

General Computer Science
320201 GenCS I & II Lecture Notes Spring 2010

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



©: Michael Kohlhase

1



1 Circuits

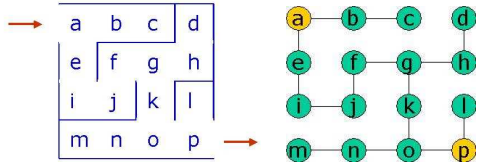
We will now study a new model of computation that comes quite close to the circuits that execute computation on today's computers. Since the course studies computation in the context of computer science, we will abstract away from all physical issues of circuits, in particular the construction of gates and timing issues. This allows us to present a very mathematical view of circuits at the level of annotated graphs and concentrate on qualitative complexity of circuits. Some of the material in this section is inspired by [?].

We start out our foray into circuits by laying the mathematical foundations of graphs and trees in Subsection 1.1, and then build a simple theory of combinational circuits in Subsection 1.2 and study their time and space complexity in Subsection 1.3. We introduce combinational circuits for computing with numbers, by introducing positional number systems and addition in Subsection 1.4 and covering 2s-complement numbers and subtraction in Subsection 1.5. A basic introduction to sequential logic circuits and memory elements in Subsection 1.6 concludes our study of circuits.

1.1 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)



©: Michael Kohlhase

2



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 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 green circles)
 - ▷ $E \subseteq \{\{v, v'\} \mid v, v' \in V, v \neq v'\}$ is the set of its **undirected edges** (draw as lines)
 - ▷ **Definition 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 3** Given a graph $G = \langle V, E \rangle$. The **in-degree** $\text{indeg}(v)$ and the **out-degree** $\text{outdeg}(v)$ of a vertex $v \in V$ are defined as
 - ▷ $\text{indeg}(v) = \#\{\{w \mid \langle w, v \rangle \in E\}$
 - ▷ $\text{outdeg}(v) = \#\{\langle v, w \rangle \mid \langle v, w \rangle \in E\}$
- Note:** For an undirected graph, $\text{indeg}(v) = \text{outdeg}(v)$ for all nodes v .



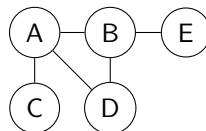
We will mostly concentrate on directed graphs in the following, since they are most important for the applications we have in mind. Many of the notions can be defined for undirected graphs with a little imagination. For instance the definitions for indeg and outdeg are the obvious variants: $\text{indeg}(v)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

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

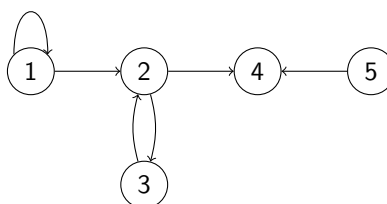
This is a very abstract yet elementary definition. We only need very basic concepts like sets and ordered pairs to understand them. The main difference between directed and undirected graphs can be visualized in the graphic representations below:

Examples

- ▷ **Example 4** An undirected graph $G_1 = \langle V_1, E_1 \rangle$, where $V_1 = \{A, B, C, D, E\}$ and $E_1 = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{B, E\}\}$



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



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!)



©: Michael Kohlhase

5



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

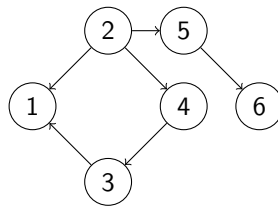
▷ **Definition 6** Let $G = \langle V, E \rangle$ be a directed graph, then we call a node $v \in V$

▷ **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 7** The node 2 is initial, and the nodes 1 and 6 are terminal in



©: Michael Kohlhase

6



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.

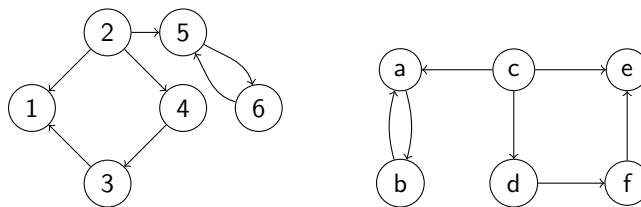
Graph Isomorphisms

▷ **Definition 8** A **graph isomorphism** between two graphs $G = \langle V, E \rangle$ and $G' = \langle V', E' \rangle$ is a bijective function $\psi: V \rightarrow V'$ 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 9** Two graphs G and G' are **equivalent** iff there is a graph-isomorphism ψ between G and G' .

▷ **Example 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.



©: Michael Kohlhase

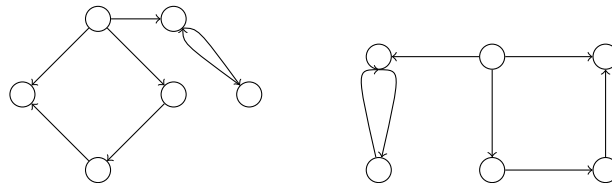
7



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 priori these two have nothing to do with each other. They are certainly not equivalent via the graph equivalence 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 11** A **labeled graph** G is a triple $\langle V, E, f \rangle$ where $\langle V, E \rangle$ is a graph and $f: V \cup E \rightarrow R$ is a partial function into a set R of **labels**.
- ▷ **Notation 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 13** $G = \langle V, E, f \rangle$ with $V = \{A, B, C, D, E\}$, where
 - ▷ $E = \{\langle A, A \rangle, \langle A, B \rangle, \langle B, C \rangle, \langle C, B \rangle, \langle B, D \rangle, \langle E, D \rangle\}$
 - ▷ $f: V \cup E \rightarrow \{+, -, \emptyset\} \times \{1, \dots, 9\}$ with
 - ▷ $f(A) = 5, f(B) = 3, f(C) = 7, f(D) = 4, f(E) = 8,$
 - ▷ $f(\langle A, A \rangle) = -0, f(\langle A, B \rangle) = -2, f(\langle B, C \rangle) = +4,$
 - ▷ $f(\langle C, B \rangle) = -4, f(\langle B, D \rangle) = +1, f(\langle E, D \rangle) = +4$

SOME RIGHTS RESERVED

©: Michael Kohlhase

8

JACOBS
UNIVERSITY

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 14** Given a directed graph $G = \langle V, E \rangle$, then we call a vector $p = \langle v_0, \dots, v_n \rangle \in V^{n+1}$ 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 $\text{start}(p)$)

▷ v_n is called the **end** of p (write $\text{end}(p)$)

▷ n is called the **length** of p (write $\text{len}(p)$)

Note: Not all v_i -s in a path are necessarily different.

▷ **Notation 15** For a graph $G = \langle V, E \rangle$ and a path $p = \langle v_0, \dots, v_n \rangle \in V^{n+1}$, 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 16** We write $\Pi(G)$ for the set of all paths in a graph G .



©: Michael Kohlhase

9



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 17** Given a graph $G = \langle V, E \rangle$, then

▷ a path p is called **cyclic** (or a **cycle**) iff $\text{start}(p) = \text{end}(p)$.

▷ a cycle $\langle v_0, \dots, v_n \rangle$ 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 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)



©: Michael Kohlhase

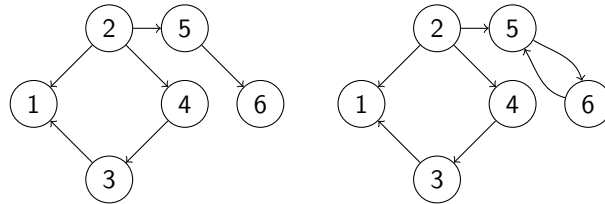
10



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 19** Let $G := \langle V, E \rangle$ be a digraph, then the **depth** $\text{dp}(v)$ of a vertex $v \in V$ is defined to be 0, if v is a source of G and $\sup\{\text{len}(p) \mid \text{indeg}(\text{start}(p)) = 0 \wedge \text{end}(p) = v\}$ otherwise, i.e. the length of the longest path from a source of G to v . (\triangle can be infinite)
- ▷ **Definition 20** Given a digraph $G = \langle V, E \rangle$. The **depth** ($\text{dp}(G)$) of G is defined as $\sup\{\text{len}(p) \mid p \in \Pi(G)\}$, i.e. the maximal path length in G .
- ▷ **Example 21** The vertex 6 has depth two in the left graphs and infinite 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)



©: Michael Kohlhase

11



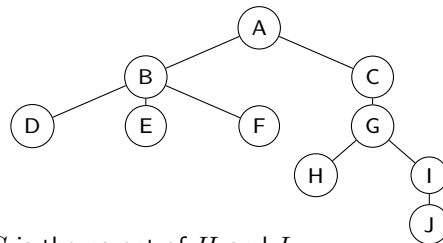
We now come to a very important special class of graphs, called trees.

Trees

- ▷ **Definition 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 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 24** For any node $v \in V$ except the root v_r , there is exactly one path $p \in \Pi(G)$ with $\text{start}(p) = v_r$ and $\text{end}(p) = v$. (*proof by induction on the number of nodes*)



©: Michael Kohlhase

12



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.

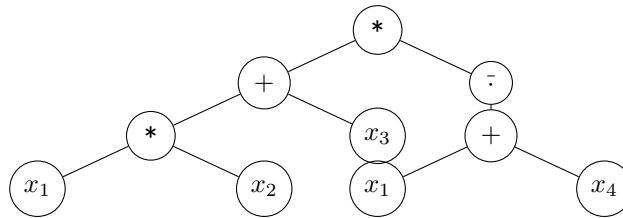
This allows us to view Boolean expressions as trees and apply all the mathematics (nomenclature and results) we will develop for them.

The Parse-Tree of a Boolean Expression

▷ **Definition 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 $e = \bar{e}'$ 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 $V_e := V_{e_1} \cup V_{e_2} \cup \{v\}$, $E_e := E_{e_1} \cup E_{e_2} \cup \{\langle v, v'_1 \rangle, \langle v, v'_2 \rangle\}$, 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 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 26** the parse tree of $x_1 * x_2 + x_3 * \overline{x_1 + x_4}$ is



©: Michael Kohlhase

13



Introduction to Combinatorial Circuits

1.2 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

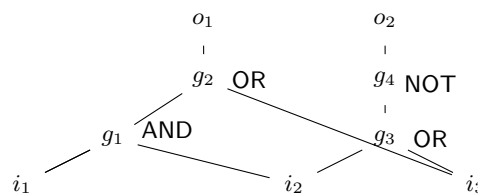
▷ **Definition 27** A **combinatorial circuit** is a labeled acyclic graph $G = \langle V, E, f_g \rangle$ with label set $\{\text{OR}, \text{AND}, \text{NOT}\}$, such that

▷ $\text{indeg}(v) = 2$ and $\text{outdeg}(v) = 1$ for all nodes $v \in (f_g)^{-1}(\{\text{AND}, \text{OR}\})$

▷ $\text{indeg}(v) = \text{outdeg}(v) = 1$ for all nodes $v \in (f_g)^{-1}(\{\text{NOT}\})$

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

▷ **Example 28** The following graph $G_{\text{cir1}} = \langle V, E \rangle$ is a combinatorial circuit



▷ **Definition 29** 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)

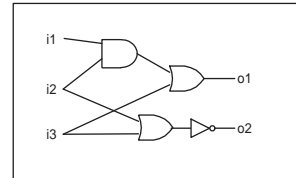
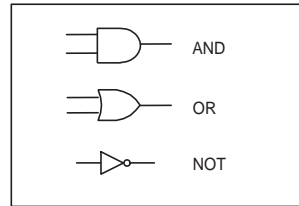


So combinatorial circuits are simply a class of specialized labeled directed graphs. As such, they inherit the nomenclature and equality conditions we introduced for graphs. The motivation for the restrictions is simple, we want to model computing devices based on gates, i.e. simple computational devices 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.

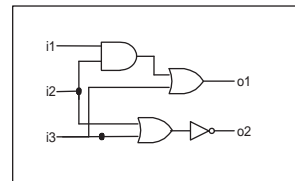
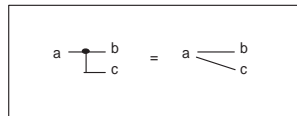
Since combinatorial circuits are a primary tool for understanding logic circuits, they have their own traditional visual display format. Gates are drawn with special node shapes and edges are traditionally drawn on a rectangular grid, using bifurcating edges instead of multiple lines with blobs distinguishing 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



©: Michael Kohlhase

15



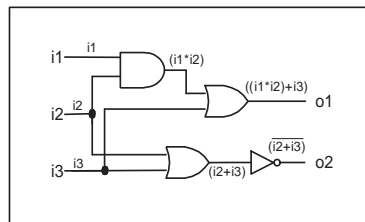
In particular, the diagram on the lower right is a visualization for the combinatory circuit G_{circ1} from the last slide.

To view combinational 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 combinational 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 combinational circuit with Boolean Expressions!
- ▷ **Definition 30** Given a combinational 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	$v \in I(G)$
$\overline{f_L(\langle u, v \rangle)}$	$f_g(v) = \text{NOT}$
$f_L(\langle u, v \rangle) * f_L(\langle u', v \rangle)$	$f_g(v) = \text{AND}$
$f_L(\langle u, v \rangle) + f_L(\langle u', v \rangle)$	$f_g(v) = \text{OR}$



©: Michael Kohlhase

16



Armed with the expression label of edges we can now make the computational behavior of combinational circuits explicit. The intuition is that a combinational 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 31** A combinatorial circuit $G = \langle V, E, f_g \rangle$ with input vertices i_1, \dots, i_n and output vertices o_1, \dots, o_m **computes** an n -ary Boolean function

$$f: \{0, 1\}^n \rightarrow \{0, 1\}^m; \langle i_1, \dots, i_n \rangle \mapsto \langle f_{e_1}(i_1, \dots, i_n), \dots, f_{e_m}(i_1, \dots, i_n) \rangle$$

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

▷ **Example 32** The circuit example on the last slide defines the Boolean function $f: \{0, 1\}^3 \rightarrow \{0, 1\}^2; \langle i_1, i_2, i_3 \rangle \mapsto \langle f_{i_1 * i_2 + i_3}, f_{\overline{i_2 * i_3}} \rangle$

▷ **Definition 33** 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 .



©: Michael Kohlhase

17



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 .

Realizing Complex Gates Efficiently

1.3 Realizing Complex Gates Efficiently

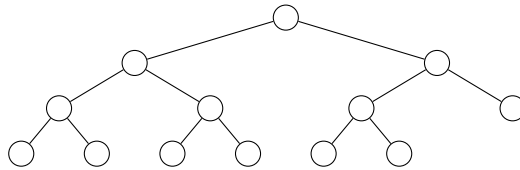
The main properties of combinatory circuits we are interested in studying will be the the number of gates and the depth of a circuit. The number of gates is of practical importance, since it is a measure of the cost that is needed for producing the circuit in the physical world. The depth is interesting, 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.

1.3.1 Balanced Binary Trees

Balanced Binary Trees

- ▷ **Definition 34** A **binary tree** is a tree where all nodes have out-degree 2 or 0.
 - ▷ **Definition 35** 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 $\text{outdeg}(u) = 0$, $(E_i := E_{i-1} \cup \{\langle u, v \rangle, \langle u, w \rangle\})$
 - ▷ repeat steps 2 and 3 until $i = n$ $(V = V_n, E = E_n)$
- ▷ **Example 36** 7 leaves



©: Michael Kohlhase

18



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 37** Let $G = \langle V, E \rangle$ be a balanced binary tree of depth $n > i$, then the set $V_i := \{v \in V \mid \text{dp}(v) = i\}$ 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 $\text{dp}(G) = n > i$, 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 38** A fully balanced tree of depth d has $2^{d+1} - 1$ nodes.
 - ▷ **Proof:**
 - P.1** Let $G := \langle V, E \rangle$ be a fully balanced tree
 - P.2** Then $\#(V) = \sum_{i=0}^d 2^i = 2^{d+1} - 1$. □



©: Michael Kohlhase

19



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 39** 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 $V' := V \setminus W$, 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 $d = \log_2(\#(V)) - \log_2(c) = \lfloor \log_2(\#(V)) \rfloor$. □



©: Michael Kohlhase

20



Leaves of Binary Trees

▷ **Lemma 40** 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 $\#(V) = 2m - 3 + 2 = 2m - 1$. □



©: Michael Kohlhase

21



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

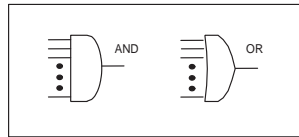
1.3.2 Realizing n -ary Gates

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 41** $\text{AND}x_1, \dots, x_n := \prod_{i=1}^n x_i$ and $\text{OR}x_1, \dots, x_n := \sum_{i=1}^n x_i$
- ▷ **Note:** These can be realized as balanced binary trees G_n
- ▷ **Corollary 42** $C(G_n) = n - 1$ and $dp(G_n) = \lceil \log_2(n) \rceil$.

▷ **Notation 43**



©: Michael Kohlhase

22



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 44** *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 $\lceil \log_2(2^n) \rceil = n$. □



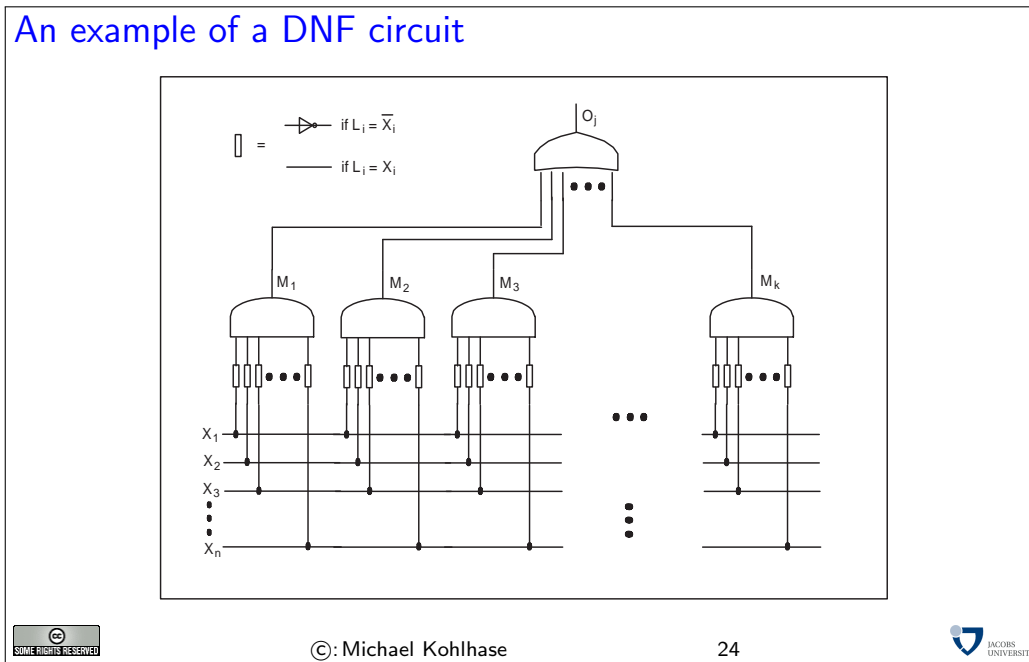
©: Michael Kohlhase

23



Of course, the depth result is related to the first worst-case complexity result for Boolean expressions (??); 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 combinational circuit.

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.

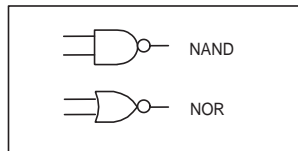
Of course, by now, we know how to do better in practice. Instead of the DNF, we can always compute the minimal polynomial for a given Boolean function using the Quine-McCluskey algorithm and derive a combinatorial circuit from this. While this does not give us any theoretical mileage (there are Boolean 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	NOR	0	1
0	1	1	0	1	0
1	1	0	1	0	0

▷ Corresponding logical connectives are written as \uparrow (NAND) and \downarrow (NOR).

▷ We will also need the **exclusive or** (XOR) connective that returns 1 iff either of its operands is 1.

XOR	0	1
0	0	1
1	1	0

▷ The gate is written as $\overset{a}{\curvearrowright} \overset{b}{\curvearrowright}$ XOR(a,b), the logical connective as \oplus .



©: Michael Kohlhase

25



The Universality of NAND and NOR

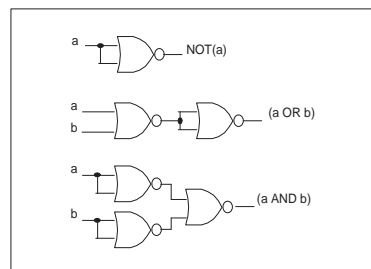
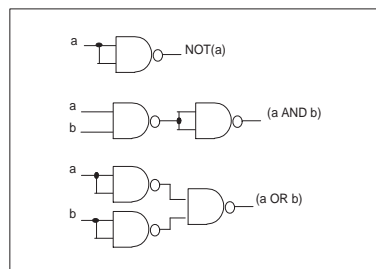
▷ **Theorem 45** NAND and NOR are universal; i.e. any Boolean function can be expressed in terms of them.

▷ **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.



©: Michael Kohlhase

26



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.

Basic Arithmetics with Combinational Circuits

1.4 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.

1.4.1 Positional Number Systems

Positional Number Systems

- ▷ **Problem:** For realistic arithmetics we need better number representations than the unary natural numbers $(|\varphi_n(\text{unary})|) \in \Theta(n)$ [number of /]
- ▷ **Recap:** the unary number system
 - ▷ build up numbers from /es (start with ' ' and add /)
 - ▷ addition \oplus as concatenation $(\odot, \oplus, \exp, \dots$ 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 46** 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})
 - ▷ $\varphi_b: D_b \rightarrow \{\epsilon, /, \dots, /^{[b-1]}\}$ is bijective (first b unary numbers)
 - ▷ $\psi_b: D_b^+ \rightarrow \{/\}^*; \langle n_k, \dots, n_1 \rangle \mapsto \bigoplus_{i=1}^k ((\varphi_b(n_i) \odot \exp(/^{[b]}/^{[i-1]}))$ (extends φ_b to string code)


©:Michael Kohlhase
27


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 $|\varphi_n(\text{unary})|$ is the “cost of representing n in unary representation”, we get $(|\varphi_n(\text{unary})|) \in \Theta(n)$. 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 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 later¹ that this give us a bijection between D_b^+ and the unary natural numbers.

EdNote(1)

Commonly Used Positional Number Systems

▷ **Example 47** 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 48** 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)

$$\triangleright \triangleright 110001101011100_2 = \underbrace{0110}_6 \underbrace{0011}_3 \underbrace{0101}_5 \underbrace{1100}_4 = 635C_{16}$$

$$\triangleright \triangleright 110001101011100_2 = \underbrace{110}_6 \underbrace{001}_1 \underbrace{101}_5 \underbrace{011}_3 \underbrace{100}_4 = 61534_8$$

$$\triangleright \triangleright FF3A16 = \underbrace{F}_{16} \underbrace{3}_{16} \underbrace{A}_{16} = 111100111010_2, \quad 4721_8 = \underbrace{4}_4 \underbrace{7}_7 \underbrace{2}_2 \underbrace{1}_1 = 100111010001_2$$



©: Michael Kohlhase

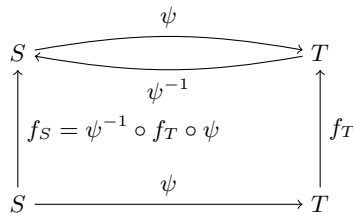
28



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 $f_T: T \rightarrow T$ 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 $f_S: S \rightarrow S$ by $f_S(s) := \psi^{-1}(f_T(\psi(s)))$ according to the following diagram.

¹EDNOTE: reference





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

- Idea:** use this to define arithmetics on \mathcal{N} .
- ▷ **Definition 50** Let $\mathcal{N} := \langle D_b, \varphi_b, \psi_b \rangle$ be a PNS of base b , then we define a binary function $+_b: \mathbb{N}_b \times \mathbb{N}_b \rightarrow \mathbb{N}_b$ by $x+_by := \psi_b^{-1}(\psi_b(x) \oplus \psi_b(y))$.
- ▷ **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)


©: Michael Kohlhase
29


1.4.2 Adders

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 51** 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 $a = \langle a_{n-1}, \dots, a_0 \rangle \in \mathbb{B}^n$, the notation $\langle\langle a \rangle\rangle$ is used, i.e.,

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

▷ **Definition 52** An n -bit **adder** is a circuit computing the function $f_{+2}^n : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}^{n+1}$ with

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



©: Michael Kohlhase

30



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 53 A **half adder** is a circuit HA implementing the function f_{HA} in the truth table on the right.

▷ $f_{\text{HA}} : \mathbb{B}^2 \rightarrow \mathbb{B}^2 \quad \langle a, b \rangle \mapsto \langle c, s \rangle$

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

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

▷ **Note:** The carry can be computed by a simple AND, i.e., $c = \text{AND}(a, b)$, and the sum bit by a XOR function.

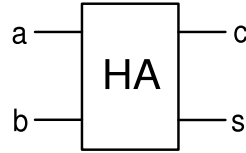
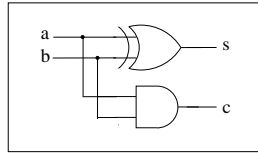


©: Michael Kohlhase

31



Building and Evaluating the Half-Adder



▷ So, the half-adder corresponds to the Boolean function $f_{\text{HA}}: \mathbb{B}^2 \rightarrow \mathbb{B}^2; \langle a, b \rangle \mapsto \langle a \oplus b, a \wedge b \rangle$

▷ **Note:** $f_{\text{HA}}(a, b) = B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle)$, i.e., it is indeed an adder.

▷ We count XOR as one gate, so $C(\text{HA}) = 2$ and $\text{dp}(\text{HA}) = 1$.



©: Michael Kohlhase

32



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

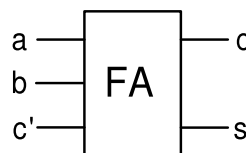
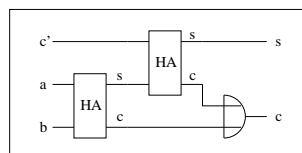
▷ **Definition 54** The 1-bit **full adder** is a circuit FA^1 that implements the function $f_{\text{FA}}^1: \mathbb{B} \times \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}^2$ with $\text{FA}^1(a, b, c') = B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + \langle\langle c' \rangle\rangle)$

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

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

▷ **Lemma 55 (Cost and Depth)** $C(\text{FA}^1) = 2C(\text{HA}) + 1 = 5$ and $\text{dp}(\text{FA}^1) = 2\text{dp}(\text{HA}) + 1 = 3$

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



©: Michael Kohlhase

33



Note: Note that in the right hand graphics, we use another notation for the OR gate.²

EdNote(2)

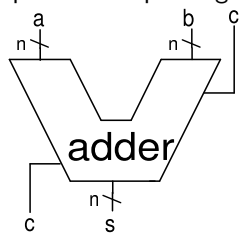
Of course adding single digits is a rather simple task, and hardly worth the effort, if this is all we can do. What we are really after, are circuits that will add n -bit binary natural numbers, so that we arrive at computer chips that can add long numbers for us.

Full n -bit Adder

▷ **Definition 56** An n -bit full adder ($n > 1$) is a circuit that corresponds to $f_{FA}^n : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{B}^n; \langle a, b, c' \rangle \mapsto B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + \langle\langle c' \rangle\rangle)$

▷ **Notation 57** 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



©: Michael Kohlhase

34



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

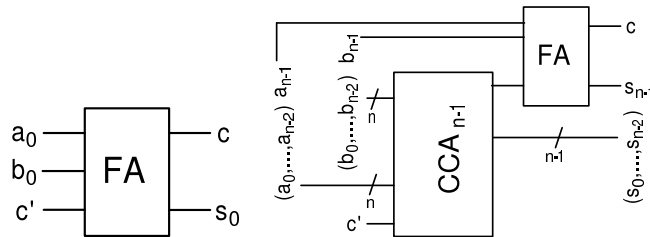
- adding the two digits with carry from the previous column
- recording the sum bit as the result, and
- passing the carry bit on to the next column

until one of the numbers ends.

²EDNOTE: Todo: introduce this earlier, or change the graphics here (or both)

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} and the last two bits of a and b

- ▷ **Definition 58** An n -bit carry chain adder CCA^n is inductively defined as

- ▷ $f_{CCA}^1(a_0, b_0, c) = FA^1(a_0, b_0, c)$
- ▷ $f_{CCA}^n(\langle a_{n-1}, \dots, a_0 \rangle, \langle b_{n-1}, \dots, b_0 \rangle, c') = \langle c, s_{n-1}, \dots, s_0 \rangle$ with
 - ▷ $\langle c, s_{n-1} \rangle = FA^{n-1}(a_{n-1}, b_{n-1}, c_{n-1})$
 - ▷ $\langle c_{n-1}, \dots, c_s \rangle_0 = f_{CCA}^{n-1}(\langle a_{n-2}, \dots, a_0 \rangle, \langle b_{n-2}, \dots, b_0 \rangle, c')$

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

- ▷ **Lemma 59 (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).

- ▷ **Definition 60** An n -bit **conditional sum adder** CSA^n is recursively defined as
 - ▷ $f_{CSA}^n(\langle a_{n-1}, \dots, a_0 \rangle, \langle b_{n-1}, \dots, b_0 \rangle, c') = \langle c, s_{n-1}, \dots, s_0 \rangle$ where
 - ▷ $\langle c_{n/2}, s_{n/2-1}, \dots, s_0 \rangle = f_{CSA}^{n/2}(\langle a_{n/2-1}, \dots, a_0 \rangle, \langle b_{n/2-1}, \dots, b_0 \rangle, c')$
 - ▷ $\langle c, s_{n-1}, \dots, s_{n/2} \rangle = \begin{cases} f_{CSA}^{n/2}(\langle a_{n-1}, \dots, a_{n/2} \rangle, \langle b_{n-1}, \dots, b_{n/2} \rangle, 0) & \text{if } c_{n/2} = 0 \\ f_{CSA}^{n/2}(\langle a_{n-1}, \dots, a_{n/2} \rangle, \langle b_{n-1}, \dots, b_{n/2} \rangle, 1) & \text{if } c_{n/2} = 1 \end{cases}$
 - ▷ $f_{CSA}^1(a_0, b_0, c) = FA^1(a_0, b_0, c)$

©: Michael Kohlhase

36

JACOBS UNIVERSITY

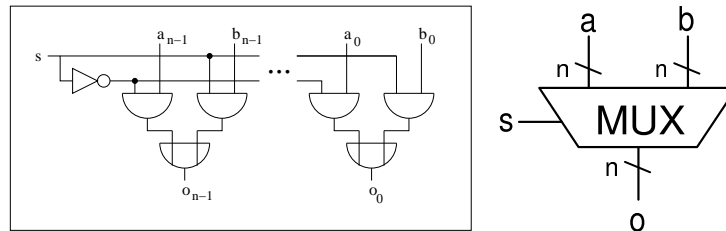
The only circuit that we still have to look at is the one that chooses the correct upper sums. Fortunately, this is a rather simple design that makes use of the classical trick that “if C , then A , else B ” can be expressed as “ $(C \text{ and } A) \text{ or } (\neg C \text{ and } B)$ ”.

The Multiplexer

▷ **Definition 61** An n -bit **multiplexer** MUX^n is a circuit which implements the function $f_{\text{MUX}}^n: \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B}^n$ with

$$f(a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0, s) = \begin{cases} \langle a_{n-1}, \dots, a_0 \rangle & \text{if } s = 0 \\ \langle b_{n-1}, \dots, b_0 \rangle & \text{if } s = 1 \end{cases}$$

▷ **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:** $C(\text{MUX}^n) = 3n + 1$ and $\text{dp}(\text{MUX}^n) = 3$.



©: Michael Kohlhase

37



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

▷ $\text{dp}(\text{CSA}^n) = \text{dp}(\text{CSA}^{n/2}) + \text{dp}(\text{MUX}^{n/2+1})$

▷ solve the recursive equation:

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



©: Michael Kohlhase

38



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

▷ $C(\text{CSA}^n) = 3C(\text{CSA}^{n/2}) + C(\text{MUX}^{n/2+1})$.

▷ **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 62 (Master's Theorem for Recursive Equations)** Given the recursively defined function $f: \mathbb{N} \rightarrow \mathbb{R}$, such that $f(1) = c \in \mathbb{R}$ and $f(b^k) = af(b^{k-1}) + g(b^k)$ for some $a \in \mathbb{R}$, $1 \leq a$, $k \in \mathbb{N}$, and $g: \mathbb{N} \rightarrow \mathbb{R}$, then $f(b^k) = ca^k + \sum_{i=0}^{k-1} a^i g(b^{k-i})$

▷ We have $C(\text{CSA}^n) = 3C(\text{CSA}^{n/2}) + C(\text{MUX}^{n/2+1}) = 3C(\text{CSA}^{n/2}) + 3(n/2 + 1) + 1 = 3C(\text{CSA}^{n/2}) + \frac{3}{2}n + 4$

▷ So, $C(\text{CSA}^n)$ 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 $c = C(\text{FA}^1) = C(\text{FA}^1) = 5$

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

▷ **Note:** $a^{\log_2(n)} = 2^{\log_2(a) \log_2(n)} = 2^{\log_2(a) \cdot \log_2(n)} = 2^{\log_2(n) \log_2(a)} = n^{\log_2(a)}$

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

▷ **Theorem 63** The cost and the depth of the conditional sum adder are in the following complexity classes:

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

▷ **Compare with:** $C(\text{CCA}^n) \in O(n)$ $dp(\text{CCA}^n) \in O(n)$

▷ 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 **carry lookahead adder** CLA^n has a cost $C(\text{CLA}^n) \in O(n)$ and a depth of $dp(\text{CLA}^n) \in O(\log_2(n))$.



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.

Arithmetics for Two's Complement Numbers

1.5 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 **sign bit**. (additional leading bit that indicates whether the number is positive)
- ▷ **Definition 64** ($(n + 1)$ -bit signed binary number system)

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

- ▷ **Note:** We need to fix string length to identify the sign bit. (leading zeroes)
- ▷ **Example 65** 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 .)
- ▷ $10\dots 0$ and $00\dots 0$ 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!



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 66** Given the binary string $a = \langle a_n, \dots, a_0 \rangle \in \mathbb{B}^{n+1}$, 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 67** 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.				\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	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



©: Michael Kohlhase

42



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 $b = \langle b_n, \dots, b_0 \rangle$ 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 $B_n^{2s}(0) = \langle 0, \dots, 0 \rangle$.
- ▷ This number system has an asymmetric range $\mathcal{R}_n^{2s} := \{-2^n, \dots, 2^n - 1\}$.



©: Michael Kohlhase

43



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 its negative sibling) can be achieved by two very simple operations: flipping all the zeros and ones, and incrementing.

The Structure Theorem for TCN

▷ **Theorem 68** Let $a \in \mathbb{B}^{n+1}$ be a binary string, then $-\langle\langle a \rangle\rangle_n^{2^s} = \langle\langle \bar{a} \rangle\rangle_n^{2^s} + 1$.

▷ **Proof:** by calculation using the definitions

$$\begin{aligned}
 \langle\langle \bar{a}_n, \bar{a}_{n-1}, \dots, \bar{a}_0 \rangle\rangle_n^{2^s} &= -\bar{a}_n \cdot 2^n + \langle\langle \bar{a}_{n-1}, \dots, \bar{a}_0 \rangle\rangle \\
 &= \bar{a}_n \cdot (-2^n) + \sum_{i=0}^{n-1} \bar{a}_i \cdot 2^i \\
 &= (1 - a_n) \cdot (-2^n) + \sum_{i=0}^{n-1} (1 - a_i) \cdot 2^i \\
 &= (1 - a_n) \cdot (-2^n) + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} a_i \cdot 2^i \\
 &= -2^n + a_n \cdot 2^n + 2^{n-1} - \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle \\
 &= (-2^n + 2^n) + a_n \cdot 2^n - \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle - 1 \\
 &= -(a_n \cdot (-2^n) + \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle) - 1 \\
 &= -\langle\langle a \rangle\rangle_n^{2^s} - 1
 \end{aligned}$$

□



©: Michael Kohlhase

44



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 $B(z) = \langle b_{n-1}, \dots, b_0 \rangle$
- ▷ complement it to $\overline{B(z)}$, i.e., the bitwise negation \bar{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 $b = \langle b_{n-1}, \dots, b_0 \rangle$, 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 $-\langle\langle \overline{B(\langle\langle b \rangle\rangle - 1)} \rangle\rangle$



©: Michael Kohlhase

45



Subtraction and Two's Complement Numbers

▷ **Idea:** With negative numbers use our adders directly

▷ **Definition 69** An n -bit **subtractor** is a circuit that implements the function $f_{\text{SUB}}^n: \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{B}^n$ such that

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

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 $\langle\langle a \rangle\rangle_n^{2s} - \langle\langle b \rangle\rangle_n^{2s} = \langle\langle a \rangle\rangle_n^{2s} + (-\langle\langle b \rangle\rangle_n^{2s}) = \langle\langle a \rangle\rangle_n^{2s} + \langle\langle \bar{b} \rangle\rangle_n^{2s} + 1$

▷ **Idea:** Can we implement an n -bit subtractor as $f_{\text{SUB}}^n(a, b, b') = \text{FA}^n(a, \bar{b}, \bar{b}')$?

▷ **not immediately:** We have to make sure that the full adder plays nice with two's complement numbers



©: Michael Kohlhase

46



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 70** An n -bit **two's complement adder** ($n > 1$) is a circuit that corresponds to the function $f_{\text{TCA}}^n: \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{B}^n$, such that $f_{\text{TCA}}^n(a, b, c') = B_n^{2s}(\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c')$ for all $a, b \in \mathbb{B}^n$ and $c' \in \mathbb{B}$.

▷ **Theorem 71** $f_{\text{TCA}}^n = f_{\text{FA}}^n$ *(first prove some Lemmas)*



©: Michael Kohlhase

47



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 $\text{TCA} \text{circ} \text{FN} n$ 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 72** Let $a = \langle a_n, \dots, a_0 \rangle \in \mathbb{B}^{n+1}$ be a binary string, then $\langle\langle a_n, \dots, a_0 \rangle\rangle_{n+1}^{2s} = \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle_n^{2s}$.

▷ **Proof:** by calculation

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

□



©: Michael Kohlhase

48



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

▷ **Definition 73** Let $a, b \in \mathbb{B}^{n+1}$ and $c \in \mathbb{B}$ with $a = \langle a_n, \dots, a_0 \rangle$ and $b = \langle b_n, \dots, b_0 \rangle$, then we call $ic_k(a, b, c)$, the k -th intermediate carry of a , b , and c , iff

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

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

▷ **Theorem 74** Let $a, b \in \mathbb{B}^n$ and $c \in \mathbb{B}$, then

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 $\langle\langle ic_{n+1}(a, b, c), s_n, \dots, s_0 \rangle\rangle = \langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + c$.



©: Michael Kohlhase

49



Unfortunately, the proof of this attractive and useful theorem is quite tedious and technical

Proof of the TCN Main Theorem

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

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

P.2 We have to consider three cases

P.2.1 $a_n = b_n = 0$:

P.2.1.1 $\langle\langle a \rangle\rangle_n^{2s}$ and $\langle\langle b \rangle\rangle_n^{2s}$ are both positive, so $\text{ic}_{n+1}(a, b, c) = 0$ and furthermore

$$\begin{aligned} \text{ic}_n(a, b, c) = 0 &\Leftrightarrow \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \leq 2^n - 1 \\ &\Leftrightarrow \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c \leq 2^n - 1 \end{aligned}$$

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

P.2.2 $a_n = b_n = 1$:

P.2.2.1 $\langle\langle a \rangle\rangle_n^{2s}$ and $\langle\langle b \rangle\rangle_n^{2s}$ are both negative, so $\text{ic}_{n+1}(a, b, c) = 1$ and furthermore $\text{ic}_n(a, b, c) = 1$, iff $\langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \geq 2^n$, which is the case, iff $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c = -2^{n+1} + \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \geq -2^n$

$$\begin{aligned} \text{P.2.2.2 Hence, } \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c &= -2^n + \langle\langle a' \rangle\rangle + -2^n + \langle\langle b' \rangle\rangle + c && \square \\ &= -2^{n+1} + \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \\ &= -2^{n+1} + \langle\langle 1, s_{n-1}, \dots, s_0 \rangle\rangle \\ &= -2^n + \langle\langle s_{n-1}, \dots, s_0 \rangle\rangle \\ &= \langle\langle s \rangle\rangle_n^{2s} \end{aligned}$$

P.2.3 $a_n \neq b_n$:

P.2.3.1 Without loss of generality assume that $a_n = 0$ and $b_n = 1$.
(then $\text{ic}_{n+1}(a, b, c) = \text{ic}_n(a, b, c)$)

P.2.3.2 Hence, the sum of $\langle\langle a \rangle\rangle_n^{2s}$ and $\langle\langle b \rangle\rangle_n^{2s}$ is in the admissible range \mathcal{R}_n^{2s} as

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

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

$$\begin{aligned} \text{P.2.3.3 So we have } \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c &= -2^n + \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \\ &= -2^n + \langle\langle \text{ic}_n(a, b, c), s_{n-1}, \dots, s_0 \rangle\rangle \\ &= -(1 - \text{ic}_n(a, b, c)) \cdot 2^n + \langle\langle s_{n-1}, \dots, s_0 \rangle\rangle \\ &= \langle\langle \text{ic}_n(a, b, c), s_{n-1}, \dots, s_0 \rangle\rangle_n^{2s} \end{aligned}$$

P.2.3.4 Furthermore, we can conclude that $\langle\langle \text{ic}_n(a, b, c), s_{n-1}, \dots, s_0 \rangle\rangle_n^{2s} = \langle\langle s \rangle\rangle_n^{2s}$ as $s_n = a_n \oplus b_n \oplus \text{ic}_n(a, b, c) = 1 \oplus \text{ic}_n(a, b, c) = \text{ic}_n(a, b, c)$. \square

P.3 Thus we have considered all the cases and completed the proof. \square

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. In the case that s is larger than $2^n - 1$, we say that an **overflow** occurred. In the opposite error case of s being smaller than -2^n , we say that an **underflow** occurred.



©: Michael Kohlhasse

51

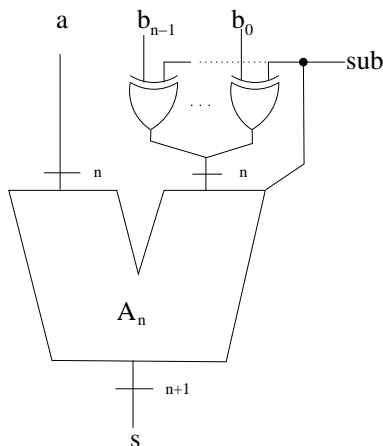


The most important application of the main TCN theorem is that we can build a combinatorial circuit that can add and subtract (depending on a control bit). This is actually the first instance of a concrete programmable computation device we have seen up to date (we interpret the control bit as a program, 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 ($\text{sub} = 1 \rightsquigarrow \text{subtract}$)
- ▷ If $\text{sub} = 0$, then the circuit acts like an adder ($a \oplus 0 = a$)
- ▷ If $\text{sub} = 1$, let $S := \langle\langle a \rangle\rangle_n^{2^s} + \langle\langle \overline{b_{n-1}}, \dots, \overline{b_0} \rangle\rangle_n^{2^s} + 1$ ($a \oplus 0 = 1 - a$)
- ▷ For $s \in \mathcal{R}_n^{2^s}$ the TCN main theorem and the TCN structure theorem together guarantee

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



- ▷ **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 ($+, -, *, /, n\text{-AND}, n\text{-OR}, \dots$)



©: Michael Kohlhasse

52



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

Sequential Logic Circuits and Memory Elements

1.6 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 75** Circuits that remember their current output or state are often called sequential logic circuits.
- ▷ **Example 76** 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



©: Michael Kohlhase

53

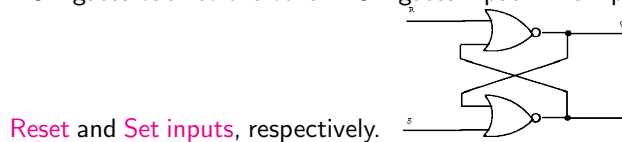


Clearly, sequential logic requires the ability to store the current state. In other words, *memory* is required by sequential logic circuits. We will investigate basic circuits that have the ability to store bits of data. We will start with the simplest possible memory element, and develop more elaborate 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 77** 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.



©: Michael Kohlhase

54



To understand the operation of the RS-flipflop we first remind 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*.

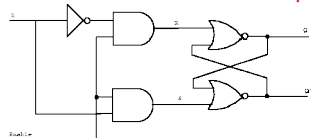
$S = 0$ **and** $R = 0$ Assume the flipflop is set ($Q = 1$ and $Q' = 0$), then the output of the top NOR gate remains at $Q = 1$ and the bottom NOR gate stays at $Q' = 0$. Similarly, when the flipflop is in a reset state ($Q = 0$ and $Q' = 1$), it will remain there with this input combination. Therefore, 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 78 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.

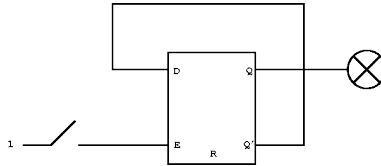
- ▷ When $E = 1$ the value of D determines the value of the output Q , when E returns to 0, the most recent input D is “remembered.”


©: Michael Kohlhase
55


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' output to its D input
(next state is inverse of current state)



©: Michael Kohlhase

56



In the on/off circuit, the external inputs (buttons) were connected to the E input.

Definition 79 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?



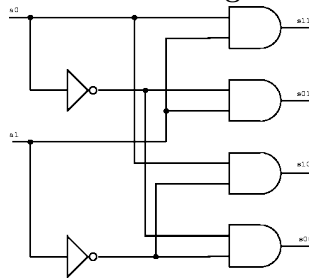
©: Michael Kohlhase

57



Address Decoder Logic

- ▷ **Idea:** Need a circuit that activates the storage element 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 element
- ▷ **Definition 80** The n -bit **address decoder** ADL^n has a n inputs and 2^n outputs. $f_{ADL}^m(a) = \langle b_1, \dots, b_{2^n} \rangle$, where $b_i = 1$, iff $i = \langle\langle a \rangle\rangle$.
- ▷ **Example 81 (Address decoder logic for 2-bit addresses)**



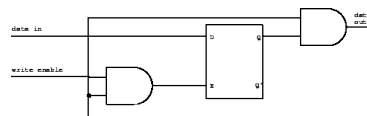
©: Michael Kohlhase

58



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 82** A Storage Element is given by the following diagram



©: Michael Kohlhase

59



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.



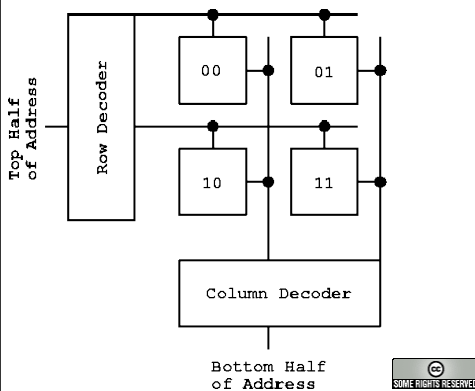
©: Michael Kohlhase

60



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.
- ▷ identify storage element 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 halves; the top half is used to select the row and the bottom half is used to select the column.



©: Michael Kohlhase

61

2 Machines

2.1 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, we just understand n -bit memory cells as devices that can store n binary values. 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 $P: \mathbb{N} \rightarrow \mathbb{B}^{2n}$)
 - ▷ **data store**: n -bit memory cells (words addressed by $D: \mathbb{N} \rightarrow \mathbb{B}^n$)
- ▷ add a **memory management unit**(MMU) (move values between RAM and registers)
- ▷ program it in **assembler language** (lowest level of programming)



©: Michael Kohlhase

62

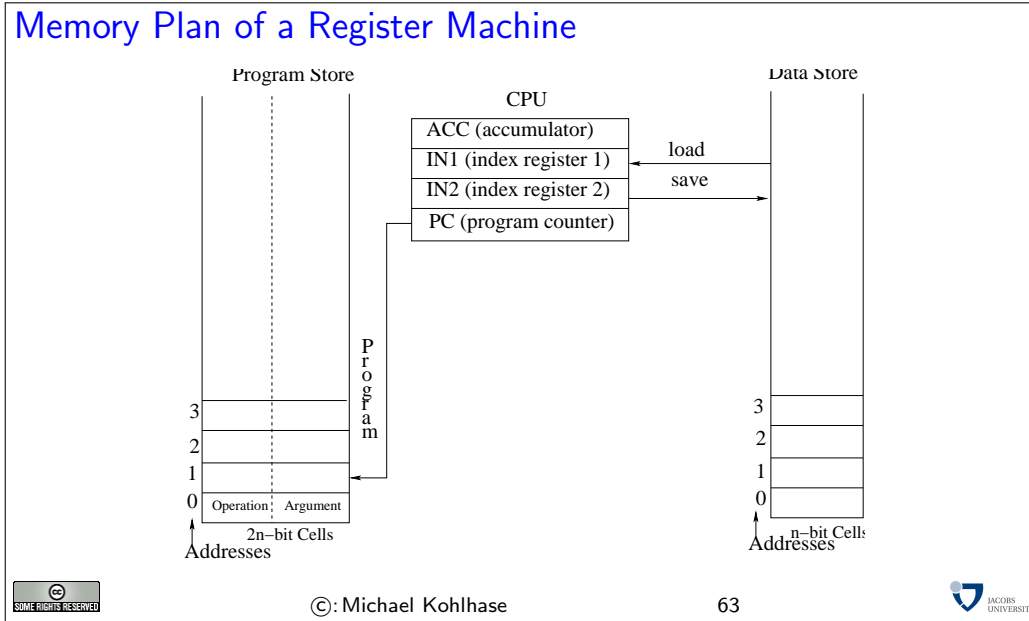


We have three kinds of memory areas in the REMA register machine: The registers (our architecture has two, which is the minimal number, real architectures have more for convenience) are just simple n -bit memory cells.

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 $P: \mathbb{N} \rightarrow \mathbb{B}^{2n}$. The data store is also RAM, but a sequence or n -bit cells, which is addressed by the function $D: \mathbb{N} \rightarrow \mathbb{B}^n$.

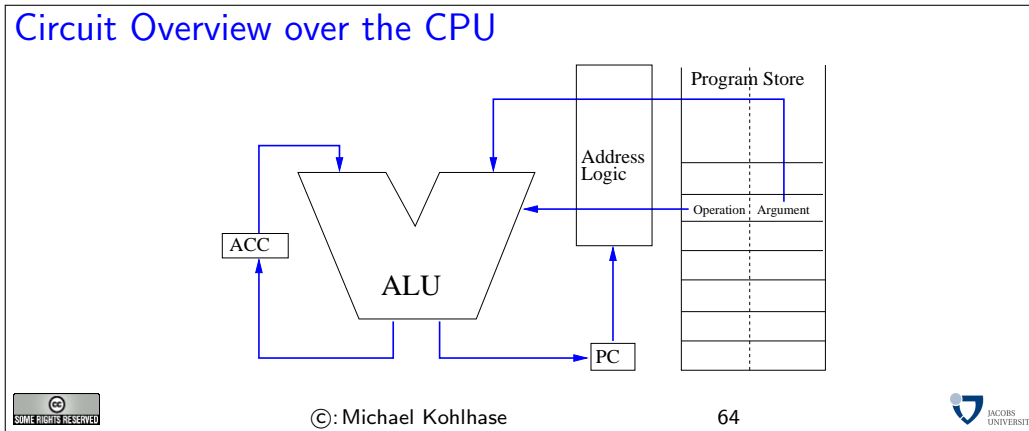
The value of the program counter is interpreted as a binary number that addresses a $2n$ -bit cell in the program store. The accumulator is the register that contains one of the inputs to the ALU before the operation (the other is given as the argument of the program instruction); the result of the ALU 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 if 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 83** **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


©: Michael Kohlhase
65


Definition 84 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 85 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 86** Exchange the values of cells 0 and 1 in the data store

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

- ▷ **Example 87** 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: = $D(1) = a$
1	ADD 2	ACC: = ACC + $D(2) = a + b$
2	ADD 3	ACC: = ACC + $D(3) = a + b + c$
3	STORE 4	$D(4)$: = ACC = $a + b + c$

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



©: Michael Kohlhase

66



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 to 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
<code>LOADIN $j\ i$</code>	ACC: = $D(INj + i)$	PC: = PC + 1	relative load
<code>STOREIN $j\ i$</code>	$D(INj + i)$: = ACC	PC: = PC + 1	relative store
<code>MOVE $S\ T$</code>	T : = S	PC: = PC + 1	move register S (source) to register T (target)

- ▷ **Problem Solution:**

P	instruction	comment
0	LOAD 0	ACC: = $D(0) = x$
1	MOVE ACC IN1	IN1: = ACC = x
2	LOAD 1	ACC: = $D(1) = y$
3	STOREIN 1 0	$D(x) = D(IN1 + 0)$: = ACC = y



©: Michael Kohlhase

67



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 above³, but we will not worry about this). EdNote(3)

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: = $\begin{cases} \text{PC} + i & \text{if } \mathcal{R}(\text{ACC}, 0) \\ \text{PC} + 1 & \text{else} \end{cases}$	conditional jump

▷ **Two more:**

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

SOME RIGHTS RESERVED

©: Michael Kohlhase

68

JACOBS UNIVERSITY

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.)

³EDNOTE: reference

Example Program

▷ **Example 88** Let $D(0) = n$, $D(1) = a$, and $D(2) = b$, copy the values of cells $a, \dots, a+n-1$ to cells $b, \dots, b+n-1$, while $a, b \geq 3$ and $|a-b| \geq n$.

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

▷ **Lemma 89** We have $D(0) = n - (i - 1)$, $\text{IN1} = a + i - 1$, and $\text{IN2} = b + i - 1$ for all $1 \leq i \leq n + 1$.
(the program does what we want)

▷ proof by induction on n .

▷ **Definition 90** The induction hypotheses are called **loop invariants**.



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

2.2.1 A Stack-based Virtual Machine

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*.

A **virtual machine** is a simple machine-language program that interprets a slightly higher-level program — the “**byte code**” — 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 91** An ASM program VM is called a **virtual machine** for a language $\mathcal{L}(VM)$, iff VM inputs a $\mathcal{L}(VM)$ 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 $\mathcal{L}(VM)$.
(simpler and more transparent)



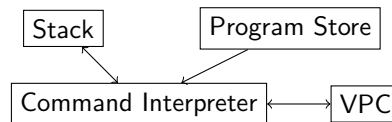
©: Michael Kohlhase

70



A Virtual Machine for Functional Programming

- ▷ 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 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.



©: Michael Kohlhase

71



A Stack-Based VM language (Arithmetic Commands)

▷ **Definition 92** VM Arithmetic Commands act on the stack

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

▷ **Example 93** The $\mathcal{L}(\text{VM})$ program “con 4 con 7 add” pushes $7 + 4 = 11$ to the stack.

▷ **Example 94** 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 · 2
con 3 con 4 mul	4 · 3
sub	4 · 3 - 7 · 2



©: Michael Kohlhase

72



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 95** 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 96** 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	



©: Michael Kohlhase

73



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 97** **Imperative access to variables:** Let $S(i)$ be the number at stack position i .

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

▷ **Example 98** The program “con 5 con 7 peek 0 peek 1 add poke 1 mul halt” computes $5 \cdot (7 + 5) = 60$.

©: Michael Kohlhase
74

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 99** Consider the following program that computes $12!$ and the corresponding $\mathcal{L}(\text{VM})$ 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 poking the stack position (must remember that)

©: Michael Kohlhase
75

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 $\mathcal{L}(\text{VM})$). We will use the example above as guiding intuition for a compiler from a simple imperative language to $\mathcal{L}(\text{VM})$ byte code below. But first we build a virtual machine for $\mathcal{L}(\text{VM})$.

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

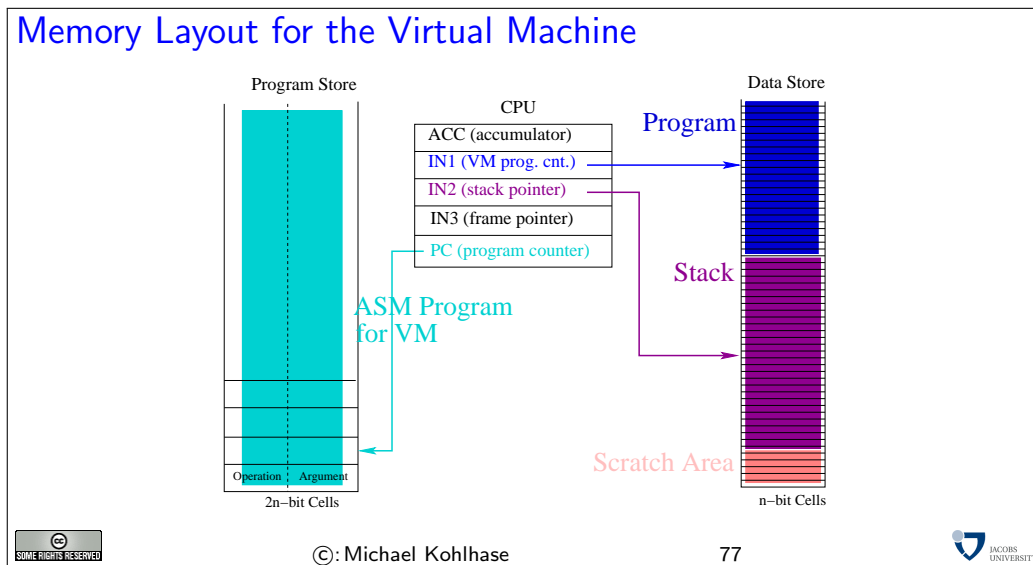
A Virtual Machine for $\mathcal{L}(\text{VM})$

- ▷ We need to build a concrete ASM program that acts as a virtual machine for $\mathcal{L}(\text{VM})$.
- ▷ 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 $\mathcal{L}(\text{VM})$ 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 $\mathcal{L}(\text{VM})$ 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 $\mathcal{L}(\text{VM})$ program have to be stored in the REMA data store (therefore we treat $\mathcal{L}(\text{VM})$ 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_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.



©: Michael Kohlhase

78



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 $\mathcal{L}(\text{VM})$ 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
⟨jt⟩	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
	LOADIN 1 0	ACC: = $D(\text{IN1})$	load instruction
	JUMP= ⟨halt⟩		goto ⟨halt⟩
	SUBI 1		next instruction code
	JUMP= ⟨add⟩		goto ⟨add⟩
	SUBI 1		next instruction code
	JUMP= ⟨sub⟩		goto ⟨sub⟩
⟨halt⟩	⋮	⋮	⋮
	STOP 0		stop
	⋮	⋮	⋮



©: Michael Kohlhase

79



Now it only remains to present the ASM programs for the individual $\mathcal{L}(\text{VM})$ 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 $\mathcal{L}(\text{VM})$ 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
<i><con></i>	<code>inc IN1</code>	$VPC := VPC + 1$	point to arg
	<code>inc IN2</code>	$SP := SP + 1$	prepare push
	<code>LOADIN 1 0</code>	$ACC := D(VPC)$	read arg
	<code>STOREIN 2 0</code>	$D(SP) := ACC$	store for push
	<code>inc IN1</code>	$VPC := VPC + 1$	point to next
	<code>JUMP <i><jt></i></code>		jump back
<i><add></i>	<code>LOADIN 2 0</code>	$ACC := D(SP)$	read arg 1
	<code>STORE 0</code>	$D(0) := ACC$	cache it
	<code>dec IN2</code>	$SP := SP - 1$	pop
	<code>LOADIN 2 0</code>	$ACC := D(SP)$	read arg 2
	<code>ADD 0</code>	$ACC := ACC + D(0)$	add cached arg 1
	<code>STOREIN 2 0</code>	$D(SP) := ACC$	store it
	<code>inc IN1</code>	$VPC := VPC + 1$	point to next
	<code>JUMP <i><jt></i></code>		jump back

▷ `sub`, `mul`, and `leq` similar to `add`.



©: Michael Kohlhase

80



For example, `mul` could be implemented as follows:

label	instruction	effect	comment
<i><mul></i>	<code>dec IN2</code>	$SP := SP - 1$	
	<code>LOADI 0</code>		initialize result
<i><loop></i>	<code>STORE 1</code>	$D(1) := 0$	read arg 1
	<code>LOADIN 2 1</code>	$ACC := D(SP + 1)$	initialize counter to arg 1
	<code>STORE 0</code>	$D(0) := ACC$	if counter=0, we are finished
	<code>JUMP_ <i><end></i></code>		read arg 2
<i><end></i>	<code>LOADIN 2 0</code>	$ACC := D(SP)$	current sum increased by arg 2
	<code>ADD 1</code>	$ACC := ACC + D(1)$	cache result
	<code>STORE 1</code>	$D(1) := ACC$	
	<code>LOAD 0</code>		
	<code>SUBI 1</code>		decrease counter by 1
	<code>STORE 0</code>	$D(0) := D(0) - 1$	repeat addition
<i><end></i>	<code>JUMP <i>loop</i></code>		load result
	<code>LOAD 1</code>		push it on stack
	<code>STOREIN 2 0</code>		
	<code>inc IN1</code>		back to jump table
	<code>JUMP <i><jt></i></code>		

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
$\langle jp \rangle$	MOVE IN1 ACC STORE 0 LOADIN 1 1 ADD 0 MOVE ACC IN1 JUMP $\langle jt \rangle$	ACC: = VPC $D(0)$: = ACC ACC: = $D(VPC + 1)$ ACC: = ACC + $D(0)$ IN1: = ACC	cache VPC load i compute new VPC value update VPC jump back
$\langle cjp \rangle$	dec IN2 LOADIN 2 1 JUMP= $\langle jp \rangle$ MOVE IN1 ACC ADDI 2 MOVE ACC IN1 JUMP $\langle jt \rangle$	SP: = SP - 1 ACC: = $D(SP + 1)$ VPC: = VPC + 2	update for pop pop value to ACC perform jump if ACC = 0 otherwise, go on point to next jump back



©: Michael Kohlhase

81



Imperative Stack Operations: peek

label	instruction	effect	comment
$\langle peek \rangle$	MOVE IN1 ACC STORE 0 LOADIN 1 1 MOVE ACC IN1 inc IN2 LOADIN 1 8 STOREIN 2 0 LOAD 0 ADDI 2 MOVE ACC IN1 JUMP $\langle jt \rangle$	ACC: = IN1 $D(0)$: = ACC ACC: = $D(VPC + 1)$ IN1: = ACC ACC: = $D(IN1 + 8)$ ACC: = $D(0)$	cache VPC load i prepare push load $S(i)$ push $S(i)$ load old VPC compute new value update VPC jump back



©: Michael Kohlhase

82



Imperative Stack Operations: poke

label	instruction	effect	comment
$\langle poke \rangle$	MOVE IN1 ACC STORE 0 LOADIN 1 1 MOVE ACC IN1 LOADIN 2 0 STOREIN 1 8 dec IN2 LOAD 0 ADD 2 MOVE ACC IN1 JUMP $\langle jt \rangle$	ACC: = IN1 $D(0)$: = ACC ACC: = $D(VPC + 1)$ IN1: = ACC ACC: = $S(i)$ $D(IN1 + 8)$: = ACC IN2: = IN2 - 1 ACC: = $D(0)$ ACC: = ACC + 2	cache VPC load i pop to ACC store in $S(i)$ get old VPC add 2 update VPC jump back



©: Michael Kohlhase

83



2.2.2 A Simple Imperative Language

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.



©: Michael Kohlhase

84



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

Abstract Syntax of SW

```

▷ Definition 100 type id = string      (* identifier *)

datatype exp =
  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 =
  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
  
```



©: Michael Kohlhase

85



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")
  
```



©: Michael Kohlhase

86



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 101** 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 (↪ the empty function)
 - ▷ insertion of a key/value pair $\langle k, v \rangle$ into an environment φ : (↪ $\varphi, [v/k]$)
 - ▷ lookup of the value v for a key k in φ (↪ $\varphi(k)$)
- ▷ 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
      
```


©: Michael Kohlhase
87


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

An SML Data Type for $\mathcal{L}(\text{VM})$ 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
      
```


©: Michael Kohlhase
88


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 $\mathcal{L}(\text{VM})$ instructions. From there, we only need to apply a simple conversion (which we omit) to numbers to obtain $\mathcal{L}(\text{VM})$ byte code.

Compiling SW programs

- ▷ SML function from SW programs (type `program`) to $\mathcal{L}(\text{VM})$ 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
```



©: Michael Kohlhase

89



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]
```



©: Michael Kohlhase

90



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 $\mathcal{L}(\text{VM})$ 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
```



©:Michael Kohlhase

91



As we anticipated above, the `compileD` function is more complex than the other two. It gives $\mathcal{L}(\text{VM})$ 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
```



©:Michael Kohlhase

92



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 $\mathcal{L}(\text{VM})$ we discussed above, we can run SW programs on the register machine REMA.

If we now use the REMA simulator from exercise⁴, then we can run SW programs on our computers outright. EdNote(4)

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 $\mathcal{L}(\text{VM})$ with new operations next.

Note that the compiler we have seen above produces $\mathcal{L}(\text{VM})$ 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.

⁴EDNOTE: include the exercises into the course materials and reference the right one here

Compiling the Extended Example: A `while` Loop

- ▷ **Example 102** Consider the following program that computes $12!$ and the corresponding $\mathcal{L}(\text{VM})$ program:

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

- ▷ 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 poking the stack position (**must remember that**)



©: Michael Kohlhase

93



Definition 103 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 **static procedures** in this case.

Example 104 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 *)

```



©: Michael Kohlhase

94



Translation of a Static Procedure

▷ **Example 105** [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, (* in *)
return, (* exp(10,2) *)
con 2, con 10, call 0, (* end *)
halt]

proc al 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 al 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)

Example 106 ▷ `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.)

▷ `[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

▷ `[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 $\mathcal{L}(\text{VM})$ program)

▷ then it jumps to the first body instruction.

▷ `[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

▷ `[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.

▷ `[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,`

32	0
10	-1
2	-2

What have we seen?

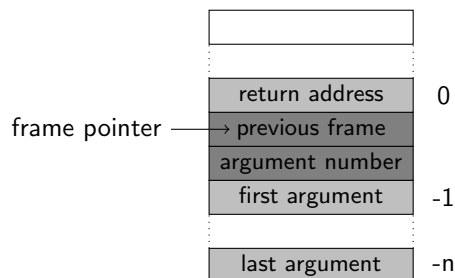
- ▷ The four new VM commands allow us to model static procedures.
 - `proc a l` contains information about the number a of arguments and the length l of the procedure
 - `arg i` pushes the i^{th} argument from the current frame to the stack.
(Note that arguments are stored in reverse order on the stack)
 - `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 another register: the **frame pointer** (FP), 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
<code><proc></code>	<code>MOVE IN1 ACC</code>	$ACC: = VPC$	
	<code>STORE 0</code>	$D(0): = ACC$	cache VPC
	<code>LOADIN 1 2</code>	$ACC: = D(VPC + 2)$	load length
	<code>ADD 0</code>	$ACC: = ACC + D(0)$	compute new VPC value
	<code>MOVE ACC IN1</code>	$IN1: = ACC$	update VPC
	<code>JUMP <jt></code>		jump back



©: Michael Kohlhase

99



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
<code><arg></code>	<code>LOADIN 1 1</code>	$ACC: = D(VPC + 1)$	load i
	<code>STORE 0</code>	$D(0): = ACC$	cache i
	<code>MOVE IN3 ACC</code>		
	<code>STORE 1</code>	$D(1): = FP$	cache FP
	<code>SUBI 1</code>		
	<code>SUB 0</code>	$ACC: = FP - 1 - i$	load argument position
	<code>MOVE ACC IN3</code>	$FP: = ACC$	move it to FP
	<code>inc IN2</code>	$SP: = SP + 1$	prepare push
	<code>LOADIN 3 0</code>	$ACC: = D(FP)$	load arg i
	<code>STOREIN 2 0</code>	$D(SP): = ACC$	push arg i
	<code>LOAD 1</code>	$ACC: = D(1)$	load FP
	<code>MOVE ACC IN3</code>	$FP: = ACC$	recover FP
	<code>MOVE IN1 ACC</code>		
	<code>ADDI 2</code>		
	<code>MOVE ACC IN1</code>	$VPC: = VPC + 2$	next instruction
	<code>JUMP <jt></code>		jump back



©: Michael Kohlhase

100



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



©: Michael Kohlhase

101



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)	LOADIN 2 0	ACC: = $D(SP)$	load top value
	STORE 0	$D(0)$: = ACC	cache it
	LOADIN 2 - 1	ACC: = $D(SP - 1)$	load return address
	MOVE ACC IN1	IN1: = ACC	set VPC to it
	LOADIN 3 - 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
	LOADIN 3 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
	STOREIN 2 0	$D(IN2)$: = ACC	pop return value
	JUMP (jt)		jump back

©: Michael Kohlhase 102

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.

2.2.3 Compiling Basic Functional Programs

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.

©: Michael Kohlhase 103

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

```



©: Michael Kohlhase

104



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 (Leq(Id"n", Con 0),
      Con 1,
      Mul(Id"x", App("exp", [Id"x", Sub(Id"n", Con 1)])))
  ])
in
  exp(2,10)
end
  App("exp", [Con 2, Con 10])

```



©: Michael Kohlhase

105



The next step is to build a compiler for μML into programs in the extended $\mathcal{L}(VM)$. 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

```



©: Michael Kohlhase

106



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)
```



©: Michael Kohlhase

107



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
```



©: Michael Kohlhase

108



Compiling μML

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



©: Michael Kohlhase

109



Where To Go Now?

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



©: Michael Kohlhase

110



2.3 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



©: Michael Kohlhase

111



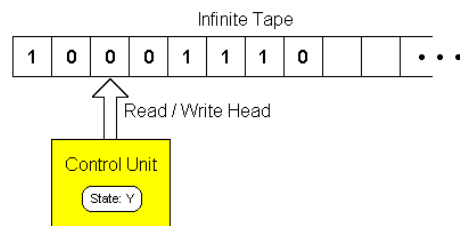
The idea we are going to pursue here is a very fundamental one for Computer Science: The Turing Machine. The main idea here is that we want to explore what the “simplest” (whatever that may mean) computing machine could be. The answer is quite surprising, we do not need wires, electricity, silicon, etc; 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 communication).

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."



More Precisely: Turing machine

- ▷ **Definition 107** 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.



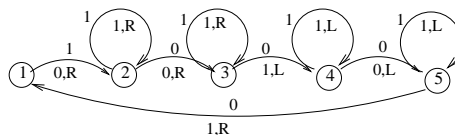
Turing Machine

Example 108 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:



©: Michael Kohlhase

114



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	1 1	9	s_2	10 0 1
2	s_2	0 1	10	s_3	100 1
3	s_2	01 0	11	s_3	1001 0
4	s_3	010 0	12	s_4	100 1 1
5	s_4	01 0 1	13	s_4	10 0 11
6	s_5	0 1 01	14	s_5	1 0 011
7	s_5	0 101	15	s_1	11 0 11
8	s_1	1 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 s_1 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



©: Michael Kohlhase

115



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 109** ([Church/Turing]) (unprovable, but accepted)

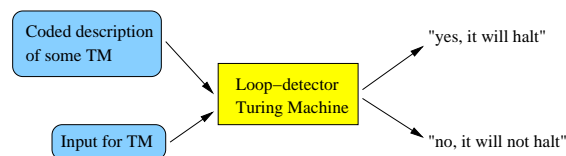
Anything that can be computed at all, can be computed by a Turing Machine



Is there anything that cannot be computed by a TM

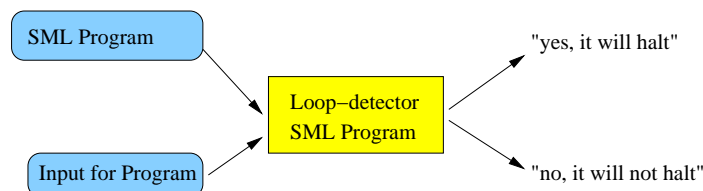
- ▷ **Theorem 110** *No Turing machine can infallibly tell if another Turing machine will get stuck in an infinite loop on some given input.*

▷



▷ **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
<pre>fun halter (n) = 1; halter : int -> int fun looper (n) = looper(n+1); looper : int -> int</pre>	<pre>will_halt (halter ,1); val true : bool will_halt (looper ,1); val false : bool</pre>

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?



©: Michael Kohlhase

118



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



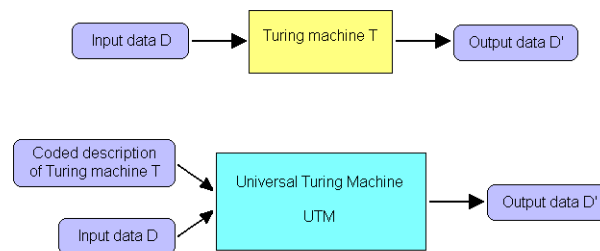
©: Michael Kohlhase

119



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 111** such a Turing machine is indeed possible (e.g. with 2 states, 18 symbols)
- ▷ **Definition 112** 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.



3 Problem Solving and Search

3.1 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 offline problem 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 113 Offline problem solving:** Acting only with complete knowledge of problem and solution
- ▷ **Definition 114 Online problem solving:** Acting without complete knowledge
- ▷ **Here:** we are concerned with **offline** problem solving only.



©: Michael Kohlhase

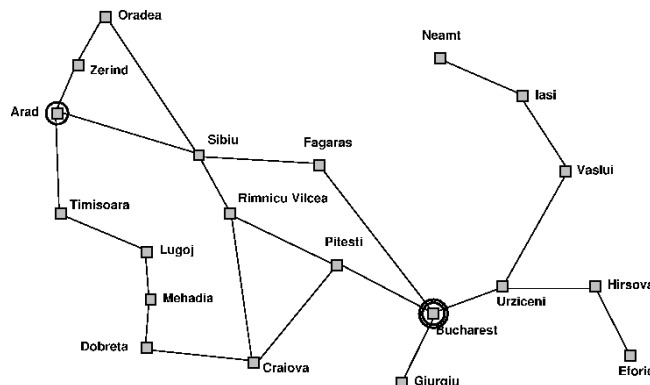
121



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



©: Michael Kohlhase

122



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 115** 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 116** A **solution** for a problem \mathcal{P} consists of a sequence of actions that bring us from \mathcal{I} to a goal state.



©: Michael Kohlhase

123



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)



©: Michael Kohlhase

124



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 $\{7, 8\}$. 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 $[right\{2, 4, 6, 8\}, suck\{4, 8\}, left\{3, 7\}, suck\{7\}]$.

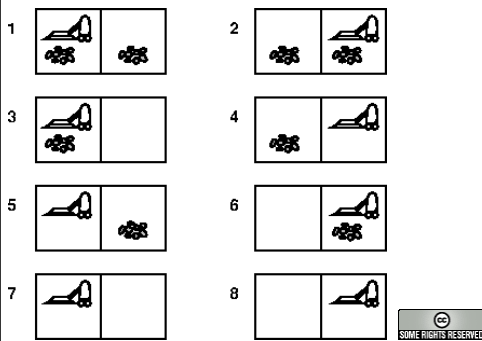
Example: vacuum-cleaner world

Single-state Problem:

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

Multiple-state Problem:

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



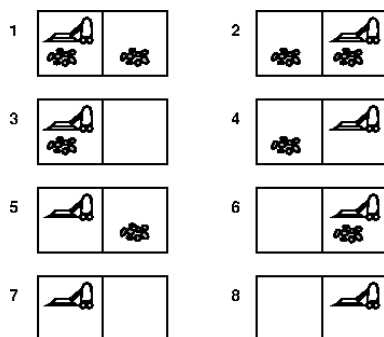
©: Michael Kohlhase

125

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:** $[suck, right, suck]$
 - $suck \rightarrow \{5,7\}$
 - $right \rightarrow \{6,8\}$
 - $suck \rightarrow \{6,8\}$



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



©: Michael Kohlhase



126



In the contingency version of \mathcal{P} a solution is the following: $[suck\{5,7\}, right \rightarrow \{6,8\}, suck \rightarrow \{6,8\}], [suck\{5,7\}]$, 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. $S(Arad) = \{\langle goZer, Zerind \rangle, \langle goSib, Sibiu \rangle, \dots\}$)
 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


©: Michael Kohlhase
127


“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

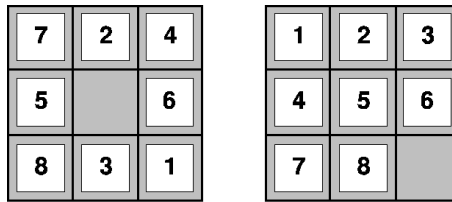
- ▷ **Abstraction:** Real world is absurdly complex
State space must be abstracted for problem solving
- ▷ **(Abstract) state:** Set of real states
- ▷ **(Abstract) operator:** Complex combination of real actions
- ▷ **Example:** $Arad \rightarrow Zerind$ represents complex set of possible routes
- ▷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


©: Michael Kohlhase
128


“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	<i>left, right, up, down</i>
Goal test	= goal state?
Path cost	1 per move



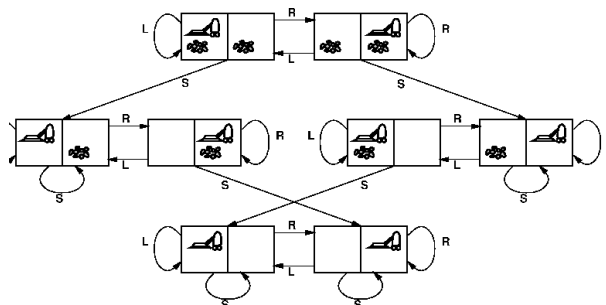
©: Michael Kohlhase

129



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	<i>left, right, suck, noOp</i>
Goal test	<i>notdirty?</i>
Path cost	1 per operation (0 for <i>noOp</i>)

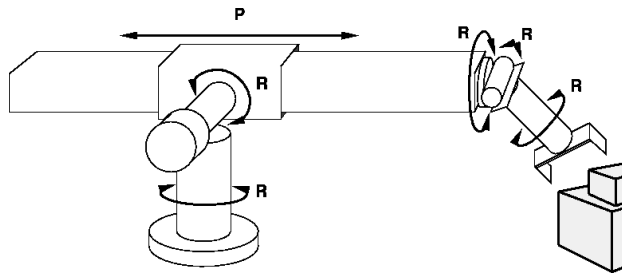


©: Michael Kohlhase

130



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



©: Michael Kohlhase

131



3.2 Search

Tree search algorithms

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

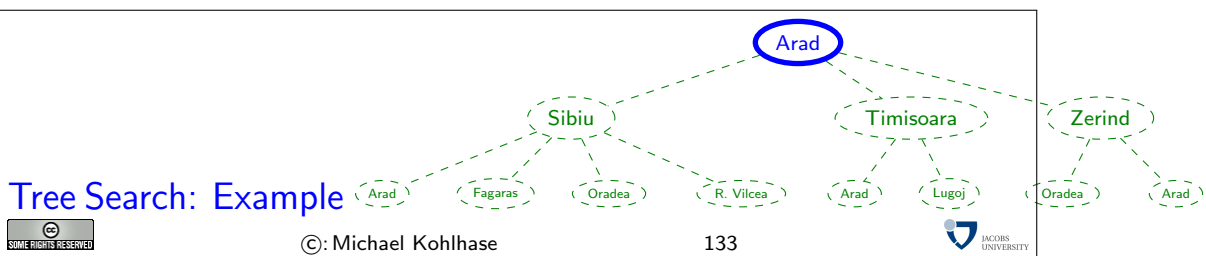
```

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 > <return failure > end if
    <choose a leaf node for expansion according to strategy >
    if <the node contains a goal state > return <the corresponding solution >
    else <expand the node and add the resulting nodes to the search tree >
    end if
  end loop
end procedure
  
```



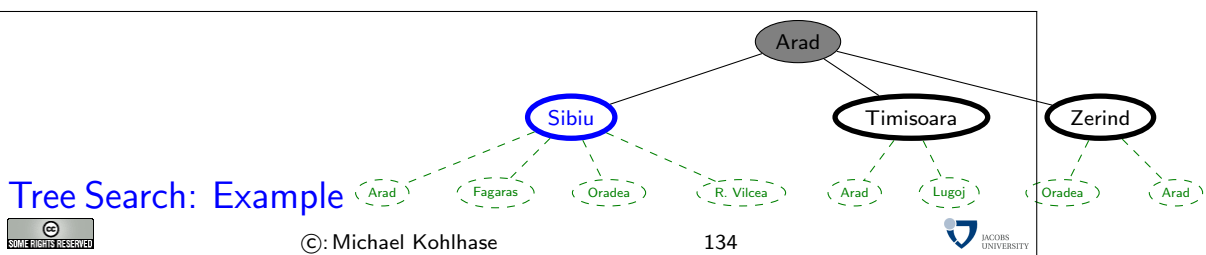
©: Michael Kohlhase

132



©: Michael Kohlhase

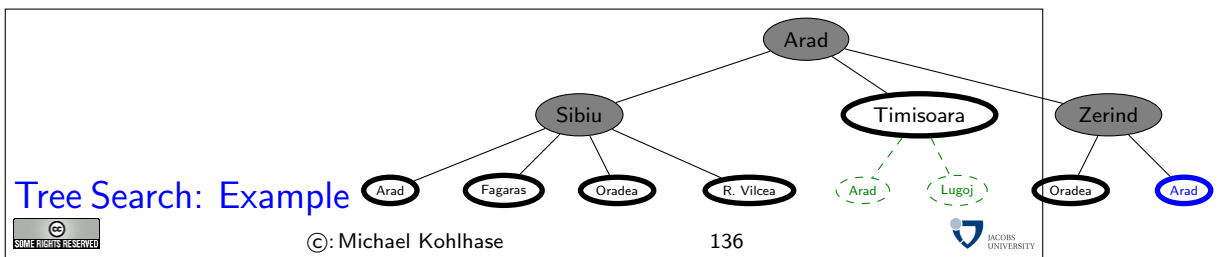
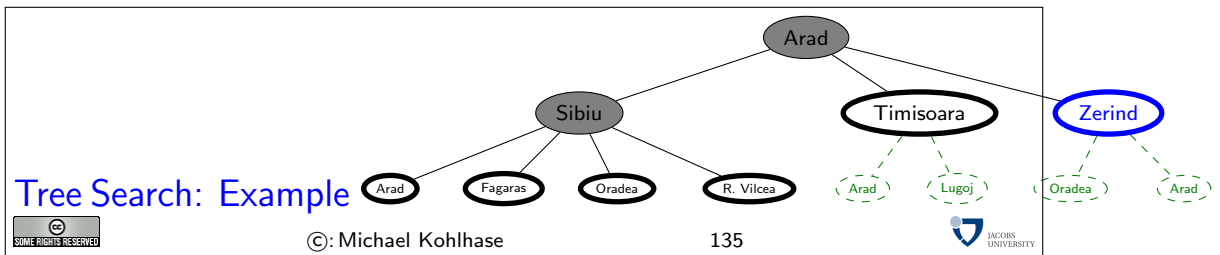
133



©: Michael Kohlhase

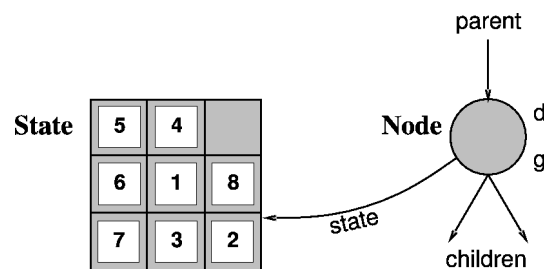
134





Implementation: States vs. nodes

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



©: Michael Kohlhase 137

Implementation of search algorithms

```

procedure Tree_Search (problem, strategy)
  fringe := insert (make_node( initial_state (problem)))
  loop
    if fringe <is empty> fail end if
    node := first (fringe, strategy)
    if NodeTest(State(node)) return State(node)
    else fringe := insert_all (expand(node, problem), strategy)
    end if
  end loop
end procedure

```

- ▷ **Definition 117** The **fringe** is a list nodes not yet considered. It is ordered by the **search strategy** (see below)

©: Michael Kohlhase 138

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 ∞)



©: Michael Kohlhase

139



Complexity means here always *worst-case* complexity.

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

3.3 Uninformed Search Strategies

Uninformed search strategies

▷ **Definition 118 (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



©: Michael Kohlhase

140

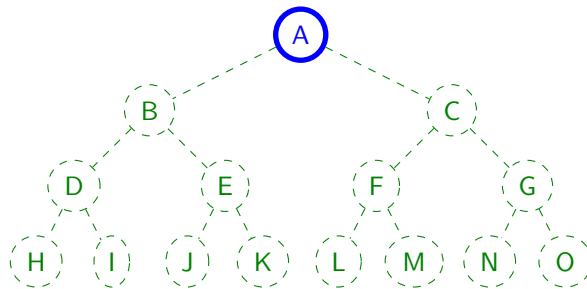


The opposite of uninformed search is informed or *heuristic* search. In the example, one could add, for instance, to prefer cities that lie in the general direction of the goal (here SE).

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



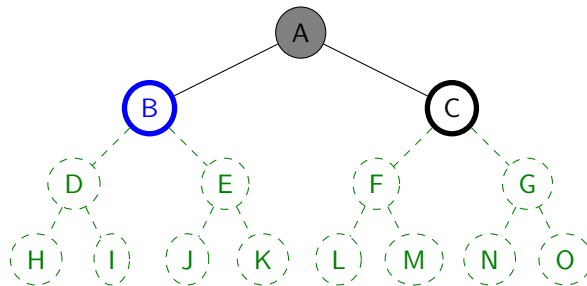
©: Michael Kohlhase

141



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



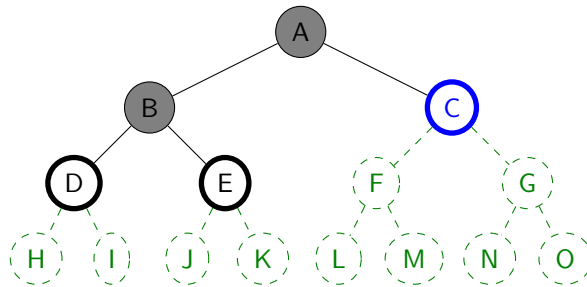
©: Michael Kohlhase

142



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



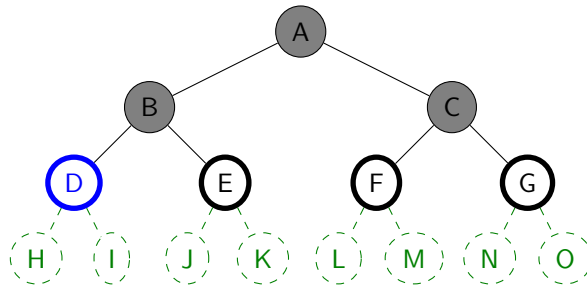
©: Michael Kohlhase

143



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



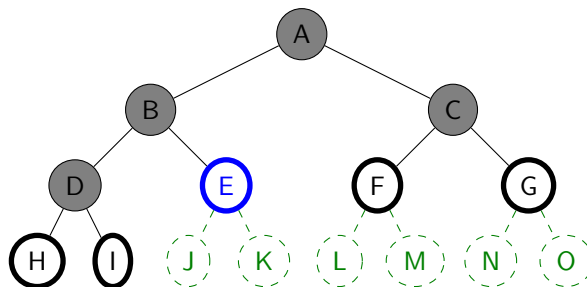
©: Michael Kohlhase

144



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



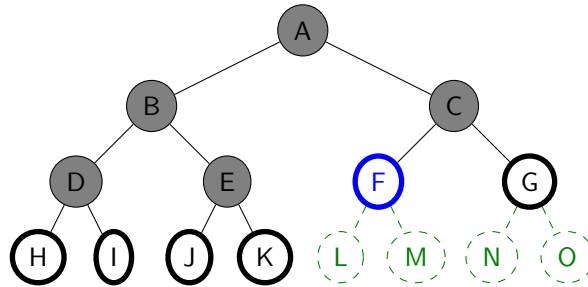
©: Michael Kohlhase

145



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



©: Michael Kohlhase

146



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

Breadth-First Search: Romania

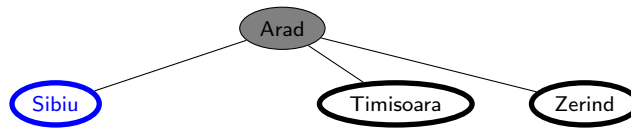


©: Michael Kohlhase

147



Breadth-First Search: Romania

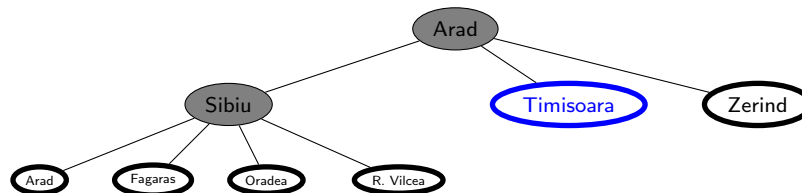


©: Michael Kohlhase

148



Breadth-First Search: Romania

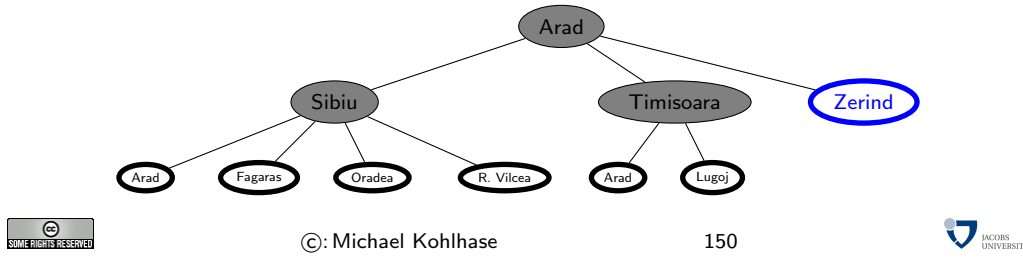


©: Michael Kohlhase

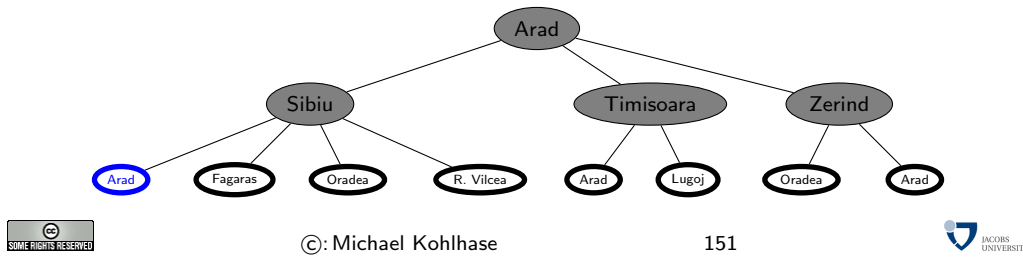
149



Breadth-First Search: Romania



Breadth-First Search: Romania



Breadth-first search: Properties

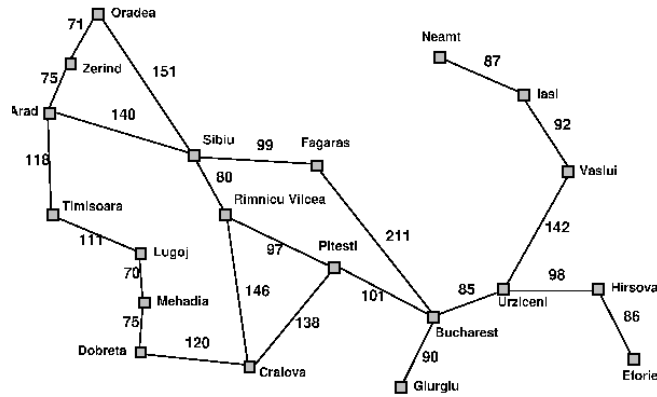
Complete	Yes (if b is finite)
Time	$1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) \in O(b^{d+1})$ i.e. exponential in d
Space	$O(b^{d+1})$ (keeps every node in memory)
Optimal	Yes (if cost = 1 per step), not optimal in general

- ▷ **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	231
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



©: Michael Kohlhase

153



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)

Arad



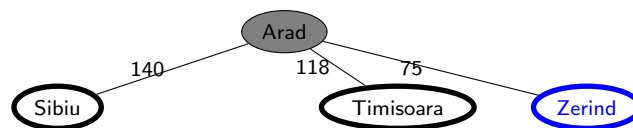
©: Michael Kohlhase

154



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)



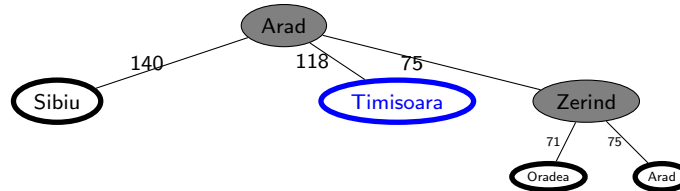
©: Michael Kohlhase

155



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)



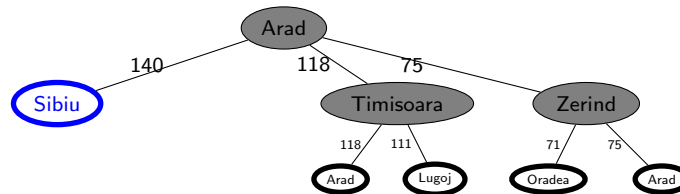
©: Michael Kohlhase

156



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)



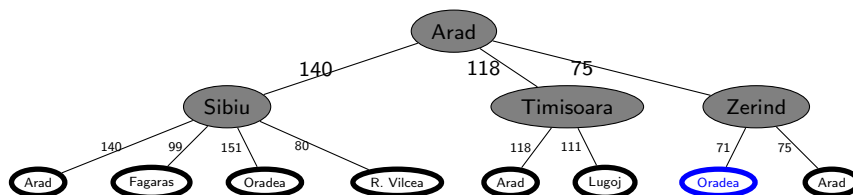
©: Michael Kohlhase

157



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)



©: Michael Kohlhase

158



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



©: Michael Kohlhase

159



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 trouble⁵

EdNote(5)

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)

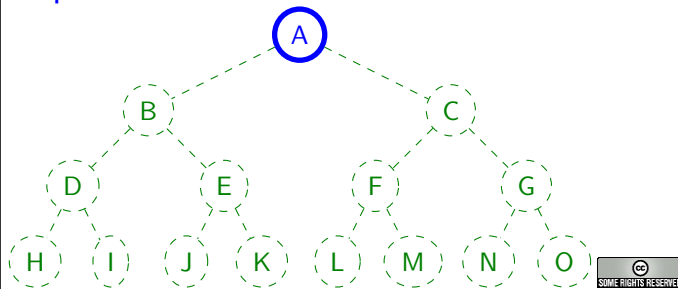


©: Michael Kohlhase

160



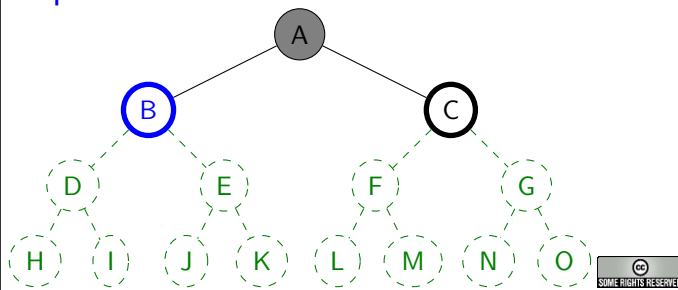
Depth-First Search



©: Michael Kohlhase

161

Depth-First Search

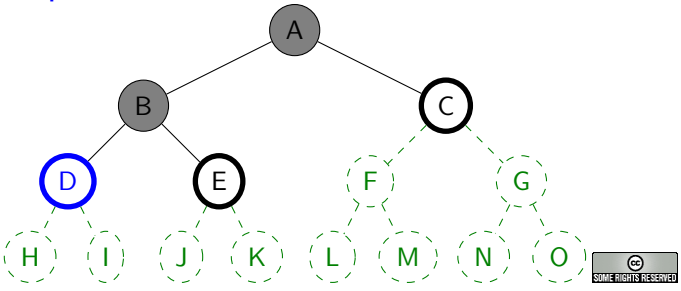


©: Michael Kohlhase

162

⁵EDNOTE: say how

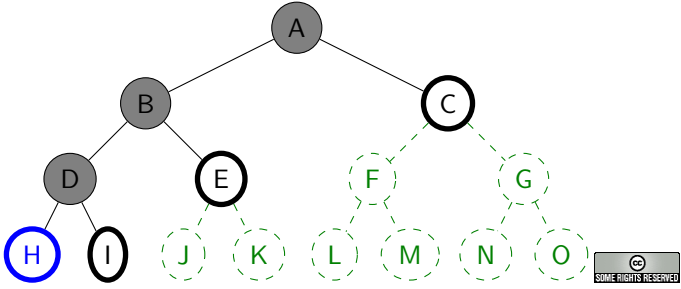
Depth-First Search



©: Michael Kohlhase

163

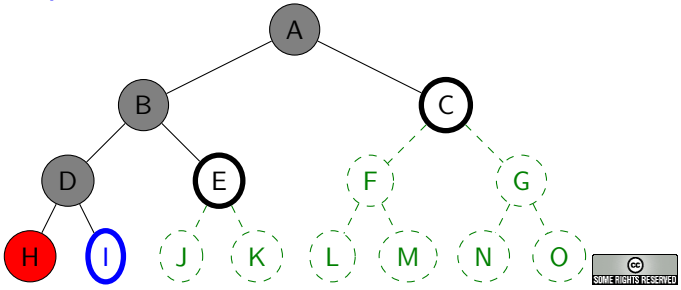
Depth-First Search



©: Michael Kohlhase

164

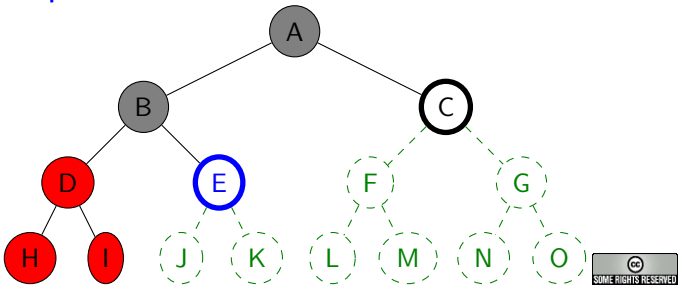
Depth-First Search



©: Michael Kohlhase

165

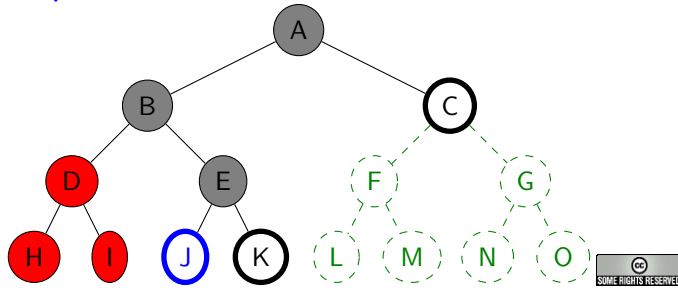
Depth-First Search



©: Michael Kohlhase

166

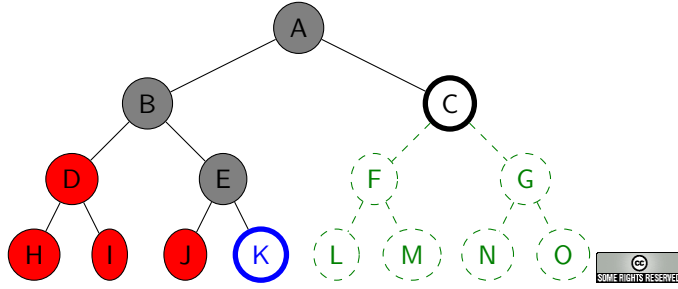
Depth-First Search



©: Michael Kohlhase

167

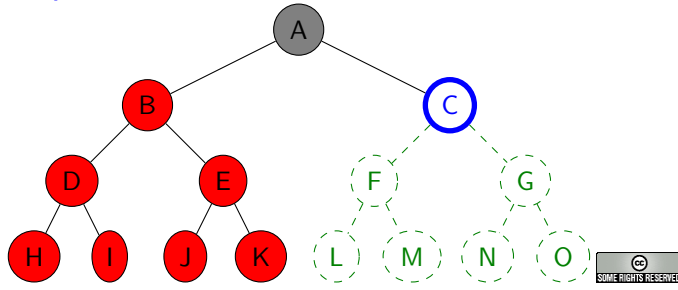
Depth-First Search



©: Michael Kohlhase

168

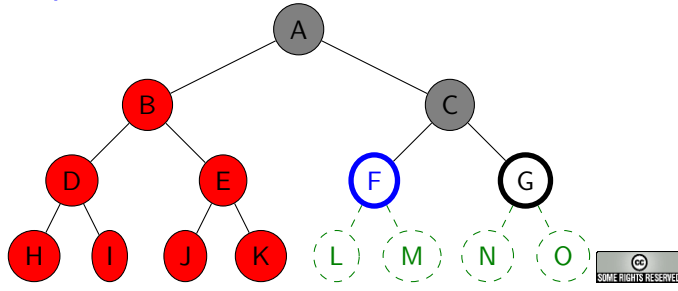
Depth-First Search



©: Michael Kohlhase

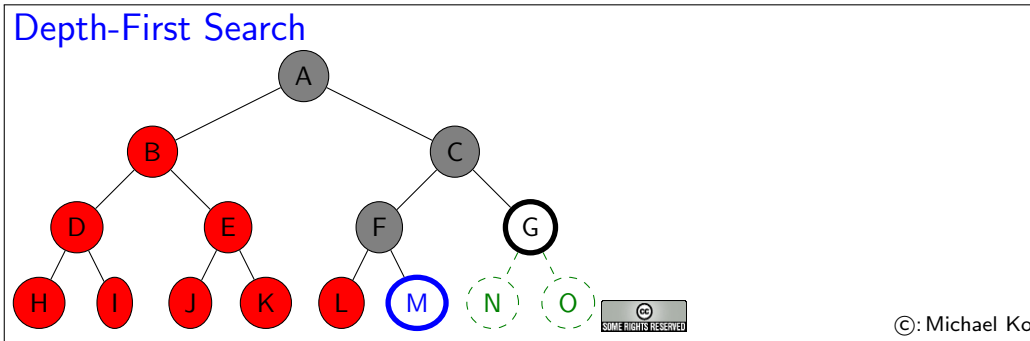
169

Depth-First Search

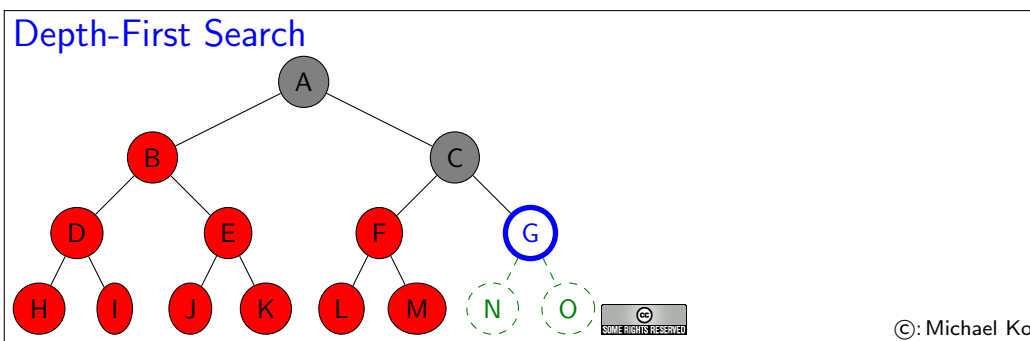


©: Michael Kohlhase

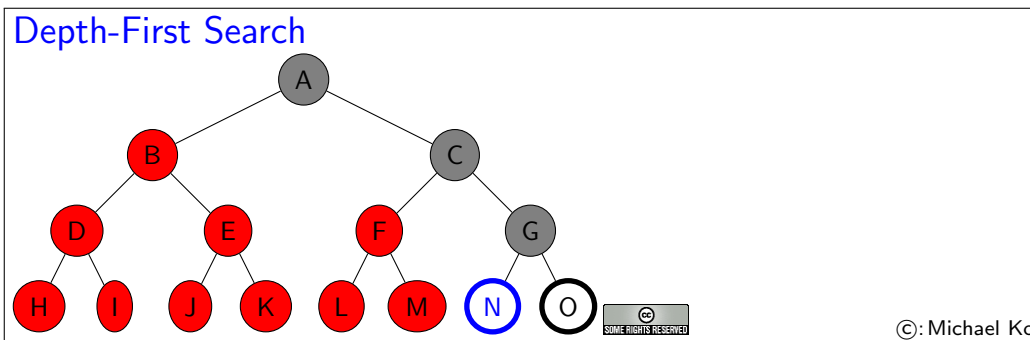
170



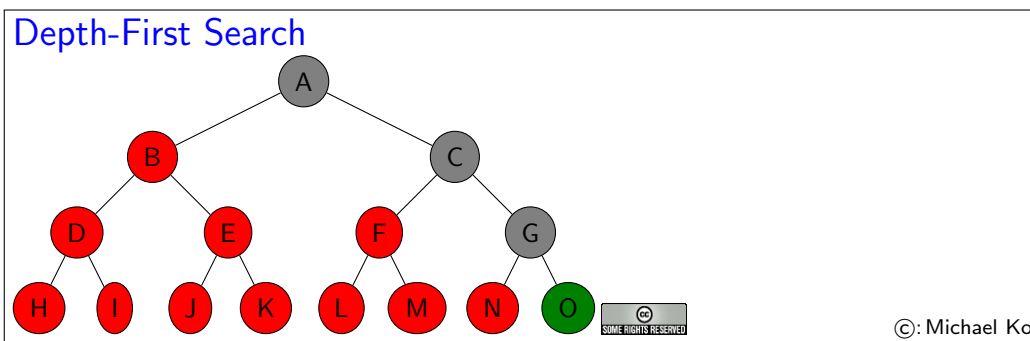
171



172



173



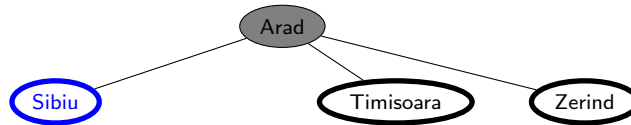
174



175



Depth-First Search: Romania

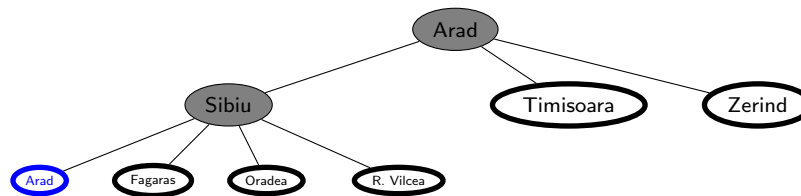


©: Michael Kohlhase

176



Depth-First Search: Romania

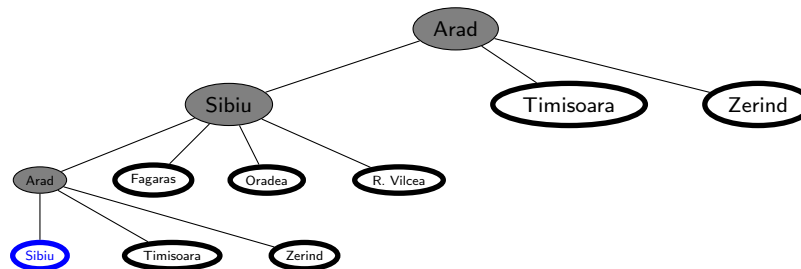


©: Michael Kohlhase

177



Depth-First Search: Romania



©: Michael Kohlhase

178



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.



©: Michael Kohlhase

179



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 ∞
    result := Depth_Limited_search(problem,depth)
    if depth ≠ cutoff return result end if
  end for
end procedure
  
```



©: Michael Kohlhase

180



Iterative Deepening Search at Limit Depth 0

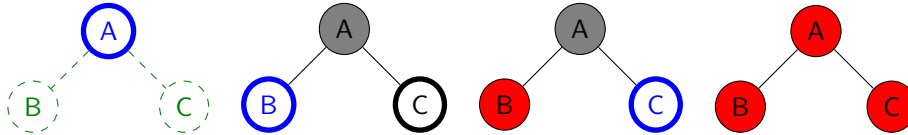


©: Michael Kohlhase

181



Iterative Deepening Search at Limit Depth 1

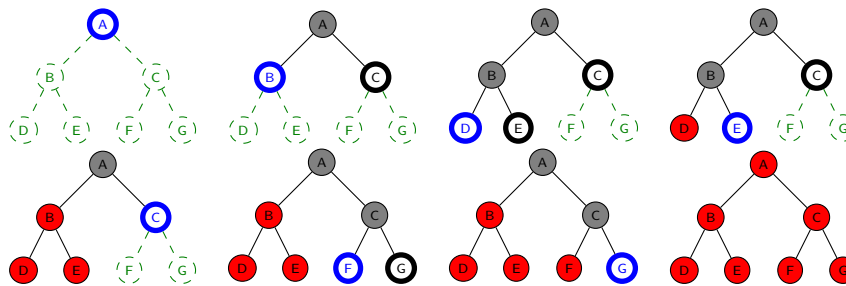


©: Michael Kohlhase

182



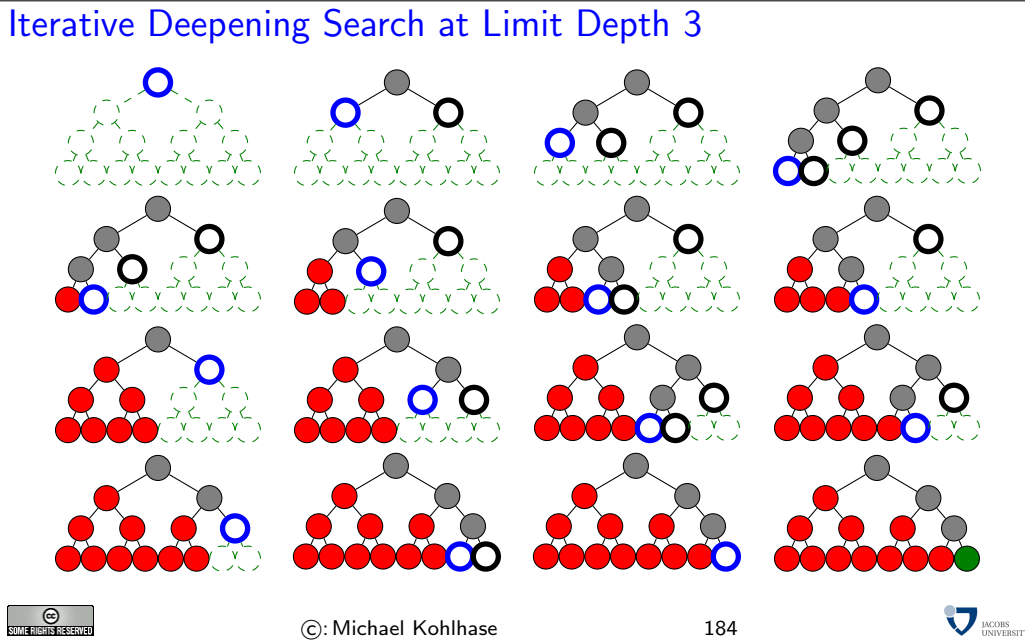
Iterative Deepening Search at Limit Depth 2



©: Michael Kohlhase

183





Iterative deepening search: Properties

Complete	Yes
Time	$(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d \in O(b^{d+1})$
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.

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

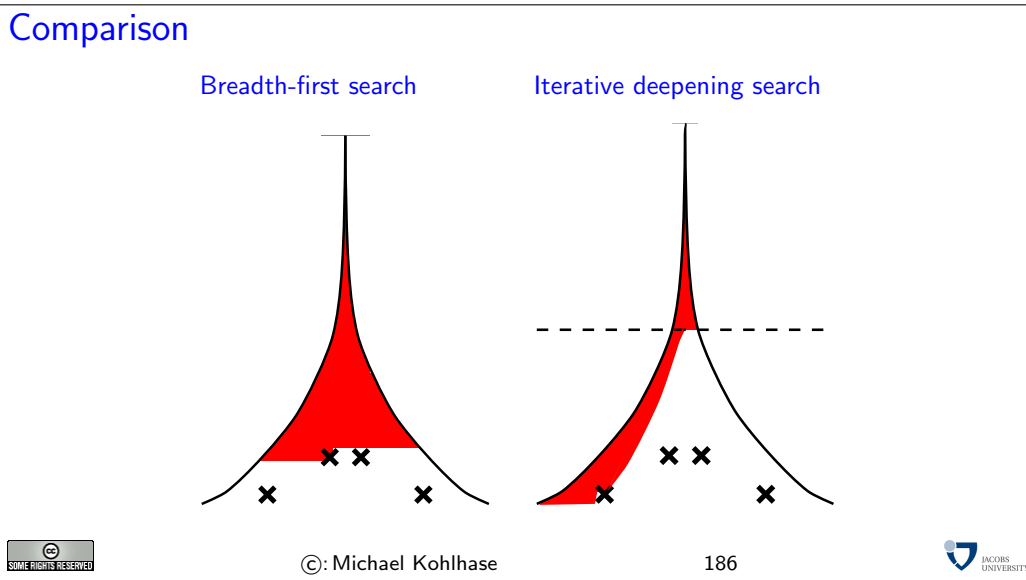
▷ Comparison:

©: Michael Kohlhase 185

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.



3.4 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

©: Michael Kohlhase 187

Best-first search

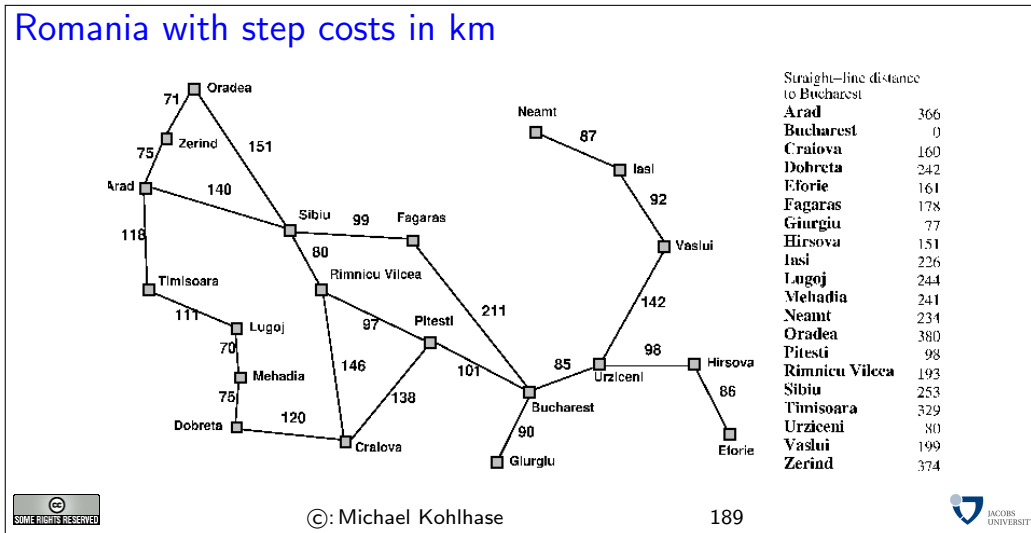
- ▷ **Idea:** Use an **evaluation function** for each node (estimate of "desirability") Expand most desirable unexpanded node
- ▷ **Implementation:** *fringe* is a queue sorted in decreasing order of desirability
- ▷ **Special cases:** Greedy search, A* search

©: Michael Kohlhase 188

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.



Greedy search

▷ **Definition 119** 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 120** $h_{SLD}(n) =$ 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 → external search control
partial solution cost → goal cost estimation

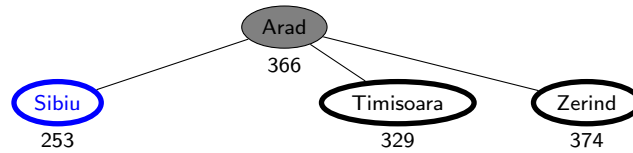
In greedy search we replace the *objective* cost to *construct* the current solution with a heuristic or *subjective* measure from which we think it gives a good idea how far we are from a *solution*. Two things have shifted:

- 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

Arad
366

Greedy Search: Romania

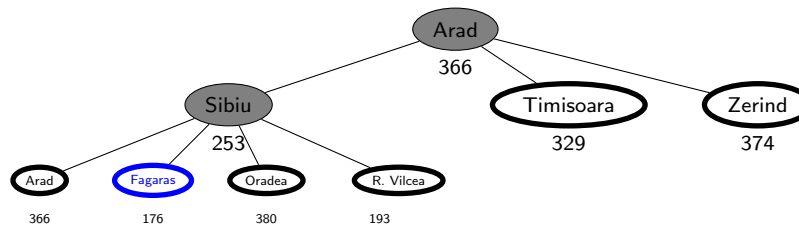


©: Michael Kohlhase

192



Greedy Search: Romania

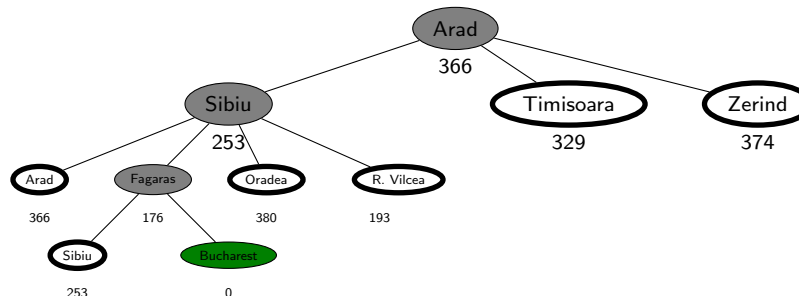


©: Michael Kohlhase

193



Greedy Search: Romania



©: Michael Kohlhase

194



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 121** 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



©: Michael Kohlhase

195



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 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 optional, 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 122** 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 123** Best-First-Search with evaluation function $g + h$ is called **astarSearch search**.



©: Michael Kohlhase

196



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 124 (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 125** Straight-line distance never overestimates the actual road distance (triangle inequality)

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



©: Michael Kohlhase

197

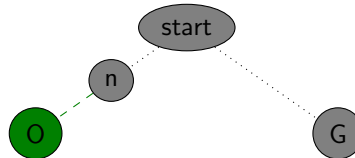


A* Search: Admissibility

▷ **Theorem 126** *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

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

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



©: Michael Kohlhase

198



A* Search Example

Arad
366=0+366

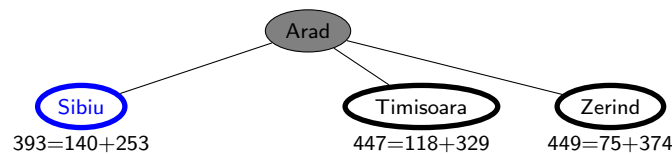


©: Michael Kohlhase

199



A* Search Example

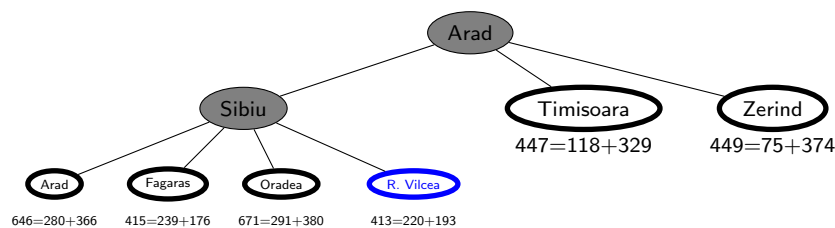


©: Michael Kohlhase

200



A* Search Example

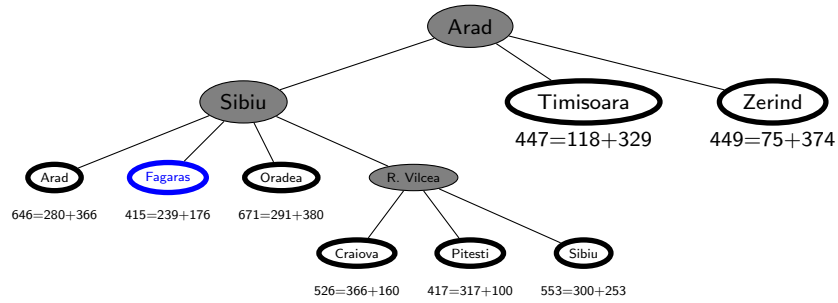


©: Michael Kohlhase

201



A* Search Example

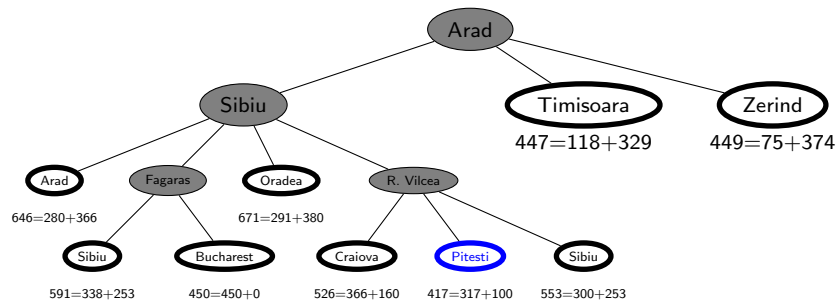


©: Michael Kohlhase

202



A* Search Example

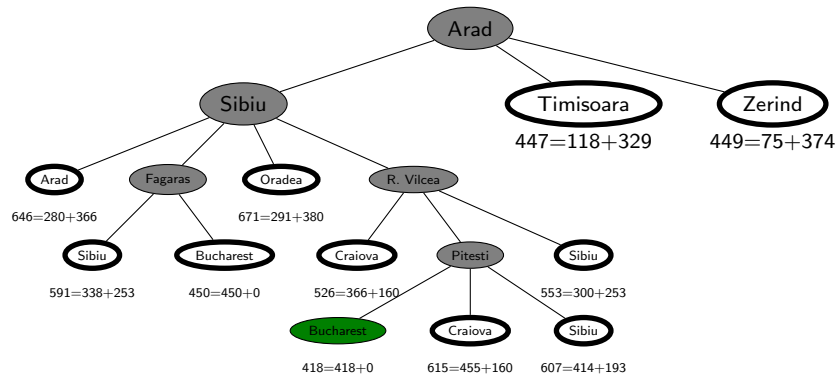


©: Michael Kohlhase

203



A* Search Example



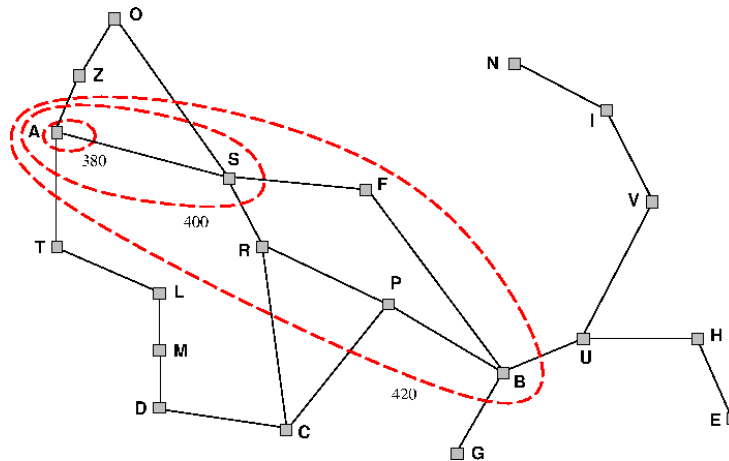
©: Michael Kohlhase

204



A* search: f -contours

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



©: Michael Kohlhase

205



A* search: Properties

Complete	Yes (unless there are infinitely many nodes n with $f(n) \leq f(0)$)
Time	Exponential in [relative error in $h \times$ 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 .



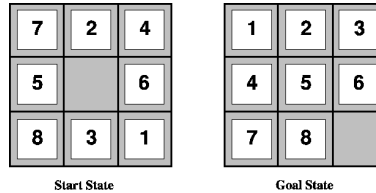
©: Michael Kohlhase

206



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 127** Let $h_1(n)$ be the number of misplaced tiles in node n ($h_1(S) = 6$)
- ▷ **Example 128** Let $h_2(n)$ be the total manhattan distance from desired location of each tile. ($h_2(S) = 2 + 0 + 3 + 1 + 0 + 1 + 3 + 4 = 14$)
- ▷ **Observation 129** (Typical search costs) ($IDS \hat{=} \textit{iterative deepening search}$)

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



©: Michael Kohlhase

207



Dominance

- ▷ **Definition 130** Let h_1 and h_2 be two admissible heuristics we say that h_2 **dominates** h_1 if $h_2(n) \geq h_1(n)$ or all n .
- ▷ **Theorem 131** If h_2 dominates h_1 , then h_2 is better for search than h_1 .



©: Michael Kohlhase

208



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 132** If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then we get heuristic h_1 .
- ▷ **Example 133** 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



©: Michael Kohlhase

209

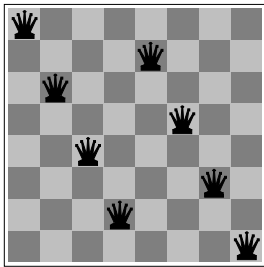


Relaxation means to remove some of the constraints or requirements of the original problem, so that a solution becomes easy to find. Then the cost of this easy solution can be used as an optimistic approximation of the problem.

3.5 Local Search

Local Search Problems

- ▷ **Idea:** Sometimes the path to the solution is irrelevant
- ▷ **Example 134 (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 135** 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,...



©: Michael Kohlhase

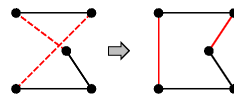
210



Local Search: Iterative improvement algorithms

- ▷ **Definition 136 (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



- ▷ **Definition 137 (n -queens problem)** Put n queens on $n \times n$ board such that no two queens in the same row, column, or diagonal.

- ▷ **Idea:** Move a queen to reduce number of conflicts



©: Michael Kohlhase

211



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]
      return [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.



©: Michael Kohlhase

212



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.

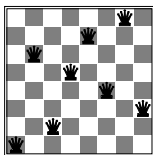
Of course, this will only get us into local maxima, and has no guarantee of getting us into global ones (remember, we are blind). The solution to this problem is to re-start the search at random (we do not have any information) places, and hope that one of the random jumps will get us to a slope that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

Example Hill-Climbing with 8 Queens

- ▷ **Idea:** Heuristic function h is number of queens that threaten each other.
- ▷ **Example 138** An 8-queens state with heuristic cost estimate $h = 17$ showing h -values for moving a queen within its column

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	👤	13	16	13	16
👤	14	17	15	👤	14	16	16
17	👤	16	18	15	👤	15	👤
18	14	👤	15	15	14	👤	16
14	14	13	17	12	14	12	18

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



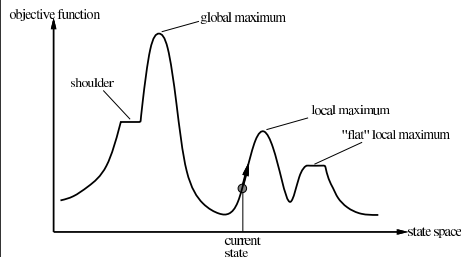
©: Michael Kohlhase

213



Hill-climbing

- ▷ **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”



- ▷ **Idea:** Escape local maxima by allowing some “bad” or random moves.
- ▷ **Example 139** local search, simulated annealing. . .
- ▷ **Properties:** All are incomplete, non-optimal.
- ▷ Sometimes performs well in practice (if (optimal) solutions are dense)



©: Michael Kohlhase

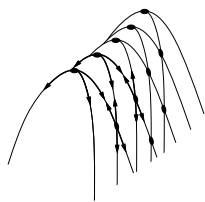
214



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

Simulated annealing (Idea)

- ▷ **Definition 140** **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 modelling
- ▷ Widely used in VLSI layout, airline scheduling, etc.



©: Michael Kohlhase

215



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 ∞
    T := schedule[t]
    if T = 0 return current end if
    next := <a randomly selected successor of current>
    Δ(E) := Value[next]-Value[current]
    if Δ(E) > 0 current := next
    else
      current := next <only with probability>  $e^{\Delta(E)/T}$ 
    end if
  end for
end procedure

```

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



©: Michael Kohlhase

216



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 \implies 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?



©: Michael Kohlhase

217



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!)



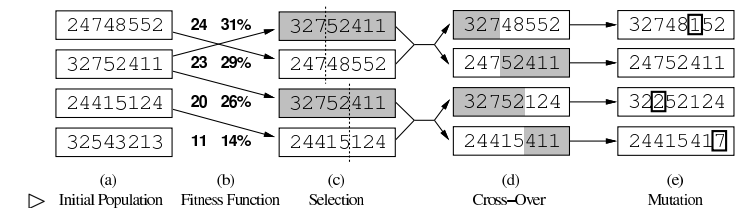
©: Michael Kohlhase

218



Genetic algorithms (very 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)



©: Michael Kohlhase

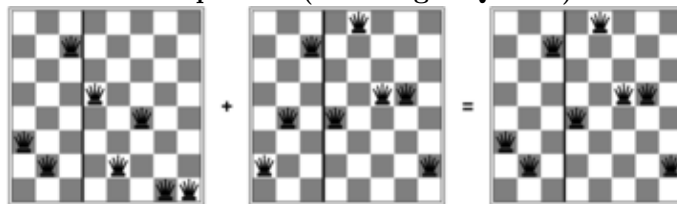
219



Genetic algorithms (continued)

- ▷ Problem: Genetic Algorithms require states encoded as strings (GPs use programs)
- ▷ Crossover helps iff substrings are meaningful components

▷ Example 141 (Evolving 8 Queens)



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



©: Michael Kohlhase

220



4 Logic Programming

4.1 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 computing 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 142

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



©: Michael Kohlhase

221



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 theoretical 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. ⁶

EdNote(6)

⁶EDNOTE: reference

Representing a Knowledge base in ProLog

- ▷ A knowledge base is represented (symbolically) by a set of facts and rules.
- ▷ **Definition 143** 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 144** We can state that Mia is a woman as `woman(mia)`.
- ▷ **Definition 145** A **rule** states information that is *conditionally* true in the domain.
- ▷ **Example 146** Write "something is a car if it has a motor and four wheels" as `car(X) :- has_motor(X), has_wheels(X,4)` (variables are upper-case)
this is just an ASCII notation for $(m(x) \wedge w(x,4)) \Rightarrow car(x)$
- ▷ **Definition 147** 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 $\wedge I$.

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



Knowledge Base (Example)

- ▷ **Example 148** `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).`

$$\frac{\frac{m(c) \quad w(c,4)}{m(c) \wedge w(c,4)} \wedge I \quad \frac{(m(x) \wedge w(x,4)) \Rightarrow car(x)}{(m(c) \wedge w(c,4)) \Rightarrow car(c)} \text{Subst}}{car(c)} \text{MP}$$



Querying the Knowledge base

▷ **Idea:** We want to see whether a fact is in the knowledge base.

▷ **Definition 149** 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 150** The the knowledge base induced by the program

```
nat(zero).
nat(s(X)) :- nat(X).
```

is the set $\{\text{nat}(\text{zero}), \text{nat}(\text{s}(\text{zero})), \text{nat}(\text{s}(\text{s}(\text{zero}))), \dots\}$.

▷ **Idea:** interpret this as a search problem.

▷ state = tuple of goals; goal state = empty list (of goals).

▷ $\text{next}(\langle G, R_1, \dots, R_l \rangle) := \langle \sigma(B_1), \dots, \sigma(B_m), R_1, \dots, R_l \rangle$ (**backchaining**) if there is a rule $H : -B_1, \dots, B_m.$ and a substitution σ with $\sigma(H) = \sigma(G)$.

```
?- nat(s(s(zero))).
?- nat(s(zero)).
?- nat(zero).
Yes
```

▷ If a query contains variables, then ProLog will return an **answer substitution**.

```
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 ¹ to the knowledge base

```
unat(zero).
```

which asserts that the predicate `unat`² is **true** on the term `zero`. Generally, we can add a fact to the

¹for “unary natural numbers”; we cannot use the predicate `nat` and the constructor functions here, since their meaning is predefined in ProLog

²for “unary natural numbers”.

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))).
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`³. 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)



Note: We have seen before⁷ that depth-first search has the problem that it can go into loops. EdNote(7)
 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 language would not be Turing-complete otherwise. The argument can be sketched as follows: we have seen that for Turing machines the halting problem⁸ is undecidable. So if all ProLog programs were terminating, then ProLog would EdNote(8)
 be weaker than Turing machines and thus not Turing complete.

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
  
```

©: Michael Kohlhase 226

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 151 $add(X,Y,Z)$ 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).
  
```

©: Michael Kohlhase 227

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

³Type “control” key together with “h” then press “m” to get an exhaustive mode help.
⁷EdNOTE: reference
⁸EdNOTE: reference

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 `add(X, Y, Z)` means $X + Y = Z$. We start with the same defining equations for addition, rewriting them to relational style.

The first equation is straight-forward 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 `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 `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 152** 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 153** 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 154** 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.

4.2 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 155 (rule)** $H : \neg B_1, \dots, B_n$. corresponds to $H \vee \neg(B_1) \vee \dots \vee \neg(B_n)$ (head the only positive literal)
 - ▷ **Observation 156 (goal setid)** $? - G_1, \dots, G_n$. corresponds to $\neg(G_1), \dots, \neg(G_n)$
 - ▷ **Observation 157 (fact)** F . corresponds to the unit clause F .
- ▷ **Definition 158** A **Horn clause** is a clause with at most one positive literal.

Note: backchaining becomes (hyper)-resolution (special case for rule with facts)

$$\frac{P^T \vee A \quad P^F \vee B}{A \vee B} \quad \frac{H : \neg B_1, \dots, B_n. \quad B_1 \quad \dots \quad B_n}{H}$$

positive unit-resulting hyperresolution (PURR)



©: Michael Kohlhase

229



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 $\hat{=}$ program) cannot be understood/affected by the programmer.
 - ▷ problems with termination



©: Michael Kohlhase

230



PROLOG (Horn clauses)

- ▷ **Definition 159** Each clause contains at most one positive literal
 - ▷ $B_1 \vee \dots \vee B_n \vee \neg(A)$ ($A : \neg B_1, \dots, B_n$)
 - ▷ **Rule clause:** fallible(X) : \neg human(X)
 - ▷ **Fact clause:** human(socrates).
 - ▷ **Program:** set of rule and fact clauses
 - ▷ **Query:** ? \neg fallible(X), greek(X).



©: Michael Kohlhase

231



PROLOG (SLD Resolution)

- ▷ Strategy for Resolution: SLDNF (LUSH)
 - ▷ Selected Literal Definite clauses
 - ▷ Linear resolution
(Always continue work with the focussed clause)
 - ▷ Select the leftmost unsolved positive literal
 - ▷ Always resolve on the positive literal
- ▷ **Theorem 160** (*Strongly*) complete on horn clauses
 - ▷ Each instance of the query that is entailed by the program is subsumed by a positive answer.



©: Michael Kohlhase

232



PROLOG: Our Example

- ▷ **Program:**

```
human(socrates).
human(leibniz).
greek(socrates).
fallible(X) :- human(X).
```
- ▷ **Example 161 (Query)** ? – fallible(X), greek(X).
- ▷ **Answer substitution:** [socrates/X]



©: Michael Kohlhase

233



Why Only Horn Clauses?

- ▷ General clauses of the form $A_1, \dots, A_n :- B_1, \dots, B_n$.
- ▷ e.g. `greek(socrates), greek(perikles)`
 - ▷ **Question:** Are there fallible greeks?
 - ▷ **Indefinite answer:** Yes, Perikles or Sokrates
 - ▷ **Warning:** how about Sokrates and Perikles?
- ▷ e.g. `greek(socrates), roman(socrates):-`
 - ▷ **Query:** Are there fallible greeks?
 - ▷ **Answer:** Yes, Sokrates, if he is not a roman
 - ▷ **Is this abduction?????**



©: Michael Kohlhase

234



Three Principal Modes of Inference

▷ **Deduction**: knowledge extension

$$\frac{\text{rains} \Rightarrow \text{wet_street} \quad \text{rains}}{\text{wet_street}} \quad D$$

▷ **Abduction** explanation

$$\frac{\text{rains} \Rightarrow \text{wet_street} \quad \text{wet_street}}{\text{rains}} \quad A$$

▷ **Induction** learning rules

$$\frac{\text{wet_street} \quad \text{rains}}{\text{rains} \Rightarrow \text{wet_street}} \quad I$$



©: Michael Kohlhase

235



4.2.1 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 slogan⁹

EdNote(9)

$$\text{computation} = \text{logic} + \text{control}$$

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.

⁹EDNOTE: find the reference, and see what he really said

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 162** 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 163** 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. $\leq [\text{free}(\mathbf{A}) \cup \text{free}(\mathbf{B})]$.



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 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 algorithm itself. We will define it in a way that is very similar to logic programming: we first define a calculus that generates “solved forms” (formulae from which we can read off the solution) and reason about control later. In this case we will reason that control does not matter.

Unification (Equational Systems)

▷ **Idea:** Unification is equation solving.

▷ **Definition 164** We call a formula $\mathbf{A}^1=?\mathbf{B}^1 \wedge \dots \wedge \mathbf{A}^n=?\mathbf{B}^n$ an **equational system**.

▷ We consider equational systems as sets of equations (\wedge is ACI), and equations as two-element multisets ($=?$ is C).

▷ **Definition 165** We say that $X^1=?\mathbf{B}^1 \wedge \dots \wedge X^n=?\mathbf{B}^n$ is a **solved form**, iff the X^i are distinct and $X^i \notin \text{free}(\mathbf{B}^j)$.

▷ **Lemma 166** If $\mathcal{E} = X^1=?\mathbf{B}^1 \wedge \dots \wedge X^n=?\mathbf{B}^n$ is a solved form, then \mathcal{E} has the unique most general unifier $\sigma_{\mathcal{E}} := [\mathbf{B}^1/X^1], \dots, [\mathbf{B}^n/X^n]$.

▷ **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 $\theta = \theta \circ \sigma_{\mathcal{E}}[\text{supp}(\sigma)]$. □



©: Michael Kohlhase

237



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 Lemma¹⁰ shows.¹¹

Given the idea that unification problems can be expressed as formulae, we can express the algorithm in three simple rules that transform unification problems into solved forms (or unsolvable ones).

EdNote(10)

EdNote(11)

¹⁰EDNOTE: reference

¹¹EDNOTE: say something about the occurs-in-check,...

Unification Algorithm

▷ **Definition 167** Inference system \mathcal{U}

$$\frac{\mathcal{E} \wedge f(\mathbf{A}^1, \dots, \mathbf{A}^n) = ? f(\mathbf{B}^1, \dots, \mathbf{B}^n)}{\mathcal{E} \wedge \mathbf{A}^1 = ? \mathbf{B}^1 \wedge \dots \wedge \mathbf{A}^n = ? \mathbf{B}^n} \mathcal{U}_{\text{dec}} \quad \frac{\mathcal{E} \wedge \mathbf{A} = ? \mathbf{A}}{\mathcal{E}} \mathcal{U}_{\text{triv}}$$

$$\frac{\mathcal{E} \wedge X = ? \mathbf{A} \quad X \notin \text{free}(\mathbf{A}) \quad X \in \text{free}(\mathcal{E})}{[\mathbf{A}/X](\mathcal{E}) \wedge X = ? \mathbf{A}} \mathcal{U}_{\text{elim}}$$

▷ **Lemma 168** \mathcal{U} is *correct* ($\mathcal{E} \vdash_{\mathcal{U}} \mathcal{F}$ implies $\mathbf{U}(\mathcal{F}) \subseteq \mathbf{U}(\mathcal{E})$)

▷ **Lemma 169** \mathcal{U} is *complete* ($\mathcal{E} \vdash_{\mathcal{U}} \mathcal{F}$ implies $\mathbf{U}(\mathcal{E}) \subseteq \mathbf{U}(\mathcal{F})$)

▷ **Lemma 170** \mathcal{U} is *confluent* (order of derivations does not matter)

▷ **Corollary 171** First-Order Unification is *unitary* (unique most general unifiers)
(\mathcal{U} trivially branching)



©: Michael Kohlhase

238



Unification Examples

Example 172 Two similar unification problems

$\frac{f(g(x, x), h(a)) = ? f(g(a, z), h(z))}{g(x, x) = ? g(a, z) \wedge h(a) = ? h(z)} \mathcal{U}_{\text{dec}}$	$\frac{f(g(x, x), h(a)) = ? f(g(b, z), h(z))}{g(x, x) = ? g(b, z) \wedge h(a) = ? h(z)} \mathcal{U}_{\text{dec}}$
$\frac{x = ? a \wedge x = ? z \wedge h(a) = ? h(z)}{x = ? a \wedge x = ? z \wedge a = ? z} \mathcal{U}_{\text{dec}}$	$\frac{x = ? b \wedge x = ? z \wedge h(a) = ? h(z)}{x = ? b \wedge x = ? z \wedge a = ? z} \mathcal{U}_{\text{dec}}$
$\frac{x = ? a \wedge a = ? z \wedge a = ? z}{x = ? a \wedge z = ? a \wedge a = ? a} \mathcal{U}_{\text{elim}}$	$\frac{x = ? b \wedge b = ? z \wedge a = ? z}{x = ? a \wedge z = ? a \wedge a = ? b} \mathcal{U}_{\text{elim}}$
$\frac{x = ? a \wedge z = ? a \wedge a = ? a}{x = ? a \wedge z = ? a} \mathcal{U}_{\text{triv}}$	$\frac{x = ? a \wedge z = ? a \wedge a = ? b}{x = ? a \wedge z = ? a \wedge a = ? b} \mathcal{U}_{\text{triv}}$
MGU: $[a/x], [a/z]$	not unifiable



©: Michael Kohlhase

239



Unification (Termination)

▷ **Definition 173** Let S and T be multisets and \prec a partial ordering on $S \cup T$. Then we define $S \prec^m T$, iff $S = C \uplus \{s\}$ and $T = C \uplus T'$, where $s \prec t$ for all $t \in T'$. We call \prec^m the **multiset ordering** induced by \prec .

▷ **Lemma 174** If \prec is total/terminating on S , then \prec^m is total/terminating on $\mathcal{P}(S)$.

▷ **Lemma 175** \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. □



Unification (decidable)

▷ **Definition 176** 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 177** 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 $\mathbf{A}, \mathbf{B} \notin \mathcal{V}_i$:

P.2.1.1 then $\mathbf{A} = f(\mathbf{A}^1 \dots \mathbf{A}^n)$ and $\mathbf{B} = f(\mathbf{B}^1 \dots \mathbf{B}^n)$, and thus $\mathcal{U}\text{dec}$ is applicable □

P.2.2 $\mathbf{A} = X \in (\mathcal{V}_i \cap \text{free}(\mathcal{E}))$:

P.2.2.1 then $\mathcal{U}\text{elim}$ (if $\mathbf{B} \neq X$) or $\mathcal{U}\text{triv}$ (if $\mathbf{B} = X$) is applicable. □

□

▷ **Corollary 178** Unification is decidable in PL1

▷ **Proof Idea:** \mathcal{U} -irreducible set of equations are either solved or unsolvable □



4.3 Topics in Logic Programming

Adding Lists to ProLog

- ▷ Lists are represented by terms of the form $[a,b,c,\dots]$
- ▷ 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).
```



©: Michael Kohlhase

242



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).
```



©: Michael Kohlhase

243



Use ProLog for Talking/Programming about Logics

▷ Idea: We will use PL_{NQ} (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,yfx,~). % 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.



©: Michael Kohlhase

244



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).
```



©: Michael Kohlhase

245



5 The Information and Software Architecture of the Internet and WWW

We will now look at the information and software architecture of the Internet and the World Wide Web (WWW) from the ground up.

The Internet and the Web

- ▷ **Definition 179** The **Internet** is a worldwide computer network that connects hundreds of thousands of smaller networks. (The mother of all networks)
- ▷ **Definition 180** The **World Wide Web** is the interconnected system of servers that support multimedia documents, i.e. the multimedia part of the Internet.
- ▷ The Internet and WWW form critical infrastructure for modern society and commerce.
- ▷ The Internet/WWW is huge:

Year	Web	Deep Web	eMail
1999	21 TB	100 TB	11TB
2003	167 TB	92 PB	447 PB
2010	????	?????	?????

- ▷ We want to understand how it works (services and scalability issues)



Units of Information

Bit (b)	<i>binary digit</i>
Byte (B)	<i>8 bit</i>
2 Bytes	A Unicode character.
10 Bytes	your name.
Kilobyte (KB)	<i>1,000 bytes OR 10^3 bytes</i>
2 Kilobytes	A Typewritten page.
100 Kilobytes	A low-resolution photograph.
Megabyte (MB)	<i>1,000,000 bytes OR 10^6 bytes</i>
1 Megabyte	A small novel OR a 3.5 inch floppy disk.
2 Megabytes	A high-resolution photograph.
5 Megabytes	The complete works of Shakespeare.
10 Megabytes	A minute of high-fidelity sound.
100 Megabytes	1 meter of shelved books.
500 Megabytes	A CD-ROM.
Gigabyte (GB)	<i>1,000,000,000 bytes or 10^9 bytes</i>
1 Gigabyte	a pickup truck filled with books.
20 Gigabytes	A good collection of the works of Beethoven.
100 Gigabytes	A library floor of academic journals.

Terabyte (TB)	<i>1,000,000,000,000 bytes or 10^{12} bytes</i>
1 Terabyte	50000 trees made into paper and printed.
2 Terabytes	An academic research library.
10 Terabytes	The print collections of the U.S. Library of Congress.
400 Terabytes	National Climactic Data Center (NOAA) database.
Petabyte (PB)	<i>1,000,000,000,000,000 bytes or 10^{15} bytes</i>
1 Petabyte	3 years of EOS data (2001).
2 Petabytes	All U.S. academic research libraries.
20 Petabytes	Production of hard-disk drives in 1995.
200 Petabytes	All printed material (ever).
Exabyte (EB)	<i>1,000,000,000,000,000,000 bytes or 10^{18} bytes</i>
2 Exabytes	Total volume of information generated in 1999.
5 Exabytes	All words ever spoken by human beings.



A Timeline of the Internet and the Web

- ▷ Early 1960s: introduction of the network concept
- ▷ 1970: ARPANET, scholarly-aimed networks
- ▷ 62 computers in 1974
- ▷ 1975: Ethernet developed by Robert Metcalf
- ▷ 1980: TCP/IP
- ▷ 1982: The first computer virus, Elk Cloner, spread via Apple II floppy disks
- ▷ 500 computers in 1983
- ▷ 28,000 computers in 1987
- ▷ 1989: Web invented by Tim Berners-Lee
- ▷ 1990: First Web browser based on HTML developed by Berners-Lee
- ▷ Early 1990s: Andreessen developed the first graphical browser (Mosaic)
- ▷ 1993: The US White House launches its Web site
- ▷ 1993 –: commercial/public web explodes



©: Michael Kohlhase

248



We will now look at the information and software architecture of the Internet and the World Wide Web (WWW) from the ground up. We will show aspects of how the Internet can cope with this enormous growth of numbers of computers, connections and services.

5.1 Internet Basics

The growth of the Internet rests on three design decisions taken very early on. The Internet

1. is a packet-switched network rather than a network, where computers communicate via dedicated physical communication lines.
2. is a network, where control and administration are decentralized as much as possible.
3. is an infrastructure that only concentrates on transporting packets/datagrams between computers. It does not provide special treatment to any packets, or try to control the content of the packets.

The first design decision is a purely technical one that allows the existing communication lines to be shared by multiple users, and thus save on hardware resources. The second decision allows the administrative aspects of the Internet to scale up. Both of these are crucial for the scalability of the Internet. The third decision (often called “net neutrality”) is hotly debated. The defenders cite that net neutrality keeps the Internet an open market that fosters innovation, where as the attackers say that some uses of the network (illegal file sharing) disproportionately consume resources.

Package-Switched Networks

- ▷ **Definition 181** A **packet-switched network** divides messages into small **network packets** that are transported separately and re-assembled at the target.
- ▷ **Advantages:**
 - ▷ many users can share the same physical communication lines.
 - ▷ packets can be routed via different paths. (bandwidth utilization)
 - ▷ bad packets can be re-sent, while good ones are sent on. (network reliability)
 - ▷ packets can contain information about their sender, destination.
 - ▷ no central management instance necessary (scalability, resilience)



©: Michael Kohlhase

249



These ideas are implemented in the Internet Protocol Suite, which we will present in the rest of the section. A main idea of this set of protocols is its layered design that allows to separate concerns and implement functionality separately.

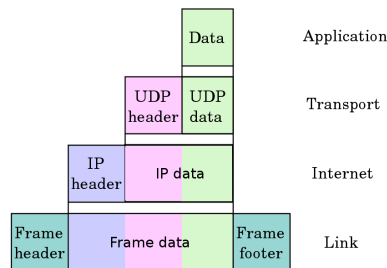
The Internet Protocol Suite

- ▷ **Definition 182** The **Internet Protocol Suite** (commonly known as **TCP/IP**) is the set of communications protocols used for the Internet and other similar networks. It structured into 4 layers.

Layer	e.g.
Application Layer	HTTP, SSH
Transport Layer	UDP, TCP
Internet Layer	IPv4, IPsec
Link Layer	Ethernet, DSL

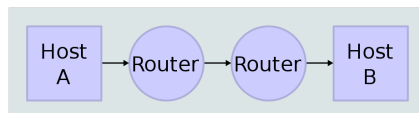
- ▷ **Layers in TCP/IP:** TCP/IP uses encapsulation to provide abstraction of protocols and services.

An application (the highest level of the model) uses a set of protocols to send its data down the layers, being further encapsulated at each level.

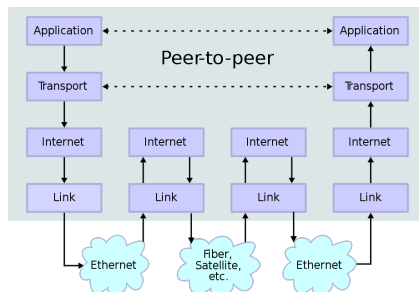


- ▷ **Example 183 (TCP/IP Scenario)** Consider a situation with two Internet host computers communicate across local network boundaries.
- ▷ network boundaries are constituted by internetworking gateways (routers).
- ▷ **Definition 184** A **router** is a purposely customized computer used to forward data among computer networks beyond directly connected devices.
- ▷ A router implements the link and internet layers only and has two network connections.

Network Connections



Stack Connections



We will now take a closer look at each of the layers shown above, starting with the lowest one.



Instead of going into network topologies, protocols, and their implementation into physical signals that make up the link layer, we only discuss the devices that deal with them. Network Interface controllers are specialized hardware that encapsulate all aspects of link-level communication, and we take them as black boxes for the purposes of this course.

Network Interfaces

- ▷ The nodes in the Internet are computers, the edges communication channels
- ▷ **Definition 185** A **network interface controller (NIC)** is a hardware device that handles an interface to a computer network and thus allows a network-capable device to access that network.
- ▷ **Definition 186** Each NIC contains a unique number, the **media access control address (MAC address)**, identifies the device uniquely on the network.
- ▷ MAC addresses are usually 48-bit numbers issued by the manufacturer, they are usually displayed to humans as six groups of two hexadecimal digits, separated by hyphens (-) or colons (:), in transmission order, e.g. 01-23-45-67-89-AB, 01:23:45:67:89:AB.
- ▷ **Definition 187** A **network interface** is a software component in the operating system that implements the higher levels of the network protocol (the NIC handles the lower ones).

Layer	e.g.
Application Layer	HTTP, SSH
Transport Layer	TCP
Internet Layer	IPv4, IPsec
Link Layer	Ethernet, DSL

- ▷ A computer can have more than one network interface. (e.g. a router)


©: Michael Kohlhase
251


The next layer is the Internet Layer.

Internet Protocol and IP Addresses

- ▷ **Definition 188** The **Internet Protocol (IP)** is a protocol used for communicating data across a packet-switched internetwork. The Internet Protocol defines addressing methods and structures for datagram encapsulation. The Internet Protocol also routes data packets between networks
- ▷ **Definition 189** An Internet Protocol (IP) address is a numerical label that is assigned to devices participating in a computer network, that uses the Internet Protocol for communication between its nodes.
- ▷ An IP address serves two principal functions: host or network interface identification and location addressing.
- ▷ **Definition 190** The global IP address space allocations are managed by the **Internet Assigned Numbers Authority (IANA)**, delegating allocate IP address blocks to five Regional Internet Registries (RIRs) and further to Internet service providers (ISPs).
- ▷ **Definition 191** The Internet mainly uses **Internet Protocol Version 4 (IPv4)** [RFC80], which uses 32-bit numbers (**IPv4 addresses**) for identification of network interfaces of Computers.
- ▷ IPv4 was standardized in 1980, it provides 4,294,967,296 (2^{32}) possible unique addresses. With the enormous growth of the Internet, we are fast running out of IPv4 addresses
- ▷ **Definition 192** **Internet Protocol Version 6 (IPv6)** [DH98], which uses 128-bit numbers (**IPv6 addresses**) for identification.
- ▷ Although IP addresses are stored as binary numbers, they are usually displayed in human-readable notations, such as 208.77.188.166 (for IPv4), and 2001:db8:0:1234:0:567:1:1 (for IPv6).



The Internet infrastructure is currently undergoing a dramatic retooling, because we are moving from IPv4 to IPv6 to counter the depletion of IP addresses. Note that this means that all routers and switches in the Internet have to be upgraded. At first glance, it would seem that that this problem could have been avoided if we had only anticipated the need for more the 4 million computers. But remember that TCP/IP was developed at a time, where the Internet did not exist yet, and it's precursor had about 100 computers. Also note that the IP addresses are part of every packet, and thus reserving more space for them would have wasted bandwidth in a time when it was scarce.

The Transport Layer

- ▷ **Definition 193** The **transport layer** is responsible for delivering data to the appropriate application process on the host computers by forming data packets, and adding source and destination port numbers in the header.
- ▷ **Definition 194** The internet protocol mainly suite uses the **Transmission Control Protocol (TCP)** and **Transmission Control Protocol (UDP)** protocols at the transport layer.
- ▷ TCP is used for communication, UDP for multicasting and broadcasting.
- ▷ TCP supports virtual circuits, i.e. provide connection oriented communication over an underlying packet oriented datagram network. (hide/reorder packets)
- ▷ TCP provides end-to-end reliable communication (error detection & automatic repeat)



©: Michael Kohlhase

253



The Application Layer

- ▷ **Definition 195** The **application layer** of the internet protocol suite contains all protocols and methods that fall into the realm of process-to-process communications via an Internet Protocol (IP) network using the Transport Layer protocols to establish underlying host-to-host connections.
- ▷ **Example 196 (Some Application Layer Protocols and Services)**

BitTorrent	Peer-to-peer	Atom	Syndication
DHCP	Dynamic Host Configuration	DNS	Domain Name System
FTP	File Transfer Protocol	HTTP	HyperText Transfer
IMAP	Internet Message Access	IRCP	Internet Relay Chat
NFS	Network File System	NNTP	Network News Transfer
NTP	Network Time Protocol	POP	Post Office Protocol
RPC	Remote Procedure Call	SMB	Server Message Block
SMTP	Simple Mail Transfer	SSH	Secure Shell
TELNET	Terminal Emulation	WebDAV	Write-enabled Web



©: Michael Kohlhase

254



Domain Names

- ▷ **Definition 197** The **DNS (Domain Name System)** is a distributed set of servers that provides the mapping between (static) IP addresses and domain names.
- ▷ **Example 198** e.g. `www.kwarc.info` stands for the IP address 212.201.49.189.
- ▷ networked computers can have more than one DNS name. (virtual servers)
- ▷ Domain names must be registered to ensure uniqueness (registration fees vary, cybersquatting)
- ▷ **Definition 199 ICANN** is a non-profit organization was established to regulate human-friendly domain names. It approves domain name registrars and delegates the actual registration to them.



©: Michael Kohlhase

255



Domain Name Top-Level Domains

- ▷ .com (.commercial) is a generic top-level domain. It was one of the original top-level domains, and has grown to be the largest in use.
- ▷ .org (.organization) is a generic top-level domain, and is mostly associated with non-profit organizations. It is also used in the charitable field, and used by the open-source movement. Government sites and Political parties in the US have domain names ending in .org
- ▷ .net (.network) is a generic top-level domain and is one of the original top-level domains. Initially intended to be used only for network providers (such as Internet service providers). It is still popular with network operators, it is often treated as a second .com. It is currently the third most popular top-level domain.
- ▷ .edu (.education) is the generic top-level domain for educational institutions, primarily those in the United States. One of the first top-level domains, .edu was originally intended for educational institutions anywhere in the world. Only post-secondary institutions that are accredited by an agency on the U.S. Department of Education's list of nationally recognized accrediting agencies are eligible to apply for a .edu domain.
- ▷ .info (.information) is a generic top-level domain intended for informative website's, although its use is not restricted. It is an unrestricted domain, meaning that anyone can obtain a second-level domain under .info. The .info was one of many extension(s) that was meant to take the pressure off the overcrowded .com domain.
- ▷ .gov (.government) a generic top-level domain used by government entities in the United States. Other countries typically use a second-level domain for this purpose, e.g., .gov.uk for the United Kingdom. Since the United States controls the .gov Top Level Domain, it would be impossible for another country to create a domain ending in .gov.
- ▷ .biz (business) the name is a phonetic spelling of the first syllable of "business." A generic top-level domain to be used by businesses. It was created due to the demand for good domain names available in the .com top-level domain, and to provide an alternative to businesses whose preferred .com domain name which had already been registered by another.



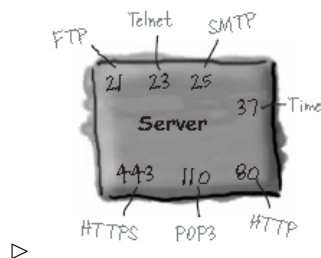
©: Michael Kohlhase

256



Ports

- ▷ **Definition 200** To separate the services and protocols of the network application layer, network interfaces assign them specific **port**, referenced by a number.



▷

Port	use	comment
22	SSH	remote shell
53	DNS	Domain Name System
80	HTTP	World Wide Web
443	HTTPS	HTTP over SSL



©: Michael Kohlhase

257



Internet Governance

- ▷ The Internet is a critical infrastructure for world society and commerce.



©: Michael Kohlhase

258



5.2 Basics Concepts of the World Wide Web

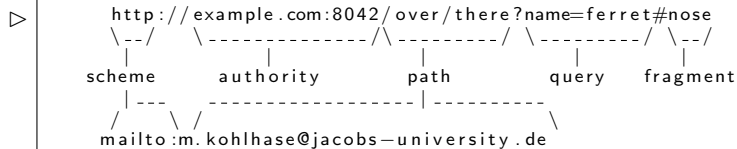
Uniform Resource Identifier (URI), Plumbing of the Web

- ▷ **Definition 201** A **uniform resource identifier** is a global identifiers of network-retrievable documents (**web resources**). URIs adhere a uniform syntax (grammar) defined in RFC-3986 [BLFM05]. Rules contain:

$$\text{URI} ::= (\text{scheme}), ' : ', \text{hierPart}, [('?' \text{query})], [('#' \text{fragment})]$$

$$\text{hier-part} ::= ('/' \text{pathAbempty} | \text{pathAbsolute} | \text{pathRootless} | \text{pathEmpty})$$

- ▷ **Example 202** The following are two example URIs and their component parts:



Note: URIs only **identify** documents, they do not have to be provide access to them (e.g. in a browser).



©: Michael Kohlhase

259



Uniform Resource Locators and relative URIs

- ▷ **Definition 203** A **uniform resource locator** is a URI that that gives access to a web resource via the http protocol.

- ▷ **Example 204** The following URI is a URL (try it in your browser)

`http://kwarc.info/kohlhase/index.html`

- ▷ **Note:** URI/URLs are one of the core features of the web infrastructure, they are considered to be the **plumbing of the WWW**. (direct the flow of data)

- ▷ **Definition 205** URIs can be abbreviated to **relative URIs**; missing parts are filled in from the context

- ▷ **Example 206**

relative URI	abbreviates	in context
#foo	⟨current-file⟩#foo	current file
../bar.txt	file:///home/kohlhase/foo/bar.txt	file system
../bar.html	http://example.org/foo/bar.html	on the web



©: Michael Kohlhase

260



Web Browsers

- ▷ **Definition 207** A **web Browser** is a software application for retrieving, presenting, and traversing information resources on the World Wide Web, enabling users to view Web pages and to jump from one page to another.
- ▷ **Practical Browser Tools:**
 - ▷ Status Bar: security info, page load progress
 - ▷ Favorites (bookmarks)
 - ▷ View Source: view the code of a Web page
 - ▷ Tools/Internet Options, history, temporary Internet files, home page, auto complete, security settings, programs, etc.
- ▷ **Example 208** e.g. IE, Mozilla Firefox, Safari, etc.
- ▷ **Definition 209** A **web page** is a document on the Web that can include multimedia data
- ▷ **Definition 210** A **web site** is a collection of related Web pages usually designed or controlled by the same individual or company.
- ▷ a web site generally shares a common domain name.



HTTP: Hypertext Transfer Protocol

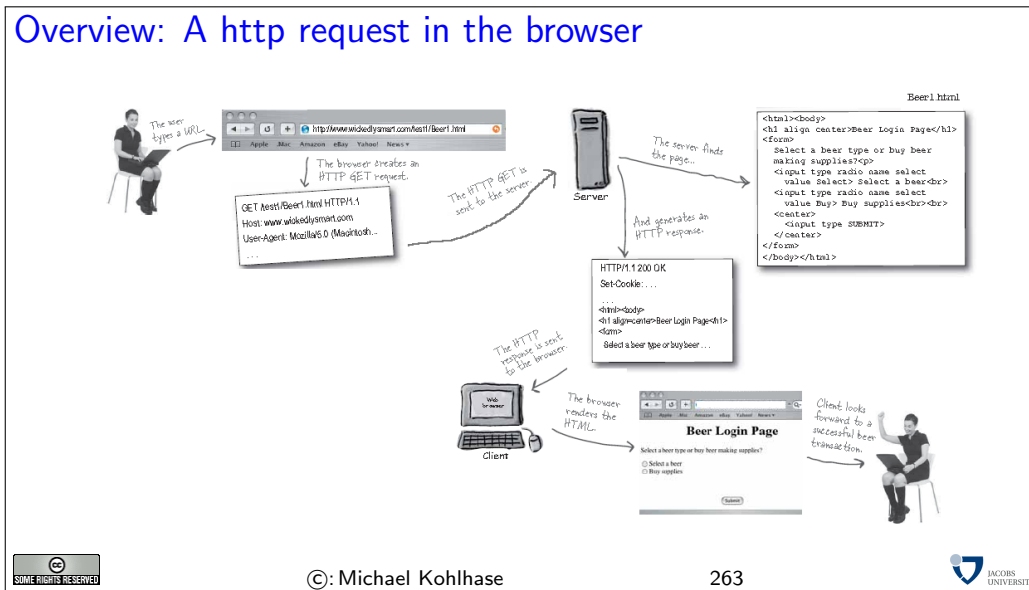
- ▷ **Definition 211** The **Hypertext Transfer Protocol (HTTP)** is an application layer protocol for distributed, collaborative, hypermedia information systems.
- ▷ June 1999: HTTP/1.1 is defined in RFC 2616 [FGM⁺99]
- Definition 212** HTTP is used by a client (called **user agent**) to access web resources (addressed by Uniform Resource Locators (URLs)) via a **http request**. The **web server** answers by supplying the resource
- ▷ Most important HTTP requests (5 more less prominent)

GET	Requests a representation of the specified resource.	safe
PUT	Uploads a representation of the specified resource.	idempotent
DELETE	Deletes the specified resource.	idempotent
POST	Submits data to be processed (e.g., from an HTML form) to the identified resource.	

- ▷ **Definition 213** We call a HTTP request **safe**, iff it does not change the state in the web server. (except for server logs, counters, . . . ; no side effects)
- ▷ **Definition 214** We call a HTTP request **idempotent**, iff executing it twice has the same effect as executing it once.
- ▷ HTTP is a stateless protocol (very memory-efficient for the server.)



Overview: A http request in the browser



©: Michael Kohlhase

263



Example: An http request in real life

- ▷ Connect to the web server (port 80) (so that we can see what is happening)

```
telnet www.kwarc.info 80
```

- ▷ Send off the GET request

```
GET /teaching/GenCS2.html http/1.1
Host: www.kwarc.info
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-US; rv:1.9.2.4)
Gecko/20100413 Firefox/3.6.4
```

- ▷ Response from the server

```
HTTP/1.1 200 OK
Date: Mon, 03 May 2010 06:48:36 GMT
Server: Apache/2.2.9 (Debian) DAV/2 SVN/1.5.1 mod_fastcgi/2.4.6 PHP/5.2.6-1+lenny8 with
Suhosin-Patch mod_python/3.3.1 Python/2.5.2 mod_ssl/2.2.9 OpenSSL/0.9.8g
Last-Modified: Sun, 02 May 2010 13:09:19 GMT
ETag: "1c78b-db1-4859c2f221dc0"
Accept-Ranges: bytes
Content-Length: 3505
Content-Type: text/html
```

```
<!-- This file was generated by ws2html.xsl. Do NOT edit manually! -->
<html xmlns="http://www.w3.org/1999/xhtml"><head>...</head></html>
```



©: Michael Kohlhase

264



HTML: Hypertext Markup Language

- ▷ **Definition 215** The **HyperText Markup Language (HTML)**, is a representation format for web pages. Current version 4.01 is defined in [RHJ98].
- ▷ **Definition 216 (Main markup tags of HTML)** HTML marks up the structure and appearance of text with tags of the form `<e1>` (begin) and `</e1>` (end), where `e1` is one of the following

structure	html, head, body	metadata	title, link, meta
headings	h1, h2, ..., h6	paragraphs	p, br
lists	ul, ol, dl, ..., li	hyperlinks	a
images	img	tables	table, th, tr, td, ...
CSS style	style, div, span	old style	b, u, tt, i, ...
interaction	script	forms	form, input, button

- ▷ **Example 217 A** (very simple) HTML file.

```
<html>
  <body>
    <p>Hello GenCSIII!</p>
  </body>
</html>
```

- ▷ **Example 218** Forms contain input fields and explanations.

```
<form name="input" action="html_form_submit.asp" method="get" >
  Username: <input type="text" name="user" />
  <input type="submit" value="Submit" />
</form>
```

Username:



CSS: Cascading Style Sheets

- ▷ **Idea:** Separate structure/function from appearance.

Definition 219 The **Cascading Style Sheets (CSS)**, is a style sheet language that allows authors and users to attach style (e.g., fonts and spacing) to structured documents. Current version 2.1 is defined in [BCHL09].

- ▷ **Example 220** Our text file from Example 217 with embedded CSS

```
<html>
  <head>
    <style type="text/css" >
      body {background-color:#d0e4fe;}
      h1   {color:orange;
           text-align:center;}
      p    {font-family:"Verdana";
           font-size:20px;}
    </style>
  </head>
  <body>
    <h1>CSS example</h1>
    <p>Hello GenCSII!</p>
  </body>
</html>
```

CSS example

Hello GenCSII!



©: Michael Kohlhase

266



References

- [BCHL09] Bert Bos, Tantek Celik, Ian Hickson, and Høakon Wium Lie. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. W3C Candidate Recommendation, World Wide Web Consortium (W3C), 2009.
- [BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internet Engineering Task Force, 2005.
- [DH98] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2460, Internet Engineering Task Force, 1998.
- [ECM09] ECMAScript language specification. ECMA Standard ECMA-262, ECMA International, December 2009. 5th Edition.
- [FGM⁺99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, 1999.
- [RFC80] DOD standard internet protocol. RFC 760, Internet Engineering Task Force, 1980.
- [RHJ98] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.0 Specification. W3C Recommendation REC-html40, World Wide Web Consortium (W3C), April 1998.
- [XML] Extensible Markup Language (XML) 1.0 (Fourth Edition). Web site at <http://www.w3.org/TR/REC-xml/>.

Index

- astarSearch* search, 96
- n*-bit
 - full adder, 23
- access
 - random (), 41
- accumulator, 41
- acyclic, 7
 - directed (), 7
- adder, 21
 - carry chain, 24
 - carry lookahead, 27
 - conditional sum, 25
 - full, 22
 - half, 21
 - twos-complement, 32
- addition
 - carry chain, 20
 - rules, 20
- address
 - decoder, 39
 - IPv4, 127
 - IPv6, 127
 - MAC, 126
- admissible, 96
- agent
 - user, 131
- algorithm
 - search, 71
- ALU, 35, 41–43
- anchor
 - cell, 61
- answer
 - substitution, 108
- application
 - layer, 128
- arithmetic
 - logic
 - unit, 35, 36
- assembler, 46
 - language, 41, 43
- asynchronous, 38
- balanced
 - fully (), 13
 - tree, 13
- base, 18
 - knowledge, 107
- bijection, 19
- binary
 - natural number, 28
 - tree, 13
- bit
 - carry, 21
 - most significant, 28
 - sign, 28
 - sum, 21
- Boolean
 - expressions, 11
- bootstrapping
 - process, 19
- borrow
 - input (), 32
- Browser
 - web, 131
- byte
 - code, 46, 55
- C, 58
- carry
 - bit, 21
 - input, 22
 - intermediate, 33
- carry chain
 - adder, 24
 - addition, 20
- carry lookahead
 - adder, 27
- cell
 - anchor, 61
 - internal, 61
- character
 - code, 19
- child, 8
- circuit
 - combinational, 9
 - combinatorial, 9, 10
- clause
 - fact, 112
 - Horn, 112
 - rule, 112
- clock, 38
- code
 - byte, 46, 55
 - character, 19
 - string, 19
- combinational
 - circuit, 9
- combinatorial
 - circuit, 9, 10

- command interpreter, 47
- compiler, 46, 47
- complete, 117
- computes, 12
- conditional sum
 - adder, 25
- control, 26
- correct, 117
- cost, 12
- counter
 - program, 41, 42
- CPU, 41
- CSS, 134
- Cascading Style Sheets, 134
- current
 - instruction, 42
 - state, 101
- cycle, 7
- cyclic, 7
- DAG, 7
- data
 - store, 41
- decoder
 - address, 39
- digit, 18
- digits, 18
- digraph, 3
- directed
 - acyclic
 - graph, 7
 - edge, 3
 - graph, 3, 4
- DNF, 16
- DNS, 128
- Domain Name System, 128
- dominates, 100
- edge, 4
 - directed, 3
 - undirected, 3
- end, 7
- environment, 55
- equational
 - system, 116
- equivalent
 - graph, 5
- evaluation
 - function, 96
- exclusive
 - or, 17
- expression
 - label, 11
- expressions
 - Boolean, 11
- fact, 107
 - clause, 112
- firmware, 43
- form
 - solved, 116
- \mathcal{U} -reducible, 118
- frame
 - pointer, 61
- fringe, 78
- full
 - n -bit (), 23
 - adder, 22
- fully
 - balanced
 - tree, 13
- function
 - evaluation, 96
- functional
 - programming
 - language, 43
- gate, 10
- general
 - more, 115
- goal, 108
 - state, 73
- graph
 - directed, 3, 4
 - equivalent, 5
 - isomorphism, 5
 - labeled, 6
 - undirected, 3, 4
- depth, 8
- greedy
 - search, 94
- half
 - adder, 21
- heuristic, 94
- hexadecimal
 - numbers, 43
- Horn
 - clause, 112
- HTML, 133
- HyperText Markup Language, 133
- HTTP, 131
- Hypertext Transfer Protocol, 131
- http
 - request, 131
- IANA, 127
- Internet Assigned Numbers Authority, 127

- ICANN, 128
- idempotent, 131
- imperative
 - programming
 - language, 43
- in-degree, 3
- initial, 5
 - node, 5
 - state, 73
- input
 - borrow
 - bit, 32
 - carry, 22
 - vertex, 10
- instruction
 - current, 42
 - program, 43
- interface
 - network, 126
- intermediate
 - carry, 33
- internal
 - cell, 61
- Internet, 121
- Internet Protocol, 127
- invariants
 - loop, 46
- IP, 127
- IPv4, 127
 - address, 127
- Internet Protocol Version 4, 127
- IPv6, 127
 - address, 127
- Internet Protocol Version 6, 127
- isomorphism
 - graph, 5
- Java, 58
- jump
 - table, 51
- key, 55
- knowledge
 - base, 107
- label, 6
 - expression, 11
- labeled
 - graph, 6
- language
 - assembler, 41, 43
- layer
 - application, 128
 - transport, 128
- leaf, 8
- leak
 - memory, 57
- length, 7
- LIFO, 47
- local
 - search, 101
- logic
 - arithmetic (), 35, 36
 - sequential (), 36
- logical
 - program, 112
- loop
 - invariants, 46
- MAC
 - address, 126
- media access control address, 126
- machine, 46
 - register, 41, 42, 46
 - virtual, 46, 47
- management
 - memory (), 41
- memory
 - leak, 57
 - management
 - unit, 41
 - random access, 40, 41
- most general
 - unifier, 115
- minimal
 - polynomial, 16
- MMU, 41–43
- more
 - general, 115
- most significant
 - bit, 28
- multiplexer, 26
- multiset
 - ordering, 118
- natural
 - binary (), 28
- network
 - interface, 126
 - packet-switched, 124
 - packets, 124
- NIC, 126
- network interface controller, 126
- node, 3, 4
 - initial, 5
 - terminal, 5
- number
 - positional (), 18

- numbers
 - hexadecimal, 43
- offline
 - problem
 - solving, 72
- operator, 73
- or
 - exclusive, 17
- ordered
 - pair, 3, 4
- ordering
 - multiset, 118
- out-degree, 3
- output
 - vertex, 10
- overflow, 35
- packet-switched
 - network, 124
- packets
 - network, 124
- page
 - web, 131
- pair, 3
 - ordered, 3, 4
- parent, 8
- parse-tree, 9
- Pascal, 58
- path, 7
- pointer
 - frame, 61
 - stack, 50
- polarity, 28
- polynomial
 - minimal, 16
- port, 129
- positional
 - number
 - system, 18
- positive
 - unit-resulting
 - hyperresolution, 112
- problem
 - offline (), 72
- procedure
 - static, 58
- process
 - bootstrapping, 19
- program, 41
 - counter, 41, 42
 - instruction, 43
 - logical, 112
 - store, 41
- program store, 47
- programming
 - functional (), 43
 - imperative (), 43
- pull-back, 19, 21
- query, 108, 112
- Quine-McCluskey, 16
- radix, 18
- RAM, 40–42
- random access
 - memory, 41
- random
 - access
 - memory, 41
- random access
 - memory, 40
- register, 41, 42
 - machine, 41, 42, 46
- relation, 4
- relative
 - URI, 130
- request
 - http, 131
- resource
 - uniform (), 130
 - web, 130
- Ridge, 103
- root, 8
- router, 125
- RS-flipflop, 36
- RS-latch, 36
- rule, 107
 - clause, 112
- rules
 - addition, 20
- safe, 131
- search
 - algorithm, 71
 - greedy, 94
 - local, 101
 - strategy, 78
- sequential
 - logic
 - circuit, 36
- server
 - web, 131
- set, 3
- sign
 - bit, 28
- sink, 5
- site

- web, 131
- solution, 73
- solved
 - form, 116
- source, 5
- stack, 47
 - pointer, 50
- start, 7
- state, 73
 - current, 101
 - goal, 73
 - initial, 73
- static
 - procedure, 58
- store
 - data, 41
 - program, 41
- strategy
 - search, 78
- string
 - code, 19
- substitution
 - answer, 108
- subtractor, 32
- sum
 - bit, 21
- synchronous, 38
- system
 - equational, 116
- table
 - jump, 51
- TCP, 128
- Transmission Control Protocol, 128
- Internet Protocol Suite, 125
- TCP/IP, 125
- terminal, 5
 - node, 5
- transport
 - layer, 128
- tree, 8
 - balanced, 13
 - binary, 13
- Turing
 - universal (), 71
- two's complement, 30
- twos-complement
 - adder, 32
- UDP, 128
- Transmission Control Protocol, 128
- underflow, 35
- undirected
 - edge, 3
- graph, 3, 4
- unifier, 115
 - most general, 115
- uniform
 - resource
 - identifier, 130
 - locator, 130
- unit-resulting
 - positive (), 112
- unitary, 117
- universal
 - Turing
 - machine, 71
- URI
 - relative, 130
- user
 - agent, 131
- vertex, 3
 - input, 10
 - output, 10
- depth, 8
- virtual
 - machine, 46, 47
- virtual program counter, 47
- VPC, 47
- web
 - Browser, 131
 - page, 131
 - resource, 130
 - server, 131
 - site, 131
- word, 41, 46
- World Wide Web, 121