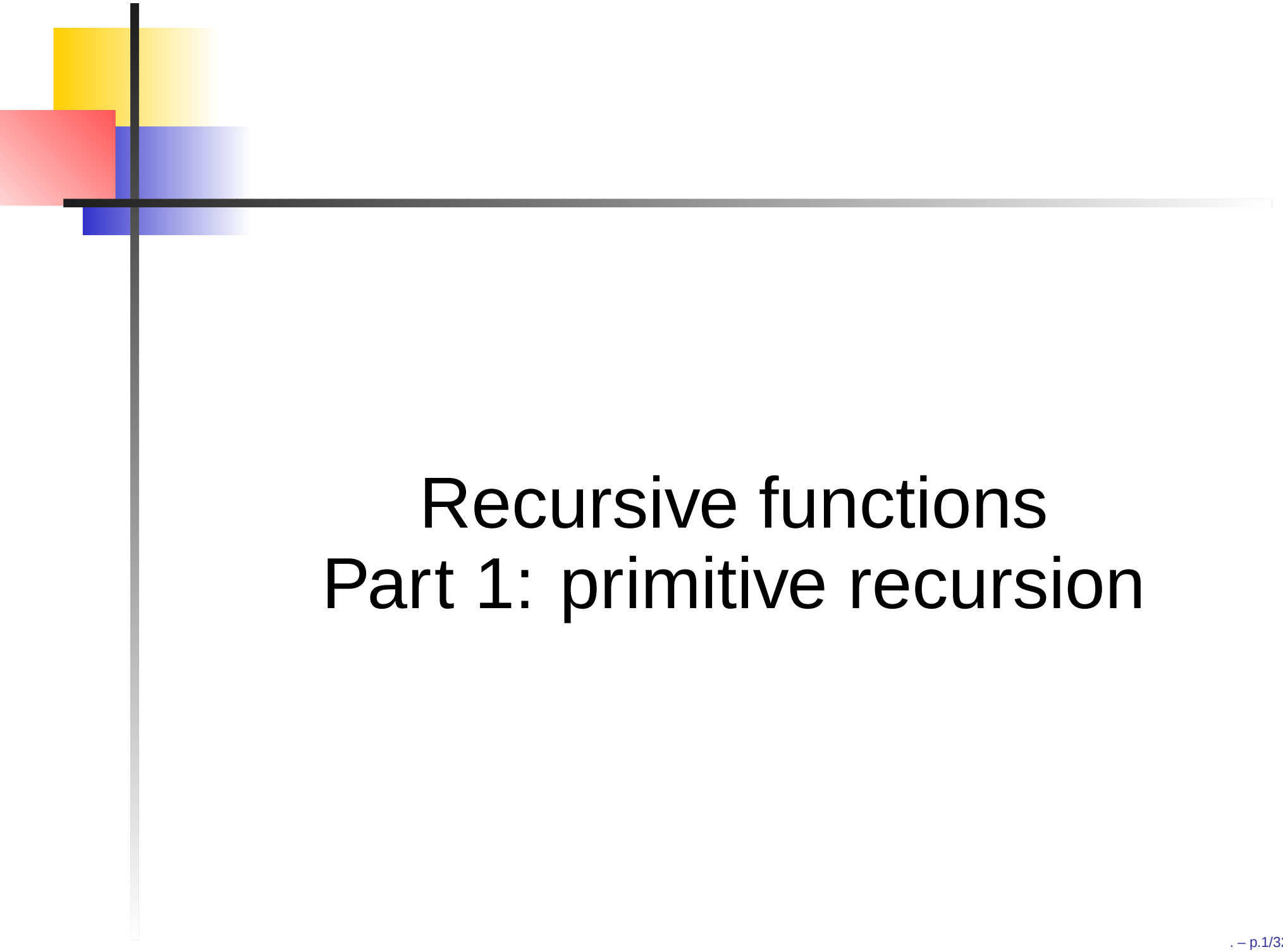

Recursive functions
Part 1: primitive recursion

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

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

$$
\begin{aligned}
x^0 &= 1 \\
x^1 &= x \\
x^2 &= x \cdot x \\
&\vdots \\
x^y &= x \cdot x \cdot \cdots \cdot x \quad (y \text{ occurrences of } x) \\
x^{y+1} &= x \cdot x \cdot \cdots \cdot x \quad (y+1 \text{ occurrences of } x) \\
&= x \cdot x^y
\end{aligned}
$$

Motivation for primitive recursion

The two “rewriting rules”

$$x^0 = 1 \tag{1}$$

$$x^{y+1} = x \cdot x^y \tag{2}$$

are enough to define the $\exp$ function.

Motivation for primitive recursion

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

$$x \cdot 0 = 0 \tag{1}$$

$$x \cdot (y + 1) = x + x \cdot y \tag{2}$$

So the rules for $\cdot$ reduce multiplication to addition.

Motivation for primitive recursion

Now consider

$$x + 0 = x \tag{1}$$

$$x + (y + 1) = 1 + (x + y) \tag{2}$$

So the rules for $+$ reduce addition to adding 1.

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.

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**.

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**

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

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

If $g_1, g_2, \ldots, g_m$ are functions $N^k \to N$, and f is a function $N^m \to N$, then the function $h : N^k \to N$ 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

$$h = \mathrm{Cn}[f, g_1, \ldots, g_m].$$

Composition: example

Consider

$$h(x_1, x_2, x_3) = f(g(x_1, x_2, x_3))$$

where $f = s$ and $g = p_2^3$. Thus h returns the successor of the second argument. Formally:

$$h = \mathrm{Cn}[f, g] = \mathrm{Cn}[s, p_2^3].$$

Composition: example

Constant functions. Consider

$$h(x) = f(g(x)) \qquad \text{where } f = s \text{ and } g = z.$$

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

$$\begin{aligned} z &\quad \text{the constant } 0 \text{ function} \\ \mathrm{Cn}[s, z] &\quad \text{the constant } 1 \text{ function} \\ \mathrm{Cn}[s, \mathrm{Cn}[s, z]] &\quad \text{the constant } 2 \text{ function} \\ &\vdots \end{aligned}$$

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

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

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

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

$$h = \mathrm{Pr}[f, g]$$

Sum

$$\mathrm{sum}(x, 0) = x$$
$$\mathrm{sum}(x, s(y)) = s(\mathrm{sum}(x, y))$$

So

$$f(x) = x = p_1^1(x)$$
$$g(x, y, u) = s(u) = \mathrm{Cn}[s, p_3^3]$$

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

Multiplication

Multiplication can be defined as follows:

$$\mathrm{prod} = \mathrm{Pr}[z, \mathrm{Cn}[\mathrm{sum}, p_1^3, p_3^3]].$$

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_i^k are primitive recursive.
- Functions which arise by **composition** Cn or **primitive recursion** Pr from primitive recursive functions are also primitive recursive.

Exercise

The **predecessor function** $pred$ takes one argument y and returns $y - 1$ if y is greater than 0, and returns 0 otherwise. Show that $pred$ is primitive recursive by using (not necessarily all of) s, z, p_i^k, Cn, 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 $c : N \times N \rightarrow N$; show for one such encoding c that it is primitive recursive.

Prim. rec. $\Longrightarrow$ abacus-computable

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 successor

Increase R_1:

```
0: 1+; goto 99
```

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 projection p_i^k

- 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
```

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. On the next slide, `x,y,z,i` are registers that must not be used by f and g.

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
   if [y]=0 then { goto X } else y-;goto Y
Y: [x] -> 1;
   [i] -> 2;
   [z] -> 3;
   Program for g;
   [1] -> z;
   i+; // now again z = h(x,i) and i+y = y0
   goto Y;
X: [z] -> 1; // return z
```