COURSENOTES

# CS5034: Models of Computation

Clifford A. Shaffer
Department of Computer Science
Virginia Tech

# Forms of Proof

Proofs:

- Truth Tables
- Proof by Example
  - Existence proof and disproving only
- Proof by Exhaustive Checking
- Direct Proof
  - ad hoc
  - chain of inference
- Indirect Proof
  - Proof of Contrapositive
  - Proof by Contradiction
- If and Only If
- Induction (later)

# Set Notation

$\mathbb{N}$: Natural Numbers

$\mathbb{Z}$: Integers

$\mathbb{Q}$: Rational Numbers

$\mathbb{R}$: Real Numbers

If $S = \{a, b, c\}$, then

$$\text{power}(S) = \{ \emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

# Sets, Bags and Tuples

Bag or Multiset: A collection of objects that may contain a finite number of redundant occurrences of elements.

$$[h, h, g, u]$$

Tuples have $n$ **members** (tuple of **length** $n$) that appear in an **order**.

$$<12, R, 9>$$

Three elements, the first is 12, the second is $R$, the third is 9.

- Set: $\{h, u, g, h\} = \{h, u, g\} = \{u, g, h\}$.
- Bag:
  $[h, u, g, h] = [h, h, g, u] \neq [h, u, g] = [u, g, h]$.
- Tuple:
  $<h, u, g, h> \neq <h, h, g, u>$ and
  $<h, u, g> \neq <u, g, h>$ .

# Russell's Paradox

(An example of the need for clear definitions.)

Sets should not be "too complicated."

Ex: Never let a set be defined by a test that checks whether a set is a member of itself.

$$T = \{\text{A} \mid \text{A is a set and A} \notin \text{A}\}.$$

Is $T \in T$?

Redefine sets using a hierarchy of **types**.

Lowest type: $T_0$: individual elements.

Other types $T_i$ contain only sets whose elements are from the next lower type in the hierarchy, $T_{i-1}$.

$T = \{A \mid A \text{ is a set}\}$. Not legal.
$T = \{A \mid A \text{ has type } T_k\}$. Clearly, $T \notin T$.

# Tuples are Sets

Goal: Define tuples as special sets such that, for example, two ordered pairs $<a, b>$ and $<c, d>$ are equal iff $a = c$ and $b = d$.

O-tuple $<>$: Simply the empty set.

1-tuple: $<x>$ is just $\{x\}$.

Ordered pair: Define $<x, y> = \{\{x\}, \{x, y\}\}$.

Now, $<a, b> = \{\{a\}, \{a, b\}\}$ and
$<b, a> = \{\{b\}, \{b, a\}\} = \{\{b\}, \{a, b\}\}$.

Ordered pairs $<a, b>$ and $<c, d>$ are now equal iff $a = c$ and $b = d$.

Note: Read about relations.

# Functions

Programs can be functions and functions can be programs.

Suppose $A$ and $B$ are sets. Then a **function** is an association between $A$ and $B$ such that each element of $A$ is associated with exactly one element of $B$.

WARNING: The possible associations are much richer than the relatively small set of associations that can be defined by equations.

$A \to B$ denotes the set of all functions from $A$ to $B$.

$$f : A \to B$$

$f$ has **type** $A \to B$.

$A$ is the **domain** of $f$, $B$ is the **codomain** of $f$.

$$\text{range}(f) = \{f(a) \mid a \in A\}.$$

A function can have multiple arguments.

# Functions (cont.)

The **image** of $C \subset A$, denoted $f(C)$, is

$$f(C) = \{f(x) \mid x \in C\}.$$

If $D \subset B$, then the **pre-image** of $D$ under $f$, denoted $f^{-1}(D) = \{a \in A \mid f(a) \in D\}$.

Note: $f^{-1}(B) = A$.

Tuples can be viewed as functions. Ex: The tuple

$$<22, 14, 55, 1, 700, 67>$$

can be considered a listing of the values of a function

$$f : \{0, 1, 2, 3, 4, 5\} \to \mathbb{N}.$$

$f(0) = 22$; $f(2) = 55$.

# Partial Function

A **partial function** from $A$ to $B$ is like a function except that it might not be defined for some elements of $A$.

Example: division is a partial function of type $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$.

A **total function** is defined on all its domain.

# Composing Functions

Given: functions $f : A \to B$ and $g : B \to C$.

The **composition** of $f$ and $g$ is the function $g \circ f : A \to C$ defined by

$$(g \circ f)(x) = g(f(x)).$$

Composition is associative:

$$((f \circ g) \circ h)(x) = (f \circ (g \circ h))(x) = f(g(h(x))).$$

# Tupling Functions

Given $f : A \to B$, $g : A \to C$, and $h : B \times C \to D$, we can define function $< f, g >$ by

$$< f, g > (x) = < f(x), g(x) > .$$

The function $< f, g >$ has type $A \to B \times C$.

Notation:

$$h(x) = f(x) + g(x)$$

$$h(x) = (+ \circ < f, g >)(x)$$

$h = + \circ < f, g >$ (Cancel variable).

# Selector Function

A **selector function** selects one component of a tuple.

$$\mathbf{2}(x, y, z) = y.$$

Example:

$$\begin{aligned}
&\text{max3}(x, y, z) \\
&\quad = \text{max}(\text{max}(x, y), z) \\
&\quad = \text{max}(\text{max}(\mathbf{1}(x, y, z), \mathbf{2}(x, y, z)), \mathbf{3}(x, y, z)) \\
&\quad = (\text{max} \circ <\text{max} \circ <\mathbf{1}, \mathbf{2}>, \mathbf{3}>)(x, y, z).
\end{aligned}$$

$$\text{max3} = \text{max} \circ <\text{max} \circ <\mathbf{1}, \mathbf{2}>, \mathbf{3}>.$$

# Higher Order Functions

A **higher order function** may have functions for its arguments and value.

Consider $f : A \to B$ and $g : B \to C$. Write $g \circ f$ in prefix form: $\circ(f, g)$.

$$(A \to B) \times (B \to C) \to (A \to C).$$

Map function:

$$\begin{aligned} &\text{map}(+)(<<1,2>,<3,4>,<5,6>>) \\ &\quad =<+<1,2>,+<3,4>,+<5,6>> \\ &\quad =<3,7,11> . \end{aligned}$$

Apply function: Given function $f : A \to B$ and an element $x \in A$, return the result of applying $f$ to $a$.

Example:

$$\begin{aligned} g(f,x) &= f(x) + x \\ &= (+\circ <\text{apply}, \mathbf{2}>)(f,x). \end{aligned}$$

$$g = +\circ <\text{apply}, \mathbf{2}> .$$

# Properties of Functions

A function $f : A \to B$ is called **one to one** or **injective** if no two elements in $A$ map to the same element in $B$.

Equivalently:

For all $x, y \in A$, if $f(x) = f(y)$ then $x = y$.

For all $x, y \in A$, if $x \neq y$ then $f(x) \neq f(y)$.

A function $f : A \to B$ is called **onto** or **surjective** if each element $b \in B$ can be written as $b = f(x)$ for some element $x$ in $A$.

Equivalently: range$(f) = B$.

A function $f : A \to B$ is called **bijective** if it is both injective and surjective (equivalently: one to one and onto).

Every bijective function $f : A \to B$ has an **inverse function** $g : B \to A$ defined by $g(b) = a$ if $f(a) = b$.

**The Pigeonhole Principle**: If $A$ and $B$ are nonempty finite sets and $|A| > |B|$, then there is no one-to-one function from $A$ to $B$.

# Cardinality

Two sets have the **same cardinality** if there is a bijection between them.

Notation: $|A| = |B|$.

This concept can also be applied to infinite sets.

Example: Let Odd and Even be the sets of odd and even natural numbers, respectively.
Then, $|\text{Odd}| = |\text{Even}|$ because the function $f : |\text{Odd} \to \text{Even}|$ defined by $f(x) = x - 1$ is a bijection.

How about $|\text{Even}| = |\mathbb{N}|$?

# Counting Infinite Sets

A set $C$ is **countable** if it is finite or if $|C| = |\mathbb{N}|$.

If a set is not countable, then it is **uncountable**.

If $A$ is a finite alphabet, then $A^*$ is countably infinite.

Proof: Arrange the strings in order by length, and within a given length by alphabetical order. This provides a bijection.

As a corollary, the set of all computer programs is countable.

# Diagonalization

Theorem: The set power($\mathbb{N}$) is uncountable.

Proof by contradiction: Assume there exists a bijection $f : \mathbb{N} \to$ power($\mathbb{N}$).

power($\mathbb{N}$) can be enumerated as power($\mathbb{N}$) $= \{S_0, S_1, ...\}$ where $S_i(x) = 1$ if $x$ is in the $i$'th set of the enumeration, and 0 if it is not.

Now, construct a new set $D$ by setting $D(x) = 0$ if $S_x(x) = 1$, and setting $D(x) = 1$ otherwise.

Since $D$ corresponds to a set of natural numbers, it should be $S_k$ for some number $k$.

Is $k \in S_k$? NO, because it differs from $S_k$ at the $k$'th position.

By contradiction, the assumption that there is a bijection must be false.

# Diagonalization (cont.)

Visualization: consider a 2D array, with position $j$ of row $i$ being 1 if $j \in S_i$, 0 otherwise.

The complement of the diagonal is not in any set.

Note: In general, any description for an algorithm must be formulated as a string of finite length. What is the implication of this, combined with the proof that |power($\mathbb{N}$)| is uncountable?

Relationship to language:

- Consider a language to be a subset of the strings generated by some alphabet.
- This language is one row of the array.
- A program "recognizes" a language.
- Not all languages can have a corresponding program.

# Inductively Defined Sets

Define set $A = \{3, 5, 7, 9, ...\}$.

We have an expectation that this means something like:

$$A = \{2k + 3 | k \in \mathbb{N}\}$$

Alternatively:

- $3 \in A$.
- if $x \in A$, then $x + 2 \in A$.
- Nothing else is in $A$.

An **inductive definition** for a set consists of:

- Basis: a finite list of one or more elements.
- Induction: One or more rules for creating new elements from existing elements.
- Closure: State that only elements in the basis or derived by the induction are in the set (usually assumed).

# Examples

Natural Numbers:

- Basis:

- Induction:

Note: succ($n$) $= n + 1$.

What about $A = \{2, 3, 4, 7, 8, 11, 15, 16, ...\}$?

Consider $A = B \cup C$ for

$B = \{2, 4, 8, 16, ...\}$ and $C = \{3, 7, 11, 15...\}$.

# Natural Numbers Defined by Sets

Inductively define "Nat" using only set notation.

- Basis: $\emptyset$
- Induction: If $s \in$ Nat, then $s \cup \{s\} \in$ Nat.

Consequence:

- 0 is:
- 1 is:
- 2 is:

Note that $2 = \{0, 1\}$
Note that succ$(x) = x \cup \{x\}$.

# Lists

Given: list $< x, y, z >$.

Useful functions:

- head($< x, y, z >$) $= x$.
- tail($< x, y, z >$) $=< y, z >$.
- cons($x, < y, z >$) $=< x, y, z >$.
- cons($x, <>$) $=< x >$.

Inductive definition for Lists[$A$], all lists made from set $A$:

- Basis: $<> \in$ Lists[$A$].
- Induction: If $x \in A$ and $t \in$ Lists[$A$], then cons($x, t$) $\in$ Lists[$A$].

Inductively define the set $S$ of all non-empty lists over the set $\{0, 1\}$, where the elements in each list alternate between 0 and 1.

- Basis:
- Induction:

# Strings

Empty String: $\Lambda$.
Append letter $a$ to string $s$: $a \cdot s$.

$a \cdot \Lambda = a\Lambda = a$.

Given alphabet $A$, define $A^*$.

- Basis:
- Induction:

For $A = \{0, 1\}$, create set $S \subset A^*$ such that no string in S contains a leading zero except 0.

- Basis:
- Induction:

# Product Sets

Define $\mathbb{N} \times \mathbb{N}$.

- Basis: $< 0, 0 > \in \mathbb{N} \times \mathbb{N}$.
- Induction: If $< x, y > \in \mathbb{N} \times \mathbb{N}$, then $< \mathsf{succ}(x), y >$, $< x, \mathsf{succ}(y) > \in \mathbb{N} \times \mathbb{N}$.

# Languages

For alphabet $A$, a **language** over $A$ is a collection of strings whose components come from $A$.

$A^*$ is the biggest possible language over $A$, all other possible languages over $A$ are subsets of $A^*$.

Four languages over $A$: $\emptyset, \{\Lambda\}, A, A^*$

A string in the language is called a **well formed formula** (wff).

Concatenation of string:

$$\text{cat}(aab, ba) = aabba.$$

# Grammar Rules

If $L$ is a language over an alphabet $A$, then a grammar for $L$ consists of a set of **grammar rules** of the form:

$$\alpha \to \beta$$

where $\alpha$ and $\beta$ denote strings of symbols taken from $A$ and from a set of grammar symbols disjoint from $A$.

A grammar rule is also called a **production**.

Every grammar has a **start symbol** $S$ and at least one production of the form:

$$S \to \beta.$$

Example: For $A = \{a, b, c\}$, $A^*$ can be defined by the grammar:

$$\begin{aligned} S &\to \Lambda \\ S &\to aS \\ S &\to bS \\ S &\to cS. \end{aligned}$$

# Derivation

A **derivation** for a string is a series of productions that lead from the start symbol to the string.

Ex: Derive string $aab$.

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aabS \Rightarrow aab\Lambda = aab.$$

Notation:

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

# Definition of a Grammar

A grammar has four parts.

- An alphabet $N$ of **nonterminals**
- An alphabet $T$ of **terminals**. Note: $N \cap T = \emptyset$.
- A specific nonterminal called the **start symbol**.
- A finite set of productions of the form $\alpha \to \beta$, where $\alpha$ and $\beta$ are strings over the alphabet $N \cup T$ with these restrictions:
  - $\alpha$ is not the empty string.
  - A least one production has only the start symbol on the left hand side.
  - Each non-terminal must appear on the left side of some production.

# Grammars and Languages

If $G$ is a grammar, then the language of $G$ is:

$$L(G) = \{s | s \in T^* \text{ and } S \Rightarrow^+ s\}.$$

Example: infinite language $\{a^n b | n \geq 0\}$ can be described by the grammar:

$$S \rightarrow b | aS.$$

A grammar for an infinite language must be recursive.

Any language defined by a grammar is an inductively defined set.

# Grammar Examples

Define a grammar for even decimal numbers.

Define a grammar for identifiers, where an identifier contains digits or letters, but must start with a letter.

# Ambiguity

A grammar is **ambiguous** if its language contains some string that has two different parse trees

- Equivalent: Two distinct leftmost or rightmost derivations.

Example:

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

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

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

We can solve this problem with the following grammar:

- $E \rightarrow E - T|T$
- $T \rightarrow a|b.$

# Notation for Recursive Functions

Sum the first $n$ odd numbers

Pattern-matching definition:

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

Conditional form:

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

Alternative conditional form:

- $f(n) =$ if $n = 0$ then 1 else $f(n-1) + 2n + 1$.

# Example: Recursively Construct the Power Set

power(<>) =<<>>,
power($a :: L$) =

# Streams

A stream is an infinite list.

Any inductively defined countable set can be represented by a stream.

We can write recursive functions to process streams.

Ex: Sum the first $n$ elements of a stream of integers.

```
sum(n, s) =
  if n = 0 then 0
           else head(s) + sum(n-1, tail(s)).
```

Define ints(4) to be the stream of integers beginning with 4.

What is sum(3, ints(4))?

# Prime Sieve

Starting with stream ints(2), "sieve" out all the non-primes by extracting all multiples of 2, then 3, etc.

sieve(ints(2)) starts us.

Assume removeM($n$, $t$) removes all multiples of $n$ (including $n$) from stream $t$.

sieve($n$, $t$) = $n$::sieve(removeM($n$, $t$)).

To remove the extra parameter:

sieve($s$) =
  head($s$)::sieve(removeM(head($s$),tail($s$))).

$m$ is a multiple of $n$ iff $m$ mod $n = 0$.

```
removeM(n, t) = if head(t) mod n = 0
                then removeM(n, tail(t))
                else head(t)::removeM(n, tail(t)).
```

# Binary Relations

A binary relation $R$ over set $A$ can be represented as a subset of $A \times A$.

Special properties that may be held by binary relations:

- **Reflexive**: $\forall a \in A, aRa$.
- **Symmetric**: $\forall a, b \in A$,
  if $aRb$ then $bRa$.
- **Transitive**: $\forall a, b, c \in A$,
  if $aRb$ and $bRc$, then $aRc$.
- **Irreflexive**: $\forall a \in A,\ a \not R a$.
- **Antisymmetric**: $\forall a, b \in A$,
  if $a \neq b$ and $aRb$, then $b \not R a$.
  Alt: if $aRb$ and $bRa$, then $a = b$.

# Examples

- isParentof
- isSiblingOf
- hasSameBirthdayAs
- $<$
- $\leq$
- $=$
- $\{< n, n+1 > | n \in \mathbb{N}\}$

# Composing Binary Relations

isGrandparentof: $< a, c > \in$ isGrandparentof iff there exists some $b$ such that $< a, b >, < b, c > \in$ isParentof.

$< a, c > \in R \circ S$ iff $< a, b > \in R$ and $< b, c > \in S$ for some $b \in B$.

isGrandparentof $=$ isParentOf $\circ$ isParentOf.

Is $R \circ S = S \circ R$?

greater $\circ$ less $= \mathbb{R} \times \mathbb{R}$.

equal $\circ$ notEqual $=$

notEqual $\circ$ notEqual $=$

Since relations are sets of ordered pairs, they can be combined by set operations.

- equal $\cap$ less $=$
- less $\cup$ greater $=$

# Properties of Composition

- $R \circ (S \circ T) = (R \circ S) \circ T$. (Associativity)
- $R \circ (S \cup T) = R \circ S \cup R \circ T$.
- $R \circ (S \cap T) \subset R \circ S \cap R \circ T$.

# Self-Composition

$R^n$: $R$ composed with itself $n$ times.

Define inductively:

- $R^0 = \{< a, a > | a \in A\}$.
- $R^{n+1} = R^n \circ R$.

Note:

$R^1 = R^{0+1} = R^0 \circ R = \{< a, a > | a \in A\} \circ R = R$.

$R^n$ inherits the reflexive, symmetric and transitive properties of $R$, but not the irreflexive or antisymmetric properties.

# Closure

For binary relation $R$ and property $p$, $p(R)$ denotes the "$p$ closure of $R$."

$p(R)$ is the smallest possible relation containing $R$ and satisfying $p$.

If $R$ satisfies $p$, then what is $p(R)$?

- Reflexive closure: $r(R)$
- Symmetric closure: $s(R)$
- Transitive closure: $t(R)$

Example: For $A = \{a, b, c\}$,

$$R = \{< a, a >, < a, b >, < b, a >, < b, c >\}.$$

$r(R) =$

$s(R) =$

$t(R) =$

# Closures (cont.)

If $R$ is a binary relation over a set $A$, then

- $r(R) = R \cup R^0$.
- $s(R) = R \cup R^c$. ($R^c$ is the converse relation)
- $t(R) = R \cup R^2 \cup R^3 \cup \cdots$
- If $A$ is finite with $n$ elements,
  then $t(R) = R \cup R^2 \cup \cdots \cup R^n$.

Examples:

- $r(<) =$
- $s(<) =$
- $t(<) =$
- $r(\neq) =$
- $s(\neq) =$
- $t(\neq) =$

Notation:

- $R^+$ denotes $t(R)$.
- $R^*$ denotes $rt(R)$.

# Equality

What does "equality" mean?

Basic equality: $\{< x, x > | x \in A\}$.

What about equality of arithmetic expressions?

- Here, two syntactic objects are equal if they have the same value (semantics).

In general, what about reflexive, symmetric, and transitive properties?

Any binary relation that is reflexive, symmetric and transitive is called an **equivalence relation**.

# Equivalence Example

Two binary trees are equivalent if they have the same structure (ignore node values).

Basis:

Induction:

What if we don't care about left/right order?

# Partitioning

An equivalence relation on a set partitions the set into **equivalence classes**.

- Notation: $[x] = \{y | y \in S \text{ and } xRy\}$.

What are the equivalence classes for equality on set $\{a, b, c\}$?

A partitioning can also be used as the definition for equivalence.

$$\{\{a, b\}, \{c, d, e\}\}$$

```
if x > 0 then q(x);
         else if x is even then r(x);
                           else s(x);
```

This program has infinite inputs. But, it has a finite number of equivalence classes, so all paths can be tested.

# Generating Equivalence Relations

What is the smallest equivalence relation containing a relation $R$?

Apply closure. Does it matter in what order?

- $A = \{a, b, c\}$
- $R = \{< a, b >, < a, c >, < b, b >\}$

What is $str(R)$?

What is $tsr(R)$?

**Theorem**: $tsr(R)$ is the smallest equivalence relation on $R$.

# Kernel Relations

Given function $f : A \to B$, relate elements $x$ and $y$ if $f(x) = f(y)$.

This is called the **kernel relation** of $f$ on $A$, and is an equivalence relation (why?).

Example: $f$ takes an English word and returns the set of letters in the word.

- What is the domain?
- What is the codomain?
- What is [too]?

What changes if $g$ takes an English word and returns the BAG of letters in the word?

# Meaning and Equality

If objects have no established meaning, use simple equality.

Otherwise, use the kernel relation.

Example: Let $S$ denote the set of arithmetic expressions on nonempty unary strings.

- For string $1^n$, $m(1^n) = n$.
- For $e + e'$ an expression in $S$, $m(e + e') = m(e) + m(e')$.
- Evaluate $+$ left-to-right.

$$\begin{aligned} m(1 + 111 + 11) &= m((1 + 111) + 11) \\ &= m(1 + 111) + m(11) \\ &= m(1) + m(111) + 2 \\ &= 1 + 3 + 2 \\ &= 6 \end{aligned}$$

(Don't get confused by the two different meanings of '+'!)

# Maintaining Equivalences

Problem: Given a set of objects and a series of stated equivalence pairs, how do we compute the equivalence classes efficiently?

Answer: Parent Pointer tree structure (UNION/FIND algorithm).

Example: Set $A$ is the integers from 1 to 10.

$$1 \sim 8, 4 \sim 5, 9 \sim 2, 4 \sim 10, 3 \sim 7, 6 \sim 3, 4 \sim 9$$

# Order Relations

What are the properties of an ordering relationship?

- Reflexive?
- Symmetric?
- Transitive?
- Irreflexive?
- Antisymmetric?

Consider the following **partial order**:

# Partial Orders

A binary relation is a **partial order** if it is antisymmetric and transitive.

The set is a **partially ordered set** or **poset**.

- Given Set $S$ and Relation $R$ that makes $S$ a poset, write $\langle S, R \rangle$.
- $\langle \mathbb{N}, < \rangle$.
- $\langle \mathbb{N}, \leq \rangle$.
- $\langle \text{power}(\{a, b, c\}), \subset \rangle$.

It is called "partial" because some elements are not related.

$x$ and $y$ are **comparable** if either $xRy$ or $yRx$.

# Total Order

**Total order**: every pair of distinct elements in a partial order are comparable.

Example: $a|b$ means "$a$ divides $b$", that is, $b$ is a multiple of $a$.

- $\langle \mathbb{N}, | \rangle$.
- $\langle \{2^n | n \in \mathbb{N}\}, | \rangle$.
- $\langle \{1, 3, 9, 45\}, | \rangle$.
- $\langle \{1, 2, 3, 4\}, | \rangle$.

Notation:

- $a \prec b$ for partial order. $(<)$
- $a \preceq b$ for reflexive partial order. $(\leq)$

# Chains

A set of elements in a poset is called a **chain** if all the elements are comparable – linked – to each other.

$$x_1 \prec x_2 \prec x_3 \prec \cdots$$

Example: For $\langle \text{power}(\{a, b, c\}, \subset \rangle$,

$$\{a, b, c\} \subset \{a, b\} \subset \{a\} \subset \emptyset$$

# Bounds

$S$ is a non-empty subset of poset $P$, and $x \in S$.

$x$ is **minimal element** of $S$ if $x$ has no predecessors in $S$.

$x$ is the **least element** of $S$ if $x$ is minimal and $x \preceq y$ for all $y \in S$.

For poset $\langle \mathbb{N}, | \rangle$,

- Subset $\{2, 4, 5, 10\}$
- Subset $\{2, 4, 12\}$
- $\mathbb{N}$

For poset $\langle \text{power}(\{a, b, c\}), \subset \rangle$,
what about subset $\{\{a, b\}, \{a\}, \{b\}\}$?

Similarly, define **maximal elements** and the **greatest element**.

# Bounds (cont.)

The set of positive rational numbers has no least element, but is bounded below by zero.

For $S$ a subset of $P$ and $x \in P$, $x$ is a **lower bound** of $S$ if $x \preceq y$ for all $y \in S$.

$x \in P$ is the **greatest lower bound** of $S$ if $x$ is a lower bound and $z \preceq x$ for all lower bounds $z$ of $S$.

Example: What is the glb for subset $\mathbb{Q}^+$ of poset $\langle \mathbb{Q}, \leq \rangle$?

Equivalently, **upper bound** and **least upper bound** (lub).

Example: Subset $\{2, 3\}$ of the following poset has upper bounds, but no least upper bound:

# Lattice

A **lattice** is a poset with the property that every pair of elements has a glb and a lub.

Examples:

- $\langle \mathbb{N}, \leq \rangle$.
- $\langle \text{power}(A), \subset \rangle$.

Diagrams:

# Well-Founded Orders

Every descending chain of natural numbers is finite in length. (Why?)

Is this true for integers?

A poset is a **well-founded set** if every descending chain of elements is finite.

Its partial order is a **well-founded order**.

Example:

- Power set of any finite set is well-founded by the subset relation.

What about "less" and the positive rationals?

Well-founded sets are basic tools for use in inductive proofs.

Property: If $A$ is a well-founded set, then every non-empty subset of $A$ has a minimal element.

Conversely, if every non-empty subset of $A$ has a minimal element, then $A$ is well-founded.

# Examples of Orders

Lexicographic order on $\mathbb{N}^k$:

$$\langle x_1, ..., x_k \rangle \prec \langle y_1, ..., y_k \rangle$$

iff there is an index $j \geq 1$ such that $x_j < y_j$ and for each $i < j, x_i = y_i$.

Is this a total ordering? Is it well-ordered?

Consider Dictionary ordering on $A^*$.

Is this a total ordering? Is it well-ordered?

What can we do?

# Inductively Defined Sets

An inductively defined set $W$ can be made into a well-founded set.

Method 1:

Define a function $f : W \to \mathbb{N}$ as follows:

- $f(c) = 0$ for all basis elements $c$ of $W$.
- If $x \in W$ and $x$ is constructed from elements $y_1, ..., y_n$ in $W$, then define $f(x) = 1 + \max\{f(y_1), ..., f(y_n)\}$.
- Let $x \prec y$ mean $f(x) < f(y)$.

Method 2:

The ordering $\prec$ is defined as follows:

- Let the basis elements of $W$ be minimal elements.
- If $x \in W$ and $x$ is constructed from elements $y_1, ..., y_n$ in $W$, then define $y_i \prec x$ for each $i = 1, ..., n$.
- The actual ordering is the transitive closure of $\prec$.

# A Basis of Mathematical Induction

Theorem: Let $m \in \mathbb{Z}$ and

$$W = \{n | n \in \mathbb{Z}, n \geq m\}.$$

Let $S$ be a nonempty subset of $W$ such that the following conditions hold:

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

Then $S = W$.

Proof: Suppose $S \neq W$.

- Then $W - S$ has a least element $x$. (Why?)
- Since $m \in S$, $x > m$.
- $x - 1 \geq m$ and $x - 1 \in S$.
- But, if $x - 1 \in S$, then $x \in S$.
- Therefore, $S = W$.

# Principle of Mathematical Induction

Let $m \in \mathbb{Z}$.

**IF** the following two statements are true:

1. $P(m)$ is true.
2. For $n > m$,
   $P(n-1)$ is true $\rightarrow$ $P(n)$ is true.

... **THEN** we may conclude: $\forall n \geq m$, $P(n)$.

Proof: Let $W = \{n | n \geq m\}$ and let
$S = \{n | n \geq m$ and $P(n) =$ TRUE$\}$.

- Assume the two steps of MI are true.
- Then $S$ satisfies the conditions of the basis.
- Therefore $S = W$.
- Therefore $P(n)$ is true for all $n \geq m$.

# Induction Example

**Theorem**: The sum of the first $n$ numbers is $n(n+1)/2$, for $n \geq 1$.

# Induction Example

**Theorem**: The following program computes the sum of 1 to $n$.

```
f(n) = if (n == 0) then return 0
       else return f(n-1) + n;
```

**Proof**: Let $P(n) =$ "$f(n) = 1 + 2 + ... + n$."

Base case: $f(0) = 0$.

Induction Hypothesis: Assume $P(k)$ is true.

Induction step:

$$\begin{aligned} f(k+1) &= f(k+1-1) + (k+1) \\ &= f(k) + (k+1) \\ &= (1 + 2 + ... + k) + (k+1) \\ &= 1 + 2 + ... + (k+1). \end{aligned}$$

Thus, if $P(k)$ true, $P(k+1)$ is also true.

By MI, $f(n) = 1 + ... + n$ for all $n \in \mathbb{N}$.

# The Basis of Well-Founded Induction

Let $W$ be a well-founded set, let $S$ be a nonempty subset of $W$ such that:

1. $S$ contains all the minimal elements of $W$.
2. Whenever an element $x \in W$ has the property that all of its predecessors are in $S$, then $x \in S$.

Then $S = W$.

Proof: Suppose $S \neq W$.

- Then $W - S$ has a minimal element $x$.
- All predecessors of $x$ must be in $S$. (Why?)
- By the second condition, $x \in S$
- Therefore $S = W$.

# Well-Founded Induction

Suppose $P(x)$ is a statement for each $x$ in a well-founded set $W$.

To prove that $P(x)$ is true for all $x$, show:

1. $P(m)$ is true for all minimal elements $m \in W$.
2. For $x$ an arbitrary element of $W$, assuming that $P(y)$ is true for all predecessors $y$ of $x$, prove $P(x)$ is true.

Proof: Let $S = \{x | x \in W$ and $P(x) =$ true$\}$.

- Assume that we have performed the steps of well-founded induction.
- Then $S$ satisfies the conditions of the basis.
- Therefore, $S = W$, that is, $P(x)$ is true for all $x \in W$.

Strong Induction: For $n > m$,

$P(i), i = 1, 2, ..., n - 1$ is true $\rightarrow P(n)$ is true.

# Induction Example

**Theorem**: For $n > 1$, $n$ is divisible by some prime number.

# Induction Example

**Theorem**: $2¢$ and $5¢$ stamps can be used to form any denomination (for denominations $\geq$ 4).

# Induction Example

**Theorem**: The set of regions formed by $n$ infinite lines in the plane can be two-colored.

# Induction Example

**Theorem**: The following function computes $y^{x+1}$ for any natural numbers $x$ and $y$.

```
f(x, y) = if (x == 0) then return y
          else return f(x-1, y)* y;
```

# Induction Example

**Theorem**: All horses are the same color.

**Proof**:

**Base Case**: $n = 1$ is easy.

**Induction Step**: Assume P($i$) for $i \leq n - 1$.

Let $S$ be a set of horses, $|S| = n$.

Let $S'$ be $S - \{h\}$ for some horse $h$.

By induction hypothesis, all horses in $S'$ have the same color.

Let $h'$ be some horse in $S'$.

Induction hypothesis implies $\{h, h'\}$ have all the same color.

Thus, we conclude by Mathematical Induction that the theorem holds.

# The Pigeonhole Principle

If $A$ and $B$ are nonempty finite sets and $|A| > |B|$, then there is no one-to-one function from $A$ to $B$.

Proof:

**Base case:** $|B| = 1$ (that is, $B = \{b\}$) and $|A| > 1$. For any $f : A \to B$ there are at least two distinct elements $a_1, a_2 \in A$, with $f(a_1) = f(a_2) = b$.

**Induction Hypothesis**: Suppose that $f : A \to B$ is not one-to-one, $|A| > |B|$ and $|B| \leq n$, where $n \geq 1$.

**Induction Step**: Suppose that $f : A \to B$ and $|A| > |B|$ for $|B| = n + 1 \geq 2$.

Pick some element $b \in B$.

If $|f^{-1}(b)| \geq 2$ then $f$ is not one-to-one by definition.

Otherwise, remove $b$ from $B$, remove $f^{-1}(b)$ from $A$, and set $g$ to be $f$ without $f^{-1}(b)$.

By the induction hypothesis, $g$ is not one-to-one.

# Closure Property of Inductive Definitions

If

- $S$ is an inductively defined set and
- $T$ is a set that also satisfies the basis and inductive cases for the definition of $S$ and
- $T \subset S$

... then $T = S$.

Example:

- Basis: $1 \in S$.
- Induction: If $x \in S$, then $x + 2 \in S$.

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

Base case: $1 = 2 * 0 + 1$.

Induction case: Assume $x = 2k + 1 \in T$.

Then $x + 2 = 2(k + 1) + 1 \in T$.

Show $T \subset S$.

Prove this by induction that $2k + 1 \in S$ for all $k \in (N)$.

# Induction Example

Find a grammar for language

$$M = \{ab^n | n \in \mathbb{N}\}.$$

How about: $S \to a|Sb$?

Use an inductive definition:

- Basis: $a \in L(G)$.
- Induction: If $x \in L(G)$, then put $xb$ in $L(G)$.

Now, we need only prove that $M = L(G)$ using induction.

# Calculus

A **calculus** is a language with

- Expressions.
- Rules for forming expressions.
- Values or meanings for expressions.
- Rules for transforming expressions into other expressions with the same value.

Examples:

- Calculus of real functions
- Propositional Calculus
- Predicate Calculus

The last two are "logical" calculi.

What is a proof?

# Propositional Calculus

We will use letters for **atomic formulas** or **variables**.

Truth Symbols (literals): true, false.

Connectives: $\neg$, $\wedge$ , $\vee$ , $\rightarrow$.

Punctuation: (, ).

The formula $(F \vee G)$ is a **disjunction**.

The formula $(F \wedge G)$ is a **conjunction**.

The formula $\neg F$ is a **negation**.

# Inductive Definition

A Well Formed Formula (wff) is:

- A truth symbol.
- A variable.
- The negation of a wff.
- The conjunction of two wffs.
- The disjunction of two wffs.
- The implication of one wff from another.
- A wff surrounded by parentheses.

Hierarchy of operators:

- $\neg$
- $\wedge$
- $\vee$
- $\rightarrow$

These operators are left associative.

Parentheses overrides precedence.

# Truth Table

Any wff has a unique truth table.

For $\neg P \rightarrow Q \wedge R$:

| $P$ | $Q$ | $R$ | $\neg P$ | $Q \wedge R$ | $\neg P \rightarrow Q \wedge R$ |
|---|---|---|---|---|---|
| $T$ | $T$ | $T$ | $F$ | $T$ | $T$ |
| $T$ | $T$ | $F$ | $F$ | $F$ | $T$ |
| $T$ | $F$ | $T$ | $F$ | $F$ | $T$ |
| $T$ | $F$ | $F$ | $F$ | $F$ | $T$ |
| $F$ | $T$ | $T$ | $T$ | $T$ | $T$ |
| $F$ | $T$ | $F$ | $T$ | $F$ | $F$ |
| $F$ | $F$ | $T$ | $T$ | $F$ | $F$ |
| $F$ | $F$ | $F$ | $T$ | $F$ | $F$ |

The "meaning" of a wff is its truth table.

**Tautology**: All truth table values are true.

**Contradiction**: All truth table values are false.

**Contingency**: Some truth values are true, some false.

# Equivalence

Two wffs are equivalent if their truth tables are identical.

There is no meaningful distinction between equivalent wffs.

Equivalence is an equivalence relation.

Any subwff of a wff can be replaced by an equivalent wff without changing the meaning.

# Useful Equivalences

| | |
|---|---|
| *Idempotency* | $(F \vee F) \equiv F$ |
| | $(F \wedge F) \equiv F$ |
| *Commutativity* | $(F \vee G) \equiv (G \vee F)$ |
| | $(F \wedge G) \equiv (G \wedge F)$ |
| | $(F \leftrightarrow G) \equiv (G \leftrightarrow F)$ |
| *Associativity* | $((F \vee G) \vee H) \equiv (F \vee (G \vee H))$ |
| | $((F \wedge G) \wedge H) \equiv (F \wedge (G \wedge H))$ |
| *Absorption* | $(F \vee (F \wedge G)) \equiv F$ |
| | $(F \wedge (F \vee G)) \equiv F$ |
| *Distributivity* | $(F \wedge (G \vee H)) \equiv$ |
| | $((F \wedge G) \vee (F \wedge H))$ |
| | $(F \vee (G \wedge H)) \equiv$ |
| | $((F \vee G) \wedge (F \vee H))$ |
| *Double Negation* | $\neg\neg F \equiv F$ |
| *DeMorgan* | $\neg(F \vee G) \equiv (\neg F \wedge \neg G)$ |
| | $\neg(F \wedge G) \equiv (\neg F \vee \neg G)$ |
| *Tautology* | If $F$ is a tautology: |
| | $(F \vee G) \equiv F$ |
| | $(F \wedge G) \equiv G$ |
| *Unsatisfiability* | If $F$ is a contradiction: |
| | $(F \vee G) \equiv G$ |
| | $(F \wedge G) \equiv F$ |

# A Proof

Theorem:

$$A \to (B \to C) \equiv B \to (A \to C).$$

Proof:

$$\begin{aligned} A \to (B \to C) &\equiv A \to (\neg B \vee C) \\ &\equiv \neg A \vee (\neg B \vee C) \\ &\equiv (\neg A \vee \neg B) \vee C \\ &\equiv (\neg B \vee \neg A) \vee C \\ &\equiv \neg B \vee (\neg A \vee C) \\ &\equiv B \to (\neg A \vee C) \\ &\equiv B \to (A \to C). \end{aligned}$$

# Checking Tautology

Can we check all of the assignments?

For $A$ a variable in wff $W$, $W$ is a tautology iff $W(A/true)$ and $W(A/false)$ are tautologies.

Example: For wff W:

$$[(A \wedge B \rightarrow C) \wedge (A \rightarrow B)] \rightarrow (A \rightarrow C).$$

Compute and simplify $W(A/T)$ and $W(A/F)$.

$$\begin{aligned} W(A/T) &\equiv [(T \wedge B \rightarrow C) \wedge (T \rightarrow B)] \rightarrow (T \rightarrow C) \\ &\equiv (B \rightarrow C) \wedge (T \rightarrow B)] \rightarrow (T \rightarrow C) \\ &\equiv [(B \rightarrow C) \wedge B] \rightarrow C. \end{aligned}$$

$$\begin{aligned} W(A/F) &\equiv [(F \wedge B \rightarrow C) \wedge (F \rightarrow B)] \rightarrow (F \rightarrow C) \\ &\equiv [(F \rightarrow C) \wedge T] \rightarrow T \\ &\equiv T. \end{aligned}$$

Setting $X \equiv [(B \rightarrow C) \wedge B] \rightarrow C$, we continue to completion.

# Truth Functions

A **truth function** is a function whose arguments are Boolean and whose result is Boolean.

Any truth table can be turned into a wff. (Why?)

# Normal Form

A **literal** is an atomic formula or the negation of an atomic formula.

A formula is in **conjunctive normal form** if it is a conjunction of disjunctions of literals.

A formula is in **disjunctive normal form** if it is a disjunction of conjunctions of literals.

Example:

- $((A \wedge B) \vee (\neg A \wedge C))$ is in disjunctive normal form.
- $((A \vee B) \wedge (B \vee C \vee \neg A) \wedge (A \vee C))$ is in conjunctive normal form.

**Theorem**: Every formula has at least one conjunctive normal form and at least one disjunctive normal form. There is an algorithm that transforms a formula $F$ into a conjunctive or disjunctive normal form.

# Complete Sets of Connectives

Do we need all of the connectives to create all wffs?

$$\{\neg,\ \wedge\ ,\ \vee\ ,\rightarrow\}$$

A set of connectives is **complete** if all wffs can be formed with that set.

Conjunctive, Disjunctive normal forms.

How about $\{\neg,\ \vee\ \}$?

$$A \wedge B \equiv \neg(\neg A \vee \neg B).$$

NAND, NOR

$$\neg P \equiv \mathsf{NAND}(P, P).$$

# Formal Reasoning Systems

A formal reasoning system requires

- A set of wffs.
- A set of axioms.
- Inference rules to map one or more wffs to a wff called the **conclusion** or **consequent**.

Axioms are the wffs that we use as a basis from which to "reason" using the inference rules.

A proof is a finite sequence of wffs with the property that each wff in the sequence either is an axiom, or can be inferred (using the inference rules) from previous wffs in the sequence.

The last wff in the sequence is called a **theorem**.

# Inference Rules

Modus Ponens (MP): Given:

- $A$
- $A \to B$

... we can infer $B$.

Notation:

$$\frac{A \to B, A}{\therefore B}$$

Rule $R$ preserves truth if, given tautologies in our premises, the result is a tautology.

Equivalently, $R(P_1, ..., P_k) = C$ preserves truth if the following is a tautology:

$$P_1 \wedge ... \wedge P_k \to C.$$

What about Modus Ponens?

# Other Inference Rules

*Modus tollens (MT)* $\frac{A \rightarrow B, \neg B}{\therefore \neg A}$

*Conjunction* $\frac{A, B}{\therefore A \ \wedge \ B}$

*Simplification* $\frac{A \ \wedge \ B}{\therefore A}$

(Do we need one for B?)

*Disjunctive syllogism* $\frac{A \ \vee \ B, \neg A}{\therefore B}$

*Hypothetical syllogism* $\frac{A \rightarrow B, B \rightarrow C}{\therefore A \rightarrow C}$

# Formal Theories

A formal reasoning system is also called a **formal theory**.

If a formal theory allows both $W$ and $\neg W$ to be proved as theorems, it is **inconsistent**.

Formal theories without this behavior are **consistent**.

In propositional calculus, a consistent theory is obtained if we use only tautologies as axioms and inference rules that map tautologies to tautologies.

# Conditional Proof

Want to prove

$$A_1 \wedge A_2 \wedge ... \wedge A_n \rightarrow B.$$

Procedure: Treat $A_1$, $A_2$, ..., $A_n$ as axioms and construct a proof of $B$.

Prove: $(A \vee B) \wedge (A \vee C) \wedge \neg A \rightarrow B \wedge C$.

| | | |
|---|---|---|
| 1. | $A \vee B$ | $P$ |
| 2. | $A \vee C$ | $P$ |
| 3. | $\neg A$ | $P$ |
| 4. | $B$ | $1, 3, DS$ |
| 5. | $C$ | $2, 3, DS$ |
| 6. | $B \wedge C$ | $4, 5, Conj$ |
| | $QED$ | $1, 2, 3, 6, CP.$ |

# Subproofs

Prove:

$$((A \vee B) \rightarrow (B \wedge C)) \rightarrow (B \rightarrow C) \vee D.$$

The conclusion contains a conditional.

| | | |
|---|---|---|
| 1. | $(A \vee B) \rightarrow (B \wedge C)$ | $P$ |
| 2. | $\quad B$ | $P$ |
| 3. | $\quad A \vee B$ | $2, Add$ |
| 4. | $\quad B \wedge C$ | $1, 3, MP$ |
| 5. | $\quad C$ | $4, Simp$ |
| 6. | $B \rightarrow C$ | $2, 5, CP$ |
| 7. | $(B \rightarrow C) \vee D$ | $6, Add$ |
| | $QED$ | $1, 7, CP$ |

WARNING: You may not use lines in a subproof to infer a line later!

# Proof of Contrapositive

Notice that

$$A \to B \equiv \neg B \to \neg A.$$

So, instead of proving $A \to B$, we can try to prove $\neg B \to \neg A$.

# Indirect Proof

An Indirect Proof is similar to a proof by contradiction.

$$A \to B \equiv A \wedge \neg B \to \text{ false.}$$

Now we can use both $A$ and $\neg B$ as premises in an attempt to derive "FALSE".

Prove

$$(W \vee S) \wedge (W \to M) \wedge (S \to B) \wedge \neg B \to M.$$

| | | |
|---|---|---|
| 1. | $W \vee S$ | $P$ |
| 2. | $W \to M$ | $P$ |
| 3. | $S \to B$ | $P$ |
| 4. | $\neg B$ | $P$ |
| 5. | $\neg M$ | $PforIP$ |
| 6. | $\neg W$ | $2, 5, MT$ |
| 7. | $\neg S$ | $3, 4, MT$ |
| 8. | $\neg W \wedge \neg S$ | $6, 7, Conj$ |
| 9. | $\neg(W \vee S)$ | $8, T$ |
| 10. | $(W \vee S) \wedge \neg(W \vee S)$ | $1, 9, Conj$ |
| 11. | $FALSE$ | $10, T$ |
| | $QED$ | $1-4, 5, 11, IP.$ |

# Reasoning Systems

So far, we allowed all of the stated inference rules and all known tautologies.

Do we need to be so loose?

Goals:

- Minimize sets of axioms and inference rules.
- **Soundness**: Want all proofs to yield tautologies.
- **Completeness**: Want all tautologies to be provable as theorems.

What if we permit the following inference rule?

$$\frac{A \vee B}{\therefore A}$$

# Hilbert's System

Here is one system of axioms and proof rules that is sound and complete.

Inference Rules:

1. modus ponens
2. conditional proof rule

Axioms:

1. $A \vee A \rightarrow A$
2. $A \rightarrow A \vee B$
3. $A \vee B \rightarrow B \vee A$
4. $(A \rightarrow B) \rightarrow (C \vee A \rightarrow C \vee B)$
5. $A \rightarrow B \equiv \neg A \vee B \equiv \neg(A \wedge \neg B)$

Small systems are easier to test, like small instruction sets.

Can we rebuild a "fuller" set of axioms?

# Theorem 1

Prove $(A \to B) \wedge (B \to C) \to (A \to C)$.

| | | |
|---|---|---|
| 1. | $A \to B$ | $P$ |
| 2. | $B \to C$ | $P$ |
| 3. | $A$ | $P$ |
| 4. | $B$ | $1, 3, MP$ |
| 5. | $C$ | $2, 4, MP$ |
| 6. | $A \to C$ | $3, 5, CP$ |
| | $QED$ | $1, 2, 6, CP$ |

# Theorem 2

Prove $A \to A$.

1. $A \to A \vee A$ $Axiom2$
2. $A \vee A \to A$ $Axiom1$
3. $A \to A$ $1, 2, Theorem1$

$QED$

# Limits to Propositional Calculus

Many forms of everyday arguments cannot be expressed by Propositional Calculus.

Example:

- All X have property Y.
- Individual S does not have property Y.
- Therefore, S is not in X.

X needs to take a non-boolean value.

"Has property Y" is a **predicate**: A boolean relation.

Write $p(x)$ for "$x$ has property $p$."

$q(x, y)$ could mean "$x < y$" for integers $x$ and $y$.

# Quantifiers

Let $D = \{2, 3, 4, 5\}$.

$$p(2) \vee p(3) \vee p(4) \vee p(5).$$

"There exists an $x$ in $D$ that has property $p$."

Write $\exists x \in D : p(x)$.

For all integers $x$ (rather than set $D$), write

$$\exists x \ p(x).$$

Existential Quantifier.

# Quantifiers (cont.)

$$p(1) \wedge p(3) \wedge p(5) \wedge p(7)$$

For $D = \{1, 3, 5, 7\}$,

$$\forall x \in D : p(x).$$

Universal quantifier.

$$[p(0,0) \vee p(0,1)] \wedge [p(1,0) \vee p(1,1)].$$

$$\forall x \in D : \exists y \in D : p(x, y).$$

$$\exists y \in D : \forall x \in D : p(x, y).$$

$$[p(0,0) \wedge p(0,1)] \vee [p(1,0) \wedge p(1,1)].$$

# Examples

Every natural number has a successor.

$$\forall x \in \mathbb{N} : \exists y \in \mathbb{N} : s(x, y).$$

There is no natural number whose successor is 0.

$$\neg \exists x \in \mathbb{N} : s(x, 0).$$

**First-order** predicated calculus: quantifiers can quantify only variables that occur in predicates.

In "higher order" logics, quantifiers can quantify additional things.

# Well Formed Formulas

- Variables: $x$, $y$, $z$.
- Constants: $a$, $b$, $c$.
- Functions: $f$, $g$, $h$.
- Predicates: $p$, $q$, $r$.
- Connectives: $\neg$, $\rightarrow$, $\wedge$ , $\vee$ .
- Quantifiers: $\exists$, $\forall$.
- Punctuation: (, ).

Variables, constants, functions and predicates may be subscripted.

A predicate with no arguments is a proposition.

**Term**: variable, constant, or function applied to arguments.

**Atom**: a predicate applied to arguments that are terms.

A wff is:

- An atom.
- If $W$ and $V$ are wffs, and $x$ is a variable, then the following are wffs:

  $(W), \neg W, W \vee V, W \wedge V, W \rightarrow V, \exists x\ W, \forall x\ W.$

# Quantifier Scope

The **scope** of a quantifier is the wff it precedes.

$$\exists x \ p(x, y) \rightarrow q(x).$$

Here, $\exists x$ influences $p(x, y)$, NOT $q(x)$.

$$\exists x \ (p(x, y) \rightarrow q(x)).$$

Here, $\exists x$ influences $p(x, y) \rightarrow q(x)$.

A variable $x$ in a wff is **bound** if it lies within the scope of a quantifier; otherwise, it is **free**.

$$\exists x \ p(x, y) \rightarrow q(x).$$

The first two occurrences of $x$ are bound, the third is free, $y$ is free.

# WFF Semantics

For a wff to have "meaning," we must give an interpretation to its symbols.

- The interpretation must allow the wff to evaluate to TRUE or FALSE.

Example:

- $p(x)$ means "$x$ is an even integer."
- $x$ means 236.
- Now the wff can be evaluated.

$$\forall x \exists y \ s(x, y).$$

- Let $s(x, y)$ mean "the successor of $x$ is $y$."
- Let $x$ and $y$ be natural numbers.

# Interpretation

An **interpretation** for a wff consists of:

- Set $D$ called the domain of the wff,
- An assignment of symbols of the wff to symbols in $D$.
- Each predicate assigned to a relation over $D$.
- A predicate with no arguments is a constant with a truth value.
- Each function letter assigned a function over $D$.
- Each free variable assigned a value in $D$.
- Each constant assigned a value in $D$.

$W(x/t)$ is the wff obtained by replacing $x$ with $t$.

For free variable $x$,

- $W(x) = p(x) \vee \exists x \; q(x, y)$.
- $W(x/a) = p(a) \vee \exists x \; q(x, y)$.

# Meaning of a WFF

If wff W contains quantifiers, evaluate as:

- $\exists x\ W$ is true if $W(x/d)$ is true for some $d \in D$.
- Otherwise, $\exists x\ W$ is FALSE.
- $\forall x\ W$ is true if $W(x/d)$ is true for every $d \in D$.
- Otherwise, $\forall x\ W$ is FALSE.

Examples:

$$W = \exists x \forall y\ (p(y) \rightarrow q(x, y)).$$

Let $q(x, y)$ mean "$x = y$."

- $D = \{a\}$, $p(a) =$ true.
- $D = \{a\}$, $p(a) =$ false.
- $D = \{a, b\}$, $p(a) = p(b) =$ true.
- What if $p(d)$ is true for at most one element $d \in D$?

# Example

$$W = \forall x \ (p(f(x,x),x) \rightarrow p(x,y)).$$

- $D = \mathbb{N}$, $p$ is equality, $y = 0$, $f(a,b) = (a+b) \bmod 3$.

$$\forall x \ ((2x \bmod 3 = x) \rightarrow x = 0).$$

- $D = \{a,b\}$, $f(a,a) = a$, $f(b,b) = b$, $p$ is equality, $y = a$.

An interpretation for wff $W$ is a **model** if it makes $W$ true.

# Validity

A wff is **valid** if it is true for every interpretation.

- Otherwise, it is **invalid**.

A wff is **satisfiable** if there is some interpretation that makes it true.

- Otherwise, it is **unsatisfiable**.

Possibilities:

- Valid and satisfiable: tautology.
- satisfiable and invalid: contingency.
- unsatisfiable and invalid: contradiction.

How can we tell if a wff is a tautology?

- Truth tables?

# Methods of Proof

Indirect:

- Assume W is invalid, try to obtain a contradiction.
- For $A \rightarrow B$, show that you can't have $A = T$ while $B = F$.

Direct:

- For $A \rightarrow B$, assume an arbitrary interpretation is a model for $A$.
- Now, prove this interpretation is a model for $B$.

# Proof Example

$$W = \exists y \forall x \ p(x, y) \rightarrow \forall x \exists y \ p(x, y).$$

Direct proof:

- Let $M$ be an interpretation (with domain D) for $W$ such that $M$ is a model for $A$.
- There is an element $d \in D$ such that $\forall x \ p(x, d)$ is true.
- Therefore $p(e, d)$ is true for all $e \in D$.
- Thus, $M$ is a model for $B$.
- Therefore, $W$ is valid for all $M$.

Indirect proof:

- Assume $W$ is invalid.
- Then it has countermodel with domain $D$ such that $A$ is true and $B$ is false.
- There is $d \in D$ such that $\exists y \ p(d, y)$ is false.
- Thus, $p(d, e)$ is false for all $e \in D$.
- Since $A$ is true, there is a $c \in D$ such that $\forall x \ p(x, c)$ is true.
- So, $p(d, c)$ is true.
- This contradicts $p(d, e)$ is false for all $d \in D$.
- Therefore, $W$ is valid.

# Closures

**Universal closure**: Universally quantify all free variables.

**Existential closure**: Existentially quantify all free variables.

Closure properties:

1. A wff is valid iff its universal closure is valid.
2. A wff is unsatisfiable iff its existential closure is unsatisfiable.

# Validity Problem

A **decision problem** has a yes or no answer.

A decision problem is **decidable** if there is an algorithm that halts with an answer to the problem.

- Otherwise, the problem is **undecidable.**

A decision problem is **partially decidable** if there is an algorithm that halt with "yes" iff and only iff the answer is yes.

- It may not halt if the answer is "no."

How do we decide if a wff is valid?

Propositional calculus: check the truth table.

For first-order predicate calculus, validity is partially decidable.

- natural deduction.
- resolution.

# Equivalence

Two wffs are equivalent if they both have the same truth value with respect to every interpretation of both.

- free variables, constants, predicates interpreted with respect to same domain.
- In general, can't use truth tables to determine equivalence.

Alternatively: $(A \to B) \wedge (B \to A)$ must be valid.

A wff $W$ is an instance of propositional wff $V$ if $W$ is obtained from $V$ by replacing each propositional letter of $V$ by a wff where all occurrences of each propositional letter in $V$ are replaced by the same wff.

- $\forall x\ p(x) \to \forall x\ p(x) \vee q(x)$.
- This is an instance of $P \to P \vee Q$.

# Two equivalences

$$\neg(\forall x\ W) \equiv \exists x\ \neg W$$

Proof:

$I$ is a model for $\neg(\forall x\ W)$
iff $\neg(\forall x\ W)$ is true for $I$
iff $\forall x\ W$ is false for $I$
iff $W(x/d)$ is false for some $d \in D$
iff $\neg W(x/d)$ is true for some $d \in D$
iff $\exists x\ \neg W$ is true for $I$
iff $I$ is a model for $\exists x\ \neg W$.

$$\neg(\exists x\ W) \equiv \forall x\ \neg W$$

Proof: Since $W$ is arbitrary, replace $W$ by $\neg W$:

$$\neg(\forall x\ \neg W) \equiv \exists x\ \neg\neg W$$

Now, negate both sides and simplify.

# More equivalences

$$\forall x \forall y \ W \equiv \forall y \forall x \ W$$

$$\exists x \exists y \ W \equiv \exists y \exists x \ W$$

Renaming: If $y$ does not occur in $W(x)$, then the following equivalences hold:

- $\exists x \ W(x) \equiv \exists y \ W(y)$
- $\forall x \ W(x) \equiv \forall y \ W(y)$

# Normal Forms

A wff is in **Prenix normal form** if all its quantifiers are on the left of the expression.

$$\forall x \exists y \forall z \ (p(x) \vee q(y) \wedge r(x, z)).$$

Algorithm:

1. Rename the variables so that no quantifiers use the same variable name, and quantified variable names are distinct from free variable names.
2. Use known equivalences to shift quantifiers to the left.

Prenix conjunctive normal form and prenix disjunctive normal form.

# Inference Rules

We need some inference rules to do proofs.

Theorem: Modus Ponens maps valid wffs to a valid wff.

Proof:

- Let $A$ and $A \rightarrow B$ be valid wffs.
- Select an interpretation for $B$ with domain $D$.
- Generate an interpretation for $A$ in $D$.
- $A$ and $A \rightarrow B$ must be true under the interpretation.
- We can apply MP for propositions to this interpretation.
- Since the interpretation was arbitrary, it must apply for all interpretations.
- Thus, B is valid.

Similar arguments hold for all propositional inference rules.

# Inference Rules for Quantifiers

How do we deal with the quantifiers?

Can we infer $\exists y\ p(y, y)$ from $\forall x \exists y\ p(x, y)$?

When does this go wrong?

A term $t$ is **free to replace** $x$ in $W(x)$ if both $W(t)$ and $W(x)$ have the same bound occurrences of variables.

Universal Instantiation Rule (UI):

$$\frac{\forall x W(x)}{\therefore W(t)}$$ if $t$ is free to replace $x$ in $W(x)$.

# Existential Instantiation

What can we infer from $\exists x\ W(x)$?

What about $W(c)$?

$$\exists x\ p(x) \wedge \exists x\ q(x) \rightarrow p(c) \wedge q(c).$$

What is the problem here?

Existential Quantification Rule (EI):

$\dfrac{\exists x\ W(x)}{\therefore W(c)}$ if $c$ is a new constant in the proof.

# Universal Generalization

Consider the following argument:

- Let $x$ be an arbitrary, but fixed, element of $D$.
- Prove $W(x)$ is true.
- Since $x$ is arbitrary, $W(x)$ is true for all $x \in D$.

Can we infer $\forall x\ p(x)$ from $p(x)$?

Obviously not, in general.

What are appropriate restrictions?

A variable $x$ in a wff $W$ is a **flagged** variable in $W$ if $x$ is free in $W$ and either $W$ is a premise or $W$ is inferred by a wff containing $x$ as a flagged variable.

| | | | |
|---|---|---|---|
| 1. | $p(x)$ | $P$ | $x$ is flagged |
| 2. | $\forall x\ q(x)$ | $P$ | |
| 3. | $q(x)$ | $2, UI$ | |
| 4. | $p(x) \wedge q(x)$ | $1, 3, Conj$ | $x$ is flagged |

Do not infer $\forall x\ W(x)$ from $W(x)$ if $x$ is a flagged variable.

# Universal Generalization (cont.)

| | | |
|---|---|---|
| 1. | $\forall x \exists y \ x < y$ | $P$ |
| 2. | $\exists y \ x < y$ | $1, UI$ |
| 3. | $x < c$ | $2, EI$ |
| 4. | $\forall x \ x < c$ | 3, proposed $UG$ |

Variable $x$ is **subscripted** if $x$ is free in $W$ and there is a constant $c$ in $W$ created by EI, with $c$ and $x$ in the same predicate.

Do not infer $\forall x \ W(x)$ from $W(x)$ if $x$ is a subscripted variable.

Universal Generalization (UG):

$$\frac{W(x)}{\therefore \forall x \ W(x)}$$

... if $x$ is not flagged and $x$ is not subscripted.

# A Proof from Lewis Carroll

- Babies are illogical.
- Nobody is despised who can manage a crocodile.
- Illogical persons are despised.
- Therefore, babies cannot manage crocodiles.

Proof:

| | | | |
|---|---|---|---|
| 1. | $\forall x(B(x) \rightarrow \neg L(x))$ | $P$ | |
| 2. | $\forall x(C(x) \rightarrow \neg D(x))$ | $P$ | |
| 3. | $\forall x(\neg L(x) \rightarrow D(x))$ | $P$ | |
| 4. | $B(x) \rightarrow \neg L(x)$ | $1, UI$ | |
| 5. | $C(x) \rightarrow \neg D(x)$ | $2, UI$ | |
| 6. | $\neg L(x) \rightarrow D(x)$ | $3, UI$ | |
| 7. | $B(x)$ | $P$ | $x$ is flagged |
| 8. | $\neg L(x)$ | $4, 7, MP$ | $x$ is flagged |
| 9. | $D(x)$ | $6, 8, MP$ | $x$ is flagged |
| 10. | $\neg C(x)$ | $5, 9, MT$ | $x$ is flagged |
| 11. | $B(x) \rightarrow \neg C(x)$ | $7, 10, CP$ | |
| 12. | $\forall x(B(x) \rightarrow \neg C(x))$ | $11, UG$ | |
| | $QED$ | $1, 2, 3, 12, CP.$ | |

# Existential Generalization(EG)

Given that 5 is prime, it makes sense to infer there exists a prime number.

Consider the following:

| | | |
|---|---|---|
| 1. | $\forall x \exists y \ x < y$ | $P$ |
| 2. | $\exists y \ x < y$ | $1, UI$ |
| 3. | $x < c$ | $2, EI$ |
| 4. | $\exists x \ x < x$ | $3, EG$? |
| 5. | $\exists z \ x < z$ | $3, EG$ |
| 6. | $\exists x \ x < c$ | $3, EG$ |

What restrictions do we need?

Existential Generalization Rule (EG):

$$\frac{W(t)}{\therefore \exists x \ W(x)}$$

.. if the following two restrictions hold:

- $W(t) = W(x)(x/t)$.
- $t$ is free to replace $x$ in $W(x)$.

# Proof Example

Prove $\exists x\ W(x) \rightarrow \exists y\ W(y)$.

| | | |
|---|---|---|
| 1. | $\exists x\ W(x)$ | $P$ |
| 2. | $W(c)$ | $1, EI$ |
| 3. | $\exists y\ W(y)$ | $2, EG$ |
| | $QED$ | $1, 3, CP.$ |

# Proof Example

Three statements:

- Every computer scientist is a logical thinker.
- Dr. Shaffer is a computer scientist.
- Therefore, there is some logical thinker.

$$\forall x\ (C(x) \rightarrow L(x)) \wedge C(b) \rightarrow \exists x\ L(x).$$

Proof:

| | | |
|---|---|---|
| 1. | $\forall x\ (C(x) \rightarrow L(x)) \wedge C(b)$ | $P$ |
| 2. | $\forall x\ (C(x) \rightarrow L(x))$ | $1, Simp$ |
| 3. | $C(b)$ | $1, Simp$ |
| 4. | $C(b) \rightarrow L(b)$ | $2, UI$ |
| 5. | $L(b)$ | $3, 4, MP$ |
| 6. | $\exists x\ L(x)$ | $5, EG$ |
| | $QED$ | $1, 6, CP.$ |

# Equality

Syntactic Equality: $x = x$ regardless of domain of discourse.

Semantic Equality:

- $3 + 4 = 2 + 5$.
- If $x + y = 2z$, then $(x + y) + w = 2z + w$.

A first-order theory is called a **first-order theory with equality** if it contains a two-argument predicate that captures the properties of equality: $e(x, y)$.

# Equality Axioms

Equality Axiom (EA):

$$\forall x\ (x = x).$$

What about terms?

| | | |
|---|---|---|
| 1. | $\forall x\ (x = x)$ | $EA$ |
| 2. | $t = t$ | $1, UI$ |

Equals for Equals rule (EE):

$$t = u \wedge p(...t...) \rightarrow p(...u...)$$

Equivalently,

$$\frac{t = u, p(...t...)}{\therefore p(...u...)}.$$

Is this symmetric?

| | | |
|---|---|---|
| 1. | $t = u$ | $P$ |
| 2. | $p(t, t)$ | $EA$ |
| 3. | $p(u, t)$ | $1, 2, EE$ |
| | $QED$ | $1, 3, CP$ |

# Proof Example

Prove

$$\forall x \ (x + x = x \rightarrow x = 0).$$

| | | |
|---|---|---|
| 1. | $x + x = x$ | $P$ |
| 2. | $-x = -x$ | $EA$ |
| 3. | $(x + x) + -x = x + -x$ | $1, 2, EE$ |
| 4. | $x + (x + -x) = (x + x) + -x$ | $Assoc.$ |
| 5. | $x + (x + -x) = x + -x$ | $3, 4, Trans.$ |
| 6. | $x + -x = 0$ | $(Axiom)$ |
| 7. | $x + 0 = 0$ | $5, 6, EE$ |
| 8. | $x = x + 0$ | $(Axiom)$ |
| 9. | $x = 0$ | $7, 8, Transitivity$ |
| 10. | $x + x = x \rightarrow x = 0$ | $1, 9, CP$ |
| 11. | $\forall x \ (x + x = x \rightarrow x = 0)$ | $10, UG$ |
| | $QED.$ | |

What is the big deal about this?

- It means that you can automate the process (Larch).
- Predicate calculus machinery supports arithmetic.

# Program Correctness

Prove that a program is correct.

- What does this mean?
- Warning: In general, this is not computable.

Imperative programming language: a sequence of statements that represent commands.

- Assignment.
- Loops.
- If statements.

What does a program $S$ do?

Call $P$ the relevant state of the world prior to the program: Precondition.

Call $Q$ the relevant state of the world after the program: Postcondition.

$$\{P\}S\{Q\}$$

$P$ and $Q$ are wffs from an appropriate first-order theory.

# Meaning of a Program

$\{P\}S\{Q\}$ is also a wff.

The meaning of $\{P\}S\{Q\}$ is the truth value of:

- If $P$ is true before $S$ is executed, and the execution of $S$ halts, then $Q$ is true after $S$ halts.

If this is true, we say that $S$ is correct with respect to $P$ and $Q$.

We generally agree this is true:

$$\{x > 4\}x := x + 1\{x > 5\}$$

We generally agree that this is false:

$$\{x > 4\}x := x + 1\{x > 6\}$$

# A Programming Language

We need axioms and inference rules, but first a language.

Assignment: For $x$ a variable and $t$ a term,

$$x := t.$$

Assignment Axiom (AA):

$$\{Q(x/t)\}x := t\{Q\}.$$

"Backwards" assignment axiom: Construct the precondition from the postcondition.

$$\{\}x := x + 1\{x < 5\}$$

yields

$$\{x + 1 < 5\}x := x + 1\{x < 5\}.$$

What about

$$\{x < 3\}x := x + 1\{x < 5\}?$$

# Consequence Rule

$$\frac{P \to R \text{ and } \{R\}S\{Q\}}{\therefore \{P\}S\{Q\}} \text{ and } \frac{\{P\}S\{T\} \text{ and } T \to Q}{\therefore \{P\}S\{Q\}}.$$

This requires both proof of correctness and proof of implication.

Example:

$$\{x < 3\}x := x + 1\{x < 5\}$$

requires a proof that

$$x < 3 \to x + 1 < 5.$$

# Composition

We need to support sequences of statements.

Composition rule:

$$\frac{\{P\}S_1\{R\} \text{ and } \{R\}S_2\{Q\}}{\therefore \{P\}S_1; S_2\{Q\}}.$$

Example: Prove

$$\{x > 2 \wedge y > 3\}x := x + 1; y := y + x\{y > 6\}.$$

Do AA on second statement:

$$\{y + x > 6\}y := y + x\{y > 6\}$$

Now, go through the first statement:

$$\{y + x + 1 > 6\}x := x + 1\{y + x > 6\}$$

Use the composition rule to put them together.

Now, we just need

$$\{x > 2 \wedge y > 3\} \rightarrow \{y + x + 1 > 6\}$$

# If-Then Statements

If-then Rule:

$$\frac{\{P \wedge C\}S\{Q\} \text{ and } P \wedge \neg C \rightarrow Q}{\therefore \{P\} \textbf{ if } C \textbf{ then } S\{\mathrm{Q}\}}.$$

Example: Prove

$$\{\text{true}\}\textbf{if } x < 0 \textbf{ then } x := -x \ \{x \geq 0\}$$

This fits the pattern.

Need to prove:

- true $\wedge\ x < 0\}x := -x\{x \geq 0\}$.
- true $\wedge\ \neg(x < 0) \rightarrow x \geq 0$.

# Proof

| | | |
|---|---|---|
| 1. | $\{-x \geq 0\}x := -x\{x \geq 0\}$ | $AA$ |
| 2. | true $\wedge\ x < 0$ | $P$ |
| 3. | $x < 0$ | $2, Simp$ |
| 4. | $-x > 0$ | $3, T$ |
| 5. | $-x \geq 0$ | $4, Add$ |
| 6. | true $\wedge\ x < 0 \rightarrow -x \geq 0$ | $2, 5, CP$ |
| 7. | $\{$true $x < 0\}x := -x\{x \geq 0\}$ | $1, 6, Consequence$ |
| 8. | true $\wedge\ \neg(x < 0)$ | $P$ |
| 9. | $\neg(x < 0)$ | $8, Simp$ |
| 10. | $x \geq 0$ | $9, T$ |
| 11. | true $\wedge\ \neg(x < 0) \rightarrow x \geq 0$ | $8, 10, CP$ |
| | $QED$ | $7, 11, If-then$ |

# If-Then-Else Rule

If-then-else Rule:

$$\frac{\{P \wedge C\}S_1\{Q\} \text{ and } \{P \wedge \neg C\}S_2\{Q\}}{\therefore \{P\} \textbf{ if } C \textbf{ then } S_1 \textbf{ else } S_2\{\mathrm{Q}\}}.$$

Prove:

$$\{\text{true}\}\textbf{if } \text{even}(x) \quad \textbf{then} \quad y := x$$
$$\textbf{else} \quad y := x + 1\{\text{even}(y)\}$$

Must show that:

- $\{\text{true} \quad \wedge \text{even}(x)\}y := x\{\text{even}(y)\}$.
- $\{\text{true} \quad \wedge \text{odd}(x)\}y := x + 1\{\text{even}(y)\}$.

$$\text{even}(x) = \exists k \ (x = 2k)$$

$$\text{odd}(x) = \exists k \ (x = 2k + 1)$$

# While Rule

While:

$$\frac{\{P \wedge C\} S \{P\}}{\therefore \{P\} \textbf{ while } C \textbf{ do } S \{P \wedge \neg C\}}.$$

$P$ is called the **loop invariant**.

# While Example

$\{a > 0 \wedge b \geq 0\}$

```
  i := 0;
  p := 1;
  while i < b do
    p := p * a;
    i := i + 1;
  od
```

$\{p = a^b\}$

Want to prove:

$\{a > 0 \wedge b \geq 0\}$ `i := 0; p := 1;` $\{P\}$,

$\{P\}W\{p = a^b\}$.

What should we use for $P$?

What is the loop invariant?

# Loop Invariant

$\{P\}$

```
while i < b do
  p := p*a; i := i+1;
od
```

$\{P \wedge \neg(i < b)\}$.

Try 1: Set $[P \wedge \neg(i < b)] = [p = a^b]$ and solve for $P$.

Try 2: Want $P$ such that
$(P \wedge \neg(i < b)) \rightarrow (p = a^b)$.

- Note that $p = a^i$ holds before and after loop.
- Note that $i \leq b$ holds before loop.
- Try this for $P$:

$$(p = a^i) \wedge (i \leq b).$$

Does this fit the original precondition of the program?

What else do we need to prove?

# Completing the Proof

Final step: Prove

$$\{P \wedge (i < b)\}\texttt{p := p*a; i := i+1;}\{P\}.$$

The full story:

$\{(a > 0) \wedge (b \geq 0)\}$

```
  i := 0;
  p := 1;
```

$\{P\} = \{(p = a^i) \wedge (i \leq b)\}$

```
  while i < b do
    p := p * a;
    i := i + 1;
  od
```

$\{P \wedge \neg C\} = \{(p = a^i) \wedge (i \leq b) \wedge \neg(i < b)\}$
$\{p = a^b\}$

# Proving Termination

Our while loop axiom is adequate only to show **Partial Correctness**.

- It does not prove that the loop terminates – only the conditions if it does terminate.

**Total Correctness** includes proof of termination.

When does the following loop terminate?

```
x := a;
y := b;
while x != y do
  x := x - 1;
  y := y + 1;
  c := c + 1;
od;
```

We need a precondition that is strong enough to guarantee termination.

# Program State

A program **state** is a tuple representing the values of all variables at some instant in the computation.

- Define state of previous program by $< x, y, c >$.

Assignment changes the state.

What happens to the program for state $< 10, 6, 0 >$?

Termination occurs when program action on the initial state forms a decreasing sequence on a well-founded set.

- For this program, the set is $\mathbb{N}$, the value to decrease is $x - y$.

# Termination Condition

For precondition $P$ of loop **while** $C$ **do** $S$:

- $P(s)$ denotes the wff obtained from $P$ by replacing its variables by their corresponding values in $s$.
- $C(s)$ denotes the wff obtained from $C$ by replacing its variables by their corresponding values in $s$.

Let $s$ represent state prior to execution of $S$; let $t$ be state after execution of $S$.

Let $< W, \prec >$ be a well-founded set with (partial) function

$$f : \text{States} \rightarrow W.$$

Termination Condition:

$$P(s) \wedge C(s) \wedge f(s) \in W \rightarrow f(t) \in W \wedge f(s) \prec f(t).$$

# Termination Example

$$P = (x \geq y) \wedge \text{even}(x - y).$$

$$f(< x, y, c >) = x - y.$$

If $s =< x, y, c >$ is state prior to executing loop body, then $t =< x - 1, y + 1, c + 1 >$ is state after.

$$\begin{array}{rcl} f(s) & = & f(x, y, c) = x - y. \\ f(t) & = & f(x - 1, y + 1, c + 1) \\ & = & (x - 1) - (y + 1) = x - y - 2. \end{array}$$

Termination Condition is:

$$\begin{array}{l} (x \geq y) \wedge \ \text{even}(x - y) \wedge (x \neq y) \wedge \\ (x - y) \in \mathbb{N} \rightarrow (x - y - 2) \in \mathbb{N} \wedge \\ (x - y) > (x - y - 2) \end{array}$$

# Higher Order Logic

First-order predicate calculus allows quantification of variables, and terms as the argument of a predicate.

- A term is a constant, variable, or function expression with terms as variables.

Can this statement be written in first-order logic?

- There is a function that grows faster than the log function.

$$\exists f \forall x \ (f(x) > \log x).$$

This is a second-order wff.

$$\exists f \forall x \ p(f(x), g(x)).$$

Equality: $x$ and $y$ are the same if all of their properties are the same.

$$\forall P \ ((P(x) \rightarrow P(y)) \wedge (P(y) \rightarrow P(x))).$$

# Classifying Logics

Predicates are sets.

- If $P$ is a predicate of one argument, think of $P$ as a set in which $x \in P$ iff $P(x)$ is true.
- If $P$ is a predicate of $n$ arguments, the set consists of $n$-tuples.

$$\forall x \ (A(x) \rightarrow B(x)).$$

New interpretation: If $x \in A$ then $x \in B$.

- $A$ is a subset of $B$.

A logic is **higher order** if it allows sets to be quantified or if it allows sets to be elements of other sets.

$\exists S \ S(t)$. Set $S$ is quantified.
$S(x) \wedge T(S)$. Set $S$ is an element of set $T$.

# Functions

Functions can also be viewed as a set of tuples (input and output).

A function is a predicate of two variables (input and output).

Higher order wffs can quantify functions.

$\exists f \forall x \ p(f(x), g(x))$. Function (set) $f$ is quantified.

$p(f(s)) \wedge q(f)$. Function (set) $f$ is an element of set $q$.

# Classification

Propositional Calculus is **zero-order** logic.

Predicate Calculus is **first-order** logic.

Order:

- A predicate has order 1 if all its arguments are terms.
- A predicate has order $n + 1$ where $n$ is the highest order among its arguments that are not terms.
- A quantifier has order 1 if it quantifies a variable.
- A quantifier has order $n + 1$ if it quantifies a predicate of order $n$.

$$\forall x \exists S \exists T \exists f \ (S(x, f(x)) \wedge T(S)).$$

# Semantics

We attach meaning to a higher-order wff by constructing an interpretation.

Need to start with a domain $D$.

$$\exists S \exists T \forall x \ (S(x) \rightarrow \neg T(x)).$$

Let $D = \{a, b\}$.

$S$ and $T$ can be any subsets of $D$.

- Example: $S = \{a, b\}$, $T = \emptyset$.
- Thus, the wff is true for this interpretation.

# Gödel's Theorem

If a formal system is powerful enough to describe all arithmetic formulas of the natural numbers, and the system is consistent, then it is not complete.

The formulas of arithmetic can be described in a first-order logic with equality.

- So, first-order logic with equality is not complete.

The formulas of arithmetic can be described in a second-order logic.

- So, second-order logic with equality is not complete.

What does this mean?

- We must leave the formalism to prove certain tautologies in the system.
- This limits what can be automated.

# Family Tree Problem

Given a set of Parent-Child relationships, find answers to questions like "Is $x$ a second cousin of $y$?"

Define functions:

- isGrandparentOf($x$, $y$)
- isGrandchildOf($x$, $y$)
- isSiblingOf($x$, $y$)

Define $p(x, y)$ to mean "$x$ is the parent of $y$."

Define $g(x, y)$ to mean "$x$ is the grandparent of $y$" as follows:

$$g(x, y) \text{ if } p(x, z) \text{ and } p(z, y).$$

Conceptually easy to compute, but what if there are thousands of parent relationships in the database?

# Logic Queries

Find all pairs $x$, $y$ such that $x$ is a grandparent of $y$.

Logic programming style: $\leftarrow g(x, y)$.

- This is a goal.
- The program answers yes if it can satisfy the goal (list all grandparent relationships).

Specific goal: $\leftarrow g(\text{Jack}, \text{Mary})$.

- This is true iff relations $p(\text{Jack}, x)$ and $p(x, \text{Mary})$ are in the database.

Fact: Write $p(\text{Jack}, \text{Fred}) \leftarrow$.

Conditional: If $A$ then $B$.

- Write: $B \leftarrow A$.

Form of general grandparent function:

$$g(x, y) \leftarrow p(x, z), p(z, y).$$

# Computing Grandparent

How does the program solve the query $\leftarrow g(v, w)$?

It works by pattern matching.

First step: Match the atom $g(v, w)$ to a left hand side.

- This matches the left hand side of

$$g(x, y) \leftarrow p(x, z), p(z, y).$$

Now we need to prove the antecedent.

This is done by pattern matching the first rule to the fact database.

- The result is a binding for $z$ in the second rule, and a check for a matching fact.

Repeat until exhaustion.

# Automated Reasoning

Goal: Find inference rule(s) that can be applied automatically.

**Literal**: An atom or negation of an atom.

**Clause**: A disjunction of zero or more literals.

- The empty clause is assumed to have value FALSE.

**Clausal Form**: The universal closure of a conjunction of clauses.

Prenex conjunctive normal form:

- All quantifiers are universal.
- No free variables.

Notation: represent as a set of clauses.

- $S = \{C_1, ..., C_n\}$ represents

$$\forall x_1 \cdots \forall x_m (C_1 \wedge \cdots \wedge C_n).$$

# Examples

| | |
|---|---|
| $\forall x\ p(x)$ | $\{p(x)\}$ |
| $\forall x\ \neg q(x,b)$ | $\{\neg q(x,b)\}$ |
| $\forall x \forall y\ (p(x) \wedge \neg q(y,b))$ | $\{p(x), \neg q(y,b)\}$ |
| $\forall x \forall y\ (p(y,f(x)) \wedge (q(y) \vee \neg q(a)))$ | $\{p(y,f(x)), q(y) \vee \neg q(a)\}$ |
| $(p(a) \vee p(b)) \wedge q(a,b)$ | $\{p(a) \vee p(b), q(a,b)\}$ |

The last is a proposition.

Since a set of clauses represents a wff, we can apply the terms "valid", "satisfiable."

# Skolemization

Any wff can be turned into prenix conjunctive form.

An existentially quantified variable outside of scope for all universal quantifiers can be replaced with a constant.

$$\forall x \exists y \; p(x, y).$$

Here, $y$ may depend on $x$.

- Consider successor function.

Solution: Replace $y$ with $f(x)$.

All wffs can be converted to clauses.

# Resolution (for Propositions)

Consider the pair of clauses $p \vee q$ and $\neg p \vee r$.

This pair is satisfiable iff $q \vee r$ is satisfiable.

Resolution:

- Take a pair of clauses $C_1$ with literal $A$ and $C_2$ with its negation $\neg A$.
- Remove all occurrences of $A$ from $C_1$ and $\neg A$ from $C_2$.
- Or together the remaining literals.

Resolution proof: To prove a wff is unsatisfiable, use resolution to generate the empty clause.

Prove a wff is valid by proving its negation is invalid.

This can be automated (try all pairs of clauses).

# Resolution for Predicate Calculus

We need some extra rules.

Substitution: Replace variable $y$ with constant $b$ to resolve the following pair:

- IN: $\{p(x,y) \vee q(y), r(z) \vee \neg q(b)\}$.
- Conversion: $\{p(x,b) \vee q(b), r(z) \vee \neg q(b)\}$.
- Result: $p(x,b) \vee r(z)$.

Example: Resolve

$$p(x,a) \vee \neg q(x)$$
$$\neg p(b,y) \vee \neg q(a)$$

Substitute to get:

$$p(b,a) \vee \neg q(b)$$
$$\neg p(b,a) \vee \neg q(a)$$

Result: $\neg q(b) \vee \neg q(a)$.

# Resolution Example

Grandparent: $p(x,z) \wedge p(z,y) \rightarrow g(x,y)$.

Given facts: $p(a,b) \wedge p(c,b) \wedge p(b,d) \wedge p(a,e)$.

Prove or disprove $g(a,d)$.

Here is the wff to prove is valid:

$$p(a,b) \wedge p(c,d) \wedge p(b,d) \wedge p(a,e) \wedge$$
$$(p(x,z) \wedge p(z,y) \rightarrow g(x,y)) \rightarrow g(a,d).$$

Negate:

$$p(a,b) \wedge p(c,d) \wedge p(b,d) \wedge p(a,e) \wedge$$
$$(p(x,z) \wedge p(z,y) \rightarrow g(x,y)) \wedge \neg g(a,d).$$

Replace $p(x,z) \wedge p(z,y) \rightarrow g(x,y)$ with
$\neg p(x,z) \vee \neg p(z,y) \vee g(x,y)$.

# Proof

| | | |
|---|---|---|
| 1. | $p(a, b)$ | $P$ |
| 2. | $p(c, b)$ | $P$ |
| 3. | $p(b, d)$ | $P$ |
| 4. | $p(a, e)$ | $P$ |
| 5. | $\neg p(x, z) \vee \neg p(z, y) \vee g(x, y)$ | $P$ |
| 6. | $\neg g(a, d)$ | $P$ |
| 7. | $\neg p(a, z) \vee \neg p(z, d)$ | $5, 6, R, \{x/a, y/d\}$ |
| 8. | $\neg p(b, d)$ | $1, 7, R, \{z/b\}$ |
| 9. | $\square$ | $3, 8, R, \{\}$ |
| | $QED.$ | |

# Logic Programming Notation

Consider a wff of the form:

$$B_1 \wedge \cdots \wedge B_n \rightarrow A.$$

This is often written in logic programming as:

$$A \leftarrow B_1, ..., B_n.$$

Note that this is the same as the wff:

$$A \vee \neg B_1 \vee \cdots \vee \neg B_n.$$

A **Horn Clause** contains at most one positive literal.

# Logic Programming

Fact:

$$A \leftarrow .$$

Goal:

$$\leftarrow B_1, ..., B_n.$$

Given the following:

$$\begin{array}{lll} q(a) & \leftarrow & \\ r(a) & \leftarrow & \\ p(x) & \leftarrow & q(x), r(x) \end{array}$$

Is $p(a)$ true?

Given:

- $P$ is a logic program.
- $G$ is a goal.

The goal question is answered "yes" iff the set of clauses $P \cup \{G\}$ is unsatisfiable.

Since $P$ and $G$ are in clause form, we use resolution.

# Logic Program Example

Program $P$:

$$\begin{aligned} q(a) &\leftarrow \\ p(f(x)) &\leftarrow q(x). \end{aligned}$$

Goal:

$$\leftarrow p(y).$$

Interpretation: Does $P$ imply $\exists y\ p(y)$?

Equivalent: Is there a substitution $\Theta$ such that $p(y)\Theta$ is inferred from $P$?

Answer: Let $\Theta = \{y/f(a)\}$.

- Then,

$$p(y)\Theta = p(y)\{y/f(a)\} = p(f(a)).$$

# Proof

| | | |
|---|---|---|
| 1. | $q(a)$ | $P$ Clause: $q(a) \leftarrow$ |
| 2. | $p(f(x)) \vee \neg q(x)$ | $P$ Clause: $p(f(x)) \leftarrow q(x)$ |
| 3. | $\neg p(y)$ | $P$ Goal Clause: $\leftarrow p(y)$ |
| 4. | $\neg q(x)$ | $2, 3, R\{y/f(x)\}$ |
| 5. | $\square$ | $1, 4, R\{x/a\}$ |
| | $QED.$ | |

The appropriate value for $y$ is generated by the proof: $f(a)$.

# Use

Advantages of this notation:

- Easy to write down.
- Can interpret $A \leftarrow B_1, ..., B_n$ as an implication, or can interpret as a procedure $A$ executed by executing procedures $B_1, ..., B_n$ in order.
- Can apply resolution rule.

Applying resolution requires a lot of choosing pairs of clauses to resolve.

Want to follow some algorithm to maximize efficiency.

- Example: Pick current line as a goal, and go through program in order looking for a suitable clause.

# SLD Rule

Here is an algorithm for resolution:

1. Pick the goal clause on last line of partial proof, select its leftmost atom $B_1$.
2. Find a program clause whose head unifies with $B_1$ using substitution $\Theta$. (Rename variables to keep distinct).
3. Replace $B_1$ in goal clause with body of the program clause.
4. Apply $\Theta$ to new goal clause to get resolvant, to generate next line of proof.

# Example

Program: ($p$ means isParentof, $g$ means isGrandParentOf)

$$\begin{aligned} p(a,b) &\leftarrow \\ p(d,b) &\leftarrow \\ p(b,c) &\leftarrow \\ g(x,y) &\leftarrow p(x,z), p(z,y). \end{aligned}$$

Goal:

$$\leftarrow g(w,c).$$

Proof:

| | | | |
|---|---|---|---|
| 1. | $p(a,b) \leftarrow$ | $P$ | |
| 2. | $p(d,b) \leftarrow$ | $P$ | |
| 3. | $p(b,c) \leftarrow$ | $P$ | |
| 4. | $g(x,y) \leftarrow p(x,z), p(z,y)$ | $P$ | |
| 5. | $\leftarrow g(w,c)$ | $P$ | Initial Goal |
| 6. | $\leftarrow p(x,z), p(z,c)$ | $4,5,R,\Theta_1 = \{w/x, y/c\}$ | |
| 7. | $\leftarrow p(b,c)$ | $1,6,R,\Theta_2 = \{x/a, z/b\}$ | |
| 8. | $\square$ | $3,7,R,\Theta_3 = \{\}$ | |
| | $QED.$ | | |

$w\Theta_1\Theta_2\Theta_3 = a.$

Note that there is another answer: $w = d$.

# Computation Tree

- The goal is the root.
- The children of a node correspond to any resolvants obtained by resolving the node with a program clause.
- Children are ordered from left to right to match order of program clauses.
- Leaves marked with empty clause represent success.
- Leaves marked with a clause indicate no further resolution is possible.

A logic program must implicitly search at least part of the tree.

- Prolog searches depth first.

# Examples

Program 1:

$$\begin{aligned} p(a) &\leftarrow \\ p(succ(x)) &\leftarrow p(x). \end{aligned}$$

Goal:

$$\leftarrow p(x)$$

Program 2:

$$\begin{aligned} p(f(x)) &\leftarrow p(x) \\ p(a) &\leftarrow \\ p(b) &\leftarrow \end{aligned}$$

Goal:

$$\leftarrow p(x)$$

What happens if we reorder the statements in the program?

Alternative search strategies:

- Breadth first search.
- Breadth first search with list of active goals.

# Logic Programs

Transitive Closure:

$$\begin{aligned} tc(x,y) &\leftarrow r(x,y) \\ tc(x,y) &\leftarrow r(x,z), tc(z,y). \end{aligned}$$

Interpretation:

- $x$ is an ancestor of $y$ if $x$ is a parent of $y$.
- $x$ is an ancestor of $y$ if $x$ is a parent of $z$ and $z$ is an ancestor of $y$.

Factorial (model of recursion):

$$\begin{aligned} fact(0,1) &\leftarrow \\ fact(x, x*y) &\leftarrow fact(x-1, y). \end{aligned}$$

Interpretation:

- $0! = 1$.
- If $(x-1)! = y$, then $x! = x * y$.

# More Programs

Length of a list:

$$\begin{aligned} length(<>, 0) &\leftarrow \\ length(x :: L, z + 1) &\leftarrow length(L, z). \end{aligned}$$

Delete first occurrence of an element:

Procedural:

$$\begin{aligned} \text{delete}(x, L) = {} & \text{if } L = <> \text{ then } <> \\ & \text{else if head}(L) = x \text{ then tail}(L) \\ & \text{else head}(L)\text{::delete}(x, \text{ tail}(L)). \end{aligned}$$

Logic programming:

$$\begin{aligned} \text{remove}(x, <>, <>) &\leftarrow \\ \text{remove}(x, x :: T, T) &\leftarrow \\ \text{remove}(x, y :: T, y :: U) &\leftarrow \text{remove}(x, T, U). \end{aligned}$$

# Regular Languages

Given alphabet $A$, a **language** is a subset of $A^*$.

Can we write a program to dermine whether string $w$ is in language $L$?

The following languages are called **regular**:

- Basis: $\emptyset$, $\{\Lambda\}$, and $\{a\}$ are regular languages for all $a \in A$.
- Induction: If $L$ and $M$ are regular languages, then the following languages are also regular: $L \cup M$, $M \cdot L$ and $L^*$.

# Regular Expressions

- Basis: $\Lambda$, $\emptyset$, and $a$ are regular expressions for all $a \in A$.
- Induction: If $R$ and $S$ are regular expressions, then the following expressions are also regular: $(R)$, $R + S$, $R \cdot S$, and $R^*$.

Order of precedence: $(),^*, \cdot, +$.

Every regular expression represents a regular language, and every regular language is represented by a regular expression.

Given regular expression $R$, $L(R)$ stands for the language represented by $R$.

Some properties:

- $R^* = R^*R^* = (R^*)^* = R + R^*$.
- $R(SR)^* = (RS)^*R$.
- $(R^*S)^* = \Lambda + (R + S)^*S$.
- $(RS^*)^* = \Lambda + R(R + S)^*$.

# Property Proof

Theorem: $R^* = R^*R^*$

Proof:

- Need to show that $L(R^*) = L(R^*R^*)$.
- Note that $L(R^*) = L(R)^*$ and $L(R^*R^*) = L(R)^*L(R)^*$.
- Let $x \in L(R)*$. Then $x = x\Lambda \in L(R)^*L(R)^*$.
- Therefore, $L(R)^* \subset L(R)^*L(R)^*$.
- Suppose $x \in L(R)^*L(R)^*$.
- Then, $x = yz$ where $y \in L(R)^*$ and $z \in L(R)^*$.
- Thus, $y \in L(R)^k$ and $z \in L(R)^n$ for some $k$ and $n$.
- Therefore, $yz \in L(R)^{k+n}$, so $yz \in L(R)^*$.
- Therefore, $L(R)^*L(R)^* \subset L(R)^*$.

# Finite Automata

Regular languages and regular expressions describe (generate) a certain class of languages.

How do we **recognize** if a string is a member of a particular regular language?

Finite Automata – a severely restricted model of computation.

- No "stored program" concept – the machine is the computation.
- No auxiliary memory – the automaton fixes memory by its definition.
- Input: a string on a "tape"
- Read head moves over string (left to right).
- Output: "Accept" or "Not Accept"

Advantages of Finite Automata

- Well understood
- Clearly defined
- Provides good solutions to useful problems:
  - Lexical analysis
  - Pattern matching

# Deterministic Finite Automata (DFA)

DFA operation is completely determined by the input.

Input tape: made of squares, one symbol per square.

CPU is a finite collection of **states**.

At any instant, DFA is "in" some state.

As symbols are read, DFA may change to anther state.

# DFA Definition

A **Deterministic Finite Automaton** is a quintuple $M = (K, \Sigma, \delta, s, F)$ where

- $K$ is a finite set of **states**.
- $\Sigma$ is an alphabet.
- $s \in K$ is the **initial state**.
- $F \subset K$ is the set of **final states**.
- $\delta$, the **transition function**, is from $K \times \Sigma$ to $K$.

If $M$ is in state $q \in K$ and the symbol read from input is $\sigma \in \Sigma$, then $\delta(q, \sigma) \in K$ is the uniquely determined state to which $M$ passes.

# Example DFA

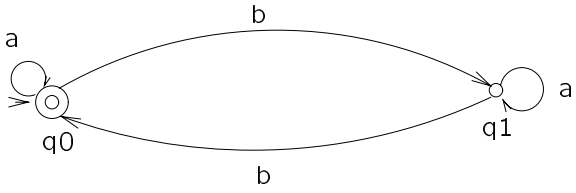
Define $M = (K, \Sigma, \delta, s, F)$ to be

- $K = \{q_0, q_1\}$
- $\Sigma = \{a, b\}$
- $s = q_0$
- $F = \{q_0\}$

and $\delta$ is the following function:

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $q_0$ |
| $q_0$ | $b$ | $q_1$ |
| $q_1$ | $a$ | $q_1$ |
| $q_1$ | $b$ | $q_0$ |

Can be drawn as:




Consider input $aabba$.

# Configurations

Read head only moves right – what has passed has no effect beyond determining current state.

Thus, remaining string and current state completely determine a **configuration**.

A configuration of $M = (K, \Sigma, \delta, s, F)$ is any element of $K \times \Sigma^*$. Example: $(q_2, aabbab)$.

Notation: $\vdash$: "yields" is a binary relation on configurations. The relation holds if $M$ can pass from one configuration to the other.

$(q, w) \vdash_M (q', w')$ iff $w = \sigma w'$ for $\sigma \in \Sigma$ and $\delta(q, \sigma) = q'$.

$\vdash_M^*$: transitive closure of $\vdash_M$:
$(q, w) \vdash_M^* (q', w')$ means $(q, w)$ yields $(q', w')$ after some number, possibly zero, of steps.

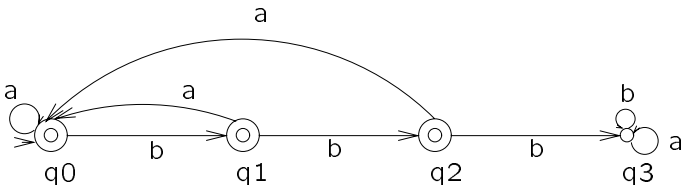
# Example

$$\begin{aligned}(q_0, aabba) &\vdash_M (q_0, abba)\\ &\vdash_M (q_0, bba)\\ &\vdash_M (q_1, ba)\\ &\vdash_M (q_0, a)\\ &\vdash_M (q_0, \Lambda)\end{aligned}$$

Therefore, $(q_0, aabba) \vdash_M^* (q_0, \Lambda)$ and so $aabba$ is accepted by $M$.

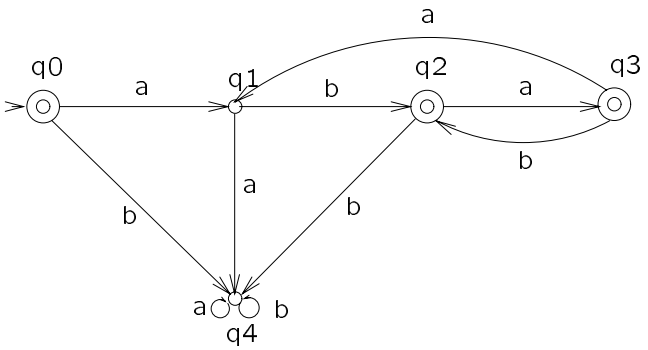
# DFA Example

Design a DFA $M$ that accepts language $L(M) = \{w : w \in \{a, b\}^*$ and $w$ does not contain three consecutive $b$'s$\}$.



# Example DFA

Design a DFA to accept language $L = (ab \cup aba)^*$.

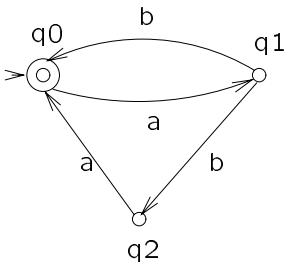


# Nondeterministic Finite Automata (NDFA)

Nondeterminism: Allow a choice of more than one "next state" for a given input symbol.

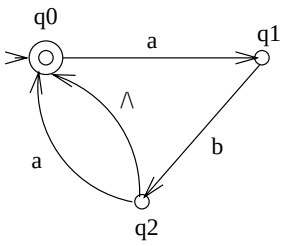
The choice is "nondeterministic"

The language is accepted if *any one* of the choices leads to terminating in an accepting state.

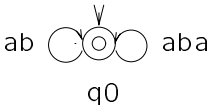


# Extensions to NDFA

Add "free" transitions on $\Lambda$.




Add transitions on substrings.



# Formal Definition of NDFA

A nondeterministic finite automaton is a quintuple $M = (K, \Sigma, \Delta, s, F)$ where

- $K$ is a finite set of states,
- $\Sigma$ is an alphabet,
- $s \in K$ is the initial state,
- $F \subset K$ is the set of final states,
- $\Delta$, the **transition relation** is a finite subset of $K \times \Sigma^* \times K$.

$M$, when in state $q$, may "consume" a substring $u$ from the input string and enter state $p$ iff $(q, u, p) \in \Delta$.

Triple $(q, u, p)$ is a **transition** of $M$.

Clearly, a DFA is a special case of a NDFA.

# Equivalence of DFA and NDFA

Finite automata $M_1$ and $M_2$ are **equivalent** iff $L(M_1) = L(M_2)$.

**Theorem:** For each NDFA, there is an equivalent DFA.

Difference between the two:

Read multiple characters at once.

Solution: Create separate states for each character.

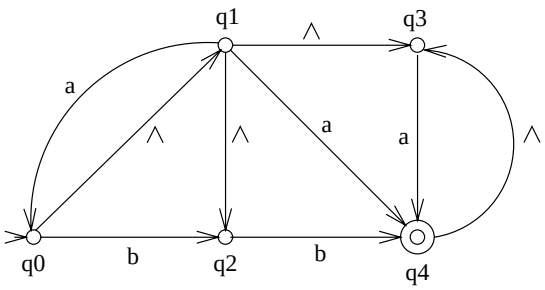
Transitions on $\Lambda$.

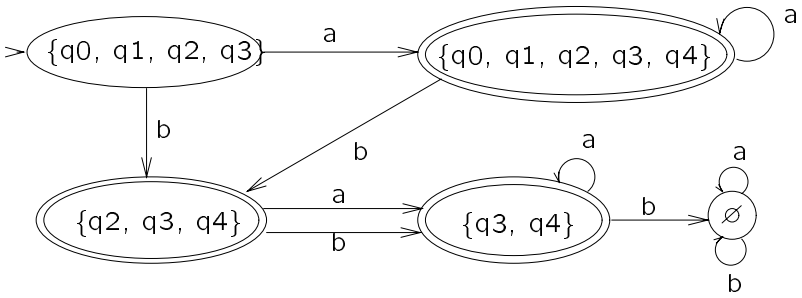
Solution: Given $(q_a, u, q_b)$ and $(q_b, \Lambda, q_c)$, replace the second with $(q_a, u, q_c)$.

Only remaining problem: multiple transitions on the same input character.

# Eliminating Multiple Transitions

Key idea: imagine the NDFA as occupying at any moment not a single state, but a *set* of states: all the states that can be reached from the initial state by means of the input consumed so far.



# Closure

The class of languages accepted by DFA is closed under

1. Union
2. Intersection
3. Concatenation
4. Kleene star (closure)
5. Complementation

In each case, a NDFA can be constructed from 1 or 2 NDFA for the appropriate input languages.

Examples:

(1) Nondeterministicly choose to enter the start state of the appropriate NDFA.

(5) Complement the set of accepting states.

# Other Useful Properties

The following questions can be answered about finite automata:

For FA $M$ and string $w$, is $w \in L(M)$?

For FA $M$, is $L(M) = \emptyset$?

For FA $M$, is $L(M) = \Sigma^*$?

For FA $M_1$ and $M_2$, is $L(M_1) \subset L(M_2)$?

For FA $M_1$ and $M_2$, is $L(M_1) = L(M_2)$?

# Grammar Rules

RE: $a(a^* \cup b^*)b$.

Consider this alternative view:

$$\begin{aligned} S &\rightarrow aM \\ M &\rightarrow A \\ M &\rightarrow B \\ A &\rightarrow b \\ A &\rightarrow aA \\ B &\rightarrow b \\ B &\rightarrow bB \end{aligned}$$

To generate the string $aaab$:

$$S \rightarrow aM \rightarrow aA \rightarrow aaA \rightarrow aaaA \rightarrow aaab$$

# Regular Grammars

A grammar is called a **regular grammar** iff the right-hand side of every rule contains at most one non-terminal, which, if present, must be the last symbol in the string.

$$\begin{aligned} S &\rightarrow w, \\ S &\rightarrow wT. \end{aligned}$$

Examples:

| Regular Expression | Regular Grammar |
|---|---|
| $a^*$ | $S \rightarrow \Lambda \vert aS$ |
| $(a+b)^*$ | |
| $a^*+b^*$ | |
| $a^*b$ | |
| $ba^*$ | |
| $(ab)^*$ | |
| $a^*bc^*$ | |

# FA and Regular Expressions

**Theorem**: A language is regular iff it is accepted by some FA.

Proof

(a) Only if:

- By definition, regular languages are the smallest class containing $\emptyset$ and each member of $\Sigma$, and closed under union, concatenation and Kleene star.
- Clearly, FA can recognize $\emptyset$ and each member of $\Sigma$.
- By an earlier theorem, FA are closed under union, concatenation and Kleene star.

(b) If: There is an algorithm for constructing a regular expression from a FA.

# What Languages are Regular?

Heuristics:

The amount of memory available to the FA is bounded and fixed in advance.
Thus, $\{a^n b^n : n \geq 0\}$ is not likely to be regular.

Infinite regular languages have regular structure arising from a star in the regular expression.

**Theorem**: Let $L$ be an infinite regular language. Then there are strings $x$, $y$ and $z$ such that $y \neq \Lambda$ and $xy^k z \in L$ for each $k \geq 0$.

**Proof**:

- $L$ is accepted by some DFA $M$ (with a finite number of states).
- By Pigeonhole Principle, there must be a cycle in $M$.
- This cycle corresponds to $y$, and can be repeated an arbitrary number of times.

This is an example of a **pumping theorem**.
Use: Since $\{a^n b^n : n \geq 0\}$ can't be rewritten as $xy^n z$, it can't be regular.

# Context-Free Grammar

A **context-free grammar** $G$ is a quadruple $(V, \Sigma, R, S)$ where

- $V$ is an alphabet,
- $\Sigma$ is a subset of $V$, (**terminals**),
- $R$ is a finite subset of $(V - \Sigma) \times V^*$, (**rules**),
- $S$ is an element of $V - \Sigma$ (**start symbol**).

Members of $V - \Sigma$ are called **nonterminals**.

For strings $u, v \in V^*$, write $u \Rightarrow v$ iff there are strings $x, y, v' \in V^*$ and $A \in V - \Sigma$ such that $u = xAy$, $v = xv'y$, and $A \to v'$.

$\overset{*}{\Rightarrow}$ is the reflexive, transitive closure of $\Rightarrow$.

$L(G)$, the language generated by $G$ is $\{w \in \Sigma^* : S \overset{*}{\Rightarrow} w\}$ (a **context-free language**).

Context free: For $aSb$, the possible replacements for $S$ are not affected by the values of $a$ and $b$.

# CFG Example

- $V = \{S, a, b\}$,
- $\Sigma = \{a, b\}$,
- $R = \{S \to aSb, S \to \Lambda\}$.

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

$L(G) = \{a^n b^n : n \geq 0\}$.

What does this say about the relationship between CFG and RE?

# CFG Example

$G = (V, \Sigma, R, E)$ as follows:

- $V = \{x, 1, 2, +, *, (, ), T, F, E\}$,
- $\Sigma = \{x, 1, 2, +, *, (, )\}$,

$$
\begin{aligned}
R = \{E &\rightarrow E + T, \\
E &\rightarrow T, \\
T &\rightarrow T * F, \\
T &\rightarrow F, \\
F &\rightarrow (E), \\
F &\rightarrow x1, \\
F &\rightarrow x2\}
\end{aligned}
$$

Example derivation:

$$(x1 * x2 + x1) * (x1 + x2)$$

# CFG Example

$G = (V, \Sigma, R, S)$ as follows:

- $V = \{S, (, )\}$,
- $\Sigma = \{(, )\}$,

$$R = \{\begin{array}{rcl} S & \rightarrow & \Lambda, \\ S & \rightarrow & SS, \\ S & \rightarrow & (S)\}. \end{array}$$

Example derivations:

$$\begin{array}{rcl} S & \Rightarrow & SS \Rightarrow S(S) \Rightarrow S((S)) \\ & \Rightarrow & S(()) \Rightarrow (S)(()) \Rightarrow ()(()). \end{array}$$

$$\begin{array}{rcl} S & \Rightarrow & SS \Rightarrow (S)S \Rightarrow ()S \\ & \Rightarrow & ()(S) \Rightarrow ()((S)) \Rightarrow ()(()). \end{array}$$

A derivation of a string from a CFG is a **leftmost derivation** if the nonterminal symbol replaced at every step is the leftmost nonterminal in the string.

# Extending FA

Not every CFL can be recognized by an FA.

Can we extend the FA concept to create FA-like machines that recognize a language iff it is a Context-Free language?

Consider $\{wcw^R : w \in \{a, b\}^*\}$.

An automaton that recognizes this must "remember" $w$ until it sees 'c' and then check's for $w^R$.

This can be done with a stack memory.

Checking for matching parentheses can also be done with a stack memory.

# Pushdown Automata

Consider a *non-deterministic* FA with stack memory.

This is called a **pushdown automaton**.

For any CFG rule $A \to wBx$, a pushdown automaton can imitate it by reading $w$, putting $x$ on the stack and going to state $B$.

The PDA can decide (nondeterministically) to pop values on the stack and compare against the next input characters.

**Theorem**: A language is Context-Free iff it can be recognized by a Pushdown Automaton.

# PDA Definition

A **pushdown automaton** is a sextuple $M = (K, \Sigma, , , \Delta, s, F)$, where

- $K$ is a finite set of **states**,
- $\Sigma$ is an alphabet (**input symbols**),
- , is an alphabet (**stack symbols**),
- $s \in K$ is the **initial state**,
- $F \subset K$ is the set of **final states**,
- $\Delta$ is a finite subset of $(K \times \Sigma^* \times , ^*) \times (K \times , ^*)$.

If $((p, u, \beta), (q, \gamma)) \in \Delta$, then when in state $p$ with $\beta$ at the top of the stack, $M$ may read $u$ from input, replace $\beta$ with $\gamma$ on the top of the stack, and enter state $q$.

$M$ accepts $w \in \Sigma^*$ iff $(s, w, \Lambda) \vdash_M^* (p, \Lambda, \Lambda)$ for $p \in F$.

**Push**: $((p, u, \Lambda), (q, a))$ pushes $a$ onto the stack.

**Pop**: $((p, u, a), (q, \Lambda))$ pops $a$ from the stack.

A DFA is simply a PDA that never operates on its stack.

# PDA Example

$L = \{wcw^R : w \in \{a, b\}^*\}$.

Let $M = (K, \Sigma, , , \Delta, s, F)$ where

- $K = \{s, f\}$,
- $\Sigma = \{a, b, c\}$,
- $, = \{a, b\}$,
- $F = \{f\}$,
- $\Delta = \begin{array}{l} ((s, a, \wedge), (s, a)) \\ ((s, b, \wedge), (s, b)) \\ ((s, c, \wedge), (f, \wedge)) \\ ((f, a, a), (f, \wedge)) \\ ((f, b, b), (f, \wedge)) \end{array}$

Example: abbcbba

# PDA Example

$L = \{ww^R : w \in \{a, b\}^*\}$.

Let $M = (K, \Sigma, , , \Delta, s, F)$ where

- $K = \{s, f\}$,
- $\Sigma = \{a, b\}$,
- $, \ = \{a, b\}$,
- $F = \{f\}$,
- $\Delta = \begin{array}{l} ((s, a, \Lambda), (s, a)) \\ ((s, b, \Lambda), (s, b)) \\ ((s, \Lambda, \Lambda), (f, \Lambda)) \\ ((f, a, a), (f, \Lambda)) \\ ((f, b, b), (f, \Lambda)) \end{array}$

Just like previous example, except for Transition 3.

Example: abbbba

# PDA Example

$L = \{w \in \{a,b\}^* : w$ has the same number of $a$'s and $b$'s $\}$.

Let $M = (K, \Sigma, , , \Delta, s, F)$ where

- $K = \{s, q, f\}$,
- $\Sigma = \{a, b\}$,
- $, \ = \{a, b\}$,
- $F = \{f\}$,
- $\Delta = \begin{array}{l} ((s, \wedge, \wedge), (q, c)) \\ ((q, a, c), (q, ac)) \\ ((q, a, a), (q, aa)) \\ ((q, a, b), (q, \wedge)) \\ ((q, b, c), (q, bc)) \\ ((q, b, b), (q, bb)) \\ ((q, b, a), (q, \wedge)) \\ ((q, \wedge, c), (f, \wedge)) \end{array}$

'c' is a special symbol to mark the bottom of the stack.

Keep either the excess $a$'s or excess $b$'s in the stack.

# Deterministic PDA

A PDA is deterministic if we can always decide which transition to use next.

$\{wcw^R : w \in \{a,b\}^*\}$ is deterministic – once a '$c$' is read, start matching $w^R$ against the stack.

$\{ww^R : w \in \{a,b\}^*\}$ is not deterministic:

$$\begin{aligned}\Delta \;=\; & ((s,a,\Lambda),(s,a)) \\ & ((s,b,\Lambda),(s,b)) \\ & ((s,\Lambda,\Lambda),(f,\Lambda)) \\ & ((f,a,a),(f,\Lambda)) \\ & ((f,b,b),(f,\Lambda))\end{aligned}$$

Rule 3 is confused with rules 1 or 2.

A language $L$ is **Deterministic context-free** if there is some deterministic PDA $M$ that recognizes $L\$$.

# Making PDA Deterministic

$L(G) = \{a^n b^n : n \geq 0\}$.

$$S \to aSb \qquad S \to \Lambda$$

PDA $M_1 = (\{p, q\}, \{a, b\}, \{a, b, S\}, \Delta_1, p, \{q\})$

$$\begin{aligned} \Delta_1 \;=\; & ((p, \Lambda, \Lambda), (q, S)) \\ & ((q, \Lambda, S), (q, aSb)) \\ & ((q, \Lambda, S), (q, \Lambda)) \\ & ((q, a, a), (q, \Lambda)) \\ & ((q, b, b), (q, \Lambda)) \end{aligned}$$

This is not deterministic.

Can convert to a deterministic PDA that recognizes $L\$$ by realizing that we can decide what to do by looking ahead by one input symbol.

# Deterministic Version

$M_2 =$
$(\{p, q, q_a, q_b, q_\$\}, \{a, b, \$\}, \{a, b, S\}, \Delta_2, p, \{q_\$\})$

$$\begin{aligned}
\Delta_2 \;=\; & ((p, \wedge, \wedge), (q, S)) \\
& ((q, a, \wedge), (q_a, \wedge)) \\
& ((q_a, \wedge, a), (q, \wedge)) \\
& ((q, b, \wedge), (q_b, \wedge)) \\
& ((q_b, \wedge, b), (q, \wedge)) \\
& ((q, \$, \wedge), (q_\$, \wedge)) \\
& ((q_a, \wedge, S), (q_a, aSb)) \\
& ((q_b, \wedge, S), (q_b, \wedge))
\end{aligned}$$

| | | | | | |
|---|---|---|---|---|---|
| 0 | $p$ | $ab\$$ | $\wedge$ | – | |
| 1 | $q$ | $ab\$$ | $S$ | 1 | |
| 2 | $q_a$ | $b\$$ | $S$ | 2 | |
| 3 | $q_a$ | $b\$$ | $aSb$ | 7 | $S \to aSb$ |
| 4 | $q$ | $b\$$ | $Sb$ | 3 | |
| | $q_b$ | $\$$ | $Sb$ | 4 | |
| 6 | $q_b$ | $\$$ | $b$ | 8 | $S \to \wedge$ |
| 7 | $q$ | $\$$ | $\wedge$ | 5 | |
| 8 | $q_\$$ | $\wedge$ | $\wedge$ | 6 | |

$M_2$ is an example of a **top-down parser**.

# Restricted CFG's

For CFL $\Lambda$, we can replace any grammar whose rules contain $\Lambda$ with an equivalent grammar that does not.

We can find a CFG for any CFL whose rules have the following form (Chomsky Normal Form):

- Righthand side is a single terminal, or
- Righthand side is exactly two nonterminals.
- The rule $S \to \Lambda$ is allowed if $\Lambda$ is part of the language.

We can find a CFG for any CFL whose rules have the following form (Greibach Normal Form):

- Righthand side is a single terminal followed by zero or more nonterminals.
- The rule $S \to \Lambda$ is allowed if $\Lambda$ is part of the language.

# Pumping Lemma for CFL

Pumping Lemma for CFL:

- Let $L$ be an infinite CFL.
- There is a positive integer $m$ such that for all strings $z \in L$ with $|z| \geq m$, $z$ can be written in the form $z = uvwxy$, with the following properties:
  - $|vx| \geq 1$.
  - $|vwx| \leq m$.
  - $uv^k wx^k y \in L$ for all $k \geq 0$.

Equivalently, there is a derivation for $w$ of the form

$$S \stackrel{*}{\Rightarrow} uAz \stackrel{*}{\Rightarrow} uvAyz \stackrel{*}{\Rightarrow} uvxyz$$

# Using the Pumping Lemma

**Theorem**: $L = \{a^n b^n c^n\}$ is not context-free.

Proof:

- Assume that it is.
- Then there is a string $z = a^m b^m c^m \in L$.
- Since $|z| \geq m$, we can write $z = uvwxy$.
- Here, $|vx| \geq 1$, $|vwx| \leq m$, $uv^k w^k y \in L$ for $k \geq 0$.
- Observation: Pumped variable $v$ can't contain two distinct letters.
- Likewise for $x$.
- At least one of $v, x$ is non-empty.
- Therefore, not all three letters can get pumped up equally.
- This is a contradiction.

# Closure

**Theorem**: The CF Languages are closed under union, concatenation and Kleene star.

**Theorem**: The intersection of a CF language and a regular language is a CF language.

**Theorem**: The CF languages are not closed under intersection or complementation.

**Theorem**: There is an algorithm to answer the following question:

- Given a CF grammar $G$ and a string $w$, is $w \in L(G)$?

# A More General Model

Neither FA nor PDA are general models of computation

- Language Recognizers

Want a general model of computation that is as simple as possible.

- Wish to be able to reason about the model.
- Automata are simple.

Necessary features:

- Read
- Write
- Compute

# Turing Machines

A single I/O head:

- Read current symbol
- Change current symbol

Control Unit Actions:

- Put the control unit into a new state.
- Either:
  1. Write a symbol in current tape square.
  2. Move I/O head one square left or right.

Tape has a fixed left end, infinite right end.

- Machine ceases to operate if head moves off left end.
- Input is placed on left end of tape.

A **halt** state ($h$) signals end of computation (no concept of accepting state).

"#" indicates a blank tape square.

# Formal definition of Turing Machine

A **Turing Machine** is a quadruple $(K, \Sigma, \delta, s)$ where

- $K$ is a finite set of **states** (not including $h$).
- $\Sigma$ is an alphabet (containing #, not $L$ or $R$).
- $s \in K$ is the **initial** state.
- $\delta$ is a function from $K \times \Sigma$ to $(K \cup \{h\}) \times (\Sigma \cup \{L, R\})$.

If $q \in K$, $a \in \Sigma$ and $\delta(q, a) = (p, b)$, then when in state $q$ and scanning $a$, enter state $p$ and

1. If $b \in \Sigma$ then replace $a$ with $b$.
2. Else ($b$ is $L$ or $R$) Move head.

# Turing Machine Example

$M = (K, \Sigma, \delta, s)$ where

- $K = \{q_0, q_1\}$,
- $\Sigma = \{a, \#\}$,
- $s = q_0$,
- $\delta =$

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $(q_1, \#)$ |
| $q_0$ | $\#$ | $(h, \#)$ |
| $q_1$ | $a$ | $(q_0, a)$ |
| $q_1$ | $\#$ | $(q_0, R)$ |

# Turing Machine Example

$M = (K, \Sigma, \delta, s)$ where

- $K = \{q_0\}$,
- $\Sigma = \{a, \#\}$,
- $s = q_0$,
- $\delta =$

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $(q_0, L)$ |
| $q_0$ | $\#$ | $(h, \#)$ |

# Notation

Configuration: $(q, aaba\#\underline{\#}a)$

**Halted configuration**: $q$ is $h$.

**Hanging configuration**: Move left from leftmost square.

A **computation** is a sequence of configurations for some $n \geq 0$. Such a computation is of **length** $n$.

$$
\begin{aligned}
(q_0, \underline{a}aaa) &\vdash_M (q_1, \underline{\#}aaa) \\
&\vdash_M (q_0, \#\underline{a}aa) \\
&\vdash_M (q_1, \#\underline{\#}aa) \\
&\vdash_M (q_0, \#\#\underline{a}a) \\
&\vdash_M (q_1, \#\#\underline{\#}a) \\
&\vdash_M (q_0, \#\#\#\underline{a}) \\
&\vdash_M (q_1, \#\#\#\underline{\#}) \\
&\vdash_M (q_0, \#\#\#\#\underline{\#}) \\
&\vdash_M (h, \#\#\#\#\underline{\#})
\end{aligned}
$$

# Computations

$M$ is said to **halt on input** $w$ iff $(s, \#w\underline{\#})$ yields some halted configuration.

$M$ is said to **hang on input** $w$ if $(s, \#w\underline{\#})$ yields some hanging configuration.

Turing machines compute functions from strings to strings.

Formally: Let $f$ be a function from $\Sigma_0^*$ to $\Sigma_1^*$. Turing machine $M$ is said to **compute** $f$ if for any $w \in \Sigma_0^*$, if $f(w) = u$ then

$$(s, \#w\underline{\#}) \vdash_M^* (h, \#u\underline{\#}).$$

$f$ is said to be a **Turing-computable function**.

Multiple parameters: $f(w_1, ..., w_k) = u$, $(s, \#w_1\#w_2\#...\#w_k\underline{\#}) \vdash_M^* (h, \#u\underline{\#})$.

# Functions on Natural Numbers

Represent numbers in **unary** notation on symbol $I$ (zero is represented by the empty string).

$f : \mathbb{N} \to \mathbb{N}$ is computed by $M$ if $M$ computes $f' : \{I\}^* \to \{I\}^*$ where $f'(I^n) = I^{f(n)}$ for each $n \in \mathbb{N}$.

Example: $f(n) = n + 1$ for each $n \in \mathbb{N}$.

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $I$ | $(h, R)$ |
| $q_0$ | $\#$ | $(q_0, I)$ |

$$(q_0, \#II\underline{\#}) \vdash_M (q_0, \#II\underline{I}) \vdash_M (h, \#III\underline{\#}).$$

In general, $(q_0, \#I^n\underline{\#}) \vdash_M^* (h, \#I^{n+1}\underline{\#})$.

What about $n = 0$?

# Turing-decidable Languages

A language $L \subset \Sigma_0^*$ is **Turing-decidable** iff function $\chi_L : \Sigma_0^* \to \{\boxed{\text{Y}}, \boxed{\text{N}}\}$ is Turing-computable, where for each $w \in \Sigma_0^*$,

$$\chi_L(w) = \begin{cases} \boxed{\text{Y}} & \text{if } w \in L \\ \boxed{\text{N}} & \text{otherwise} \end{cases}$$

Ex: Let $\Sigma_0 = \{a\}$, and let $L = \{w \in \Sigma_0^* : |w| \text{ is even}\}$.

$M$ erases the marks from right to left, with current parity encode by state. Once blank at left is reached, mark $\boxed{\text{Y}}$ or $\boxed{\text{N}}$ as appropriate.

# Turing-acceptable Languages

$M$ **accepts** a string $w$ if $M$ halts on input $w$.

- $M$ accepts a language iff $M$ halts on $w$ iff $w \in L$.
- A language is **Turing-acceptable** if there is some Turing machine that accepts it.

Ex: $\Sigma_0 = \{a, b\}$,
$L = \{w \in \Sigma_0^* : w$ contains at least one $a\}$.

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $(h, a)$ |
| $q_0$ | $b$ | $(q_0, L)$ |
| $q_0$ | # | $(q_0, L)$ |

Every Turing-decidable language is Turing-acceptable.

# Combining Turing Machines

**Lemma**: If

$$(q_1, w_1\underline{a_1}u_1) \vdash_M^* (q_2, ww_2\underline{a_2}u_2)$$

for string $w$ and

$$(q_2, w_2\underline{a_2}u_2) \vdash_M^* (q_3, w_3\underline{a_3}u_3),$$

then

$$(q_1, w_1\underline{a_1}u_1) \vdash_M^* (q_3, ww_3\underline{a_3}u_3).$$

Insight: Since $(q_2, w_2\underline{a_2}u_2) \vdash_M^* (q_3, w_3\underline{a_3}u_3)$, this computation must take place without moving the head left of $w_2$

- The machine cannot "sense" the left end of the tape

# Combining Turing Machines (Cont)

Thus, the head won't move left of $w_2$ even if it is not at the left end of the tape.

This means that Turing machine computations can be combined into larger machines:

- $M_2$ prepares string as input to $M_1$.
- $M_2$ passes control to $M_1$ with I/O head at end of input.
- $M_2$ retrieves control when $M_1$ has completed.

# Some Simple Machines

Basic machines:

- $|\Sigma|$ symbol-writing machines (one for each symbol).
- Head-moving machines R and L move the head appropriately.

More machines:

- First do $M_1$, then do $M_2$ or $M_3$ depending on current symbol.
- (For $\Sigma = \{a, b, c\}$) Move head to the right until a blank is found.
- Find first blank square to left: $L_{\#}$
- Copy Machine: Transform $\#w\underline{\#}$ into $\#w\#w\underline{\#}$.
- Shift a string left or right.

# A Non-CF Language

Construct a Turing machine to recognize the following language:

$$\{a^n b^n c^n | n \geq 0\}$$

If the start cell is empty, then HALT.
If the current cell contains $a$,
  then write $X$ and scan right.
    Look for $b$, replace with $Y$.
    Look for $c$, replace with $Z$.
  Now, scan left to an $a$ to right of $X$.
  Repeat process.
Make sure nothing to right of $c$'s.

# Extensions

The following extensions do not increase the power of Turing Machines.

- 2-way infinite tape
- Multiple tapes
- Multiple heads on one tape
- Two-dimensional "tape"
- Non-determinism

# Grammars

**Unrestricted grammar**, also known as a **rewriting system**.

A **Grammar** is a quadruple $(V, \Sigma, R, S)$ where

- $V$ is an alphabet.
- $\Sigma \subset V$ is the set of **terminal** symbols, and $V - \Sigma$ is the set of **nonterminal** symbols.
- $S \in V - \Sigma$ is the **start symbol**.
- $R$, the set of **rules**, is a finite subset of $(V^*(V - \Sigma)V^*) \times V^*$.

Generalizes CFG: the left hand side of a rule need not consist of only a single non-terminal.

Example: $L = \{w \in \{a, b, c\}^* : w$ has equal numbers of $a$'s, $b$'s, and $c$'s$\}$.

$$\begin{array}{ccc} S \to \Lambda & S \to ABCS & \\ AB \to BA & AC \to CA & BC \to CB \\ BA \to AB & CA \to AC & CB \to BC \\ A \to a & B \to b & C \to c \end{array}$$

# Grammars and Turing Machines

**Theorem**: Let $M$ by any Turing machine. Then there is a grammar $G$ such that for any configurations $(q, u\underline{a}v)$ and $(q', u'\underline{a'}v')$ of $M$,

$$(q, u\underline{a}v) \vdash_M^* (q', u'\underline{a'}v')$$

if and only if

$$[uqav] \stackrel{*}{\Rightarrow} [u'q'a'v']$$

Proof is by simulation: Each transition in $M$ is replaced by a corresponding grammar rule

A function is **grammatically computable** iff there is a grammar $G$ such that $f(u) = v$ iff $xuy \stackrel{*}{\Rightarrow} x'vy'$.

(Consider $x, y, x', y'$ as end markers to set the context.)

Conclusion: Every Turing-computable function from strings to strings or numbers to numbers is grammatically computable.

# Partial Recursive Functions

A function is **Partial Recursive** if it can be generated from the following operations.

- $zero() = 0$.
- $succ(x) = x + 1$ for all $x \in \mathbb{N}$.
- $p_i(x_1, ..., x_n) = x_i$ for $1 \leq i \leq n$.
- **Composition**: for partial recursive functions $h, g_1, ..., g_m$, we can form a new partial recursive function $f$ as follows:

  $$f(x) = h(g_1(x), ..., g_m(x)).$$

  We can use any number of arguments in place of $x$.
- **Primitive Recursion**: We can construct new function $f$ from parital recursive functions $h$ and $g$ as follows:

  $$f(x, 0) = h(x),$$
  $$f(x, succ(y))) = g(x, y, f(x, y)).$$

# Parital Recursive Functions (Cont.)

**unbounded minimalization**: for $g$ a partial recursive function,

$$f(x) = \min_y(g(x, y) = 0)$$

that is, the least value $y$ that satisfies $g(x, y) = 0$.

# Examples of Partial Recursive Functions

All constants: Apply the successor function to the zero function a constant number of times.

Addition:

$$n + 0 = n.$$

$$n + (m + 1) = (n + 1) + m.$$

Multiplication:

$$n \cdot 0 = 0.$$

$$n \cdot (m + 1) = n \cdot m + n.$$

Non-total function:

$$h(x, y) = \min_z(x = y + z).$$

# Gödelization

We can convert numbers to strings by using Unary notation.

We can convert strings to unique numbers as well.

Assign each character in $\Sigma$ a unique number.

- The $i$th character has value $i$.

Denote the $i$th prime number as $p_i$.

Represent string $S = s_{i_1} s_{i_2} ... s_{i_k}$ of length $k$ as:

$$g(s_{i_1} s_{i_2} ... s_{i_k}) = p_1^{i_1} p_2^{i_2} \cdots p_k^{i_k}.$$

Example: CAT is $2^3 3^1 5^{20}$.

Now we can use partial recursive functions to compute functions from strings to strings.

# Church's Thesis

We now have three "general purpose" forms of computation: Turing machines, grammars and partial recursive functions.

Each of the three can be transformed into one of the other two. Thus, they are all "equally powerful."

We can create a **Universal Turing machine**:

- input is a specification for a Turing machine $M$ and a string $w$.
- output is the output from $M$ on $w$.

**Church's Thesis** (also **Church-Turing Thesis**): Turing machines are formal versions of algorithms, and no computational procedure will be considered an algorithm unless it can be presented as a Turing machine.

A "thesis" because it is unprovable: asserts that a certain informal concept (algorithm) corresponds to a mathematical concept (Turing machine).

# Church's Thesis (cont)

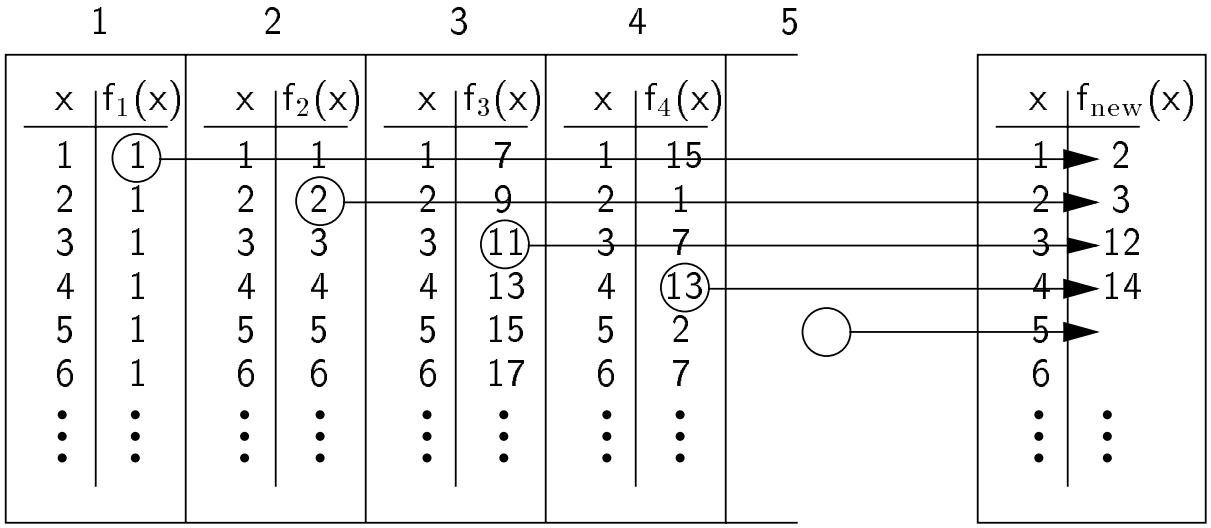
**How to disprove Church's Thesis**: Devise another model of computation that does finite labor at each step, and which can compute functions not computable by any Turing machine.

# Uncomputable Functions

Consider the set of functions $f(x) = y$ for $x, y$ natural numbers.

The set of such functions is uncountable.

Diagonalization argument:




Thus, not all functions on natural numbers are computable.

# Turing Decidable vs. Turing Acceptable Languages

Any Turing Decidable language is Turing Acceptable.

Problem: Given a Turing Machine $M$ and a string $w$, will $M$ halt when run on $w$?

If there is a Turing machine $M_0$ that solves this problem, then an algorithm to convert a Turing-accepting machine $M_1$ to a Turing-deciding machine is:

- Feed $M_1$ and its input $w$ to $M_0$.
- If the answer is "halts" then output Y.
- Otherwise, output N.

This crucial question is called **the Halting Problem**.

# Halting Problem for Programs

Can a **C** program be written to solve the following problem?

**Halting Problem**:

- Input: A program $P$ and input $X$.
- Output: "Halts" if $P$ halts when run with $X$ as input. "Does not Halt" otherwise.

# Halting Problem Proof

**Theorem**: There is no program to solve the Halting Problem.

**Proof**: (by contradiction).

Assumption: There is a **C** program that solves the Halting Problem.

```
bool halt(char* prog, char* input)
{
  Code to solve halting problem
  if (prog does halt on input) then
    return(TRUE);
  else
    return(FALSE);
}
```

# Two More Procedures

```
bool selfhalt(char *prog) {
  // Return TRUE if program halts
  //  when given itself as input.
  if (halt(prog, prog))
    return(TRUE);
  else
    return(FALSE);
}

void contrary(char *prog) {
  if (selfhalt(prog))
    while(TRUE); // Go into an infinite loop
}
```

# The Punchline

What happens when function `contrary` is run on itself?

Case 1: `selfhalt` returns `TRUE`.

- `contrary` will go into an infinite loop.
- This contradicts the result from `selfhalt`.

`selfhalt` returns `FALSE`.

- `contrary` will halt.
- This contradicts the result from `selfhalt`.

Either result is impossible.

The only flaw in this argument is the assumption that `halt` exists.

Therefore, `halt` cannot exist.

# Turing Machine Noncomputable Functions

Turing machine form of the Halting Problem: Given Turing Machine $M$ and string $w$, does $M$ halt on input $w$?

- Accept proof for programs that this is noncomputable.

Does a program (machine) halt on the EMPTY input?

No. Proof (Turing machine style):

- Suppose that $M_0$ decided the language

$$\{L(M) : M \text{ accepts } \Lambda\}.$$

- Given arbitrary machine $M$ and string $w$, we can create a new machine $M_w$ that operates as follows on empty input:
  - Write $w$ on the tape.
  - Simulate the execution of $M$.
- Now, we can apply $M_0$ to $M_w$ to solve the original halting problem.
- Thus, $M_0$ must not exist.

# Noncomputable Functions

Given a Turing machine $M$, is there any string at all on which $M$ halts?

Given a Turing machine $M$, does $M$ halt on every input?

Given two Turing machines $M_1$ and $M_2$, do they halt on the same input strings?

Given a Turing machine $M$, is the language $M$ accepts regular? Is it context-free? Is it Turing-decidable?

# Other Noncomputable Functions

(Note: there are equivalent unanswerable questions for grammars.)

Is there ANY input for which a program (machine) halts?

Does a program (machine) halt on EVERY input?

Do two programs (machines) compute the SAME function?

Does a particular line (transition) in a program (machine) get executed?

Does a program (machine) compute a particular function?

Does a program contain a "computer virus"?

# Evaluation of Expressions

How do we evaluate expressions?

Is there more than one way to evaluate an expression?

Can we tell if two expressions have the same value?

To evaluate expressions automatically, we need ways to transform one expression to another.

A **transformation rule** or **reduction rule** or **rewrite rule**:

- A production of the form $E \to F$ where $E$ and $F$ are expressions.
- "$E$ reduces to $F$."

# Reduction Sequences

Can apply reduction rules to subexpressions.

Given $G_0$ that contains $E$, we can reduce one or more instances of $E$ to $F$ to get $G_1$.

- Can have a **reduction sequence** $G_0 \to G_1 \to G_2 \to ...$

Reduction rules may come from axioms.

- The axiom $E = F$ yields rules $E \to F$ and $F \to E$.
- Given succ(0) = 1, succ(0) $\to$ 1 may be the more useful of the pair.

# Reduction Sequences (cont)

A sequence of reductions might not be unique.

$$s(0) + s(s(0)) \to 1 + s(s(0)) \to 1 + s(1).$$

$$s(0) + s(s(0)) \to s(0) + s(1) \to 1 + s(1).$$

Not all reduction sequences result in the same value.

$$a \to b, \quad a \to c, \quad b \to d, \quad b \to a$$

$$a \to b \to d$$
$$a \to c$$

The rule $g(x) \to g(g(x))$ leads to an infinite chain of reductions.

A set of rules has the **Church-Rosser property** if whenever $E$ evaluates – in two different ways – to $F$ and $G$, then there is $H$ and reductions from $F$ to $H$ and $G$ to $H$.

# Lambda Calculus

Wffs of the Lambda calculus are called **lambda expressions**.

$$\begin{aligned} E \rightarrow &\ \ V | (EE) | \lambda V.E \\ V \rightarrow &\ \ \text{identifier.} \end{aligned}$$

Expression of the form $\lambda x.M$ is an **abstraction**.

- A function definition.

Expression of the form $(MN)$ is an **appliction** of $M$ to $N$.

Write $f(x) = a$ and $g(x) = x$ as $\lambda x.a$ and $\lambda x.x$.

Write $f(g(b))$ as $(\lambda x.a(\lambda x.x\ b))$

$x$ is bound if $x$ occurs in a subexpression of from $\lambda x.M$. Otherwise, $x$ is free.

- $\lambda x.x$: $x$ is bound.
- $\lambda y.x$: $y$ is bound, $x$ is free.
- $(\lambda y.(\lambda x.x\ y)x)$: Only the rightmost $x$ is free.

Expression is **closed** if it has no free variables.

# Variable Substitution

If $E$ is an expression, then replacing all free occurances of $x$ by $N$ is denoted

$$E[x/N].$$

$\beta$-reduction:

$$(\lambda x.M\ \ N) \rightarrow M[x/N].$$

Evaluate: $(\lambda x.x\ \ a)$

$$(\lambda x.x\ \ a) \rightarrow a.$$

Evaluate: $(\lambda x.y\ \ a)$

$$(\lambda x.y\ \ a) \rightarrow y.$$

# Normal Form

An expression of the form $(\lambda x.M\ N)$ is called a **redex**: It has a reducible expression.

An expression is called a **normal form** if it cannot be reduced.

- $x$, $(y\ a)$, $\lambda x.y$: Yes.
- $(x(\lambda y.z\ a))$: No.

Does every expression have a normal form?

Given: $\lambda x.(x\ x)$.

- Evaluate $(\lambda x.(x\ x)\ \lambda x.(x\ x))$.

# Reduction Order

Given: $(\lambda x.x(\lambda y.y\ a))$.

$$\begin{array}{lcl}(\lambda x.x(\lambda y.y\ a)) & \rightarrow & (\lambda x.x\ a) \rightarrow a. \\ (\lambda x.x(\lambda y.y\ a)) & \rightarrow & (\lambda y.y\ a) \rightarrow a.\end{array}$$

A redex is **innermost** if it does not contain any redexes.

A redex is **outermost** if it is not contained in any redexes.

**Applicative Order Reduction** (call by value): Reduce the leftmost of the innermost redexes first.

- $(\lambda x.x(\lambda y.y\ a)) \rightarrow (\lambda x.x\ a) \rightarrow a.$
- $(\lambda y.(\lambda x.y\ z)\ w) \rightarrow (\lambda y.y\ w) \rightarrow w.$

**Normal Order Reduction** (call by name): Reduce the leftmost of the outermost redexes first.

- $(\lambda x.x(\lambda y.y\ a)) \rightarrow (\lambda y.y\ a) \rightarrow a.$
- $(\lambda y.(\lambda x.y\ z)\ w) \rightarrow (\lambda x.w\ z) \rightarrow w.$

# Reduction Order (Cont.)

Regardless of which we use, *if* we reach normal form, they are the same except possibly for variable name changes.

Example:

$$(\lambda x.\lambda y.y\ (\lambda z.(z\ z)\ \lambda z.(z\ z))).$$

NOR:

$$(\lambda x.\lambda y.y\ (\lambda z.(z\ z)\ \lambda z.(z\ z))) \rightarrow \lambda y.y.$$

AOR:

$$\begin{aligned}(\lambda x.\lambda y.y\ (\lambda z.(z\ z)\ \lambda z.(z\ z)))& \\ \rightarrow (\lambda x.\lambda y.y\ (\lambda z.(z\ z)\ \lambda z.(z\ z)))& \\ \rightarrow (\lambda x.\lambda y.y\ (\lambda z.(z\ z)\ \lambda z.(z\ z)))& \\ \rightarrow \ldots&\end{aligned}$$

# Renaming Rules

We must be careful about how we rename things.

$$((\lambda x.\lambda y.(y\ x)\ y)\ x) \rightarrow (\lambda y.(y\ y)\ x) \rightarrow (x\ x).$$

Rename $y$ as $z$.

$$((\lambda x.\lambda z.(z\ x)\ y)\ x) \rightarrow (\lambda z.(z\ y)\ x) \rightarrow (x\ y).$$

$\alpha$-Conversion:

$$\lambda x.M = \lambda y.M[x/y]$$

... if $y$ is a new variable not occurring in $M$.

$\beta$-Reduction:

$$(\lambda x.M\ N) \rightarrow M[x/N]$$

... if $N$ is free to replace $x$ in $M$. This occurs if the bound variables and the free variables in $\lambda x.M$ are disjoint.

$\eta$-Reduction:

$$\lambda x.(M\ x) = M \quad \text{if } x \text{ is not free in } M.$$

$\beta$-reduction and $\eta$-reduction both have the Church-Rossler property.

# Simplified Syntax

Write $g = \lambda x.M$ as

$$gx = M.$$

Write $h = \lambda x.\lambda y.M$ as

$$hxy = M.$$

Define booleans as:

$$\begin{aligned} \text{true} &= \lambda x.\lambda y.x \\ \text{false} &= \lambda x.\lambda y.y \end{aligned}$$

Define "if $C$ then $M$ else $N$" as $C\ M\ N$.

- Evaluate: true $M\ N$
- Evaluate: false $M\ N$

$$\begin{aligned} \text{And}\ M\ N &= M\ N\ \text{false}. \\ \text{Or}\ M\ N &= M\ \text{true}\ N. \\ \text{Not}\ M &= M\ \text{false true}. \end{aligned}$$

# Lists

$$
\begin{aligned}
\text{emptyList} &= \lambda x.\text{true} \\
\text{isEmpty} &= \lambda s.(s\lambda h.\lambda t.\text{false}) \\
\text{cons} &= \lambda h.\lambda t.\lambda s.(s\ h\ t) \\
\text{head} &= \lambda x.(x\ \text{true}) \\
\text{tail} &= \lambda x.(x\ \text{false})
\end{aligned}
$$

# Knuth-Bendix Completion

Given: A set of axioms.

Goal: CONSTRUCT a set of reduction rules that satisfy the following **completion** property:

- An equation $s = t$ is derivable from the axioms if and only if $s$ and $t$ can be reduced to the same **normal** form.

In general, this problem is unsolvable.

However, for many sets of axioms, Knuth-Bendix completion will work.

# Simple Example

Axioms:

- $g^5(x) = x$.
- $g^3(x) = x$.

Observation:

$$g(x) = g(g^5(x)) = g^6(x) = g^3(g^3(x)) = g^3(x) = x.$$

Therefore, the axioms imply the following:

- $g(x) = x$.

In fact, we can eliminate the original axioms!

From this, we see that $g^k(x) = x$.

Can we automate the process of locating the new axiom?

# Observations

To obtain $g(x) = g(g^5(x))$ we replaced $x$ by $g^5(x)$.

- Used $x = g^5(x)$ instead of $g^5(x) = x$.

In contrast, to obtain $g^3(g^3(x)) = g^3(x) = x$, we replaced $g^3(x)$ by $x$.

For the final axiom, we replace $g(x)$ with $x$.

- Reduction rule $g(x) \to x$.

# Terminology

If $u$ is an expression containing subexpression $s$, write $u[s]$.

Let $u[t]$ be an expression obtained from $u[s]$ by replacing one occurrence of $s$ by $t$.

Example:

- $u = f(g(x), g(x))$.
- Write $u[g(x)]$.
- Then $u[t]$ can be $f(t, g(x))$ or $f(g(x), t)$.

Assume there is a well-founded order $\succ$ defined on the expressions such that the following **monotonic property** holds for all expressions $s$, $t$, and $u[s]$ and all substitutions $\Theta$:

- If $s \succ t$, then $u[s] \succ u[t]$ and $s\Theta > t\Theta$.

Example: For expressions of the form $g^n(x)$:

- Define $g^n(x) \succ g^m(x)$ for all $n > m$.

Note that we cannot compare $g^2(x)$ and $g(y)$.

- Consider the substitution $\{y/g^2(x)\}$.

# Terminology (Cont.)

Specialization:

- For expressions $s$ and $l$, write $s \rhd l$ if a subexpression of $s$ is an instance of $l$ but no subexpression of $l$ is an instance of $s$.
- Example: $f(x, g(z)) \rhd g(z)$.

If $s = t$, write $s \leftrightarrow t$.

For inference rules, write $< E, R >$ with $E$ a set of equations and $R$ a set of reduction rules.

$t \rightarrow_R u$ means that $t$ reduces to $u$ by using the rules of $R$.

# Knuth-Bendix Inference Rules

Delete:

$$\frac{< E \cup \{s \leftrightarrow s\}, R >}{\therefore < E, R >}.$$

Compose:

$$\frac{< E, R \cup \{s \to t\} >, t \to_R u}{\therefore < E, R \cup \{s \to u\} >}.$$

Simplify:

$$\frac{< E \cup \{s \leftrightarrow t\}, R >, t \to_R u}{\therefore < E \cup \{s \leftrightarrow u\}, R >}.$$

Orient:

$$\frac{< E \cup \{s \leftrightarrow t\}, R >, s \succ t}{\therefore < E, R \cup \{s \to t\} >}.$$

Collapse:

$$\frac{\begin{array}{c} <E, R \cup \{s \to t\}>, s \to_R u \\ \text{by rule } l \to r \text{ in } R \text{ with } s \rhd l \end{array}}{\therefore < E \cup \{u \leftrightarrow t\}, R >}.$$

Deduce:

$$\frac{< E, R >, s \leftarrow_R u \to_R t}{\therefore < E \cup \{s \leftrightarrow t\}, R >}.$$

# Completion

Begin with $< E_0, R_0 >$, with $E_0$ the original set of equations (axioms) and $R_0$ a (usually empty) set of reduction rules.

Apply the inference rules to get

$$< E_0, R_0 >, < E_1, R_1 >, ..., < E_f, R_f > .$$

The goal is to reach $E_f = \emptyset$ and $R_f$ satisfying the completion property:

- Equation $s \leftrightarrow t$ can be derived from equations $E_0$ and reduction rules $R_0$ if and only if $s$ and $t$ can be reduced to the same normal form by the reduction rules $R_f$.

# Example

Initial axioms:

$$\begin{array}{lcl} g^5(x) & \leftrightarrow & x \\ g^3(x) & \leftrightarrow & x \end{array}$$

Assume $g^m(x) \succ g^n(x)$ for any $m > n$.

| | | | |
|---|---|---|---|
| 0 | $g^5(x) \leftrightarrow x$ | $\emptyset$ | |
| | $g^3(x) \leftrightarrow x$ | | |
| 1 | $g^5(x) \leftrightarrow x$ | $g^3(x) \rightarrow x$ | Orient |
| 2 | $g^2(x) \leftrightarrow x$ | $g^3(x) \rightarrow x$ | Simplify |
| 3 | $\emptyset$ | $g^2(x) \rightarrow x$ | Orient |
| | | $g^3(x) \rightarrow x$ | |
| 4 | $g(x) \leftrightarrow x$ | $g^2(x) \rightarrow x$ | Collapse |
| 5 | $\emptyset$ | $g(x) \rightarrow x$ | Orient |
| | | $g^2(x) \rightarrow x$ | |
| 6 | $g(x) \leftrightarrow x$ | $g(x) \rightarrow x$ | Collapse |
| 7 | $x \leftrightarrow x$ | $g(x) \rightarrow x$ | Simplify |
| 8 | $\emptyset$ | $g(x) \rightarrow x$ | Delete |

# Precedence for Expressions

For an expression $s$, let $n(x, s)$ denote the number of occurrences of $x$ in $s$.

Let $w(s)$ be the sum of $n(e, s)$ and all $n(x, s)$ for variables $x$ in $s$. Ex: If $s = (x \cdot e) \cdot (x \cdot y)$, then

$$w(s) = n(e, s) + n(x, s) + n(y, s) = 1 + 2 + 1 = 4.$$

Define $s \succ t$ if either:

1. $w(s) > w(t)$ and $n(x, s) \geq n(x, t)$ for each variable $x$ in $t$.
2. $w(s) = w(t)$ and $n(s, x) = n(t, x)$ for each variable $x$ in either $s$ or $t$, and $s \succ t$ matches one of:

$x^{-n} \succ x$ for any $x$ and $n \geq 1$.
$e^{-n} \succ e$ for $n \geq 1$.
$(a_1 \cdot a_2)^{-1} \succ b_1 \cdot b_2$.
$a^{-1} \succ b^{-1}$ if $a \succ b$.
$a_1 \cdot a_2 \succ b_1 \cdot b_2$
if $a_1 \succ b_1$ or $a_1 = b_1$ and $a_2 \succ b_2$.

Example: By (1), $(x \cdot y) \succ x$, so by (2), $(x \cdot y) \cdot z \succ x \cdot (y \cdot z)$.

# Examples

(1) Axioms for a group, where $e$ is the identity:

$$\begin{aligned} e \cdot x &\leftrightarrow x \\ x^{-1} \cdot x &\leftrightarrow e \\ (x \cdot y) \cdot z &\leftrightarrow x \cdot (y \cdot z). \end{aligned}$$

(2) Single axiom: $(w, (x, y, w), z) \rightarrow y$.