Recall From Last Time

Definition: We say that S is *reducible* to T, $S \leq T$, iff $\exists$ total, recursive $f : \mathbf{N} \to \mathbf{N}$,

$$(\forall w \in \mathbf{N}) \quad (w \in S) \qquad \Leftrightarrow \qquad (f(w) \in T)$$

[In the future we will insist that $f \in F(\mathbf{L})$.]

Theorem: Suppose $S \leq T$. Then,

1. If T is **r.e.**, then S is **r.e.**.
2. If T is co-**r.e.**, then S is co-**r.e.**.
3. If T is **Recursive**, then S is **Recursive**.

Definition: C is r.e.-*complete* iff

1. $C \in$ **r.e.**, and
2. $(\forall A \in \textbf{r.e.}) \quad (A \leq C)$

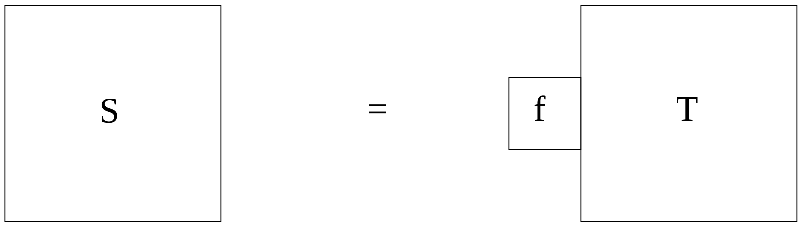
Theorem: K, HALT, and $A_{0,17}$ are r.e. complete, hence not recursive.

Definition: We say that S is *reducible* to T, $S \le T$, iff $f \in F(\mathbf{L})$,

$$(\forall w \in \mathbf{N}) \quad (w \in S) \qquad \Leftrightarrow \qquad (f(w) \in T)$$

Intuition: $S \le T$ iff the placement of a **very simple front end** f before a T-recognizer creates an S-recognizer.

$$\chi_S \quad = \quad \chi_T \circ f, \qquad \text{i.e.,} \quad (\forall x)(\chi_S(x) = \chi_T(f(x)))$$



The Reduction Game: To build a reduction, f, from S to T you must solve the following puzzle:

"For each input, w, what membership question, $f(w)$, can I ask T such that the answer is the membership question $w \in ?S$."

Rice-Myhill-Shapiro Theorem:

Our proof that $A_{0,17}$ was r.e.-complete had little to do with the numbers 0 or 17. A very similar argument can be used to show that "any non-trivial property of Turing machines is undecidable". (See [P], Theorem 3.2, page 62.)

Definition 11.1 Two Turing machines M and N are **equivalent** if for any input x, $M(x)\downarrow$ iff $N(x)\downarrow$ and if both output strings $M(x)$ and $N(x)$ are defined, they are equal. ♠

Definition 11.2 A language is a **property of machines** if for any numbers i and j such that M_i and M_j are equivalent machines, i and j are either both in A or both not in A. ♠

Theorem 11.3 *Let A be a language other than $\emptyset$ or* **N** *that is a property of machines. Then A is not recursive.*

Proof:

Suppose that the (numbers of) machines that never halt are not in A. (If they aren't, we replace A by $\overline{A}$ and prove that the latter is not recursive.) Since A is nonempty, pick some number ℓ so that $M_\ell \in A$.

We will reduce K to A, which means we must define a total recursive function f so that $n \in K$ iff $f(n) \in A$. This means that for any machine M_n, we must build a machine $M_{f(n)}$ that will have the property necessary for A iff M_n accepts n. We'll do this using our assumptions. If M_n accepts n, $M_{f(n)}$ will be equivalent to M_ℓ and thus $f(n)$ will be in A. If M_n does not accept n, $M_{f(n)}$ will never halt and thus $f(n)$ will not be in A.

This is easy. We design $M_{f(n)}$ so that it first runs M_n on n, then (if it finishes that job) runs M_ℓ on the original input. This machine simulates M_ℓ if M_n accepts n and never halts otherwise.

Since $K \leq A$, A cannot be recursive. (We cannot say that A is r.e.-complete because it might not be r.e., in fact "most" such A's are not.)

♠

Primitive Recursive Functions

In Lecture 1 and HW#1 we defined the programming language Bloop, with integer variables and bounded loops. We will now see that the class of functions from **N** to **N** that are implementable in Bloop are a very well studied class called the **primitive recursive functions**.

You may have wondered whether "recursion" as you know it from programming has anything to do with "recursive functions" in this course. The name indeed comes from defining functions recursively. Later this lecture we'll define the **general recursive functions** that are the same as the partial recursive functions. But first we define a less powerful kind of recursion. It defines functions that are guaranteed to halt, but can't define all total recursive functions.

On HW#4 we'll prove that there are recursive functions that are not primitive recursive. So the primitive recursive functions are a *proper* subset of the total recursive functions.

Initial functions:

$\zeta() = 0$

$\sigma(x) = x + 1$

$\pi_i^n(x_1, \ldots, x_n) = x_i, \quad n = 1, 2, \ldots, \quad 1 \leq i \leq n$

Composition: $g_i : \mathbf{N}^k \to \mathbf{N}, 1 \leq i \leq m;\ ; h : \mathbf{N}^m \to \mathbf{N}$:

$$\mathcal{C}(h; g_1, \ldots, g_m)(x_1, \ldots, x_k) \ = \ h(g_1(\overline{x}), \ldots, g_m(\overline{x}))$$

Primitive Recursion: $g : \mathbf{N}^k \to \mathbf{N};\ h : \mathbf{N}^{k+2} \to \mathbf{N}$: $f(n, y_1, \ldots, y_k) = \mathcal{P}(g, h)(n, y_1, \ldots y_k)$, given by:

$$\begin{aligned} f(0, y_1, \ldots y_k) \ &= \ g(y_1, \ldots, y_k) \\ f(n+1, y_1, \ldots y_k) \ &= \ h(f(n, y_1, \ldots, y_k), n, y_1, \ldots, y_k) \end{aligned}$$

Definition 11.4 The **primitive recursive functions (PrimRecFcns)** are the smallest class of functions containing the Initial functions and closed under Composition and Primitive Recursion. ♠

Proposition 11.5 *The following are in* **PrimRecFcns***:*

1. $M_1(x) \ = \ \textbf{if } (x > 0) \textbf{ then } (x - 1) \ \textbf{ else } 0$
2. $x \ominus y \ = \ \textbf{if } (y \leq x) \textbf{ then } (x - y) \ \textbf{ else } 0$
3. $+$
4. $*$
5. $\exp(x, y) \ = y^x$
6. $\exp^*(x) \ = \textbf{if } (x = 0) \textbf{ then } 1 \ \textbf{ else } 2^{\exp^*(x-1)}$

$$\exp^*(x) \ = \ 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\}x$$

Proposition 11.6 *The class* **PrimRecFcns** *is closed under the* **bounded μ-operator**. *If $f(x)$ is a p.r. function, then $\mu\{x < y : f(x) = 0\}$ is defined to be the least x such that $f(x) = 0$ if such an x exists with $x < y$, or to be y otherwise.*

On HW#3 you are asked to relate p.r. functions (in the form of Bloop functions) to Turing machines. A key tool in doing this is the coding of **sequences of numbers** as single numbers, first developed by Gödel. He began with elementary number theory:

Proposition 11.7 Prime, PrimeF $\in$ **PrimRecFcns**, *where,*

$\mathrm{Prime}(x) = \textbf{if}\ (\text{“}x\ \textit{is prime}\text{”})\ \textbf{then}\ 1\ \textbf{else}\ 0$

$\mathrm{PrimeF}(n) =$ *prime number* n, *i.e,* $\mathrm{PrimeF}(0) = 2$, $\mathrm{PrimeF}(1) = 3$, $\mathrm{PrimeF}(2) = 5$, $\mathrm{PrimeF}(3) = 7$, $\mathrm{PrimeF}(4) = 11$.

Proof:

$$\begin{aligned}
x|y &= (\exists z \le y)(xz = y) \\
\mathrm{Prime}(x) &= x > 1 \ \wedge\ (\forall y < x)(y|x \to y = 1) \\
\mathrm{NextPrime}(x) &= \mu\{t \le (x+1)^{x+1} \mid t > x \ \wedge\ \mathrm{Prime}(t)\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{PrimeF}(0) &= 2 \\
\mathrm{PrimeF}(x+1) &= \mathrm{NextPrime}(\mathrm{PrimeF}(x))
\end{aligned}$$

♠

Proposition 11.8

IsSeq, Length, Item $\in$ **PrimRecFcns**, *where,*

$$\begin{aligned}
\text{Seq}(a_0, a_1, \ldots, a_n) &= 2^{a_0+1}3^{a_1+1}\cdots\text{PrimeF}(n)^{a_n+1} \\
\text{IsSeq}(S) &= \textbf{if } (S \textit{ codes a Sequence}) \\
&\qquad\qquad \textbf{then } 1 \textbf{ else } 0 \\
\text{Length}(\text{Seq}(a_0, a_1, \ldots, a_n)) &= n+1 \\
\text{Item}(\text{Seq}(a_0, a_1, \ldots, a_n), i) &= a_i
\end{aligned}$$

Proof:

$$\begin{aligned}
\text{Good}(x, S) &= (\forall y < S)((y < x \wedge \text{PrimeF}(y) | S) \\
&\qquad \vee (y \geq x \wedge \text{PrimeF}(y) \not| S))
\end{aligned}$$

$$\text{IsSeq}(S) = (\exists x < S)\text{Good}(x, S)$$

$$\text{Length}(S) = \mu\{x < S \mid \text{Good}(x, S)\}$$

$$\begin{aligned}
\text{Item}(S, i) &= \mu\{y < S \mid \text{IsSeq}(S) \ \wedge \ \text{PrimeF}(i)^{y+1} | S \\
&\qquad \wedge \text{PrimeF}(i)^{y+2} \not| S\}
\end{aligned}$$

♠

As your intuition about Bloop should begin to tell you, almost anything computable can be computed in Bloop. (On HW#4 you'll find a t.r. function that can't.) In particular, we can simulate Turing machines:

Primitive Recursive COMP Theorem: [Kleene]
Let $\mathrm{COMP}(n, x, c, y)$ mean $M_n(x) = y$, and that c is M_n's complete computation on input x.
Then COMP is a Primitive Recursive predicate.

Proof: We will encode TM computations:

$$c = \mathrm{Seq}(\mathrm{ID}_0, \mathrm{ID}_1, \ldots, \mathrm{ID}_t)$$

Where each ID_i is a sequence number of tape-cell contents:

$$\mathrm{ID}_i = \mathrm{Seq}(\triangleright, a_1, \ldots, a_{i-1}, [\sigma, a_i], a_{i+1}, \ldots, a_r)$$

$$\begin{aligned}
&\mathrm{COMP}(n, x, c, y) \;\equiv \\
&\quad \mathrm{START}(\mathrm{Item}(c, 0), x) \;\wedge\; \mathrm{END}(\mathrm{Item}(c, \mathrm{Length}(c) - 1), y) \;\wedge \\
&\qquad (\forall i < \mathrm{Length}(c))\mathrm{NEXT}(n, \mathrm{Item}(c, i), \mathrm{Item}(c, i + 1))
\end{aligned}$$

♠

Primitive Recursive = Bloop

We have two sets of functions from $\mathbf{N}^k$ to $\mathbf{N}$, each defined recursively:

- $\zeta()$, $\sigma(x)$, and $\pi_i^n(x_1, \ldots, x_n)$ are p.r.,
- the composition of p.r. functions is p.r., and
- the function made from two p.r. functions by the primitive recursion rule is p.r.

- "++" statements are Bloop program blocks
- an assignment of a call-by-value function call is a block
- the concatenation of two blocks is a block
- a variable-bounded loop containing a block is a block

To be more precise about Bloop we would have to be more careful about a formal semantics. Esssentially, given a binding of some variables at the start of a block, we get a new binding at the end, and the exact transformation of the binding could be defined by induction. Here we'll rely on our intuitions about real programs.

Theorem 11.9 *Every primitive recursive function can be implemented in Bloop.*

Proof:

Base Cases:

```
declare zeta() {
   return x;}

declare sigma(x) {
   return x++;}

declare pi_2_4 (x1, x2, x3, x4) {
   return x2;}
```

Composition Example:

$f(x, y) = g(h_1(x, y), h_2(x, y), h_3(x, y))$

```
    declare f(x,y) {
       a = h1(x,y);
       b = h2(x,y);
       c = h3(x,y);
       return g(a,b,c);}
```

Primitive Recursion Example: $f(0, y, z) = g(y, z)$, $f(n + 1, y, z) = h(f(n, y, z), n, y, z)$

```
    declare f(n,y,z) {
       a = g(y,z);
       i = zeta();
       for (n) {
          a = h(a,i,y,z);
          i++;}
       return a;}
```

♠

Theorem 11.10 *The output of any Bloop program is a primitive recursive function.*

What is easier to prove by induction is the following:

Theorem 11.11 *After any Bloop program block, any variable defined at the end is a primitive recursive function of those variables defined at the beginning.*

Proof:

Successor: If the block is "`xi++`", and the variables defined are $\{x_1, \ldots, x_n\}$, then the end value of x_j for $j \neq i$ is $\pi^n_j(x_1, \ldots, x_n)$ and the end value of x_i is $\sigma(\pi^n_i(x_1, \ldots, x_n)$.

Assignment: If the block is "`x = f(y1,...,yk)`" then by the IH since f is defined elsewhere, it is a p.r. function of the y's. By projections we can make x a p.r. function of all defined variables, and other variables are unchanged.

Concatenation: If the block is “`B;C`” where B and C are blocks, by the IH every variable y_i defined after B is a p.r. function of those variables x_i defined before B, and every variable z_i defined after C is a p.r. function of the y’s. By composition we write each z_i as a p.r. function of the x’s.

Loops: Suppose the block is “`for (n) B;`” where B is a block, and let us first consider the case where only one variable y is defined during B. By the IH the effect of B on y is given by a p.r. function $b(y)$.

We define $f(n, y_0)$ to be the value of y after B has been executed n times starting from $y = y_0$. Clearly we can define f by primitive recursion, with $g(y_0) = y_0$ and $h(f(n, y_0), n, y_0) = b(f(n, y_0))$.

Now say we have k different variables defined during B. We use the sequence tools developed above. Let C be a block that takes its single argument, decodes it into k variables, runs B on them, and codes the k answers as a single number. By composition, C’s effect on its variable is p.r., and by the argument above so is that of “`for (x) C;`”. Encoding and decoding gives us p.r. functions for “`for (x) B;`”.

♠

General Recursive Functions

We will now increase the power of the primitive recursive functions by adding one more rule:

Definition 11.12 If $f(x, y_1, \ldots, y_k)$ is a function, we define the μ-**operator** on f with respect to x. $\mu\{x : f(x, y_1, \ldots, y_k)\}$ is the following partial function $g(y_1, \ldots, y_k)$. If there is a value of x such that $x < y$ and $f(x, y_1, \ldots, y_k) = 0$, then $g(y_1, \ldots, y_k)$ is the least such value. If there is no such value of x, $g(y_1, \ldots, y_k)$ is undefined. ♠

Definition 11.13 The **general recursive functions** are the least set of partial functions containing the initial functions and closed under composition and the μ operator. ♠

Definition 11.14 **Floop** is the programming language consisting of Bloop augmented with one more statement type. If B is a block, possibly changing the value of x, "`while (x) B`" is a block that keeps executing B as long as x is positive. (It may run forever – if so its behavior is a strictly partial function.) ♠

Theorem 11.15 *A partial function from* $\mathbf{N}^k$ *to* $\mathbf{N}$ *is general recursive iff it is computed by a Floop program.*

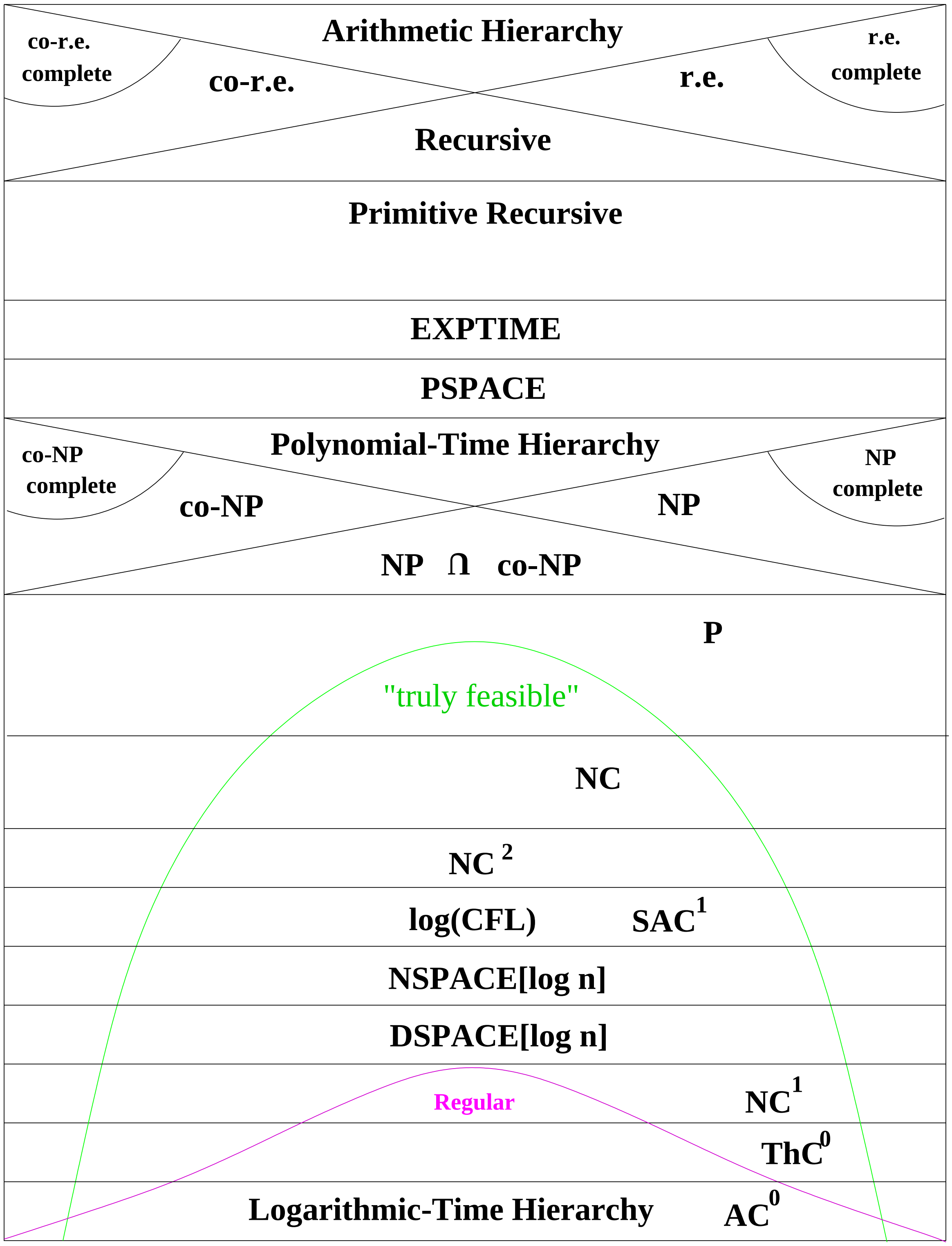
Proof: Exercise (HW#4). ♠

Theorem 11.16 *A partial function from* $\mathbf{N}$ *to* $\mathbf{N}$ *is general recursive iff it is partial recursive.*

Proof:

- $\rightarrow$: Inductively simulate Floop blocks by TM's.
- $\leftarrow$: Let f be a partial recursive function. Then $f(x) = y$ iff there is a halting computation of f's TM on input x yielding y. Using the COMP predicate and the μ operator we construct a general recursive function that outputs $f(x)$ if it is defined.

♠


Arithmetic Hierarchy
co-r.e. complete
co-r.e.
r.e.
r.e. complete
Recursive
Primitive Recursive
EXPTIME
PSPACE
Polynomial-Time Hierarchy
co-NP complete
co-NP
NP
NP complete
NP ∩ co-NP
P
"truly feasible"
NC
NC 2
log(CFL)
SAC1
NSPACE[log n]
DSPACE[log n]
Regular
NC1
ThC0
Logarithmic-Time Hierarchy
AC0