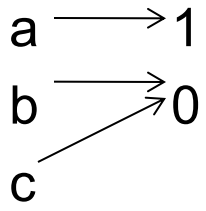
Computable Functions

Chapter 25

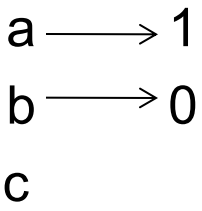
Total and Partial Functions

- *f* is a ***total function*** on the domain *Domain* iff *f* is defined on all elements of *Domain*.
- *f* is a ***partial function*** on the domain *Domain* iff *f* is defined on zero or more elements of *Domain*.
- Let *Dom* $\subseteq$ *Domain* be the set of elements on which *f* is defined, then *f* is total on *Dom*.

f: {a,b,c} -> {0,1}



g: {a,b,c} -> {0,1}



In *f*, *Domain* = *Dom* = {a,b,c}, total

In *g*, *Domain* = {a,b,c}, *Dom*={a,b}, partial on *Domain*, total on *Dom*

Total and Partial Functions

- The successor function *succ*:
  - Total on *Domain* = $\mathbb{N}$.

- *midchar*, which returns the middle character of its argument string if there is one.
  - Partial on *Domain* = {*w*} = set of strings.
  - Total on *Dom* = {*w* : |*w*| is odd for string *w*}

- *steps*, defined on inputs of the form <*M*, *w*>, returns the number of steps that Turing machine *M* executes, on input *w*, before it halts.
  - Partial on *Domain* = {<*M*, *w*>}
  - Total on *Dom* = {<*M*, *w*>: Turing machine *M* halts on input *w*}.

Why can't we just narrow *Domain* to *Dom,* then all functions become total?

Why Bother to Have Partial Functions

- Look at *steps*, defined on inputs of the form <*M*, *w*>, returns the number of steps that Turing machine *M* executes, on input *w*, before it halts.
  - Partial on *Domain* = {<*M*, *w*>}
  - Total on *Dom* = {<*M*, *w*>: Turing machine *M* halts on input *w*}.
- *Dom* for *steps* is not in *D*, i.e., not decidable
  - We cannot have a function whose domain is not decidable
  - Impossible for any implementation of *steps* to check its precondition.
  - The only way is to define its domain as some decidable set and then allow elements of that domain for which *steps* will not return a value.
  - Thus, *steps* becomes a partial function.
  - Such program will fail to halt on some inputs.
- Now look at *midchar*, which returns the middle character of its argument string if there is one.
  - Partial on *Domain* = {*w*} = set of strings.
  - Total on *Dom* = {*w* : |*w*| is odd for string *w*}
- *midchar* is also partial but *Dom* for *midchar* is in *D*

Partially Computable Functions

Let $M = (K, \Sigma, \Gamma, \delta, s, \{h\})$.
The initial configuration of M will be $(s, \underline{\sqcup}w)$.
Define $M(w) = z$ iff $(s, \underline{\sqcup}w) \vdash_M^* (h, \underline{\sqcup}z)$.

A function f is ***partially computable*** or ***partial recursive*** iff there is a Turing machine M_p such that for all $w \in \Sigma^*$:

1. If w is an input on which f is defined, $M_p(w) = \langle f(w) \rangle$

   (In other words, M_p halts with the encoding of $f(w)$ on its tape)

2. Otherwise, M_p does not halt

   (In some references, " … or it might get stuck at some point, but it must not pretend to produce a value for f at w")

Computable Functions

Let $M = (K, \Sigma, \Gamma, \delta, s, \{h\})$.
The initial configuration of M will be $(s, \underline{\sqcup} w)$.
Define $M(w) = z$ iff $(s, \underline{\sqcup} w) \vdash_M^* (h, \underline{\sqcup} z)$.

A function f is ***computable*** or ***recursive*** iff there is a Turing machine M_c such that for all $w \in \Sigma^*$:

1. If w is an input on which f is defined, $M_c(w) = \langle f(w) \rangle$
2. Otherwise, M_c halts and output "Error"

•M_c always halts

•Every computable function is also partially computable.

- Because if there exists M_c for f, there must exist M_p for it.
- We can use M_c as a subroutine of M_p. Seeing "Error", loop

•Not vice versa, unless there's some additional condition

- *Dom* of f is decidable (will see later)

Computable Total Functions

A function *f* is ***computable*** or ***recursive*** iff there is a Turing machine M_c such that for all $w \in \Sigma^*$:

1. If *w* is an input on which *f* is defined, $M_c(w) = \langle f(w) \rangle$
2. Otherwise, M_c halts and output "Error"

Note, if *f* is total, 2 is never reached.

Thus, a function *f* is computable if it is total and partially computable.
Or, a total function *f* is computable iff it's partially computable.
Or, for any total function, computable = partially computable.

- e.g., *succ* (we need to show we can find M_p for it, skip here)

- e.g., the characteristic function of a language *L* (or a set).
- A function *f* with domain *Domain* is a characteristic function of a set *L* iff *f*(*x*) = *True* if *x* is an element of *L* and *False* otherwise.
- Such *f* must be total coz on any element, in or out of *L, f* must be defined.
- So, a language is decidable iff its characteristic function is computable.

- This doesn't mean all computable functions needs to be total. see next

Computable Partial Functions

Equivalently, a function *f* is ***computable*** iff *f* is partially computable and the *Dom* of *f* is decidable.

- since *f* is partially computable, we can find M_p for it
- since *Dom* of *f* is decidable, we can find a decider M_d for it
- Then we can build M_c like this:
  - (1) Run M_d on *w* (to check precondition)
  - (2) If M_d rejects (*w* is undefined), M_c halts and output "Error"
  - (3) Otherwise (M_d accepts and *w* is undefined), run M_p on *w*

- e.g., *midchar* is partial but its *Dom* is decidable (we need to show we can find M_p for it, skip here)

For any *f*, if its *Dom* is not decidable (like *steps*, *Dom* = {<*M*, *w*> : *M* halts on *w*>}), it cannot be computable. At most partially computable.

- So, for any partial function, computable -> partially computable but not vice versa

Alternatively …

The text first defines "computes", and then uses it to define "computable" and "partially computable". I found it less intuitive and more confusing … However, it can be convenient when we say "There is no Turing machine that computes it" …

Let $M = (K, \Sigma, \Gamma, \delta, s, \{h\})$.
The initial configuration of M will be $(s, \underline{\sqcup} w)$.
Define $M(w) = z$ iff $(s, \underline{\sqcup} w) \mid\!-_M^*\ (h, \underline{\sqcup} z)$.

M ***computes*** a function f iff, for all $w \in \Sigma^*$:

- If w is an input on which f is defined, $M(w) = \langle f(w) \rangle$
- Otherwise $M(w)$ does not halt

A function f is ***partially computable*** or ***partial recursive*** iff there is a Turing machine M that computes it.

- M does not always halt (for w that is undefined)

A function f is ***computable*** or ***recursive*** iff there is a Turing machine M that computes it and that always halts.

- Equivalently, a function f is computable **iff** it's partially computable and its domain is a decidable set.

Note: in some references, computable is used for partially computable

Alternatively …

$$M(w) \equiv \begin{cases} y & \text{if } M \text{ on input "}\triangleright w \sqcup\text{" eventually halts with output "}\triangleright y \sqcup\text{"} \\ \nearrow & \text{otherwise} \end{cases}$$

$\Sigma_0 \equiv \Sigma - \{\triangleright, \sqcup\}$; Usually, $\Sigma_0 = \{0, 1\}$; $w, y \in \Sigma_0^\star$

Definition 4.1 Let $f : \Sigma_0^\star \to \Sigma_0^\star$ be a total or partial function. We say that f is a **partial, recursive function** iff $\exists$ TM $M (f = M(\cdot))$, i.e., $\forall w \in \Sigma_0^\star (f(w) = M(w))$. □

Remark 4.2 *There is an easy to compute 1:1 and onto map between* $\{0, 1\}^\star$ *and* **N** *[Hw2]. Thus we can think of the contents of a TM tape as a natural number and talk about* $f : \mathbf{N} \to \mathbf{N}$ *being a* **recursive** *function.*

If the partial, recursive function f is total, i.e., $f : \mathbf{N} \to \mathbf{N}$ then we say that f is a **total, recursive function**. A partial function that is not total is called **strictly partial**.

Computing and Deciding

A language is decidable iff its characteristic function is computable.

- In other words, a language *L* is decidable iff there exists a Turing machine that always halts and that outputs *True* if its input is in *L* and *False* otherwise.

Actually, a language is decidable iff its characteristic function is partially computable.

- Because the function must be total

We might be attempted to say the following:

- A language is semi-decidable iff its characteristic function is partially computable.

This is wrong, but the following is true:

- A language is semi-decidable if its characteristic function is partially computable.
- Because if decidable, then semi-decidable (*D* is a subset of *SD*)

If *L* is in *SD* but not *D* (semidecidable but not decidable), either:

- *L* does not have a characteristic function
- Its characteristic function is not partially computable

Computing and Deciding

If *L* is in *SD* but not *D* (semidecidable but not decidable), either:

- *L* does not have a characteristic function
- Its characteristic function is not partially computable

But then we can define for *L* a "partial characteristic function" *f*, whose *Dom* = *L*

- i.e., for only elements in *L*, *f* is defined and maps them to 1.
- Then we can say the following:
  A language is semi-decidable iff its partial characteristic function is partially computable.

- Then, we can say the following:
  Deciding or semideciding a language *L* is also computing the characteristic function or partial characteristic function of *L*.

If *L* is not in *SD* (not semidecidable), either:

- *L* does not have a partial characteristic function (what could be the case)
- Its partial characteristic function is not partially computable (next slide $\neg$SD)

Three Views

The Problem View	The Language View	The Functional View
Given three natural numbers, x, y, and z, is $z = x \cdot y$?	$\{x * y = z : x, y, z \in \{0, 1\}^*, num(x) \cdot num(y) = num(z)\}$ **D**	$f: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, $f(x, y) = x \cdot y$ **computable**
Given a TM M, does M have an even number of states?	$\{<M> : M$ has an even number of states$\}$ **D**	$f: \{<M>\} \rightarrow$ Boolean, $f(<M>) =$ *True* if M has an even number of states, *False* otherwise **computable**
Given a TM M and a string w, does M halt on w in n steps?	$\{<M, w, n> : M$ halts on w in n steps$\}$ **SD/D**	$f: \{<M, w>\} \rightarrow \mathbb{N}$, $f(<M, w>) =$ If TM M halts on w, then the number of steps it executes before halting, else undefined. **partially computable**
Given a TM M, does M halt on all strings in no more than n steps?	$\{<M, n> : M$ halts on each element of Σ^* in no more than n steps$\}$ **¬SD**	$f: \{<M\} \rightarrow \mathbb{N}$, $f(<M>) =$ If TM M halts on all strings, then the maximum number of steps it executes before halting, else undefined. **not partially computable**

Example of Computing a Function

Let Σ = {a, b}. Let *f*(*w*) = *ww.* //duplicating a string

Input: ❑*w*❑❑❑❑❑❑ Output: ❑*ww*❑

1. copy:

❑*w*❑❑❑❑❑❑ → ❑*w*❑*w*❑

2. shift:

❑*w*❑*w*❑ → ❑*ww*❑

Computing Numeric Functions

Example: $succ(n) = n + 1$

We will represent n in binary. So $n \in 0 \cup 1\{0, 1\}^*$

Input: ❑n❑❑❑❑❑❑ Output: ❑n+1❑

❑1111❑❑❑❑ Output: ❑10000❑

1. Scan right until the first ❑. Then move on square back left so that the head is on the last digit of n.
2. Loop:
    1. If the current digit is 0, write a 1, move head left to the first ❑, and halt
    2. If the current digit is 1, we need to carry. So write a 0, move one square to the left, and go back to the top of the loop
    3. If the current digit is ❑, we have carried all the way to the left. Write a 1, move one square to the left, and halt

There Exist Functions That Are Not Partially Computable

Theorem: There exist (a very large number of) functions that are not partially computable.

Proof: By a counting argument. Let *U* be the set of unary functions from some subset of $\mathbb{N}$ to $\mathbb{N}$. Encode both inputs and outputs as binary strings.

There Exist Functions That Are Not Partially Computable

Lemma: There is a countably infinite number of partially computable functions in *U*.

Proof of Upper bound: Every partially computable function in *U* is computed by some Turing machine *M* with Σ and Γ equal to {0, 1}, There are countably infinitely many such machines. There cannot be more partially computable functions than there are Turing machines.

Proof of Lower bound: The number of partially computable functions must be infinite because it includes all the constant functions:

$$cf_1(x) = 1,\ cf_2(x) = 2,\ cf_3(x) = 3,\ \ldots$$

So there is a countably infinite number of partially computable functions in *U*.

There Exist Functions That Are Not Partially Computable

Lemma: There is an uncountably infinite number of functions in *U*.

Proof of Lemma: Let *S* be $\mathcal{P}(\mathbb{N})$. For any element *s* of *S*, let f_s be the characteristic function of *s*. No two elements of *S* have the same characteristic function. There is an uncountably infinite number of elements in *S*, so there is an uncountably infinite number of such characteristic functions, each of which is in *U*.

There Exist Functions That Are Not Partially Computable

Proof of Theorem: Since there is only a countably infinite number of partially computable functions in *U* and an uncountably infinite number of functions in *U*, there is an uncountably infinite number of functions in *U* that are not partially computable.

Can We Describe One?

Yes, use diagnolization.

Let E be a lexicographic enumeration of the TMs that compute the partially computable functions in U.

Let M_i be the i^{th} machine in that enumeration.

Define a new function *notcomp*(x) as follows:

$$\textit{notcomp}: \mathbb{N} \rightarrow \{0, 1\},$$
$$\textit{notcomp}(x) = \begin{cases} 1 \text{ if } M_x(x) = 0, \\ 0 \text{ otherwise.} \end{cases}$$

So *notcomp*(x) = 0 if either:

- $M_i(x)$ is defined and the value is something other than 0, or
- $M_i(x)$ is not defined.

This new function *notcomp* is in U, but it differs, in at least one place, from every function that is computed by a Turing machine whose encoding is listed in E. So there is no Turing machine that computes it. Thus it is not partially computable.

Busy Beaver Functions

There are functions with straightforward definitions that are not partially computable.

Busy beaver functions

Tibor Radó:

- Hungarian mathematician
- moved to USA after WWI
- lectured at Harvard and Rice, before obtaining a faculty position in the Department of Mathematics at Ohio State
- In 1962, published his most famous results: The Busy Beaver function and its non-computability

Tibor Radó (1895-1965)

Busy Beaver Functions

Let T be the set of all standard TMs that:

- Have tape alphabet Γ = {❑, 1}, and
- Halt on a blank tape.

Define the **busy beaver functions** $S(n)$ and $\Sigma(n)$:

- $S(n)$: the maximum number of steps that are executed by any element of T with n-nonhalting states, when started on a blank tape, before it halts.

- $\Sigma(n)$: the maximum number of 1's that are left on the tape by any element of T with n-nonhalting states, when it halts.

The Busy Beaver Functions

- $S(n)$: the maximum number of steps.
- $\Sigma(n)$: the maximum number of `1`'s.

n	$S(n)$	$\Sigma(n)$
1	1	1
2	6	4
3	21	6
4	107	13
5	$\geq$ 47,176,870	4098
6	$\geq 3*10^{1730}$	$\geq 1.29*10^{865}$

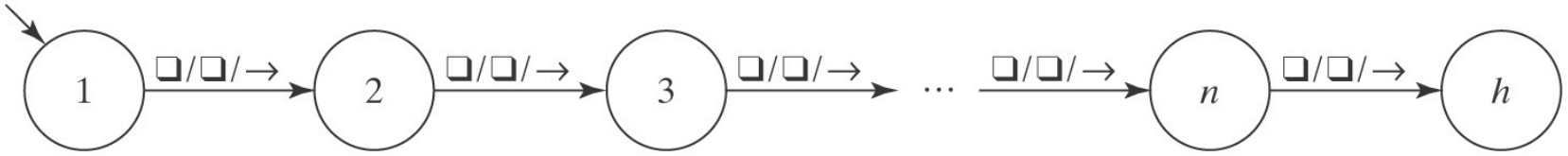
The Busy Beaver Functions are Total

Theorem: Both S and Σ are total functions on the positive integers.

Proof: For any value n, both $S(n)$ and $\Sigma(n)$ are defined iff there exists some standard Turing machine M, with tape alphabet $\Gamma = \{❑, 1\}$, where:

- M has n nonhalting states, and
- M halts on a blank tape.

Name the nonhalting states of M with the integers 1, …, n. Then M is:



Monotonicity

Theorem: Both S and Σ are strictly monotonically increasing functions. In other words:

$$S(n) < S(m) \text{ iff } n < m, \text{ and}$$
$$\Sigma(n) < \Sigma(m) \text{ iff } n < m.$$

Proof: We prove four claims:

- $n < m \to S(n) < S(m)$.
- $S(n) < S(m) \to n < m$.
- $n < m \to \Sigma(n) < \Sigma(m)$.
- $\Sigma(n) < \Sigma(m) \to n < m$.

Computability

Theorem: Neither *S* nor Σ is computable.

Proof that S is not computable: If it were, there would be some TM *BB*, with b states, that computes it. Define:

- For any positive integer *n*, a Turing machine $Write_n$ that writes *n* `1`'s on its tape, one at a time, moving rightwards, and then halts with its read/write head on the blank square immediately to the right of the rightmost `1`. $Write_n$ has *n* nonhalting states plus one halting state.

- *Multiply*, which multiplies two unary numbers, written on its tape and separated by the character #. Let *m* be the number of states in *Multiply*.

Computability

Define *Trouble*$_n$:

>*Write*$_n$; R *Write*$_n$ L$_\square$ *Multiply* L$_\square$ *BB*

Trouble$_n$:

- Writes a string of the form $1^n;1^n$.
- Moves its read/write head back to the left
- Invokes *Multiply*, which results in the tape containing a string of exactly n^2 1's.
- It moves its read/write head back to the left.
- Invokes *BB*, which outputs $S(n^2)$.

The Number of States in $Trouble_n$

>$Write_n$; R $Write_n$ $L_{❑}$ *Multiply* $L_{❑}$ *BB*

Component	Number of States
$Write_n$	$n + 1$
; R	1
$Write_n$	$n + 1$
$L_{❑}$	2
Multiply	m
$L_{❑}$	2
BB	b
Total	$2n + m + b + 7$

The Bottom Line

Since *BB* writes a string of length $S(n^2)$ and it can write only one character per step, $Trouble_n$ must run for at least $S(n^2)$ steps. If $n > 0$, $Trouble_n$ is a TM with $2n + m + b + 7$ states that runs for at least $S(n^2)$ steps. So:

$$S(2n + m + b + 7) \geq S(n^2)$$

S is monotonically increasing, so it must also be true that, for any $n > 0$:

$$2n + m + b + 8 \geq n^2$$

But, since n^2 grows faster than *n* does, that cannot be true.

In assuming that *BB* exists, we have derived a contradiction. So *BB* does not exist. So *S* is not computable.

Recursive Function Theory

A function is computable iff there is a Turing machine that computes it and always halts.

Is there an alternative, nonprocedural definition?

The more traditional terminology:

- ***recursive*** for decidable
- ***recursively enumerable*** for semidecidable
- ***recursive*** for computable
- ***partial recursive*** for partially computable

Note: in some references, computable and recursive are used for partially computable and partial recursive

Why use ***recursive*** as a synonym for ***computable***?

Primitive Recursive Functions

The set of primitive recursive functions is the smallest set that includes:

- The constant function 0.
- $succ(n) = n + 1$.
- A family of projection functions: $(n_1, n_2, \ldots n_k) = n_j$.

and is closed under:

- Composition of g with $h_1, h_2, \ldots h_k$:
  $g(h_1(\ \ ), h_2(\ \ ), \ldots h_k(\ \ ))$

- Primitive recursion of f in terms of g and h:
  $f(n_1, n_2, \ldots n_k, \quad 0) = g(n_1, n_2, \ldots n_k)$
  $f(n_1, n_2, \ldots n_k, m+1) = h(n_1, n_2, \ldots n_k, m, f(n_1, n_2, \ldots n_k, m))$

Examples

$plus(n, 0) = p1,1(n) = n$.
$plus(n, m+1) = succ(p3,3(n, m, plus(n, m)))$.

For clarity, we will simplify our future definitions by omitting the explicit calls to the projection functions. Doing that here, we get:

$plus(n, 0) = n$.
$plus(n, m+1) = succ(plus(n, m))$.

Examples

times(*n*, 0) = 0.
times(*n*, *m*+1) = *plus*(*n*, *times*(*n*, *m*)).

factorial(0) = 1.
factorial(*n* + 1) = *times*(*succ*(*n*), *factorial*(*n*)).

exp(*n*, 0) = 1.
exp(*n*, *m*+1) = *times*(*n*, *exp*(*n*, *m*)).

pred(0) = 0.
pred(*n*+1) = *n*.

Primitive Recursive Functions and Computability

Theorem: Every primitive recursive function is computable.

Proof: Each of the basic functions, as well as the two combining operations can be implemented in a straightforward fashion on a Turing machine or using a standard programming language.

Primitive Recursive Functions and Computability

Theorem: Not all computable functions are primitive recursive.

Proof: Lexicographically enumerate the unary primitive recursive functions, f_0, f_1, f_2, f_3, ….

Define $g(n) = f_n(n) + 1$. g is computable, but it is not on the list. Suppose it were f_m for some m. Then

$f_m(m) = f_m(m) + 1$, which is absurd.

	0	1	2	3	4
f_0					
f_1					
f_2					
f_3				27	
f_4					

Suppose g were f_3. Then $g(3)$ 27 + 1.

A Function that Isn't Primitive Recursive

Ackermann's function:

$A(0, y) = y + 1.$
$A(x + 1, 0) = A(x, 1).$
$A(x + 1, y + 1) = A(x, A(x + 1, y)).$

Ackermann's Function

	0	**1**	**2**	**3**	**4**
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	$2^{65536}-3$ *	$2^{2^{65536}}-3$ #	$2^{2^{2^{65536}}}-3$

* 19,729 digits $3 \cdot 10^{17}$ seconds since Big Bang

\# 10^{5940} digits 10^{79} atoms in observable universe

Thus writing digits at the speed of light on all atoms in the universe starting at the Big Bang would have produced $3 \cdot 10^{96}$ digits.

Minimalization

The ***minimalization*** *f* of a function *g* (of *k* + 1 arguments) is a function of *k* arguments:

$f(n_1,n_2,\ldots n_k)$ = the smallest *m* such that:
$g(n_1, n_2, \ldots n_k, m) = 1$, if there is such an *m*,
0, otherwise.

Given any function *g* and any set of *k* arguments to it, there either is at least one value *m* such that $g(n_1, n_2, \ldots n_k, m) = 1$ or there isn't.

If there is at least one such value, then there is a smallest one. So there always exists a function *f* that is the minimalization of *g*.

Minimalization

If g is computable, then we can build a Turing machine T_{min} that almost computes f as follows:

$T_{\text{min}}(n_1, n_2, \ldots n_k)$ =

1. $m = 0$.
2. While $g(n_1, n_2, \ldots n_k, m) \neq 1$ do:
   $m = m + 1$.
3. Return m.

The Problem with T_{min}

$T_{min}(n_1, n_2, \ldots n_k)$ =

1. $m = 0$.
2. While $g(n_1, n_2, \ldots n_k, m) \neq 1$ do:
   $m = m + 1$.
3. Return m.

T_{min} will not halt if no value of m exists. There is no way for T_{min} to discover that no such value exists and thus return 0.

A function g is ***minimalizable*** iff,

for every $n_1, n_2, \ldots n_k$,

there is an m such that $g(n_1, n_2, \ldots n_k, m) = 1$.

Recursive Functions

A function is ***μ-recursive*** iff it can be obtained from the basic functions:

- The constant function 0.
- *succ*(*n*) = *n* + 1.
- A family of projection functions: (n_1, n_2, … n_k) = n_j,

Using the operations of:

- Composition,
- Recursive definition, and
- Minimalization of minimalizable functions.

Primitive Recursive vs μ-Recursive Functions

- Primitive recursive: Use iteration.
- μ-recursive: Use while loop.

Equivalence of μ-Recursion and Computability

***Theorem**:* A function is μ-recursive iff it is computable.

Proof: By two constructions that show that each can simulate the other:

- Every partial μ-recursive function is partially computable. We show this by showing how to build a Turing machine for each of the basic functions and for each of the combining operations.

- Every partially computable function is partial μ-recursive. We show this by showing how to construct μ-recursive functions to perform each of the operations that a Turing machine can perform.

Partial μ-Recursive Functions

A function is ***partial μ-recursive*** iff it can be obtained from the basic functions:

- the constant function 0,
- the successor function: $succ(n) = n + 1$,
- the family of projection functions: for any $k \geq j > 0$, $p_{k,j}(n_1, n_2, \ldots n_k) = n_j$,

and that is closed under the operations:

- composition of g with $h_1, h_2, \ldots h_k$.,
- primitive recursion of f in terms of g and h, and
- minimalization (of any, possibly nonminimalizable function).

Equivalence of Partial μ-Recursion and Partial Computability

***Theorem**:* A function is partial μ-recursive iff it is partially computable.

Proof: By two constructions that show that each can simulate the other.

Functions and Machines