# Lecture 13:
# Introduction to Recursive Function Theory

## Lecture Outline

- Review of Language/Machine Class Equivalences: The Chomsky Hierarchy
- Introduction to Recursive Function Theory
- Partial vs Total Functions
- Initial Functions
- Combination, Composition and Primitive Recursion

**Reading**

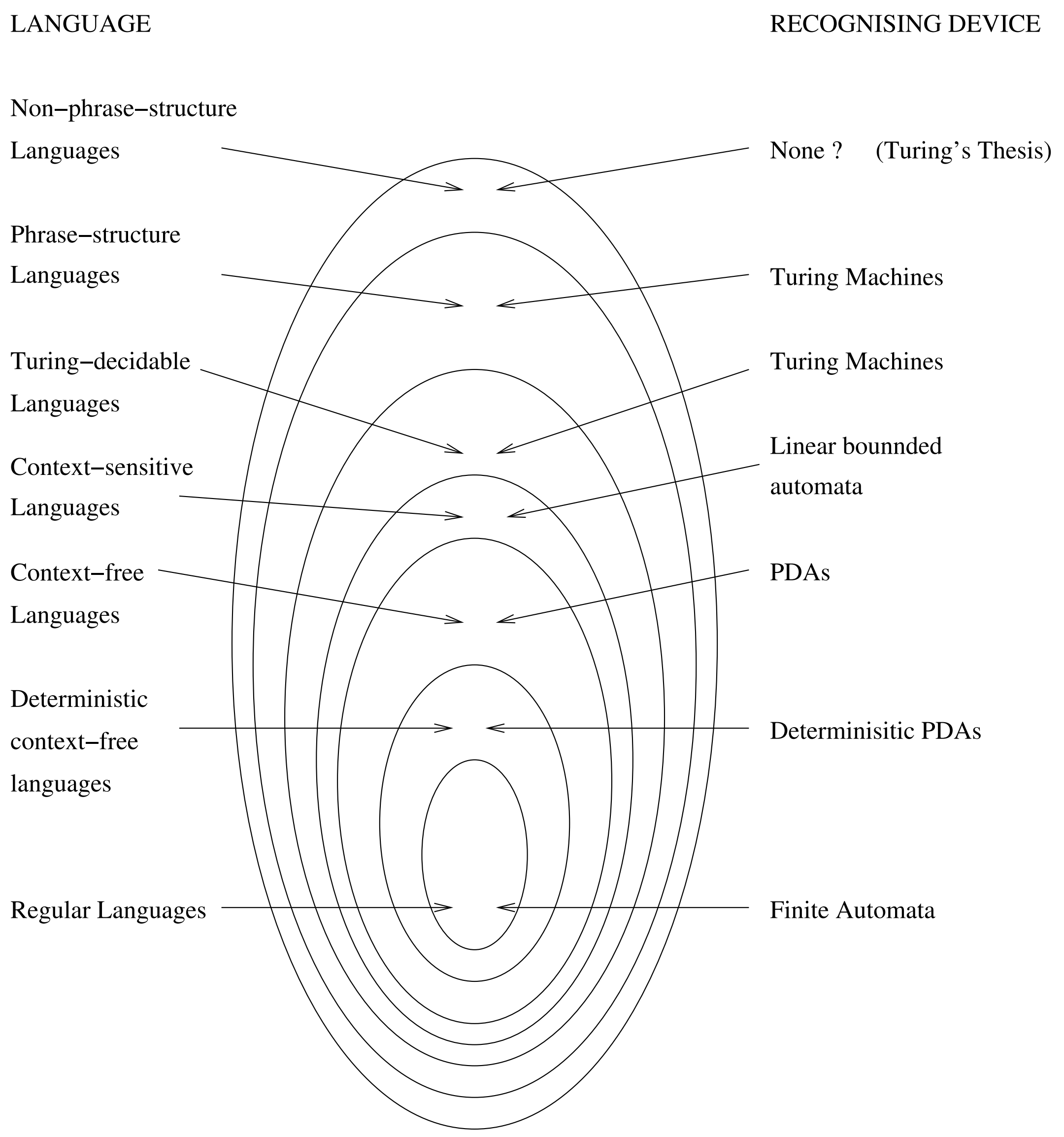
Chapter 4.1 of Brookshear

Chapter 13 of Martin (2nd ed)

N.J. Cutland *Computability: An Introduction to Recursive Function Theory*, Cambridge University Press, Cambridge, 1980. Chapters 1-2.

## Enhanced Chomsky Hierarchy

- The language/machine results we have studied so far can be presented in a single diagram:



- This is a variation of the **Chomsky hierarchy**, named after Noam Chomsky who did pioneering work in formal language theory in the 1950's.

  He identified four language classes: regular, context-free, context-sensitive and phrase-structure languages.

(We have not investigated context-sensitive languages – they can be recognised by a class of two stack pushdown automata called *linear bounded automata* – automata whose amount of memory is bounded by the length of their input).

## The Chomsky Hierarchy

- Recall again that the difference between classes of phrase structure grammar are due to the form of their rules.
- The differences in form of rules between language classes in the *Chomsky hierarchy* may be summarised as follows.

  A generative grammar $G = \langle \Sigma_N, \Sigma_T, S, R \rangle$ is said to be of type $i$ if:

  1. $i = 0$: Every rule in $R$ has the form
  $$P \rightarrow Q$$
  where $P$ and $Q$ are in $(\Sigma_N \cup \Sigma_T)^*$ and $P$ contains at least one symbol from $\Sigma_N$ (*phrase structure grammars*)
  2. $i = 1$: Every rule in $R$ has the form
  $$Q_1 A Q_2 \rightarrow Q_1 P Q_2$$
  with $Q_1, Q_2, P \in (\Sigma_N \cup \Sigma_T)^*$, $A \in \Sigma_N$, and $P \neq \lambda$, except possibly for the rule $A \rightarrow \lambda$ which may occur in $R$ in which case $S$ does not occur on the righthand side of the rules (*context-sensitive grammars*)
  3. $i = 2$: Every rule in $R$ has the form
  $$A \rightarrow P$$
  where $A \in \Sigma_N$ and $P \in (\Sigma_N \cup \Sigma_T)^*$ (*context-free grammars*)
  4. $i = 3$: Every rule in $R$ has the form
  $$A \rightarrow PB \quad \text{or} \quad A \rightarrow P$$
  where $A, B \in \Sigma_N$ and $P \in \Sigma_T$ (*regular or finite state grammars*).

- A language is then said to be of type $i$ if it is generated by a grammar of type $i$. The class of all languages of type $i$ is denoted $\mathcal{L}_i$.

- The following relationships have been proven to hold amongst the language classes defined by the Chomsky hierarchy:

  $$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0.$$

  Thus, each of these language classes is a **proper** subset of the one containing it – i.e. there are languages in each larger class that are not in the class below.

## Introduction to Recursive Function Theory

- Turing's thesis states that TM's form a bound on what can be computed.
- This claim is supported by equivalent bounds which emerge in other areas.
- One such equivalent bound is that imposed by phrase structure grammars.
- **Recursive function theory**, the study of a particular class of integer functions, provides another such equivalent bound.
- Rather than concentrate on designing a simple machine which serves to embody the essence of computation (the Turing machine), recursive function theory starts with a small class of self-evidently computable functions, and then explores how large a class of functions can be built up from these using certain simple operations.
- Using this approach recursive function theory aims to identify the class of computable functions.
- One practical consequence of this approach is that programming languages that can be shown to be sufficiently powerful to implement the simple functions underlying recursive function theory, and the means for building new functions from them, can compute any function in the class (and of course are equivalent to any other programming language that can do this).

## Partial vs Total Functions

- Since any data can be represented as strings of 0's and 1's, any computable function can be considered to be a function from non-negative integers to non-negative integers.

- However, not every computable function has the form:

  $$f : \mathbf{N}^m \to \mathbf{N}^n \;\; n, m, \text{ integers}$$

  since some functions are not defined for every m-tuple.

  E.g.

  $$div(x, y) = \text{integer part of } x/y, \text{ for } x, y \in \mathbf{N}, y \neq 0$$

  is not defined for $y = 0$ and hence is not a function of the form $f : \mathbf{N}^2 \to \mathbf{N}$.

- **Partial** functions are functions which are only defined for a subset of their domains.

  **Strictly partial** functions are functions which are only defined for a proper subset of their domains.

  **Total** functions are functions which are defined for every value in their domains.

  E.g. $div$ is a strictly partial function on $\mathbf{N}$ and $plus(x, y) = x + y$ is a total function on $\mathbf{N}$. Both $div$ and $plus$ are partial functions on $\mathbf{N}$.

## Initial Functions

- Recursive function theory supposes a stock of simple, self-evidently computable functions, called **initial functions**, and some computable mechanism(s) for building more complex functions from the initial functions.

- A notational convention: we use $\bar{x}$ to stand for the $n$-tuple $(x_1, \ldots, x_n)$ when the details of the expanded form are not necessary.

- The initial functions are:

  1. the **zero function** $\zeta$. $\zeta$ maps the zero-tuple $()$ to 0. I.e. $\zeta() = 0$.
     It corresponds to writing a 0 on a blank piece of paper, or initialising a tape cell, or a memory location to 0.

  2. the **successor function** $\sigma$. Given an integer $n$, $\sigma(n) = n + 1$.

  3. the **projection functions** $\pi_n^m$. The collection of projection functions map $m$-tuples onto the $n$-th component of the $m$-tuple.
     E.g. $\pi_2^3(7, 11, 4)$ returns the $2$nd element of the $3$-tuple $(7, 11, 4)$; i.e., $\pi_2^3(7, 11, 4) = 11$.
     Similarly, $\pi_2^2(4, 6) = 6$.
     By convention $\pi_0^n(\bar{x}) = ()$, for any arbitrary $n$-tuple $\bar{x}$

- All of these functions are held to be obviously computable – certainly they can be carried out in a mechanical fashion.

## Combination, Composition and Primitive Recursion

- Three basic techniques for constructing complex functions from the initial functions are: **combination**, **composition**, and **primitive recursion**.

  Functions constructed from the initial functions by a finite number of applications of combination, composition and primitive recursion are called **primitive recursive functions**.

- The *combination* of the functions $f : \mathbf{N}^k \rightarrow \mathbf{N}^m$ and $g : N^k \rightarrow N^n$, where $k, m, n \in \mathbf{N}$, is written $f \times g : \mathbf{N}^k \rightarrow N^{m+n}$ and is defined by

  $$f \times g(\bar{x}) = (f(\bar{x}), g(\bar{x})).$$

  I.e. $f \times g$ takes $k$-tuples as input and produces $m + n$-tuples where the first $m$ elements of the output are the output of $f$ when applied to the $k$-tuple and the next $n$ elements of the output are the output of $g$ when applied to the $k$-tuple.

  E.g.

  $$\begin{aligned} \pi_2^4 \times \pi_3^4(1,5,3,7) &= (\pi_2^4(1,5,3,7), \pi_3^4(1,5,3,7)) \\ &= (5,3). \end{aligned}$$

- If each of $f$ and $g$ is computable then clearly $f \times g$ is computable, since it just involves computing $f$ separately and $g$ separately and then combining their outputs into a single tuple.

## Composition

- The *composition* of the functions $f : \mathbf{N}^k \rightarrow \mathbf{N}^m$ and $g : N^m \rightarrow N^n$, where $k, m, n \in \mathbf{N}$, is written $g \circ f : \mathbf{N}^k \rightarrow N^n$ and is defined by

  $$g \circ f(\bar{x}) = g(f(\bar{x})).$$

  I.e. to form the output of $g \circ f$ apply $f$ to the input first, then apply $g$ to the output of $f$.

  E.g.

  $$\sigma \circ \zeta() = 1$$

  since $\zeta() = 0$ and $\sigma(0) = 1$.

- If each of $f$ and $g$ is computable then clearly $f \circ g$ is computable, since we can compute $f$ applied to the input first, the use the output of this as input to $g$.

## Primitive Recursion

- A function $f : \mathbf{N}^{k+1} \to \mathbf{N}^m$ is constructed from functions and $g : N^k \to N^m$ and $h : \mathbf{N}^{k+m+1} \to \mathbf{N}^m$, where $k, m, n \in \mathbf{N}$, using **primitive recursion** if

$$\begin{aligned} f(\bar{x}, 0) &= g(\bar{x}) \\ f(\bar{x}, y+1) &= h(\bar{x}, y, f(\bar{x}, y)) \end{aligned}$$

  where $\bar{x}$ is an arbitrary k-tuple.

- This can be paraphrased as:

  If the last component in the input $k+1$-tuple is $0$, $f$ is computed by applying $g$ to the first $k$ elements of the input $k+1$-tuple.

  If the last component in the input $k+1$-tuple is not $0$, then $f$ is computed by applying $h$ to the $k+m+1$-tuple consisting of:

  - the first $k$ elements of the input $k+1$-tuple (i.e. $\bar{x}$)
  - the predecessor of the last component of the input $k+1$-tuple (i.e. $y$)
  - the result of applying $f$ to the $k+1$-tuple obtained by subtracting 1 from the last element of the original $k+1$ tuple (i.e. $f(\bar{x}, y)$).

- The hallmarks of recursion are

  1. the occurrence on the right hand side of the defining equation of the function being defined on the left,
  2. the step-by-step reduction of arbitrary cases to a base or initial case.

## Primitive Recursion (cont)

E.g. the $plus$ function can be defined via primitive recursion as follows:

$$\begin{aligned} plus(x,0) &= \pi^1_1(x) \\ plus(x,y+1) &= \sigma \circ \pi^3_3(x,y,plus(x,y)) \end{aligned}$$

Letting $f = plus$, $g = \pi^1_1$ and $h = \sigma \circ \pi^3_3$ we can see how this definition of $plus$ fits the form of definition, given on the previous page, required for primitive recursion.

Note how we can compute $plus(3,2)$ for example:

$$\begin{aligned} plus(3,2) &= \sigma \circ \pi^3_3(3,1,plus(3,1)) \\ &= \sigma \circ \pi^3_3(3,1,\sigma \circ \pi^3_3(3,0,plus(3,0))) \\ &= \sigma \circ \pi^3_3(3,1,\sigma \circ \pi^3_3(3,0,\pi^1_1(3))) \\ &= \sigma \circ \pi^3_3(3,1,\sigma \circ \pi^3_3(3,0,3)) \\ &= \sigma \circ \pi^3_3(3,1,\sigma \circ 3) \\ &= \sigma \circ \pi^3_3(3,1,4) \\ &= \sigma \circ 4 \\ &= 5 \end{aligned}$$

- If each of $g$ and $h$ is computable then any function $f$ defined from them by primitive recursion will be computable since we can calculate any $f(\bar{x},y)$ by first computing $f(\bar{x},0)$ then $f(\bar{x},1)$, then $f(\bar{x},2)$ and so on until we reach $f(\bar{x},y)$.

- Note that since all the initial functions are total functions and since the construction techniques of combination, composition and primitive recursion yield total functions when applied to total functions, every primitive recursive function of the form $f : \mathbf{N}^m \to \mathbf{N}^n$ is a total function.

## Primitive Recursion – Example

- Show that the function $mult : \mathbf{N}^2 \rightarrow \mathbf{N}$ which returns the product of two natural numbers is primitive recursive.

  Define $mult$ by

$$\begin{aligned} mult(x, 0) &= \zeta \circ \pi_0^1(x) \\ mult(x, y+1) &= plus \circ (\pi_1^3 \times \pi_3^3(x, y, mult(x, y))) \end{aligned}$$

  This has the form

$$\begin{aligned} f(\bar{x}, 0) &= g(\bar{x}) \\ f(\bar{x}, y+1) &= h(\bar{x}, y, f(\bar{x}, y)) \end{aligned}$$

  where

$$\begin{aligned} f &= mult \\ g &= \zeta \circ \pi_0^1 \\ h &= plus \circ (\pi_1^3 \times \pi_3^3) \end{aligned}$$

  Since each of $f$, $g$, and $h$ is primitive recursive and $mult$ is defined from them by primitive recursion, it follows that $mult$ is primitive recursive.

  $mult$ can be defined more readably as:

$$\begin{aligned} mult(x, 0) &= 0 \\ mult(x, y+1) &= plus(x, mult(x, y)) \end{aligned}$$

  and we will shortly introduce conventions to allow us to do so.

## Summary

- Recursive function theory describes a class of integer functions built up from **initial functions** using **function-building operations**.

- The initial functions are:
  - zero function;
  - successor function;
  - projection functions.

- The function-building operations are:
  - combination;
  - composition;
  - primitive recursion.

- The class of functions consisting of the three initial functions plus those built up from them using the three function-building operations are called **primitive recursive functions** (later we will extend this class by adding an additional function-building operation).

  Note: Not all primitive recursive functions use the function-building operation of primitive recursion (e.g. the successor function $\sigma$).

- Since the initial functions are total and the three function-building operations preserve totality, all primitive recursive functions are total.