What is this?

- An alternate approach to computability, based on numeric functions.
- Sometimes having this alternate viewpoint will be helpful, and often more elegant.
- Also, much common terminology is derived from this approach rather than from Turing machines.
- The family of primitive recursive functions is first defined, then partial recursive functions are built on that.

Numbers vs. Strings

- What do we lose with numbers? Nothing really, as strings over any alphabet can be regarded as numerals:
  Over $\Sigma = \{a, b, c\}$ for example:
  
  - $i \leftrightarrow 0$
  - $a \leftrightarrow 1$
  - $b \leftrightarrow 2$
  - $c \leftrightarrow 3$
  - $aa \leftrightarrow 4$
  - $ab \leftrightarrow 5$

  - For an $n$-letter alphabet, this is called the $n$-adic encoding.

String Manipulation by Numbers

- In an $n$-adic encoding:
  - dividing by $n$ is truncating the last character.
  - modding by $n$ is getting the last character.
  - multiplying by $n$ and adding the value of a character is like consing the a character on the end.

- A Turing machine tape can be represented as two stacks, then the stacks pushed and pop by the above numeric operations, to simulate a Turing machine.

Primitive Recursive Functions

- The set of primitive recursive functions is defined inductively.

- Every function is $k$-ary, for some $k > 0$.

- The domain and co-domain of each function is the set of natural numbers $\{0, 1, 2, 3, \ldots\}$, or $k$-tuples in the case of domain.

Basis Functions (1 of 3)

- The zero function is primitive recursive:

  $$\text{zero}(x) = 0$$
Basis Functions (2 of 3)
- The **projection** functions are all primitive recursive:
  \[ \pi_k^j(x_1, x_2, \ldots, x_k) = x_j \]
  for each arity \( k \geq 1 \) and each \( i, 1 \leq i \leq k \).

Basis Functions (3 of 3)
- The **successor** function is primitive recursive:
  \[ S(x) = x + 1 \]

Induction Rules (1 of 2)
- The **composition** of primitive recursive functions is primitive recursive:
  \[ h(x_1, x_2, \ldots, x_k) = \]
  \[ f(g_1(x_1, x_2, \ldots, x_k), \]
  \[ g_2(x_1, x_2, \ldots, x_k), \]
  \[ \ldots \]
  \[ g_r(x_1, x_2, \ldots, x_k)) \]
  for each pair of arities \( k, r > 0 \).

Constant Functions
- A consequence of the rules up to this point is that **constant** functions are all primitive recursive:
  \[ C^k_c(x_1, x_2, \ldots, x_k) = c \]
  for each natural number \( c \).
  
  This is so because it is just a composition of the zero and successor functions:
  \[ C^k_c(x_1, x_2, \ldots, x_k) = S(S(S(zero(\pi^k_1(x_1, x_2, \ldots, x_k)))) \ldots)) \]

Explicit Definition (ED)
- This is a convenient shorthand for combining compositions, projections, and constants. We can just use definitions such as:
  \[ f(x, y, z) = g(h(y, x), s, k(z, z)) \]
  and know that if \( g, h, \) and \( k \) are primitive recursive, so is \( f \),
  because we can exhibit the corresponding composition of zero, \( S \), and projections.

Example of Explicit Definition
- \( f(x, y, z) = g(h(y, x), s, k(z, z)) \) is equivalent to:
  \[ f(x, y, z) = g(h(\pi^3_2(x, y, z), \pi^3_1(x, y, z)), \]
  \[ S(S(S(S(zero(\pi^3_2(x_1, x_2, \ldots, x_k)))))), \]
  \[ k(\pi^3_2(x, y, z), \pi^3_2(x, y, z)) \]
- ED is also sometimes called ET (Explicit Transformation)
Induction Rules (2 of 2)

- A function \( f \) defined from already-defined primitive recursive functions \( b \) and \( r \) of the appropriate arities by the following primitive recursion pattern is primitive recursive:
  \[
  f(0, x_1, x_2, \ldots, x_k) = b(x_1, x_2, \ldots, x_k)
  \]
  \[
  f(n+1, x_1, x_2, \ldots, x_k) = r(x_1, x_2, \ldots, x_k, n, f(n, x_1, x_2, \ldots, x_k))
  \]
  - Note that primitive recursion is a very stylized template. Not every recursion fits into this template.

Examples of Primitive Recursive Functions
- \( \text{add}(x, y) \): addition
- \( \text{mult}(x, y) \): multiplication
- \( \text{pred}(x) \): predecessor
- \( \text{sub}(x, y) \): proper subtraction
- \( \text{mod}(x, y) \): modulus
- \( \text{div}(x, y) \): integer division (quotient)
- \( \text{sqrt}(x) \): integer square root
  * \( \text{sub}(x, y) \) is 0 if \( x < y \)

rex implementations
- I will demonstrate some of these using explicit definition in rex. This allows the definitions to be tested readily.
- rex does not restrict to natural numbers and does not enforce a primitive recursive formalism, so we have to be careful not to “cheat”.

add implementation in rex
- \( S(n) = n + 1; \) // pretend this definition is built in
- \( \text{add}(0, y) \Rightarrow y; \)
- \( \text{add}(n+1, y) \Rightarrow S(\text{add}(n, y)); \)
  - For reference (identify \( b \) and \( r \) above):
    \[
    f(0, x_1, x_2, \ldots, x_k) = b(x_1, x_2, \ldots, x_k)
    \]
    \[
    f(n+1, x_1, x_2, \ldots, x_k) =
    \]
    \[
    r(x_1, x_2, \ldots, x_k, n, f(n, x_1, x_2, \ldots, x_k))
    \]

mult implementation
- \( \text{mult}(0, y) \Rightarrow 0; \)
- \( \text{mult}(n+1, y) \Rightarrow \text{add}(y, \text{mult}(n, y)); \)
  - For reference (identify \( b \) and \( r \) above):
    \[
    f(0, x_1, x_2, \ldots, x_k) = b(x_1, x_2, \ldots, x_k)
    \]
    \[
    f(n+1, x_1, x_2, \ldots, x_k) =
    \]
    \[
    r(x_1, x_2, \ldots, x_k, n, f(n, x_1, x_2, \ldots, x_k))
    \]

pred (predecessor) implementation
- informally \( \text{pred}(y) = y = 0? 0 : y-1; \)
- \( \text{pred}(0) \Rightarrow \)
- \( \text{pred}(n+1) \Rightarrow \)
  - For reference (identify \( b \) and \( r \) above):
    \[
    f(0, x_1, x_2, \ldots, x_k) = b(x_1, x_2, \ldots, x_k)
    \]
    \[
    f(n+1, x_1, x_2, \ldots, x_k) =
    \]
    \[
    r(x_1, x_2, \ldots, x_k, n, f(n, x_1, x_2, \ldots, x_k))
    \]
sub implementation

- sub is **proper** subtraction (aka "monus"): If \( a > b \), then \( \text{sub}(a, b) = a - b \).
  If \( a < b \), then \( \text{sub}(a, b) = 0 \).

- \( \text{sub}(y, 0) => \)

- \( \text{sub}(y, n+1) => \)

Primitive Recursive *Predicates*

- For some definitions we want to have predicates, which we can equate to functions that return only values 0 (false) and 1 (true).

  - \( \text{sgn}(0) => 0; \)
  - \( \text{sgn}(n+1) => 1; \)
  - \( \text{sgn} \) converts arbitrary values to \( \{0, 1\} \).

Negation

- \( \text{not}(0) => 1; \)

- \( \text{not}(n+1) => 0; \)

Equality Predicate

- \( \text{eq}(x, y) = \text{not}(\text{add}(\text{sub}(x, y), \text{sub}(y, x))); \)

if-then-else function

- \( \text{ifthenelse}(0, x, y) => y; \)

- \( \text{ifthenelse}(n+1, x, y) => x; \)

mod and div

- \( \text{mod}(0, y) => 0; \)

- \( \text{mod}(n+1, y) => \text{ifthenelse}(\text{eq}(S(\text{mod}(n, y)), y), 0, S(\text{mod}(n, y))); \)

- \( \text{div}(0, y) => 0; \)

- \( \text{div}(n+1, y) => \text{ifthenelse}(\text{eq}(S(\text{mod}(n, y)), y), S(\text{div}(n, y)), S(\text{div}(n, y))); \)
**Pragmatic Perspective**

- Primitive recursive functions are functions that can be defined using only **definite iteration** (e.g. the equivalent of a for-loop with upper bound pre-determined)

  and not requiring **indefinite iteration** (while-loops) or the full power of recursion.

- Primitive recursion **as given** is not a special case of tail recursion, although there is an equivalent version that is.

  The standard version of primitive recursion is “top-down”, whereas tail-recursion is “bottom-up”.

**Primitive Recursion = Definite Iteration**

- The function \( f \) defined in the primitive recursion scheme can be computed by the following for-loop:

  ```
  // To compute acc == f(n, x_1, x_2, ..., x_k)
  // where f is defined by primitive recursion
  // from b and r
  acc := b(x_1, x_2, ..., x_k);
  for( j := 0; j < n; j++ )
  {
      acc := r(x_1, x_2, ..., x_k, j, acc);
  }
  ```

**Proof by Invariant**

- The function \( f \) defined in the primitive recursion scheme can be computed by the following for-loop:

  ```
  // To compute acc == f(n, x_1, x_2, ..., x_k)
  // where f is defined by primitive recursion
  // from b and r
  acc := b(x_1, x_2, ..., x_k);
  for( j := 0; j < n; j++ )
  { // invariant: acc = f(j, x_1, x_2, ..., x_k)
      acc := r(x_1, x_2, ..., x_k, j, acc);
  }
  ```

**Tail-Recursion Theorem**

- The function \( f(n, x_1, x_2, ..., x_k) \) defined by primitive recursion can be computed as \( t(n, b(x_1, x_2, ..., x_k)) \) where \( t \) is defined in the following tail-recursion:

  ```
  t(0, acc) => acc;
  t(n+1, acc) => t(n, mult(n+1, acc));
  ```

- Proof: This version can be “read off” from the previous loop version. The connection to the original primitive recursion was established by the loop invariant.

**Example: Factorial**

- Primitive-recursive version (uses the primitive-recursion pattern):

  ```
  fac(0) => 1;
  fac(n+1) => mult(n+1, fac(n));
  ```

- Tail-recursive version (doesn’t use the pattern, but equivalent):

  ```
  fac_tr(n) = t(n, 1);
  t(0, acc) => acc;
  t(n+1, acc) => t(n, mult(n+1, acc));
  ```

**Totality Theorem**

- Every primitive recursive function is a total function.

  Two levels of induction are involved:

  - For each individual use of the primitive-recursion pattern, there is an induction to show that \( f \) is defined for all \( n \), assuming that \( b \) and \( r \) are total.

  - Structural induction is used to ascertain that anything defined from the derivation rules is a function.
Computability Theorem
- Every primitive-recursive function is computable by a Turing machine.
- This follows from the Church/Turing thesis.
- It can also be shown in significant detail by showing how a Turing machine can be constructed by composing functions using the basis functions and induction rules.

Primitive Recursion
Diagonalization Theorem
- There is a computable function that is not primitive recursive.
- Proof: A Turing machine can effectively enumerate the primitive recursive functions of one argument, by applying the rules in some orderly fashion:
  \[ p_0, p_1, p_2, \ldots \]
  Then define \( q(x) = p_x(x) + 1 \). This function is clearly total, since each \( p_x \) is, but \( q \) cannot be \( p_k \) for any \( k \).

The Ackermann Hierarchy
- We notice that add and mult have similar definitions.
  - add uses S as a base
  - mult uses add as a base
  We can go on to define exp analogously:
    - exp uses mult as a base
  When does this stop?
  Never, but we quickly reach functions that have very large values for small arguments.
  Ackermann observed that is possible to diagonalize over this hierarchy.

The Ackermann Hierarchy
- \( A_0(m) = S(m) \)
- \( A_{n+1}(0) = A_n(1) \)
- \( A_{n+1}(m+1) = A_n(A_{n+1}(m)) \)
  In effect, \( A_{n+1}(m+1) = A^n(1) \), the \( m \)-fold application of \( A_n \).
- Each function in the list: \( A_0, A_1, A_2, \ldots \) is clearly primitive-recursive.
- Define \( A(n, m) = A(n, m) \) (called Ackermann’s Function)
  It can be proved that for any primitive recursive function \( p \) of one variable, there is an \( n \) such that
    \[ \forall m \in \mathbb{N} \quad p(m) < A(n, m) \]
  Then the function \( q(m) = A(m, m) \) cannot be primitive recursive.

Partial-Recursive Functions
- These extend the primitive recursive functions by using the \( \mu \) operator”.

Partial-Recursive Functions
- Start with the primitive-recursive functions as a base.
- Add one more induction rule: If \( h \) is a \( k+1 \) ary partial–recursive function (prf), then \( f \) is a \( k \)-ary prf:
  \[ f(x_1, x_2, \ldots, x_k) = \mu x_k \left[ h(x_0, x_1, \ldots, x_k) = 0 \right] \]
  “the least value of \( x_k \) such that \( h(x_0, x_1, \ldots, x_k) = 0 \)”.
- Note: It is understood that if \( h(x_0, x_1, \ldots, x_k) \) is undefined for any \( y < \) the least \( x_k \), then the value of \( f(x_1, x_2, \ldots, x_k) \) is also undefined.
- \( \mu \) is called the “minimalization operator”.
Example of Using the $\mu$ Operator

- Suppose we want to compute the integer square root of a number. We could define
  $$\text{sqrt}(n) = \mu k [\text{sub}(n, \text{mult}(k, k)) = 0]$$
- It turns out that this particular use of $\mu$ is not essential; $\text{sqrt}$ can be computed by primitive-recursive means. Still, it is convenient.

Example of Non-Total Functions Using $\mu$ Operator

- Consider
  $$\text{diverge}(n) = \mu k [\text{sub}(k+1, k) = 0]$$
  $\text{diverge}(n)$ is undefined for all $n$.
- Consider
  $$\text{strange}(m, n) = \mu k [\text{not}(\text{eq}(k+m, n)) = 0]$$

Note on $\mu$ Operator and Ackermann

- It is not obvious why the $\mu$ operator would give us a way to compute Ackermann’s function.
- The “double-recursion” equations given for Ackermann’s function actually fit within a different formalism, Herbrand-Gödel-Kleene general recursive functions (GRF) rather than the partial-recursive functions. [The formalism is similar to a set of rex definitions over functions on the natural numbers.]
- The two formalisms are equivalent, but this is often proved in a way that does not make a clear connection that bridges the gap between primitive and partial recursive functions in a manner applicable to Ackermann’s function.

Computability Theorem for Partial-Recursive Functions

- Again we can appeal to the Church-Turing thesis to convince ourselves that the partial-recursive functions are computable partial functions.
- An explicit construction can also be given. Please think about how this could be done.
- It is clear that partial-recursive functions are not always total.

Converse of the Computability Theorem

- Every Turing computable partial function is computable by a partial-recursive function.
- Moreover, the $\mu$ operator needs to be used only once to achieve any partial-recursive function.

Importance of the Computability Theorem and its Converse

- Turing-computable partial functions and partial-recursive functions are established as being virtually the same thing.
- One was defined using strings, the other using numbers.
Strings vs. Numbers

• We recognize that natural numbers and strings are equivalent.

• Strings can be enumerated in a straightforward way, for example the strings over a 2-letter alphabet \( \{a, b\} \):
  
  \[
  \begin{align*}
  0 & \rightarrow \Lambda \\
  1 & \rightarrow a \\
  2 & \rightarrow b \\
  3 & \rightarrow aa \\
  4 & \rightarrow ab \\
  5 & \rightarrow ba \\
  \end{align*}
  \]

• So a set of numbers is equivalent to a language (set of strings).

Establishing the Converse

• The converse shows that any Turing-computable partial function is a partial-recursive function.

• To do this involves encoding TM tapes and configurations as numbers.

• Then it can be shown that there are primitive recursive functions that:
  
  - Simulate a single step of a Turing machine.
  - Tell whether an encoded configuration is halting.

Primitive Recursive Functions for TMs

• \( R(x) \) is the encoding of the configuration resulting after 1 step from encoded configuration \( x \).

• \( T(i, x) \) is the encoding of the configuration resulting from encoded configuration \( x \) after \( i \) steps.

• \( P(x) \) indicates whether or not an encoded configuration is halting (0 or 1).

Recursive TM equivalents, using \( \mu \)

• Halting in \( i \) steps is expressed by:
  
  \[
  \mu [ P(T(i, x_0)) = 0 ]
  \]

• The halting configuration, if any, resulting from \( x_0 \) is:
  
  \[
  T(\mu [ P(T(i, x_0)) = 0 ], x_0)
  \]

Encodings

• Using primitive recursive functions to encode and decode tapes and configurations requires a lengthy, but interesting, excursion.

• One way (but not the only way) to encode arbitrary sequences of numbers is to use "Gödel numbering":
  
  Any sequence of natural numbers \( (x_1, x_2, \ldots, x_k) \)
  
  can be encoded as a single natural number:
  
  \[
  p_1^{1+x_1} p_2^{1+x_2} \cdots p_k^{1+x_k}
  \]

Universal Partial-Recursive Functions

• Most results for Turing machines have parallels for the partial-recursive functions.

• The partial-recursive functions are programs that can be coded and effectively enumerated just like Turing machines can:
  
  \( \psi_0, \psi_1, \psi_2, \psi_3, \ldots \)
  
  are the \( k \)-ary partial-recursive functions for any fixed \( k \).

• "Effective" here means that there is an algorithm that, given \( i \), can construct \( \psi_i \).
Kleene’s Normal Form Theorem

• For each \( k > 1 \), there exists a 1-ary primitive recursive function \( U \) and a \((k+2)\)-ary primitive recursive predicate \( T_k \) such that
  \[
  \phi^*_k(n, x_1, x_2, \ldots, x_k) \text{ converges iff } (\exists z) \ T_k(n, x_1, x_2, \ldots, x_k, z)
  \]
  \[
  \phi^*_k(n, x_1, x_2, \ldots, x_k) = U(\mu z \ T(n, x_1, x_2, \ldots, x_k, z) = 0)
  \]
  Essentially, \( T \) is like the function that tells whether the \( n \)th configuration of a TM computation is halting, while \( U \) gives the result from that halting configuration.
  The numbers \( z \) code both the program for the partial recursive function in question and the number of steps.

Universal Partial-Recursive Functions

• For each \( k \), there is a partial-recursive function \( \psi \) of \( k+1 \) variables such that
  \[
  \psi(n, x_1, x_2, \ldots, x_k) = \phi^*_k(n, x_1, x_2, \ldots, x_k)
  \]
  \( \psi \) is a universal function for \( k \) arguments.

Important: Terminology

• Henceforth, Turing-computable and “recursive” are used interchangeably:
  • Partial-recursive function = partial function computable by a Turing machine
  • Recursive function = total function computable by a Turing machine.
  These are not to be confused with “recursive” as used in programming language parlance.

Languages of Indices

• Set of indices of Turing machines (equivalently partial-recursive functions) provide a good testing ground for understanding the distinctions between recursive and recursive-enumerable languages.
  Suppose that \( \psi_0^k, \psi_1^k, \psi_2^k, \psi_3^k, \ldots \) is an effective enumeration of all \((k-ary)\) partial recursive functions.

Divergence Notation

• \( \psi(x) \downarrow \) is used to mean that \( \psi \) is defined for argument \( x \).
• \( \psi(x) \uparrow \) is used to mean that \( \psi \) diverges on argument \( x \).

Divergence Problem Re-Cast

• The set \( \{ j \in \mathbb{N} \mid \psi_0(j) \uparrow \} \) is not recursively-enumerable; this is the divergence problem.
  Suppose that \( D \) were r.e. Then by the alternate characterization, there is a \( k \) such that \( q_k \) has \( D \) as its domain.
  By definition of \( D \), \( k \in D \) iff \( q_k(k) \uparrow \).
  But since \( D \) is the domain of \( q_k \), \( k \notin D \) iff \( q_k(k) \uparrow \), by definition of “domain”.
Halting vs. Divergence

- The set $H = \{ j | \psi_j(j) \downarrow \}$ is recursively-enumerable (why?).

- But $H$ is not recursive; this is the **halting problem**.

- If $H$ were recursive, then so would its complement be.

- But its complement is $D$ on the previous slide, which is not even recursively-enumerable.

The Set of Indices of **Total** Recursive Functions is not Recursively-Enumerable

- Let $A = \{ j | \forall x \psi_j(x) \downarrow \}$.

- Suppose that $A$ is r.e.

- Let $T$ be a total recursive function that enumerates $A$, i.e. $A = \{ T(0), T(1), \ldots \}$.

- Then the function $T'$ defined by:
  $$
  T'(j) = \psi_{T(j)}(j) + 1
  $$
  is also total and obviously computable (**recursive**).

- Thus $T'$ has an index $k \in A$:
  $$
  T' = \psi_k
  $$

- But then $T'(k) = \psi_k(k) = \psi_{T(k)}(k) + 1 = T'(k) + 1$, which is contradictory.