6.044J/18.423J: Computability, Programming, and Logic Handout 29
Massachusetts Institute of Technology 1 December 1989

Notes on Programming (Part I)

The following notes will serve as the main text for the remainder of the course. The goal of this final unit will be to work with a language that in some sense "captures" the essential features of the programming language Scheme and other applicative functional languages. Our language, called FKS, we claim is the functional kernel of Scheme. The syntax FKS will be basically that of the simply-typed λ-calculus, with a single base type ι which we will take to be the natural numbers. One of the most important properties of FKS that makes it possible for us to analyze in this class is that it has **NO SIDE EFFECTS**.

We will present definitions of the language at several levels. Our first level will be that of rewrite rules. Rewrite rules, via an immediate reduction relation between pieces of code, specify how, at a high level, programs can be evaluated. It will take a program M and in one step reduce it to another program that in some sense will be closer to what we would like to call *the answer*. Although rewrite rules provide a wonderful way of defining a language, the way in which they work is very far from a reasonable implementation strategy. We will present an automaton, called a SECD machine, which will reduce the task of interpreting our language to that of basic, well-understood pointer manipulations. This SECD machine will be very close in spirit to the way in which functional languages are actually implemented. Bridging the gap between the SECD machine and the rewrite rules we will present *eval* a recursive characterization of the rewrite rules. All of these definitions with respect to a language $\mathcal{L}$ are referred to as the operational semantics of $\mathcal{L}$. In this case our language will be "FKS", and we will provide a definition of the semantics operationally, via rewrite rules.

We will also present a denotational semantics for FKS. The motivation behind this sort of semantics is the desire to say that the meaning of a piece of code that computes a certain function *is* the function which it computes. The goal of denotational semantics is to develop an interpretation (which we will from now on call a *model*) of all of the terms (fragments of code) in the language. Unlike first order logic, where an interpretation can be any first order structure, we will limit our consideration of semantic interpretations for FKS to a single model. In the field of denotational semantics, a model is specified by two entities: a domain (just like in logic), and a *meaning* function which maps syntactic elements of the language into the domain. This meaning function does a job analogous to that of the interpretation function operating on terms. In logic $\mathcal{I}(t)$ (referred to as the meaning or *denotation* of term t in interpretation $\mathcal{I}$) was an object in $D_{\mathcal{I}}$ that was implicitly defined by giving the denotations of the constant and function symbols of the language for which $\mathcal{I}$ is a model. In devising a model for FKS, however, we need to explicitly define this meaning function which when given a term yields that term's denotation. The most logical question to ask at this point is "what good is this model?" The answer is a lot. For example we will show that if a program M evaluates to a constant c then the model will assign the same object to both M and to c we will also see that the converse is true, namely that if M and c have the same denotation then M will evaluate to c. We will then show as a corollary to this that if M and N have the same denotation, then (not considering time or space issues) these two pieces of code are completely interchangeable—a very nice property to be able to state. Wouldn't it be nice if after you optimize a piece of code in an already working system you could then prove rigorously that the new code was functionally equivalent to the old code? This is what denotational semantics can do for you...

Given this cultural background it is now time to consider the task immediately at hand. In order to define FKS, we will first define its syntax. We will then define its semantics operationally by a set of rewrite rules. This combination of syntax and semantics will fully define the language FKS. We will then present alternate operational definitions for the semantics of a language with the same syntax (but not necessarily the same semantics) as FKS. We will then sketch a proof that the semantics of these alternate definitions coincide with that of the rewrite rules—thus they define the same language. Finally, we will provide a denotational definition of a language over that same syntax. We will then argue that the semantics defined denotationally coincides in a nice way with the operational semantics.

1 Syntax of FKS

What is a term. The basis of the syntax for FKS is what is called the simply typed λ-calculus. Fragments of code in this framework are referred to as terms. A term in this framework is analogous to the code that appears between a balances set of parenthesis in unsugared Scheme, or a constant symbol, or a non-binding occurrence of a variable.

Example 1. The following is a short fragment of Scheme code:

$$((\text{lambda } (x)\ (*\ x\ x))\ 5)$$

The terms in this program are: $5, x, (*\ x\ x), (\text{lambda } (x)\ (*\ x\ x))$, and $((\text{lambda } (x)\ (*\ x\ x))\ 5)$. Note: the x immediately after the lambda is not really a term. Scheme's notation is not optimal. Scheme should have been defined so that code would have been written something like:

$$((\text{lambda } x.(*\ x\ x))\ 5)$$

Types. Unlike Scheme every term in our language will all have an associated type. The set of types is called **Types**; we define it inductively as follows:

- ι is a type. It will correspond to our notion of the natural numbers.
- If σ is a type and τ is a type then $\sigma \rightarrow \tau$ is a type. It will correspond to functions which take a single argument of type σ and return a single result of type τ.

Example 2. When writing a type we will let $\rightarrow$ associate *right*. Thus $\sigma_1 \rightarrow (\sigma_2 \rightarrow \sigma_3)$ is the same type as $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3$, which is distinctly different from $(\sigma_1 \rightarrow \sigma_2) \rightarrow \sigma_3$.

The following are some basic functions, and their types.

1. SQUARE$\overset{\text{def}}{=}$ (lambda (x) ($*$ x x)). SQUARE is of type $\iota \rightarrow \iota$, as it is a function that takes a natural number as an argument and returns a natural number as a result.
2. 5. 5 is of type ι.
3. APP5$\overset{\text{def}}{=}$ (lambda (foo) (foo 5)). APP5 is of type $(\iota \rightarrow \iota) \rightarrow \iota$. It takes a a function from natural numbers to natural numbers, and then returns a natural number. For example (APP5 SQUARE) is 25.
4. PLUS1$\overset{\text{def}}{=}$ (lambda (foo) (lambda (x) (+1 (foo x)))). PLUS1 is of type $(\iota \rightarrow \iota) \rightarrow (\iota \rightarrow \iota)$. It takes a function from natural numbers to natural numbers and returns a function from natural numbers to natural numbers. For example (PLUS1 SQUARE) is a function that returns its argument squared, plus 1.

Note that this language does not contain any booleans, characters, reals, strings, lists or other such structures. We claim that this simple type hierarchy with only the base type ι is the core of Scheme.

Notice that our set of types does not include "pair" types. For example the binary plus operator which we are familiar with from arithmetic takes two arguments, both of type ι and returns one value, also of type ι. So we would write 2 plus 3 as (+ 2 3), and + has type $(\iota \times \iota \rightarrow)\iota$. In our framework we will not have pair types. But, via a process called "currying", we will show that there is no loss of generality in not having pair types. Before giving the formal definition of currying, we will provide as an example a definition of a curried version of plus ($+_c$) defined in terms of the standard plus:

$$+_c \overset{\text{def}}{=} (\lambda x(\lambda y((+\ x\ )y)))$$

So consider the two ways we would now write the expression for 2 plus 3:

$$\text{curried: } ((+_c\ 2)\ 3)$$

$$\text{uncurried: } (+\ 2\ 3)$$

Note that the type of $+_c$ is $\iota \rightarrow \iota \rightarrow \iota$. Thus $(+_c\ 2)$ is of type $\iota \rightarrow \iota$ and represents the "plus 2 function". In general, given an arbitrary function foo of type $\sigma = (\sigma_1 \times \ldots \times \sigma_n) \rightarrow \sigma'$ we can easily curry it to foo_c which works right. The function foo_c will have type $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma'$. It is defined from foo as follows:

$$foo_c \overset{\text{def}}{=} (\lambda x_1^{\sigma_1} \ldots \lambda x_n^{\sigma_n} (foo\ x_1^{\sigma_1} \ldots x_n^{\sigma_n}))$$

The last comment to be made is that every type σ is of the form $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma'$ this type will occasionally be abbreviated as $\sigma = (\sigma_1, \ldots, \sigma_n, \sigma')$.

Terms. Now that we know what types are, we are ready to define what terms are. Terms and their types are defined inductively as follows:

- x^σ is a term of type sigma. (representing a variable of type σ)
- c^σ is a term of type sigma. (representing a constant of type σ)
- (MN) is a term of type τ if M is a term of type $\sigma \rightarrow \tau$ and N is a term of type σ.
- (cond M N_1 N_2) is a term of type ι and M, N_1, and N_2 are all terms of type ι.
- $(\lambda x^\sigma.\ M)$ is a term of type $\sigma \rightarrow \tau$ if M is a term of type τ.

Note that we are very informal with parenthesis and as with arithmetic we will define certain conventions that allows most parenthesis to be eliminated. The conventions are as follows:

- Application associates left. Thus (MNP) is the same as $((MN)P)$, which is very different from $(M(NP))$.
- λ's bind out as far as they can (*e.g.*until the end of the expression or overridden by a parenthesis). So $(\lambda\ x.\ M\ N)$ is the same as $(\lambda\ x.\ (M\ N))$ which is distinctly different from $((\lambda\ x.\ M)N)$.
- Never drop the parenthesis that surrounds "cond M N_1 N_2 ."
  Thus (cond M N_1 N_2) is correct, and cond M N_1 N_2 will only lead to confusion.

We will call any term which a constant, variable or λ-abstraction a *value*. A term of the form (MN) is can be called either a *combination* or an *application*. A term of the form (cond...) is called a *conditional*.

Aside from presenting the set of constants, and their affiliated types, this completely defines the syntax of FKS.

Unsugaring Scheme. So terms are either variables, constants, applications (something of the form (MN)), conditional expressions (something of the form (cond M N_1 N_2)), or λ-abstractions (something of the from $(\lambda x^\sigma.\ M)$. Applications can be viewed as function calls, conditional expressions can be viewed as our only special form, and λ-abstraction can be viewed as procedure construction. Remember from 6.001 that most of the friendly structures in Scheme are syntactic sugar for other forms.

Example 3. The most prominent case of syntactic sugar is "let". For example:

(let ((*var*1 *exp*1)) *exp*)

is syntactic sugar for

((lambda (*var*1) *exp*) *exp*1)

Example 4. Another example of syntactic sugar is "define" (when used to define a function that is not recursive). For example:

(define (*foo arg*) body)

is syntactic sugar for:

(define *foo* (lambda (*arg*) body))

Considering that FKS does not allow side-effects, and it does not have lists (or streams or other fancy stuff) this syntax really is almost as expressive as full fledged Scheme. Two missing elements of the syntax require further justification. These two problems are as follows: conditionals can only return objects of type ι, and the lack of "define". We will argue later on in Example 5 that there is a straightforward way to program a "higher-type" conditional from the one given here. On the surface we can argue away "define" by saying that since we have no side effects then the following:

(define P_1 B_1)
(define P_2 B_2)
⋮
(define P_n B_n)
body

is equivalent to

$$(\text{let } ((P_1\ B_1)\ (P_2\ B_2) \ldots (P_n\ B_n))\ \text{body}$$

Since "let" is only sugar, "define" is only sugar.

We have, however, slipped something very important under the rug. In Example 4, the unsugaring of (define $(foo\ arg)$ body) assumed that foo was not recursive. We will introduce a special constant Y that will allow for the unsugaring of recursive procedures, and in Sections 2 and 2 we will show how Y can be used to generate recursive and mutually recursive functions.

Constants. The constants of FKS and their types are as follows:

- $0, 1, 2, \ldots$ all of type ι
- *succ*, *pred* both of type $\iota \rightarrow \iota$
- Y_σ of type $(\sigma \rightarrow \sigma) \rightarrow \sigma)$ (for all types $\sigma \neq \iota$)

Free and Bound variables. In this section we will write the variable x^σ simply as x. When we do not need to discuss the type of a bound variable, it is sometimes convenient to drop the superscript. We must be careful, however, for example if we use x^ι and $x^{(\iota \rightarrow \iota)}$ as distinct variables, we need to be careful when we abbreviate one by x so as to prevent confusion.

We will define the function $FV : Terms \rightarrow \{Variables\}$ So, if M is a term, it has a set $FV(M)$ of free variables. It is defined inductively on the structure of the term M by:

- $FV(x) = \{x\}$
- $FV(c) = \{\}$
- $FV((MN)) = FV(M) \cup FV(N)$
- $FV(\text{cond } M\ N_1\ N_2\ ) = FV(M) \cup FV(N_1) \cup FV(N_2)$
- $FV((\lambda x\ M)) = FV(M) \backslash \{x\}$

A term is *closed* iff $FV(M) = \emptyset$, otherwise it is *open*.

A function $BV : Terms \rightarrow \{Variables\}$ can be defined analogously to give the bound variables of a term.

The free variables of a term are simply those variables that are not under the scope of a λ-abstraction over that variable. For example y is free in $(\lambda x.\ x\ y)$

since there is no λ binding it. The bound variables of a term are those variables that are bound by λ's. For example y is bound in both $(\lambda y.\ y)$ and $(\lambda y.\ 5)$. Note that it is possible for a variable to occur both free and bound in the same term. For example y is both free and bound in $(y(\lambda y.\ (x\ y)))$. The free occurrences of y in a term M are those occurrences of y in M that are not bound. The bound occurrences of y in a term M are partitioned (divided into disjoint sets that cover everything) into occurrences bound by an individual λ-abstraction.

A *program* is a closed term of ground type. Since our only ground type is ι, a *program* is a closed term of type ι.

Given an infinite list $x_1, \ldots$ of distinct variables (which we will assign types when we use them), the substitution prefix is defined inductively in the structure of terms as follows:

- $x[x := M] = M$; $y[x := M] = y\ (if\ x \neq y)$
- $a[x := M] = a$
- $(NN')[x := M] = (N[x := M])\ (N'[x := M])$
- $(\text{cond } N\ N_1\ N_2)[x := M] = (\text{cond } N[x := M]\ N_1[x := M]\ N_2[x := M])$
- $(\lambda x N)[x := M] = (\lambda x N)$; $(\lambda y N)[x := N] = \lambda z N[y := z][x := M]$, if $x \neq y$, where z is the variable defined by:
  1. If $x \notin FV(N)$ or $y \notin FV(M)$ then $z = y$.
  2. Otherwise, z is the first variable in the list $x_1, x_2, \ldots$ such that $z \notin FV(N) \cup FV(M)$—and z is made to have the same type as y.

We now introduce a relation $=_\alpha$in order to capture the notion that two terms are equivalent up to the renaming of bound variables. It is defined inductively in the structure of terms as follows: The relation $=_\alpha$of alpha equivalence, is defined inductively by:

1. $x =_\alpha x$
2. $a =_\alpha a$.
3. If $M =_\alpha M'$ and $N =_\alpha N'$ then $(MN) =_\alpha (M'N')$.
4. If $M =_\alpha M'$ and $N_1 =_\alpha N_1'$ and $N_2 =_\alpha N_2'$ then $(\text{cond } M\ N_1\ N_2) =_\alpha (\text{cond } M'\ N_1'\ N_2')$
5. If $M =_\alpha M'[y := x]$, where either $x = y$ or $x \notin FV(M')$ then $(\lambda x M) =_\alpha (\lambda y M')$.

Contexts. A *context* is a term with one or more "holes" in it. It is normally written in the from $C[\cdot]$. The term that results from filling the term M into all of the holes in the context $C[\cdot]$ is written as $C[M]$. $C[\cdot]$ is called a program context (implicitly with respect to the term M iff $C[M]$ is a program.

This concludes the definition of the syntax of FKS.

2 Operational Semantics: Rewrite Rules

One way of providing an operational semantics for a language is via a set of rewrite rules. From the rewrite rules we will arrive at a partial function *Eval* from Programs (closed terms of type ι) to constants. *Eval* is defined by means of an immediate reduction relation, $\rightarrow$ between terms by:

Eval:

$$Eval(M) = k \text{ iff } M \twoheadrightarrow k, \text{for any program } M \text{ and constant } k.$$

Where $\twoheadrightarrow$ is the reflexive transitive closure of $\rightarrow$ (sometimes written as $\xrightarrow{*}$).

It will be the case that $M \twoheadrightarrow k$ and $M \twoheadrightarrow k'$ implies that k and k' are identical. Notice that for constants c, $Eval(c) = c$.

The following rules together are called *rewrite rules* and together they define the desired immediate reduction relation $\rightarrow$.

1. (a) $(succ\ n) \rightarrow (n+1)$
   (b) $(pred\ 0) \rightarrow 0$
   (c) $(pred\ (n+1)) \rightarrow n$
   (d) $(Y_\sigma V) \rightarrow (V(\lambda x^\sigma (Y_\sigma V) x^\sigma))$ (where $x \notin FV(V)$)
2. (a) $(\lambda x V) \rightarrow M[x := V]$ (for V a value)
   (b) (cond 0 N_1 N_2) $\rightarrow N_1$
   (c) (cond $(n+1)$ N_1 N_2) $\rightarrow N_2$
3. (a) if $M \rightarrow M'$ then $(MN) \rightarrow (M'N)$
   (b) if $N \rightarrow N'$ then $(VN) \rightarrow (VN')$ (for V a value)
   (c) if $M \rightarrow M'$ then (cond M N_1 N_2) $\rightarrow$ (*cond* M' N_1 N_2)

That is all. We have just defined a language completely. We have given a definition of the syntax, and we have given an operational definition of what it means for a program P to evaluate to a constant c. This is operational in the sense that it is sort of how a compiler works, and it made no appeal to semantic domains or models.

Some Useful Definitions. A term M is said to *diverge* iff $mEval(M)$ is undefined. This is equivalent to saying that for all N if $M \twoheadrightarrow N$ then their is a term N' such that $N \rightarrow N'$.

We now introduce a notion of equality between terms that is generated by the rewrite rules. This notion is called observational equality ($\equiv_{obs}$)[1] and it captures the idea of the interchangeability of code. Two pieces of code are observationally equivalent if they can be exchanged freely in FKS programs with out changing the value to which the program evaluates. This is defined formally as follows:

$$N \equiv_{obs} M$$

iff for all program contexts

$$Eval(C[M]) \simeq Eval(C[N])$$

Two objects are $\simeq$ iff either they are both undefined, or they are both defined and equal. There is a "Context Lemma" for FKS that says that:

$$\text{if } M \twoheadrightarrow N \text{ then } Eval(C[M] \simeq Eval(C[N]))$$

Thus $M \twoheadrightarrow N$ implies $M \equiv_{obs} N$. This "Context Lemma" will be a Corollary of the Adequacy Theorem which we will prove later. [2]

Now, in order to fully appreciate the richness of this language we provide some examples of coding tasks.

Example 5. Before we can do much of anything, we really need a "macro" which will enable us to do a higher order conditional. Something we can abbreviate into the form ($cond_\sigma$ M N_1 N_2) where N_1 and N_2 both have type σ. In this side-effect free, typed world of ours, we have no need for a conditional where N_1 and N_2 have different types.

Supposing $\sigma = (\sigma_1, \ldots, \sigma_n, \sigma')$ $cond_\sigma$ is:

$$(cond_\sigma\ M\ N_1\ N_2) \overset{\text{def}}{=} (\lambda x_1^{\sigma_1} \cdots \lambda x_n^{\sigma_n}.\ (\text{cond } M(N_1 x_1 \cdots x_n)(N_2 x_1 \cdots x_n)))$$

It is a rather grungy, but manageable task to show that ($cond_\sigma$ 0 N_1 N_2) behaves just like (cond_σ 0 N_1 N_2) would if it were in the language (except that if ($cond_\sigma$ 0 N_1 N_2) is computed, but never used, then if N_1 diverges a program using the cond_σ would also diverge, but a program using the $cond_\sigma$ might still evaluate to a value).[3]

[1] It is important to note that $\equiv_{obs}$is an equivalence relation.

[2] One way of stating the Adequacy Theorem is: "If M and N have the same denotation, then $M \equiv_{obs} N$."

[3] If you did not understand that parenthetic remark it is ok, this is a watered down version of a concept called *observational approximation* which we will might cover later. For your information, a term M observationally approximates a term N iff for all program contexts $C[\cdot]$, $C[M] \twoheadrightarrow c$ implies $C[N] \twoheadrightarrow c$. In this setting we can say that M is observationally congruent to N iff M observationally approximates N and N observationally approximates M.

Example 6. FKS does not include constants or special forms which allow the pairing of objects (we do not have cons). This lack is easily overcome, since using λ-terms we can define combinators that act like typed pairing operators. We will define "macros" which will provide typed pairing operators for us. Thus $pair_\sigma$ pairs together two objects of type σ. The macros $left_\sigma$ and $right_\sigma$ will unpair the result of pairing together two objects of type σ. These three macros are defined as follows:

- $pair_\sigma(M, N) \stackrel{\text{def}}{=} (\lambda z^{(\sigma\to\sigma\to\sigma)}.\ M\ N)$. Where this object is of type: $(\sigma \to \sigma \to \sigma) \to \sigma)$.
- $left_\sigma(P) \stackrel{\text{def}}{=} (P(\lambda x^\sigma.\lambda y^\sigma.x))$
- $right_\sigma(P) \stackrel{\text{def}}{=} (P(\lambda x^\sigma.\lambda y^\sigma.y))$

It is a straightforward task to check the correctness of these definitions and that $left(pair(M, N)) = M$ and that $right(pair(M, N)) = N$.

It is fairly simple to generalize this technique to allow the pairing of terms which do not have the same type. This can be done many ways. One such way is to coerce the arguments to be of the same type. If M has type $\sigma_1 = (\sigma_1, \ldots, \sigma_n, \iota)$ and N has type $\tau = (\tau_1, \ldots, \tau_m, \iota)$ then we can coerce M into M' and N into N', M' and N' of the type $\sigma' = (\sigma_1, \ldots, \sigma_n, \tau_1, \ldots, \tau_m, \iota)$ So

$$M' \stackrel{\text{def}}{=} \lambda x_1^{\sigma_1} \ldots \lambda x_n^{\sigma_n} \lambda y_1^{\tau_1} \ldots \lambda y_m^{\tau_m} (M x_1^{\sigma_1} \ldots x_n^{\sigma_n})$$

and

$$N' \stackrel{\text{def}}{=} \lambda x_1^{\sigma_1} \ldots \lambda x_n^{\sigma_n} \lambda y_1^{\tau_1} \ldots \lambda y_m^{\tau_m} (M x_1^{\tau_1} \ldots y_m^{\tau_m})$$

It should be clear how to recover terms equivalent to the original M and N from M' and N' through appropriate abstraction and application to dummy terms.

From here on we will assume that we have a "smart" macro system which will do the appropriate coercions and such and we will assume we have a single macro for each of pair, left, and right that does what we would like it to do (pair will coerce its args to the right type, but left and right will not "uncoerce" their results).

Finally it should be clear how to make n-tuples. Call the operation $tuple_n^\sigma$ which makes an n-tuple out of objects of type $sigma$. It should be equally clear how to define the projection on the i^{th} component—$proj_{(n,i)}^\sigma$. These can be defined either directly (introducing new expressions using λ's for each) or they can be defined from $pair$, $left$ and $right$.

Debunking Recursion: Fixed Points. An important element of any functional language is the ability to define a function recursively. Consider, for example the standard definition of the factorial function in Scheme:

$$\text{(define } fact\ (\lambda\ n)\ \text{(cond (= } n\ 0)\ 1\ (*\ n\ (fact\ (-1\ n)))))$$

Unfortunately the syntax of Scheme does not make it immediately obvious that *fact* is a recursive function (actually a "simply recursive"[4]). The syntax of Common Lisp, however, does make this fact evident. Consequently, we will draw the examples of defining recursion from Common Lisp instead of Scheme (it should be immediately obvious how to convert from one definition to the other). So *fact* would be defined in Common Lisp as follows:

$$\text{(letrec } fact = \text{(lambda (}n\text{) (cond (= } n\ 0)\ 1\ (*\ n\ (fact\ (-1\ n))))))$$

This syntax for defining a recursive function makes it immediately clear that *fact* is a simply recursive function. Now consider the form of a mutually recursive definition of functions $f_1, \ldots, f_n$ by bodies $b_1, \ldots, b_n$ where each body b_i has n "holes". We write $b_i[f_1, \ldots, f_n]$ to denote b_i with its hole #1 filled by f_1,...,#n filled by f_n. So the definitions of $f_1, \ldots, f_n$ would be as follows:

$$\begin{array}{ll} \text{(letrec} & \\ \quad (f_1 = & b_1[f_1, \ldots, f_n]) \\ \quad \ldots & \\ \quad (f_n = & b_n[f_1, \ldots, f_n]) \\ &) \end{array}$$

Now that we have a syntax that makes it clear what is being defined recursively, we can go ahead and explain how recursive functions can be defined in FKS. Let us consider the following function:

$$F \overset{\text{def}}{=} \text{(lambda (}f\text{) (lambda (}n\text{) (cond (= } n\ 0)\ 1\ (*\ n\ (f\ (-1\ n))))))$$

F has a very interesting property, namely $F(fact) = fact$. For no other argument x is $F(x) = x$. There is a special name for this property, namely, *fact* is a *fixed point* of F. In mathematics, if you have any function G and object x, x is called a fixed point of G iff $G(x) = x$. Let fix be a "fixed point operator", namely a function which returns a fixed point of its argument. Then *fact* could be defined as $fact \overset{\text{def}}{=} (fix\ F)$. Thus any recursive definition of the form:

$$\text{(letrec } foo\ =\ \text{body}[foo])$$

[4] A function *foo* is called ***simply recursive*** iff it is recursive and its definition does not use another function whose definition depends (either directly or indirectly) on itself. This is to be contrasted with a ***mutually recursive*** function (or set of functions) where the definition of f_1 uses f_2 and f_2 either directly or indirectly uses f_1.

can be thought of as:

$$(\text{let } foo \;=\; (\text{fix } (\lambda\ (f)\ \text{body}[f]))$$

Where now foo is no longer defined in terms of itself.

Luckily, FKS has a fixed point operator, namely Y (actually Y is a "least" fixed point operator, but it is beyond the scope of this section to explain that here). Let us now look at how to define $fact$ in FKS. First we need to translate F into FKS. So:

$$F \overset{\text{def}}{=} (\lambda f^{(\iota\to\iota)} \lambda n^{\iota}.\ (\text{cond } (=\ n)\ 0\ 1)\ ((*\ n)\ (f\ (pred\ n))))$$

Finally we can now define $fact$:

$$fact \overset{\text{def}}{=} (Y_{(\iota\to\iota)} F)$$

So, this is how to define a simply recursive function. Using tupling we can define mutually recursive functions. This is a slight modification of the preceding example of what mutually recursive definitions should look like. The difference is that we will use b_i' where:

$$b_i' \overset{\text{def}}{=} (\lambda h.(\lambda g_1 \ldots \lambda g_n.\ (b_i[g_1,\ldots,g_n])\ proj_{n,1}(h) \cdots proj_{n,n}(h)))$$

and the final letrec is:

$$\begin{array}{lll}
(\text{letrec} & & \\
 & (f_1 = & (b_1'\ tuple_n(f_1,\ldots,f_n)) \\
 & \ldots & \\
 & (f_n = & (b_n'\ tuple_n(f_1,\ldots,f_n)) \\
 & &)
\end{array}$$

So we have simplified the definitions of each of the functions to be of the form $f_i = b_i' foo$ where foo is $tuple_n(f_1,\ldots,f_n))$. So if we can define an expression which generates foo then we are done (because $f_i = proj_{n,i}(foo)$). So we want to define a function F that has foo as its least fixed point. Here is is:

$$F \overset{\text{def}}{=} (\lambda H.\ tuple_n((b_1'\ H),\ldots,(b_n'\ H)))$$

So in conclusion, we have:

$$f_i \overset{\text{def}}{=} proj_{n,i}(Y\ F)$$

Debunking Y: The Fixed Point Operator. Up until now we have taken the tack that Y is a fixed point operator. We have not, however justified this statement. Using our rewrite rules we can check that Y is a fixed point operator. Our goal would to prove a statement analogous to $F(fix\ F) = F)$. For FKS this statement takes the form $(V\ (Y\ V))\equiv_{obs}(Y\ V)$, where V can be any value that is not of type ι (thus probably, a λ-abstraction). But look:

$$(Y_\sigma V) \rightarrow (V(\lambda x^{(\sigma\rightarrow\sigma)}.(Y_\sigma V)x))\ \ ($$

for

$$x^\sigma \notin FV(V))$$

You can check yourself the following is a general rule:

If M is of type $\sigma \neq \iota$, M does not diverge, and $x^\sigma \notin FV(M)$, then $M\equiv_{obs}(\lambda x^\sigma.Mx)$

Since $(Y_\sigma\ V)$ and x^σ meet the antecedent of this rule, then

$$(Y_\sigma\ V)\equiv_{obs}(\lambda x^\sigma.(Y\ V)\ x)$$

Since we can interchange terms which are $\equiv_{obs}$to each other with impunity, we have:

$$(V\ (Y_\sigma\ V))\equiv_{obs}(V\ (\lambda x^\sigma.(Y_\sigma\ V)\ x))$$

and, given that $\equiv_{obs}$is an equivalence relation:

$$(V\ (Y_\sigma\ V))\equiv_{obs}(Y_\sigma\ V)$$

this is the desired result.

CONTINUED ON THE NEXT PAGE

Example 7. To really see how Y works in defining recursion, consider the computation of (fact 2), where:

$$\begin{aligned}
F &\overset{\text{def}}{=} (\lambda f^{(\iota\rightarrow\iota)}.\lambda n^{\iota}.(\text{cond } (=\ n\ 0)\ 1\ (*\ n\ (f\ (-1\ n)))))\\
fact &\overset{\text{def}}{=} (Y_{(\iota\rightarrow\iota)}F)\\
\text{body1} &\overset{\text{def}}{=} (\lambda n^{\iota}.(\text{cond } (=\ n\ 0)\ 1\ (*\ n\ (f\ (-1\ n)))))\\
\text{body2} &\overset{\text{def}}{=} (\text{cond } (=\ n\ 0)\ 1\ (*\ n\ (f\ (-1\ n))))\\
\text{foo} &\overset{\text{def}}{=} (\lambda x^{(\iota\rightarrow\iota)}.\ (Y\ F)\ x)
\end{aligned}$$

Here is the evaluation, in hideously gory detail (we will use $\equiv$ to mean syntactic equality):

$$\begin{aligned}
(fact\ 2) &\equiv ((YF)2)\\
&\rightarrow ((F(\lambda x.(YF)x)2)\\
&\rightarrow (\text{body1}[f := foo]2)\\
&\rightarrow \text{body2}[f := foo][n := 2]\\
&\equiv (\text{cond } (=\ 2\ 0)\ 1\ (*\ 2(foo(pred2))))\\
&\rightarrow (\text{cond } (=_2\ \ 0)\ 1\ (*\ 2(foo(pred2))))\\
&\rightarrow (\text{cond } 1\ 1\ (*\ 2(foo(pred2))))\\
&\rightarrow (*\ 2(foo(pred2)))\\
&\rightarrow (*_2\ (foo(pred2)))\\
&\equiv (*_2\ ((lambdax^{\cdots}.\ (Y\ F)\ x)\ (pred\ 2)))\\
&\rightarrow (*_2\ ((lambdax^{\cdots}.\ (Y\ F)\ x)\ 1))\\
&\rightarrow (*_2\ ((Y\ F)\ 1)\\
&\rightarrow (*_2\ (F\ (\lambda x.\ (Y\ F)\ x)\ 1))\\
&\rightarrow (*_2\ (\text{body1}[f := foo]\ 1))\\
&\rightarrow (*_2\ \text{body2}[f := foo][n := 1]\\
&\equiv (*_2\ (\text{cond } (=\ 1\ 0)\ 1\ (*\ 1(foo\ (pred\ 1)))))\\
&\rightarrow (*_2\ (\text{cond } (=_1\ \ 0)\ 1\ (*\ 1(foo\ (pred\ 1)))))\\
&\rightarrow (*_2\ (\text{cond } 1\ 1\ (*\ 1(foo\ (pred\ 1)))))\\
&\rightarrow (*_2\ (*\ 1\ (foo\ (pred\ 1))))\\
&\rightarrow (*_2\ (*_1\ (foo\ (pred\ 1))))\\
&\equiv (*_2\ (*_1\ ((lambdax^{\cdots}.(Y\ F)\ x)\ (pred\ 1))))\\
&\rightarrow (*_2\ (*_1\ ((lambdax^{\cdots}.(Y\ F)\ x)\ 0)))\\
&\rightarrow (*_2\ (*_1\ ((Y\ F)\ 0)\\
&\rightarrow (*_2\ (*_1\ (F\ (\lambda x.\ (Y\ F)\ x)\ 0))
\end{aligned}$$

$$
\begin{aligned}
&\rightarrow (*_2\ (*_1\ (\text{body1}[f := foo]\ 0)) \\
&\rightarrow (*_2\ (*_1\ \text{body2}[f := foo][n := 0] \\
&\equiv (*_2\ (*_1\ (\text{cond}\ (=\ 0\ 0)\ 1\ (*\ 0(foo\ (pred\ 0)))))) \\
&\rightarrow (*_2\ (*_1\ (\text{cond}\ (=_0\ \ 0)\ 1\ (*\ 0(foo\ (pred\ 0)))))) \\
&\rightarrow (*_2\ (*_1\ (\text{cond}\ 0\ 1\ (*\ 0(foo\ (pred\ 0))))) \\
&\rightarrow (*_2\ (*_1\ 1)) \\
&\rightarrow (*_2\ 1) \\
&\rightarrow 2
\end{aligned}
$$

Example 8. Definability of the basic operation "+" (in curried form). The curried form of "+" has type $\iota \rightarrow \iota \rightarrow \iota$.

$$+ \overset{\text{def}}{=} (Y^{(\iota\rightarrow\iota\rightarrow\iota)}\ (\lambda f^{(\iota\rightarrow\iota\rightarrow\iota)}\lambda x^{\iota}\lambda y^{\iota}.\ (\text{cond}xy(\text{succ}((f(\text{pred}x))y))))$$

A good exercise to enhance your understanding of how the rewrite rules and recursion works would be to hand evaluate ((+3)10). Another good exercise would be to define * and exp in this manner.

Example 9. Definability of a Primitive Recursion operator PR.

Recall the definition of primitive recursion. An $n+1$-ary function h can be defined by primitive recursion from an n-ary function f and an $n+2$-ary function g as follows:

- $h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n)$
- $h(x_1, \ldots, x_n, (succy)) = g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y))$

I am going to provide a definition of a primitive recursion operator PR_1 for the case of $n = 1$. It is straightforward to then define PR_n for all n. PR_1 has type: $\sigma_f \rightarrow \sigma_g \rightarrow \sigma_h$, where $\sigma_f = \iota \rightarrow \iota$, $\sigma_g = \iota \rightarrow \iota \rightarrow \iota \rightarrow \iota$, and $\sigma_h = \iota \rightarrow \iota \rightarrow \iota$. So, here is the definition:

$$
\begin{aligned}
PR_1 \overset{\text{def}}{=}\ & \lambda f^{\sigma_f}\lambda g^{\sigma_g}. \\
& \quad Y_{\sigma_h}(\lambda h^{\sigma_h}\lambda x^{\iota}\lambda y^{\iota}. \\
& \qquad\quad (\text{cond}\ y\quad (f\ x) \\
& \qquad\qquad\qquad (((g\ x)\ (\text{pred}y))\ (h\ x(\text{pred}y)))))
\end{aligned}
$$

Example 10. At this point it should be clear how to program all of the primitive recursive functions in FKS (at least their curried versions). To check this, we simply need to verify that we have all of the basic functions and operations needed to define them. Let's go through a check:

- We have the successor function built in.
- The zero function is: $(\lambda x^{\iota}.0)$
- The identity functions can be defined analogous to the pairing operators defined earlier. In general $ID_i^n = (\lambda x_1^{\iota} \ldots \lambda x_n^{\iota}.\, n_i)$.
- Composition of m n-ary functions. Again the set of combinators $CN_{m,n}$ is easily λ-definable. It is left as an exercise.
- Primitive recursion has just been given.

So we can code any primitive recursive function of n-arguments by a function in FKS of type:

$$\underbrace{\iota \to \ldots \to \iota}_{n\ times} \to \iota$$

This ability to code all of the functions of a class like this in our language is called *numeralwise representability*. We say that the all of the primitive recursive functions are *numeralwise representable* FKS.

Example 11. Now that we know that all of the primitive recursive functions are numeralwise representable in FKS, we would like to show that all of the partial recursive functions are numeralwise representable in FKS.

This simply involves demonstrating that we can code the set of minimization operations Mn_n. We will define Mn_1. It is of type: $\sigma_f \to \iota \to \iota$, where $\sigma_f = \iota \to \iota$. In the definition we will use: $\sigma_h = \iota \to \iota$. So, here it is:

$$\begin{array}{lll} Mn_1 & \overset{\text{def}}{=} & (\lambda f^{\sigma_f} \lambda x^{\iota}. \\ & & \quad ((Y_{\sigma_h}(\lambda h^{\sigma_h} \lambda y^{\iota}. \\ & & \qquad\qquad (\text{cond}\ ((f\ x)\ y) \quad y \\ & & \qquad\qquad\qquad\qquad\qquad (h\ (succ\ y))))) \\ & & \quad 0)) \end{array}$$

And, we are done.

Thus FMS can define all of the partial recursive functions, so it is just as powerful as any of the other models of computation which we have seen. In addition, it is not too powerful. Although it may not be immediately clear how you could write an interpreter for FKS, the SECD machine, which will begin the next section of notes, operates by very simple pointer manipulations and will obviously be implementable via a turing machine or μ-recursive functions.

Notes on Programming (Part III)

by Arthur Lent

This handout assumes that you are familiar with the material contained in handout #29 (Part I of these notes). In particular you need to know the definition of terms, FV, BV, the substitution operation, $=_\alpha$, and the rewrite rules.

1 The SECD Machine

Introduction. The SECD machine was first introduced by Landin in 1963[1]. The importance of the machine's introduction was that it was the first time that a language was described abstracted away from a particular implementation—in the past, a language was defined by the first compiler written for it. The SECD machine takes a role between the further abstraction of rewrite rules and the concreteness of an actual implementation—making explicit the control structure of evaluation, yet still leaving unspecified arbitrary details.

The treatment here is taken largely from a work by Plotkin in 1975[2].

What is a SECD Machine. SECD stands for Stack–Environment–Control-string–Dump. The precise technical definition of each of these terms will be given in the next section. First, we wish to cultivate some intuitions about the SECD machine. The SECD machine is an automaton whose basic actions are pointer manipulations. In fact, its basic actions involve a constant number of pointer manipulations. Once you see the full definitions it should be immediately clear how to built an interpreter for FKS. You should be able to do it in about an afternoon. You could not say the same thing about the rewrite rules presented in the first section of the notes. Not only does this description lend itself much more to being implemented, an implementation based directly on the SECD machine is far more efficient than one based upon the rewrite rules. Yet this does not mean that the rewrite rules are without value—they probably make a more understandable operational definition of FKS, and they will be essential for our work on denotational semantics.

Before we give technical definitions of the four components of the SECD machine we will tell what each piece will be used for:

Stack The stack will be used to store intermediate results in the evaluation of a function.

Environment The environment is the environment in which the function is being evaluated.

Controlstring The controlstring contains whatever operations still need to be performed in order to complete the current function call.

Dump The dump stores the state of the machine that existed immediately prior to the current function call. (It can be viewed as a stack of the preceding activation records).

Definitions. The first new concepts to be defined are the sets **Environments** and **Closures**. These correspond very closely to the notions defined in 6.001. In fact, they are the same notions, but, in a language without side effects, they have some additional nice properties. But first we should back up a step and address why the SECD machine needs environments and closures. It uses them in order to implement substitution. The process of turning an arbitrary term M into the term $M[x := N]$ is nontrivial. The traditional way of doing substitution is to say that the term $M[x := N]$ can represented by M paired with an *environment* an object that states that x really is N. M paired with this environment is called a closure, and it "represents" the term $M[x := N]$. Now what if we want to substitute $(M[x := N])$ for y into P to get the term $P[y := (M[x := N])]$. We want to represent this by the closure $[P, E]$ where $E(y) = (M[x := N])$. Unfortunately, we do not actually have our hands on the term $M[x := N]$—we have our hands on a closure representing it. Thus we do not really want environments to map from variables to terms, but, rather, we want them to map from variables to closures. This may look like a circular definition, but then so does defining two functions in a mutually recursive way. Here is a simultaneous mutual inductive definition of the set **Closures** of closures, the set **Value Closures** of value closures and **Environments** of environments:

- $\emptyset$, is an abbreviation for the totally undefined function. It is an environment.
- A partial function with finite domain, that maps variables of type σ to value closures of type σ is an environment.
- If E is an environment, and M of type σ such that $FV(M) \subseteq Domain(E)$ then $[M, E]$ is a closure of type σ. (In other words, E is defined on all of the free variables of M).
- If M is a value of type σ and $[M, E]$ is a closure then $[M, E]$ is a value closure.

Note that any closed term M can be represented by the closure $[M, \emptyset]$.

We also define $E\{Cl/x\}$ (x and Cl must have the same type) to be the unique environment E' such that $E'(y) = E(y)$ if $y \neq x$ and $E'(x) = Cl$ (for any $Cl \in$ **Closures**).

Finally, to drive home the point that a closure can mechanically be "unwound" into the term it represents we define a function Realterm : **Closures** → **Terms**. It is defined inductively by:

$$\text{Realterm}([M, E]) = M[x_1 := \text{Realterm}(E(x_1))] \ldots [x_n := \text{Realterm}(E(x_n))]$$

where

$$FV(M) = \{x_1, \ldots, x_n\}$$

Note that this unwinding property is only possible when there are no side-effects in the terms inside the closures being unwound, so for Scheme it will not work, but there are many cases where closures *are* built up from terms without side-effects, in which case you really can think of those closures in terms of this unwinding process.

The set of stacks, **Stacks**, is the set of all finite sequences of closures, formally written as **Stacks** = (**Closures**)*.

The set of controlstrings, **Controlstrings** = (**Terms** $\cup$ ap, cd)* where ap, cd are special symbols that are not elements of **Terms**. The function FV is easily extended to work on controlstrings as follows:

- $FV(ap) = \emptyset$
- $FV(cd) = \emptyset$
- $FV(C_1, \ldots, C_n) = \bigcup_{i=1}^{n} FV(C_i)$ $(n \geq 0)$

Finally, the set of dumps, **Dumps**, is defined inductively by:

- $nil \in$ **Dumps**
- If $S \in$ **Stacks**, $E \in$ **Environments**, $C \in$ **Controlstrings** and C is such that $FV(C) \subseteq Domain(E)$, and $D \in$ **Dumps** then $[S, E, C, D] \in$ **Dumps**

This concludes the definitions of the primary data structures manipulated by the SECD machine.

The functions constapply and Constapply. The SECD machine model which we will present in the next section will be defined in a manner that abstracts away from the constants that we have chosen to include in FKS. Thus we could, in principle, add constants to the language (such as a curried plus operator) and the main proofs about the SECD machine would carry through directly. In addition this abstraction separates out the "constant stuff" for a particular language from the general principles of interpreting a functional language.

The SECD machine will employ the function **Constapply** when it hits an operator that is a constant. Since constant operators (namely Y) in our language can take values as arguments, and values are general terms which might need closures to fully define them, Constapply needs to be a partial function of the type:

$$\textbf{Constants} \times \textbf{Closures} \rightarrow \textbf{Value Closures}$$

We will also be presenting a recursive characterization of the rewrite rules which does not use stacks, closures, environments, or dumps, yet still captures explicitly the order of evaluation of the SECD machine. But for this recursive characterization, which we will from now on call *eval*, we still need a **Constapply** sort of function, but it must live wholly in the world of terms (no closures allowed). We will call it **constapply**, and it needs to be a partial function of type:

$$\textbf{Constants} \times \textbf{Closed Values} \rightarrow \text{Closed Terms}$$

In actuality we will not define **Constapply** directly. Instead we will define **constapply**, and then state that **Constapply** is as determined as it needs to be by the following restriction:

$$\text{Realterm}(\textbf{Constapply}(a, Cl)) =_\alpha \textbf{constapply}(a, \text{Realterm}(Cl))$$

What this requires is that **Constapply** gives a result that is independent of how the closure that is its argument represents the term it is acting upon. So **Constapply** cannot distinguish between:

$$[x, \{[x, (MN)]\}]$$

and

$$[(x\ y), \{[x, M], [y, N]\}]$$

Here is the definition of **constapply** which we will use for FKS:

$$\begin{array}{lllll}
\text{succ} & \textbf{constapply}(\text{succ}, \boldsymbol{n}) & \rightarrow & \boldsymbol{n+1} \\
\text{pred} & \textbf{constapply}(\text{pred}, \boldsymbol{n+1}) & \rightarrow & \boldsymbol{n} \\
 & \textbf{constapply}(\text{pred}, \mathbf{0}) & \rightarrow & \mathbf{0} \\
Y_\sigma & \textbf{constapply}(Y_\sigma, V) & \rightarrow & (V(\lambda x^{(\sigma\rightarrow\sigma)}.(Y_\sigma V)x)) \\
 & & & \text{(for an } x \notin FV(V) \\
 & & & \text{and for } V \text{ a value} \\
 & & & \text{and for } \sigma \neq \iota)
\end{array}$$

Order of evaluation. The SECD machine will evaluate the operands of a combination before the operators. This is to be contrasted with the order of the rewrite rules which evaluate operators before operands. We apologize for this confusion, and it should be obvious how to change the rewrite rules or the SECD machine so that the order is reversed. Our main theorem, however, carries through whichever order used in whichever scheme. This is because, for FKS, the order does not matter (in fact for a particular scheme expression, where the evaluating the operator or operands does not produce any side effects, the order does not matter). Note that the definition of Scheme does not even specify which is the correct order. The following is an exerpt from the Scheme manual given to 6.001 students in Spring '89:

"A procedure call is written by enclosing in parenthesis expressions for the procedure to be called and the arguments to be passed to it. The operator and operand expressions are evaluated **(in an indeterminate order)** and the resulting procedure is passed the resulting arguments... Procedure calls are also called *combinations*" (emphasis added).

The function SECD. The state transition function, a partial function from **Dumps** to **Dumps** is defined as follows:

1. $[Cl : S, E, nil, [S', E', C', D']] \Rightarrow [Cl : S', E', C', D']$
2. $[S, E, x : C, D] \Rightarrow [E(x) : S, E, C, D]$
3. $[S, E, a : C, D] \Rightarrow [[a, \emptyset] : S, E, C, D]$
4. $[S, E, (\lambda x.\ M) : C, D] \Rightarrow [[(\lambda x.\ M), E] : S, E, C, D]$
5. $[[(\lambda x.\ M), E'] : Cl : S, E, ap : C, D] \Rightarrow [nil, E'\{Cl/x\}, M, [S, E, C, D]]$
6. $[[a, \emptyset] : [V, E''] : S, E, ap : C, D] \Rightarrow [nil, E', M', [S, E, C, D]]$
   (where **Constapply**$(a, [V, E'']) = [M, E'])$
7. $[S, E, (MN) : C, D] \Rightarrow [S, E, N : M : ap : C, D]$
8. $[S, E, (\text{cond } M N_1 N_2) : C, D] \Rightarrow [[N_1, E] : [N_2, E] : S, E, M : cd : C, D]$
9. $[[\mathbf{o}, E_0] : [N_1, E_1] : [N_2, E_2] : S, E, cd : C, D] \Rightarrow [S, E_1, N_1 : C, D]$
10. $[[[\boldsymbol{n} + \mathbf{1}, E_0] : [N_1, E_1] : [N_2, E_2] : S, E, cd : C, D] \Rightarrow [S, E_2, N_2 : C, D]$

We now need two functions **Load** and **Unload** which convert terms into SECD machine state, and SECD machine state into terms. Specifically they are defined by:

$$\textbf{Load}(M) = [nil, \emptyset, M, nil]$$

$$\mathbf{Unload}([Cl, \emptyset, nil, nil]) = \mathrm{Realterm}(CL)$$

We can now define an evaluation function, which is a partial function from terms to values as follows:

$$\mathrm{SECD}(M) = V \text{ iff } \mathbf{Load}(M) \overset{*}{\Rightarrow} D, \text{ and } V = \mathbf{Unload}(D) \text{ for some dump D}$$

The punchline of the section on SECD machines is the following theorem:

Theorem 1. SECD(M)$=_\alpha N$ iff $Eval(M)=_\alpha N$ for all terms M and N.

Remember that M and N are $=_\alpha$ iff they differ only in the names of their bound variables—called α-equivalence or *equal up to renaming of bound variables.*

In order to prove this theorem we will introduce another scheme for evaluating programs that is midway between the rewrite rules and the SECD machine. This will be a simple recursive definition that uses substitution rather than closures.

We would to like to find a (partial) function *eval* : **Closed Terms** → **Values** such that:

$$eval(a) = a; \quad eval(\lambda x M) = \lambda x M$$

$$eval(MN) = \begin{cases} eval(M'[x := N']) & \text{(if } eval(M) = \lambda x M' \\ & \text{and } eval\text{(N)=N')} \\ eval(\mathbf{constapply}(a, N')) & \text{(if } eval(M) = a \\ & \text{and } eval(N) = N') \end{cases}$$

$$eval(\text{cond } M\ N_1\ N_2) = \begin{cases} eval(N_1) & \text{(if } eval(M) = \mathbf{0}) \\ eval(N_2) & \text{(if } eval(M) = \boldsymbol{n} + \mathbf{1}) \end{cases}$$

Now this may look like a good recursive definition of a partial function, and it turns out that it *is* good, but the precise sense in which equational recursive definitions of partial functions work requires avoiding some mathematical pitfalls, which we must not take for granted. So, to be perfectly precise about how *eval* is defined, we define the predicate "M evals to N at stage t" by induction on t, for closed terms M and closed values N

1. a evals to a at stage 1; $(\lambda x M)$ evals to $(\lambda x M)$ at stage 1.
2. If M evals to $(\lambda x M')$ at stage t and N evals to N' at stage t' and $[N'/x]M'$ has value L at stage t'' then (MN) evals to L at stage $t + t' + t'' + 1$.
3. If M evals to $\mathbf{0}$ at stage t and N_1 evals to L at stage t' then (cond M N_1 N_2) evals to L at stage $t + t' + 1$. If M evals to $\boldsymbol{n} + \mathbf{1}$ at stage t and N_2 evals to L at stage t' then (cond M N_1 N_2) evals to L at stage $t + t' + 1$.

4. If M evals to a at stage t and N evals to N' at stage t' and if constapply(a,N') is defined and evals to N'' at stage t'', then (MN) evals to N'' at stage $t + t' + t'' + 1$.

It is a fairly simple induction on t to show that for all M, there is *at most one* pair (N, t) such that M evals to N at stage t. Consequently this is a good definition of a partial function:

$$eval(M) = N \text{ iff } M \text{ evals to } N \text{ at some stage.}$$

There is a better way of defining *eval*via an inference relation $eval_r$; however, time has not permitted working out this better definition.

Proving the equivalence of SECD and *eval*. In before we prove Theorem 1 we first prove the following, easier Theorem (notice the little "e"):

Theorem 2. SECD(M)$=_\alpha N$ iff $eval(M)$$=_\alpha N$ for all terms M and N.

This theorem will be proven using three lemmas. The first says using closures and environments to model substitution "works right". The second will prove direction ⇒ of this theorem, and the third will prove direction ⇐ of this theorem.

Lemma 1. Suppose $[\lambda y.\ M, E]$ and $[N, E']$ are value closures. Also suppose that Realterm($[\lambda y.\ M, E]$)$=_\alpha(\lambda x.M')$ and

Realterm($[N, E']$)$=_\alpha N'$. Then Realterm($[M, E\{[N, E']/y\}]$)$=_\alpha M'[x := N']$.

Proof Sketch: Observe that if $\lambda y.\ M =_\alpha \lambda x.\ M'$ then $M =_\alpha M'[x := y]$, hence $M[y := N] =_\alpha M'[x := N]$. The rest is a simple unwinding of the closures and simply examining the definition of Realterm. ■

The proof of this next Lemma captures how the SECD machine really works. It is quite long, however, thus we will leave out the details of a few of the cases.

Lemma 2. Suppose E is an environment and $[M, E]$ is a closure. Suppose Realterm($[M, E]$) evals to M''. Suppose C is a controlstring with $FV(C) \subseteq Domain(E)$. Then there is a $t' \geq t$, such that for all S, D,

$$[S, E, M : C, D] \overset{t'}{\Rightarrow} [[M', E'] : S, E, C, D]$$

where $[M', E']$ is a value closure and Realterm($[M', E']$)$=_\alpha M''$.

This Lemma really does entail the right hand direction of Theorem 2, but requires in its statement a rather hefty induction hypothesis.

Proof: This is a proof by induction on t. It is quite similar to that presented by Plotkin [2]. There are 5 main cases:

1. M is a constant. Here Realterm($[M, E]$) $= M = M''$ and $t = 1$. As
$$[S, E, M : C, D] \Rightarrow [[M, \emptyset] : S, E, C, D]$$
we can take $[M', E'] = [M, \emptyset]$ and $t' = 1$.
2. M is a λ-abstraction. Almost the same as the previous case.
3. M is a variable. Take $[M', E'] = E(M)$, and $t = 1$.
4. $M = (\text{cond } P\ N_1\ N_2)$ is a conditional. Apply the inductive hypothesis to P and then divide by cases according to P' the value that P evals to at stage t_1
5. $M = (M_1\ M_2)$ is a combination. Then
$$\begin{aligned} \text{Realterm}([M, E]) &= (\text{Realterm}([M_1, E])\ \text{Realterm}([M_2, E])) \\ &= (N_1\ N_2) \text{ say.} \end{aligned}$$
This now divides into two subcases, depending on whether or not the value to which N_1 evals to is a λ-abstraction or a constant.
   (a) $(\lambda x.\ N_3)$ is the value that N_1 evals to at stage t_1, N_4 is the value that N_2 evals to at stage t_2, M'' is the value that $N_3[x := N_4]$ evals to at stage t_3 and $t = t_1 + t_2 + t_3 + 1$.
   Then by the induction hypothesis there are $t'_i \geq t_i$ $(i = 1,\ 2)$ such that:
$$\begin{aligned} [S, E, (M_1\ M_2) : C, D] &\Rightarrow [S, E, M_2 : M_1 : ap : C, D] \\ &\overset{t'_1}{\Rightarrow} [[M'_2, E'_2] : S, E, M_1 : ap : C, D] \\ &\overset{t'_2}{\Rightarrow} [[M'_1, E'_2] : [M'_2, E'_2] : S, E, ap : C, D] \end{aligned}$$
   where
$$\text{Realterm}([M_1, E_1]) =_\alpha (\lambda x.\ M) \text{ and } \text{Realterm}([M'_2, E'_2]) =_\alpha N_4,$$
   and the $[M'_i, E'_i]$ are value closures.
   Here $M'_1 = (\lambda y.\ M'_3)$ for some M'_3, and
$$\text{Realterm}([M'_3, E'_1\{[y := [M'_2, E'_2]]\}]) =_\alpha [N_4/x]N_3 \text{ (by Lemma 1).}$$

Now,

$$\begin{aligned}[][M'_1, E'_1] : \quad & [M'_2, E'_2] : S, E, ap : C, D] \\ & \Rightarrow [nil, E'_1\{[M'_2, E'_2]/y\}, M'_3, [S, E, C, D]] \\ & \overset{t'_3}{\Rightarrow} [[M', E'], E'_1\{[M'_2, E'_2]/y\}, nil, [S, E, C, D]] \\ & \Rightarrow [[M', E'] : S, E, C, D]\end{aligned}$$

where, by the induction hypothesis, Realterm($[M', E']$) is to within α-equivalence the value that Realterm($[M'_3, E'_1\{[M'_2, E'_2]/y\}]$) evals to at stage $t_3 \leq t'_3$ and $[M', E']$ is a value closure. Taking $t' = t'_1 + t'_2 + t'_3 + 3$ concludes this subcase.

(b) a is the value that N_1 evals to at stage t_1, V is the value that N_2 evals to at stage t_2. Then, by the inductive hypothesis there are $t'_i \geq t_i$ (i=1, 2) , and a value closure VC such that:

$$\begin{aligned}[S, E, (M_1\ M_2), C, D] & \Rightarrow & [S, E, M_2 : M_1 : ap : C, D] \\ & \overset{t'_2}{\Rightarrow} & [VC : S, E, M_1 : ap : C, D] \\ & \overset{t'_1}{\Rightarrow} & [[a, \emptyset] : VC : S, E, ap : C, D]\end{aligned}$$

where Realterm(VC) $= V$. Now, finally, suppose that we have **Constapply**(a, VC) $= [M'', E'']$, and N'' is the value to which Realterm($[M'', E'']$) evals at stage t_3 (thus N'' is the value to which **constapply**(a, Realterm(VC)) evals at stage t_3). By the induction hypothesis there are t'_3 and VC' such that:

$$\begin{aligned}[S, E, (M_1\ M_2), C, D] & \overset{t'_1+t'_2+1}{\Rightarrow} & [[a, \emptyset] : VC : S, E, ap : C, D] \\ & \Rightarrow & [nil, E'', M'', [S, E, C, D]] \\ & \overset{t'_3}{\Rightarrow} & [VC', E'', nil, [S, E, C, D]] \\ & \Rightarrow & [VC' : S, E, C, D]\end{aligned}$$

where Realterm(VC') $= N''$. Then taking $t' = t'_1 + t'_2 + t'_3 + 3$ and $[M', E'] = VC'$ concludes the proof of the lemma.

■

Before we introduce the next lemma, we need a definition. If $D \overset{t}{\Rightarrow} D'$, where D' does not have the form $[Cl, \emptyset, nil, nil]$ and $D' \not\Rightarrow D''$ for any D'' then D is said to *hit an error state* (viz. D').

Lemma 3. Suppose E is a value environment and $[M, E]$ is a closure. If Realterm($[M, E]$) does not eval to a value at any $t' \leq t$, then either for all S, C, D, with $FV(C) \subseteq Domain(E)$, $[S, E, M : C, D]$ hits an error state or else $[S, E, M : C, D] \overset{t}{\Rightarrow} D'$ for some D'.

Proof Sketch: This is proved by induction on t—the number of steps used by the SECD machine to evaluate M. It is just a horrible counting exercise that can just be grunged through. ■

Proof: **(Theorem 2).** Suppose $eval(M) = M''$. Then at some stage t, M'' is the value that M evals to at stage t. By lemma 2,

$$[nil, \emptyset, M, nil] \overset{t'}{\Rightarrow} [[M', E'], \emptyset, nil, nil],$$

where Realterm($[M', E']$)$=_\alpha M''$. So $Eval(M) =_\alpha M''$.

Suppose, on the other hand, that M does not eval to a value at any stage. Then by Lemma 3 either $[nil, \emptyset, M]$ hits an error state or else for every t there is a D such that $[nil, \emptyset, M, nil] \overset{t}{\Rightarrow} D$. In either case SECD($M$) is also not defined. ■

Proving the equivalence of eval and Eval.

Theorem 3. For all well-typed, closed terms M with constants in **Constants** then $M \twoheadrightarrow M'$ (M' a value) iff M evals to M' at some stage t ($eval(M) = M'$).

But first we need several facts:

Fact 1. $\rightarrow$ is deterministic. That is: if $M \rightarrow M'$ then $\not\exists M'' \neq M'$ such that $M \rightarrow M''$. Thus if $M \xrightarrow{n} M''$, $M \xrightarrow{m} M'$ and $m \leq n$ then $M' \xrightarrow{n-m} M''$.

Fact 2. If $M_1 \xrightarrow{n} M_1'$ then $(M_1 M_2) \xrightarrow{n} (M_1' M_2)$ and $(aM_1) \xrightarrow{n} (aM_1')$

Fact 3. If M is a closed value, then $(cM) \rightarrow$ **constapply**(c, M) which is to say that if **constapply**(c,M) is defined then (cM) reduces to it, and if **constapply**(c,M) is not defined then $\not\exists M' : (cM) \rightarrow M'$.

Proof: $(M \xrightarrow{n} M' \Rightarrow eval(M) = M')$. By induction on n.

Basis. $n = 0$. M is a constant c, or M is an abstraction $(\lambda x N)$. In either case $M = M'$ and M evals to M' at stage 1.

Inductive Step. M is a **combination**, say $(M_1 M_2)$. For $(M_1 M_2) \twoheadrightarrow M'$, a value, then it must be the case that $M_1 \xrightarrow{n_1} M_1'$, and $M_2 \xrightarrow{n_2} M_2'$, where M_1' and M_2' are values. By Fact 2, $(M_1 M_2) \xrightarrow{n_1} (M_1' M_2) \xrightarrow{n_2} (M_1' M_2')$. The proof now breaks down into two cases depending on what kind of value M_1' is.

1. $M_1' = \lambda x N$. Then
$$(M_1M_2)\xrightarrow{n_1+n_2}((\lambda xN)M_2') \to (N[x := M_2'])\xrightarrow{n-(n_1+n_2+1)}M'.$$
By the inductive hypothesis then $eval(M_1) = \lambda xN$, $eval(M_2) = M_2'$, and $eval(N[x := M_2']) = M'$. Thus:
$$eval(M_1M_2) = eval(N[x := M_2']) = M'.$$

2. $M_1' = c$.
$$(M_1M_2)\xrightarrow{n_1+n_2}(cM_2)\xrightarrow{n_2}(cM_2') \to \ \mathbf{constapply}(c, M_2')\xrightarrow{n-(n_1+n_2+1)}M'$$
By the inductive hypothesis:
$$eval(M_1) = c,\ eval(M_2) = M_2',\ \text{and } eval(\mathbf{constapply}(c, M_2')) = M'.$$
Thus:
$$eval(M_1M_2) = eval(\mathbf{constapply}(c, M_2')) = M'$$

The case for when M is a conditional is left as an exercise. ■

Proof: **(M evals to M' at stage $t \Rightarrow M \twoheadrightarrow M'$)**. By induction on t.

Basis. $t = 1$. $M = M'$, and is either a constant or an abstraction. In either case $M \xrightarrow{0} N$ and we are done.

Inductive Step. $t > 1$. M is neither a constant nor an abstraction so it must be an application or conditional. We consider the case of an application, that of the conditional is left as an exercise. So $M = (M_1M_2)$.

M_1 must eval to a value at stage some $t_1 \leq t-2$. So say M_1 evals to M_1' at stage t_1. Then by the induction hypothesis $M_1 \twoheadrightarrow M_1'$, and then $(M_1M_2) \twoheadrightarrow (M_1'M_2)$. In addition M_2 must eval to a value at some stage $t_2 \leq t - (t_1 + 1)$. So say M_2 evals to M_2' at stage t_2. Then by the induction hypothesis $M_2 \twoheadrightarrow M_2'$, thus $(M_1'M_2) \twoheadrightarrow (M_1'M_2')$.

The analysis now breaks down into 2 cases based upon M_1'.

1. $M_1' = \lambda xN$. In this case $N[x := M_2]$ evals to M' at stage $t - (t_1 + t_2)$. But then
$$\begin{array}{rcl} M = (M_1M_2) & \twoheadrightarrow & (\lambda xN)M_2 \\ & \to & N[x := M_2] \end{array}$$
and by the inductive hypothesis $N'[x := M_2] \twoheadrightarrow M'$ and so $M \twoheadrightarrow M'$.

2. M_1' is a constant. Let $N = \mathbf{constapply}(M_1', M_2')$. By fact 3 we know that $(M_1'M_2') \to N$. Finally, N must eval to value M' at stage $t - (t_1 + t_2 + 1)$, thus by the induction hypothesis $N \twoheadrightarrow M'$ and more importantly, $M \twoheadrightarrow M'$.

■

References

[1] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[2] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

①

Arthur Lent 11/27/89

~~[illegible]~~ call-by-value for 6.044

Language:

$x^\sigma : \sigma$

$(M^{\sigma\to\tau}\ N^\sigma) : \tau$

$(\lambda x^\sigma.\ M^\tau) : \sigma\to\tau$

succ, pred : $\iota\to\iota$

$0, 1, 2, \ldots : \iota$

~~(cond M^ι N_ι~~

(cond M^ι N_1^ι N_2^ι) : ι

$Y_\sigma : (\sigma\to\sigma)\to\sigma$ (for all $\sigma \neq \iota$)

definition of value:

~~6~~ λ-abstr, variable, constant)

Operational semantics - rewrite rules ($\to$)

ref trans closure ($\twoheadrightarrow$)

I. a succ $n \to$ $(n+1)$

b. pred $(n+1) \to n$

c. pred $0 \to 0$

~~d. $(\lambda x. M)\ V \to M_x\ V$ $(M[x:=V])$~~

~~e.~~ d ~~(cond 0 N_1 N_2) $\to$ N_1~~

~~f.~~ e. ~~(cond $(n+1)$ N_1 N_2) $\to$ N_2~~

~~g.~~ ~~f.~~ d. $(Y_\sigma\ V^{\sigma\to\sigma}) \to V^{\sigma\to\sigma}\,(\lambda x^\sigma.\ Y_\sigma V^{\sigma\to\sigma}\ x^\sigma)$ (x is fresh (actually only n $x \notin FV(V)$)

II. a. $(\lambda x. M)\ V \to (M[x:=V])$

b. (cond 0 N_1 N_2) $\to N_1$

c. (cond 1 N_1 N_2) $\to N_2$

III a.
$$\frac{M \to M'}{(M\ N) \to (M'\ N)}$$

b.
$$\frac{N \to N'}{(V\ N) \to (V\ N')}$$

c.
$$\frac{M \to M'}{(\text{cond } M\ N_1\ N_2) \to (\text{cond } M'\ N_1\ N_2)}$$

recursive characterization of Language. eval

eval (c) = c; eval $(\lambda x M) = \lambda x M$

$$\text{eval}(M\ N) = \begin{cases} \text{eval}(M'[x:=N']) & (\text{if eval}(M) = \lambda x M' \text{ and eval}(N) = N') \\ \text{constapply}(c, V) & (\text{if eval}(M) = c \text{ and eval}(N) = V) \end{cases}$$

$$\text{eval}(\text{cond}\ M N_1 N_2) = \begin{cases} N_1 & (\text{if eval}(M) = 0) \\ N_2 & (\text{if eval}(M) = (n+1)) \end{cases}$$

constapply is defined as follows:

succ	(succ, n)	→	(n+1)
pred	(pred, 0)	→	0
	(pred, n+1)	→	n
Y	(Y, V)	→	$V(\lambda x. Y V x)$ $(x \notin FV(V))$
{ cond	(cond, 0)	→	$\lambda x \lambda y\ x$
	(cond, 1)	→	$\lambda x \lambda y\ y$ }

If we have an interest in telling the student what it would be like to prove this equivalent to a secd machine we also need the predicate "M has value N at time t" defined by induction on t for closed terms M and N.

bag this part rest of lecture

c has value c at time 1;

2. If M has value a at time t, and N has value N' at time t' and M'[x:=N'] has value L at time t'' then (M N) has value L at time t + t' + t'' + 1

3. If M has value 0 at time t and N_1 has value L at time t', (cond 0 N_1 N_2) has value L at time t+t'+1
If M has value (n+1) at time t and N_2 has value L at time t' (cond (n+1) N_1 N_2) has value L at time t+t'+1

4. If M has value c at time t and N has value N' at time t' then if constapply (c, N') is defined and has value N'' at time t'' then (M N) has value N'' at time t + t' + t'' + 1

It is a fairly simple induction on ~~on~~ t to show that If M has value N at time t, then it has no value at time t' < t and its value at time t is unique. Consequently this is a good ~~defn~~ of a partial function: eval (M) = N iff M has value N at some time closures, environments and Real

Real : Closures → Terms

$$\text{Real}([M, E]) = M[x_1 := \text{Real}(E(x_1)), \ \vdots \ x_n := \text{Real}(E(x_n))]$$

Constapply : Constants × Closed Values $\xrightarrow{p}$ Closed Terms

Constapply : Constants × Value Closures $\xrightarrow{p}$ Closures

$$\text{Real}(\text{Constapply}(c, Cl)) = \text{constapply}(a, \text{Real}(Cl))$$

(1) nil is a Dump

(2) if S is a stack E an environment C a ctl string such that FV(C) ⊆ … and D is a dump [S, E, C, D] is a dump

Control Strings = (Terms ∪ {ap, cd})* where ap, cd ∉ Terms

Def of ⇒

(1) $[Cl : S, E, \text{nil}, [S', E', C', D']] \Rightarrow [Cl : S', E', C', D']$

(2) $[S, E, x : C, D] \Rightarrow [E(x) : S, E, C, D]$

(3) $[S, E, a : C, D] \Rightarrow [[a, \phi] : S, E, C, D]$

(4) $[S, E, (\lambda x M) : C, D] \Rightarrow [[(\lambda x M), E] : S, E, C, D]$

(5) $[[(\lambda x M), E'] : Cl : S, E, ap : C, D] \Rightarrow [\text{nil}, E'\{Cl/x\}, M, [S, E, C, D]]$

(6) $[[a, E'] : [V, E'''] : S, E, ap : C, D] \Rightarrow$ [~~S, E, Constapply(a, V)~~

$[\text{nil}, E', M', [S, E, C, D]]$

where $\text{Constapply}[a, [V, E''']] = [M', E']$

(7) $[S, E, (M N) : C, D] \Rightarrow [S, E, N : M : ap : C, D]$

(8) $[S, E, (\text{Cond } M \ N_1 \ N_2) : C, D] \Rightarrow [[N_1, E], [N_2, E] : S, E, M : cd : C, D]$

(9) a. $[[0, E_1] : [N_1, E_2] : [N_2, E_3] : S, E, cd : C, D] \Rightarrow [S, E_2, N_1 : C, D]$

b. $[[\text{nil}, E_1] : [N_1, E_2] : [N_2, E_3] : S, E, cd : C, D] \Rightarrow [S, E_3, N_2 : C, D]$

SEMANTICS :

$[\![\iota]\!] = \mathbb{N}$ - ordered discretely

$[\![\sigma \to \tau]\!] = [\![\sigma]\!] \to_c [\![\tau]\!]$ (partial continuous fcns)

$[\![succ]\!] =$ successor fcn

$[\![pred]\!] =$ predecessor fcn

$[\![\cdot]\!] : Term \times Env \to Value$

Env : Var → Value (partial)

$[\![c]\!]\rho \simeq c$ (c = 0 ... succ, pred)

$[\![x]\!]\rho \simeq \rho(x)$

$[\![M^{\sigma\to\tau} N^{\sigma}]\!]\rho \simeq [\![M^{\sigma\to\tau}]\!]\rho \cdot [\![N^{\sigma}]\!]\rho$

$$[\![cond\ M\ N_1\ N_2]\!]\rho \simeq \begin{cases} [\![N_1]\!]\rho & \text{if } [\![M]\!]\rho = 0 \\ [\![N_2]\!]\rho & \text{if } [\![M]\!]\rho = (n+1) \end{cases}$$

$[\![\lambda x^{\sigma}. M^{\tau}]\!]\rho \simeq f$

where $f(d) \simeq [\![M^{\tau}]\!]\rho[x^{\sigma} \mapsto d]$

~~$[\![Y]\!]\rho \simeq \bigsqcup_{n}$~~

$[\![Y]\!]\rho(d) \simeq \bigsqcup_{n \geq 0} f_n$

f_0 does not exist

$f_{n+1} = d(f_n)$

Soundness: For any ~~closed~~ Program M, cst k, $M \twoheadrightarrow k$ implies $[\![M]\!](\bot) = k$

Adequacy: For any program M, cst k $[\![M]\!](\bot) = k$ implies $M \twoheadrightarrow k$.

Soundness Proof:

Key Lemma: Substitution

$\llbracket M[x \mapsto N] \rrbracket \rho = \llbracket M \rrbracket \rho[x \mapsto \llbracket N \rrbracket \rho]$

–easy proof by induction on structure of M

only catch: $M = \lambda y\, M'$

$M = \lambda y\, M'$ cases: $y \neq x$

$\llbracket \lambda y\, M'[x \mapsto N] \rrbracket \rho(d) = \llbracket M'[x \mapsto N] \rrbracket \overbrace{\rho[y \mapsto d]}^{=\rho'}$

$\llbracket \lambda y\, M' \rrbracket \rho[x \mapsto \llbracket N \rrbracket \rho](d) = \llbracket M' \rrbracket \rho[x \mapsto \llbracket N \rrbracket \rho][y \mapsto d]$

$= \llbracket M' \rrbracket \rho'[x \mapsto \llbracket N \rrbracket \rho]$

$y = x$

$\llbracket (\lambda x . M')[x \mapsto N] \rrbracket \rho = \llbracket \lambda x\, M' \rrbracket \rho$

$\llbracket \lambda x . M' \rrbracket \rho[x \mapsto \llbracket N \rrbracket \rho] = \llbracket \lambda x\, M' \rrbracket \rho$

Soundness: ~~pf by induct~~ if $M \to M'$ then $\llbracket M \rrbracket \rho = \llbracket M' \rrbracket \rho$ for all ρ

pf. By induction on structure of terms, and cases according to how $M \to M'$

Basis: I a–d, II a–c.

Ind step: III a–c.

Only interesting case is Y, rule I.d

$M = Y V$, $M' = V(\lambda x\, (Y V)\, x)$ (x is first var not free in V)

call "$\llbracket V \rrbracket \rho$" "f".

$\llbracket Y V \rrbracket \rho = \llbracket Y \rrbracket \rho \cdot \llbracket V \rrbracket \rho$

$= \llbracket Y \rrbracket \rho\, (f)$

$\llbracket V(\lambda x\, Y V x) \rrbracket \rho = \llbracket V \rrbracket \rho \cdot \llbracket (\lambda x\, (Y V) x) \rrbracket \rho$

$= f\, (\llbracket \lambda x\, (Y V) x) \rrbracket \rho)$

$\llbracket \lambda x\, Y V x \rrbracket \rho \cdot d = \llbracket (Y\, V)\, x \rrbracket \underbrace{\rho[x \mapsto d]}_{\rho'}$

$= \llbracket Y V \rrbracket \rho'\, d$

$= (\llbracket Y \rrbracket \rho' \cdot \llbracket V \rrbracket \rho')\, d$ but $\llbracket V \rrbracket \rho' = \llbracket V \rrbracket \rho = f$ since $x \notin FV($

$= (\llbracket Y \rrbracket \rho' \cdot f)\, d$

$[\![(\lambda x\, Y V\, x)]\!]\, \rho\, d =$

$\vdots$

$$\left([\![Y]\!]\, \rho' \cdot f\right) \cdot d$$

$$= \left(\bigsqcup_{n \geq 0} f_n\right) \cdot d$$

where f_0 does not exist

$f_{n+1} = f(f_n)$

$$[\![\lambda x\, Y V x]\!]\, \rho = \left(\bigsqcup_{n \geq 0} f_n\right)$$

celim d's

$\therefore$

$$[\![V\, (\lambda x\, Y V x)]\!]\, \rho \simeq f\left(\bigsqcup_{n \geq 0} f_n\right)$$

$$\simeq \bigsqcup_{n \geq 0} (f(f_n))$$

Since f must be continuous

$$\simeq \bigsqcup_{n \geq 1} f_n \simeq \bigsqcup_{n \geq 0} f_n$$

6.044J/18.423J: Computability, Programming, and Logic — Handout 1
Massachusetts Institute of Technology — 13 September 1989

Course Information

Staff.

Lecturer:	Prof. Albert R. Meyer `meyer@theory.lcs.mit.edu`	NE43-315	x3-6024
Teaching Assistant:	Jon G. Riecke `riecke@theory.lcs.mit.edu`	NE43-328	x3-1365
Grader:	Arthur F. Lent `aflent@theory.lcs.mit.edu`	Baker House	225-7178
Secretary:	David Jones `6044-secretary@theory.lcs.mit.edu`	NE43-316	x3-5936

Lectures and Tutorials. Class meets MWF from 1:00–2:00PM in 24-115. There will be no recitation sections, but tutorial/review sessions may be organized in response to requests. The TA will have one regularly scheduled office hour to be announced the first day of class. Further meetings with the TA or instructor can be scheduled by appointment.

Prerequisites. The official requirement for the course is either 18.063 *Introduction to Algebraic Systems*, or 18.310 *Principles of Applied Mathematics*. If you know the basic vocabulary of mathematics and how to do elementary proofs, then you may take this course with the permission of the instructor.

Contrarequisites. There will be up to a 40% overlap in topics (namely, basic computability theory) between 6.045J/18.400J and this course. For this reason, Course 6 students are discouraged from taking both courses. There will be a smaller overlap with 6.840J/18.404J; students, especially Math majors, *may* take both this course and 6.840J/18.404J.

Textbook. The required text for the course is

G. Boolos and R. Jeffrey, *Computability and Logic (Second Edition)*, Cambridge University Press, 1980.

Grading. There will be regular problem sets, quizzes, and most likely a regular three hour final exam. This will be decided the first day of class. The problem sets, quizzes, and final each *count about equally* toward the final grade.

Problem Sets. There will be four to six problem sets. Homework will usually be assigned on a Friday and due 7–10 days later.

Handouts and Notebook. You may find it useful to get a loose-leaf notebook for use with the course, since all handouts and homework will be on standard three-hole punched paper. If you fail to obtain a handout in lecture, you can get a copy from the file cabinet outside David's office (NE43-316). If you take the last copy of a handout, please inform David so that more copies can be made.

Handouts will also be available from the machine `theory.lcs.mit.edu` through the use of the program `ftp`. To use access and transfer these files on a UNIX machine, run `ftp`, and open `theory.lcs.mit.edu`, supplying "anonymous" as the name (account) and "guest" as the password. Files may then be transferred. All handouts will be placed in the directory "/pub/6044" and are written in LaTeX. The macro file "/pub/6044/6044-macros.tex" and the file "/pub/6044/handouts-6044-fall-89" (which serves as an index to the handouts) must also be transferred to run LaTeX on the handout files.

Electronic mail. To facilitate communication in the class, there are three electronic mail addresses:

```
6044-secretary@theory.lcs.mit.edu
6044-forum@theory.lcs.mit.edu
6044-staff@theory.lcs.mit.edu
```

The `6044-forum` mailing list is for general communication by students, the instructor, and the TA to the class; a message sent here will automatically be distributed to those on the mailing list. Students are strongly encouraged to use `6044-forum` to arrange study sessions, discuss ambiguities and problems with homework, and send comments to the whole class. The TA and instructor may also post bugs and corrections to homeworks and handouts to `6044-forum`. Send email to `6044-secretary` to subscribe to the list; other administrative requests should also be directed to this address. Messages to the instructor, TA, or grader should be sent to `6044-staff`.

Pictures. You can help us learn who you are by giving us your photograph with your name on it. This is especially helpful if you later need a recommendation.

Diagnostic Quiz

You will not be graded on this quiz. Do not discuss it with anyone before taking it, Take it sometime after class, and return it to the TA on Friday, September 15. Be sure to indicate your name, the date, "6.044 Diagnostic Quiz", and the time it took you, on your answer sheet.

Problem 1. Describe the function which is the composition of the integer successor function, *i.e.*, successor(x) = $x + 1$, with itself.

Problem 2. How many strings of length four are there over the alphabet $\{a, b, c\}$?

Problem 3. Give an example of an uncountable set.

Problem 4. Which is a synonym for "injective"?

(a) epi

(b) onto

(c) mono

(d) isomorphism

(e) one-to-one

(f) one-to-one and onto

What sets have the property that there is *no* injection from the set into itself?

What sets have the property that there is *no* injection from the set into a *proper* subset of itself?

Problem 5. Define a binary relation, $\preceq$, between sets A, B as follows:

$$A \preceq B \quad \text{iff} \quad (\exists f : A \rightarrow B)(f \text{is injective}).$$

Which of the following properties does the relation $\preceq$ have? For those properties it fails, describe some simple sets $A, B, \ldots$ which provide a countererxample.

(a) reflexive

(b) symmetric

(c) transitive

(d) equivalence relation

(e) partial order

Problem 6. Describe a propositional, *i.e.*, Boolean, connective which is not commutative.

Problem 7. Two Boolean formulas, $F_i(x_1, \ldots, x_n)$ for $i = 1, 2$, are *equivalent* iff they yield the same 0-1 truth value for all 0-1 assignments to the variables $x_1, \ldots, x_n$.

(a) Exhibit three simple, syntactically distinct, but equivalent formulas with two variables.

(b) Explain why "equivalence" is actually an equivalence relation on formulas.

(c) Explain why there are only a finite number of equivalence classes of formulas with (at most) variables $x_1, \ldots, x_n$. How many?

6.044J/18.423J: Computability, Programming, and Logic — Handout 3
Massachusetts Institute of Technology — 15 September 1989

Solutions to Diagnostic Quiz

Problem 1. Describe the function which is the composition of the integer successor function, *i.e.*, successor$(x) = x + 1$, with itself. Answer: $x + 2$.

Problem 2. How many strings of length four are there over the alphabet $\{a, b, c\}$? Answer: $3 * 3 * 3 * 3 = 81$; for each position there are three possible letters, and there are 4 possible positions.

Problem 3. Give an example of an uncountable set. Examples: the real numbers, and the real numbers between 0 and 1.

Problem 4. Which is a synonym for "injective"? Answer: (e) one-to-one.

What sets have the property that there is *no* injection from the set into itself? Answer: NONE. The identify function from a set onto itself is always well-defined, and always an injection.

What sets have the property that there is *no* injection from the set into a *proper* subset of itself? Answer: Precisely the finite sets.

Problem 5. Define a binary relation, $\preceq$, between sets A, B as follows:

$$A \preceq B \quad \text{iff} \quad (\exists f : A \rightarrow B)(f \text{ is injective}).$$

Which of the following properties does the relation $\preceq$ have? For those properties it fails, describe some simple sets $A, B, \ldots$ which provide a counterexample.

(a) reflexive. Answer: YES. The identity from A to A always exists and is always injective.

(b) symmetric. Answer: NO. Consider $A = \{1\}$ and $B = \{1, 2\}$. $A \preceq B$ but $B \npreceq A$.

(c) transitive. Answer: YES. If f_1 is an injection from A to B and f_2 is an injection from B to C then $f_2 \circ f_1$ is an injection from A to C.

(d) equivalence relation. Answer: NO. A relation is an equivalence relation iff it is reflexive, symmetric and transitive. $\preceq$ is not symmetric.

(e) partial order. Answer: Depends on how we define equality. A relation is a partial order iff it is reflexive, transitive, and *anti-symmetric*, *i.e.*, if A is related to B and B is related to A then $A = B$. If we define equality to be "set equality," $\preceq$ is not anti-symmetric, since for $A = \{1\}$ and $B = \{2\}$, $A \preceq B$ and $B \preceq A$, but $A \neq B$. If we define equality as "same cardinality," then $\preceq$ *is* anti-symmetric.

Problem 6. Describe a propositional, *i.e.*, Boolean, connective which is not commutative. Answer: Implies ($\supset$) is a propositional connective which is not commutative. (8 of the 16 propositional connectives are not commutative).

Problem 7. Two Boolean formulas, $F_i(x_1, \ldots, x_n)$ for $i = 1, 2$, are *equivalent* iff they yield the same 0-1 truth value for all 0-1 assignments to the variables $x_1, \ldots, x_n$.

(a) Exhibit three simple, syntactically distinct, but equivalent formulas with two variables. Example: $x_1 \supset x_2$, $\overline{x_1} \vee x_2$ and $\overline{x_1} \vee x_2 \vee x_2$ are true for all assignments *except* $x_1 = \mathbf{true}$ and $x_2 = \mathbf{false}$, in which case all are false.

(b) Explain why "equivalence" is actually an equivalence relation on formulas. Answer: Because it is reflexive (obviously), symmetric (if F_2 agrees with F_1 on all input values, then the opposite must also be the case), and transitive (if F_1 agrees with F_2 on all inputs values, and F_2 agrees with F_3 on all input values, then F_1 agrees with F_3 on all input values), by definition the relation "equivalence" is an equivalence relation on formulas.

(c) Explain why there are only a finite number of equivalence classes of formulas with (at most) variables $x_1, \ldots, x_n$. How many? Answer: For n variables there are exactly 2^n different 0-1 assignments to the variables. For each assignment to the variables there are two possible truth values to yield. Consequently there can be at most only 2^{2^n} different equivalence classes. Why? By the pigeonhole principle if there were more than this 2^{2^n} equivalence classes then at least two of them would have to have the same input/output behavior, in which case they would be the same equivalence classes, so there can be at most 2^{2^n} distinct equivalence classes.

Instructions for Problem Sets

Form of Solutions. Each problem is to be done on a *separate sheet* of *three-hole punched paper.* If a problem requires more than one sheet, staple these sheets together, but keep each problem separate. Do not use red ink. Mark the top of the paper with:

- Your name,
- "6.044J/18.423J",
- the assignment number,
- the problem number, and
- the date.

Try to be as clear and precise as possible in your presentations. Problem grades are based not only on getting the right answer or otherwise demonstrating that you understand how a solution goes, but also on your ability to explain the solution or proof in a way helpful to a reader.

If you have doubts about the way your homework has been graded, first see the TA. Other questions and suggestions will be welcomed by both the instructor and the TA.

Problem sets will be collected at the beginning of class; graded problem sets will be returned at the end of class. Solutions will generally be available with the graded problem sets, one week after their submission.

Collaboration and References. You must write your own problem solutions and other assigned course work in your own words and entirely alone. On the other hand, you are encouraged to discuss the problems with one or two classmates before you write your solutions. If you do so, please be sure to

> *indicate the members of your discussion group*

on your solution.

Similarly, you are welcome to use other texts and references in doing homework, but if you find that a solution to an assigned problem has been given in such a reference, you should nevertheless rewrite the solution in your own words and *cite your source.*

Late Policy. Late homeworks should be submitted to the TA. If they can be graded without inconvenience, they will be. Late homeworks that are not graded will be kept for reference until after the final. No homework will be accepted after the solutions have been given out.

Problem Set 1

Due: 22 September 1989.

Remark. Before beginning the assignment, please be sure to read Handout 4, "Instructions for Problem Sets."

Problem 1. Let x be any string, and let x^R denote the *reversal* of x. For example, if x is a string over the alphabet $\{\mathsf{a},\mathsf{b}\}$ and $x = \mathsf{babb}$, then $x^R = \mathsf{bbab}$.

1(a). Give an inductive definition of x^R based on the definition of strings.

1(b). Prove, by induction on the string x, that

$$(x \cdot y)^R = y^R \cdot x^R.$$

Problem 2. Consider the following inductive definition of a subset M of the *natural numbers* $\mathbf{N} = \{0, 1, 2, 3, \ldots\}$:

(i) $2 \in M$,

(ii) if $n \in M$, then $n^2 \in M$,

(iii) if $n, m \in M$, then $(n \cdot m) \in M$.

2(a). Prove by induction on the definition of M that

$$M = \{2^k \mid k \geq 1\}\ .$$

Let $f : M \to \mathbf{N}$ be any (possibly partial) function. Then f is said to be a *counter function* if

(i) $f(2) = 1$,

(ii) $f(n^2) = 2 \cdot f(n)$,

(iii) $f(n \cdot m) = f(n) + f(m)$.

2(b). Define $f_1 : M \rightarrow \mathbf{N}$ to be

$$f_1(2^k) = k$$

Prove that f_1 is a counter function. (Hint: Induction on the definition of M.)

2(c). Prove that f_1 is the *unique* counter function, *i.e.*, if f_2 is any counter function, then $f_1(x) = f_2(x)$ for all $x \in M$.

2(d). Consider the function $g : M \rightarrow \mathbf{N}$ defined inductively by:

(i) $g(2) = 1$,
(ii) $g(n^2) = 5$,
(iii) $g(n \cdot m) = 10$.

Carefully prove that $1 = 0$! Explain why this contradiction occurs here, but not for counter functions.

Problem 3. Recall that Σ^*, where Σ is an alphabet (a set of symbols), is the set of strings created from symbols in Σ. Pick any string $x_0 \in \Sigma^*$, and also pick any total function $g : \Sigma \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$. We say that $f : \Sigma^* \rightarrow \Sigma^*$ is defined by *string recursion on notation* (from x_0 and g) if

(i) $f(e) = x_0$,
(ii) $f(\sigma x) = g(\sigma, x, f(x))$.

Prove that there exists a unique total function f defined by recursion on notation from x_0 and g. (Hint: Prove that $f(x)$ exists and is uniquely determined by induction on x.)

6.044J/18.423J: Computability, Programming, and Logic Handout 7
Massachusetts Institute of Technology 22 September 1989

Problem Set 2

Due: 29 September 1989

Problem 1. (Enumerability and Diagonalization, Boolos & Jeffrey, Chapters 1 and 2)

1(a). Let A be any set, and $P(A)$ be the set of all subsets of A. Prove that there is no onto function from $f : A \rightarrow P(A)$. (Hint: Use a diagonalization argument with the "diagonal" set $\{a \in A : a \notin f(a)\}$. If the case $A = \emptyset$ bothers you, ignore it.)

1(b). Let A be a denumerably infinite set, and let $P_{\mathit{fin}}(A)$ be the set of all *finite* subsets of A. Prove that $P_{\mathit{fin}}(A)$ is enumerable.

Problem 2. (Diagonalization) A function $u : \mathbf{N}^2 \rightarrow \mathbf{N}$ is called a *universal function* for the primitive recursive functions of one argument iff, for every $n \in \mathbf{N}$, $\lambda x.u(n,x)$ is primitive recursive, and moreover, for every primitive recursive function $g : \mathbf{N} \rightarrow \mathbf{N}$, there is an $n_g \in \mathbf{N}$ such that $g = \lambda x.u(n_g, x)$.

2(a). Show that no such universal function can be primitive recursive.

2(b). Explain informally why there is such a universal function which is computable (programmable) in say, SCHEME.

Problem 3. (Partial Orders, Handout 6) Let A, B be partially-ordered sets, *i.e.*, sets with partial order relations $\leq_A$, $\leq_B$ respectively. Suppose that $f : A \rightarrow B$ is a total function. Then f is said to be *monotone* if for all $x, y \in A$,

$$x \leq_A y \text{ implies } f(x) \leq_B f(y).$$

Monotone functions are sometimes called order-preserving functions.

Suppose A is a *finite* partially-ordered set (with ordering $\leq_A$) having a unique least element a_0. Suppose $f : A \rightarrow A$ is a total monotone function. Prove that f has a *least fixed point fix(f)* $\in A$. That is, if $x =$ *fix(f)*, then

$$f(x) = x$$

and moreover, $x \leq_A y$ for any y with $f(y) = y$. (Hint: Consider the sequence $a_0, f(a_0), f(f(a_0)), \ldots$. Show that if $f(y) \leq_A y$, then every element in the sequence is $\leq_A y$. Then show that because A is finite, the sequence has an lub (least upper bound) and this lub must be *fix(f)*.)

Problem 4. (Primitive Recursion, Boolos & Jeffrey, Chapter 7) Suppose the functions $f : \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ and $g : \mathbf{N}^n \rightarrow \mathbf{N}$ are primitive recursive. Let the function $GenSum_n[f, g] : \mathbf{N}^n \rightarrow \mathbf{N}$ be defined by

$$\begin{aligned} &GenSum_n[f, g](x_1, \ldots, x_n) = \\ &\quad f(x_1, \ldots, x_n, 0) + f(x_1, \ldots, x_n, 1) + \cdots + f(x_1, \ldots, x_n, g(x_1, \ldots, x_n)) \end{aligned}$$

Give a formal primitive recursive definition of $GenSum_n[f, g]$. You may use the constants f, g, and sum (defined on page 84 of the text) in your definition, in addition to the usual functions id_k^l, s, and z, and functionals Pr_k and $Cn_{k,l}$. (Hint: First carefully translate the inductive definition

$$\begin{aligned} &\text{(i)} \quad h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n, 0), \\ &\text{(ii)} \quad h(x_1, \ldots, x_n, n+1) = h(x_1, \ldots, x_n, n) + f(x_1, \ldots, x_n, n+1) \end{aligned}$$

into a primitive recursive definition, and then use composition to obtain the function $GenSum_n[f, g]$.)

6.044J/18.423J: Computability, Programming, and Logic — Handout 8

Massachusetts Institute of Technology — 29 September 1989

Problem Set 1 Solutions

General Information. This handout includes some of the best solutions submitted by students for Problem Set 1. These solutions are a good representation of the level of detail expected.

The grades went as follows:

	number submitted	min	max	mean	median
1	14	10	25	21.4	24
2	14	10	25	21.4	23
3	13	5	25	18.3	15

Aaron Wallach
6.044 (25-25-24)

① x

x^R is defined as

1A
① if $x = \epsilon$, $x^R = \epsilon$ ✓
② if $x = (\sigma, x')$, $x^R = x' \cdot (\sigma, e)$

(1B) Assume $(x \cdot y)^R = y^R \cdot x^R$

if $x = \epsilon$, then $(x \cdot y)^R = (\epsilon \cdot y)^R = y^R$ [because $\epsilon \cdot y = y$] ✓

if $x = (\sigma, x')$ then $(x \cdot y)^R = ((\sigma, x') \cdot y)^R = (\sigma, (x' \cdot y))^R =$

$$\underbrace{(x' \cdot y)^R \cdot \sigma = (y^R \cdot x'^R) \cdot \sigma}_{\text{Inductive step}} = y^R \cdot (x'^R \cdot \sigma) = y^R \, x^R$$ ✓

Ted Hoatz
6.044J/18.423J
Problem Set 1
#3
9/22/89

3. Given: $x_0 \in \Sigma^*$ and a total function $g : \Sigma \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$

Prove: $\exists!$ total $f : \Sigma^* \rightarrow \Sigma^*$, defined by string recursion on notation as: ✓

(i) $f(e_\Sigma) = x_0$
(ii) $f(\langle \sigma, x \rangle) = g(\sigma, x, f(x))$

~~(a) Proof~~

Pf:

(1) $f(e_\Sigma) \in \Sigma^*$ exists, and is uniquely determined, because it is equal to x_0, which was uniquely chosen from Σ^* (by (i)) ✓

~~(2) For $\sigma \in \Sigma^*$, and $x \in$~~

(2) For $\sigma \in \Sigma$ and $x \in \Sigma^*$,
if $f(x) \in \Sigma^*$ exists and is uniquely determined, then $g(\sigma, x, f(x)) \in \Sigma^*$ exists and is uniquely determined, because g was uniquely chosen and is total, σ and x were uniquely chosen, and $f(x) \in \Sigma^*$ exists and is uniquely determined (by the inductive hypothesis) ✓

$\Rightarrow f(\langle \sigma, x \rangle) \in \Sigma^*$ exists and is uniquely determined because it is equal to $g(\sigma, x, f(x))$ (by (ii)) ✓

$\therefore$ $\forall x_0 \in \Sigma^*$, $\forall$ total $g : \Sigma \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ [illegible] total $f : \Sigma^* \rightarrow \Sigma^*$ defined by string recursion on notation (from x_0 and g) ✓

25/25

/

Jeff Johnson
6.044J
PS1
#2
9/15/89

2(a). Prove by induction on def of M that $M = \{2^k \mid k \geq 1\}$

ON K

(1) Base: $k=1 \rightarrow 2^k = 2^1 = 2$ ✓ by (i) $2 \in M$ ✓

(2) Induction: given $2^{k-1}, 2 \in M \rightarrow 2 \cdot 2^{k-1} = 2^k \in M$ by (iii) with $n = 2^{k-1}, m = 2$. ✓

This shows that all $\{2^k \mid k \geq 1\}$ is in M. ✓ I now show that all M is in $L = \{2^k \mid k \geq 1\}$ by considering the 3 cases.

(i) $2 \in M$ checks by def. ✓

inductive (ii) if $n \in M$, then $n^2 \in M$: $n \in L$ by ind then $n = 2^k$ for some $k \geq 1$. Thus $n^2 = 2^{2k}$ which implies $n^2 \in L$ ✓

inductive (iii) if $n, m \in M$, then $(n \cdot m) \in M$: $m, n \in L$ by ind thus $n = 2^k$ and $m = 2^j$ for some $j, k \geq 1$. Thus, $(n \cdot m) \in L$ since $2^k \cdot 2^j = 2^{k+j}$. ✓

Thus M consists exclusively of L and vice versa ($M = L$). □

(b) Prove f_1 is a counter function by inductive examination of def of a c.f.

(i) $f_1(2) = f_1(2^1) = 1$ ✓ ✓

(ii) $n = 2^k$ for some $k \geq 1$ $f_1(2^k) = k$ thus $n^2 = 2^{2k}$
and $f_1(2^{2k}) = 2 \cdot k$.
Therefore $f_1(n^2) = f_1(2^{2k}) = 2 \cdot k = 2 \cdot f_1(2^k) = 2 \cdot f_1(n)$ ✓ ✓

(iii) $n = 2^k$ for some $k \geq 1$ $\rightarrow$ $f_1(n) = f_1(2^k) = k$
$m = 2^j$ for some $j \geq 1$ $f_1(m) = f_1(2^j) = j$

$n \cdot m = 2^k \cdot 2^j = 2^{k+j} \rightarrow f_1(n \cdot m) = f_1(2^{k+j}) = k+j$

$f_1(n) + f_1(m) = k+j = f_1(n+m)$ ✓ ✓

25/25

Proof by induction on x
$\forall x \in N\ f_1(x) = f_2(x)$

Jeff Johnson
6.044J
PS1
#2
9/15/85

2c. Suppose that f_2 is a counter function, but that for some x $f_2(x) \neq f_1(x)$.

(1) Base: $f_2(2) = f_1(2)$ by defintion of a c.f.

(2) Induction given $f_2(y) = f_1(y)$

$$f_2(2y) = f_2(2) + f(y)$$

But $f_2(2) = f_1(2)$ and $f_2(y) = f_1(y)$.

Thus $f_1(x) = f_2(x)$ for all x showing that $f_1(x)$ is unique ✓

2d This problem seems a bit contrived, however here is A solution

1. $\dfrac{g(4)-5}{5} = \dfrac{g(4)-5}{5}$

2. $\dfrac{g(2\cdot 2)-5}{5} = \dfrac{g(2^2)-5}{5}$

3. $\dfrac{10-5}{5} = \dfrac{5-5}{5}$

4. $1 = 0$! ✓

This contradiction occurs because of conflict between (ii) and (iii).
Since (ii) is a special case of (iii) they should yield the same
result as they do in the def of $f(x)$ c.f.
Moral: your inductive def. should be self-consistent. ✓

6.044J/18.423J: Computability, Programming, and Logic — Handout 9
Massachusetts Institute of Technology — 29 September 1989

Problem Set 3

Due: 6 October 1989

Instructions. Do Problem 1 and *either* Problem 2 or Problem 3.

Problem 1. (Turing Machines, Boolos & Jeffrey, Chapter 3) Construct the flow graph of a Turing machine that converts unary numbers to binary numbers. The converter should start with a number n in standard format (*i.e.*, reading the leftmost 1 of a block of $n + 1$ 1's, with blanks everywhere else on the tape.) The machine should halt reading the leftmost digit of the binary representation of n, a string of a's and b's, with a representing the binary digit 0 and b representing the binary digit 1. Show each configuration of your machine on the input 1111 (*i.e.*, the number 3, so in the final configuration, the tape would look like $\cdots 00$bb$00 \cdots$.)

Problem 2. (Abacus Machines, Boolos & Jeffrey, Chapter 6) For parts (a), (b), and (c), use only registers 1, 2, and 3.

2(a). Write an abacus program (a flow chart) using *only* registers 2 and 3 that multiplies the contents of register 2 by 5 and sets register 3 to zero as a side effect. That is, show how an abacus machine can simulates the following instructions given in the notation of the text:

$$\boxed{\begin{array}{l} 5 \cdot [2] \rightarrow 2 \\ 0 \rightarrow 3 \end{array}}$$

2(b). Write an abacus program for

$$\boxed{\begin{array}{l} 5^{[1]} \rightarrow 2 \\ 0 \rightarrow 1 \\ 0 \rightarrow 3 \end{array}}$$

(Hint: Use part (a).)

2(c). Write an abacus program that simulates the following branch instruction:

```
if 5 divides [2]
    then 0 → 3
        ⟨ code ⟩
    else 0 → 3
        ⟨ code ⟩
```

(Hint: Divide [2] by 5, using register 3 as a temp. Check remainder to branch, then reverse procedure to restore [2].)

2(d). We can uniquely code the contents of four registers into a single number: if the registers contain n_1, n_2, n_3, n_4, use the number $2^{n_1} \cdot 3^{n_2} \cdot 5^{n_3} \cdot 7^{n_4}$ as the code. Suppose this encoding is placed in register 1. Show how, using a three-register abacus machine, to simulate the increment instruction on any of the four registers. That is, write an abacus program that updates register 1 so that register 1 contains an encoding of the contents of the four registers after the increment. Likewise, show how to simulate the test-and-decrement instruction on any of the four registers.

2(e). Conclude from the above that a three-register abacus machine can compute exactly the same partial functions from $\mathbf{N} \to \mathbf{N}$ as an arbitrary abacus machine.

Problem 3. (Computability and Church's Thesis, Boolos & Jeffrey, Chapter 6) Consider the following two instructions which add indirect addressing to the abacus machine model:

- Copy the contents of register i to the register whose *address* is stored in register j:

  $$[i] \to [j]$$

  For example, if j holds 56, "$[i] \to [j]$" stores the contents of register i in register 56 (destroying the old contents of register 56.)

- Copy the contents of the register whose address is stored in register i to register j:

  $$[[i]] \to j$$

  For example, if i holds 32, "$[[i]] \to j$" copies the contents of register 32 into register j (again, destroying the old contents of register j.)

(For simplicity, assume the registers of the abacus are labelled $0, 1, 2, \ldots$) Give an informal but convincing explanation of why the class of abacus-computable functions does not change when these instructions are added to the model.

6.044J/18.423J: Computability, Programming, and Logic — Handout 10
Massachusetts Institute of Technology — 2 October 1989

Notes on Computability Theory

The following notes outline the main definitions and results about computability theory indicated in the course lectures and homework which go beyond Boolos & Jeffrey's text.

1 Simulation

The "simulation thesis" says that all general computation models are capable of simulating one another given sufficient resources and ignoring issues of efficiency. That is, we insist only that the simulations *eventually* get the right answer, no matter how slowly they proceed or how much more memory or energy they require than the machine being simulated. Thus, an apparently very weak machine model such as a Turing machine computes exactly the same set of partial functions on, say, numbers as are computable by Scheme programs, random-access register machines (RAM'S), or enhanced Turing machines with multidimensional, multihead, multitapes.

Not just machine models, but certain inductively defined classes of functions such as the μ-recursive functions also characterize the same class of computable partial functions. This is proved, for example, by showing for any Turing machine, how to devise a μ-recursive function that simulates the step-by-step computation of the Turing machine (see Boolos & Jeffrey, Chapter 8), and by showing how to write a Turing machine interpreter for the μ-recursive programming language (not described in Boolos & Jeffrey, but one could construct such an interpreter just as one constructs interpreters for Scheme.)

The significance of all these simulations is that we can take a robust, machine and programming language *independent* view of the computable functions. The Turing computable partial functions from tuples of natural numbers to natural numbers are the same as the enhanced-Turing computable partial functions are the same as the abacus coumputable functions are the same as the μ-recursive partial computable functions are the same as the functions computable by Scheme procedures.... This class of functions is called the *partial computable functions*; a synonym is the *partial recursive functions.*

These notes will follow the mathematical literature in regularly referring to Turing machines when we need a concrete machine/programming language model, but the reader can safely think of "Turing Machine" as replaced by, say, "Scheme procedure text" if that seems more familiar.

What we have just called the "simulation thesis" is called "Church's thesis" by Boolos & Jeffrey. In the wider literature, Church's thesis is often described as equating any intuitively "effective procedure" with a Turing machine program. This leaves open misinterpretations of Church's thesis: for example, a cookbook recipe might in ordinary discourse be regarded as describing an effective procedure, but no one claims a Turing machine can bake a cake! A more interesting example might be the 0-1 valued function $f(n)$ which is equal to one iff there will be an earthquake with epicenter in Cambridge, Massachusetts during the n^{th} minute after noon on October 2, 1989. There is an obvious effective procedure to compute $f(n)$ with some seismographs and a clock—by waiting up to n minutes—but again, Church surely did not mean to imply that a Turing machine could compute f.

2 Binary Codes for Finite Objects

In adopting the Turing machine model over the other models, it will be more natural to talk about computing on ***strings*** over an arbitrary alphabet Σ instead of computing on natural numbers. It is important to note, however, that this feature of Turing machines—the ability to compute string functions—adds no power. Abacus machines and μ-resursive functions can also compute the same class of string functions—using a suitable encoding function of strings into natural numbers (say, the ASCII representation that turns a string into a number.) We say that a string function is ***partial recursive*** if there is a Turing machine computing it.

One can also talk about computing functions over other data objects: integers, finite graphs, lists, flowcharts, ordered pairs and finite sets of these, and various other finite or finitely representable mathematical objects. In each case we assume, without going into detail, that there are encodings of these objects into finite strings over some standard alphabet. We will use the alphabet $\{\mathtt{a}, \mathtt{b}\}$ as our standard alphabet; one could think of strings in this alphabet as "binary" numerals.[1] A function on, say, graphs, is "partial recursive" iff the corresponding function on strings which code graphs is partial recursive.

In a basic argument below, we will speak of the Turing Machine (Self-)Halting Problem, K_1. The "problem" is represented by the set of Turing machines that halt "when given themselves as input." More precisely, we must define a coding function

$$d : \{\text{Turing machines}\} \to \{\mathtt{a}, \mathtt{b}\}^*$$

under which every Turing machine is coded as a "binary" string. It is straightforward enough, though a little tedious, to do this; one encoding would change the flow graph of a Turing machine into a string. It is technically convenient if *every* string in $\{\mathtt{a}, \mathtt{b}\}^*$ is the code of some Turing machine. We can always

[1] To avoid the ambiguous representation of the "blank" symbol in Boolos & Jeffrey, we will use the symbol "#" to represent blanks.

make this hold by invoking the convention that every string not in the range of d is to be interpreted as the code of a particular fixed Turing machine which, say, halts with output a on any input. We use the notation M_x for the Turing machine denoted by $x \in \{\mathtt{a}, \mathtt{b}\}^*$.

We now define

$$K_1 = \{x \in \{\mathtt{a}, \mathtt{b}\}^* : M_x \text{ is a Turing machine that halts on input } x\} .$$

(Note that, by convention, a string $x \in \{\mathtt{a}, \mathtt{b}\}^*$ is in K_1 only if $\{\mathtt{a}, \mathtt{b}\} \subseteq \Sigma_{M_x}$, where Σ_{M_x} is the tape alphabet (the alphabet used plus the blank symbol) of M_x.)

Similarly, we can straightforwardly code strings over an arbitrary enumerably infinite alphabet into binary strings, and of course we can code a pair of binary strings into a single string. Thus, when we say the (General) Turing Machine Halting Problem, K_0, is

$$K_0 = \{(M, x) : \text{Turing machine } M \text{ halts on input } x\}$$

we really mean

$$\begin{aligned} K_0 = \{z \in \{\mathtt{a}, \mathtt{b}\}^* \quad : \quad & z = \mathit{pair}(y_1, y_2),\ y_2 \text{ codes } x \in \Sigma^*_{M_{y_1}}, \\ & \text{and } M_{y_1} \text{ halts on input } x\}. \end{aligned}$$

The precise set of binary words equal to K_0 or K_1 depends of course on how we choose the coding function d, but the salient properties we establish about these sets are independent of the details of the coding (see the Appendix, §A of these notes).

3 Terminology

Definition 1. A *language* is a subset of Σ^*.

In other words, a language is an arbitrary set of strings over some alphabet. For example, the set of strings with equal numbers of a's and b's is a language over the alphabet $\{\mathtt{a}, \mathtt{b}\}$. Given a language A over the alphabet Σ, we denote its *complement* by

$$\overline{A} = \{x \in \Sigma^* : x \notin A\}.$$

Definition 2. A partial recursive function that happens to be totally defined on $(\Sigma^*)^n$ is called *total recursive*.

A language $D \subseteq \Sigma^*$ is ***decidable*** iff its characteristic function $c_D : \Sigma^* \rightarrow \{0, 1\}$ is total recursive, where

$$c_D(x) = \begin{cases} 1 & \text{if } x \in D, \\ 0 & \text{otherwise.} \end{cases}$$

For any Turing machine M, let

$$\text{domain}(M) = \{x \in \Sigma_M^* : M \text{ halts on input } x\}.$$

A language $R \subseteq \Sigma^*$ is ***recursively enumerable*** (r.e.) iff $R = \text{domain}(M)$ for some Turing machine M.

Like the notion of "partial recursive," the above definitions can be extended to other datatypes. For example, a set of finite graphs is r.e. iff the language consisting of binary codes of elements of the set is r.e.

Synonyms:

- recursive set = decidable set = Turing-decidable set.
- r.e. set = recursively enumerable set = Turing-acceptable set.
- total recursive function = recursive function = [Turing] computable total function= general recursive function.

If P is a property of sets, then "A is co-P" means $P(\overline{A})$ holds.

4 Basic Properties of R.E. and Recursive Sets

Lemma 1. Recursive sets are closed under complement. (That is, if A is recursive, then $\overline{A}$ is recursive.)

Theorem 1. A is recursive iff both A and $\overline{A}$ are r.e. Another way to say this is, A is recursive iff A is both r.e. and co-r.e.

Theorem 2. The recursive sets are closed under union, intersection, and complementation.

Theorem 3. The r.e. sets are closed under union and intersection.

Theorem 4. The following are equivalent for a language A:

(a) A is r.e.

(b) A is the domain of a partial recursive function of one argument.

(c) A is the range of a partial recursive function.

(d) $A = \emptyset$ or A is the range of a total recursive function.

(e) A is finite or A is the range of a one-to-one total recursive function.

Proof: To show, for example, that (b) implies (c), suppose A is the domain of a partial recursive function of one argument computed by a Turing machine M. Define a new Turing machine M' that works as follows:

> "On input x, run M on x until it halts; then output x (that is, write x on the tape and erase the rest of the tape.)"

The language A is then the range of a partial recursive function computed by M'. The other implications may be proved along similar lines. ■

Definition 3. The canonical order $<_{\text{can}}$ of strings in Σ^* (where Σ is ordered) is defined for all $w, u \in \Sigma^*$ as follows: $w <_{\text{can}} u$ iff $|w| < |u|$ or $|w| = |u|$ and w precedes u alphabetically.

Note that canonical order is different from alphabetical (dictionary) order. For example, $b <_{\text{can}} ab$ even though ab precedes b alphabetically. The canonical ordering of $\{a, b\}^*$, where a precedes b in the alphabet, is

$$e, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots, bbb, aaaa, \ldots.$$

Some authors use the term "lexicographic order" for canonical order.

Definition 4. A function $f : \Sigma_0^* \rightarrow \Sigma_1^*$ is an *increasing* function iff, for all strings $x, y \in \text{domain}(f)$, if $x <_{\text{can}} y$, then $f(x) <_{\text{can}} f(y)$.

Theorem 5. A language A is recursive iff it is finite or the range of an increasing total recursive function.

5 Undecidability of the Halting Problem

Let K_0 and K_1 be respectively the Turing machine Halting and the Self-halting Problems defined in §2 above.

Theorem 6. $\overline{K_1}$ is not r.e.

Proof: By the definition of $\overline{K}_1$, we have for any $x \in \{\mathtt{a}, \mathtt{b}\}^*$ that

$$x \in \overline{K}_1 \quad \text{iff} \quad M_x \text{ does not halt on input } x.$$

Assume that $\overline{K}_1$ is r.e. Then there is string x_0 such that M_{x_0} halts on precisely the strings in $\overline{K}_1$. By the definition of M_{x_0}, we have for every string x that

$$x \in \overline{K}_1 \quad \text{iff} \quad M_{x_0} \text{ halts on } x.$$

Hence, for all x,

$$M_{x_0} \text{ halts on } x \quad \text{iff} \quad M_x \text{ does not halt on } x.$$

Now, let $x = x_0$ and we obtain an immediate contradiction. ■

Corollary 1. K_1 is not recursive.

Proof: If it were, then $\overline{K}_1$ would be recursive too by Lemma 1, and so would be r.e. by Theorem 1, contradicting what we just proved. ■

Corollary 2. K_0 is not recursive.

Proof: A simple modification of any program which decided membership in K_0 would yield a program which decided membership in K_1, contradicting Corollary 1. ■

Theorem 7. K_0 and K_1 are r.e.

Proof: This follows easily from the fact that we can write a Turing machine program which is a ***universal function***: on input strings $x, y \in \{\mathtt{a}, \mathtt{b}\}^*$, it simulates the computation of M_x on the input string over Σ_{M_x} coded by y. Constructing the universal machine or "interpreter" itself is a long but nowadays familiar programming project. ■

Corollary 3. K_0 is not co-r.e., and hence the r.e. sets are not closed under complement.

Proof: Theorem 1, Corollary 2, and Theorem 7. ■

6 Many-one Reducibility

Besides diagonalization, the technique of *reducing* one language to another is often used to show that certain languages are not recursive or r.e.

Definition 5. Given two languages $A \subseteq \Sigma_A^*$, $B \subseteq \Sigma_B^*$, we say A is *many-one reducible* to B, in symbols $A \leq_{\mathrm{m}} B$, iff there is a total computable function $f : \Sigma_A^* \rightarrow \Sigma_B^*$ such that

$$x \in A \quad \text{iff} \quad f(x) \in B$$

for all $x \in \Sigma_A^*$. We say $A \equiv_{\mathrm{m}} B$ iff $[A \leq_{\mathrm{m}} B$ and $B \leq_{\mathrm{m}} A]$.

The following properties are easily verified:

Transitivity. $A \leq_{\mathrm{m}} B$ and $B \leq_{\mathrm{m}} C$ implies $A \leq_{\mathrm{m}} C$.

Recursiveness Inherits Down. $A \leq_{\mathrm{m}} B$ and B recursive implies A recursive.

Non-recursiveness Inherits Up. $A \leq_{\mathrm{m}} B$ and A not recursive implies B not recursive.

R.E. Inherits Down. $A \leq_{\mathrm{m}} B$ and B r.e. implies A r.e.

Non-R.E. Inherits Up. $A \leq_{\mathrm{m}} B$ and A not r.e. implies B not r.e.

Symmetry w.r.t. Complement. $A \leq_{\mathrm{m}} B$ iff $\overline{A} \leq_{\mathrm{m}} \overline{B}$.

Lemma 2. If A is r.e. but not recursive, then A and $\overline{A}$ are $\leq_{\mathrm{m}}$-incomparable.

Proof: Say A is r.e. If $\overline{A} \leq_{\mathrm{m}} A$, then since r.e. inherits down, $\overline{A}$ is also r.e. Therefore, A is recursive by Theorem 1. If $A \leq_{\mathrm{m}} \overline{A}$, then by symmetry w.r.t. complement, $\overline{A} \leq_{\mathrm{m}} A$, which is the previous case. So if A is r.e. and not recursive, neither $\overline{A} \leq_{\mathrm{m}} A$ nor $A \leq_{\mathrm{m}} \overline{A}$ holds. ■

Note that the ability to decide membership in a set obviously implies the ability to decide membership in the complement of the set. But the last property above reveals that a set and its complement may be $\leq_{\mathrm{m}}$-incomparable! This highlights the point that $\leq_{\mathrm{m}}$ is a precise *restricted* version of the intuitive notion of "reducing" one problem to another.

Corollary 4. $K_0 \leq_{\mathrm{m}} A$ implies A is not co-r.e.

Proof: $K_0 \leq_{\mathrm{m}} A$ implies $\overline{K_0} \leq_{\mathrm{m}} \overline{A}$. By Corollary 3, $\overline{K_0}$ is not r.e., and non-r.e. inherits up $\leq_{\mathrm{m}}$. ■

Corollary 5. $K_0 \times \overline{K}_0$ is neither r.e. nor co-r.e.

Proof: Obviously, $K_0 \leq_m K_0 \times \overline{K}_0$, so is not co-r.e. by Corollary 4. Likewise $\overline{K_0} \leq_m K_0 \times \overline{K}_0$, and so it's not r.e. either. ■

Lemma 3. If A is r.e., then $A \leq_m K_0$.

Proof: Suppose M accepts A. Let $f(x) = (M, x)$. Clearly, f is recursive, and also $x \in A$ iff $f(x) \in K_0$. ■

Theorem 7 and Lemma 3 show that K_0 is, in some sense, the ***hardest*** r.e. set—any r.e. set can be reduced to it. The following definition formalizes this notion:

Definition 6. If R is a class of sets and $\leq$ is a relation on sets, then a set K is $\leq$-***hard*** for R iff $C \leq K$ for all $C \in R$. A set K is $\leq$-***complete*** for R iff K is both $\leq$-hard for R and $K \in R$.

Lemma 4. Every language other than the empty set and Σ^* is $\leq_m$-hard for recursive languages over Σ.

Proof: An exercise. ■

7 Undecidability of Blank-Tape Halting Problem

Theorem 8. Let $K_2 = \{M : M \text{ halts on blank tape}\}$, that is, the set of encodings of Turing machines that halt on blank tape. Then K_2 is a $\leq_m$-complete r.e. set.

Proof: Clearly K_2 is r.e. (*cf.* Theorem 7), so we need only show that it is $\leq_m$-hard for r.e. sets.

Let $R \subseteq \Sigma^*$ be any r.e. set, and say $R = \text{domain}(M_R)$ for some Turing machine M_R. We show that $R \leq_m K_2$ as follows.

For any string $x \in \Sigma^*$, we can define a new Turing machine $M_{f(x)}$ (that is, $f(x)$ is the code of this Turing machine) that operates as follows:

> "On input w, erase w, print x on the tape as input, and then act exactly like M_R."

By definition, the behavior of $M_{f(x)}$ does not depend on its input—it always halts or it never halts. In fact,

$$\begin{array}{rll} x \in R & \text{iff} & M_R \text{ halts on } x \\ & \text{iff} & M_{f(x)} \text{ halts on some input} \\ & \text{iff} & M_{f(x)} \text{ halts on input } e \\ & \text{iff} & f(x) \in K_2. \end{array}$$

But it is not hard to see that the function f is total recursive. (Think of writing a Scheme procedure that, when applied to character string x, prints out a Turing machine flow-chart for $M_{f(x)}$. The Scheme program has a flowchart for M_R as a "built-in" constant.) Hence, $R \leq_{\mathrm{m}} K_2$. ■

Theorem 9. K_1 is a $\leq_{\mathrm{m}}$-complete r.e. set.

Proof: Replace "2" by "1" in the preceding proof. ■

8 Rice's Theorem

Definition 7. A property of languages is ***nontrivial on the r.e. languages***, "nontrivial" for short, iff there is some r.e. language that has the property and some r.e. language that does not.

For example, the property of being an r.e. language is trivial (since all r.e. languages have it). The properties of

- containing the empty word,
- being empty, or
- being infinite

are each nontrivial.

Theorem 10. **(Rice)** The set $K_P = \{M : P(\text{domain}(M))\}$ is not decidable for any nontrivial property P of r.e. sets. In fact, if $P(\emptyset)$ is false, then K_P is $\leq_{\mathrm{m}}$-hard for r.e. sets.

Proof: Suppose that $P(\emptyset)$ is false. Since P is nontrivial, there exists a machine M_1 with domain$(M_1) \neq \emptyset$ such that $P(\text{domain}(M_1))$.

Let $R \subseteq \Sigma^*$ be any r.e. set and say $R = \text{domain}(M_R)$ for some Turing machine M_R. We show that $R \leq_{\mathrm{m}} K_P$ as follows.

For any string $x \in \Sigma^*$, we can define a new Turing machine $M_{f(x)}$ (that is, $f(x)$ is the code of this Turing machine) that operates as follows

> "On input w, save w temporarily, and simulate M_R on input x. If this simulation halts, then act exactly like M_1 on input w."

Now, by definition of $M_{f(x)}$,

$$\begin{aligned}&M_R \text{ halts on } x\\ &\text{implies } M_{f(x)} \text{ acts like } M_1 \text{ on every input}\\ &\text{implies } \text{domain}(M_{f(x)}) = \text{domain}(M_1)\\ &\text{implies } f(x) \in K_P,\end{aligned}$$

and conversely,

$$\begin{aligned}&M_R \text{ does not halt on } x\\ &\text{implies } \text{domain}(M_{f(x)}) = \emptyset\\ &\text{implies } f(x) \notin K_P.\end{aligned}$$

So $x \in R$ iff M_R halts on x iff $f(x) \in K_P$. Moreover, as in the proof of Theorem 8, the function f is total recursive. Hence, $R \leq_m K_P$. ■

Theorem 11. The set $K_{\text{tot}} = \{M : \text{domain}(M) = \Sigma_M^*\}$ is neither r.e. nor co-r.e.

Proof: By Rice's theorem, K_{tot} is $\leq_m$-hard for r.e. sets, and so is not co-r.e. To show that K_{tot} is not r.e., it is enough to show that $\overline{K_2} \leq_m K_{\text{tot}}$. To do this, for any string $x \in \{\mathsf{a}, \mathsf{b}\}^*$, we can define a new Turing machine $M_{f(x)}$ that operates as follows

> "On input w, simulate M_x running on blank tape for $|w|$ steps. If M_x does *not* halt in this number of steps, then halt, otherwise go into an infinite loop."

Then $M_x \in \overline{K_2}$ iff $M_{f(x)}$ halts on all inputs w iff $M_{f(x)} \in K_{\text{tot}}$. ■

9 Turing Reducibility and Relativization

Many-one reducibility is a good technical tool, but as we noted in §6, it does not completely capture the idea of reducing one problem to another. We introduce ***Turing reducibility***, $\leq_T$, as a better formulation of this general notion.

Informally, A is ***recursive in*** B iff there is some program which decides membership in A, where the program is allowed to repeatedly "call a subroutine" to answer questions about membership in B. It may use the answers about B however it likes. (In contrast, many-one reducibility allows only a single question about membership in B, *viz.*, "$f(x) \in B$", and must return the same answer as that question.)

We formalize "calling a subroutine" for B by defining Turing machines with *language inputs* as well as string inputs. These are Turing machines with an extra tape, called the *language tape*. The head on the ordinary tape operates as usual. The head on the language tape is two-way read-only.

Although we can define a Turing machine tape as containing an infinite number of squares all but a finite number of which are blank, we can equally well think of the tape as finite but able to grow further at any point in a computation. In fact, we formalized Turing machine computations in just this way using configurations which contained only the finite, nonblank portion of the tape at any step of the computation. Now likewise thinking of the language tape as finite but growing, we can define configurations and computations involving the language tape as was done for the regular worktape, except that when the language tape needs to be extended by an additional tape square, the new square, instead of always being marked with a blank, is sometimes marked with a 1 and sometimes with a 0. In particular, as the language-tape head moves into new squares, the n^{th} tape square added to the language tape contains 1 iff the n^{th} word in Σ_B^* in canonical order is in B.

Of course if B is not decidable, there is no *effective* way to extend the language tape with properly marked squares as computations proceed. For this reason, machines with language inputs are sometimes called *oracle machines*, and the language input B is called "the oracle" since the language tape for B miraculously gives correct answers about a language which may be undecidable. But the binary values of the language-tape cells, and hence the computational behavior of oracle machines, are perfectly well-defined mathematically.

To run an oracle Turing machine on string input x and oracle B, we start with $\#x\#$ on the work tape in the normal way. We start the language-tape head on the leftmost square of the language tape, and have this square contain a blank. Switching back to describing the language tape as a completed infinite tape, we see that it contains the values of the characteristic function, c_B, of B on the successive words over the alphabet of B in canonical order. For example, if the alphabet of B consists of the symbols a and b, in that order, the complete, infinite language tape looks like:

#	$c_B(e)$	$c_B(a)$	$c_B(b)$	$c_B(aa)$	$c_B(ab)$	$c_B(ba)$	$c_B(bb)$	...

When we run an oracle machine M on inputs x and B, we are essentially doing a computation as if we had a subroutine for deciding membership in B.

Definition 8. A is *Turing-reducible to* B, in symbols $A \leq_{\mathrm{T}} B$, iff there is an oracle Turing machine M which, given fixed oracle B, halts on all string inputs $x \in \Sigma_A^*$ with output $c_A(x)$.

A is *r.e. in* B if there is an oracle Turing machine M which, given fixed oracle B, halts on input $x \in \Sigma_A^*$ iff $x \in A$.

Synonyms: $A \leq_{\mathrm{T}} B$ iff A is *recursive* in B iff A is *decidable* in B.

Theorem 12. Basic Facts about Turing Reducibility:

- $A \leq_{\mathrm{m}} B$ implies that $A \leq_{\mathrm{T}} B$.
- If R is a recursive set, then A is recursive iff A is recursive in R, and A is r.e. iff A is r.e. in R.
- $A \leq_{\mathrm{T}} B$ and $B \leq_{\mathrm{T}} C$ imply $A \leq_{\mathrm{T}} C$.
- $A \leq_{\mathrm{T}} B$ iff $A \leq_{\mathrm{T}} \overline{B}$ iff $\overline{A} \leq_{\mathrm{T}} B$.
- It is not the case that $A \leq_{\mathrm{T}} B$ implies $A \leq_{\mathrm{m}} B$. (For example, $K_0 \leq_{\mathrm{T}} \overline{K}_0$, but by Lemma 2 it is false that $K_0 \leq_{\mathrm{m}} \overline{K}_0$.)

We define the *Halting Problem relative to* B to be

$$B' = \{(M, x) \ : \ M \text{ is an oracle Turing machine which halts on input } x \text{ and oracle } B\}\ .$$

So K_0 amounts to $\emptyset'$. B' is also called the *jump* of B.

Theorem 13. (Relativized Halting Problem) B' is r.e. but not recursive in B.

Proof: Same as the corresponding proofs (Corollary 2 and Theorem 7) for the ordinary halting problem $K_0 (= \emptyset')$, with all the ordinary Turing machines in the original proofs replaced by oracle Turing machines with fixed oracle B. ■

We say $A <_{\mathrm{T}} B$ iff $A \leq_{\mathrm{T}} B$ and $B \not\leq_{\mathrm{T}} A$. Thus, we have

Corollary 6. $B <_{\mathrm{T}} B'$.

Define

$$B^{(0)} = \emptyset\ ,$$
$$B^{(n+1)} = (B^{(n)})'.$$

By the preceding theorems, $B^{(n)} <_{\mathrm{T}} B^{(n+1)}$ for all n. So the sequence of sets

$$\emptyset <_{\mathrm{T}} \emptyset' <_{\mathrm{T}} \emptyset^{(2)} <_{\mathrm{T}} \cdots$$

has strictly more difficult successive membership problems. This sequence, or more precisely the sequence of families $\{L : L \leq_{\mathrm{m}} \emptyset^{(n)}\}$ for $n = 0, 1, \ldots$, is called the *Arithmetic Hierarchy.*

Thus, there is a rich classification possible among undecidable problems. Various natural decision lie along the arithmetic hierarchy. For example, $\overline{K_{\mathrm{tot}}}$ turns out to be $\equiv_{\mathrm{m}} \emptyset^{(2)}$.

Theorems and proofs about machines that carry over without change to oracle machines are said to *relativize.* Most of our theorems relativize. For example, the remark that a set A is r.e. iff $A \leq_{\mathrm{m}} K_0$ (which follows immediately from the facts that K_0 is a $\leq_{\mathrm{m}}$-complete r.e. set and that r.e. inherits down $\leq_{\mathrm{m}}$) relativizes to:

Theorem 14. A is r.e. in B iff $A \leq_{\mathrm{m}} B'$.

Likewise Theorem 1 relativizes to:

Theorem 15. $A \leq_{\mathrm{T}} B$ iff A is both r.e. and co-r.e. in B.

Some further relativizations:

Theorem 16. The following are equivalent:

- A is r.e. in B.
- $A = \text{domain}(\varphi)$, where φ is a partial recursive function in B.
- $A = \text{range}(f)$, where f is a total recursive function in B, or $A = \emptyset$.

Theorem 17. Define the *blank-tape halting problem relative to* B, $K_2^{(B)}$, to be

$$\{M : \text{oracle machine } M \text{ with oracle } B \text{ halts on blank input}\}.$$

Then $K_2^{(B)} \equiv_{\mathrm{m}} B'$.

Putting these relativized facts together, we can conclude as a final result in these notes:

Theorem 18. A'' is neither r.e. nor co-r.e. in A.

Proof: To show a language B is not r.e. in A, it suffices to show that $\overline{A'} \leq_{\mathrm{m}} B$. This follows because $\overline{A'}$ is not r.e. in A and non-r.e.-ness inherits up $\leq_{\mathrm{m}}$.

But $A' \leq_{\mathrm{T}} A'$ trivially, so by Theorem 15, A' is r.e. in A'. Therefore, by Theorem 14, $A' \leq_{\mathrm{m}} A''$ so $\overline{A'} \leq_{\mathrm{m}} \overline{A''}$, and we conclude that A'' is not co-r.e. in A. But also $\overline{A'} \leq_{\mathrm{T}} A'$, so similarly $\overline{A'} \leq_{\mathrm{m}} A''$, and we conclude that A'' is not r.e. in A. ■

Corollary 7. K_0' is neither r.e. nor co-r.e.

A Axioms for Coding Computable Functions (Optional)

There is a simple set of axioms which characterize the properties of the set of codewords abstractly without having to mention d or Turing machines at all; only the general notion of partial recursive function need be known. For simplicity we'll state the axioms for recursive functions on strings over the alphabet {a,b}. First, saying that d is a coding certainly implies that $d(M)$ and x uniquely determine the behavior of M on x. The important part of d is this mapping from $d(M)$ and x to the output of M on x. This is captured the idea of a *coder* function.

Definition 9. A ***partial-recursive-function coder*** is a partial function

$$v : \{\mathtt{a},\mathtt{b}\}^* \times \{\mathtt{a},\mathtt{b}\}^* \to \{\mathtt{a},\mathtt{b}\}^*$$

such that for every partial recursive function $\varphi : \{\mathtt{a},\mathtt{b}\}^* \to \{\mathtt{a},\mathtt{b}\}^*$, there is a string $z_\varphi \in \{\mathtt{a},\mathtt{b}\}^*$ with the property that for all $x \in \{\mathtt{a},\mathtt{b}\}^*$,

$$v(z_\varphi, x) = \varphi(x).$$

Such a z_φ is called a ***code*** or ***Gödel number*** for φ. Coders are also called ***universal functions*** for the partial recursive functions.

Having chosen a partial-recursive-function coder v, the axiomatic definition of K_1 becomes

Definition 10.

$$K_1 = \{x \in \{\mathtt{a},\mathtt{b}\}^* : (x,x) \in \mathrm{domain}(v)\}.$$

The intuitive requirement that d be recursive serves to guarantee that the coding is effectively decipherable—there is some recursive way to recover information about M from $d(M)$. This is abstractly captured by the ***universal machine theorem***:

Theorem 19. There is a partial-recursive-function coder v which is itself a partial recursive function.

One other property of the coder will be needed. In addition to determining the input/output behavior of M, the code $z = d(M)$ can be modified to obtain the code for simple variants of M. For example, from z and $y \in \{a, b\}^*$, one can easily construct a machine $M_{z,y}$ which replaces every input x given to it by the word $pair(y, x) \in \{a, b\}^*$ which codes the pair (y, x), and then acts like M on the new input $pair(y, x)$. This is abstractly captured by thinking of the function $s(z, y) = d(M_{z,y})$ and saying it is total and recursive. For historical reasons, this is known as the *s^m_n-Theorem*:

Theorem 20. There is a total recursive function $s : \{a, b\}^* \times \{a, b\}^* \to \{a, b\}^*$ such that for all $x, y, z \in \{a, b\}^*$,

$$v(z, pair(y, x)) = v(s(z, y), x).$$

In these notes we gave explanations in terms of a function d coding Turing machines into binary strings. One can give the details of how such an encoding can be accomplished, but we can confidently gloss over these details because a careful reading of all the arguments above will reveal that Theorems 19 and 20 are the only facts we needed about the coding to obtain all the results given in these notes. In fact, there is an elegant "recursive isomorphism" theorem due to Hartley Rogers which explains why all reasonable codings—namely those that satisfy the universal machine and s^m_n-theorems—have the same properties with respect to computability.

Theorem 21. Let v_1 and v_2 be two coders satisfying the universal machine Theorem 19 and the s^m_n-Theorem 20. Then there is a total recursive one-one and onto function $t : \{a, b\}^* \to \{a, b\}^*$ such that

$$v_1(z, x) = v_2(t(z), x)$$

for all $z, x \in \{a, b\}^*$.

The proof is not very hard but a bit long, and to save time we'll skip it.

Theorem 21 can be understood as saying that there is a *one-one onto* recursive function t translating, say, Scheme programs into equivalent CLU programs. So Scheme and CLU (and Turing machines, RAM's, *etc.*) are indistinguishable from the point of view of general computability theory. This is a clear warning that the conclusions of the theory will *not* bear on some central Computer Science issues—such as which language features which make Scheme more desirable then CLU for certain problems, and vice-versa. On the other hand, the conclusions of the theory, especially *negative* conclusions about the limitations of computability, will hold with great generality and can't be gotten around by changing programming languages.

6.044J/18.423J: Computability, Programming, and Logic — Handout 11
Massachusetts Institute of Technology — 6 October 1989

Problem Set 2 Solutions

General Information. This handout includes some of the best solutions submitted by students for Problem Set 2.

The grades went as follows:

	number submitted	min	max	mean	median
1	13	15	25	19.5	21
2	12	3	25	18.3	20
3	13	3	25	17.2	19
4	12	10	25	19.7	22

page 1

Jeff Johnson
6.044
PS2
#1

1(a) $f: A \to P(A)$ To show that f doesn't map A onto P(A), I will generate a subset of A that f never mapps an element of A to.

$\bar{D} = \{a \in A : a \notin f(a)\}$. The claim is that $\bar{D}$ is never mapped to by f. Proof is by contradiction:

Assume $\bar{D} = f(a)$ for some a.

Case 1: $a \in f(a)$. In this case $\bar{D}$ does NOT contain a by def. Then in this case $\bar{D} \neq f(a)$.

Case 2: $a \notin f(a)$. In this case $\bar{D}$ contains a by def. Thus in this case also $\bar{D} \neq f(a)$. *

Thus by contradiction we have shown that ∃ no onto function from $f: A \to P(A)$.

clear + concise ✓

b) To show that $P_{fin}(A)$ is enumerable, it suffices to show a scheme to list all possible sets. This will be done as follows: Let the elements of A be called $a_1, a_2, a_3, \ldots$ List first all combinations in order, using just a_1, then a_1 & a_2, then a_1, a_2, a_3, etc.

List: $\emptyset$, a_1, a_1a_2, a_2, $a_1a_2a_3$, a_1a_3, a_2a_3, a_3, $a_1a_2a_3a_4$, $a_1a_2a_4$, $a_1a_3a_4$, a_1a_4, … ✓

Problem 2a: Suppose there is a universal function $u : \mathbf{N}^2 \rightarrow \mathbf{N}$ for the primitive recursive functions that is primitive recursive. Then in particular, the function $f = u(x,x) + 1$ is primitive recursive. Now let n_f be the "encoding" of the primitive recursive function f. We then have $f(n_f) = u(n_f, n_f) + 1$. But since u is universal,

$$u(n_f, n_f) = f(n_f) = u(n_f, n_f) + 1$$

hence $0 = 1$, a contradiction. Thus, u cannot be primitive recursive.

Problem 2b: One should invoke "Church's Thesis" here instead of giving a complete program. It's not hard to see that all of the basic functions, plus composition and primitive recursion, can be simulated in Scheme. Using these subroutines, one can write an *interpreter* for the primitive recursive notation. Since a "universal function" is basically an interpreter that takes the encoding of a program and a number and simulates the program on the number, the universal function is programmable in Scheme (and hence computable.)

Pahl Law
6.044J
PS 2
Problem 3
9/29/89

(3) (i) Prove the f has a fixed point.

$a_0 \le f(a_0)$ ∵ $f(a_0) \in A$ and a_0 is the least element

$f(a_0) \le f(f(a_0))$ ∵ f is monotone

let $f^n(a_0) = \underbrace{f(f(f\ (f(a_0))))}_{n\text{ times}}$

Assume that $f^{k-1}(a_0) \le f^k(a_0)$

Prove $f^k(a_0) \le f^{k+1}(a_0)$

$f^{k-1}(a_0) \le f^k(a_0)$ by assumption

$f(f^{k-1}(a_0)) \le f(f^k(a_0))$ f is monotone

∴ $f^k(a_0) \le f^{k+1}(a_0)$

Since A is finite, $\exists b \in A$ $f^n(a_0) = b$ $f(b) = b$

This is a fixed point of f thus $\forall n' > n$ $f^{n'}(a_0) = f^n(a_0) = b$

(ii) Prove if $f(y) \le y$, then $f^k(a_0) \le y$ $\forall k \in \{1, 2, \ldots n\}$ of the sequence $f^0(a_0), f^1(a_0), f^2(a_0), \ldots$ $(f^0(a_0) = a_0)$

$a_0 \le y$ ∵ a_0 is the least element

$f(a_0) \le f(y)$ ∵ f is monotone

$f(a_0) \le y$ ∵ $f(y) \le y$

[redacted] $f^i(a_0) \le y$

ove $f^{i+1}(a_0) \le y$

$f^i(a_0) \le y$ by induction hypothesis

$f(f^i(a_0)) \le f(y)$ f is monotone

$f^{i+1}(a_0) \le f(y)$

∴ $f^{i+1}(a_0) \le y$ ∵ $f(y) \le y$

✓

25/25

Ted Hontz
6.044J/18.423J
Assignment #2
Problem #4
10/2/89

4. Let $f_n'[f] : \mathbb{N}^n \longrightarrow \mathbb{N}$ be defined by

$f_n'[f] = Cn_{n+1,n}[f, id_1^n, id_2^n, id_3^n, \ldots, id_n^n, Cn_{1,n}[z, id_1^n]]$

and

$g_n'[f] : \mathbb{N}^{n+2} \longrightarrow \mathbb{N}$ be defined by

$g_n'[f] = Cn_{2,n+2}[sum, id_{n+2}^{n+2}, Cn_{n+1,n+2}[f, id_1^{n+2}, \ldots, id_n^{n+2}, Cn_{1,n+2}[s, id_{n+1}^{n+2}]]]$

and

$h_n[f] : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ be defind by

$h_n[f] = Pr_n[f_n'[f], g_n'[f]]$

Now $GenSum_n[f,g] : \mathbb{N}^n \to \mathbb{N}$ may be defined by

$GenSum_n[f,g] = Cn_{n+1,n}[h_n[f], id_1^n, id_2^n, \ldots, id_n^n, g]$ ✓

25/25

6.044J/18.423J: Computability, Programming, and Logic — Handout 12
Massachusetts Institute of Technology — 6 October 1989

Problem Set 4

Due: 13 October 1989

Problem 1. (Recursive and Recursively Enumerable Sets)

1(a). Show that the recursive sets are closed under union and intersection. That is, given any two recursive sets $S_1 \subseteq \mathbf{N}$ and $S_2 \subseteq \mathbf{N}$, prove that $S_1 \cup S_2$ and $S_1 \cap S_2$ are recursive.

1(b). Show that the r.e. sets are closed under union and intersection.

Problem 2. Show that a set $S \subseteq \mathbf{N}$ is r.e. iff $S = \emptyset$ or S is the range of a *primitive* recursive function $p : \mathbf{N} \rightarrow \mathbf{N}$. (Hint: Suppose M is any Turing machine. Using the primitive recursive functions defined in class for proving that μ-recursive functions simulate Turing machines, show that the predicate of n and m:

> "M halts on input n in exactly m steps"

is primitive recursive.)

Problem 3. Show that a set $S \subseteq \mathbf{N}$ is recursive iff S is finite or S is the range of an increasing total recursive function. (A function f is *increasing* if $n < m$ implies $f(n) < f(m)$.)

6.044J/18.423J: Computability, Programming, and Logic — Handout 13
Massachusetts Institute of Technology — 11 October 1989

Problem Set 3 Solutions

General Information. This handout includes some of the best solutions submitted by students for Problem Set 3.

The grades went as follows:

	number submitted	min	max	mean	median
1	14	20	25	24.5	25
2	10	10	25	22.2	25
3	5	10	25	20.0	25

Aaron Wallach

P1 ~~My~~ algorithm counts down the 1's and increments the binary rep.

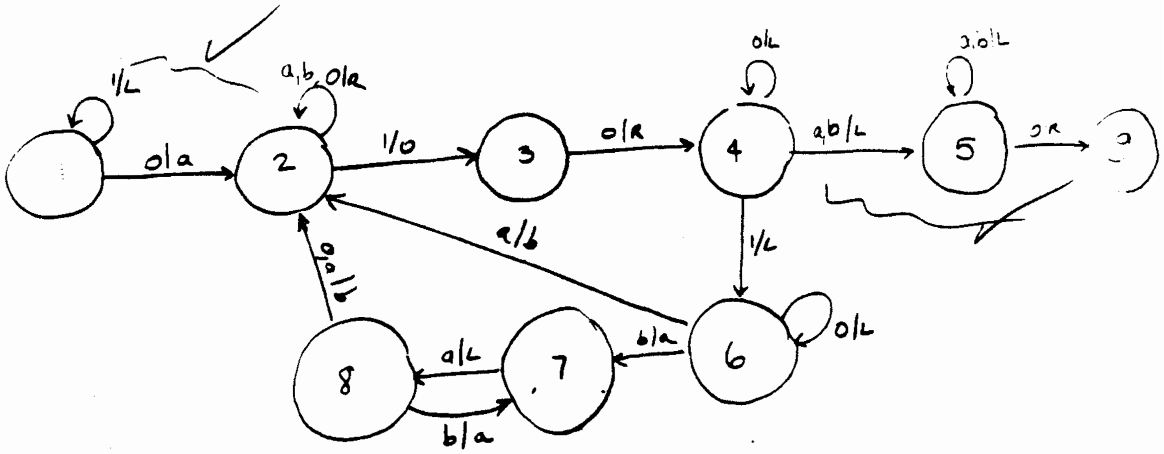
Intalization - write an a to the left of the 1s ✓

Loop - Erase a 1. If it was the last 1 (the block to the right has a 0), halt ✓
Otherwise, increment the binary representation. ✓

How to increment the binary representation ✓

1) Move left until you get to either an a or a b ✓
2) If the cell is a zero or an a, change it to a b ✓
Otherwise it is a b, change it to an a, move left and go to ② ✓

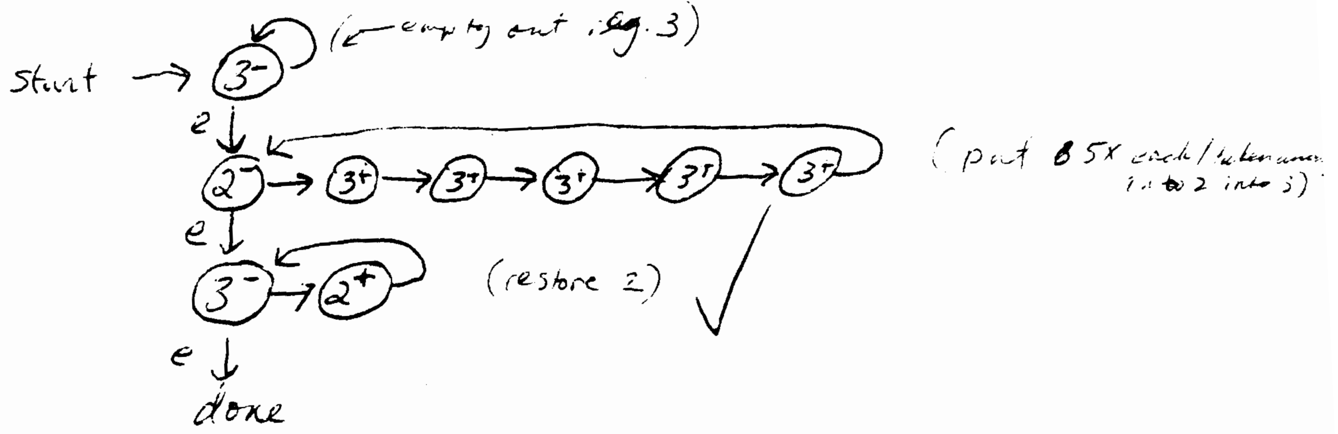
FSM



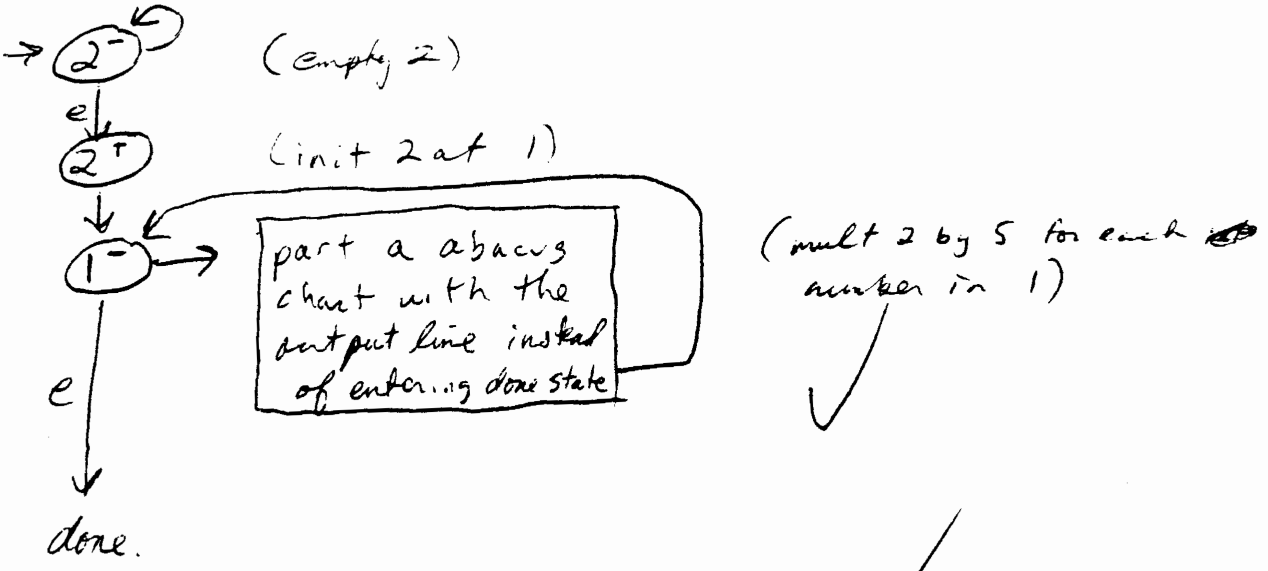
David Taylor 6.044J/18.423J PS 3 due 10/6/89

prob 2.

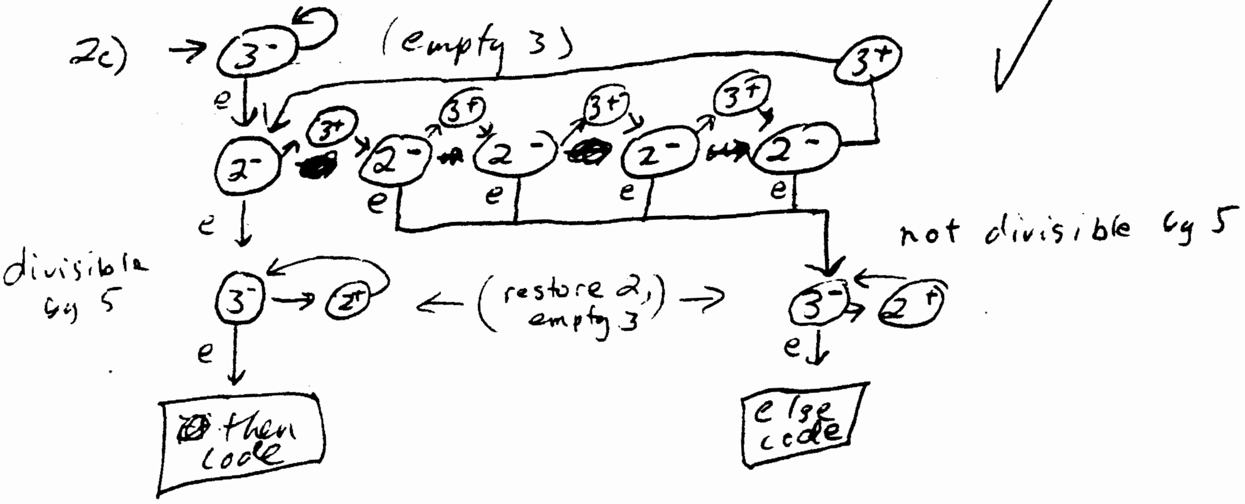
2a)



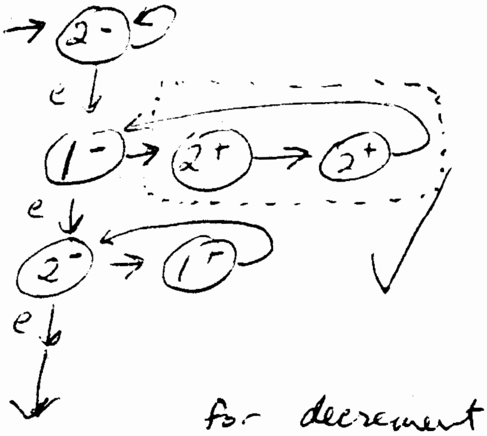
2b)


2⁻
(empty 2)
e
2⁺
(init 2 at 1)
1⁻
part a abacus chart with the output line instead of entering done state
(mult 2 by 5 for each number in 1)
e
done.

2c)

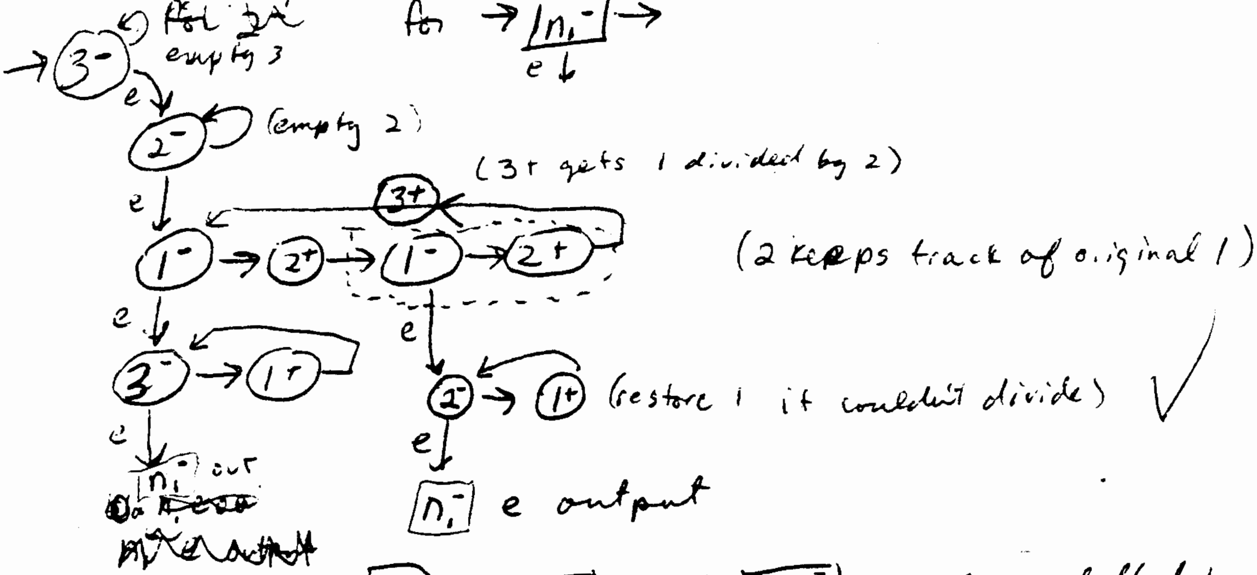


2d. for (black box) $\boxed{n_1^+}$ we want to multiply reg. 1 by 2, which can be done by



likewise, ~~fore~~ $\boxed{n_2^+}$ we can ~~re~~place the dotted box with 3 2^+'s to mult by 3, ~~and~~ or with 5 2^+'s to do $\to\boxed{n_3^+}\to$ or with 7 for $\to\boxed{n_4^+}\to$

for decrement and test, ~~one~~ modify part c, ~~and~~ ~~replace code A~~ for $\to\boxed{n_1^-}\to$ e↓



for $\boxed{n_2^-}$, $\boxed{n_3^-}$, and $\boxed{n_4^-}$ replace dotted box with 2, 4, and 6 sequences of $1^- \to 2^+ \to$ e↓ with all e outputs going to the same next step.

(This could be done using only reg 1 and 2, but this seems to be clearer)

25/25

2e) ~~being~~ using these techniques, we can encode all registers as prime powers, and since primes are infinite this will be able to encode infinite registers. We can use 2 and 3 to manipulate the infinite registers of 1, and can get any output. Or else, instead of using all primes, we could encode n_1 to n_4 else of form $2^{n_{11}}\cdot 3^{n_{12}}\cdot 5^{n_{13}}\cdot 7^{n_{14}}$ to $2^{n_{41}}\cdot 3^{n_{42}}\cdot 5^{n_{43}}\cdot 7^{n_{44}}$ and repeat this indefinitely to get our infinite registers, stacking above functions.

Problem 3: No one provided the answer we had in mind, though many did solve the problem correctly. Each correct solution depended on the ideas stated in Problem 2; an example of such a solution follows.

One can solve the problem without depending on Problem 2, and the argument is important enough to merit discussion. We must show that the set of functions computed by the extended abacus machines are the same as the set computed by the ordinary abacus machines. The idea is to prove, using simulation arguments, that

"extended abacus computable implies Turing computable"

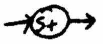
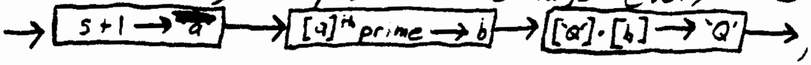
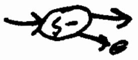
and vice versa. Since the Turing computable functions are the same as the abacus computable functions, this will suffice to prove the theorem.

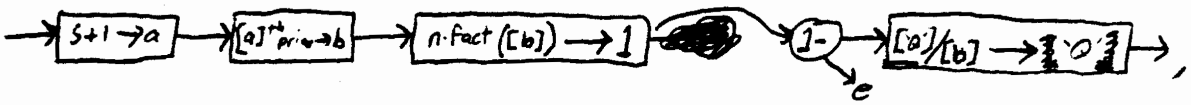
To show that "extended abacus" computable functions are Turing computable, we need to show how to construct a Turing machine to simulate any extended abacus. Use the method given in class to encode abacus registers on the tape. Then to simulate the instruction "$[i] \rightarrow [j]$", for example, we move to the $[j]$ block and copy $[i]$ over it. This requires some care—for example, using special symbols to mark block j since we cannot store $[j]$ in the finite state—but the idea is easy enough to see. Thus, Turing machines can simulate extended abacus machines. Since we proved in class that ordinary abacus machines can simulate Turing machines, the argument is finished.

Ted Hontz
6.044J/18.423
Problem Set 3
Problem #3
10/6/89

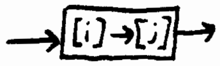
3. I will show, informally, how any abacus program (flowchart) which contains indirection instructions may be translated into one without indirection instructions, thus showing that the class of abacus-machine-with-indirection-computable functions is the same as the class of abacus-computable functions (since the former encompasses the latter, and can be simulated by it)

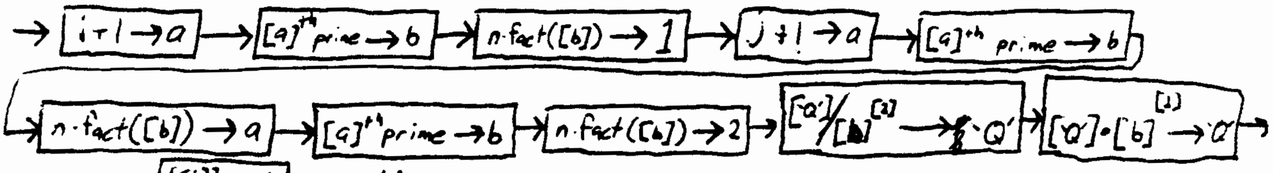
To change a flowchart with indirection into one without, we will need to first write a couple of abacus programs (without indirection): One that, given an input n in a particular register, a, computes the n^{th} prime number, which ends up in another particular register, b, and another program, which I will call n-facts, which takes an input p in a particular registar, c, and computes the number of factors of p that $[Q]$, the number of stones in register 'Q', has. We know that these functions are computable, because they require only addition, subtraction, computation of $x \bmod y$, and ~~compariso~~ branching on a 0 value in a regist all of which are abacus-computable. I will not give explicit abacus programs for these functions. We also need the restriction that register 'Q' is not used by the original abacus-with-indirection flowchart (this is why it is not labeled with an integer), and that it starts out containing a 1.

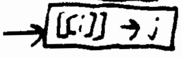
To make the translation, simply change every  in the flow chart to , replace every  with

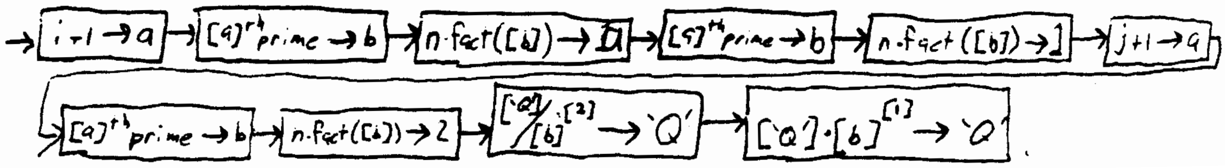


,

every  with



and every  with



.

If the output of the original abacus was in register O, the new output will be the number of factors of the O+1st prime number in 'Q'.

All of the blocks that we replaced the original abacus and indirection instructions with can be expanded to abacus instructions without indirection ($\textcircled{s-}$'s and $\textcircled{s+}$'s), because each of the blocks computes an abacus-computable function and puts the result in a register, so the new flowchart represents an abacus-computable function whose output (which can be decoded with a post-processor from register 'Q') is the same as the abacus-with-indirection flowchart

∴ abacus-with-indirection-computable functions are abacus computable.

25/25

6.044J/18.423J: Computability, Programming, and Logic — Handout 14
Massachusetts Institute of Technology — 13 October 1989

Quiz 1

Instructions. Do all problems in the provided blue book, carefully labeling solutions with their corresponding numbers. All problems count equally.

Problem 1. Suppose $f : \mathbf{N} \to \mathbf{N}$ and $g : \mathbf{N} \to \mathbf{N}$. We say that f and g are defined by *simultaneous primitive recursion* from functions $h : \mathbf{N}^3 \to \mathbf{N}$, $h' : \mathbf{N}^3 \to \mathbf{N}$ and natural numbers n_0, n_1 iff

$$\begin{aligned} f(0) &= n_0 \\ g(0) &= n_1 \\ f(x+1) &= h(x, f(x), g(x)) \\ g(x+1) &= h'(x, f(x), g(x)) \end{aligned}$$

Show that if h and h' are primitive recursive, so are f and g. (Hint: Consider the function $p(x) = pair(f(x), g(x))$.)

Problem 2. Explain why the set of multi-variable, Diophantine polynomials with an integer root vector is r.e.

Problem 3. Suppose $S \subseteq \mathbf{N}$. Prove that S is an infinite r.e. set iff S is the range of a *one-to-one* (*i.e.*, injective) total recursive function $f : \mathbf{N} \to \mathbf{N}$.

Quiz 1 Solutions

Remark. The solutions below contain roughly what we'd expect to see in a "perfect" answer. Quiz solutions, like the ones below, may be somewhat sketchy but should convey the right ideas.

First, here are the relevant statistics from the quiz.

Score	Students
100 - 91:	**
90 - 81:	**
80 - 71:	**
70 - 61:	*
60 - 51:	**
50 - 41:	**
40 - 31:	
30 - 21:	*
20 - 11:	**
10 - 0:	*

The median score was 61; the mean was 55.1.

Problem 1. Suppose $f : \mathbf{N} \rightarrow \mathbf{N}$ and $g : \mathbf{N} \rightarrow \mathbf{N}$. We say that f and g are defined by *simultaneous primitive recursion* from functions $h : \mathbf{N}^3 \rightarrow \mathbf{N}$, $h' : \mathbf{N}^3 \rightarrow \mathbf{N}$ and natural numbers n_0, n_1 iff

$$\begin{aligned} f(0) &= n_0 \\ g(0) &= n_1 \\ f(x+1) &= h(x, f(x), g(x)) \\ g(x+1) &= h'(x, f(x), g(x)) \end{aligned}$$

Show that if h and h' are primitive recursive, so are f and g. (Hint: Consider the function $p(x) = pair(f(x), g(x))$.)

Solution. Define the function $p : \mathbf{N} \rightarrow \mathbf{N}$ as follows:

$$\begin{aligned} p(0) &= pair(n_0, n_1) \\ p(x+1) &= pair(h(x, left(p(x)), right(p(x))), h'(x, left(p(x)), right(p(x)))) \\ &= h''(x, p(x)) \end{aligned}$$

where *pair* : $\mathbf{N}^2 \longrightarrow \mathbf{N}$ is some primitive recursive pairing function like

$$pair(n, m) = 2^n \cdot 3^m$$

or perhaps

$$pair(n, m) = \frac{1}{2}(n + m)(n + m + 1) + m$$

(*cf.* Boolos & Jeffrey, page 161), and *left*, *right* extract the left and right components of a coded pair. Since *pair*, *left*, *right*, h, and h' are primitive recursive, so is $h''(x, y)$ and hence so is p. Then let

$$\begin{aligned} f(x) &= left(p(x)) \\ g(x) &= right(p(x)) \end{aligned}$$

Then f and g are also primitive recursive.

Problem 2. Explain why the set of multi-variable, Diophantine polynomials with an integer root vector is r.e.

Solution. We must give a program that halts on precisely the Diophantine equations with an integer root vector:

> "Given a multi-variable Diophantine polynomial as input, determine the number m of variables in the polynomial. Then for $k = 0, 1, 2, \ldots$, evaluate the polynomial on all m-tuples of integers where each integer has absolute value $\leq k$. Halt if the polynomial ever equals 0."

This algorithm halts at each stage k, because there are only a finite number of m-tuples with each integer having absolute value $\leq k$. The algorithm thus gets to try each possible integer root vector, and so meets the requirements. Thus, the specified set of polynomials is r.e.

Problem 3. Suppose $S \subseteq \mathbf{N}$. Prove that S is an infinite r.e. set iff S is the range of a *one-to-one* (*i.e.*, injective) total recursive function $f : \mathbf{N} \longrightarrow \mathbf{N}$.

Solution. ($\Rightarrow$) Suppose S is an infinite r.e. set. Let P be the program whose domain is S. Define a new program as follows:

> "On input k, dovetail P on all inputs. Stop when the machine has seen $k + 1$ inputs for which P halts, and output the $(k + 1)$st input on which P halted."

This program computes a partial recursive function $f : \mathbf{N} \rightarrow \mathbf{N}$. Since S is infinite, and since the program uses dovetailing, f is total and $range(f) = S$. Since f never outputs the same answer for distinct inputs, f is one-to-one.

($\Leftarrow$) Suppose S is the range of a one-to-one total recursive function f. Then S is r.e. since it is the range of a partial recursive function, and S is infinite since it is the range of a total one-to-one function on $\mathbf{N}$.

Problem Set 5

Due: 27 October 1989

Problem 1. Let S be the set of Turing machines (TM's) that halt on input 3 and run forever on input 4. Show that S is neither r.e. nor co-r.e.

Problem 2. Classify the following r.e. sets according to whether they are recursive or nonrecursive. If the set is not recursive, state whether its nonrecursiveness follows from Rice's theorem. If the set is not recursive and this fact does not follow from Rice's theorem, prove that the set is not recursive.

2(a). The set of TM's that halt on input 3 and halt on input 4.

2(b). The set of TM's that halt on input 5 in exactly 10^9 steps.

2(c). The set of TM's each of whose domain is *not* a set of prime numbers.

2(d). The set of TM's which, on input 0, run for exactly 2^i steps for some $i \geq 0$.

2(e). The set of SCHEME[1] S-expressions which *either* halt when applied to an infinite number of different integers, *or* there is a number k such that the expression does not halt when applied to any integer $\geq k$.

2(f). The set of SCHEME S-expressions P which compute a partial recursive function $f : \mathbf{N} \rightarrow \mathbf{N}$, such that $range(f)$ contains an integer $k \geq$ number of atoms in P.

[1] Idealized language with an unbounded heap (memory).

6.044J/18.423J: Computability, Programming, and Logic — Handout 17
Massachusetts Institute of Technology — 20 October 1989

Problem Set 4 Solutions

General Information. This handout includes some of the best solutions submitted by students for Problem Set 4.

The grades went as follows:

	number submitted	min	max	mean	median
1	11	17	25	21.9	24
2	10	5	25	15.5	15
3	11	14	25	20.0	23

Ted Hoatz
6.044J/18.423J
Problem Set 4
Problem #1
10/13/89

1. (a.) $S_1 \subseteq \mathbb{N}$ and $S_2 \subseteq \mathbb{N}$ are recursive $\Rightarrow$
there exist total computable functions $C_{S_1}: \mathbb{N} \to \{0,1\}$ and $C_{S_2}: \mathbb{N} \to \{0,1\}$ such that
$C_{S_1}(n) = 1$ for $n \in S_1$ and $C_{S_1}(n) = 0$ for $n \notin S_1$
and $C_{S_2}(n) = 1$ for $n \in S_2$ and $C_{S_2}(n) = 0$ for $n \notin S_2$

Let $C_{S_1 \cup S_2}: \mathbb{N} \to \{0,1\}$ be defined by
$C_{S_1 \cup S_2}(n) = C_{S_2} + C_{S_1} \dot{-} C_{S_1} \cdot C_{S_2} = C_{S_1}(n)$ OR $C_{S_2}(n)$

$C_{S_1 \cup S_2}$ is total recursive, and ~~defines~~ determines $S_1 \cup S_2$
$\therefore S_1 \cup S_2$ is recursive

Let $C_{S_1 \cap S_2}: \mathbb{N} \to \{0,1\}$ be defined by
$C_{S_1 \cap S_2}(n) = C_{S_1} \cdot C_{S_2}$
$C_{S_1 \cap S_2}$ is total recursive, and determines $S_1 \cap S_2$
$\therefore S_1 \cap S_2$ is recursive ✓

(Solution due to Michael Frumkin)

1. b) If S_1 and S_2 are r.e., then there are programs P_{S_1} + P_{S_2} s.t. $dom(P_{S_1}) = S_1$ + $dom(P_{S_2}) = S_2$

we can write $P_{S_1 \cup S_2}$ as follows

"Given input n, run P_{S_1} and P_{S_2} on n in parallel (by dovetailing them)
If <u>either</u> P_{S_1} <u>or</u> P_{S_2} halts then halt with output $= n$, else diverge"

It is obvious that the doman of $P_{S_1 \cup S_2}$ is an r.e. set $\therefore$ $S_1 \cup S_2$ is r.e.

Similarly,

we can write $P_{S_1 \cap S_2}$ as follows

"Given input n, run P_{S_1} and P_{S_2} on n in parallel by dovetailing
If <u>both</u> P_{S_1} <u>and</u> P_{S_2} halt then halt with output $= n$, else diverge"

$P_{S_1 \cap S_2}$ is obviously defining an r.e. process

$\therefore$ $S_1 \cap S_2$ is r.e.

no need to dove tail in this instance

25/25

Paul Law
6044J
Problem Set 4

10/13/89

2. $\Leftarrow$

(i) $S = \emptyset$, then the Turing machine that diverges on all inputs has S as its domain. So S is r.e.

(ii) f $S \neq \emptyset$. $S = \text{range}(p)$, p is a p.r. function, then S is the range of a partial recursive function, since every p.r. function is a partial recursive function.
We showed in class that it is equivalent to S being r.e

$\Rightarrow$

(i) if S is r.e and $S = \emptyset$, then the theorem stated in the problem holds trivially.

(ii) if $S \neq \emptyset$, then $S = \text{dom}(M)$ for some Turing machine M.
Let P be the predicate that holds for a pair (n, m) iff M halts on input n in exactly m steps.
Define $\underline{P}(n,m) = 1$ if $g(n,m) = 0$
$= 0$ otherwise
where g is defined as a primitive recursive function with the property: if t is a stage not later than the stage where M halts when computing $f(x)$, then $g(x,t) = \langle l, c, r \rangle$ (Abbreviations as in class and in text)

$$g(x, 0) = \langle 0, 1, S(x) \rangle$$
$$g(x, t+1) = \Big\{ \text{ as in book p. 94}$$

Define $p : \mathbb{N} \to \mathbb{N}$ as follows
Since $S \neq \emptyset$, $\exists s_0 \in S$
$p(n) = s_0$ if $\underline{P}(\text{left}(n), \text{right}(n)) = 0$
$= \text{left}(n)$ otherwise

Then $S = \text{range}(p)$ as desired

Jeff Johnson
PS4
6.044
#3

3 Trivial: If S is finite $\to$ S is recursive.
PF: all finite sets are recursive since they can be broken up into individual cases. □

Thm: If S is infinite and recursive then S is the range of an increasing total recursive function.

PF: Since S is recursive then let R be the decidable TM that returns 1 on input $m \in \mathbb{N}$ iff $m \in S$.

We now define a TM as follows:

"On input K run R on inputs $m = 0, 1, \ldots$ until R has returned exactly K 1's. Output the last m."

This finds the k^{th} item in S and returns it. We are guaranteed that the output is increasing since $k+1$ must look at least 1 place further than k.

Also the machine is guaranteed to halt. Since $|S| = \infty$ and $k < |S|$ then the incremental m will eventually find the k^{th} accept state. □

✓

Thm: If S is the range of an increasing total rec function, then S is recursive. Let T = total rec fun.

Build R as follows: "On input K run T on $0, 1, \ldots, k$ if T returns K for any of the $k+1$ inputs, accept, otherwise reject."

This machine can clearly be constructed. Certainly if $T(l)$ returns k, then $k \in S$. Also, if $k \in S$, it will occur before the $k+1$ element. This is true because the function is increasing. For a minimal mapping $T(l) = l$, however if any item prior to l is not in S then $T(l) > l$. Thus, ~~to~~ so if $l \in S$ we only need to run a finite number of checks which means the algorithm will terminate and thus is recursive.

✓

25/25

Review for Quiz 2

MANY-ONE REDUCTIONS

Everyone seems to have their own way of thinking about many-one reductions, and unfortunately most of these ways are misleading or wrong. Let's look once more at the definition:

$A \leq_m B$ means there exists some total recursive function f such that $x \in A$ iff $f(x) \in B$.

This is really all there is to it. Now we also have the following theorem:

MANY-ONE THEOREM: If $A \leq_m B$ then

- *if B is recursive then A is recursive*
- *if B is r.e. then A is r.e.*

Many people seem to think about $A \leq_m B$ as: If I could decide B then I could decide A. Now this is good intuitively, maybe, but it can easily lead you astray. Remember, many-one reductions and our many-one theorem are two different things! If $A \leq_m B$ then we can use the theorem to say that if B is decidable then A is decidable, but this doesn't always hold in the other direction.

What is a good way to think of many-one reductions then? Think of a many-one reduction as "disguising" one problem as another. The question we really want to know the answer to is: Is $x \in A$? But we ask this question: Is $f(x) \in B$? Since $x \in A$ iff $f(x) \in B$, the answer to this question will be the same answer that we wanted! Whoever is telling us the answer to "Is $f(x) \in B$?" is really telling us the answer to "Is $x \in A$?" but we have disguised our question in such a way that we fool the answerer into telling us what we want to know. Now sometimes the answerer will always tell us yes or no (which corresponds to the "recursive" part of the many-one theorem), sometimes the answerer will say yes but refuse to answer if the answer was really no (which is the "r.e." part) and sometimes the answerer will do something different. The many-one reduction doesn't say anything about what kind of answer we will get to our question; all it tells us is how an answer to the question we pose relates to the answer of the question we really wanted to ask.

This is still pretty abstract, but we'll use the idea as the basis of giving hopefully intuitive explanations of how one goes about thinking up many-one reductions in the solutions that follow.

Problem 1. For each of the following languages, indicate whether it is decidable, r.e. but not decidable, co-r.e. but not decidable, or none of these.

1(a). $L_1 = \{M \mid \text{there exist at least 100 Turing Machines with } domain(M)\}$.

DEC. This one is easy. You can add to any TM dummy states that are unreachable or whatever...and easily get 100, no 1000, oh, maybe even more TM's accepting the same language as M without having to do any real thinking at all. You can thus rephrase L_1: $L_1 = \{M \mid M \text{ is a TM}\}$. You know this is easy to check for.

1(b). $L_2 = \{M \mid domain(M) \text{ contains only even numbers}\}$.

CO-RE. L_2 is clearly at least co-r.e. $\overline{L_2} = \{M : \text{there exists some odd number } n \in domain(M)\}$. Just run M on all odd numbers in the standard dovetail fashion—if it halts on any of them, then $M \in \overline{L_2}$. L_2 is not decidable, though, by Rice's Theorem. "Only even numbers" is a nontrivial language property.

1(c). $L_3 = \{M \mid M \text{ halts on } 2, 4, 6 \text{ and diverges on } 27, 28, 31\}$.

NONE. The details are left to you.

1(d). $L_4 = \{(M, k) \mid M \text{ accepts some } w \in \mathbf{N} \text{ in } k \text{ or fewer steps}\}$.

DEC. At first glance this might seem undecidable, but the trick is that you only have to consider inputs of length k or less. You can't even read more than k of the input, so it doesn't really matter if there's more lying around. Finiteness is easy. Just try all encodings of numerals w, where the length of the encoding $(w+1)$ is $\leq k$, one after another, running each for k steps. If $M(w)$ ever halts within that time, accept; otherwise, reject.

Problem 2. Let $L_{MIN} = \{(M, M') \mid domain(M) = domain(M') \text{ and } M' \text{ has the fewest number of states of any TM accepting } domain(M)\}$. Prove that L_{MIN} is undecidable. This demonstrates that there is no algorithm for state minimization for Turing Machines.

At first glance it seems that you can't use Rice's Theorem since "fewest number of states" is clearly a machine property and not a language property. So the next best thing is your favorite: many-one reductions! But what problem should you reduce to L_{MIN}? Well, if you remember earlier problems involving two Turing Machines, the idea was always to fix one machine and then show that if the other machine had such-and-such relation to the first machine it was the same as having some property. In this case it seems best to fix M' into a nice simple form. What is one possible form? Well, we want M' to have the fewest number of states of any machine accepting $domain(M')$. The fewest number of states any TM can have in the model given is one: a start state. One simple behavior is to just immediately halt. Then $domain(M') = \mathbf{N}$ and M' has the fewest number of states possible. So now maybe it's obvious: Testing an arbitrary M to see if $(M, M') \in L_{MIN}$ is really testing whether $M \in K_4 = \{M \mid domain(M) = \mathbf{N}\}$. Thus you see how you are "disguising" your question "$M \in K_4$?" as the question "$(M, M') \in L_{MIN}$?" and *fooling* L_{MIN} into answering this question for you. To make this formal, so you can really be sure you're right, you use this standard form. First you want to show

$$K_4 \leq_m L_{MIN}$$

You write out what this means: There exists some total recursive function f such that

$$x \in K_4 \text{ iff } f(x) \in L_{MIN}$$

This is pretty abstract so the next step is to keep rephrasing this statement until it makes sense:

$$M_1 \in K_4 \text{ iff } (M_2, M_3) \in L_{MIN}$$

where $f(M_1) = (M_2, M_3)$. This is just stating what x and $f(x)$ are. Now you say what it means to be in the language:

$$domain(M_1) = \mathbf{N} \text{ iff } domain(M_2) = domain(M_3)$$

and also remembering the condition that M_3 must have the fewest number of states of any machine accepting M_2's language.

Now you've got the statement of what you're trying to prove into an easily understood form, so you want to say what f does—how does it transform M_1 into M_2 and M_3? Well, you want M_3 to be the one-state machine that halts immediately on all inputs, so you can build that into f (which is just a TM that always halts, remember). You want to test if M_1 halts everywhere, so you just let $M_2 = M_1$. This is all easily computable. Now finally you need to make sure the "iff" holds:

If $domain(M_1) = \mathbf{N}$ *then* $domain(M_2) = domain(M_3)$ *and* M_3 *has the smallest number of states of any machine with* $domain(M_2)$. Since $domain(M_3) = \mathbf{N}$ and $M_2 = M_1$, certainly if $domain(M_1) = \mathbf{N}$ this holds.

It is best to state the other direction as the contrapositive: *If* $domain(M_1) \neq \mathbf{N}$ *then* $domain(M_2) \neq domain(M_3)$ *or* M_3 *doesn't have the smallest number of states of any machine halting on* $domain(M_2)$. Clearly this is true as well, since $domain(M_3) = \mathbf{N}$. So you have proven that $K_4 \leq_m L_{MIN}$. You now use the many-one theorem and since K_4 is not decidable, it follows that L_{MIN} is also undecidable.

Problem 3. Let $B = \{M \mid domain(M) = \emptyset \text{ or } domain(M) = \mathbf{N}\}$. Recall that $K_2 = \{M \mid M \text{ halts on input } 0\}$.

3(a). Show that $K_2 \leq_m B$.

You proceed to keep rewriting this until it makes sense: $K_2 \leq_m B$ means there exists some total recursive function f such that

$$x \in K_2 \text{ iff } f(x) \in B$$

$$M_1 \in K_2 \text{ iff } M_2 \in B$$

where $f(M_1) = M_2$.

$$M_1 \text{ halts on } 0 \text{ iff } domain(M_2) = \emptyset \text{ or } domain(M_2) = \mathbf{N}$$

So the problem is to design a TM M_2 based on M_1 so that M_2's language is $\emptyset$ or $\mathbf{N}$ exactly when M_1 halts on input 0. Thus you are disguising your question by hiding the "M_1 halts on input 0" inside M_2. Perhaps you recall a common trick is to have M_2 throw out its input and run M_1 on 0. This is the "fooling" part—the answerer thinks it's answering questions about M_2 on some input, but you're ignoring the input entirely and running M_1 on 0, all inside M_2. So you specify M_2 as follows: M_2 on any input erases it and runs M_1 on 0, halting iff M_1 halts. The function f to build M_2 out of M_1 is easily computable. Now you have to make sure this works—just stating things, even if they are true, is not enough. You do each part separately for clarity:

If M_1 halts on 0, then $domain(M_2) = \emptyset$ or $domain(M_2) = \mathbf{N}$. Clearly if M_1 halts on 0, then $domain(M_2) = \mathbf{N}$ since M_2 then halts on every input (since it ignores all inputs). So this is true.

If M_1 doesn't halt on input 0, then $domain(M_2) \neq \emptyset$ and $domain(M_2) \neq \mathbf{N}$. Oh no! Trouble. If M_1 doesn't halt on 0, then $L(M_2) = \emptyset$, so this doesn't work. You've found a bug!

What you need is a way to make sure this second part is true. Here's one easy fix: Have M_2 always halt on 0, and on any other input erase it and run M_1 on input 0. Now if M_1 halts on 0, $domain(M_2) = \mathbf{N}$ and the first part still works, but if M_1 doesn't halt on 0, then $domain(M_2) = \{0\}$ and thus $domain(M_2)$ is neither $\emptyset$ nor $\mathbf{N}$.

Many other ideas also work for M_2. The basic idea for this problem is to make sure that M_2 halts on at least one input when M_1 on 0 doesn't halt. You've got to make sure not to run M_1 on 0 in that case, for obvious reasons!

3(b). Show that $\overline{K_2} \leq_m B$.

This one is nearly identical, but now what you want is:

$$M_1 \text{ doesn't halt iff on input } 0, \mathit{domain}(M_2) = \emptyset \text{ or } \mathit{domain}(M_2) = \mathbf{N}$$

Here is an M_2 that works: M_2 diverges on 0, and erases the input and runs M_1 on 0 otherwise. You should go through the "iff" yourself to see how this works.

3(c). Conclude that B is neither r.e. nor co-r.e.

Both non-r.e.ness and non-co-re.ness inherit up $\leq_m$. Since $\overline{K_2}$ is not r.e. and $\overline{K_2} \leq_m B$, B is not r.e.. Similarly, since K_2 is not co-r.e. and $K_2 \leq_m B$, B is not co-r.e. Remember that what you're really doing here is applying the many-one theorem.

Problem 4. Let $K_{\text{infinite}} = \{M \mid \mathit{domain}(M) \text{ is infinite}\}$, the set of programs with infinite domain. Show that $K_{\text{infinite}} \equiv_{\text{m}} K_{\text{tot}}$, *i.e.*, $K_{\text{infinite}} \leq_{\text{m}} K_{\text{tot}}$ and $K_{\text{tot}} \leq_{\text{m}} K_{\text{infinite}}$.

This is an exercise for you.

6.044J/18.423J: Computability, Programming, and Logic Handout 19
Massachusetts Institute of Technology 30 October 1989

Problem Set 5 Solutions

Grades. The grades went as follows:

	number submitted	min	max	mean	median
1	12	5	25	19.6	25
2	12	25	50	40.9	43

Problem 1. Let S be the set of Turing machines (TM's) that halt on input 3 and run forever on input 4. Show that S is neither r.e. nor co-r.e.

Solution. To see that S is not co-r.e., we use Rice's theorem. The property

$$\mathcal{P} = \text{Contains the number 3 and doesn't contain 4?}$$

is a nontrivial property of r.e. sets, and $\mathcal{P}$ does not hold on the empty set. Thus, S is many-one hard for the r.e. sets; in particular, $K_0 \leq_{\mathrm{m}} S$. Since non-co-r.e.-ness inherits up, S is not co-r.e.

We cannot use Rice's theorem, however, to show that $\overline{S}$, the set of TM's that halt on input 4 or run forever on input 3, is not r.e., since the property

$$\mathcal{P}' = \text{Does not contain the number 3 or contains 4?}$$

does hold on the empty set. We need to use a direct reduction instead. Consider the set

$$A = \{M \mid M \text{ halts on } 4\}$$

which is not co-r.e. by Rice's theorem. To see that $A \leq_{\mathrm{m}} \overline{S}$, we use the reduction function $f : \mathbf{N} \rightarrow \mathbf{N}$, where $f(M)$ is a TM that behaves as follows:

> "On input n, check the input. If $n = 3$, halt; otherwise, run machine M on input 4, halting iff M halts."

It's not hard to see that f is total recursive. To see that it's a legal reduction, suppose $M \in A$. Then M halts on input 4, so $f(M)$ halts on input 4. Thus, $f(M) \in \overline{S}$. Conversely, suppose $M \notin A$. Then M does not halt on input 4, so $f(M)$ does not halt on input 4 and halts on input 3, and thus $f(M) \in S$—in other words, $f(M) \notin \overline{S}$. Thus, $A \leq_{\mathrm{m}} \overline{S}$, so $\overline{S}$ is not co-r.e. This just says that S is not r.e., which is what we wanted to show.

Note the form of the "converse" used in this argument. This idea was pointed out in Handout 18.

Problem 2. Classify the following r.e. sets according to whether they are recursive or nonrecursive. If the set is not recursive, state whether its nonrecursiveness follows from Rice's theorem. If the set is not recursive and this fact does not follow from Rice's theorem, prove that the set is not recursive.

2(a). The set of TM's that halt on input 3 and halt on input 4.

Solution. Not recursive by Rice's theorem.

2(b). The set of TM's that halt on input 5 in exactly 10^9 steps.

Solution. Recursive; to decide whether a machine M is in the set, run the machine for 10^9 steps on input 5. If M halts in exactly this number of steps, halt and output 1, otherwise halt and output 0.

2(c). The set of TM's each of whose domain is *not* a set of prime numbers.

Solution. Not recursive by Rice's theorem.

2(d). The set of TM's which, on input 0, run for exactly 2^i steps for some $i \geq 0$.

Solution. Not recursive; does not follow from Rice's theorem, since there are two TM's whose domains are the set $\{0\}$, but one of which is in the set and one of which isn't. This problem was harder than I anticipated; the grader was consequently lenient. To show that this set—call it B—is not co-r.e., we reduce K_2 to it. The reduction function $f : \mathbf{N} \rightarrow \mathbf{N}$, when given a TM M, returns a machine which works as follows:

> "On input n, simulate machine M on input 0. If the simulated TM halts, finish on an even power of 2 steps and halt."

The problem here is making sure that the *simulation* halts in a power of 2 steps. Some tricky programming is needed; one needs to simulate each step of M in some constant number of steps, and keep track of just how many simulated steps are taken. Using this information, the machine then performs some number of "dummy" moves to get it up to a power of 2. A rigorous argument here might take pages of programming.

At any rate, the reduction function can be shown to be total recursive. If M halts on input 0, then $f(M)$ halts in an even number of steps. Conversely, if M does not halt on 0, then $f(M)$ does not halt either. Thus, $K_2 \leq_{\mathrm{m}} B$, so B is not co-r.e.

2(e). The set of SCHEME S-expressions which *either* halt when applied to an infinite number of different integers, *or* there is a number k such that the expression does not halt when applied to any integer $\geq k$.

Solution. Recursive; it's a trivial property, though stated unclearly.

2(f). The set of SCHEME S-expressions P which compute a partial recursive function $f : \mathbf{N} \rightarrow \mathbf{N}$, such that *range*$(f)$ contains an integer $k \geq$ number of atoms in P.

Solution. Not recursive; does not follow from Rice's theorem, since it talks about the *ranges* of programs. To show that this set, call it C, is not r.e., we show that $K_2 \leq_{\mathrm{m}} C$. The reduction function $f : \mathbf{N} \rightarrow \mathbf{N}$, when given a TM M, returns a SCHEME expression that works as follows:

> "On input n, run machine M on input 0. If M halts, output an integer k larger than the number of atoms of this program."

This last step looks a bit fishy; but since k can be determined by looking at the size of the encoding for M, this function f is total recursive. Also, M halts on input 0 iff $f(M)$ outputs an integer $\geq$ the number of atoms in $f(M)$. Thus, $K_2 \leq_{\mathrm{m}} C$, so C is not co-r.e.

Quiz 2

Instructions. This exam is *closed book*. Do all problems in the provided white book, carefully labeling solutions with their corresponding numbers. Points are listed for each problem. You have one and a half hours. Good luck.

Problem 1. [25 points] A total function $f : \mathbf{N} \rightarrow \mathbf{N}$ is said to be *nondecreasing* if $n \leq m$ implies $f(n) \leq f(m)$. Show that the range of any nondecreasing total recursive function f is recursive. (Hint: Divide the problem into two cases—either $range(f)$ is finite or $range(f)$ is infinite.)

Problem 2. [25 points] Let $S = \{(M, M') \mid dom(M) = dom(M')\}$. Prove that S is neither r.e. nor co-r.e.

Problem 3. [25 points] Prove Rice's Theorem. Specifically, let P be a property of sets such that the empty set does not have property P, and there is an r.e. set R that has property P. Define $K_P = \{M \mid dom(M) \text{ has property } P\}$. Show that for any r.e. set A,

$$A \leq_{\mathrm{m}} K_P.$$

Problem 4. [25 points] Classify the following sets into one of four categories: decidable (D), r.e. but not co-r.e. (RE), co-r.e. but not r.e. (CO), or neither r.e. nor co-r.e. (NONE). No explanation or proof is necessary.

4(a). The set of arabic digits $0, 1, \ldots, 9$ that occur infinitely often among the base ten representations of elements of K_1.

4(b). The set of TM's that halt on input 9 and enter at least 637 different states during the computation.

4(c). The set of SCHEME S-expressions computing functions from $\mathbf{N} \rightarrow \mathbf{N}$ such that the domain of the function has a prime number of elements.

4(d). The set of TM's which, on input 4, run for more than some prime number of steps.

4(e). The set of TM's which, on every input n, run for at least n^2 steps.

6.044J/18.423J: Computability, Programming, and Logic — Handout 21
Massachusetts Institute of Technology — 6 November 1989

Quiz 2 Solutions

Statistics. A histogram of the scores:

Score	**Students**
100 - 91:	***
90 - 81:	*
80 - 71:	*
70 - 61:	*
60 - 51:	****
50 - 41:	*
40 - 31:	*
30 - 21:	*
20 - 11:	
10 - 1:	

The median score was 57; the mean was 63.8.

Problem 1. [25 points] A total function $f : \mathbf{N} \rightarrow \mathbf{N}$ is said to be *nondecreasing* if $n \leq m$ implies $f(n) \leq f(m)$. Show that the range of any nondecreasing total recursive function f is recursive. (Hint: Divide the problem into two cases—either $range(f)$ is finite or $range(f)$ is infinite.)

Solution. If $range(f)$ is finite, then it is trivially recursive (any finite set is recursive.) Now suppose $range(f)$ is infinite; here's a program that decides whether a number is in $range(f)$:

> "On input n, compute $f(0), f(1), f(2), \ldots$ until $f(i) \geq n$. If $f(i) = n$, output 1, else output 0."

A useful method for proving that a program meets some specification is to show that (1) if it terminates, it gets the correct answer, and (2) it always terminates. If the above procedure halts, we know it gets the right answer—if it answers 1 it is obviously correct, and if it answers 0 we know that the input n cannot be in the range since f is nondecreasing. To see that the program always halts, each computation $f(0), f(1), f(2), \ldots$ halts since f is total recursive. Also, since the range is infinite and the function is nondecreasing, we can never "get stuck" at an element in the range. Thus, the procedure will always find an i with $f(i) \geq n$, so the program always halts.

Problem 2. [25 points] Let $S = \{(M, M') \mid dom(M) = dom(M')\}$. Prove that S is neither r.e. nor co-r.e.

Solution. There are a number of correct ways to tackle this problem. The most straightforward method is to pick two sets, one that is not r.e. and one that is not co-r.e. and reduce these sets to S. A slightly easier method is to reduce a single set that is neither r.e. nor co-r.e. to S.

We'll take the latter approach. Recall that K_{tot} is the set of TM's that halt on all inputs, and that K_{tot} is neither r.e. nor co-r.e. Fix TM M_1 to be the one-state TM that simply halts. Define the reduction function f such that, given a TM M, $f(M) = (M, M_1)$. Clearly, f is total recursive. Also, if M halts on all inputs, then $domain(M) = domain(M_1)$; if M does not halt on all inputs, $domain(M) \neq domain(M_1)$. Thus, $M \in K_{\text{tot}}$ iff $f(M) \in S$, so $K_{\text{tot}} \leq_{\text{m}} S$. Since K_{tot} is neither r.e. nor co-r.e., S is neither r.e. nor co-r.e.

Problem 3. [25 points] Prove Rice's Theorem. Specifically, let P be a property of sets such that the empty set does not have property P, and there is an r.e. set R that has property P. Define $K_P = \{M \mid dom(M) \text{ has property } P\}$. Show that for any r.e. set A,

$$A \leq_{\text{m}} K_P.$$

Solution. See the computability notes (Handout 10) for the solution.

Problem 4. [25 points] Classify the following sets into one of four categories: decidable (D), r.e. but not co-r.e. (RE), co-r.e. but not r.e. (CO), or neither r.e. nor co-r.e. (NONE). No explanation or proof is necessary.

4(a). The set of arabic digits $0, 1, \ldots, 9$ that occur infinitely often among the base ten representations of elements of K_1.

Solution. (D), decidable. The set is finite; even though we may not know how to write the program, there exists a TM that decides membership in the set.

4(b). The set of TM's that halt on input 9 and enter at least 637 different states during the computation.

Solution. (RE), r.e. but not co-r.e. Call this set D_1; a Turing machine with domain D_1 works as follows:

> "Given any TM as input simulate the TM on input 9 and count the number of states used. If it halts and uses at least 637 states, halt."

The set is thus r.e. To see that the set is not co-r.e., consider the set

$$K_5 = \{M \mid M \text{ halts on input } 9\}$$

By Rice's Theorem, K_5 is not co-r.e. We want to show that $K_5 \leq_{\mathrm{m}} D_1$. Here, we use the reduction function $f : \mathbf{N} \rightarrow \mathbf{N}$, where $f(M)$ is M with 637 "dummy" states at the beginning; transitions out of a dummy state lead to the next dummy state without changing the tape contents or head position. Thus, M halts on 9 iff $f(M)$ halts on 9 and enters at least 637 different states, so $K_5 \leq_{\mathrm{m}} D_1$. Since non-co-r.e.-ness inherits up, D_1 is not co-r.e.

4(c). The set of SCHEME S-expressions computing functions from $\mathbf{N} \rightarrow \mathbf{N}$ such that the domain of the function has a prime number of elements.

Solution. (NONE), neither r.e. nor co-r.e. Call this set D_2; by Rice's theorem (where P is a SCHEME S-expression)

$$D_2 = \{P \mid \mathit{domain}(P) \text{ has a prime number of elements}\}$$

is not co-r.e. To see that D_2 is not r.e., we need to use a reduction. Let

$$A = \{P \mid 0 \notin \mathit{domain}(P)\};$$

A is not r.e., since $\overline{A}$ is not co-r.e. by Rice's Theorem. We claim that $A \leq_{\mathrm{m}} D_2$. The reduction is $f : \mathbf{N} \rightarrow \mathbf{N}$, where $f(P)$ behaves as follows:

> "On input n, halt if n is 0, 1, or 2. If $n = 3$, run P on input 0 and halt if P halts. If $n > 3$, diverge."

If $P \in A$, then $\mathit{domain}(f(P))$ has 3 elements and hence $f(P) \in D_2$. If $P \notin A$, then $\mathit{domain}(f(P))$ has 4 elements and so $f(P) \notin D_2$. Thus, $A \leq_{\mathrm{m}} D_2$, so D_2 is not r.e.

4(d). The set of TM's which, on input 4, run for more than some prime number of steps.

Solution. (D), decidable. A TM deciding this set just checks to see whether its input halts in not more than 2 steps on input 4. If it does, output 0, else output 1.

4(e). The set of TM's which, on every input n, run for at least n^2 steps.

Solution. (CO), co-r.e. but not r.e. Call this set D_3. A program with domain $\overline{D_3}$ works as follows:

> "Given any TM as input, dovetail the TM over all inputs. If it ever halts in less than n^2 steps on a particular input, halt."

To see that D_3 is not r.e., we reduce $\overline{K_2}$ to it. The reduction function f returns $f(M)$ which behaves as follows:

> "On input n, ignore n and run M on input 0. If M halts, halt."

If M does not halt on 0, then $f(M)$ never halts on any input and so is in D_3. If M halts on input 0 in, say, k steps, then $f(M)$ will halt in, say, $g(k)$ steps. Thus, we can pick an n large enough so that $f(M)$ runs in fewer than n^2 steps, so $f(M) \notin D_3$. Thus, $\overline{K_2} \leq_{\mathrm{m}} D_3$, so D_3 is not r.e.

6.044J/18.423J: Computability, Programming, and Logic — Handout 22
Massachusetts Institute of Technology — 6 November 1989

Problem Set 6

Due: 13 November 1989

Problem 1. Prove Rice's Theorem for the partial recursive functions. Specifically, suppose P is a property of partial recursive functions $\mathbf{N} \to \mathbf{N}$. Define

$$F_P = \{M \mid M \text{ computes } f : \mathbf{N} \to \mathbf{N} \text{ and } f \text{ has property } P\}$$

Suppose the totally undefined function does not have property P, and assume there is a partial recursive function $g : \mathbf{N} \to \mathbf{N}$ with property P. Prove that for any r.e. set A,

$$A \leq_{\mathrm{m}} F_P.$$

Problem 2. Consider any formulas F_1 and F_2, where $x_1, x_2, \ldots x_n$ are the free variables appearing in either F_1 or F_2. Show that $F_1 \simeq F_2$ iff

$$\vdash \forall x_1 \forall x_2 \ldots \forall x_n (F_1 \leftrightarrow F_2)$$

Problem 3. Determine whether each of the following sentences is valid. For those that are not valid, give an interpretation in which the sentence is not true.

3(a). $\forall x\ x + 0 = x$

3(b). $(\exists x \forall y\ P\ x\ y) \to (\forall y \exists x\ P\ x\ y)$

3(c). $(\forall x \exists y\ P\ x\ y) \to (\exists y \forall x\ P\ x\ y)$

3(d). $[\forall x(F_1 \to F_2) \land \forall x F_1] \to (\forall x F_2)$

3(e). $[\exists x(F_1 \to F_2) \land \exists x F_1] \to (\exists x F_2)$

3(f). $(\mathit{MinArith} \land \mathit{Commutativity}) \to \mathit{Cancellation}$,

where the sentence *MinArith* is the conjunction of the seven axioms for the theory Q given in Boolos & Jeffrey, page 107,

$$\begin{aligned} \mathit{Commutativity} &= (\forall x.\forall y.x + y = y + x)), \text{ and} \\ \mathit{Cancellation} &= (\forall x.\forall y.\forall z.(x + y = z + y) \to x = z) \end{aligned}$$

(Hint: Consider $\mathbf{N} \cup \{\infty\}$.)

Problem Set 7

Due: 17 November 1989

Problem 1. [25 points] Explain how to write a SCHEME program which, given any first-order sentence S and an interpretation over a finite domain for the constants, functions, and predicate symbols appearing in S, prints the truth value of S in the interpretation. Describe the input conventions for the interpretation. (Don't actually write the whole SCHEME program, just give enough explanation to make it clear you could.)

Problem 2. [25 points] Convert the following sentences into prenex normal form. Try to minimize the number of alternations of quantifiers in the resultant sentence.

2(a). $(\exists x\ (A\ x) \wedge \exists x\ (B\ x)) \rightarrow \exists x\ (C\ x)$

2(b). $\forall y\ \exists x\ (x < y) \rightarrow (\neg \exists z\ \forall y\ (z < y))$

Problem 3. [50 points] This problem will prove that the validity problem is undecidable for first-order sentences restricted to have a single ternary predicate symbol as their only nonlogical symbol. The proof follows from a series of sentence "simplifications" which you are to exhibit in subproblems below. You must define each simplification so that it ***preserves validity*** (*i.e.*, the old sentence is valid iff the simplified sentence is valid).

3(a). Let F be any atomic formula of the form $P_i(t_1, t_2)$ or $t_1 = t_2$, where t_1 and t_2 are terms containing only the function symbol $'$ (successor) and constant $\mathbf{0}$, and P_i is a binary predicate symbol for $i = 1, \ldots, m$. Explain how to write a formula F' equivalent to F so that F' is of the form

$$\exists x_1 \ldots \exists x_k\ (F_1 \wedge F_2 \wedge \ldots F_n)$$

where each F_i has the form $x = \mathbf{0}$, $x = y'$, or $P_i(x, y)$, with x, y variables.

3(b). Let S be a sentence whose atomic formulas have the form $x = \mathbf{0}$, $x = y'$, or $P(x, y)$. Explain how to "simplify" S into a sentence S' with only binary predicate symbols and no function symbols. (Hint: Write a sentence that states that a binary predicate $G(x, y)$ is the graph of a total function of x. Let S' be the conjunction of this sentence and a simplified form of S in which subformulas $x = y'$ are replaced by $G(x, y)$. Note that S' is *not equivalent to* S, but it is the case that either both are valid or neither is valid.)

3(c). One can simplify any sentence S into a sentence S' with the same non-logical predicate and function symbols but no names; the trick is to replace distinct names by fresh variables and then existentially quantifying over the new variables. For instance, the sentence

$$\forall x\ (x = a \vee x = b)$$

becomes

$$\exists y\ \exists z\ \forall x\ (x = y \vee x = z).$$

Explain how to obtain a model of S' from any model of ***any*** sentence S, and vice-versa. Conclude that S is valid iff S' is, and also that S is satisfiable iff S' is.

3(d). Let S be any sentence whose only nonlogical symbols are binary predicate symbols P_i. Show how to simplify S into a sentence S' whose only nonlogical symbols are names and a single ternary predicate symbol T.

3(e). Conclude that the validity of sentences whose only nonlogical symbol is a single ternary predicate symbol is undecidable. (Hint: In Boolos & Jeffrey, Chapter 10, only binary predicate symbols and the unary function symbol $'$ and the name $\mathbf{0}$ appear in the formulas corresponding to a Turing machine.)

Problem Set 6 Solutions

Problem 1. Prove Rice's Theorem for the partial recursive functions. Specifically, suppose P is a property of partial recursive functions $\mathbf{N} \to \mathbf{N}$. Define

$$F_P = \{M \mid M \text{ computes } f : \mathbf{N} \to \mathbf{N} \text{ and } f \text{ has property } P\}$$

Suppose the totally undefined function does not have property P, and assume there is a partial recursive function $g : \mathbf{N} \to \mathbf{N}$ with property P. Prove that for any r.e. set A,

$$A \leq_{\mathrm{m}} F_P.$$

Solution. There is very little difference between the proof of this theorem and the proof of Rice's theorem for sets. This may be somewhat surprising, since the theorems seem to say quite different things.

Here's the whole proof, written out once again. Let Turing machine M_g compute the partial recursive function g which, by hypothesis, has property P. Let A be any r.e. set, with $A = \text{domain}(M_A)$ for some Turing machine M_A.

For any $n \in \mathbf{N}$, we can define a new Turing machine $M_{f(n)}$ (that is, $f(n)$ is the code of this Turing machine) that operates as follows:

> "On input k, save k and simulate M_A on input n. If this simulation halts, then act exactly like M_g on input k."

Let $h : \mathbf{N} \to \mathbf{N}$ be the partial recursive function computed by $M_{f(n)}$. By definition of $M_{f(n)}$,

$$\begin{aligned}&M_A \text{ halts on } n\\ &\Rightarrow M_{f(n)} \text{ acts like } M_g \text{ on every input}\\ &\Rightarrow h = g \quad \Rightarrow M_{f(n)} \in F_P\end{aligned}$$

and conversely,

$$\begin{aligned}&M_A \text{ does not halt on } n\\ &\Rightarrow h \text{ is the totally undefined function}\\ &\Rightarrow M_{f(n)} \notin K_P.\end{aligned}$$

So $n \in A$ iff M_A halts on n iff $M_{f(n)} \in K_P$. Moreover, the function f is total recursive. Hence, $A \leq_{\mathrm{m}} F_P$.

Problem 2. Consider any formulas F_1 and F_2, where $x_1, x_2, \ldots x_n$ are the free variables appearing in either F_1 or F_2. Show that $F_1 \simeq F_2$ iff

$$\vdash \forall x_1 \forall x_2 \ldots \forall x_n (F_1 \leftrightarrow F_2)$$

Solution. We were trying to get you to work through some of the basic definitions (*e.g.*, of when two formulas are $\simeq$) in this problem. It's important to see that the statement is not obvious—given the way we've defined things—although if the statement weren't true, there would be something wrong with our definitions.

Here's a proof using a chain of iff's, something like what we've seen in class. Let $a_1, \ldots a_n$ be fresh, distinct names, and let F_1^* be F_1 with free occurrences of x_i replaced by a_i; define F_2^* in the same manner. Furthermore, let $\mathcal{I}$ be any interpretation for the nonlogical symbols occurring in F_1 and F_2. Then

$\vdash \forall x_1 \forall x_2 \ldots \forall x_n (F_1 \leftrightarrow F_2)$

iff $\mathcal{I}(\forall x_1 \forall x_2 \ldots \forall x_n (F_1 \leftrightarrow F_2)) = 1$ (by definition of validity)

iff $\mathcal{I}^{a_1 \cdots a_n}_{o_1 \ldots o_n}(F_1^* \leftrightarrow F_2^*)) = 1$ (by the definition of satisfaction)

iff $\mathcal{I}^{a_1 \cdots a_n}_{o_1 \ldots o_n}(F_1^*) = \mathcal{I}^{a_1 \cdots a_n}_{o_1 \ldots o_n}(F_2^*)$ (by the definition of satisfaction)

Since $\mathcal{I}$ was arbitrary, $\mathcal{I}^{a_1 \cdots a_n}_{o_1 \ldots o_n}$ is in fact an arbitrary interpretation giving meaning to the constants in F_1^* and F_2^*. Thus, the last line holds iff $F_1 \simeq F_2$, completng the proof.

Problem 3. Determine whether each of the following sentences is valid. For those that are not valid, give an interpretation in which the sentence is not true.

3(a). $\forall x \; x + 0 = x$

Solution. Not valid. For example, let $\mathcal{I}$ be the interpretation with domain **N** which assigns **+** the usual addition function, but assigns **0** the element 1. Then the sentence above is not true in $\mathcal{I}$, so the sentence cannot be valid.

3(b). $(\exists x \forall y \; P \; x \; y) \rightarrow (\forall y \exists x \; P \; x \; y)$

Solution. Valid.

3(c). $(\forall x \exists y\ P\ x\ y) \rightarrow (\exists y \forall x\ P\ x\ y)$

Solution. Not valid. For instance, let $\mathcal{I}$ be the interpretation with domain the integers (both positive and negative), where **P** is given the meaning "less than" in this domain. Then the hypothesis of the implication says, "For any x, there is a y less than x." This is true in the interpretation. However, the conclusion says, "There is a y such that for any x, y is less than x." The conclusion is false, since there is no "minimal element" in the domain. Thus, the sentence is not valid.

3(d). $[\forall x(F_1 \rightarrow F_2) \wedge \forall x F_1] \rightarrow (\forall x F_2)$

Solution. Valid.

3(e). $[\exists x(F_1 \rightarrow F_2) \wedge \exists x F_1] \rightarrow (\exists x F_2)$

Solution. Not valid. Suppose, for example, F_2 is the sentence letter A and F_1 is the atomic formula $(P\ x)$. Let $\mathcal{I}$ be an interpretation with domain $\{d_1, d_2\}$; further set $\mathcal{I}$ so that it assigns false to A and makes P true only on d_1. Then $\mathcal{I}$ satisfies

$$\exists x((P\ x) \rightarrow A) \wedge \exists x(P\ x)$$

but does not satisfy $\exists x A$.

3(f). (*MinArith* $\wedge$ *Commutativity*) $\rightarrow$ *Cancellation*,

where the sentence *MinArith* is the conjunction of the seven axioms for the theory Q given in Boolos & Jeffrey, page 107,

$$\begin{aligned} \textit{Commutativity} &= (\forall x.\forall y.x + y = y + x)), \text{ and} \\ \textit{Cancellation} &= (\forall x.\forall y.\forall z.(x + y = z + y) \rightarrow x = z) \end{aligned}$$

(Hint: Consider $\mathbf{N} \cup \{\infty\}$.)

Solution. Not valid. Some people did not understand the hint; "∞" is meant to represent a ***single*** element. Let $\mathcal{I}$ be the interpretation with domain $\mathbf{N} \cup \{\infty\}$. In this interpretation, $\mathcal{I}(\mathbf{0}) = 0$, and the addition and successor constants are given the usual meaning when the arguments are elements of $\mathbf{N}$. To define their behavior elsewhere, we use the following equations (being slightly pedantic but careful):

$$\begin{aligned} \mathcal{I}(')(\infty) &= \infty \\ \mathcal{I}(+)(n, \infty) &= \infty \\ \mathcal{I}(+)(\infty, n) &= \infty \end{aligned}$$

One can check that $\mathcal{I}$ is a model of $(\mathit{MinArith} \wedge \mathit{Commutativity})$; it is *not* a model of *Cancellation*, though, since

$$\mathcal{I}(+)(3, \infty) = \mathcal{I}(+)(4, \infty)$$

but 3 is not equal to 4 in $\mathcal{I}$.

6.044J/18.423J: Computability, Programming, and Logic — Handout 25
Massachusetts Institute of Technology — 20 November 1989

Problem Set 7 Solutions

Problem 1. [25 points] Explain how to write a SCHEME program which, given any first-order sentence S and an interpretation over a finite domain for the constants, functions, and predicate symbols appearing in S, prints the truth value of S in the interpretation. Describe the input conventions for the interpretation. (Don't actually write the whole SCHEME program, just give enough explanation to make it clear as you could.)

Solution. One first needs to pick some suitable representation of a finite domain and the assignments it makes to names, function symbols, and predicate symbols. Suppose S contains the names $a_1, \ldots, a_k$, the function symbols $f_1, \ldots, f_l$, and predicate symbols $P_1, \ldots, P_m$, all written in ASCII. To represent an n element domain, we'll use the numbers $1, \ldots, n$ (these are just symbols representing arbitrary elements of the domain.) To represent the assignments of the names to elements of the domain, create a list of pairs

$$((a_1\ o_1)\ (a_2\ o_2) \ldots (a_k\ o_k))$$

where each o_i is an number in the range $1, \ldots, n$. To represent the assignments of function symbols (say, of arity j), create a list of the form

$$(((1\ 1 \ldots 1\ 1)\ o_1)\ ((1\ 1 \ldots 1\ 2)\ o_2) \ldots ((n\ n \ldots n\ n) o_{j^n}))$$

where each j-tuple using integers $1, \ldots, n$ appears in this list as an "input", and where each o_i is again an integer in the range $1, \ldots, n$ (the "output.") Finally, assign to each predicate symbol (say, of arity j) a list consisting of lists

$$(o_1\ o_2 \ldots o_j)$$

An interpretation will thus be given by a list containing 4 elements:

1. The size of the domain, say n;
2. The list for the interpreting the names;
3. The list for interpreting the function symbols;
4. The list for interpreting the predicate symbols.

To determine whether a sentence is true in one of these interpretations, one proceeds by calling a procedure `is-true?`, passing the sentence and an interpretation in the above form. The procedure `is-true?` works recursively. For example, if the sentence is $\forall x F$, the procedure calls itself recursively substituting each element of the domain into the sentence for the variable x (maybe by substituting each number $1, \ldots, n$ into the sentence. This would avoid the problems of obtaining new fresh names, but we'd have to make sure the procedure can tell the difference between elements in the domain and names.) If all are true, the procedure returns true. For atomic formulas $P(t_1, \ldots, t_i)$ (the base case of the recursion), the procedure will calculate the value of each term by using the tabled values of the names and function symbols, and then looking up this value in the table of P. If the value is there, we return true. Finally, for atomic formulas of the form $t_1 = t_2$, we calculate the value of each term and see if they are equivalent.

There are, of course, many ways to write the actual code, but this gives the flavor of such a program.

Problem 2. [25 points] Convert the following sentences into prenex normal form. Try to minimize the number of alternations of quantifiers in the resultant sentence.

2(a). $(\exists x\ (A\ x) \wedge \exists x\ (B\ x)) \rightarrow \exists x\ (C\ x)$

Solution. Here's a possible answer (one of many):

$$\forall x \forall y \exists z\ ((A\ x) \wedge (B\ y)) \rightarrow (C\ z)$$

2(b). $\forall y\ \exists x\ (x < y) \rightarrow (\neg \exists z\ \forall y\ (z < y))$

Solution. Again, here's one possible answer:

$$\exists y \forall x \forall z \exists w\ \neg((x < y) \wedge (z < w))$$

Problem 3. [50 points] This problem will prove that the validity problem is undecidable for first-order sentences restricted to have a single ternary predicate symbol as their only nonlogical symbol. The proof follows from a series of sentence "simplifications" which you are to exhibit in subproblems below. You must define each simplification so that it *preserves validity* (*i.e.*, the old sentence is valid iff the simplified sentence is valid).

3(a). Let F be any atomic formula of the form $P_i(t_1, t_2)$ or $t_1 = t_2$, where t_1 and t_2 are terms containing only the function symbol $'$ (successor) and constant $\mathbf{0}$, and P_i is a binary predicate symbol for $i = 1, \ldots, m$. Explain how to write a formula F' equivalent to F so that F' is of the form

$$\exists x_1 \ldots \exists x_k \, (F_1 \wedge F_2 \wedge \ldots F_n)$$

where each F_i has the form $x = \mathbf{0}$, $x = y'$, or $P_i(x, y)$, with x, y variables.

Solution. The basic idea here is to ***unwind*** a number of applications of the function symbol $'$ into single applications. By the way we have described the problem, t_1 and t_2 must be $'$ applied to $\mathbf{0}$ some number of times. Suppose, without loss of generality, t_1 has l applications of $'$ to $\mathbf{0}$, t_2 has m applications of $'$ to $\mathbf{0}$, and $l \geq m$. Let F_1 be the formula $x_1 = \mathbf{0}$, F_2 be $x_2 = x_1'$, F_3 be $x_3 = x_2'$, and so forth up to F_{l+1} which we set to be $x_{l+1} = x_l'$. Also, let let F_{l+2} be $x_{l+2} = x_{m+1}$. Finally, let F_{l+3} be either $P_i(x_{l+1}, x_{l+2})$ or $x_{l+1} = x_{l+2}$ depending on the form of F. Then existentially quantifying over all the variables gives the sentence above.

Note that this can easily be modified to work if t_1 or t_2 contain free variables, although you weren't asked to do this.

3(b). Let S be a sentence whose atomic formulas have the form $x = \mathbf{0}$, $x = y'$, or $P(x, y)$. Explain how to "simplify" S into a sentence S' with only binary predicate symbols and no function symbols. (Hint: Write a sentence that states that a binary predicate $G(x, y)$ is the graph of a total function of x. Let S' be the conjunction of this sentence and a simplified form of S in which subformulas $x = y'$ are replaced by $G(x, y)$. Note that S' is ***not equivalent to*** S, but it is the case that either both are valid or neither is valid.)

Solution. Define the sentence S_1 to be

$$\forall x \exists y \forall z \; G(x, y) \wedge ((z \neq y) \rightarrow \neg G(x, z))$$

Informally, S_1 says that every x is related to some y (in other words, for every input there is an output), and that everything other than y is ***not*** an output. In other words, S_1 says that the relation $G(x, y)$ is a function. (Recall that functions are just special kinds of relations.)

Now we use the hint: replace all atomic formulas $x = y'$ in S with $G(x, y)$; the result is still a sentence. Call the result S_2; then S' is $S_1 \wedge S_2$.

3(c). One can simplify any sentence S into a sentence S' with the same nonlogical predicate and function symbols but no names; the trick is to replace distinct names by fresh variables and then existentially quantifying over the new variables. For instance, the sentence

$$\forall x\,(x = a \vee x = b)$$

becomes

$$\exists y\, \exists z\, \forall x\,(x = y \vee x = z).$$

Explain how to obtain a model of S' from any model of ***any*** sentence S, and vice-versa. Conclude that S is valid iff S' is, and also that S is satisfiable iff S' is.

Solution. The idea here is fairly simple—any model of S will automatically be a model of S'! More precisely, suppose $\mathcal{I}$ is a model of S, where S has constants $a_1, \ldots, a_n$. Let S' be defined as above, *i.e.*, S' has the form

$$\exists x_1 \ldots \exists x_n S^{\dagger}$$

where $S^{\dagger}$ is S with a_i replaced by x_i. Now let $b_1, \ldots, b_n$ be fresh, distinct names, and let o_i be the value in the domain of $\mathcal{I}$ whose meaning is assigned to a_i. Then $\mathcal{I}^{b_1 \ldots b_n}_{o_1 \ldots o_n}(S^*) = 1$, where S^* is $S^{\dagger}$ with x_i replaced by b_i, since $\mathcal{I}$ is a model of S. Thus, any model of S is a model of S', and so S is valid (or satisfiable) iff S' is.

3(d). Let S be any sentence whose only nonlogical symbols are binary predicate symbols P_i. Show how to simplify S into a sentence S' whose only nonlogical symbols are names and a single ternary predicate symbol T.

Solution. Suppose the sentence S contains binary predicate symbols $P_1, \ldots, P_n$. Let $a_1, \ldots, a_n$ be fresh distinct names. Let S' be S with every occurrence of $P_i(t_1, t_2)$ replaced by $T(t_1, t_2, a_i)$. Then S is valid iff S' is.

3(e). Conclude that the validity of sentences whose only nonlogical symbol is a single ternary predicate symbol is undecidable. (Hint: In Boolos & Jeffrey, Chapter 10, only binary predicate symbols and the unary function symbol $'$ and the name **0** appear in the formulas corresponding to a Turing machine.)

Solution. We merely put the parts together. Take any sentence S gotten from the $\leq_{\mathrm{m}}$-reduction of K_2 into the set of valid sentences. Note that S contains only binary predicate symbols, the unary function symbol $'$, and the name **0**. By part (a), we can eliminate all nested applications of $'$ to obtain a formula S' with atomic formulas of the form $x = 0$, $x = y'$, or $P_i(x, y)$. By part (b), we can replace each atomic formula $x = y'$ of S' by $G(x, y)$, and taking the conjunction of this sentence with S_1. Call the result S''. By part (d), we can eliminate turn all occurrences of binary predicates in S'' into one ternary predicate; call the resultant sentence S'''. Finally, by part (c), we can eliminate all constants. Thus, we obtain a sentence S'''' which is valid iff S is; note that the translation is effective, too.

Therefore, K_2 many-one reduces to the problem of determining whether a sentence with a single ternary predicate symbol is valid. Since K_2 is undecidable, so is this problem.

Quiz 3

Instructions. This exam is *closed book.* Do all problems in the provided white book, carefully labeling solutions with their corresponding numbers. Points are listed for each problem. You have one and a half hours. Good luck.

Problem 1. [40 points] Write first-order sentences for each part below.

1(a). A prenex normal form of

$$(\exists x \forall z(P\ x\ \wedge\ Q\ z)) \rightarrow (\forall z(f(z) = z))$$

1(b). A sentence with no nonlogical symbols whose models are precisely those interpretations whose domains have at most three elements. (Remember that "=" is allowed since it is considered to be a logical symbol.)

1(c). A sentence, whose only nonlogical symbol is a binary predicate, R, which is true in exactly those interpretations in which R is an equivalence relation.

1(d). A satisifiable sentence, whose only nonlogical symbol is a binary predicate, R, which is not true in any finite model. (Hint: Try saying that R is the graph of the successor function on **N**; alternatively, try saying R is a strict order with no least element.)

Problem 2. [30 points] For each of the sentences below, state whether the sentence is valid or not valid. If the sentence is not valid, give an interpretation in which the sentence is not true. (You do not need to prove that a sentence is valid if you think it is.)

2(a). $(\forall x \exists y(P\ x\ y)) \rightarrow (\forall x(P\ x\ f(x)))$

2(b). $(\forall x(P\ x\ f(x))) \rightarrow (\forall x \exists y(P\ x\ y))$

2(c). $(\forall x \forall z \exists y(P\ x\ y\ z)) \rightarrow (\forall u \exists v \forall w(P\ u\ w\ v))$

Problem 3. [30 points] (Refutation proofs)

3(a). Prove that the following sentence is valid using a refutation proof:

$$(\forall x(P\ x\ \wedge\ Q\ x)) \wedge (\exists x\ \neg(P\ x)) \rightarrow (\exists x(Q\ x))$$

3(b). Give a canonical derivation using the two sentences $\exists y \forall x(P\ x\ y)$ and $\forall x(P\ x\ x)$ as your "Δ set." Conclude *from the form of your derivation* that the conjunction of the two sentences must be satisfiable.

Problem Set 8

Due: 4 December 1989

Problem 1. Let Γ be a set of quantifier-free sentences which contain no function symbols and do not contain "=". Give a simplified proof from scratch that if every finite subset of Γ is satisfiable, *i.e.*, Γ is o.k., then Γ is satisfiable. (Hint: Rework and simplify the proof of Lemma 3 from the proof of the Completeness Theorem; the proof here should be easier, since the only terms to consider are names and there are no equality constraints.)

Problem 2. We say that a first-order sentence S is a $\forall$-*sentence* if it has the form

$$\forall x_1 \ldots \forall x_n F$$

where F is a quantifier-free formula (note that F may contain function symbols in addition to predicates and names.) Show that the set

$$\{S \mid S \text{ is a valid } \forall\text{-sentence}\}$$

is decidable. (Hint: Show that a canonical refutation from the negation of any $\forall$-sentence is finite.)

Problem 3. Let S be a sentence. Prove that if $\Gamma \vdash S$, then there is a finite subset $\Gamma' \subseteq \Gamma$ such that $\Gamma' \vdash S$. (Hint: Compactness.)

6.044J/18.423J: Computability, Programming, and Logic — Handout 28
Massachusetts Institute of Technology — 29 November 1989

Quiz 3 Solutions

Statistics. A histogram of the scores:

Score	**Students**
100 - 91:	***
90 - 81:	****
80 - 71:	**
70 - 61:	
60 - 51:	*
50 - 41:	
40 - 31:	
30 - 21:	
20 - 11:	
10 - 1:	

The median score was 87; the mean was 83.7.

Problem 1. [40 points] Write first-order sentences for each part below.

1(a). A prenex normal form of

$$(\exists x \forall z (P\ x\ \wedge\ Q\ z)) \rightarrow (\forall z (f(z) = z))$$

Solution. Here is one possible solution, where the number of alternations of quantifiers has been minimized:

$$\forall x \forall y \exists z (P\ x\ \wedge\ Q\ z) \rightarrow (f(y) = y)$$

1(b). A sentence with no nonlogical symbols whose models are precisely those interpretations whose domains have at most three elements. (Remember that "=" is allowed since it is considered to be a logical symbol.)

Solution. One possible answer is the sentence

$$\exists x \exists y \exists z \forall w\ (w = x) \vee (w = y) \vee (w = z)$$

1(c). A sentence, whose only nonlogical symbol is a binary predicate, R, which is true in exactly those interpretations in which R is an equivalence relation.

Solution. Recall that R is an equivalence relation if it is reflexive, symmetric, and transitive. The first-order formalization of these properties is

$$[\forall x(R\ x\ x)] \land [\forall x \forall y(R\ x\ y) \to (R\ y\ x)] \land [\forall x \forall y \forall z(R\ x\ y) \land (R\ y\ z) \to (R\ x\ z)]$$

1(d). A satisifiable sentence, whose only nonlogical symbol is a binary predicate, R, which is not true in any finite model. (Hint: Try saying that R is the graph of the successor function on **N**; alternatively, try saying R is a strict order with no least element.)

Solution. Let's try working with the first hint. Recall (from lecture) that we get a copy of the natural numbers if the successor function is required to be one-to-one and have an element (namely 0) which is not the successor of any element. We therefore want the conjunction of the sentences

- R is a functional relation: $\forall x \forall y \forall z(R\ x\ y) \land (R\ x\ z) \to (y = z)$
- R is total: $\forall x \exists y(R\ x\ y)$
- R is one-to-one: $\forall x \forall y \forall z(R\ x\ z) \land (R\ y\ z) \to (x = y)$
- There is an element d such that for any e, R does not relate (e, d): $\exists x \forall y \neg(R\ y\ x)$

In point of fact, the first sentence above is not necessary.

Using the other hint can work just as well. To say that R is a strict order with no least element is to say that

- R is transitive: $\forall x \forall y \forall z(R\ x\ y) \land (R\ y\ z) \to (R\ x\ z)$
- R is strict: $\forall x \forall y(R\ x\ y) \to \neg(R\ y\ x)$
- Every element has an element R-below it: $\forall x \exists y(R\ y\ x)$

Transitivity is essential here—without it, there are models with three elements!

Problem 2. [30 points] For each of the sentences below, state whether the sentence is valid or not valid. If the sentence is not valid, give an interpretation in which the sentence is not true. (You do not need to prove that a sentence is valid if you think it is.)

2(a). $(\forall x \exists y (P\ x\ y)) \rightarrow (\forall x (P\ x\ f(x)))$

Solution. Not valid. Consider the interpretation $\mathcal{I}$ whose domain has elements $\{d, e\}$, where P is true precisely on (d, e) and (e, e), and where f always returns d as its output. Then the hypothesis is true in this interpretation, but the consequent is false. Thus, the sentence is not valid.

2(b). $(\forall x (P\ x\ f(x))) \rightarrow (\forall x \exists y (P\ x\ y))$

Solution. Valid.

2(c). $(\forall x \forall z \exists y (P\ x\ y\ z)) \rightarrow (\forall u \exists v \forall w (P\ u\ w\ v))$

Solution. Not valid. There are many ways to pick an interpretation in which the above sentence is not true. For example, consider the interpretation $\mathcal{J}$ with domain $\mathbf{N}$, and where P is true of (k, m, n) iff $k+n = m$. Then the antecedent is true in the interpretation—the sum of two natural numbers is always a natural number. However, the consequent is not true; for example, there is no natural number m with $6 + m = 2$.

Problem 3. [30 points] (Refutation proofs)

3(a). Prove that the following sentence is valid using a refutation proof:

$$(\forall x (P\ x\ \wedge\ Q\ x)) \wedge (\exists x\ \neg(P\ x)) \rightarrow (\exists x (Q\ x))$$

Solution. Consider the set $\Delta = \{(\forall x (P\ x\ \wedge\ Q\ x)), (\exists x\ \neg(P\ x)), (\forall x \neg(Q\ x))\}$. This set is unsatisfiable iff the original sentence is valid. A refutation from this set:

1.	$\exists x\ \neg(P\ x)$	Δ
2.	$\neg(P\ a)$	1
3.	$\forall x (P\ x\ \wedge\ Q\ x)$	Δ
4.	$P\ a\ \wedge\ Q\ a$	3

Lines 2 and 4 are unsatisfiable, so the set Δ is unsatisfiable.

Note that we didn't need the third element in the set Δ in the refutation. If the first part of the sentence had read $(\forall x (P\ x\ \vee\ Q\ x))$, though, we would have needed it. Note also that one can negate the whole sentence above, put the result into prenex form, and carry out a refutation from there.

3(b). Give a canonical derivation using the two sentences $\exists y \forall x (P\ x\ y)$ and $\forall x (P\ x\ x)$ as your "Δ set." Conclude *from the form of your derivation* that the conjunction of the two sentences must be satisfiable.

Solution. Here is a canonical derivation, using the procedure developed in class:

1.	$\exists y \forall x (P\ x\ y)$	Δ
2.	$\forall x (P\ x\ a)$	1
3.	$(P\ a\ a)$	2
4.	$\forall x (P\ x\ x)$	Δ
5.	$(P\ a\ a)$	4

and that's it! Since there are no function symbols, one can only use terms that are names. By starting off with the first sentence, we get precisely one name, and hence only one term, to use for universal instantiation.

Note that the derivation does not yield a contradictory set of quantifier-free sentences. Thus, Δ must be satisfiable—if it weren't, we'd have hit a contradictory set of quantifier-free sentences in the canonical derivation.

Notes on Programming (Part I)

The following notes will serve as the main text for the remainder of the course. The goal of this final unit will be to work with a language that in some sense "captures" the essential features of the programming language Scheme and other applicative functional languages. Our language, called FKS, we claim is the functional kernel of Scheme. The syntax FKS will be basically that of the simply-typed λ-calculus, with a single base type ι which we will take to be the natural numbers. One of the most important properties of FKS that makes it possible for us to analyze in this class is that it has **NO SIDE EFFECTS**.

We will present definitions of the language at several levels. Our first level will be that of rewrite rules. Rewrite rules, via an immediate reduction relation between pieces of code, specify how, at a high level, programs can be evaluated. It will take a program M and in one step reduce it to another program that in some sense will be closer to what we would like to call *the answer*. Although rewrite rules provide a wonderful way of defining a language, the way in which they work is very far from a reasonable implementation strategy. We will present an automaton, called a SECD machine, which will reduce the task of interpreting our language to that of basic, well-understood pointer manipulations. This SECD machine will be very close in spirit to the way in which functional languages are actually implemented. Bridging the gap between the SECD machine and the rewrite rules we will present *eval* a recursive characterization of the rewrite rules. All of these definitions with respect to a language $\mathcal{L}$ are referred to as the operational semantics of $\mathcal{L}$. In this case our language will be "FKS", and we will provide a definition of the semantics operationally, via rewrite rules.

We will also present a denotational semantics for FKS. The motivation behind this sort of semantics is the desire to say that the meaning of a piece of code that computes a certain function *is* the function which it computes. The goal of denotational semantics is to develop an interpretation (which we will from now on call a *model*) of all of the terms (fragments of code) in the language. Unlike first order logic, where an interpretation can be any first order structure, we will limit our consideration of semantic interpretations for FKS to a single model. In the field of denotational semantics, a model is specified by two entities: a domain (just like in logic), and a *meaning* function which maps syntactic elements of the language into the domain. This meaning function does a job analogous to that of the interpretation function operating on terms. In logic $\mathcal{I}(t)$ (referred to as the meaning or *denotation* of term t in interpretation $\mathcal{I}$) was an object in $D_{\mathcal{I}}$ that was implicitly defined by giving the denotations of the constant and function symbols of the language for which $\mathcal{I}$ is a model. In devising a model for FKS, however, we need to explicitly define this meaning function which when given a term yields that term's denotation. The most logical question to ask at this point is "what good is this model?" The answer is a lot. For example we will show that if a program M evaluates to a constant c then the model will assign the same object to both M and to c we will also see that the converse is true, namely that if M and c have the same denotation then M will evaluate to c. We will then show as a corollary to this that if M and N have the same denotation, then (not considering time or space issues) these two pieces of code are completely interchangeable—a very nice property to be able to state. Wouldn't it be nice if after you optimize a piece of code in an already working system you could then prove rigorously that the new code was functionally equivalent to the old code? This is what denotational semantics can do for you...

Given this cultural background it is now time to consider the task immediately at hand. In order to define FKS, we will first define its syntax. We will then define its semantics operationally by a set of rewrite rules. This combination of syntax and semantics will fully define the language FKS. We will then present alternate operational definitions for the semantics of a language with the same syntax (but not necessarily the same semantics) as FKS. We will then sketch a proof that the semantics of these alternate definitions coincide with that of the rewrite rules—thus they define the same language. Finally, we will provide a denotational definition of a language over that same syntax. We will then argue that the semantics defined denotationally coincides in a nice way with the operational semantics.

1 Syntax of FKS

What is a term. The basis of the syntax for FKS is what is called the simply typed λ-calculus. Fragments of code in this framework are referred to as terms. A term in this framework is analogous to the code that appears between a balances set of parenthesis in unsugared Scheme, or a constant symbol, or a non-binding occurrence of a variable.

Example 1. The following is a short fragment of Scheme code:

$$((\text{lambda }(x)\ (*\ x\ x))\ 5)$$

The terms in this program are: $5, x, (*\ x\ x), (\text{lambda }(x)\ (*\ x\ x))$, and $((\text{lambda }(x)\ (*\ x\ x))\ 5)$. Note: the x immediately after the lambda is not really a term. Scheme's notation is not optimal. Scheme should have been defined so that code would have been written something like:

$$((\text{lambda }x.(*\ x\ x))\ 5)$$

Types. Unlike Scheme every term in our language will all have an associated type. The set of types is called **Types**; we define it inductively as follows:

- ι is a type. It will correspond to our notion of the natural numbers.
- If σ is a type and τ is a type then $\sigma \rightarrow \tau$ is a type. It will correspond to functions which take a single argument of type σ and return a single result of type τ.

Example 2. When writing a type we will let $\rightarrow$ associate *right*. Thus $\sigma_1 \rightarrow (\sigma_2 \rightarrow \sigma_3)$ is the same type as $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3$, which is distinctly different from $(\sigma_1 \rightarrow \sigma_2) \rightarrow \sigma_3$.

The following are some basic functions, and their types.

1. SQUARE$\overset{\text{def}}{=}$ (lambda (x) $(* \ x \ x)$). SQUARE is of type $\iota \rightarrow \iota$, as it is a function that takes a natural number as an argument and returns a natural number as a result.
2. 5. 5 is of type ι.
3. APP5$\overset{\text{def}}{=}$ (lambda (foo) $(foo$ 5$)$). APP5 is of type $(\iota \rightarrow \iota) \rightarrow \iota$. It takes a a function from natural numbers to natural numbers, and then returns a natural number. For example (APP5 SQUARE) is 25.
4. PLUS1$\overset{\text{def}}{=}$ (lambda (foo) (lambda (x) $(+1$ $(foo \ x))))$. PLUS1 is of type $(\iota \rightarrow \iota) \rightarrow (\iota \rightarrow \iota)$. It takes a function from natural numbers to natural numbers and returns a function from natural numbers to natural numbers. For example (PLUS1 SQUARE) is a function that returns its argument squared, plus 1.

Note that this language does not contain any booleans, characters, reals, strings, lists or other such structures. We claim that this simple type hierarchy with only the base type ι is the core of Scheme.

Notice that our set of types does not include "pair" types. For example the binary plus operator which we are familiar with from arithmetic takes two arguments, both of type ι and returns one value, also of type ι. So we would write 2 plus 3 as (+ 2 3), and + has type $(\iota \times \iota \rightarrow)\iota$. In our framework we will not have pair types. But, via a process called "currying", we will show that there is no loss of generality in not having pair types. Before giving the formal definition of currying, we will provide as an example a definition of a curried version of plus $(+_c)$ defined in terms of the standard plus:

$$+_c \overset{\text{def}}{=} (\lambda x(\lambda y((+ \ x \ )y)))$$

So consider the two ways we would now write the expression for 2 plus 3:

$$\text{curried: } ((+_c\ 2)\ 3)$$

$$\text{uncurried: } (+\ 2\ 3)$$

Note that the type of $+_c$ is $\iota \rightarrow \iota \rightarrow \iota$. Thus $(+_c\ 2)$ is of type $\iota \rightarrow \iota$ and represents the "plus 2 function". In general, given an arbitrary function foo of type $\sigma = (\sigma_1 \times \ldots \times \sigma_n) \rightarrow \sigma'$ we can easily curry it to foo_c which works right. The function foo_c will have type $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma'$. It is defined from foo as follows:

$$foo_c \stackrel{\text{def}}{=} (\lambda x_1^{\sigma_1} \ldots \lambda x_n^{\sigma_n}(foo\ x_1^{\sigma_1} \ldots x_n^{\sigma_n}))$$

The last comment to be made is that every type σ is of the form $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma'$ this type will occasionally be abbreviated as $\sigma = (\sigma_1, \ldots, \sigma_n, \sigma')$.

Terms. Now that we know what types are, we are ready to define what terms are. Terms and their types are defined inductively as follows:

- x^σ is a term of type sigma. (representing a variable of type σ)
- c^σ is a term of type sigma. (representing a constant of type σ)
- (MN) is a term of type τ if M is a term of type $\sigma \rightarrow \tau$ and N is a term of type σ.
- (cond M N_1 N_2) is a term of type ι and M, N_1, and N_2 are all terms of type ι.
- $(\lambda x^\sigma.\ M)$ is a term of type $\sigma \rightarrow \tau$ if M is a term of type τ.

Note that we are very informal with parenthesis and as with arithmetic we will define certain conventions that allows most parenthesis to be eliminated. The conventions are as follows:

- Application associates left. Thus (MNP) is the same as $((MN)P)$, which is very different from $(M(NP))$.
- λ's bind out as far as they can (*e.g.*until the end of the expression or overridden by a parenthesis). So $(\lambda\ x.\ M\ N)$ is the same as $(\lambda\ x.\ (M\ N))$ which is distinctly different from $((\lambda\ x.\ M)N)$.
- Never drop the parenthesis that surrounds "cond M N_1 N_2 ."
  Thus (cond M N_1 N_2) is correct, and cond M N_1 N_2 will only lead to confusion.

We will call any term which a constant, variable or λ-abstraction a *value*. A term of the form (MN) is can be called either a *combination* or an *application*. A term of the form (cond ...) is called a *conditional.*

Aside from presenting the set of constants, and their affiliated types, this completely defines the syntax of FKS.

Unsugaring Scheme. So terms are either variables, constants, applications (something of the form (MN)), conditional expressions (something of the form (cond M N_1 N_2)), or λ-abstractions (something of the from $(\lambda x^\sigma.\ M)$. Applications can be viewed as function calls, conditional expressions can be viewed as our only special form, and λ-abstraction can be viewed as procedure construction. Remember from 6.001 that most of the friendly structures in Scheme are syntactic sugar for other forms.

Example 3. The most prominent case of syntactic sugar is "let". For example:

(let ((*var*1 *exp*1)) *exp*)

is syntactic sugar for

((lambda (*var*1) *exp*) *exp*1)

Example 4. Another example of syntactic sugar is "define" (when used to define a function that is not recursive). For example:

(define (*foo arg*) body)

is syntactic sugar for:

(define *foo* (lambda (*arg*) body))

Considering that FKS does not allow side-effects, and it does not have lists (or streams or other fancy stuff) this syntax really is almost as expressive as full fledged Scheme. Two missing elements of the syntax require further justification. These two problems are as follows: conditionals can only return objects of type ι, and the lack of "define". We will argue later on in Example 5 that there is a straightforward way to program a "higher-type" conditional from the one given here. On the surface we can argue away "define" by saying that since we have no side effects then the following:

$$\begin{array}{c}(\text{define } P_1\ B_1)\\(\text{define } P_2\ B_2)\\\vdots\\(\text{define } P_n\ B_n)\\\text{body}\end{array}$$

is equivalent to

$$(\text{let } ((P_1\ B_1)\ (P_2\ B_2) \ldots (P_n\ B_n))\ \text{body}$$

Since "let" is only sugar, "define" is only sugar.

We have, however, slipped something very important under the rug. In Example 4, the unsugaring of (define $(foo\ arg)$ body) assumed that foo was not recursive. We will introduce a special constant Y that will allow for the unsugaring of recursive procedures, and in Sections 2 and 2 we will show how Y can be used to generate recursive and mutually recursive functions.

Constants. The constants of FKS and their types are as follows:

- $0, 1, 2, \ldots$ all of type ι
- $succ, pred$ both of type $\iota \rightarrow \iota$
- Y_σ of type $(\sigma \rightarrow \sigma) \rightarrow \sigma)$ (for all types $\sigma \neq \iota$)

Free and Bound variables. In this section we will write the variable x^σ simply as x. When we do not need to discuss the type of a bound variable, it is sometimes convenient to drop the superscript. We must be careful, however, for example if we use x^ι and $x^{(\iota\rightarrow\iota)}$ as distinct variables, we need to be careful when we abbreviate one by x so as to prevent confusion.

We will define the function $FV : Terms \rightarrow \{Variables\}$ So, if M is a term, it has a set $FV(M)$ of free variables. It is defined inductively on the structure of the term M by:

- $FV(x) = \{x\}$
- $FV(c) = \{\}$
- $FV((MN)) = FV(M) \cup FV(N)$
- $FV(\text{cond } M\ N_1\ N_2\ ) = FV(M) \cup FV(N_1) \cup FV(N_2)$
- $FV((\lambda x\ M)) = FV(M) \backslash \{x\}$

A term is *closed* iff $FV(M) = \emptyset$, otherwise it is *open*.

A function $BV : Terms \rightarrow \{Variables\}$ can be defined analogously to give the bound variables of a term.

The free variables of a term are simply those variables that are not under the scope of a λ-abstraction over that variable. For example y is free in $(\lambda x.\ x\ y)$

since there is no λ binding it. The bound variables of a term are those variables that are bound by λ's. For example y is bound in both $(\lambda y.\ y)$ and $(\lambda y.\ 5)$. Note that it is possible for a variable to occur both free and bound in the same term. For example y is both free and bound in $(y(\lambda y.\ (x\ y)))$. The free occurrences of y in a term M are those occurrences of y in M that are not bound. The bound occurrences of y in a term M are partitioned (divided into disjoint sets that cover everything) into occurrences bound by an individual λ-abstraction.

A *program* is a closed term of ground type. Since our only ground type is ι, a *program* is a closed term of type ι.

Given an infinite list $x_1, \ldots$ of distinct variables (which we will assign types when we use them), the substitution prefix is defined inductively in the structure of terms as follows:

- $x[x := M] = M$; $y[x := M] = y$ $(if\ x \neq y)$
- $a[x := M] = a$
- $(NN')[x := M] = (N[x := M])\ (N'[x := M])$
- (cond N N_1 N_2)$[x := M]$ = (cond $N[x := M]$ $N_1[x := M]$ $N_2[x := M]$)
- $(\lambda xN)[x := M] = (\lambda xN)$; $(\lambda yN)[x := N] = \lambda zN[y := z][x := M]$, if $x \neq y$, where z is the variable defined by:
    1. If $x \notin FV(N)$ or $y \notin FV(M)$ then $z = y$.
    2. Otherwise, z is the first variable in the list $x_1, x_2, \ldots$ such that $z \notin FV(N) \cup FV(M)$—and z is made to have the same type as y.

We now introduce a relation $=_\alpha$in order to capture the notion that two terms are equivalent up to the renaming of bound variables. It is defined inductively in the structure of terms as follows: The relation $=_\alpha$of alpha equivalence, is defined inductively by:

1. $x =_\alpha x$
2. $a =_\alpha a$.
3. If $M =_\alpha M'$ and $N =_\alpha N'$ then $(MN) =_\alpha (M'N')$.
4. If $M =_\alpha M'$ and $N_1 =_\alpha N_1'$ and $N_2 =_\alpha N_2'$
then (cond M N_1 N_2)$=_\alpha$(cond M' N_1' N_2')
5. If $M =_\alpha M'[y := x]$, where either $x = y$ or $x \notin FV(M')$
then $(\lambda xM) =_\alpha (\lambda yM')$.

Contexts. A *context* is a term with one or more "holes" in it. It is normally written in the from $C[\cdot]$. The term that results from filling the term M into all of the holes in the context $C[\cdot]$ is written as $C[M]$. $C[\cdot]$ is called a program context (implicitly with respect to the term M iff $C[M]$ is a program.

This concludes the definition of the syntax of FKS.

2 Operational Semantics: Rewrite Rules

One way of providing an operational semantics for a language is via a set of rewrite rules. From the rewrite rules we will arrive at a partial function *Eval* from Programs (closed terms of type ι) to constants. *Eval* is defined by means of an immediate reduction relation, $\rightarrow$ between terms by:

$$Eval(M) = k \text{ iff } M \twoheadrightarrow k, \text{for any program } M \text{ and constant } k.$$

Where $\twoheadrightarrow$ is the reflexive transitive closure of $\rightarrow$ (sometimes written as $\overset{*}{\rightarrow}$).

It will be the case that $M \twoheadrightarrow k$ and $M \twoheadrightarrow k'$ implies that k and k' are identical. Notice that for constants c, $Eval(c) = c$.

The following rules together are called *rewrite rules* and together they define the desired immediate reduction relation $\rightarrow$.

1. (a) $(succ\ n) \rightarrow (n+1)$
   (b) $(pred\ 0) \rightarrow 0$
   (c) $(pred\ (n+1)) \rightarrow n$
   (d) $(Y_\sigma V) \rightarrow (V(\lambda x^\sigma (Y_\sigma V)x^\sigma))$ (where $x \notin FV(V)$)
2. (a) $(\lambda x V) \rightarrow M[x := V]$ (for V a value)
   (b) (cond 0 N_1 N_2) $\rightarrow N_1$
   (c) (cond $(n+1)$ N_1 N_2) $\rightarrow N_2$
3. (a) if $M \rightarrow M'$ then $(MN) \rightarrow (M'N)$
   (b) if $N \rightarrow N'$ then $(VN) \rightarrow (VN')$ (for V a value)
   (c) if $M \rightarrow M'$ then (cond M N_1 N_2) $\rightarrow$ (*cond* M' N_1 N_2)

That is all. We have just defined a language completely. We have given a definition of the syntax, and we have given an operational definition of what it means for a program P to evaluate to a constant c. This is operational in the sense that it is sort of how a compiler works, and it made no appeal to semantic domains or models.

Some Useful Definitions. A term M is said to *diverge* iff $mEval(M)$ is undefined. This is equivalent to saying that for all N if $M \twoheadrightarrow N$ then their is a term N' such that $N \rightarrow N'$.

We now introduce a notion of equality between terms that is generated by the rewrite rules. This notion is called observational equality ($\equiv_{obs}$)[1] and it captures the idea of the interchangeability of code. Two pieces of code are observationally equivalent if they can be exchanged freely in FKS programs with out changing the value to which the program evaluates. This is defined formally as follows:

$$N \equiv_{obs} M$$

iff for all program contexts

$$Eval(C[M]) \simeq Eval(C[N])$$

Two objects are $\simeq$ iff either they are both undefined, or they are both defined and equal. There is a "Context Lemma" for FKS that says that:

$$\text{if } M \twoheadrightarrow N \text{ then } Eval(C[M] \simeq Eval(C[N]))$$

Thus $M \twoheadrightarrow N$ implies $M \equiv_{obs} N$. This "Context Lemma" will be a Corollary of the Adequacy Theorem which we will prove later. [2]

Now, in order to fully appreciate the richness of this language we provide some examples of coding tasks.

Example 5. Before we can do much of anything, we really need a "macro" which will enable us to do a higher order conditional. Something we can abbreviate into the form ($cond_\sigma$ M N_1 N_2) where N_1 and N_2 both have type σ. In this side-effect free, typed world of ours, we have no need for a conditional where N_1 and N_2 have different types.

Supposing $\sigma = (\sigma_1, \ldots, \sigma_n, \sigma')$ $cond_\sigma$ is:

$$(cond_\sigma\ M\ N_1\ N_2) \stackrel{\text{def}}{=} (\lambda x_1^{\sigma_1} \cdots \lambda x_n^{\sigma_n}.\ (\text{cond } M(N_1 x_1 \cdots x_n)(N_2 x_1 \cdots x_n)))$$

It is a rather grungy, but manageable task to show that ($cond_\sigma$ 0 N_1 N_2) behaves just like (cond$_\sigma$ 0 N_1 N_2) would if it were in the language (except that if ($cond_\sigma$ 0 N_1 N_2) is computed, but never used, then if N_1 diverges a program using the cond$_\sigma$ would also diverge, but a program using the $cond_\sigma$ might still evaluate to a value).[3]

[1] It is important to note that $\equiv_{obs}$is an equivalence relation.

[2] One way of stating the Adequacy Theorem is: "If M and N have the same denotation, then $M \equiv_{obs} N$."

[3] If you did not understand that parenthetic remark it is ok, this is a watered down version of a concept called ***observational approximation*** which we will might cover later. For your information, a term M observationally approximates a term N iff for all program contexts $C[\cdot]$, $C[M] \twoheadrightarrow c$ implies $C[N] \twoheadrightarrow c$. In this setting we can say that M is observationally congruent to N iff M observationally approximates N and N observationally approximates M.

Example 6. FKS does not include constants or special forms which allow the pairing of objects (we do not have cons). This lack is easily overcome, since using λ-terms we can define combinators that act like typed pairing operators. We will define "macros" which will provide typed pairing operators for us. Thus $pair_\sigma$ pairs together two objects of type σ. The macros $left_\sigma$ and $right_\sigma$ will unpair the result of pairing together two objects of type σ. These three macros are defined as follows:

- $pair_\sigma(M, N) \overset{\text{def}}{=} (\lambda z^{(\sigma\to\sigma\to\sigma)}.\ M\ N)$. Where this object is of type: $(\sigma \to \sigma \to \sigma) \to \sigma)$.

- $left_\sigma(P) \overset{\text{def}}{=} (P(\lambda x^\sigma.\lambda y^\sigma.x))$

- $right_\sigma(P) \overset{\text{def}}{=} (P(\lambda x^\sigma.\lambda y^\sigma.y))$

It is a straightforward task to check the correctness of these definitions and that $left(pair(M, N)) = M$ and that $right(pair(M, N)) = N$.

It is fairly simple to generalize this technique to allow the pairing of terms which do not have the same type. This can be done many ways. One such way is to coerce the arguments to be of the same type. If M has type $\sigma_1 = (\sigma_1, \ldots, \sigma_n, \iota)$ and N has type $\tau = (\tau_1, \ldots, \tau_m, \iota)$ then we can coerce M into M' and N into N', M' and N' of the type $\sigma' = (\sigma_1, \ldots, \sigma_n, \tau_1, \ldots, \tau_m, \iota)$ So

$$M' \overset{\text{def}}{=} \lambda x_1^{\sigma_1} \ldots \lambda x_n^{\sigma_n} \lambda y_1^{\tau_1} \ldots \lambda y_m^{\tau_m} (M x_1^{\sigma_1} \ldots x_n^{\sigma_n})$$

and

$$N' \overset{\text{def}}{=} \lambda x_1^{\sigma_1} \ldots \lambda x_n^{\sigma_n} \lambda y_1^{\tau_1} \ldots \lambda y_m^{\tau_m} (M x_1^{\tau_1} \ldots y_m^{\tau_m})$$

It should be clear how to recover terms equivalent to the original M and N from M' and N' through appropriate abstraction and application to dummy terms.

From here on we will assume that we have a "smart" macro system which will do the appropriate coercions and such and we will assume we have a single macro for each of pair, left, and right that does what we would like it to do (pair will coerce its args to the right type, but left and right will not "uncoerce" their results).

Finally it should be clear how to make n-tuples. Call the operation $tuple_n^\sigma$ which makes an n-tuple out of objects of type *sigma*. It should be equally clear how to define the projection on the i^{th} component—$proj_{(n,i)}^\sigma$. These can be defined either directly (introducing new expressions using λ's for each) or they can be defined from *pair*, *left* and *right*.

Debunking Recursion: Fixed Points. An important element of any functional language is the ability to define a function recursively. Consider, for example the standard definition of the factorial function in Scheme:

$$\text{(define } fact\ (\lambda\ n)\ \text{(cond (= } n\ 0)\ 1\ (*\ n\ (fact\ (-1\ n)))))$$

Unfortunately the syntax of Scheme does not make it immediately obvious that *fact* is a recursive function (actually a "simply recursive"[4]). The syntax of Common Lisp, however, does make this fact evident. Consequently, we will draw the examples of defining recursion from Common Lisp instead of Scheme (it should be immediately obvious how to convert from one definition to the other). So *fact* would be defined in Common Lisp as follows:

$$\text{(letrec } fact = \text{(lambda } (n)\ \text{(cond (= } n\ 0)\ 1\ (*\ n\ (fact\ (-1\ n))))))$$

This syntax for defining a recursive function makes it immediately clear that *fact* is a simply recursive function. Now consider the form of a mutually recursive definition of functions $f_1, \ldots, f_n$ by bodies $b_1, \ldots, b_n$ where each body b_i has n "holes". We write $b_i[f_1, \ldots, f_n]$ to denote b_i with its hole #1 filled by f_1,. . .,#n filled by f_n. So the definitions of $f_1, \ldots, f_n$ would be as follows:

$$\begin{array}{lll}\text{(letrec} & & \\ & (f_1 = & b_1[f_1, \ldots, f_n]) \\ & \ldots & \\ & (f_n = & b_n[f_1, \ldots, f_n]) \\ & &) \end{array}$$

Now that we have a syntax that makes it clear what is being defined recursively, we can go ahead and explain how recursive functions can be defined in FKS. Let us consider the following function:

$$F \stackrel{\text{def}}{=} \text{(lambda } (f)\ \text{(lambda } (n)\ \text{(cond (= } n\ 0)\ 1\ (*\ n\ (f\ (-1\ n))))))$$

F has a very interesting property, namely $F(fact) = fact$. For no other argument x is $F(x) = x$. There is a special name for this property, namely, *fact* is a *fixed point* of F. In mathematics, if you have any function G and object x, x is called a fixed point of G iff $G(x) = x$. Let fix be a "fixed point operator", namely a function which returns a fixed point of its argument. Then *fact* could be defined as $fact \stackrel{\text{def}}{=} (fix\ F)$. Thus any recursive definition of the form:

$$\text{(letrec } foo\ =\ \text{body}[foo])$$

[4] A function *foo* is called ***simply recursive*** iff it is recursive and its definition does not use another function whose definition depends (either directly or indirectly) on itself. This is to be contrasted with a ***mutually recursive*** function (or set of functions) where the definition of f_1 uses f_2 and f_2 either directly or indirectly uses f_1.

can be thought of as:

$$(\text{let } foo \;=\; (\text{fix } (\lambda\ (f)\ \text{body}[f]))$$

Where now foo is no longer defined in terms of itself.

Luckily, FKS has a fixed point operator, namely Y (actually Y is a "least" fixed point operator, but it is beyond the scope of this section to explain that here). Let us now look at how to define $fact$ in FKS. First we need to translate F into FKS. So:

$$F \overset{\text{def}}{=} (\lambda f^{(\iota\rightarrow\iota)}\lambda n^{\iota}.\ (\text{cond } (=\ n)\ 0\ 1)\ ((*\ n)\ (f\ (pred\ n))))$$

Finally we can now define $fact$:

$$fact \overset{\text{def}}{=} (Y_{(\iota\rightarrow\iota)}F)$$

So, this is how to define a simply recursive function. Using tupling we can define mutually recursive functions. This is a slight modification of the preceding example of what mutually recursive definitions should look like. The difference is that we will use b'_i where:

$$b'_i \overset{\text{def}}{=} (\lambda h.(\lambda g_1 \ldots \lambda g_n.\ (b_i[g_1,\ldots,g_n])\ proj_{n,1}(h)\cdots proj_{n,n}(h)))$$

and the final letrec is:

$$\begin{array}{lll}
(\text{letrec} & & \\
 & (f_1 = & (b'_1\ tuple_n(f_1,\ldots,f_n)) \\
 & \ldots & \\
 & (f_n = & (b'_n\ tuple_n(f_1,\ldots,f_n)) \\
 & &)
\end{array}$$

So we have simplified the definitions of each of the functions to be of the form $f_i = b'_i foo$ where foo is $tuple_n(f_1,\ldots,f_n))$. So if we can define an expression which generates foo then we are done (because $f_i = proj_{n,i}(foo)$). So we want to define a function F that has foo as its least fixed point. Here is is:

$$F \overset{\text{def}}{=} (\lambda H.\ tuple_n((b'_1\ H),\ldots,(b'_n\ H)))$$

So in conclusion, we have:

$$f_i \overset{\text{def}}{=} proj_{n,i}(Y\ F)$$

Debunking Y: The Fixed Point Operator. Up until now we have taken the tack that Y is a fixed point operator. We have not, however justified this statement. Using our rewrite rules we can check that Y is a fixed point operator. Our goal would to prove a statement analogous to $F(\mathit{fix}\ F) = F)$. For FKS this statement takes the form $(V\ (Y\ V)){\equiv_{obs}}(Y\ V)$, where V can be any value that is not of type ι (thus probably, a λ-abstraction). But look:

$$(Y_\sigma V) \rightarrow (V(\lambda x^{(\sigma\rightarrow\sigma)}.(Y_\sigma V)x))\ \ ($$

for

$$x^\sigma \notin FV(V))$$

You can check yourself the following is a general rule:

If M is of type $\sigma \neq \iota$, M does not diverge, and $x^\sigma \notin FV(M)$, then $M{\equiv_{obs}}(\lambda x^\sigma.Mx)$

Since $(Y_\sigma\ V)$ and x^σ meet the antecedent of this rule, then

$$(Y_\sigma\ V){\equiv_{obs}}(\lambda x^\sigma.(Y\ V)\ x)$$

Since we can interchange terms which are ${\equiv_{obs}}$to each other with impunity, we have:

$$(V\ (Y_\sigma\ V)){\equiv_{obs}}(V\ (\lambda x^\sigma.(Y_\sigma\ V)\ x))$$

and, given that ${\equiv_{obs}}$is an equivalence relation:

$$(V\ (Y_\sigma\ V)){\equiv_{obs}}(Y_\sigma\ V)$$

this is the desired result.

CONTINUED ON THE NEXT PAGE

Example 7. To really see how Y works in defining recursion, consider the computation of (fact 2), where:

$$\begin{aligned}
F &\overset{\text{def}}{=} (\lambda f^{(\iota\rightarrow\iota)}.\lambda n^{\iota}.(\text{cond } (=\ n\ 0)\ 1\ (*\ n\ (f\ (-1\ n)))))\\
fact &\overset{\text{def}}{=} (Y_{(\iota\rightarrow\iota)}F)\\
\text{body1} &\overset{\text{def}}{=} (\lambda n^{\iota}.(\text{cond } (=\ n\ 0)\ 1\ (*\ n\ (f\ (-1\ n)))))\\
\text{body2} &\overset{\text{def}}{=} (\text{cond } (=\ n\ 0)\ 1\ (*\ n\ (f\ (-1\ n))))\\
\text{foo} &\overset{\text{def}}{=} (\lambda x^{(\iota\rightarrow\iota)}.\ (Y\ F)\ x)
\end{aligned}$$

Here is the evaluation, in hideously gory detail (we will use $\equiv$ to mean syntactic equality):

$$\begin{aligned}
(fact\ 2) &\equiv ((YF)2)\\
&\rightarrow (F(\lambda x.(YF)x)2)\\
&\rightarrow (\text{body1}[f := foo]2)\\
&\rightarrow \text{body2}[f := foo][n := 2]\\
&\equiv (\text{cond } (=\ 2\ 0)\ 1\ (*\ 2(foo(pred2))))\\
&\rightarrow (\text{cond } (=_2\ 0)\ 1\ (*\ 2(foo(pred2))))\\
&\rightarrow (\text{cond } 1\ 1\ (*\ 2(foo(pred2))))\\
&\rightarrow (*\ 2(foo(pred2)))\\
&\rightarrow (*_2\ (foo(pred2)))\\
&\equiv (*_2\ ((lambdax^{\cdots}.\ (Y\ F)\ x)\ (pred\ 2)))\\
&\rightarrow (*_2\ ((lambdax^{\cdots}.\ (Y\ F)\ x)\ 1))\\
&\rightarrow (*_2\ ((Y\ F)\ 1)\\
&\rightarrow (*_2\ (F\ (\lambda x.\ (Y\ F)\ x)\ 1))\\
&\rightarrow (*_2\ (\text{body1}[f := foo]\ 1))\\
&\rightarrow (*_2\ \text{body2}[f := foo][n := 1]\\
&\equiv (*_2\ (\text{cond } (=\ 1\ 0)\ 1\ (*\ 1(foo\ (pred\ 1)))))\\
&\rightarrow (*_2\ (\text{cond } (=_1\ 0)\ 1\ (*\ 1(foo\ (pred\ 1)))))\\
&\rightarrow (*_2\ (\text{cond } 1\ 1\ (*\ 1(foo\ (pred\ 1)))))\\
&\rightarrow (*_2\ (*\ 1\ (foo\ (pred\ 1))))\\
&\rightarrow (*_2\ (*_1\ (foo\ (pred\ 1))))\\
&\equiv (*_2\ (*_1\ ((lambdax^{\cdots}.(Y\ F)\ x)\ (pred\ 1))))\\
&\rightarrow (*_2\ (*_1\ ((lambdax^{\cdots}.(Y\ F)\ x)\ 0)))\\
&\rightarrow (*_2\ (*_1\ ((Y\ F)\ 0)\\
&\rightarrow (*_2\ (*_1\ (F\ (\lambda x.\ (Y\ F)\ x)\ 0))
\end{aligned}$$

$$\begin{array}{ll}
\rightarrow & (*_2\ (*_1\ (\text{body1}[f := foo]\ 0)) \\
\rightarrow & (*_2\ (*_1\ \text{body2}[f := foo][n := 0] \\
\equiv & (*_2\ (*_1\ (\text{cond}\ (=\ \ 0\ 0)\ 1\ (*\ 0(foo\ (pred\ 0)))))) \\
\rightarrow & (*_2\ (*_1\ (\text{cond}\ (=_0\ \ 0)\ 1\ (*\ 0(foo\ (pred\ 0)))))) \\
\rightarrow & (*_2\ (*_1\ (\text{cond}\ 0\ 1\ (*\ 0(foo\ (pred\ 0))))) \\
\rightarrow & (*_2\ (*_1\ 1)) \\
\rightarrow & (*_2\ 1) \\
\rightarrow & 2
\end{array}$$

Example 8. Definability of the basic operation "+" (in curried form). The curried form of "+" has type $\iota \rightarrow \iota \rightarrow \iota$.

$$+ \overset{\text{def}}{=} (Y^{(\iota\rightarrow\iota\rightarrow\iota)}\ (\lambda f^{(\iota\rightarrow\iota\rightarrow\iota)} \lambda x^{\iota} \lambda y^{\iota}.\ (\text{cond}xy(\text{succ}((f(\text{pred}x))y))))$$

A good exercise to enhance your understanding of how the rewrite rules and recursion works would be to hand evaluate ((+3)10). Another good exercise would be to define * and exp in this manner.

Example 9. Definability of a Primitive Recursion operator PR.

Recall the definition of primitive recursion. An $n+1$-ary function h can be defined by primitive recursion from an n-ary function f and an n+2-ary function g as follows:

- $h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n)$
- $h(x_1, \ldots, x_n, (succy)) = g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y))$

I am going to provide a definition of a primitive recursion operator PR_1 for the case of $n = 1$. It is straightforward to then define PR_n for all n. PR_1 has type: $\sigma_f \rightarrow \sigma_g \rightarrow \sigma_h$, where $\sigma_f = \iota \rightarrow \iota$, $\sigma_g = \iota \rightarrow \iota \rightarrow \iota \rightarrow \iota$, and $\sigma_h = \iota \rightarrow \iota \rightarrow \iota$. So, here is the definition:

$$\begin{array}{lll}
PR_1 & \overset{\text{def}}{=} & \lambda f^{\sigma_f} \lambda g^{\sigma_g}. \\
& & \quad Y_{\sigma_h}(\lambda h^{\sigma_h} \lambda x^{\iota} \lambda y^{\iota}. \\
& & \qquad\qquad (\text{cond}\ y \quad (f\ x) \\
& & \qquad\qquad\qquad\qquad (((g\ x)\ (\text{pred}y))\ (h\ x(\text{pred}y)))))
\end{array}$$

Example 10. At this point it should be clear how to program all of the primitive recursive functions in FKS (at least their curried versions). To check this, we simply need to verify that we have all of the basic functions and operations needed to define them. Let's go through a check:

- We have the successor function built in.
- The zero function is: $(\lambda x^{\iota}.0)$
- The identity functions can be defined analogous to the pairing operators defined earlier. In general $ID_i^n = (\lambda x_1^{\iota} \ldots \lambda x_n^{\iota}.\ n_i)$.
- Composition of m n-ary functions. Again the set of combinators $CN_{m,n}$ is easily λ-definable. It is left as an exercise.
- Primitive recursive has just been given.

So we can code any primitive recursive function of n-arguments by a function in FKS of type:

$$\underbrace{\iota \rightarrow \ldots \rightarrow \iota}_{n\ times} \rightarrow \iota$$

This ability to code all of the functions of a class like this in our language is called ***numeralwise representability.*** We say that the all of the primitive recursive functions are ***numeralwise representable*** FKS.

Example 11. Now that we know that all of the primitive recursive functions are numeralwise representable in FKS, we would like to show that all of the partial recursive functions are numeralwise representable in FKS.

This simply involves demonstrating that we can code the set of minimization operations Mn_n. We will define Mn_1. It is of type: $\sigma_f \rightarrow \iota \rightarrow \iota$, where $\sigma_f = \iota \rightarrow \iota$. In the definition we will use: $\sigma_h = \iota \rightarrow \iota$. So, here it is:

$$\begin{array}{lll} Mn_1 & \overset{\text{def}}{=} & (\lambda f^{\sigma_f} \lambda x^{\iota}. \\ & & \quad ((Y_{\sigma_h}(\lambda h^{\sigma_h} \lambda y^{\iota}. \\ & & \qquad\qquad (\text{cond } ((f\ x)\ y) \quad y \\ & & \qquad\qquad\qquad\qquad\qquad\quad (h\ (succ\ y))))) \\ & & \qquad 0)) \end{array}$$

And, we are done.

Thus FMS can define all of the partial recursive functions, so it is just as powerful as any of the other models of computation which we have seen. In addition, it is not too powerful. Although it may not be immediately clear how you could write an interpreter for FKS, the SECD machine, which will begin the next section of notes, operates by very simple pointer manipulations and will obviously be implementable via a turing machine or μ-recursive functions.

Notes on Programming (Part II)

by Arthur Lent

1 More on Recursion and Y

This section is included in order to help you understand, from a computational point of view, why Y works in defining recursive functions. We will return to the example of factorial which occurs in pages 11-13 of the first part of the notes. Consider how *fact* is defined in SCHEME:

$$\text{(define } (\mathit{factn}) \text{ (cond } (= \ n\ 0)\ 1\ (*\ n\ (\mathit{fact}\ (-1\ n)))))$$

As users of SCHEME, we know how this factorial function works. But it is important to see that, from a naive mathematical perspective, this "definition" of fact is **not** a definition in the traditional sense—the "defined" value appears in its own definition! Rather, it states a **property** that any "*fact*" function must satisfy. So, we are going to have to play some tricks in order to get a true definition of a factorial function.

Now look at the following code:

$$\text{(lambda } (n) \text{ (cond } (= \ n\ 0)\ 1\ (*\ n\ (\mathit{fact}\ (-1\ n)))))$$

This code can be thought of as a function of one variable, whose definition uses whatever function is bound to the identifier *fact*. So let us abstract this expression over the identifier *fact*, and rename the thusly bound identifier to f, and call the resulting expression F:

$$F \overset{\text{def}}{=} \text{(lambda } (f) \text{ (lambda } (n) \text{ (cond } (= \ n\ 0)\ 1\ (*\ n\ (f\ (-1\ n))))))$$

F is now a function which takes two arguments. If you were to apply F to the factorial function, the result would be the factorial function. Thus we get an equation:

$$\mathit{fact} = F(\mathit{fact})$$

Since *fact* has this property, it is called a *fixed point* of the function F. In general, we define a fixed of a function G to be any element x of the domain of G such that $G(x) = x$. We also define a ***fixed point operator***, call it ***fix***, to be a functional that takes a function as its argument, and returns the fixed point of its argument. It turns out that a fixed point of a function G can be found by repeatedly applying it to itself "infinitely many times"

Now consider the infinite list of functions:

$$\begin{array}{lll}
h_0 & \overset{\text{def}}{=} & \text{a function that is undefined on all inputs} \\
h_1 & \overset{\text{def}}{=} & h_1(0) = 0!, h_1(x) \text{ undefined for } x \geq 1 \\
h_2 & \overset{\text{def}}{=} & h_2(0) = 0!, h_2(1) = 1!, h_2(x) \text{ undefined for } x \geq 2 \\
h_3 & \overset{\text{def}}{=} & h_3(0) = 0!, h_3(1) = 1!, h_3(2) = 2, h_3(x) \text{ undefined for } x \geq 3 \\
 & & \ldots \\
h_n & \overset{\text{def}}{=} & h_n(0) = 0! \ldots h_2(n-1) = (n-1)!, \text{ undefined for } x \geq n \\
 & & \ldots
\end{array}$$

We can imagine that this list goes on forever, approaching a single function, h_∞. But this supposed limit, h_∞, *is* the factorial function. Another way of looking at it is that for every x there is an element in the list, h_i, such that $h_i(x) = x!$. This fact will be important later when we look at mathematical models.

What else can we see? Well, notice that $(F(h_0)) = h_1$. More generally, $(F(h_i)) = h_{i+1}$. But then:

$$(\underbrace{(F(F \ldots F(}_{n\ F's} h_0))) = h_n$$

We can pretend n goes to ∞ and we have:

$$(F(\ldots F(h_0))) = \text{factorial}$$

We could then almost say, then that:

$$(F(F(\ldots F(h_0)))) = F(\text{factorial})$$

But then we could almost say:

$$\text{factorial} = F(\text{factorial})$$

So we are almost truly convinced that:

$$(\text{let } fact = (fix\ F))$$

really defines *fact* to be the factorial function. By looking carefully at the example on pages 14-15 of the previous set of notes you can see this process at work (note: In FKS, Y does the job of *fix*). You can witness that this process evaluates the body of F, then just when it needs it, $(Y F)$ appears with another copy of the body of F for the next recursive call.

There is one difficulty with this description. Joseph Stoy characterizes it quite well in [1]:

"It [the argument just given] certainly leaves some unanswered questions, which must be settled before we can be happy with the above definition of $fact$. If B has more than one fixed point [such as $(\lambda x^{\iota}.x)$], which one does $(Y\ B)$ produce? What happens when Y is applied to such expressions as $\lambda x.x+1$ which ... has no fixed points? It is questions like this last one that have earned Y the title of 'paradoxical combinator'. "

In the case of $((Y\ G)$ args) (with args appropriate to make that expression of type ι) where G has no fixed point, that term will diverge. But the question of multiple fixed points is much more subtle. It turns out that Y is a "least" fixed point operator, picking the "least" such result that is still a fixed point, yet is compatible with all of the other fixed points. This notion of "least" will be made more precise in the section on denotational semantics.

References

[1] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

Notes on Programming (Part III)

by Arthur Lent

This handout assumes that you are familiar with the material contained in handout #29 (Part I of these notes). In particular you need to know the definition of terms, FV, BV, the substitution operation, $=_\alpha$, and the rewrite rules.

1 The SECD Machine

Introduction. The SECD machine was first introduced by Landin in 1963[1]. The importance of the machine's introduction was that it was the first time that a language was described abstracted away from a particular implementation—in the past, a language was defined by the first compiler written for it. The SECD machine takes a role between the further abstraction of rewrite rules and the concreteness of an actual implementation—making explicit the control structure of evaluation, yet still leaving unspecified arbitrary details.

The treatment here is taken largely from a work by Plotkin in 1975[2].

What is a SECD Machine. SECD stands for Stack–Environment–Control-string–Dump. The precise technical definition of each of these terms will be given in the next section. First, we wish to cultivate some intuitions about the SECD machine. The SECD machine is an automaton whose basic actions are pointer manipulations. In fact, its basic actions involve a constant number of pointer manipulations. Once you see the full definitions it should be immediately clear how to built an interpreter for FKS. You should be able to do it in about an afternoon. You could not say the same thing about the rewrite rules presented in the first section of the notes. Not only does this description lend itself much more to being implemented, an implementation based directly on the SECD machine is far more efficient than one based upon the rewrite rules. Yet this does not mean that the rewrite rules are without value—they probably make a more understandable operational definition of FKS, and they will be essential for our work on denotational semantics.

Before we give technical definitions of the four components of the SECD machine we will tell what each piece will be used for:

Stack The stack will be used to store intermediate results in the evaluation of a function.

Environment The environment is the environment in which the function is being evaluated.

Controlstring The controlstring contains whatever operations still need to be performed in order to complete the current function call.

Dump The dump stores the state of the machine that existed immediately prior to the current function call. (It can be viewed as a stack of the preceding activation records).

Definitions. The first new concepts to be defined are the sets **Environments** and **Closures**. These correspond very closely to the notions defined in 6.001. In fact, they are the same notions, but, in a language without side effects, they have some additional nice properties. But first we should back up a step and address why the SECD machine needs environments and closures. It uses them in order to implement substitution. The process of turning an arbitrary term M into the term $M[x := N]$ is nontrivial. The traditional way of doing substitution is to say that the term $M[x := N]$ can represented by M paired with an *environment* an object that states that x really is N. M paired with this environment is called a closure, and it "represents" the term $M[x := N]$. Now what if we want to substitute $(M[x := N])$ for y into P to get the term $P[y := (M[x := N])]$. We want to represent this by the closure $[P, E]$ where $E(y) = (M[x := N])$. Unfortunately, we do not actually have our hands on the term $M[x := N]$—we have our hands on a closure representing it. Thus we do not really want environments to map from variables to terms, but, rather, we want them to map from variables to closures. This may look like a circular definition, but then so does defining two functions in a mutually recursive way. Here is a simultaneous mutual inductive definition of the set **Closures** of closures, the set **Value Closures** of value closures and **Environments** of environments:

- $\emptyset$, is an abbreviation for the totally undefined function. It is an environment.
- A partial function with finite domain, that maps variables of type σ to value closures of type σ is an environment.
- If E is an environment, and M of type σ such that $FV(M) \subseteq Domain(E)$ then $[M, E]$ is a closure of type σ. (In other words, E is defined on all of the free variables of M).
- If M is a value of type σ and $[M, E]$ is a closure then $[M, E]$ is a value closure.

Note that any closed term M can be represented by the closure $[M, \emptyset]$.

We also define $E\{Cl/x\}$ (x and Cl must have the same type) to be the unique environment E' such that $E'(y) = E(y)$ if $y \neq x$ and $E'(x) = Cl$ (for any $Cl \in$ **Closures**).

Finally, to drive home the point that a closure can mechanically be "unwound" into the term it represents we define a function Realterm : **Closures** $\rightarrow$ **Terms**. It is defined inductively by:

$$\text{Realterm}([M, E]) = M[x_1 := \text{Realterm}(E(x_1))] \ldots [x_n := \text{Realterm}(E(x_n))]$$

where

$$FV(M) = \{x_1, \ldots, x_n\}$$

Note that this unwinding property is only possible when there are no side-effects in the terms inside the closures being unwound, so for Scheme it will not work, but there are many cases where closures *are* built up from terms without side-effects, in which case you really can think of those closures in terms of this unwinding process.

The set of stacks, **Stacks**, is the set of all finite sequences of closures, formally written as **Stacks** = (**Closures**)*.

The set of controlstrings, **Controlstrings** = (**Terms** $\cup\, ap, cd)^*$ where ap, cd are special symbols that are not elements of **Terms**. The function FV is easily extended to work on controlstrings as follows:

- $FV(ap) = \emptyset$
- $FV(cd) = \emptyset$
- $FV(C_1, \ldots, C_n) = \bigcup_{i=1}^{n} \; FV(C_i)$ $(n \geq 0)$

Finally, the set of dumps, **Dumps**, is defined inductively by:

- $nil \in$ **Dumps**
- If $S \in$ **Stacks**, $E \in$ **Environments**, $C \in$ **Controlstrings** and C is such that $FV(C) \subseteq Domain(E)$, and $D \in$ **Dumps** then $[S, E, C, D] \in$ **Dumps**

This concludes the definitions of the primary data structures manipulated by the SECD machine.

The functions constapply and Constapply. The SECD machine model which we will present in the next section will be defined in a manner that abstracts away from the constants that we have chosen to include in FKS. Thus we could, in principle, add constants to the language (such as a curried plus operator) and the main proofs about the SECD machine would carry through directly. In addition this abstraction separates out the "constant stuff" for a particular language from the general principles of interpreting a functional language.

The SECD machine will employ the function **Constapply** when it hits an operator that is a constant. Since constant operators (namely Y) in our language can take values as arguments, and values are general terms which might need closures to fully define them, Constapply needs to be a partial function of the type:

$$\textbf{Constants} \times \textbf{Closures} \rightarrow \textbf{Value Closures}$$

We will also be presenting a recursive characterization of the rewrite rules which does not use stacks, closures, environments, or dumps, yet still captures explicitly the order of evaluation of the SECD machine. But for this recursive characterization, which we will from now on call *eval*, we still need a **Constapply** sort of function, but it must live wholly in the world of terms (no closures allowed). We will call it **constapply**, and it needs to be a partial function of type:

$$\textbf{Constants} \times \textbf{Closed Values} \rightarrow \text{Closed Terms}$$

In actuality we will not define **Constapply** directly. Instead we will define **constapply**, and then state that **Constapply** is as determined as it needs to be by the following restriction:

$$\text{Realterm}(\textbf{Constapply}(a, Cl)) =_{\alpha} \textbf{constapply}(a, \text{Realterm}(Cl))$$

What this requires is that **Constapply** gives a result that is independent of how the closure that is its argument represents the term it is acting upon. So **Constapply** cannot distinguish between:

$$[x, \{[x, (MN)]\}]$$

and

$$[(x\ y), \{[x, M], [y, N]\}]$$

Here is the definition of **constapply** which we will use for FKS:

succ	**constapply**(succ, $\mathbf{n}$)	→	$\mathbf{n} + \mathbf{1}$
pred	**constapply**(pred, $\mathbf{n} + \mathbf{1}$)	→	$\mathbf{n}$
	constapply(pred, $\mathbf{0}$)	→	$\mathbf{0}$
Y_σ	**constapply**(Y_σ, V)	→	$(V(\lambda x^{(\sigma\rightarrow\sigma)}.(Y_\sigma V)x))$ (for an $x \notin FV(V)$ and for V a value and for $\sigma \neq \iota$)

Order of evaluation. The SECD machine will evaluate the operands of a combination before the operators. This is to be contrasted with the order of the rewrite rules which evaluate operators before operands. We apologize for this confusion, and it should be obvious how to change the rewrite rules or the SECD machine so that the order is reversed. Our main theorem, however, carries through whichever order used in whichever scheme. This is because, for FKS, the order does not matter (in fact for a particular scheme expression, where the evaluating the operator or operands does not produce any side effects, the order does not matter). Note that the definition of Scheme does not even specify which is the correct order. The following is an exerpt from the Scheme manual given to 6.001 students in Spring '89:

"A procedure call is written by enclosing in parenthesis expressions for the procedure to be called and the arguments to be passed to it. The operator and operand expressions are evaluated **(in an indeterminate order)** and the resulting procedure is passed the resulting arguments... Procedure calls are also called *combinations*" (emphasis added).

The function SECD. The state transition function, a partial function from **Dumps** to **Dumps** is defined as follows:

1. $[Cl : S, E, \mathit{nil}, [S', E', C', D']] \Rightarrow [Cl : S', E', C', D']$
2. $[S, E, x : C, D] \Rightarrow [E(x) : S, E, C, D]$
3. $[S, E, a : C, D] \Rightarrow [[a, \emptyset] : S, E, C, D]$
4. $[S, E, (\lambda x.\ M) : C, D] \Rightarrow [[(\lambda x.\ M), E] : S, E, C, D]$
5. $[[(\lambda x.\ M), E'] : Cl : S, E, \mathit{ap} : C, D] \Rightarrow [\mathit{nil}, E'\{Cl/x\}, M, [S, E, C, D]]$
6. $[[a, \emptyset] : [V, E''] : S, E, \mathit{ap} : C, D] \Rightarrow [\mathit{nil}, E', M', [S, E, C, D]]$
   (where $\mathbf{Constapply}(a, [V, E'']) = [M, E']$)
7. $[S, E, (M\,N) : C, D] \Rightarrow [S, E, N : M : \mathit{ap} : C, D]$
8. $[S, E, (\text{cond } M N_1 N_2) : C, D] \Rightarrow [[N_1, E] : [N_2, E] : S, E, M : \mathit{cd} : C, D]$
9. $[[\mathbf{o}, E_0] : [N_1, E_1] : [N_2, E_2] : S, E, \mathit{cd} : C, D] \Rightarrow [S, E_1, N_1 : C, D]$
10. $[[[\boldsymbol{n} + \mathbf{1}, E_0] : [N_1, E_1] : [N_2, E_2] : S, E, \mathit{cd} : C, D] \Rightarrow [S, E_2, N_2 : C, D]$

We now need two functions **Load** and **Unload** which convert terms into SECD machine state, and SECD machine state into terms. Specifically they are defined by:

$$\mathbf{Load}(M) = [\mathit{nil}, \emptyset, M, \mathit{nil}]$$

$$\textbf{Unload}([Cl, \emptyset, nil, nil]) = \text{Realterm}(CL)$$

We can now define an evaluation function, which is a partial function from terms to values as follows:

$$\text{SECD}(M) = V \text{ iff } \textbf{Load}(M) \overset{*}{\Rightarrow} D, \text{ and } V = \textbf{Unload}(D) \text{ for some dump D}$$

The punchline of the section on SECD machines is the following theorem:

Theorem 1. SECD(M)$=_\alpha N$ iff $Eval(M) =_\alpha N$ for all terms M and N.

Remember that M and N are $=_\alpha$ iff they differ only in the names of their bound variables—called α-equivalence or *equal up to renaming of bound variables.*

In order to prove this theorem we will introduce another scheme for evaluating programs that is midway between the rewrite rules and the SECD machine. This will be a simple recursive definition that uses substitution rather than closures.

We would to like to find a (partial) function *eval* : **Closed Terms → Values** such that:

$$eval(a) = a; \quad eval(\lambda x M) = \lambda x M$$

$$eval(MN) = \begin{cases} eval(M'[x := N']) & (\text{if } eval(M) = \lambda x M' \\ & \text{and } eval\text{(N)=N')} \\ eval(\textbf{constapply}(a, N')) & (\text{if } eval(M) = a \\ & \text{and } eval(N) = N') \end{cases}$$

$$eval(\text{cond } M\ N_1\ N_2) = \begin{cases} eval(N_1) & (\text{if } eval(M) = \mathbf{o}) \\ eval(N_2) & (\text{if } eval(M) = \boldsymbol{n} + \mathbf{1}) \end{cases}$$

Now this may look like a good recursive definition of a partial function, and it turns out that it *is* good, but the precise sense in which equational recursive definitions of partial functions work requires avoiding some mathematical pitfalls, which we must not take for granted. So, to be perfectly precise about how *eval* is defined, we define the predicate "M evals to N at stage t" by induction on t, for closed terms M and closed values N

1. a evals to a at stage 1; $(\lambda x M)$ evals to $(\lambda x M)$ at stage 1.
2. If M evals to $(\lambda x M')$ at stage t and N evals to N' at stage t' and $[N'/x]M'$ has value L at stage t'' then (MN) evals to L at stage $t + t' + t'' + 1$.
3. If M evals to $\mathbf{o}$ at stage t and N_1 evals to L at stage t' then (cond M N_1 N_2) evals to L at stage $t + t' + 1$. If M evals to $\boldsymbol{n} + \mathbf{1}$ at stage t and N_2 evals to L at stage t' then (cond M N_1 N_2) evals to L at stage $t + t' + 1$.

4. If M evals to a at stage t and N evals to N' at stage t' and if constapply(a,N') is defined and evals to N'' at stage t'', then $(M\,N)$ evals to N'' at stage $t+t'+t''+1$.

It is a fairly simple induction on t to show that for all M, there is *at most one* pair (N,t) such that M evals to N at stage t. Consequently this is a good definition of a partial function:

$$eval(M) = N \text{ iff } M \text{ evals to } N \text{ at some stage.}$$

There is a better way of defining *eval*via an inference relation $eval_r$; however, time has not permitted working out this better definition.

Proving the equivalence of SECD and *eval*. In before we prove Theorem 1 we first prove the following, easier Theorem (notice the little "e"):

Theorem 2. SECD(M)$=_\alpha N$ iff $eval(M)=_\alpha N$ for all terms M and N.

This theorem will be proven using three lemmas. The first says using closures and environments to model substitution "works right". The second will prove direction ⇒ of this theorem, and the third will prove direction ⇐ of this theorem.

Lemma 1. Suppose $[\lambda y.\ M, E]$ and $[N, E']$ are value closures. Also suppose that Realterm($[\lambda y.\ M, E]$)$=_\alpha(\lambda x.M')$ and

Realterm($[N,E']$)$=_\alpha N'$. Then Realterm($[M, E\{[N,E']/y\}]$)$=_\alpha M'[x := N']$.

Proof Sketch: Observe that if $\lambda y.\ M =_\alpha \lambda x.\ M'$ then $M =_\alpha M'[x := y]$, hence $M[y := N] =_\alpha M'[x := N]$. The rest is a simple unwinding of the closures and simply examining the definition of Realterm. ■

The proof of this next Lemma captures how the SECD machine really works. It is quite long, however, thus we will leave out the details of a few of the cases.

Lemma 2. Suppose E is an environment and $[M, E]$ is a closure. Suppose Realterm($[M, E]$) evals to M''. Suppose C is a controlstring with $FV(C) \subseteq Domain(E)$. Then there is a $t' \geq t$, such that for all S, D,

$$[S, E, M : C, D] \overset{t'}{\Rightarrow} [[M', E'] : S, E, C, D]$$

where $[M', E']$ is a value closure and Realterm($[M', E']$)$=_\alpha M''$.

This Lemma really does entail the right hand direction of Theorem 2, but requires in its statement a rather hefty induction hypothesis.

Proof: This is a proof by induction on t. It is quite similar to that presented by Plotkin [2]. There are 5 main cases:

1. M is a constant. Here Realterm($[M, E]$) $= M = M''$ and $t = 1$. As

   $$[S, E, M : C, D] \Rightarrow [[M, \emptyset] : S, E, C, D]$$

   we can take $[M', E'] = [M, \emptyset]$ and $t' = 1$.

2. M is a λ-abstraction. Almost the same as the previous case.

3. M is a variable. Take $[M', E'] = E(M)$, and $t = 1$.

4. $M =$ (cond P N_1 N_2) is a conditional. Apply the inductive hypothesis to P and then divide by cases according to P' the value that P evals to at stage t_1

5. $M = (M_1\ M_2)$ is a combination. Then

   $$\begin{aligned} \text{Realterm}([M, E]) &= (\text{Realterm}([M_1, E])\ \text{Realterm}([M_2, E])) \\ &= (N_1\ N_2) \text{ say.} \end{aligned}$$

   This now divides into two subcases, depending on whether or not the value to which N_1 evals to is a λ-abstraction or a constant.

   (a) $(\lambda x.\ N_3)$ is the value that N_1 evals to at stage t_1, N_4 is the value that N_2 evals to at stage t_2, M'' is the value that $N_3[x := N_4]$ evals to at stage t_3 and $t = t_1 + t_2 + t_3 + 1$.
   Then by the induction hypothesis there are $t_i' \geq t_i$ $(i = 1,\ 2)$ such that:

   $$\begin{aligned} [S, E, (M_1\ M_2) : C, D] &\Rightarrow [S, E, M_2 : M_1 : ap : C, D] \\ &\overset{t_1'}{\Rightarrow} [[M_2', E_2'] : S, E, M_1 : ap : C, D] \\ &\overset{t_2'}{\Rightarrow} [[M_1', E_2'] : [M_2', E_2'] : S, E, ap : C, D] \end{aligned}$$

   where

   $$\text{Realterm}([M_1, E_1]) =_\alpha (\lambda x.\ M) \text{ and Realterm}([M_2', E_2']) =_\alpha N_4,$$

   and the $[M_i', E_i']$ are value closures.
   Here $M_1' = (\lambda y.\ M_3')$ for some M_3', and

   $$\text{Realterm}([M_3', E_1'\{[y := [M_2', E_2']]\}]) =_\alpha [N_4/x]N_3 \text{ (by Lemma 1).}$$

Now,

$$\begin{aligned}
[[M_1', E_1'] : \quad & [M_2', E_2'] : S, E, ap : C, D] \\
& \Rightarrow [nil, E_1'\{[M_2', E_2']/y\}, M_3', [S, E, C, D]] \\
& \overset{t_3'}{\Rightarrow} [[M', E'], E_1'\{[M_2', E_2']/y\}, nil, [S, E, C, D]] \\
& \Rightarrow [[M', E'] : S, E, C, D]
\end{aligned}$$

where, by the induction hypothesis, Realterm($[M', E']$) is to within α-equivalence the value that Realterm($[M_3', E_1'\{[M_2', E_2']/y\}]$) evals to at stage $t_3 \leq t_3'$ and $[M', E']$ is a value closure. Taking $t' = t_1' + t_2' + t_3' + 3$ concludes this subcase.

(b) a is the value that N_1 evals to at stage t_1, V is the value that N_2 evals to at stage t_2. Then, by the inductive hypothesis there are $t_i' \geq t_i$ (i=1, 2) , and a value closure VC such that:

$$\begin{aligned}
[S, E, (M_1\ M_2), C, D] \quad & \Rightarrow \quad [S, E, M_2 : M_1 : ap : C, D] \\
& \overset{t_2'}{\Rightarrow} \quad [VC : S, E, M_1 : ap : C, D] \\
& \overset{t_1'}{\Rightarrow} \quad [[a, \emptyset] : VC : S, E, ap : C, D]
\end{aligned}$$

where Realterm(VC) $= V$. Now, finally, suppose that we have **Constapply**(a, VC) $= [M'', E'']$, and N'' is the value to which Realterm($[M'', E'']$) evals at stage t_3 (thus N'' is the value to which **constapply**(a, Realterm(VC)) evals at stage t_3). By the induction hypothesis there are t_3' and VC' such that:

$$\begin{aligned}
[S, E, (M_1\ M_2), C, D] \quad & \overset{t_1'+t_2'+1}{\Rightarrow} \quad [[a, \emptyset] : VC : S, E, ap : C, D] \\
& \Rightarrow \quad [nil, E'', M'', [S, E, C, D]] \\
& \overset{t_3'}{\Rightarrow} \quad [VC', E'', nil, [S, E, C, D]] \\
& \Rightarrow \quad [VC' : S, E, C, D]
\end{aligned}$$

where Realterm(VC') $= N''$. Then taking $t' = t_1' + t_2' + t_3' + 3$ and $[M', E'] = VC'$ concludes the proof of the lemma.

■

Before we introduce the next lemma, we need a definition. If $D \overset{t}{\Rightarrow} D'$, where D' does not have the form $[Cl, \emptyset, nil, nil]$ and $D' \not\Rightarrow D''$ for any D'' then D is said to *hit an error state* (viz. D').

Lemma 3. Suppose E is a value environment and $[M, E]$ is a closure. If Realterm($[M, E]$) does not eval to a value at any $t' \leq t$, then either for all S, C, D, with $FV(C) \subseteq Domain(E)$, $[S, E, M : C, D]$ hits an error state or else $[S, E, M : C, D] \overset{t}{\Rightarrow} D'$ for some D'.

Proof Sketch: This is proved by induction on t—the number of steps used by the SECD machine to evaluate M. It.is just a horrible counting exercise that can just be grunged through. ■

Proof: **(Theorem 2).** Suppose $eval(M) = M''$. Then at some stage t, M'' is the value that M evals to at stage t. By lemma 2,

$$[nil, \emptyset, M, nil] \overset{t'}{\Rightarrow} [[M', E'], \emptyset, nil, nil],$$

where Realterm($[M', E']$)$=_\alpha M''$. So $Eval(M) =_\alpha M''$.

Suppose, on the other hand, that M does not eval to a value at any stage. Then by Lemma 3 either $[nil, \emptyset, M]$ hits an error state or else for every t there is a D such that $[nil, \emptyset, M, nil] \overset{t}{\Rightarrow} D$. In either case SECD($M$) is also not defined. ■

Proving the equivalence of eval and Eval.

Theorem 3. For all well-typed, closed terms M with constants in **Constants** then $M \twoheadrightarrow M'$ (M' a value) iff M evals to M' at some stage t ($eval(M) = M'$).

But first we need several facts:

Fact 1. $\rightarrow$ is deterministic. That is: if $M \rightarrow M'$ then $\nexists M'' \neq M'$ such that $M \rightarrow M''$. Thus if $M \xrightarrow{n} M''$, $M \xrightarrow{m} M'$ and $m \leq n$ then $M' \xrightarrow{n-m} M''$.

Fact 2. If $M_1 \xrightarrow{n} M_1'$ then $(M_1 M_2) \xrightarrow{n} (M_1' M_2)$ and $(aM_1) \xrightarrow{n} (aM_1')$

Fact 3. If M is a closed value, then $(cM) \rightarrow$ **constapply**(c, M) which is to say that if **constapply**(c,M) is defined then (cM) reduces to it, and if **constapply**(c,M) is not defined then $\nexists M' : (cM) \rightarrow M'$.

Proof: $(\boldsymbol{M \xrightarrow{n} M' \Rightarrow eval(M) = M'})$. By induction on n.

Basis. $n = 0$. M is a constant c, or M is an abstraction $(\lambda x N)$. In either case $M = M'$ and M evals to M' at stage 1.

Inductive Step. M is a **combination**, say $(M_1 M_2)$. For $(M_1 M_2) \twoheadrightarrow M'$, a value, then it must be the case that $M_1 \xrightarrow{n_1} M_1'$, and $M_2 \xrightarrow{n_2} M_2'$, where M_1' and M_2' are values. By Fact 2, $(M_1 M_2) \xrightarrow{n_1} (M_1' M_2) \xrightarrow{n_2} (M_1' M_2')$. The proof now breaks down into two cases depending on what kind of value M_1' is.

1. $M_1' = \lambda x N$. Then
$$(M_1M_2) \xrightarrow{n_1+n_2} ((\lambda x N)M_2') \to (N[x := M_2']) \xrightarrow{n-(n_1+n_2+1)} M'.$$
By the inductive hypothesis then $eval(M_1) = \lambda x N$, $eval(M_2) = M_2'$, and $eval(N[x := M_2']) = M'$. Thus:
$$eval(M_1M_2) = eval(N[x := M_2']) = M'.$$

2. $M_1' = c$.
$$(M_1M_2) \xrightarrow{n_1+n_2} (cM_2) \xrightarrow{n_2} (cM_2') \to \ \mathbf{constapply}(c, M_2') \xrightarrow{n-(n_1+n_2+1)} M'$$
By the inductive hypothesis:
$$eval(M_1) = c,\ eval(M_2) = M_2',\ \text{and } eval(\mathbf{constapply}(c, M_2')) = M'.$$
Thus:
$$eval(M_1M_2) = eval(\mathbf{constapply}(c, M_2')) = M'$$

The case for when M is a conditional is left as an exercise. ■

Proof: **(M evals to M' at stage $t \Rightarrow M \twoheadrightarrow M'$)**. By induction on t.

Basis. $t = 1$. $M = M'$, and is either a constant or an abstraction. In either case $M \xrightarrow{0} N$ and we are done.

Inductive Step. $t > 1$. M is neither a constant nor an abstraction so it must be an application or conditional. We consider the case of an application, that of the conditional is left as an exercise. So $M = (M_1M_2)$.

M_1 must eval to a value at stage some $t_1 \leq t-2$. So say M_1 evals to M_1' at stage t_1. Then by the induction hypothesis $M_1 \twoheadrightarrow M_1'$, and then $(M_1M_2) \twoheadrightarrow (M_1'M_2)$. In addition M_2 must eval to a value at some stage $t_2 \leq t - (t_1 + 1)$. So say M_2 evals to M_2' at stage t_2. Then by the induction hypothesis $M_2 \twoheadrightarrow M_2'$, thus $(M_1'M_2) \twoheadrightarrow (M_1'M_2')$.

The analysis now breaks down into 2 cases based upon M_1'.

1. $M_1' = \lambda x N$. In this case $N[x := M_2]$ evals to M' at stage $t - (t_1 + t_2)$. But then
$$\begin{array}{rcl} M = (M_1M_2) & \twoheadrightarrow & (\lambda x N)M_2 \\ & \to & N[x := M_2] \end{array}$$
and by the inductive hypothesis $N'[x := M_2] \twoheadrightarrow M'$ and so $M \twoheadrightarrow M'$.

2. M_1' is a constant. Let $N = \mathbf{constapply}(M_1', M_2')$. By fact 3 we know that $(M_1'M_2') \to N$. Finally, N must eval to value M' at stage $t - (t_1 + t_2 + 1)$, thus by the induction hypothesis $N \twoheadrightarrow M'$ and more importantly, $M \twoheadrightarrow M'$.

■

References

[1] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[2] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

Supplemental Problems on FKS

Remark. The following supplemental problems are not required exercises. Nevertheless, in preparation for Quiz 4, you are strongly encouraged to attempt all problems. Solutions will be handed out before the quiz so that you may study them.

Problem 1. Recall the definition of the addition function *plus* given in lecture:

$$plus \overset{\text{df}}{=} Y(\lambda p.\lambda x.\lambda y.\mathsf{cond}\ y\ x\ (\mathsf{succ}\ (p\ x\ (\mathsf{pred}\ y))))$$

Using the rewrite rules, show all the steps in the evaluation of $((plus\ 31)\ 2)$.

Problem 2. Show all the steps of the SECD machine on the evaluation of $((plus\ 31)\ 1)$. That is, starting from the dump

$$[nil, \emptyset, ((plus\ 31)\ 1), nil]$$

show all steps of the SECD machine till it halts.

Problem 3. Give a rigorous proof that $((plus\ n)\ m)$, under the rewrite rules, evaluates to $n + m$, *i.e.*, $Eval(((plus\ 31)\ 1)) = n + m$. (Hint: Use induction on m. Note that this is a familiar fact, but is not obvious since the rewrite rules could be defined in a bizarre way.)

6.044J/18.423J: Computability, Programming, and Logic Handout 33
Massachusetts Institute of Technology 8 December 1989

Notes on Programming (Part III revised)

by Arthur Lent

This handout assumes that you are familiar with the material contained in handout #29 (Part I of these notes). In particular you need to know the definition of terms, FV, BV, the substitution operation, $=_\alpha$, and the rewrite rules.

1 The SECD Machine

Introduction. The SECD machine was first introduced by Landin in 1963[1]. The importance of the machine's introduction was that it was the first time that a language was described abstracted away from a particular implementation—in the past, a language was defined by the first compiler written for it. The SECD machine takes a role between the further abstraction of rewrite rules and the concreteness of an actual implementation—making explicit the control structure of evaluation, yet still leaving unspecified arbitrary details.

The treatment here is taken largely from a work by Plotkin in 1975[2].

What is a SECD Machine. SECD stands for Stack–Environment–Control-string–Dump. The precise technical definition of each of these terms will be given in the next section. First, we wish to cultivate some intuitions about the SECD machine. The SECD machine is an automaton whose basic actions are pointer manipulations. In fact, its basic actions involve a constant number of pointer manipulations. Once you see the full definitions it should be immediately clear how to built an interpreter for FKS. You should be able to do it in about an afternoon. You could not say the same thing about the rewrite rules presented in the first section of the notes. Not only does this description lend itself much more to being implemented, an implementation based directly on the SECD machine is far more efficient than one based upon the rewrite rules. Yet this does not mean that the rewrite rules are without value—they probably make a more understandable operational definition of FKS, and they will be essential for our work on denotational semantics.

Before we give technical definitions of the four components of the SECD machine we will tell what each piece will be used for:

Stack The stack will be used to store intermediate results in the evaluation of a function.

Environment The environment is the environment in which the function is being evaluated.

Controlstring The controlstring contains whatever operations still need to be performed in order to complete the current function call.

Dump The dump stores the state of the machine that existed immediately prior to the current function call. (It can be viewed as a stack of the preceding activation records).

Definitions. The first new concepts to be defined are the sets **Environments** and **Closures**. These correspond very closely to the notions defined in 6.001. In fact, they are the same notions, but, in a language without side effects, they have some additional nice properties. But first we should back up a step and address why the SECD machine needs environments and closures. It uses them in order to implement substitution. The process of turning an arbitrary term M into the term $M[x := N]$ is nontrivial. The traditional way of doing substitution is to say that the term $M[x := N]$ can represented by M paired with an *environment* an object that states that x really is N. M paired with this environment is called a closure, and it "represents" the term $M[x := N]$. Now what if we want to substitute $(M[x := N])$ for y into P to get the term $P[y := (M[x := N])]$. We want to represent this by the closure $[P, E]$ where $E(y) = (M[x := N])$. Unfortunately, we do not actually have our hands on the term $M[x := N]$— we have our hands on a closure representing it. Thus we do not really want environments to map from variables to terms, but, rather, we want them to map from variables to closures. This may look like a circular definition, but then so does defining two functions in a mutually recursive way. Here is a simultaneous mutual inductive definition of the set **Closures** of closures, the set **Value Closures** of value closures and **Environments** of environments:

- $\emptyset$, is an abbreviation for the totally undefined function. It is an environment.
- If E is an environment, and M of type σ such that $FV(M) \subseteq Domain(E)$ then $[M, E]$ is a closure of type σ. (In other words, E is defined on all of the free variables of M).
- If M is a value of type σ and $[M, E]$ is a closure then $[M, E]$ is a value closure of type σ.
- If Cl_i are value closures of types σ_i (for $i = 1, \ldots, n$), and $Dom(E) = \{x_i^{\sigma_i} | 1 \leq i \leq n\}$, and $E(x_i^{\sigma_i}) = Cl_i$ (for $i = 1, \ldots, n$) then E is an environment.

Note that any closed term M can be represented by the closure $[M, \emptyset]$.

We also define $E\{Cl/x\}$ (x and Cl must have the same type) to be the unique environment E' such that $E'(y) = E(y)$ if $y \neq x$ and $E'(x) = Cl$ (for any $Cl \in$ **Closures**).

Finally, to drive home the point that a closure can mechanically be "unwound" into the term it represents we define a function Realterm : **Closures** → **Terms**. It is defined inductively by:

$$\text{Realterm}([M, E]) = M[x_1 := \text{Realterm}(E(x_1))] \ldots [x_n := \text{Realterm}(E(x_n))]$$

where

$$FV(M) = \{x_1, \ldots, x_n\}$$

Note that this unwinding property is only possible when there are no side-effects in the terms inside the closures being unwound, so for Scheme it will not work, but there are many cases where closures *are* built up from terms without side-effects, in which case you really can think of those closures in terms of this unwinding process.

The set of stacks, **Stacks**, is the set of all finite sequences of closures, formally written as **Stacks** = (**Closures**)*.

The set of controlstrings, **Controlstrings** = (**Terms** ∪ ap, cd)* where ap, cd are special symbols that are not elements of **Terms**. The function FV is easily extended to work on controlstrings as follows:

- $FV(ap) = \emptyset$
- $FV(cd) = \emptyset$
- $FV(C_1, \ldots, C_n) = \bigcup_{i=1}^{n} FV(C_i)$ $(n \geq 0)$

Finally, the set of dumps, **Dumps**, is defined inductively by:

- $nil \in$ **Dumps**
- If $S \in$ **Stacks**, $E \in$ **Environments**, $C \in$ **Controlstrings** and C is such that $FV(C) \subseteq Domain(E)$, and $D \in$ **Dumps** then $[S, E, C, D] \in$ **Dumps**

This concludes the definitions of the primary data structures manipulated by the SECD machine.

The functions constapply and Constapply. The SECD machine model which we will present in the next section will be defined in a manner that abstracts away from the constants that we have chosen to include in FKS. Thus we could, in principle, add constants to the language (such as a curried plus operator) and the main proofs about the SECD machine would carry through directly. In addition this abstraction separates out the "constant stuff" for a particular language from the general principles of interpreting a functional language.

The SECD machine will employ the function **Constapply** when it hits an operator that is a constant. Since constant operators (namely Y) in our language can take values as arguments, and values are general terms which might need closures to fully define them, Constapply needs to be a partial function of the type:

$$\textbf{Constants} \times \textbf{Closures} \rightarrow \textbf{Value Closures}$$

We will also be presenting a recursive characterization of the rewrite rules which does not use stacks, closures, environments, or dumps, yet still captures explicitly the order of evaluation of the SECD machine. But for this recursive characterization, which we will from now on call *eval*, we still need a **Constapply** sort of function, but it must live wholly in the world of terms (no closures allowed). We will call it **constapply**, and it needs to be a partial function of type:

$$\textbf{Constants} \times \textbf{Closed Values} \rightarrow \text{Closed Terms}$$

In actuality we will not define **Constapply** directly. Instead we will define **constapply**, and then state that **Constapply** is as determined as it needs to be by the following restriction:

$$\text{Realterm}(\textbf{Constapply}(a, Cl)) =_\alpha \textbf{constapply}(a, \text{Realterm}(Cl))$$

What this requires is that **Constapply** gives a result that is independent of how the closure that is its argument represents the term it is acting upon. So **Constapply** cannot distinguish between:

$$[x, \{[x, (MN)]\}]$$

and

$$[(x\ y), \{[x, M], [y, N]\}]$$

Here is the definition of **constapply** which we will use for FKS:

succ	$\mathbf{constapply}(\text{succ}, \boldsymbol{n})$	$\longrightarrow$	$\boldsymbol{n} + \mathbf{1}$
pred	$\mathbf{constapply}(\text{pred}, \boldsymbol{n} + \mathbf{1})$	$\longrightarrow$	$\boldsymbol{n}$
	$\mathbf{constapply}(\text{pred}, \mathbf{0})$	$\longrightarrow$	$\mathbf{0}$
Y_σ	$\mathbf{constapply}(Y_\sigma, V)$	$\longrightarrow$	$(V(\lambda x^{(\sigma\rightarrow\sigma)}.(Y_\sigma V)x))$ (for an $x \notin FV(V)$ and for V a value and for $\sigma \neq \iota$)

Order of evaluation. The SECD machine will evaluate the operands of a combination before the operators. This is to be contrasted with the order of the rewrite rules which evaluate operators before operands. We apologize for this confusion, and it should be obvious how to change the rewrite rules or the SECD machine so that the order is reversed. Our main theorem, however, carries through whichever order used in whichever scheme. This is because, for FKS, the order does not matter (in fact for a particular scheme expression, where the evaluating the operator or operands does not produce any side effects, the order does not matter). Note that the definition of Scheme does not even specify which is the correct order. The following is an exerpt from the Scheme manual given to 6.001 students in Spring '89:

"A procedure call is written by enclosing in parenthesis expressions for the procedure to be called and the arguments to be passed to it. The operator and operand expressions are evaluated (**in an indeterminate order**) and the resulting procedure is passed the resulting arguments... Procedure calls are also called *combinations*" (emphasis added).

The function SECD. The state transition function, a partial function from **Dumps** to **Dumps** is defined as follows:

1. $[Cl : S, E, nil, [S', E', C', D']] \Rightarrow [Cl : S', E', C', D']$
2. $[S, E, x : C, D] \Rightarrow [E(x) : S, E, C, D]$
3. $[S, E, a : C, D] \Rightarrow [[a, \emptyset] : S, E, C, D]$
4. $[S, E, (\lambda x.\ M) : C, D] \Rightarrow [[(\lambda x.\ M), E] : S, E, C, D]$
5. $[[(\lambda x.\ M), E'] : Cl : S, E, ap : C, D] \Rightarrow [nil, E'\{Cl/x\}, M, [S, E, C, D]]$
6. $[[a, \emptyset] : [V, E''] : S, E, ap : C, D] \Rightarrow [nil, E', M', [S, E, C, D]]$
   (where **Constapply**$(a, [V, E'']) = [M, E']$)
7. $[S, E, (MN) : C, D] \Rightarrow [S, E, N : M : ap : C, D]$

8. $[S, E, (\text{cond } M N_1 N_2) : C, D] \Rightarrow [[N_1, E] : [N_2, E] : S, E, M : cd : C, D]$
9. $[[\mathbf{o}, E_0] : [N_1, E_1] : [N_2, E_2] : S, E, cd : C, D] \Rightarrow [S, E_1, N_1 : C, D]$
10. $[[[\boldsymbol{n} + \mathbf{1}, E_0] : [N_1, E_1] : [N_2, E_2] : S, E, cd : C, D] \Rightarrow [S, E_2, N_2 : C, D]$

We now need two functions **Load** and **Unload** which convert terms into SECD machine state, and SECD machine state into terms. Specifically they are defined by:

$$\mathbf{Load}(M) = [nil, \emptyset, M, nil]$$

$$\mathbf{Unload}([Cl, \emptyset, nil, nil]) = \text{Realterm}(CL)$$

We can now define an evaluation function, which is a partial function from terms to values as follows:

$$\text{SECD}(M) = V \text{ iff } \mathbf{Load}(M) \overset{*}{\Rightarrow} D, \text{ and } V = \mathbf{Unload}(D) \text{ for some dump D}$$

The punchline of the section on SECD machines is the following theorem:

Theorem 1. SECD(M)$=_\alpha N$ iff $Eval(M)=_\alpha N$ for all terms M and N.

Remember that M and N are $=_\alpha$ iff they differ only in the names of their bound variables—called α-equivalence or *equal up to renaming of bound variables.*

In order to prove this theorem we will introduce another scheme for evaluating programs that is midway between the rewrite rules and the SECD machine. This will be a simple inductive definition that uses substitution rather than closures.

We introduce the binary relation $\boldsymbol{eval_{rel}}$on closed terms:

- $\boldsymbol{eval_{rel}}(V, V)$ for V a value.
- if $\boldsymbol{eval_{rel}}(M_1, \lambda x.M)$ and $\boldsymbol{eval_{rel}}(N_1, N_2)$ and $\boldsymbol{eval_{rel}}(M[x := N_2], L)$ then $\boldsymbol{eval_{rel}}((M_1, N_1), L)$
- if $\boldsymbol{eval_{rel}}(M_1, c)$ and $\boldsymbol{eval_{rel}}(N_1, N_2)$ and $\boldsymbol{eval_{rel}}(\mathbf{constapply}(c, N_2)), L)$ then $\boldsymbol{eval_{rel}}((M_1, N_1), L)$
- if $\boldsymbol{eval_{rel}}(M, \mathbf{o})$ and $\boldsymbol{eval_{rel}}(N_1, N_1')$ then $\boldsymbol{eval_{rel}}((\text{cond } M\ N_1\ N_2), N_1')$
- if $\boldsymbol{eval_{rel}}(M, \boldsymbol{n} + \mathbf{1})$ and $\boldsymbol{eval_{rel}}(N_2, N_2')$ then $\boldsymbol{eval_{rel}}((\text{cond } M\ N_1\ N_2), N_2')$

The following two facts about $\boldsymbol{eval_{rel}}$can be proven by induction on its definition:

(a) $\boldsymbol{eval}_{rel}$is the graph of a function

(b) If $\boldsymbol{eval}_{rel}(M, N)$ then N is a value

Thus we can make the following definition of a partial function *eval*:

$$eval(M) \overset{\text{def}}{=} \text{ the unique } V\text{, if any, such that } \boldsymbol{eval}_{rel}(M, V)$$

Note that this function *eval*is the same function as the Metacircular Evaluator in scheme. Only here we have give a rigorous mathematical definition of *eval*—which the 6.001 definition is not.

Neverless you can check that *eval*does "work like" the metacircular environment in that it satisfies the equations:

$$eval(a) = a; \quad eval(\lambda x M) = \lambda x M$$

$$eval(MN) = \begin{cases} eval(M'[x := N']) & (\text{if } eval(M) = \lambda x M' \\ & \text{and } eval(\text{N})\text{=N'}) \\ eval(\mathbf{constapply}(a, N')) & (\text{if } eval(M) = a \\ & \text{and } eval(N) = N') \end{cases}$$

$$eval(\text{cond } M\ N_1\ N_2) = \begin{cases} eval(N_1) & (\text{if } eval(M) = \mathbf{o}) \\ eval(N_2) & (\text{if } eval(M) = \boldsymbol{n} + \mathbf{1}) \end{cases}$$

From the definition of $\boldsymbol{eval}_{rel}$ and *eval*, it should be clear that we could define the predicate "M evals to N at stage t". Where this simply captures the notion that determining $\boldsymbol{eval}_{rel}(M, N)$ takes exactly t steps. An induction on t for $\boldsymbol{eval}_{rel}(M, N)$ is called an "induction on the length of the derivation". So concluding $\boldsymbol{eval}_{rel}(V, V)$ for V a value takes one step. For all of the inductive cases, if it takes t steps to establish all of the hypotheses, then it takes $t+1$ steps to establish the consequents. Thus, for example, if M evals to (0) in t_1 steps, and N_1 evals to L in t_2 steps, then (cond M N_1 N_2) evals to L in $t_1 + t_2 + 1$ steps.

Proving the equivalence of SECD and *eval*. In before we prove Theorem 1 we first prove the following, easier Theorem (notice the little "e"):

Theorem 2. SECD(M)$=_\alpha N$ iff $eval(M) =_\alpha N$ for all terms M and N.

This theorem will be proven using three lemmas. The first says using closures and environments to model substitution "works right". The second will prove direction ⇒ of this theorem, and the third will prove direction ⇐ of this theorem.

Lemma 1. Suppose $[\lambda y.\ M, E]$ and $[N, E']$ are value closures. Also suppose that Realterm($[\lambda y.\ M, E]$)$=_\alpha(\lambda x.M')$ and

Realterm($[N, E']$)$=_\alpha N'$. Then Realterm($[M, E\{[N, E']/y\}]$)$=_\alpha M'[x := N']$.

Proof Sketch: Observe that if $\lambda y.\ M=_\alpha\lambda x.\ M'$ then $M=_\alpha M'[x := y]$, hence $M[y := N]=_\alpha M'[x := N]$. The rest is a simple unwinding of the closures and simply examining the definition of Realterm. ■

The proof of this next Lemma captures how the SECD machine really works. It is quite long, however, thus we will leave out the details of a few of the cases.

Lemma 2. Suppose E is an environment and $[M, E]$ is a closure. Suppose Realterm($[M, E]$) evals to M''. Suppose C is a controlstring with $FV(C) \subseteq Domain(E)$. Then there is a $t' \geq t$, such that for all S, D,

$$[S, E, M : C, D] \overset{t'}{\Rightarrow} [[M', E'] : S, E, C, D]$$

where $[M', E']$ is a value closure and Realterm($[M', E']$)$=_\alpha M''$.

This Lemma really does entail the right hand direction of Theorem 2, but requires in its statement a rather hefty induction hypothesis.

Proof: This is a proof by induction on t. It is quite similar to that presented by Plotkin [2]. There are 5 main cases:

1. M is a constant. Here Realterm($[M, E]$) $= M = M''$ and $t = 1$. As
$$[S, E, M : C, D] \Rightarrow [[M, \emptyset] : S, E, C, D]$$
we can take $[M', E'] = [M, \emptyset]$ and $t' = 1$.

2. M is a λ-abstraction. Almost the same as the previous case.

3. M is a variable. Take $[M', E'] = E(M)$, and $t = 1$.

4. $M =$ (cond P N_1 N_2) is a conditional. Apply the inductive hypothesis to P and then divide by cases according to P' the value that P evals to at stage t_1

5. $M = (M_1\ M_2)$ is a combination. Then
$$\begin{aligned} \text{Realterm}([M, E]) &= (\text{Realterm}([M_1, E])\ \text{Realterm}([M_2, E])) \\ &= (N_1\ N_2) \text{ say.} \end{aligned}$$
This now divides into two subcases, depending on whether or not the value to which N_1 evals to is a λ-abstraction or a constant.

(a) $(\lambda x.\ N_3)$ is the value that N_1 evals to at stage t_1, N_4 is the value that N_2 evals to at stage t_2, M'' is the value that $N_3[x := N_4]$ evals to at stage t_3 and $t = t_1 + t_2 + t_3 + 1$.
Then by the induction hypothesis there are $t'_i \geq t_i$ $(i = 1,\ 2)$ such that:

$$\begin{aligned} [S, E, (M_1\ M_2) : C, D] &\Rightarrow [S, E, M_2 : M_1 : ap : C, D] \\ &\overset{t'_1}{\Rightarrow} [[M'_2, E'_2] : S, E, M_1 : ap : C, D] \\ &\overset{t'_2}{\Rightarrow} [[M'_1, E'_2] : [M'_2, E'_2] : S, E, ap : C, D] \end{aligned}$$

where

$$\text{Realterm}([M_1, E_1]) =_\alpha (\lambda x.\ M) \text{ and } \text{Realterm}([M'_2, E'_2]) =_\alpha N_4,$$

and the $[M'_i, E'_i]$ are value closures.
Here $M'_1 = (\lambda y.\ M'_3)$ for some M'_3, and

$$\text{Realterm}([M'_3, E'_1\{[y := [M'_2, E'_2]\}]) =_\alpha [N_4/x]N_3 \text{ (by Lemma 1).}$$

Now,

$$\begin{aligned} [[M'_1, E'_1] :\ & [M'_2, E'_2] : S, E, ap : C, D] \\ &\Rightarrow [nil, E'_1\{[M'_2, E'_2]/y\}, M'_3, [S, E, C, D]] \\ &\overset{t'_3}{\Rightarrow} [[M', E'], E'_1\{[M'_2, E'_2]/y\}, nil, [S, E, C, D]] \\ &\Rightarrow [[M', E'] : S, E, C, D] \end{aligned}$$

where, by the induction hypothesis, Realterm($[M', E']$) is to within α-equivalence the value that Realterm($[M'_3, E'_1\{[M'_2, E'_2]/y\}]$) evals to at stage $t_3 \leq t'_3$ and $[M', E']$ is a value closure. Taking $t' = t'_1 + t'_2 + t'_3 + 3$ concludes this subcase.

(b) a is the value that N_1 evals to at stage t_1, V is the value that N_2 evals to at stage t_2. Then, by the inductive hypothesis there are $t'_i \geq t_i$ (i=1, 2) , and a value closure VC such that:

$$\begin{aligned} [S, E, (M_1\ M_2), C, D] &\Rightarrow [S, E, M_2 : M_1 : ap : C, D] \\ &\overset{t'_2}{\Rightarrow} [VC : S, E, M_1 : ap : C, D] \\ &\overset{t'_1}{\Rightarrow} [[a, \emptyset] : VC : S, E, ap : C, D] \end{aligned}$$

where Realterm(VC) $= V$. Now, finally, suppose that we have **Constapply**$(a, VC) = [M'', E'']$, and N'' is the value to which Realterm($[M'', E'']$) evals at stage t_3 (thus N'' is the value to which

constapply$(a, \text{Realterm}(VC))$ evals at stage t_3). By the induction hypothesis there are t'_3 and VC' such that:

$$\begin{array}{rcl} [S, E, (M_1\ M_2), C, D] & \overset{t'_1+t'_2+1}{\Rightarrow} & [[a, \emptyset] : VC : S, E, ap : C, D] \\ & \Rightarrow & [nil, E'', M'', [S, E, C, D]] \\ & \overset{t'_3}{\Rightarrow} & [VC', E'', nil, [S, E, C, D]] \\ & \Rightarrow & [VC' : S, E, C, D] \end{array}$$

where $\text{Realterm}(VC') = N''$. Then taking $t' = t'_1 + t'_2 + t'_3 + 3$ and $[M', E'] = VC'$ concludes the proof of the lemma.

■

Before we introduce the next lemma, we need a definition. If $D \overset{t}{\Rightarrow} D'$, where D' does not have the form $[Cl, \emptyset, nil, nil]$ and $D' \not\Rightarrow D''$ for any D'' then D is said to *hit an error state* (viz. D').

Lemma 3. Suppose E is a value environment and $[M, E]$ is a closure. If $\text{Realterm}([M, E])$ does not eval to a value at any $t' \leq t$, then either for all S, C, D, with $FV(C) \subseteq Domain(E)$, $[S, E, M : C, D]$ hits an error state or else $[S, E, M : C, D] \overset{t}{\Rightarrow} D'$ for some D'.

Proof Sketch: This is proved by induction on t—the number of steps used by the SECD machine to evaluate M. It is just a horrible counting exercise that can just be grunged through. ■

Proof: **(Theorem 2).** Suppose $eval(M) = M''$. Then at some stage t, M'' is the value that M evals to at stage t. By lemma 2,

$$[nil, \emptyset, M, nil] \overset{t'}{\Rightarrow} [[M', E'], \emptyset, nil, nil],$$

where $\text{Realterm}([M', E']) =_\alpha M''$. So $Eval(M) =_\alpha M''$.

Suppose, on the other hand, that M does not eval to a value at any stage. Then by Lemma 3 either $[nil, \emptyset, M]$ hits an error state or else for every t there is a D such that $[nil, \emptyset, M, nil] \overset{t}{\Rightarrow} D$. In either case SECD($M$) is also not defined. ■

Proving the equivalence of eval and Eval.

Theorem 3. For all well-typed, closed terms M with constants in **Constants** then $M \twoheadrightarrow M'$ (M' a value) iff M evals to M' at some stage t ($eval(M) = M'$).

But first we need several facts:

Fact 1. $\rightarrow$ is deterministic. That is: if $M \rightarrow M'$ then $\nexists M'' \neq M'$ such that $M \rightarrow M''$. Thus if $M \xrightarrow{n} M''$, $M \xrightarrow{m} M'$ and $m \leq n$ then $M' \xrightarrow{n-m} M''$.

Fact 2. If $M_1 \xrightarrow{n} M_1'$ then $(M_1 M_2) \xrightarrow{n} (M_1' M_2)$ and $(a M_1) \xrightarrow{n} (a M_1')$

Fact 3. If M is a closed value, then $(cM) \rightarrow \mathbf{constapply}(c, M)$ which is to say that if **constapply**(c,M) is defined then (cM) reduces to it, and if **constapply**(c,M) is not defined then $\nexists M' : (cM) \rightarrow M'$.

Proof: $(M \xrightarrow{n} M' \Rightarrow \boldsymbol{eval}(M) = M')$. By induction on n.

Basis. $n = 0$. M is a constant c, or M is an abstraction $(\lambda x N)$. In either case $M = M'$ and M evals to M' at stage 1.

Inductive Step. M is a **combination**, say $(M_1 M_2)$. For $(M_1 M_2) \twoheadrightarrow M'$, a value, then it must be the case that $M_1 \xrightarrow{n_1} M_1'$, and $M_2 \xrightarrow{n_2} M_2'$, where M_1' and M_2' are values. By Fact 2, $(M_1 M_2) \xrightarrow{n_1} (M_1' M_2) \xrightarrow{n_2} (M_1' M_2')$. The proof now breaks down into two cases depending on what kind of value M_1' is.

1. $M_1' = \lambda x N$. Then

   $$(M_1 M_2) \xrightarrow{n_1+n_2} ((\lambda x N) M_2') \rightarrow (N[x := M_2']) \xrightarrow{n-(n_1+n_2+1)} M'.$$

   By the inductive hypothesis then $eval(M_1) = \lambda x N$, $eval(M_2) = M_2'$, and $eval(N[x := M_2']) = M'$. Thus:

   $$eval(M_1 M_2) = eval(N[x := M_2']) = M'.$$

2. $M_1' = c$.

   $$(M_1 M_2) \xrightarrow{n_1+n_2} (c M_2) \xrightarrow{n_2} (c M_2') \rightarrow \ \mathbf{constapply}(c, M_2') \xrightarrow{n-(n_1+n_2+1)} M'$$

   By the inductive hypothesis:

   $$eval(M_1) = c,\ eval(M_2) = M_2',\ \text{and } eval(\mathbf{constapply}(c, M_2')) = M'.$$

   Thus:

   $$eval(M_1 M_2) = eval(\mathbf{constapply}(c, M_2')) = M'$$

The case for when M is a conditional is left as an exercise. ■

Proof: (**M evals to M' at stage $t \Rightarrow M \twoheadrightarrow M'$**). By induction on t.

Basis. $t = 1$. $M = M'$, and is either a constant or an abstraction. In either case $M \xrightarrow{0} N$ and we are done.

Inductive Step. $t > 1$. M is neither a constant nor an abstraction so it must be an application or conditional. We consider the case of an application, that of the conditional is left as an exercise. So $M = (M_1 M_2)$.

M_1 must eval to a value at stage some $t_1 \leq t-2$. So say M_1 evals to M_1' at stage t_1. Then by the induction hypothesis $M_1 \twoheadrightarrow M_1'$, and then $(M_1 M_2) \twoheadrightarrow (M_1' M_2)$. In addition M_2 must eval to a value at some stage $t_2 \leq t - (t_1 + 1)$. So say M_2 evals to M_2' at stage t_2. Then by the induction hypothesis $M_2 \twoheadrightarrow M_2'$, thus $(M_1' M_2) \twoheadrightarrow (M_1' M_2')$.

The analysis now breaks down into 2 cases based upon M_1'.

1. $M_1' = \lambda x N$. In this case $N[x := M_2]$ evals to M' at stage $t - (t_1 + t_2)$. But then
$$\begin{array}{rcl} M = (M_1 M_2) & \twoheadrightarrow & (\lambda x N) M_2 \\ & \rightarrow & N[x := M_2] \end{array}$$
and by the inductive hypothesis $N'[x := M_2] \twoheadrightarrow M'$ and so $M \twoheadrightarrow M'$.

2. M_1' is a constant. Let $N = \textbf{constapply}(M_1', M_2')$. By fact 3 we know that $(M_1' M_2') \rightarrow N$. Finally, N must eval to value M' at stage $t - (t_1 + t_2 + 1)$, thus by the induction hypothesis $N \twoheadrightarrow M'$ and more importantly, $M \twoheadrightarrow M'$.

■

References

[1] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[2] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

Notes on Programming (Part IV)

by Arthur Lent

1 Models

Models. In general, a model is a method of assigning meaning to terms. We would like to pick a model such that the desired equations between terms hold (in the case of FKS we desire that if M is a program and $Eval(M)$ = n then our model assigns the meaning of M to n). There will be two levels of meaning in or model so that we can accomplish two different tasks—we wish to assign a meaning even to terms with free variables, and we want to be able to obtain a meaning for closed terms and for open terms+an assignment of meanings to its free variables. This will be done by having a "meaning function" $[\![\cdot]\!]$ that map the syntactic entities of terms into semantic objects. These semantic objects will be functions which take environments (maps from variables to the domain of the model) and return elements of the domain. For closed terms, this function will be a constant function. Moreover, we would like our semantics to be such that the denotation of a closed term M of type ι is a constant function that returns the natural number which the syntactic object M represents. In addition, for terms M of higher type we would map to bring us to **the unique function** which the term M computes. So the whole motivation behind this is to give a mathematical precision behind the notaion of a piece of code "computing" a certain function. So, for example, for our semantics to be satisfactory we want the meaning of a closed term M of type $\iota \rightarrow \iota$ to literally be the partial recursive function f iff $Eval((Mn)) = f(n)$. In order for this to work out our model will have some properties which may not be familiar to you.

It is important that our model properties:

- Can give meaning to numerals, succ, pred, Y and conditionals.
- Is closed under λ-definability. That is, if I have a meaning for a term with free variable x, and I abstract over x to obtain a new function, that function must be in the model.
- Is closed under application.

Given a particular set of functions and elements, it is not obvious that it has these properties.

We wll now be more precise about the elements and functions of the *particular* model we shall study. We define the collection $\{D^\sigma\}$ of sets (for any type σ) inductively as follows:

- $D^{\iota} = \mathbb{N}$

- $D^{\sigma\rightarrow\tau} = D^{\sigma} \rightharpoonup_c D^{\tau}$

where $A \rightharpoonup_c B$ is the set of partial continuous functions from A to B.

At this point, the reader is not expected to understand the definition of $D_{\sigma\rightarrow\tau}$. The next few pages will explain precisely these sets of functions.

Our sets will be ordered by the partial order $\sqsubseteq$. Remember that a partial order is a relation that is reflexive, transitive, and anti-symmetric (if $a \sqsubseteq b$ and $b \sqsubseteq a$ then $a = b$). We will write $a \sqsupseteq b$ for $b \sqsubseteq a$. $\sqsubseteq$ will be ordering objects based upon *information content.* So any two natural numbers will be *incomparable* since no natural number has any more or less information contained in it than any other, yet they have distinct information. Consider a function f which agrees with g on all arguments for which g is defined, but is also defined on more arguments; f has a greater information content than g, and its information is compatible with that of g. But if for even one argument, both f and g were defined, but took on different values, then their information content would be incomparable.

We will provide a definition of $\sqsubseteq$ by induction on types. Note that it only makes sense to ask the question $a \sqsubseteq b$ if a and b are elements of the same set; to do otherwise is a "type error". We now formally define $\sqsubseteq$ by induction on types (*i.e.*, we define $\sqsubseteq$ for elements of D^{ι} then we define $\sqsubseteq$ for elements of $D^{(\sigma\rightarrow\tau)}$ assuming $\sqsubseteq$ is defined for elements of D^{σ} and D^{τ}). So, $a \sqsubseteq b$ (a, $b \in D^{\kappa}$) iff:

- Case of $\kappa = \iota$. $D^{\iota} = \mathbb{N}$ ordered discretely, we have $a \sqsubseteq b$ iff $a = b$. This is what we mean when we say $\mathbb{N}$ is *ordered discretely.* We forget the normal ordering on $\mathbb{N}$ (namely $\leq$) and use this new ordering ($\sqsubseteq$) under which two distinct natural numbers are incomparable (*e.g.* if $a \neq b$ then $a \not\sqsubseteq b$ and $b \not\sqsubseteq a$).

- Case of $\kappa = \sigma \rightarrow \tau$. So, we are comparing two functions from D^{σ} to D^{τ}. $f \sqsubseteq g$ iff for all $d \sqsubseteq e$, d, e $\in D^{\sigma}$ then either $f(d)$ is undefined or $f(d) \sqsubseteq g(e)$.

Note we will use $a \sqsubset b$ as an abbreviation for ($a \sqsubseteq b$ and $a \neq b$); $\sqsupset$ is defined similarly.

Example. Consider the following infinite set of functions:

$$\begin{array}{lcl}
h_0 & \overset{\text{def}}{=} & \text{a function that is undefined on all arguments} \\
h_1 & \overset{\text{def}}{=} & h_1(0) = 0!, h_1(x) \text{ undefined for } x \geq 1 \\
h_2 & \overset{\text{def}}{=} & h_2(0) = 0!, h_2(1) = 1!, h_2(x) \text{ undefined for } x \geq 2 \\
h_3 & \overset{\text{def}}{=} & h_3(0) = 0!, h_3(1) = 1!, h_3(2) = 2!, h_3(x) \text{ undefined for } x \geq 3 \\
 & & \dots \\
h_n & \overset{\text{def}}{=} & h_n(0) = 0! \dots h_2(n-1) = (n-1)!, \text{ undefined for } x \geq n \\
 & & \dots
\end{array}$$

These are all elements of $D^{(\iota \to \iota)}$. You can easily check that:

$$h_0 \sqsubset h_1 \sqsubset h_2 \sqsubset \dots$$

Definition 1. Now we define the *least upper bound* (LUB) of a set $X \subseteq D^\sigma$ for some σ (written $\sqcup X$). We say $d = \sqcup X$ (if it exists) is uniquely if d has the following two properties:

d is an upper bound For all $d' \in X$, $d' \sqsubseteq d$.

d is least For all $d' \neq d$ in D^σ such that d' is an upper bound on X, $d \sqsubset d'$.

Note that not all sets X have an least upper bound (written LUB).

Note the following interesting fact:

Fact 1.

$$\bigsqcup_{i \geq 0} h_i = \textbf{Fact} \text{ the factorial function.}$$

Proof: We know that **Fact** is an upper bound on the set $\{h_i\}$. Suppose f is also an upper bound on the set $\{h_i\}$. Then show that $\textbf{Fact} \sqsubseteq f$. Why? f must be total, since **Fact** is. Why else? $\textbf{Fact}(n) \sqsubseteq f(n)$ since $h_n(n) = \textbf{Fact}(n)$, for all n, and $h_n(n) \sqsubseteq f(n)$ (note that if $A = B$ and $A \sqsubseteq C$ then $B \sqsubseteq C$). But then $\textbf{Fact} \sqsubseteq f$. ■

Example. We now show that there exist sets $X \in D^\sigma$ such that X has no upper bound. Such sets are found throughout almost all of the D^σ's.

We take $\sigma = \iota \to \iota$ and $X = \{f, g\}$ where $f(x) = x!$ and $g(x) = x! + 1$. X has no upper bound. Why not? First, $f \neq g$. But f and g are maximally defined. Thus there are no elements k of $D^{(\iota \to \iota)}$ such that $f \sqsubset k$ or $g \sqsubset k$. But if k were an upper bound of X then it would have to be the case that $f = k = g$. Since $f \neq g$, X has no upper bound (much less a LEAST upper bound).

There also exist sets $X \in D^\sigma$ such that X has an upper bound, but no least upper bound.

Definition 2. A subset X of D^σ is *directed* iff:

> For every pair of elements w, $y \in X$ there is a $z \in X$ such that: $w \sqsubseteq z$ and $y \sqsubseteq z$.

Fact 2. The set $\{h_i\}$ is directed.

Proof: Consider h_i, h_j, and withour loss of generality, assume $j \geq i$. Then $h_i \sqsubseteq h_j$, $h_j \sqsubseteq h_j$. So, h_j is an upper bound. ■

Definition 3. A *complete partial order (cpo)* is a pair $[D, \sqsubseteq]$ (where D is any set, and $\sqsubseteq$ is a partial order on that set), that meets the following additional property:

$$\bigsqcup X \in D \text{ (for all directed } X \subseteq D)$$

Note that each of $[D^\sigma, \sqsubseteq]$, for all types σ, is a cpo.

Definition 4. A partial function $f : D^\sigma \rightarrow D^\tau$ is *continuous* iff $f(\sqcup X) = \sqcup\{f(x) | x \in X\}$.

Fact 3. The function f whose definition is:

$$f(k) = m \text{ where } m(x) = \begin{cases} 1 & (\text{if } x = 0) \\ x * (k(x-0)) & \text{otherwise} \end{cases}$$

is a partial continuous function.

We now have enough definitions in order to fully understand the definition of our type frame.

The meaning function: $[\![\cdot]\!]$. Here is one more definition which will help in defining our "meaning" function.

We use the notation $f \simeq g$, for expressions f and g, to be true iff either both f and g are undefined, or they are both defined and have the same value. We also write $f \bullet d$ to mean "undefined" if either f or d do not exist, and $f(d)$ otherwise.

There is a two step process to map a term into an element of the domain of our model. The first step is to apply the meaning function ($[\![\cdot]\!]$) to the term. Because an arbitrary term might have free variables, it is not possible for the meaning function to assign a term directly to an element of the model. Instead the meaning function maps a term to a functional which takes an environment as argument. This functional then uses the environment to find the values

$$
\begin{array}{lcl}
[\![\mathbf{cst}]\!]\rho & \simeq & cst \text{ for } cst = 0, 1, \ldots, \text{succ}, \text{pred—but not } Y \\
[\![x]\!]\rho & \simeq & \rho(x) \\
[\![(M\ N)]\!]\rho & \simeq & [\![M]\!]\rho \cdot [\![N]\!]\rho \\
[\![cond\ M N_1 N_2]\!]\rho & \simeq & \left\{ \begin{array}{l} [\![N_1]\!]\rho \text{ if } [\![M]\!]\rho = 0 \\ [\![N_2]\!]\rho \text{ if } [\![M]\!]\rho = n+1 \end{array} \right. \\
[\![\lambda x^\sigma .\ M]\!]\rho & = & f \quad \text{where} f(d) \simeq [\![M]\!]\rho[x^\sigma := d] \\
[\![Y]\!]\rho \cdot d & \simeq & \sqcup_{n \geq 0} f_n \\
 & \text{where} & f_0 \text{ is the totally undefined function} \\
 & & f_{n+1} = d(F_n)
\end{array}
$$

Figure 1: The definition of $[\![\cdot]\!]$ for FKS

of the original term's free variables, and can then return a unique element of the domain of our model. Thus $[\![\cdot]\!]$ is a partial function of type: **Terms** $\longrightarrow$ (**Environments** $\longrightarrow D^\sigma$), where a term of type σ is mapped to an element of D^σ. Environments (written as ρ) are from **Variables** to D that are total on their domain. The meaning of the term M in environment ρ is written as follows:

$$[\![M]\!]\rho$$

In order for this to make sense, however, we need the following condition:

$$\rho(x^\sigma) \in D^\sigma \text{ defined for all } x^\sigma \in FV(M)$$

Thus environment ρ is total on its domain, which is the free variables of the term for which it is being used. We need our meaning to obey certain constraints. These constraints are defined in Figure 1.

By looking at this definition of $[\![\cdot]\!]$ we can see from where some of the constraints upon our model have arisen. It is closed under application, since $f \in D^{\sigma \rightarrow \tau}$ must be a partial continuous function from D^σ to D^τ. This simply follows from the fact that $Range(f) \subseteq D^\tau$. Closure under λ-definability is a much more subtle issue. We know that in the case mentioned above, such an f exists. It is even a function from D^σ to D^τ; however, it is not at all obvious that f is continuous. Well, it is, but it is beyond the scope of these notes to justify that statement.

Our last desire about this model in general is that the meaning which we have given to Y really is that of a least fixed point operator. Trust us, it is.

2 Soundness of the Model

A property that is absolutely essential about any model of anything is that reasoning in the model be *sound.* For our partial function model of the language FKS, this soundness takes the form of the following theorem:

Theorem 1. (Soundness) For any program M, and constant k, $Eval(M) = k$ implies $[\![M]\!](\rho) = k$.

So soundness says that if programs M, M' evaluate to the same value then they have the same denotation.

3 Adequacy of the Model

The adequacy is a statement of a sort of completeness of the model. It is complete for knowing how programs in isolation will behave. The formulation of adequacy which we will prove is as follows:

Theorem 2. (Adequacy) For any program M, and constant k, $Eval(M) = k$ iff $[\![M]\!](\bot) = k$.

Proof Sketch: The direction $\Rightarrow$ is simply soundness. The direction $\Leftarrow$ will be proving that divergent programs do not denote anything (their denotation is undefined). ■

A direct corollary of adequacy is the following:

Corollary 1. If $[\![M]\!]\rho \simeq [\![N]\!]\rho$ for all environments ρ then $M \equiv_{obs} N$

Proof: Well, suppose not. In other words $[\![M]\!]\rho \simeq [\![N]\!]\rho$ for all environments ρ, yet $M \not\equiv_{obs} N$. Let $C[\cdot]$ be a program context of which can distinguish between M and N. *i.e.* $Eval(C[M]) \neq Eval(C[N])$. But by our adequacy theorem, this implies that $[\![C[M]]\!](\bot) \neq [\![C[N]]\!](\bot)$. But our hypothesis ($[\![M]\!]\rho \simeq [\![N]\!]\rho$ for all environments ρ) implies $[\![C[M]]\!](\bot) = [\![C[N]]\!](\bot)$. Contradiction. ■

It is this adequacy theorem that we have been looking for in terms of the usefulness of our semantic model. Given adequacy, we now have a powerful piece of machinery that enables us to prove that two pieces of code are completely interchangeable. This is quite a robust notion. Unfortunately, this notion is not quite as robust a notion as we might like. In particular, we might wish also to have the converse—namely that if $M \equiv_{obs} N$ then M and N have the same denotation. When you have both adequacy and its converse, the semantics for the language is termed to be *fully abstract*. What full abstraction gives you is: if two terms are interchangeable, then they have the same denotation. Moreover, when you have fully abstract semantics, if you can prove that two terms do not have the same denotation, then the code they represent is not completely interchangeable.

6.044J/18.423J: Computability, Programming, and Logic — Handout 35
Massachusetts Institute of Technology — 8 December 1989

Problem Set 8 Solutions

Grades. The grades went as follows:

	number submitted	min	max	mean	median
1	8	15	25	21.5	23.5
2	9	10	25	21.7	25
3	9	5	25	21.7	25

Problem 1. Let Γ be a set of quantifier-free sentences which contain no function symbols and do not contain "$=$". Give a simplified proof from scratch that if every finite subset of Γ is satisfiable, *i.e.*, Γ is o.k., then Γ is satisfiable. (Hint: Rework and simplify the proof of Lemma 3 from the proof of the Completeness Theorem; the proof here should be easier, since the only terms to consider are names and there are no equality constraints.)

Solution. The point of this problem is to see that equivalence classes, and the extra headaches that go along with them, can be eliminated from the proof of Lemma 3 under certain conditions.

We start the same way as we did in the original proof. Let $T = c_1, c_2, \ldots$ be the constants appearing in Γ; since we have no function symbols, these are the *only* terms. Also, let $A_1, A_2, \ldots$ be the atomic formulas constructed from terms in T and predicates in Γ—we do not consider formulas with equality, however.

Next, let $\Gamma_1 = \Gamma$, and let

$$\Gamma_{n+1} = \begin{cases} \Gamma_n \cup \{A_n\} & \text{if o.k.} \\ \Gamma_n \cup \{\neg A_n\} & \text{otherwise} \end{cases}$$

By a fact proved in lecture, each Γ_i is o.k. Finally, let B_i be whichever of A_i or $\neg A_i$ is in Γ_{i+1}.

Now define an interpretation $\mathcal{I}$ that matches Γ by the following:

- $D_{\mathcal{I}} = T$;
- To each name c_i, assign the value c_i;
- For each predicate R, we set R to be true precisely when $(R\, t_1 \ldots t_n) = B_i$ for some i.

- For each sentence letter R, R is true iff $R = B_i$ for some i.

Since we don't have equality, there is no checking to do here—this is a good definition of an interpretation! Also, by the way we've set up the predicates, each B_i is true in $\mathcal{I}$.

Now to see that $\mathcal{I}$ is a model of Γ, suppose $S \in \Gamma$. Then for some k, S is built out of the atomic formula $\{A_1, \ldots, A_k\}$ (maybe not all of them) using logical symbols but no quantifiers. Consider the set

$$\gamma = \{B_1, \ldots, B_k, S\}.$$

Since γ is a subset of the o.k. set Γ_{k+1}, there is a *model* $\mathcal{J}$ of γ. But this model assigns the same truth values to $A_1, A_2, \ldots, A_k$ as $\mathcal{I}$ does. Since S is true in $\mathcal{J}$ and since it is built from $A_1, \ldots, A_k$, S is true in $\mathcal{I}$.

Problem 2. We say that a first-order sentence S is a $\forall$-*sentence* if it has the form

$$\forall x_1 \ldots \forall x_n F$$

where F is a quantifier-free formula (note that F may contain function symbols in addition to predicates and names.) Show that the set

$$\{S \mid S \text{ is a valid } \forall\text{-sentence}\}$$

is decidable. (Hint: Show that a canonical refutation from the negation of any $\forall$-sentence is finite.)

6044 PS#8

Solution due to Carl de Marcken

2 The set $\{S \mid S$ is valid $\forall$-sentence$\}$ ~~if~~ is decidable if we can, for an arbitrary S, decide whether $\neg S$ is refutable. We can use a canonical refutation to decide, if it is finite. $\neg S$ has the form

$$\exists x_1 \exists x_2 \exists x_3 \ldots \exists x_n \neg F$$

Thus the length of a canonical refutation will be finite (no sentence may be used more than once for an application of EI).

25/25

Problem 3. Let S be a sentence. Prove that if $\Gamma \vdash S$, then there is a finite subset $\Gamma' \subseteq \Gamma$ such that $\Gamma' \vdash S$. (Hint: Compactness.)

Michael Frumkin
6.044J
PS # 8
prob. 3
12/3/89

3.

- $\Gamma \vdash S$ iff
- $\Gamma \cup \{\neg S\}$ is not satisfiable iff (def of $\vdash$)
- There is a finite subset of $\Gamma \cup \{\neg S\}$, X, (compactness) is unsatisfiable.
- $X \cup \{\neg S\}$ is not satisfiable if X is (compactness) unsatisfiable ~~So that if we~~ [illegible]
- $X \cup \{\neg S\}$ is unsatisfiable $\Leftrightarrow$ $X \vdash S$

Therefore if $\Gamma \vdash S$ there is a finite set X, s.t. $X \subseteq \Gamma$ & $X \vdash S$.

6.044J/18.423J: Computability, Programming, and Logic — Handout 36
Massachusetts Institute of Technology — 11 December 1989

Solutions to Supplemental Problems

Problem 1. Recall the definition of the addition function *plus* given in lecture:

$$\mathit{plus} \overset{\text{df}}{=} Y(\lambda p.\lambda x.\lambda y.\mathsf{cond}\ y\ x\ (\mathsf{succ}\ (p\ x\ (\mathsf{pred}\ y))))$$

Using the rewrite rules, show all the steps in the evaluation of $((\mathit{plus}\ 31)\ 2)$.

Solution. Before giving the complete reduction, let's define some names for terms (as the notes do) to make the reduction easier to read:

$$\begin{aligned}
\mathit{body} &\overset{\text{df}}{=} \lambda x.\lambda y.\mathsf{cond}\ y\ x\ (\mathsf{succ}\ (p\ x\ (\mathsf{pred}\ y))) \\
H &\overset{\text{df}}{=} \lambda x.\lambda y.\mathsf{cond}\ y\ x\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ x\ (\mathsf{pred}\ y))) \\
G &\overset{\text{df}}{=} \lambda y.\mathsf{cond}\ y\ 31\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ 31\ (\mathsf{pred}\ y)))
\end{aligned}$$

The complete reduction sequence, carefully written and showing each step, is:

$$\begin{aligned}
((\mathit{plus}\ 31)\ 2) &\to ((((\lambda p.\mathit{body})\ (\lambda z.\mathit{plus}\ z))\ 31)\ 2) \\
&\to ((H\ 31)\ 2) \\
&\to (G\ 2) \\
&\to \mathsf{cond}\ 2\ 31\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ 31\ (\mathsf{pred}\ 2))) \\
&\to \mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ 31\ (\mathsf{pred}\ 2)) \\
&\to \mathsf{succ}\ (\mathit{plus}\ 31\ (\mathsf{pred}\ 2)) \\
&\to \mathsf{succ}\ (((\lambda p.\mathit{body})\ (\lambda z.\mathit{plus}\ z))\ 31\ (\mathsf{pred}\ 2)) \\
&\to \mathsf{succ}\ (H\ 31\ (\mathsf{pred}\ 2)) \\
&\to \mathsf{succ}\ (G\ (\mathsf{pred}\ 2)) \\
&\to \mathsf{succ}\ (G\ 1) \\
&\to \mathsf{succ}\ (\mathsf{cond}\ 1\ 31\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ 31\ (\mathsf{pred}\ 1)))) \\
&\to \mathsf{succ}\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ 31\ (\mathsf{pred}\ 1))) \\
&\to \mathsf{succ}\ (\mathsf{succ}\ (\mathit{plus}\ 31\ (\mathsf{pred}\ 1)) \\
&\to \mathsf{succ}\ (\mathsf{succ}\ (((\lambda p.\mathit{body})\ (\lambda z.\mathit{plus}\ z))\ 31\ (\mathsf{pred}\ 1))) \\
&\to \mathsf{succ}\ (\mathsf{succ}\ (H\ 31\ (\mathsf{pred}\ 1))) \\
&\to \mathsf{succ}\ (\mathsf{succ}\ (G\ (\mathsf{pred}\ 1))) \\
&\to \mathsf{succ}\ (\mathsf{succ}\ (G\ 0)) \\
&\to \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{cond}\ 0\ 31\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ 31\ (\mathsf{pred}\ 0))))) \\
&\to \mathsf{succ}\ (\mathsf{succ}\ 31) \\
&\to \mathsf{succ}\ 32 \\
&\to 33
\end{aligned}$$

A lot of parentheses have been left out of the terms in this reduction sequence—remember that $(M\ N\ P)$ is shorthand for $((M\ N)\ P)$.

Problem 2. Show all the steps of the SECD machine on the evaluation of $((\mathit{plus}\ 31)\ 1)$. That is, starting from the dump

$$[\mathit{nil}, \emptyset, ((\mathit{plus}\ 31)\ 1), \mathit{nil}]$$

show all steps of the SECD machine till it halts.

Solution. This problem should exercise all of the rules of the SECD machine. Here is the first part of the history of the evaluation of the SECD machine:

$$\begin{aligned}
[\mathit{nil},\ & \emptyset, ((\mathit{plus}\ 31)\ 1), \mathit{nil}] \Rightarrow [\mathit{nil}, \emptyset, 1 : (\mathit{plus}\ 31) : \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [[1, \emptyset], \emptyset, (\mathit{plus}\ 31) : \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [[1, \emptyset], \emptyset, 31 : \mathit{plus} : \mathit{ap} : \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [[31, \emptyset] : [1, \emptyset], \emptyset, \mathit{plus} : \mathit{ap} : \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [[31, \emptyset] : [1, \emptyset], \emptyset, (\lambda p.\mathit{body}) : Y : \mathit{ap} : \mathit{ap} : \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [[\lambda p.\mathit{body}, \emptyset] : [31, \emptyset] : [1, \emptyset], \emptyset, Y : \mathit{ap} : \mathit{ap} : \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [[Y, \emptyset] : [\lambda p.\mathit{body}, \emptyset] : [31, \emptyset] : [1, \emptyset], \emptyset, \mathit{ap} : \mathit{ap} : \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [\mathit{nil}, \emptyset, ((\lambda p.\mathit{body})\ (\lambda z.\mathit{plus}\ z)), [S_1, \emptyset, \mathit{ap} : \mathit{ap}, \mathit{nil}]] \\
& \Rightarrow [\mathit{nil}, \emptyset, (\lambda z.\mathit{plus}\ z) : (\lambda p.\mathit{body}) : \mathit{ap}, [S_1, \emptyset, \mathit{ap} : \mathit{ap}, \mathit{nil}]] \\
& \Rightarrow [[(\lambda z.\mathit{plus}\ z), \emptyset], \emptyset, (\lambda p.\mathit{body}) : \mathit{ap}, [S_1, \emptyset, \mathit{ap} : \mathit{ap}, \mathit{nil}]] \\
& \Rightarrow [[(\lambda p.\mathit{body}), \emptyset] : [(\lambda z.\mathit{plus}\ z), \emptyset], \emptyset, \mathit{ap}, [S_1, \emptyset, \mathit{ap} : \mathit{ap}, \mathit{nil}]] \\
& \Rightarrow [\mathit{nil}, \emptyset\{[(\lambda z.\mathit{plus}\ z), \emptyset]/p\}, \mathit{body}, [\mathit{nil}, \emptyset, \mathit{nil}, [S_1, \emptyset, \mathit{ap} : \mathit{ap}, \mathit{nil}]]] \\
& \Rightarrow [[\mathit{body}, E_1], E_1, \mathit{nil}, [\mathit{nil}, \emptyset, \mathit{nil}, [S_1, \emptyset, \mathit{ap} : \mathit{ap}, \mathit{nil}]]] \\
& \Rightarrow [[\mathit{body}, E_1], \emptyset, \mathit{nil}, [S_1, \emptyset, \mathit{ap} : \mathit{ap}, \mathit{nil}]] \\
& \Rightarrow [[\mathit{body}, E_1] : [31, \emptyset] : [1, \emptyset], \emptyset, \mathit{ap} : \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [\mathit{nil}, E_1\{[31, \emptyset]/x\}, \mathit{body}_1, [[1, \emptyset], \emptyset, \mathit{ap}, \mathit{nil}]] \\
& \Rightarrow [[\mathit{body}_1, E_2], E_2, \mathit{nil}, [[1, \emptyset], \emptyset, \mathit{ap}, \mathit{nil}]] \\
& \Rightarrow [[\mathit{body}_1, E_2] : [1, \emptyset], \emptyset, \mathit{ap}, \mathit{nil}] \\
& \Rightarrow [\mathit{nil}, E_3, \mathit{body}_2, [\mathit{nil}, \emptyset, \mathit{nil}, \mathit{nil}]] \\
& \Rightarrow [[x, E_3] : [\mathit{body}_3, E_3], E_3, y : \mathit{cd}, [\mathit{nil}, \emptyset, \mathit{nil}, \mathit{nil}]] \\
& \Rightarrow [[1, \emptyset] : [x, E_3] : [\mathit{body}_3, E_3], E_3, \mathit{cd}, [\mathit{nil}, \emptyset, \mathit{nil}, \mathit{nil}]] \\
& \Rightarrow [\mathit{nil}, E_3, \mathit{body}_3, [\mathit{nil}, \emptyset, \mathit{nil}, \mathit{nil}]]
\end{aligned}$$

where

$$\begin{aligned}
\mathit{body} &\overset{\text{df}}{=} \lambda x.\lambda y.\mathsf{cond}\ y\ x\ (\mathsf{succ}\ (p\ x\ (\mathsf{pred}\ y)))\\
\mathit{body}_1 &\overset{\text{df}}{=} \lambda y.\mathsf{cond}\ y\ x\ (\mathsf{succ}\ (p\ x\ (\mathsf{pred}\ y)))\\
\mathit{body}_2 &\overset{\text{df}}{=} \mathsf{cond}\ y\ x\ (\mathsf{succ}\ (p\ x\ (\mathsf{pred}\ y)))\\
\mathit{body}_3 &\overset{\text{df}}{=} (\mathsf{succ}\ (p\ x\ (\mathsf{pred}\ y)))\\
E_1 &\overset{\text{df}}{=} \emptyset\{[(\lambda z.\mathit{plus}\ z),\emptyset]/p\}\\
E_2 &\overset{\text{df}}{=} E_1\{[31,\emptyset]/x\}\\
E_3 &\overset{\text{df}}{=} E_2\{[1,\emptyset]/y\}
\end{aligned}$$

For brevity, the evaluation has been terminated here although there are many more reductions to do.

Problem 3. Give a rigorous proof that $((\mathit{plus}\ n)\ m)$, under the rewrite rules, evaluates to $n + m$, *i.e.*, $\mathit{Eval}(((\mathit{plus}\ n)\ m)) = n + m$. (Hint: Use induction on m. Note that this is a familiar fact, but is not obvious since the rewrite rules could be defined in a bizarre way.)

Solution. The point of this problem is to see that the rewrite rules work in the expected way, *i.e.*, the code for *plus* works as its informal description says it does. The notation may be a bit confusing: $(n + m)$ is a *numeral* representing the addition of the numerals n and m.

We prove the fact by induction on m. First, let's define *body* and H as above, and for any numeral n, define

$$G_n \overset{\text{df}}{=} \lambda y.\mathsf{cond}\ y\ n\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ n\ (\mathsf{pred}\ y)))$$

In the base case, $m = 0$. Then $((\mathit{plus}\ n)\ 0)$ has the following reduction sequence using the rewrite rules:

$$\begin{aligned}
((\mathit{plus}\ n)\ 0) &\rightarrow ((((\lambda p.\mathit{body})\ (\lambda z.\mathit{plus}\ z))\ n)\ 0)\\
&\rightarrow ((H\ n)\ 0)\\
&\rightarrow (G_n\ 0)\\
&\rightarrow \mathsf{cond}\ 0\ n\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ n\ (\mathsf{pred}\ 0)))\\
&\rightarrow n
\end{aligned}$$

as desired.

The induction case, where $m = k + 1$, requires a delicate argument. Note that $((\mathit{plus}\ n)\ m)$ has the following reduction sequence:

$$((\mathit{plus}\ n)\ m) \rightarrow ((((\lambda p.\mathit{body})\ (\lambda z.\mathit{plus}\ z))\ n)\ m)$$

$$\begin{aligned}
&\rightarrow\ ((H\ n)\ m)\\
&\rightarrow\ (G_n\ m)\\
&\rightarrow\ \mathsf{cond}\ m\ n\ (\mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ n\ (\mathsf{pred}\ m)))\\
&\rightarrow\ \mathsf{succ}\ ((\lambda z.\mathit{plus}\ z)\ n\ (\mathsf{pred}\ m))\\
&\rightarrow\ \mathsf{succ}\ (\mathit{plus}\ n\ (\mathsf{pred}\ m))\\
&\rightarrow\ \mathsf{succ}\ ((((\lambda p.\mathit{body})\ (\lambda z.\mathit{plus}\ z))\ n)\ (\mathsf{pred}\ m))\\
&\rightarrow\ \mathsf{succ}\ ((H\ n)\ (\mathsf{pred}\ m))\\
&\rightarrow\ \mathsf{succ}\ (G_n\ (\mathsf{pred}\ m))\\
&\rightarrow\ \mathsf{succ}\ (G_n\ k)
\end{aligned}$$

We cannot yet apply the induction hypothesis, since we have yet to run across a subterm of the form $((\mathit{plus}\ n)\ k)$. However, notice that

$$\begin{aligned}
((\mathit{plus}\ n)\ k)\ &\rightarrow\ ((((\lambda p.\mathit{body})\ (\lambda z.\mathit{plus}\ z))\ n)\ k)\\
&\rightarrow\ ((H\ n)\ k)\\
&\rightarrow\ (G_n\ k)
\end{aligned}$$

which, by induction, must finally reduce to the numeral $(n+k)$. The definition of the rewrite rules guarantees that operands evaluated until they become values; hence, succ $(G_n\ k)$ must reduce to succ $(n+k)$ in some number of steps. Thus, $((\mathit{plus}\ n)\ m)$ reduces to the numeral $(n+m)$, and we are done.

6.044J/18.423J: Computability, Programming, and Logic Handout 37
Massachusetts Institute of Technology 11 December 1989

Quiz 4

Instructions. This exam is *closed book.* Do all problems in the provided white book, carefully labeling solutions with their corresponding numbers. Points are listed for each problem. You have one and a half hours. Good luck.

Problem 1. [20 points] (Refutation Procedure) An $\forall\exists$-sentence is a sentence of first-order predicate calculus of the form

$$\forall x_1 \forall x_2 \ldots \forall x_n \exists y_1 \exists y_2 \ldots \exists y_m F(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

where F is quantifier-free and has *no* function symbols. Show that it is decidable whether an $\forall\exists$-sentence is valid.

Problem 2. [30 points] (Evaluation of FKS terms)

2(a). Using the rewrite rules, work out the first 6 steps of the evaluation of $((Y\ (\lambda f^{\iota\to\iota}.f))\ 3)$. Briefly describe the remainder of the evaluation (see the appendix for the rewrite rules of FKS.)

2(b). Describe precisely the partial recursive function represented by the term $(Y\ (\lambda f^{\iota\to\iota}.f))$.

Problem 3. [25 points] (Compactness) Show that there is no first-order sentence which means ***precisely*** "the binary relation $\prec$ is a strict partial order on an infinite domain." (Hint: Note that there is a sentence, PO, which means "$\prec$ is a strict partial order," namely

$$[\forall x \forall y \forall z (x \prec y) \land (y \prec z) \rightarrow (x \prec z)] \land [\forall x \forall y (x \prec y) \rightarrow \neg (y \prec x)].)$$

Problem 4. [25 points] (Arithmetic definability)

4(a). Show that the set of true atomic sentences of arithmetic is decidable. Conclude that the set of true quantifier-free sentences of arithmetic is also decidable.

Let *pair* : $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ be a recursive, bijective (*i.e.*, one-to-one and onto) function. For $A \subseteq \mathbf{N}$, we write $\mathit{right}(A)$ as shorthand for $\{m \mid \mathit{pair}(n,m) \in A\}$.

4(b). Let F_1 be a quantifier-free formula of arithmetic with free variables x_0, x_1, and x_2, and let G_1 be the formula

$$\exists x_1 \forall x_2 F_1(x_0, x_1, x_2).$$

Let

$$\begin{aligned} A_1 &\overset{\text{df}}{=} \{\mathit{pair}(n_2, (\mathit{pair}(n_1, n_0))) \mid F_1(n_0, n_1, n_2) \text{ is true in arithmetic}\} \\ B_1 &\overset{\text{df}}{=} \mathit{right}(\overline{\mathit{right}(\overline{A_1})}) \end{aligned}$$

Then it is not hard to see that B_1 is the arithmetically definable set defined by G_1. Now let G_2 be a formula

$$\forall x_1 \exists x_2 \exists x_3 F_2(x_0, x_1, x_2, x_3).$$

and let

$$A_2 \overset{\text{df}}{=} \{\mathit{pair}(n_3, \mathit{pair}(n_2, (\mathit{pair}(n_1, n_0)))) \mid F_2(n_0, n_1, n_2, n_3) \text{ is true in arithmetic}\}$$

Write an expression for the corresponding set B_2 in terms of A_2 and the operators "*right*" and complement. No explanation is required.

4(c). Conclude that every arithmetically definable set may be obtained from some recursive set by applications of the complement and *right* operations. (Hint: Prenex form.)

Appendix—Rewrite Rules of FKS.

1. (a) (succ n) → $(n+1)$

   (b) (pred 0) → 0

   (c) (pred $(n+1)$) → n

   (d) $(Y_\sigma\ V) \rightarrow (V\ (\lambda x^\sigma.(Y_\sigma\ V)\ x^\sigma))$ (where $x \notin FV(V)$)

2. (a) $((\lambda x.M)\ V) \rightarrow M[x := V]$ (for V a value)

   (b) (cond 0 N_1 N_2) → N_1

   (c) (cond $(n+1)$ N_1 N_2) → N_2

3. (a) if $M \rightarrow M'$ then $(M\ N) \rightarrow (M'\ N)$

   (b) if $N \rightarrow N'$ then $(V\ N) \rightarrow (V\ N')$ (for V a value)

   (c) if $M \rightarrow M'$ then (cond M N_1 N_2) → (cond M' N_1 N_2)

6.044J/18.423J: Computability, Programming, and Logic — Handout 38
Massachusetts Institute of Technology — 13 December 1989

Quiz 4 Solutions

Problem 1. [20 points] (Refutation Procedure) An $\forall\exists$-sentence is a sentence of first-order predicate calculus of the form

$$\forall x_1 \forall x_2 \ldots \forall x_n \exists y_1 \exists y_2 \ldots \exists y_m F(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

where F is quantifier-free and has *no* function symbols. Show that it is decidable whether an $\forall\exists$-sentence is valid.

Solution. Note first that the negation of an $\forall\exists$-sentence is an $\exists\forall$-sentence (similarly defined.) Now consider the canonical derivation beginning from a single $\exists\forall$-sentence; we first do n applications of the EI rule, followed by as many applications of the UI rule as we can do. Since there are no function symbols, the only terms we need to use in the applications of the UI rule are the constants that have appeared previously in the derivation (n of them) plus those that appear in the original sentence. This is a finite number—hence, the canonical derivation is finite.

Now we can check to see whether the $\exists\forall$-sentence is satisfiable or not by looking at the finite canonical derivation. If it is satisfiable (*i.e.*, every finite set of quantifier-free sentences in the derivation is satisfiable), the original $\forall\exists$-sentence is *not* valid. If the $\exists\forall$-sentence is not satisfiable, the original sentence is valid. This gives us a decision procedure, so the set of valid $\forall\exists$-sentences is decidable.

Problem 2. [30 points] (Evaluation of FKS terms)

2(a). Using the rewrite rules, work out the first 6 steps of the evaluation of $((Y\ (\lambda f^{\iota\to\iota}.f))\ 3)$. Briefly describe the remainder of the evaluation (see the appendix for the rewrite rules of FKS.)

Solution. The first six steps of the reduction sequence are

$$\begin{aligned}
((Y\ (\lambda f^{\iota\to\iota}.f))\ 3) &\to (((\lambda f.f)(\lambda x.Y\ (\lambda f^{\iota\to\iota}.f)\ x))\ 3)\\
&\to ((\lambda x.Y\ (\lambda f^{\iota\to\iota}.f)\ x)\ 3)\\
&\to ((Y\ (\lambda f^{\iota\to\iota}.f))\ 3)\\
&\to (((\lambda f.f)(\lambda x.Y\ (\lambda f^{\iota\to\iota}.f)\ x))\ 3)\\
&\to ((\lambda x.Y\ (\lambda f^{\iota\to\iota}.f)\ x)\ 3)\\
&\to ((Y\ (\lambda f^{\iota\to\iota}.f))\ 3)
\end{aligned}$$

The remainder of the reduction sequence simply repeats this pattern *ad infinitum.*

2(b). Describe precisely the partial recursive function represented by the term $(Y\ (\lambda f^{\iota \rightarrow \iota}.f))$.

Solution. The fact that we used the numeral 3 used in the above sequence is inconsequential—on ***any*** numeral the function $(Y\ (\lambda f^{\iota \rightarrow \iota}.f))$ will loop. Thus, this term represents the totally undefined partial function.

Problem 3. [25 points] (Compactness) Show that there is no first-order sentence which means ***precisely*** "the binary relation $\prec$ is a strict partial order on an infinite domain." (Hint: Note that there is a sentence, PO, which means "$\prec$ is a strict partial order," namely

$$[\forall x \forall y \forall z (x \prec y) \wedge (y \prec z) \rightarrow (x \prec z)] \wedge [\forall x \forall y (x \prec y) \rightarrow \neg (y \prec x)].)$$

Solution. Proceed by contradiction—suppose there is a first-order sentence S which is true precisely in interpretations with an infinite domain where $\prec$ is a strict partial order. Now consider the sentence $(\neg S)$—this sentence is true precisely in those interpretations either with a finite domain *or* where $\prec$ is not a strict partial order. Finally, consider the sentence $(\neg S) \wedge PO$; this sentence is true precisely in those interpretations in which $\prec$ is a strict partial order on a finite domain.

Let R_n be the sentence that says that the domain "has at least n elements," *i.e.*,

$$R_n \overset{\text{df}}{=} \exists x_1 \ldots \exists x_n \bigwedge_{1 \leq i < j \leq n} (x_i \neq x_j)$$

The set of sentences $\{(\neg S) \wedge PO, R_1, R_2, R_3, \ldots\}$ is finitely satisfiable (ok), and hence by Compactness has a model. But this model must have an infinite number of elements (as it satisifies each R_n) and satisfies $(\neg S) \wedge PO$, a contradiction.

Problem 4. [25 points] (Arithmetic definability)

4(a). Show that the set of true atomic sentences of arithmetic is decidable. Conclude that the set of true quantifier-free sentences of arithmetic is also decidable.

Solution. The atomic sentences of arithmetic are just equalities between closed terms, where closed terms are built from the constant **0** and the function symbols $'$, $\cdot$, and $+$. It is easy to decide whether two closed terms over this language are equal—simply calculate the values and check to see whether they are equal! The set of true quantifier-free sentences may be decided using this calculation procedure as a subroutine—calculate the values of all the terms and see whether the sentence comes out to be true using truth tables.

Let *pair* : $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ be a recursive, bijective (*i.e.*, one-to-one and onto) function. For $A \subseteq \mathbf{N}$, we write $\mathit{right}(A)$ as shorthand for $\{m \mid \mathit{pair}(n,m) \in A\}$.

4(b). Let F_1 be a quantifier-free formula of arithmetic with free variables x_0, x_1, and x_2, and let G_1 be the formula

$$\exists x_1 \forall x_2 F_1(x_0, x_1, x_2).$$

Let

$$\begin{aligned} A_1 &\overset{\text{df}}{=} \{\mathit{pair}(n_2,(\mathit{pair}(n_1,n_0))) \mid F_1(n_0,n_1,n_2) \text{ is true in arithmetic}\} \\ B_1 &\overset{\text{df}}{=} \mathit{right}(\overline{\mathit{right}(\overline{A_1})}) \end{aligned}$$

Then it is not hard to see that B_1 is the arithmetically definable set defined by G_1. Now let G_2 be a formula

$$\forall x_1 \exists x_2 \exists x_3 F_2(x_0, x_1, x_2, x_3).$$

and let

$$\begin{aligned} A_2 \overset{\text{df}}{=} \; & \{\mathit{pair}(n_3,\mathit{pair}(n_2,(\mathit{pair}(n_1,n_0)))) \mid \\ & \qquad F_2(n_0,n_1,n_2,n_3) \text{ is true in arithmetic}\} \end{aligned}$$

Write an expression for the corresponding set B_2 in terms of A_2 and the operators "*right*" and complement. No explanation is required.

Solution. $B_2 = \overline{\mathit{right}(\overline{\mathit{right}(\mathit{right}(A_2))})}$.

4(c). Conclude that every arithmetically definable set may be obtained from some recursive set by applications of the complement and *right* operations. (Hint: Prenex form.)

Solution. Suppose a set $S \subseteq \mathbf{N}$ is definable by a formula $G(x_0)$ with a single free variable x_0. Without loss of generality, we can assume that $G(x_0)$ is in prenex form (if not, put it in that form); that is,

$$G(x) = Q_1 x_1 \ldots Q_n x_k \; F(x_0, x_1, \ldots x_k)$$

where F is quantifier-free, and the Q_i's are either $\exists$ or $\forall$. By part (a) and the fact that *pair* is recursive, the set

$$\begin{aligned} A \;=\; & \{\mathit{pair}(n_k,\mathit{pair}(n_{k-1},(\ldots,\mathit{pair}(n_1,n_0)\ldots))) \mid \\ & \qquad F(n_0,\ldots,n_k) \text{ is true in arithmetic}\} \end{aligned}$$

is recursive.

Now recall a formula $\forall x H$ is equivalent to $\neg\exists x(\neg H)$. To determine the set defined by $G(x)$, we use a little recursive procedure based on the outermost quantifier of a prenex formula H:

- If no quantifiers appear in H, return A;
- If $H = \exists x H'$, apply *right* to the set obtained by the recursive call on H';
- If $H = \forall x H'$, apply complement, *right*, and then complement to the set obtained by the recursive call on H'.

When we put in $G(x)$ into this procedure, we get the set defined by $G(x)$ in terms of applications of complement and *right* to the recursive set A. We are thus done.

Appendix—Rewrite Rules of FKS.

1. (a) (succ n) → $(n+1)$
   (b) (pred 0) → 0
   (c) (pred $(n+1)$) → n
   (d) $(Y_\sigma\ V) \rightarrow (V\ (\lambda x^\sigma.(Y_\sigma\ V)\ x^\sigma))$ (where $x \notin FV(V)$)
2. (a) $((\lambda x.M)\ V) \rightarrow M[x := V]$ (for V a value)
   (b) (cond 0 N_1 N_2) → N_1
   (c) (cond $(n+1)$ N_1 N_2) → N_2
3. (a) if $M \rightarrow M'$ then $(M\ N) \rightarrow (M'\ N)$
   (b) if $N \rightarrow N'$ then $(V\ N) \rightarrow (V\ N')$ (for V a value)
   (c) if $M \rightarrow M'$ then (cond M N_1 N_2) → (cond M' N_1 N_2)

6.044J

Introduction

Lecture 1, 9/13/89

Structure 4 quizzes, no final
*Must send pink sheet to registrar

9-10 | 4
6-3 | R·C

Question What is computability? We might want to call

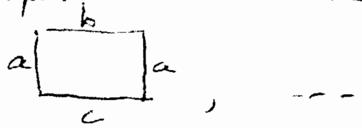
$$f: \mathbb{N} \to \mathbb{N} \quad \text{or} \quad g: \{a, b, c\}^* \longrightarrow \{0, 1\}^*$$

Some possibilities:

$id(n) = n$	identity
$sum(n, m) = n + m$	addition
$mult(n, m) = n * m$	multiplication

Once you solve I/O problems, the algorithms are eas,
So we'll call these functions computable.

Surprise: Can't compute all functions! Something you can't compute: A function which takes colored tiles



Can you tile the plane? Undecidable.

Note Uncomputable functions are interesting; computable functions may or may not.

Other two sections Logic & Programming Languages

~~Circulate another signup next time~~

~~Memo to Anne Hunter — course info with note on front~~

~~Have David set up file drawer~~

$\boxed{\text{Inductive Definitions}}$ Lecture 2, 9/15/99

<u>Def</u>: A set Σ whose elements are "symbols" will denote an "alphabet."

Examples: ASCII characters, $\{0, 1\}$, etc.

<u>Pick</u>: Some object called "e_Σ" also called the "empty string" (over Σ) with the property that e_Σ is <u>not</u> an ordered pair.

<u>Def</u>: "Strings over Σ" inductively by (call it Σ^*)

(1) e_Σ is a string

(2) if $\sigma \in \Sigma$, and x is a string, then (σ, x) is a string.

(3) that's all

<u>Def</u>: Concatenation : $\Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. $x \cdot y$ is infix for Concatenation (x, y). Define by induction on definition of x being a string:

(1) $e_\Sigma \cdot y = y$ for all $y \in \Sigma^*$

(2) $(\sigma, x) \cdot y = (\sigma, x \cdot y)$

<u>Lemma 1</u>: $x \cdot e = x$ for all $x \in \Sigma^*$

<u>Proof</u>: By induction on x:

(1) If $x = e$, then

$$x \cdot e = e \cdot e \overset{\text{by cat (1)}}{=} e = x$$

(2) If $x = (\sigma, x')$, then

$$x \cdot e = (\sigma, x') \cdot e \overset{\text{by cat (2)}}{=} (\sigma, x' \cdot e)$$
$$= (\sigma, x') \quad \text{(by induction)}$$
$$= x$$

QED

<u>Lemma 2</u>: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

<u>Proof</u>: By induction on x:

(1) $(e \cdot y) \cdot z \overset{\text{cat (1)}}{=} y \cdot z \overset{\text{cat (1)}}{=} e \cdot (y \cdot z)$

(2) Say $x = (\sigma, x')$

$$(x \cdot y) \cdot z = ((\sigma, x') \cdot y) \cdot z \overset{\text{cat (2)}}{=} (\sigma, x' \cdot y) \cdot z$$
$$\underset{\text{cat (2)}}{=} (\sigma, (x' \cdot y) \cdot z) \underset{\text{induction}}{=} (\sigma, x' \cdot (y \cdot z)) \underset{\text{cat (2)}}{=} (\sigma, x') \cdot$$

$= x \cdot (y \cdot z)$

QED

Lemma 3: $(\sigma, x) = (\sigma, e \cdot x) = (\sigma, e) \cdot x$

Corollary: Every string can be obtained from strings of "length 1," of form (σ, e), by finite # of concatenations.

Convenience: We'll be sloppy about difference between σ and (σ, e)

Def: $|x| : \Sigma^* \to \mathbb{N}$

(1) $|e| = 0$

(2) $|(\sigma, x)| = 1 + |x|$

Lemma: (Exercise) $|x \cdot y| = |x| + |y|$

Diagonal Arguments Lecture 3, 9/18/89

Diagonal Argument: Let $f_i : \mathbb{N} \to \mathbb{N}$ — may be partial.

Side Remark: Let $g : A \to B$

Basic: $\text{domain}(g) = \{ a \in A : g(a) \text{ is defined} \}$

$\text{range}(g) = \{ b \in B : g(a) = b \text{ for some } a \in A \}$

A is "soure", B is "target"

Def: $d : \mathbb{N} \to \mathbb{N}$ a diagonal function by

$$d(n) = \begin{cases} f_n(n) + 1 & \text{if } n \in \text{dom}(f_n) \\ 0 & \text{otherwise} \end{cases}$$

Claim: d is not in the list, i.e. for all $n \geq 0$, $d \neq f_n$

Proof: Suppose $d = f_{n_0}$ for some n_0. We know

$$f_{n_0}(n) \neq f_n(n)$$

for all n, since $f_{n_0} = d$, such that $f_n(n)$ is define

Also, $f_{n_0}(n_0)$ is defined, since d is total by def.

Therefore, let n be n_0 — thus

$$f_{n_0}(n_0) \neq f_{n_0}(n_0) \quad \rightarrow\!\leftarrow$$

(A contradiction.) ☒

Remark: Whatever "programs" are, there are only countably many "programmable" (computable) functions from $\mathbb{N}$ to $\mathbb{N}$

Why? Consider Scheme programs — they can be enumerated. Then d for this list is not computable. This argument applies to any programming language.

Def: $h : \mathbb{N} \to \mathbb{N}$

$$h(n) = \begin{cases} 1 \dot{-} f_n(n) & \text{if } n \in \text{dom}(f_n) \\ 0 & \text{otherwise} \end{cases}$$

Recursive Functions Lecture 4, 9/20/89

Reading: Chapters 1, 2, 3, 5, 7 of Boolos & Jeffrey. For coming week, 6 & 8. Skip chapter 4, 16, 19, 20, 22, 23, 24, 26, 27.

We will read: Chapters 9-15; maybe 17, 18, 21, 25

Composition: $f: N \to N$, $g: N \to N$. Then

$Cn_{1,1}[f, g]' = f \circ g =$ the function $h: N \to N$ such that

$$h(x) = f(g(x))$$

Since we're working with partial functions, we have some problems. Convention: if g is partial, then

$$x \in dom(h) \quad \text{iff} \quad [x \in dom(g) \wedge g(x) \in dom(f)]$$

Example: $s(x) = x+1$; then $Cn_{1,1}[s, s](y) = y+2$

Generalize: $f': N^3 \to N$, $g_i': N^2 \to N$ $i = 1, 2, 3.$

$h' = Cn_{3,2}[f', g_1', g_2', g_3'] : N^2 \to N$

given by rule

$$h'(x_1, x_2) = f'(g_1(x_1, x_2), g_2(x_1, x_2), g_3(x_1, x_2))$$

In general, $Cn_{k,l}[\ldots]$.

Def: Let $id_i^3(x_1, x_2, x_3) = x_i$ $i = 1, 2, 3.$

Can generalize too.

Let $z: N \to N$ be $z(x) = 0$

Example: (1) $Cn_{1,2}[s, id_2^2](x_1, x_2) = x_2 + 1$

$Cn_{1,2}[s, id_2^2] = \lambda(x_1, x_2) = x_2 + 1$

(2) $Cn_{1,2}[s, Cn_{1,2}[s, id_2^2]] = \lambda(x_1, x_2).\ x_2 + 2$

(3) $Cn_{2,2}[\dot{-}, id_1^2, \lambda(x_1, x_2).\ x_2 + 2] = \lambda(x_1, x_2) = x_1 \dot{-} (x_2$

Example: Say $h: N^4 \to N$; want $g(x, y) = h(y, y, 2, x)$

$g = Cn_{4,2}[h, id_2^2, id_2^2, Cn_{1,2}[Cn_{1,1}[s, s], Cn_{1,2}[z, id_2^2$
$id_1^2]$

An example of explicit transformation

<u>Primitive Recursion</u>: $f: \mathbb{N} \to \mathbb{N}$, $g: \mathbb{N}^3 \to \mathbb{N}$. Let $h: \mathbb{N}^2 \to \mathbb{N}$ be given by

$$h(x, 0) = f(x)$$
$$h(x, y+1) = g(x, y, h(x,y))$$

$h = Pr_1[f, g]$. 1 here is # of parameters — can generalize to $f: \mathbb{N}^k \to \mathbb{N}$, $g: \mathbb{N}^{k+2} \to \mathbb{N}$, $h = Pr_k[f, g]: \mathbb{N}^{k+1} \to \mathbb{N}$

<u>Example</u>: $x + 0 = x$
$x + (y+1) = (x+y) + 1$ <u>or</u>
$x + s(y) = s(x+y)$

Let $f = id_1^1$, $g(u, v, w) = w + 1 = Cn_{1,3}[s, id_3^3]$.
Then

$$+ = Pr_1[f, g].$$

<u>Example</u>: Multiplication — $x * 0 = 0$
$x * (y+1) = (x * y) + x$

This will be Problem 4.

<u>Example</u>: Exponentiation — $x^0 = 1$
$x^{y+1} = x^y * x$

<u>Example</u>: $\dot{-}1$: $0 \dot{-} 1 = 0$
$(y+1) \dot{-} 1 = y$

<u>Example</u>: $x \dot{-} y$: $x \dot{-} 0 = x$
$x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1$

μ-Recursive Functions Lecture 5, 9/22/89

Def: P a predicate on N^k

$P(x_1, \ldots, x_k)$ is true or false

Examples

1. $Eq(x, y)$ iff $x = y$
2. $Q(x, y, z)$ iff $x * y = z$

Def: Characteristic function of P is $c_P : N^k \to \{0, 1\}$ (total) s.t.

$$c_P(\vec{x}_k) = 1 \quad \text{iff} \quad P(\vec{x}_k)$$

Lemma: P_1, P_2 primitive recursive implies $P_1 \wedge P_2$, $\neg P_1$ are also.

Proof:
$$c_{P_1 \wedge P_2}(\vec{x}) = c_{P_1}(\vec{x}) * c_{P_2}(\vec{x})$$
$$c_{\neg P_1}(\vec{x}) = 1 \dot{-} c_{P_1}(\vec{x}). \qquad \boxtimes$$

Note: "$x \geq y$" is primitive recursive because
$$c_{\geq}(x, y) = 1 \dot{-} (y \dot{-} x)$$

Also, "$x = y$" is prim. rec. since
$$x = y \quad \text{iff} \quad (x \geq y) \wedge (y \geq x)$$

Also, $Q(x, y, z)$ above is prim. rec.

Note: "if $p(\vec{x})$ then $f(\vec{x})$ else $g(\vec{x})$" is prim. recursive; simulate by
$$f(\vec{x}) * c_p(\vec{x}) + g(\vec{x}) * c_{\neg p}(\vec{x})$$

Def: Let P be a predicate on N^2. Then
$$f(w,x) = BMn_2\,[P]\,(w, x) = \begin{cases} \text{smallest } y \leq x \text{ s.t. } P(x,y) & \text{if there is such} \\ 0 & \text{otherwise} \end{cases}$$

Note: This is prim. rec. if P is:

$$f(0, x) = 0$$
$$f(w+1, x) = \text{if } f(w,x) \neq 0 \text{ then } f(w,x)$$
$$\text{else if } P(x,0) \text{ then } 0$$
$$\text{else if } P(x, w+1) \text{ then } w+1$$
$$\text{else } 0$$

Notation: $BMn\,[P]\,(w, x)$ is written as
$$\mu y \leq w \; P(x,y).$$

Bounded Quantification: $\exists y \le w .\ P(x,y)$. If P is primitive recursive, then this predicate is. Programmed:

$$[(\mu y \le w.\ P(x,y) \ne 0) \vee P(x,0)]$$

iff $\exists y \le w\ \ P(x,y)$

Also,

$$\forall y \le w .\ P(x,y) \quad \text{iff} \quad \neg \exists y \le w .\ \neg P(x,y)$$

Example: $\text{pair}(x,y) = 2^x * 3^y$ is prim. rec.
Then $\text{left}(z) = \mu x \le z .\ \exists y \le z\ \ \text{pair}(x,y) = z$ is also prim. rec.

Example: (1) $x \mid y$ iff $\exists z \le y .\ x * z = y$

(2) p is a prime iff $[(p > 1) \wedge \forall y \le p$ if $y \mid p$ then $(y=1) \vee (y=$

(3) y is a power of prime p iff
$(p$ is prime$) \wedge y \ne 0 \wedge$
$[\forall z \le y\ \ z \mid y$ implies $(z = 1 \vee p \mid z)]$

(4) $n(z,p) = p^n$ where n is the length of the base p representation of z

$n(z,p) = \mu y \le p * z .\ y$ is a power of p and $y > z$

(5) $y *_p z =$ concatenation of base p rep. of y ;
$= y\, n(z,p) + z$

(This gives us ability to program on strings in prim. rec. functions)

Def: $\text{Iterate}(f).(n,x) = f(\ldots f(x) \ldots)$
$g(0,x) = x$
$g(n+1,x) = f(g(n,x))$.

Def: $f_3 \; = 2^x$
$f_4(x) = f_3(\ldots f_3(1)\ldots) = \text{Iterate}(f_3)(x,1)$
$f_5(x) =$
$\vdots$

These are unimaginably large! Ackermann's function (roughly $f_n(n)$) is not prim. rec.

Note: Unbounded minimization – read in book. Gives a substantial increase in power that one can use, even more total fcns.

Lecture 6: Turing Machines

I. Introduction

A. Reading for today's lecture — Chapters 3, 5, and first 3 pages of 6 (standard form)

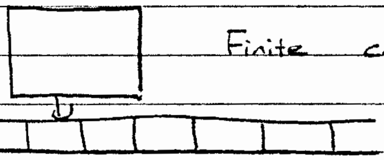
B. Today, we'll do a simple <u>machine - based</u> model of computation

1. Closer to underlying hardware than μ-recursi functions

2. We'll be programming in hardware, not in a higher-level language. But able to write complex progra

3. Punch line — it won't matter! The seemingly disparate notions of computability will yield <u>exactly</u> the same functions (numeric & string)

II. Overview of the model



Finite control

Infinite tape w/ distinct cells — input (fini on tape, rest blank

A. Infinite tape — can hold any symbols drawn from a finite alphabet. Book calls symbols $S_1, \ldots, S_n$. Also, special blank character S_0. These will be abbreviated 0, 1, ..., n.

B. Finite control — like a piece of hardware.

1. States — $q_1, \ldots, q_k$
q_1 is <u>starting</u> state.

2. Depending on what's on tape under read/w head, can either
   a. Move left or right; or
   b. Write a character
   Then go to different state

3. Important fact — "program" <u>cannot</u> change. Truly like a piece of hardware then.

III. Examples

Not in book

A. Given a string of $n \geq 1$ 1's, remove last "1" and halt reading rightmost 1.

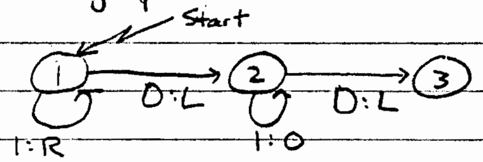
1. Alphabet here is $\{1\}$ with blank symbol (which we'll call 0 today — book uses both 0 and B.)
2. State chart

		Symbol	Read	
		0	1	
	q_1	Lq_2	Rq_1	
State	q_2	Lq_3	$0q_2$	
	q_3			← halt state

Explain this!

3. Flow graph — easier to use; we'll stick with the

But same as chart!!



read a 1 — move right

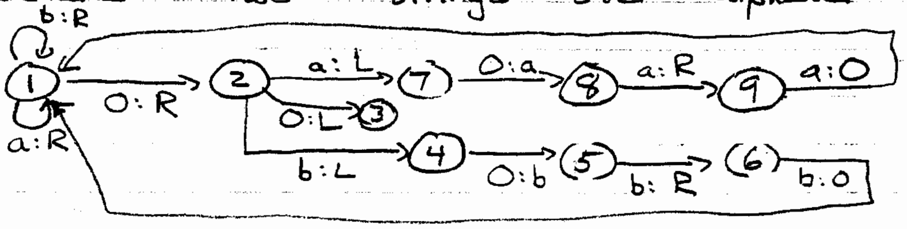
4. Example run — using a configuration (complete snapshot of the machine)

$0\underset{1}{1}110 \rightarrow 01\underset{1}{1}10 \rightarrow$

$011\underset{1}{1}0 \rightarrow 0111\underset{1}{0} \rightarrow$

$011\underset{2}{1}0 \rightarrow 011\underset{2}{0}0 \rightarrow 01\underset{3}{1}0$

B. Concatenate two strings over alphabet $\{a, b\}$

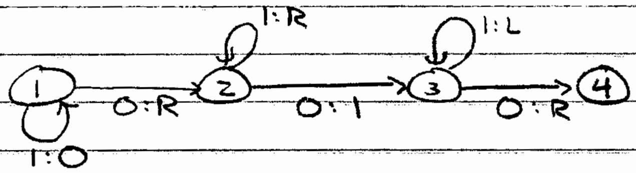


$0\underset{1}{a}b0ba0 \xrightarrow{*} 0ab\underset{1}{0}ba0 \rightarrow$

$0ab0\underset{2}{b}a0 \rightarrow 0ab\underset{4}{0}ba0 \rightarrow$

D. Examole: Addition

Overwrite 1 with 0, move right, Overwrite 0 with 1, read left to 0, move right.

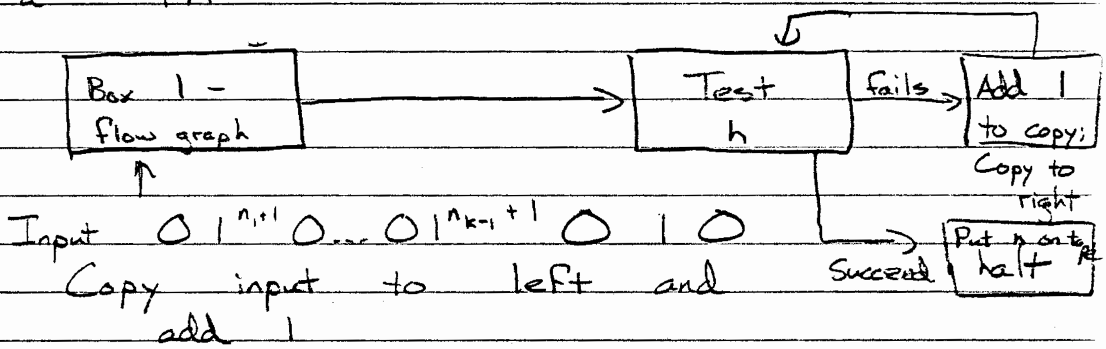


E. Read about multiplication; can also do exponentiation. Seems close to prim. recursiv functions — although may not stop!

F. Suppose we have a TM which, on input

$$0\,1^{n_1+1}\,0\,1^{n_2+1}\ldots 0\,1^{n_k+1}\,0$$

writes either "1" or something else. Call this h; Can then simulate $Mn[h] = \mu x_k . h(x_1, \ldots, x_{k-1}, \ldots$ by a TM



$0\ a\ b\ \underset{5}{b}\ b\ a\ 0 \rightarrow 0\ a\ b\ b\ \underset{6}{b}\ a\ 0 \rightarrow$

$0\ a\ b\ b\ \underset{1}{0}\ a\ 0 \rightarrow 0\ a\ b\ b\ 0\ \underset{2}{a}\ 0 \rightarrow$

$0\ a\ b\ b\ \underset{7}{0}\ a\ 0 \rightarrow 0\ a\ b\ b\ \underset{8}{a}\ a\ 0 \rightarrow$

$0\ a\ b\ b\ a\ \underset{9}{a}\ 0 \rightarrow 0\ a\ b\ b\ a\ \underset{1}{0}\ 0 \rightarrow$

$0\ a\ b\ b\ a\ 0\ \underset{3}{0} \rightarrow 0\ a\ b\ b\ a\ \underset{\uparrow}{0}\ 0$

[Note technique – states remember symbol read.]

c. Read books example on how to double a string of "1"s. [Erasing technique] – describe informally. <u>Combining technique</u>
[How would we quadruple 1's ?]

IV. Computing numeric functions

A. We've got a symbol pusher here – how do we expect to compute numeric fcns? Same way we do on digital computers – use <u>numerals</u> to represent numbers.

B. Standard format:

1. Use <u>unary</u> numerals – $(n+1)$ string of 1s represents n
2. For fcns of 2 or more args,
   $$0 1^{n_1+1} 0 1^{n_2+1} 0 1^{n_3+1} 0 \dots 0 1^{n_k+1} 0$$
   is standard form
3. Start machine reading leftmost 1.
4. End with
   $$0 1^{f(n_1,\dots,n_k)+1} 0$$
5. Never go past 00 input ---

C. Why use unary? WARNING BELLS shou[ld] be going off in heads.

1. Doesn't matter – could use binary!
2. Translator: Binary $\underset{0\ \ 1}{\overset{\{a,b\}}{\uparrow\ \uparrow}}$ to unary

Move left to right, doubling each time

Lecture 7 – Abacus Machines, Simulation

I. Introduction

A. Reading: Chapters 6, 7 of the text

B. Review

1. Def: A $^{\text{partial}}$ function $f: \mathbb{N}^k \to \mathbb{N}$ is μ-recursive (called recursive $^{\text{in book}}$) if it can be computed by a μ-recursive definition.
2. Def: A partial function $f: \mathbb{N}^k \to \mathbb{N}$ is Turing-computable if it can be computed by a TM in standard form (input / output conventions)

C. Today: • Abacus machines, another model of comput
• Simulation & Church's Thesis

II. Abacus machines – Definition   Abstract RAMs & Assenby$^{\text{Lan}}$

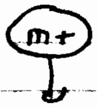
A. We shall try to abstract away the details of a modern digital computer, the Random Access Machine. Limited instructions.

B. Memory

1. Infinite # of registers labelled 1, 2, 3, ...
   a. Only use a finite # in a program
   b. More important – makes the model easier to work with; fcns not computable here will not be computable on RAMs!
2. Registers hold unbounded size natural numbers
   a. Only hold finite #'s at any stage
   b. Again – any fcns not computable here will not be computable on RAMs

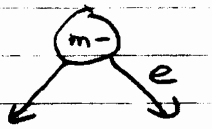
C. Program with flowcharts

1. Increment box



increments register m
$m := m + 1$
$[m] + 1 \to m$

2. Decrement & test



If $[m] = 0$ then e
else $[m] - 1 \to m$

On problem set, we'll have you add instructions and show power does not change

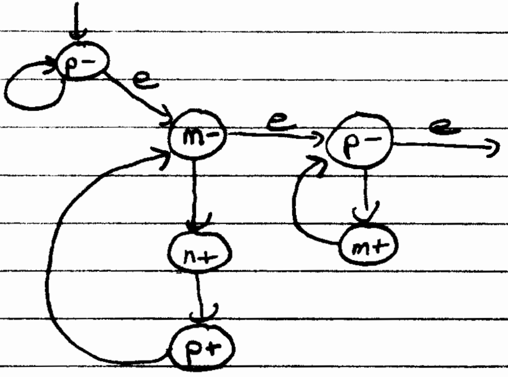
III. Examples of abacus

A. Add register m to register n

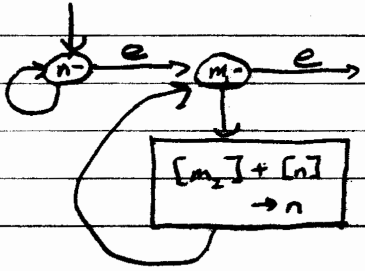
$0 \rightarrow p$

$[m] + [n] \rightarrow n$

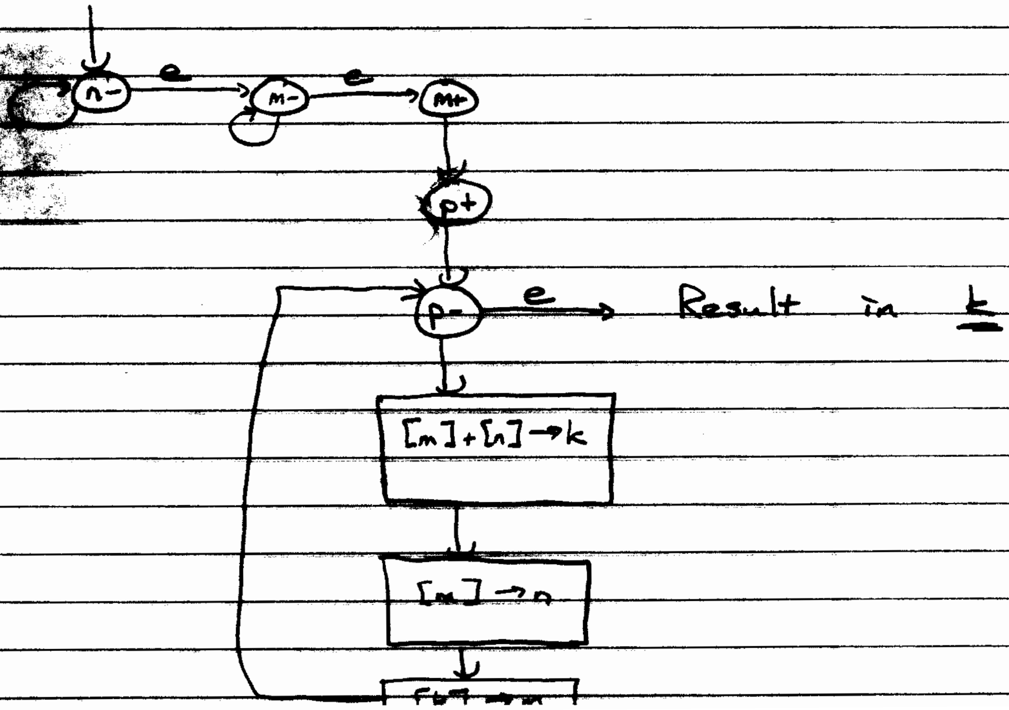
(I'll need p as a temporary register)



B. $[m_1] \cdot [m_2] \rightarrow n$



C. Fibonacci numbers — ONLY IF TIME



IV. Abacus-computable fcns

A. Def: A partial function $f : \mathbb{N}^k \to \mathbb{N}$ is <u>abacus-compltable</u> if there is an abacus progra which, when inputs $x_1, \ldots, x_k$ are placed in registers $1, \ldots, k$, leaves the output

$$f(x_1, \ldots, x_k) = [m]$$

(if any answer.)

B. Notation: $A^k_m(x_1, \ldots, x_k) = f(x_1, \ldots, x_k)$

C. <u>Examples</u>: Addition, multiplication, Fibonacci

V. Church's Thesis

A. All 3 classes of computable fcns seem to b intuitively computable.

B. Church's Thesis: "Intuitively" computable = Turing-computabl

C. Evidence: Turing-computable, abacus-computable, and μ-recursive are same

D. Proofs: <u>Simulation</u> of one program by another — e.g. show how to simulate any abacus-machine program by a TM.

$$\mu\text{-}R \underset{2}{\subseteq} A \underset{1}{\subseteq} T \underset{3}{\subseteq} \mu\text{-}R$$

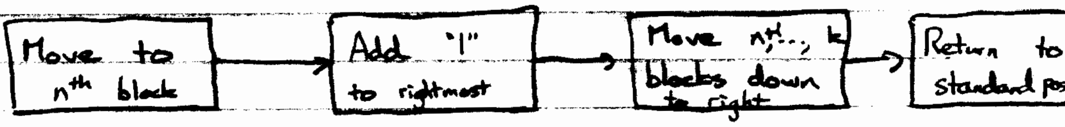
VI. Turing Simulates Abacus

A. Tape stores registers — All reg. past $k = 0$ (Program uses first k registers)

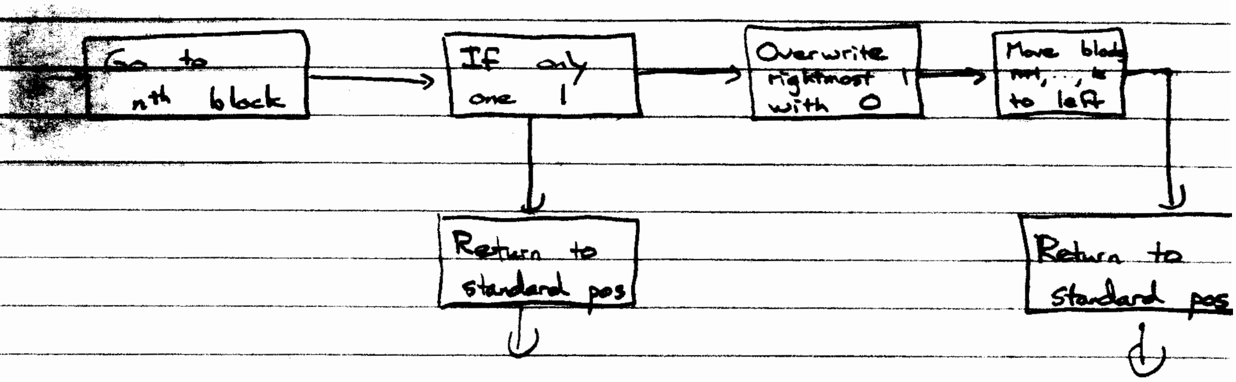
$$0\ 0\ 1^{[1]+1}\ 0\ 1^{[2]+1}\ 0 \ldots 0\ 1^{[k]+1}\ 0 \ldots$$

Standard position: leftmost 1

B. Increment Box n+



C. Decrement-and-test Box



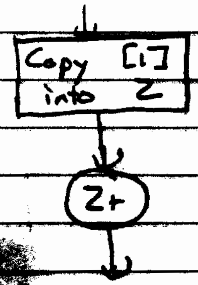
D. Final steps — Erase all but mth block

VII. Abacus Simulates μ-Recursive

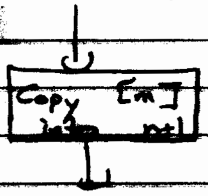
A. Assume input $\in \mathbb{N}^k$ in 1, …, k register
Put output into k+1 (0 initially; all others to
<u>Never</u> <u>change</u> <u>input</u>!

B. Simulate z : Argument in 1 — ignore!

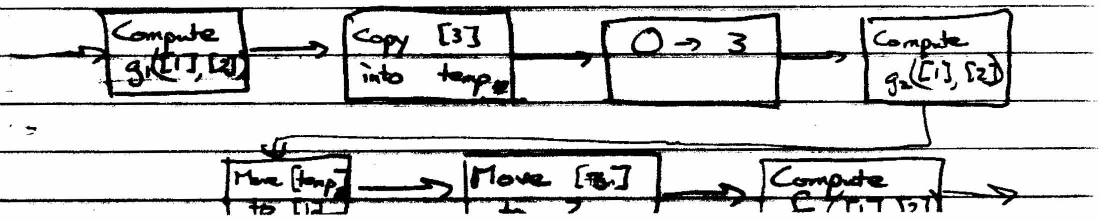
C. Simulate $s : \mathbb{N} \to \mathbb{N}$



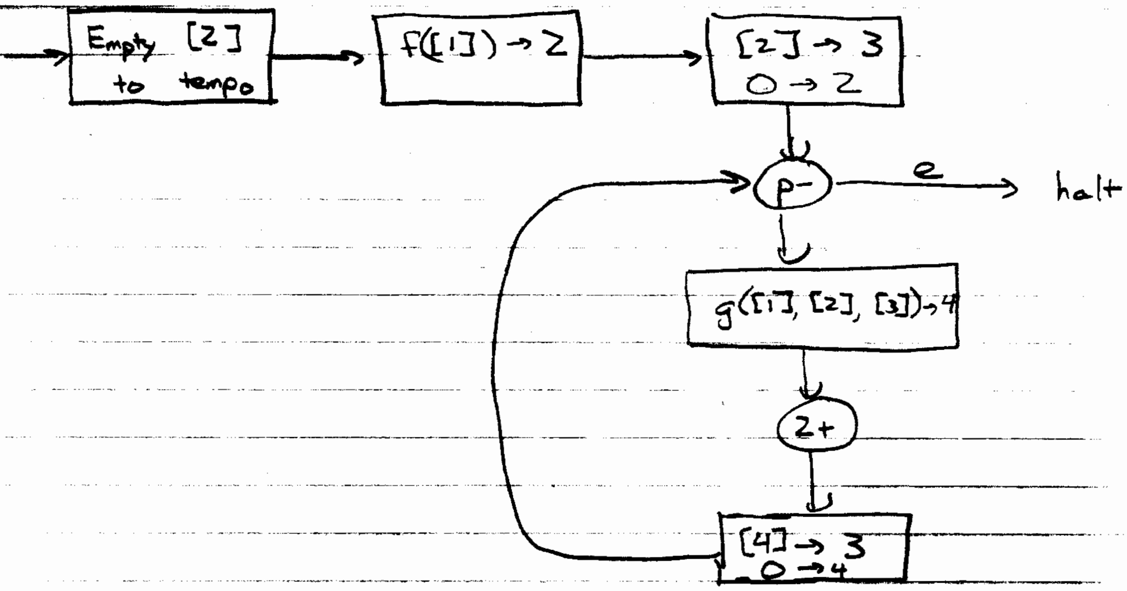
id^n_m



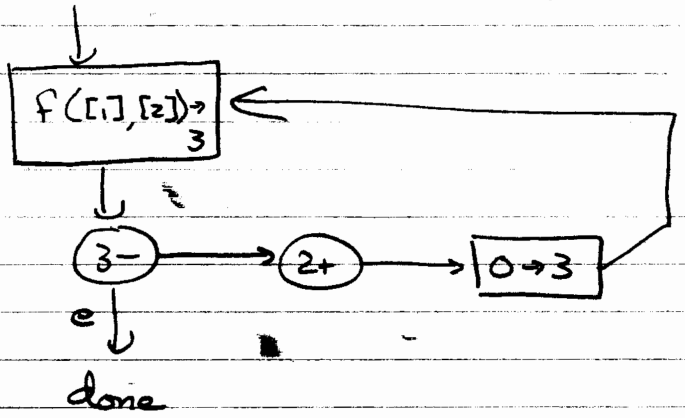
E. Simulate $Cn_{2,2}[f, g_1, g_2]$



F. Simulate $Pr_1[f, g]$ — inputs x_1, x_2



G. Simulate $Mn_1[f]$ — input x_1



Lecture 8 – μ-Recursive Simulates Turing

I. Introduction

A. Last time we talked about Church's Thesis and started a proof that adds a lot of evidence in favor of Church's Thesis:

$$\mu\text{-Rec} \subseteq A \subseteq T \subseteq \mu\text{-Rec}$$

We showed Turing simulates abacus.

B. Today:
- μ-Recursive Simulates Turing
- Kleene Normal Form Theorem
- Abacus simulates μ-Rec (if time)

C. Temptation: One may think any similar model of computation is equiv. to TMs.

1. Primitive rec. functions are not — proved this in your homework, that there was a computabl fcn not represented. Also, some computable fcns are part
2. Similarly, TMs that can only take linear time (eg.) are not equiv. to general TM even though they can have "undefined" valu

Main point: Theorem has a lot of content.

II. Overview of simulation – $\boxed{\text{Get } f(x_1, x_2)}$ computed by TM into μ-recursive form

A. Figure out how to encode the tape

B. Changing input and output of TM into this encoding

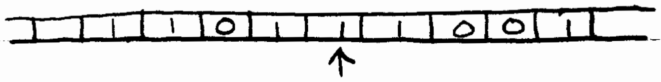
C. Encoding the transition relation — finite control — TM

D. [illegible] finite control together with encoded tape.

III. Encoding the tape

A. Could code as a single number — blanks n bla — but then couldn't keep tack of position of hea

B. The trick:



$\ell = 1101_2 = 13$
$r = 10011_2 = 19$ } Two #'s represent tape

C. How to move tape: 4 cases

- ℓ even, move left
- ℓ odd, move left (in book)
- r even, move right
- r odd, move right (we'll do)

Above example:

- Moving right chops off rightmost bit — hence, divide $r/2$ — prim. rec.
- Since r is odd, moving right means new $\ell := \ell * 2 + 1$ — prim. rec.

Try: $\ell = 13$, $r = 19$ $\longrightarrow$ $\ell = 11011_2 = 27$; $r = 100$

D. How to read tape: Just check to see if r is even! Thus need $e : \mathbb{N} \to \mathbb{N}$ with

$$e(x) = \begin{cases} 0 & \text{if } x \text{ even} \\ 1 & \text{if } x \text{ odd} \end{cases}$$

e is also prim. rec.

III. Changing input / output

A. Changing input into encoded numbers

- We're always at leftmost "1"
- Input looks like

$$\cdots 0\, 1^{x_1+1}\, 0\, 1^{x_2+1}\, 0 \cdots$$

- $\ell = 0$ initially
- $r = (2^{x_1+1} \dot{-} 1) + 2^{x_1+2}(2^{x_2+1} \dot{-} 1)$

This is prim. rec. — function is

$$s(x_1, x_2) = (2^{x_1+1} \dot{-} 1) + 2^{x_1+2}(2^{x_2+1} \dot{-} 1)$$

B. Changing output (encoded) into number

- Simulation will yield r = right of tape encoded as a binary number. Should be $2^{f(x_1,x_2)+1} - 1$
- Thus, take

$$lo(x) = \text{largest } w \text{ s.t. } 2^w \leq x$$

This is prim. rec. (bounded search)

- Note that $lo(r) = f(x_1, x_2)$.

V. Encoding the finite control

A. What happens on move? In a state, reading a symbol → new state, action

B. Two functions needed

1. $a(i, j) =$ 0 (write a 0)
   1 (write a 1)
   2 (move left)
   3 (move right)

   state symbol

2. $q(i, j)$ = new state

C. Halt state = 0 — so when entry in table is undefined, can recognize

D. $a(i,j)$ and $q(i,j)$ are prim. rec., sinc can be defined by cases.

VI. Putting it all together

A. Define a fcn $g(x_1, x_2, t)$ such that

$g(x_1, x_2, t)$ = snapshot of TM at step t in computation

We'll get it by prim. rec.

B. Need to encode snapshot — tape & state — by single number

Trick — Use $2^{l} 3^{i} 5^{r} = \langle l, i, r \rangle$

- Function $tpl(l, i, r) = 2^{l} 3^{i} 5^{r}$ is prim. rec.
- Also need to extract encodings

$lft(x)$ = greatest $w \le x$ with $2^{w} \le$

$ctr(x)$ = " " $3^{w} \le x$

$rgt(x)$ =

C. Starting off; initial config.

$g(x_1, x_2, 0) = tpl(0, 1, s(x_1, x_2))$

Next step,

$g(x_1, x_2, t+1) =$ let $l = lft(g(x_1, x_2, t))$

$r = rgt(g(x_1, x_2, t))$

$c = ctr(g(x_1, x_2, t))$

$q = q(c, e(r))$

$a = a(c, e(r))$

```
in
    if  $a = 0 \wedge (e(r)=0)$ then $tpl(\ell, c, r$
          ⋮
        $a = 1 \wedge (e(r)=0)$ then $tpl(\ell, c, r+$
          ⋮
                  (move right)
        $a = 3 \wedge (e(r) = 1)$ then
                                      $tpl(\ell * 2 + 1, c, r/2$
```

This is still prim. rec!

D. Final step — want to loop until machine halts, then read tape.

$$t' = \mu t. (ctr(g(x_1, x_2, t)) = 0)$$

Thus,

$$f(x_1, x_2) = lo(rgt(g(x_1, x_2, t'))) \; !$$

VII. Note: Kleene Normal Form — use only One least number operator

A. Translates to other models as well — can simulate TMs with only one outer loop; rest of loops have bounds on th

Intro Computability — Lecture 9, 10/2/89

Summary: "Simulation Thesis" - models can simulate each other up to ignoring efficiency. We'll ignore details of the model.

Today: Recursive and r.e. sets.

Synonyms: 1. Recursive Set = Decidable Set
2. R. E. Set = Half-decidable Set = Turing-acceptable = Turing-recogniz

Def: A set $S \subseteq \mathbb{N}$ is decidable iff $c_S : \mathbb{N} \to \{0,1\}$ is computable. (c_S = characteristic fcn)

Def: domain$(M) = \{n \in \mathbb{N} : M$ halts on input $n\}$ (M is a Turing machine)

Def: $R \subseteq \mathbb{N}$ is r.e. iff $R = \mathrm{dom}(M)$ for some M

Facts: (1) Recursive ⇒ r.e.

(2) S and $\overline{S}$ $(= \mathbb{N} - S)$ are both r.e. ⇒ S is recursive.

Proof: Run two machines for S and $\overline{S}$ in paralle Show that machine's answers are correct if th machine halts. Then show machine always halts

(3) Recursive ⇒ co-Recursive

Proof: Flip answers

(4) Thus, ⇔ in (2).

Question: Why is notion of "r.e." important? Logic and proving facts in a theory. Theorems are r.e. Non-r.e. sets cannot be captured by a logical proof system.

Note: Polynomial equality is decidable; polynomial unequality has no axiomatization! Hilbert's Tenth Problem:

$\{p(x_1, \ldots, x_n) : p$ has integer coeffs and an integer root $\}$

= Diophantine polynomials

This is undecidable! But it is r.e.!

R.e. Characterizations   Lecture 10, 10/4/89

Recall: Theorems are r.e. – can generate all true theorems

Theorem: The following are equivalent for any set $S \subseteq N$:

(1) S is r.e.

(2) S is domain of partial recursive fcn

(3) S is range " "

(4) $S = \emptyset$ or S is the range of f for some total recursive $f: N \to N$.

Proof: We'll do $4 \Rightarrow 3 \Rightarrow 2 \Rightarrow 1 \Rightarrow 4$.

($4 \Rightarrow 3$) This is trivial – if $S = \emptyset$, $S = \text{range}(\omega)$ where $\text{domain}(\omega) = \emptyset$; else $S = \text{range}(f)$ with f a partial recursive fcn.

($3 \Rightarrow 2$) Given $S = \text{range}(\varphi)$ where $\varphi: N \to N$ is a partial recursive fcn computed by a program P_φ. Program P' for $\varphi': N \to N$ s.t. $\text{dom}(\varphi') = S$.

"Given input n, begin computing *in parallel* (by time-slicing) $\varphi(0)$, $\varphi(1)$, --- using P_φ. In more detail,

```
for i = 0, 1, 2, ___   until $P_\varphi(\text{right}(i))$ halts w/ output n   (Than output 0)
    simulate left(i) steps on $P_\varphi$ on input right(i)
```

This is a dovetailing argument. Clearly, the quoted procedure is effective, so P' exists, and computes φ' s.t. $\text{domain}(\varphi') = S$ (and $\text{range}(\varphi') = \{0\}$.)

($2 \Rightarrow 1$) Have $S = \text{dom}(\varphi)$ and P_φ computes φ. Want $S = \{n : P' \text{ halts on input } n\}$, with a new P'. P' works as follows:

"Given input n, simulate P_φ on n until halt. If output is a well-fosed integer, then halt, else diverge."

($1 \Rightarrow 4$) Next time

More R.e. and Recursive — Lecture 11, 10/6/89

Lemma: S r.e. $\Rightarrow$ $S = \emptyset$ or $S = \text{range}(f)$ where $f: \mathbb{N} \to \mathbb{N}$ is total recursive

Proof. Assume wlog $5 \in S$. Have $S = \text{dom}(P)$ for some program P. We want to construct the appropriat f for above. To compute f:

> "Given input n, dovetail P on all inputs, but only run the overall dovetail process for n steps. If on this n^{th} step the process discovers that P halts on some input k, then output k Else, output 5."

We have to agree:

(1) What's in quotes clearly describes an effective process for computing a partial rec. fcn. f.

(2) f is total.

(3) $\text{range}(f) \subseteq S$

(4) $S \subseteq \text{range}(f)$

Proofs of these steps aren't hard.

An alternative way to compute f:

$$f(n) = \begin{cases} 5 & \text{if } P \text{ on input left}(n \text{ does not halt in exactly right}(n) \text{ steps} \\ \text{left}(n) & \text{o.w.} \end{cases}$$

☒

Theorem: S is r.e. iff S is finite or $S = \text{range}(f$ for some total recursive 1-1 function f'.

Proof: Same kind of argument as above.

☒

Theorem: Let $\text{Graph}(f) = \{(n,m) : n \in \text{dom}(f) \text{ and } f(n) = m$ for partial $f: \mathbb{N} \to \mathbb{N}$. f is partial recursive iff $\text{Graph}(f)$ is r.e.

Proof: Note that $\text{Graph}(f)$ is not $\subseteq \mathbb{N}$! But we can encode pairs (and other finite, familiar objects into $\mathbb{N}$; that's what we'll mean.

($\Leftarrow$) Suppose $\text{Graph}(f) = \text{domain}(P)$. To compute

> "Given input n, run P on (n,k) for $k = 0, 1, 2, \ldots$ in a dovetailed way. Output k if ever halt."

To see that program works, go back to definition of Graph(f).

($\Rightarrow$) Left as exercise.

☒

Def: Self-Halting Problem:

$$K_1 = \{ n \in \mathbb{N} : P_n \text{ halts on input } n \}$$

Programs. $P_0, P_1, P_2, \ldots$ Theorems will be *independen* of enumeration, as long as ordering is "sensible."

Theorem: $\overline{K_1}$ is *not* r.e.

Theorem: K_1 is r.e.

$\boxed{\text{Sets not R.e.}}$ Lecture 12, 10/11/89

Recall: $K_1 = \{ n : P_n$ halts on input $n \}$

Lemma: $\overline{K_1}$ is not r.e.

Proof: By contradiction — suppose $\overline{K_1} = \text{dom}(P_{n_0})$ for some $n_0 \in \mathbb{N}$. Wlog, $n_0 = 5$. By definition of $\overline{K_1}$

$n \in \overline{K_1}$ iff P_n on input n does not ha

In particular, by choice of 5

$n \in \overline{K_1}$ iff P_5 on input n does halt.

Thus,

P_5 on input 5 does not halt iff P_5 on input 5 does halt.

This is a contradiction, so $\overline{K_1}$ is not r.e.

$\boxtimes$

Remark: For "sensible" enumerations, K_1 is r.e. Appendix of Computability Notes gives formal definition

Theorem: K_1 is r.e. but not recursive.

Corollary: $K_0 \triangleq \{ (n,m) : P_n$ halts on input $m \}$ (The "General Halting Problem") is also r.e. but not recursive.

Proof: Reduction. $\boxtimes$

Note: "Subsets" does not preserve any properties in this setup.

Def: $A \leq_m B$ (where $A, B \subseteq \mathbb{N}$) iff there is a total recursive $f : \mathbb{N} \to \mathbb{N}$ s.t.

$n \in A$ iff $f(n) \in B$

Example: $K_1 \leq_m K_0$ via $f(n) = (n,n)$. Precisely, we'd have to encode pairs — so $f(n) = \text{pair}(n,n)$.

Lemma: $\leq_m$ is transitive and reflexive.

Proof: Reflexive is easy. Suppose $A \overset{f_1}{\leq_m} B$ & $B \overset{f_2}{\leq_m}$ C

Then $A \leq_m C$ by $f_3 = f_2 \circ f_1$.

$\boxtimes$

Lemma: Recursiveness Inherits Down

$A \leq_m^{cf} B$ & B recursive $\Rightarrow$ A recursive.

Proof: $c_A = c_B \circ f$. ☒

Lemma: $A \leq_m^{f} B$ & B is r.e. then A is r.

Proof: Write program which, on input n,

"Compute $f(\ )$ and run machine for B on it

That is, if $B = \text{domain}(g)$, $A = \text{domain}(g \circ f)$ (partial rec. g) ☒

Thus: R.e.-ness inherits down

Lemma: $A \leq_m B$ iff $\overline{A} \leq_m \overline{B}$.

Corollary: Neither $K_1 \leq_m \overline{K_1}$ nor $\overline{K_1} \leq_m K_1$.

Reducibility

Lecture 13, 10/16/89

Recall: We talked about many-one reducibility; $A \le_m B$. R.e. inherits downward; non-r.e. inherits up.

Corollary: If $\overline{K_0} \le_m B$, then B is not r.e.

Corollary: $K_0 \times \overline{K_0}$ is neither r.e. nor co-r.e.

Proof: Note that $\overline{K_0} \le_m K_0 \times \overline{K_0}$, using reduction fcn $f(n) = (5, n)$ — where, wlog, $5 \in K_0$. Thus, $K_0 \times \overline{K_0}$ is not r.e. Also, $K_0 \le_m K_0 \times \overline{K_0}$ similarly; thus, $K_0 \times \overline{K_0}$ is not co-r.e. ☒

Lemma: If A is r.e., then $A \le_m K_0$.

Proof: Say $A = \mathrm{dom}(P_n)$. Then use reduction function

$$f(m) = (n, m)$$

Then $n \in A$ iff $f(n) \in K_0$. ☒

Def: A set B is $\le_m$-complete for r.e. sets if (a) B is r.e.

(b) For any r.e. A, $A \le_m B$.

Recall: $K_1 \le_m K_0$. What about $K_0 \le_m K_1$?

Def: $K_2 = \{ M : M$ halts $\overset{\text{when started}}{\wedge}$ on blank tape$\}$

Alternatively, $K_2 = \{ P_n : P_n$ halts on input $0 \}$

Theorem: K_2 is $\le_m$-complete for r.e. sets.

Proof: Two steps:

(1) K_2 is r.e. (Obvious)

(2) Let A be $\mathrm{dom}(P_q)$ for program P_q. Want to show $A \le_m K_2$. For any constant $n \in \mathbb{N}$, define a procedure $P_{f(n)}$:

"Given input k, ignore k and act like P_q on input n."

Then

$$
\begin{aligned}
n \in A \quad & \text{iff} \quad n \in \mathrm{dom}(P_q) \\
& \text{iff} \quad P_{f(n)} \text{ halts on } \underline{\text{all}} \text{ inputs} \\
& \text{iff} \quad 0 \in \mathrm{dom}(P_{f(n)}) \\
& \text{iff} \quad f(n) \in K_2 .
\end{aligned}
$$

☒

$\boxed{K_2 \text{ is r.e.-complete}}$ Lecture 14, 10/18/89

Def: A set is diophantine iff there is a polynomial p s.t. $S = \{n : p(n, a_1, \ldots, a_k) = 0$ for some $a_1, \ldots, a_k \in \mathbb{N}\}$

Theorem: (Matijasevič) A set is r.e. iff it is diophantine.

This comes along with

Theorem: (Davis, Putnam, Robinson) A set is r.e. iff it is exponential diophantine.

Recall: $P_{f(n)}$: "Given input k, ignore it, and act like P_5 on input n" (P_5 is machine w/ domain A)

Thus, $n \in \mathrm{dom}(P_5)$ iff $0 \in \mathrm{dom}(P_{f(n)})$
iff $f(n) \in K_2$.

Also f is computable — why? Using Scheme, f is

```
(LAMBDA (n)
    (CODE '(LAMBDA (k)
                (P5 ,n))))
```

Thus, K_2 is r.e.-complete, and hence not recursive.

Note: $K_0 \equiv_m K_1 \equiv_m K_2$

Rice's Theorem: Any nontrivial property of the net behavior of a program is undecidable. Doesn't cover syntactic properties, nor running-times, etc.

Rice's Theorem   Lecture 15, 10/20/89

__Def:__ A __property__ of r.e. sets is __nontrivial__ if there is an r.e. set with the property, and an r.e. set without the property.

__Examples__
- Is empty?
- Contains 0?
- Is infinite?
- Is finite or contains an inf. # of even #s?
- Is finite or " "
  or contains an inf. # of odd #s?

All but last is a nontrivial property of r.e. sets.

__Def:__ $K_\wp = \{ P :$ P is a program and dom(P) has property $\wp \}$

__Theorem:__ (Rice) If $\wp$ is nontrivial, then $K_\wp$ is not recursive. In fact, if ϕ does not have property $\wp$, then $A \leq_m K_\wp$ for any r.e. set A.

__Note:__ Doesn't apply to all nonrecursive sets; e.g. K_1 contains a program P_1 but doesn't contain a program P_2 which has same domain as P_1.

__Proof:__ Suppose $\wp(\phi)$ holds. Let $\wp' \equiv \neg\wp$ and work with $\wp'$ instead below. ($K_\wp \equiv_m \overline{K_{\neg\wp}}$)

Suppose $\wp(\phi)$ does __not__ hold. Let A be any r.e. set. Consider the program $P_{g(n)}$: (where $\wp(R)$ holds for some r.e. R)

"Given input k,
(1) Save k for awhile, and see if $n \in A$.
(2) If n is in A, (diverging otherwise), then see if $k \in R$. (diverge if $k \notin R$)
(3) halt."

Clearly, the quoted spec. is programmable for any n. Moreover, g is clearly total recursive. Also

$n \notin A$ iff $\mathrm{dom}(P_{g(n)}) = \phi$ iff $g(n) \notin K_\wp$

$n \in A$ iff $\mathrm{dom}(P_{g(n)}) = R$ iff $g(n) \in K_\wp$

Thus $n \in A$ iff $g(n) \in K_\wp$, so $A \leq_m K_\wp$.

☒

Finish Rice's Theorem   Lecture 16, 10/23/89

Examples:

Properties of subsets of $\mathbb{N}$	Families of machines
• is nonempty	$K_{\text{non-empty}} = \{M : M$ halts on some integer$\}$
• has 0 as an element	$K_{[illegible]} = \{M : M$ halts on $0\}$
• is recursive	$K_{[illegible]} = \{M :$ dom(M) is recursive$\}$
• is finite	$K_{[illegible]} = \{M : M$ diverges on all suff. large integers$\}$
• (has no corr. property)	$\{M : M$ runs for ≤ 35 steps on input $11\}$
	$K = \{M : M$ halts on blank tape in an even # of steps$\}$

Lemma: $K_2 \le_m K$, hence $\overline{K}$ is not r.e.

Proof: $f(M) = M'$, where M' is specified by "Given input k, run M on input k but skip a beat — i.e. expand states into two."

Thus, $\overline{K}$ is not r.e. ☒

Example: $\{M :$ dom(M) is recognized by a TM with an even # of states$\}$

Rice's Thm is applicable — it's recursive.

Example: $\{M :$ dom(M) is also the domain of a machine with at most 10 states & 14 symbols$\}$

There are only a finite # of domains of this form, but an infinite # of r.e. sets.

Discussion of Decidability   Lecture 17, 10/25/89

<u>Fermat's Conjecture</u>: A <u>pythagorean triple</u> is a triple of three positive integers x, y, z s.t. $x^2 + y^2 = z^2$

A <u>k-pythagorean triple</u> is three pos. integers x, y, z with $x^k + y^k = z^k$.

The conjecture: the only k-pythagorean triples are those for $k \leq 2$.

<u>Define</u>:

$$f(k) = \begin{cases} 0 & \text{if there is a } j\text{-pythagorean triple for some } j \geq k+3 \\ 1 & \text{otherwise} \end{cases}$$

Let

$$Z_k(n) = \begin{cases} 0 & \text{if } n \leq k \\ 1 & \text{otherwise} \end{cases}$$

$$Z_\infty(n) = 0$$

<u>Claim</u>: f is in $\{Z_k\}_{k \geq 0} \cup \{Z_\infty\}$

<u>Proof</u>: Suppose there are infinitely many k's with a k-pythagorean triple — thus $f = Z_\infty$. Suppose not; then for largest k with a k-pythagorean triple, $f = Z_{k+1}$. Thus, f is computable. ☒

<u>Example</u>: $\{ M : dom(M) \supseteq \overline{K_0} \}$. Rice's Theorem applies — thu its not recursive.

$\{ M : dom(M) = \overline{K_0} \}$ is decidable — it's $\emptyset$

<u>Example</u>: $\{ M :$ dom $(M) = \{289\}$ <u>and</u> M has the min. # of states of any 2-letter alphabet machine whose domain is empty$\}$

$= \{ M :$ dom $(M) = \{289\}$ <u>and</u> M has 292 state and 2 symbols$\}$

It's finite, hence decidable.

$\boxed{\text{First-Order Logic}}$ Lecture 18, 10/27/89

__Language__: Of arithmetic

Name: $\underset{\sim}{0}$

Function symbols: $\underset{\sim}{\cdot}$, $\underset{\sim}{+}$, $\underset{\sim}{'}$

} Non-logical symbols

__Def__: "True in all interpretations" — or __valid__ — A sentence is __valid__ if it is true in all interpr. of the names & function symbols & relation symbols.

__Note__: We'll show that the valid formulas is r.e but not decidable. We'll also show true formulas of arithmetic is neither r.e. nor co-r.e. These are Gödel's Completeness and Incompleteness Theorems.

__Def__: __Invalid__ — not true in all interpretations.

__Grammar__: $\overset{\text{term}}{t} ::= \overset{\text{var}}{x} \mid \overset{\text{name}}{a} \mid f(t) \mid g(t_1, t_2) \mid \ldots$

__Def__: An __interpretation__ $\mathcal{I}$ consists of

1. A nonempty set called $D_{\mathcal{I}}$ (the __domain__)
2. An association, with every name a, an element $\mathcal{I}(a) \in D_{\mathcal{I}}$; with every function symbol f, a __total__ function $\mathcal{I}(f) : D_{\mathcal{I}}^{n} \to D_{\mathcal{I}}$, with $n =$ arity f; with every predicate P, a total predicate $\mathcal{I}(p) : D_{\mathcal{I}}^{n} \to \{0, 1\}$, with $n =$ arity P.

Interpretation of Sentences   Lecture 19, 10/30/89

<u>Recall</u>: We were defining meaning of terms

<u>Def</u>. $\mathcal{I}(t)$ for <u>closed</u> t:

<u>Case $t \equiv a$</u>: $\mathcal{I}(t) = \mathcal{I}(a)$

<u>Case $t \equiv f(t_1, t_2)$</u>: $\mathcal{I}(t) = \mathcal{I}(f)\,(\mathcal{I}(t_1), \mathcal{I}(t_2))$
(etc. for ~~higher order~~ n-ary function symbols)

<u>Formulas</u>: Given by grammar

$$F ::= p \mid P(t_1, t_2) \ldots \mid t_1 = t_2 \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F_1 \mid F_1 \rightarrow F_2 \mid F_1 \leftrightarrow F_2 \mid \forall x\ F \mid \exists x\ F$$

sentence letter

predicate symbols

<u>atomic formulae</u>

<u>Example</u>: Of free & bound variables

$$F_0 \equiv f(a,a) \neq a \rightarrow \exists x_2\, \exists x_3\, [a = f(x_2, x_3) \wedge b \neq x_2 \wedge f(x_2, x_2) \neq x_2 \wedge \forall x_1\, \forall x_3\, (x_2 = f(x_1, x_3) \rightarrow (x_2 = x_1 \vee x_2 = x_3))]$$

<u>Def</u>: A <u>sentence</u> is a formula w/ no free variables.

<u>Def</u>: For a <u>sentence</u> F, define $\mathcal{I}(F)$ by cases:

<u>$F \equiv p$</u>: $\mathcal{I}(F) = \mathcal{I}(p)$

<u>$F \equiv P(t_1, t_2)$</u>: $\mathcal{I}(t_i)$ are defined; then
$\mathcal{I}(F) = \mathcal{I}(P)\,(\mathcal{I}(t_1), \mathcal{I}(t_2))$

<u>$F \equiv t_1 = t_2$</u>: $\mathcal{I}(t_i)$ are defined; then

$$\mathcal{I}(F) = \begin{cases} 1 & \text{if } \mathcal{I}(t_1) = \mathcal{I}(t_2) \\ 0 & \text{otherwise} \end{cases}$$

<u>$F \equiv F_1 \wedge F_2$</u>: $\mathcal{I}(F_i)$ are defined; then

$$\mathcal{I}(F) = \begin{cases} 1 & \text{if } \mathcal{I}(F_1) = \mathcal{I}(F_2) = 1 \\ 0 & \text{otherwise} \end{cases}$$

<u>$F \equiv F_1 \rightarrow F_2$</u>:

$$\mathcal{I}(F) = \begin{cases} 1 & \text{if } \mathcal{I}(F_1) = 0, \\ 1 & \text{if } \mathcal{I}(F_2) = 1, \\ 0 & \text{otherwise} \end{cases}$$

$\underline{\forall_x F}$: $\mathcal{I}(\forall_x F) = \begin{cases} 1 & \text{if } \forall o.\ \mathcal{I}_o^a(F_x\, a) = 1 \\ 0 & \text{otherwise} \end{cases}$

where "a" is a fresh name, and $\mathcal{I}_o^a$ works same as $\mathcal{I}$ excapt on a, with $\mathcal{I}(a) = o$.

$\underline{\exists_x F}$: $\mathcal{I}(\exists_x F) = \begin{cases} 1 & \text{if there is some } o \text{ w/ } \mathcal{I}_o^a(F_x a) = 1 \\ 0 & \text{otherwise} \end{cases}$

Think about: Recursiveness of truth over Finite domains

Logical Equivalence

Lecture 20, 11/1/89

Recall: Interpretation of sentences — $\mathcal{I}(S) = 1$. We say "S is true under (in) interpretation $\mathcal{I}$" or $\mathcal{I}$ is a <u>model</u> of S.

Def: $\vdash S$ iff $\mathcal{I}(S) = 1$ for all $\mathcal{I}$. We say S is then <u>valid</u>.

Example: $\vdash S \vee \neg S$

$\left.\vdash (\forall x.\ P\,x) \rightarrow (\exists x.\ P\,x)\right\} S$

Proof: Of second. Need to show $\mathcal{I}(S) = 1$.

<u>Case 1</u>: $\mathcal{I}(\forall x.\ P\,x) = 0$. Then $\mathcal{I}(S) = 1$ by rule for $\mathcal{I}(\rightarrow)$

<u>Case 2</u>: $\mathcal{I}(\forall x.\ P\,x) = 1$. Then $\mathcal{I}^a_o((P_x)_x\,a) = 1$ for all $o \in D_{\mathcal{I}}$ by def. of $\mathcal{I}(\forall)$. Thus $\mathcal{I}^a_o(P\,a) = 1$ for all o. Choose an $o_1 \in D_{\mathcal{I}}$; possible since $D_{\mathcal{I}}$ is nonempty. Then $\mathcal{I}^a_{o_1}(P\,a) = 1$. Thus $\mathcal{I}(\exists x.\ P\,x) = 1$ by def. of $\mathcal{I}(\exists)$. Thus, $\mathcal{I}(S) = 1$.

Thus, $\mathcal{I}(S) = 1$.

Def: $F_1 \simeq F_2$ (are <u>logically</u> <u>equivalent</u>.) Let $F_1^* = F_1$ with "fresh" names replacing free variables.

E.g. $F_1 \equiv f(x_1, x_2) = x_1 \wedge \forall x_1\ x_2 = x_1$

$F_1^* \equiv f(a_1, a_2) = a_1 \wedge \forall x_1\ a_2 = x_1$

For same $(^*)$,

$$F_1 \simeq F_2 \quad \text{iff} \quad \mathcal{I}(F_1^*) = \mathcal{I}(F_2^*)$$

Example: $\neg(\exists x.\ F) \simeq (\forall x.\ \neg F)$

Proof: $\mathcal{I}((\neg \exists x.\ F)^*) = 1$

iff $\mathcal{I}(\neg(\exists x.\ F)^*) = 1$

iff $\mathcal{I}((\exists x.\ F)^*) = 0$

iff $\mathcal{I}(\exists x.\ (F)^*) = 0$

iff it's not true that "$\mathcal{I}^a_o(F^{\#}_x\ a) = 1$ for some $o \in D_{\mathcal{I}}$"

iff $\mathcal{I}^a_o((F^*)_x\ a) = 0$ for all $o \in D_{\mathcal{I}}$

iff $\mathcal{I}^a_o(\neg((F^*_x)\ a)) = 1$ " "

iff $\mathcal{I}^a_o((\neg F^*)_x\ a) = 1$ " "

iff $\mathcal{I}((\forall x.\ \neg F)^*) = 1$. ☒

Prenex Normal Form

Lecture 21, 11/3/89

Recall: $\neg \exists_x F \simeq \forall_x \neg F$

$(\forall_x F) \vee G \simeq \forall_x (F \vee G)$ $\quad x \notin \text{Free Var}(G)$

$(\exists_x F) \vee G \simeq \exists_x (F \vee G)$ $\quad$ " $\quad$ "

Also works for F & G reversed; also works $\wedge$.

More Equivalences: $Q \in \{\forall, \exists\}$

1. $Q_x F \simeq Q_y (F_x\ y)$ $\quad$ (y is fresh)

   (renaming bound variables)

2. Congruence rules – e.g.

   $F_1 \simeq F_2 \Rightarrow Q_x F_1 \simeq Q_x F_2$

3. $\neg \forall_x F \simeq \exists_x \neg F$

4. Boolean equivalences – e.g. $F \to G \simeq (\neg F \vee G)$

Prenex Procedure: $\quad$ Example:

$$\forall_y [\neg \exists_x P_{xy} \leftrightarrow \forall_z P_{zz}]$$

$$\simeq \forall_y [((\neg\neg \exists_x P_{xy}) \vee (\forall_z P_{zz})) \wedge ((\neg \forall_z P_{zz}) \vee (\neg \exists_x P_{xy}))]$$

$$\simeq \forall_y [((\neg\neg \exists_x P_{xy}) \vee (\forall_z P_{zz})) \wedge \forall_x((\neg \forall_z P_{zz}) \vee (\neg P_{xy}))]$$

$$\simeq \forall_y \forall_x [(\ \_\_\ ) \wedge ((\neg \forall_z P_{zz}) \vee (\neg P_{xy}))]$$

$$\simeq \forall_y \forall_x \exists_{x_1} \forall_z [(\neg\neg P_{x_1 y} \vee P_{zz}) \wedge (\neg \forall_z P_{zz} \vee \neg P_{xy})]$$

$$\simeq \forall_y \forall_x \exists_{x_1} \forall_z \exists_{z_1} [(\neg\neg P_{x_1 y} \vee P_{zz}) \wedge (P_{z_1 z_1}) \vee \neg P_{xy})]$$

$\boxed{\text{Arithmetic \& Interpretations}}$ Lecture 22, 11/6/89

Lemma: If $\vec{x} = FV(F_1) \cup FV(F_2)$, then $F_1 \simeq F_2$ iff $\vdash \forall \vec{x}.\ F_1 \Leftrightarrow F_2$.

Notation: $\vdash S$ is notation for "S is valid." If Γ is a set of sentences then $\Gamma \vdash S$ iff $\mathcal{I}(S)=1$ for all $\mathcal{I}$ s.t. $\mathcal{I}(S')=1$ $\forall S' \in \Gamma$.

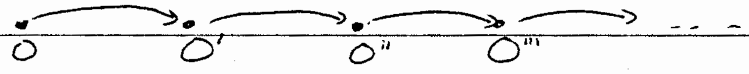
Remark: If Γ is finite, let $\bigwedge \Gamma$ be the conjunction of all sentences in Γ. Then $\Gamma \vdash S$ iff $\vdash \bigwedge\Gamma \rightarrow S$.

If Γ is a set of formulas and F is a formula, $\Gamma \vdash F$ iff $\vdash \forall \vec{x}\ (\bigwedge \Gamma \rightarrow F)$

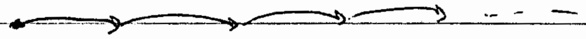
Arithmetic: Let (language = $(', 0, +, *)$)

$$\text{Min Arith}_1 = [\forall x\ \forall y\ ((x' = y') \rightarrow x = y)] \wedge [\forall x\ (0 \neq x')] \wedge [\forall x\ (x \neq 0 \rightarrow \exists y\ (x = y'))]$$

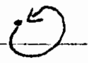
Possible interpretations: $\rightarrow$ = $'$ function



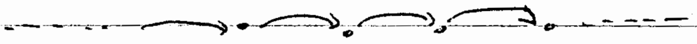
In any interpretation satisfying first two sentences, no → enters 0 (second axiom) and no point has two arrows into it. We get a copy of the natural #s here. Could have another chain



but third axiom rules this out. But can have



as extra elts. Can also have chains like



and any # of copies. Wow!

Let: $\text{Min Arith} = \text{Min Arith}_1 \wedge [\forall x\ (x+0) = x] \wedge [\forall x,y\ \ x+(y') = (x+y)'] \wedge$ Axioms for mult

Addition is not commutative

Undecidability of First-Order Logic

Lecture 23, 11/8/89

Theorem: The set of valid sentences of first-order logic is undecidable.

Proof: Due to Büchi. We'll reduce $K_{z} \leq_m$ Valid. Given a TM M with states $Q_0, \ldots, Q_n$ and symbols $S_0, \ldots, S_k$, we'll map this to a sentence over the language

$$\underbrace{Q_0, \ldots, Q_n, S_0, \ldots, S_k, <}_{\text{binary predicates}},\ \overset{\leftarrow \text{unary fcn}}{'},\ 0$$

Think of domain $(\mathcal{D}) = \mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$;

Think of $t\,Q_i\,x$ as meaning that $t > 0$, and in step t of M's computation on input 0, M is in state Q_i reading the x^{th} tape square. Also, $t\,S_j\,x$ means that at step t, the y^{th} tape square contains symbol S_j.

Our formula will be S_M with

- If "integer-like" and "Q_i's and S_j's are M-like", then "M halts on input 0"

Programming "M-like":

1. $0\ Q_0\ 0$ — at time 0 in state Q_0, scanning 0 sq.
2. $\forall x.\ 0\ S_0\ x$ — have all blanks at time 0
3. $\forall t.\ \forall x\ \forall y\ [t\,Q_i\,x \wedge t\,Q_i\,y \rightarrow x = y]$ (one for each i) — says can be in at most one ~~state~~ tape square
4. $\forall t\ \forall x\ \forall y\ [t\,Q_j\,x \rightarrow \bigwedge_{i \neq j} \neg(t\,Q_i\,y)]$ (for all j) — says can be in at most one state
5. $\forall t\ \forall x\ (t\,S_j\,x \rightarrow \bigwedge_{i \neq j} \neg(t\,S_i\,x))$ — can be at most one symbol per square
6. $\forall t\ \forall x\ [\bigvee_{j=0}^{t} t\,S_j\,x]$ — at least one symbol per square

Now to program flow graph of TM,

$$\textcircled{i} \xrightarrow{S_j : S_k} \textcircled{m} \;=\; \forall t\ \forall x\ ((t\,Q_i\,x \wedge t\,S_j\,x) \rightarrow (t'\,Q_m\,x \wedge t'\,S_k\,x)) \wedge (\bigwedge_{\ell=0}^{k} \forall y\ y \neq x \rightarrow (t\,S_\ell\,y \rightarrow t'\,S_\ell\,y))$$

Can keep going with this.

Now "M halts": $\exists x\ \exists t\ [t\,Q_{halt}\,x]$

Lecture 24: Refutation System

I. Introduction

A. Reading: Chapter 11 of text

B. Review

1. Last time we showed that valid sentences are not decidable by showing $K_2 \leq_m \{\text{Valid sentences}\}$

2. This says valid sentences are ... not co-r.e. Fortunately, says nothing about non-r.e-ne

C. Outline for today:

1. Proof systems in general
2. Refutation system
3. Expansions of interpretations

II. Proof systems - Mechanical (mindless) way of proving theorems; should be easily checka

A. Usual form: Start with a set of axioms (judiciously chosen); these sentences should all be valid. Then need a set of proof rules, ways of generating more valid sentence from old.

1. Example axioms:

a. $F_1 \wedge F_2 \rightarrow F_2 \wedge F_1$ (propositional tautology)

b. $(\forall x . F) \rightarrow F_x t$, where $F_x t$ is F, with x replaced by any term (with renaming to avoid capture by quantifiers.

2. Rule: Modus ponens

$$\frac{F \qquad F \rightarrow F'}{F'}$$

3. This is way mathematicians (and you) do proofs

B. Our form: Proof by refutation.

1. Negate what we wish to prove; show that we must reach an absurdity.

2. Why study?

a. System needs only two rules (and nc axioms)

b. Slightly easier to work with than full system

C. Properties of proof system that we will prove

Prove exactly the valid sentences

1. Soundness: If we have a refutation, then sentence is not valid.
2. Completeness: If a sentence is not valid, we can show by a refutation
3. Others: Compactness, Skolem – Löwenheim theore

D. Terminology – Encourage you to keep straight

1. Unsatisfiable – S is <u>unsatisfiable</u> iff it has no models. Thus, if S is unsatisfiable, $(\neg S)$ is <u>valid</u>.
   This is book's terminology
2. Satisfiable – S is <u>satisfiable</u> iff it has a model. Thus, S is satisfiable iff $(\neg S)$ is <u>not</u> valid.

III. Refutation system

A. More general: Show that a set Δ of sentences has no model – i.e. the set is unsatisfiable.

B. Refutation proof –

1. Each line is

   n F "reason"

   A "reason" is either "Δ" – meaning $F \in \Delta$ – or a number k less than n – meaning n follows from k by a "yields" rule. Obtain a prop. contradiction in end

2. The two "yields" rules:

   a. UI – <u>Universal instantiation</u>. The line k is

   k $\forall x.F$ reason

   yields

   n $F_x t$ k

   where t is <u>any</u> closed term (book says any term, but they mean closed)

b. EI - <u>existential instantiation</u>. The like k ·

k $\exists x.\ F$ reason

yields

n $F_x\ a$ k

where "a" is a name that does nc appear in any previous line.

We're naming what makes $\exists x.\ F$ true — hence, <u>no</u> assumptions should be made about what this is. That's why we pick "fresh" name

C. Examples

1. Prove $(\forall x.\ x \geq 0) \rightarrow (0''' \geq 0)$

a. Step 1 — negate and convert to prenex nf

$$-((-\forall x.\ x \geq 0) \vee (0''' \geq 0))$$

Skip this

gives

$$(\forall x.\ x \geq 0) \wedge -(0''' \geq 0)$$

gives

<u>Proof</u>:

1	$(\forall x.\ (x \geq 0 \wedge -(0''' \geq 0)))$	Δ
2	$(0''' \geq 0) \wedge -(0''' \geq 0)$	1

Done!

2. That's pretty boring — slightly less boring:

$$(\forall x\ \forall y\ P\ x\ y) \rightarrow (\forall y\ P\ y\ y)$$

Convert to <u>set</u> of formulas this time

$$(\forall x\ \forall y\ P\ x\ y)\ ,\quad -(\forall y\ P\ y\ y)$$
$$= (\exists y\ -P\ y\ y)$$

Proof:

1	$\exists y\ -P\ y\ y$	Δ
2	$-P\ a\ a$	1
3	$\forall x\ \forall y\ P\ x\ y$	Δ
4	$\forall y\ P\ a\ y$	3
5	$P\ a\ a$	4

Contradiction!

Note that cannot do in a different order. Thus, we can use some cleverness, although our machine won't have to

3. Final example:

$$\exists x \, \forall y \, (y + x = y) \quad \wedge \quad \forall x \, \forall y \quad x + y = y + x \quad \rightarrow$$
$$\exists x \, \forall y \, (x + y = y)$$

Set $\Delta =$

$$\exists x \, \forall y \, (y + x = y) \, , \quad \forall x \, \forall y \quad x + y = y + x \, ,$$
$$\forall x \, \exists y \, -(x + y = y)$$

Proof:

1	$\exists x \, \forall y \, (y + x = y)$	Δ
2	$\forall y \, (y + a = y)$	1
3	$\forall x \, \exists y \, -(x + y = y)$	Δ
4	$\exists y \, -(a + y = y)$	3
5	$-(a + b = b)$	4
6	$(b + a = b)$	2
7	$\forall x \, \forall y \, (x + y = y + x)$	Δ
8	$\forall y \, (a + y = y + a)$	7
9	$(a + b = b + a)$	8

Lecture 25 - Soundness of Refutation System

I. Introduction

A. Reading: Chapter 11 of text

B. Review — UI, EI are "yields" rules

Example: $\exists x \forall y (Py \rightarrow y = x)$, $-\exists x Px = \forall x -$f

1	$\exists x \forall y (Py \rightarrow y = x)$	Δ
2	$\forall x (-Px)$	Δ
3	$\forall y (Py \rightarrow y = a)$	1
4	$Pa \rightarrow a = a$	3
5	$-Pa$	2

Derivation

Can be infinite

Note: Legal to insert "bogus" lines!

C. Outline

1. Soundness
2. Decidability of unsatisfiabilty of q.f. sententces
3. Completeness - outline

II. Thm: (Soundness) If Δ has a refutation, Δ is unsatifiable.

A. UI is a perfectly natural rule — in fact, many logics include it

However, EI is not a natural rule! Does n preserve validity. Example:

$\forall y \exists x (Py \leftrightarrow y = x)$ is valid

$(Pa \leftrightarrow a = b)$ is not

B. We're doing refutations — want to reach something that's unsatisfiable. So with our rules, if we reach an unsatisfiable set of quantifier-free sentences, we want our Δ to be unsatisfiable

This is soundness.

C. Def: Let $\mathcal{I}$ be an interpretation. An expansio $\mathcal{J}$ of $\mathcal{I}$ is any interpretation with domain equal to $\mathcal{I}$'s, and assigns names, fcn symbols, predicates the same as $\mathcal{I}$, but possibly more.

Example: $D_{\mathcal{I}} = \mathbb{N}$, $\mathcal{I}(\underset{\sim}{0}) = 0$.

$\mathcal{J} = \mathcal{I}^{a}_{0}$

$\mathcal{J}' = \mathcal{J}$ with $\mathcal{J}'(\,'\,) =$ successor fcr

D. Lemma 1: (Basic Prop) Suppose Γ = set of sent. satisfiable in $\mathcal{I}$. Let $F_x a$ be conclusion of application o EI from $(\exists x F) \in \Gamma$. (name a is fresh) Then there is some $o \in D_{\mathcal{I}}$ with $\mathcal{I} = \mathcal{I}^a_o$ a model of $\Gamma \cup \{F_x a\}$

Proof: Let $\mathcal{I} = \mathcal{I}^a_o$. Then we know that since a does not occur in Γ, $\mathcal{I}$ satisfies Γ. Also, $\mathcal{I}(F_x a) = 1$, since o is chosen to satisfy the existential. ☒

E. Lemma 2: (Main Lemma) If $\mathcal{I}$ is a model of Δ and $\mathcal{D}$ is a derivation from Δ, then set of sentences occuring in $\mathcal{D}$ has a model

F. Proof: Of soundness. Suppose $\mathcal{D}$ is a refutatio from Δ. If $\mathcal{I}$ were a model of Δ, by Lemma 2 sentences in $\mathcal{D}$ would have a model. But a finite set of sentences in $\mathcal{D}$ don't have a model. Thus, Δ has no mod i.e. Δ is unsatisfiable. ☒

G. Proof: Of Lemma 2. Define

$$\Delta_0 = \Delta$$

$$\Delta_{i+1} = \Delta_i \cup \{S_{i+1}\}$$

Where S_i = sentence in ith line of derivatio

We'll define an interpretation (an expansion) for each Δ_n — this will be used to get one for whole set of sentences in $\mathcal{D}$. (Might have infinite # of names, fcn symbols, though — but could have had that to start.) its a model of Δ_0.

Define $\mathcal{I}_0 = \mathcal{I}$; and $\mathcal{I}_{k+1}$ as follows

Case 1: S_{k+1} has reason Δ in derivatio

Then $\mathcal{I}_{k+1} = \mathcal{I}_k$; its a model of Δ_{k+1} then trivially

Case 2: S_{k+1} has reason $j < k+1$, and follo by UI. Then $S_{k+1} = F_x t$.

If $\mathcal{D}_k$ gives meaning to all fcn symbols & constants in t, set $\mathcal{D}_{k+1} = \mathcal{D}_k$. Then $\mathcal{D}_{k+1}$ is a model of Δ_{k+1}. Otherwise, let $d \in D_{\mathcal{D}}$. Then let $\mathcal{D}_{k+1}$ assign each "new" constant a the value d, and each "new" fcn symbol f the constant function d. Then $\mathcal{D}_{k+1}$ is a model of Δ_{k+1}.

Case 3: S_{k+1} has reason $j < k+1$, and follows by EI. Then $S_{k+1} = F_x\, a$, with a fresh. By Lemma 1, then, $(\mathcal{D}_k)^{a}_{o}$, for some $o \in D_{\mathcal{D}}$, is a model of Δ_{k+1}.

Let $\mathcal{L}$ = jamming together of $\mathcal{D}_k$'s — i.e.

$$D_{\mathcal{L}} = D_{\mathcal{D}}$$

and $\mathcal{L}$ gives same meaning as $\mathcal{D}_{k+1}$ to any constant of fcn symbol in S_{k+1}, and gives same meaning as $\mathcal{D}$ to symbols in Δ. Then $\mathcal{L}$ is a model of every sentence in $\mathfrak{D}$, so done. ☒

III. Recursive check for unsat. of finite set of q.f. sentences

A. Why important!

B. Special case — without fcn symbols or equality

Example:

$(P\ a\ b) \wedge (P\ b\ c)$,

$(P\ a\ c) \vee -(P\ a\ b) \vee -(P\ b\ c)$,

$-(P\ a\ c)$

Replace by distinct sentence letters —

$A \wedge B$

$C \vee -A \vee -B$

$-C$

Use truth tables

C. With equality and fcn symbols —

Let n = # of distinct terms in finite set; e.g.

$0' + a = a$

has terms 0, $0'$, a, $0' + a$.

Look at <u>all</u> interpretations with $\leq n$ elements in domain. If unsatisfiable here, it doesn't matter how many elts are in domain! they will always be unsatisfiable, cause I can never refer to extra elements.

<u>Proof</u>: Do formally (maybe)

IV. Outline of Completeness

A. <u>Thm</u>: If Δ is unsatisfiable, there is a refutation.

B. Canonical derivations

C. Proof of completeness: If no refutation, build a model for Δ.

Lecture 26 - Completeness

I. Introduction

A. Reading : Chapter 12

B. Review

1. Last time we covered
   Thm: If Δ has a refutation, Δ is unsatisfiable.
2. Proved by showing that if Δ is satisfiable then any derivation is satisfiable.

C. Today : Completeness - An important theorem!
   Thm: If Δ is unsatisfiable, then Δ has a refutation.

D. Our mission : If Δ has no refutation, Δ is sat.

1. Find a derivation $\mathfrak{D}$ that will give us a refutation if Δ has one
2. Build a model of Δ if no refutation (i.e. no finite set of q f sentences in $\mathfrak{D}$ is unsat.
   This will be tricky part

II. Canonical Derivations

A. Df: Let Δ be a set of sentences, $\mathfrak{D}$ a derivation from Δ. Then $\mathfrak{D}$ is canonical if

(1) Every sent. in Δ is in $\mathfrak{D}$

(2) If $\exists x\, F$ occurs in $\mathfrak{D}$, then $F_x t$ (some t closed) occurs in $\mathfrak{D}$

(3) If $\forall x\, F$ " " , then "
"

(4) If $\forall x\, F$ occurs in $\mathfrak{D}$, then for every any term t from names and fcn symbols : $\mathfrak{D}$, $F_x t$ occurs in $\mathfrak{D}$

(5) $\mathfrak{D}$ uses only fcn symbols in Δ

B. Idea: Every possible conclusion of UI or E: from fcns symbols in Δ, is in the canonical derivation. We won't need new fcn symbols (this makes sense

C. <u>Lemma 1</u>: Every Δ has a canonical derivation.
<u>Proof</u>: Generate $\mathcal{D}$ as follows:

"At stage i,

a. Write down a new member of Δ

b. Apply EI as much as possible, subject to restrictions

c. Apply UI as much " " "
" " "

Restrictions:

1. No sentence appears twice in $\mathcal{D}$
2. No sentence may be used as premise of E: more than once
3. At stage i, we always use <u>less</u> than i function symbols in instantial terms of U
Also, only use names and fcn symbols of previous lines

<u>Example</u>:

Stage 1	1	$\exists x\ \forall y\ (Py \to y = x)$	Δ
	2.	$\forall y\ (Pa \to y = a)$	1
	3	$Pa \to a = a$	2
Stage 2	4	$\exists x\ P(f\ x)$	Δ
	5	$P(f\ b)$	4
	6	$Pa \to (f\ a) = a$	2
	7	$Pa \to (f\ b) = b$	2

⋮

One can check that each stage terminates; also, one can check each part of def. of canonical. ☒

D. <u>Def</u>: Let Γ be a set of <u>qf</u> sentences. Then $\mathcal{I}$ <u>matches</u> Γ if $\mathcal{I}$ is a model of Γ, and if any terms occur in Γ, then every elt of $D_{\mathcal{I}}$ is denoted by some term.
We can "express" all elements of domain.

E. <u>Lemma 2</u>: Suppose $\mathcal{D}$ from Δ is canonical and Γ is set of all qf sentences in $\mathcal{D}$ and $\mathcal{I}$ matches Γ. Then $\mathcal{I}$ models <u>all</u> sent in $\mathcal{D}$.

<u>Proof</u>: By contradiction — suppose $\mathcal{I}$ assigns false to a sentence in Γ. Then let S be a sentence with <u>minimal</u> <u>length</u> with $\mathcal{I}(s) =$ S must be either $\exists x\ F$ or $\forall x\ F$. If first, since

$$F_x\, t \qquad (t \text{ closed})$$

is in $\mathfrak{D}$ (by "canonical"), $\mathcal{I}(F_x\, t) = 1$. Thus, $\mathcal{I}(\exists x\ t) = 1$, a contradiction. So $S = \forall x\ F$. But then (by "canonical")

$$F_x\, t$$

occurs in $\mathfrak{D}$ for <u>any</u> term t built from constants and fcns in $\mathfrak{D}$. Since $\mathcal{I}$ matches Γ, every elt of $D_{\mathcal{I}}$ is given by term. Thus, since $\mathcal{I}(F_x\, t) = 1$, $\mathcal{I}(\forall x\ F) = 1$. →←. ☒

III. Building model

A. Where have we come —

a. Canonical derivation

b. Models of Γ that match Γ will model Δ

Now all we need is to construct $\mathcal{I}$ s.t.

"if every finite subset of Γ is satis, then $\mathcal{I}$ matches Γ"

Thus, we'll have a model of Δ then $\nexists$ <u>no</u> refutation

B. Equivalence relations

1. Axioms — $x R x$ $\quad x R y \Rightarrow y R x$

$x R y$ and $y R z \Rightarrow x R z$

2. <u>Def</u>: Let R be an equiv. relation. Then $[x] = \{y : x R y\}$

3. <u>Fact 1</u>: $x \in [x]$.

<u>Proof</u>: Since $x R x$, $x \in [x]$. ☒

4. <u>Fact 2</u>: $x R y$ iff $[x] = [y]$
<u>Proof</u>: Suppose $[x] = [y]$. Then $y \in [x]$ so $x R y$. Other is easy also. ☒

C. OK sets

1. <u>Def</u>: Let Θ be a set of sent. Θ is <u>O</u> if every finite subset of Θ is satisf.

2. <u>Fact 3</u>: Let Θ be OK. Then for a sent S, either $\Theta \cup \{S\}$ is OK or $\Theta \cup \{-S\}$ is.
<u>Proof</u>: Suppose neither is true. Then

$\{A_1, \ldots, A_n, S\}$ is unsat
$\{B_1, \ldots, B_m, -S\}$ is unsat

Thus, $\{A_1, \ldots, A_n, B_1, \ldots, B_m\}$ is unsat
$\rightarrow\leftarrow$. ☒

D. <u>Main Lemma 3</u>: If Γ is an enumerable, OK set of qf sentences, then there is an interp. $\mathcal{I}$ matching Γ.

<u>Proof</u>: This is a long one! We will give most of t highlights, but skip some details; read about them!
Let T = set of terms in Γ (<u>including</u> <u>subterr</u>
Let

$$A_1, A_2, A_3, \ldots$$

be the closed <u>atomic</u> formula built from either sentence letters or predicates applied to T, where sent. letters or predicates occur in Γ, or $t_1 = t_2$.
Define $\Gamma_1 = \Gamma$, and let

$$\Gamma_{n+1} = \begin{cases} \Gamma_n \cup \{A_n\} & \text{if ok} \\ \Gamma_n \cup \{-A_n\} & \text{otherwise} \end{cases}$$

By Fact 3, Γ_{n+1} is ok. So all Γ_n are ok.
Finally, let

$$B_i = \text{whichever } A_i \text{ or } -A_i \text{ is in } \Gamma_{i+1}$$

and define

$$r \sim s \quad \text{iff there is an } i \text{ with } B_i \equiv (r = s$$

We claim $\sim$ is an equivalence relation. (Details left.) Thus $[t]$ for any term t in Γ is an equiv. class.

Now define an interpretation $\mathcal{I}$ matching Γ:

(1) $D_{\mathcal{I}} = \{ [t] : t \text{ appears in } \Gamma \}$
(or any nonempty set if none appear)

The big trick!

(2) For each name, a, assign $[a]$

(3) For each fcn f, assign $f([t_1], \ldots, [t_n]) = [f(s_1, \ldots, s_n)]$ for any $s_i \in [t_i]$.
Details: Gives unique values

(4) For each predicate R, R is true of $[t_1], \ldots, [t_n]$ iff $R t_1 \ldots t_n = B_i$ some i.
Details: Doesn't depend on t_i's

(5) For each sentence letter C, C is true if $B_i = C$

Notice that each B_i is true in $\mathcal{I}$ — (easy to check.) Also, every element in $\mathcal{I}$ is rep. by a term in Γ.

Thus, we just need to show that $\mathcal{I}$ is a model of Γ. Let $S \in \Gamma$. Then for some k, S is built out of $\{A_1, \ldots, A_k\}$ (maybe not all.) using $\wedge$, $\vee$, $-$, $\rightarrow$, $\leftrightarrow$. Consider

$$\gamma = \{B_1, \ldots, B_k, S\} \subseteq \Gamma_{k+1}$$

Then γ is satis. by, say, interp. $\mathcal{J}$. Thus, $\mathcal{J}(S) = 1$. But A_i has same value in $\mathcal{I}$ as in $\mathcal{J}$, since B_i does. Thus,

$\mathcal{I}(S) = 1$.

☒

$\boxed{\forall\exists\text{-Sentences, Compactness}}$ Lecture 28, 11/22/89

Theorem: Define an $\forall\exists$-sentence to be a first-order sentence in prenex form

$$\forall x_1 \dots \forall x_{n_1} \exists y_1 \dots \exists y_{n_2} \; F \quad (\Leftarrow \text{no fcn symbols, no quantifiers}$$

(Example: R is the graph of a total function –

$$\forall x \exists y \; R(x,y) \;\wedge\; (\forall x \forall y \forall z \; R(x,y) \wedge (x,z \to y=z)$$

Ø R is 1-1:

$$\forall x \forall y \forall z \; R(x,y) \wedge R(z,y) \Rightarrow (x=z)$$

and $\forall z \; \neg R(z,0)$

The conjuction here is an $\forall\exists$-sentence) Then th set of valid $\forall\exists$-sentences is decidable.

Proof: To decide if an $\forall\exists$-sentence S is valid, take prenex form of $\neg S$ and see if it has a refutation. B S' is an $\exists\forall$-sentence. The canonical derivatio (of kind generated in book) contains only n_1 names besides those already in S. The derivation contains no other terms (because there are no fcn symbols), and therefore is finite! Then I can check (decide) i $\neg S$ has a refutation. ☒

Compactness Theorem: Let Δ be a set of first-order sentences. Then Δ is ok iff Δ is satisfiable. Alternatively, if $\Delta \vdash S$ iff $\Delta' \vdash S$ for some finite $\Delta' \subset \Delta$.

Proof: For first part, $(\Leftarrow)$ is trivial. So suppose Δ is ok. Apply refutation procedure to Δ, and get Δ' which is also ok. Thus Δ' has a model (as we showed Thus, $\Delta \subseteq \Delta'$ has a model too.

Other part is propositionally equivalent to first part. ☒

Lemma: There is no sentence S such that $\mathfrak{M} \models S$ $D_{\mathfrak{M}}$ is infinite.

Proof: By contradiction. Suppose there were such an S. Then $\neg S$ means $D_{\mathfrak{M}}$ is finite. Thus, $\neg S \cup \{\exists x_1 \dots x_n [\bigwedge x_i \neq x_j]\}$ is ok. By compactness, there is a model of whole set Thus, domain is infinite! $\to \Leftarrow$

☒

<u>Exercise:</u> $\bigvee_{n=0}^{\infty} a = f(f(\_\_(f(b))\ldots))$

This is <u>not</u> equiv. to any first-order sentence.

Incompleteness Theorem Lecture 29, 11/27/89

__Def.__ Let $\mathcal{T} = \{S : S$ is a __true__ sentence of arithmetic$\}$ (language has $0, +, \cdot, '$).

__Introduce:__ A new name ω. Let

$$0_n \equiv \omega \neq 0 \wedge \omega \neq 0' \wedge \ldots \wedge \omega \neq 0^{\overbrace{''\cdots'}^{n}}$$

We remark that $\mathcal{T} \cup \{0_n : n \geq 0\}$ is ok. By compactness, there is a model $\mathcal{M}$ for this set. This is called a __nonstandard__ __model__ of arithmetic.

Limitation of power of first-order logic!

__Gödel's Incompleteness Thm__ (First, short form) $\mathcal{T}$ is not r.e.

__Corollary:__ Any sound, effective, axiom system for arithmetic fails to ~~be~~ have some true sentence as a theorem.

Want axioms to be decidable; also want rules to be decidable.

__Proof:__ By def., all __provable__ sentences are true. Also, theorems are r.e.. Thus, the theorems are $\subsetneq \mathcal{T}$. ☒

__Def:__ A subset $D \subseteq \mathbb{N}$ is __definable__ if there is a formula $F_D(x)$, s.t.

$$\vdash_{\mathcal{T}} F_D(\underset{\sim}{n}) \quad \text{iff} \quad n \in D.$$

__Example__ (1) Even numbers are definable: $\exists y\,(y+y=x)$

(2) Definable sets are closed under intersection:

$D_1 \cap D_2$ is definable by $F_{D_1}(x) \wedge F_{D_2}(x)$

Also complement, etc.

__Lemma:__ Every r.e. set is definable.

__Lemma:__ If D is definable, then $D \leq_m \mathcal{T}$.

__Proof:__ $n \in D$ iff $F_D(\underset{\sim}{n}) \in \mathcal{T}$. ☒

__Proof:__ Of incompleteness. By first Lemma, K_0 is definable. Thus, $\overline{K_0}$ is definable by example above. Thus, by second Lemma, $\overline{K_0} \leq_m \mathcal{T}$. Thus, $\mathcal{T}$ is not r.e. ☒

Finish Incompleteness Lecture 30, 11/29/89

__Lemma:__ Every r.e. set is arithmetically definable.

__Proof:__ Remark: $n *_p m = \ell$ is definable by an arithmetic formula. $*_p$ = concatenation of p-base integers. Actually, we showed that $n *_p m$ is prim. rec. In fact, we used bounded quantification so that it's clear how to write an arithmetic formula $\text{Concat}(n, m, \ell, p)$ which is true iff p is prime and $n *_p m = \ell$.

Thus, we can show every r.e. set of __strings__ over an alphabet $\{0,1\}$ is definable in the language of strings under concatenation (over $\Sigma \supseteq \{0,1\}$)

Let $\Sigma = \{0,1\} \cup \underset{\text{TM states}}{Q} \cup \underset{\text{Tape symbols}}{S} \cup \{\text{delimiters}\}$. To say $x \in \Sigma_0 \subseteq \Sigma$,

$$x = \sigma_1 \lor x = \sigma_2 \lor \dots \lor x = \sigma_k$$

where $\Sigma_0 = \{\sigma_1, \dots, \sigma_k\}$. To say $x \in \Sigma_0^*$,

$$\neg \exists y, z, w \; [x = yzw \;\land\; \neg(\text{“}z \in \Sigma_0\text{”}) \land \text{“}z \in \Sigma\text{”}]$$

To say x_1 is a suffix of x_2,

$$\exists y \; (x_2 = y x_1)$$

To say x_2 and x_3 are consecutive substrings of x_1, delimited by $\sigma_0 \in \Sigma$, where $x_2, x_3 \in (\Sigma - \{\sigma_0\})^*$, $(\Sigma_0 = \Sigma - \{\sigma_0\})$

$$\exists x_4, x_5 \; (x_1 = x_4 \sigma_0 x_2 \sigma_0 x_3 \sigma_0 x_5) \;\land\; (\text{“}x_2 \in \Sigma_0^*\text{”}) \land (\text{“}x_3 \in \Sigma_0^*\text{”})$$

Now we can start talking about TMs. Let M be a TM. Suppose M in state p reading symbol a changes to state q and prints b. Let

$$F_{p,a,q,b} \equiv \text{“}x_1 \text{ and } x_2 \text{ are configuration words, and } x_2 \text{ results from } x_1 \text{ by making the move”}$$

$$\equiv \exists x_3, x_4 \; [\, x_1 = x_3 \, p \, a \, x_4 \;\land\; \text{“}x_3 \in \Sigma_{\text{tape}}^*\text{”} \;\land\; \text{“}x_4 \in \Sigma_{\text{tape}}^*\text{”} \;\land\; x_2 = x_3 \, q \, b \, x_4 \,]$$

Now for moving,

$$F_{p,a,q,R} \equiv \exists x_3, x_4 \; [\, x_1 = x_3 \, p a \, x_4 \;\land\; \text{“}x_3 \in \Sigma_{\text{tape}}^*\text{”} \;\land\; \text{“}x_4 \in \Sigma_{\text{tape}}^*\text{”} \land [((x_2 = x_3 \, a \, q \, x_4) \land x_4 \neq x_4 x_4)) \lor (x_2 = x_3 \, a q \,(\text{blank}) \land x_4 = x_4 x_4)]$$

$$F_{p,a,q,L} \equiv \text{similar.}$$

Then

"x_1 moves to x_2 in one step"

$$\equiv \bigvee_{p,q,a,-} F_{p,a,q,-}$$

Now "x_1 goes to x_2 in 0 or more steps of M" $\equiv$ "there is a string x_3 with consecutive config. words delimited by σ_0, starting with x_1, ending with x_2, and s.t. every pair of consecutive words follows in one step"

$\equiv \exists x_3$ ["$\sigma_0 x_2 \sigma_0$ is a suffix of x_3" $\wedge$
"$\sigma_0 x_1 \sigma_0$ is a prefix of x_3" $\wedge$
"x_1 is a config. word" $\wedge$
$\forall x_4, x_5$ ["x_4 and x_5 are sonsec. substrings of x_3 delimited by σ_0" $\rightarrow$
"x_5 follows by one step of M by x_4"]]

Finally, to say "$x_1 \in \mathrm{dom}(M)$", ($x_1 \neq$ empty)

$\exists x_2, x_3, x_4, x_5$ [$x_2 = q_0 x_1$ $\wedge$ $x_3 = x_4 q_{\text{halt}} x_5$
$\wedge$ "x_2 goes to x_3 in 0 or more steps of M"]

$\boxed{\text{Intro to Call-by-value}}$ Lecture 31, 12/1/89

Language: FKS = functional kernel of Scheme. Our language will also be simply-typed.

Terms: * Types are $\sigma ::= \iota \mid \sigma_1 \to \sigma_2$. Terms are

$$x^\sigma : \sigma \qquad\qquad \frac{M : \sigma \to \tau \qquad N : \sigma}{(M\ N) : \tau}$$

$$c^\sigma : \sigma$$

$$\frac{M : \tau}{(\lambda x^\sigma . M) : \sigma \to \tau} \qquad\qquad \frac{M, N_1, N_2 : \iota}{(\text{cond}\ M\ N_1\ N_2) : \iota}$$

Constants are

- $0, 1, 2, \ldots \quad : \iota$
- succ, pred $\quad : \iota \to \iota$
- $Y_\sigma \quad : (\sigma \to \sigma) \to \sigma$

Currying: $+_\iota : \iota \to (\iota \to \iota)$ So $(+_\iota\ 5) = \lambda x^\iota .$ "$5 + x$"

Pairing: $\text{pair}_\sigma(M, N) \overset{\text{df}}{=} (\lambda z. ((z\ M)\ N))$ $\quad M, N : \sigma$

$z : \sigma \to (\sigma \to \sigma)$

Note that $\text{pair}_\sigma(M, N) : (\sigma \to (\sigma \to \sigma)) \to \sigma$. Let

$\text{left}_\sigma = (\lambda x^\sigma (\lambda y^\sigma .\ x))$

Let: (let ((var M)) N) in Scheme will be written
((λv. N) M)

Define: >(define (F x) M) $\leftarrow$ No F's in M
> (F N)

is translated to

$((\lambda f.\ (f\ N))\ (\lambda x. M))$

Recursion: Let f be $\lambda x.$ if $x = 0$ then 0 else $f(x-1) + 2$.

We let f be

fixpoint $(\lambda f.\ \lambda x.$ if $x = 0$ then 0 else $f(x-1) + 2)$

or

$(Y\ (\lambda f.\ \lambda x.\ \text{cond}\ x\ 0\ ^{(\text{succ}\ (\text{succ}}(f\ (\text{pred}\ x))))))$

$\boxed{\text{Recursion, Rewrite Rules}}$ Lecture 32, 12/4/89

<u>Example</u>: Recursive SCHEME program

```
(define (plus x Y)
        (if (zero? Y)
            x
            (succ (plus x (pred Y)))))
```

How do we think of this in FKS?

$$\text{let } plus = \lambda x.\ \lambda y.\ (cond\ \ y\ \ x\ \ (succ\ (plus\ \ x\ \ (pred\ y))))$$

It's not clear how to think of this as a <u>function</u>, since we don't have a <u>definition</u>; it's a <u>constraint</u>.

An x that satisfies

$$x = Body\ (x)$$

is called a <u>fixed</u> <u>point</u> of Body. In this case, $\lambda p. Body$ is

$$\lambda p^{\iota\to\iota\to\iota}.\ \lambda x^{\iota}.\ \lambda y^{\iota}.\ (cond\ \ y\ \ x\ \ (succ\ (p\ \ x\ \ (pred\ y))))$$

Now let

$$plus = (Y\ \ G)$$

We're rid of recursion!

<u>Rewrite rules</u>: Specifies an interpreter. Use → as "rewrites in one step." Will be a partial fcn.

$$(succ\ \ \underset{\sim}{n}) \to \underset{\sim}{n+1}$$
$$(pred\ \ \underset{\sim}{0}) \to \underset{\sim}{0}$$
$$(pred\ \ \underset{\sim}{n+1}) \to \underset{\sim}{n}$$
$$(cond\ \ \underset{\sim}{0}\ \ N_1\ \ N_2) \to N_1$$
$$(cond\ \ \underset{\sim}{m+1}\ \ N_1\ \ N_2) \to N_2$$

$$\frac{M \to M'}{(M\ N) \to (M'\ N)}$$

$$((\lambda x.\ M)\ V) \to M\,[x := V] \qquad (V \text{ a value})$$

anything but an application or conditional

Remark: Check that V is a closed value iff $V \not\to$

$$\frac{N \to N'}{(V\ N) \to (V\ N')}$$

$$(Y\ \ V) \to V\ (\lambda x.\ ((Y\ V)\ \ x))$$

$$\frac{M \to M'}{(\text{cond } M\ N_1\ N_2) \to (\text{cond } M'\ N_1\ N_2)}$$

Def: $\text{Eval}(M) : \text{Terms} \longrightarrow_p \text{Values}$. Claim Eval is well-defined

Def: A closed term $M^{\iota\to\iota}$ represents a partial fcn $f: N \to N$ iff

$$\text{Eval}((M\ \underset{\sim}{n})) = \underset{\sim}{m} \quad \text{iff} \quad f(n) = m$$

and $\text{Eval}((M\ \underset{\sim}{n}))$ is defined $n \in \text{dom}(f)$

Every fcn that ~~th~~ is representable is partial rec.; also, every partial rec. is representable.

Overview, SECD Machine   Lecture 33, 12/6/89

Informally: Environments are finite maps Variables → Value-closures. Closures are terms with an environment for free variables. Formally

(1) The empty function, $\emptyset$, is an environment

(2) A closure is a pair $[M, E]$ where M is a term and E is an environment and $FV(M) \subseteq dom(E)$.

(3) A value-closure is a closure $[M, E]$ s.t. M is a value (variable, abstraction, or constant)

(4) If $Cl_i^{\sigma_i}$ $(i = 1, \dots, n)$ are value closures and $x_i^{\sigma_i}$ are distinct variables, then E is an environment, where

$$dom(E) = \{x_1, \dots, x_n\} \quad \text{and}$$
$$E(x_i) = Cl_i$$

Def: Define the function Realterm: Closures → Closed Terms as follows:

- $\text{Realterm}([M, \emptyset]) = M$
- $\text{Realterm}([M, E]) = M[x_1 := \text{Realterm}(E(x_1)), \dots, x_n := \text{Realterm}(E(x_n))]$

where $FV(M) = \{x_1, \dots, x_n\}$.

Theorem 1: $\text{Realterm}(\text{SECD}(M)) = \text{Eval}(M)$.

We'll need: $\text{evalrel}(M, N)$ defined inductively. (Metacircular evaluator)

- $\text{evalrel}(V, V)$ V a value

- $$\frac{\text{eval}(M_1, M_2) \quad \text{eval}(N_1, N_2) \quad \text{eval}((M_2, N_2), N_3)}{\text{eval}((M, N_1), N_3)}$$

(Need other rules, too, like $\dfrac{\text{evalrel}(M[x := V], N)}{\text{evalrel}((\lambda x . M) V, N)}$.)

Prove, by induction on def. of evalrel, that

a) evalrel is the graph of a function eval (partial)

b) if $\text{evalrel}(M, N)$ then N is a value

Theorem 2: eval = Eval. Also, $\text{eval}(V) = V$ and $\text{eval}((M\ N)) = \text{eval}((\text{eval}(M))(\text{eval}(N)))$

Def: Stacks = $(\text{Closures})^*$

Controlstrings = $(\text{Term} \cup \{ap, cd\})^*$

$$\text{Dumps} = \begin{cases} \text{nil} & \text{or} \\ [S, E, C, D] & \text{where} \quad FV(C) \subseteq E \end{cases}$$

Also: Need Constapply and constapply. Constapply: $\overset{\text{Constants} \times}{\text{Value-Clos}} \to \text{Clos.}$

where

$$\text{Constapply}\ (a,\ [V, E]) \to Cl$$

e.g.

$$\text{Castapply}\ (Y,\ [V, E]) = [V(\lambda x.\ Y\ V\ x),\ E]$$

We define this via constapply: Constants × Terms → Terms

SECD Machine

Lecture 34, 12/8/89

SECD Machine: See notes for actual rules of machine.

Def: $M =_\alpha M'$ iff M and M' agree except on bound variables.

We want: $\text{Realterm}([M', E']) =_\alpha \text{Eval}(\text{Realterm}[M, E])$
if

$$[S, E, M{:}C, D] \Rightarrow^* [[M', E']{:}S, E, C, D]$$

Example: $[S, E, (\text{succ } 1){:}C, D] \Rightarrow [S, E, 1{:}\text{succ}{:}\text{ap}{:}C, D]$

$\Rightarrow [[1, \emptyset]{:}S, E, \text{succ}{:}\text{ap}{:}C, D]$

$\Rightarrow [[\text{succ}, \emptyset]{:}[1, \emptyset]{:}S, E, \text{ap}{:}C, D]$

$\Rightarrow [\text{nil}, \emptyset, 2, [S, E, C, D]]$

$\Rightarrow [[2, \emptyset], \emptyset, \text{nil}, [S, E, C, D]]$

$\Rightarrow [[2, \emptyset]{:}S, E, C, D]$

Example: $[S, E, ((\lambda x. x)(\lambda y. 3)){:}C, D]$

$\Rightarrow [S, E, (\lambda y. 3){:}(\lambda x. x){:}\text{ap}{:}C, D]$

$\Rightarrow^2 [[(\lambda x. x), E]{:}[(\lambda y. 3), E]{:}S, E, \text{ap}{:}C, D]$

$\Rightarrow [\text{nil}, E\{[\lambda y. 3, E]/x\}, x, [S, E, C, D]]$

$\Rightarrow [[\lambda y. 3, E], E\{[\lambda y. 3, E]/x\}, \text{nil}, [S, E, C, D]]$

$\Rightarrow [[\lambda y. 3, E]{:}S, E, C, D]$

Def: Load: Closed-Terms $\longrightarrow$ Dump where
$\text{Load}(M) = [\text{nil}, \emptyset, M, \text{nil}]$
and $\text{Unload}([[M', E], \emptyset, \text{nil}, \text{nil}]) =$ ~~Then~~ $\text{Realterm}([M', E])$. Then
$\text{SECD}(M) = \text{Unload}(D')$ where
$\text{Load}(M) \Rightarrow^* D'$ and D' is stopped

Theorem: $\text{SECD}(M) =_\alpha N$ iff $\text{Eval}(M) =_\alpha N$

$\boxed{\text{Semantics of FKS}}$ Lecture 35, 12/11/89

Today: Building a model for FKS. Some particular problems:

1. Give meaning to constants and conditional expressions.
2. Must be closed under λ-abstraction
3. Must be closed under application

Def: D^σ inductively:

- $D^\iota = \mathbb{N}$
- $D^{\sigma \to \tau} = D^\sigma \to_c D^\tau$ (partial continuous fcns)

Recall: Partial orders. We use $\sqsupseteq$ notation, with

- $a \sqsupseteq b$ iff $b \sqsubseteq a$
- $a \sqsubset b$ iff $a \neq b$ and $a \sqsubseteq b$

Ordering:

- For D^ι, $a \sqsubseteq b$ iff $a = b$
- For $D^{\sigma \to \tau}$, $f \sqsubseteq g$ iff for all $a \sqsubseteq b$, $f(a) \sqsubseteq g(b)$.

Example: Approximations to factorial function – H_n

Def: $\bigsqcup X$ = least upper bound of X.

Note: $\bigsqcup_i H_i$ = factorial fcn.

Def: A subset X of D^σ is directed if for every pair of elements $w, y \in X$, there is a $z \in X$ with $w \sqsubseteq z$, $y \sqsubseteq z$.

Def: A cpo is a $[D, \sqsubseteq]$ s.t. every directed set X has a $\bigsqcup X \in D$.

Def: $f \in D^\sigma \to D^\tau$ is continuous if for all directed $X \subseteq D^\sigma$
$f(\bigsqcup X) = \bigsqcup \{f(x)\}$

Def: $[\![\cdot]\!]$: Terms × Environments $\to D^\sigma$. See notes for definition.

Def: $M \equiv_{obs} N$ iff M and N are completely interchangable piezes of code.

Theorem 1: (Soundness) For any program M and constant k, $\mathrm{Eval}(M) = k \Rightarrow [\![M]\!]\rho = k$.

Theorem 2: (Adequacy) For any program M and constant k, $\mathrm{Eval}(M) = k \Leftrightarrow [\![M]\!]\rho = k$.

Example: $[\![\lambda x^{\sigma}. \Omega_{\sigma}]\!] = [\![Y_{\sigma\to\sigma}(\lambda x^{\sigma\to\sigma}. x)]\!]$. Thus,

$$\lambda x. \Omega_{\sigma} \equiv_{Obs} Y_{\sigma\to\sigma}(\lambda x^{\sigma\to\sigma}. x)$$