Definitions of Functions

- Two basic mechanisms for defining computational functions are
  - *explicit* definitions and
  - *recursive* definitions.

- An *explicit definition* of a function $f$ consists of an expression that indicates for each possible argument value $x$ how $f(x)$ is obtained from previously defined functions (and constants) by composition.

- **Examples**

  $$
  \begin{align*}
  \text{zero}(x) &= 0 \\
  \text{add3}(x) &= x + 3 \\
  \text{gt}(x, y) &= \text{if } x > y \text{ then } 1 \text{ else } 0 \\
  \chi_A(x) &= \text{if } x \in A \text{ then } 1 \text{ else } 0
  \end{align*}
  $$

  The function $\chi_A$ is called the *characteristic function* of the set $A$.

  The if-then-else function allows for case distinctions in function definitions.
Recursive Definitions

- A recursive definition of a function consists of two parts:

  **Basis**
  
  Provide explicit function values \( f(x_i) \) for one or more “smallest” arguments \( x_1, \ldots, x_k \).

  **Recursion**
  
  Define \( f(x) \) in terms of (previously defined functions and) values \( f(y) \) for arguments \( y \) “smaller” than \( x \).

- The factorial function can be defined recursively by:

  \[
  \text{fact}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  n \times \text{fact}(n-1) & \text{otherwise}
  \end{cases}
  \]

  The value of \( \text{fact}(n) \), also written \( n! \), is the number of permutations of \( n \) elements.

- The following binary function yields the greatest common divisor of its two arguments:

  \[
  \text{gcd}(a, b) = \begin{cases} 
  a & \text{if } b = 0 \\
  \text{gcd}(b, a \mod b) & \text{otherwise}
  \end{cases}
  \]

- A key part of a recursive definition is the order according to which one value is “smaller” than another value.
Recursive Evaluation

- The evaluation of a recursively defined function for a specific argument involves two operations:
  - **Substitutions** use the definition of a function $f$ to expand an application $f(x)$.
  - **Simplifications** use information about previously defined functions to evaluate subexpressions.

- **Example**

  
  $\text{fact}(5) = 5 \times \text{fact}(5 - 1)$ \hspace{1cm} \text{(substitution)}
  
  $= 5 \times \text{fact}(4)$ \hspace{1cm} \text{(simplification)}
  
  $= 5 \times (4 \times \text{fact}(4 - 1))$ \hspace{1cm} \text{(substitution)}
  
  $= 20 \times \text{fact}(3)$ \hspace{1cm} \text{(simplification)}
  
  $\vdots$
  
  $= 120$

- If a recursive definition is based on a “well-founded” order, then the evaluation process is guaranteed to terminate.

- The two concepts, definition by recursion and evaluation by substitution and simplification, form the foundation of **functional programming languages**.
Example - Squares

• A function that squares its argument can be defined *explicitly*, via multiplication, by

\[ sq(x) = x \times x, \]

or *recursively* by

\[
\text{square}(x) = \begin{cases} 
0 & \text{if } x = 0 \\
\text{square}(x - 1) + 2x - 1 & \text{else}
\end{cases}
\]

• The recursive definition yields the following function values:

\[
\begin{align*}
\text{square}(0) &= 0 \\
\text{square}(1) &= \text{square}(0) + 1 = 1 \\
\text{square}(2) &= \text{square}(1) + 3 = 4 \\
\text{square}(3) &= \text{square}(2) + 5 = 9 \\
\text{square}(4) &= \text{square}(3) + 7 = 16 \\
\end{align*}
\]

• The two definitions specify equal functions, as

\[ x \times x = (x - 1) \times (x - 1) + 2x - 1. \]

A formal proof requires a mathematical induction argument.
The following function is defined by nested recursive function applications:

\[ M(n) = \begin{cases} 
  n - 10 & \text{if } n > 100 \\
  M(M(n + 11)) & \text{if } n \leq 100
\end{cases} \]

It is known as McCarthy’s 91 function.

For example,

\[
\begin{align*}
M(99) & = M(M(110)) & (\text{since } 99 \leq 100) \\
      & = M(100) & (\text{since } 110 > 100) \\
      & = M(M(111)) & (\text{since } 100 \leq 100) \\
      & = M(101) & (\text{since } 111 > 100) \\
      & = 91 & (\text{since } 101 > 100)
\end{align*}
\]

It turns out that this function is defined for all positive integers and, remarkably, takes the value 91 for all arguments less than or equal to 101.
Partially Defined Functions

• Recursive definitions often specify only partial definitions of functions.

• For example, take the following definition of a function intended to be applied to nonnegative integers,

\[
F(x) = \text{if } x = 0 \text{ then } 0 \text{ else } F(x + 1) + 1.
\]

Most attempts at computing function values will not succeed:

\[
\begin{align*}
F(0) & = 0 \\
F(1) & = F(2) + 1 \\
& = F(3) + 2 \\
& = F(4) + 3 \\
& \vdots
\end{align*}
\]

The function is not defined for positive integers.
Another Partial Function

• What about the function $G$, defined for positive integers by

$$G(n) = \begin{cases} 
0 & \text{if } n = 1 \\
1 + G(n/2) & \text{if } n \text{ is even} \\
G(3n - 1) & \text{if } n \text{ is odd and } n > 1
\end{cases}$$

Is it defined for all nonnegative integers?

• We have

$\begin{align*}
G(1) & = 0 \\
G(2) & = 1 + G(1) = 1 \\
G(3) & = G(8) = 1 + G(4) = 1 + (1 + G(2)) = 3 \\
G(4) & = 1 + G(2) = 2 \\
G(5) & = G(14) = 1 + G(7) = 1 + G(20) \\
& = 1 + (1 + G(10)) = 3 + G(5)
\end{align*}$

• Of course, there is no integer $k$ that satisfies the identity $k = 3 + k$. In other words, $G(5)$ is undefined.
A Possibly Total Function

- The following definition uses a slight modification,

\[ H(n) = \begin{cases} 
0 & \text{if } n = 1 \\
1 + H(n/2) & \text{if } n \text{ is even} \\
H(3n + 1) & \text{if } n \text{ is odd and } n > 1 
\end{cases} \]

- It has been conjectured that this defines a function on all positive integers.

- Here are a few sample sequences of recursive calls.
  - \( H(2) \): \( H(1) \)
  - \( H(10) \): \( H(5) \)–\( H(16) \)–\( H(8) \)–\( H(4) \)–\( H(2) \)–\( H(1) \)
  - \( H(17) \): \( H(52) \)–\( H(26) \)–\( H(13) \)–\( H(40) \)–\( H(20) \)–\( H(10) \)–\( H(5) \)–\( H(16) \)–\( H(8) \)–\( H(4) \)–\( H(2) \)–\( H(1) \)
  - \( H(21) \): \( H(64) \)–\( H(32) \)–\( H(16) \)–\( H(8) \)–\( H(4) \)–\( H(2) \)–\( H(1) \)
  - \( H(35) \): \( H(106) \)–\( H(53) \)–\( H(160) \)–\( H(80) \)–\( H(40) \)–\( H(20) \)–\( H(10) \)–\( H(5) \)–\( H(16) \)–\( H(8) \)–\( H(4) \)–\( H(2) \)–\( H(1) \)

- Note that \( H(n) \) counts the number of downward steps in the sequence of recursive calls.
Definitions of Sets

- Sets can often be defined constructively via recursion. A *recursive (or inductive) definition* of a set $S$ consists of three parts:

**Basis**
Define certain objects to be elements of $S$.

**Recursion/Induction**
Give *rules* to produce new elements of $S$ from elements already known to be in $S$.

**Restriction/Closure**
Limit the elements of $S$ to those that can be obtained by the two preceding steps.

- For example, the set $\mathbb{O}$ of odd natural numbers can be defined by as follows:
  
  (i) $1 \in \mathbb{O}$ and
  (ii) if $x \in \mathbb{O}$ then $x + 2 \in \mathbb{O}$.

- Usually only the basis and induction are explicitly stated, whereas the closure condition is assumed implicitly.
Inductive Sets

- A set is called *inductive* if it can be specified by an inductive definition.

- An example of an inductive set is the set \( A = \{2^n - 1 : n \in \mathbb{N}\} \), which can be specified by:
  
  (i) \( 0 \in A \) and  
  (ii) if \( x \in A \) then \( 2x + 1 \in A \).

- The set \( B = \{2, 3, 4, 7, 8, 11, 15, 16, \ldots\} \) is also inductive. Let us first give a formal (non-recursive) definition:

  \[
  B = \{ x \in \mathbb{N} : x \mod 4 = 3 \lor \exists k \in \mathbb{N}, (k \geq 1 \land x = 2^k) \}.
  \]

  A possible recursive definition is:

  (i) \( 2 \in B \) and \( 3 \in B \), and  
  (ii) if \( x \in B \) and \( x \) is even, then \( 2x \in B \);  
  if \( x \in B \) and \( x \) is odd, then \( x + 4 \in B \).
Let $\Sigma$ be a finite set. We define the set $\Sigma^*$ of all strings (over $\Sigma$) recursively by:

1. The empty string, denoted by $\epsilon$, is an element of $\Sigma^*$.
2. If $w$ is an element of $\Sigma^*$ and $a$ is an element of $\Sigma$, then $a \cdot w$ is an element of $\Sigma^*$.

In other words, strings can be defined in terms of (i) the empty string and (ii) the function $\cdot : \Sigma \times \Sigma^* \rightarrow \Sigma^*$ that maps a pair $(a, w)$ to the string beginning with the symbol $a$ and followed by $w$.

For example, if $\Sigma = \{a, b, c, \ldots, x, y, z\}$ then

$$v \cdot i \cdot o \cdot l \cdot a \cdot \epsilon$$

is a string. Strings are usually written in more succinct form, e.g., the above string is written as *viola*.  

---

**Example - Strings**

- Let $\Sigma$ be a finite set. We define the set $\Sigma^*$ of all strings (over $\Sigma$) recursively by:
  
  1. The empty string, denoted by $\epsilon$, is an element of $\Sigma^*$.
  2. If $w$ is an element of $\Sigma^*$ and $a$ is an element of $\Sigma$, then $a \cdot w$ is an element of $\Sigma^*$.

- In other words, strings can be defined in terms of (i) the empty string and (ii) the function $\cdot : \Sigma \times \Sigma^* \rightarrow \Sigma^*$ that maps a pair $(a, w)$ to the string beginning with the symbol $a$ and followed by $w$.

- For example, if $\Sigma = \{a, b, c, \ldots, x, y, z\}$ then

$$v \cdot i \cdot o \cdot l \cdot a \cdot \epsilon$$

is a string. Strings are usually written in more succinct form, e.g., the above string is written as *viola*. 

---
Recursive String Operations

- Recursive definitions of sets provide a natural basis for recursive definitions of functions defined on these domains.

- For example, the length function for strings can be defined by:
  
  1. $|\epsilon| = 0$
  2. $|a \cdot w| = |w| + 1$

- Concatenation can be defined by:
  
  1. If $v$ is the string $\epsilon$ and $w$ is any string, then $vw = w$.
  2. If $v$ is a string $a \cdot v'$ and $w$ is a string, then $vw = a \cdot w'$ where $w'$ is $v'w$. 
Example - Lists

• Let $A$ be a set. The set $lists_A$ of lists of elements of type $A$ can be defined in a similar way as strings:
  
  1. The empty list, denoted by $[]$ or $nil$, is an element of $lists_A$.
  2. If $x$ is an element of $A$ and $L$ is an element of $lists_A$, then $x :: L$ is an element of $lists_A$.

• Keep in mind that we implicitly assume closure: only objects obtained by the above two steps are elements of $lists_A$.

• We sometimes write $cons(x, L)$ instead of $x :: L$ and, in general, simplify the notation by writing

  $$[a_1, a_2, \ldots, a_{n-1}, a_n]$$

  instead of

  $$a_1 :: a_2 :: \cdots :: a_{n-1} :: a_n :: nil.$$
• We may view \textit{nil} and \textit{cons} as constructor operations that are used to build lists.

• Other operations, called destructors, may be viewed as decomposing given lists into simpler components.

• The latter operations include the head and tail functions, which are defined as follows:

\[
\text{If } L \text{ is a non-empty list } x :: L', \text{ then } \text{head}(L) = x \text{ and } \text{tail}(L) = L'.
\]

• Note that the two functions are not defined for the empty list, and hence are partial functions on the domain \textit{lists}_A.

• The above definition implies that the following identity is valid for all non-empty lists \( L \):

\[
L = \text{head}(L) :: \text{tail}(L).
\]
Examples

- The set $S$ of all lists over $A$ of even length can be defined by:
  
  (i) $\text{nil} \in S$ and  
  (ii) if $x \in A$, $y \in A$, and $L \in S$, then $x :: (y :: L) \in S$.

- Let $T$ be defined by:
  
  (i) if $a \in A$ and $L \in \text{lists}_A$ then $(a, a :: L) \in T$, and 
  (ii) if $a \in A$ and $(b, L) \in T$, then $(b, a :: L) \in T$.

  What are the elements of this set?

- Finally, we inductively define the set $U$ of all lists over $\{0, 1\}$ with alternating occurrences of elements:
  
  (i) the lists $\text{nil}$, $[0]$, and $[1]$ are elements of $U$;  
  (ii) if $0 :: L \in U$ then $1 :: (0 :: L) \in U$; and 
  (iii) if $1 :: L \in U$ then $0 :: (1 :: L) \in U$.  

Structural Induction

Let $S$ be a recursively defined set and $P$ be a property defined on the elements of $S$. Suppose the following two statements are true:

1. The property $P$ is satisfied by all elements specified in the basis part of the definition of $S$.
2. If a rule in the recursion part of the definition of $S$ is applied to elements that satisfy $P$, then the objects generated by the rule also satisfy $P$.

Then $P$ is true for all elements of $S$.

For example, let $A$ be the following recursively defined set of strings of (left and right) parentheses:

1. The empty string is an element of $A$.
2. If $x$ and $y$ are elements of $A$, then $xy$ and $(x)$ are also elements of $A$.

Let $P(t)$ be the property that $t$ contains the same number of left and right parentheses, and $Q(t)$ be the property that no prefix of $t$ contains fewer left than right parentheses.

Structural induction can be used to prove that both properties, $P$ and $Q$, are satisfied by all elements of $A$. 