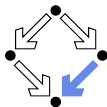
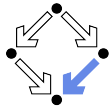


# Turing Complete Computational Models

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)  
Johannes Kepler University, Linz, Austria  
<http://www.risc.jku.at>





---

## 1. Random Access Machines

## 2. Loop and While Programs

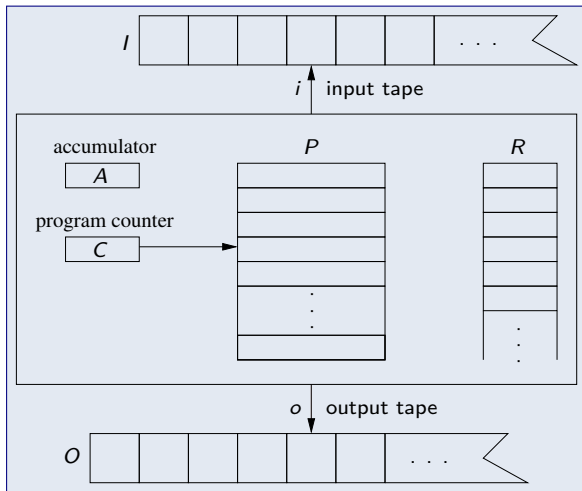
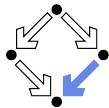
## 3. Primitive Recursive and $\mu$ -recursive Functions

## 4. Further Turing Complete Models

## 5. The Chomsky Hierarchy

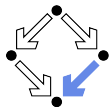
## 6. Real Computers

# A Random Access Machine



A model closer to a real computer.

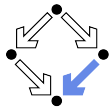
# A Random Access Machine



- A **random access machine (RAM)**:
  - an infinite input tape  $I$  (whose cells can hold natural numbers of arbitrary size) with a read head position  $i \in \mathbb{N}$ ,
  - an infinite output tape  $O$  (whose cells can hold natural numbers of arbitrary size) with a write head position  $o \in \mathbb{N}$ ,
  - an accumulator  $A$  which can hold a natural number of arbitrary size,
  - a program counter  $C$  which can hold an arbitrary natural number,
  - a program consisting of a finite number of instructions  $P[1], \dots, P[m]$ ,
  - a memory consisting of a countably infinite sequence of registers  $R[1], R[2], \dots$ , each of which can hold an arbitrary natural number.
- **Execution**:
  - Initially,  $i = 0$ ,  $o = 0$ ,  $A = 0$ ,  $C = 1$ ,  $R[1] = R[2] = \dots = 0$ .
  - In every step, the RAM reads  $P[C]$ , increments  $C$  by 1, and then performs the action indicated by the instruction.
  - Execution terminates when  $C = 0$ .

Program is a sequence of machine instructions.

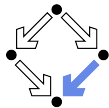
# RAM Instructions



Instruction	Description	Action
IN	Read value from input tape into accumulator	$A := I[i]; i := i + 1$
OUT	Write value from accumulator to output tape	$O[o] := A; o := o + 1$
LOAD # $n$	Load constant $n$ into accumulator	$A := n$
LOAD $n$	Load content of register $n$ into accumulator	$A := R[n]$
LOAD ( $n$ )	Load content of register referenced by reg. $n$	$A := R[R[n]]$
STORE $n$	Store content of accumulator into register $n$	$R[n] := A$
STORE ( $n$ )	Store content into register referenced by reg. $n$	$R[R[n]] := A$
ADD # $n$	Increment content of accumulator by constant	$A := A + n$
SUB # $n$	Decrement content of accumulator by constant	$A := \max\{0, A - n\}$
JUMP $n$	Unconditional jump to instruction $n$	$C := n$
BEQ $i, n$	Conditional jump to instruction $n$	if $A = i$ then $C := n$

Immediate addressing, direct addressing, indirect addressing.

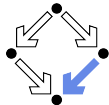
# Example



```
START:  LOAD #1      A := 1
        STORE 1     R[1] := A
READ:   LOAD 1      A := R[1]
        ADD #1     A := A + 1
        STORE 1     R[1] := A
        IN         A := I[i]; i := i + 1
        BEQ 0,WRITE if A = 0 then C := WRITE
        STORE (1)   R[R[1]] := A
        JUMP READ   C := READ
WRITE:  LOAD 1      A := R[1]
        SUB #1     A := A - 1
        STORE 1     R[1] := A
        BEQ 1,HALT if A = 1 then C := HALT
        LOAD (1)   A := R[R[1]]
        OUT        O[o] := A; o := o + 1
        JUMP WRITE  C := WRITE
HALT:   JUMP 0      C := 0
```

Reads  $x_1, \dots, x_n, 0$  and writes  $x_n, \dots, x_1$  using stack  $R[2], \dots, R[n+1]$ .

# RAMs versus Turing Machines

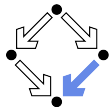


**Theorem:** Every Turing machine can be simulated by a RAM.

- RAM uses registers  $R[1], \dots, R[c - 1]$  for its own purposes,
- stores in  $R[c]$  the position of the tape head of the Turing machine,
- uses  $R[c + 1], R[c + 2], \dots$  as a virtual Turing machine tape.
  - Using “indirect addressing” operations  $\text{LOAD}(n)$  and  $\text{STORE}(n)$ .
- RAM copies the input from the input tape into its virtual tape, then it mimics the execution of the Turing machine on the virtual tape.
- When the simulated Turing machine terminates, the content of the virtual tape is copied to the output tape.

RAMs represent a Turing complete computational model.

# RAMs versus Turing Machines



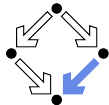
**Theorem:** Every RAM can be simulated by a Turing machine.

- The Turing machine uses 5 tapes to simulate the RAM:
  - Tape 1 represents the input tape of the RAM.
  - Tape 2 represents the output tape of the RAM.
  - Tape 3 holds a representation of that part of the memory that has been written by the simulation of the RAM.
  - Tape 4 holds a representation of the accumulator of the RAM.
  - Tape 5 serves as a working tape.
- Tape 3 holds a sequence of (address,contents) pairs that represent those registers of the RAM that have been written during the simulation (the contents of all other registers hold 0).
- Every instruction of the RAM is simulated by a sequence of steps of the Turing machine which reads respectively writes Tape 1 and 2 and updates on Tape 3 and 4 the tape representations of the contents of the memory and the accumulator.

**RAMs are not more powerful than Turing machines.**



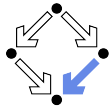
# Random Access Stored Program Machine



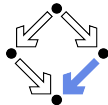
The program of a RAM is “read-only”.

- **Random Access Stored Program Machine (RASP).**
  - A RAM variant where the program is stored in memory  $R$  (there is no separate program store  $P$ ).
- **Every RASP can be simulated by a RAM.**
  - RAM is interpreter for RASP instructions (like a *microprogram* in a processor interprets machine instructions).
- **Every RAM can be simulated by a RASP.**
  - Even if indirect addressing is removed from RASP.
  - RAM instructions  $\text{LOAD}(n)$  and  $\text{STORE}(n)$  can be interpreted by self-modifying RASP code.

**Self modifying programs do not add computational power to a RAM.**



- 
1. Random Access Machines
  - 2. Loop and While Programs**
  3. Primitive Recursive and  $\mu$ -recursive Functions
  4. Further Turing Complete Models
  5. The Chomsky Hierarchy
  6. Real Computers



# Loop Programs

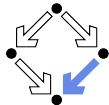
---

- Loop Program  $P$ :

$$P ::= x_i := 0 \mid x_i := x_j + 1 \mid x_i := x_j - 1 \mid P; P \\ \mid \text{loop } x_i \text{ do } P \text{ end.}$$

- Set  $\{x_0, x_1, x_2, \dots\}$  of program variables.
- Initial value of  $x_i$  determines the number of loop iterations.
- Loop must eventually terminate.

Programs with bounded iteration that necessarily terminate.



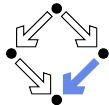
- **Semantics**  $\llbracket P \rrbracket(m)$  maps the start memory  $m : \mathbb{N} \rightarrow \mathbb{N}$  to the final memory after the termination of  $P$ :

$$\begin{aligned}\llbracket x_i := 0 \rrbracket(m) &:= m[i \leftarrow 0] \\ \llbracket x_i := x_j + 1 \rrbracket(m) &:= m[i \leftarrow m(j) + 1] \\ \llbracket x_i := x_j - 1 \rrbracket(m) &:= m[i \leftarrow \max\{0, m(j) - 1\}] \\ \llbracket P_1; P_2 \rrbracket(m) &:= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(m)) \\ \llbracket \mathbf{loop} \ x_i \ \mathbf{do} \ P \ \mathbf{end} \rrbracket(m) &:= \llbracket P \rrbracket^{m(i)}(m)\end{aligned}$$

- $m[i \leftarrow n]$ : memory  $m$  after updating the value  $x_i$  by value  $n$ .
- $\llbracket P \rrbracket^n(m)$ : memory  $m$  after  $n$  times executing  $P$ :

$$\begin{aligned}\llbracket P \rrbracket^0(m) &:= m \\ \llbracket P \rrbracket^{n+1}(m) &:= \llbracket P \rrbracket(\llbracket P \rrbracket^n(m))\end{aligned}$$

A loop program denotes a function over memories.



# Syntactic Abbreviations

- $x_i := x_j$

```
 $x_i := x_j + 1; x_i := x_i - 1$ 
```

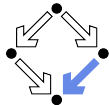
- $x_i := n$

```
 $x_i := 0; x_i := x_i + 1; x_i := x_i + 1; \dots; x_i := x_i + 1$ 
```

- **if**  $x_i = 0$  **then**  $P_t$  **else**  $P_e$  **end**

```
 $x_t := 1; \text{loop } x_i \text{ do } x_t := 0; \text{end};$   
 $x_e := 1; \text{loop } x_t \text{ do } x_e := 0; \text{end};$   
loop  $x_t$  do  $P_t$  end; loop  $x_e$  do  $P_e$  end;
```

The usual programming language constructs (except for unbounded iteration) can be represented.



# Loop Computability

---

We consider the computability of functions over the natural numbers.

$f : \mathbb{N}^n \rightarrow \mathbb{N}$  is **loop computable**, if there exists a loop program  $P$  such that for all  $x_1, \dots, x_n \in \mathbb{N}$  and memory  $m : \mathbb{N} \rightarrow \mathbb{N}$  defined as

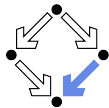
$$m(i) := \begin{cases} x_i & \text{if } 1 \leq i \leq n \\ 0 & \text{else} \end{cases}$$

we have

$$\llbracket P \rrbracket(m)(0) = f(x_1, \dots, x_n)$$

When started in a state where  $x_1, \dots, x_n$  contain the arguments of  $f$ , the program terminates in a state where  $x_0$  holds the result of  $f$ .

# Example



- Addition is computable by the program  $x_0 := x_1 + x_2$  :

```
x0 := x1;  
loop x2 do  
  x0 := x0 + 1  
end
```

- Multiplication is computable by the program  $x_0 := x_1 \cdot x_2$  :

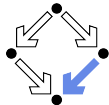
```
x0 := 0;  
loop x2 do  
  x0 := x0 + x1  
end
```

- Exponentiation is computable by the program  $x_0 := x_1^{x_2}$  :

```
x0 := 1;  
loop x2 do  
  x0 := x0 · x1  
end
```

Natural number arithmetic is loop computable.

# Arithmetic



- $x_0 := x_1 \cdot x_2$ :

```
x0 := 0;  
loop x2 do  
  x0 := x0 + x1  
end
```

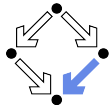
$\rightsquigarrow$

```
x0 := 0;  
loop x2 do  
  x0 := x0;  
  loop x1 do  
    x0 := x0 + 1  
  end  
end
```

Higher arithmetic needs multiply nested loops.



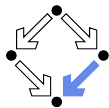
# Beyond Exponentiation



$$a \uparrow^n b := \begin{cases} a^b & \text{if } n = 1 \\ 1 & \text{if } b = 0 \\ a \uparrow^{n-1} (a \uparrow^n (b-1)) & \text{else} \end{cases}$$

- $a \uparrow^1 b = a^b$   
 $a \uparrow^1 b = a \cdot a \cdot \dots \cdot a$  ( $b$  times)
- $a \uparrow^2 b = a^{a^{\dots^a}}$  ( $b$  times)  
 $a \uparrow^2 b = a \uparrow^1 a \uparrow^1 \dots \uparrow^1 a$  ( $b$  times)
- $a \uparrow^3 b$ :  
 $a \uparrow^3 b = a \uparrow^2 a \uparrow^2 \dots \uparrow^2 a$  ( $b$  times)

The notation allows to define arbitrary “complex” arithmetic functions.



# Limits of Loop Computability

- **Theorem:** for every  $n > 0$  and  $f(a, b) := a \uparrow^n b$ 
  - $f$  is loop computable, and
  - every loop program computing  $f$  requires at least  $n + 2$  nested loops.
- **Theorem:**  $g : \mathbb{N}^3 \rightarrow \mathbb{N}, g(a, b, n) := a \uparrow^{n+1} b$  is not loop computable.
  - Assume  $g$  can be computed by a program  $P$  with  $n$  loops.
  - Then the computation of  $g(a, b, n) = a \uparrow^{n+1} b$  requires  $n + 3$  loops.
  - Thus  $P$  cannot compute  $g$ .
- Also the **Ackermann Function** is not loop computable:

$$\text{ack}(0, m) := m + 1$$

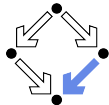
$$\text{ack}(n, 0) := \text{ack}(n - 1, 1)$$

$$\text{ack}(n, m) := \text{ack}(n - 1, \text{ack}(n, m - 1)), \text{ if } n > 0 \wedge m > 0$$

- $\text{ack}(n, m) = 2 \uparrow^{n-2} (m + 3) - 3$
- $\text{ack}(4, 2)$  has 20,000 digits.

Some arithmetic functions grow “too fast” to be loop computable.

# While Programs

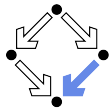


- While Program  $P$ :

$$P ::= \dots \text{ (as for loop programs)}$$
$$| \text{ while } x_i \text{ do } P \text{ end.}$$

- Set  $\{x_0, x_1, x_2, \dots\}$  of program variables.
- Loop is repeated as long as  $x_i \neq 0$ .
- If  $x_i \neq 0$  forever, loop does not terminate.

Programs with unbounded iteration that may not terminate.



- **Semantics**  $\llbracket P \rrbracket(m)$  maps start memory  $m : \mathbb{N} \rightarrow \mathbb{N}$ 
  - to the final memory, if  $P$  terminates, and
  - to the special value  $\perp$  (bottom), if  $P$  does not terminate.
- Semantics generalizes that of loop programs:

$$\llbracket P \rrbracket(m) := \begin{cases} \perp & \text{if } m = \perp \\ \llbracket P \rrbracket'(m) & \text{else} \end{cases}$$

$$\llbracket \dots \rrbracket'(m) := \dots \text{ (as for loop programs)}$$

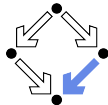
- Semantics of unbounded iteration:

$$\llbracket \mathbf{while} \ x_i \ \mathbf{do} \ P \ \mathbf{end} \rrbracket'(m) := \begin{cases} \perp & \text{if } L_i(P, m) \\ \llbracket P \rrbracket^{T_i(P, m)}(m) & \text{else} \end{cases}$$

$$L_i(P, m) := \Leftrightarrow \forall k \in \mathbb{N} : \llbracket P \rrbracket^k(m)(i) \neq 0$$

$$T_i(P, m) := \min \{ k \in \mathbb{N} \mid \llbracket P \rrbracket^k(m)(i) = 0 \}$$

**A while program denotes a function whose result is either a memory or  $\perp$ .**



# Syntactic Abbreviations

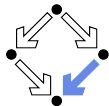
---

- **while**  $x_i < x_j$  **do**  $P$  **end**

```
 $x_k := x_j - x_i;$   
while  $x_k$  do  $P; x_k := x_j - x_i;$  end
```

Analogous constructions possible for other termination conditions.

# While Computability



$f : \mathbb{N}^n \rightarrow_p \mathbb{N}$  is **while computable**, if there exists a while program  $P$  such that for all  $x_1, \dots, x_n \in \mathbb{N}$  and memory  $m : \mathbb{N} \rightarrow \mathbb{N}$  defined as

$$m(i) := \begin{cases} x_i & \text{if } 1 \leq i \leq n \\ 0 & \text{else} \end{cases}$$

the following holds:

- If  $x_1, \dots, x_n \in \text{domain}(f)$ , then  $\llbracket P \rrbracket(m) : \mathbb{N} \rightarrow \mathbb{N}$  and

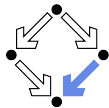
$$\llbracket P \rrbracket(m)(0) = f(x_1, \dots, x_n)$$

- If  $x_1, \dots, x_n \notin \text{domain}(f)$ , then

$$\llbracket P \rrbracket(m) = \perp$$

For a defined value of  $f(x_1, \dots, x_n)$ ,  $P$  terminates with this value in variable  $x_0$ . If  $f(x_1, \dots, x_n)$  is undefined, the program does not terminate.

# Example



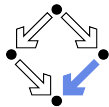
The Ackermann function is while computable with the help of a stack.

```
function ack(n, m):  
  if n = 0 then  
    return m + 1  
  else if m = 0 then  
    return ack(n - 1, 1)  
  end if  
  return ack(n - 1, ack(n, m - 1))  
end function
```

```
function ack(x1, x2):  
  push(x1); push(x2)  
  while size() > 1 do  
    x2 ← pop(); x1 ← pop()  
    if x1 = 0 then  
      push(x2 + 1)  
    else if x2 = 0 then  
      push(x1 - 1); push(1);  
    else  
      push(x1 - 1);  
      push(x1); push(x2 - 1)  
    end if  
  end while  
  return pop()  
end function
```

While programs are computationally more powerful than loop programs.

# Normal Form of a While Program



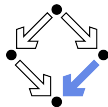
**Kleene's Normal Form Theorem:** every while computable function can be computed by a while program in Kleene's normal form:

```
 $x_c := 1;$   
while  $x_c \neq 0$  do  
  if  $x_c = 1$  then  $P_1$   
  else if  $x_c = 2$  then  $P_2$   
  ...  
  else if  $x_c = n$  then  $P_n$   
  end if  
end while
```

- $P_1, \dots, P_n$  do *not* contain while loops.
- Control variable  $x_c$  determines which  $P_i$  to execute next.

A single while loop is all that is needed.





# Normal Form of a While Program

We sketch the proof of Kleene's Normal Form Theorem.

- A while program can be translated into a goto program:

```
while  $x_i$  do  $P$  end
```

$\rightsquigarrow$

```
 $L_i$  :   if  $x_i = 0$  goto  $L_{i+1}$   
        ;  
        goto  $L_i$   
 $L_{i+1}$  : ...
```

- Gotos can be translated to control variable assignments:

```
goto  $L_j$ 
```

$\rightsquigarrow$

```
 $x_c := j$ 
```

- The resulting program can be translated into normal form:

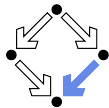
```
 $L_1$  :  $P_1$   
 $L_2$  :  $P_2$   
...  
 $L_n$  :  $P_n$ 
```

$\rightsquigarrow$

```
 $x_c := 1$ ;  
while  $x_c \neq 0$  do  
  if  $x_c = 1$  then  $x_c := 2$ ;  $P_1$   
  else if  $x_c = 2$  then  $x_c := 3$ ;  $P_2$   
  ...  
  else if  $x_c = n$  then  $x_c := 0$ ;  $P_n$   
  end if  
end while
```

In essence, the execution loop of a processor.

# Turing Machines and While Programs

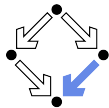


- **Theorem:** Every Turing machine can be simulated by a while program and vice versa.
  - Consequence: every Turing computable function is while computable and vice versa.

Proof  $\Rightarrow$ : construct  $P$  to simulate  $M$ .

- $x_0$  holds initial tape content.
  - Determines initial configuration.
- Machine configuration  $(x_l, x_q, x_r)$ :
  - $x_q$ : the current state.
  - $x_l$ : the tape left to the tape head,
  - $x_r$ : the tape under/right to head.
- State  $x_q$  and symbol  $x_a$  under head determine the state transition.
  - If none is possible, final tape content is written to  $x_0$ .

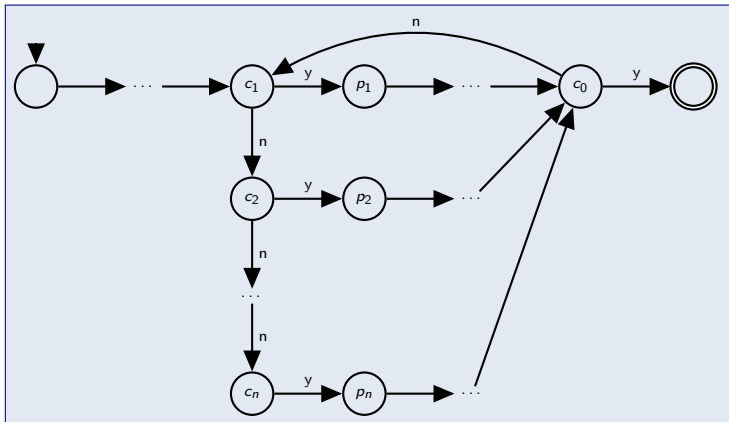
```
( $x_l, x_q, x_r$ ) := input( $x_0$ )
 $x_a$  := head( $x_r$ )
while transition( $x_q, x_a$ ) do
  if  $x_q = q_1 \wedge x_a = a_1$  then
     $P_1$ 
  else if  $x_q = q_2 \wedge x_a = a_2$  then
     $P_2$ 
  else if ... then
    ...
  else if  $x_q = q_n \wedge x_a = a_n$  then
     $P_n$ 
  end
   $x_a$  := head( $x_r$ )
end
 $x_0$  := output( $x_l, x_q, x_r$ )
```

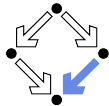


# Turing Machines and While Programs

Proof  $\Leftarrow$ : construct  $M$  to simulate  $P$  (given in normal form).

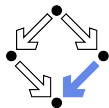
- Each program fragment  $P_i$  is translated into a corresponding fragment of the transition function of  $M$  with sequence of states  $c_i, p_i, \dots, c_0$ .





- 
1. Random Access Machines
  2. Loop and While Programs
  - 3. Primitive Recursive and  $\mu$ -recursive Functions**
  4. Further Turing Complete Models
  5. The Chomsky Hierarchy
  6. Real Computers

# Primitive Recursive Functions



The following functions over the natural numbers are **primitive recursive**:

- The **constant null** function  $0 \in \mathbb{N}$ .
- The **successor** function  $s : \mathbb{N} \rightarrow \mathbb{N}, s(x) := x + 1$ .
- The **projection** functions  $p_i^n : \mathbb{N}^n \rightarrow \mathbb{N}, p_i^n(x_1, \dots, x_n) := x_i$ .
- Every function  $h : \mathbb{N}^n \rightarrow \mathbb{N}$  defined by **composition**

$$h(x_1, \dots, x_n) := f(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$$

from primitive recursive  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  and  $g_1, \dots, g_k : \mathbb{N}^n \rightarrow \mathbb{N}$ .

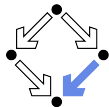
- Every function  $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  defined by **primitive recursion**

$$h(y, x_1 \dots x_n) := \begin{cases} f(x_1, \dots, x_n) & \text{if } y = 0 \\ g(y-1, h(y-1, x_1, \dots, x_n), x_1, \dots, x_n) & \text{else} \end{cases}$$

from primitive recursive  $f : \mathbb{N}^n \rightarrow \mathbb{N}$  and  $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ .

**Starting with the base functions, by composition and primitive recursion new primitive recursive functions can be defined.**

# Understanding Primitive Recursion



- Primitive recursion can be defined by a **pattern matching equation**:

$$\begin{aligned}h(0, x_1 \dots, x_n) &:= f(x_1, \dots, x_n) \\ h(y + 1, x_1 \dots, x_n) &:= g(y, h(y, x_1, \dots, x_n), x_1, \dots, x_n)\end{aligned}$$

- Primitive recursion can be defined by a **pattern matching construct**:

$$\begin{aligned}h(y, x_1 \dots x_n) &:= \\ \text{case } y \text{ of} & \\ 0: & \quad f(x_1, \dots, x_n) \\ z + 1: & \quad g(z, h(z, x_1, \dots, x_n), x_1, \dots, x_n)\end{aligned}$$

- $h(y, x)$  denotes the  $(y - 1)$ -times application of  $g$  starting with  $f(x)$ :

$$h(0, x) = f(x)$$

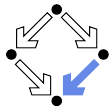
$$h(1, x) = g(0, h(0, x), x) = g(0, f(x), x)$$

$$h(2, x) = g(1, h(1, x), x) = g(1, g(0, f(x), x), x)$$

$$h(3, x) = g(2, h(2, x), x) = g(2, g(1, g(0, f(x), x), x), x)$$

...

$$h(y, x) = g(y - 1, h(y - 1, x), x) = g(y - 1, g(y - 2, \dots, g(0, f(x), x), \dots, x), x)$$



# Example

---

We consider arithmetic on natural numbers.

- Addition  $y + x$  is primitive recursive:

$$0 + x := x$$

$$(y + 1) + x := (y + x) + 1$$

- Multiplication  $y \cdot x$  is primitive recursive:

$$0 \cdot x := 0$$

$$(y + 1) \cdot x := y \cdot x + x$$

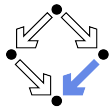
- Exponentiation  $x^y$  is primitive recursive:

$$x^0 := 1$$

$$x^{y+1} := x^y \cdot x$$

Natural number arithmetic is primitive recursive.

# Primitive Recursion and Loop Computability



Both the execution of a loop program and the evaluation of a primitive recursive function are bounded; are they equally expressive?

**Example:** Compute in  $x_0$  the smallest  $n < x_1$  for which  $p(n) = 1$  holds (respectively  $x_0 = x_1$ , if  $p(n) \neq 1$  for all  $n < x_1$ ).

```
x0 := x1
x2 := 0
loop x1 do
  if x0 = x1  $\wedge$  p(x2) = 1 then
    x0 := x2
  end
  x2 := x2 + 1
end
```

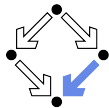
Assume  $n = 3$ :

$x_0$	$x_1$	$x_2$
5	5	0
5	5	1
5	5	2
5	5	3
3	5	4
3	5	5

We will construct a primitive recursive function computing the same value.



# Primitive Recursion and Loop Computability



We mimic the execution of the **loop** by a primitive recursive function *loop* whose recursion parameter denotes the number of loop iterations.

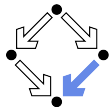
$$\text{min}(x_1) := \text{loop}(x_1, x_1)$$

$$\text{loop}(x_2, x_1) := \begin{cases} x_1 & \text{if } x_2 = 0 \\ \text{if}(x_2 - 1, \text{loop}(x_2 - 1, x_1), x_1) & \text{else} \end{cases}$$

$$\text{if}(x_2, x_0, x_1) := \begin{cases} x_2 & \text{if } x_0 = x_1 \wedge p(x_2) = 1 \\ x_0 & \text{else} \end{cases}$$

- $\text{min}(x_1) := \text{loop}(x_1, x_1)$  computes the value assigned to  $x_0$  for input  $x_1$  (2nd argument) after  $x_1$  iterations of the **loop** (1st argument).
- $\text{loop}(x_2, x_1)$  computes the value assigned to  $x_0$  for input  $x_1$  after  $x_2$  iterations of the **loop**.
- $\text{if}(x_2, x_0, x_1)$  computes the new value assigned to  $x_0$  from the old value of  $x_0$  for input  $x_1$  after  $x_2$  iterations by the **if** statement.

# Primitive Recursion and Loop Computability



Evaluation of  $\text{min}(5) = \text{loop}(5,5)$ .

$$\text{loop}(0,5) = 5$$

$$\text{loop}(1,5) = \text{if}(0, \text{loop}(0,5), 5) = \text{if}(0, 5, 5) = 5$$

$$\text{loop}(2,5) = \text{if}(1, \text{loop}(1,5), 5) = \text{if}(1, 5, 5) = 5$$

$$\text{loop}(3,5) = \text{if}(2, \text{loop}(2,5), 5) = \text{if}(2, 5, 5) = 5$$

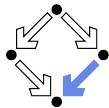
$$\text{loop}(4,5) = \text{if}(3, \text{loop}(3,5), 5) = \text{if}(3, 5, 5) = 3$$

$$\text{loop}(5,5) = \text{if}(4, \text{loop}(4,5), 5) = \text{if}(4, 3, 5) = 3$$

$x_0$	$x_1$	$x_2$
5	5	0
5	5	1
5	5	2
5	5	3
3	5	4
3	5	5

In sequence of evaluations of  $\text{loop}(x_2, x_1) = x_0$  the values  $(x_0, x_1, x_2)$  correspond to the program trace of the loop program.

# Primitive Recursion and Loop Computability



**Theorem:** every prim. recursive function is loop computable and vice versa.

Proof  $\Rightarrow$ : we show that primitive recursive function  $h$  is loop computable.

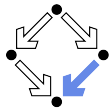
- If  $h$  is one of the basic functions, it is clearly loop computable.
- Case  $h(x_1, \dots, x_n) := f(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$

```
y1 := g1(x1, ..., xn);  
y2 := g2(x1, ..., xn);  
...  
yk := gk(x1, ..., xn);  
x0 := f(y1, ..., yk)
```

- Case  $h(y, x_1 \dots x_n) := \begin{cases} f(x_1, \dots, x_n) & \text{if } y = 0 \\ g(y - 1, h(y, x_1, \dots, x_n), x_1, \dots, x_n) & \text{else} \end{cases}$

```
x0 := f(x1, ..., xn); xy := 0;  
loop y do  
  x0 := g(xy, x0, x1, ..., xn);  
  xy := xy + 1  
end
```

# Primitive Recursion and Loop Computability



Proof  $\Leftarrow$ : let  $h$  be computable by loop program  $P$ . Let  $f_P : \mathbb{N}^{n+1} \rightarrow \mathbb{N}^{n+1}$  be the function that maps the initial values of the variables used by  $P$  to their final values. We show by induction on  $P$  that  $f_P$  is primitive recursive.

- Case  $x_j := k$  :

$$f_P(x_0, \dots, x_n) := (x_0, \dots, x_{j-1}, k, x_{j+1}, \dots, x_n)$$

- Case  $x_j := x_j \pm 1$  :

$$f_P(x_0, \dots, x_n) := (x_0, \dots, x_{j-1}, x_j \pm 1, x_{j+1}, \dots, x_n)$$

- Case  $P_1; P_2$  :

$$f_P(x_0, \dots, x_n) := f_{P_2}(f_{P_1}(x_0, \dots, x_n))$$

- Case **loop**  $x_j$  **do**  $P'$  **end** :

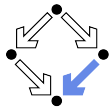
$$f_P(x_0, \dots, x_n) := g(x_j, x_0, \dots, x_n)$$

$$g(0, x_0, \dots, x_n) := (x_0, \dots, x_n)$$

$$g(m+1, x_0, \dots, x_n) := f_{P'}(g(m, x_0, \dots, x_n))$$

Thus the Ackermann function is also not primitive recursive.

# $\mu$ -Recursive Functions



A partial function over the natural numbers is  $\mu$ -recursive, if it

- is the constant null, successor, or a projection function,
- can be constructed from other  $\mu$ -recursive functions by composition or primitive recursion, or
- is a function  $h : \mathbb{N}^n \rightarrow_p \mathbb{N}$  defined as

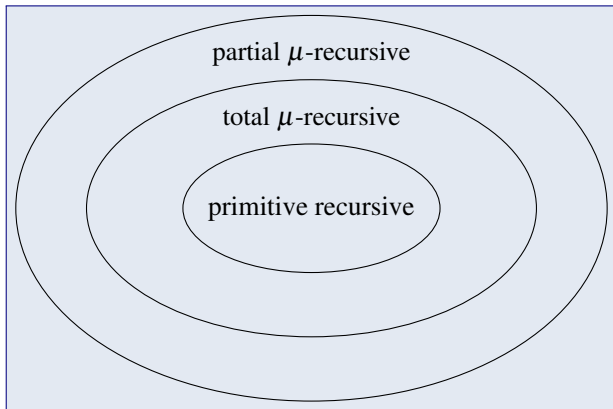
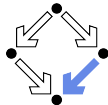
$$h(x_1, \dots, x_n) := (\mu f)(x_1, \dots, x_n)$$

with  $\mu$ -recursive  $f : \mathbb{N}^{n+1} \rightarrow_p \mathbb{N}$  and  $(\mu f) : \mathbb{N}^n \rightarrow_p \mathbb{N}$  defined as

$$(\mu f)(x_1, \dots, x_n) := \min \left\{ y \in \mathbb{N} \mid \begin{array}{l} f(y, x_1, \dots, x_n) = 0 \wedge \\ \forall z \leq y : (z, x_1, \dots, x_n) \in \text{domain}(f) \end{array} \right\}$$

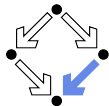
$(\mu f)(x_1, \dots, x_n)$  is the smallest  $y$  such that  $f(y, x_1, \dots, x_n) = 0$  (and  $f$  is defined for all  $z \leq y$ ); the result of  $h$  is undefined, if no such  $y$  exists.

# $\mu$ -Recursive Functions



Every primitive recursive function is a total  $\mu$ -recursive function; a  $\mu$ -recursive function may or may not be total.

# $\mu$ -Recursion and While Computability



**Theorem:** every  $\mu$ -recursive function is while computable and vice versa.

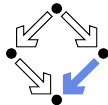
Proof  $\Rightarrow$ : we show that  $\mu$ -recursive  $h$  is while computable.

- If  $h$  is one of the basic functions or defined by composition or primitive recursion, it is clearly while computable.
- Case  $h(x_1, \dots, x_n) := (\mu f)(x_1, \dots, x_n)$

```
x0 := 0;  
y := f(x0, x1, ..., xn);  
while y ≠ 0 do  
  x0 := x0 + 1;  
  y := f(x0, x1, ..., xn)  
end
```

$\mu$ -recursion denotes unbounded iterative search.

# $\mu$ -Recursion and While Computability



Proof  $\Leftarrow$ : let  $h : \mathbb{N}^k \rightarrow_p \mathbb{N}$  be computable by while program  $P$  that uses variables  $x_0, \dots, x_n$ . Then  $h(x_1, \dots, x_k) := \text{var}_0(f_P(0, x_1, \dots, x_k, 0, \dots, 0))$  where  $\text{var}_i(x_0, \dots, x_n) := x_i$ . We show that  $f_P : \mathbb{N}^n \rightarrow_p \mathbb{N}^n$  is  $\mu$ -recursive by induction on  $P$ .

- If  $P$  is an assignment, a sequence, or a bounded loop, then  $f_P$  is clearly  $\mu$ -recursive.
- Case **while  $x_i$  do  $P'$  end** :

$$f_P(x_0, \dots, x_n) := g((\mu g_i)(x_0, \dots, x_n), x_0, \dots, x_n)$$

$$g_i : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$$

$$g_i(m, x_0, \dots, x_n) := \text{var}_i(g(m, x_0, \dots, x_n))$$

$$g(0, x_0, \dots, x_n) := (x_0, \dots, x_n)$$

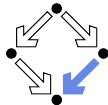
$$g(m+1, x_0, \dots, x_n) := f_{P'}(g(m, x_0, \dots, x_n))$$

- $g_i(m, x_0, \dots, x_n)$ : the value of program variable  $i$  after  $m$  iterations
- $g(m, x_0, \dots, x_n)$ : the values of all variables after  $m$  iterations.

Thus the Ackermann function is also  $\mu$ -recursive.



# Normal Form of a $\mu$ -Recursive Function



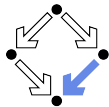
**Kleene's Normal Form Theorem:** every  $\mu$ -recursive function  $h$  can be defined in Kleene's normal form:

$$h(x_1, \dots, x_k) := f_2(x_1, \dots, x_k, (\mu g)(f_1(x_1, \dots, x_k)))$$

- $f_1, f_2, g$  are **primitive** recursive functions.

A single application of  $\mu$  is all that is needed.

# Normal Form of a $\mu$ -Recursive Function



We sketch the proof of Kleene's Normal Form Theorem.

Since  $h$  is  $\mu$ -recursive, it is computable by a while program in normal form

```
 $x_c := 1$ ; while  $x_c$  do ... end
```

with memory function

$$f_P(x_0, \dots, x_n) := g((\mu g_C)(\text{init}(x_0, \dots, x_n)), \text{init}(x_0, \dots, x_n))$$

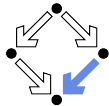
with primitive recursive  $g$  and  $g_C$  and  $\text{init}(x_0, \dots, x_c, \dots, x_n) := (x_0, \dots, 1, \dots, x_n)$ .

Thus we can define

$$\begin{aligned} h(x_1, \dots, x_k) &:= \text{var}_0(f_P(0, x_1, \dots, x_k, 0, \dots, 0)) \\ &= \text{var}_0(g((\mu g_C)(\text{init}(0, x_1, \dots, x_k, 0, \dots, 0)), \text{init}(0, x_1, \dots, x_k, 0, \dots, 0))) \\ &= f_2(x_1, \dots, x_k, (\mu g_C)(f_1(x_1, \dots, x_k))) \end{aligned}$$

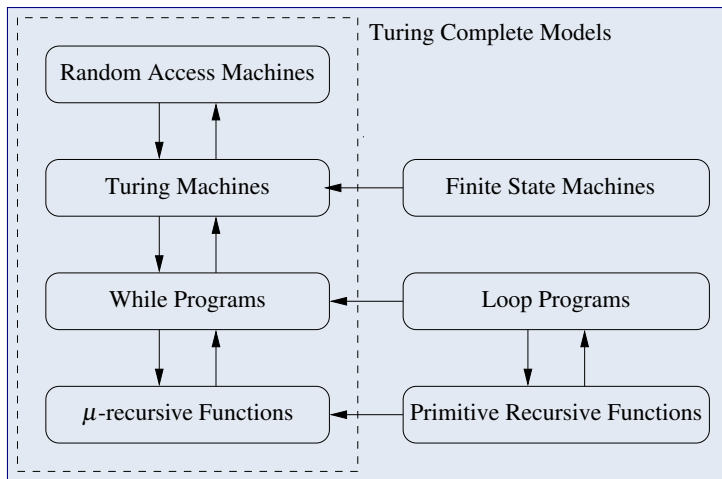
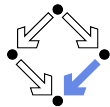
with primitive recursive

$$\begin{aligned} f_1(x_1, \dots, x_k) &:= \text{init}(0, x_1, \dots, x_k, 0, \dots, 0) \\ f_2(x_1, \dots, x_k, r) &:= \text{var}_0(g(r, \text{init}(0, x_1, \dots, x_k, 0, \dots, 0))) \end{aligned}$$



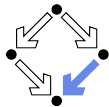
- 
1. Random Access Machines
  2. Loop and While Programs
  3. Primitive Recursive and  $\mu$ -recursive Functions
  - 4. Further Turing Complete Models**
  5. The Chomsky Hierarchy
  6. Real Computers

# The Big Picture So Far



We are going to sketch some more Turing complete models.

# Goto Programs



- A **goto program** has form

$$L_1 : P_1; L_2 : P_2; \dots; P_n : A_n$$

where  $L_k$  denotes a label and  $P_k$  an action:

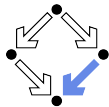
$$P ::= x_i := 0 \mid x_i := x_j + 1 \mid x_i := x_j - 1 \mid \text{if } x_i \text{ goto } L_j$$

- **Semantics**  $\llbracket P \rrbracket(k, m)$ :

- A partial function which maps the initial state  $(k, m)$  of  $P$ , consisting of program counter  $k \in \mathbb{N}$  and memory  $m : \mathbb{N} \rightarrow \mathbb{N}$ , to its final state (unless the program does not terminate).

$$\begin{aligned} \llbracket P \rrbracket(0, m) &:= m \\ \llbracket P = \dots; P_k : x_i := 0; \dots \rrbracket(k, m) &:= \llbracket P \rrbracket(k + 1, m[i \leftarrow 0]) \\ \llbracket P = \dots; P_k : x_i := x_j + 1; \dots \rrbracket(k, m) &:= \llbracket P \rrbracket(k + 1, m[i \leftarrow m[j] + 1]) \\ \llbracket P = \dots; P_k : x_i := x_j - 1; \dots \rrbracket(k, m) &:= \llbracket P \rrbracket(k + 1, m[i \leftarrow \max\{0, m[j] - 1\}]) \\ \llbracket P = \dots; P_k : \text{if } x_i \text{ goto } L_j; \dots \rrbracket(k, m) &:= \begin{cases} \llbracket P \rrbracket(k + 1, m), & \text{if } m(i) = 0 \\ \llbracket P \rrbracket(j, m), & \text{if } m(i) \neq 0 \end{cases} \end{aligned}$$

We have already seen how goto programs can be translated to while programs and vice versa; goto programs are therefore Turing complete.



- A  $\lambda$ -term  $T$ :

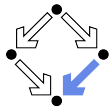
$$T ::= x_i \mid (T T) \mid (\lambda x_i. T)$$

- $x_i$ : a variable.
  - $(T T)$ : an **application**.
  - $(\lambda x_i. T)$ : an **abstraction**.
- **Reduction relation**  $\rightarrow$ :

$$((\lambda x_i. T_1) T_2) \rightarrow (T_1[x_i \leftarrow T_2])$$

- The result of the application of a function to an argument.
- **Reduction sequence**  $T_1 \rightarrow^* T_2$   
 $T_1 \rightarrow \dots \rightarrow T_2$ 
  - $T_2$  is in **normal form**, if no further reduction is possible.
- **Church-Rosser Theorem**: If  $T_1 \rightarrow^* T_2$  and  $T_1 \rightarrow^* T'_2$  such that both  $T_2$  and  $T'_2$  are in normal form, then  $T_2 = T'_2$ .

Every computable function can be represented by a  $\lambda$ -term.



How can we represent unbounded iteration (recursion)?

- Can define **fixpoint operator**  $Y$ :

$$(YF) \rightarrow^* (F(YF))$$

- $Y := (\lambda f.(\lambda x.(f(xx)))(\lambda x.(f(xx))))$
- Can translate **recursive function definition to  $\lambda$ -term**:

$$f(x) := \dots f(g(x)) \dots \rightsquigarrow f := YF$$

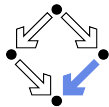
$$F := \lambda h.\lambda x.\dots h(g(x)) \dots$$

- $\lambda$ -term **behaves like recursive function**.

$$fa = (YF)a \rightarrow^* F(YF)a \rightarrow^* \dots (YF)(g(a)) \dots = \dots f(g(a)) \dots$$

**Formal basis of functional programming languages.**

# Rewriting Systems



- A **term rewriting system** is a set of rules of form

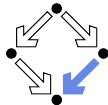
$$L \rightarrow R$$

- $L, R$ : terms such that  $L$  is not a variable and every variable that appears in  $R$  must also appear in  $L$ .
- **Rewriting Step**  $T \rightarrow T'$ :
  - There is some rule  $L \rightarrow R$  and a substitution  $\sigma$  (a mapping of variables to terms) such that
  - some subterm  $U$  of  $T$  matches the left hand side  $L$  of the rule under the substitution  $\sigma$ , i.e.,  $U = L\sigma$ ,
  - $T'$  is derived from  $T$  by replacing  $U$  with  $R\sigma$ , i.e. with the right hand side of the rule after applying the variable replacement.
- **Rewriting Sequence**  $T_1 \rightarrow^* T_2$   
 $T_1 \rightarrow \dots \rightarrow T_2$ 
  - $T_2$  is in **normal form**, if no further reduction is possible.

Every computable function can be represented by a term rewriting system.



# Rewriting Systems



- Term rewriting system:

$$f(x, f(y, z)) \rightarrow f(f(x, y), z)$$

$$f(x, e) \rightarrow x$$

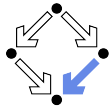
$$f(x, i(x)) \rightarrow e$$

- Rewriting sequence:

$$f(a, f(i(a), e)) \rightarrow f(f(a, i(a)), e) \rightarrow f(e, e) = e$$

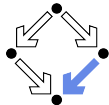
$$f(a, f(i(a), e)) \rightarrow f(a, i(a)) \rightarrow e$$

Rewriting systems can be also defined over strings and graphs; the later form the basis of tools for model driven architectures.



- 
1. Random Access Machines
  2. Loop and While Programs
  3. Primitive Recursive and  $\mu$ -recursive Functions
  4. Further Turing Complete Models
  - 5. The Chomsky Hierarchy**
  6. Real Computers

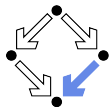
# Languages and Machines



- Regular languages:
  - Representable by regular expressions.
  - Recognizable by finite state machines.
- Recursively enumerable languages:
  - Representable by ... ?
  - Recognizable by Turing machines.
- Relationship:
  - Every regular language is recursively enumerable.
  - Every finite state machine can be simulated by a Turing machine.  
But not vice versa.

Are there any other interesting classes of languages and associated machine models and how do they relate to those above?

# Grammars



**Grammar**  $G = (N, \Sigma, P, S)$ :

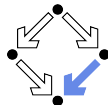
- $N$ : a finite set of **nonterminal symbols**,
- $\Sigma$ : a finite set of **terminal symbols** disjoint from  $N$ .  
 $N \cap \Sigma = \emptyset$
- $P$ : a finite set of **production rules** of form  $l \rightarrow r$  such that  
 $l \in (N \cup \Sigma)^* \circ N \circ (N \cup \Sigma)^*$   
 $r \in (N \cup \Sigma)^*$ 
  - $l$  and  $r$  consist of nonterminal and/or terminal symbols.
  - $l$  must contain at least one nonterminal symbol.
  - Multiple rules  $l \rightarrow r_1, l \rightarrow r_2, \dots, l \rightarrow r_n$  can be abbreviated:

$$l \rightarrow r_1 \mid r_2 \mid \dots \mid r_n$$

- $S$ : the **start symbol**.  
 $S \in N$

**Grammar**  $G$  describes a language over alphabet  $\Sigma$ .

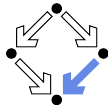
# The Language of a Grammar



Grammar  $G = (N, \Sigma, P, S)$ , words  $w, w_1, w_2 \in (N \cup \Sigma)^*$ .

- **Direct derivation**  $w_1 \Rightarrow w_2$  in  $G$ :  
 $w_1 = ulv$  and  $w_2 = urv$   
for  $u, v \in (N \cup \Sigma)^*$  and  $(l \rightarrow r) \in P$
- **Derivation**  $w_1 \Rightarrow^* w_2$  in  $G$ :  
 $w_1 \Rightarrow \dots \Rightarrow w_2$  in  $G$ .
- $w$  is a **sentential form** in  $G$ :  
 $S \Rightarrow^* w$
- $w$  is a **sentence** in  $G$ :
  - $w$  is a sentential form in  $G$  and  $w \in \Sigma^*$ .
- **Language**  $L(G)$  of  $G$ :  
 $L(G) := \{w \text{ is a sentence in } G\}$

The language of a grammar is the set of all words that consist only of terminal symbols and that are derivable from the start symbol.



# Example

- Grammar  $G = (N, \Sigma, P, S)$ :

$$N = \{S, A, B\}$$

$$\Sigma = \{a, b, c\}$$

$$P = \{S \rightarrow Ac, A \rightarrow aB, A \rightarrow BBb, B \rightarrow b, B \rightarrow ab\}$$

- Derivations:

$$S \Rightarrow Ac \Rightarrow aBc \Rightarrow abc$$

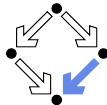
$$S \Rightarrow Ac \Rightarrow BBbc \Rightarrow abBbc \Rightarrow ababbc$$

- Language:

$$L(G) = \{abc, aabc, bbbc, babbc, abbbc, ababbc\}$$

This grammar defines a finite language.

# Example



- Grammar  $G = (N, \Sigma, P, S)$ :

$$N = \{S\}$$

$$\Sigma = \{(' , ') , '[' , '']\}$$

$$P = \{S \rightarrow \varepsilon \mid SS \mid [S] \mid (S)\}$$

- Derivations:

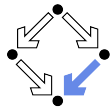
$$S \Rightarrow [S] \Rightarrow [SS] \Rightarrow [(S)S] \Rightarrow [(S)] \Rightarrow [(S)] \Rightarrow [(S)] \Rightarrow [(S)] \Rightarrow [(S)] \Rightarrow [(S)]$$

- Language: the “Dyck-Language”

$L(G)$  is the language of all expressions with matching pairs of parentheses “( )” and brackets “[ ]”

This grammar defines an infinite language.

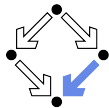
# Right-Linear Grammars and Regular Lang.



- Grammar  $G = (N, \Sigma, P, S)$  is **right linear** if each rule in  $P$  has form
  - $A \rightarrow \varepsilon, A \rightarrow a, A \rightarrow aB$with nonterminal symbols  $A, B \in N$  and terminal symbol  $a \in \Sigma$ .
- **Theorem:** The languages of right linear grammars are exactly the regular languages.
  - For every right linear grammar  $G$ , there exists a FSM  $M$  with  $L(M) = L(G)$  and vice versa.
  - Proof  $\Rightarrow$ : we construct from right linear grammar  $G$  a NFSM  $M$ . The states are the nonterminal symbols extended by a final state  $q_f$ ; the start state is the start symbol.
    - For every rule  $A \rightarrow \varepsilon$ , the state  $A$  becomes final.
    - For every rule  $A \rightarrow a$ , we add a transition  $\delta(A, a) = q_f$ .
    - For every rule  $A \rightarrow aB$ , we add a transition  $\delta(A, a) = B$ .
  - Proof  $\Leftarrow$ : we construct from DFSM  $M$  right linear grammar  $G$ . The nonterminal symbols are the states; the start symbol is the start state.
    - For every transition  $\delta(q, a) = q'$  we add a production rule  $q \rightarrow aq'$ .
    - For every final state  $q$ , we add a production rule  $q \rightarrow \varepsilon$ .



# Grammars and Recursively Enum. Lang.



**Theorem:** The languages of (unrestricted) grammars are exactly the recursively enumerable languages.

- Proof  $\Rightarrow$ : construct 2-tape nondeterministic  $M$  with  $L(M) = L(G)$ .

$M$  uses the second tape to construct some sentence of  $L(G)$ : it starts by writing  $S$  on the tape and then nondeterministically chooses some rule  $l \rightarrow r$  and applies it to some occurrence of  $l$  on the tape, replacing it by  $r$ . Then  $M$  checks whether the result equals the word on the first tape. If yes,  $M$  accepts the word, otherwise, it continues with another production rule.

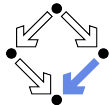
- Proof  $\Leftarrow$ : construct grammar  $G$  with  $L(G) = L(M)$ .

Sentential forms encode pairs  $(w, c)$  of input  $w$  and configuration  $c$  of  $M$ ; every form contains a non-terminal symbol such that by a rule application the current configuration is replaced by the successor configuration. The rules ensure that

- from the start symbol, every pair matching  $(w, c)$  of  $M$  can be derived;
- for every transition that moves  $c$  to  $c'$ , a rule is constructed that allows a derivation  $(w, c) \Rightarrow (w, c')$ ;
- if configuration  $c$  describes a final state from which no further transition is possible, the derivation  $(w, c) \Rightarrow w$  is possible.

Unrestricted grammars represent another Turing complete model.

# The Chomsky Hierarchy



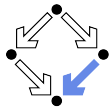
Noam Chomsky, 1959.

Type $i$	Grammar $G(i)$	Language $L(i)$	Machine $M(i)$
0	unrestricted	recursively enumerable	Turing machine
1	context sensitive	context sensitive	linear bounded automaton
2	context free	context free	push down automaton
3	right linear	regular	finite state machine

$L(i)$  is the set of languages of grammars  $G(i)$  and machines  $M(i)$ .

- For  $i > 0$ , the set of languages of type  $L(i)$  is a proper subset of the set of languages  $L(i-1)$ , i.e.  $L(i) \subset L(i-1)$ .
- For  $i > 0$ , every machine in  $M(i)$  can be simulated by a machine in  $M(i-1)$  (but not vice versa).

Grammars correspond to machine models.

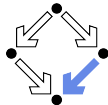


## Context Free Languages (Type 2)

- **Context free grammar  $G$ :** every rule has form  $A \rightarrow r$  with  $A \in N$ .
  - Independent of the context, any occurrence of  $A$  can be replaced.
- **Example:**  $L := \{a^i b^i \mid i \in \mathbb{N}\}$ 
  - $S \rightarrow \varepsilon \mid aSb$
  - $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$
- **Pushdown automaton  $M$ :** nondeterministic FSM with unbounded stack of symbols as “working memory”:
  - in every transition  $\delta(q, a, b) = (q', w)$ ,
  - $M$  reads the next input symbol  $a$  ( $a$  may be  $\varepsilon$ , i.e.,  $M$  may not read a symbol) and the symbol  $b$  on the top of the stack, and
  - replaces  $b$  by a (possibly empty) sequence  $w$  of symbols.

Most languages in computer science are context free.

# Generation of Syntax Analyzers



“Compiler generators” for the generation of syntax analyzers (parsers).

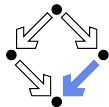
- Input: a (deterministic) context free grammar.

```
statement: assignment | conditional | whileloop | ... ;
whileloop: 'while' '(' valexp ')' statement ;
```

- Output: a (deterministic) push down automaton (as a program)

```
public final LoopStatement whileloop() throws ... {
    ...
    pushFollow(FOLLOW_valexp_in_whileloop1457);
    valexp();
    state._fsp--;
    if (state.failed) return value;
    ...
    pushFollow(FOLLOW_statement_in_whileloop1484);
    statement();
    state._fsp--;
    if (state.failed) return value;
    ...
}
```

# Context Sensitive Languages (Type 1)



## ■ Context sensitive grammar $G$ :

- in every rule  $l \rightarrow r$ , we have  $|l| \leq |r|$ , i.e., the length of left side  $l$  is less than or equal the length of right side  $r$ ,
- the rule  $S \rightarrow \varepsilon$  is only allowed, if the start symbol  $S$  does not appear on the right hand side of any rule.

## ■ Example: $L := \{a^i b^i c^i \mid i \in \mathbb{N}\}$

$S \rightarrow ABC \mid SABC$

$BA \rightarrow AB, CB \rightarrow BC, CA \rightarrow AC$

$AB \rightarrow ab, BC \rightarrow bc, Aa \rightarrow aa, bB \rightarrow bb, cC \rightarrow cc$

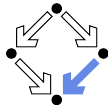
$\underline{S} \Rightarrow \underline{S}ABC \Rightarrow \underline{ABC}ABC \Rightarrow \underline{ABAC}BC \Rightarrow \underline{AABC}BC \Rightarrow \underline{AABB}CC$

$\Rightarrow \underline{Aab}BCC \Rightarrow \underline{aab}BC \underline{C} \Rightarrow \underline{aabb}c \underline{C} \Rightarrow aabbcc$

## ■ Linear bounded automaton $M$ : nondeterministic Turing machine with $k$ tapes (for some $k$ ).

- For input of length  $n$ , only the first  $n$  cells of each tape are used.
- The "space" used is a fixed multiple of the length of the input word.

Less practical importance.



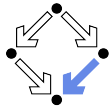
# Summary

---

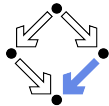
We have seen examples of each type of language.

- **Type 3:**  $\{(ab)^n \mid n \in \mathbb{N}\}$ 
  - Language is regular.
- **Type 2:**  $\{a^n b^n \mid n \in \mathbb{N}\}$ 
  - Language is context free.
- **Type 1:**  $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ 
  - Language is context sensitive.
- **Type 0:**  $\{a^{2^n} \mid n \in \mathbb{N}\}$ 
  - Language is recursively enumerable.

None of these languages of type  $i$  is also of type  $i + 1$ .



- 
1. Random Access Machines
  2. Loop and While Programs
  3. Primitive Recursive and  $\mu$ -recursive Functions
  4. Further Turing Complete Models
  5. The Chomsky Hierarchy
  - 6. Real Computers**



Are real computers Turing complete?

## ■ Hardware view:

- Finite number of digital elements and thus a finite number of states.
- Cannot simulate the infinite Turing machine tape.
- Cannot perform unbounded arithmetic.
- A computer is thus a **finite state machine** (i.e., not Turing complete).

View taken by model checkers.

## ■ Algorithm theory view:

- On demand, arbitrary much (e.g., virtual) memory may be added.
- Can thus simulate arbitrary large portion of the Turing machine tape.
- Can thus perform unbounded arithmetic.
- A computer is **Turing complete**.

View taken by algorithm design.

**A matter of the point of view respectively the goal of the modeling.**