Recursive functions
Part 1: primitive recursion

Recursive functions and computability

- **Recursive functions** are another class of effectively-computable functions.
- Unlike for Turing machines and abacus machines, the description of recursive functions is **inductive**: certain basic functions are recursive, and functions build from recursive functions in a certain way are also recursive.

Church’s thesis

- **Church’s thesis**: “Every effectively-computable function is recursive.”
- Analogous to Turing’s thesis.

Computability: the big picture

- Later, we show that recursive functions can simulate TMs, and abacus machines can simulate recursive functions.
- Because TMs can simulate abacus machines, we get a cycle of simulations.
- So all three kinds of computable functions are the same.
- In particular, Church’s thesis and Turing’s thesis are equivalent.
Roadmap

- First, we introduce the primitive recursive functions.
- Then we introduce the recursive function by adding a construct called minimization.
- Intuitively, primitive recursive functions can do only FOR loops and always terminate, whereas minimization corresponds to a WHILE loop that may not terminate.

Motivation for primitive recursion

The two “rewriting rules”

\[ x^0 = 1 \]  \hspace{1cm} (1)
\[ x^{y+1} = x \cdot x^y \]  \hspace{1cm} (2)

are enough to define the \( \text{exp} \) function.

Motivation for primitive recursion

Consider the function \( \exp(x, y) = x^y \):

\[
\begin{align*}
x^0 &= 1 \\
x^1 &= x \\
x^2 &= x \cdot x \\
&\vdots \\
x^y &= x \cdot x \cdots \cdot x \quad (y \text{ occurrences of } x) \\
x^{y+1} &= x \cdot x \cdots \cdot x \quad (y + 1 \text{ occurrences of } x) \\
&= x \cdot x^y
\end{align*}
\]

Motivation for primitive recursion

The rules for \( \exp \) reduce exponentiation to multiplication; now consider

\[
\begin{align*}
x \cdot 0 &= 0 \quad \text{(1)} \\
x \cdot (y + 1) &= x + x \cdot y \quad \text{(2)}
\end{align*}
\]

So the rules for \( \cdot \) reduce multiplication to addition.
Motivation for primitive recursion

Now consider

\[ x + 0 = x \] (1)
\[ x + (y + 1) = 1 + (x + y) \] (2)

So the rules for + reduce addition to adding 1.

Building blocks for prim. rec. functions

On the next slides, we introduce the building blocks for primitive recursive functions. There will be

- three classes of basic functions: successor, zero, and projections, and
- two ways of building new primitive recursive functions from old: composition and primitive recursion.

Motivation for primitive recursion

- Primitive recursion is in the spirit of our “computation by rewriting” definitions of \( \exp \), \( \cdot \), and +.
- It consists of one rule for \( y = 0 \) and one rule for \( y > 0 \).
- \( y \) acts as a “countdown” for the number of remaining steps in the computation.

The successor function

- The function that takes \( x \) to \( x + 1 \) can be taken apart no further.
- Therefore, it will be a basic building block for primitive recursive functions.
- We denote it by \( s \) (for “successor”).
The zero function

- The zero is used in every computation and will therefore be a basic building block for primitive recursive functions.
- For technical reasons, we shall use the zero function

\[ z : \begin{cases} N & \rightarrow N \\ z(x) & = 0 \end{cases} \]

Composition

If \( g_1, g_2, \ldots, g_m \) are functions \( N^k \rightarrow N \), and \( f \) is a function \( N^m \rightarrow N \), then the function \( h : N^k \rightarrow N \) given by

\[ h(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k)) \]

is said to arise by composition from \( f, g_1, \ldots, g_k \).
We write

\[ h = Cn[f, g_1, \ldots, g_m]. \]

The projections

- A projection function is of the form:

\[ p_i^k : \begin{cases} N^k & \rightarrow N \\ p(x_1, x_2, \ldots, x_k) & = x_i \end{cases} \]

- Called so because it goes from \( k \)-dimensional “space” into one-dimensional “space”.
- Projections occur in almost every computation and will therefore be basic building blocks for primitive recursive functions.

Composition: example

Consider

\[ h(x_1, x_2, x_3) = f(g(x_1, x_2, x_3)) \]

where \( f = s \) and \( g = p_3^2 \). Thus \( h \) returns the successor of the second argument. Formally:

\[ h = Cn[f, g] = Cn[s, p_3^2]. \]
**Composition:**

**Constant functions.** Consider

\[ h(x) = f(g(x)) \]

where \( f = s \) and \( g = z \).

Thus \( h \) is the constant function that returns 1. Formally: \( h = Cn[f, g] = Cn[s, z] \). We have

- \( z \) the constant 0 function
- \( Cn[s, z] \) the constant 1 function
- \( Cn[s, Cn[s, z]] \) the constant 2 function

\[ \vdots \]

**Primitive recursion**

If \( f : N^k \to N \) and \( g : N^{k+2} \to N \), then the function \( h : N^{k+1} \to N \) is said to be defined by **primitive recursion** from \( f \) and \( g \) if

\[
\begin{align*}
    h(\bar{x}, 0) &= f(\bar{x}) \\
    h(\bar{x}, s(y)) &= g(\bar{x}, y, h(\bar{x}, y))
\end{align*}
\]

where \( \bar{x} \) stands for \( x_1, \ldots, x_k \). We write

\[ h = Pr[f, g] \]

**Sum**

\[
\begin{align*}
    \text{sum}(x, 0) &= x \\
    \text{sum}(x, s(y)) &= s(\text{sum}(x, y))
\end{align*}
\]

So

\[
\begin{align*}
    f(x) &= x = p_1^1(x) \\
    g(x, y, u) &= s(u) = Cn[s, p_3^3]
\end{align*}
\]

Thus

\[ \text{sum} = Pr[f, g] = Pr[p_1^1, Cn[s, p_3^3]] \]

**Multiplication**

Multiplication can be defined as follows:

\[ \text{prod} = Pr[z, Cn[\text{sum}, p_1^3, p_3^3]] \]
Definition of primitive recursive functions

**Definition.** The class of primitive recursive functions is defined as follows:

- The zero function \( z \), the successor function \( s \), and all projection functions \( p^k_i \) are primitive recursive.
- Functions which arise by composition \( C_n \) or primitive recursion \( Pr \) from primitive recursive functions are also primitive recursive.

**Exercise**

The predecessor function \( \text{pred} \) takes one argument \( y \) and returns \( y - 1 \) if \( y \) is greater than 0, and returns 0 otherwise. Show that \( \text{pred} \) is primitive recursive by using (not necessarily all of) \( s, z, p^k_i, C_n, \) and \( Pr \).

**Exercise**

Show that the factorial function is primitive recursive.

**Exercise**

We have seen that there are encodings of pairs of natural numbers, i.e. that there are total injections \( c : N \times N \to N \); show for one such encoding \( c \) that it is primitive recursive.
Next, we will show that every primitive recursive function is computable by an abacus machine (and therefore also by a Turing machine).

**Abacus program for the zero function**

Decrease the content of $R_1$ until it contains zero:

0: if $[1]=0$ then
    {goto 99}
else
    {1-;goto 0}

**Abacus program for the successor**

Increase $R_1$:

0: 1+; goto 99

**Abacus program for the projection $p^k_i$**

- If $i = 1$, the program needs to do nothing, because the result is already in $R_1$.
  
  0: goto 99

- For $i \neq 1$, the program makes $R_1$ zero and then empties $R_i$ into $R_1$:

  0: if $[1]=0$ then {goto 1} else {1-;goto 0}
  1: if $[i]=0$ then {goto 99} else {i-;goto 2}
  2: 1+; goto 1
Suppose that $h$ is defined by composition from $f, g_1, g_2$ as follows:

$$h(x_1, x_2, x_3) = f(g_1(x_1, x_2, x_3), g_2(x_1, x_2, x_3)).$$

The next slide contains an abacus program for $h$, where $x_1, x_2, x_3$ and aux are register that must not be used by $f, g_1,$ or $g_2$.

We build the result of $h$ in a register $z$, while $y$ acts as a “countdown”. Example for $y = 2$:

<table>
<thead>
<tr>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$h(x, 0) = f(x)$</td>
</tr>
<tr>
<td>1</td>
<td>$h(x, 1) = g(x, 0, f(x))$</td>
</tr>
<tr>
<td>0</td>
<td>$h(x, 2) = g(x, 1, g(x, 0, f(x)))$</td>
</tr>
</tbody>
</table>

We use a register $i$ for the increasing counter. On the next slide, $x, y, z, i$ are registers that must not be used by $f$ and $g$. 

[Abacus program for composition]

1. $\rightarrow x_1$; // save R1
2. $\rightarrow x_2$; // save R2
3. $\rightarrow x_3$; // save R3
Program for $g_1$;
1. $\rightarrow$ aux; // save result of $g_1$
[x1] $\rightarrow 1$; // restore R1
[x2] $\rightarrow 2$; // restore R2
[x3] $\rightarrow 3$; // restore R3
Program for $g_2$;
1. $\rightarrow 2$; // move result of $g_2$ to R2
[aux] $\rightarrow 1$; // restore result of $g_1$ to R1
Program for $f$
Summary of primitive recursion

Definition of primitive recursive functions

Definition. The class of primitive recursive functions is defined as follows:

- The zero function \( z \), the successor function \( s \), and all projection functions \( p_i^k \) are primitive recursive.

- Functions which arise by composition \( C_n \) or primitive recursion \( Pr \) from primitive recursive functions are also primitive recursive.

Primitive recursion

If \( f : \mathbb{N}^k \rightarrow \mathbb{N} \) and \( g : \mathbb{N}^{k+2} \rightarrow \mathbb{N} \), then the function \( h : \mathbb{N}^{k+1} \rightarrow \mathbb{N} \) is said to be defined by primitive recursion from \( f \) and \( g \) if

\[
\begin{align*}
h(\bar{x}, 0) &= f(\bar{x}) \\
h(\bar{x}, s(y)) &= g(\bar{x}, y, h(\bar{x}, y))
\end{align*}
\]

where \( \bar{x} \) stands for \( x_1, \ldots, x_k \). We write

\[ h = \text{Pr}[f, g] \]

Abacus program for composition

Suppose that \( h \) is defined by composition from \( f, g_1, g_2 \) as follows:

\[ h(x_1, x_2, x_3) = f(g_1(x_1, x_2, x_3), g_2(x_1, x_2, x_3)). \]

The next slide contains an abacus program for \( h \), where \( x_1, x_2, x_3 \) and \( \text{aux} \) are register that must not be used by \( f, g_1, \) or \( g_2 \).
Abacus program for composition

1. -> x1; // save R1
2. -> x2; // save R2
3. -> x3; // save R3

Program for g1;
1. -> aux; // save result of g1
[x1] -> 1; // restore R1
[x2] -> 2; // restore R2
[x3] -> 3; // restore R3

Program for g2;
1. -> 2; // move result of g2 to R2
[aux] -> 1; // restore result of g1 to R1

Program for f

We build the result of \( h \) in a register \( z \), while \( y \) acts as a “countdown”. Example for \( y = 2 \):

<table>
<thead>
<tr>
<th>( y )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>( h(x, 0) = f(x) )</td>
</tr>
<tr>
<td>1</td>
<td>( h(x, 1) = g(x, 0, f(x)) )</td>
</tr>
<tr>
<td>0</td>
<td>( h(x, 2) = g(x, 1, g(x, 0, f(x))) )</td>
</tr>
</tbody>
</table>

We use a register \( i \) for the increasing counter.

Abacus program for

\[ h = \Pr(f, g) \]

On the next slide, \( x, y, z, i \) are registers that must not be used by \( f \) or \( g \), and \( y_0 \) stands for the initial value of Register 2.

\[ h = \Pr(f, g) \]

1. -> x;
2. -> y;

Program for f;
1. -> z;
0 -> i; // now z = h(x, i) and i+y = y_0
A: if \( y \)=0 then { goto C } else { y-; goto B }
B: [x] -> 1;
[i] -> 2;
[z] -> 3;
Program for g;
1. -> z;
i+; // now again z = h(x, i) and i+y = y_0
goto A;
C: [z] -> 1; // return z
The Ackermann function is defined as follows:

\[
A(0, y) = y + 1 \quad (1)
\]
\[
A(x + 1, 0) = A(x, 1) \quad (2)
\]
\[
A(x + 1, y + 1) = A(x, A(x + 1, y)) \quad (3)
\]

Compute \( A(2, 1) \):

\[
A(2, 1) = A(1, A(2, 0)) = A(1, A(1, 1))
\]
\[
= A(1, A(0, A(1, 0))) = A(1, A(0, A(0, 1)))
\]
\[
= A(1, A(0, 2)) = A(1, 3) = A(0, A(1, 2))
\]
\[
= A(0, A(0, A(1, 1))) = A(0, A(0, A(0, 1)))
\]
\[
= A(0, A(0, A(0, A(0, 1)))) = A(0, A(0, A(0, 2)))
\]
\[
= A(0, A(0, 3)) = A(0, 4) = 5
\]

Define the lexicographical order on \( \mathbb{N} \times \mathbb{N} \) as follows:

\( (x, y) > (x', y') \) if \( x > x' \) or \( x = x' \) and \( y > y' \).

The clauses (2) and (3) lead to lexicographically smaller arguments; this cannot go on forever, so \( A \) must finally halt.

We define a configuration to be an expression of the form

\[
A(x_1, A(x_2, \ldots (A(x_{n-1}, x_n))))
\]

Here is an algorithm for computing \( A(x, y) \):

While there is an \( A(\ldots, \ldots) \) left {

Apply the suitable rule (1, 2, or 3) to the innermost \( A \) }
Ackermann is not primitive recursive

- If $h(x, y)$ is defined by primitive recursion, then $y$ operates as a “countdown”.
- By contrast, the totality of the Ackermann function is shown with the lexicographical ordering on pairs.
- Fact: $A(y, y)$ gets greater than any primitive recursive function $h(y)$ for sufficiently great $y$.
- So in particular, $A$ is not primitive recursive.

Towards general recursion

- As we have seen, the Ackermann function is not primitive recursive.
- Some other computable functions are not primitive recursive simply because they are not total.
- In both cases, the algorithms can be written in the form “WHILE some condition holds, DO X”.
- Technically, instead of WHILE loops we add a construct called minimization which does something equivalent.

Definition of minimization

The minimization of a function $f : N^{k+1} \rightarrow N$ is defined as follows:

$$M_n[f](x_1, \ldots, x_k) = \begin{cases} y & \text{if } f(x_1, \ldots, x_k, y) = 0 \\
& \text{and for all } i < y, \\
& f(x_1, \ldots, x_k, i) \neq 0 \\
\bot & \text{otherwise} \end{cases}$$

Algorithm for $M_n[f]$

The algorithm for $M_n[f]$ (presented in pseudocode) goes as follows:

```pseudocode
y = 0;
while(not(f(x, y) = 0)) {
    y = y+1;
}
return y;
```

This can fail to halt for two reasons: either because $f(x, i)$ fails to halt for some $i$, or because $f(x, i) \neq 0$ for all $i$. 
Definition of recursive functions

Definition. The class of recursive functions is defined as follows:

- The functions $s$ and $z$ are recursive, and so are all projections $p^k_i$.
- Functions built from recursive ones by using composition $C_n$ or primitive recursion $P_r$ are also recursive.
- Functions built from recursive ones by minimization $M_n$ are also recursive.

Exercise

Let $f$ be a two-argument recursive function. Show that the following functions are also recursive:

1. $g(x, y) = f(y, x)$;
2. $h(x) = f(x, x)$;
3. $k_{17}(x) = f(17, x)$, and $k^{17}_1(x) = f(x, 17)$.

Exercise

Give a reasonable way of assigning code numbers to recursive functions.

Exercise

Given a reasonable way of coding recursive functions by natural numbers, let $d(x) = 1$ if the one-argument function with the code number $x$ is defined and has value 0 for argument $x$, and $d(x) = 0$ otherwise. Show that this function is not recursive.
Exercise

Let \( h(x, y) = 1 \) if the one-argument recursive function with code number \( x \) is defined for argument \( y \), and \( h(x, y) = 0 \) otherwise. Show that this function is not recursive.

Rec. functions are abacus-computable

- Evidently, every primitive recursive function is recursive.
- We have seen earlier that all primitive recursive functions are abacus-computable.
- We have also seen that minimization is abacus-computable.
- Therefore, all recursive functions are abacus-computable.

Abacus program for \( Mn[f] \)

Registers \( x \) and \( y \) must not be used by the program for \( f \).

\[
\begin{align*}
[1] & \to x; \\
0 & \to y; \\
A: & x \to 1; \\
& y \to 2; \\
& \text{program for } f; \\
& \text{if } [1]=0 \text{ then } \{ \text{goto C} \} \text{ else } \{ \text{goto B} \}; \\
B: & y++; \ \text{goto A}; \\
C: & [y] \to 1;
\end{align*}
\]

Recursive functions: a reminder
Primitive recursion, informally

To define a function $h(x, z)$ by primitive recursion, we need to describe what happens
- in the case where $z = 0$, and
- in the case where $z$ is of the form $y + 1$ for some $y$, using the value of $h(x, y)$.

Example: multiplication

Below we have a primitive recursive definition of multiplication, in terms of $+$.

\[
\begin{align*}
x \cdot 0 &= 0 \\
x \cdot (y + 1) &= x \cdot y + x
\end{align*}
\]

Note that this looks almost like a realistic computer program. An even more realistic version might look like

\[
mult(x, y) = \\
\quad \text{if } (y = 0) \text{ then } 0 \text{ else } mult(x, y - 1) + x.
\]

Definition of primitive recursive functions

Definition. The class of primitive recursive functions is defined as follows:

- The zero function $z$, the successor function $s$, and all projection functions $p_{k}^{i}$ are primitive recursive.
- Functions which arise by composition $C_{n}$ or primitive recursion $Pr$ from primitive recursive functions are also primitive recursive.
Towards general recursion

- Some functions (e.g. the Ackermann function) are not primitive recursive.
- Informally, this is because primitive recursive functions do not allow while-loops, i.e. constructs of the form “WHILE some condition holds, DO X”.
- Formally, instead of WHILE loops, we add a construct called minimization.

Definition of minimization

The minimization of a function \( f : N^{k+1} \to N \) is defined as follows (where \( x \) stands for \( x_1, \ldots, x_k \)):

\[
\text{Mn}[f](x) = \begin{cases} 
  y & \text{if } f(x, y) = 0 \text{ and for all } i < y, \ f(x, i) \text{ is defined and } \neq 0 \\
  \perp & \text{otherwise}
\end{cases}
\]

Algorithm for \( \text{Mn}[f](x) \)

The algorithm for \( \text{Mn}[f](x) \) looks as follows:

\[
y = 0; \\
\text{while(}\neg(f(x, y) = 0)) \{ \\
  y = y+1; \\
\}\text{ return } y;
\]

This can fail to halt for two reasons: either because \( f(x, i) \) fails to halt for some \( i \), or because \( f(x, i) \neq 0 \) for all \( i \).

Definition of recursive functions

Definition. The class of recursive functions is defined as follows:

- The functions \( s \) and \( z \) are recursive, and so are all projections \( p^k_i \).
- Functions built from recursive ones by using composition \( C_n \) or primitive recursion \( P_r \) are also recursive.
- Functions built from recursive ones by minimization \( \text{Mn} \) are also recursive.