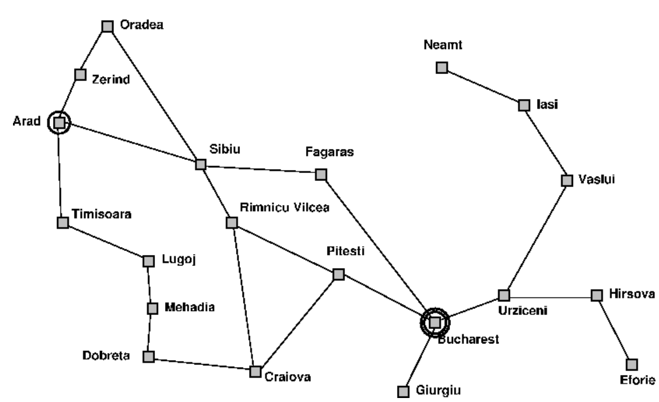
  Using these, we can view a sequence of actions as a solution, if it brings us into a situation, where the problem is solved.
- Definition 13.1: Offline problem solving: Acting only with complete knowledge of problem and solution
- Definition 13.2: Online problem solving: Acting without complete knowledge
- Here: we are concerned with offline problem solving only.

 161

We will use the following problem as a running example. It is simple enough to fit on one slide and complex enough to show the relevant features of the problem solving algorithms we want to talk about.

Example: Traveling in Romania

- Scenario: On holiday in Romania; currently in Arad, Flight leaves tomorrow from Bucharest.
- Formulate problem: *States*: various cities *Actions*: drive between cities
- Solution: Appropriate sequence of cities, e.g.: Arad, Sibiu, Fagaras, Bucharest



 162

Problem Formulation

- The problem formulation models the situation at an appropriate level of abstraction. (do not model things like "put on my left sock", etc.)
  - it describes the initial state (we are in Arad)
  - it also limits the objectives. (excludes, e.g. to stay another couple of weeks.)
- Finding the right level of abstraction and the required (not more!) information is often the key to success.
- Definition 13.3: A problem (formulation) $\mathcal{P} := \langle \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ consists of a set $\mathcal{S}$ of states and a set $\mathcal{O}$ of operators that specify how states can be accessed from each other. Certain states in $\mathcal{S}$ are designated as goal states ($\mathcal{G} \subseteq \mathcal{S}$) and there is a unique initial state $\mathcal{I}$.
- Definition 13.4: A solution for a problem $\mathcal{P}$ consists of a sequence of actions that bring us from $\mathcal{I}$ to a goal state.

 163

Problem types

- Single-state problem
  - observable (at least the initial state)
  - deterministic (i.e. the successor of each state is determined)
  - static (states do not change other than by our own actions)
  - discrete (a countable number of states)
- Multiple-state problem:
  - initial state not/partially observable (multiple initial states?)
  - deterministic, static, discrete
- Contingency problem:
  - non-deterministic (solution can branch, depending on contingencies)
  - unknown state space(like a baby, agent has to learn about states and actions)

 164

We will explain these problem types with another example. The problem $\mathcal{P}$ is very simple: We have a vacuum cleaner and two rooms. The vacuum cleaner is in one room at a time. The floor can be dirty or clean.

The possible states are determined by the position of the vacuum cleaner and the information, whether each room is dirty or not. Obviously, there are eight states: $\mathcal{S} = \{1,2,3,4,5,6,7,8\}$ for simplicity.

The goal is to have both rooms clean, the vacuum cleaner can be anywhere. So the set $\mathcal{G}$ of goal states is $\{\mathbf{7}, \mathbf{8}\}$. In the single-state version of the problem, $[right, suck]$ shortest solution, but $[suck, right, suck]$ is also one. In the multiple-state version we have $[right\{(2,4,6,8)\}, suck\{(4,8)\}, left\{(3,7)\}, suck\{(7)\}]$.

Example: vacuum-cleaner world

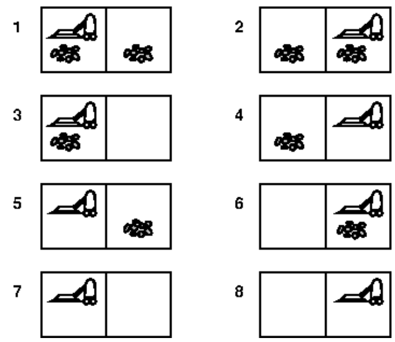
Single-state Problem:

- ▷ Start in *5*
- ▷ Solution: $[right, suck]$

Multiple-state Problem:

- ▷ Start in $\{1,2,3,4,5,6,7,8\}$
- ▷ Solution: $[right, suck, left, suck]$

$$\begin{array}{ll} right & \rightarrow \{2,4,6,8\} \\ suck & \rightarrow \{4,8\} \\ left & \rightarrow \{3,7\} \\ suck & \rightarrow \{7\} \end{array}$$

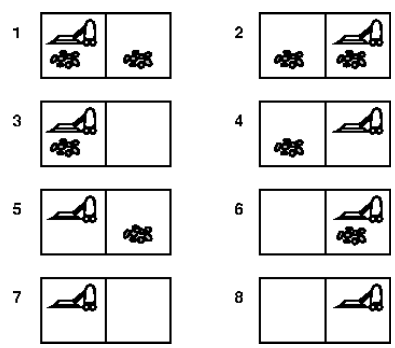


Example: vacuum-cleaner world (continued)

Contingency Problem:

- ▷ Murphy's Law: $suck$ can dirty a clean carpet
- ▷ Local sensing: *dirty*/*not dirty* at location only
- ▷ Start in: $\{1,3\}$
- ▷ Solution: $[suck, right, suck]$

$$\begin{array}{ll} suck & \rightarrow \{5,7\} \\ right & \rightarrow \{6,8\} \\ suck & \rightarrow \{6,8\} \end{array}$$



better: $[suck, right, \mathbf{if}\ dirt\ \mathbf{then}\ suck]$ (decide whether in 6 or 8 using local sensing)

In the contingency version of $\mathcal{P}$ a solution is the following: $[suck\{(5,7)\}, right \rightarrow \{(6,8)\}, suck \rightarrow \{(\mathbf{6},8)\}]$, $[suck\{(\mathbf{5},7)\}]$, etc. Of course, local sensing can help: narrow $\{6,8\}$ to $\{6\}$ or $\{8\}$, if we are in the first, then suck.

Single-state problem formulation

- ▷ Defined by the following four items
  1. Initial state: (e.g. *Arad*)
  2. Successor function S: (e.g. $S(Arad) = \{\langle goZer, Zerind\rangle, \langle goSib, Sibiu\rangle, \ldots\}$)
  3. Goal test: (e.g. $x = Bucharest$ (explicit test); $noDirt(x)$ (implicit test))
  4. Path cost (optional): (e.g. sum of distances, number of operators executed, etc.)
- ▷ Solution: A sequence of operators leading from the initial state to a goal state

"Path cost": There may be more than one solution and we might want to have the "best" one in a certain sense.

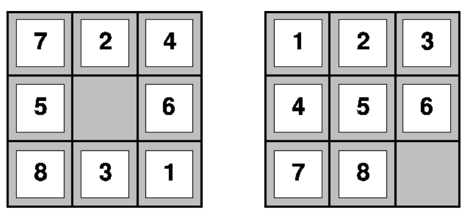
Selecting a state space

- Abstraction: Real world is absurdly complex
  State space must be abstracted for problem solving
- (Abstract) state: Set of real states
- (Abstract) operator: Complex combination of real actions
- Example: *Arad* → *Zerind* represents complex set of possible routes
- (Abstract) solution: Set of real paths that are solutions in the real world

"State": e.g., we don't care about tourist attractions found in the cities along the way. But this is problem dependent. In a different problem it may well be appropriate to include such information in the notion of state.

"Realizability": one could also say that the abstraction must be sound wrt. reality.

Example: The 8-puzzle

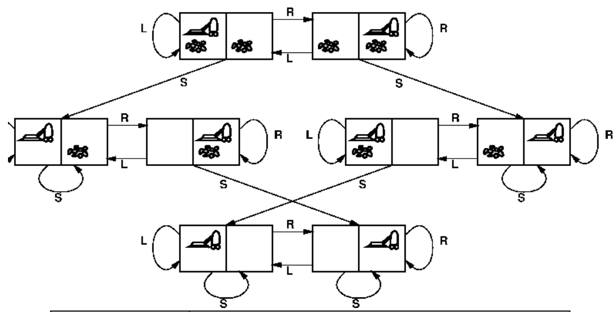


Start State Goal State

States	integer locations of tiles
Actions	$left, right, up, down$
Goal test	= goal state?
Path cost	1 per move

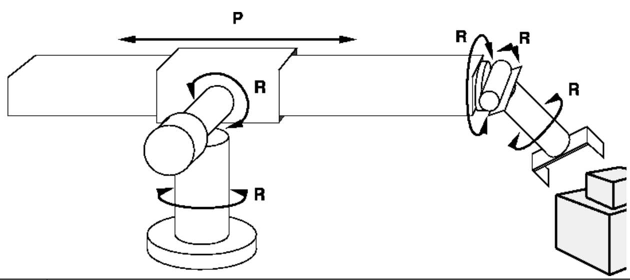
How many states are there? N factorial, so it is not obvious that the problem is in NP. One needs to show, for example, that polynomial length solutions do always exist. Can be done by combinatorial arguments on state space graph (really ?).

Example: Vacuum-cleaner



States	integer dirt and robot locations
Actions	$left, right, suck, noOp$
Goal test	$notdirty?$
Path cost	1 per operation (0 for $noOp$)

Example: Robotic assembly



States	real-valued coordinates of robot joint angles and parts of the object to be assembled
Actions	continuous motions of robot joints
Goal test	assembly complete?
Path cost	time to execute

14 Midterm Analysis

Cheating

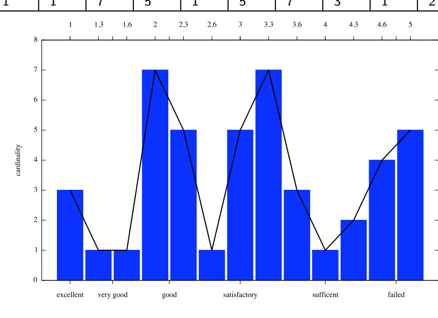
- ▷ Remember the code of academic integrity? (you've signed it)
- ▷ Crucial elements are
  - ▷ honest academic work
  - ▷ respect intellectual property of others
- ▷ Please keep this in mind!
- ▷ Copying from others is a bad idea! (and you know it)
  - ▷ It violates the AI code
  - ▷ You don't learn anything by doing it (in the end you hurt yourself)

  JACOBS UNIVERSITY

Mid-Term Results

- ▷ Midterm: 7 Problems, 30 Points, 27 counted, 3 bonus
- ▷ This has been too long (Apologies for this)
- ▷ New score: 22 points
- ▷ Grades: Average: 14 points ($\hat{=}$ 65%), grade 3.00 ("satisfactory")

Performance	exc.	very good		good		satisfactory			sufficient			failing		
IUB Grade	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33	3.67	4.00	4.33	4.67	5.00	∅
Cardinality	3	1	1	7	5	1	5	7	3	1	2	4	5	6



  JACOBS UNIVERSITY

Procedure, Consequences

- ▷ Procedure (So that we do not have any accusations of late edits!)
  - ▷ We will give back the graded tests at April, 3, 13:00-14:00. (as announced on the forums)
  - ▷ You will check the grading, points summation, ...
  - ▷ We will answer questions, and correct mistakes.
  - ▷ You will take home the test, when you leave the room the grade is final!
- ▷ Consequences
  - ▷ You need more practice (and to practice more!)
  - ▷ We will provide a repository of previous exams (so you have plenty to practice with)
  - ▷ You need better time management (Don't panic!)
  - ▷ Take full advantage of tutorials and TAs (some TAs report you are not well prepared)
  - ▷ We are here to help you (we don't aim at making you fail)

 174 JACOBS UNIVERSITY

15 Search

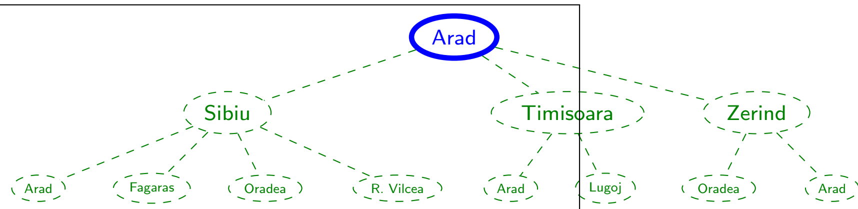
Tree search algorithms

- ▷ Simulated exploration of state space in a search tree by generating successors of already-explored states (Offline Algorithm)

```
procedure TREE-SEARCH(problem, strategy)                ▷ a solution or failure
    initialize the search tree using the initial state of problem
    loop
        if there are no candidates for expansion then
            return failure
        end if
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then
            return the corresponding solution
        else
            expand the node and add the resulting nodes to the search tree
        end if
    end loop
end procedure
```

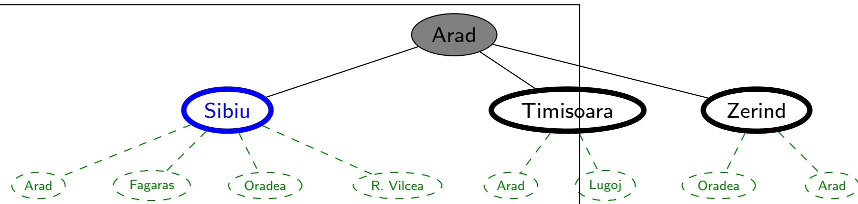
 175 JACOBS UNIVERSITY

Tree Search: Example



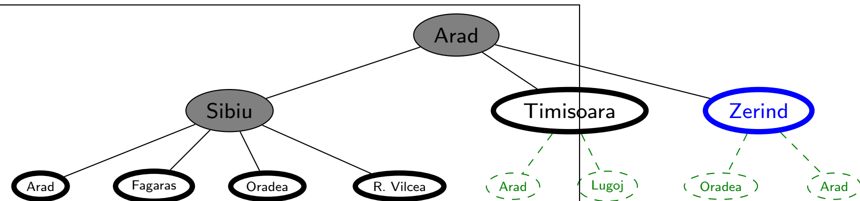
 176

Tree Search: Example



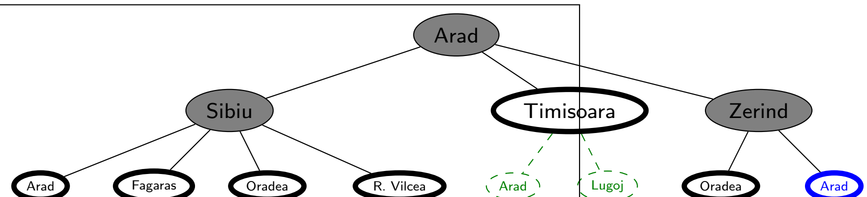
 177

Tree Search: Example



 178

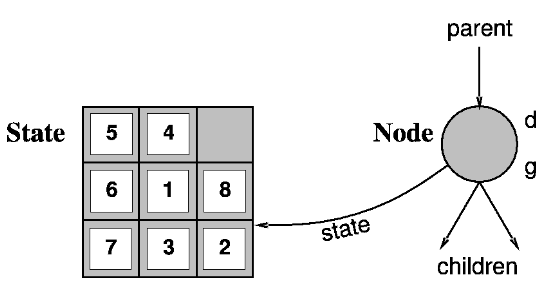
Tree Search: Example



 179

Implementation: States vs. nodes

- ▷ State: A (representation of) a physical configuration
- ▷ Node: A data structure constituting part of a search tree (includes *parent*, *children*, *depth*, *path cost*, etc.)



 180

Implementation of search algorithms

```
procedure TREE_SEARCH(problem, strategy)
    fringe ← insert(make_node(initial_state(problem)))
    loop
        if fringe empty then
            fail
        end if
        node ← first(fringe, strategy)
        if NodeTest(State(node)) then
            return State(node)
        else
            fringe ← insert_all(expand(node, problem), strategy)
        end if
    end loop
end procedure
```

▷ Definition 15.1: The fringe is a list nodes not yet considered. It is ordered by the search strategy (see below)

STATE gives the state that is represented by *node*

EXPAND = creates new nodes by applying possible actions to *node*

A node is a data structure representing states, will be explained in a moment.

MAKE-QUEUE creates a queue with the given elements.

fringe holds the queue of nodes not yet considered.

REMOVE-FIRST returns first element of queue and as a side effect removes it from *fringe*.

STATE gives the state that is represented by *node*.

EXPAND applies all operators of the problem to the current node and yields a set of new nodes.

INSERT inserts an element into the current *fringe* queue. This can change the behavior of the search.

INSERT-ALL Perform INSERT on set of elements.

Search strategies

▷ Strategy: Defines the order of node expansion

▷ Important properties of strategies:

completeness	does it always find a solution if one exists?
time complexity	number of nodes generated/expanded
space complexity	maximum number of nodes in memory
optimality	does it always find a least-cost solution?

▷ Time and space complexity measured in terms of:

b	maximum branching factor of the search tree
d	depth of a solution with minimal distance to root
m	maximum depth of the state space (may be ∞)

Complexity means here always *worst-case* complexity.

Note that there can be infinite branches, see the search tree for Romania.

16 Uninformed Search Strategies

Uninformed search strategies

- ▷ Definition 16.1: (Uninformed search)

  Use only the information available in the problem definition

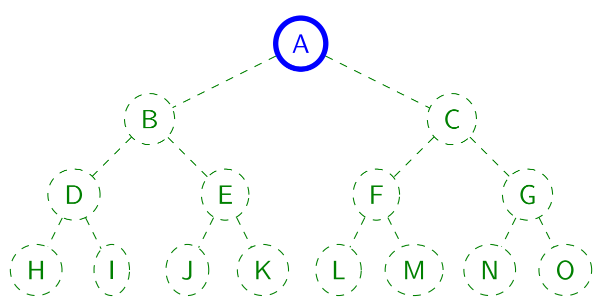
- ▷ Frequently used strategies:
  - ▷ Breadth-first search
  - ▷ Uniform-cost search
  - ▷ Depth-first search
  - ▷ Depth-limited search
  - ▷ Iterative deepening search

The opposite of uninformed search is informed or *heuristic* search. In the example, one could add, for instance, to prefer cities that lie in the general direction of the goal (here SE).

Uninformed search is important, because many problems do not allow to extract good heuristics.

Breadth-first search

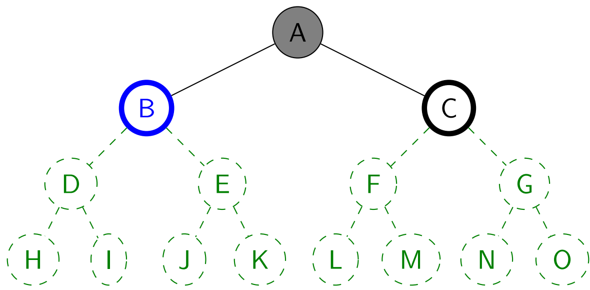
- ▷ Idea: Expand shallowest unexpanded node
- ▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



Breadth-First Search

▷ Idea: Expand shallowest unexpanded node

▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue

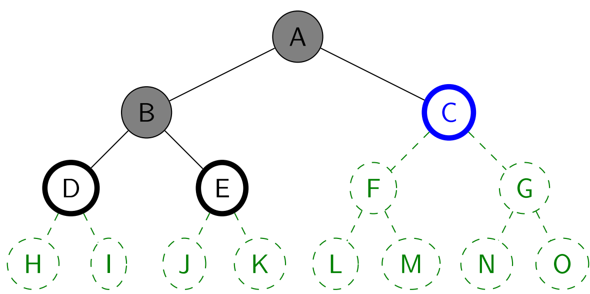


 185

Breadth-First Search

▷ Idea: Expand shallowest unexpanded node

▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue

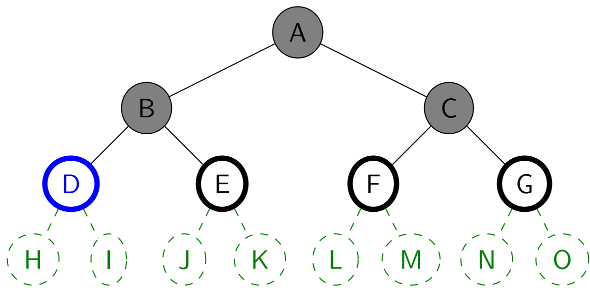


 186

Breadth-First Search

▷ Idea: Expand shallowest unexpanded node

▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue

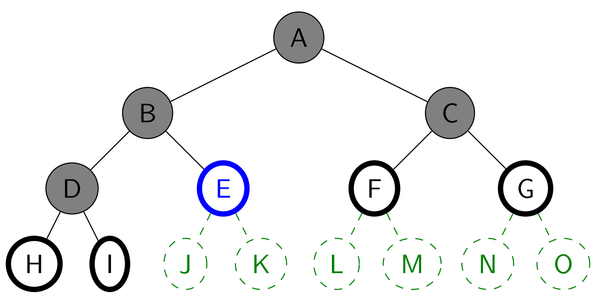


 187

Breadth-First Search

▷ Idea: Expand shallowest unexpanded node

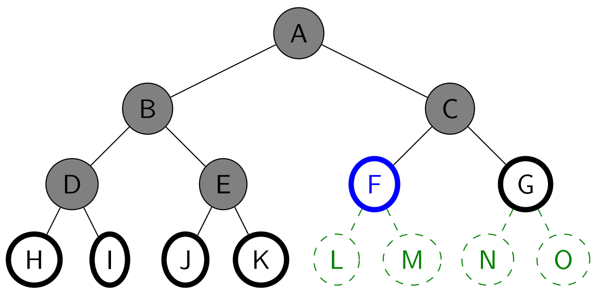
▷ Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



 188

Breadth-First Search

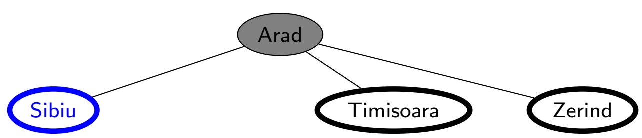
- Idea: Expand shallowest unexpanded node
- Implementation: *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



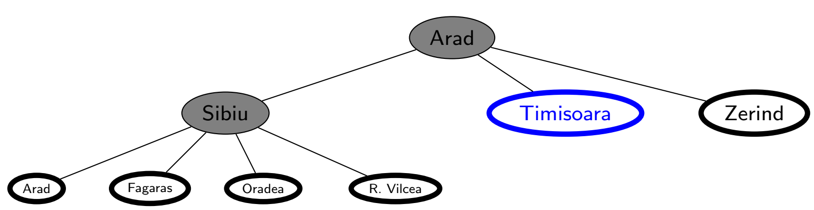
We will now apply the breadth-first search strategy to our running example: Traveling in Romania. Note that we leave out the green dashed nodes that allow us a preview over what the search tree will look like (if expanded). This gives a much more realistic view.

Breadth-First Search: Romania

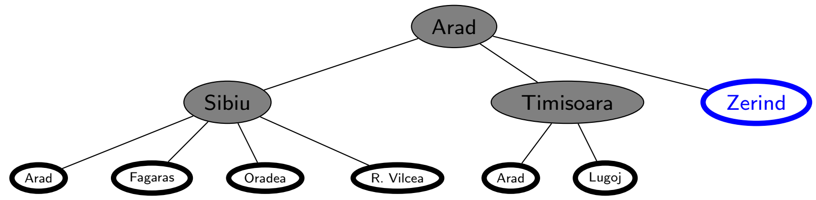
Breadth-First Search: Romania



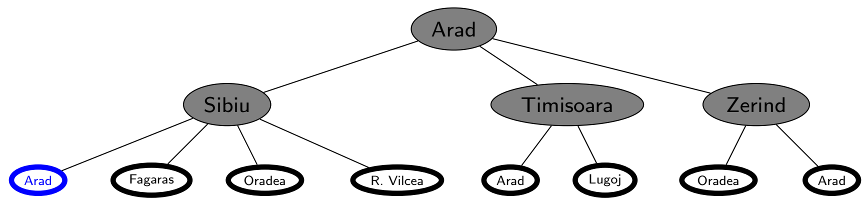
Breadth-First Search: Romania



Breadth-First Search:Romania



Breadth-First Search:Romania



Breadth-first search: Properties

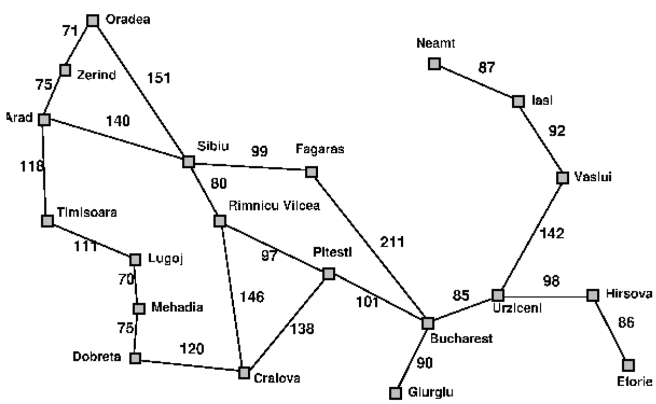
Complete	Yes (if b is finite)
Time	$1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) \in O(b^{d+1})$ i.e. exponential in d
Space	$O(b^{d+1})$ (keeps every node in memory)
Optimal	Yes (if cost = 1 per step), not optimal in general

- ▷ Disadvantage: Space is the big problem (can easily generate nodes at 5MB/sec so 24hrs = 430GB)
- ▷ Optimal?: if cost varies for different steps, there might be better solutions below the level of the first solution.
- ▷ An alternative is to generate *all* solutions and then pick an optimal one. This works only, if m is finite.

The next idea is to let cost drive the search. For this, we will need a non-trivial cost function: we will take the distance between cities, since this is very natural. Alternatives would be the driving time, train ticket cost, or the number of tourist attractions along the way.

Of course we need to update our problem formulation with the necessary information.

Romania with Step Costs as Distances



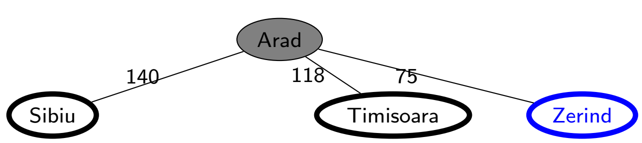
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Uniform-cost search

- ▷ Idea: Expand least-cost unexpanded node
- ▷ Implementation: $fringe$ is queue ordered by increasing path cost.
- ▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)

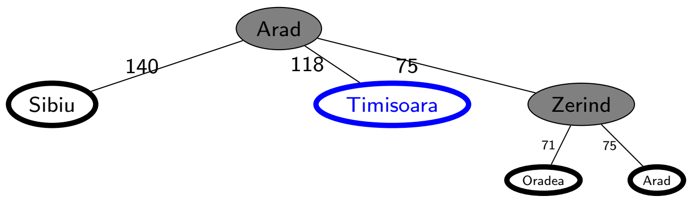
Uniform Cost Search: Romania

- ▷ Idea: Expand least-cost unexpanded node
- ▷ Implementation: $fringe$ is queue ordered by increasing path cost.
- ▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)



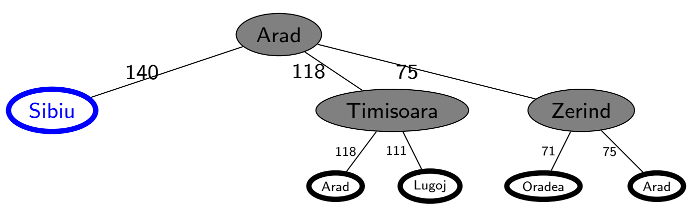
Uniform Cost Search: Romania

- ▷ Idea: Expand least-cost unexpanded node
- ▷ Implementation: $fringe$ is queue ordered by increasing path cost.
- ▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)



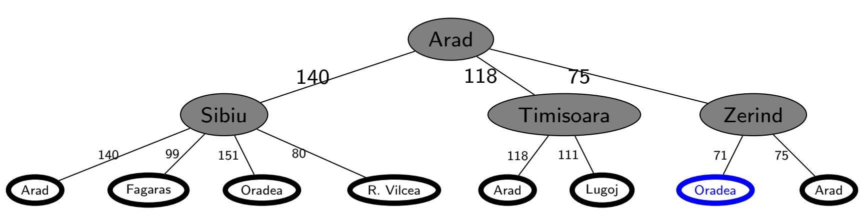
Uniform Cost Search: Romania

- ▷ Idea: Expand least-cost unexpanded node
- ▷ Implementation: $fringe$ is queue ordered by increasing path cost.
- ▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)



Uniform Cost Search: Romania

- ▷ Idea: Expand least-cost unexpanded node
- ▷ Implementation: $fringe$ is queue ordered by increasing path cost.
- ▷ Note: Equivalent to breadth-first search if all step costs are equal (DFS: see below)



Note that we must sum the distances to each leaf. That is, we go back to the first level after step 3.

Uniform-cost search: Properties

Complete	Yes (if step costs $\geq \epsilon > 0$)
Time	number of nodes with past-cost less than that of optimal solution
Space	number of nodes with past-cost less than that of optimal solution
Optimal	Yes

If step cost is negative, the same situation as in breadth-first search can occur: later solutions may be cheaper than the current one.

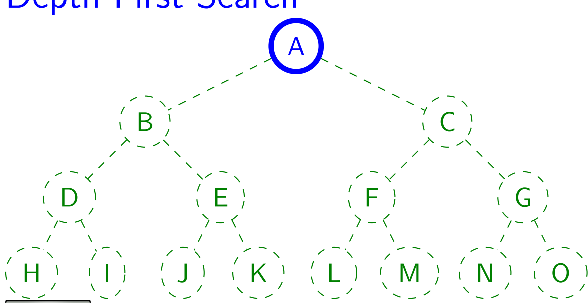
If step cost is 0, one can run into infinite branches. UC search then degenerates into depth-first search, the next kind of search algorithm. Even if we have infinite branches, where the sum of step costs converges, we can get into trouble[7] EdNote(7)

Worst case is often worse than BF search, because large trees with small steps tend to be searched first. If step costs are uniform, it degenerates to BF search.

Depth-first search

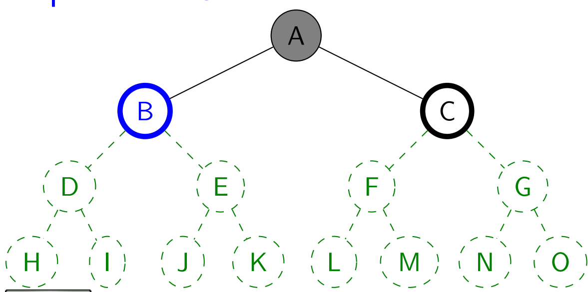
- ▷ Idea: Expand deepest unexpanded node
- ▷ Implementation: *fringe* is a LIFO queue (a stack), i.e. successors go in at front of queue
- ▷ Note: Depth-first search can perform infinite cyclic excursions
  Need a finite, non-cyclic search space (or repeated-state checking)

Depth-First Search



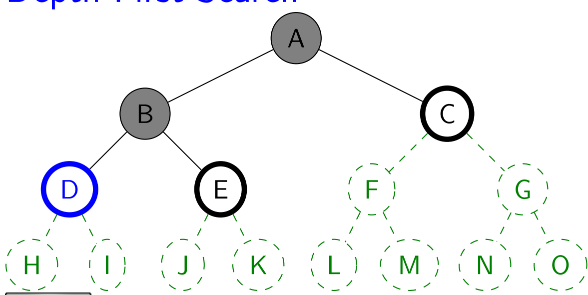
[7] EdNote: say how

Depth-First Search



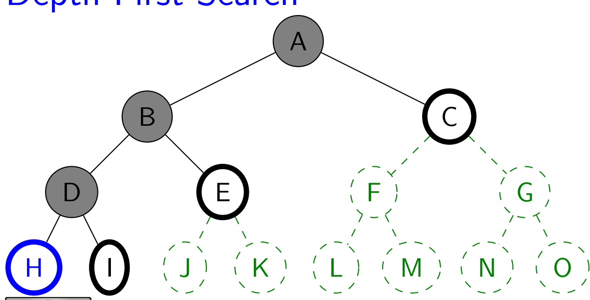
 205 JACOBS UNIVERSITY

Depth-First Search



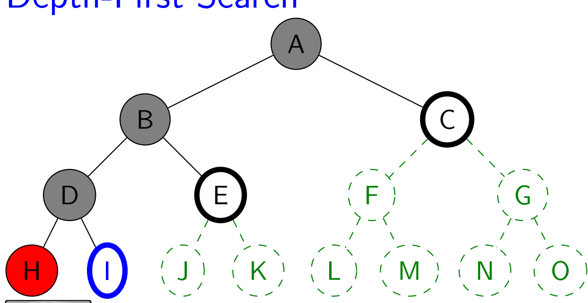
 206 JACOBS UNIVERSITY

Depth-First Search



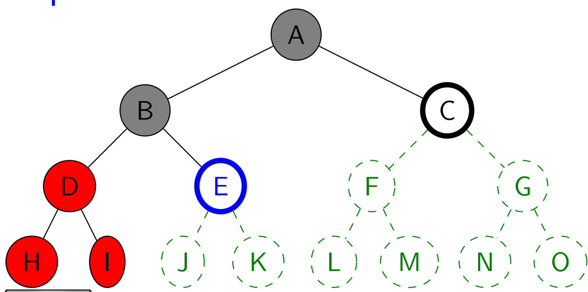
 207 JACOBS UNIVERSITY

Depth-First Search

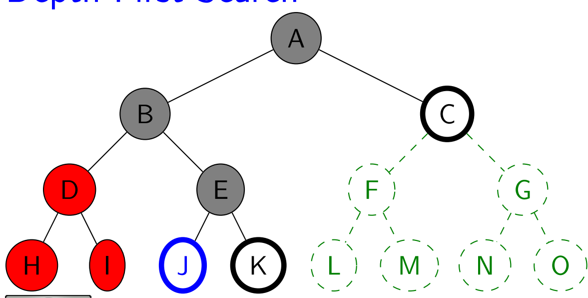


 208 JACOBS UNIVERSITY

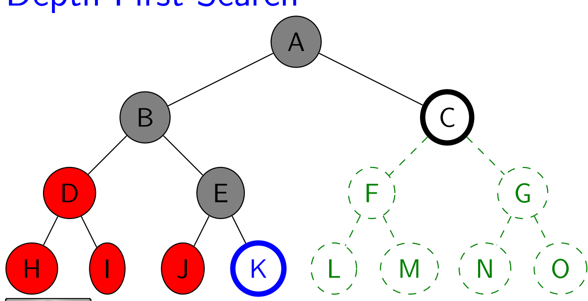
Depth-First Search



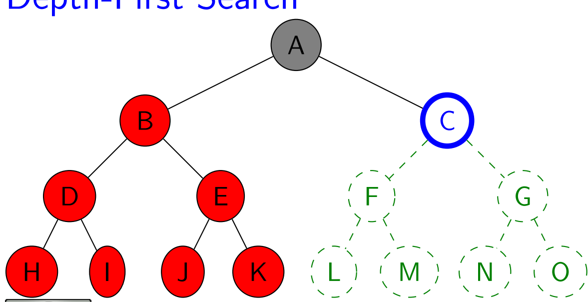
Depth-First Search



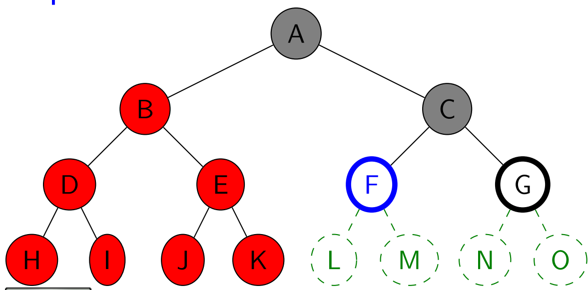
Depth-First Search



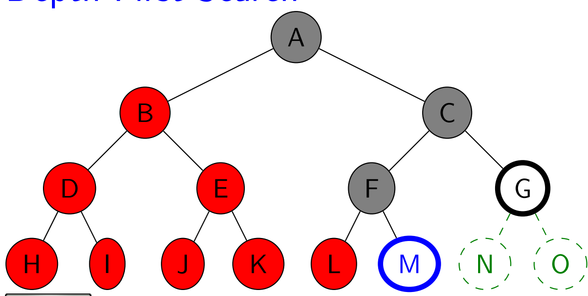
Depth-First Search



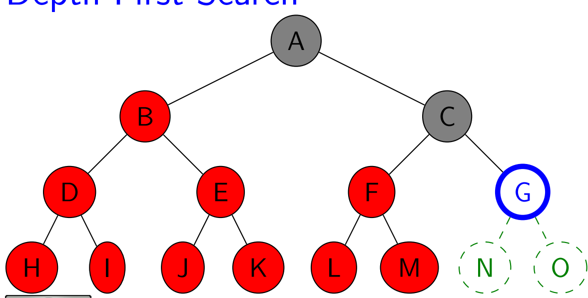
Depth-First Search



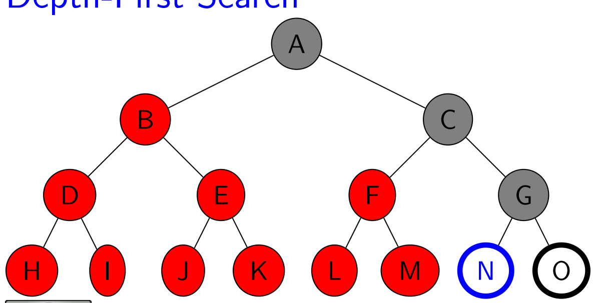
Depth-First Search



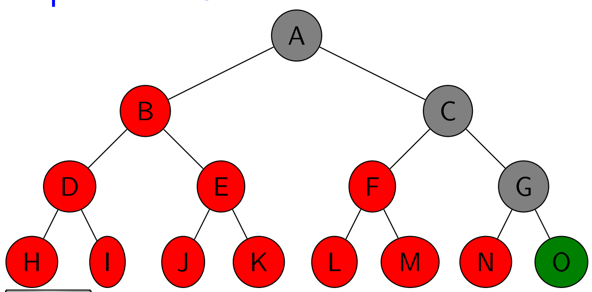
Depth-First Search



Depth-First Search

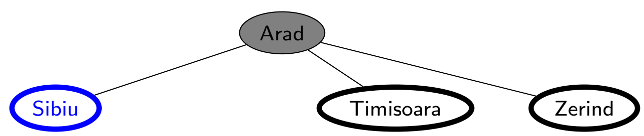


Depth-First Search

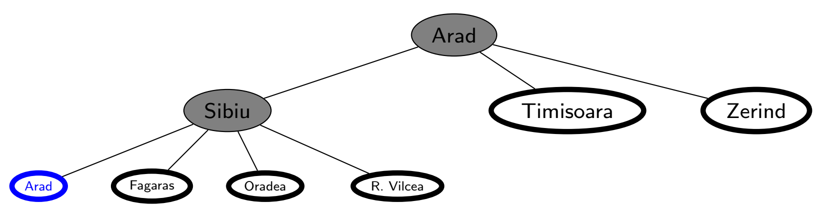


Depth-First Search: Romania

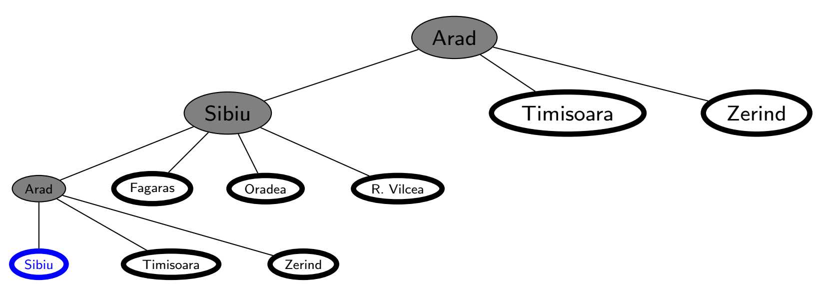
Depth-First Search: Romania



Depth-First Search: Romania



Depth-First Search: Romania



Depth-first search: Properties

Complete	Yes: if state space finite No: if state contains infinite paths or loops
Time	$O(b^m)$ (we need to explore until max depth m in any case!)
Space	$O(b \cdot m)$ (i.e. linear space) (need at most store m levels and at each level at most b nodes)
Optimal	No (there can be many better solutions in the unexplored part of the search tree)

▷ Disadvantage: Time terrible if m much larger than d.

▷ Advantage: Time may be much less than breadth-first search if solutions are dense.

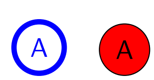
Iterative deepening search

▷ Depth-limited search: Depth-first search with depth limit

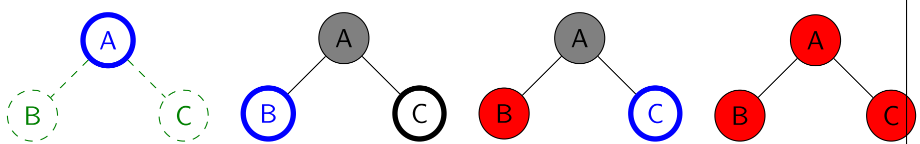
▷ Iterative deepening search: Depth-limit search with ever increasing limits

```
procedure TREE_SEARCH(problem)
    initialize the search tree using the initial state of problem
    for depth = 0 to ∞ do
        result ← Depth_Limited_search(problem, depth)
        if depth ≠ cutoff then
            return result
        end if
    end for
end procedure
```

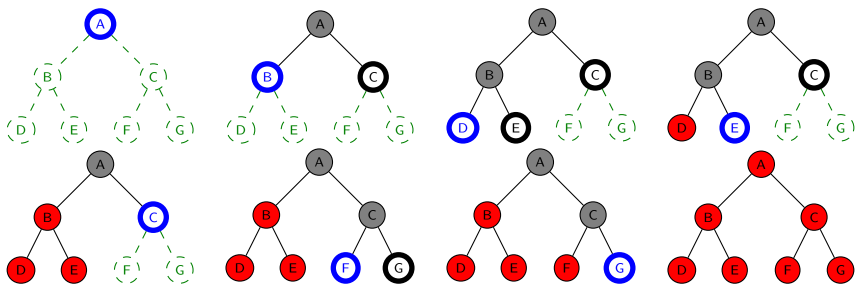

Iterative Deepening Search at Limit Depth 0



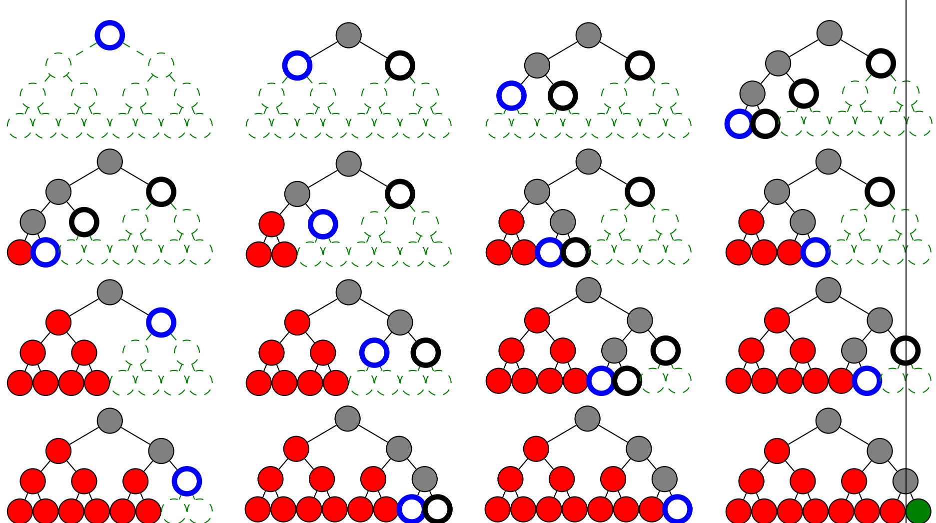
Iterative Deepening Search at Limit Depth 1



Iterative Deepening Search at Limit Depth 2



Iterative Deepening Search at Limit Depth 3



Iterative deepening search: Properties

Complete	Yes
Time	$(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d \in O(b^{d+1})$
Space	$O(bd)$
Optimal	Yes (if step cost = 1)

- ▷ (Depth-First) Iterative-Deepening Search often used in practice for search spaces of large, infinite, or unknown depth.
- ▷ Comparison:

Criterion	Breadth-first	Uniform-cost	Depth-first	Iterative deepening
Complete?	Yes*	Yes*	No	Yes
Time	b^{d+1}	$\approx b^d$	b^m	b^d
Space	b^{d+1}	$\approx b^d$	bm	bd
Optimal?	Yes*	Yes	No	Yes

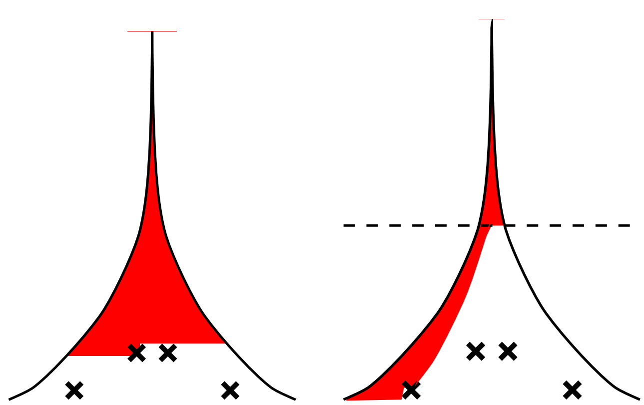
Note: To find a solution (at depth d) we have to search the whole tree up to d. Of course since we do not save the search state, we have to re-compute the upper part of the tree for the next level. This seems like a great waste of resources at first, however, iterative deepening search tries to be complete without the space penalties.

However, the space complexity is as good as depth-first search, since we are using depth-first search along the way. Like in breadth-first search, the whole tree on level d (of optimal solution) is explored, so optimality is inherited from there. Like breadth-first search, one can modify this to incorporate uniform cost search.

As a consequence, variants of iterative deepening search are the method of choice if we do not have additional information.

Comparison

Breadth-first search | Iterative deepening search



17 Informed Search Strategies

Summary: Uninformed Search/Informed Search

- ▷ Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- ▷ Variety of uninformed search strategies
- ▷ Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- ▷ Next Step: Introduce additional knowledge about the problem (informed search)
  - ▷ Best-first-, A*-search (guide the search by heuristics)
  - ▷ Iterative improvement algorithms

Best-first search

- ▷ Idea: Use an evaluation function for each node (estimate of "desirability") Expand most desirable unexpanded node
- ▷ Implementation: *fringe* is a queue sorted in decreasing order of desirability
- ▷ Special cases: Greedy search, A^* search

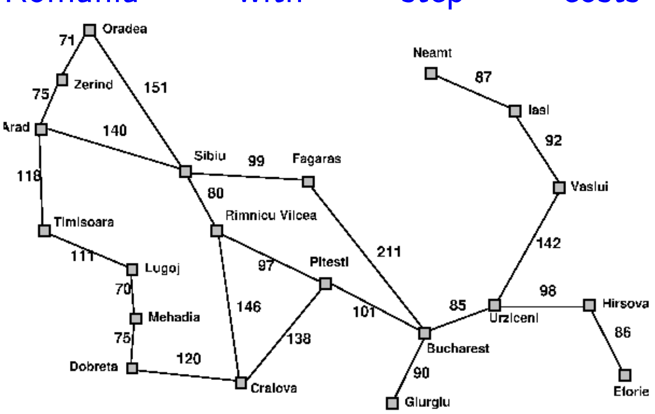
This is like UCS, but with evaluation function related to problem at hand replacing the path cost function.

If the heuristics is arbitrary, we expect incompleteness!

Depends on how we measure "desirability".

Concrete examples follow.

Romania with step costs in km



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

▷ Definition 17.1: A heuristic is an evaluation function h on nodes that estimates of cost from n to the nearest goal state.

Idea: Greedy search expands the node that appears to be closest to goal

▷ Example 17.2: $h_{SLD}(n) = $ straight-line distance from n to Bucharest

▷ Note: Unlike uniform-cost search the node evaluation function has nothing to do with the nodes explored so far

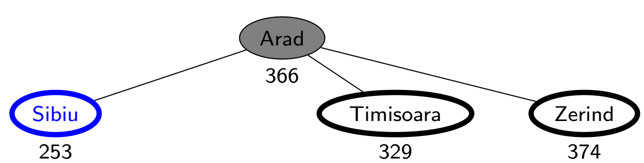
internal search control $\longrightarrow$ external search control
partial solution cost $\longrightarrow$ goal cost estimation

In greedy search we replace the *objective* cost to *construct* the current solution with a heuristic or *subjective* measure from which we think it gives a good idea how far we are from a *solution*. Two things have shifted:

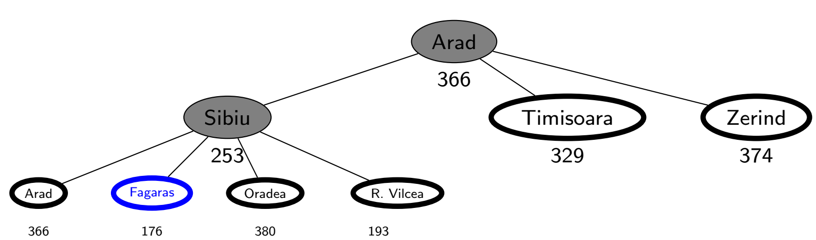
- we went from internal (determined only by features inherent in the search space) to an external/heuristic cost
- instead of measuring the cost to build the current partial solution, we estimate how far we are from the desired goal

Greedy Search: Romania

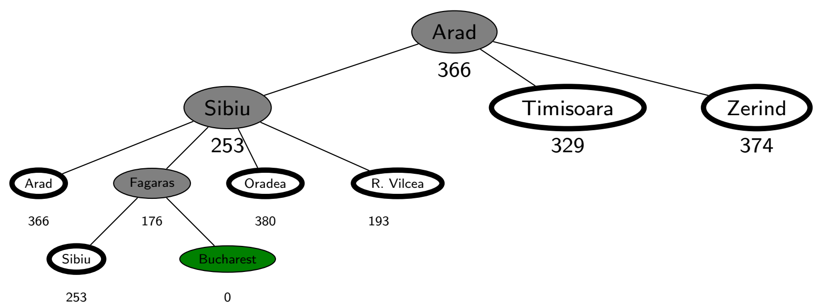
Greedy Search: Romania



Greedy Search: Romania



Greedy Search: Romania



Greedy search: Properties

Complete	No: Can get stuck in loops Complete in finite space with repeated-state checking
Time	$O(b^m)$
Space	$O(b^m)$
Optimal	No

- ▷ Example 17.3: Greedy search can get stuck going from Iasi to Oradea: Iasi → Neamt → Iasi → Neamt → ···
- ▷ Worst-case time same as depth-first search,
- ▷ Worst-case space same as breadth-first
- ▷ But a good heuristic can give dramatic improvement

Greedy Search is similar to UCS. Unlike the latter, the node evaluation function has nothing to do with the nodes explored so far. This can prevent nodes from being enumerated systematically as they are in UCS and BFS.

For completeness, we need repeated state checking as the example shows. This enforces complete enumeration of state space (provided that it is finite), and thus gives us completeness.

Note that nothing prevents from *all* nodes being searched in worst case; e.g. if the heuristic function gives us the same (low) estimate on all nodes except where the heuristic mis-estimates the distance to be high. So in the worst case, greedy search is even worse than BFS, where d (depth of first solution) replaces m.

The search procedure cannot be optimal, since actual cost of solution is not considered.

For both, completeness and optimality, therefore, it is necessary to take the actual cost of partial solutions, i.e. the path cost, into account. This way, paths that are known to be expensive are avoided.

A^* search

- Idea: Avoid expanding paths that are already expensive (make use of actual cost)

  The simplest way to combine heuristic and path cost is to simply add them.

- Definition 17.4: The evaluation function for A^*-search is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost for n and $h(n)$ is the estimated cost to goal from n.

- Thus $f(n)$ is the estimated total cost of path through n to goal

- Definition 17.5: Best-First-Search with evaluation function $g + h$ is called $astarSearch$ search.

 239 JACOBS UNIVERSITY

This works, provided that h does not overestimate the true cost to achieve the goal. In other words, h must be *optimistic* wrt. the real cost h^*. If we are too pessimistic, then non-optimal solutions have a chance.

A^* search: Admissibility

- Definition 17.6: (Admissibility of heuristic)

  $h(n)$ is called admissible if $0 \leq h(n) \leq h^*(n)$ for all nodes n, where $h^*(n)$ is the true cost from n to goal. (In particular: $h(G) = 0$ for goal G)

- Example 17.7: Straight-line distance never overestimates the actual road distance (triangle inequality)

  Thus $h_{SLD}(n)$ is admissible.

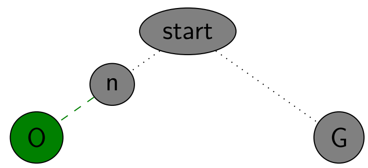
 240 JACOBS UNIVERSITY

A^* Search: Admissibility

▷ Theorem 17.8: A^* *search with admissible heuristic is optimal*

▷ Proof: We show that sub-optimal nodes are never selected by A^*

P.1 Suppose a suboptimal goal G has been generated then we are in the following situation:

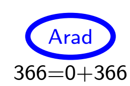


P.2 Let n be an unexpanded node on a path to an optimal goal O, then

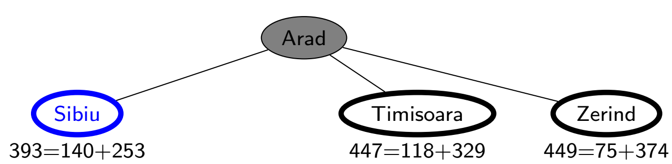
$$\begin{array}{rcll} f(G) & = & g(G) & \text{since } h(G) = 0 \\ & > & g(O) & \text{since } G \text{ suboptimal} \\ & = & g(n) + h^*(n) & n \text{ on optimal path} \\ & \geq & g(n) + h(n) & \text{since } h \text{ is admissible} \\ & = & f(n) & \end{array}$$

P.3 Thus, $f(G) > f(n)$ and $astarSearch$ never selects G for expansion. □

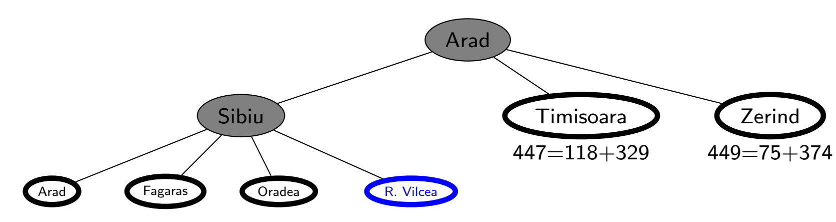
A^* Search Example



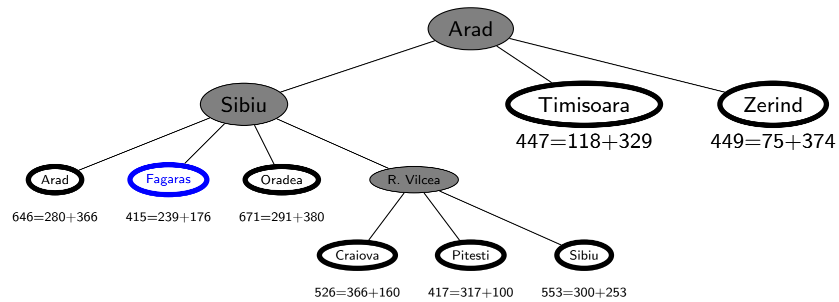
A^* Search Example



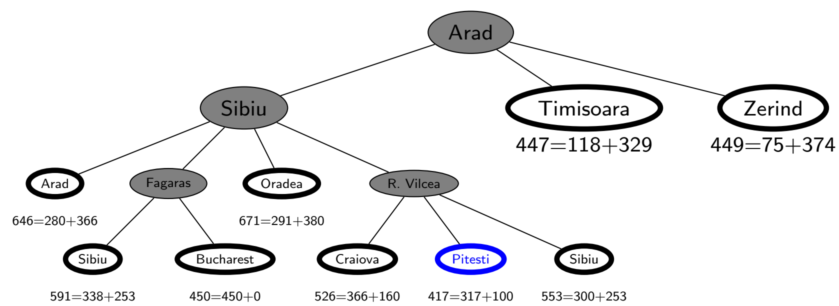
A^* Search Example



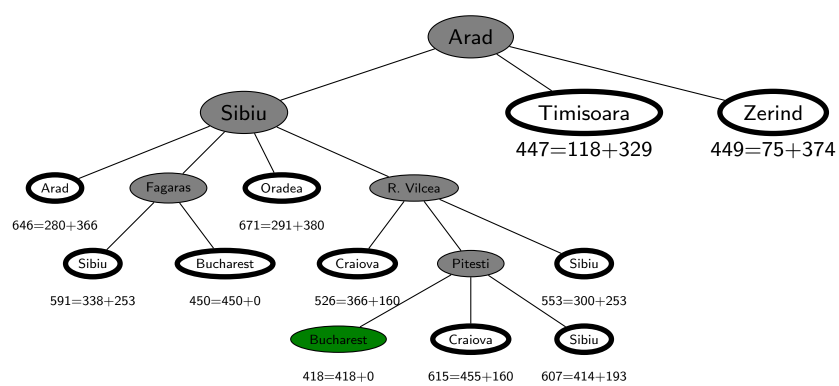
A^* Search Example



A^* Search Example

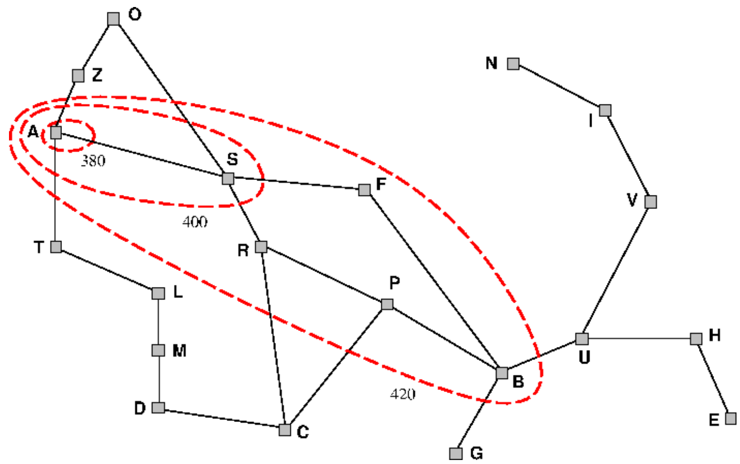


A^* Search Example



A^* search: f-contours

- ▷ A^* gradually adds "f-contours" of nodes



 248

A^* search: Properties

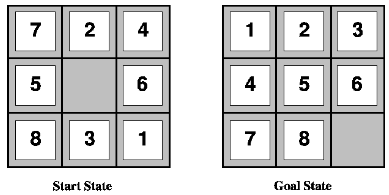
Complete	Yes (unless there are infinitely many nodes n with $f(n) \leq f(0)$)
Time	Exponential in [relative error in h × length of solution]
Space	Same as time (variant of BFS)
Optimal	Yes

- ▷ A^* expands all (some/no) nodes with $f(n) < h^*(n)$
- ▷ The run-time depends on how good we approximated the real cost h^* with h.

 249

Since the availability of admissible heuristics is so important for informed search (particularly for A^*), let us see how such heuristics can be obtained in practice. We will look at an example, and then derive a general procedure from that.

Admissible heuristics: Example 8-puzzle



- Example 17.9: Let $h_1(n)$ be the number of misplaced tiles in node n ($h_1(S) = 6$)
- Example 17.10: Let $h_2(n)$ be the total manhattan distance from desired location of each tile. ($h_2(S) = 2+0+3+1+0+1+3+4 = 14$)
- 17.11: (Typical search costs)

  (IDS $\widehat{=}$ iterative deepening search)

nodes explored	*IDS*	$A^*(h_1)$	$A^*(h_2)$
$d = 14$	*3,473,941*	*539*	*113*
$d = 24$	*too many*	*39,135*	*1,641*

Dominance

- Definition 17.12: Let h_1 and h_2 be two admissible heuristics we say that h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all n.
- Theorem 17.13: *If h_2 dominates h_1, then h_2 is better for search than h_1.*

Relaxed problems

- Finding good admissible heuristics is an art!
- Idea: Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem.
- Example 17.14: If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then we get heuristic h_1.
- Example 17.15: If the rules are relaxed so that a tile can move to *any adjacent square*, then we get heuristic h_2.
- Key point: The optimal solution cost of a relaxed problem is not greater than the optimal solution cost of the real problem.

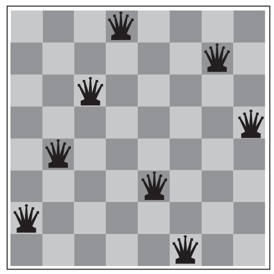
Relaxation means to remove some of the constraints or requirements of the original problem, so that a solution becomes easy to find. Then the cost of this easy solution can be used as an optimistic approximation of the problem.

18 Local Search

Local Search Problems

- ▷ Idea: Sometimes the path to the solution is irrelevant
- ▷ Example 18.1: (8 Queens Problem)

  Place 8 queens on a chess board, so that no two queens threaten each other.



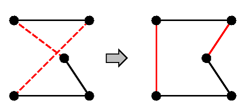
- ▷ This problem has various solutions, e.g. the one on the right
- ▷ Definition 18.2: A local search algorithm is a search algorithm that operates on a single state, the current state (rather than multiple paths). (advantage: constant space)
- ▷ Typically local search algorithms only move to successors of the current state, and do not retain search paths.
- ▷ Applications include: integrated circuit design, factory-floor layout, job-shop scheduling, portfolio management, fleet deployment,. . .

Local Search: Iterative improvement algorithms

- ▷ Definition 18.3: (Traveling Salesman Problem)

  Find shortest trip through set of cities such that each city is visited exactly once.

- ▷ Idea: Start with any complete tour, perform pairwise exchanges

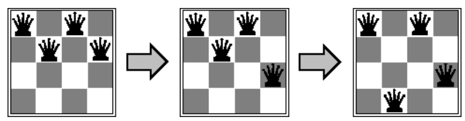


Local Search: Iterative improvement algorithms

- Definition 18.4: (n-queens problem)

  Put n queens on $n \times n$ board such that no two queens in the same row, columns, or diagonal.

- Idea: Move a queen to reduce number of conflicts



- Increasing number of solutions with increasing n (in general)

n	1	2	3	4	5	6	7	8	9	10	...	15
solutions	1	0	0	2	10	4	40	92	352	724	...	2,279,184

Hill-climbing (gradient ascent/descent)

- Idea: Start anywhere and go in the direction of the steepest ascent.
- Depth-first search with heuristic and w/o memory

```
procedure HILL-CLIMBING(problem)          ▷ a state that is a local minimum
    local current, neighbor                                          ▷ nodes
    current ← Make_Node(Initial_State[problem])
    loop
        neighbor ← a highest-valued successor of current
        if Value[neighbor] < Value[current] then
            return State[current]
            current ← neighbor
        end if
    end loop
end procedure
```

- Like starting anywhere in search tree and making a heuristically guided DFS.
- Works, if solutions are dense and local maxima can be escaped.

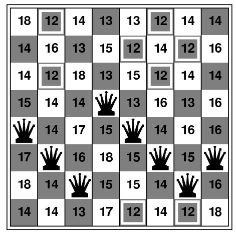
In order to understand the procedure on a more intuitive level, let us consider the following scenario: We are in a dark landscape (or we are blind), and we want to find the highest hill. The search procedure above tells us to start our search anywhere, and for every step first feel around, and then take a step into the direction with the steepest ascent. If we reach a place, where the next step would take us down, we are finished.

Of course, this will only get us into local maxima, and has no guarantee of getting us into global ones (remember, we are blind). The solution to this problem is to re-start the search at random (we do not have any information) places, and hope that one of the random jumps will get us to a slope that leads to a global maximum.

Example Hill-Climbing with 8 Queens

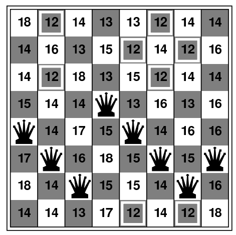
▷ Idea: Heuristic function h is number of queens that threaten each other.

▷ Example 18.5: An 8-queens state with heuristic cost estimate $h = 17$ showing h-values for moving a queen within its column



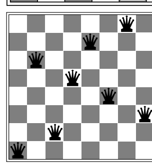
 257 JACOBS UNIVERSITY

Example Hill-Climbing with 8 Queens



▷ Successor function: The successor function returns all possible states generated by moving a single queen to another square in the same column.

▷ Problem: The state space has local minima. e.g. the board on the right has $h = 1$ but every successor has $h > 1$.



 258 JACOBS UNIVERSITY

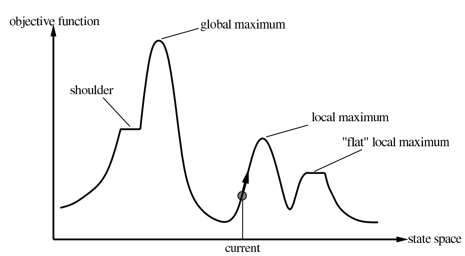
Hill-climbing

▷ Hill-climbing also called greedy local search

▷ Often makes very rapid progress towards a solution (just five steps to reach $h = 1$ solution!)

▷ Problem: Depending on initial state, can get stuck on local maxima/minima and plateaux

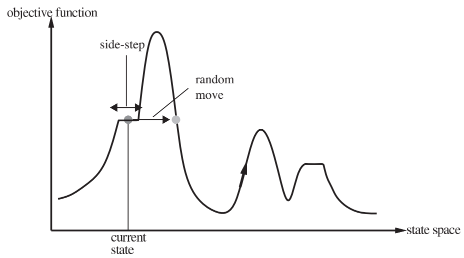
▷ "Hill-climbing search is like climbing Everest in thick fog with amnesia"



 259 JACOBS UNIVERSITY

Hill-climbing

- ▷ Idea: Escape local maxima by allowing some side-step, "bad" or random moves.



- ▷ Problem: Always allowing side-steps may end in infinite-loops. (if flat local maximum is not a shoulder)
- ▷ Solution: Restrict number of consecutive side-steps.

Hill-climbing

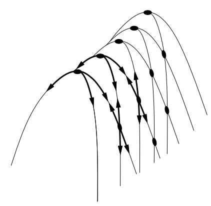
- ▷ Problem: Hill-climbing is incomplete and non-optimal
- ▷ Random-restart hill climbing conducts series of hill-climbing searches from randomly generated initial states. (hill-climbing is fast, so this is cheap)
- ▷ Example 18.6: Other examples: local search, simulated annealing...
- ▷ Properties: All are incomplete, non-optimal.
- ▷ Sometimes performs well in practice (if optimal solutions are dense)

Recent work on hill-climbing algorithms tries to combine complete search with randomization to escape certain odd phenomena occurring in statistical distribution of solutions.

Random-restart hill climbing is complete with probability approaching 1 because given a high enough number of restarts at some point a goal state will be generated as initial state.

Simulated annealing (Idea)

▷ Definition 18.7: Ridges are ascending successions of local maxima



▷ Problem: They are extremely difficult to navigate for local search algorithms

▷ Idea: Escape local maxima by allowing some "bad" moves, but gradually decrease their size and frequency

▷ Annealing is the process of heating steel and let it cool gradually to give it time to grow an optimal cristal structure.

▷ Simulated Annealing is like shaking a ping-pong ball occasionally on a bumpy surface to free it. (so it does not get stuck)

▷ Devised by Metropolis et al., 1953, for physical process modeling; adapted to CS in the early 80's.

▷ Widely used in VLSI layout, airline scheduling, etc.

 262 JACOBS UNIVERSITY

Simulated annealing (Implementation)

```
procedure SIMULATED-ANNEALING(problem,schedule)            ▷ a solution state
    local node, next                                        ▷ nodes
    local T          ▷ a "temperature" controlling prob. of downward steps
    current ← Make_Node(Initial_State[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then
            return current
        end if
        next ← a randomly selected successor of current
        ΔE ← Value[next] − Value[current]
        if ΔE > 0 then
            current ← next
        else
            current ← next only with probabilitye^(ΔE/T)
        end if
    end for
end procedure
```

a problem schedule is a mapping from time to "temperature"

 263 JACOBS UNIVERSITY

Properties of simulated annealing

- At fixed "temperature" T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

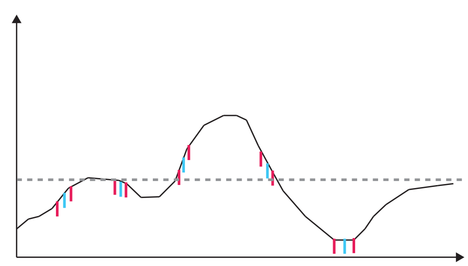
  T decreased slowly enough $\Longrightarrow$ always reach best state x^* because $\frac{e^{\frac{E(x^*)}{kT}}}{e^{\frac{E(x)}{kT}}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$ for small T.

- Is this necessarily an interesting guarantee?

In fact, it turns out that while simulated annealing is guaranteed to find the global optimum given a temperature decrease that is slow enough, often this will result in a search that takes longer than a complete search of the solution space.

Local beam search

- Idea: Keep k states instead of 1; choose top k of all their successors
- Not the same as k searches run in parallel! (Searches that find good states recruit other searches to join them)



- Problem: quite often, all k states end up on same local hill
- Idea: Choose k successors randomly, biased towards good ones. (Observe the close analogy to natural selection!)

Genetic algorithms (briefly)

- Idea:
  Use local beam search (keep a population of k)
  randomly modify population (mutation)
  generate successors from pairs of states (sexual reproduction)
  optimize a fitness function (survival of the fittest)

-

- GAs $\neq$ evolution: e.g., real genes encode replication machinery!

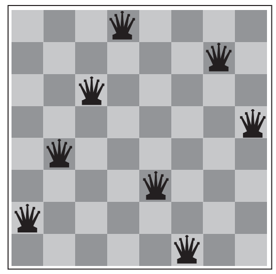
Genetic algorithms (continued)

▷ Problem: Genetic Algorithms require states encoded as strings (Genetic programming uses programs)

▷ Crossover helps iff substrings are meaningful components

Example 18.8: (Encoding 8 Queens)

▷ Encode 8 Queens states by position in their column, from top to bottom: 47382516



 267 JACOBS UNIVERSITY

Genetic algorithms (continued)

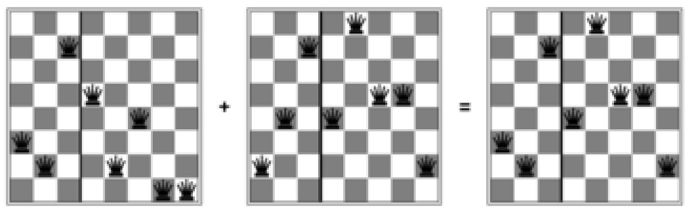
▷ Basic algorithm

```
procedure GENETIC ALGORITHM
    local population
    Generate randomly initial population
    while not termination criterion reached do
        Evaluate fitness of each individual in population
        Randomly select pairs of individuals for reproduction  ▷ probability based on fitness value
        Generate new individuals by crossover at randomly chosen point.
        Mutate the two offspring with probabilty p_m.
        Replace current population with new population  ▷ Alternative: keep k fittest individuals
    end while
    returnpopulation
end procedure
```

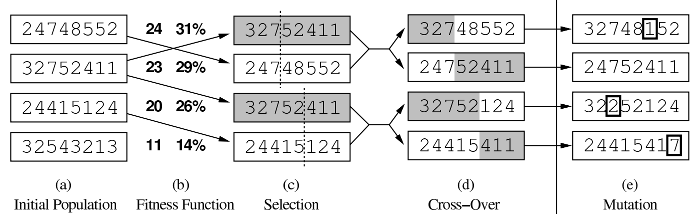
 268 JACOBS UNIVERSITY

Genetic algorithms (continued)

▷ Example 18.9: (Evolving 8 Queens)



▷ Generation of offspring:



 269

19 Programming as Search: Introduction to Logic Programming and ProLog

We will now learn a new programming paradigm: "logic programming" (also called "Declarative Programming"), which is an application of the search techniques we looked at last, and the logic techniques. We are going to study ProLog (the oldest and most widely used) as a concrete example of the ideas behind logic programming.

Logic Programming is a programming style that differs from functional and imperative programming in the basic procedural intuition. Instead of transforming the state of the memory by issuing instructions (as in imperative programming), or comupting the value of a function on some arguments, logic programming interprets the program as a body of knowledge about the respective situation, which can be queried for consequences. This is actually a very natural intuition; after all we only run (imperative or functional) programs if we want some question answered.

Logic Programming

- ▷ Idea: Use logic as a programming language!
- ▷ We state what we know about a problem (the program) and then ask for results (what the program would compute)
- ▷ Example 19.1:

Program	Leibniz is human Sokrates is is human Sokrates is a greek Every human is fallible	$x + 0 = x$ If $x + y = z$ then $x + s(y) = s(z)$ 3 is prime
Query	Are there fallible greeks?	is there a z with $s(s(0)) + s(0) = z$
Answer	Yes, Sokrates!	yes $s(s(s(0)))$

How to achieve this?: Restrict the logic calculus sufficiently that it can be used as computational procedure.

- ⧐ Slogan: Computation = Logic + Control ([Kowalski '73])
- ▷ We will use the programming language ProLog as an example

 270

ProLog is a simple logic programming language that exemplifies the ideas we want to discuss quite nicely. We will not introduce the language formally, but in concrete examples as we explain the theortical concepts. For a complete reference, please consult the online book by Blackburn & Bos & Striegnitz `http://www.coli.uni-sb.de/~kris/learn-prolog-now/`.

Of course, this the whole point of writing down a knowledge base (a program with knowledge about the situation), if we do not have to write down *all* the knowledge, but a (small) subset, from which the rest follows. We have already seen how this can be done: with logic. For logic programming we will use a logic called "first-order logic" which we will not formally introduce here. We have already seen that we can formulate propositional logic using terms from an abstract data type instead of propositional variables. For our purposes, we will just use terms with variables instead of the ground terms used there. [8] EdNote(8)

[8] EdNote: reference

Representing a Knowledge base in ProLog

- A knowledge base is represented (symbolically) by a set of facts and rules.
- Definition 19.2: A fact is a statement written as a term that is unconditionally true of the domain of interest. (write with a term followed by a ".")
- Example 19.3: We can state that Mia is a woman as $\mathtt{woman}(\mathtt{mia})$.
- Definition 19.4: A rule states information that is *conditionally* true in the domain.
- Example 19.5: Write "something is a car if it has a motor and four wheels" as $(\mathtt{car}(\mathtt{X}) : -\mathtt{has\_motor}(\mathtt{X}), \mathtt{has\_wheels}(\mathtt{X}, 4))$ (variables are upper-case)
  this is just an ASCII notation for $m(x) \wedge w(x, 4) \Rightarrow car(x)$
- Definition 19.6: The knowledge base given by a set of facts and rules is that set of facts that can be derived from it by Modus Ponens (MP) and $\wedge I$.

$$\frac{A \quad A \Rightarrow B}{B} MP \qquad \frac{A \quad B}{A \wedge B} \wedge I \qquad \frac{\mathbf{A}}{[\mathbf{B}/X]\mathbf{A}} Subst$$

 271 JACOBS UNIVERSITY

Knowledge Base (Example)

- Example 19.7: `car(c).` is in the knowlege base generated by

```
has_motor(c).
has_wheels(c,4).
car(X):- has_motor(X),has_wheels(X,4).
```

$$\cfrac{\cfrac{m(c) \quad w(c,4)}{m(c) \wedge w(c,4)} \wedge I \qquad \cfrac{m(x) \wedge w(x,4) \Rightarrow car(x)}{m(c) \wedge w(c,4) \Rightarrow car(c)} Subst}{car(c)} MP$$

 272 JACOBS UNIVERSITY

Querying the Knowledge base

- Idea: We want to see whether a fact is in the knowledge base.
- Definition 19.8: A query or goal is a statement of which we want to know whether it is in the knowledge base. (write as $?-$ A., if A statement)
- Problem: Knowledge bases can be big and even infinite.
- Example 19.9: The the knowledge base induced by the program

```
nat(zero).
nat(s(X)) :- nat(X).
```

  is the set $\{\mathtt{nat(zero)}, \mathtt{nat(s(zero))}, \mathtt{nat(s(s(zero)))}, \ldots\}$.

- Idea: interpret this as a search problem.
  - state = tuple of goals; goal state = empty list (of goals).
  - $next(\langle \mathtt{G}, R_1, \ldots R_l\rangle) := \langle\sigma(\mathtt{B}_1), \ldots, \sigma(\mathtt{B}_m), R_1, \ldots, R_l\rangle$ (backchaining) if there is a rule $\mathtt{H} : -\mathtt{B}_1, \ldots \mathtt{B}_m$. and a substitution σ with $\sigma\mathtt{H} = \sigma\mathtt{G}$.

```
?- nat(s(s(zero))).
?- nat(s(zero)).
?- nat(zero).
Yes
```

- If a query contains variables, then ProLog will return an .

```
has_wheels(mybmw,4).
has_motor(mybmw).
car(X):-has_wheels(X,4),has_motor(X).
?- car(Y)
?- has_wheels(Y,4),has_motor(Y).
Y = mybmw
?- has_motor(mybmw).
Y = mybmw
Yes
```

- If no instance of the statement in a query can be derived from the knowledge base, then the ProLog interpreter reports failure.

```
?- nat(s(s(0))).
?- nat(s(0)).
?- nat(0).
FAIL
No
```


We will now discuss how to use a ProLog interpreter to get to know the language. The SWI ProLog interpreter can be downloaded from http://www.swi-prolog.org/. To start the ProLog interpreter with pl or prolog or swipl from the shell. The SWI manual is available at http://gollem.science.uva.nl/SWI-Prolog/Manual/

We will introduce working with the interpreter using unary natural numbers as examples: we first add the fact [2] to the knowledge base

```
unat(zero).
```

which asserts that the predicate unat[3] is true on the term zero. Generally, we can add a fact to

[2]for "unary natural numbers"; we cannot use the predicate nat and the constructor functions here, since their meaning is predefined in ProLog

[3]for "unary natural numbers".

the knowledge base either by writing it into a file (e.g. `example.pl`) and then "consulting it" by writing one of the following commands into the interpreter:

```
[example]
consult('example.pl').
```

or by directly typing

```
assert(unat(zero)).
```

into the `ProLog` interpreter. Next tell `ProLog` about the following rule

```
assert(unat(suc(X)) :- unat(X)).
```

which gives the `ProLog` runtime an initial (infinite) knowledge base, which can be queried by

```
?- unat(suc(suc(zero))).
\smlout{Yes}
```

Running `ProLog` in an `emacs` window is incredibly nicer than at the command line, because you can see the whole history of what you have done. Its better for debugging too. If you've never used `emacs` before, it still might be nicer, since its pretty easy to get used to the little bit of `emacs` that you need. (Just type "`emacs &`" at the `UNIX` command line to run it; if you are on a remote terminal like `putty`, you can use "`emacs -nw`".).

If you don't already have a file in your home directory called "`.emacs`" (note the dot at the front), create one and put the following lines in it. Otherwise add the following to your existing `.emacs` file:

```
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
    (autoload 'prolog-mode "prolog" "Major mode for editing Prolog programs." t)
    (setq prolog-program-name "swipl") ; or whatever the prolog executable name is
    (add-to-list 'auto-mode-alist '("\\pl$" . prolog-mode))
```

The file `prolog.el`, which provides `prolog-mode` should already be installed on your machine, otherwise download it at `http://turing.ubishops.ca/home/bruda/emacs-prolog/`

Now, once you're in `emacs`, you will need to figure out what your "meta" key is. Usually its the alt key. (Type "control" key together with "h" to get help on using `emacs`). So you'll need a "`meta-X`" command, then type "`run-prolog`". In other words, type the meta key, type "`x`", then there will be a little window at the bottom of your `emacs` window with "`M-x`", where you type `run-prolog`[4]. This will start up the `SWI ProLog` interpreter, ...et voilà!

The best thing is you can have two windows "within" your `emacs` window, one where you're editing your program and one where you're running `ProLog`. This makes debugging easier.

Depth-First Search with Backtracking

- So far, all the examples led to direct success or to failure. (simpl. KB)
- Search Procedure: top-down, left-right depth-first search
  - Work on the queries in left-right order.
  - match first query with the head literals of the clauses in the program in top-down order.
  - if there are no matches, fail and backtrack to the (chronologically) last point.
  - otherwise backchain on the first match , keep the other matches in mind for backtracking. (backtracking points)

 274 JACOBS UNIVERSITY

[4]Type "control" key together with "h" then press "m" to get an exhaustive mode help.

Note: We have seen before[9] that depth-first search has the problem that it can go into loops. And in fact this is a necessary feature and not a bug for a programming language: we need to be able to write non-terminating programs, since the langugage would not be Turing-complete ogtherwise. The argument can be sketched as follows: we have seen that for Turing machines the halting problem[10] is undecidable. So if all ProLog programs were terminating, then ProLog would be weaker than Turing machines and thus not Turing complete. EdNote(9) EdNote(10)

Backtracking by Example

```
has_wheels(mytricycle,3).
has_wheels(myrollerblade,3).
has_wheels(mybmw,4).
has_motor(mybmw).
car(X):-has_wheels(X,3),has_motor(X).      % cars sometimes have 3 wheels
car(X):-has_wheels(X,4),has_motor(X).
?- car(Y).
?- has_wheels(Y,3),has_motor(Y). % backtrack point 1
Y = mytricycle}}                   % backtrack point 2
?- has_motor(mytricycle).
FAIL                             % fails, backtrack to 2
Y = myrollerblade               % backtrack point 2
?- has_motor(myrollerblade).
FAIL                             % fails, backtrack to 1
?- has_wheels(Y,4),has_motor(Y).
Y = mybmw
?- has_motor(mybmw).
Y=mybmw
Yes
```


Can We Use This For Programming?

- Question: What about functions? E.g. the addition function?
- Question: We do not have (binary) functions, in ProLog
- Idea (back to math): use a three-place predicate.
  Example 19.10: `add(X,Y,Z)` stands for `X+Y=Z`
- Now we can directly write the recursive equations $X + 0 = X$ (base case) and $X + s(Y) = s(X + Y)$ into the knowledge base.

```
add(X,zero,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

- similarly with multiplication and exponentiation.

```
mult(X,o,o).
mult(X,s(Y),Z) :- mult(X,Y,W), add(X,W,Z).
expt(X,o,s(o)).
expt(X,s(Y),Z) :- expt(X,Y,W), mult(X,W,Z).
```


Note: Viewed through the right glasses logic programming is very similar to functional programming; the only difference is that we are using $n+1$-ary relations rather than n-ary functions. To see how this works let us consider the addition function/relation example above: instead of a binary function $+$ we program a ternary relation add, where relation $\mathtt{add}(X, Y, Z)$ means $X + Y = Z$. We start with the same defining equations for addition, rewriting them to relational style.

[9] EdNote: reference
[10] EdNote: reference

The first equation is straight-foward via our correspondance and we get the `ProLog` fact `add(X,zero,X)`.. For the equation $X + s(Y) = s(X + Y)$ we have to work harder, the straight-forward relational translation $\mathtt{add}(X, s(Y), s(X + Y))$ is impossible, since we have only partially replaced the function $+$ with the relation `add`. Here we take refuge in a very simple trick that we can always do in logic (and mathematics of course): we introduce a new name Z for the offending expression $X + Y$ (using a variable) so that we get the fact $\mathtt{add}(X, s(Y), s(Z))$. Of course this is not universally true (remember that this fact would say that "$X + s(Y) = s(Z)$ for all X, Y, and Z"), so we have to extend it to a `ProLog` rule (`add(X,s(Y),s(Z)) :−add(X,Y,Z)`) which relativizes to mean "$X + s(Y) = s(Z)$ for all X, Y, and Z with $X + Y = Z$".

Indeed the rule implements addition as a recursive predicate, we can see that the recursion relation is terminating, since the left hand sides are have one more constructor for the successor function. The examples for multiplication and exponentiation can be developed analogously, but we have to use the naming trick twice.

More Examples from elementary Arithmetics

▷ Example 19.11: We can also use the add relation for subtraction without changing the implementation. We just use variables in the "input positions" and ground terms in the other two (possibly very inefficient since "generate-and-test approach")

```
?−add(s(zero),X,s(s(s(zero)))).
X = s(s(zero))
Yes
```

▷ Example 19.12: Computing the the n^{th} Fibonacci Number (0,1,1,2,3,5,8,13,...; add the last two to get the next), using the addition predicate above.

```
fib(zero,zero).
fib(s(zero),s(zero)).
fib(s(s(X)),Y):−fib(s(X),Z),fib(X,W),add(Z,W,Y).
```

▷ Example 19.13: using `ProLog`'s internal arithmetic: a goal of the form `?−D is e.` where e is a ground arithmetic expression binds D to the result of evaluating e.

```
fib(0,0).
fib(1,1).
fib(X,Y):− D is X − 1, E is X − 2,fib(D,Z),fib(E,W), Y is Z + W.
```


Note: Note that the `is` relation does not allow "generate-and-test" inversion as it insists on the right hand being ground. In our example above, this is not a problem, if we call the `fib` with the first ("input") argument a ground term. Indeed, if match the last rule with a goal `?−fib(g,Y).`, where g is a ground term, then `g−1` and `g−2` are ground and thus `D` and `E` are bound to the (ground) result terms. This makes the input arguments in the two recursive calls ground, and we get ground results for `Z` and `W`, which allows the last goal to succeed with a ground result for `Y`. Note as well that re-ordering the body literals of the rule so that the recursive calls are called before the computation literals will lead to failure.

20 Logic Programming as Resolution Theorem Proving

We know all this already

- Goals, goal-sets, rules, and facts are just clauses. (so-called Horn clauses)
  - observation 20.1: (rule)
    $\mathtt{H} : -\mathtt{B}_1, \ldots, \mathtt{B}_n.$ *corresponds to* $\mathtt{H} \vee \neg \mathtt{B}_1 \vee \ldots \vee \neg \mathtt{B}_n$ (*head the only positive literal*)
  - observation 20.2: (goal setid)
    $? - \mathtt{G}_1, \ldots, \mathtt{G}_n.$ *corresponds to* $\neg \mathtt{G}_1, \ldots, \neg \mathtt{G}_n$
  - observation 20.3: (fact)
    $\mathtt{F}.$ *corresponds to the unit clause* $\mathtt{F}$.
- Definition 20.4: A Horn clause is a clause with at most one positive literal.
- Note: backchaining becomes (hyper)-resolution (special case for rule with facts)

PROLOG (Horn clauses)

- Logic programming by resolution theorem proving
- Question: With full predicate logic (with equality)?
- Answer: No, since
  - Search spaces are immense
  - Control (of proof search $\widehat{=}$ program) cannot be understood/affected by the programmer.
  - problems with termination

PROLOG (Horn clauses)

- Definition 20.5:Each clause contains at most one positive literal
  - $B_1 \vee \ldots \vee B_n \vee \neg A$ $((\mathtt{A} : -\mathtt{B1}, \ldots, \mathtt{Bn}))$
  - Rule clause: $(\mathtt{fallible}(\mathtt{X}) : -\mathtt{human}(\mathtt{X}))$
  - Fact clause: $\mathtt{human}(\mathtt{sokrates})$.
  - Program: set of rule and fact clauses
  - Query: $? - \mathtt{fallible}(\mathtt{X}), \mathtt{greek}(\mathtt{X})$.

PROLOG (SLD Resolution)

- Strategy for Resolution: SLDNF (LUSH)
  - Selected Literal Definite clauses
  - Linear resolution
    (Always continue work with the focussed clause)
  - Select the lefmost unsolved positive literal
  - Always resolve on the positive literal
- Theorem 20.6: *(Strongly) complete on horn clauses*
  - *Each instance of the query that is entailed by the program is subsumed by a positive answer.*

PROLOG: Our Example

- Program:

```
human(sokrates).
human(leibniz).
greek(sokrates).
fallible(X) :- human(X).
```

- Example 20.7: (Query)
  ? − `fallible(X), greek(X)`.
- Answer substitution: `[sokrates/X]`

Why Only Horn Clauses?

- General clauses of the form `A1,...,An :- B1,...,Bn.`
- e.g. `greek(sokrates),greek(perikles)`
  - Question: Are there fallible greeks?
  - Indefinite answer: Yes, Perikles or Sokrates
  - Warning: how about Sokrates and Perikles?
- e.g. `greek(sokrates),roman(sokrates):-.`
  - Query: Are there fallible greeks?
  - Answer: Yes, Sokrates, if he is not a roman
  - Is this abduction?????

20.1 First-Order Unification

We will now look into the problem of finding a substitution σ that make two terms equal (we say it unifies them) in more detail. The presentation of the unification algorithm we give here "transformation-based" this has been a very influential way to treat certain algorithms in theoretical computer science.

A transformation-based view of algorithms: The "transformation-based" view of algorithms divides two concerns in presenting and reasoning about algorithms according to Kowalski's slogan[11] EdNote(11)

computation = logic + control

The computational paradigm highlighted by this quote is that (many) algorithms can be thought of as manipulating representations of the problem at hand and transforming them into a form that makes it simple to read off solutions. Given this, we can simplify thinking and reasoning about such algorithms by separating out their "logical" part, which deals with is concerned with how the problem representations can be manipulated in principle from the "control" part, which is concerned with questions about when to apply which transformations.

It turns out that many questions about the algorithms can already be answered on the "logic" level, and that the "logical" analysis of the algorithm can already give strong hints as to how to optimize control.

In fact we will only concern ourselves with the "logical" analysis of unification here.

The first step towards a theory of unification is to take a closer look at the problem itself. A first set of examples show that we have multiple solutions to the problem of finding substitutions that make two terms equal. But we also see that these are related in a systematic way.

Unification (Definitions)

- Problem: For given terms $\mathbf{A}$ and $\mathbf{B}$ find a substitution σ, such that $\sigma\mathbf{A} = \sigma\mathbf{B}$.
  - term pairs $\mathbf{A} =^? \mathbf{B}$ e.g. $f(X) =^? f(g(Y))$
  - Solutions: $[g(a)/X], [a/Y]$
    $[g(g(a))/X], [g(a)/Y]$
    $[g(Z)/X], [Z/Y]$
  - are called unifiers, $\mathbf{U}(\mathbf{A} =^? \mathbf{B}) := \{\sigma \mid \sigma\mathbf{A} = \sigma\mathbf{B}\}$

  Idea: find representatives in $\mathbf{U}(\mathbf{A} =^? \mathbf{B})$, that generate the set of solutions
- Definition 20.8: Let σ and θ be substitutions and $W \subseteq \mathcal{V}_\iota$, we say that a σ more general than θ (on W write $\sigma \leq \theta[W]$), iff there is a substitution ρ, such that $\theta = \rho \circ \sigma[W]$, where $\sigma = \rho[W]$, iff $\sigma X = \rho X$ for all $X \in W$.
- Definition 20.9: σ is called a most general unifier of $\mathbf{A}$ and $\mathbf{B}$, iff it is minimal in $\mathbf{U}(\mathbf{A} =^? \mathbf{B})$ wrt. $\leq [\mathbf{free}(\mathbf{A}) \cup \mathbf{free}(\mathbf{B})]$.

 284

The idea behind a most general unifier is that all other unifiers can be obtained from it by (further) instantiation. In an automated theorem proving setting, this means that using most general unifiers is the least committed choice — any other choice of unifiers (that would be necessary for completeness) can later be obtained by other substitutions.

Note that there is a subtlety in the definition of the ordering on substitutions: we only compare on a subset of the variables. The reason for this is that we have defined substitutions to be total on (the infinite set of) variables for flexibility, but in the applications (see the definition of a most general unifiers), we are only interested in a subset of variables: the ones that occur in the initial problem formulation. Intuitively, we do not care what the unifiers do off that set. If we did not

[11] EdNote: find the reference, and see what he really said

have the restriction to the set W of variables, the ordering relation on substitutions would become much too fine-grained to be useful (i.e. to guarantee unique most general unifiers in our case).

Now that we have defined the problem, we can turn to the unification itself.

Unification (Equational Systems)

- ▷ Idea: Unification is equation solving.
- ▷ Definition 20.10: We call a formula $\mathbf{A}^1 =^? \mathbf{B}^1 \wedge \ldots \wedge \mathbf{A}^n =^? \mathbf{B}^n$ an equational system.
- ▷ We consider equational systems as sets of equations ($\wedge$ is ACI), and equations as two-element multisets ($=^?$ is C).
- ▷ Definition 20.11: We say that $X^1 =^? \mathbf{B}^1 \wedge \ldots \wedge X^n =^? \mathbf{B}^n$ is a solved form, iff the X^i are distinct and $X^i \notin \mathbf{free}(\mathbf{B}^j)$.
- ▷ Lemma 20.12: *If $\mathcal{E} = X^1 =^? \mathbf{B}^1 \wedge \ldots \wedge X^n =^? \mathbf{B}^n$ is a solved form, then $\mathcal{E}$ has the unique most general unifier $\sigma_\mathcal{E} := [\mathbf{B}^1/X^1], \ldots, [\mathbf{B}^n/X^n]$.*
- ▷ Proof:

P.1 Let $\theta \in \mathbf{U}(\mathcal{E})$, then $\theta X^i = \theta \mathbf{B}^i = \theta \circ \sigma_\mathcal{E}(X^i)$

P.2 and thus $\theta = \theta \circ \sigma_\mathcal{E}[\mathbf{supp}(\sigma)]$. □

In principle, unification problems are sets of equations, which we write as conjunctions, since all of them have to be solved for finding a unifier. Note that it is not a problem for the "logical view" that the representation as conjunctions induces an order, since we know that conjunction is associative, commutative and idempotent, i.e. that conjuncts do not have an intrinsic order or multiplicity, if we consider two equational problems as equal, if they are equivalent as propositional formulae. In the same way, we will abstract from the order in equations, since we know that the equality relation is symmetric. Of course we would have to deal with this somehow in the implementation (typically, we would implement equational problems as lists of pairs), but that belongs into the "control" aspect of the algorithm, which we are abstracting from at the moment.

It is essential to our "logical" analysis of the unification algorithm that we arrive at equational problems whose unifiers we can read off easily. Solved forms serve that need perfectly as the Lemma[12] shows.[13] EdNote(12) EdNote(13)

[12] EdNote: reference

[13] EdNote: say something about the occurs-in-check,...

Unification Algorithm

▷ Definition 20.13: Inference system $\mathcal{U}$

$$\frac{\mathcal{E} \wedge (f\mathbf{A}^1 \ldots \mathbf{A}^n)=^?(f\mathbf{B}^1 \ldots \mathbf{B}^n)}{\mathcal{E} \wedge \mathbf{A}^1=^?\mathbf{B}^1 \wedge \ldots \wedge \mathbf{A}^n=^?\mathbf{B}^n}\,\mathcal{U}dec \qquad \frac{\mathcal{E} \wedge \mathbf{A}=^?\mathbf{A}}{\mathcal{E}}\,\mathcal{U}triv$$

$$\frac{\mathcal{E} \wedge X=^?\mathbf{A} \quad X \notin \mathbf{free}(\mathbf{A}) \quad X \in \mathbf{free}(\mathcal{E})}{[\mathbf{A}/X](\mathcal{E}) \wedge X=^?\mathbf{A}}\,\mathcal{U}elim$$

▷ Lemma 20.14: *$\mathcal{U}$ is correct ($\mathcal{E} \vdash_{\mathcal{U}} \mathcal{F}$ implies $\mathbf{U}(\mathcal{F}) \subseteq \mathbf{U}(\mathcal{E})$)*

▷ Lemma 20.15: *$\mathcal{U}$ is complete ($\mathcal{E} \vdash_{\mathcal{U}} \mathcal{F}$ implies $\mathbf{U}(\mathcal{E}) \subseteq \mathbf{U}(\mathcal{F})$)*

▷ Lemma 20.16: *$\mathcal{U}$ is confluent (order of derivations does not matter)*

▷ Corollary 20.17: *First-Order Unification is unitary (unique most general unifiers)* ($\mathcal{U}$ trivially branching)

 286 JACOBS UNIVERSITY

Unification (Termination)

▷ Lemma 20.18: *$\mathcal{U}$ is terminating* (any $\mathcal{U}$-derivation is finite)

▷ Proof:

P.1 Let $\mu(\mathcal{E}) := \langle m, \mathcal{N}, n \rangle$, where

- ▷ m is the number of unsolved variables in $\mathcal{E}$
- ▷ $\mathcal{N}$ is the multi-set of term depths in $\mathcal{E}$
- ▷ n the number of term pairs in $\mathcal{E}$

P.2 The lexicographic order $\prec$ on triples $\mu(\mathcal{E})$ is decreased by all inference rules. □

 287 JACOBS UNIVERSITY

Unification (decidable)

▷ Definition 20.19: We call an equational problem $\mathcal{E}$ $\mathcal{U}$-reducible, iff there is a $\mathcal{U}$-step $\mathcal{E} \vdash_{\mathcal{U}} \mathcal{F}$ from $\mathcal{E}$.

▷ Lemma 20.20: *If $\mathcal{E}$ is unifiable but not solved, then it is $\mathcal{U}$-reducible*

▷ Proof:

P.1 There is an unsolved pair $\mathbf{A}{=}^?\mathbf{B}$ in $\mathcal{E} = \mathcal{E}' \wedge \mathbf{A}{=}^?\mathbf{B}$.

P.2 we have two cases

P.2.1 $\mathbf{A}, \mathbf{B} \in \mathcal{V}_\iota$:

P.2.1.1 then $\mathbf{A} = f(\mathbf{A}^1 \ldots \mathbf{A}^n)$ and $\mathbf{B} = f(\mathbf{B}^1 \ldots \mathbf{B}^n)$, and thus $\mathcal{U}dec$ is applicable □

P.2.2 $\mathbf{A} = X \in \mathcal{V}_\iota \cap \mathbf{free}(\mathcal{E})$:

P.2.2.1 then $\mathcal{U}elim$ (if $\mathbf{B} \neq X$) or $\mathcal{U}triv$ (if $\mathbf{B} = X$) is applicable. □

▷ Corollary 20.21: *Unification is decidable in PL1*

▷ Proof Idea: $\mathcal{U}$-irreducible set of equations are either solved or unsolvable □

21 Topics in Logic Programming

Adding Lists to ProLog

▷ Lists are represented by terms of the form `[a,b,c,...]`

▷ first/rest representation `F|R`, where `R` is a rest list.

▷ predicates for `member`, `append` and `reverse` of lists in default `ProLog` representation.

```
member(X,[X|_]).
member(X,[_|R]):-member(X,R).
append([],L,L).
append([X|R],L,[X|S]):-append(R,L,S).
reverse([],[]).
reverse([X|R],L):-reverse(R,S),append(S,[X],L).
```

Relational Programming Techniques

- ▷ Parameters have no unique direction "in" or "out"

```
:- rev(L,[1,2,3]).
:- rev([1,2,3],L1).
:- rev([1,X],[2,Y]).
```

- ▷ Symbolic programming by structural induction

```
rev([],[]).
rev([X,Xs],Ys) :- ...
```

- ▷ Generate and test

```
sort(Xs,Ys) :- perm(Xs,Ys), ordered(Ys).
```


Use ProLog for Talking/Programming about Logics

- ▷ Idea: We will use $PLNQ$ (prop. logic where prop. variables are ADT terms)
- ▷ represent the ADT as facts of the form

```
constant(mia).
pred(love,2).
pred(run,1).
fun(father,1)
```

this licenses ProLog terms like `run(mia).` and `love(mia,father(mia)).`

- ▷ represent propositional connectives as ProLog operators, which we declare with the following declarations.

```
:- op(900,yfx,<>).  % equivalence
:- op(900,yfx,>).   % implication
:- op(850,yfx,\/).  % disjunction
:- op(800,yfx,\&).   % conjunction
:- op(750,fx,~).    % negation
```

The first argument of op is the operator precedence, the second the fixity. This licenses ProLog terms like `X > Y.` and `~(X).`

- ▷ Use the ProLog built-in predicate `=..` to deconstruct terms: a literal `f(a,b)=..Z` binds `Z` to the list `[f,a,b]`, i.e. the first element of the list is the function/predicate symbol, followed by the arguments.

Example: A complete first-order Tableau Theorem Prover

```
prove((E,F),A,B,C,D) :- !,prove(E,[F|A],B,C,D).
prove((E;F),A,B,C,D) :- !,prove(E,A,B,C,D),prove(F,A,B,C,D).
prove(all(I,J),A,B,C,D) :- !,
   \+length(C,D),copy_term((I,J,C),(G,F,C)),
   append(A,[all(I,J)],E),prove(F,E,B,[G|C],D).
prove(A,_,[C|D],_,_) :-
    ((A= -(B);-(A)=B) -> (unify(B,C);prove(A,[],D,_,_))).
prove(A,[E|F],B,C,D) :- prove(E,F,[A|B],C,D).
```


Index