COMP9020 Week 4
Term 3, 2019
Recursion

Summary of topics

- Recursion
- Recursive Data Types
- Recursive programming

Summary of topics

- Recursion
- Recursive Data Types
- Recursive programming

Recursion

Fundamental concept in Computer Science

- Recursion in algorithms: Solving problems by reducing to smaller cases
  - Factorial
  - Towers of Hanoi
  - Mergesort, Quicksort

Recursion

Fundamental concept in Computer Science

- Recursion in algorithms: Solving problems by reducing to smaller cases
  - Factorial
  - Towers of Hanoi
  - Mergesort, Quicksort
- Recursion in data structures: Finite definitions of **arbitrarily large** objects
  - Natural numbers
  - Words
  - Linked lists
  - Formulas
  - Binary trees

Recursion

Fundamental concept in Computer Science

- Recursion in algorithms: Solving problems by reducing to smaller cases
  - Factorial
  - Towers of Hanoi
  - Mergesort, Quicksort
- Recursion in data structures: Finite definitions of **arbitrarily large** objects
  - Natural numbers
  - Words
  - Linked lists
  - Formulas
  - Binary trees
- Analysis of recursion: Proving properties
  - Recursive sequences (e.g. Fibonacci sequence)
  - Structural induction

Recursion

Consists of a basis (B) and recursive process (R).
A sequence/object/algorithm is recursively defined when (typically)
(B) some initial terms are specified, perhaps only the first one;
(R) later terms stated as functional expressions of the earlier terms.

NB

(R) also called **recurrence formula (especially when dealing with sequences)**

Example: Factorial

Example

Factorial:

(B) $0! = 1$

(R) $(n+1)! = (n+1) \cdot n!$

$\texttt{fact}(n)$:

(B) $\quad$ $\texttt{if}(n = 0)$: 1

(R) $\quad$ $\texttt{else}$: $n * \texttt{fact}(n-1)$

Example: Euclid's gcd algorithm

Example

$$\gcd(m,n) = \begin{cases} m & \text{if } m = n \\ \gcd(m-n,n) & \text{if } m > n \\ \gcd(m,n-m) & \text{if } m < n \end{cases}$$

Example: Towers of Hanoi

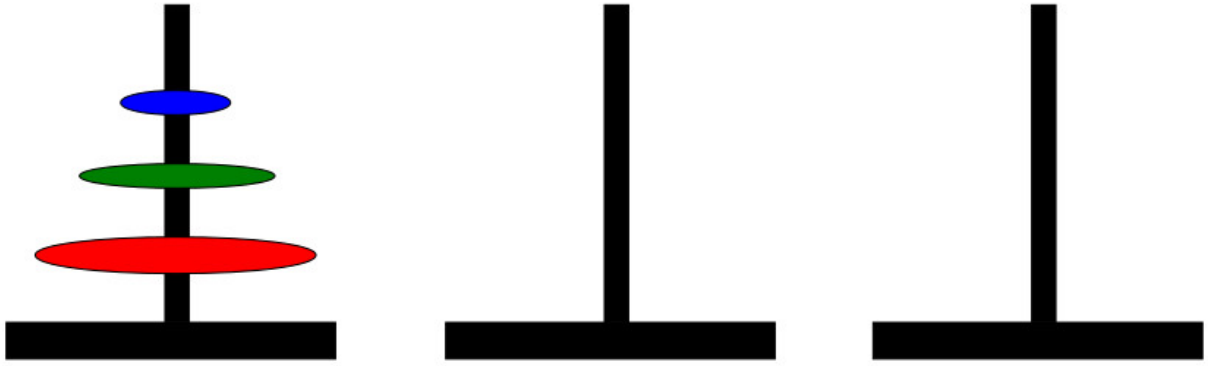
- There are 3 towers (pegs)
- n disks of decreasing size placed on the first tower
- You need to move all disks from the first tower to the last tower
- Larger disks cannot be placed on top of smaller disks
- The third tower can be used to temporarily hold disks

Example: Towers of Hanoi

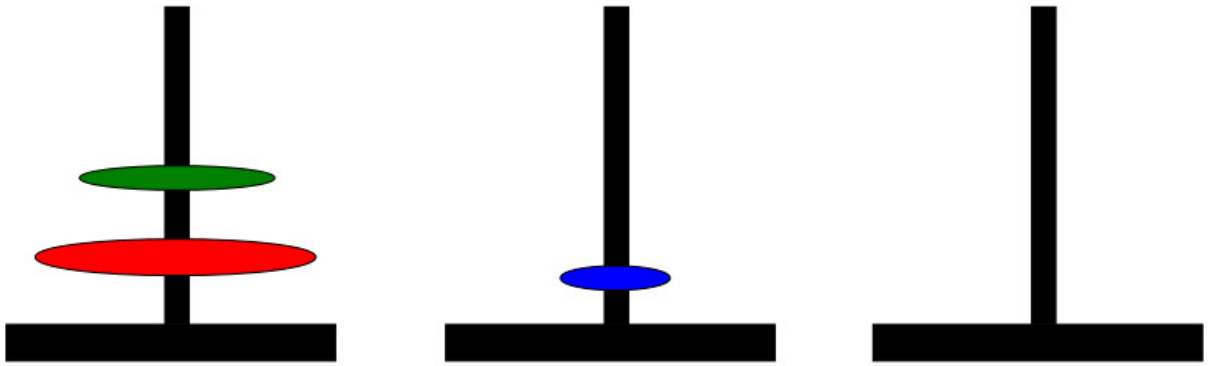
Questions

- Describe a general solution for n disks
- How many moves does it take?

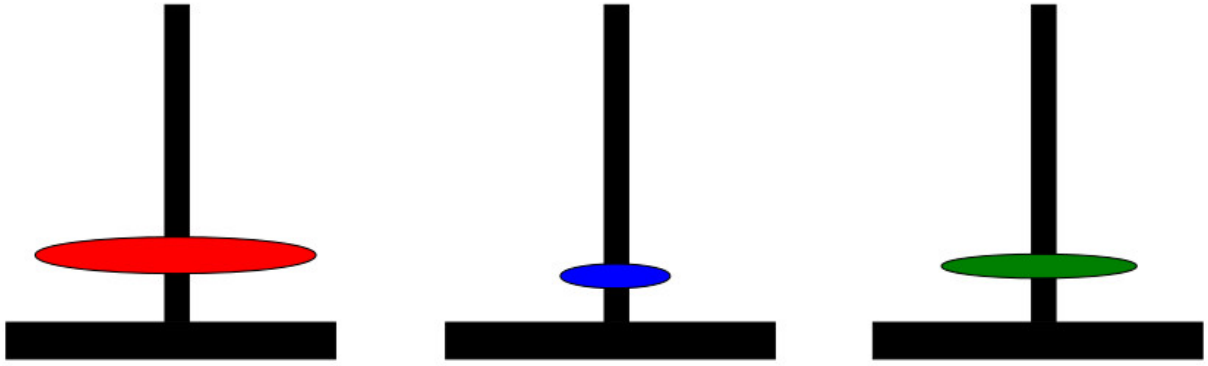
Example: Towers of Hanoi



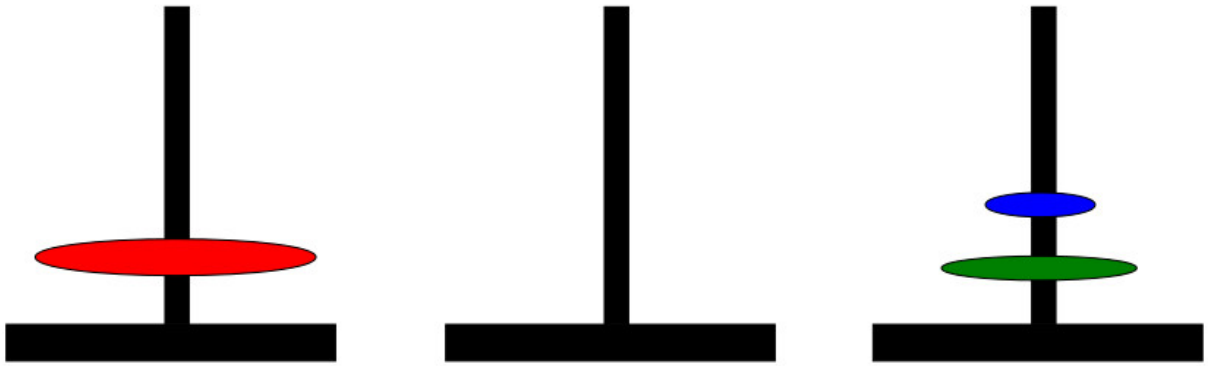
Example: Towers of Hanoi



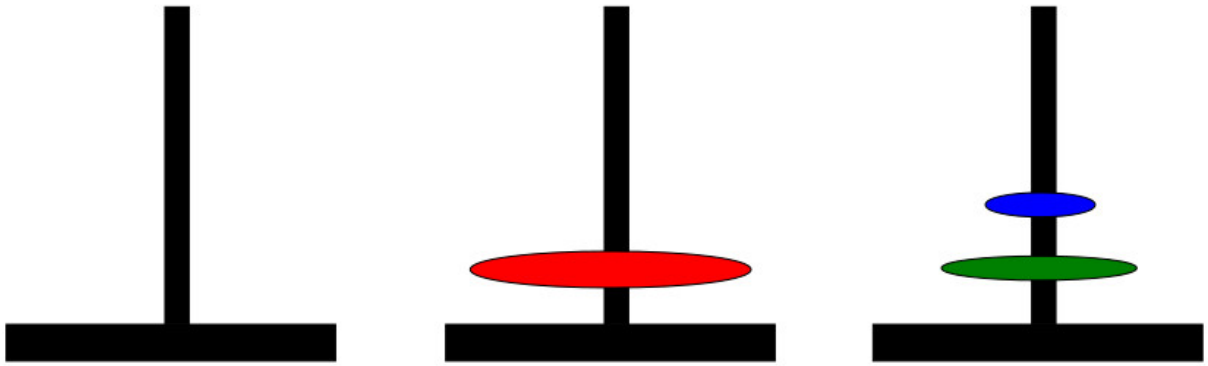
Example: Towers of Hanoi



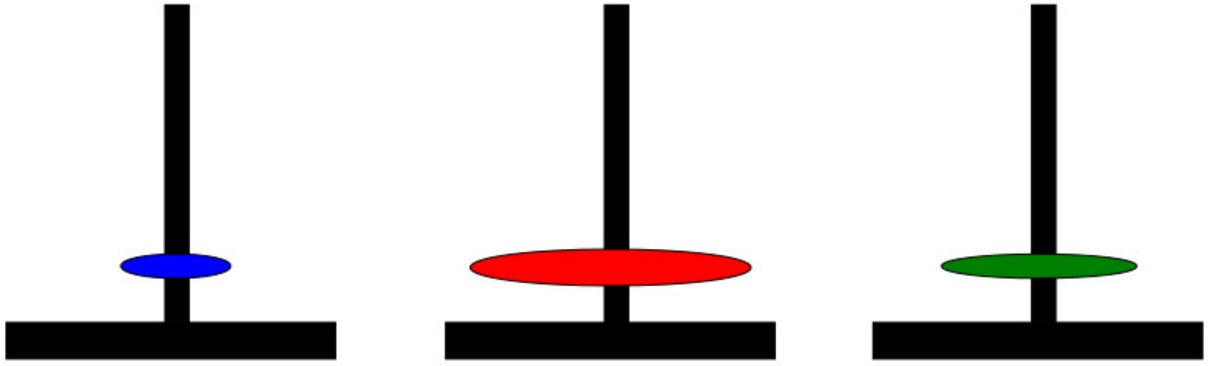
Example: Towers of Hanoi



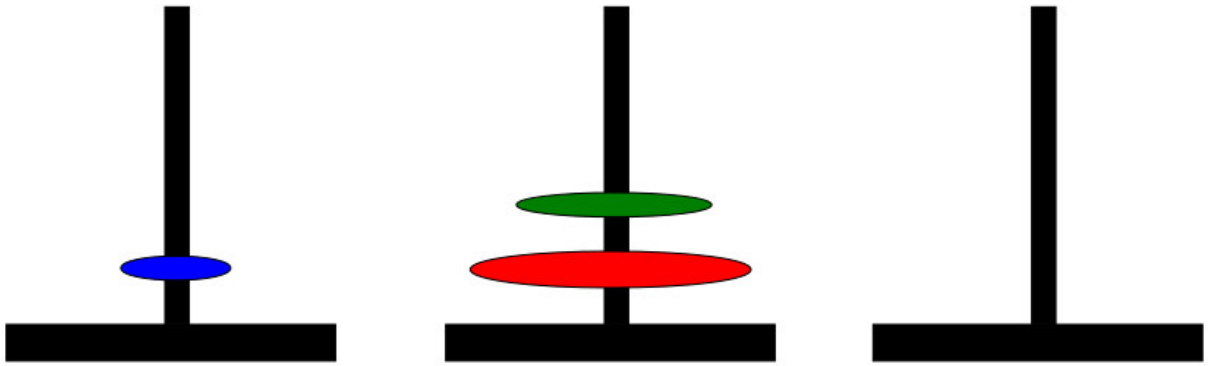
Example: Towers of Hanoi



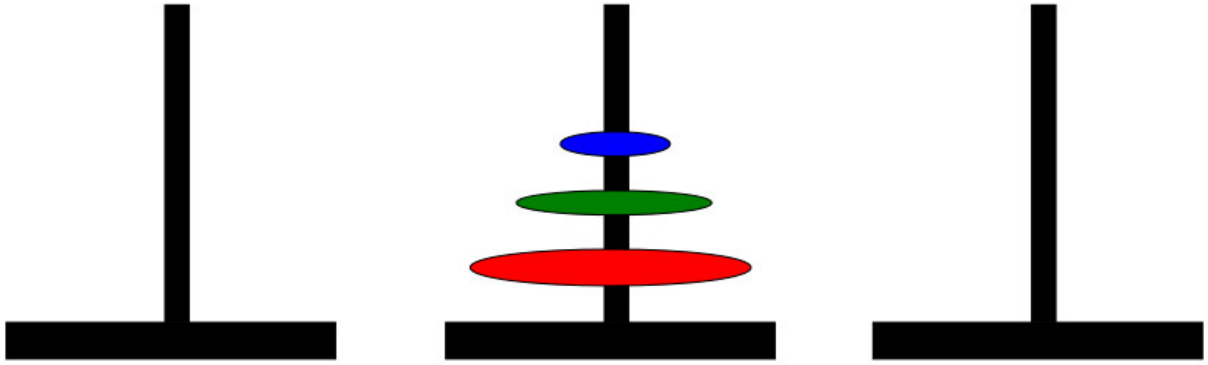
Example: Towers of Hanoi



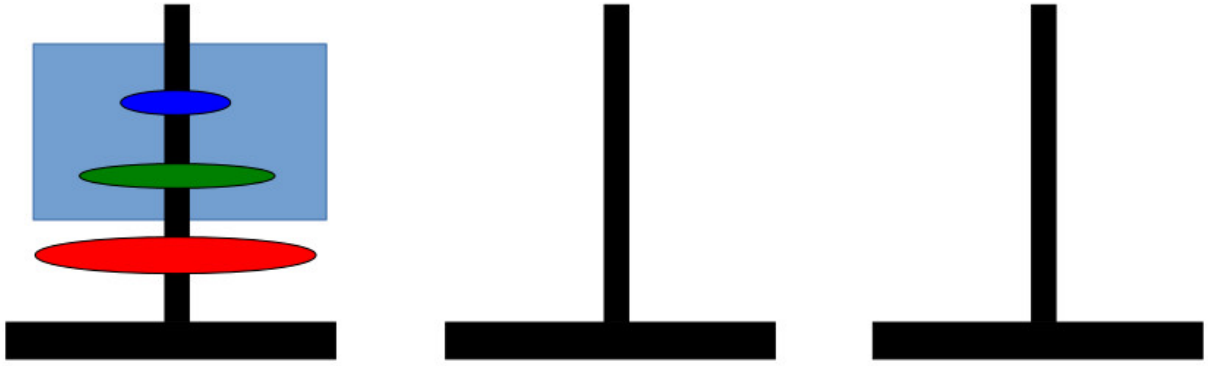
Example: Towers of Hanoi



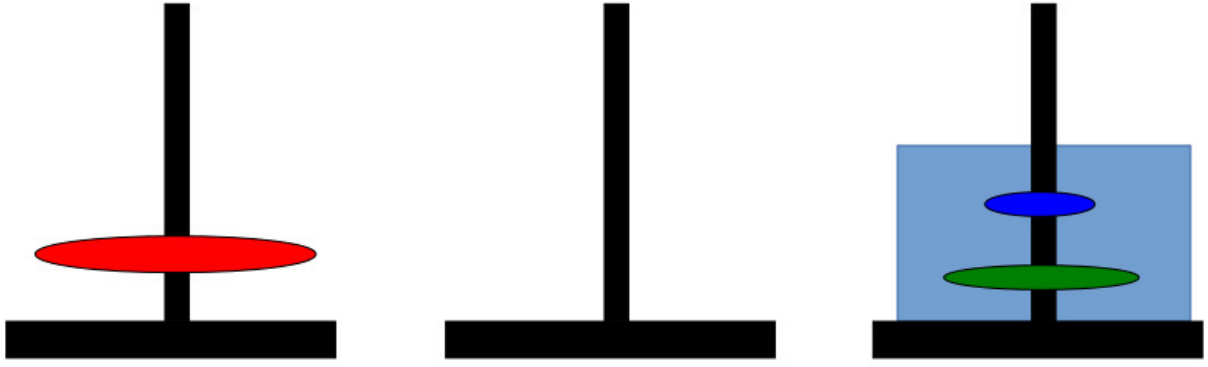
Example: Towers of Hanoi



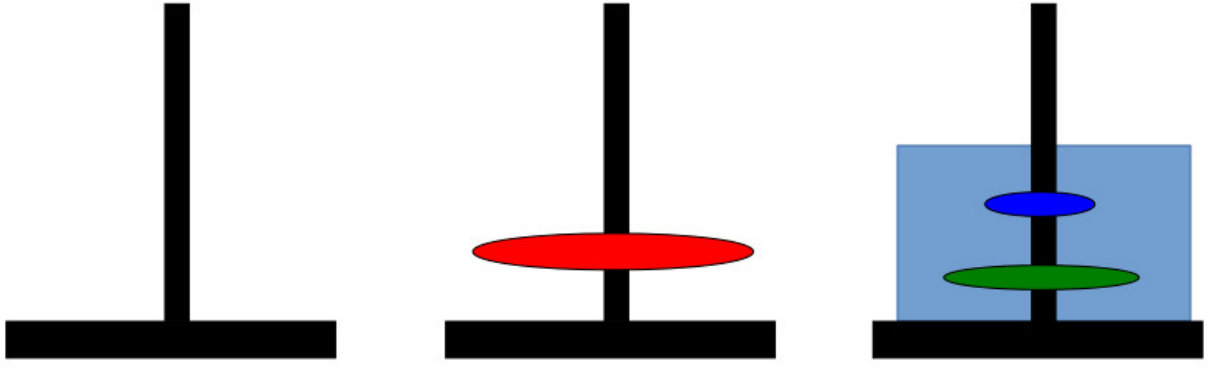
Example: Towers of Hanoi



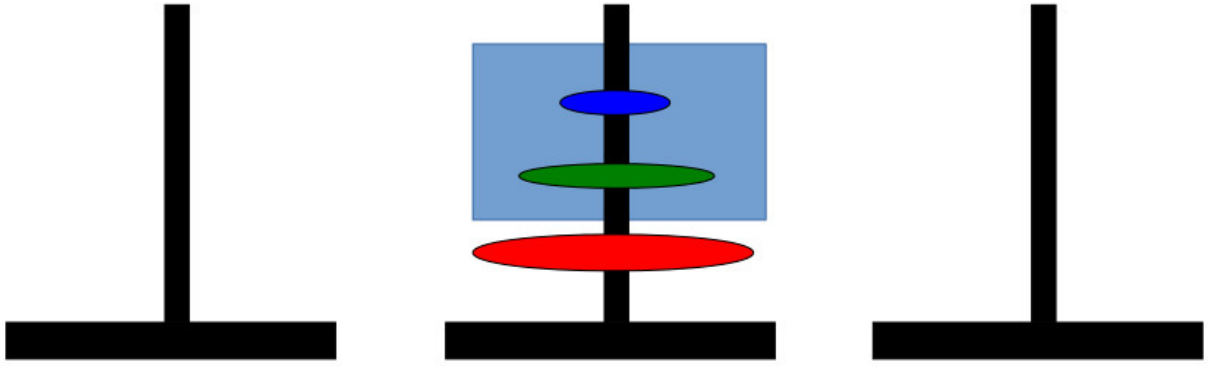
Example: Towers of Hanoi



Example: Towers of Hanoi



Example: Towers of Hanoi



Example: Towers of Hanoi

Questions

- Describe a general solution for n disks
- How many moves does it take? ?

Summary of topics

- Recursion
- Recursive Data Types
- Recursive programming

Example: Natural numbers

Example

A natural number is either 0 (B) or one more than a natural number (R).

Formal definition of $\mathbb{N}$:

- (B) $0 \in \mathbb{N}$
- (R) If $n \in \mathbb{N}$ then $(n+1) \in \mathbb{N}$

Example: Fibonacci numbers

Example 斐波纳契 5 8 13 21 34 55

The Fibonacci sequence starts $0, 1, 1, 2, 3, \ldots$ where, after $0, 1$, each term is the sum of the previous two terms.

Formally, the set of Fibonacci numbers: $\mathbb{F} = \{F_n : n \in \mathbb{N}\}$, where the n-th Fibonacci number F_n is defined as:

- (B) $F_0 = 0$,
- (B) $F_1 = 1$,
- (I) $F_n = F_{n-1} + F_{n-2}$

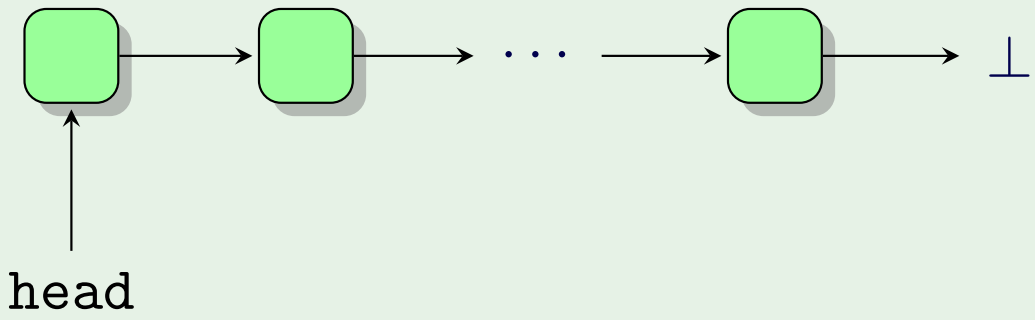
NB

Could also define the Fibonacci sequence as a function $\text{FIB} : \mathbb{N} \to \mathbb{F}$. *Choice of perspective depends on what structure you view as your base object (***ground type***).*

Example: Linked lists

Example

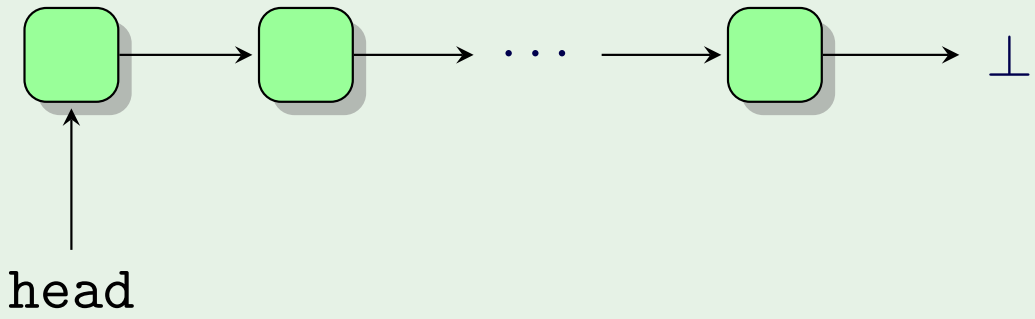
A linked list is zero or more linked list nodes:



Example: Linked lists

Example

A linked list is zero or more linked list nodes:



In C:

```
struct node{
   int data;
   struct node *next;
}
```

Example: Linked lists

Example

We can view the linked list **structure** abstractly. A linked list is either:

- (B) an empty list, or
- (R) an ordered pair $(\texttt{Data}, \text{List})$.

Example: Words over Σ

Example

A word over an alphabet Σ is either λ (B) or a symbol from Σ followed by a word (R).

Formal definition of Σ^*:

- (B) $\lambda \in \Sigma^*$
- (R) If $w \in \Sigma^*$ then $aw \in \Sigma^*$ for all $a \in \Sigma$

NB

This matches the recursive definition of a **Linked List** *data type.*

Example: Propositional formulas

Example

A well-formed formula (wff) over a set of propositional variables, $\textsc{Prop}$ is defined as:

- (B) $\top$ is a wff
- (B) $\bot$ is a wff
- (B) p is a wff for all $p \in \textsc{Prop}$
- (R) If φ is a wff then $\neg\varphi$ is a wff
- (R) If φ and ψ are wffs then:
  - $(\varphi \wedge \psi)$,
  - $(\varphi \vee \psi)$,
  - $(\varphi \rightarrow \psi)$, and
  - $(\varphi \leftrightarrow \psi)$ are wffs.

Exercises

Exercises

4.4.4 (a) Give a recursive definition for the sequence

$$(2,\ 4,\ 16,\ 256,\ \ldots)$$

(b) Give a recursive definition for the sequence

$$(2,\ 4,\ 16,\ 65536,\ \ldots)$$

Exercises

Exercises

4.4.4 (a) Give a recursive definition for the sequence

$$(2,\ 4,\ 16,\ 256,\ \ldots)$$

An = a_(n–1)^2

?

(b) Give a recursive definition for the sequence

$$(2,\ 4,\ 16,\ 65536,\ \ldots)$$

An = 2^(a_(n –1))

?

Summary of topics

- Recursion
- Recursive Data Types
- Recursive programming

Programming over recursive datatypes

Recursive datatypes make recursive programming/functions easy.

Example

The factorial function:

$$
\begin{array}{ll}
 & \texttt{fact}(n): \\
(B) & \quad \texttt{if}(n = 0)\text{: } 1 \\
(R) & \quad \texttt{else}\text{: } n * \texttt{fact}(n-1)
\end{array}
$$

Programming over recursive datatypes

Recursive datatypes make recursive programming/functions easy.

Example

Summing the first n natural numbers:

$$
\begin{array}{ll}
 & \texttt{sum}(n)\text{:} \\
(B) & \quad \texttt{if}(n = 0)\text{: } 0 \\
(R) & \quad \texttt{else}\text{: } n + \texttt{sum}(n-1)
\end{array}
$$

Programming over recursive datatypes

Recursive datatypes make recursive programming/functions easy.

Example

Summing elements of a linked list:

```
          sum(L):
(B)         if(L.isEmpty()):
              return 0
(R)         else:
              return L.data + sum(L.next)
```

Programming over recursive datatypes

Recursive datatypes make recursive programming/functions easy.

Example

Concatenation of words (defining wv):

For all $w, v \in \Sigma^*$ and $a \in \Sigma$:

(B) $\lambda v = v$

(R) $(aw)v = a(wv)$

Programming over recursive datatypes

Recursive datatypes make recursive programming/functions easy.

Example

Length of words:

$$\begin{array}{ll} (B) & \text{length}(\lambda) = 0 \\ (R) & \text{length}(aw) = 1 + \text{length}(w) \end{array}$$

Programming over recursive datatypes

Recursive datatypes make recursive programming/functions easy.

Example

"Evaluation" of a propositional formula

Correctness of Recursive Definition

A recurrence formula is correct if the computation of any later term can be reduced to the initial values given in (B).

Example (Incorrect definition)

- Function $g(n)$ is defined recursively by

$$g(n) = g(g(n-1)-1)+1, \qquad g(0) = 2.$$

The definition of $g(n)$ is incomplete — the recursion may not terminate:
Attempt to compute $g(1)$ gives

$$g(1) = g(g(0)-1)+1 = g(1)+1 = \ldots = g(1)+1+1+1\ldots$$

When implemented, it leads to an overflow; most static analyses cannot detect this kind of ill-defined recursion.

Example (continued)

However, the definition could be repaired. For example, we can add the specification specify $g(1) = 2$.

Then $g(2) = g(2-1) + 1 = 3$,
$g(3) = g(g(2)-1) + 1 = g(3-1) + 1 = 4$,
$\ldots$

In fact, by induction $\ldots$ $g(n) = n + 1$

This illustrates a very important principle: the boundary (limiting) cases of the definition are evaluated *before* applying the recursive construction.

Example

Function $f(n)$ is defined by

$$f(n) = f(\lceil n/2 \rceil), \quad f(0) = 1$$

When evaluated for $n = 1$ it leads to

$$f(1) = f(1) = f(1) = \ldots$$

This one can also be repaired. For example, one could specify that $f(1) = 1$.
This would lead to a constant function $f(n) = 1$ for all $n \geq 0$.

Mutual Recursion

Several more sophisticated programs employ a technique of two procedures calling each other. Of course, it should be designed so that each consecutive call refers to ever smaller parameters, so that the entire process terminates. This method is often used in computer graphics, in particular for generating fractal images (basis of various imaginary landscapes, among others).