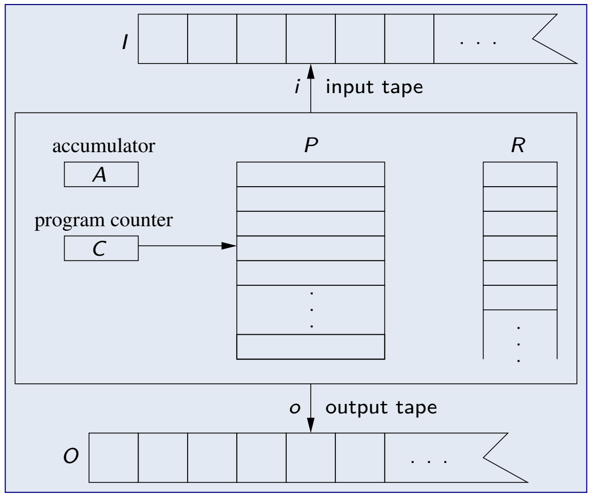
Turing Complete Computational Models

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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

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 $P[1], \ldots, P[m]$,
  - a memory consisting of a countably infinite sequence of registers $R[1], R[2], \ldots$, each of which can hold an arbitrary natural number.
- Execution:
  - Initially, $i = 0$, $o = 0$, $A = 0$, $C = 1$, $R[1] = R[2] = \ldots = 0$.
  - 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 $x_1, \ldots, x_n, 0$ and writes $x_n, \ldots, x_1$ using stack $R[2], \ldots, R[n+1]$.

RAMs versus Turing Machines

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

- RAM uses registers $R[1], \ldots, R[c-1]$ for its own purposes,
- stores in $R[c]$ the position of the tape head of the Turing machine,
- uses $R[c+1], R[c+2], \ldots$ as a virtual Turing machine tape.
  - Using "indirect addressing" operations `LOAD(n)` and `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 `LOAD(n)` and `STORE(n)` can be interpreted by self-modifying RASP code.

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

Loop Programs

- Loop Program P:

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

- Set $\{x_0, x_1, x_2, \ldots\}$ 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

- Semantics $[\![P]\!](m)$ maps the start memory $m : \mathbb{N} \to \mathbb{N}$ to the final memory after the termination of P:

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

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

$$\begin{array}{lcl}
[\![P]\!]^0(m) & := & m \\
[\![P]\!]^{n+1}(m) & := & [\![P]\!]([\![P]\!]^n(m))
\end{array}$$

A loop program denotes a function over memories.

Syntactic Abbreviations

- $x_i := x_j$

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

- $x_i := n$

  $x_i := 0; x_i := x_i + 1; x_i := x_i + 1; \ldots; x_i := x_i + 1$

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

  $x_t := 1;$ **loop** x_i **do** $x_t := 0;$ **end**;
  $x_e := 1;$ **loop** x_t **do** $x_e := 0;$ **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.

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

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

we have

$$[\![P]\!](m)(0) = f(x_1, \ldots, x_n)$$

When started in a state where $x_1, \ldots, x_n$ 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 $x_0 := x_1 + x_2$:

  $x_0 := x_1;$
  loop x_2 **do**
    $x_0 := x_0 + 1$
  end

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

  $x_0 := 0;$
  loop x_2 **do**
    $x_0 := x_0 + x_1$
  end

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

  $x_0 := 1;$
  loop x_2 **do**
    $x_0 := x_0 \cdot x_1$
  end

Natural number arithmetic is loop computable.

Arithmetic

- $x_0 := x_1 \cdot x_2$:

  $x_0 := 0;$
  loop x_2 **do**
    $x_0 := x_0 + x_1$
  end

  $\rightsquigarrow$

  $x_0 := 0;$
  loop x_2 **do**
    $x_0 := x_0;$
    loop x_1 **do**
      $x_0 := x_0 + 1$
    end
  end

Higher arithmetic needs multiply nested loops.

Beyond Exponentiation

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

- $a \uparrow^1 b = a^b$
  $a \uparrow^1 b = a \cdot a \cdot \ldots \cdot a$ (b times)
- $a \uparrow^2 b = a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}$ (b times)
  $a \uparrow^2 b = a \uparrow^1 a \uparrow^1 \ldots \uparrow^1 a$ (b times)
- $a \uparrow^3 b$:
  $a \uparrow^3 b = a \uparrow^2 a \uparrow^2 \ldots \uparrow^2 a$ (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: $g : \mathbb{N}^3 \to \mathbb{N}, g(a,b,n) := a \uparrow^{n+1} b$ 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:

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

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

Some arithmetic functions grow "too fast" to be loop computable.

While Programs

- While Program P:

$$P ::= \ldots \text{ (as for loop programs)} \mid \textbf{while } x_i \textbf{ do } P \textbf{ end}.$$

- Set $\{x_0, x_1, x_2, \ldots\}$ 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

- Semantics $[\![P]\!](m)$ maps start memory $m : \mathbb{N} \to \mathbb{N}$
  - to the final memory, if P terminates, and
  - to the special value $\bot$ (bottom), if P does not terminate.
- Semantics generalizes that of loop programs:

$$[\![P]\!](m) := \begin{cases} \bot & \text{if } m = \bot \\ [\![P]\!]'(m) & \text{else} \end{cases}$$
$$[\![\ldots]\!]'(m) := \ldots \text{(as for loop programs)}$$

- Semantics of unbounded iteration:

$$[\![\textbf{while } x_i \textbf{ do } P \textbf{ end}]\!]'(m) := \begin{cases} \bot & \text{if } L_i(P,m) \\ [\![P]\!]^{T_i(P,m)}(m) & \text{else} \end{cases}$$
$$L_i(P,m) :\Leftrightarrow \forall k \in \mathbb{N} : [\![P]\!]^k(m)(i) \neq 0$$
$$T_i(P,m) := \min\{k \in \mathbb{N} \mid [\![P]\!]^k(m)(i) = 0\}$$

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

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

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

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

the following holds:

- If $x_1, \ldots, x_n \in domain(f)$, then $[\![P]\!](m) : \mathbb{N} \to \mathbb{N}$ and
$$[\![P]\!](m)(0) = f(x_1, \ldots, x_n)$$
- If $x_1, \ldots, x_n \notin domain(f)$, then
$$[\![P]\!](m) = \bot$$

For a defined value of $f(x_1, \ldots, x_n)$, P terminates with this value in variable x_0. If $f(x_1, \ldots, x_n)$ 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 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
```

- $P_1, \ldots, P_n$ 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}
            P;
            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 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.

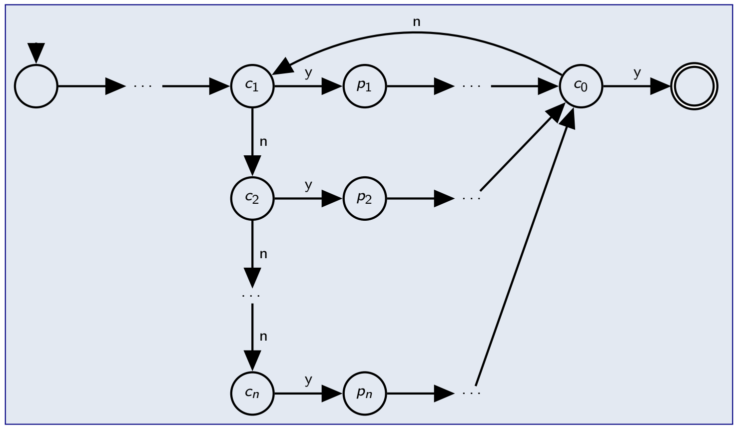
- x_0 holds initial tape content.
  - Determines initial configuration.
- Machine configuration (x_l, x_q, x_r):
  - x_q: the current state.
  - x_l: the tape left to the tape head,
  - x_r: the tape under/right to head.
- State x_q and symbol x_a under head determine the state transition.
  - If none 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 ∧ x_a = a_1 then
    P_1
  else if x_q = q_2 ∧ x_a = a_2 then
    P_2
  else if ... then
    ...
  else if x_q = q_n ∧ 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 $c_i, p_i, \ldots, c_0$.



Primitive Recursive Functions

The following functions over the natural numbers are primitive recursive:

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

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

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

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

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

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

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:

$$h(0,x_1\ldots,x_n) := f(x_1,\ldots,x_n)$$
$$h(y+1,x_1\ldots,x_n) := g(y,h(y,x_1,\ldots,x_n),x_1,\ldots,x_n)$$

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

$h(y,x_1\ldots x_n) :=$
case y **of**
$0:$ $f(x_1,\ldots,x_n)$
$z+1:$ $g(z,h(z,x_1,\ldots,x_n),x_1,\ldots,x_n)$

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

$h(0,x) = f(x)$
$h(1,x) = g(0,h(0,x),x) = g(0,f(x),x)$
$h(2,x) = g(1,h(1,x),x) = g(1,g(0,f(x),x),x)$
$h(3,x) = g(2,h(2,x),x) = g(2,g(1,g(0,f(x),x),x),x)$
$\ldots$
$h(y,x) = g(y-1,h(y-1,x),x) = g(y-1,g(y-2,\ldots,g(0,f(x),x),\ldots,x),x)$

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 ∧ 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.

$$min(x_1) := loop(x_1, x_1)$$

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

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

- $min(x_1) := loop(x_1, x_1)$ computes the value assigned to x_0 for input x_1 (2nd argument) after x_1 iterations of the **loop** (1st argument).
- $loop(x_2, x_1)$ computes the value assigned to x_0 for input x_1 after x_2 iterations of the **loop**.
- $if(x_2, x_0, x_1)$ 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 $min(5) = loop(5,5)$.

$loop(0,5) = 5$
$loop(1,5) = if(0, loop(0,5), 5) = if(0,5,5) = 5$
$loop(2,5) = if(1, loop(1,5), 5) = if(1,5,5) = 5$
$loop(3,5) = if(2, loop(2,5), 5) = if(2,5,5) = 5$
$loop(4,5) = if(3, loop(3,5), 5) = if(3,5,5) = 3$
$loop(5,5) = if(4, loop(4,5), 5) = if(4,3,5) = 3$

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 $loop(x_2, x_1) = x_0$ 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 ⇒: we show that primitive recursive function h is loop computable.

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

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

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

```
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 ⇐: let h be computable by loop program P. Let $f_P : \mathbb{N}^{n+1} \to \mathbb{N}^{n+1}$ 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 $x_i := k$:

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

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

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

- Case $P_1; P_2$:

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

- Case **loop** x_i **do** P' **end**:

$$f_P(x_0, \ldots, x_n) := g(x_i, x_0, \ldots, x_n)$$

$$g(0, x_0, \ldots, x_n) := (x_0, \ldots, x_n)$$
$$g(m+1, x_0, \ldots, x_n) := f_{P'}(g(m, x_0, \ldots, x_n))$$

Thus the Ackermann function is also not primitive recursive.

μ-Recursive Functions

A partial function over the natural numbers is μ-recursive, if it

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

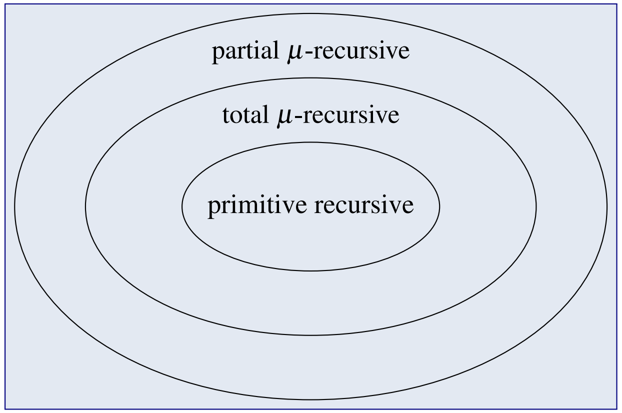
$$h(x_1, \dots, x_n) := (\mu f)(x_1, \dots, x_n)$$

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

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

$(\mu f)(x_1, \dots, x_n)$ 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.

μ-Recursive Functions



Every primitive recursive function is a total μ-recursive function; a μ-recursive function may or may not be total.

A μ-recursive Function

Consider particular sequences of numbers.

$$f^k(n) = \underbrace{f(f(f(\dots f(n))))}_{k \text{ applications of } f}$$

$$f(n) := \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n+1 & \text{otherwise} \end{cases}$$

$$\begin{aligned}
&f^0(10) = 10 \\
&f^1(10) = f(f^0(10)) = f(10) = 5 \\
&f^2(10) = f(f^1(10)) = f(5) = 16 \\
&f^3(10) = f(f^2(10)) = f(16) = 8 \\
&f^4(10) = f(f^3(10)) = f(8) = 4 \\
&f^5(10) = f(f^4(10)) = f(4) = 2 \\
&f^6(10) = f(f^5(10)) = f(2) = 1
\end{aligned}$$

Collatz Conjecture: for every $n \in \mathbb{N}$, $f^k(n) = 1$ for some $k \in \mathbb{N}$.

A μ-recursive Function

We define $C(n)$ to denote the smallest k with $f^k(n) = 1$.

$$\begin{aligned}
C(n) &:= (\mu D)(n) \\
D(k, n) &:= f^k(n) - 1 \\
f^k(n) &:= \begin{cases} n & \text{if } k = 0 \\ f(f^{k-1}(n)) & \text{otherwise} \end{cases}
\end{aligned}$$

(see lecture notes for completely formal definition)

Truth of conjecture is unknown: C may or may not be total (and may or may not be primitive recursive).

μ-Recursion and While Computability

Theorem: every μ-recursive function is while computable and vice versa.

Proof $\Rightarrow$: we show that μ-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 $h(x_1,\ldots,x_n) := (\mu f)(x_1,\ldots,x_n)$

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

μ-recursion denotes unbounded iterative search.

μ-Recursion and While Computability

Proof $\Leftarrow$: let $h : \mathbb{N}^k \to_p \mathbb{N}$ be computable by while program P with variables $x_0,\ldots,x_n$. Then $h(x_1,\ldots,x_k) := var_0(f_P(0,x_1,\ldots,x_k,0,\ldots,0))$ where $var_i(x_0,\ldots,x_n) := x_i$. We show that $f_P : \mathbb{N}^{n+1} \to_p \mathbb{N}^{n+1}$ is μ-recursive by induction on P.

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

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

$$g_i : \mathbb{N}^{n+1} \to \mathbb{N}$$
$$g_i(m,x_0,\ldots,x_n) := var_i(g(m,x_0,\ldots,x_n))$$

$$g(0,x_0,\ldots,x_n) := (x_0,\ldots,x_n)$$
$$g(m+1,x_0,\ldots,x_n) := f_{P'}(g(m,x_0,\ldots,x_n))$$

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

Thus the Ackermann function is also μ-recursive.

Normal Form of a μ-Recursive Function

Kleene's Normal Form Theorem: every μ-recursive function h can be defined in Kleene's normal form:

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

- f_1, f_2, g are primitive recursive functions.

A single application of μ is all that is needed.

Normal Form of a μ-Recursive Function

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

Since h is μ-recursive, it is computable by a while program in normal form

$x_c := 1;$ **while** xc **do** ... **end**

with memory function

$$f_P(x_0,\ldots,x_n) := g((\mu g_c)(init(x_0,\ldots,x_n)),init(x_0,\ldots,x_n))$$

with primitive recursive g and g_c and $init(x_0,\ldots,x_c,\ldots,x_n) := (x_0,\ldots,1,\ldots,x_n)$.

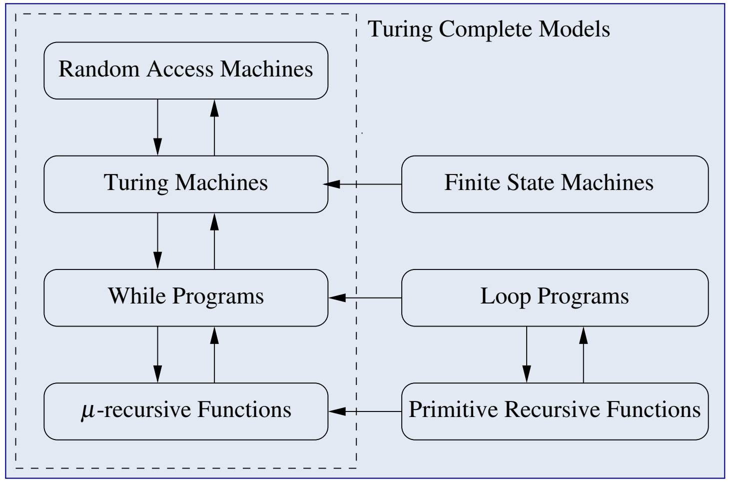
Thus we can define

$$\begin{aligned} h(x_1,\ldots,x_k) &:= var_0(f_P(0,x_1,\ldots,x_k,0,\ldots,0)) \\ &= var_0(g((\mu g_c)(init(0,x_1,\ldots,x_k,0,\ldots,0)),init(0,x_1,\ldots,x_k,0,\ldots,0))) \\ &= f_2(x_1,\ldots,x_k,(\mu g_c)(f_1(x_1,\ldots,x_k))) \end{aligned}$$

with primitive recursive

$$f_1(x_1,\ldots,x_k) := init(0,x_1,\ldots,x_k,0,\ldots,0)$$
$$f_2(x_1,\ldots,x_k,r) := var_0(g(r,init(0,x_1,\ldots,x_k,0,\ldots,0)))$$

The Big Picture So Far



We are going to sketch some more Turing complete models.

Goto Programs

- A goto program has form
  $L_1 : P_1; L_2 : P_2; \ldots ; P_n : A_n$
  where L_k denotes a label and P_k an action:
  $P ::= x_i := 0 \mid x_i := x_j + 1 \mid x_i := x_j - 1 \mid \textbf{if } x_i \textbf{ goto } L_j$
- Semantics $[\![P]\!](k,m)$:
  - A partial function which maps the initial state (k,m) of P, consisting of program counter $k \in \mathbb{N}$ and memory $m : \mathbb{N} \to \mathbb{N}$, to its final state (unless the program does not terminate).

$$\begin{array}{lcl}
[\![P]\!](0,m) & := & m \\
[\![P = \ldots; P_k : x_i := 0; \ldots]\!](k,m) & := & [\![P]\!](k+1, m[i \leftarrow 0]) \\
[\![P = \ldots; P_k : x_i := x_j + 1; \ldots]\!](k,m) & := & [\![P]\!](k+1, m[i \leftarrow m[j]+1]) \\
[\![P = \ldots; P_k : x_i := x_j - 1; \ldots]\!](k,m) & := & [\![P]\!](k+1, m[i \leftarrow \max\{0, m[j]-1\}]) \\
[\![P = \ldots; P_k : \textbf{if } x_i \textbf{ goto } L_j; \ldots]\!](k,m) & := & \begin{cases} [\![P]\!](k+1,m), \text{ if } m(i) = 0 \\ [\![P]\!](j,m), \text{ if } m(i) \neq 0 \end{cases}
\end{array}$$

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

λ-Calculus

- A λ-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 $\to$:

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

  - The result of the application of a function to an argument.
- Reduction sequence $T_1 \to^* T_2$
  $T_1 \to \ldots \to T_2$
  - T_2 is in normal form, if no further reduction is possible.
- Church-Rosser Theorem: If $T_1 \to^* T_2$ and $T_1 \to^* T_2'$ 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 λ-term.

λ-Calculus

How can we represent unbounded iteration (recursion)?

- Can define fixpoint operator Y:

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

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

$$f(x) := \ldots f(g(x)) \ldots \;\rightsquigarrow\; f := YF$$

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

- λ-term behaves like recursive function.

$$fa = (YF)a \to^* F(YF)a \to^* \ldots (YF)(g(a)) \ldots = \ldots f(g(a)) \ldots$$

Formal basis of functional programming languages.

Rewriting Systems

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

$$L \to R$$

  - L, R: terms such that L is not a variable and every variable that appears in R must also appear in L.
- Rewriting Step $T \to T'$:
  - There is some rule $L \to R$ and a substitution σ (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 σ, 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 $T_1 \to^* T_2$
  $T_1 \to \ldots \to T_2$
  - 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:

$$f(x, f(y, z)) \to f(f(x, y), z)$$
$$f(x, e) \to x$$
$$f(x, i(x)) \to e$$

- Rewriting sequence:

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

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

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,
- Σ: a finite set of terminal symbols disjoint from N.

  $N \cap \Sigma = \emptyset$
- P: a finite set of production rules of form $l \to r$ 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 $l \to r_1, l \to r_2, \ldots, l \to r_n$ can be abbreviated:

$$l \to r_1 \mid r_2 \mid \ldots \mid r_n$$

- S: the start symbol.

  $S \in N$

Grammar G describes a language over alphabet Σ.

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 $(l \to r) \in P$
- Derivation $w_1 \Rightarrow^* w_2$ in G:

  $w_1 \Rightarrow \ldots \Rightarrow w_2$ 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 $G = (N, \Sigma, P, S)$:

$$N = \{S, A, B\}$$
$$\Sigma = \{a, b, c\}$$
$$P = \{S \to Ac,\ A \to aB, A \to BBb, B \to b, B \to ab\}$$

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

$$N = \{S\}$$
$$\Sigma = \{`(', `)', `[', `]'\}$$
$$P = \{S \to \varepsilon \mid SS \mid [S] \mid (S)\}$$

- Derivations:

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

- 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
  - $A \to \varepsilon$, $A \to a$, $A \to aB$

  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 $A \to \varepsilon$, the state A becomes final.
    - For every rule $A \to a$, we add a transition $\delta(A, a) = q_f$.
    - For every rule $A \to aB$, 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 $q \to aq'$.
    - For every final state q, we add a production rule $q \to \varepsilon$.

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 $l \to r$ 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 matching pair (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 $A \to r$ with $A \in N$.
  - Independent of the context, any occurrence of A can be replaced.
- Example: $L := \{a^i b^i \mid i \in \mathbb{N}\}$

  $S \to \varepsilon \mid aSb$

  $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$
- 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 ε, 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 $l \to r$, we have $|l| \le |r|$, i.e., the length of left side l is less than or equal the length of right side r,
  - the rule $S \to \varepsilon$ is only allowed, if the start symbol S does not appear on the right hand side of any rule.
- Example: $L := \{a^i b^i f^i \mid i \in \mathbb{N}\}$

  $S \to \varepsilon \mid T, T \to ABC \mid TABC$

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

  $AB \to ab, bC \to bc, Aa \to aa, bB \to bb, cC \to cc$

  $$\underline{S} \Rightarrow \underline{T} \Rightarrow \underline{T}ABC \Rightarrow ABC\underline{A}BC \Rightarrow A\underline{BA}CBC \Rightarrow AAB\underline{CB}C \Rightarrow A\underline{AB}BCC$$

  $$\Rightarrow \underline{Aa}bBCC \Rightarrow aa\underline{bB}CC \Rightarrow aab\underline{bC}C \Rightarrow aabb\underline{cC} \Rightarrow aabbcc$$
- 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: $\{a^n b^n f^n \mid n \in \mathbb{N}\}$
  - Language is context-sensitive.
- Type 0: $\{a^i b^j f^k \mid k = ack(i, j)\}$
  - Language is recursively enumerable (also recursive).

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

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.