Discrete Structures for Computer Science

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU CSci 341

Subsection 1

Inductively Defined Sets

Inductive Definitions Informally

- Consider the set $A = \{3, 5, 7, 9, \ldots\}$.
- This set can also be described as $A = \{2k + 3 : k \in \mathbb{N}\}$.
- Another way to describe A is as follows:
  - $3 \in A$;
  - Whenever $x \in A$, then $x + 2 \in A$;
  - The only way an element gets in A is by the previous two steps.
- The three ingredients involved in such an *inductive definition* of an *inductive set*:
  1. There is a starting element (3 in this case).
  2. There is a construction operation to build new elements from existing elements (addition by 2 in this case).
  3. There is a statement that no other elements are in the set.

Inductive Definitions Formally

- An **inductive definition** of a set S consists of three steps:

Basis: List some specific elements of S (at least one element must be listed).

Induction: Give one or more rules to construct new elements of S from existing elements of S.

Closure: State that S consists exactly of the elements obtained by the basis and induction steps.

- The Closure step is usually assumed rather than stated explicitly, but should not be underestimated!
- Example: Consider the two steps
  - $3 \in S$;
  - If $x \in S$, then $x + 2 \in S$.

  Without the Closure step, these two steps do not define $\{3, 5, 9, \ldots\}$ (as intended), since the set $\mathbb{N}$ is another set satisfying these requirements (and there are many others!).

Constructors of an Inductive Set

- The **constructors** of an inductive set are:
  - the basis elements;
  - the rules for constructing new elements.
- Example: Consider again the inductive set $S = \{3, 5, 7, 9, \ldots\}$ defined inductively by:
  - $3 \in S$;
  - $x \in S$ implies $x + 2 \in S$;
  - Closure.

  It has two constructors:
  - the number 3;
  - the operation of adding 2 to a number.

Inductive Definition of the Natural Numbers

- The set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ is an inductive set:

Basis: $0 \in \mathbb{N}$;

Induction: If $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$.

- The constructors of $\mathbb{N}$ are:
  - the integer 0;
  - the operation that adds 1 to an element of $\mathbb{N}$, called the **successor** function and written

$$\text{succ}(n) = n + 1.$$

- Rephrasing, we can say that $\mathbb{N}$ is the inductive set defined by:

Basis: $0 \in \mathbb{N}$;

Induction: If $n \in \mathbb{N}$, then $\text{succ}(n) \in \mathbb{N}$.

- So $\mathbb{N}$ is an inductive set with the two constructors 0 and succ.

Example

- Give an inductive definition of $A = \{1, 3, 7, 15, 31, \ldots\}$.

  The basis case should place 1 in A.

  If $x \in A$, then we can construct another element of A with the expression $2x + 1$.

  So the constructors of A are:

  - the number 1;
  - the operation of multiplying by 2 and adding 1.

  An inductive definition of A is:

Basis: $1 \in A$;
Induction: If $x \in A$, then $2x + 1 \in A$.

Example

- Write an inductive definition of $A = \{2, 3, 4, 7, 8, 11, 15, 16, \ldots\}$.

  We think of A as the union of the two sets

$$\begin{aligned} B &= \{2, 4, 8, 16, \ldots\}; \\ C &= \{3, 7, 11, 15, \ldots\}. \end{aligned}$$

  Both these sets are inductive.
  - The constructors of B are:
    - the number 2;
    - the operation of multiplying by 2.
  - The constructors of C are:
    - the number 3;
    - the operation of adding 4.
- We can combine these definitions to give an inductive definition of A:

Basis: $2, 3 \in A$;

Induction: If $x \in A$ and x is odd, then $x + 4 \in A$;
If $x \in A$ and x is even, then $2x \in A$.

The Set of All Strings Over an Alphabet

- Let A be an alphabet.
- The set A^* of all strings over A is defined inductively as follows:

Basis: $\Lambda \in A^*$;

Induction: If $s \in A^*$ and $a \in A$, then $as \in A^*$.

- Here, we used the two constructors:
  - the empty string Λ;
  - the operation of concatenation (placing two strings next to each other in juxtaposition to form a new string).

Inductively Defined Languages Over $\{a, b\}$

- Recall that a **language** over an alphabet A is a subset of A^*.
- Give inductive definitions of the following languages over $A = \{a, b\}$:
  - (a) $S = \{a, ab, abb, abbb, \ldots\} = \{ab^n : n \in \mathbb{N}\}$;
  - (b) $S = \{\Lambda, ab, aabb, aaabbb, \ldots\} = \{a^n b^n : n \in \mathbb{N}\}$;
  - (c) $S = \{\Lambda, ab, abab, ababab, \ldots\} = \{(ab)^n : n \in \mathbb{N}\}$.

(a) S is defined inductively as follows:

Basis: $a \in S$;

Induction: If $s \in S$, then $sb \in S$.

(b) S is defined inductively as follows:

Basis: $\Lambda \in S$;

Induction: If $s \in S$, then $asb \in S$.

(c) S is defined inductively as follows:

Basis: $\Lambda \in S$;

Induction: If $s \in S$, then $abs \in S$.

Example

- Define inductively the following language over $A = \{a, b\}$:

  $$S = \{a, b, ab, ba, aab, bba, aaab, bbba, \ldots\}.$$

  Think of S as the union of two simpler sets

  $$A = \{a, ba, bba, \ldots\} \quad \text{and} \quad B = \{b, ab, aab, \ldots\}.$$

  The definition of S can be written as follows:

Basis: $a, b, ba, ab \in S$;

Induction: Let $s \in S$.

- If $s \neq a$ and s starts with a, then $as \in S$;
- If $s \neq b$ and s starts with b, then $bs \in S$.

Inductively Defined Languages Over $\{0, 1\}$

- Define inductively the language S of strings over $A = \{0, 1\}$ such that each string in S contains exactly one occurrence of 0 and that occurrence is on the right.

  Note that this is the language

  $$S = \{0, 10, 110, 1110, \ldots\}.$$

  The inductive definition of S can be written as follows:

  Basis: $0 \in S$;

  Induction: If $s \in S$, then $1s \in S$.

- Give an inductive definition of the set S of strings over $A = \{0, 1\}$ with the following property:

  No string contains a leading zero except 0 itself.

  S should contain strings like $0, 1, 10, 11, 100, 101, 110, 111, \ldots$.

  The inductive definition of S can be written as follows:

  Basis: $0, 1 \in S$;

  Induction: If $s \in S$ and $s \neq 0$, then $s0, s1 \in S$.

Lists Over A Revisited

- Recall that Lists$[A]$ is the set of all lists over A.
- **Destructors** are the operations:
  - head : Lists$[A] - \{\langle\rangle\} \to A$ returning the head of a list;
  - tail : Lists$[A] \to$ Lists$[A]$ returning the tail of a list.
- **Constructor** is the operation:
  - cons : $A \times$ Lists$[A] \to$ Lists$[A]$ that given an element in A and a list over A constructs a new list with the element as head and the list as tail.
- For any nonempty list L, we have

$$L = \text{cons}(\text{head}(L), \text{tail}(L)).$$

- Example: We have
  - $\text{head}(\langle x, y, z\rangle) = x$;
  - $\text{tail}(\langle x, y, z\rangle) = \langle y, z\rangle$;
  - $\text{cons}(x, \langle y, z\rangle) = \langle x, y, z\rangle$.

Inductive Definition of Lists

- To write an inductive definition of lists over A we use the following constructors:
  - the empty list $\langle\rangle$;
  - the list constructor cons.
- Then the set $\mathrm{Lists}[A]$ is defined inductively by:

Basis: $\langle\rangle \in \mathrm{Lists}[A]$;

Induction: If $x \in A$ and $L \in \mathrm{Lists}[A]$, then $\mathrm{cons}(x, L) \in \mathrm{Lists}[A]$.

Infix Notation for cons

- The infix form of $\text{cons}(x, L)$ is $x :: L$.
- Example: The list $\langle a, b, c\rangle$ can be constructed by using cons as follows:

$$\begin{aligned} \text{cons}(a, \text{cons}(b, \text{cons}(c, \langle\rangle))) &= \text{cons}(a, \text{cons}(b, \langle c\rangle)) \\ &= \text{cons}(a, \langle b, c\rangle) \\ &= \langle a, b, c\rangle. \end{aligned}$$

  Using the infix form, we construct $\langle a, b, c\rangle$ as follows:

$$a :: (b :: (c :: \langle\rangle)) = a :: (b :: \langle c\rangle) = a :: \langle b, c\rangle = \langle a, b, c\rangle.$$

- When parentheses are omitted in infix, we associate on the right.
- Example: $a :: b :: c :: \langle\rangle$ means $a :: (b :: (c :: \langle\rangle))$.

Example

- Define inductively the set S of all nonempty lists over the set $\{0,1\}$, where the elements in each list alternate between 0 and 1.

  We get an idea about S by listing a few elements:

  $$S = \{\langle 0\rangle, \langle 1\rangle, \langle 1,0\rangle, \langle 0,1\rangle, \langle 0,1,0\rangle, \langle 1,0,1\rangle, \ldots\}.$$

  We use the constructors:

  - the lists $\langle 0\rangle, \langle 1\rangle$;
  - the list constructor cons.

  An inductive definition for S can be written as follows:

Basis: $\langle 0\rangle, \langle 1\rangle \in S$;

Induction: If $L \in S$ and $\text{head}(L) = 0$, then $\text{cons}(1, L) \in S$.
If $L \in S$ and $\text{head}(L) = 1$, then $\text{cons}(0, L) \in S$.

Example

- Define inductively the set S of all lists over $\{a, b\}$ that begin with the single letter a followed by zero or more occurrences of b.

  We can describe S informally by writing a few of its elements:

$$S = \{\langle a\rangle, \langle a, b\rangle, \langle a, b, b\rangle, \langle a, b, b, b\rangle, \ldots\}.$$

  We make $\langle a\rangle$ the basis element of S.

  Then we can construct a new list from any list $L \in S$ by attaching the letter b on the left end of the tail of L by using

$$\text{cons}(a, \text{cons}(b, \text{tail}(L))) \in S.$$

  So we have the following inductive definition of S:

Basis: $\langle a\rangle \in S$;
Induction: If $L \in S$, then $a :: b :: \text{tail}(L) \in S$.

Inductive Definition of a Product

- To give an inductive definition for a product $A \times B$ of two sets A and B, we may use one of two techniques:
  - The first can be used if both A and B are inductively defined:
    - For the basis case, put $(a, b) \in A \times B$ whenever a is a basis element of A and b is a basis element of B;
    - For the inductive part, if $(x, y) \in A \times B$ and $x' \in A$ and $y' \in B$ are elements constructed from x and y, respectively, then put the elements $(x, y'), (x', y)$ in $A \times B$.
  - The second technique can be used if only one of the sets, say A, is inductively defined:
    - For the basis case, put $(a, b) \in A \times B$ for all basis elements $a \in A$ and all elements $b \in B$;
    - For the induction case, if $(x, y) \in A \times B$ and $x' \in A$ is constructed from x, then put $(x', y) \in A \times B$.

  A similar definition of $A \times B$ can also be made that uses only the fact that B is inductively defined.

Example

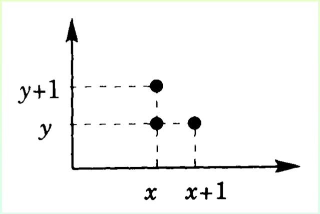
- Recall the inductive definition of $\mathbb{N}$:

Basis: $0 \in \mathbb{N}$;
Induction: If $n \in \mathbb{N}$ then $\mathrm{succ}(n) \in \mathbb{N}$.

Give an inductive definition of the set $\mathbb{N} \times \mathbb{N}$.

According to the method of the preceding slide, we can define $\mathbb{N} \times \mathbb{N}$ inductively as follows:

Basis: $(0, 0) \in \mathbb{N} \times \mathbb{N}$;
Induction: If $(x, y) \in \mathbb{N} \times \mathbb{N}$, then $(x, \mathrm{succ}(y)), (\mathrm{succ}(x), y) \in \mathbb{N} \times \mathbb{N}$.



Example

- Let $A = \{n \in \mathbb{N} : n \mod 5 = 2\}$ and $B = \{2^n : n \in \mathbb{N}\}$.

  Give an inductive definition for $A \times B$.

  Note that

  $$A = \{2, 7, 12, 17, 22, \ldots\}, \quad B = \{1, 2, 4, 8, 16, \ldots\}.$$

  Now we have the following inductive definitions:

  - For A:

    Basis: $2 \in A$;
    Induction: If $x \in A$, then $x + 5 \in A$.

  - For B:

    Basis: $1 \in B$;
    Induction: If $y \in B$, then $2y \in B$.

  Thus, for $A \times B$, we obtain:

Basis: $(2, 1) \in A \times B$;
Induction: If $(x, y) \in A \times B$, then $(x + 5, y), (x, 2y) \in A \times B$.

Subsection 2

Recursive Functions and Procedures

Recursively Defined Functions and Procedures

- A **procedure** is a subprogram that:
    - carries out one or more actions;
    - returns any number of values - including none - through its arguments.
- A function or a procedure is said to be **recursively defined** if it is defined in terms of itself:
    - A function f is recursively defined if at least one value $f(x)$ is defined in terms of another value $f(y)$, where $x \neq y$.
    - A procedure P is recursively defined if the actions of P for some argument x are defined in terms of the actions of P for another argument y, where $x \neq y$.

Constructing a Recursively Defined Function/Procedure

- Suppose S is an inductively defined set.
  - We can construct a function f with domain S as follows:

Basis: For each basis element $x \in S$, specify a value for $f(x)$.

Induction: For any inductively defined element $x \in S$, give rules that define $f(x)$ in terms of previously defined values of f.

  - We can construct a procedure P to process the elements of S as follows:

Basis: For each basis element $x \in S$, specify a set of actions for $P(x)$.

Induction: For any inductively defined element $x \in S$, give rules that define the actions of $P(x)$ in terms of previously defined actions of P.

Example: Pattern-Matching Definitions

- Let $f : \mathbb{N} \to \mathbb{N}$ denote the function that, for each $n \in \mathbb{N}$, returns the sum of the odd numbers from 1 to $2n+1$:

$$f(n) = 1 + 3 + \cdots + 2n + 1.$$

  - Since $f(0) = 1$, we have a definition for f applied to the basis element $0 \in \mathbb{N}$.
  - For the inductive part of the definition, we have:

$$\begin{array}{rcl} f(n+1) & = & 1 + 3 + \cdots + (2n+1) + [2(n+1)+1] \\ & = & 1 + 3 + \cdots + (2n+1) + (2n+3) \\ & = & f(n) + 2n + 3. \end{array}$$

  This gives us the necessary ingredients for a recursive definition of f:

Basis: $f(0) = 1$;
Induction: $f(n+1) = f(n) + 2n + 3$.

- A definition like this is often called a **pattern-matching definition** because the evaluation of an expression $f(x)$ depends on $f(x)$ matching either $f(0)$ or $f(n+1)$.

Example: Conditional Definitions

- Consider again $f : \mathbb{N} \to \mathbb{N}$ defined by
$$f(n) = 1 + 3 + \cdots + 2n + 1.$$
An alternative form for the definition of f is the **conditional form**.
  - One conditional form consists of equations with conditionals:

Basis: $f(0) = 1$;
Induction: $f(n) = f(n-1) + 2n + 1$, if $n > 0$.

  - A second conditional form is the familiar if-then-else equation as follows:
$$f(n) = \text{if } n = 0 \text{ then } 1 \text{ else } f(n-1) + 2n + 1.$$

Evaluation of Recursive Functions

- A recursively defined function can be easily evaluated by a technique called **unfolding the definition**.
- Example: Consider the function f defined recursively by

Basis: $f(0) = 1$;
Induction: $f(n+1) = f(n) + 2n + 3$.

Unfold the definition to find the value of $f(3)$.

$$\begin{array}{rcl} f(3) & = & f(2) + 2 \cdot 2 + 3 \\ & = & f(1) + 2 \cdot 1 + 3 + 2 \cdot 2 + 3 \\ & = & f(0) + 2 \cdot 0 + 3 + 2 \cdot 1 + 3 + 2 \cdot 2 + 3 \\ & = & 1 + 2 \cdot 0 + 3 + 2 \cdot 1 + 3 + 2 \cdot 2 + 3 \\ & = & 1 + 3 + 5 + 7 \\ & = & 16. \end{array}$$

Example: The Rabbit Problem and Fibonacci Numbers

- Start with a single pair of rabbits.

  Assume that every month each pair produces a new pair that becomes productive after one month.

(a) Find the number of new pairs produced each month for the first nine months.

Month	0	1	2	3	4	5	6	$\cdots$
Old	1	1	2	3	5	8	13	$\cdots$
Young	0	1	1	2	3	5	8	$\cdots$

(b) Let $\text{fib}(n)$, the n-th **Fibonacci number**, be the number of new rabbits produced after month n.

Give a recursive definition of n.

Basis: $\text{fib}(0) = 0,\ \text{fib}(1) = 1;$

Induction: $\text{fib}(n+2) = \text{fib}(n+1) + \text{fib}(n).$

Example: Complements of Strings

- Let $A = \{a, b\}$ be a 2-symbol alphabet.
- Given a string s over A, its **complement** is the string formed by changing every letter in s.
- Example: The complement of $bbab$ is $aaba$.
- Find a recursive definition of the function $f : A^* \to A^*$, defined by

$$f(s) = \text{the complement of } s.$$

First, an arbitrary string over A is either Λ or has the form as or bs for some string s.

So, we have the following recursive definition:

Basis: $f(\Lambda) = \Lambda$;
Induction: $f(as) = bf(s)$;
$f(bs) = af(s)$.

Example: Prefixes of Strings

- A string p is a **prefix** of the string x if x can be written in the form $x = ps$ for some string s.
- Define the function $\text{pfix}(x, y)$ giving the longest common prefix of two strings x, y over $\{a, b\}$.
- Example: The longest common prefix of the two strings *aabbab* and *aababb* is
$$\text{pfix}(aabbab, aababb) = aab.$$

Example: Pattern-Matching Definition of pfix

- An arbitrary string over $\{a, b\}$ matches one of the following patterns:
  - the empty string Λ;
  - the form *as*, for some string *s*;
  - the form *bs* for some string *s*.

  A pattern-matching inductive definition of f is as follows:

Basis: $\text{pfix}(\Lambda, x) = \Lambda$;
$\text{pfix}(x, \Lambda) = \Lambda$;

Induction: $\text{pfix}(as, bt) = \Lambda$;
$\text{pfix}(bs, at) = \Lambda$;
$\text{pfix}(as, at) = a\text{pfix}(s, t)$;
$\text{pfix}(bs, bt) = b\text{pfix}(s, t)$.

Example: Conditional Definition of pfix

- The set of pairs of strings $S = \{a, b\}^* \times \{a, b\}^*$ over $\{a, b\}$ is defined by induction as follows:

Basis: $(\Lambda, \Lambda) \in S$;
Induction: If $(x, y) \in S$, then $(sx, y), (x, sy) \in S$, for any $s \in \{a, b\}$;

- To write a conditional form of the definition of pfix we assume that we have available functions head and tail for strings with the obvious functionalities.
- A conditional form of an inductive definition of pfix is as follows:

Basis: $\text{pfix}(\Lambda, \Lambda) = \Lambda$;
Induction: $\text{pfix}(sx, y) = \text{if head}(y) = s \text{ then } s\text{pfix}(x, \text{tail}(y)) \text{ else } \Lambda$;
$\text{pfix}(x, sy) = \text{if head}(x) = s \text{ then } s\text{pfix}(\text{tail}(x), y) \text{ else } \Lambda$;

Natural Numbers to Binary Strings

- The division algorithm gives the following representation of a natural number x:
$$x = 2 \cdot \text{floor}(x/2) + x \mod 2.$$

- Give a recursive definition of a function bin converting a decimal representation of x to a binary representation of x.

  This formula above can be used to create a binary string representation of x:
  - $x \mod 2$ is the rightmost bit of the representation;
  - The next bit is found by computing $\text{floor}(x/2) \mod 2$.
  - The next bit is $\text{floor}(\text{floor}(x/2)/2) \mod 2$, and so on.

  The recursive definition is as follows:

Basis: $\text{bin}(0) = 0;$
$\text{bin}(1) = 1;$
Induction: $\text{bin}(x) = \text{cat}(\text{bin}(\text{floor}(x/2)), x \mod 2), \text{ if } x > 1.$

Natural Numbers to Binary Strings (Cont'd)

- We gave the recursive definition:

Basis: $\text{bin}(0) = 0$; $\text{bin}(1) = 1$;

Induction: $\text{bin}(x) = \text{cat}(\text{bin}(\text{floor}(x/2)), x \mod 2)$, if $x > 1$.

Unfold the definition to compute bin(9).

$$\begin{aligned}
\text{bin}(9) &= \text{cat}(\text{bin}(4), 1) \\
&= \text{cat}(\text{cat}(\text{bin}(2), 0), 1) \\
&= \text{cat}(\text{cat}(\text{cat}(\text{bin}(1), 0), 0), 1) \\
&= \text{cat}(\text{cat}(\text{cat}(1, 0), 0), 1) \\
&= \text{cat}(\text{cat}(10, 0), 1) \\
&= \text{cat}(100, 1) \\
&= 1001.
\end{aligned}$$

Example: Computing a List

- Give a recursive definition of the function $f : \mathbb{N} \to \text{Lists}[\mathbb{N}]$ that computes the list
$$\langle n, n-1, \ldots, 1, 0\rangle.$$
We have
$$\begin{array}{rcl} f(n) & = & \langle n, n-1, \ldots, 1, 0\rangle \\ & = & \text{cons}(n, \langle n-1, \ldots, 1, 0\rangle) \\ & = & \text{cons}(n, f(n-1)). \end{array}$$
Therefore, the inductive definition is as follows:

Basis: $f(0) = \langle 0\rangle$;
Induction: $f(n) = \text{cons}(n, f(n-1))$, for $n \neq 0$.

Length of a List

- Let S be a set.

  Define length : Lists$[S] \to \mathbb{N}$ as the function that returns the number of elements in a list.

  Give a recursive definition of length.

  Note that:
  - the length of an empty list is zero;
  - the length of a nonempty list is one plus the length of its tail.

  A recursive definition is as follows:

Basis: $\text{length}(\langle\rangle) = 0$;
Induction: $\text{length}(L) = 1 + \text{length}(\text{tail}(L))$, if $L \neq \langle\rangle$.

- Observe that the induction step can also be written as:

Induction: $\text{length}(\text{cons}(x, t)) = 1 + \text{length}(t)$.
Induction: $\text{length}(x :: t) = 1 + \text{length}(t)$.

The Distribute Function

- Write a recursive definition for the distribute function, denoted $\text{dist} : A \times \text{Lists}[A] \to \text{Lists}[A \times A]$ defined by

$$\text{dist}(a, \langle a_1, a_2, \ldots, a_n \rangle) = \langle (a, a_1), (a, a_2), \ldots, (a, a_n) \rangle.$$

We use induction on the second argument (a list) to define dist.

Note the following:

$$\begin{array}{l} \text{dist}(a, \langle a_1, a_2, \ldots, a_n \rangle) \\ = \langle (a, a_1), (a, a_2), \ldots, (a, a_n) \rangle \\ = (a, a_1) :: \langle (a, a_2), \ldots, (a, a_n) \rangle \\ = (a, a_1) :: \text{dist}(a, \langle a_2, \ldots, a_n \rangle). \end{array}$$

So we have the following definition:

Basis: $\text{dist}(a, \langle\rangle) = \langle\rangle$;

Induction: $\text{dist}(a, L) = (a, \text{head}(L)) :: \text{dist}(a, \text{tail}(L))$, if $L \neq \langle\rangle$.

The Pairs Function

- The function pairs is defined by

$$\text{pairs}(\langle a_1, a_2, \ldots, a_n\rangle, \langle b_1, b_2, \ldots, b_n\rangle) = \langle (a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\rangle.$$

Give a recursive definition of pairs.

The key is to compute:

$$\begin{aligned}&\text{pairs}(\langle a_1, a_2, \ldots, a_n\rangle, \langle b_1, b_2, \ldots, b_n\rangle)\\ &= \langle (a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\rangle\\ &= (a_1, b_1) :: \langle (a_2, b_2), \ldots, (a_n, b_n)\rangle\\ &= (a_1, b_1) :: \text{pairs}(\langle\langle a_2, \ldots, a_n\rangle, \langle b_2, \ldots, b_n\rangle).\end{aligned}$$

The pairs function can be defined recursively by:

Basis: $\text{pairs}(\langle\rangle, \langle\rangle) = \langle\rangle$;

Induction: $\text{pairs}(L, L') = (\text{head}(L), \text{head}(L')) :: \text{pairs}(\text{tail}(L), \text{tail}(L'))$.

Appending an Element on the Right

- Write a recursive definition of the function $\text{consR} : \text{Lists}[A] \times A \to \text{Lists}[A]$, defined by

$$\text{consR}(\langle a_1, a_2, \ldots, a_n \rangle, a) = \langle a_1, a_2, \ldots, a_n, a \rangle.$$

  We compute:

$$\begin{array}{rcl} \text{consR}(\langle a_1, a_2, \ldots, a_n \rangle, a) & = & \langle a_1, a_2, \ldots, a_n, a \rangle \\ & = & a_1 :: \langle a_2, \ldots, a_n, a \rangle \\ & = & a_1 :: \text{consR}(\langle a_2, \ldots, a_n \rangle, a). \end{array}$$

  So the inductive definition is as follows:

Basis: $\text{consR}(\langle\rangle, a) = \langle a \rangle$;
Induction: $\text{consR}(L, a) = \text{head}(L) :: \text{consR}(\text{tail}(L), a)$.

Concatenation of Lists

- Let $\mathrm{cat} : \mathrm{Lists}[A] \times \mathrm{Lists}[A] \to \mathrm{Lists}[A]$ denote the concatenation function, defined by

$$\mathrm{cat}(\langle a_1, \ldots, a_n\rangle, \langle b_1, \ldots, b_m\rangle) = \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle.$$

  Write a recursive definition of cat.

  Note that

$$\begin{aligned}&\mathrm{cat}(\langle a_1, \ldots, a_n\rangle, \langle b_1, \ldots, b_m\rangle)\\ &= \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle\\ &= a_1 :: \langle a_2, \ldots, a_n, b_1, \ldots, b_m\rangle\\ &= a_1 :: \mathrm{cat}(\langle a_2, \ldots, a_n\rangle, \langle b_1, \ldots, b_m\rangle).\end{aligned}$$

  So, we have the following recursive definition:

Basis: $\mathrm{cat}(\langle\rangle, L') = L'$;
Induction: $\mathrm{cat}(L, L') = \mathrm{head}(L) :: \mathrm{cat}(\mathrm{tail}(L), L')$.

Inserting a Number in a Sorted List

- Give a recursive procedure insert(x, S) for inserting a new number into a sorted list of numbers.

  Assume that whenever the number to be inserted is already in the list, then a new redundant copy will be placed to the left of the one already there.

  We work by induction on the length of the second argument.
  - The empty list is sorted, and to insert x into $\langle\rangle$, we simply create the singleton list $\langle x\rangle$.
  - Otherwise if the sorted list is not empty, then:
    - Either x belongs on the left of the list or
    - x should be inserted somewhere else in the list.

  The inductive definition is as follows:

Basis: $\text{insert}(x, \langle\rangle) = \langle x\rangle$;
Induction: If $L \neq \langle\rangle$, $\text{insert}(x, L) =$
if $x \leq \text{head}(L)$ then $x :: L$ else $\text{head}(L) :: \text{insert}(x, \text{tail}(L))$.

The Function map

- Let $f : A \to B$ be a function.

  Give a recursive definition of the function $\text{map}(f) : \text{Lists}[A] \to \text{Lists}[B]$, defined by

  $$\text{map}(f)(\langle a_1, a_2, \ldots, a_n \rangle) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle.$$

  Note that:

  $$\begin{array}{rcl} \text{map}(f)(\langle a_1, a_2, \ldots, a_n \rangle) & = & \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle \\ & = & \text{cons}(f(a_1), \langle f(a_2), \ldots, f(a_n) \rangle) \\ & = & \text{cons}(f(a_1), \text{map}(f)(\langle a_2, \ldots, a_n \rangle)). \end{array}$$

  So we have the following inductive definition:

Basis: $\text{map}(f)(\langle\rangle) = \langle\rangle$;

Induction: $\text{map}(f)(L) = \text{cons}(f(\text{head}(L)), \text{map}(f)(\text{tail}(L)))$, if $L \neq \langle\rangle$.

Streams (Infinite Sequences)

- A **stream** is an infinite list.
- We define the constructor $\text{cons} : A \times \text{Stream}[A] \to \text{Stream}[A]$, with

$$\text{cons}(a_0, \langle a_1, a_2, \ldots \rangle) = \langle a_0, a_1, a_2, \ldots \rangle.$$

- We also define the destructors:
  - $\text{head} : \text{Stream}[A] - \{\langle\rangle\} \to A$, with

$$\text{head}(\langle a_0, a_1, \ldots \rangle) = a_0;$$

  - $\text{tail} : \text{Stream}[A] \to \text{Stream}[A]$, with

$$\text{tail}(\langle a_0, a_1, a_2, \ldots \rangle) = \langle a_1, a_2, \ldots \rangle.$$

  - Note the property that, for any stream s,

$$\text{cons}(\text{head}(s), \text{tail}(s)) = s.$$

Example

- Example: Consider the function $\mathsf{ints} : \mathbb{Z} \to \mathsf{Stream}[\mathbb{Z}]$, defined by

$$\mathsf{ints}(x) = \langle x, x+1, x+2, \ldots \rangle.$$

  We have the following:
  - $\mathsf{ints}(x) = x :: \mathsf{ints}(x+1)$;
  - $\mathsf{head}(\mathsf{ints}(x)) = x$;
  - $\mathsf{tail}(\mathsf{ints}(x)) = \mathsf{ints}(x+1)$.

Pseudo-Recursive Definitions

- The following can be taken as the definition of ints:

$$\text{ints}(x) = x :: \text{ints}(x+1).$$

- Strictly speaking, it is not recursive since it does not have a basis.
- It is still "recursive" in the sense that it is defined in terms of itself.
- If we executed the definition, an infinite loop would construct the stream, e.g.:

$$\begin{array}{lcl} \text{ints}(0) & = & 0 :: \text{ints}(1) \\ & = & 0 :: 1 :: \text{ints}(2) \\ & = & 0 :: 1 :: 2 :: \text{ints}(3) \\ & = & \cdots \end{array}$$

Lazy Evaluation

- In practice, when a stream like $\text{ints}(x)$ is used as an argument to another function, it is evaluated only when some of its values are needed, a technique called **lazy evaluation**.
- Example: Use lazy evaluation to compute

$$\text{head}(\text{tail}(\text{ints}(3))).$$

$$\begin{aligned}
\text{head}(\text{tail}(\text{ints}(3))) &= \text{head}(\text{tail}(3 :: \text{ints}(4))) \\
&= \text{head}(\text{ints}(4)) \\
&= \text{head}(4 :: \text{ints}(5)) \\
&= 4.
\end{aligned}$$

Summing

(a) Build a function sum : $\mathbb{N} \times \text{Stream}[\mathbb{Z}] \to \mathbb{Z}$ that takes a natural number n and a stream s of integers and returns the sum of the first n elements of s.

We can make the following general definition:

$$\text{sum}(n,s) = \text{if } n = 0 \text{ then } 0 \text{ else head}(s) + \text{sum}(n-1, \text{tail}(s)).$$

(b) Unfold the definition to evaluate sum(3, ints(4)).

$$\begin{aligned} \text{sum}(3,\text{ints}(4)) &= 4 + \text{sum}(2,\text{ints}(5)) \\ &= 4 + 5 + \text{sum}(1,\text{ints}(6)) \\ &= 4 + 5 + 6 + \text{sum}(0,\text{ints}(7)) \\ &= 4 + 5 + 6 + 0 \\ &= 15. \end{aligned}$$

Skipping

(a) Construct the function skipTwo : $\mathbb{Z} \to$ Stream[$\mathbb{Z}$] defined by:

$$\text{skipTwo}(x) = \langle x, x+2, x+4, \ldots \rangle.$$

The definition is as follows:

$$\text{skipTwo}(x) = x :: \text{skipTwo}(x+2).$$

(b) Unfold the definition to compute skipTwo(5).

$$\begin{array}{rcl} \text{skipTwo}(5) & = & 5 :: \text{skipTwo}(7) \\ & = & 5 :: 7 :: \text{skipTwo}(9) \\ & = & 5 :: 7 :: 9 :: \text{skipTwo}(11) \\ & = & \langle 5, 7, 9, 11, \ldots \rangle. \end{array}$$

More General Skipping

(a) Construct a function named skip : $\mathbb{Z} \times \mathbb{N} \to \text{Stream}[\mathbb{Z}]$, defined by

$$\text{skip}(x, k) = \langle x, x + k, x + 2k, \ldots \rangle.$$

We may define skip by

$$\text{skip}(x, k) = x :: \text{skip}(x + k, k).$$

(b) Evaluate the expression skip$(-5, 3)$.

$$\begin{array}{rcl} \text{skip}(-5, 3) & = & -5 :: \text{skip}(-2, 3) \\ & = & -5 :: -2 :: \text{skip}(1, 3) \\ & = & -5 :: -2 :: 1 :: \text{skip}(4, 3) \\ & = & \langle -5, -2, 1, 4, \ldots \rangle. \end{array}$$

Subsection 3

Grammars

Grammars

- A **grammar** consists of four parts:
  1. An alphabet N of grammar symbols called **nonterminals**.
  2. An alphabet T of symbols called **terminals**.
     The terminals are distinct from the nonterminals.
  3. A specific nonterminal S, called the **start symbol**.
  4. A finite set of **grammar rules** or **productions** of the form $\alpha \to \beta$, where α and β are strings over the alphabet $N \cup T$ with the restriction that α is not the empty string.
     - There is at least one production with only S on its left side.
     - Each nonterminal must appear on the left side of some production.
- We will only deal with grammar rules that have a single nonterminal on the left side.
- Under this hypothesis, all restrictions concerning productions can be summarized by saying that

  Each nonterminal appears as the left side of some production.

Derivation Steps

- Consider the grammar $G = \langle N, T, S, P \rangle$, where:
  1. $N = \{S\}$ is the set of nonterminals;
  2. $T = \{\Lambda, a, b\}$ is the set of terminals;
  3. S is the start symbol;
  4. $P = \{S \to \Lambda, S \to aS, S \to bS\}$ is the set of productions.
- A rule $\alpha \to \beta$ is read "**replace** α **by** β", "α **produces** β", "α **rewrites to** β" or "α **reduces to** β".
- We show that this grammar "derives" the string *aabab*:

$$\begin{array}{rcl} S & \Rightarrow & aS \Rightarrow aaS \Rightarrow aabS \Rightarrow aabaS \\ & \Rightarrow & aababS \Rightarrow aabab\Lambda = aabab. \end{array}$$

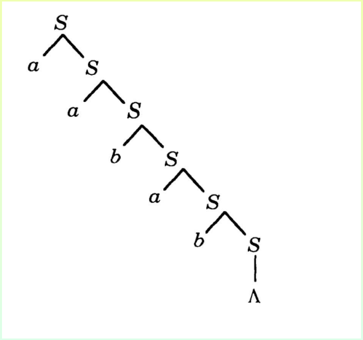
- Each step $\Rightarrow$ is called a **derivation step** and is read "**derives in one step**".
- A **derivation** is a sequence of derivation steps.

The Tree of a Derivation

- Consider again the derivation

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aabS \Rightarrow aabaS \Rightarrow aababS \Rightarrow aabab\Lambda = aabab.$$

- It has the following **derivation tree**:
  - Its root is the start symbol;
  - Each step in the derivation corresponds to attaching a new subtree to the existing derivation tree.



Notational and Other Conventions

- When two or more productions have the same left side, we can simplify the notation by writing one production with alternate right sides separated by the vertical line $|$.
- Example: The three productions $S \to \Lambda$, $S \to aS$, $S \to bS$ can be written in the following shorthand form: $S \to \Lambda|aS|bS$.
- For a particular grammar, we may only write down the productions of the grammar, where the nonterminals are capital letters and the first production listed contains the start symbol on its left side.
- Example: Suppose given the grammar

$$\begin{array}{rcl} S & \to & AB \\ A & \to & \Lambda|aA \\ B & \to & \Lambda|bB \end{array}$$

From this we can deduce that $N = \{S, A, B\}$, $T = \{\Lambda, a, b\}$, S is the start symbol and $P = \{S \to AB, A \to \Lambda, A \to aA, B \to \Lambda, B \to bB\}$.

Derivations

- In discussing derivations we use the symbols:

$\Rightarrow$	**derives in one step**
$\Rightarrow^+$	**derives in one or more steps**
$\Rightarrow^*$	**derives in zero or more steps**

- Example: Consider the grammar:

$$S \to AB, \quad A \to \Lambda | aA, \quad B \to \Lambda | bB.$$

$S \Rightarrow^+ aab$ means that there exists a derivation of aab that takes one or more steps. In fact we have

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aab.$$

Leftmost and Rightmost Derivations

- When a grammar contains more than one nonterminal, it may be possible to find several different derivations of the same string.
- A derivation is called a **leftmost derivation** if at each step the leftmost nonterminal is reduced by some production.
- A derivation is called a **rightmost derivation** if at each step the rightmost nonterminal is reduced by some production.
- Example: Consider again the grammar:

$$S \to AB, \quad A \to \Lambda | aA, \quad B \to \Lambda | bB.$$

The following are the leftmost and rightmost derivations, respectively of aab:

$$\begin{aligned} S &\Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aab; \\ S &\Rightarrow AB \Rightarrow AbB \Rightarrow Ab \Rightarrow aAb \Rightarrow aaAb \Rightarrow aab. \end{aligned}$$

The Language of a Grammar

- If G is a grammar, then the **language of** G, denoted $L(G)$, is the set of terminal strings derived from the start symbol of G.
- More formally, if
$$G = \langle N, T, S, P \rangle$$
is a grammar, then the **language of** G is the set
$$L(G) = \{s \in T^* : S \Rightarrow^+ s\}.$$

Grammars for Finite Languages

- Suppose we want to produce a grammar for a language L.
- If the language L is finite, say $L = \{w_1, w_2, \ldots, w_n\}$, then a grammar G can consist of the rules:

$$S \to w_1 | w_2 | \cdots | w_n.$$

- Example: Consider the language

$$L = \{a, ab, aab\}.$$

A grammar G, such that $L(G) = L$ is:

$$S \to a | ab | aab.$$

Grammars for Infinite Languages

- If the language is infinite, then some production or sequence of productions must be used repeatedly to construct the derivations.
- Example:
  - The infinite language
    $$L = \{a^n b : n \geq 0\}$$
    can be described by the grammar G
    $$S \to b | aS.$$
  - To derive the string $a^n b$:
    - use the production $S \to aS$ repeatedly n times;
    - stop the derivation by using the production $S \to b$.

  For example, we derive the string $a^5 b$:
  $$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaaaS \Rightarrow aaaaaS \Rightarrow aaaaab.$$

Recursive Grammars

- A production is called **recursive** if its left side occurs on its right side.
- Example: The production $A \to aA$ is recursive.
- A production $A \to \alpha$ is **indirectly recursive** if A derives a string that contains A.
- Example: Suppose we have the following grammar:

$$\begin{array}{rcl} S & \to & b|aA \\ A & \to & c|bS \end{array}$$

  The productions $S \to aA$ and $A \to bS$ are both indirectly recursive because of the following derivations:

$$S \Rightarrow aA \Rightarrow abS, \qquad A \Rightarrow bS \Rightarrow baA.$$

- A grammar is **recursive** if it contains either a recursive production or an indirectly recursive production.
- A grammar for an infinite language must be recursive.

Example

- Consider the grammar G:

$$\begin{array}{rcl} S & \rightarrow & \Lambda | aB \\ B & \rightarrow & b | bB. \end{array}$$

Find the language $L(G)$.
First, look at derivations that do not contain recursive productions:

$$S \Rightarrow \Lambda, \qquad S \Rightarrow aB \Rightarrow ab.$$

So, we have $\Lambda, ab \in L(G)$.
Every other derivation must use the recursive production $B \rightarrow bB$ one or more times, each time adding a new letter b to the derived string:

$$\begin{array}{rcl} S & \Rightarrow & aB \Rightarrow abB \Rightarrow abb; \\ S & \Rightarrow & aB \Rightarrow abB \Rightarrow abbB \Rightarrow abbb. \end{array}$$

So if $B \rightarrow bB$ is used n times ($n \geq 1$), then we can conclude that $ab^n \in L(G)$. Therefore, $L(G) = \{\Lambda\} \cup \{ab^n : n \geq 1\}$.

Some Simple Languages and Their Grammars

- We use the following simple languages and associated grammars as building blocks together with methods to combine grammars to obtain grammars for more complex languages.

Language	Grammar
$\{a, ab, abb, abbb\}$	$S \to a \vert ab \vert abb \vert abbb$
$\{\Lambda, a, aa, \ldots, a^n, \ldots\}$	$S \to \Lambda \vert aS$
$\{\Lambda, ab, aabb, \ldots, a^n b^n, \ldots\}$	$S \to \Lambda \vert aSb$
$\{\Lambda, ab, abab, \ldots, (ab)^n, \ldots\}$	$S \to \Lambda \vert abS$
$\{b, bbb, \ldots, b^{2n+1}, \ldots\}$	$S \to b \vert bbS$
$\{b, abc, aabcc, \ldots, a^n bc^n, \ldots\}$	$S \to b \vert aSc$
$\{ac, abc, abbc, \ldots, ab^n c, \ldots\}$	$S \to aBc$
	$B \to \Lambda \vert bB$

Combining Grammars

- Given two grammars, by renaming nonterminal symbols we can assume that they have disjoint sets of nonterminals.
- Suppose M and N are languages whose grammars have disjoint sets of nonterminals.
- Suppose also that the start symbols for the grammars of M and N are A and B, respectively.

Union Rule: The language $M \cup N$ starts with the two productions

$$S \to A|B$$

Product Rule: The language $M \cdot N$ starts with the production

$$S \to AB.$$

Closure Rule: The language M^* starts with the production

$$S \to AS|\Lambda.$$

Example: Union Rule

- Write a grammar for the language
$$L = \{\Lambda, a, b, aa, bb, \ldots, a^n, b^n, \ldots\}.$$
L is the union of the two languages
$$M = \{a^n : n \in \mathbb{N}\} \quad \text{and} \quad N = \{b^n : n \in \mathbb{N}\}.$$
Recall that these have grammars:
$$A \to \Lambda | aA \quad \text{and} \quad B \to \Lambda | bB.$$
Thus, we can write a grammar for L as follows:
$$\begin{array}{lcll} S & \to & A|B & \text{union rule} \\ A & \to & \Lambda|aA & \text{grammar for } M \\ B & \to & \Lambda|bB & \text{grammar for } N \end{array}$$

Example: Product Rule

- Write a grammar for the language
$$L = \{a^m b^n : m, n \in \mathbb{N}\}.$$
L is the product of the two languages
$$M = \{a^m : m \in \mathbb{N}\} \quad \text{and} \quad N = \{b^n : n \in \mathbb{N}\}.$$
These have grammars:
$$A \to \Lambda | aA \quad \text{and} \quad B \to \Lambda | bB.$$
Thus we can write a grammar for L as follows:
$$\begin{array}{lcll} S & \to & AB & \text{product rule} \\ A & \to & \Lambda | aA & \text{grammar for } M \\ B & \to & \Lambda | bB & \text{grammar for } N \end{array}$$

Example: Closure Rule

- Construct the language L of all possible strings made up from zero or more occurrences of aa or bb.
  The language is
  $$L = \{aa, bb\}^*.$$
  The grammar for $\{aa, bb\}$ is
  $$A \to aa|bb.$$
  So we can write a grammar for L as follows:
  $$\begin{array}{lcll} S & \to & AS|\Lambda & \text{closure rule} \\ A & \to & aa|bb & \text{grammar for } \{aa, bb\} \end{array}$$
  We can simplify this grammar as follows:
  $$S \to aaS|bbS|\Lambda.$$

Example: Decimal Numerals

- Find a grammar for the language of decimal numerals.

  A decimal numeral is either a digit or a digit followed by a decimal numeral.

  The following grammar rules (inspired by the union rule) reflect this idea:

$$\begin{array}{rcl} S & \to & D|DS \\ D & \to & 0|1|2|3|4|5|6|7|8|9. \end{array}$$

  We can say that:

  - S is replaced by either D or DS;
  - D can be replaced by any decimal digit.

Example: Even Decimal Numerals

- Find a grammar for the language of decimal numerals for the even natural numbers.

  Each numeral must have an even digit on its right side.

  So an even decimal numeral is an even digit or it can be built as a decimal digit followed by an even decimal numeral.

  The following grammar will do the job:

$$\begin{array}{lcl} S & \to & E|DS \\ E & \to & 0|2|4|6|8 \\ D & \to & 0|1|2|3|4|5|6|7|8|9. \end{array}$$

Example: Palindromes

- A **palindrome** is a string that is the same when written in reverse order.
- Write a grammar for the set of all palindromes over the alphabet $A = \{a, b, c\}$.

  Let P be the start symbol.

  Then the language of palindromes over the alphabet A has the grammar
  $$P \to aPa|bPb|cPc|a|b|c|\Lambda.$$
- Use this grammar to derive the palindrome *abcba*.
  $$P \Rightarrow aPa \Rightarrow abPba \Rightarrow abcba.$$

Meaning and Ambiguity

- Let G be a grammar and $L(G)$ its language.
- The **meaning** of a string $w \in L(G)$ is the parse tree produced by a derivation of w in G.
- A grammar G is called **ambiguous** if its language $L(G)$ contains a string that has two different parse trees, i.e., two different meanings.
- This is equivalent to saying that some string has two distinct leftmost or two distinct rightmost derivations in G.

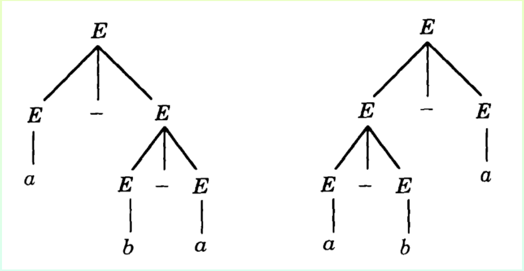
Example of an Ambiguous Grammar

- Suppose we define a set of arithmetic expressions by the grammar

$$E \to a|b|E - E.$$

This grammar is ambiguous because it has a string, namely, $a - b - a$, that has two distinct leftmost derivations:

$$E \Rightarrow E - E \Rightarrow a - E \Rightarrow a - E - E \Rightarrow a - b - E \Rightarrow a - b - a$$
$$E \Rightarrow E - E \Rightarrow E - E - E \Rightarrow a - E - E \Rightarrow a - b - E \Rightarrow a - b - a$$



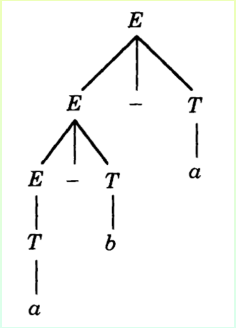
Disambiguation

- To make sure there is only one parse tree for every string in the language, we can try to find a different grammar for the same set of strings.
- Example: Suppose we want $a - b - a$ to mean $(a - b) - a$.

  To give the first minus sign higher precedence than the second, we introduce a new nonterminal:

$$\begin{aligned} E &\to E - T | T \\ T &\to a | b. \end{aligned}$$

  Notice that T can be replaced in a derivation only by either a or b.
  Therefore every derivation of $a - b - a$ produces the unique parse tree:



Subsection 4

Inductive Proofs

Theoretical Basis for Mathematical Induction

- **Theoretical Basis of Mathematical Induction**

  Let $m \in \mathbb{Z}$ and $W = \{n \in \mathbb{Z} : n \geq m\}$.
  Let S be a nonempty subset of W such that:

  1. $m \in S$;
  2. Whenever $k \in S$, then $k + 1 \in S$.

  Then $S = W$.

  To prove $S = W$ we use contradiction.

  Suppose $S \neq W$. Then $W - S$ has a least element x because every nonempty subset of W has a least element. Condition 1 tells us that $m \in S$. It follows that $x > m$. Thus $x - 1 \geq m$. It follows that $x - 1 \in S$. Apply the second condition to obtain $x = (x - 1) + 1 \in S$. This is a contradiction, since we cannot have both $x \in S$ and $x \in W - S$ at the same time. Therefore, $S = W$.

The Principle of Mathematical Induction

- **The Principle of Mathematical Induction**

  Let $m \in \mathbb{Z}$.
  To prove that a property $P(n)$ is true for all integers $n \geq m$, perform the following two steps:

  1. Prove that $P(m)$ is true;
  2. Assume that $P(k)$ is true for an arbitrary $k \geq m$.
     Prove that $P(k+1)$ is true.

  Let $W = \{n : n \geq m\}$, and let $S = \{n \in W : P(n) \text{ holds}\}$.

  Assume that we have performed the two steps of the principle.

  Then S satisfies the hypothesis of the theoretical basis.

  Therefore $S = W$.

  In other words, $P(n)$ is true for all $n \geq m$.

Example: A Closed Form

- Prove that the following property $P(n)$ holds, for all $n \geq 1$:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}.$$

We use the Principle of Mathematical Induction:

1. We show $P(1)$ holds:

$$1 = \frac{1(1+1)}{2}.$$

2. Assume that $P(k)$ holds for some $k \geq 1$.
   We prove that $P(k+1)$ holds.

$$\begin{array}{rcl} 1 + 2 + \cdots + k + (k+1) & = & (1 + 2 + \cdots + k) + (k+1) \\ & \overset{P(k)}{=} & \frac{k(k+1)}{2} + (k+1) \\ & = & (k+1)[\frac{k}{2} + 1] \\ & = & (k+1)\frac{k+2}{2} \\ & = & \frac{(k+1)((k+1)+1)}{2}. \end{array}$$

Example: Correctness of a Summation Program

- Show that the following if-then-else program computes the sum $1 + 2 + \cdots + n$ for all $n \geq 0$:

$$f(n) = \text{if } n = 0 \text{ then } 0 \text{ else } f(n-1) + n.$$

For each $n \in \mathbb{N}$, let $P(n)$ be the property that $f(n) = 1 + 2 + \cdots + n$. We use the Principle of Mathematical Induction to show that $P(n)$ holds for all n.

1. $f(0) = 0$ is correct.
2. Assume that $P(k)$ is true for some $k \in \mathbb{N}$.
   We must prove $P(k+1)$.

$$\begin{aligned} f(k+1) &= f(k+1-1) + (k+1) \quad \text{(definition of } f\text{)} \\ &= f(k) + (k+1) \\ &= (1 + 2 + \cdots + k) + (k+1) \quad (P(k)) \\ &= 1 + 2 + \cdots + (k+1). \end{aligned}$$

Example: Arithmetic Progressions

- An **arithmetic progression** with **first term** a and **difference** d is the sequence defined by

$$a,\ a+d,\ a+2d,\ \ldots,\ a_n = a+(n-1)d,\ \ldots$$

Prove that $a_1 + a_2 + \cdots + a_n = \frac{n(a_1+a_n)}{2}$.

We show that, for all $n \geq 1$, $a_1 + a_2 + \cdots + a_n = \frac{n(2a+(n-1)d)}{2}$.

1. For $n = 1$, $a_1 = \frac{1(2a+0\cdot d)}{2}$.
2. Suppose that $a_1 + \cdots + a_k = \frac{k(2a+(k-1)d)}{2}$.
   Now we have

$$\begin{array}{rcl} a_1+\cdots+a_k+a_{k+1} & = & (a_1+\cdots+a_k)+a_{k+1} \\ & = & \frac{k(2a+(k-1)d)}{2}+a+kd \\ & = & \frac{k(2a+kd-d)+2a+2kd}{2} \\ & = & \frac{k(2a+kd)-kd+2a+2kd}{2} \\ & = & \frac{k(2a+kd)+2a+kd}{2} \\ & = & \frac{(k+1)(2a+kd)}{2}. \end{array}$$

Example: Geometric Progressions

- A **geometric progression** with **first term** a and **ratio** r is the sequence defined by

$$a,\ ar,\ ar^2,\ \ldots,\ a_n = ar^{n-1},\ \ldots$$

- Prove that, for all $n \geq 0$,

$$a + ar + ar^2 + \cdots + ar^n = \frac{a(r^{n+1} - 1)}{r - 1}.$$

We use the Principle of Mathematical Induction:

1. If $n = 0$, $a = \frac{a(r-1)}{r-1}$.
2. Assume that, for some $k \geq 0$, $a + ar + ar^2 + \cdots + ar^k = \frac{a(r^{k+1}-1)}{r-1}$.
   Then, $a + ar + ar^2 + \cdots + ar^k + ar^{k+1}$

$$\begin{aligned} &= \frac{a(r^{k+1}-1)}{r-1} + ar^{k+1} = \frac{a(r^{k+1}-1)+(r-1)ar^{k+1}}{r-1} \\ &= \frac{ar^{k+1}-a+ar^{k+2}-ar^{k+1}}{r-1} = \frac{a(r^{(k+1)+1}-1)}{r-1}. \end{aligned}$$

Principle of Generalized Induction

- **The Principle of Generalized Induction**

  Let $m \in \mathbb{Z}$.
  To prove that $P(n)$ is true for all integers $n \geq m$, perform the following two steps:

  1. Prove that $P(m)$ is true;
  2. Assume that n is an arbitrary integer $n > m$, and assume that $P(k)$ is true for all k, $m \leq k < n$.
     Prove that $P(n)$ is true.

The Fundamental Theorem of Arithmetic

- Every natural number $n \geq 2$ is a product of primes.

  For $n \geq 2$, let $P(n)$ be the statement "n is a product of prime numbers".

  We need to show that $P(n)$ is true for all $n \geq 2$.

  1. Since 2 is prime, it follows that $P(2)$ is true.
  2. Assume that $n > 2$ and $P(k)$ is true for $2 \leq k < n$.
     We must show that $P(n)$ is true.
     If n is prime, then $P(n)$ is true.
     If n is not prime, then $n = xy$, where $2 \leq x < n$ and $2 \leq y < n$. By our assumption, $P(x)$ and $P(y)$ are both true, i.e., x and y are products of primes. But, then $n = xy$ is also a product of primes.
     So $P(n)$ is true.

  By the Principle of Generalized Induction, $P(n)$ is true for all $n \geq 2$.

Correctness of a Grammar

- Show that the grammar G

  $$S \to a|Sb$$

  generates the language $L = \{ab^n : n \in \mathbb{N}\}$.
  We first show $L \subseteq L(G)$.
  For each $n \in \mathbb{N}$, let $P(n)$ be the statement "There is a derivation of ab^n in G".

  1. If $n = 0$, then the production $S \to a$ gives a derivation of $a = ab^0$.
  2. Assume that $P(k)$ is true for some $k \geq 0$.
     We show that $P(k+1)$ is true. Since $P(k)$ is true, there is a derivation $S \Rightarrow^+ ab^k$. The last step of the derivation must use the production $S \to a$:

     $$S \Rightarrow^* Sb^k \Rightarrow ab^k.$$

     Replacing this step with two steps, we get

     $$S \Rightarrow^* Sb^k \Rightarrow Sbb^k \Rightarrow ab^{k+1}.$$

     This gives a derivation $S \Rightarrow^+ ab^{k+1}$. Therefore, $P(k+1)$ is true.

Correctness of a Grammar (Converse)

- Next, we show $L(G) \subseteq L$.

  For each $k \in \mathbb{N} - \{0\}$, let $P(k)$ be the statement "Any derivation of length n (that derives a terminal string) derives a string in L".

  1. For $k = 1$, there is only one derivation of length 1 that uses the production $S \to a$, and $a = ab^0 \in L$.
  2. Assume that $P(k)$ is true for some $k \geq 1$.

     We show that $P(k+1)$ is true. Any derivation of length $k+1$ must start with the production $S \to Sb$. The remaining k steps of the derivation take the form $Sb \Rightarrow^+ xb$, where $S \Rightarrow^+ x$ is a derivation of length k:

     $$S \Rightarrow Sb \Rightarrow^+ xb, \text{ with } S \Rightarrow^+ x.$$

     Since $P(k)$ is true, it follows that $x \in L$. Therefore, $xb \in L$. In other words, $P(k+1)$ is true.

Inducting on a Single Variable

- Prove that the following function computes the number y^{x+1} for any natural numbers x and y:

$$f(x,y) = \text{if } x = 0 \text{ then } y \text{ else } f(x-1,y)y.$$

In other words, show that $f(x,y) = y^{x+1}$, for all $(x,y) \in \mathbb{N} \times \mathbb{N}$.

We use induction on the variable x:

1. The definition of f gives $f(0,y) = y = y^{0+1}$.
2. Assume that $x > 0$ and $f(n,y) = y^{n+1}$ for $n < x$.
   We must show $f(x,y) = y^{x+1}$.
   The definition of f and the induction assumption give the following:

$$\begin{aligned} f(x,y) &= f(x-1,y)y \quad \text{(definition of } f\text{)} \\ &= y^{x-1+1}y \quad \text{(hypothesis)} \\ &= y^{x+1}. \end{aligned}$$

Proofs About Inductively Defined Sets

- Let S be the set defined inductively by:

Basis: $1 \in S$;

Induction: If $x \in S$, then $x + 2 \in S$.

Prove that $S = \{2k + 1 : k \in \mathbb{N}\}$.

Let $T = \{2k + 1 : k \in \mathbb{N}\}$.

We want to prove $S = T$.

Recall this entails showing $S \subseteq T$ and $T \subseteq S$.

$S \subseteq T$: This part requires showing that T satisfies the basis and the induction steps in the definition of S, since, by the closure step, S is the smallest set satisfying these conditions.

$T \subseteq S$: This part is usually done by induction.

Proofs About Inductively Defined Sets (Cont'd)

- We show $S \subseteq \{2k+1 : k \in \mathbb{N}\}$.
  - Clearly, $1 \in \{2k+1 : k \in \mathbb{N}\}$.
  - Suppose that $x \in \{2k+1 : k \in \mathbb{N}\}$.
    Then there exists $n \in \mathbb{N}$ such that $x = 2n+1$.
    Therefore $x+2 = 2n+1+2 = 2(n+1)+1 \in \{2k+1 : k \in \mathbb{N}\}$.

  Since T satisfies the basis and induction steps defining S, we have, by closure, that $S \subseteq T$.
- Now we show $\{2k+1 : k \in \mathbb{N}\} \subseteq S$.
  We prove $2k+1 \in S$ by induction on $k \in \mathbb{N}$.
  1. For $k = 0$, $1 \in S$ by the basis step.
  2. Suppose $2k+1 \in S$.
     We must show that $2(k+1)+1 \in S$.
     We have $2(k+1)+1 = (2k+1)+2 \in S$, by the hypothesis and the induction step.