We now know the syntax + semantics of (pure) logic programming.
In this section, we show that LP is a universal (= Turing-complete) programming language.
In other words, it is as expressive as Java, C, Haskell...
Thus, every computable function can also be computed by a logic program.

What are the computable functions?

Turing: Turing Machines
Church: Lambda Terms
Kleene: μ-recursive Functions

Yield the same set of functions

Church’s Thesis: every formalism to define computability yields the same set of computable functions

Here, we show that every μ-recursive function can be computed by a logic program.

We only regard functions on N.

μ-recursive functions are defined by 6 rules to characterize all computable functions.
Rules 1-3 define basic functions.
Rules 4-6 allow us to construct new computable functions from existing ones.

**Def 4.2.1 (μ-recursive Functions)**

The set of μ-recursive functions is the smallest set of functions with:

1. For all \( n \in \mathbb{N} \), the function \( \text{null}_n : \mathbb{N}^n \to \mathbb{N} \) with \( \text{null}_n (k_1, \ldots, k_n) = 0 \) is μ-recursive.

2. The successor function \( \text{succ} : \mathbb{N} \to \mathbb{N} \) with \( \text{succ} (k) = k + 1 \) is μ-recursive.

3. For all \( n \geq 1 \) and all \( 1 \leq i \leq n \), the projection function \( \text{proj}_{n,i} : \mathbb{N}^n \to \mathbb{N} \) with \( \text{proj}_{n,i} (k_1, \ldots, k_n) = k_i \) is μ-recursive.

4. The μ-recursive functions are closed under composition: For all \( n \geq 1 \) and all \( n \geq 0 \) we have:
   - If \( f : \mathbb{N}^n \to \mathbb{N} \), \( f_1, \ldots, f_m : \mathbb{N}^n \to \mathbb{N} \) are μ-recursive,
   - then the following function \( g : \mathbb{N}^n \to \mathbb{N} \) is also μ-recursive:
     \[
g(k_1, \ldots, k_n) = f \left( f_n (k_1, \ldots, k_n), \ldots, f_m (k_1, \ldots, k_n) \right)
\]

5. The μ-recursive functions are closed under primitive recursion: For all \( n \geq 0 \) we have:
primitive recursion: For all $n \geq 0$ we have:

If $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$ are $\mu$-recursive, then the following fact $h : \mathbb{N}^{n+2} \to \mathbb{N}$ is also $\mu$-recursive:

\[
h(k_0, \ldots, k_n, 0) = f(k_0, \ldots, k_n)
\]
\[
h(k_0, \ldots, k_n, k + 1) = g(k_0, \ldots, k_n, k, h(k_0, \ldots, k_n, k))
\]

6. The $\mu$-recursive functions are closed under (unbounded) minimization:

If $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is $\mu$-recursive, then the following fact $g : \mathbb{N}^n \to \mathbb{N}$ is also $\mu$-recursive:

\[
g(k_0, \ldots, k_n) = k \quad \text{iff} \quad f(k_0, \ldots, k_n, k) = 0 \quad \text{and}
\]
\[
\text{for all } 0 \leq k' < k, \quad f(k_0, \ldots, k_n, k') \text{ is defined and } f(k_0, \ldots, k_n, k') > 0.
\]

If there is no such $k$, then $g(k_0, \ldots, k_n)$ is undefined.

In an imperative language, $g$ is easy to implement: initialize $k$ with 0 and increase it repeatedly until $f(k, \ldots, k, k) = 0$.

The class of functions that can be constructed with the principles 1-5 are the primitive recursive
The principles 1-5 are the primitive recursive functions.

There are computable functions that are not primitive recursive:

- partial functions
- total functions like the Ackermann function

**Ex. 4.22.** The addition function \( \text{plus} : \mathbb{N}^2 \rightarrow \mathbb{N} \) is primitive recursive:

\[
\begin{align*}
\text{f}(x, y, z) &= \text{succ} \left( \text{proj}_3 \left( x, y, z \right) \right) \\
\text{plus}(x, 0) &= \text{proj}_2(x) \\
\text{plus}(x, y+1) &= f(x, y, \text{plus}(x, y))
\end{align*}
\]

\[
\begin{align*}
\text{f}(x, y, z) &= 2 + 1 \\
\text{plus}(x, 0) &= x \\
\text{plus}(x, y+1) &= \text{plus}(x, y) + 1
\end{align*}
\]

The multiplication function \( \text{times} : \mathbb{N}^2 \rightarrow \mathbb{N} \) is also primitive recursive:

\[
\begin{align*}
\text{g}(x, y, z) &= \text{plus} \left( \text{proj}_3, \left( x, y, z \right), \text{proj}_2, \left( x, y, z \right) \right) \\
\text{times}(x, 0) &= \text{null}(x) \\
\text{times}(x, y+1) &= g \left( x, y, \text{times}(x, y) \right)
\end{align*}
\]

\[
\begin{align*}
\text{g}(x, y, z) &= x + z \\
\text{times}(x, 0) &= 0 \\
\text{times}(x, y+1) &= x + \text{times}(x, y)
\end{align*}
\]

The predecessor function \( \text{p} : \mathbb{N} \rightarrow \mathbb{N} \) is primitive recursive, where \( \text{p}(0) = 0 \) and \( \text{p}(x+1) = x \).

\[
\text{p}(0) = \text{null}_0
\]
\( p(x+1) = \text{proj}_{2,1} (x, p(x)) \)

The subtraction function \( \text{minus} : \mathbb{N}^2 \to \mathbb{N} \) is defined recursively, where \( \text{minus} (x, y) = 0 \) if \( x \leq y \) and \( \text{minus} (x, y) = x - y \) otherwise.

More precisely:

\[
\text{minus} (x, y) = \underbrace{p(p(\ldots p(x)))}_{y \text{ times}}
\]

\[
h(x, y, z) = p(\text{proj}_{3,2} (x, y, z))
\]

\[
\text{minus} (x, 0) = \text{proj}_{1,1} (x) \leftarrow x
\]

\[
\text{minus} (x, y+1) = h(x, y, \text{minus} (x, y)) \leftarrow p(\text{minus} (x, y))
\]

Finally, we show that \( \text{div} : \mathbb{N}^2 \to \mathbb{N} \) is \( \mu \)-recursive, where

\[
\text{div} (x, y) = \left\lfloor \frac{x}{y} \right\rfloor \quad \text{if} \quad y \neq 0
\]

\[
\text{div} (0, 0) = 0
\]

\[
\text{div} (x, 0) \text{ undefined if} \quad x > 0
\]

Idea:

\[
\frac{x}{y} = z \quad \text{if} \quad x = y \cdot z
\]

\[
\text{iff} \quad x - y \cdot z = 0
\]

\[
\overset{\text{i} (x, y, z)}{\text{i} (x, y, z)}
\]

We search for the smallest \( z \) where \( i (x, y, z) \)

is \( 0 \).

\[
\text{div} (x, y) = z \quad \text{iff} \quad i (x, y, z) = 0 \quad \text{and}
\]
for all $0 \leq t' \leq t$, $i(x, y, t')$ is defined and $i(x, y, t') > 0$

Here, $i$ is primitive recursive:

$$i(x, y, t) = \text{min} \left( \text{proj}_{3\alpha} (x, y, t), j(x, y, t) \right) \leq x - y \cdot t$$

$$j(x, y, t) = \text{times} \left( \text{proj}_{3\beta} (x, y, t), \text{proj}_{3\gamma} (x, y, t) \right) \leq y \cdot t$$

The $\mu$-recursive functions are exactly the computable functions. To show that logic programming is Turing-complete, we have to show that every $\mu$-rec. fun. can be computed by a logic program.

We first have to make clear when a log. prog. "computes" a function on nat. numbers.

Problems:

@. Logic prog. operate on terms, not on numbers.

@. Logic prog. implement relations/predicates, not functions.

Solution for @.

Represent numbers by terms over $0 \in \Sigma_0$ and $s \in \Sigma_n$.

$$0 \equiv 0$$
\[ 1 \triangleq s(0) \\
2 \triangleq s(s(0)) \]

Solution for (b):

We compute a function \( f : \mathbb{N}^n \to \mathbb{N} \) by a predicate \( f \) of arity \( n+1 \).

**Def 4.2.3 (Computing arithmetic functions by Logic Programs)**

- Every number \( k \in \mathbb{N} \) is represented by the term \( k \in \Gamma \left( \Sigma, \Delta \right) \) with \( k = \underbrace{s(\ldots s(0)\ldots)}_{\text{k times}} \).
  - \( 0 \in \Sigma_0 \) and \( s \in \Sigma_1 \).
  - \( 0 = 0 \)
  - \( 1 = s(0) \)
  - \( 2 = s(s(0)) \)

- A logic prog \( \mathcal{S} \) over \( (\Sigma, \Delta) \) computes a function \( f : \mathbb{N}^n \to \mathbb{N} \) iff there is a predicate symbol \( f \in \Delta_{n+1} \) such that \( f(k_1, \ldots, k_n) = k \) iff \( \mathcal{S} \vdash f(k_1, \ldots, k_n, k) \).

Then one can use the LP \( \mathcal{S} \) to compute \( f \)'s
The following logic program implements the functions from Ex. 4.2.2.

\[ ? - f(k_1, \ldots, k_n, X). \]
\[ X = Y \]

**Ex. 4.2.4** The following logic program implements the functions from Ex. 4.2.2.

\[ \text{plus}(X, 0, X). \]
\[ \text{plus}(X, s(Y), s(Z)) :- \text{plus}(X, Y, Z). \]
\[ \text{times}(X, 0, 0). \]
\[ \text{times}(X, s(Y), Z) :- \text{times}(X, Y, U), \text{plus}(U, X, Z). \]
\[ P(0, 0). \]
\[ P(s(X), X). \]
\[ \text{minus}(X, 0, X). \]
\[ \text{minus}(X, s(Y), Z) :- \text{minus}(X, Y, U), P(U, Z). \]
\[ \text{div}(0, Y, 0). \]
\[ \text{div}(s(X), s(Y), s(Z)) :- \text{minus}(X, Y, U), \text{div}(U, s(Y), Z). \]

This computes a partial function \( \text{div} \), because
\[ ? - \text{div}(1, 0, Z). \]
fails. So we can implement partiality by failure or by non-termination.

\textbf{Thm 4.2.5 (Universality of LP)}

Every \( \mu \)-recursive \textit{fact} can be computed by a logic program.

\textbf{Proof:} We prove the \textit{thm.} by induction on the construction of the class of \( \mu \)-rec. facts.

1. The function \( \text{null} \) can be computed by the following LP:
   \[
   \text{null} \equiv (X_1, \ldots, X_n, 0).
   \]

2. The fact. \( \text{succ} \) is implemented by
   \[
   \text{succ} \equiv (X, s(X)).
   \]

3. The fact. \( \text{proj}_{n,i} \) is implemented by
   \[
   \text{proj}_{n,i} \equiv (X_1, \ldots, X_n, X_i).
   \]

4. Composition can also be realized by a LP.
   By the ind. hyp. there is a LP with \( \bar{f} \in \Delta_{m+1}, \)
   \( f_1, \ldots, f_m \in \Delta_{n+1} \) that computes \( f, f_1, \ldots, f_m. \)
   We extend this LP by the following rule:
   \[
   g(X_1, \ldots, X_n, z) \equiv f_1(X_1, \ldots, X_n, Y_1), \ldots, f_m(X_1, \ldots, X_n, Y_m), f(Y_1, \ldots, Y_m, z).
   \]

5. \textbf{Prim. Recursion}
   By the ind. hyp there is a LP with \( f \in \Delta_{m+3}, g \in \Delta_{n+3} \)
   that computes \( f \) and \( g. \) We extend it by the following
Two clauses:

\[ h(X_0, \ldots, X_n, 0, \varepsilon) := f(X_0, \ldots, X_n, \varepsilon). \]

\[ h(X_0, \ldots, X_n, s(X), \varepsilon) := h(X_0, \ldots, X_n, X, Y), \]
\[ g(X_0, \ldots, X_n, X, Y, \varepsilon). \]

6. Unbounded Minimization

By the ind. hyp., there is a LP with \( f \in \Delta_{n+2} \) which computes \( f \). We extend it by the following rules.

Here \( f'(X_0, \ldots, X_n, Y, \varepsilon) \) holds iff

\[ f(X_0, \ldots, X_n, \varepsilon) = 0 \quad \text{and} \]
\[ \text{for all } Y \leq \varepsilon' < \varepsilon \text{ we have } f(X_0, \ldots, X_n, \varepsilon') > 0 \]

\[ g(X_0, \ldots, X_n, \varepsilon) := f'(X_0, \ldots, X_n, 0, \varepsilon). \]

\[ f'(X_0, \ldots, X_n, Y, Y) := f(X_0, \ldots, X_n, Y, 0). \]

\[ f'(X_0, \ldots, X_n, Y, \varepsilon) := f(X_0, \ldots, X_n, s(Y), Y), \]
\[ f'(X_0, \ldots, X_n, s(Y), \varepsilon). \]

\[ \square \]

Ex 4.26: If one uses the construction of the above proof, then one would obtain the following LP for plus from the \( \mu \)-recursive plus-function in Ex 4.2.2.
\[\text{proj}_{3,3}(X, Y, Z, W)\].
\[\text{succ}(X, s(X))\].
\[f(X, Y, Z, W) :\text{ proj}_{3,3}(X, Y, Z, U), \text{ succ}(U, V)\].
\[\text{proj}_{1,n}(X, X)\].
\[\text{plus}(X, 0, U) :\text{ proj}_{1,n}(X, U)\].
\[\text{plus}(X, s(Y), U) :\text{ plus}(X, Y, Z), f(X, Y, Z, U)\].

If one applies the construction from the proof to the \(\mu\)-recursive fact, \text{div}, then one gets:
\[\text{div}(X_1, X_2, Z) :\text{ i'}(X_1, X_2, 0, Z)\].
\[\text{i'}(X_1, X_2, Y, Y) :\text{ i}(X_1, X_2, Y, 0)\].
\[\text{i'}(X_1, X_2, Y, Z) :\text{ i}(X_1, X_2, Y, s(U))\].
\[\text{i'}(X_1, X_2, s(Y), Z)\].

: clauses for \text{i}.