## Contents

- Turing-Complete Systems
  - Unrestricted Grammars
  - Lindenmayer Systems
  - Gödel’s $\mu$-Recursive Functions
  - While Programs
Turing-Complete Systems
## 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

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Are there models of computation more powerful than Turing machines?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
</table>
| • 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**. |

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• How do you prove the functional equivalence of two given computation models $M_1$ and $M_2$, i.e., $M_1 \Leftrightarrow M_2$?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
</table>
| • Simulation!  
  • Simulate $M_1$ from $M_2$. Simulate $M_2$ from $M_1$. |
### 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

<table>
<thead>
<tr>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Modern computers (assuming $\infty$ memory)</td>
</tr>
<tr>
<td>• Church’s lambda calculus.</td>
</tr>
<tr>
<td>• Gödel’s $\mu$-recursive functions (building computable functions).</td>
</tr>
<tr>
<td>• Post’s tag systems aka Post machines (NFA + FIFO queue)</td>
</tr>
<tr>
<td>• Post production systems (has grammar-like rules)</td>
</tr>
<tr>
<td>• Unrestricted grammars (generalization of CFG’s).</td>
</tr>
<tr>
<td>• Markov algorithms.</td>
</tr>
<tr>
<td>• Conway’s Game of Life.</td>
</tr>
<tr>
<td>• One dimensional cellular automata.</td>
</tr>
<tr>
<td>• Theoretical models of DNA-based computing.</td>
</tr>
<tr>
<td>• Lindenmayer systems or L-systems.</td>
</tr>
<tr>
<td>• While programs.</td>
</tr>
</tbody>
</table>
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,

2. $\Sigma$: 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 $\alpha$ contains at least one nonterminal.
   - **Time (computation)**
   - **Space (computer memory)**
4. $S$: The **start nonterminal** (belongs to $N$).
Construct an UG for \( L = \{ a^{2^n} \mid n \geq 0 \} \)

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language ( L = { a^{2^n} \mid n \geq 0 } )</td>
</tr>
</tbody>
</table>
Construct an UG for \( L = \{ a^{2^n} \mid n \geq 0 \} \)

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language ( L = { a^{2^n} \mid n \geq 0 } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
</table>
| • \( S \to LaR \)  
  \( L \to LD \)  
  \( Da \to aaD \)  
  \( DR \to R \)  
  \( L \to \epsilon \)  
  \( R \to \epsilon \)  
  ▶ \( D \) acts as a doubling operator |

• Can you derive the string \( a \) from the grammar?
Construct an UG for \( L = \{a^{2^n} \mid n \geq 0\} \)

### Solution (continued)

- **Grammar:**
  
  \[
  \begin{align*}
  S & \rightarrow LaR \\
  L & \rightarrow LD \\
  Da & \rightarrow aaD \\
  DR & \rightarrow R \\
  L & \rightarrow \epsilon \\
  R & \rightarrow \epsilon
  \end{align*}
  \]

- **Recognizing \( a \):**
  
  \[
  \begin{align*}
  S & \Rightarrow LaR \\
  & \Rightarrow aR \\
  & \Rightarrow a
  \end{align*}
  \]

- Can you derive the string \( aa \) from the grammar?
Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Solution (continued)

- **Grammar:**
  
  $S \rightarrow LaR$
  
  $L \rightarrow LD$
  
  $Da \rightarrow aaD$

  $DR \rightarrow R$
  
  $L \rightarrow \epsilon$

  $R \rightarrow \epsilon$

- **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:**
  
  \[
  \begin{align*}
  S &\rightarrow LaR \\
  L &\rightarrow LD \\
  Da &\rightarrow aaD \\
  DR &\rightarrow R \\
  L &\rightarrow \epsilon \\
  R &\rightarrow \epsilon
  \end{align*}
  \]

- **Recognizing **aaa**a:**
  
  \[
  \begin{align*}
  S &\Rightarrow LaR \\
  &\Rightarrow LDaR \\
  &\Rightarrow LDDaR \\
  &\Rightarrow LDaaDR \\
  &\Rightarrow LaaDaDR \\
  &\Rightarrow LaaaDDR \\
  &\Rightarrow LaaaaDR \\
  &\Rightarrow LaaaaR \\
  &\Rightarrow aaaaaR \\
  &\Rightarrow aaaa
  \end{align*}
  \]

- **Can you derive the string **aaa**aa**aa**aa**aa**aa**aa**aa**aa**aa**aa** from the grammar?**
Construct an UG for $L = \{a^{2^n} \mid n \geq 0\}$

Solution (continued)

• Grammar:
  
  \begin{align*}
  S & \rightarrow LaR \\
  L & \rightarrow LD \\
  Da & \rightarrow aaD \\
  DR & \rightarrow R \\
  L & \rightarrow \epsilon \\
  R & \rightarrow \epsilon
  \end{align*}

• Recognizing $aaaaaaaaaa$:
  
  \begin{align*}
  S & \Rightarrow LaR \\
  \Rightarrow LDaR \\
  \Rightarrow LDDaR \\
  \Rightarrow LDDDaR \\
  \Rightarrow LDDDaR \\
  \Rightarrow LDDaaDR \\
  \Rightarrow LDaaDaDR \\
  \Rightarrow LDaaaaDDR \\
  \Rightarrow LaaDaaaDDR
  \end{align*}

• Can you identify the generic technique in deriving the string $a^{2^k}$ from the grammar?
**Construct an UG for** \( L = \{ a^{2^n} \mid n \geq 0 \} \)

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language ( L = { a^{2^n} \mid n \geq 0 } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution (continued)</th>
</tr>
</thead>
</table>
| • **Recognizing** \( a^{2^k} \):  
  \[ S \Rightarrow^* L a R \]  
  \[ \Rightarrow^* L D^k a R \]  
  \[ \Rightarrow^* L a^{2^k} D^k R \]  
  \[ \Rightarrow^* L a^{2^k} R \]  
  \[ \Rightarrow^* a^{2^k} R \]  
  \[ \Rightarrow^* a^{2^k} \]  
  ▶ Most important step |
Construct an UG for $L = \{a^n b^n c^n \mid n \geq 0\}$

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language $L = {a^n b^n c^n \mid n \geq 0}$</td>
</tr>
</tbody>
</table>
**Construct an UG for** \( L = \{a^n b^n c^n | n \geq 0\} \)

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language ( L = {a^n b^n c^n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
</table>
| • \( S \rightarrow ABCS \)  
  \( S \rightarrow T_c \)  
  \( T_c \rightarrow T_b \)  
  \( T_b \rightarrow T_a \)  
  \( T_a \rightarrow \epsilon \)  
  \( CA \rightarrow AC \)  
  \( BA \rightarrow AB \)  
  \( CB \rightarrow BC \)  
  \( CT_c \rightarrow T_c c \)  
  \( BT_b \rightarrow T_b b \)  
  \( AT_a \rightarrow T_a a \) |
Construct an UG for \( L = \{a^n b^n c^n \mid n \geq 0\} \)

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language ( L = {a^n b^n c^n \mid n \geq 0} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Recognizing ( abc ):</td>
</tr>
</tbody>
</table>
| \( S \Rightarrow ABCS \) \( \Rightarrow ABCTc \) (\( \because S \rightarrow T_c \)) \( \Rightarrow ABTc_{}c \) (\( \because CT_c \rightarrow T_c c \)) \( \Rightarrow ABTbc \) (\( \because T_c \rightarrow T_b \)) \( \Rightarrow AT_bbc \) (\( \because BT_b \rightarrow T_b b \)) \( \Rightarrow AT_{}abc \) (\( \because T_b \rightarrow T_a \)) \( \Rightarrow T_{}abc \) (\( \because AT_a \rightarrow T_a a \)) \( \Rightarrow abc \) (\( \because T_a \rightarrow \epsilon \))
Construct an UG for \( L = \{a^n b^n c^n \mid n \geq 0 \} \)

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language ( L = {a^n b^n c^n \mid n \geq 0 } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Recognizing</strong> ( aabbcc ):</td>
</tr>
<tr>
<td>( S \Rightarrow ABCS ) &amp; ( \Rightarrow AABBT_bcc )</td>
</tr>
<tr>
<td>( \Rightarrow ABCABCS ) &amp; ( \Rightarrow AABT_bbcc )</td>
</tr>
<tr>
<td>( \Rightarrow ABACBCS ) &amp; ( \Rightarrow AAT_bbbcc )</td>
</tr>
<tr>
<td>( \Rightarrow AABCBCS ) &amp; ( \Rightarrow AAT_aabbcc )</td>
</tr>
<tr>
<td>( \Rightarrow AABBCCS ) &amp; ( \Rightarrow AT_aabbcc )</td>
</tr>
<tr>
<td>( \Rightarrow AABBCCT_c ) &amp; ( \Rightarrow T_aabbcc )</td>
</tr>
<tr>
<td>( \Rightarrow AABBCT_ccc ) &amp; ( \Rightarrow aabbcc )</td>
</tr>
<tr>
<td>( \Rightarrow AABBT_{c}ccc )</td>
</tr>
</tbody>
</table>
### Construct an UG for $L = \{a^n b^n c^n \mid n \geq 0\}$

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language $L = {a^n b^n c^n \mid n \geq 0}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Recognizing</strong> $aaabbbcccc$:</td>
</tr>
</tbody>
</table>
| \[ S \Rightarrow ABCS \]
| \[ \Rightarrow ABCABCS \]
| \[ \Rightarrow ABCABCA BCS \]
| \[ \Rightarrow ABACBCABCS \]
| \[ \Rightarrow AABCBCABCS \]
| \[ \Rightarrow AABCBACBCS \]
| \[ \Rightarrow AABCABCBS \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow AAABBBCT_{ccc} \]
| \[ \Rightarrow aaabbbcccc \] |
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 $a^k b^k c^k$:**
  
  \[
  S \Rightarrow ABCS \\
  \Rightarrow^* (ABC)^k S \\
  \Rightarrow^* A^k B^k C^k S \\
  \Rightarrow^* A^k B^k C^k T_c \\
  \Rightarrow^* A^k B^k T_c c^k \\
  \Rightarrow^* A^k B^k T_b c^k \\
  \Rightarrow^* A^k T_b b^k c^k \\
  \Rightarrow^* A^k T_a b^k c^k \\
  \Rightarrow^* T_a a^k b^k c^k \\
  \Rightarrow^* a^k b^k c^k
  \]

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

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language $L = { a^n b^n c^n \mid n \geq 1 }$</td>
</tr>
</tbody>
</table>
Problem

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

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 $L = \{a^n b^n c^n \mid n \geq 1\}$

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an UG that accepts all strings from the language $L = {a^n b^n c^n \mid n \geq 1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Recognizing $abc$: $S \Rightarrow LABC \Rightarrow aBC \Rightarrow abC \Rightarrow abc$</td>
</tr>
</tbody>
</table>
| • Recognizing $aabccc$:

```
S \Rightarrow SABC
⇒ LABCABC
⇒ LABACBC
⇒ LABABCC
⇒ LAABBCC
⇒ aABBCC
⇒ aaBBCC
⇒ aabBCC
⇒ aabbCC
⇒ aabbcC
⇒ aabbcC
⇒ aabbcc
```
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** \( a^k b^k c^k \):

\[
\begin{align*}
S & \Rightarrow SABC \\
\Rightarrow & \ast S(ABC)^{k-1} \\
\Rightarrow & \ast L(ABC)^k \\
\Rightarrow & \ast L A^k B^k C^k \\
\Rightarrow & \ast a^k B^k C^k \\
\Rightarrow & \ast a^k b^k C^k \\
\Rightarrow & \ast a^k b^k c^k
\end{align*}
\]

▷ Toughest step
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 $\alpha \rightarrow \beta$,  
   $\alpha, \beta \in (V \cup C)^*$ and $\alpha$ contains at least one variable.

- **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 \( \alpha \rightarrow \beta \), \( \alpha, \beta \in (V \cup C)^* \) and \( \alpha \) 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?

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Generate <strong>self-similar fractals</strong>.</td>
</tr>
<tr>
<td>• Model the <strong>growth processes</strong> of a variety of organisms (e.g.: plants, algae, etc).</td>
</tr>
<tr>
<td>• Compose music, predict protein folding, and design buildings.</td>
</tr>
</tbody>
</table>

Example: Rabbit population

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an L-system to model rabbit population.</td>
</tr>
</tbody>
</table>

Variables:
\[ I, M \]

Terminals:
\[ \phi \]

Start:
\[ I \]

Rules:
\[ I \rightarrow M, M \rightarrow MI \]

\( n = 0: I \)
\( n = 1: M \)
\( n = 2: MI \)
\( n = 3: MIM \)
\( n = 4: MIMMI \)

Lengths of strings:
1, 1, 2, 3, 5, ...

Fibonacci sequence.
Example: Rabbit population

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an L-system to model rabbit population.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
</table>
| • Variables = \{I, M\}, Terminals = \(\phi\),  
    Start = \(I\), Rules = \{I \rightarrow M, M \rightarrow MI\}.  
    \((I = \text{immature}, M = \text{mature})\) 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: Sierpinski triangle

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an L-system to draw a Sierpinski triangle.</td>
</tr>
</tbody>
</table>
### Example: Sierpinski triangle

**Problem**

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

**Solution**

- **L-system.**
  - Variables = \{A, B\}.
  - Terminals = \{+, −\}.
  - Starting string = $ABA − − AA − − AA$.
  - Rules = \{A → AA, B → − − ABA + + ABA + + ABA − − \}.

- **Meaning.**
  - A, B = go forward a unit length.
  - + = turn left by 60°.
  - − = turn right by 60°.
Example: Sierpinski triangle

Solution (continued)

Source: Robert M. Dickau
### Problem

- Construct an L-system to draw a tree.
Example: Trees

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Construct an L-system to draw a tree.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• L-system.</td>
</tr>
<tr>
<td>Variables = {F}. Terminals = {+, −, [, ,]}.</td>
</tr>
<tr>
<td>Start = F. Rules = {F → F[−F]F[+F][F]}.</td>
</tr>
<tr>
<td>• Meaning.</td>
</tr>
<tr>
<td>F = go forward a unit length.</td>
</tr>
<tr>
<td>+ = turn left by 36°. − = turn right by 36°.</td>
</tr>
<tr>
<td>[ = push the current pen position and direction onto the stack.</td>
</tr>
<tr>
<td>] = 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.</td>
</tr>
</tbody>
</table>
Example: Trees

Solution (continued)

Gödel’s $\mu$-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 $\mu$-recursive functions?

$\mu$-Recursive functions

- Primitive recursive functions
  - Ackermann function

Equivalent to Turing-computable algorithms or 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

**Examples**

- Arithmetic operations, logical operations, several mathematical functions (such as factorial, combination, etc), and so on.
Zero function \((\mathbb{W}^k \rightarrow \mathbb{W})\)

**Definition**

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

**Examples**

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

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

Examples

- proj for $k = 0$ is not defined
- \( \text{proj}_{1,1}(n) = n \)
- \( \text{proj}_{2,1}(n_1, n_2) = n_1 \)
- \( \text{proj}_{100,57}(n_1, n_2, \ldots, n_{100}) = n_{57} \)
The successor function is defined as 
\[ \text{succ}(n) = n + 1, \text{ for all } n \in \mathbb{W} \]

Examples
- \( \text{succ}(-1) \) is not defined for negative numbers
- \( \text{succ}(0) = 1 \)
- \( \text{succ}(1) = 2 \)
- \( \text{succ}(100) = 101 \)
- For what value of \( x \) we have \( \text{succ}(x) = 0? \)
### Combining functions

#### Composition function \((\mathbb{W}^k \rightarrow \mathbb{W})\)

- 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

\[
f(x) = g(h_1(X), h_2(X), \ldots, h_\ell(X))
\]

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

#### Primitive recursion \((\mathbb{W}^{k+1} \rightarrow \mathbb{W})\)

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

\[
f(X, 0) = g(X)
\]

\[
f(X, m + 1) = h(f(X, m), X, m)
\]

where \(X = (n_1, \ldots, n_k)\) and \(n_1, \ldots, n_k, m \in \mathbb{W}\)
## Primitive recursive functions

### Examples

- **Constant.**
  
  \[
  3 = \text{succ}(\text{succ}(\text{succ}(\text{zero}(m))))
  \]
  \[
  k = \text{succ}(\cdots (\text{succ}(\text{zero}(m)) \cdots )
  \]
  \[\text{for } k \text{ times}\]

- **Addition.**
  
  \[
  \text{add}(m, 0) = m
  \]
  \[
  \text{add}(m, n + 1) = \text{succ}(\text{add}(m, n))
  \]

- **Multiplication.**
  
  \[
  \text{mult}(m, 0) = \text{zero}(m)
  \]
  \[
  \text{mult}(m, n + 1) = \text{plus}(\text{mult}(m, n), m)
  \]

- **Exponentiation.**
  
  \[
  \text{pow}(m, 0) = \text{succ}(\text{zero}(m))
  \]
  \[
  \text{pow}(m, n + 1) = \text{mult}(\text{pow}(m, n), m)
  \]

- **Predecessor.**
  
  \[
  \text{pred}(0) = 0
  \]
  \[
  \text{pred}(n + 1) = n
  \]

\[\text{constant}\]

\[\text{add}(m, n) = m + n\]

\[\text{mult}(m, n) = m \times n\]

\[\text{pow}(m, n) = m^n\]

\[\text{pred}(n) = \max(n - 1, 0)\]
Examples

- **Nonnegative subtraction.**  \( \triangleright \) \( \text{sub}(m, n) = \max(m - n, 0) \)
  
  \( \text{sub}(m, 0) = m \)
  
  \( \text{sub}(m, n + 1) = \text{pred}(\text{sub}(m, n)) \)

- **Sign.**  \( \triangleright \) \( \text{sign}(n) = 0 \) if \( n = 0 \), \( 1 \) if \( n > 0 \)
  
  \( \text{sign}(0) = 0 \)
  
  \( \text{sign}(n + 1) = \text{succ}(\text{zero}(n)) \)

- **Positive.**
  
  \( \text{positive}(n) = \text{sign}(n) \)

- **IsZero.**  \( \triangleright \) \( \text{iszero}(n) = 1 \) if \( n = 0 \), \( 0 \) otherwise
  
  \( \text{iszero}(0) = 1 \)
  
  \( \text{iszero}(n + 1) = 0 \)

- **IsOne.**  \( \triangleright \) \( \text{isone}(n) = 1 \) if \( n = 1 \), \( 0 \) otherwise
  
  \( \text{isone}(0) = 0 \)
  
  \( \text{isone}(n + 1) = \text{iszero}(n) \)
Examples

- **Greater than or equal to.**
  \[ \text{ge}(m, n) = 1 \text{ if } m \geq n \]
  \[ \text{ge}(m, n) = \text{iszero}(\text{sub}(n, m)) \]

- **Negation.**
  \[ \text{neg}(p(m, n)) = \text{sub}(1, p(m, n)) \]

- **Disjunction.**
  \[ \text{or}(p(m, n), q(m, n)) = \text{sub}(1, \text{iszero}(\text{add}(p(m, n), q(m, n)))) \]

- **Conjunction.**
  \[ \text{and}(p(m, n), q(m, n)) = \text{sub}(1, \text{iszero}(\text{mult}(p(m, n), q(m, n)))) \]

- **How do you define le(m, n), gt(m, n), lt(m, n), and eq(m, n)?**
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.

\[
\text{rem}(0, n) = 0
\]

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

- Integer quotient.

\[
\text{div}(0, n) = 0
\]

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

- **Digit.**
  \[
  \text{digit}(m, p, n) = \text{div}(\text{rem}(n, \text{pow}(p, m)), \text{pow}(p, \text{sub}(m, 1)))
  \]

- **Series sum.**
  \[
  \sum_f(n, m) = f(n, 0) + f(n, 1) + \cdots + f(n, m)
  \]
  If \( f(n, m) \) is primitive recursive, so is \( \sum_f(n, m) \)

- 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 $\mu$-recursive functions?

**Definition**

The $\mu$-recursive functions are the smallest class of functions from $W \times W \times \cdots \times W$ to $W$ 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
6. minimalization of minimalizable functions

$\mu$-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.

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

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

1. $m \leftarrow 0$
2. while $g(X, m) \neq 1$ do
3. $m \leftarrow m + 1$
4. return $m$

$\text{TM-Min}$ 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.

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

<table>
<thead>
<tr>
<th>TM-MIN$(g, X)$</th>
<th>$\triangleright f(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $m \leftarrow 0$</td>
<td></td>
</tr>
<tr>
<td>2. while $g(X, m) \neq 1$ do</td>
<td></td>
</tr>
<tr>
<td>3. $m \leftarrow m + 1$</td>
<td></td>
</tr>
<tr>
<td>4. return $m$</td>
<td></td>
</tr>
</tbody>
</table>

**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 **TM-MIN** always halts.
(Primitive vs. $\mu$) recursive functions

<table>
<thead>
<tr>
<th></th>
<th>Primitive rec. functions</th>
<th>$\mu$-recursive functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparable to</td>
<td>Halting for-loops</td>
<td>Halting while-loops</td>
</tr>
<tr>
<td>#Iterations</td>
<td>Known beforehand</td>
<td>Not known beforehand</td>
</tr>
</tbody>
</table>

$\mu$-Recursive functions
- Primitive recursive functions
  - Ackermann function

Equivalent to Turing-computable algorithms or functions
While Programs
What are for and while programs?

<table>
<thead>
<tr>
<th>Operations</th>
<th>For programs</th>
<th>While programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignments</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>e.g. ( x \leftarrow y + 5 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential compositions</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>e.g. ( p; q )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditionals</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>e.g. if ((x &lt; y)) then (p) else (q)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>For loops</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>e.g. for (y) do (p)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>While loops</td>
<td>X</td>
<td>✔️</td>
</tr>
<tr>
<td>e.g. while (x &lt; y) do (p)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### What are for and while programs?

<table>
<thead>
<tr>
<th>Difference</th>
<th>For programs</th>
<th>While programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>For programs are computer programs without the while construct.</td>
<td>While programs are computer programs with the while construct.</td>
</tr>
<tr>
<td>#Iterations</td>
<td>Known beforehand. Does change after the execution of the loop body.</td>
<td>Might change after the execution of the loop body.</td>
</tr>
<tr>
<td>Halting</td>
<td>Always halt.</td>
<td>Might not halt.</td>
</tr>
</tbody>
</table>
## Relationship with recursive functions

<table>
<thead>
<tr>
<th>Time</th>
<th>Formal functions</th>
<th>Computer programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>Primitive rec. functions</td>
<td>For programs</td>
</tr>
<tr>
<td></td>
<td>$\mu$-recursive functions</td>
<td>Halting while programs</td>
</tr>
<tr>
<td>Infinite</td>
<td>Partially rec. functions</td>
<td>Non-halting while programs</td>
</tr>
</tbody>
</table>