PHIL 222
Philosophical Foundations of Computer Science
Week 4, Tuesday

Sept. 17, 2024

Recursive Functions: Primitive Recursion (cont'd)

All the cases so far can be unified by

```
h(x, y, z, 0) := f(x, y, z)
for i in (0, ..., n-1):
    h(x, y, z, i+1) := g(x, y, z, i, h(x, y, z, i))
```

All the cases so far can be unified by

```
h(x, y, z, 0) := f(x, y, z)
for i in (0, ..., n-1):
    h(x, y, z, i+1) := g(x, y, z, i, h(x, y, z, i))
```

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

All the cases so far can be unified by

```
h(x, y, z, 0) := f(x, y, z)
for i in (0, ..., n-1):
    h(x, y, z, i+1) := g(x, y, z, i, h(x, y, z, i))
```

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

If f and g are computable so is h!

All the cases so far can be unified by

```
h(x, y, z, 0) := f(x, y, z)
for i in (0, ..., n-1):
    h(x, y, z, i+1) := g(x, y, z, i, h(x, y, z, i))
```

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

If f and g are computable so is h!

N.B. Keep track of the numbers of inputs / arguments:

- if f takes n inputs, g takes $n+2$ inputs, and h takes $n+1$ inputs.

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

E.g., we just showed that + can be defined this way: We saw

$$x + 0 = x, \qquad x + s(i) = s(x + i),$$

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad\qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

E.g., we just showed that + can be defined this way: We saw

$$x + 0 = x, \qquad\qquad x + s(i) = s(x + i),$$

i.e., we have $\qquad h(x, 0) = x, \qquad\qquad h(x, s(i)) = s(h(x, i)).$

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

E.g., we just showed that + can be defined this way: We saw

$$x + 0 = x, \qquad x + s(i) = s(x + i),$$

i.e., we have $$h(x, 0) = x, \qquad h(x, s(i)) = s(h(x, i)).$$

So we have $$h(x, 0) = f(x), \qquad h(x, s(i)) = g(x, i, h(x, i))$$

if we define $$f(x) = x, \qquad g(x, i, k) = s(k).$$

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

E.g., we just showed that + can be defined this way: We saw

$$x + 0 = x, \qquad x + s(i) = s(x + i),$$

i.e., we have $\quad h(x, 0) = x, \qquad h(x, s(i)) = s(h(x, i)).$

So we have $\quad h(x, 0) = f(x), \qquad h(x, s(i)) = g(x, i, h(x, i))$

if we define $\quad f(x) = x, \qquad g(x, i, k) = s(k).$

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

E.g., we just showed that + can be defined this way: We saw

$$x + 0 = x, \qquad x + s(i) = s(x + i),$$

i.e., we have $\quad h(x, 0) = x, \qquad h(x, s(i)) = s(h(x, i)).$

So we have $\quad h(x, 0) = f(x), \qquad h(x, s(i)) = g(x, i, h(x, i))$

if we define $\quad f(x) = x, \qquad g(x, i, k) = s(k).$

This means that + is defined by primitive recursion from these f and g.

We just showed that + is defined by primitive recursion from

$$f(x) = x, \qquad\qquad g(x, i, k) = s(k).$$

We just showed that + is defined by primitive recursion from

$$f(x) = x, \qquad\qquad g(x, i, k) = s(k).$$

But these f and g are computable because they are

We just showed that + is defined by primitive recursion from

$$f(x) = x, \qquad\qquad g(x, i, k) = s(k).$$

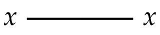
But these f and g are computable because they are

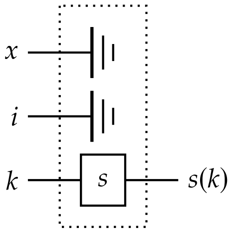
x ——— x

We just showed that + is defined by primitive recursion from

$$f(x) = x, \qquad\qquad g(x, i, k) = s(k).$$

But these f and g are computable because they are

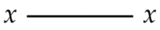


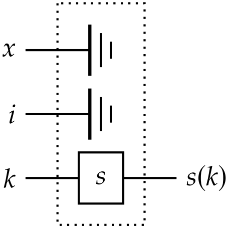


We just showed that + is defined by primitive recursion from

$$f(x) = x, \qquad\qquad g(x, i, k) = s(k).$$

But these f and g are computable because they are





Therefore + is computable!

$\times$ is also computable:

$\times$ is also computable: It satisfies

$$x \times 0 = 0, \qquad x \times s(i) = x \times i + x,$$

i.e.

$$h(x, 0) = 0, \qquad h(x, s(i)) = h(x, i) + x.$$

$\times$ is also computable: It satisfies

$$
\begin{array}{lll}
 & x \times 0 = 0, & x \times s(i) = x \times i + x, \\
\text{i.e.} & h(x,0) = 0, & h(x,s(i)) = h(x,i) + x. \\
\text{So} & h(x,0) = f(x), & h(x,s(i)) = g(x,i,h(x,i)) \\
\text{if} & f(x) = 0, & g(x,i,k) = k + x.
\end{array}
$$

$\times$ is also computable: It satisfies

$$
\begin{array}{llll}
 & x \times 0 = 0, & & x \times s(i) = x \times i + x, \\
\text{i.e.} & h(x,0) = 0, & & h(x,s(i)) = h(x,i) + x. \\
\text{So} & h(x,0) = f(x), & & h(x,s(i)) = g(x,i,h(x,i)) \\
\text{if} & f(x) = 0, & & g(x,i,k) = k + x.
\end{array}
$$

$\times$ is also computable: It satisfies

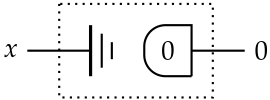
$$
\begin{array}{lll}
 & x \times 0 = 0, & x \times s(i) = x \times i + x, \\
\text{i.e.} & h(x, 0) = 0, & h(x, s(i)) = h(x, i) + x. \\
\text{So} & h(x, 0) = f(x), & h(x, s(i)) = g(x, i, h(x, i)) \\
\text{if} & f(x) = 0, & g(x, i, k) = k + x.
\end{array}
$$

i.e., $\times$ is defined by primitive recursion from these f and g.

$\times$ is also computable: It satisfies

$$
\begin{array}{lll}
 & x \times 0 = 0, & x \times s(i) = x \times i + x, \\
\text{i.e.} & h(x,0) = 0, & h(x,s(i)) = h(x,i) + x. \\
\text{So} & h(x,0) = f(x), & h(x,s(i)) = g(x,i,h(x,i)) \\
\text{if} & f(x) = 0, & g(x,i,k) = k + x.
\end{array}
$$

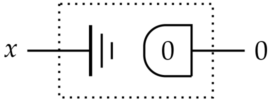
i.e., $\times$ is defined by primitive recursion from these f and g.
But these f and g are computable because they are

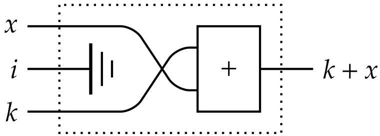


$\times$ is also computable: It satisfies

$$
\begin{array}{lrl}
 & x \times 0 = 0, & x \times s(i) = x \times i + x, \\
\text{i.e.} & h(x,0) = 0, & h(x,s(i)) = h(x,i) + x. \\
\text{So} & h(x,0) = f(x), & h(x,s(i)) = g(x,i,h(x,i)) \\
\text{if} & f(x) = 0, & g(x,i,k) = k + x.
\end{array}
$$

i.e., $\times$ is defined by primitive recursion from these f and g.
But these f and g are computable because they are

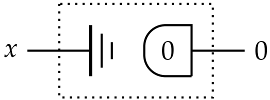


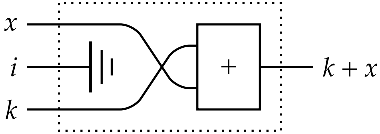


$\times$ is also computable: It satisfies

$$\begin{array}{llll} & x \times 0 = 0, & & x \times s(i) = x \times i + x, \\ \text{i.e.} & h(x,0) = 0, & & h(x,s(i)) = h(x,i) + x. \\ \text{So} & h(x,0) = f(x), & & h(x,s(i)) = g(x,i,h(x,i)) \\ \text{if} & f(x) = 0, & & g(x,i,k) = k + x. \end{array}$$

i.e., $\times$ is defined by primitive recursion from these f and g.
But these f and g are computable because they are





Therefore $\times$ is computable!

Primitive Recursive Functions

So far we have introduced:

1. six basic functions and
2. two operations of building more and more complicated functions: (serial & parallel) composition and primitive recursion.

Primitive Recursive Functions

So far we have introduced:

1. six basic functions and
2. two operations of building more and more complicated functions: (serial & parallel) composition and primitive recursion.

Definition. We say that a function is "primitive recursive" if (& only if)

1. it is one of the six basic functions, or
2. it can be obtained from other primitive recursive functions by composition and primitive recursion.

Primitive Recursive Functions

So far we have introduced:

1. six basic functions and
2. two operations of building more and more complicated functions: (serial & parallel) composition and primitive recursion.

Definition. We say that a function is "primitive recursive" if (& only if)

1. it is one of the six basic functions, or
2. it can be obtained from other primitive recursive functions by composition and primitive recursion.

How does this criterion of computability compare to Turing's?

Primitive Recursive Functions

So far we have introduced:

1. six basic functions and
2. two operations of building more and more complicated functions: (serial & parallel) composition and primitive recursion.

Definition. We say that a function is "primitive recursive" if (& only if)

1. it is one of the six basic functions, or
2. it can be obtained from other primitive recursive functions by composition and primitive recursion.

How does this criterion of computability compare to Turing's?

Every primitive recursive function is Turing computable,

Primitive Recursive Functions

So far we have introduced:

1. six basic functions and
2. two operations of building more and more complicated functions: (serial & parallel) composition and primitive recursion.

Definition. We say that a function is "primitive recursive" if (& only if)

1. it is one of the six basic functions, or
2. it can be obtained from other primitive recursive functions by composition and primitive recursion.

How does this criterion of computability compare to Turing's?

Every primitive recursive function is Turing computable,
but the converse is not the case: there are Turing computable functions that are not primitive recursive (i.e. cannot be obtained by 1 and 2).

Primitive Recursive Functions

So far we have introduced:

1. six basic functions and
2. two operations of building more and more complicated functions: (serial & parallel) composition and primitive recursion.

Definition. We say that a function is "primitive recursive" if (& only if)

1. it is one of the six basic functions, or
2. it can be obtained from other primitive recursive functions by composition and primitive recursion.

How does this criterion of computability compare to Turing's?

Every primitive recursive function is Turing computable,
but the converse is not the case: there are Turing computable functions that are not primitive recursive (i.e. cannot be obtained by 1 and 2).

We need one more operation to match the power of Turing machines.

Recursive Functions:
Partial Recursive Functions

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation,

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

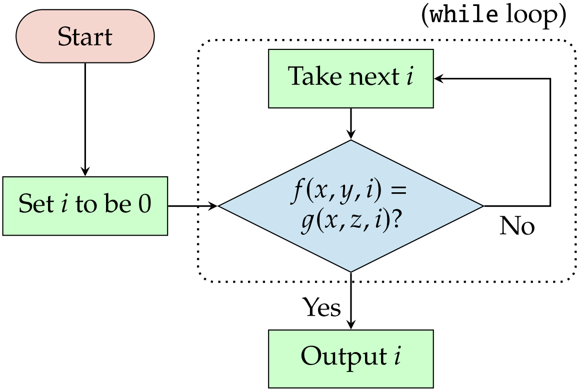
$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```



Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3$?

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3\,?$

0

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3$?

$0 \quad\quad 3$

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3\,?$

$0 \quad \neq \quad 3$

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3\,?$ $\quad$ $1 \times 2 = 1 + 3\,?$

$0 \quad \neq \quad 3$

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3\,?$ $\quad$ $1 \times 2 = 1 + 3\,?$

$0 \neq 3$ $\quad$ $2 \neq 4$

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3\,?$	$1 \times 2 = 1 + 3\,?$	$2 \times 2 = 2 + 3\,?$
$0 \neq 3$	$2 \neq 4$	

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3$?	$1 \times 2 = 1 + 3$?	$2 \times 2 = 2 + 3$?
$0 \neq 3$	$2 \neq 4$	$4 \neq 5$

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3$?	$1 \times 2 = 1 + 3$?	$2 \times 2 = 2 + 3$?	$3 \times 2 = 3 + 3$?
$0 \neq 3$	$2 \neq 4$	$4 \neq 5$	

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3\,?$	$1 \times 2 = 1 + 3\,?$	$2 \times 2 = 2 + 3\,?$	$3 \times 2 = 3 + 3\,?$
$0 \quad \neq \quad 3$	$2 \quad \neq \quad 4$	$4 \quad \neq \quad 5$	$6 \quad = \quad 6$

Unbounded Search

"Unbounded search" (also called "minimization") searches for a solution (indeed the smallest solution) to an equation, e.g.,

$$f(x, y, n) = g(x, z, n).$$

Regarding x, y, z as parameters, we want to find an n that makes the equation hold. There is a simple algorithm for that:

```
i := 0
while (f(x, y, i) != g(x, z, i)):
    i := i+1
return i
```

E.g., let's solve $n \times 2 = n + 3$.

$0 \times 2 = 0 + 3\,?$	$1 \times 2 = 1 + 3\,?$	$2 \times 2 = 2 + 3\,?$	$3 \times 2 = 3 + 3\,?$
$0 \neq 3$	$2 \neq 4$	$4 \neq 5$	$6 = 6$

This search is computable if f and g are!

But what if there is no solution?

But what if there is no solution?

E.g., let's try the same trick to solve $n \times 3 = n + 3$.

But what if there is no solution?

E.g., let's try the same trick to solve $n \times 3 = n + 3$.

$$0 \times 3 = 0 \neq 3 = 0 + 3,$$

But what if there is no solution?

E.g., let's try the same trick to solve $n \times 3 = n + 3$.

$$0 \times 3 = 0 \neq 3 = 0 + 3,$$

$$1 \times 3 = 3 \neq 4 = 1 + 3,$$

But what if there is no solution?

E.g., let's try the same trick to solve $n \times 3 = n + 3$.

$$\begin{aligned} 0 \times 3 = 0 &\neq 3 = 0 + 3, \\ 1 \times 3 = 3 &\neq 4 = 1 + 3, \\ 2 \times 3 = 6 &\neq 5 = 2 + 3, \end{aligned}$$

But what if there is no solution?

E.g., let's try the same trick to solve $n \times 3 = n + 3$.

$$0 \times 3 = 0 \neq 3 = 0 + 3,$$

$$1 \times 3 = 3 \neq 4 = 1 + 3,$$

$$2 \times 3 = 6 \neq 5 = 2 + 3,$$

$$3 \times 3 = 9 \neq 6 = 3 + 3,$$

But what if there is no solution?

E.g., let's try the same trick to solve $n \times 3 = n + 3$.

$$\begin{aligned} 0 \times 3 = 0 &\neq 3 = 0 + 3, \\ 1 \times 3 = 3 &\neq 4 = 1 + 3, \\ 2 \times 3 = 6 &\neq 5 = 2 + 3, \\ 3 \times 3 = 9 &\neq 6 = 3 + 3, \\ 4 \times 3 = 12 &\neq 7 = 4 + 3, \\ &\vdots \end{aligned}$$

But what if there is no solution?

E.g., let's try the same trick to solve $n \times 3 = n + 3$.

$$\begin{aligned} 0 \times 3 = 0 &\neq 3 = 0 + 3, \\ 1 \times 3 = 3 &\neq 4 = 1 + 3, \\ 2 \times 3 = 6 &\neq 5 = 2 + 3, \\ 3 \times 3 = 9 &\neq 6 = 3 + 3, \\ 4 \times 3 = 12 &\neq 7 = 4 + 3, \\ &\vdots \end{aligned}$$

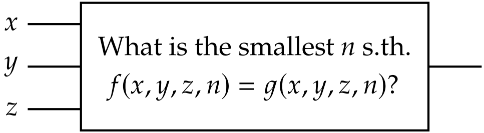
This process never halts.

Upshot: according to a model of computation involving unbounded search, some computation fails to halt due to bad `while` loops.
(But we already know that some computation fails to halt, since some Turing machines do on some inputs.)

Upshot: according to a model of computation involving unbounded search, some computation fails to halt due to bad `while` loops.
(But we already know that some computation fails to halt, since some Turing machines do on some inputs.)

To sum up,

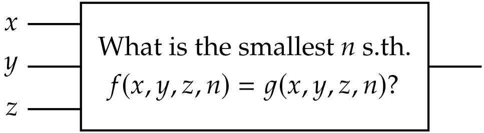
1. We have a function: "Given inputs x, y, z, output the smallest solution n to the equation $f(x, y, z, n) = g(x, y, z, n)$".



Upshot: according to a model of computation involving unbounded search, some computation fails to halt due to bad `while` loops.
(But we already know that some computation fails to halt, since some Turing machines do on some inputs.)

To sum up,

1. We have a function: "Given inputs x, y, z, output the smallest solution n to the equation $f(x,y,z,n) = g(x,y,z,n)$".



2. But this function may be partial and not total, since depending on inputs, it may fail to output values.

It is enough to write

$$f(x, y, z, n) = 0$$

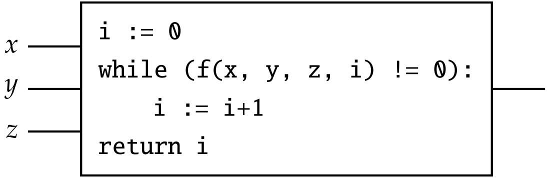
for equations, since we can always move everything to the LHS.

It is enough to write

$$f(x, y, z, n) = 0$$

for equations, since we can always move everything to the LHS.

Definition. Given a (partial or total) function $f(x, y, z, n)$, we write $\mu f(x, y, z)$ for the (partial or total) function



It is enough to write

$$f(x, y, z, n) = 0$$

for equations, since we can always move everything to the LHS.

Definition. Given a (partial or total) function $f(x, y, z, n)$, we write $\mu f(x, y, z)$ for the (partial or total) function

x
y
z

```
i := 0
while (f(x, y, z, i) != 0):
    i := i+1
return i
```

We may note that there are two ways $\mu f(x, y, z)$ can end up undefined:

It is enough to write

$$f(x, y, z, n) = 0$$

for equations, since we can always move everything to the LHS.

Definition. Given a (partial or total) function $f(x, y, z, n)$, we write $\mu f(x, y, z)$ for the (partial or total) function

```
i := 0
while (f(x, y, z, i) != 0):
    i := i+1
return i
```

We may note that there are two ways $\mu f(x, y, z)$ can end up undefined:

1. Even if $f(x, y, z, n)$ is defined for all n, the equation $f(x, y, z, n) = 0$ has no solution.
2. Even if $f(x, y, z, n) = 0$ has solutions, for some i smaller than them $f(x, y, z, i)$ is undefined.

Partial Recursive Functions

Definition. We say that a function is "primitive recursive" if (& only if)

1. it is one of the zero, successor function, discarding, duplication, identity, swap, or
2. it can be obtained from other primitive recursive functions by (serial & parallel) composition and primitive recursion.

Partial Recursive Functions

Definition. We say that a function is "primitive recursive" if (& only if)

1. it is one of the zero, successor function, discarding, duplication, identity, swap, or
2. it can be obtained from other primitive recursive functions by (serial & parallel) composition and primitive recursion.

Definition. We say that a partial function is "partial recursive" if (& only if) either 1 above or

3. it can be obtained from other partial recursive functions by (serial & parallel) composition, primitive recursion, and unbounded search.

Partial Recursive Functions

Definition. We say that a function is "primitive recursive" if (& only if)

1. it is one of the zero, successor function, discarding, duplication, identity, swap, or
2. it can be obtained from other primitive recursive functions by (serial & parallel) composition and primitive recursion.

Definition. We say that a partial function is "partial recursive" if (& only if) either 1 above or

3. it can be obtained from other partial recursive functions by (serial & parallel) composition, primitive recursion, and unbounded search.

Theorem. Given any partial function f,

$$f \text{ is Turing computable } \iff f \text{ is partial recursive.}$$

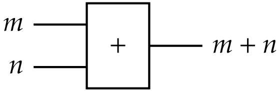
Recursive Functions: One Last Bit of Reflection

Computation as symbol manipulation.

In computing, expressions matter:

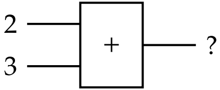
Computation as symbol manipulation.

In computing, expressions matter:



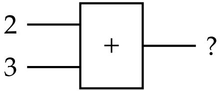
Computation as symbol manipulation.

In computing, expressions matter:



Computation as symbol manipulation.

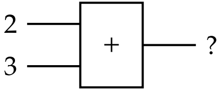
In computing, expressions matter:



- You: "What is 2 + 3?" Calculator: "5."

Computation as symbol manipulation.

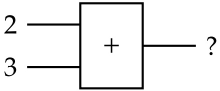
In computing, expressions matter:



- You: "What is $2 + 3$?" Calculator: "5."
- Mathematicians (usually): "$2 + 3 = 5$."

Computation as symbol manipulation.

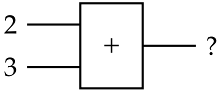
In computing, expressions matter:



- You: "What is $2 + 3$?" Calculator: "5."
- Mathematicians (usually): "$2 + 3 = 5$."
- Logicians (usually): " '$2 + 3$' and '5' refer to the same number."

Computation as symbol manipulation.

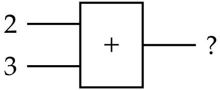
In computing, expressions matter:



- You: "What is $2 + 3$?" Calculator: "5."
- Mathematicians (usually): "$2 + 3 = 5$."
- Logicians (usually): " '$2 + 3$' and '5' refer to the same number."
- You: "What is $2 + 3$?" Sili: "$2 + 3$."

Computation as symbol manipulation.

In computing, expressions matter:

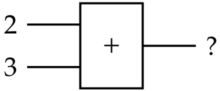


- You: “What is 2 + 3?” Calculator: “5.”
- Mathematicians (usually): “2 + 3 = 5.”
- Logicians (usually): “ ‘2 + 3’ and ‘5’ refer to the same number.”
- You: “What is 2 + 3?” Sili: “2 + 3.”

Moral: an important part of computation is to transform one expression, “2 + 3”, into another, “5”.

Computation as symbol manipulation.

In computing, expressions matter:



- You: "What is $2 + 3$?" Calculator: "5."
- Mathematicians (usually): "$2 + 3 = 5$."
- Logicians (usually): " '$2 + 3$' and '5' refer to the same number."
- You: "What is $2 + 3$?" Sili: "$2 + 3$."

Moral: an important part of computation is to transform one expression, "$2 + 3$", into another, "5".

- You: "110111." A Turing machine: "11111."

Recursive functions do this, too, as we can see when we unpack primitive recursion.

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$
$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$

$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$ss0 + sss0$$

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$

$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$ss0 + sss0$$

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$

$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$\begin{aligned} & ss0 + sss0 \\ = \ & s(ss0 + ss0) \end{aligned}$$

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$

$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$\begin{aligned} & ss0 + sss0 \\ = {} & s(ss0 + ss0) \end{aligned}$$

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$

$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$\begin{aligned} & ss0 + sss0 \\ = {} & s(ss0 + ss0) \\ = {} & ss(ss0 + s0) \end{aligned}$$

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$

$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$\begin{aligned} &ss0 + sss0 \\ &= s(ss0 + ss0) \\ &= ss(ss0 + s0) \end{aligned}$$

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$

$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$\begin{aligned} &ss0 + sss0 \\ &= s(ss0 + ss0) \\ &= ss(ss0 + s0) \\ &= sss(ss0 + 0) \end{aligned}$$

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$
$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$\begin{aligned} & ss0 + sss0 \\ = {} & s(ss0 + ss0) \\ = {} & ss(ss0 + s0) \\ = {} & sss(ss0 + 0) \end{aligned}$$

Recursive functions do this, too, as we can see when we unpack primitive recursion.

$$x + 0 = x$$

$$x + s(i) = s(x + i)$$

"$2 + 3$" is transformed to "5":

$$\begin{aligned} & ss0 + sss0 \\ = {} & s(ss0 + ss0) \\ = {} & ss(ss0 + s0) \\ = {} & sss(ss0 + 0) \\ = {} & sssss0 \end{aligned}$$

$$x \times 0 = 0$$
$$x \times s(i) = (x \times i) + x$$

"$(2 + 3) \times 2$" is transformed to "10":

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2+3) \times 2$" is transformed to "10":

$$(ss0 + sss0) \times ss0$$

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2 + 3) \times 2$" is transformed to "10":

$$\begin{aligned} &(ss0 + sss0) \times ss0 \\ &\vdots \\ = {} & sssss0 \times ss0 \end{aligned}$$

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2+3) \times 2$" is transformed to "10":

$$\begin{aligned} &(ss0 + sss0) \times ss0 \\ &\vdots \\ = {}& sssss0 \times ss0 \end{aligned}$$

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2+3) \times 2$" is transformed to "10":

$$\begin{aligned} &(ss0 + sss0) \times ss0 \\ &\vdots \\ &= sssss0 \times ss0 \\ &= (sssss0 \times s0) + sssss0 \end{aligned}$$

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2 + 3) \times 2$" is transformed to "10":

$$\begin{aligned} &(ss0 + sss0) \times ss0 \\ &\vdots \\ &= sssss0 \times ss0 \\ &= (sssss0 \times s0) + sssss0 \end{aligned}$$

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2+3) \times 2$" is transformed to "10":

$$\begin{aligned}
&(ss0 + sss0) \times ss0 \\
&\vdots \\
&= sssss0 \times ss0 \\
&= (sssss0 \times s0) + sssss0 \\
&= ((sssss0 \times 0) + sssss0) + sssss0
\end{aligned}$$

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2 + 3) \times 2$" is transformed to "10":

$$\begin{aligned}
&(ss0 + sss0) \times ss0 \\
&\vdots \\
&= sssss0 \times ss0 \\
&= (sssss0 \times s0) + sssss0 \\
&= ((sssss0 \times 0) + sssss0) + sssss0
\end{aligned}$$

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2+3) \times 2$" is transformed to "10":

$$\begin{aligned}
&(ss0 + sss0) \times ss0 \\
&\vdots \\
&= sssss0 \times ss0 \\
&= (sssss0 \times s0) + sssss0 \\
&= ((sssss0 \times 0) + sssss0) + sssss0 \\
&= (0 + sssss0) + sssss0
\end{aligned}$$

$$x \times 0 = 0$$

$$x \times s(i) = (x \times i) + x$$

"$(2+3) \times 2$" is transformed to "10":

$$\begin{aligned}
&(ss0 + sss0) \times ss0 \\
&\vdots \\
&= sssss0 \times ss0 \\
&= (sssss0 \times s0) + sssss0 \\
&= ((sssss0 \times 0) + sssss0) + sssss0 \\
&= (0 + sssss0) + sssss0 \\
&\vdots \\
&= sssss0 + sssss0
\end{aligned}$$

$$x \times 0 = 0$$
$$x \times s(i) = (x \times i) + x$$

"$(2+3) \times 2$" is transformed to "10":

$$\begin{aligned}
& (ss0 + sss0) \times ss0 \\
& \vdots \\
&= sssss0 \times ss0 \\
&= (sssss0 \times s0) + sssss0 \\
&= ((sssss0 \times 0) + sssss0) + sssss0 \\
&= (0 + sssss0) + sssss0 \\
& \vdots \\
&= {\color{red} sssss0 + sssss0} \\
& \vdots \\
&= {\color{green} ssssssssss0}
\end{aligned}$$