Theory of Computation

(Turing-Complete Systems)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

January 24, 2021

Contents

Contents

Turing-Complete Systems

Models more powerful than TM's

Problem

- Are there models of computation more powerful than Turing machines?

Models more powerful than TM's

Problem

- Are there models of computation more powerful than Turing machines?

Solution

- Nobody knows if there are more powerful models.
- However, there are many computational models equivalent in power to TM's. They are called Turing-complete systems.

Problem

- How do you prove the functional equivalence of two given computation models M_1 and M_2, i.e., $M_1 \Leftrightarrow M_2$?

Models more powerful than TM's

Problem

- Are there models of computation more powerful than Turing machines?

Solution

- Nobody knows if there are more powerful models.
- However, there are many computational models equivalent in power to TM's. They are called Turing-complete systems.

Problem

- How do you prove the functional equivalence of two given computation models M_1 and M_2, i.e., $M_1 \Leftrightarrow M_2$?

Solution

- Simulation!
- Simulate M_1 from M_2. Simulate M_2 from M_1.

Turing-complete systems

Variants of TM's

- TM's with a two-way infinite tape
- TM's with multiple heads
- TM's with a multidimensional tape
- TM's with multiple tapes
- TM's with random access memory
- TM's with nondeterminism
- TM's with stacks
- TM's with queues
- TM's with counters

- None of these variants are more powerful than a TM.

More Turing-complete systems

Systems

- Modern computers (assuming ∞ memory)
- Church's lambda calculus.
- Gödel's μ-recursive functions (building computable functions).
- Post's tag systems aka Post machines (NFA + FIFO queue)
- Post production systems (has grammar-like rules)
- Unrestricted grammars (generalization of CFG's).
- Markov algorithms.
- Conway's Game of Life.
- One dimensional cellular automata.
- Theoretical models of DNA-based computing.
- Lindenmayer systems or L-systems.
- While programs.

Unrestricted Grammars

What is an unrestricted grammar (UG)?

- Grammar = A set of rules for a language
- Unrestricted = No restrictions/constraints on production rules

What is an unrestricted grammar (UG)?

- Grammar = A set of rules for a language
- Unrestricted = No restrictions/constraints on production rules

Definition

An unrestricted grammar (UG) M is a 4-tuple
$G = (N, \Sigma, S, P)$, where,

1. N: A finite set (set of nonterminals/variables).
2. Σ: A finite set (alphabet).
3. P : A finite set of productions/rules of the form $\alpha \rightarrow \beta$, $\alpha, \beta \in (N \cup \Sigma)^*$ and α contains at least one nonterminal.

   ▷ Time (computation)

   ▷ Space (computer memory)
4. S: The start nonterminal (belongs to N).

Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^{2^n} \mid n \geq 0\}$

Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^{2^n} \mid n \geq 0\}$

Solution

- $S \rightarrow LaR$
  $L \rightarrow LD$
  $Da \rightarrow aaD$ $\quad \triangleright$ D acts as a doubling operator
  $DR \rightarrow R$
  $L \rightarrow \epsilon$
  $R \rightarrow \epsilon$

- Can you derive the string a from the grammar?

Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Solution (continued)

- Grammar:

$$S \to LaR$$
$$L \to LD$$
$$Da \to aaD$$
$$DR \to R$$
$$L \to \epsilon$$
$$R \to \epsilon$$

- Recognizing a:

$S \Rightarrow LaR$
$\Rightarrow aR$
$\Rightarrow a$

- Can you derive the string aa from the grammar?

Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Solution (continued)

- Grammar:

$$S \to LaR$$
$$L \to LD$$
$$Da \to aaD$$
$$DR \to R$$
$$L \to \epsilon$$
$$R \to \epsilon$$

- Recognizing aa:

$S \Rightarrow LaR$
$\Rightarrow LDaR$
$\Rightarrow LaaDR$
$\Rightarrow LaaR$
$\Rightarrow aaR$
$\Rightarrow aa$

- Can you derive the string $aaaa$ from the grammar?

Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Solution (continued)

- Grammar:

$$S \rightarrow LaR$$
$$L \rightarrow LD$$
$$Da \rightarrow aaD$$
$$DR \rightarrow R$$
$$L \rightarrow \epsilon$$
$$R \rightarrow \epsilon$$

- Recognizing $aaaa$:

$$S \Rightarrow LaR$$
$$\Rightarrow LDaR$$
$$\Rightarrow LDDaR$$
$$\Rightarrow LDaaDR$$
$$\Rightarrow LaaDaDR$$
$$\Rightarrow LaaaaDDR$$
$$\Rightarrow LaaaaDR$$
$$\Rightarrow LaaaaR$$
$$\Rightarrow aaaaR$$
$$\Rightarrow aaaa$$

- Can you derive the string $aaaaaaaa$ from the grammar?

Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Solution (continued)

- Grammar:

$$S \to LaR$$
$$L \to LD$$
$$Da \to aaD$$
$$DR \to R$$
$$L \to \epsilon$$
$$R \to \epsilon$$

- Recognizing $aaaaaaaa$:

$$S \Rightarrow LaR$$
$$\Rightarrow LDaR$$
$$\Rightarrow LDDaR$$
$$\Rightarrow LDDDaR$$
$$\Rightarrow LDDaaDR$$
$$\Rightarrow LDaaDaDR$$
$$\Rightarrow LDaaaaDDR$$
$$\Rightarrow LaaDaaaDDR$$
$$\Rightarrow LaaaaDaaDDR$$
$$\Rightarrow LaaaaaaDaDDR$$
$$\Rightarrow LaaaaaaaaDDDR$$
$$\Rightarrow LaaaaaaaaDDR$$
$$\Rightarrow LaaaaaaaaDR$$
$$\Rightarrow LaaaaaaaaR$$
$$\Rightarrow aaaaaaaaR$$
$$\Rightarrow aaaaaaaa$$

- Can you identify the generic technique in deriving the string a^{2^k} from the grammar?

Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^{2^n} \mid n \geq 0\}$

Solution (continued)

- Recognizing a^{2^k}:

  $S \Rightarrow^* LaR$

  $\Rightarrow^* LD^k aR$

  $\Rightarrow^* La^{2^k} D^k R$ ▷ Most important step

  $\Rightarrow^* La^{2^k} R$

  $\Rightarrow^* a^{2^k} R$

  $\Rightarrow^* a^{2^k}$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 0\}$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 0\}$

Solution

- $S \rightarrow ABCS$
  $S \rightarrow T_c$
  $T_c \rightarrow T_b$
  $T_b \rightarrow T_a$
  $T_a \rightarrow \epsilon$
  $CA \rightarrow AC$
  $BA \rightarrow AB$
  $CB \rightarrow BC$
  $CT_c \rightarrow T_c c$
  $BT_b \rightarrow T_b b$
  $AT_a \rightarrow T_a a$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 0\}$

Solution (continued)

- Recognizing abc:

$S \Rightarrow ABCS$
$\Rightarrow ABCT_c \qquad (\because S \rightarrow T_c)$
$\Rightarrow ABT_cc \qquad (\because CT_c \rightarrow T_cc)$
$\Rightarrow ABT_bc \qquad (\because T_c \rightarrow T_b)$
$\Rightarrow AT_bbc \qquad (\because BT_b \rightarrow T_bb)$
$\Rightarrow AT_abc \qquad (\because T_b \rightarrow T_a)$
$\Rightarrow T_aabc \qquad (\because AT_a \rightarrow T_aa)$
$\Rightarrow abc \qquad (\because T_a \rightarrow \epsilon)$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 0\}$

Solution (continued)

- Recognizing $aabbcc$:

$S \Rightarrow ABCS$
$\Rightarrow ABCABCS$
$\Rightarrow ABACBCS$
$\Rightarrow AABCBCS$
$\Rightarrow AABBCCS$
$\Rightarrow AABBCCT_c$
$\Rightarrow AABBCT_cc$
$\Rightarrow AABBT_ccc$
$\Rightarrow AABBT_bcc$
$\Rightarrow AABT_bbcc$
$\Rightarrow AAT_bbbcc$
$\Rightarrow AAT_abbcc$
$\Rightarrow AT_aabbcc$
$\Rightarrow T_aaabbcc$
$\Rightarrow aabbcc$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 0\}$

Solution (continued)

- Recognizing $aaabbbccc$:

$S \Rightarrow ABCS$
$\Rightarrow ABCABCS$
$\Rightarrow ABCABCABCS$
$\Rightarrow ABACBCABCS$
$\Rightarrow AABCBCABCS$
$\Rightarrow AABCBACBCS$
$\Rightarrow AABCABCBCS$
$\Rightarrow AABACBCBCS$
$\Rightarrow AAABCBCBCS$
$\Rightarrow AAABBCCBCS$
$\Rightarrow AAABBCBCCS$
$\Rightarrow AAABBBCCCS$
$\Rightarrow AAABBBCCCT_c$
$\Rightarrow AAABBBCCT_c c$
$\Rightarrow AAABBBCT_c cc$
$\Rightarrow AAABBBT_c ccc$
$\Rightarrow AAABBBT_b ccc$
$\Rightarrow AAABBT_b bccc$
$\Rightarrow AAABT_b bbccc$
$\Rightarrow AAAT_b bbbccc$
$\Rightarrow AAAT_a bbbccc$
$\Rightarrow AAT_a abbbccc$
$\Rightarrow AT_a aabbbccc$
$\Rightarrow T_a aaabbbccc$
$\Rightarrow aaabbbccc$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 0\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 0\}$

Solution (continued)

- Recognizing $a^k b^k c^k$:

$S \Rightarrow ABCS$
$\Rightarrow^* (ABC)^k S$
$\Rightarrow^* A^k B^k C^k S$ ▷ Toughest step
$\Rightarrow^* A^k B^k C^k T_c$
$\Rightarrow^* A^k B^k T_c c^k$
$\Rightarrow^* A^k B^k T_b c^k$
$\Rightarrow^* A^k T_b b^k c^k$
$\Rightarrow^* A^k T_a b^k c^k$
$\Rightarrow^* T_a a^k b^k c^k$
$\Rightarrow^* a^k b^k c^k$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 1\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 1\}$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 1\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 1\}$

Solution

- $S \to SABC$
  $S \to LABC$
  $BA \to AB$
  $CB \to BC$
  $CA \to AC$
  $LA \to a$
  $aA \to aa$
  $aB \to ab$
  $bB \to bb$
  $bC \to bc$
  $cC \to cc$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 1\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 1\}$

Solution (continued)

- Recognizing abc: $S \Rightarrow LABC \Rightarrow aBC \Rightarrow abC \Rightarrow abc$
- Recognizing $aabbcc$:

$$
\begin{aligned}
S &\Rightarrow SABC \\
&\Rightarrow LABCABC \\
&\Rightarrow LABACBC \\
&\Rightarrow LABABCC \\
&\Rightarrow LAABBCC \\
&\Rightarrow aABBCC
\end{aligned}
\qquad
\begin{aligned}
&\Rightarrow aaBBCC \\
&\Rightarrow aabBCC \\
&\Rightarrow aabbCC \\
&\Rightarrow aabbcC \\
&\Rightarrow aabbcc
\end{aligned}
$$

Construct an UG for $L = \{a^n b^n c^n \mid n \geq 1\}$

Problem

- Construct an UG that accepts all strings from the language $L = \{a^n b^n c^n \mid n \geq 1\}$

Solution (continued)

- Recognizing $a^k b^k c^k$:
  $S \Rightarrow SABC$
  $\Rightarrow^* S(ABC)^{k-1}$
  $\Rightarrow^* L(ABC)^k$
  $\Rightarrow^* LA^k B^k C^k$ $\quad\triangleright$ Toughest step
  $\Rightarrow^* a^k B^k C^k$
  $\Rightarrow^* a^k b^k C^k$
  $\Rightarrow^* a^k b^k c^k$

Lindenmayer Systems

What is an L-system?

Definition

A Lindenmayer system (L-system) is a 4-tuple
$L = (V, C, S, R)$, where,

1. V: A finite set (set of variables).
2. C: A finite set of constants.
3. S: The starting string (belongs to $(V \cup C)^*$), aka axiom.
4. R : A finite set of rules of the form $\alpha \to \beta$,
   $\alpha, \beta \in (V \cup C)^*$ and α contains at least one variable.

$\triangleright$ Time (computation) and Space (computer memory)

What is an L-system?

Definition

A Lindenmayer system (L-system) is a 4-tuple
$L = (V, C, S, R)$, where,

1. V: A finite set (set of variables).
2. C: A finite set of constants.
3. S: The starting string (belongs to $(V \cup C)^*$), aka axiom.
4. R : A finite set of rules of the form $\alpha \to \beta$, $\alpha, \beta \in (V \cup C)^*$ and α contains at least one variable.

$\triangleright$ Time (computation) and Space (computer memory)

Difference

A Lindenmayer system (L-system) differs from an unrestricted grammar in three major ways:

1. You apply all rules in parallel or simultaneously.
2. You start with a starting string.
3. All strings produced are in the language.

What are the applications of L-systems?

Applications

- Generate self-similar fractals.
- Model the growth processes of a variety of organisms (e.g.: plants, algae, etc).
- Compose music, predict protein folding, and design buildings.

Source: Wikipedia

Example: Rabbit population

Problem

- Construct an L-system to model rabbit population.

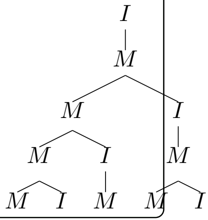
Example: Rabbit population

Problem

- Construct an L-system to model rabbit population.

Solution

- Variables $= \{I, M\}$, Terminals $= \phi$,
  Start $= I$, Rules $= \{I \to M, M \to MI\}$.
  ($I =$ immature, $M =$ mature) rabbit pair.
- $n = 0$: I
  $n = 1$: M
  $n = 2$: MI
  $n = 3$: MIM
  $n = 4$: $MIMMI$
- Lengths of strings:
  $1, 1, 2, 3, 5, \ldots$ Fibonacci sequence



Example: Sierpinksi triangle

Problem

- Construct an L-system to draw a Sierpinksi triangle.

Example: Sierpinksi triangle

Problem

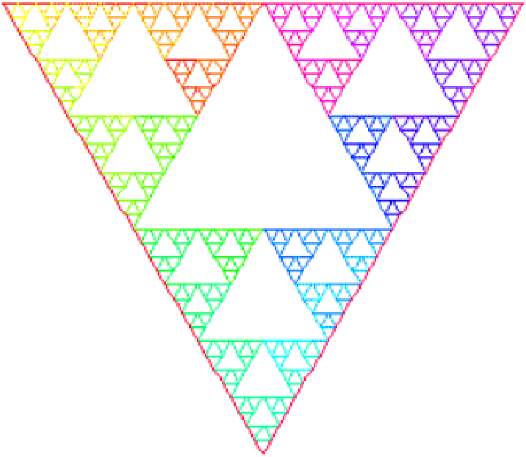
- Construct an L-system to draw a Sierpinksi triangle.

Solution

- L-system.
  Variables $= \{A, B\}$.
  Terminals $= \{+, -\}$.
  Starting string $= ABA - -AA - -AA$.
  Rules $= \{A \to AA, B \to - - ABA + +ABA + +ABA - -\}$.
- Meaning.
  $A, B =$ go forward a unit length.
  $+ =$ turn left by 60°.
  $- =$ turn right by 60°.

Example: Sierpinksi triangle

Solution (continued)



Source: Robert M. Dickau

Example: Trees

Problem

- Construct an L-system to draw a tree.

Example: Trees

Problem

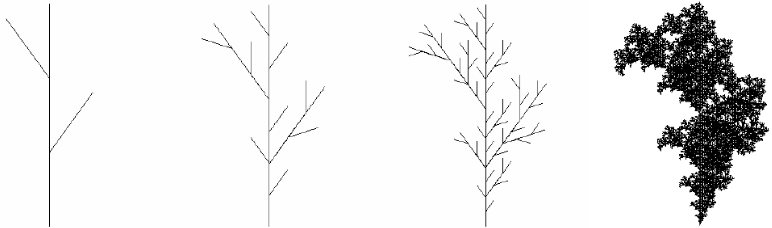
- Construct an L-system to draw a tree.

Solution

- L-system.
  Variables $= \{F\}$. Terminals $= \{+, -, [,]\}$.
  Start $= F$. Rules $= \{F \to F[-F]F[+F][F]\}$.
- Meaning.
  $F =$ go forward a unit length.
  $+ =$ turn left by 36°. $- =$ turn right by 36°.
  $[\;=$ push the current pen position and direction onto the stack.
  $] =$ pop the top pen position/direction off the stack, lift up the pen, move it to the position that is now on the top of the stack, put it back down, and set its direction to the one on the top of the stack.

Example: Trees

Solution (continued)



Source: Elaine Rich's Automata, Computability and Complexity: Theory and Applications.

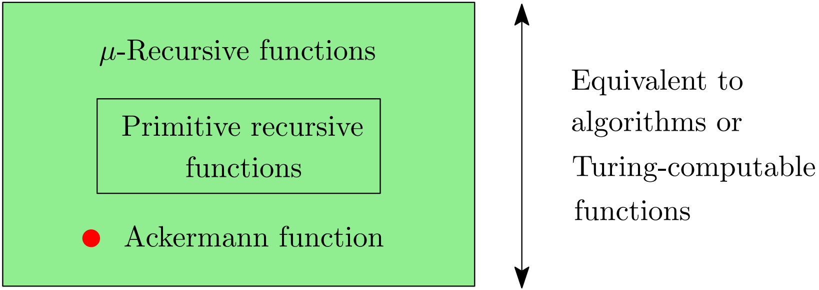
Gödel's μ-Recursive Functions

What are computable functions?

Concept

- Computable functions are comparable to algorithms.
- Gödel developed primitive recursive functions to model all computable functions.
- Ackermann showed a computable function that was not primitive recursive.
- Gödel expanded his definition and developed μ-recursive functions to model all computable functions.
- Gödel's μ-recursive functions are computationally equivalent to algorithms or Turing-computable functions.

What are μ-recursive functions?



What are primitive recursive functions?

Definition

The primitive recursive functions are the smallest class of functions from $\mathbb{W} \times \mathbb{W} \times \cdots \times \mathbb{W}$ to $\mathbb{W}$ that includes:

1. zero function
2. successor function
3. projection function

and that is closed under the operations:

4. composition of functions
5. primitive recursion $\triangleright$ for loop

Examples

- Arithmetic operations, logical operations, several mathematical functions (such as factorial, combination, etc), and so on.

Zero function ($\mathbb{W}^k \to \mathbb{W}$)

Definition

- The k-ary zero function for any $k \in \mathbb{W}$ is defined as zero$_k(X) = 0$, where $X = (n_1, n_2, \ldots, n_k)$ for all $n_1, n_2, \ldots, n_k \in \mathbb{W}$

Examples

- zero$_0() = 0$
- zero$_1(n) = 0$
- zero$_2(n_1, n_2) = 0$
- zero$_{100}(n_1, n_2, \ldots, n_{100}) = 0$

Projection function ($\mathbb{W}^k \to \mathbb{W}$)

Definition

- The projection function for any $i, k \in \mathbb{N}$ and $i \leq k$ is defined as
  $\text{proj}_{k,i}(X) = n_i$, where $X = (n_1, n_2, \ldots, n_k)$
  for all $n_1, \ldots, n_k \in \mathbb{W}$

Examples

- proj for $k = 0$ is not defined
- $\text{proj}_{1,1}(n) = n$ ▷ identity function
- $\text{proj}_{2,1}(n_1, n_2) = n_1$
- $\text{proj}_{100,57}(n_1, n_2, \ldots, n_{100}) = n_{57}$

Successor function ($\mathbb{W} \to \mathbb{W}$)

Definition

- The successor function is defined as succ$(n) = n + 1$, for all $n \in \mathbb{W}$

Examples

- succ(-1) is not defined for negative numbers
- succ$(0) = 1$
- succ$(1) = 2$
- succ$(100) = 101$
- For what value of x we have succ$(x) = 0$?

Combining functions

Composition function ($\mathbb{W}^k \to \mathbb{W}$)

- The k-ary composition function of g and $h_1, h_2, \ldots, h_\ell$ for any $k, \ell \in \mathbb{W}$ is defined as

  $$\boxed{f(x) = g(h_1(X), h_2(X), \ldots, h_\ell(X))}$$

  where $X = (n_1, \ldots, n_k)$ and $n_1, \ldots, n_k \in \mathbb{W}$

Primitive recursion ($\mathbb{W}^{k+1} \to \mathbb{W}$)

- The $(k+1)$-ary function defined recursively by g and h for any $k, \ell \in \mathbb{W}$ is defined as

  $$\boxed{f(X, 0) = g(X)}$$

  $$\boxed{f(X, m+1) = h(f(X, m), X, m)}$$

  where $X = (n_1, \ldots, n_k)$ and $n_1, \ldots, n_k, m \in \mathbb{W}$

Primitive recursive functions

Examples

- **Constant.** $\triangleright$ constant

  $3 = \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}(m))))$

  $k = \underbrace{\mathsf{succ}(\cdots(\mathsf{succ}(}_{k \text{ times}}\mathsf{zero}(m))\cdots)$

- **Addition.** $\triangleright$ $\mathsf{add}(m,n) = m+n$

  $\mathsf{add}(m,0) = m$

  $\mathsf{add}(m,n+1) = \mathsf{succ}(\mathsf{add}(m,n))$

- **Multiplication.** $\triangleright$ $\mathsf{mult}(m,n) = m \times n$

  $\mathsf{mult}(m,0) = \mathsf{zero}(m)$

  $\mathsf{mult}(m,n+1) = \mathsf{plus}(\mathsf{mult}(m,n),m)$

- **Exponentiation.** $\triangleright$ $\mathsf{pow}(m,n) = m^n$

  $\mathsf{pow}(m,0) = \mathsf{succ}(\mathsf{zero}(m))$

  $\mathsf{pow}(m,n+1) = \mathsf{mult}(\mathsf{pow}(m,n),m)$

- **Predecessor.** $\triangleright$ $\mathsf{pred}(n) = \mathsf{max}(n-1,0)$

  $\mathsf{pred}(0) = 0$

  $\mathsf{pred}(n+1) = n$

Primitive recursive functions

Examples

- Nonnegative subtraction. $\triangleright$ $\mathsf{sub}(m,n) = \mathsf{max}(m-n,0)$
  $\mathsf{sub}(m,0) = m$
  $\mathsf{sub}(m,n+1) = \mathsf{pred}(\mathsf{sub}(m,n))$
- Sign. $\triangleright$ $\mathsf{sign}(n) = 0$ if $n = 0$, 1 if $n > 0$
  $\mathsf{sign}(0) = 0$
  $\mathsf{sign}(n+1) = \mathsf{succ}(\mathsf{zero}(n))$
- Positive.
  $\mathsf{positive}(n) = \mathsf{sign}(n)$
- IsZero. $\triangleright$ $\mathsf{iszero}(n) = 1$ if $n = 0$, 1 otherwise
  $\mathsf{iszero}(0) = 1$
  $\mathsf{iszero}(n+1) = 0$
- IsOne. $\triangleright$ $\mathsf{isone}(n) = 1$ if $n = 1$, 0 otherwise
  $\mathsf{isone}(0) = 0$
  $\mathsf{isone}(n+1) = \mathsf{iszero}(n)$

Primitive recursive functions

Examples

- Greater than or equal to. $\triangleright$ $\mathsf{ge}(m,n) = 1$ if $m \geq n$

  $\mathsf{ge}(m,n) = \mathsf{iszero}(\mathsf{sub}(n,m))$
- Negation. $\triangleright$ $\mathsf{neg}(p) = \sim p$

  $\mathsf{neg}(p(m,n)) = \mathsf{sub}(1, p(m,n))$
- Disjunction. $\triangleright$ $\mathsf{or}(p,q) = p \vee q$

  $\mathsf{or}(p(m,n), q(m,n)) = \mathsf{sub}(1, \mathsf{iszero}(\mathsf{add}(p(m,n), q(m,n))))$
- Conjunction. $\triangleright$ $\mathsf{and}(p,q) = p \wedge q$

  $\mathsf{and}(p(m,n), q(m,n)) = \mathsf{sub}(1, \mathsf{iszero}(\mathsf{mult}(p(m,n), q(m,n))))$
- How do you define $\mathsf{le}(m,n)$, $\mathsf{gt}(m,n)$, $\mathsf{lt}(m,n)$, and $\mathsf{eq}(m,n)$?

Primitive recursive functions

Examples

- Function defined by cases.

$$f(x) = \begin{cases} g(x) & \text{if } p(x), \\ h(x) & \text{if } \sim p(x). \end{cases}$$

where $x = (n_1, n_2, \ldots, n_k)$ and $n_1, n_2, \ldots, n_k \in \mathbb{W}$

$f(x) = p(x) \cdot g(x) + (1 - p(x)) \cdot h(x)$

i.e., $f(x) = \text{add}(\text{mult}(p(x), g(x)), \text{mult}(\text{sub}(1, p(x)), h(x)))$

- Remainder. $\triangleright$ $\text{rem}(m, n) = m\%n$

$\text{rem}(0, n) = 0$

$$\text{rem}(m+1, n) = \begin{cases} 0 & \text{if eq}(\text{rem}(m, n), \text{pred}(n)), \\ \text{rem}(m, n) + 1 & \text{otherwise.} \end{cases}$$

- Integer quotient. $\triangleright$ $\text{div}(m, n) = \text{floor}(m/n)$

$\text{div}(0, n) = 0$

$$\text{div}(m+1, n) = \begin{cases} \text{div}(m, n) + 1 & \text{if eq}(\text{rem}(m, n), \text{pred}(n)), \\ \text{div}(m, n) & \text{otherwise.} \end{cases}$$

Primitive recursive functions

Examples

- Digit.
  $\mathsf{digit}(m, p, n) = \mathsf{div}(\mathsf{rem}(n, \mathsf{pow}(p, m)), \mathsf{pow}(p, \mathsf{sub}(m, 1)))$
- Series sum.
  $\mathsf{sum}_f(n, m) = f(n, 0) + f(n, 1) + \cdots + f(n, m)$
  If $f(n, m)$ is primitive recursive, so is $\mathsf{sum}_f(n, m)$

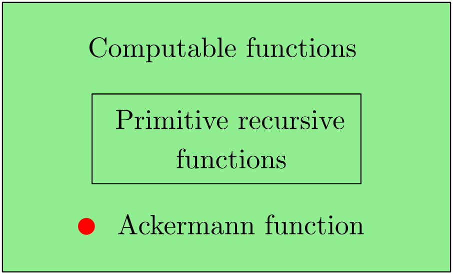
- Do primitive recursive functions represent all computable functions? No!
  There are computable functions that are not primitive recursive.

Ackermann function

Definition

- Ackermann function is the simplest example of an intuitively computable total function that is not primitive recursive.
- It is defined as:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ A(m-1,1) & \text{if } n = 0, \\ A(m-1, A(m,n-1)) & \text{otherwise.} \end{cases}$$



What are μ-recursive functions?

Definition

The μ-recursive functions are the smallest class of functions from $\mathbb{W} \times \mathbb{W} \times \cdots \times \mathbb{W}$ to $\mathbb{W}$ that includes:

1. zero function
2. successor function
3. projection function

and that is closed under the operations:

4. composition of functions
5. primitive recursion $\triangleright$ halting for-loop
6. minimalization of minimalizable functions $\triangleright$ halting while-loop

- μ-recursive functions are computationally equivalent to algorithms or Turing-computable functions.

What are minimizable functions?

Definition

- Let g be a $(k+1)$-ary function, for some $k \geq 0$. The minimalization of g is the k-ary function f defined as follows.

$$f(X) = \begin{cases} \text{least } m \in \mathbb{W} \text{ such that } g(X,m) = 1 & \text{if } m \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

$\textsc{TM-Min}(g, X)$ $\triangleright\ f(X)$

```
1. m ← 0
2. while g(X, m) ≠ 1 do
3.   m ← m + 1
4. return m
```

$\textsc{TM-Min}$ might not halt if no value of m exists.

What are minimizable functions?

Definition

- Let g be a $(k+1)$-ary function, for some $k \geq 0$. The minimalization of g is the k-ary function f defined as follows.

$$f(X) = \begin{cases} \text{least } m \in \mathbb{W} \text{ such that } g(X, m) = 1 & \text{if } m \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

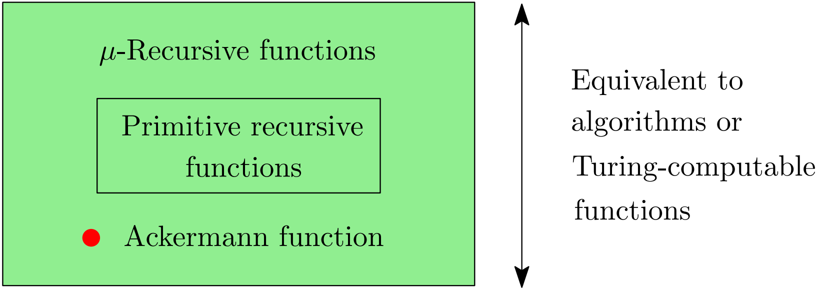
$\textsc{TM-Min}(g, X)$	$\triangleright\ f(X)$
1. $m \leftarrow 0$	
2. while $g(X, m) \neq 1$ do	
3. $\quad m \leftarrow m + 1$	
4. return m	

$\textsc{TM-Min}$ might not halt if no value of m exists.

- A function g is called minimalizable function iff for every X, there is an m such that $g(X, m) = 1$.
  A function g is minimalizable iff $\textsc{TM-Min}$ always halts.

(Primitive vs. μ) recursive functions

	Primitive rec. functions	μ-recursive functions
Comparable to	Halting for-loops	Halting while-loops
#Iterations	Known beforehand	Not known beforehand



While Programs

What are for and while programs?

Operations	For programs	While programs
Assignments e.g. $x \leftarrow y + 5$	✓	✓
Sequential compositions e.g. $p; q$	✓	✓
Conditionals e.g. if $(x < y)$ then p else q	✓	✓
For loops e.g. for y do p	✓	✓
While loops e.g. while $x < y$ do p	✗	✓

What are for and while programs?

Difference	For programs	While programs
Definition	For programs are computer programs without the while construct.	While programs are computer programs with the while construct.
#Iterations	Known beforehand. Does change after the execution of the loop body.	Might change after the execution of the loop body.
Halting	Always halt.	Might not halt.

Relationship with recursive functions

Time	Formal functions	Computer programs
Finite	Primitive rec. functions	For programs
	μ-recursive functions	Halting while programs
Infinite	Partially rec. functions	Non-halting while programs