Recall :

Turing Machines

are specified by:

- Q, finite set of machine states
- Σ, finite set of tape symbols (disjoint from Q) containing distinguished symbols $\triangleright$ (left endmarker) and $\sqcup$ (blank)
- $s \in Q$, an initial state
- $\delta \in (Q \times \Sigma) \to (Q \cup \{\mathtt{acc}, \mathtt{rej}\}) \times \Sigma \times \{L, R, S\}$, a transition function, satisfying:

  for all $q \in Q$, there exists $q' \in Q \cup \{\mathtt{acc}, \mathtt{rej}\}$ with $\delta(q, \triangleright) = (q', \triangleright, R)$
  (i.e. left endmarker is never overwritten and machine always moves to the right when scanning it)

We've seen that a Turing machine's computation can be implemented by a register machine.

The converse holds: the computation of a register machine can be implemented by a Turing machine.

To make sense of this, we first have to fix a tape representation of RM configurations and hence of numbers and lists of numbers...

Tape encoding of lists of numbers

Definition. A tape over $\Sigma = \{\triangleright, \sqcup, 0, 1\}$ codes a list of numbers if precisely two cells contain 0 and the only cells containing 1 occur between these.

Such tapes look like:

$$\triangleright \sqcup \cdots \sqcup 0 \underbrace{1 \cdots 1}_{n_1} \sqcup \underbrace{1 \cdots 1}_{n_2} \sqcup \cdots \sqcup \underbrace{1 \cdots 1}_{n_k} 0 \underbrace{\sqcup \cdots}_{\text{all } \sqcup\text{'s}}$$

which corresponds to the list $[n_1, n_2, \ldots, n_k]$.

E.g. $\triangleright 0 \sqcup \sqcup \sqcup 11 \sqcup 1 \sqcup 0 \sqcup \sqcup \ldots$

codes the list

Tape encoding of lists of numbers

Definition. A tape over $\Sigma = \{\triangleright, \sqcup, 0, 1\}$ codes a list of numbers if precisely two cells contain 0 and the only cells containing 1 occur between these.

Such tapes look like:

$$\triangleright \sqcup \cdots \sqcup 0 \underbrace{1 \cdots 1}_{n_1} \sqcup \underbrace{1 \cdots 1}_{n_2} \sqcup \underset{\cdots}{\cdots} \sqcup \underbrace{1 \cdots 1}_{n_k} 0 \underbrace{\sqcup \cdots}_{\text{all } \sqcup\text{'s}}$$

which corresponds to the list $[n_1, n_2, \ldots, n_k]$.

E.g. $\triangleright 0 \sqcup \sqcup \sqcup 11 \sqcup 1 \sqcup 0 \sqcup \sqcup \ldots$

codes the list $[0,0,0,2,1,0]$

Recall:

Computable functions

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is (register machine) computable if there is a register machine M with at least $n+1$ registers R_0, R_1, . . . , R_n (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$,

> the computation of M starting with $\mathrm{R}_0 = 0$, $\mathrm{R}_1 = x_1$, . . . , $\mathrm{R}_n = x_n$ and all other registers set to 0, halts with $\mathrm{R}_0 = y$

if and only if $f(x_1, \ldots, x_n) = y$.

Turing computable function

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is Turing computable if and only if there is a Turing machine M with the following property:

> Starting M from its initial state with tape head on the left endmarker of a tape coding $[0, x_1, \ldots, x_n]$, M halts if and only if $f(x_1, \ldots, x_n)\downarrow$, and in that case the final tape codes a list (of length ≥ 1) whose first element is y where $f(x_1, \ldots, x_n) = y$.

Theorem. A partial function is Turing computable if and only if it is register machine computable.

Proof (sketch). We've seen how to implement any TM by a RM. Hence

f TM computable implies f RM computable.

For the converse, one has to implement the computation of a RM in terms of a TM operating on a tape coding RM configurations. To do this, one has to show how to carry out the action of each type of RM instruction on the tape. It should be reasonably clear that this is possible in principle, even if the details (omitted) are tedious.

Notions of computability

- Church (1936): λ-calculus [see later]
- Turing (1936): Turing machines.

Turing showed that the two very different approaches determine the same class of computable functions. Hence:

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

Notions of computability

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

Further evidence for the thesis:

- Gödel and Kleene (1936): partial recursive functions
- Post (1943) and Markov (1951): canonical systems for generating the theorems of a formal system
- Lambek (1961) and Minsky (1961): register machines
- Variations on all of the above (e.g. multiple tapes, non-determinism, parallel execution. . .)

All have turned out to determine the same collection of computable functions.

Notions of computability

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

In rest of the course we'll look at

- Gödel and Kleene (1936): partial recursive functions
  ($\rightsquigarrow$ branch of mathematics called recursion theory)
- Church (1936): λ-calculus
  ($\rightsquigarrow$ branch of CS called functional programming)

Aim

A more abstract, machine-independent description of the collection of computable partial functions than provided by register/Turing machines:

> they form the smallest collection of partial functions containing some basic functions and closed under some fundamental operations for forming new functions from old—composition, primitive recursion and minimization.

The characterization is due to Kleene (1936), building on work of Gödel and Herbrand.

Basic functions

- Projection functions, $\texttt{proj}_i^n \in \mathbb{N}^n \to \mathbb{N}$:

$$\texttt{proj}_i^n(x_1, \ldots, x_n) \triangleq x_i$$

- Constant functions with value $\mathbf{0}$, $\texttt{zero}^n \in \mathbb{N}^n \to \mathbb{N}$:

$$\texttt{zero}^n(x_1, \ldots, x_n) \triangleq \mathbf{0}$$

- Successor function, $\texttt{succ} \in \mathbb{N} \to \mathbb{N}$:

$$\texttt{succ}(x) \triangleq x + \mathbf{1}$$

Basic functions

are all RM computable:

- Projection $\texttt{proj}_i^n$ is computed by

$$\texttt{START} \rightarrow \boxed{R_0 ::= R_i} \rightarrow \texttt{HALT}$$

- Constant $\texttt{zero}^n$ is computed by

$$\texttt{START} \rightarrow \texttt{HALT}$$

- Successor $\texttt{succ}$ is computed by

$$\texttt{START} \rightarrow R_1^+ \rightarrow \boxed{R_0 ::= R_1} \rightarrow \texttt{HALT}$$

Composition

Composition of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ is the partial function $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in \mathbb{N}$

$$f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) \equiv f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

where $\equiv$ is “Kleene equivalence” of possibly-undefined expressions: **LHS** $\equiv$ **RHS** means “either both **LHS** and **RHS** are undefined, or they are both defined and are equal.”

Composition

Composition of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ is the partial function $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in \mathbb{N}$

$$f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) \equiv f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

So $f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) = z$ iff there exist $y_1, \ldots, y_n$ with $g_i(x_1, \ldots, x_m) = y_i$ (for $i = 1..n$) and $f(y_1, \ldots, y_n) = z$.

Composition

Composition of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ is the partial function $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in \mathbb{N}$

$$f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) \equiv f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

So $f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) = z$ iff there exist $y_1, \ldots, y_n$ with $g_i(x_1, \ldots, x_m) = y_i$ (for $i = 1..n$) and $f(y_1, \ldots, y_n) = z$.

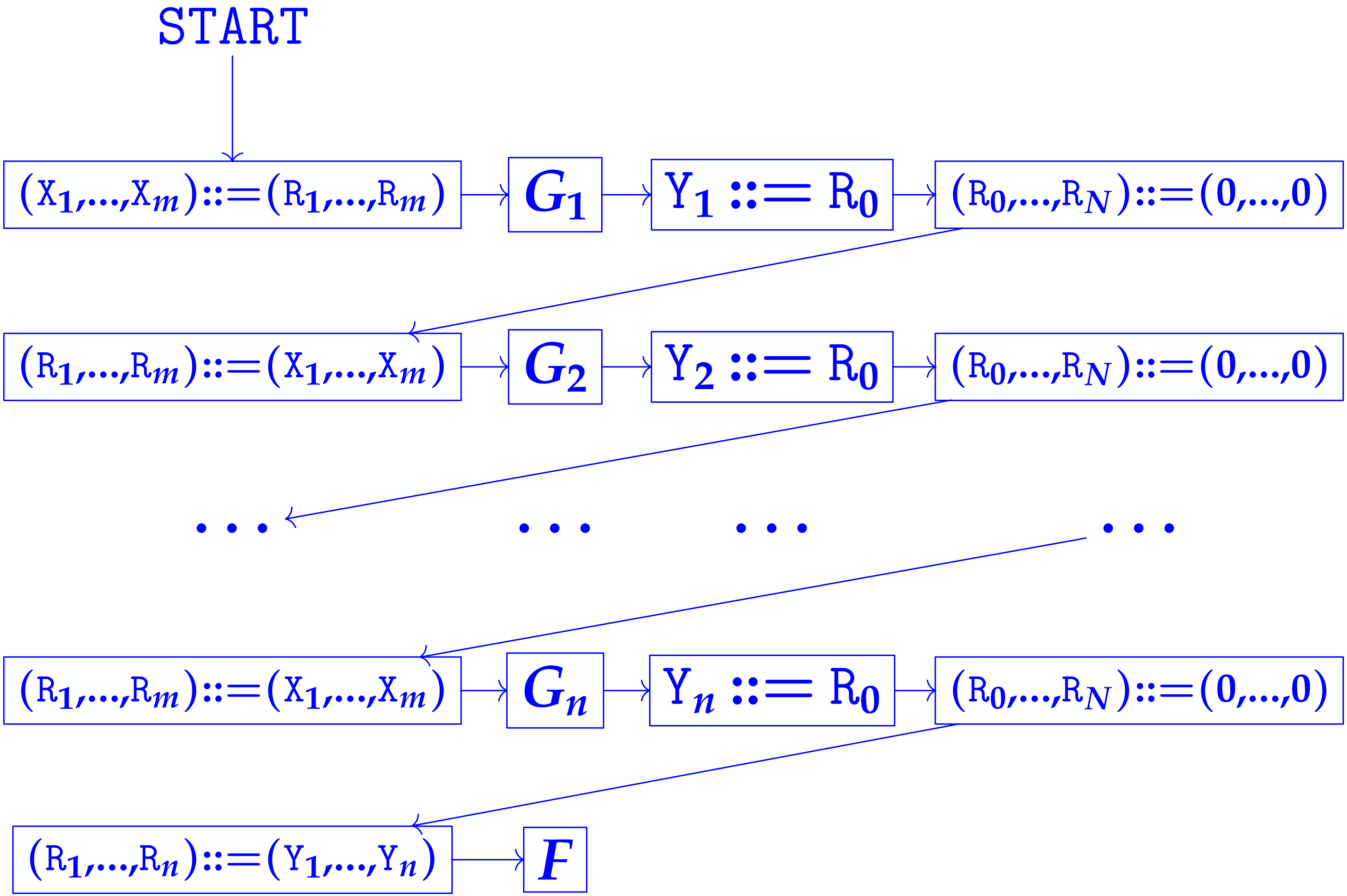
N.B. in case $n = 1$, we write $f \circ g_1$ for $f \circ [g_1]$.

Composition

$f \circ [g_1, \ldots, g_n]$ is computable if f and $g_1, \ldots, g_n$ are.

Proof. Given RM programs $\begin{cases} F \\ G_i \end{cases}$ computing $\begin{cases} f(y_1, \ldots, y_n) \\ g_i(x_1, \ldots, x_m) \end{cases}$ in $\mathtt{R}_0$ starting with $\begin{cases} \mathtt{R}_1, \ldots, \mathtt{R}_n \\ \mathtt{R}_1, \ldots, \mathtt{R}_m \end{cases}$ set to $\begin{cases} y_1, \ldots, y_n \\ x_1, \ldots, x_m \end{cases}$, then the next slide specifies a RM program computing $f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m)$ in $\mathtt{R}_0$ starting with $\mathtt{R}_1, \ldots, \mathtt{R}_m$ set to $x_1, \ldots, x_m$.

(**Hygiene** [caused by the lack of *local names* for registers in the RM model of computation]: we assume the programs $F, G_1, \ldots, G_n$ only mention registers up to $\mathtt{R}_N$ (where $N \geq \max\{n, m\}$) and that $\mathtt{X}_1, \ldots, \mathtt{X}_m, \mathtt{Y}_1, \ldots, \mathtt{Y}_n$ are some registers $\mathtt{R}_i$ with $i > N$.)


START
(X1,...,Xm)::=(R1,...,Rm)
G1
Y1 ::= R0
(R0,...,RN)::=(0,...,0)
(R1,...,Rm)::=(X1,...,Xm)
G2
Y2 ::= R0
(R0,...,RN)::=(0,...,0)
(R1,...,Rm)::=(X1,...,Xm)
Gn
Yn ::= R0
(R0,...,RN)::=(0,...,0)
(R1,...,Rn)::=(Y1,...,Yn)
F

Partial recursive functions

Examples of recursive definitions

$$\begin{cases} f_1(0) & \equiv 0 \\ f_1(x+1) & \equiv f_1(x) + (x+1) \end{cases}$$

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

Examples of recursive definitions

$$\begin{cases} f_1(0) & \equiv 0 \\ f_1(x+1) & \equiv f_1(x) + (x+1) \end{cases}$$

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

$$\begin{cases} f_2(0) & \equiv 0 \\ f_2(1) & \equiv 1 \\ f_2(x+2) & \equiv f_2(x) + f_2(x+1) \end{cases}$$

$f_2(x) = x$th Fibonacci number

Examples of recursive definitions

$$\begin{cases} f_1(0) & \equiv 0 \\ f_1(x+1) & \equiv f_1(x) + (x+1) \end{cases}$$

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

$$\begin{cases} f_2(0) & \equiv 0 \\ f_2(1) & \equiv 1 \\ f_2(x+2) & \equiv f_2(x) + f_2(x+1) \end{cases}$$

$f_2(x) = x$th Fibonacci number

$$\begin{cases} f_3(0) & \equiv 0 \\ f_3(x+1) & \equiv f_3(x+2) + 1 \end{cases}$$

$f_3(x)$ undefined except when $x = 0$

Examples of recursive definitions

$$\begin{cases} f_1(0) & \equiv 0 \\ f_1(x+1) & \equiv f_1(x) + (x+1) \end{cases}$$

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

$$\begin{cases} f_2(0) & \equiv 0 \\ f_2(1) & \equiv 1 \\ f_2(x+2) & \equiv f_2(x) + f_2(x+1) \end{cases}$$

$f_2(x) = x$th Fibonacci number

$$\begin{cases} f_3(0) & \equiv 0 \\ f_3(x+1) & \equiv f_3(x+2) + 1 \end{cases}$$

$f_3(x)$ undefined except when $x = 0$

$$f_4(x) \equiv \textit{if } x > 100 \textit{ then } x - 10 \textit{ else } f_4(f_4(x+11))$$

f_4 is McCarthy's "91 function", which maps x to 91 if $x \leq 100$ and to $x - 10$ otherwise

Examples of recursive definitions

$$\begin{cases} f_1(0) & \equiv 0 \\ f_1(x+1) & \equiv f_1(x) + (x+1) \end{cases}$$

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

$$\begin{cases} f_2(0) & \equiv 0 \\ f_2(1) & \equiv 1 \\ f_2(x+2) & \equiv f_2(x) + f_2(x+1) \end{cases}$$

$f_2(x) = x$th Fibonacci number

$$\begin{cases} f_3(0) & \equiv 0 \\ f_3(x+1) & \equiv f_3(x+2) + 1 \end{cases}$$

$f_3(x)$ undefined except when $x = 0$

$$\begin{aligned} f_4(x) \equiv\ & \mathit{if}\ x > 100\ \mathit{then}\ x - 10 \\ & \mathit{else}\ f_4(f_4(x+11)) \end{aligned}$$

f_4 is McCarthy's "91 function", which maps x to 91 if $x \leq 100$ and to $x - 10$ otherwise

Primitive recursion

Theorem. Given $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g \in \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$, there is a unique $h \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ satisfying

$$\begin{cases} h(\vec{x}, 0) & \equiv f(\vec{x}) \\ h(\vec{x}, x+1) & \equiv g(\vec{x}, x, h(\vec{x}, x)) \end{cases}$$

for all $\vec{x} \in \mathbb{N}^n$ and $x \in \mathbb{N}$.

We write $\rho^n(f, g)$ for h and call it the partial function defined by primitive recursion from f and g.