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** Cn or **primitive recursion** Pr from primitive recursive functions are also primitive recursive.

Primitive recursion

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

$$h(\bar{x}, 0) = f(\bar{x})$$
$$h(\bar{x}, s(y)) = g(\bar{x}, y, h(\bar{x}, y))$$

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

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

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 `x1,x2,x3` and `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
```

Abacus program for $h = \mathrm{Pr}(f, g)$

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

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

We use a register i for the increasing counter.

Abacus program for $h = \mathrm{Pr}(f, g)$

On the next slide, `x,y,z,i` are registers that must not be used by f or g, and `y0` stands for the initial value of Register 2.

Abacus program for $h = \mathrm{Pr}(f, g)$

```
    [1] -> x;
    [2] -> y;
    Program for f;
    [1]-> z;
    0 -> i; // now z = h(x,i) and i+y = y0
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 = y0
    goto A;
C: [z] -> 1; // return z
```

Limits of primitive recursion

The **Ackermann function** is defined as follows:

$$A(0, y) = y + 1 \tag{1}$$

$$A(x + 1, 0) = A(x, 1) \tag{2}$$

$$A(x + 1, y + 1) = A(x, A(x + 1, y)) \tag{3}$$

Computation for $A(2,1)$

$$
\begin{aligned}
&A(2,1) = A(1, A(2,0)) = A(1, A(1,1)) \\
&= A(1, A(0, A(1,0))) = A(1, A(0, A(0,1))) \\
&= A(1, A(0,2)) = A(1,3) = A(0, A(1,2)) \\
&= A(0, A(0, A(1,1))) = A(0, A(0, A(0, A(1,0)))) \\
&= A(0, A(0, A(0, A(0,1)))) = A(0, A(0, A(0,2))) \\
&= A(0, A(0,3)) = A(0,4) = 5
\end{aligned}
$$

Why A is a total function

$$A(0, y) = y + 1 \tag{1}$$
$$A(x+1, 0) = A(x, 1) \tag{2}$$
$$A(x+1, y+1) = A(x, A(x+1, y)) \tag{3}$$

Define the **lexicographical order** on $N \times N$ 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.

Computing $A(x, y)$ by a while loop

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(...,...) 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:

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

Algorithm for $\mathrm{Mn}[f]$

The algorithm for $\mathrm{Mn}[f]$ (presented in **pseudocode**) goes as follows:

```
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_i^k.
- Functions built from recursive ones by using composition Cn or primitive recursion Pr are also recursive.
- Functions built from recursive ones by minimization Mn 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 $k^{17}(x) = f(x, 17)$.

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.

Abacus program for $\mathrm{Mn}[f]$

Registers x and y must no be used by the program for f.

```
   [1] -> x;
   0 -> y;
A: x -> 1;
   y -> 2;
   program for f;
   if [1]=0 then {goto C} else {goto B};
B: y+; goto A;
C: [y] -> 1;
```

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.