Introduction to Computability Theory

Dag Normann
The University of Oslo
Department of Mathematics
P.O. Box 1053 - Blindern
0316 Oslo
Norway

August 16, 2010

Contents

Preface

This compendium will be the curriculum text for the course on Computability Theory at the University of Oslo, Autumn 2010. The compendium is based on chapters 3 and 4 of the compendium for "Mathematical Logic II" from 2005, [3] In its present form, the compendium may be used free of charge by anyone, but if someone uses it for an organized course, the author would like to be informed.

Blindern, 16.08.2010

Dag Normann

Introduction

This compendium is written primarily as a text for the course MAT4630 - *Computability Theory* given at the University of Oslo, Norway. The compendium is essentially consisting of two parts, *Classical Computability Theory* and *Generalized Computability Theory*. In Chapter 1 we use a Kleene-style introduction to the class of computable functions, and we will discuss the recursion theorem, c.e. sets, Turing degrees, basic priority arguments, the existence of minimal degrees and a few other results.
In Chapter 2 we give an introduction to computations relative to type 2 functionals, the hyperarithmetical sets and, to some extent, to **PCF**.

We will assume that the reader is familiar with the standard vocabulary of logic and set theory, but no advanced background from logic is required.

Both chapters are supplied with a set of exercises at the end, some simple and some hard. The exercises are integrated parts of the text, and at the end the students are assumed to have worked through most of them. The philosophy behind this is that students have to work through some of the proofs themselves in order to really understand the subject and being able to use it in other contexts.

Chapter 3 will consist of just exercises. These are not integrated parts, but challenges the eager reader/student might like to face.
The compendium is not to be considered as a complete textbook. For instance, it is scarce on references, and we do not give proper credit for the theorems we prove.

At the end, there is a subject index and a short bibliography. On request, it is possible to extend both parts, and then to make these revised parts available.

Suggestions for further reading

- Cooper [1]
- Odifreddi [4]
- Rogers [5]
- Sacks [6]
- Soare [7]
- Streicher [8]

There are many other texts available. Still maybe Rogers [5] and Odifreddi [4] are the most accessible introductory texts available.

Chapter 1

Classical Computability Theory

1.1 The foundation, Turing's analysis

In Leary [2] (the text book used locally for the introductory course on logic) the recursive functions are defined as those that can be represented in elementary number theory. $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if there is a formula $\phi(x_1, \ldots, x_k, y)$ such that for all $n_1, \ldots, n_k, m$ we have that $f(n_1, \ldots, n_k) = m$ if and only if

$$N \vdash \phi(c_{n_1}, \ldots, c_{n_k}, y) \leftrightarrow y = c_m.$$

Here c_n is the numeral for n, and N is elementary number theory.

The advantage of this definition is that it is well suited for proving Gödel's incompleteness theorem without introducing too many new concepts. The problem is that there is absolutely no conceptual analysis of the notion of computability behind this definition.

Gödel defines a class of recursive functions by recursion (pun intended). His aim is to define a sufficiently rich class for handling algorithms for e.g. substitution of a term for a variable, and for coding the meta-theory of a formal theory, but sufficiently simple to enable us to show that any recursive function will be definable, and actually, representable as described above. Gödel's computable functions are now known as the μ-recursive ones.

We are going to base our study of computability on an approach due to Kleene, and we are going to restrict ourselves to computable functions defined on the natural numbers. In many respects, computing can be described as manipulation of symbols following a given set of rules. The symbols are not then natural numbers, and different ways of representing natural numbers (binary, decadic, via numerals, Roman figures etc.) might give different concepts of computing with numbers.

The best mathematical model for computability and computations is due to Alan Turing. He defined what is now known as *Turing machines*, small

finite state machines operating on an infinite one-dimensional tape and doing symbol manipulations on this tape. The machine will, between each step of the computation, be in one of finitely many *states*. It will read one of the symbols on the tape, and dependent of the state it is in and the symbol on the tape that it reads, it will according to a fixed rule change its state, rewrite the symbol and move to the symbol to the right or to the left. It may of course stay in the same state, it may of course keep the symbol on the tape as it is, but it is hardly convenient to let it stay where it is. We think of the tape as consisting of squares, like an old fashioned movie-tape.

A Turing machine M is determined by

1. A finite alphabet Σ including one special symbol Bl for an empty square of the tape.
2. A finite set K of states, with one special state $s \in K$ called the *initial state*.
3. A partial function $\delta : (\Sigma \times K) \to (\Sigma \times K \times \{L, R\})$, where L means "left" and R means "right".

In the literature you will find many variations in the definition of a Turing machine, but they all have the same computational power. In this exposition, we decided to let the function δ, which rules the action of the machine, be partial. If $\delta(\sigma, p)$ is undefined, the machine will be in a *halting situation*, which means that the computation comes to an end. We are not going to give a precise definition of the operational semantics for Turing machines, but are content with an intuitive description:

The operational semantics of Turing Machines

Let $M = \langle \Sigma, K, s, \delta \rangle$ be a Turing machine.
The starting configuration of M will consist of a word w (called *the input*) in Σ written on an otherwise blank tape that is infinite in both directions, a position on the tape and the initial state s.
At each step, the machine M may enter a new state, rewrite the symbol at its position on the tape and shift its position one square to the right or to the left, all according to the function δ.
If M is in stage p and reads the symbol σ, and $\delta(\sigma, p)$ is undefined, M halts. Then we who observe M will know this, and we will be able to read the content of the tape, which will be called *the output*.

Normally there will be some conventions for how to organize the input configuration, e.g. that there should be no blanks in the input word and that the machine will be started at the first blank square to the left or to the right of the input word. Then the following makes sense

Definition 1.1.1 Let Σ_0 be an alphabet not containing the symbol Bl
Let Σ_0^* be the set of finite words over Σ_0.
Let $f : \Sigma_0^* \to \Sigma_0^*$ be a partial function.

We say that f is *Turing computable* if there is a Turing machine M over an alphabet $\Sigma \supset \Sigma_0$ such that if M is started with $w \in \Sigma^*$ on the tape, then it halts if and only if $f(w)$ is defined, and then with $f(w)$ as the output word.

Turing claimed that a Turing machine can

- Search systematically for pieces of information.
- Remember pieces of information
- Rewrite contents

all according to fixed rules. As an example we may consider the map $n \mapsto n!$ and the map $m \mapsto m^m$. We all agree that these maps are in principle computable, but try to think about how a computation of $10^{10}!$ could be carried out: We will need a lot of book-keeping devices in order to be at the top of the situation at each stage, but nothing that is not covered by the three items above.

We follow Turing in claiming that his model is a good mathematical model for algorithmic computations. We are, however, not going to make this compendium into a study of Turing machines. The interested reader should consult other textbooks on the subject.

The two key results are:

Theorem 1.1.2 *There is a fixed alphabet Σ such that for any alphabet Σ', any Turing machine M over Σ' may be coded as a word $[M]$ in Σ and every word w in Σ' may be coded as a word $[w]$ in Σ such that the partial function*

$$U([M][w]) = [M(w)]$$

is Turing computable, where we write $M(w)$ for the output word if the input word is w.

This is known as the existence of a *Universal Turing Machine.*

The *Halting Problem* is the following:

Given a Turing machine M and an input w, will M eventually come to a halt when started on w?

Theorem 1.1.3 *There is no Turing machine H that solves the Halting Problem in the following sense:*
$H([M][w])$ will always halt, and will halt with an empty output word if and only if M halts on w.

Our approach to computability will be more in the original style of Gödel, we will study functions defined for natural numbers only. However, the results that we obtain will be relevant for the more general approaches as well. This is based on what is known as the Church-Turing Thesis, which we phrase like this:

All algorithms can be simulated by a Turing Machine,

and the fact that Turing-computability can be reduced to the notion we will be working with. This is left as one of the minor projects in Chapter 3, see Exercise 3.2.

1.2 Computable functions and c.e. sets

1.2.1 The primitive recursive functions

The basic definition

Recursion means 'backtracking', and in pre-Church/Kleene mathematics the term *recursive function* was used for the functions defined by iterated recursion. In this pre-Church/Kleene context a recursive definition will be a definition of a function on the natural numbers, where we give one initial value $f(0)$, and define $f(k+1)$ as a function of $f(k)$ and k. E.g. Skolem used the term 'recursive function' in this way. Following Kleene, we will call these functions *primitive recursive*.

We let $\vec{x}$ and $\vec{y}$ etc. denote ordered sequences of natural numbers of some fixed length. Normally the length will not be specified, but will be clear from the context.

Definition 1.2.1 *The primitive recursive functions* $f : \mathbb{N}^n \to \mathbb{N}$ will be the least class of functions satisfying:

i) $f(x, \vec{y}) = x + 1$ is primitive recursive.

ii) $I_{i,n}(x_1, \ldots, x_n) = x_i$ is primitive recursive.

iii) $f(\vec{x}) = q$ is primitive recursive for each $q \in \mathbb{N}$.

iv) If g is n-ary and primitive recursive, and $f_1, \ldots, f_n$ are m-ary and primitive recursive, then the composition

$$h(\vec{x}) = g(f_1(\vec{x}), \ldots, f_n(\vec{x}))$$

is primitive recursive.

v) If g and h are primitive recursive of arity n and $n + 2$ resp., then f is primitive recursive where

$f(0, \vec{y}) = g(\vec{y})$
$f(x + 1, \vec{y}) = h(f(x, \vec{y}), x, \vec{y})$

We say that f is defined by *primitive recursion* or *by the recursion operator* from g and h.

The pool of primitive recursive functions and sets

Standard number theoretical functions like addition, multiplication, exponentiation and factorial will all be primitive recursive. For instance $f(x, y) = x + y$ may be defined by

- $x + 0 = x$
- $x + (y + 1) = (x + y) + 1$.

Subtraction is not primitive recursive for the simple reason that it leads us outside $\mathbb{N}$. We define a modified subtraction. This will be primitive recursive, see Exercise 1.1.

Definition 1.2.2 We let $\dot{-}$ be the modified subtraction defined by

$x \dot{-} y = x - y$ if $y \leq x$

$x \dot{-} y = 0$ if $x \leq y$.

By the definition of the primitive recursive functions, all projection maps

$$I_{i,n}(x_1, \ldots, x_n) = x_i$$

are primitive recursive. We can use this to consider any function of a set of variables as a function of a larger set of variables, as in

$$f(x_1, x_2, x_3) = g(I_{1,3}(x_1, x_2, x_3), I_{3,3}(x_1, x_2, x_3)) = g(x_1, x_3).$$

Thus we will not need to be concerned with the requirement that every function in a composition must be of the same arity. This will simplify some of the descriptions of primitive recursive functions.

Definition 1.2.3 Let $A \subseteq \mathbb{N}^n$. The *characteristic function* of A will be the function

$$K_A : \mathbb{N}^n \to \mathbb{N}$$

that is 1 on A and 0 outside A.

Definition 1.2.4 A set $A \subseteq \mathbb{N}^n$ is *primitive recursive* if the characteristic function K_A is primitive recursive.

The primitive recursive sets will form a Boolean algebra for every dimension, see Exercise 1.2.

If $A \subseteq \mathbb{N}^k$ is primitive recursive and $f, g : \mathbb{N}^k \to \mathbb{N}$ are primitive recursive, we may define a primitive recursive function h *by cases* as follows:

$$h(\vec{x}) = K_A(\vec{x}) \cdot f(\vec{x}) + (1 \dot{-} K_A(\vec{x})) \cdot g(\vec{x})).$$

We see that $h(\vec{x}) = f(\vec{x})$ when $\vec{x} \in A$ and $h(\vec{x}) = g(\vec{x})$ otherwise.

Every set defined from $=$ and $<$ using propositional calculus will be primitive recursive. We may also use functions known to be primitive recursive in the definition of primitive recursive sets, and we may use primitive recursive sets when we describe primitive recursive functions. We will leave the verification of most of this to the reader, just state the properties of primitive recursion that are useful to us. The proofs are simple, and there is no harm leaving them as exercises. We will however prove one basic (and simple) lemma:

Lemma 1.2.5 *Let $f : \mathbb{N}^{1+n} \to \mathbb{N}$ be primitive recursive. Then the function g defined as the bounded product*

$$g(x, \vec{y}) = \prod_{z \leq x} f(z, \vec{y})$$

will be primitive recursive.

Proof
We define g by primitive recursion as follows
$g(0, \vec{y}) = f(0, \vec{y})$
$g(x + 1, \vec{y}) = g(x, \vec{y}) \cdot f(x + 1, \vec{y})$.

By the same argument we can show that the primitive recursive functions will be closed under bounded sums. What is more important to us is that the primitive recursive sets will be closed under bounded quantification. This is an important strengthening of the pure relational language without quantifiers, and this will be useful when we need to construct primitive recursive sets and functions for various tasks.

Lemma 1.2.6 *Let $A \subseteq \mathbb{N}^{1+n}$ be primitive recursive. Then the following sets are primitive recursive*

a) $B = \{(x, \vec{y}) \mid \exists z \leq y((z, \vec{y}) \in A)\}$

b) $C = \{(x, \vec{y}) \mid \forall z \leq y((z, \vec{y}) \in A)\}$

The proof is left as Exercise 1.3.

In computability theory, the μ-operator is important. Within primitive recursion theory we may often use *bounded search*, or a bounded μ-operator:

Lemma 1.2.7 *Let $f : \mathbb{N}^{1+n} \to \mathbb{N}$ be primitive recursive. Then*

$$g(x, \vec{y}) = \mu_{<x} z.(f(z, \vec{y}) = 0)$$

is primitive recursive, where $g(x, \vec{y})$ is the least z such that $f(z, \vec{y}) = 0$ if there is one such $z < x$, while $g(x, \vec{y}) = x$ otherwise.

g can be defined using primitive recursion and definition by cases. The details are left as Exercise 1.4.

The interplay between the primitive recursive functions and the primitive recursive sets is ruled by the following principles:

Theorem 1.2.8 *a) Every set definable from $=$ and $<$ using primitive recursive functions, boolean operators and bounded quantifiers will be primitive recursive.*

b) Every function defined by the schemes of primitive recursion, bounded search over a primitive recursive set and definition by cases over a finite partition of $\mathbb{N}^n$ into primitive recursive sets will be primitive recursive.

There is no need to prove this theorem, since it in a sense is the synthesis of what has been proved so far. The consequence is the level of freedom in defining primitive recursive functions and relations we have obtained. This freedom will be sufficient when we later claim that certain facts are trivial to prove.

Sequence numbers

One important use of primitive recursion is the coding of finite sequences. Gödel needed an elaborate way of coding finite sequences via the so called β-function. As we mentioned above, it was important for Gödel to show that the recursive functions are definable in ordinary number theory. Since this is not important to us to the same extent, we will use full primitive recursion in coding such sequences. All proofs of the lemmas below are trivial and can safely be left for the reader.

Lemma 1.2.9 *a) The set of prime numbers (starting with 2 as the least prime number) is primitive recursive.*

b) The monotone enumeration $\{p_i\}_{i\in\mathbb{N}}$ of the prime numbers is primitive recursive (with $p_0 = 2$).

We now define the sequence numbers:

Definition 1.2.10 Let $x_0, \ldots, x_{n-1}$ be a finite sequence of numbers, $n = 0$ corresponding to the empty sequence.
We let the corresponding *sequence number* $\langle x_0, \ldots, x_{n-1}\rangle$ be the number

$$2^n \cdot \prod_{i=0}^{n-1} p_{i+1}^{x_i}$$

If $y = \langle x_0, \ldots, x_{n-1}\rangle$, we let $lh(y)$ (the *length of* y) be n and $(y)_i = x_i$.
If $x = \langle x_0, \ldots, x_{n-1}\rangle$ and $y = \langle y_0, \ldots, y_{m-1}\rangle$, we let $x * y$ be the sequence number of the *concatenation*, i.e.

$$x * y = \langle x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1}\rangle.$$

Sometimes, when it is clear from the context, we will misuse this terminology and let

$$a * \langle x_0, \ldots, x_{n-1}\rangle = \langle a, x_0, \ldots, x_{n-1}\rangle.$$

Lemma 1.2.11 *a) The set of sequence numbers is primitive recursive, and the sequence numbering is one-to-one (but not surjective).*

b) The function lh is the restriction of a primitive recursive function to the set of sequence numbers.

c) The function $coor(y, i) = (y)_i$ (the i'th coordinate of y) defined for sequence numbers y of length $> i$ is the restriction of a primitive recursive function of two variables.

d) For each n, the function

$$seq_n(x_0, \ldots, x_{n-1}) = \langle x_0, \ldots, x_{n-1} \rangle$$

is primitive recursive.

e) For all sequences $x_0, \ldots, x_{n-1}$ and $i < n$, $x_i < \langle x_0, \ldots, x_{n-1} \rangle$

f) 1 is the sequence number of the empty sequence.

h) Concatenation of sequence numbers is primitive recursive.

It makes no sense to say that the full sequence numbering is primitive recursive. However any numbering satisfying Lemma 1.2.11 can be used for the purposes of this course, so there is no need to learn the details of this definition. Occasionally it may be useful to assume that the sequence numbering is surjective. This can be achieved, see Exercise 1.5.

Primitive recursion is technically defined on successor numbers via the value on the predecessor. Sometimes it is useful to use the value on all numbers less than the argument x in order to define the value on x. In order to see that this is harmless, we use the following construction and the result to be proved in Exercise 1.6:

Definition 1.2.12 Let $f : \mathbb{N} \to \mathbb{N}$.
Let $\bar{f}(n) = \langle f(0), \ldots, f(n-1) \rangle$.

We will sometimes use an alternative coding of pairs that is both 1-1 and onto:

Definition 1.2.13 Let

$$P(x, y) = \frac{1}{2}((x + y)^2 + 3x + y)$$

It can be shown that $P : \mathbb{N}^2 \to \mathbb{N}$ is a bijection. Let π_1 and π_2 be the two projection maps such that for any x

$$P(\pi_1(x), \pi_2(x)) = x.$$

P, π_1 and π_2 are primitive recursive. The verifications are left for the reader as Exercise 1.5 e).

Ackermann proved that there is a total computable function that is not primitive recursive. His observation was that in the list of functions

$$f_0(x, y) = x + 1,\ f_1(x, y) = x + y,\ f_2(x, y) = xy,\ f_3(x, y) = x^y$$

each function but the first is defined as a y-iteration of the previous one.
For $n \geq 3$ we may then define

$$f_n(x,0) = 1,\ f_n(x,y+1) = f_{n-1}(f_n(x,y),x)$$

This defines the generalized exponentiations, or the *Ackermann-branches*. The three-place function

$$f(n,x,y) = f_n(x,y)$$

is not primitive recursive.
We will not prove Ackermann's result as stated, but in Exercise 1.7 we will see how we may define a function, using a double recursion, that is not primitive recursive. It will be easier to solve this exercise after the introduction to Kleene indexing.

1.2.2 The computable functions

The μ-operator

We extend the definition of the primitive recursive functions to a definition of the computable functions by adding one principle of infinite search. We will consider the construction

$$g(\vec{x}) = \mu x.f(x,\vec{x}) = 0.$$

In order to understand this definition, we must discuss what we actually mean, i.e. which algorithm this is supposed to represent.
Intuitively we want $\mu x.g(x,\vec{x}) = 0$ to be the least x such that $g(x,\vec{x})$ is 0. If $g(x,\vec{x}) \in \mathbb{N}$ for all x, this is unproblematic, we search for this least x by computing $g(0,\vec{x})$, $g(1,\vec{x})$ and so on until we find one value of x giving 0 as the outcome. Now, if there is no such x, we will search in vain, or in more technical terms, our procedure will be non-terminating. This forces us to introduce partial functions, i.e. functions being undefined on certain arguments. This further forces us to be careful about our interpretation of the μ-operator, we may in the search for the least x such that $g(x,\vec{x}) = 0$ face a z for which $g(z,\vec{x})$ is undefined before we find a z for which $g(z,\vec{x}) = 0$. In this case we will let $\mu x.g(x,\vec{x}) = 0$ be undefined, realizing that the algorithm for computing this number that we have in mind will be non-terminating.
With this clarification we ad the following

Definition 1.2.14 The *computable functions* is the least class of partial functions $f : \mathbb{N}^n \to \mathbb{N}$ satisfying i) - v) in the definition of the primitive recursive functions, together with the extra clause

vi) If $g : \mathbb{N}^{n+1} \to \mathbb{N}$ is computable then

$$f(\vec{x}) = \mu x.g(x,\vec{x}) = 0$$

is computable.

A set is *computable* if the characteristic function is computable.
A *total computable function* is a computable function terminating on all inputs from the domain $\mathbb{N}^n$.

Remark 1.2.15 Gödel's μ-recursive functions will be the total computable functions.

We must have in mind that the characteristic function of a set is total, so dealing with computable sets and with total computable functions will be much of the same. This is made precise in the following lemma:

Lemma 1.2.16 *Let $f : \mathbb{N}^n \to \mathbb{N}$ be total. Then the following are equivalent:*

i) f is computable.

ii) The graph of f, seen as a subset of $\mathbb{N}^{n+1}$, is computable.

Proof
Let A be the graph of f, K_A the characteristic function of A.
If f is computable, then

$$K_A(\vec{x}, y) = K_=(f(\vec{x}), y)$$

and $K_=$ is primitive recursive, see Exercise 1.1, g) and e).
If K_A is computable, then

$$f(\vec{x}) = \mu y.1 \dot{-} K_A(\vec{x}, y) = 0,$$

so f is computable.

Remark 1.2.17 This result will not hold for primitive recursion, there will be functions with primitive recursive graphs that are not primitive recursive.

There is an interplay between the total computable functions and the computable sets resembling Theorem 1.2.8 as follows

Theorem 1.2.18 *a) Any set definable from computable sets and total computable functions using boolean valued operators and bounded quantifiers will be computable.*

b) If $A \subseteq \mathbb{N}^{n+1}$ is computable, then g is computable, where $g(\vec{x})$ is the least number z such that $(z, \vec{x}) \in A$.

The proof is trivial.

Kleene's T-predicate

The definition of the computable functions is technically an inductive definition of a class of partial functions. It is, however, important to be aware of the computational interpretation of this definition, the so to say 'operational semantics'. Every partial computable function is given by a term in a language with constants for each number, the $+1$ function and symbols for the recursion operator and the μ-operator.

This operational semantics then tells us that there are actual computations going on. Now we will define the concept of a *computation tree*. A computation tree will be a number coding every step in a computation up to the final outcome. In order to be able to do so, we need a Gödel numbering, or an indexing, of the computable functions. Observe that the numbering will not be 1-1, we will enumerate the algorithms or the terms, and then only indirectly enumerate the computable functions.

Definition 1.2.19 For each number e and simultaneously for all n we define the partial function ϕ_e^n of n variables $\vec{x} = (x_1, \ldots, x_n)$ as follows:

i) If $e = \langle 1 \rangle$, let $\phi_e^n(\vec{x}) = x_1 + 1$.

ii) If $e = \langle 2, i \rangle$ and $1 \leq i \leq n$, let $\phi_e^n(\vec{x}) = x_i$.

iii) If $e = \langle 3, q \rangle$, let $\phi_e^n(\vec{x}) = q$.

iv) If $e = \langle 4, e', d_1, \ldots, d_m \rangle$ let

$$\phi_e(\vec{x}) = \phi_{e'}^m(\phi_{d_1}^n(\vec{x}), \ldots, \phi_{d_m}^n(\vec{x})).$$

v) If $e = \langle 5, d_1, d_2 \rangle$ then
$\phi_e^{n+1}(0, \vec{x}) = \phi_{d_1}^n(\vec{x})$
$\phi_e^{n+1}(x+1, \vec{x}) = \phi_{d_2}^{n+2}(\phi_e^{n+1}(x, \vec{x}), x, \vec{x})$.

vi) If $e = \langle 6, d \rangle$ then

$$\phi_e^n(\vec{x}) = \mu z.\phi_d^{n+1}(z, \vec{x}) = 0.$$

Otherwise If neither of i) - vi) above applies, let $\phi_e^n(\vec{x})$ be undefined.

Remark 1.2.20 We have defined $\phi_e(\vec{x})$ for every index e and every input $\vec{x}$, either as an undefined value or as a natural number. Indeed, if we get a natural number, this number will be unique, see Exercise 1.8.
When no misunderstanding should occur, we will drop the superscript n and write $\phi_e(\vec{x})$ instead of $\phi_e^n(\vec{x})$.

Definition 1.2.21 We write $\phi_e(\vec{x})\downarrow$ if there is a y with $\phi_e(\vec{x}) = y$. We then say that $\phi_e(\vec{x})$ *terminates*.
If $\phi_e(\vec{x})$ does not terminate, we may write

$$\phi_e(\vec{x})\uparrow .$$

We are now ready to use the sequence numbering and this indexing to define computation trees. Each terminating computation will have a unique computation tree, a number coding each step of the computation from the input to the output. We will actually be overloading this code with information, but for our purposes this is harmless. What is important to us is that information retrieval will be easy.

Before giving the formal definition, we will describe more informally how a computation may be viewed as writing down a labelled tree, where we start with a root node

$$\phi_e(\vec{x}) =?$$

and ends up, after having created the whole tree, with a root node

$$\phi_e(\vec{x}) = a,$$

where a is the output of the computation.
If ϕ_e is one of the initial functions from schemes i) - iii), we can just replace ? with the correct answer a at once.
If

$$\phi_e(\vec{x}) = \phi_{e'}(\phi_{d_1}(\vec{x}), \ldots, \phi_{d_m}(\vec{x}))$$

we first write down child nodes labelled $\phi_{d_i}(\vec{x}) =?$ for $i = 1, \ldots, m$, then follow our procedure for replacing the occurrences of ? with proper outputs $\vec{y} = y_1, \ldots, y_m$.
Then we introduce a further child node $\phi_{e'}(\vec{y}) =?$ and go through the process again replacing this ? with a value a. Then we may finally replace the topmost ? with a.

We may also describe what goes on in schemes v) and vi) as constructing subtrees until we have the information required.
It is the essential part of the information in these informal trees that we want to capture, in a coded way, in the computation trees, and we use iterated sequence numbering to obtain this code.

Definition 1.2.22 Let $\phi_e(\vec{x}) = y$. By primitive recursion on e we define the *computation tree* of $\phi_e(\vec{x}) = y$ as follows, assuming that the index e will be the index of the corresponding case:

i) $\langle e, \vec{x}, x_1 + 1\rangle$ is the computation tree for $\phi_e(\vec{x}) = x_1 + 1$.

ii) $\langle e, \vec{x}, x_i\rangle$ is the computation tree for $\phi_e(\vec{x}) = x_i$.

iii) $\langle e, \vec{x}, q\rangle$ is the computation tree for $\phi_e(\vec{x}) = q$.

iv) $\langle e, t, t_1, \ldots, t_n, y\rangle$ is the computation tree in this case, where each t_i is the computation tree for $\phi_{d_i}(\vec{x}) = z_i$ and t is the computation tree for $\phi_{e'}(\vec{z}) = y$.

v) $\langle e, 0, t, y\rangle$ is the computation tree for $\phi_e(0, \vec{x}) = y$ when t is the computation tree for $\phi_{d_1}(\vec{x}) = y$.
$\langle e, x + 1, t_1, t_2, y\rangle$ is the computation tree for $\phi_e(x + 1, \vec{x}) = y$ when t_1 is the computation tree for $\phi_e(x, \vec{x}) = z$ and t_2 is the computation tree for $\phi_{d_2}(z, x, \vec{x}) = y$.

vi) $\langle e, t_0, \ldots, t_{y-1}, t_y, y\rangle$ is the computation tree in this case, where t_i is the computation tree for $\phi_d(i, \vec{x}) = z_i \neq 0$ for $i < y$ and t_y is the computation tree for $\phi_d(y, \vec{x}) = 0$.

We say that t is a computation tree for $\phi_e^n(\vec{x})$ if for some y, t is the computation tree for $\phi_e^n(\vec{x}) = y$.

We are now ready to define Kleene's T-predicate:

Definition 1.2.23 Let

$$T_n(e, x_1, \ldots, x_n, t)$$

if t is a computation tree for $\phi_e(x_1, \ldots, x_n)$

We will normally write T instead of T_1.

Theorem 1.2.24 *a) For each n, T_n is primitive recursive.*

b) There is a primitive recursive function U such that if t is a computation tree, then $U(t)$ is the output of the corresponding computation.

c) **(Kleene's Normal Form Theorem)**
For every arity n and all e we have

$$\phi_e(x_1, \ldots, x_n) = U(\mu t.T_n(e, x_1, \ldots, x_n, t)).$$

Proof
It is only a) that requires a proof. The proof of a) is however easy, we construct the characteristic function of T_n by recursion on the last variable t. Monotonisity of the sequence numbering is important here. We leave the tedious, but simple details for the reader.

Corollary 1.2.25 *For each number n, the function*

$$f(e, x_1, \ldots, x_n) = \phi_e(x_1, \ldots, x_n)$$

is computable.

Remark 1.2.26 Corollary 1.2.25 is the analogue of the existence of a universal Turing machine, we can enumerate the computable functions in such a way that each computable function is uniformly computable in any of the numbers enumerating it. This universal function is partial. There is no universal function for the total computable functions, see Exercise 1.9.

The Recursion Theorem

The recursion theorem is one of the key insights in computability theory introduced by Kleene. In programming terms it says that we may define a set of procedures where we in the definition of each procedure refer to the other procedures in a circular way. The proof we give for the recursion theorem will be a kind of 'white rabbit out of the hat' argument based on the much more intuitive S_m^n-theorem. So let us first explain the S_m^n-theorem. Let f be a computable function of several variables. Now, if we fix the value of some of the variables, we will get a computable function in the rest of the variables. Of course, the index

of this function will vary with values we give to the variables.The S^n_m-theorem tells us that the index for this new function can be obtained in a primitive recursive way from the index of the original function and the values of the fixed variables. We have to prove one technical lemma:

Lemma 1.2.27 *There is a primitive recursive function ρ (depending on n) such that if*

$$\phi^n_e(x_1, \ldots, x_n) = t$$

and

$$1 \leq i \leq n$$

then

$$\phi^{n-1}_{\rho(e,i,x_i)}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = t.$$

Proof
We define ρ by induction on e, considering the cases i) - vi). We leave some of the easy details for the reader, see Exercise 1.11.
$\phi_e(\vec{x}) = x_1 + 1$:
If $1 < i$ let $\rho(e, i, x_i) = e$, while $\rho(e, 1, x_1) = \langle 3, x_1 + 1\rangle$.
The cases ii) and iii) are left for the reader.
If $e = \langle 4, e', d_1, \ldots, d_n\rangle$, we simply let

$$\rho(e, i, x_i) = \langle 4, e', \rho(d_1, i, x_i), \ldots, \rho(d_n, i, x_i)\rangle.$$

Case v) splits into two subcases. If $1 < i$, this case is easy. For $i = 1$ we let
$\rho(e, 1, 0) = d_1$.
$\rho(e, 1, x + 1) = \langle 4, \langle 3, x\rangle, \rho(e, 1, x), d_x, \langle 2, 1\rangle, \ldots, \langle 2, n\rangle\rangle$
where $\langle 3, x\rangle$ is the index for $f(\vec{x}) = x$ and $\langle 2, i\rangle$ is the index for the function selecting x_i from $\vec{x}$. In this case the primitive recursion is replaced by an iterated composition, the depth of which is determined by the value of x.
Case vi) is again easy, and is left for the reader.

Theorem 1.2.28 (The S^n_m-theorem)
Let $n \geq 1, m \geq 1$. There is a primitive recursive function S^n_m such that for all $e, x_1, \ldots, x_n, y_1, \ldots, y_m$

$$\phi^{n+m}_e(x_1, \ldots, x_n, y_1, \ldots, y_m) = \phi^m_{S^n_m(e,x_1,\ldots,x_n)}(y_1, \ldots, y_m)$$

Proof
Let ρ be as in Lemma 1.2.27.
Let $S^1_m(e, x) = \rho(e, 1, x)$
Let $S^{k+1}_m(e, x_1, \ldots, x_{k+1}) = \rho(S^k_{m+1}(e, x_1, \ldots, x_k), 1, x_{k+1})$. By an easy induction on k we see that this construction works for all m.

The S^n_m-theorem is a handy tool in itself, and we will use it frequently stating that we can find an index for a computable function uniformly in some parameter. Now we will use the S^n_m-theorem to prove the surprisingly strong

Theorem 1.2.29 (The Recursion Theorem)
Let $f(e, \vec{x})$ be a partial, computable function.
Then there is an index e_o such that for all $\vec{x}$

$$\phi_{e_0}(\vec{x}) \simeq f(e_0, \vec{x}).$$

Proof
Recall that by this equality we mean that either both sides are undefined or both sides are defined and equal. Let

$$g(e, \vec{x}) = f(S_n^1(e, e), \vec{x})$$

and let $\hat{g}$ be an index for g. Let

$$e_0 = S_n^1(\hat{g}, \hat{g}).$$

Then

$$\phi_{e_0}(\vec{x}) = \phi_{S_n^1(\hat{g},\hat{g})}(\vec{x}) = \phi_{\hat{g}}(\hat{g}, \vec{x}) = g(\hat{g}, \vec{x}) = f(S_n^1(\hat{g}, \hat{g}), \vec{x}) = f(e_0, \vec{x}).$$

Remark 1.2.30 Readers familiar with the fixed point construction in untyped λ-calculus may recognize this proof as a close relative, and indeed it is essentially the same proof. The recursion theorem is a very powerful tool for constructing computable functions by self reference. In Chapter 2 we will use the recursion theorem to construct computable functions essentially by transfinite induction. Here we will give a completely different application, we will prove that there is no nontrivial set of partial computable functions such that the set of indices for functions in the class is computable.

Theorem 1.2.31 (Riece)
Let $A \subseteq \mathbb{N}$ be a computable set such that if $e \in A$ and $\phi_e = \phi_d$ then $d \in A$.
Then $A = \mathbb{N}$ or $A = \emptyset$.

Proof
Assume not, and let $a \in A$ and $b \notin A$.
Let $f(e, x) = \phi_b(x)$ if $e \in A$ and $f(e, x) = \phi_a(x)$ if $e \notin A$.
By the recursion theorem, let e_0 be such that for all x

$$f(e_0, x) = \phi_{e_0}(x).$$

If $e_0 \in A$, then $\phi_{e_0} = \phi_b$ so $e_0 \notin A$.
If $e_0 \notin A$, then $\phi_{e_0} = \phi_a$ so $e_0 \in A$.
This is a clear contradiction, and the theorem is proved.

Corollary 1.2.32 (Unsolvability of the halting problem)
$\{(e, x) \mid \phi_e(x)\downarrow\}$ is not computable.

Remark 1.2.33 Riece's theorem is of course stronger than the unsolvability of the Halting Problem, for which we need much less machinery.

1.2.3 Computably enumerable sets

Four equivalent definitions

An enumeration of a set X is an onto map $F : \mathbb{N} \to X$. For subsets of $\mathbb{N}$ we may ask for computable enumerations of a set. A set permitting a computable enumeration will be called *computably enumerable* or just *c.e.* The standard terminology over many years has been *recursively enumerable* or just *r.e.*, because the expression *recursive* was used by Kleene and many with him. We will stick to the word *computable* and thus to the term *computably enumerable.*
Of course there is no enumeration of the empty set, but nevertheless we will include the empty set as one of the c.e. sets.
In this section we will give some characterizations of the c.e. sets. One important characterization will be as the semi-decidable sets. A computable set will be decidable, we have an algorithm for deciding when an element is in the set or not. In a semi-decidable set we will have an algorithm that verifies that an element is in the set when it is, but when the element is not in the set, this algorithm may never terminate. A typical semi-decidable set is the solving set of the Halting Problem

$$\{(e, x) \mid \phi_e(x)\downarrow\}.$$

Another example is the set of theorems in first order number theory or any nicely axiomatizable theory. The set of words in some general grammar will form a third class of examples.

We will show that the semi-decidable subsets of $\mathbb{N}$ will be exactly the c.e. sets. A third characterization will be that the c.e. sets are exactly the sets of projections of primitive recursive sets. In the literature this class is known as the Σ^0_1-sets. A fourth characterization will be as the ranges of partial, computable functions.

This is enough talk, let us move to the definition:

Definition 1.2.34 Let $A \subseteq \mathbb{N}$. We call A *computably enumerable* or just *c.e.* if $A = \emptyset$ or A is the range of a total computable function.

Theorem 1.2.35 *Let $A \subseteq \mathbb{N}$. Then the following are equivalent:*

i) A is c.e.

ii) A is the range of a partial computable function.

iii) There is a primitive recursive set $S \subseteq \mathbb{N}^2$ such that

$$A = \{n \mid \exists m (n, m) \in S\}$$

iv) There is a partial computable function with domain A.

Proof
Since the empty set satisfies all four properties, we will assume that $A \neq \emptyset$.
i) $\Rightarrow$ ii):

Trivial since we in this compendium consider the total functions as a subclass of the partial functions.
ii) $\Rightarrow$ iii):
Let A be the range of ϕ_e.
Then

$$n \in A \Leftrightarrow \exists y(T(e, \pi_1(y), \pi_2(y)) \wedge n = U(\pi_2(y)))$$

where π_1 and π_2 are the inverses of the pairing function P, T is Kleene's T-predicate and U is the function selecting the value from a computation tree. The matrix of this expression is primitive recursive.
iii) $\Rightarrow$ iv): Let

$$n \in A \Leftrightarrow \exists m((n, m) \in S).$$

where S is primitive recursive. Then A is the domain of the partial computable function

$$f(n) = \mu m.(n, m) \in S.$$

iv) $\Rightarrow$ i):
Let A be the domain of ϕ_e and let $a \in A$ (here we will use the assumption that A is non-empty).
Let $f(y) = \pi_1(y)$ if $T(e, \pi_1(y), \pi_2(y))$, $f(y) = a$ otherwise. Then f will be computable, and A will be the range of f.
This ends the proof of the theorem.

Clearly characterizations ii) and iii) make sense for subsets of $\mathbb{N}^n$ as well for $n > 1$, and the equivalence will still hold. From now on we will talk about c.e. sets of any dimension. The relationship between c.e. subsets of $\mathbb{N}$ and $\mathbb{N}^n$ is given in Exercise 1.14.

The following lemma will rule our abilities to construct c.e. sets:

Lemma 1.2.36 *Let $A \subseteq \mathbb{N}^n$ and $B \subseteq \mathbb{N}^n$ be c.e. Then*

a) $A \cap B$ and $A \cup B$ are both c.e.

b) If $n = m + k$ and both m and k are positive, then

$$\{\vec{x} \mid \exists \vec{y}(\vec{x}, \vec{y}) \in A\}$$

will be c.e., where $\vec{x}$ is a sequence of variables of length m and $\vec{y}$ is of length k.

Moreover, every computable set is c.e., the inverse image or direct image of a c.e. set using a partial computable function will be c.e.

All these claims follow trivially from the definition or from one of the characterizations in Theorem 1.2.35.

Clearly, a finite set will not permit a 1-1 enumeration. It turns out that any infinite c.e. set is the range of a 1-1 computable function. This will turn out to be a useful observation:

Lemma 1.2.37 *Let $f : \mathbb{N} \to \mathbb{N}$ be computable with an infinite range. Then there is a 1-1 computable function g with the same range as f.*

Proof
We define g from f by recursion as follows

$$g(0) = f(0)$$

$$g(n+1) = f(\mu m(f(m) \notin \{g(0), \ldots, g(n)\})).$$

Then g is computable and, by construction, with the same range as f. Also see Exercise 1.13

Let us introduce another standard piece of notation:

Definition 1.2.38 Let

$$W_e = \{n \mid \phi_e(n)\downarrow\} = \{n \mid \exists t T(e, n, t)\}.$$

Let

$$W_{e,m} = \{n \mid \exists t < m T(e, n, t)\}.$$

We let $\phi_{e,m}$ be the finite subfunction of ϕ_e where we restrict ourselves to computations with computation trees bounded by m.
We let $\mathcal{K}$ be the *diagonal set*

$$\mathcal{K} = \{e \mid e \in W_e\}.$$

Lemma 1.2.39 *a) $\{(e, n) \mid n \in W_e\}$ is c.e.*

b) $\mathcal{K}$ is c.e.

c) Each set $W_{e,m}$ is finite.

d) $\{(e, n, m) \mid n \in W_{e,m}\}$ is primitive recursive.

All proofs are trivial.

Selection with consequences

From now on we will prove lemmas and theorems in the lowest relevant dimension, but clearly all results will hold in higher dimensions as well.

Theorem 1.2.40 (The Selection Theorem)
Let $A \subseteq \mathbb{N}^2$ be c.e. Then there is a partial computable function f such that

i) $f(n)\downarrow \Leftrightarrow \exists m(n, m) \in A$.

ii) If $f(n)\downarrow$ then $(n, f(n)) \in A$.

Proof
This time we will give an intuitive proof. Let A be the projection of the primitive recursive set $B \subseteq \mathbb{N}^3$ (characterization iii).). For each n, search for the least m such that $(n, \pi_1(m), \pi_2(m)) \in B$, and then let $f(n) = \pi_1(m)$.

Intuitively we perform parallel searches for a witness to the fact that $(n, m) \in A$ for some m, and we choose the m for which we first observe a witness.

Corollary 1.2.41 *Let A and B be two c.e. sets. Then there are disjoint c.e. sets C and D with*

$$C \subseteq A,\ D \subseteq B \text{ and } A \cup B = C \cup D.$$

Proof
Let $E = (A \times \{0\}) \cup (B \times \{1\})$ and let f be a selection function for E.
Let $C = \{n \mid f(n) = 0\}$ and $D = \{n\,;\, f(n) = 1\}$.
Then C and D will satisfy the properties of the corollary.

Corollary 1.2.42 *A set A is computable if and only A and the complement of A are c.e.*

One way is trivial, since the complement of a computable set is computable and all computable sets are c.e. So assume that A and its complement B are c.e.
Let E be as in the proof of the corollary above, and f the selection function. Then f is the characteristic function of A, so A is computable.

Corollary 1.2.43 *Let $f : \mathbb{N} \to \mathbb{N}$ be a partial function. Then the following are equivalent:*

i) f is computable.

ii) The graph of f is c.e.

Proof
If the graph of f is c.e., then f will be the selection function of its own graph, which is computable by the selection theorem. If on the other hand f is computable, then the graph of f will be the domain of the following function $g(n, m)$: Compute $f(n)$ and see if the result equals m.

Computably inseparable c.e. sets

In Exercise 1.16 we will see that two disjoint complements of c.e. sets can be separated by a computable set. This is an improvement of Corollary 1.2.43. Here we will show that a similar separation property does not hold for c.e. sets, and we will draw some consequences of this fact.

Definition 1.2.44 Let A and B be two disjoint subsets of $\mathbb{N}$. We say that A and B are *computably separable* if there is a computable set C such that $A \subseteq C$ and $B \cap C = \emptyset$. Otherwise A and B are *computably inseparable*.

Theorem 1.2.45 *There is a pair of computably inseparable c.e. sets.*

Proof
Let $A = \{e \mid \phi_e(e) = 0\}$ and $B = \{e \mid \phi_e(e) = 1\}$.
Assume that C is a computable set that separates A and B, and assume that e_0 is an index for the characteristic function of C.
Then, if $e_0 \in C$ we have that $\phi_{e_0}(e_0) = 1$. Then $e_0 \in B$ which is disjoint from C.
Likewise, if $e_0 \notin C$ we see that $e_0 \in A \subseteq C$. In both cases we obtain a contradiction, so the existence of C is impossible.

Now this theorem has some interesting consequences concerning the difference between classical and constructive mathematics. We will end our general introduction to the basics of computability theory discussing some of the consequences.

Definition 1.2.46 a) A *binary tree* is a non-empty set D of finite 0-1-sequences such that any initial segment of an element in D is also in D. A binary tree is *computable* if the set of sequence numbers of the sequences in D is computable.

b) An *infinite branch* in a binary tree D is a function $f : \mathbb{N} \to \{0, 1\}$ such that $(f(0), \ldots, f(n-1)) \in D$ for all n.

Lemma 1.2.47 (König's Lemma)
An infinite binary tree has an infinite branch.

The proof of König's lemma is trivial, but has very little to do with computability theory. One constructs a branch by always extending the sequence in a direction where the tree is still infinite.
Well known theorems proved by similar arguments will be that a continuous function on a closed bounded interval will obtain its maximum and that any consistent first order theory has a complete extension.

Remark 1.2.48 The version of König's Lemma given above, restricting ourselves to binary trees, is often called *Weak König's lemma*, abbreviated **WKL**. The full König's lemma then refers to infinite trees with finite branching, not just binary branching. There are results showing that **WKL** is equivalent, relative to some very weak formal theory, to the mentioned theorems from analysis and topology.
Relative to the same very weak theory, **WKL** is weaker than the full König's lemma.

We will show a failure of a constructive version of König's lemma:

Lemma 1.2.49 *There is an infinite, computable binary tree without a computable, infinite branch.*

Proof
Let A and B be two computably inseparable c.e. sets and let $\{A_n\}_{n\in\mathbb{N}}$ and $\{B_n\}_{n\in\mathbb{N}}$ be primitive recursive sequences of sets A_n and B_n contained in $\{0,\ldots,n-1\}$, with $A=\bigcup_{n\in\mathbb{N}} A_n$ and $B=\bigcup_{n\in\mathbb{N}} B_n$.
If σ is a 0-1-sequence of length n, we let $\sigma\in D$ if for all $i<n$: $i\in A_n \Rightarrow \sigma(i)=1$ and $i\in B_n \Rightarrow \sigma(i)=0$. σ will be a characteristic function on $\{0,\ldots,n-1\}$ separating A_n and B_n.
Now the characteristic function of any set separating A and B will be an infinite branch in D. Since A and B are disjoint, D will be an infinite, binary tree. Moreover, D will be primitive recursive. A computable, infinite branch will on the other hand be the characteristic function of a computable set separating A and B, something assumed not to exist. This ends the proof.

In Exercises 1.17 and 1.18 we will give an application of this lemma to propositional calculus and to constructive analysis.

1.3 Degrees of Unsolvability

1.3.1 m-reducibility

Discussion

Having introduced the main concepts of computability theory there are several options. One option will be to give a further analysis of the computable functions and subclasses of them. This will lead us to complexity theory or to subrecursive hierarchies (see Section 1.7). One key concept then will be that one set or function can be reduced to another in some simple way.
We will not consider such a fragmentation of the computable functions into interesting subclasses yet. Instead we will ask for reductions between possible non-computable functions and sets, using the complexity of computability itself for defining the notion of reduction. The connection between this approach and the ones mentioned above is that both the formation of interesting concepts and the methodology of those approaches to computability theory are based on the experience gained from investigating computable reductions in general. Thus, in an introductory course, where one should learn the computability theorists way of thinking, this section is basic.

Definition 1.3.1 Let A and B be two subsets of $\mathbb{N}$.
We say that A *is m-reducible to* B, $A <_m B$, if there is a total computable function f such that for all n:

$$n\in A \Leftrightarrow f(n)\in B.$$

We read this as 'many-one-reducible', since f may be a many-one function. If we insist on f being injective, we will get 1-reducibility.

There is no way we can reduce $\mathbb{N}$ to $\emptyset$ and vice versa, and we will exclude those *trivial* sets from our further discussion. We then get the observations:

Lemma 1.3.2 *a) If A is computable and $B \neq \mathbb{N}, \emptyset$, then $A <_m B$.*

b) $A <_m B \Leftrightarrow (\mathbb{N} \setminus A) <_m (\mathbb{N} \setminus B)$.

c) $<_m$ is transitive.

The proofs are easy and are left for the reader.

Definition 1.3.3 We call two sets A and B *m-equivalent* if $A <_m B$ and $B <_m A$. We then write $A \equiv_m B$.

Clearly $\equiv_m$ will be an equivalence relation with a partial ordering inherited from $<_m$. We call the set of equivalence classes with this induced ordering *the m-degrees*.

Remark 1.3.4 Let us briefly discuss the distinction between a *complexity class* and a *degree*. A complexity class will consist of all functions and/or sets that may be computed or decided with the aid of a prefixed amount of computing resources. Resources may be the time or space available (normally as functions of the input), but to us it may also be the possible service of a non-computable oracle. This will be better explained later. A degree, on the other hand, will be some mathematical object measuring the complexity of a function or a set. It has turned out that the most useful object for this purpose simply is the class of functions of the same complexity as the given object. It is like the old Frege semantics where 17 is interpreted as the set of all sets with 17 elements, and in general a property is interpreted as the set of all objects sharing that property. It remains to decide what we mean by 'having the same complexity', and we will consider three (out of many in the literature) possible choices. Being m-equivalent is one, and later we will learn about *truth table equivalent* sets and *Turing equivalent functions*.

The m-degrees have some properties that are easy to establish:

Lemma 1.3.5 *a) Let X be a finite set of m-degrees. Then X has a least upper bound.*

b) Let X be a countable set of m-degrees. Then X has an upper bound.

Proof
The computable sets will form the minimal m-degree. Thus the empty set of m-degrees has a least upper bound. In order to prove the rest of a) it is sufficient to show that $\{A, B\}$ will have a least upper bound. Let

$$A \oplus B = \{2n \mid n \in A\} \cup \{2n+1 \mid n \in B\}.$$

This will define the least upper bound, see Problem 1.19.
In order to prove b), Let $\{A_n \mid n \in \mathbb{N}\}$ be a countable family of sets. Let

$$A = \Sigma_{n \in \mathbb{N}} A_n = \{\langle n, m \rangle \mid m \in A_n\}.$$

Then the m-degree of A will bound the m-degrees of A_n for each $n \in \mathbb{N}$.

We will invest most of our efforts in analyzing the Turing degrees below. We will however prove one negative result about the m-degrees, they do not represent a linear stratification of all nontrivial sets into complexity classes.

Lemma 1.3.6 *There are two nontrivial sets A and B such that $A \not\leq_m B$ and $B \not\leq_m A$.*

Proof
Let A be c.e. but not computable. Let $B = \mathbb{N} \setminus A$.
If $C <_m A$, C will be c.e., so B is not m-reducible to A. It follows from Lemma 1.3.2 that A is not m-reducible to B.

An m-complete c.e. set

We proved that the computable inverse of a c.e. set is c.e. A reformulation of this will be

Lemma 1.3.7 *Let $A <_m B$. If B is c.e., then A is c.e.*

Thus the c.e. sets form an initial segment of all sets pre-ordered by m-reductions. First we will show that this set has a maximal element:

Lemma 1.3.8 *Let $\mathcal{K} = \{e \mid \phi_e(e)\downarrow\}$. Then any other c.e. set is m-reducible to $\mathcal{K}$.*

Proof
Let $A = \{d \mid \phi_e(d)\downarrow\}$. Adding a dummy variable we can without loss of generality assume that $A = \{d \mid \phi_e(d, x)\downarrow\}$ where the outcome of $\phi_e(d, x)$ is independent of x. Then $A <_m \mathcal{K}$ by

$$A = \{d \mid S_1^1(e, d) \in \mathcal{K}\}.$$

Remark 1.3.9 $\mathcal{K}$ is in a sense the prototype of a complete c.e. set, and we will make further use of $\mathcal{K}$ is the sequel.

A natural question is now if there are c.e. sets that are not m-equivalent to $\mathcal{K}$ or the computable sets; are there more m-degrees among the c.e. sets? This was answered by Emil Post, when he classified a whole class of c.e. sets 'in between'.

Simple sets

Definition 1.3.10 A c.e. set A is *simple* if the complement of A is infinite, but does not contain any infinite c.e. sets.

A simple set cannot be computable (why?). We will prove that there exist simple sets, and that $\mathcal{K}$ cannot be reduced to any simple sets.

Lemma 1.3.11 *There exists a simple set.*

Proof
Let $B = \{(e, x) \mid 2e < x \wedge \phi_e(x)\downarrow\}$.
Let g be a computable selection function for B, i.e. $g(e)\downarrow$ when $(e, x) \in B$ for some x, and then $g(e)$ selects one such x.
Let A be the image of g. Then A is c.e.
Since $g(e) > 2e$ when defined, the complement of A will be infinite. This is because each set $\{0, \ldots, 2e\}$ will contain at most e elements from A.
If W_e is infinite, W_e will contain a number $> 2e$, and $g(e)$ will be defined. Then $A \cap W_e \neq \emptyset$, so W_e is not contained in the complement of A. This shows that A is simple.

Lemma 1.3.12 *Let $\mathcal{K} <_m A$ where A is c.e.*
Then the complement of A contains an infinite c.e. set.

Proof
Let f be total and computable such that

$$e \in \mathcal{K} \Leftrightarrow f(e) \in A.$$

We will construct a computable sequence $\{x_i\}_{i\in\mathbb{N}}$ of distinct numbers outside A, and use the recursion theorem to glue the whole construction together.
The induction start will be the empty sequence.
Assume that $B_n = \{x_0, \ldots, x_{n-1}\}$ has been constructed such that B_n is disjoint from A.
Let $\rho(n)$ be such that $W_{\rho(n)} = \{e \mid f(e) \in B_n\}$. By the uniformity of the construction, ρ will be computable. We will let $x_n = f(\rho(n))$.
If $x_n \in B_n$ we have that $x_n \notin A$, so $\rho(n) \notin \mathcal{K}$, and $\rho(n) \notin W_{\rho(n)}$. This contradicts the assumption that $x_n \in B_n$ and the definition of $W_{\rho(n)}$. Thus $x_n \notin B_n$.
On the other hand, if $x_n \in A$, then $\rho(n) \in \mathcal{K}$, $\rho(n) \in W_{\rho(n)}$ and by construction, $x_n \in B_n$, which we agreed was impossible. Thus x_n is a new element outside A.
We can then continue the process and in the end produce an infinite c.e. set outside A.

Corollary 1.3.13 *There is a non-computable c.e. set A such that $\mathcal{K}$ is not m-reducible to A.*

The construction above of a simple set is in a sense ad hoc. In Exercise 1.32 we give an alternative construction of simple sets, and we prove an important strengthening of Corollary 1.3.13.

1.3.2 Truth table degrees

As an intermediate step from m-reducibility to Turing-reducibility, we will discuss a concept known as *truth table reducibility* or simply *tt-reducibility*.
When $A <_m B$ we can decide if $a \in A$ by asking one question $f(a) \in B$? about

membership in B. When we work with c.e. sets A and B, we think of A and B as *semicomputable* sets, and then $A <_m B$ means that we can uniformly describe an algorithm terminating on B from any algorithm terminating on A. This concept is intimately related to reducibilities between formal theories, where $T < T'$ if there is a computable function f that to any formula Φ in the language of T gives a formula $f(\Phi)$ in the language of T' such that

$$T \vdash \Phi \Leftrightarrow T' \vdash f(\Phi).$$

If we work in another context, m-reducibility may seem artificially restricted. Why should we not be allowed to claim that the complement of a set B is reducible to B, since we can easily transform a 'yes' to a 'no'?
If we allow ourselves to use negative information about B in order to decide if a number a is in A, there is of course no reason to restrict this to only one question $f(a)$ about B. This leads us to tt-reducibility, and in turn to Turing-reducibility.

Given a set variable X (for a set $X \subseteq \mathbb{N}$) we will consider all atomic statements $a \in X$ as propositional variables. We then consider the language $\mathcal{L}$ of propositional formulas over these variables, for simplicity restricting ourselves to the connectives $\neg$ and $\vee$.
To each formula Φ in $\mathcal{L}$, we associate a number code $[\![\Phi]\!]$ as follows:

$$[\![a \in X]\!] = \langle 0, a \rangle.$$

$$[\![\neg\Phi]\!] = \langle 1, [\![\Phi]\!] \rangle.$$

$$[\![\Phi \vee \Psi]\!] = \langle 2, [\![\Phi]\!], [\![\Psi]\!] \rangle.$$

We leave it as Exercise 1.20 to prove that the set of codes for formulas in $\mathcal{L}$ is primitive recursive.

Definition 1.3.14 a) If Φ is a formula in $\mathcal{L}$ and B is a set, we let $B \models \Phi$ mean that Φ will be true under the valuation $a \in B$ when a varies over $\mathbb{N}$.

b) If $n \in \mathbb{N}$, we let $B \models n$ mean that there is a formula Φ with $[\![\Phi]\!] = n$ such that $B \models \Phi$.

Definition 1.3.15 Let A and B be two sets. We say that A is *truth table reducible* to B, $A <_{tt} B$, if there is a total, computable function $f : \mathbb{N} \to \mathbb{N}$ such that for all $a \in \mathbb{N}$

$$a \in A \Leftrightarrow B \models f(a).$$

Example 1.3.16 If B is a set and

$$A = \{2n \mid n \in B\} \cup \{2n+1 \mid n \notin B\}$$

then $B <_m A$ and $A <_{tt} B$.

We leave the verification of this claim for the reader.

Lemma 1.3.17 *a) If $A <_{tt} B$ and $B <_{tt} C$ then $A <_{tt} C$.*

b) If A, B and C are sets, and

$$A \oplus B = \{2n \mid n \in A\} \cup \{2n+1 \mid n \in B\}$$

then

$$A \oplus B <_{tt} C \Leftrightarrow A <_{tt} C \wedge B <_{tt} C.$$

We leave the proof as Exercise 1.21.

This lemma has as a consequence that the relation

$$S \equiv_{tt} B \leftrightarrow A <_{tt} B \wedge B <_{tt} A$$

is an equivalence relation. A further consequence is that the quotient ordering of the set of equivalence classes is an upper semi-lattice. We will call these equivalence classes for *tt-degrees*.
Instead of investigating this degree-structure in depth, we will move on to the Turing degrees.

1.3.3 Turing degrees

Relativised computations

In the previous sections we considered m-reducibility ant tt-reducibility. The former is a natural concept when we compare sets where the positive part has a kind of complexity that differs from that of the negative part. A typical case will be c.e.-sets, but in the next chapter we will see other examples of classes of sets definable in a prefixed manner. When we identify a set with its characteristic function, we somehow looses this aspect of a set and its complement being of a different nature, and then tt-reducibility is a more natural concept.
When $A <_{tt} B$, the set of questions about membership in B we ask may depend on the question $a \in A$?, but not on B itself. This means that we do not allow new questions about B to be asked based on the answers of previous questions. We will now extend the concept of reducibility, both allowing repeated questioning, and reducing functions to functions instead of sets to sets. We extend the definition of the computable functions by adding some extra functions as initial ones. This is technically described in the following definition:

Definition 1.3.18 Let $f_1, \ldots, f_k$ be a finite list of partial functions $f_i : \mathbb{N} \to \mathbb{N}$.

a) We extend the definition of $\phi_e(\vec{x})$ to a definition of

$$\phi_e^{f_1,\ldots,f_k}(\vec{x})$$

by adding a new clause
vii) If $e = \langle 7, i, k\rangle$ then $\phi_e^{f_1,\ldots,f_k}(x, \vec{x}) = f_i(x)$. If $f_i(x)$ is undefined, then this computation does not terminate.

b) The computation tree will in this case be $\langle 7, i, k, x, f_i(x)\rangle$.

We call these algorithms *relativized algorithms* meaning that the concept of computation has been relativized to $f_1, \ldots, f_k$. One important aspect of this definition is the *finite use principle*. Even if we in reality accept functions as inputs in algorithms as well, no terminating algorithm will use more than a finite amount of information from the functions involved.

Lemma 1.3.19 *Assume that* $\phi_e^{f_1,\ldots,f_k}(\vec{x}) = z$. *Then there are finite partial subfunctions* $f_1', \ldots, f_k'$ *of* $f_1, \ldots, f_k$ *resp. such that*

$$\phi_e^{f_1',\ldots,f_k'}(\vec{x}) = z$$

with the same computation tree.

The proof is by a tedious but simple induction on e, using the fact that a sequence number is larger than all parts of the sequence.

We have given and used an enumeration of all ordered sequences of natural numbers. In the same fashion we can construct an enumeration $\{\xi_i\}_{i\in\mathbb{N}}$ of all finite partial functions such that the following relations will be primitive recursive:

- $\xi_i(x)\downarrow$
- $\xi_i(x) = y$
- $dom(\xi_i) \subseteq \{0, \ldots, n\}$.

The reader is free to ad any other important properties, as long as they are correct.

Example 1.3.20 In order to avoid that the reader will focus on inessential details, we offer the construction of ξ_i in the guise of an example.
Let $p_0, p_1, p_2, \ldots$ be an enumeration of the prime numbers, starting with $p_0 = 2$, $p_1 = 3$ and so on.
Define ξ_i by

$\xi_i(n) = m$ if there are exactly $m + 1$ factors of p_n in i.

$\xi_i\uparrow$ if p_n is not a factor in i.

For example, if $i = 2^3 \cdot 7^4 \cdot 11$, then $\xi_i(0) = 2$, $\xi_i(3) = 3$ and $\xi_i(4) = 0$. For all other inputs x, $\xi_i(x)$ will be undefined.

The properties above are sufficient to prove the extended Kleene T-predicate:

Lemma 1.3.21 *For each* n *and* k, *the following relation is primitive recursive:* $T_{n,k}(e, \vec{x}, i_1, \ldots, i_k, t) \Leftrightarrow t$ *is the computation tree of* $\phi_e^{\xi_{i_1},\ldots,\xi_{i_k}}(\vec{x})$.

In order to save notation we will from now on mainly consider computations relativized to one function. Using the coding

$$\langle f_0, \ldots, f_{k-1} \rangle(x) = f_{\pi_1(x)(mod\ k)}(\pi_2(x))$$

we see that there is no harm in doing this.

Lemma 1.3.22 *There is a primitive recursive function c such that for any partial functions f, g and h, if for all x, $f(x) = \phi_e^g(x)$ and for all x, $g(x) = \phi_d^h(x)$, then for all x, $f(x) = \phi_{c(e,d)}^h(x)$.*

Proof
$c(e, d)$ is defined by primitive recursion on e, dividing the construction into cases i) - xi).
For cases i) - iii), we let $c(e, d) = e$.
For the cases iv) - viii) we just let $c(e, d)$ commute with the construction of e from its subindices, i.e., if the case is $e = expr(e_1, \ldots, e_n)$ then $c(e, d) = expr(c(e_1, d), \ldots, c(e_n, d))$, where $expr$ can be any relevant expression.
If $e = \langle 9, 1, 1 \rangle$ we have $\phi_e^g(x, \vec{x}) = g(x) = \phi_d^h(x)$, so we let $c(e, d) = d'$ where $\phi_{d'}^h(x, \vec{x}) = \phi_d^h(x)$. d' is primitive recursive in d.

Remark 1.3.23 There is an alternative proof of this lemma. By the finite use principle we see that if g is computable in h and f is computable in g, then the graph of f will be c.e. relative to h. Everything will be uniform, so we may extract c from this proof as well.

We can use the concepts introduced here to talk about computable *functionals*, and not just about computable functions.

Definition 1.3.24 Let $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$, a functional of type 2.
We call F *computable* if there is an index e such that for all total $f : \mathbb{N} \to \mathbb{N}$ we have

$$F(f) = \phi_e^f(0).$$

All computable functionals of type 2 will be continuous with respect to the canonical topologies. Kleene extended the notion of relativized computation to cover all functionals of any finite type as possible inputs. This will be discussed briefly in Chapter 2.

The second recursion theorem

We are now in the position to state and prove the second recursion theorem:

Theorem 1.3.25 *Let $F(f, x) = \phi_e^f(x)$ be a computable functional.*
Then there is a computable f such that for all x,

$$f(x) = F(f, x).$$

Moreover, f will be the least such function in the sense that if

$$\forall x(g(x) = F(g, x)),$$

then $f \subseteq g$ as graphs.

Proof
Let f_0 be the everywhere undefined function, and by recursion let $f_{n+1}(x) = F(f_n, x)$.
Since $f_0 \subseteq f_1$ we see by induction on n that $f_n \subseteq f_{n+1}$ for all n.
Let $f = \bigcup\{f_n \mid n \in \mathbb{N}\}$. By the finite use principle we see that

$$F(f, x) = y \Leftrightarrow \exists n F(f_n, x) = y \Leftrightarrow \exists n f_{n+1}(x) = y \Leftrightarrow f(x) = y.$$

Clearly f is the minimal solution to the equation, the minimal *fixed point* of F.
It remains to show that f is computable, i.e. that the graph of f is c.e.
We will use the effective enumeration of all finite partial functions and the effective coding of sequences. Let ξ with indices vary over all finite partial functions. Then we can rewrite the following characterization of the graph of f int a Σ^0_1-form:

$$f(x) = y \Leftrightarrow \exists n \exists \langle \xi_1, \ldots, \xi_n \rangle (\forall z \xi_0(z)\uparrow$$
$$\wedge \forall i < n (\forall z \in dom(\xi_{i+1})(\xi_{i+1}(z) = F(\xi_i, z)) \wedge \xi_n(x) = y)).$$

Remark 1.3.26 The existence of a computable fixed point for F follows by the recursion theorem. It is possible to show that the index obtained by the proof of the recursion theorem will be an index for the least fixed point, see Exercise 1.22.

Turing Degrees

We will now restrict ourselves to total functions.

Definition 1.3.27 Let f and g be two functions. We say that f *is computable in* g if there is an index e such that for all x,

$$f(x) = \phi^g_e(x).$$

We write $f <_T g$. We will then say that f is *Turing reducible to* g.

The key properties of Turing reducibility is given by

Lemma 1.3.28 *a)* $<_T$ *is a transitive relation.*

b) If f *is computable and* g *is any function, then* $f <_T g$.

c) If f *and* g *are functions, there is a function* h *such that for any other function* h'*:*

$f <_T h$ *and* $g <_T h$.

If $f <_T h'$ *and* $g <_T h'$ *then* $h <_T h'$.

Proof
a) is a consequence of Lemma 1.3.22, b) is trivial, and to prove c), let
$h(2n) = f(n)$ and $h(2n+1) = g(n)$.

Definition 1.3.29 Let f and g be two functions. f and g are *Turing equivalent*, in symbols $f \equiv_T g$, if $f <_T g$ and $g <_T f$.

$\equiv_T$ will be an equivalence relation. The equivalence classes will be called *Turing degrees* or *Degrees of unsolvability*. We will simply call them *degrees*. We will let bold-face low case letters early in the alphabet , **a**, **b**, etc. denote degrees. The set of degrees has a canonical ordering $<$ inherited from $<_T$.

We can summarize what we have observed so far in

Lemma 1.3.30 *The ordered set of degrees is an upper semi-lattice with a least element such that every countable set is bounded, and every initial segment is countable.*

We leave the verifications for the reader. We will now see that there is no maximal degree:

Lemma 1.3.31 *Let* **a** *be a degree. Then there is a degree* **b** *such that*

$$\mathbf{a} < \mathbf{b}.$$

Proof
Let $f \in \mathbf{a}$. Let $g(x) = \phi^f_{\pi_1(x)}(\pi_2(x)) + 1$ if $\phi^f_{\pi_1(x)}(\pi_2(x))\downarrow$, otherwise $g(x) = 0$. The proof of the unsolvability of the halting problem can be relativized to f so g is not computable in f. On the other hand, clearly f is computable in g.

The g constructed in the proof above is called f', *the jump of* f. The jump operator is indeed a degree-operator, see Exercise 1.23.

Before proving further results about Turing degrees in general, we will relate them to m-degrees and tt-degrees, in order to see that we have created something genuinely stronger.
By the results proved in Exercise 1.13 It follows that the Turing degrees and the m-degrees of c.e. sets are not in general the same. We will prove a similar result for tt-reducibility:

Lemma 1.3.32 *There is a set B that is Turing equivalent to $\mathcal{K}$ but not tt-reducible to $\mathcal{K}$.*

Proof
Let

$$e \in A \leftrightarrow e \notin \mathcal{K} \vee \mathcal{K} \not\models \phi_e(e)$$

and let $B = A \oplus \mathcal{K}$. Then $\mathcal{K}$ is m-reducible to B.
We first show that A is computable relative to $\mathcal{K}$: If $e \notin \mathcal{K}$ then $e \in A$. If $e \in \mathcal{K}$, then $e \in A$ exactly when $\mathcal{K} \not\models \phi_e(e)$. This can be decided with the help of $\mathcal{K}$.
We then see by a diagonal argument that A cannot be tt-reducuble to $\mathcal{K}$ via any total ϕ_e. Since A is even m-reducible to B, it follows that B cannot be tt-reducible to $\mathcal{K}$.

We have shown that there are incomparable m-degrees. The same method can be used to show that there are incomparable Turing degrees, see Exercise 1.24. We will prove a stronger result, showing that no strictly increasing sequence of degrees will have a least upper bound.

Theorem 1.3.33 *Let $\{\mathbf{a}_i\}_{i\in\mathbb{N}}$ be a strictly increasing sequence of degrees. Then there are two degrees $\mathbf{b}$ and $\mathbf{c}$ that are both upper bounds for the sequence, such that for any degree $\mathbf{d}$, if $\mathbf{d} < \mathbf{b}$ and $\mathbf{d} < \mathbf{c}$, then $\mathbf{d} < \mathbf{a}_i$ for some $i \in \mathbb{N}$.*

A pair $\mathbf{b}, \mathbf{c}$ as above is called *a perfect pair* for the sequence. The degrees in a perfect pair for a sequence will be incomparable. Further, the existence of a perfect pair shows that the sequence will have no least upper bound.

Proof
Let $\{f_i\}_{i\in\mathbb{N}}$ be a sequence of total functions of increasing Turing degrees. We will construct the functions g and h as the limits of approximations g_x and h_x, where we in the construction of g_{x+1} and h_{x+1} want to ensure that if $e_1 = \pi_1(x)$ and $e_2 = \pi_2(x)$ and $\phi^g_{e_1} = \phi^h_{e_2}$ are total, then $\phi^g_{e_1}$ is computable in f_x. In order to simplify the notation, we let g and h be defined on $\mathbb{N}^2$, but this will alter nothing.
In the construction we will preserve the following properties:

1. If $i < x$, then $g_x(i, n)$ and $h_x(i, n)$ are defined for all n.
2. If $i < x$, then $g_x(i, n) = f_i(n)$ for all but finitely many n.
3. If $i < x$, then $h_x(i, n) = f_i(n)$ for all but finitely many n.
4. $g_x(i, n)$ is defined only for finitely many (i, n) with $x \leq i$.
5. $h_x(i, n)$ is defined only for finitely many (i, n) with $x \leq i$.

This will ensure that g_x and h_x are equivalent to f_{x-1} for $x \geq 0$.
Let g_0 and h_0 both be the empty function.
Now, let $x \geq 0$ and assume that g_x and h_x are defined satisfying 1. - 5. above. Let $e_1 = \pi_1(x)$ and $e_2 = \pi_2(x)$. What we will do next will depend on the answer to the following question:
Can we find an n and finite extensions g' of g_x and h' of h_x such that
$\phi^{g'}_{e_1}(n) \neq \phi^{h'}_{e_2}(n)$ and both are defined?
(By a finite extension we mean that we ad a finite number of elements to the domain.) If the answer is 'no', we extend g_x to g_{x+1} by letting $g_{x+1}(x, n) = f_x(n)$ whenever this is not in conflict with the construction of g_x (a conflict that can appear at at most finitely many places), and we construct h_{x+1} from h_x and f_x in the same way.
If the answer is 'yes', we first choose two such finite extensions, and then we construct g_{x+1} and h_{x+1} from these extensions as above. This ends our construction.

We let $g(x, n) = g_{x+1}(x, n)$ and $h(x, n) = h_{x+1}(x, n)$. In the construction we have tried as hard as possible to avoid that $\phi^g_e = \phi^h_d$. The point is that we

have tried so hard that if they after all turn out to be equal, they will both be computable in one of the f_i's.
Claim 1
f_i is computable in both g and h.
Proof
We have that $f_i(n) = g(i, n)$ except for finitely many i, so f_i is computable in g. The same argument holds for h.
Claim 2
For $x \geq 0$ we have that g_x and h_x both are computable in f_{x-1}
Proof
This is a trivial consequence of properties 1. - 5.
Claim 3
If $\phi_e^g = \phi_d^h$ and both are total, then ϕ_e^g is computable in f_x for some x.
Proof
Let $x = P(e, d)$ where P is the pairing from Definition 1.2.13. Then in the construction of g_{x+1} and h_{x+1} we ask for a y and finite extensions g' and h' of g_x and h_x such that $\phi_e^{g'}(y) \neq \phi_d^{h'}(y)$. If we had found some, we would let g and h be further extensions of one such pair of finite extensions, and then we would have preserved that $\phi_e^g(y) \neq \phi_d^g(y)$, contradicting our assumption on e and d. On the other hand, for every y we can, by the assumption and the finite use principle, find finite extensions such that $\phi_e^{g'}(y)\downarrow$ and $\phi_d^{h'}(y)\downarrow$. The point is that all these values must be equal, otherwise we could have found two extensions giving different values. Thus we can give the following algorithm for computing $\phi_e^g(y)$ from g_x which again is computable in f_{x-1}: Search for any finite extension (by searching through the finite partial functions compatible with g_x) g' of g_x such that $\phi_e^{g'}(y)\downarrow$ The value we obtain will be the correct value.
This ends the proof of our theorem.

1.4 A minimal Turing degree

1.4.1 Trees

In the constructions of degrees we have performed so far, we have been using brute force. For instance, when we want to construct a minimal pair, we start with three properties:

1. f is not computable.
2. g is not computable
3. If h is computable in both f and g, then h is computable.

We then split these properties into infinite lists of *requirements*, which we try to satisfy during the construction:

$R_{1,e}$ If ϕ_e is total, then $f \neq \phi_e$.

$R_{2,e}$ If ϕ_e is total, then $g \neq \phi_e$.

$R_{3,e,d}$ If $\phi_e^f = \phi_d^g$ is total, then ϕ_e^f is computable.

Now, for any of these requirements and any pair σ and τ of finite sequences there will be finite extensions σ' and τ' such that any further total extension f and g of σ' and τ' resp. will satisfy the requirement. Thus by a step-by-step construction we can construct f and g via finite approximations satisfying one requirement at the time. The reader is invited to work out the full proof, see Exercise 1.25.

We will now face a problem which we cannot solve by this simple method. We will show that there is a non-computable function f such that there is no function of complexity strictly between f and the computable functions. Again we will set up the relevant properties of f, and fragmentize them into a sequence of requirements we want to satisfy during the construction. The problem will be that we cannot ensure that these requirements are satisfied by considering just a finite approximation of f. Instead we will use trees, and we will satisfy the various requirements by insisting that f is a branch in a given binary tree. Before we can go into details with the argument, we will reconsider our definition of a binary tree, and give it a formulation that will be handy for this particular application. We will not distinguish between finite sequences and sequence numbers here, but whenever we say that a function defined from a set of finite sequences to the set of finite sequences is computable, we mean that the corresponding function on sequence numbers is computable.

Definition 1.4.1 Let D be the set of finite 0-1-sequences.

a) If $\sigma \in D$ we let $\sigma * 0$ and $\sigma * 1$ be σ extended by 0 or 1 resp. We extend this to the concatenation $\sigma * \tau$ in the canonical way.

b) If σ and τ are two sequences, and $i < lh(\sigma)$, $i < lh(\tau)$ and $\sigma(i) \neq \tau(i)$, we say that σ and τ are *incompatible*.

c) $f : D \to D$ is *monotone* if $f(\sigma)$ is a proper subsequence of $f(\tau)$ whenever σ is a proper subsequence of τ.

d) *A tree* is a monotone function $T : D \to D$ mapping incompatible sequences to incompatible sequences.

e) If S and T are trees, then S is a *subtree of* T if there is a tree T' such that $S = T \circ T'$.

f) If T is a tree and $f : \mathbb{N} \to \{0, 1\}$, then f *is a branch in* T if for every n there is a sequence σ of length n such that $T(\sigma)$ is an initial segment of f.

Remark 1.4.2 These trees will sometimes be called *perfect trees*, the set of branches will form a perfect subset of the set $\{0, 1\}^{\mathbb{N}}$ in the topological sense, i.e. a set that is closed and without isolated points.
Our intuition should be focused on the set $Set(T)$ of sequences that are initial segments of the possible $T(\sigma)$'s. This will be a binary tree in the traditional

sense, and we will have the same set of infinite branches. If S is a subtree of T, then $Set(S) \subseteq Set(T)$. The converse will not hold in general.

Lemma 1.4.3 *Let $\{T_n\}_{n\in\mathbb{N}}$ be a sequence of trees such that T_{n+1} is a subtree of T_n for all n. Then there is a function f that is a branch in all trees T_n.*

Proof
Consider the set X of σ such that $\sigma \in Set(T_n)$ for all n.
X will be a binary tree. The empty sequence is in X. If $\sigma \in X$, then for all n, $\sigma * 0 \in Set(T_n)$ or $\sigma * 1 \in Set(T_n)$. At least one of these has to hold for infinitely many n and, since we are dealing with subtrees, for all n. Thus X is not a finite tree and by König's Lemma has an infinite branch, which will be a common branch for all T_n's.

Remark 1.4.4 Using topology we might just say that the intersection of a decreasing sequence of nonempty compact sets is nonempty, in order to prove this lemma.

We will now see that certain computable trees can be used to meet natural requirements. As a first case, let us prove:

Lemma 1.4.5 *Let T be a computable tree and assume that ϕ_e is total. Then there is a computable subtree S of T such that ϕ_e is not a branch in S.*

Proof
If ϕ_e is not a branch in T, we can use $S = T$. If ϕ_e is a branch in T, one of $T(0)$ and $T(1)$ will be incompatible with ϕ_e, since they are incompatible themselves. (here 0 is the sequence of length 1 with entry 0). Assume that ϕ_e is incompatible with $T(0)$.
Let $S(\sigma) = T(0*\sigma)$. Then S is a subtree as desired. The other case is essentially the same.

1.4.2 Collecting Trees

Now we will describe a property on computable trees that will ensure that if f is a branch in the tree and ϕ_e^f is total, then ϕ_e^f is computable.

Definition 1.4.6 Let T be a computable tree, e an index.
T is *e-collecting* if for all finite sequences σ, τ and all $x \in \mathbb{N}$, if $\phi_e^{T(\sigma)}(x)\downarrow$ and $\phi_e^{T(\tau)}(x)\downarrow$, then

$$\phi_e^{T(\sigma)}(x) = \phi_e^{T(\tau)}(x).$$

Lemma 1.4.7 *Let T be a computable e-collecting tree and let f be a branch in T. If ϕ_e^f is total, then ϕ_e^f is computable.*

Proof
We will give an algorithm for computing $\phi_e^f(x)$ from x.

Since $\phi_e^f(x)\downarrow$, there will be a 0-1-sequence σ such that $\phi_e^{T(\sigma)}(x)\downarrow$, and since T is e-collecting, the value $\phi_e^{T(\sigma)}(x)$ will be independent of the choice of σ. Thus our algorithm will be:
Search for a finite sequence σ such that $\phi_e^{T(\sigma)}(x)\downarrow$ and let the answer be the output of our algorithm.

Remark 1.4.8 There is more information to be gained from this proof. We see that the function ϕ_e^f itself is independent of f as long as it is total and f is a branch in an e-collecting tree.

1.4.3 Splitting Trees

We will now find a criterion that will ensure that f is computable in ϕ_e^f whenever f is a branch in a computable tree and ϕ_e^f is total:

Definition 1.4.9 Let T be a computable tree, and let e be an index.
We call T *e-splitting* if for all finite 0-1-sequences σ and τ, if σ and τ are incompatible, then there exists a number x such that

$$\phi_e^{T(\sigma)}(x)\downarrow \text{ and } \phi_e^{T(\tau)}(x)\downarrow \text{ with } \phi_e^{T(\sigma)}(x) \neq \phi_e^{T(\tau)}(x).$$

Lemma 1.4.10 *Let T be an e-splitting computable tree , let f be a branch in T and assume that ϕ_e^f is total. Then f is computable in ϕ_e^f.*

Proof
We will compute an infinite 0-1-sequence $\{k_i\}_{i\in\mathbb{N}}$ from ϕ_e^f such that $T(\sigma_n)$ is an initial segment of f for all n, where $\sigma_n = (k_0, \ldots, k_{n-1})$. The empty sequence σ_0 of course satisfies this. Assume that σ_n is constructed. Then one of $T(\sigma_n * 0)$ and $T(\sigma_n * 1)$ will be an initial segment of f. We will just have to determine which one. Now $\phi_e^{T(\sigma_n*1)}$ and $\phi_e^{T(\sigma_n*1}$) will be incompatible, so exactly one of them will be incompatible with ϕ_e^f. We can then use ϕ_e^f to find the incompatible one, which means that we can decide which direction is along f and which is not. This provides us with the induction step, and we can move on. This ends the proof of the lemma.

Remark 1.4.11 In the case of T being an e-splitting tree, we see that ϕ_e^f is a one-to one-function of the branch f, and what we just argued for is that we can compute the inverse.

1.4.4 A minimal degree

Using Lemmas 1.4.3, 1.4.5, 1.4.7 and 1.4.10 we can show the existence of a minimal degree from the following:

Lemma 1.4.12 *Let T be a computable tree and let e be an index. Then there is a computable subtree S that is either e-collecting or e-splitting.*

Proof
Case 1: There is a sequence σ such that for all τ and τ' extending σ, $\phi_e^{T(\tau)}$ and $\phi_e^{T(\tau')}$ are equal where both are defined, i.e. $\phi_e^{T(\tau)}$ and $\phi_e^{T(\tau')}$ are compatible.. Let $S(\tau) = T(\sigma * \tau)$. Then S will be a subtree of T and S will be e-collecting.
Case 2: Otherwise. Then for every σ there will be extensions τ_0 and τ_1 such that $\phi_e^{T(\tau_0)}$ and $\phi_e^{T(\tau_1)}$ are incompatible. Further, using the selection theorem,we can find these τ_0 and τ_1 as computable functions $t_0(\sigma)$ and $t_1(\sigma)$.
We then define the subtree S by

$S(()) = T(())$, i.e. S and T are equal on the empty sequence.

If $S(\sigma)$ is defined, let $S(\sigma * 0) = T(t_0(\sigma))$ and $S(\sigma * 1) = T(t_1(\sigma))$.

This defines a subtree that will be e-splitting.

We have now proved all essential steps needed in the construction of a minimal degree:

Theorem 1.4.13 *There is a non-computable function f such that if $g <_T f$ then either g is computable or g is equivalent to f.*

Proof
Using the lemmas above we construct a family $\{T_n\}$ of computable trees such that T_{n+1} is a subtree of T_n for all n, and such that for all e:

If ϕ_e is total, then ϕ_e is not a branch in T_{2e+1}.

T_{2e+2} is either e-collecting or e-splitting.

Then by Lemma 1.4.3 there is an f that is a branch in all T_n's, and this f will have the property wanted.

1.5 A priority argument

1.5.1 C.e. degrees

In the constructions of minimal pairs and functions of minimal degrees we have not been concerned with the complexity of the sets and functions constructed. We can decide upper bounds on the complexity of the objects constructed in the proofs by analyzing the complexity of properties like 'ϕ_e is total' and counting in depth how many number quantifiers we will need in order to write out a definition of the object constructed. If we, however, are interested in results about degrees with some bounded complexity, we must be more careful in our constructions. In this section we will be interested in degrees with at least one c.e. set in it:

Definition 1.5.1 Let **a** be a degree.
a is an *c.e. degree* if **a** contains the characteristic function of a c.e. set. We say that f is *of c.e. degree* if the degree of f is a c.e. degree.

There is a nice characterization of the functions of c.e. degree. We leave the proof as an exercise for the reader, see Exercise 1.29.

Theorem 1.5.2 *Let f be a function. Then the following are equivalent:*

i) f is of c.e. degree.

ii) There is a primitive recursive sequence $\{f_i\}_{i\in\mathbb{N}}$ converging pointwise to f such that the following function

$$g(x) = \mu n.\forall m \geq n(f_m(x) = f(x))$$

is computable in f.

1.5.2 Post's Problem

So far we only know two c.e. degrees, $\mathbf{O}$, the degree of the computable sets and functions, and $\mathbf{O}'$, the degree of the halting problem or of $\mathcal{K}$. Post's Problem asks if there are more c.e. degrees than those two. This is of course a nice, technical problem, but it has implications beyond that. One of the reasons why c.e. sets are so interesting is that the set of theorems in an axiomatizable theory is c.e. If there were no more c.e. degrees than those two known to us, a consequence would be that there are two kinds of axiomatizable theories, those that are decidable and those that share the complexity of Peano Arithmetic. As a consequence of Gödel's proof of the incompleteness theorem, the set of Gödel-numbers of theorems in Peano Arithmetic is a complete c.e. set, and actually of even the same m-degree as $\mathcal{K}$.

Now, in 1957 two young mathematicians, Friedberg and Muchnic, independently constructed c.e. sets of in-between degrees. They both developed what is now known as the priority method. The problem we have to face when constructing c.e. sets is that we must give an algorithm for adding elements to the set, but we cannot give an algorithm for keeping objects out of the set. If we did that, the set constructed would become computable. Thus when we have made an attempt to approximate a set with positive and negative information, we must be allowed to violate the negative information. However, we must not violate the negative information to such an extent that we ruin our global goal. We solve this dilemma by introducing priorities to our requirements. If an effort to satisfy one requirement will ruin the attempt to satisfy another requirement, we let the requirement with highest priority win. This idea will work when two properties are satisfied by the construction: If we make an attempt to satisfy a requirement and we never ruin this attempt, we actually manage to satisfy the requirement. Further, if we after a stage in the construction never make an attempt to satisfy a requirement, the requirement will automatically be satisfied.

Thus we are bound to satisfy the requirement of highest priority, either because we make an attempt which will not be ruined, or because there is no need to make an attempt.

Then we are bound to satisfy the next requirement, either because we make an attempt after the final attempt for the first requirement, or because there is no need to make such an attempt,
and so on.... In the finite injury lemma we will give a full proof along this line of arguing.

1.5.3 Two incomparable c.e. degrees

Post's problem was solved by constructing two incomparable c.e. degrees **a** and **b**. We will see below why this actually solves the original problem.

Theorem 1.5.3 *There are two c.e. sets A and B that are not computable relative to each other.*

Remark 1.5.4 If A and B are not computable in each other, neither can be computable, because any computable set will be computable in any other set. Moreover neither can have the same degree as $\mathcal{K}$, because every c.e. set is computable in $\mathcal{K}$. Thus we have not just produced an in-between degree, but two in-between degrees. In Exercise 1.30 we will see that there are infinitely many in-between c.e. degrees, and that any countable partial ordering can be embedded into the ordering of the c.e. degrees.

Proof
We will construct two c.e. sets A and B satisfying

R_{2e}**:** $\quad \mathbb{N} \setminus A \neq W_e^B$

R_{2e+1}**:** $\mathbb{N} \setminus B \neq W_e^A$

or in other terms: The complement of A is not c.e. relative to B and vice versa. If we achieve this for all e, we will have proved the theorem, since a set A is computable in B if and only if both A and the complement of A are c.e. in B, and since A is c.e. we have that A is computable in B if and only if the complement of A is c.e. in B.

We will construct two primitive recursive increasing sequences $\{A_n\}_{n\in\mathbb{N}}$ and $\{B_n\}_{n\in\mathbb{N}}$ of finite sets . We let

$$A_0 = B_0 = \emptyset.$$

We call each step in the process a *stage*. If $n = \langle 1, e, x\rangle$ we will consider to make an attempt to satisfy R_{2e} at stage n, while if $n = \langle 2, e, x\rangle$ we will consider to make an attempt to satisfy R_{2e+1}.
An attempt to satisfy R_{2e} will consist of selecting a $q \in A_{n+1} \cap W_e^{B_{n+1}}$, and then put up a *protection*, the set of points used negatively in the verification of $q \in W_e^{B_{n+1}}$. If we can keep all objects in this protection out of B throughout the construction, we will have

$$q \in A \cap W_e^B$$

and R_{2e} will be satisfied. We call the protection *active* at a later stage m if B_m is disjoint from this protection.

There is some little minor trick to observe, for different e's we will use disjoint infinite supplies of numbers that we may put into A (or B) in order to satisfy R_{2e} (or R_{2e+1}). We will use this to show that if we make only finitely many attempts, we will succeed after all.

Now let $n = \langle 1, e, x\rangle$ and assume that A_n and B_n are constructed. Assume further that we constructed certain protections, some of them active at stage n, others not. We write the following procedure for what to do next:
Let $B_{n+1} = B_n$.
Question 1: Is there a protection for R_{2e} active at stage n?
If the answer is 'yes', let $A_{n+1} = A_n$. and continue to the next stage.
If the answer is 'no', ask
Question 2: Is there a $y < n$ such that $\phi_{e,n}^{B_n}(\langle y, e\rangle)\downarrow$ and y is in no active protection for any requirement R_{2d+1} where $2d+1 < 2e$?
If the answer is 'no', let $A_{n+1} = A_n$ and proceed to the next stage.
If the answer is 'yes', choose the least y, let $A_{n+1} = A_n \cup \{\langle y, e\rangle\}$, construct a protection $\{0, \ldots, n\} \setminus B_n$ for R_{2e} and move on to the next stage.

If $n = \langle 2, e, x\rangle$ we act in the symmetric way, while for other n we just move on to the next stage, not adding anything to A or B.
This ends the construction.

Claim 1 *(The Finite Injury Lemma)*
For each requirement R_s there is a stage n_s after which we do not put up or injure any protection for R_s.
Proof
We prove this by induction on s, and as an induction hypothesis we may assume that there is a stage m_s after which we never put up a protection for any requirement R_t with $t < s$. Then after stage m_s we will never injure a protection for R_s. Thus if we never put up a protection for R_s after stage m_s we can let $n_s = m_s$, while if we construct a protection, this will never be injured and we can let n_s be the stage where this protection is constructed.

Now let $A = \bigcup_{n\in\mathbb{N}} A_n$ and $B = \bigcup_{n\in\mathbb{N}} B_n$. Then A and B are c.e. sets.

Claim 2
Each requirement R_s will be satisfied.
Proof
We prove this for $s = 2e$. There are two cases.
1. There is a protection for R_s active at stage n_s.
If this is the case, there will be a y such that $\langle y, e\rangle \in A_{n_s} \cap W_e^{B_n}$. Since the protection is not injured, we will have that $\langle y, e\rangle \in A \cap W_e^B$ and the requirement is satisfied, A and W_e^B are not complementary.
2. There is no such protection.
There are only finitely many objects of the form $\langle y, e\rangle$ in A, because we ad at most one such object for each stage before n_s, and never any at a stage after n_s.

On the other hand, there can be only finitely many objects of the form $\langle y, s\rangle$ in W_e^B, since otherwise we could choose one that is not in any protection for any R_t for $t < s$, and sooner or later we would at some stage after n_s make a new attempt to satisfy R_e, which we are not. Thus $A \cup W_e^B$ contains only finitely many objects of the form $\langle y, e\rangle$ and the sets are not the complements of each other. Thus the requirement will be satisfied in this case as well.

We have shown that all the requirements are satisfied in this construction, so the theorem is proved.

Remark 1.5.5 In Exercise 1.31, we prove the so called *splitting theorem*. The proof of the splitting theorem is by a more elaborated priority argument than the ones we have seen so far. Soare [7] is recommended for readers who would like to see priority arguments in their full power.

1.6 Models for second order number theory

One possible application of degree theory is the investigation of the relative strength of formal theories. We will prove one theorem, showing that **WKL** actually is a rather weak axiom of set existence.
In order to make this precise, we have to discuss what we mean by fragments of second order number theory and models of such fragments. Since this is not a course on logic, we will be brief.

We extend the language of number theory with variables $X_{1,n}, X_{2,n}, \ldots$ for n-ary relations on $\mathbb{N}$. In this language, we may typically express the general induction axiom

$$\forall X(0 \in X \wedge \forall x(x \in X \to x + 1 \in X) \to \forall x(x \in X)),$$

where we drop the index of the unary variable X.
A fragment of second order number theory will then be a set of axioms in this language, containing the axioms of number theory without induction, some induction axioms and some set existence axioms.
A formula in this language is *arithmetical* if all quantifiers are number quantifiers. The scheme of *arithmetical comprehension* is the axiom scheme

$$\exists X \forall \vec{x}(\vec{x} \in X \leftrightarrow \Phi(\vec{x}, \vec{y}, \vec{Y}))$$

where Φ is arithmetical. Weak König's Lemma, **WKL** is another axiom of set existence.

It is customary to assume that we have Δ_0-comprehension, i.e. the comprehension axiom for Φ as above, when Φ only contains bounded quantifiers. Then we have the expressive power to code any n-ary relation as a unary relation, and identify a set with its characteristic function. This means that we may consider a set $\mathcal{M}$ of unary functions with some basic closure properties as models of such fragments. We will use this as a definition:

Definition 1.6.1 By a *second order structure* we will mean a set $\mathcal{M} \subset \mathbb{N}^{\mathbb{N}}$ such that whenever $f_1, \ldots, f_n$ are in $\mathcal{M}$ and g is computable relative to $f_1, \ldots, f_n$, then $g \in \mathcal{M}$.

It is easy to see that any model satisfying arithmetical comprehension will also satisfy **WKL**. We will use degree theory to prove a theorem due to C. Jockusch and R.I. Soare, the converse is not true. In order to do so, we need to introduce a new concept:

Definition 1.6.2 Let **a** be a Turing degree.
We say that **a** is *low* if $\mathbf{a}' = \mathbf{O}'$.

We will indirectly prove that there are non-computable low degrees.

Theorem 1.6.3 *Let T be a computable, infinite binary tree.*
Then T has an infinite branch f of low degree.

Proof
We will construct a decreasing sequence $\{T_e\}$ of infinite, computable binary trees, and we will let f be a joint branch of all these trees.
Recall that $lh(\sigma)$ denotes the length of the sequence σ and that $\phi_{e,m}^{\sigma}\uparrow$ if there is no computation tree for $\phi_e)^{\sigma}$ with a number code below m, and that this is decidable.
Let $T_0 = T$ and assume for an arbitrary e that T_e is constructed.
Let

$$U_e = \{\sigma \in T_e; \phi_{e,lh(\sigma)}^{\sigma}\uparrow\}.$$

Then U_e is a computable tree.
If U_e is infinite, we let $T_{e+1} = U_e$, while if U_e is finite, we let $T_{e+1} = T_e$.

First, let us see that the construction is computable relative to $\mathcal{K}$. The construction is by recursion, and the critical question at each step is if U_e is finite or not. Since it is semi-decidable wether a computable binary tree is finite or not, we can use a $\mathcal{K}$-oracle to answer these questions.

Now, let f be a branch in all T_e's. We claim that

$$\phi_e^f(e)\uparrow \Leftrightarrow U_e \text{ is infinite.}$$

Since f is a branch in T_e, if $\phi_e^f(e)\uparrow$, the all initial segments of f will be in U_e, which then is infinite. On the other hand, if U_e is infinite, then $T_{e+1} = T_e$, and by the finite use principle and the assumption that f is a branch in T_{e+1} it follows that $\phi_e^f(e)\downarrow$.

Since we could compute U_e from $\mathcal{K}$ and e and decide if U_e is finite or not relative to $\mathcal{K}$, we have shown that

$$\mathcal{K}^f <_T \mathcal{K}.$$

for any f that is a branch in all T_e (and there will be exactly one such branch). This ends the proof of the theorem.

We may use this theorem to construct a second order model satisfying **WKL** only containing functions of low degree. The key observation is that the argument above can be reformulated to a proof of

Corollary 1.6.4 *Let T be an infinite binary tree of low degree. Then T has an infinite branch f such that $T \oplus f$ is of low degree.*

By this corollary, we may use brute force to construct an increasing sequence of low degrees $\mathbf{a}_n$ such that for all binary, infinite trees T, if the degree of T is below one $\mathbf{a}_n$, then T has an infinite branch of a degree below some $\mathbf{a_m}$. If we then consider all functions bounded by one of these degrees, we have a model as desired. This model cannot contain the characteristic function of $\mathcal{K}$, since this degree is not low. However, given arithmetical comprehension we can define $\mathcal{K}$, so this model cannot satisfy arithmetical comprehension.

1.7 Subrecursion theory

1.7.1 Complexity

What is to be considered as complex will be a matter of taste. Actually, a logician may alter her/his taste for complexity several times a day. The generic logician may give a class on automata theory in the morning, and then the regular languages will be the simple ones, while context free languages are more complex. Still, context free languages are decidable in polynomial time, and while our logician spends an hour contemplating on the $\mathbf{P} = \mathbf{NP}$-problem any context free language is by far simpler than the satisfiability problem for propositional logic. If our logician is working mainly in classical computability theory, all decidable languages are simple, while the undecidable ones are the complex ones. When, however, our logician gives a course on set theory with large cardinal axioms in the afternoon, all definable sets are simple, even all subsets of subsets of subsets of $\mathbb{R}$ are simple. Considering large cardinal axioms, we have to move far in order to find sets of a challenging and interesting complexity.
In this section, our view on complexity will be one shared by many proof theorists. One of the aims in proof theory is to characterize the functions and sets provably computable in certain formal theories extending elementary number theory. The idea is that if we know the functions provably computable in T, we know something worth knowing about the strength of the theory T.

We will not be concerned with proof theory in this compendium, and any references to proof-theoretical results should not be considered as a part of any curriculum based on this text.

1.7.2 Ackermann revisited

In Section 1.2.1 we defined the Ackermann branches. The idea of Ackermann was that each use of the scheme for primitive recursion involves an *iteration* of a previously defined function. Then diagonalising over a sequence of functions defined by iterated iteration would break out of the class of primitive recursive functions.

The Ackermann branches were defined using functions of two variables. We will be interested in pushing his construction through to the transfinite level,

in order to describe more complex functions from below. Then it will be convenient, from a notational point of view, to use functions of one variable.

Definition 1.7.1

- Let $F_0(x) = x + 1$
- Let $F_{k+1}(x) = F_k^{x+2}(x)$

In Exercise 1.33 we will see that diagonalising over the F_k's will lead us outside the class of primitive recursive functions.

From now on in this section we will let PA be first order Peano arithmetic; i.e. elementary number theory with the axiom scheme for first order induction. Our first order language L will be the language of PA.

Definition 1.7.2 A function $f : \mathbb{N} \to \mathbb{N}$ is *provably computable* if there is a formula $A(x, y) = \exists z B(x, y, z)$ in L where A defines the graph of f, B has only bounded quantifiers and

$$PA \vdash \forall x \exists y A(x, y).$$

This definition is extended in the obvious way to functions of several variables.

We will assume that the reader is familiar with the method of arithmetisation, sequence numbering and so forth.

Lemma 1.7.3 *Let $f(k, x) = F_k(x)$. Then f is provably computable.*

Outline of proof
Essentially we have to prove that F_0 is total and that if F_k is total then F_{k+1} is total as well. This induction step is proved by induction, where we actually must prove

$$\forall x \forall y \exists z F_k^y(x) = z$$

by induction on y.
In order to write a complete proof, we have to write down the formal definition of the graph of F and the establish a proof tree for the formula required to be a theorem in PA. This is tedious and not very exiting; the exiting part is to decide how much induction that is required.

1.7.3 Ordinal notation

The concept of *an ordinal number* will be properly introduced in a course on set theory. A *well ordering* is a total ordering $(X, \prec)$ such that each nonempty subset $Y \subseteq X$ will have a least element. An ordinal number will be a set that in a canonical way represents an isomorphism class of well orderings. For our purposes it will be sufficient to think about order types of well orderings, and we will use a term language for such order types. Exercise 3.5 is a self-service introduction to ordinal numbers.

Definition 1.7.4 ω is the smallest infinite ordinal number, and it denotes the order-type of $\mathbb{N}$ with the standard ordering.

All natural numbers will be ordinal numbers as well.
We may extend the arithmetical operations *plus*, *times* and *exponentiation* to operations on orderings, or actually, on order types. The sum of two orderings $\langle A, <_A \rangle$ and $\langle B, <_B \rangle$ will be the set

$$C = A \oplus B = (\{0\} \times A) \cup (\{1\} \times B)$$

with the ordering $<_C$ defined by

- $\langle 0, a \rangle <_C \langle 1, b \rangle$ whenever $a \in A$ and $b \in B$.
- $\langle 0, a_1 \rangle <_C \langle 0, a_2 \rangle$ if and only if $a_1 <_A a_2$.
- $\langle 1, b_1 \rangle <_C \langle 1, b_2 \rangle$ if and only if $b_1 <_B b_2$.

The product of two orderings will for our purposes be the anti-lexicographical ordering on the product of the domains.

The formal definition of the exponential of orderings is less intuitive, but it helps to think of exponentiation as iterated multiplication. Our definition only works for special orderings $\langle A, <_A \rangle$, including all well orderings.

Definition 1.7.5 Let $\langle A, <_A \rangle$ and $\langle B, <_B \rangle$ be two orderings where A has a least element a_0.
Let $p \in C$ if p is a map from B to A such that $p(b) = a_0$ for all but finitely many $b \in B$.
If $p \neq q$ are in C, there will be a maximal argument b such that $p(b) \neq q(b)$.
We then let $p <_C q \Leftrightarrow p(b) < q(b)$ for this maximal b.

Lemma 1.7.6 *If $\langle A, <_A \rangle$ and $\langle B, <_B \rangle$ are two well orderings then the sum, product and exponential will be well orderings.*

The proof is left as Exercise 1.34

As a consequence, every arithmetical expression in the constant ω and constants for the natural numbers, and using 'plus', 'times' and 'exponents', will have an interpretation as an ordinal number.
The least ordinal number that cannot be described by an expression as above is baptized ϵ_0. As for ordinary arithmetics, $\omega^0 = 1$. This is a special case of the more general rule

$$\omega^\alpha \cdot \omega^\beta = \omega^{\alpha+\beta}.$$

A consequence is that each ordinal $\alpha < \epsilon_0$ can be written in a unique way as

$$\alpha = \omega^{\alpha_n} + \cdots + \omega^{\alpha_0}$$

where $\{\alpha_0, \cdots, \alpha_n\}$ is an increasing (not necessarily strictly) sequence of ordinals less than α. We call this the *Cantor Normal Form* of α. If $\alpha_0 = 0$ the

ordinal α will be a *successor ordinal*, otherwise it will be a *limit ordinal*.

We extended the Ackermann hierarchy to the first transfinite level by diagonalising over the Ackermann branches. One advantage with the Cantor normal form is that we may find a canonical increasing unbounded sequence below every limit ordinal between ω and ϵ_0. Actually, it is possible to do so for ordinals greater than ϵ_0 too, but readers interested in how this is done and why someone would like to do it are recommended to follow a special course on proof theory and ordinal denotations.

Definition 1.7.7 Let

$$\alpha = \omega^{\alpha_m} + \cdots + \omega^{\alpha_0}$$

be given on Cantor normal form, and assume that $\alpha_0 > 0$.
We define the n'th element $\alpha[n]$ of the fundamental sequence for α as follows:

Case 1 $\alpha_0 = \beta + 1$:
Let $\alpha[n] = \omega^{\alpha_m} + \cdots + \omega^{\alpha_1} + n \cdot \omega^{\beta}$.

Case 2 α_0 is a limit ordinal:
We may then assume that $\alpha_0[n]$ is defined, and we let

$$\alpha[n] = \omega^{\alpha_m} + \cdots + \omega^{\alpha_1} + \omega^{\alpha_0[n]}$$

We may consider the map $\alpha \mapsto \alpha[n]$ as an extension of the predecessor function to limit ordinals:

Definition 1.7.8 Let $0 < \alpha < \epsilon_0$. We define the *n-predecessor of α* by

a) If α is a successor ordinal, the predecessor of α will be the n-predecessor of α.

b) If α is a limit ordinal, $\alpha[n]$ will be the n-predecessor of α.

c) We say that $\beta <_n \alpha$ if β can be reached from α by iterating the n-predecessor map.

Lemma 1.7.9 *Let $\alpha < \epsilon_0$, $\beta <_m \alpha$ and $m < n$. Then $\beta <_n \alpha$.*

Proof
It is sufficient to prove that if $m < n$ and α is a limit ordinal, then $\alpha[m] <_n \alpha$. This is left as a non-trivial exercise for the reader, see Exercise 3.3.

Lemma 1.7.10 *Let $\alpha < \epsilon_0$ and let $\beta < \alpha$. Then there is an n such that $\beta <_n \alpha$.*

Proof
This is proved by induction on α with the aid of Lemma 1.7.9. The details are left as a nontrivial exercise for the reader, see Exercise 3.3

1.7.4 A subrecursive hierarchy

We will now extend the alternative Ackermann hierarchy to all ordinals less than ϵ_0:

Definition 1.7.11 Let $\alpha < \epsilon_0$. We define $F_\alpha(x)$ by recursion on α as follows:

- $F_0(x) = x + 1$
- $F_{\beta+1}(x) = F_\beta^{x+2}(x)$
- $F_\alpha(x) = F_{\alpha[x]}(x)$ when α is a limit ordinal.

Proof theorists have shown that if a function $f : \mathbb{N} \to \mathbb{N}$ is provably computable, then f will be bounded almost everywhere by one of the F_α's where $\alpha < \epsilon_0$. The proof involves the translation of a proof in PA to a proof in ω-logic, then a cut-elimination procedure in ω-logic and finally an analysis of cut-free proofs in ω-logic of the totality of computable functions. This analysis is far beyond the scope of this compendium, and the reader is *not* recommended to work out the details.
In Exercises 3.3 and 3.4, the reader is challenged to prove that each F_α is provably computable and establish the hierarchical properties of the $\{F_\alpha\}_{\alpha<\epsilon_0}$-hierarchy. The main results are:

Lemma 1.7.12 *Let $n < x$ and $\beta <_n \alpha < \epsilon_0$.*
Then

$$F_\beta(x) \leq F_\alpha(x).$$

Theorem 1.7.13 *If $\beta < \alpha < \epsilon_0$, then*

$$\exists x_0 \forall x > x_0 (F_\beta(x) < F_\alpha(x)).$$

Remark 1.7.14 There are many aspects about subrecursive hierarchies that we have not discussed in this section. We have not discussed complexity classes. For instance, the class H_α of functions computable in polynomial time relative to a finite iteration of F_α represents a stratification of the set of all provably computable functions into a complexity hierarchy, where each complexity class is closed under composition and closed under polynomial time reductions. We will not discuss such matters further here.

1.8 Exercises

Exercise 1.1 Prove that the following functions are primitive recursive:

a) $f(x, y, \vec{z}) = x + y$

b) $f(x, y, \vec{z}) = x \cdot y$

c) $f(x, y, \vec{z}) = x^y$

d) $f(x, \vec{y}) = x!$

e) $f(x, \vec{y}) = x \dot{-} 1$

f) $f(x, y, \vec{z}) = x \dot{-} y$

g) $f(x, y, \vec{z}) = 0$ if $x = y$

$f(x, y, \vec{z}) = 1$ if $x \neq y$

h) $f(x, y, \vec{z}) = 0$ if $x < y$

$f(x, y, \vec{z}) = 1$ if $x \geq y$

Exercise 1.2 Prove the following facts:

a) $\mathbb{N}^n$ and $\emptyset$ are primitive recursive as subsets og $\mathbb{N}^n$.

b) The complement of a primitive recursive set is primitive recursive. Moreover, the union and intersection of two primitive recursive subsets of $\mathbb{N}^n$ will be primitive recursive.

Exercise 1.3 a) Prove Lemma 1.2.6.

b) Prove that we can replace the inequalities by strict inequalities in Lemma 1.2.6.

Exercise 1.4 Prove Lemma 1.2.7.

Exercise 1.5 a) Prove Lemma 1.2.11.

b) Prove that the sequence numbering is monotone in each coordinate.

c) Prove that the monotone enumeration SEQ of the sequence numbers is primitive recursive.
Hint: Find a primitive recursive bound for the next sequence number and use bounded search.

d) Define an alternative sequence numbering as follows:
$\langle\langle x_0, \ldots, x_{n-1}\rangle\rangle$ is the number z such that

$$SEQ(z) = \langle x_0, \ldots, x_{n-1}\rangle.$$

Show that this alternative numbering is surjective and still satisfies Lemma 1.2.11

e) Prove that the pairing function P in Definition 1.2.13 is 1-1 and onto.

Exercise 1.6 Let X be a set of functions closed under the schemes of primitive recursion.
Show that for any function $f : \mathbb{N} \to \mathbb{N}$ we have

$$f \in X \Leftrightarrow \bar{f} \in X.$$

Exercise 1.7 We define the function $f(k, e, y)$ by recursion on k and subrecursion on e as follows:

1. For all k, if $e = \langle 1 \rangle$ and $y = \langle x, x_1, \ldots, x_n \rangle$ we let
$$f(k, e, y) = x + 1.$$
2. For all k, if $e = \langle 2, i \rangle$, $y = \langle x_1, \ldots, x_n \rangle$ and $1 \leq i \leq n$ we let
$$f(k, e, y) = x_i.$$
3. For all k, if $e = \langle 3, q \rangle$ we let
$$f(k, e, y) = q.$$
4. For all k, if $e = \langle 4, e', d_1, \ldots, d_n \rangle$ we let
$$f(k, e, y) = f(k, e', \langle f(k, d_1, y), \ldots, f(k, d_n, y) \rangle).$$
5. For all $k > 0$, if $e = \langle 5, e_1, e_2 \rangle$ we let
   * $f(k, e, \langle 0, x_1, \ldots, x_n \rangle) = f(k - 1, e_1, \langle x_1, \ldots, x_n \rangle)$
   ** $f(k, e, \langle m + 1, x_1, \ldots, x_n \rangle) =$
   $f(k - 1, e_2, \langle f(k, e, \langle m, x_0, \ldots, x_n \rangle, m, x_1, \ldots, x_n \rangle)$
6. In all other cases, we let $f(k, e, y) = 0$.

a) Prove that f is well defined, and that f is computable.

b) Prove that if g of arity n is primitive recursive, there is a number k and an index e such that
$$g(x_1, \ldots, x_n) = f(k, e, \langle x_1, \ldots, x_n \rangle)$$
for all $x_1, \ldots, x_n \in \mathbb{N}^n$.

c) Prove that f is not primitive recursive.

Exercise 1.8 Prove that if $\phi_e(\vec{x}) = y$ and $\phi_e(\vec{x}) = z$, then $y = z$.
Hint: Use induction on e.
Discuss why this is something that needs a proof.

Exercise 1.9 Let $\{f_n\}_{n \in \mathbb{N}}$ be a sequence of total functions such that
$$g(n, m) = f_n(m)$$
is computable.
Show that each f_n is computable, and that there is a total computable function not in the sequence.
Hint: Use a diagonal argument.

Exercise 1.10 Show that there is a total computable function $\Phi(n, x)$ of two variables that enumerates all primitive recursive functions of one variable.
Is it possible to let Φ be primitive recursive?

Exercise 1.11 Complete the proof of Lemma 1.2.27.

Exercise 1.12 Prove Corollary 1.2.32.

Exercise 1.13 a) Prove that every non-empty c.e. set is the image of a primitive recursive function (easy) and that every infinite c.e. set is the image of an injective computable function (not that easy, but still...).

b) Prove that the range of a strictly increasing total computable function is computable.

Exercise 1.14 Let $A \subseteq \mathbb{N}^n$. Show that A is c.e. (by characterization ii) or iii) in Theorem 1.2.35) if and only if

$$\{\langle x_1, \ldots, x_n \rangle \mid (x_1, \ldots, x_n) \in A\}$$

is c.e.

Exercise 1.15 Give an explicit description of the selection function in the proof of Theorem 1.2.40.

Exercise 1.16 Let A and B be two disjoint sets whose complements are c.e. Show that there is a computable set C such that $A \subseteq C$ and $B \cap C = \emptyset$.
Hint: Use Corollary 1.2.41.

Exercise 1.17 Let L be the language of propositional calculus over an infinite set $\{A_i\}_{i \in \mathbb{N}}$ of propositional variables. Discuss the following statement:
There is a primitive recursive consistent set of propositions with no computable completion.

Exercise 1.18 a) Show that there is an enumeration $\{I_n\}_{i \in \mathbb{N}}$ of all closed rational intervals contained in $[0, 1]$ such that the relations $I_n \subseteq I_m$, $I_n \cap I_m = \emptyset$ and $|I_n| < 2^{-m}$ are computable, where $|I_n|$ is the length of the interval.

A real number x is *computable* if there is a computable function h such that

i) $|I_{h(n)}| < 2^{-n}$

ii) For all n, $x \in I_{h(n)}$

b) Let $f : [0, 1] \to [0, 1]$ be a continuous function.
We say that f is *computable* if there is a total computable function $g : \mathbb{N} \to \mathbb{N}$ such that

i) $I_n \subseteq I_m \Rightarrow I_{g(n)} \subseteq I_{g(m)}$

ii) $x \in I_n \Rightarrow f(x) \in I_{g(n)}$

iii) For all x and m there is an n with $x \in I_n$ and $|I_{g(n)}| < 2^{-m}$.

Show that any computable function will be continuous. Show that there is a computable function that does not take its maximal value at any computable real.

Exercise 1.19 Show that if $A \equiv_m B$ and $C \equiv_m D$, then $A \oplus C \equiv_m B \oplus D$. Show that if $A <_m E$ and $B <_m E$ then $A \oplus B <_m E$. Show that $A \oplus B$ then represents the least upper bound of A and B in the m-degrees.

Exercise 1.20 Show that the set of codes $[\![\Phi]\!]$ for formulas in the language $\mathcal{L}$ used in the definition of tt-reducibility is primitive recursive.
Will the set of tautologies in $\mathcal{L}$ be primitive recursive?

Exercise 1.21 Prove Lemma 1.3.17.

Exercise 1.22 Show that the second recursion theorem can be extracted from the proof of the recursion theorem.
Hint: Let $F(g, x)$ be computable. Use the recursion theorem on

$$f(e, x) = F(\phi_e, x).$$

Let e_0 be the index obtained from the proof of the recursion theorem.
Show that if $\phi_{e_0}(x) = y$ and we make an oracle call for $\phi_{e_0}(z)$ in the computation of $F(\phi_{e_0}, x)$ then $\phi_{e_0}(z)$ is a subcomputation of $\phi_{e_0}(x)$.

Exercise 1.23 Show that if $f_1 \equiv_T f_2$, then the jumps f_1' and f_2' are also Turing equivalent.

Exercise 1.24 Make a direct construction and show that there are two total functions f and g such that $f \not\leq_T g$ and $g \not\leq_T f$.

Exercise 1.25 Prove that there is a minimal pair of Turing degrees, i.e. a pair $\{\mathbf{a}, \mathbf{b}\}$ of degrees of non-computable functions, such that the degree $\mathbf{O}$ of computable functions is the greatest lower bound of $\mathbf{a}$ and $\mathbf{b}$.
Hint: You may use the main idea in the proof of Theorem 1.3.33, the present theorem is just simpler to prove. You may also get some ideas from the discussion of this proof in the text.

Exercise 1.26 Let $\mathbf{O}$ be the degree of the computable functions.
Recall the definition of the jump operator in the proof of Lemma 1.3.31, cfr. Exercise 1.23. We define the arithmetical hierarchy as follows:

A Σ_1^0-set is a c.e. set (of any dimension).

A Π_1^0-set is the complement of a Σ_1^0-set

A Σ^0_{k+1}-set is the projection of a Π^0_k-set

A Π^0_{k+1}-set is the complement of a Σ^0_{k+1}-set

A Δ^0_k set is a set that is both Σ^0_k and Π^0_k.

Let $\mathbf{O}^{(n)}$ be the degree obtained from $\mathbf{O}$ using the jump-operator n times.

a) Prove that if A is Σ^0_k or A is Π^0_k, then (the characteristic function of) A has a degree $\mathbf{a} < \mathbf{O}^{(k)}$.

b) Show that for $k \geq 1$ we have that A is Δ^0_{k+1} if and only if the degree of A is bounded by $\mathbf{O}^{(k)}$.
Hint: Use the relativized version of the fact that a set is computable if and only if both the set and its complement are c.e.

Exercise 1.27 We generalize the original definitiion of a binary tree to

- A tree T is a set of finite sequences σ of natural numbers such that whenever $\sigma \in T$ and $\tau \prec \sigma$, then $\tau \in T$.
- A tree T is *finitely branching* if for all $\sigma \in T$:

  $$\{a \mid \sigma a \in T\}$$

  is finite.
- A tree T is *computably bounded* if there is a computable function f such that whenever $\sigma = (a_0, \ldots, a_{n-1}) \in T$ then $a_i \leq f(i)$ when $i < n$.

a) Prove that an infinite, computably bounded computable tree will have a branch of degree $\leq \mathbf{O}$.

b) Show that there is a computable tree that is not computably bounded.

Exercise 1.28 Show that there are continuumly many minimal degrees.
Hint: Instead of constructing one tree T_n at level n we might construct 2^n trees $T_{\sigma,n}$ for $lh(\sigma) = n$, ensuring for each e that the branches of different trees will not be computable in each other via index e. The proof requires some clever book-keeping.

Exercise 1.29 Prove Theorem 1.5.2.

Exercise 1.30 Let $B \subseteq \mathbb{N}^2$ be a set. For each $n \in \mathbb{N}$, we let $B_n = \{m \mid (n,m) \in B\}$ and we let $B_{-n} = \{(k,m) \in B \mid k \neq n\}$.

a) Show that there is a c.e. set B such that for all n, B_n is not computable in B_{-n}.
Hint: Use an enumeration of $\mathbb{N}^2$ to give all the requirements

$$R_{(n,e)}: \quad \mathbb{N} \setminus B_n \neq W_e^{B_{-n}}$$

a priority rank.

b) Consider a computable partial ordering $\prec$ on the natural numbers. Show that there is an order-preserving map of $\prec$ into the c.e. degrees. Hint: Use the construction in a), and let $C_n = \{(k,m) \in B \mid k \preceq n\}$.

There is one computable partial ordering $\prec$ such that any other partial ordering of any countable set can be embedded into $\prec$, see Exercise 3.1. Thus this shows that any countable partial ordering can be embedded into the ordering of the c.e. degrees.

Exercise 1.31 Fill in the details in the proof of the following theorem:

Theorem 1.8.1 *Let* $\mathbf{a} > \mathbf{O}$ *be an c.e. degree. Then there are two incomparable c.e. degrees* $\mathbf{b}$ *and* $\mathbf{c}$ *such that* $\mathbf{a} = \mathbf{b} \oplus \mathbf{c}$.

This theorem is called *The splitting theorem*. We split the c.e. degree $\mathbf{a}$ into two simpler c.e. degrees.
Proof
Let A be a non-computable c.e. set. It is sufficient to construct two disjoint c.e. sets B and C such that
$A = B \cup C$, A is not computable in B and A is not computable in C.
Let f be a 1-1 enumeration of A. At each stage n we will put $f(n)$ into B or $f(n)$ into C, but not into both.
Let B_n be the numbers put int B before stage n and C_n be the set of numbers put into C before stage n. Further, we let
$A_n = \{f(0), \ldots, f(n-1)\}$, so $A_n = B_n \cup C_n$.
We put up requirements

$R_{2e} \quad : K_A \neq \phi_e^B$.

$R_{2e+1} : K_A \neq \phi_e^C$.

which we give priorities in the usual way.
For each requirement R_s we define three auxiliary functions. For $s = 2e$ they will be:

The match function

$$m(s,n) = \mu k < n. \forall x \leq k(\phi_{e,n}^{B_n}(x) = K_{A_n}(x)).$$

The bar function

$$b(s,n) = max\{m(s,n') \mid n' \leq n\}.$$

The protection function
$p(s,n) = \{y \mid y$ is used negatively in computing $\phi_{e,n}^{B_n}(x)$ for some $x \leq b(s,n)\}$.
In this case we call this *a protection of* B.

Now the construction at stage n is as follows: If $f(n) \notin p(s,n)$ for any $s \leq n$, put $f(n)$ into B. Otherwise, consider the requirement R_s of highest priority such that $f(n) \in p(s,n)$. If this is a protection of B, we put $f(n)$ into C, otherwise we put $f(n)$ into B.
When we put an element into a protection of B we *injure* that requirement. We will prove that for any requirement R_s there is a stage n_s after which we will never injure that requirement, and simultaneously that the bar-function $b(s,n)$ is bounded when s is fixed.
Assume that this holds for all $s' < s$. Then there is a stage n_s after which $f(n)$ is not in the protection for any $s' < s$, and then, after stage n_s, R_s will not be injured.
This in turn means that if $x < b(s,n)$ for $n \geq n_s$ and $\phi_{e,n}^{B_n}(x)\downarrow$, then $\phi_e^B(x) = \phi_{e,n}^{B_n}(x)$.
Now, if $\lim_{n\to\infty} v(s,n) = \infty$ we can use the increasing matching and the stability of $\phi_{e,n}^{B_n}$ to show that A is computable, which it is not.
On the other hand, if $K_A = \phi_e^B$ we will get increasing matching. Thus at the same time we prove that the construction of the bar and protection for R_s terminates and that the requirement is satisfied at the end.

Exercise 1.32 Post hoped to prove that a simple set cannot be of the same degree as the complete c.e. set $\mathcal{K}$. This will not be the case, which will be clear when you have solved this problem.

Let A be a non-computable c.e. set and f a total computable 1-1-enumeration of A. We know that f cannot be increasing (why?).
Let

$$B = \{n \mid \exists m > n(f(m) < f(n))\}$$

This is called the *deficiency set* of the enumeration.

a) Show that B is c.e. and that B is computable n A.

b) Show that A is computable in B.
Hint: In order to determine if $x \in A$ it is sufficient to find $n \notin B$ such that $f(n) > x$.

c) Show that B is simple.
Hint: If the complement of B contains an infinite c.e. set, the algorithm for computing A from B in b) can be turned into an algorithm for computing A.

Exercise 1.33 Let $F(k,x) = F_k(x)$ be the alternative Ackermann function.

a) Show that $x < F(k,x)$ for all k and x.

b) Show that F is monotone in both variables.

c) Let $f : \mathbb{N}^m \to \mathbb{N}$ be primitive recursive.
For $\vec{x} = (x_1, \ldots, x_m)$, let $\sum \vec{x} = x_1 + \cdots + x_m$. Show that there is a number k such that

$$f(\vec{x}) \leq F(k, \sum \vec{x})$$

for all $\vec{x} \in \mathbb{N}^m$.
Hint: Use induction on the construction of f.
In the cases $f(x, \vec{x}) = x + 1$ and $f(\vec{x}) = x_i$ you may use $k = 0$.
In the case $f(\vec{x}) = q$, use a k such that $F(k, 0) \geq q$.
In the case of composition, you may find it convenient to increase k by more than one, while in the case of primitive recursion, you should increase k by exactly one.

d) Show that F is not primitive recursive.
Hint: Show that $G(k) = F(k, k) + 1$ cannot be primitive recursive, using c).

Exercise 1.34 Prove Lemma 1.7.6

Exercise 1.35 Let $\mathcal{P}$ be the set of polynomials $P(x)$ where we only use + (not 'minus') and where all coefficients are natural numbers.
We order $\mathcal{P}$ by

$$P(x) \prec Q(x) \Leftrightarrow \exists n \forall m \geq n (P(m) < Q(m)).$$

Show that $\prec$ is a well ordering of order-type ω^ω.

Chapter 2

Generalized Computability Theory

2.1 Computing with function arguments

When we introduced the Turing degrees, we first introduced the notion of relativized computations via the notation

$$\phi_e^{f_1,\ldots,f_n}(x_1,\ldots,x_m).$$

By a small change of notation, we may view this as a partial *functional*

$$\phi_e(x_1,\ldots,x_m,f_1,\ldots,f_n)$$

where some of the inputs may be numbers and some may be functions.

Definition 2.1.1 Let $F : \mathbb{N}^m \times (\mathbb{N}^{\mathbb{N}})^n \to \mathbb{N}$.
We say that F is *computable* if there is an index e such that for all $\vec{f} \in (\mathbb{N}^{\mathbb{N}})^n$ and $\vec{f} \in \mathbb{N}^m$ we have that

$$F(\vec{x},\vec{f}) = \phi_e(\vec{x},\vec{f}).$$

We will concentrate our attention to computable functionals $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$. Let us first see why we may expect to face all relevant theoretical problems even then.

Lemma 2.1.2 *Let $n > 0$ and $m \geq 0$. Then there is a computable bijection between $\mathbb{N}^m \times (\mathbb{N}^{\mathbb{N}})^n$ and $\mathbb{N}^{\mathbb{N}}$.*

Proof
We will produce the bijections between $(\mathbb{N}^{\mathbb{N}})^2$ and $\mathbb{N}^{\mathbb{N}}$ and between $\mathbb{N} \times \mathbb{N}^{\mathbb{N}}$ and $\mathbb{N}^{\mathbb{N}}$. The rest then follows by iterating the two constructions.
Let $\langle f, g\rangle(x) = \langle f(x), g(x)\rangle$ where we use a computable pairing function with

computable projections on $\mathbb{N}$.
Clearly $\langle f, g\rangle$ is computable from f and g in the sense that

$$F(x, f, g) = \langle f, g\rangle(x)$$

is computable. In the same sense, f and g will be computable from $\langle f, g\rangle$.
If $f \in \mathbb{N}^{\mathbb{N}}$ and $x \in \mathbb{N}$, let

$$\langle x, f\rangle(0) = x$$

$$\langle x, f\rangle(y+1) = f(y)$$

This clearly defines a bijection, and it is computable in the sense above.

2.1.1 Topology

All these spaces are actually topological spaces, even metrizable spaces. The topology on $\mathbb{N}$ will be the discrete topology, and the natural metric will be the one where distinct numbers have distance 1.
The topology on $\mathbb{N}^{\mathbb{N}}$ will be the standard product topology. Since there may be readers not familiar with the product topology, we give a direct definition.

Definition 2.1.3 Let $O \subset \mathbb{N}^{\mathbb{N}}$. O will be *open* if whenever $f \in O$ there is a number n such that for all $g : \mathbb{N} \to \mathbb{N}$ we have that $g \in O$ whenever f and g agrees for all arguments between and including 0 and $n-1$.

In more technical terms, this can be expressed by

$$f \in O \Rightarrow \exists n \forall g(\bar{f}(n) = \bar{g}(n) \Rightarrow g \in O).$$

If σ is a finite sequence of numbers, σ determines an open neighborhood

$$B_\sigma = \{f \in \mathbb{N}^{\mathbb{N}} \mid \bar{f}(lh(\sigma)) = \sigma\}$$

i.e. the set of functions f extending σ. These sets B_σ will form a basis for the topology.

There is a close connection between topology and computability. There are many examples of this in the literature. Although the use of topology is restricted to fairly elementary general topology, it is very hard to read the current literature on computability on non-discrete structures without any knowledge of topology at all. In this section we will give one example of this connection.

Theorem 2.1.4 *Let $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$. Then the following are equivalent:*

a) F is continuous.

b) There is a function $f : \mathbb{N} \to \mathbb{N}$ such that F is computable relative to f.

Proof
First let F be computable relative to f, i.e. there is an index e such that

$$F(g) = \phi_e(f, g)$$

for all g.
For a fixed g, the computation tree of $\phi_e(f, g)$ is finite, and thus application of g will only be used a finite number of times in the computation tree. Let n be so large that if $g(x)$ occurs in the computation tree, then $x < n$. As a consequence we see that if $\bar{h}(n) = \bar{g}(n)$ then the computation trees of $\phi_e(f, g)$ and $\phi_e(f, h)$ will be identical. The further consequence is that

$$\forall h \in \mathbb{N}^{\mathbb{N}}(\bar{h}(n) = \bar{g}(n) \Rightarrow F(g) = F(h))$$

and this just means that F is continuous.

Now, let us prove the converse, and let F be continuous. Let $\{\sigma_n\}_{n\in\mathbb{N}}$ be a computable enumeration of all finite sequences. Let $X = \{n \mid F \text{ is constant on } B_{\sigma_n}\}$. Let f be defined by

$f(n) = 0$ if $n \notin X$

$f(n) = m + 1$ if F is constant m on B_{σ_n}.

Then for each $g \in \mathbb{N} \to \mathbb{N}$ we have that

$$F(g) = f(\mu n.f(n) > 0 \wedge \sigma_n \prec g) - 1 \tag{2.1}$$

where $\sigma \prec g$ means that the finite sequence σ is an initial segment of the infinite sequence g.

2.1.2 Associates

In the proof of Theorem 2.1.4 we constructed a function f such that $f(n) > 0$ if and only if F is constant on B_{σ_n} and in this case, $f(n) = F(g) + 1$ for all $g \in B_{\sigma_n}$. Following Kleene, we call this f the *principal associate* of F. This is of course an important concept, but from a computability theory point of view it is not completely satisfactory:

Lemma 2.1.5 *There is a computable functional F such that the principal associate is not computable.*

Proof
Let $A = W_e$ be c.e. but not computable, i.e.

$$n \in A \Leftrightarrow \exists m T(e, n, m)$$

where T is Kleene's T-predicate.
Define the computable F by

- $F(g) = 0$ if $\exists m < g(1)T(e, g(0), m)$

- $F(g) = 1$ otherwise.

Let f be the principal associate of F. Let $\nu(n) = \langle n \rangle$, i.e. the sequence number of the one-point sequence containing just n. F will be constant on $B_{\langle n \rangle}$ if and only if $n \notin A$, and then the constant value is 1, so we have

$$n \notin A \Leftrightarrow f(\nu(n)) = 2.$$

Since ν is computable and A is not computable, f cannot be computable.

When we showed that any functional F computable in f will be continuous, we referred to the computation tree of $\phi_e(f, g)$. When we look at the example showing Lemma 2.1.5 we see that the computation tree for $F(g)$ always will make use of $g(0)$ and of $g(1)$. Let f be defined by

- If $lh(\sigma_n) < 2$, let $f(n) = 0$.
- If $lh(\sigma_n) \geq 2$ and $\exists m < \sigma_n(1) T(e, \sigma_n(0), m)$, let $f(n) = 1$.
- If $lh(\sigma_n) \geq 2$ and $\forall m < \sigma_n(1) \neg T(e, \sigma_n(0), m)$, let $f(n) = 2$.

Then Equation 2.1 will hold for this F and f.
This leads us to the following definition:

Definition 2.1.6 Let $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ be continuous.
An *associate* for F will be a function $f : \mathbb{N} \to \mathbb{N}$ such that

i) If $\sigma_n \prec \sigma_m$ and $f(n) > 0$ then $f(m) = f(n)$.

ii) If $f(\bar{g}(m)) > 0$ then $f(\bar{g}(m)) = F(g) + 1$.

iii) $\forall g \in \mathbb{N}^{\mathbb{N}} \exists m (f(\bar{g}(m)) > 0)$.

It is then clear that any continuous functional will be computable in any of its associates via equation 2.1, and any computable functional will have a computable associate, see Exercise 2.2

2.1.3 Uniform continuity and the Fan Functional

It is well known that any continuous function on a compact metric space is uniformly continuous. In $\mathbb{R}$, a set will be compact if and only if it is closed and bounded. We have a similar characterization for $\mathbb{N}^{\mathbb{N}}$:

Definition 2.1.7 a) We will consider the following partial ordering of $\mathbb{N}^{\mathbb{N}}$:

$$f \leq g \Leftrightarrow \forall n (f(n) \leq g(n))$$

b) For $f \in \mathbb{N}^{\mathbb{N}}$, let $C_f = \{g \mid g \leq f\}$.

The following is left for the reader as Exercise 2.3

Lemma 2.1.8 *Let $A \subseteq \mathbb{N}^{\mathbb{N}}$. Then A will be compact if and only if A is closed and $A \subseteq C_f$ for some f.*

As a result, each continuous F will be uniformly continuous on each C_f. Let us see what this actually means. The formal definition is that for any $\epsilon > 0$ there is a $\delta > 0$ such that for all g and h in C_f, if $d(g,h) < \delta$ then $d(F(g),F(h)) < \epsilon$. The metric on $\mathbb{N}$ is trivial, so let $\epsilon = \frac{1}{2}$ and choose $\delta > 0$ accordingly. Choose n such that $2^{n-1} < \delta$. Then $\bar{g}(n) = \bar{h}(n) \Rightarrow F(g) = F(h)$ whenever g and h are in C_f. These considerations give us

Lemma 2.1.9 *Let $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ be continuous. Then*

$$\forall f \in \mathbb{N}^{\mathbb{N}} \exists n \forall g \in C_f \forall h \in C_f (\bar{g}(n) = \bar{h}(n) \to F(g) = F(h))$$

We suggest an alternative proof in Exercise 2.4.

Lemma 2.1.9 suggests that we may consider the operator Φ defined by:

Definition 2.1.10 The *Fan Functional* Φ is defined on all continuous $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ and defines a function $\Phi(F) : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ by

$$\Phi(F)(f) = \mu n. \forall g \in C_f \forall h \in C_f (\bar{g}(n) = \bar{h}(n) \to F(g) = F(h)).$$

Theorem 2.1.11 *Let Φ be the fan functional. Then $\Phi(F)$ is continuous whenever F is continuous.*

Proof
Let f be given, and let α be an associate for F. For each n there will be a finite set of sequences σ of length n that are bounded by f. Using König's lemma we may find an n such that $\alpha(\sigma) > 0$ for each σ in this set. This n can be found effectively in f and α, and will be an upper bound for $\Phi(F,f)$. We only need information from $\bar{f}(n)$ in order to verify that this n is a good one, and we may compute $\Phi(F)(f)$ from the information at hand. The details are left as an exercise for the reader.

Remark 2.1.12 In some sense originally made precise by Kleene and independently by Kreisel, the fan functional is continuous, see Exercise 2.5. In the next section we will consider a concept of computation where we may accept even functionals as inputs. This begs the question if the fan functional is even computable. The answer to this is not unique, since there is no canonical choice of a concept of computability in this case. In Exercise 2.5 we give a positive answer to this question for one possible notion of computability.

We characterized the continuous functionals as those having associates. In a way this is an old fashioned approach. In the current literature one often use *domains* as an entry to the theory of continuous functionals. We did not do so, because then we would have to introduce a lot of simple, but new, concepts. A reader interested in learning more about this kind of generalized computability is advised to consult some introduction to domain theory.

In Exercise 2.6 we will use some of the insight obtained in this section.

2.2 Computing relative to a functional of type 2

A major step in the process of generalizing computability came when Kleene and others started to relativize computations to functionals of type 2 and higher types in general. For certain indices e, we have defined the partial function $\phi_e(x_1, \ldots, x_n, f_1, \ldots, f_m)$, and of course there is no harm in accepting functionals $F_1, \ldots, F_k$ as dummy arguments. The problem is how we, in any sensible way, can use F_j actively in a computation.

The strategy will be that whenever we can supply F_j with an argument f, then we get the value $F_j(f)$ from some oracle call scheme.

Definition 2.2.1 Let $\Psi : \mathbb{N}^{n+1} \times (\mathbb{N}^{\mathbb{N}})^m \times (\mathbb{N}^{\mathbb{N}} \to \mathbb{N})^k \to \mathbb{N}$ be given.
Let $\vec{a} \in \mathbb{N}^n$, $\vec{f} \in (\mathbb{N}^{\mathbb{N}})^m$ and let $\vec{F} \in (\mathbb{N}^{\mathbb{N}} \to \mathbb{N})^k$.
By

$$\lambda x.\Psi(x, \vec{a}, \vec{f}, \vec{F})$$

we mean the function $g : \mathbb{N} \to \mathbb{N}$ defined by

$$g(x) = \Psi(x, \vec{a}, \vec{f}, \vec{F}).$$

This makes sense even when ψ is a partial functional, but then $\lambda x.\Psi(x, \vec{a}, \vec{f}, \vec{F})$ may not be total.

Kleene suggested something equivalent to

Definition 2.2.2 We extend the definition of $\phi_e(\vec{a}, \vec{f})$ to a definition of $\phi_e(\vec{a}, \vec{x}, \vec{F})$ by adding the clauses:

viii) If $e = \langle 8, d, j \rangle$ and

$$\lambda x.\phi_d(x, \vec{a}, \vec{f}, \vec{F})$$

is total, then

$$\phi_e(\vec{a}, \vec{f}, \vec{F}) = F_j(\lambda x.\phi_d(x, \vec{a}, \vec{f}, \vec{F})).$$

ix) If $e = \langle 9 \rangle$ then

$$\phi_e(d, \vec{a}, \vec{f}, \vec{F}) = \phi_d(\vec{a}, \vec{f}, \vec{F}).$$

This is an example of what is generally called an *inductive definition*. A large part of generalized computability theory is about inductive definitions and the computational principles that are behind them. In this case, we have to assume that each element in an infinite set of *subcomputations* will terminate before we accept some computations to terminate. In order to handle this properly, we will have to introduce a generalized concept of *computation tree*, now a computation tree may be an infinite, well founded tree with countable branching at certain nodes.

It will lead us too far in this introductory course to define all the concepts needed for a mathematically stringent handling of computability relative to functionals. As an appetizer, let us mention two kinds of problems:

1. What are the computable functionals of type 3?

2. What are the functions computable in a fixed functional?

Question 1 is actually hard to answer, there is no obvious characterization of this set. We know that the fan functional, which is defined only for continuous functionals of type 2, is not computable in the sense of Kleene. There are a few other positive and negative results about which functionals that are Kleene-computable, but nothing that is both general and informative. We know more about the answer to the second question, we will address this partly in the next section and partly as a small project discussed in Chapter 3. However, there is no complete and satisfactory characterization of the class of sets of functions that may turn out as the set of functions computable in some functional.

Remark 2.2.3 Scheme ix) may seem a bit dubious, and was actually considered to be a cheat. When we work with Turing machines or with Kleene's definition of computations relative to functions, the existence of a universal algorithm is an important theorem. However, without scheme ix) we will not be able to prove the existence of a universal algorithm for computations relative to functionals. We will follow Kleene, be pragmatic about it, and claim that including scheme ix) will give us a much more fruitful concept. However, introducing scheme ix) makes the two schemes v) for primitive recursion and vi) for the μ-operator redundant. This, and other facts about computability in functionals, are discussed in the nontrivial Exercise 2.10.

Definition 2.2.4 Let F be a functional of type 2, f a function of type 1. We say that f is *Kleene-computable in* F, $f <_K F$, if for some index e we have that

$$f(x) = \phi_e(x, F)$$

for all $x \in \mathbb{N}$.
This definition is extended in the canonical way to the concept $f <_K \vec{f}, \vec{F}$.
We let the *1-section* of F, $1 - sc(F)$ be the set

$$1 - sc(F) = \{f \mid f <_K F\}.$$

The next sequence of lemmas should be considered as a small project on which the reader might write an essay:

Lemma 2.2.5 *If $f <_K \vec{f}, \vec{F}$ and each f_i in $\vec{f}$ is computable in $\vec{g}, \vec{F}$, then $f <_K \vec{g}, \vec{F}$.*

Proof
We may find a primitive recursive function ρ such that if

$$f(x) = \phi_e(x, \vec{f}, \vec{F})$$

for all x and if

$$f_i(y) = \phi_{d_i}(y, \vec{g}, \vec{F})$$

for all y and i, then

$$f(x) = \phi_{\rho(e,e_1,\ldots,e_m)}(\vec{g}, \vec{F})$$

for all x.

Lemma 2.2.6 *Let F and G be functionals of type 2. Assume that $F(f) = G(f)$ whenever $f \in 1 - sc(F)$. Then $1 - sc(F) = 1 - sc(G)$.*

Proof
By induction on the ordinal rank of the computation tree we show that if $\phi_e(\vec{a}, G) = a$ then $\phi_e(\vec{a}, F) = a$ with the same computation tree.

One of Kleene's motivations for studying computations relative to functionals was to have a tool for investigating the computational power of quantifiers. Quantification over the natural numbers is captured by the discontinuous functional ${}^2E$ defined by:

Definition 2.2.7 Let

1. ${}^2E(f) = 1$ if $\exists a \in \mathbb{N}(f(a) > 0)$.
2. ${}^2E(f) = 0$ if $\forall a \in \mathbb{N}(f(a) = 0)$.

Definition 2.2.8 Let F and G be functionals of type 2.

a) $F <_K G$ if there is an index e such that $F(f) = \phi_e(G, f)$ for all functions $f : \mathbb{N} \to \mathbb{N}$.

b) F is *normal* if ${}^2E <_K F$.

The choice of the term 'normal' for these functionals reflects the focus of the early workers of higher type computability. We will investigate computability in ${}^2E$ more closely in section 4.4. We end this section by showing that the 1-section of a normal functional will be closed under jumps, see the proof of Lemma 1.3.31 for the definition.

Lemma 2.2.9 *Let F be a normal functional. Then $1 - sc(F)$ is closed under jumps.*

Proof
Generalizing Kleene's T-predicate, we see that if $g = f'$, there is a computable predicate T such that

$$g(a) = b \leftrightarrow \exists n T(a, b, n, f).$$

Given f and a we can use ${}^2E$ to decide if there are b and n such that $T(a, b, n, f)$ holds.
If it does, we may search for the relevant b and output $b + 1$. If it does not, we output 0.

2.3 ${}^2E$ versus continuity

Let f_i be defined as

$f_i(i) = 1$.

$f_i(j) = 0$ if $i \neq j$.

Then $\lim_{i\to\infty} f_i$ is the constant zero function f while ${}^2E(f) \neq \lim_{i\to\infty} {}^2E(f_i)$. This shows that ${}^2E$ is not continuous. Another way to see this is to observe that $({}^2E)^{-1}(\{0\})$ is not an open set, while $\{0\}$ is open in $\mathbb{N}$.

We will now restrict ourselves to considering computations of $\phi_e(\vec{a}, F)$, i.e. computations in one functional argument and some number arguments. We do this in order to save notation, there is no theoretical reason for this restriction.

We will define the n'th approximation $\phi_e^n(\vec{a}, F)$ to a computation of $\phi_e(\vec{a}, F)$. There will be four properties to observe

- $\phi_e^n(F, \vec{a})$ will always be defined.
- If $\phi_e(\vec{a}, F)\downarrow$ and $\phi_e(\vec{a}, F) = \lim_{n\to\infty} \phi_e^n(\vec{a}, F)$, then we can tell, in a uniform way, from which n_0 the limit is reached.
- If $\phi_e(\vec{a}, F)\downarrow$ and $\phi_e(\vec{a}, F) \neq \lim_{n\to\infty} \phi_e^n(\vec{a}, F)$ we can compute ${}^2E$ from F in a uniform way.
- If $\phi_e(\vec{a}, F)\downarrow$ we can computably distinguish between the two cases.

We will give most of the details, but the reader should be warned: The rest of this section is an example of how advanced arguments in computability theory might look like. (They often look more advanced than what they really are.)

Definition 2.3.1 We define $\phi_e^n(\vec{a}, F)$ following the cases for the definition of $\phi_e(\vec{a}, F)$ ignoring schemes v), vi) and vii).

i) If $e = \langle 1\rangle$, we let $\phi_e^n(\vec{a}, F) = \phi_e(\vec{a}, F)$.

ii) If $e = \langle 2, i\rangle$, we let $\phi_e^n(\vec{a}, F) = \phi_e(\vec{a}, F)$.

iii) If $e = \langle 3, q\rangle$, we let $\phi_e^n(\vec{a}, F) = \phi_e(\vec{a}, F)$.

iv) If $e = \langle 4, e', d_1, \ldots, d_m\rangle$, let

$$\phi_e^0(\vec{a}, F) = 0.$$
$$\phi_e^{n+1}(F, \vec{a}) = \phi_{e'}^n(\phi_{d_1}^n(\vec{a}, F), \ldots, \phi_{d_m}^n(\vec{a}, F)).$$

viii) If $e = \langle 8, d, 1\rangle$, then

$$\phi_e^0(\vec{a}, F) = 0.$$
$$\phi_e^{n+1}(\vec{a}, F) = F(\lambda x.\phi_d^n(x, \vec{a}, F)).$$

ix) If $e = \langle 9 \rangle$, let

$$\phi_e^0(d, \vec{a}, F) = 0.$$

$$\phi_e^{n+1}(d, \vec{a}, F) = \phi_d^n(\vec{a}, F).$$

$\phi_e^n(\vec{a}, F) = 0$ in all other cases.

Theorem 2.3.2 *There are three partial computable functions π, η and ν defined on the natural numbers such that $\phi_{\pi(e)}(\vec{a}, F)$, $\phi_{\eta(e)}(\vec{a}, F)$ and $\phi_{\nu(e)}(\vec{a}, F)$ terminate whenever $\phi_e(\vec{a}, F)$ terminates, and then*

- $\phi_{\pi(e)}(\vec{a}, F) = 0 \leftrightarrow \phi_e(\vec{a}, F) = \lim_{n\to\infty} \phi_e^n(\vec{a}, F)$.
- *If $\phi_e(\vec{a}, F) = \lim_{n\to\infty} \phi_e^n(\vec{a}, F)$ then $\phi_e^n(\vec{a}, F) = \phi_e(\vec{a}, F)$ whenever $n \geq \phi_{\eta(e)}(\vec{a}, F)$.*
- *If $\phi_e(\vec{a}, F) \neq \lim_{n\to\infty} \phi_e^n(\vec{a}, F)$ then ${}^2E$ is computable in F via the index $\phi_{\nu(e)}(\vec{a}, F)$.*

Proof
We will construct π, η and ν using the recursion theorem, so we will take the liberty to assume that we know the indices for these functions while defining them.
We will not hide the intuition behind a technically accurate construction of these functions, but to some extent describe in words what to do and why it works. When we explain this, we will assume as an induction hypothesis that our construction works for computations of lower complexity.
We will tell what to do when e corresponds to one of the schemes we have considered. The 'otherwise'-case is trivial, since we do not have to prove anything in this case, ϕ_e does not terminate on any input.

If e corresponds to scheme i), ii) or iii), let $\pi(e)$ be the index for the constant zero, $\eta(e)$ the same, and we may without loss of consequences let $\nu(e)$ also be the same.

Let $e = \langle 4, e', d_1, \ldots, d_m \rangle$.
Let $\pi(e)$ be the index for the following enumerated algorithm in $(\vec{a}, F)$:

1. If $\phi_{\pi(e')}(\vec{a}, F) = \phi_{\pi(d_1)}(\vec{a}, F) = \cdots = \phi_{\pi(d_m)}(\vec{a}, F) = 0$, let $\phi_{\pi(e)}(\vec{a}, F) = 0$ and go to 2., otherwise let $\phi_{\pi(e)}(\vec{a}, F) = 1$ and go to 3.
2. Let $\phi_{\eta(e)}(\vec{a}, F) = 1 + max\{\phi_{\eta(e')}(\vec{a}, F), \phi_{\eta(d_1)}(\vec{a}, F), \ldots, \phi_{\eta(d_m)}(\vec{a}, F)\}$.
3. Select the least index $d \in \{e', d_1, \ldots, d_m\}$ such that $\phi_{\pi(d)}(\vec{a}, F) \neq 0$. Then use the index $\phi_{\nu(d)}(F, \vec{a})$ for ${}^2E$ to decide if $\phi_e(\vec{a}, F) = \lim_{n\to\infty} \phi_e^n(\vec{a}, F)$ or not. If it is, move to 4., otherwise to 5.
4. We let $\phi_{\pi(e)}(\vec{a}, F) = 0$. We use ${}^2E$ to compute the proper value of $\phi_{\eta(e)}(\vec{a}, F)$.
5. We let $\phi_{\pi(e)}(\vec{a}, F) = 1$, and we let $\phi_{\nu(e)} = \phi_{\nu(d)}$.

The case ix) is handled in the same way, so we restrict our attention to case viii):

$$\phi_e(\vec{a}, F) = F(\lambda x.\phi_d(x, \vec{a}, F)).$$

Let $f(x) = \phi_e(x, \vec{a}, F)$ and let $f_n(x) = \phi_e^n(x, \vec{a}, F)$.

For each $m \in \mathbb{N}$, define $g_m(x)$ by the following algorithm:

1. Ask if $f(x) = \lim_{n\to\infty} f_n(x)$. If yes, continue with 2., otherwise continue with 3.

2. If $F(f) = F(f_n)$ for all n such that $m \leq n \leq \phi_{\eta(d)}(x, F, \vec{a})$, let $g_m(x) = f_{\phi_{\eta(d)}(x,\vec{a},F)}(x)$.
   Otherwise choose the least $n \geq m$ such that $F(f) \neq F(f_n)$ and let $g_m(x) = f_n(x)$.

3. By the induction hypothesis, $\phi_{\nu(d)}(x, \vec{a}, F)$ provides us with an index to compute ${}^2E$ from F. Use ${}^2E$ to ask if

   $$\exists n \geq m(F(f) \neq F(f_n)).$$

   If not, let $g_m(x) = f(x)$, otherwise let $g_m(x) = f_n(x)$ for the least such n.

In both cases, we will let $g_m(x) = f(x)$ if $F(f) = F(f_n)$ for all $n \geq m$, while $g_m(x) = f_n(x)$ for the least counterexample otherwise. In the case 3. this is done explicitly. In case 2. we must use that if $n > \phi_\eta(d)(\vec{a}, F)$, then $f_n(x) = f_{\phi_{\eta(d)}(x,F,\vec{a})}(x)$, a fact that follows from the induction hypothesis.
Thus for each m we may computably decide if $\exists n \geq m(F(f) \neq F(f_n))$ by

$$\exists n \geq m(F(f) \neq F(f_n)) \Leftrightarrow F(f) = F(g_m)). \tag{2.2}$$

From 2.2 we can decide if

$$\exists n(F(f) \neq F(f_n)).$$

If not, we let $\phi_{\pi(e)}(\vec{a}, F) = \phi_{\eta(e)}(\vec{a}, F) = 0$.
If there exists one such n we want to decide in a computational way if there are infinitely many of them or not. We may use the same kind of construction as that of $g_m(x)$ to compute a function $h(x)$ such that $h = f$ if $F(f) \neq F(f_n)$ for infinitely many n, while $h = f_n$ for the largest n such that $F(f) \neq F(f_n)$ otherwise. We must split the construction into the same cases as for the construction of $g_m(x)$, and we must rely on 2.2 when ${}^2E$ is not available. Then we get

$$\exists n \forall m \geq n(F(f) = F(f_m)) \Leftrightarrow F(f) \neq F(h)). \tag{2.3}$$

In case both sides of equivalence 2.3 are positive, we can use equivalence 2.2 and the μ-operator to find $\phi_{\eta(e)}(\vec{a}, F)$. It remains to show how to compute ${}^2E$ from F in case both sides of equivalence 2.3 are negative. Actually, we will use the same trick one third time. Assume that $F(f) \neq F(f_n)$ for infinitely many n. Let g be given. We want to decide

$$\exists x(g(x) \neq 0).$$

We construct $f'(x)$ as follows:

- If $f(x) \neq \lim_{n\to\infty} f_n(x)$ use ${}^2E$ to decide if $\exists x(g(x) \neq 0)$. If this is the case, select the least such x, select the least $n \geq x$ such that $F(f) \neq F(f_n)$ and let $f'(x) = f_n(x)$. If it is not the case, let $f'(x) = f(x)$.
- Otherwise, ask if there is an $x \leq \phi_{\eta(d)}(x, \vec{a}, F)$ such that $g(x) \neq 0$. In both cases, do as above.

Then

$$\exists x(g(x) \neq 0) \Leftrightarrow F(f) \neq F(f'))$$

and we are through.
This ends our proof of the theorem.

Remark 2.3.3 Our use of the recursion theorem is sound, but requires some afterthought by the inexperienced reader. What is required in order to make the definition of π, η and ν sound is to view them as Turing-computable functions operating on indices for computable functionals of higher types. We have given explicit constructions of $\phi_{\pi(e)}$ etc. with self reference, and analyzing exactly which combinations of the schemes that are required, we can express $\pi(e)$ etc. as functions of the Turing indices for π, η and ν. Then, by the recursion theorem, a solution exists.

This theorem has an interesting corollary. A *dichotomy theorem* is a theorem stating that a general situation splits into two nice, in a sense opposite, cases. One case should then contain more information than just the negation of the other. We have observed that if F is a normal functional, then $1 - sc(F)$ will be closed under jump. This is actually a characterization, but in a very strong sense. Clearly any 1-section will be closed under the relation 'computable in'. We defined a function to be of c.e.degree if it is Turing equivalent with the characteristic function of a c.e. set. This concept may of course be relativized to any function g. Finally, we say that a set Y of functions is *generated* from a set X of functions if Y consists of all functions computable in some $g_1, \ldots, g_n$ from X. We then have

Corollary 2.3.4 *Let F be a functional of type 2. Then one of two will be the case:*

1. *${}^2E <_K F$, i.e. F is normal.*
2. *There is some $f \in 1 - sc(F)$ such that $1 - sc(F)$ is generated by the elements in $1 - sc(F)$ of c.e.(f)-degree.*

Proof
If there is one terminating computation $\phi_e(\vec{a}, F)$ such that

$$\lim_{n\to\infty} \phi_e^n(\vec{a}, F) \neq \phi_e(\vec{a}, F)$$

then F is normal by Theorem 2.3.2.
Otherwise, let $f(\langle e, n, \vec{a}\rangle) = \phi_e^n(\vec{a}, F)$. Clearly, $f \in 1 - sc(F)$. Let $g \in 1 - sc(F)$

be given, $g(x) = \phi_e(x, F)$ for all x. Then g is computable in f and the c.e.(f) set

$$A = \{(x, n) \mid \exists m \geq n(\phi_e^m(x, F) \neq \phi_e^n(x, F))\}$$

which again is computable in F via $\lambda x.\phi_{\eta(e)}(x)$.

Kleene computability is extended to functionals of all finite types, and this dichotomy actually still holds. However, working with functionals in which ${}^2E$ is not computable, one may as well work with what is known as the hereditarily continuous functionals. We will touch a little bit on this in section 4.5.

2.4 The Hyperarithmetical sets

Definition 2.4.1 A set is *hyperarithmetical* if it is computable in ${}^2E$.

We have cheated a bit, and used a characterization due to Kleene as our definition. For a more systematical introduction to hyperarithmetical sets and higher computability theory in general, see Sacks [6].

The term 'hyperarithmetical' indicates that this is an extension of the arithmetical sets in a natural way, and this is exactly what the intention is. The arithmetical sets will be the sets that can be defined by first order formulas in number theory. For each Gödel number a of a first order formula $\psi(x)$ with one free variable, we can, in a primitive recursive way, find an index $\pi(a)$ such that

$$\lambda x.\phi_{\pi(a)}(x, {}^2E)$$

will be the characteristic function of

$$\{x \mid \psi(x)\}.$$

Using the enumeration scheme ix) we can then find a subset B of $\mathbb{N} \times \mathbb{N}$ that is computable in ${}^2E$ and such that each arithmetical set A is a section $B_x = \{y \ ; (x, y) \in B\}$ of B. Then every set arithmetical in B will also be hyperarithmetical, and so forth. In a sense, we may say that ${}^2E$ provides the 'arithmetical' while the enumeration scheme provides the 'hyper'.

2.4.1 Trees

If we want to analyze computations relative to functionals more closely, we need a precise notion of a computation tree. Although there is a common intuition behind all our concepts of trees, various needs require various conventions. For the rest of this section we will use

Definition 2.4.2 a) A tree will be a nonempty set T of sequences of natural numbers, identified with their sequence numbers, that is closed under initial segments. The elements of T will be called *nodes*.

b) $\langle\rangle$, i.e. the empty sequence, is the *root node* of the tree.

c) A *leaf node* of a tree T will be a node in T with no proper extension in T.

d) A *branch* in a tree T will be a maximal, totally ordered subset of T. If a branch is infinite, we identify it with the corresponding function $f : \mathbb{N} \to \mathbb{N}$ and if a branch is finite, we identify it with its maximal element.

e) A *decorated tree* will be a tree T together with a decoration, i.e. a map $f : T \to \mathbb{N}$.

f) If T_i is a tree for each $i \in I$, where $I \subseteq \mathbb{N}$, we let $\langle T_i \rangle_{i \in I}$ be the tree T consisting of the empty sequence together with all sequences $i * \sigma$ such that $i \in I$ and $\sigma \in T_i$.

g) If $a \in \mathbb{N}$ and (T_i, f_i) are decorated trees for each $i \in I$, we let

$$\langle a, (T_i, f_i) \rangle_{i \in I}$$

be the decorated tree (T, f) defined by

- $T = \langle T_i \rangle_{i \in I}$
- $f(\langle\rangle) = a$
- $f(i * \sigma) = f_i(\sigma)$ when $i \in I$ and $\sigma \in T_i$.

A computation tree will be a decorated tree. Since it actually is the decorations that will be of interest, we sometimes take the liberty to identify them with the corresponding nodes with decoration in the tree. This liberty will be visible when we discuss the root and the leaves of a computation tree.

Definition 2.4.3 Let $\phi_e({}^2E, \vec{a}) = b$. We define the *computation tree* of the computation by recursion as follows

i) $e = \langle 1 \rangle$: Let the root node also be the only leaf node and be decorated with $\langle e, \vec{a}, b \rangle$.

ii) $e = \langle 2, i \rangle$. Act as in case i).

iii) $e = \langle 3, q \rangle$. Act as in case i).

iv) $e = \langle e', d_1, \ldots, d_m \rangle$: Let $(T_1, f_1), \ldots, (T_m, f_m)$ be the computation trees of $\phi_{d_i}({}^2E, \vec{a}) = c_i$ resp. and let (T_{m+1}, f_{m+1}) be the computation tree of $\phi_{e'}({}^2E, c_1, \ldots, c_m) = b$.
Let the computation tree of $\phi_e({}^2E, \vec{a}) = b$ be

$$\langle \langle e, \vec{a}, b \rangle, T_i \rangle_{i=1}^{m+1}.$$

vii) $e = \langle 8, d \rangle$: For each $i \in \mathbb{N}$, let (T_i, f_i) be the computation tree of $\phi_d(i, \vec{a}, {}^2E)$ and let the computation tree of $\phi_e(\vec{a}, {}^2E) = b$ be

$$\langle \langle e, \vec{a}, b \rangle, (T_i, f_i) \rangle_{i \in \mathbb{N}}.$$

The rest of the cases are similar, and the details are left for the reader.

Lemma 2.4.4 *Whenever $\phi_e(\vec{a}, {}^2E)$ terminates, then the computation tree will be computable in ${}^2E$.*

Proof
We see how the computation tree is constructed from the computation trees of the immediate subcomputations. This construction is clearly computable in ${}^2E$, and using the recursion theorem for computations relative to ${}^2E$ we get a uniform algorithm for computing the computation tree from a computation.

Recall that a tree is well founded if there is no infinite branch in the tree. It is easy to see that the computation trees of terminating computations will be well founded.

Definition 2.4.5 A *pre-computation tree* will be a decorated tree that locally looks like a computation tree, i.e. each node will be of the form $\langle d, \vec{a}, b\rangle$, the leaf nodes will correspond to initial computations, and other nodes relate to their immediate subnodes as in Definition 2.4.3.

Lemma 2.4.6 *a) The concept of a pre-computation tree is arithmetical.*

b) A pre-computation tree that is well founded is actually a computation tree.

c) If $\phi_e(\vec{a}, {}^2E)$ terminates, there is exactly one pre-computation tree with a root node on the form $\langle e, \vec{a}, b\rangle$, and then b will be the value of $\phi_e(\vec{a}, {}^2E)$.

Proof
a) is trivial. In order to prove b) we observe that a subtree of a well founded pre-computation tree is itself a well founded pre-computation tree. Since facts may be proved by induction on the subtree ordering of well founded trees, we may use this kind of induction to prove the statement. The details are trivial. The set of computations relative to ${}^2E$ was defined by induction, and then we may prove facts by induction over this construction. c) is proved this way in a trivial manner.

2.4.2 Π^0_k-sets etc.

In this section we will let 'computable' mean 'Turing-computable'.

Definition 2.4.7 a) A *product set* will be any product of the sets $\mathbb{N}$ and $\mathbb{N}^{\mathbb{N}}$ in any finite number and order. We define the *arithmetical hierarchy* of subsets of product sets X as follows:

i) A set A is a Σ^0_0-set and a Π^0_0-set if A is computable. (In the literature you may find that A is supposed to be definable using only bounded quantifiers in order to be in these classes. This distinction does not matter for our applications.)

ii) A set $A \subset X$ is Σ^0_{k+1} if for some Π^0_k subset B of $\mathbb{N} \times X$ we have that

$$\vec{x} \in A \Leftrightarrow \exists x \in \mathbb{N}((x, \vec{x}) \in B).$$

iii) A set $A \subset X$ is Π^0_{k+1} if for some Σ^0_k subset B of $\mathbb{N} \times X$ we have that

$$\vec{x} \in A \Leftrightarrow \forall x \in \mathbb{N}((x, \vec{x}) \in B).$$

iv) A is a Δ^0_k-set if A is both Π^0_k and Σ^0_k.

b) We define the *analytical hierarchy* in the same fashion, but will only need the first level here:

i) A set $A \subseteq X$ is Π^1_1 if there is some arithmetical set $B \subseteq \mathbb{N}^{\mathbb{N}} \times X$ such that

$$\vec{x} \in A \Leftrightarrow \forall f \in \mathbb{N}^{\mathbb{N}}((f, \vec{x}) \in B).$$

ii) A set $A \subseteq X$ is Σ^1_1 if there is some arithmetical set $B \subseteq \mathbb{N}^{\mathbb{N}} \times X$ such that

$$\vec{x} \in A \Leftrightarrow \exists f \in \mathbb{N}^{\mathbb{N}}((f, \vec{x}) \in B).$$

iii) A is Δ^1_1 if A is both Σ^1_1 and Π^1_1.

Lemma 2.4.8 *Let X be a product set, $A \subseteq X$. Then A is Σ^0_1 if and only if A is the domain of a partial computable function of arity X.*

Proof
For product sets where all factors are $\mathbb{N}$, these are two of the characterizations of c.e.-sets. The proof of the equivalence generalizes trivially to all product sets.

In section 4.1 we showed how each product class either is homeomorphic to $\mathbb{N}$ or to $\mathbb{N}^{\mathbb{N}}$. We used pairing-functions $\langle -, - \rangle$ for pairs of numbers, pairs of functions or for a pair consisting of one number and one function for this. These functions can be used to show that multiple quantifiers of the same kind can be reduced to one, and that number quantifiers can be 'eaten up' by function quantifiers. For a complete proof, we need one more homeomorphism:

Lemma 2.4.9 $(\mathbb{N}^{\mathbb{N}})^{\mathbb{N}}$ *is homeomorphic to* $\mathbb{N}^{\mathbb{N}}$.

Proof
We use the observation $(\mathbb{N}^{\mathbb{N}})^{\mathbb{N}} \approx \mathbb{N}^{\mathbb{N} \times \mathbb{N}} \approx \mathbb{N}^{\mathbb{N}}$ since $\mathbb{N} \times \mathbb{N} \approx \mathbb{N}$. Precisely, if $\{f_i\}_{i \in \mathbb{N}}$ is a sequence of functions, we code it as one function by

$$\langle f_i \rangle_{i \in \mathbb{N}}(\langle n, m \rangle) = f_n(m).$$

This is a computable bijection.

All these coding functions will have inverses for each coordinate. We use $(-)_i$ for the inverse at coordinate i, letting pairing functions have coordinates 1 and 2. We can then perform the following reductions of quantifiers. Note that all the dual reductions will also hold.

Lemma 2.4.10 *For each relation R the following equivalences hold:*

i) $\exists n \exists m R(n,m) \Leftrightarrow \exists k R((k)_1,(k)_2)$.

ii) $\exists f \exists g R(f,g) \Leftrightarrow \exists h R((h)_1,(h)_2)$.

iii) $\exists f \exists n R(f,n) \leftrightarrow \exists g R((g)_1,(g)_2)$ (Where the decoding is different from the one in ii).).

iv) $\forall n \exists m R(n,m) \Leftrightarrow \exists f \forall n R(n,f(n))$.

v) $\forall n \exists f R(n,f) \Leftrightarrow \exists g \forall n R(n,(g)_n)$.

The proofs are trivial and are left for the reader as Exercise 2.11.

Lemma 2.4.11 *a) The classes Σ^0_k, Π^0_k, Σ^1_1 and Π^1_1 are closed under finite unions and finite intersections. Moreover, the Σ-classes are closed under $\exists n \in \mathbb{N}$ and the Π-classes are closed under $\forall n \in \mathbb{N}$.*

b) The classes Π^1_1 and Σ^1_1 are closed under number quantifiers.

c) **Π^1_1-normal form theorem**
If $A \subseteq X$ is Π^1_1, there is a computable set $R \subseteq \mathbb{N}^{\mathbb{N}} \times \mathbb{N} \times X$ such that

$$\vec{x} \in A \Leftrightarrow \forall f \exists n R(f,n,\vec{x}).$$

In the proof we use standard prenex operations and the reductions described in Lemma 2.4.10 The details are left for the reader, see Exercise 2.11.

Theorem 2.4.12 *Let Γ be one of the classes Π^0_k, Σ^0_k ($k \geq 1$), Π^1_1 or Σ^1_1 (or Π^1_k, Σ^1_k as defined in Exercise 2.12). For each product set X there is a universal Γ-set A in $\mathbb{N} \times X$, i.e. such that for each Γ-set $B \subseteq X$ there is an $n \in \mathbb{N}$ such that for all $\vec{x} \in X$:*

$$\vec{x} \in B \Leftrightarrow (n,\vec{x}) \in A.$$

Proof
Let $\tilde{\Gamma}$ be the dual of Γ, i.e. the set of complements of sets in Γ. Clearly, if the property holds for Γ, it also holds for $\tilde{\Gamma}$. Since the Π- and Σ-classes are duals of each other, it is sufficient to prove the lemma for one of each pair.
The existence of universal algorithms together with Lemma 2.4.8 ensures the theorem to hold for Σ^0_1. Then any class Γ of sets definable from Σ^0_1-sets using a fixed quantifier prefix will have universal sets for each product set. Each of our classes Γ is either one of these or the dual to one of these. For Π^1_1 we need the normal form theorem, see Lemma 2.4.11 c).

We will end this subsection by viewing the connection between Π^1_1-sets and well founded trees.
Let $A \subseteq \mathbb{N}$ be a Π^1_1-set written in its normal form

$$m \in A \Leftrightarrow \forall f \exists n R(m,n,f).$$

Since R is computable, there is, by the finite use property, a computable relation R^+ on $\mathbb{N}^3$ such that

$$m \in A \Leftrightarrow \forall f \exists n \exists k R^+(m, n, \bar{f}(k)).$$

For each m we let T_m be the tree of finite sequences σ such that

$$\forall n \leq lh(\sigma) \forall \tau \prec \sigma \neg R^+(m, n, \tau).$$

Then T_m will be a tree, and we will have

$$m \in A \Leftrightarrow T_m \text{ is well founded.}$$

If we put a bit more effort into the arguments, we can actually show that we may choose R^+ to be primitive recursive, this is connected with the relativized Kleene's T-predicate, and thus each Π^1_1 subset of $\mathbb{N}$ is m-reducible to the set of indices for well founded primitive recursive trees. On the other hand, this set is easily seen to be Π^1_1 itself. Thus the class of Π^1_1-sets contains an m-maximal element, just like the c.e. sets.
In the next subsection we will se more analogies between the c.e. sets and the Π^1_1-sets.

2.4.3 Semicomputability in ${}^2E$ and Gandy Selection

Definition 2.4.13 A subset $A \subseteq \mathbb{N}$ is *semicomputable in* ${}^2E$ if there is an index e such that for all $a \in \mathbb{N}$:

$$\phi_e(a, {}^2E)\downarrow \Leftrightarrow a \in A,$$

where $\downarrow$ still means 'terminates'.

Lemma 2.4.14 *All* Π^1_1 *subsets of* $\mathbb{N}$ *are semicomputable in* ${}^2E$.

Proof
Clearly the set of indices for computable trees is computable in ${}^2E$, so it is sufficient to show that the concept of a well founded tree is semicomputable. This can be done with a careful use of the recursion theorem: Let e be any index. Using the enumeration scheme and ${}^2E$ we can define a computable function $\Phi(e, T)$ such that

- $\Phi(e, T, {}^2E) = 0$ if T consists of exactly the empty sequence.
- $\Phi(e, T, {}^2E) = {}^2E(\lambda n \phi_e(T_n))$ if T is a more complex tree, where T_n is the set of sequences σ such that $n * \sigma \in T$.

By the recursion theorem for ${}^2E$, there is an index e_0 such that $\Phi(e_0, T, {}^2E) = \phi_{e_0}(T, {}^2E)$ for all T.
By induction on the well founded trees it is easy to se that $\phi_{e_0}(T)$ will terminate whenever T is well founded. In order to prove the other direction, we must inspect the proof of the recursion theorem and see that in this particular case, if $\phi_{e_0}(T)$ terminates, then either T consists of the empty sequence only or $\phi_{e_0}(T_n)$

must terminate for each n and be a subcomputation. This is because we only used e as an index in connection with the enumeration scheme in defining Φ. It follows that T must be well founded.

Theorem 2.4.15 *Let $A \subseteq \mathbb{N}$. Then the following are equivalent:*

i) A is Π^1_1.

ii) A is semicomputable in ${}^2E$.

Proof
Lemma 2.4.14 gives us one direction.
The set C of computation tuples $\langle e, \vec{a}, b\rangle$ such that $\phi_e({}^2E, \vec{a}) = b$ is defined by a *positive induction*, i.e. there is a formula $\Phi(x, X)$ such that atomic subformulas $t(x) \in X$ will occur positively and such that C is the least set such that

$$C = \{a \mid \Phi(a, C)\}.$$

Then

$$c \in C \Leftrightarrow \forall B(B \subseteq \{a \mid \Phi(a, B)\} \rightarrow c \in B).$$

Now is the time to assume that the reader is familiar with well orderings and preferably with ordinal numbers. For readers unfamiliar with this, we offer Exercise 3.5, a guided self-service introduction to the topic.

Definition 2.4.16 Let T be a well founded tree on $\mathbb{N}$.

a) The *rank* $|T|$ of T, is the ordinal rank of T seen as a well founded relation, where $\sigma \leq \tau \Leftrightarrow \tau \prec \sigma$.

b) The *rank* $|\sigma|_T$ *of* $\sigma \in T$, is the value of the rank function for T on σ.

c) If T is not well founded, we let $|T| = \infty$ considered to be larger than any ordinal number.

Theorem 2.4.17 *There is a two-place function Φ partially computable in ${}^2E$ such that $\phi(T, S)$ terminates exactly when both S and T are trees, and at least one of them is well founded, and then*

1. *If $|T| \leq |S|$ then $\Phi(T, S) = 0$*
2. *If $|S| < |T|$ then $\Phi(T, S) = 1$.*

Proof
We will use the recursion theorem for ${}^2E$, and the argument is an elaboration on the argument showing that the set of well founded trees is semicomputable in ${}^2E$.
We define $\Phi(S, T)$ by cases, using self reference, and the solution using the construction behind the recursion theorem will give us the result. The set of trees is computable in ${}^2E$, so for the sake of convenience, we assume that both T and S are trees. Then

- If T consists of only the empty sequence, let $\Phi(T,S)=0$.
- If S consists of only the empty sequence, but T contains more, let $\Phi(T,S)=1$.
- If neither T nor S contains just the empty sequence, let

$$\Phi(S,T)=0 \Leftrightarrow \forall n \exists m(\Phi(T_n,S_m)=0),$$

where we use ${}^2E$ to decide this, and let 1 be the alternative value.

It is easy to see by induction on the rank that if one of the trees is well founded, then Φ does what it is supposed to do. If neither T nor S are well founded, $\Phi(T,S)$ will not terminate. This is however not important, and we skip the argument.

Since every terminating computation is associated with a well founded computation tree, we may think of the rank of the computation tree as a measure of the length of the computation.

Definition 2.4.18 Let $\phi_e({}^2E,\vec{a})=b$. The *length* $|\langle e,\vec{a},b\rangle|$ of the computation will be the ordinal rank of the corresponding computation tree. If $\langle e,\vec{a},b\rangle$ is not the tuple of a terminating computation, we let $|\langle e,\vec{a},b\rangle|=\infty$.

Theorem 2.4.17 has an interesting application, the *Stage Comparison Theorem*:

Corollary 2.4.19 *There is a function Ψ partially computable in ${}^2E$ such that whenever $\sigma=\langle e,\vec{a},b\rangle$ and $\sigma'=\langle e',\vec{a}',b'\rangle$ are tuples, then $\Psi(\sigma,\sigma')$ terminates if and only if at least one of σ and σ' is a genuine computation tuple, and then*

$$\Psi(\sigma,\sigma')=0 \Leftrightarrow |\sigma| \leq |\sigma'|.$$

We have proved a selection theorem for ordinary c.e. sets, the selection was performed by searching for the least pair where the second element was a witness to the fact that the first element was in the c.e. set in question. Gandy showed how we may combine this idea and the ordinal length of computations to prove a selection theorem for computations in ${}^2E$:

Theorem 2.4.20 *(***Gandy Selection***)*
Let $A \subseteq \mathbb{N}\times\mathbb{N}$ be c.e. in ${}^2E$. Then there is a selection function for A computable in ${}^2E$.

Proof
We will use the recursion theorem for ${}^2E$. Let

$$A=\{(a,b) \mid \phi_{e_0}(a,b,{}^2E)\downarrow\}.$$

Let f be computable in ${}^2E$ satisfying

1. $f(e,a,k)=0$ if $|\langle e_0,a,k\rangle| < |\langle e,a,k+1\rangle|$.

2. $f(e,a,k) = f(e,a,k+1)+1$ if $\phi_e(a,k+1,{}^2E)\downarrow$ and $|\langle e,a,k+1\rangle| \leq |\langle e_0,a,k\rangle|$.

By the recursion theorem there is an index e_1 such that

$$f(e_1,a,k) = \phi_{e_1}(a,k,{}^2E).$$

We claim that $\lambda a.\phi_{e_1}(a,0,{}^2E)$ is a selection function for A. Let $a \in \mathbb{N}$ be given. First observe that if $\phi_{e_0}(a,k,{}^2E)\downarrow$ then $f(e,a,k)$ will terminate, so $\phi_{e_1}(a,k,{}^2E)\downarrow$. Moreover, observe that if $\phi_{e_1}(a,k,{}^2E)\downarrow$ then $\phi_{e_1}(a,k',{}^2E)\downarrow$ whenever $k' < k$. Thus

$$\exists k \phi_{e_0}(a,k,{}^2E)\downarrow \Rightarrow \phi_{e_1}(a,0,{}^2E)\downarrow .$$

Now assume that $\phi_{e_1}(a,0,{}^2E)\downarrow$. If we look at the computation of $\phi_{e_1}(a,0,{}^2E)$ we see that we will have $\phi_{e_1}(a,1,{}^2E)$, $\phi_{e_1}(a,2,{}^2E)$,... as subcomputations as long as part 2 of the algorithm for f is followed. The ranks of these computations will be a decreasing sequence of ordinals, and this sequence must come to an end. It comes to an end exactly when we hit a k_0 such that

$$|\langle e_0,a,k_0\rangle| < |\langle e_1,a,k_0+1\rangle|.$$

Then $(a,k_0) \in A$ and backtracking the value of $\phi_{e_1}(a,k',{}^2E)$ for $k' \leq k_0$ we see that $\phi_{e_1}(a,k',{}^2E) = k_0 - k'$.
Consequently $\lambda a.\phi_{e_1}(a,0,{}^2E)$ will be a selection function for A

Remark 2.4.21 This proof is of course uniform in e_0.

Corollary 2.4.22 *Let $A \subseteq \mathbb{N}$. Then the following are equivalent:*

i) A is computable in ${}^2E$.

ii) Both A and $\mathbb{N} \setminus A$ are semicomputable in ${}^2E$.

Proof
Clearly, if A is computable in ${}^2E$, then both A and its complement will be semicomputable in ${}^2E$.
Now assume that both A and $\mathbb{N} \setminus A$ are semicomputable in ${}^2E$.
Let

$$B = \{(0,n) \mid n \in A\} \cup \{(1,m) \mid m \notin A\}.$$

B is semicomputable, and B is the graph of a function. By Gandy selection this function must be computable in ${}^2E$. The result follows.

2.4.4 Characterising the hyperarithmetical sets

There is almost nothing left for us to do in this subsection. We have shown that a set A is semicomputable in ${}^2E$ if and only if it is Π^1_1. We have shown that a set B is computable in ${}^2E$ if both B and its complement are semicomputable in ${}^2E$. This gives us

Corollary 2.4.23 *A set is hyperarithmetical if and only if it is* Δ^1_1.

Remark 2.4.24 Our proof of Corollary 2.4.23 is flavored by computability theory, but there are alternative proofs in the literature. The result, in a different form, goes back to Suslin in 1917. He essentially showed that a set $A \subseteq \mathbb{N}^{\mathbb{N}}$ is Borel if and only if both A and its complement are projections of closed sets in $(\mathbb{N}^{\mathbb{N}})^2$, i.e. that relativized Δ^1_1 is the same as Borel.

There are numerous analogies between the pair (Computable, computably enumerable) and the pair (Δ^1_1, Π^1_1). Some of these are left for the reader as Exercise 2.13.
There is a close connection between hyperarithmetical theory and fragments of set theory.

2.5 Typed λ-calculus and PCF

We recommend Streicher [8] for an almost complete introduction to this subject. Kleene's definition of computations relative to functionals can be extended to functionals of even higher types than two. This will, however, be a too specialized topic to be introduced in this text. Kleene's concept turned out to be useful in definability theory, the theory of what may be defined with the help of certain principles, but we have moved quite a bit away from what what we might call "genuine computability".

The concept of higher type computability is nevertheless of interest also when genuine computability is the issue, e.g. in theoretical computer science. In this section we will give a brief introduction to PCF. PCF is a formal programming language for computing with typed objects. It has its roots in work by Platek, Scott developed the first version of it as a formal logic for computability and Plotkin gave it the form we will be investigating. The intuition should be that we are operating with hereditarily partial, monotone and continuous functionals. We will explain this better when needed.

2.5.1 Syntax of PCF

Definition 2.5.1 We define the *formal types*, or just *types* as a set of terms for types as follows:

1. ι and o are types. These are called the formal *base types*.
2. If σ and τ are types, then $(\sigma \to \tau)$ is a type.

Remark 2.5.2 When we give a semantical interpretation of PCF, we will associate a mathematical object to each formal type. We think of ι as denoting the set of natural numbers, o as denoting the set of boolean values and $(\sigma \to \tau)$ as denoting the relevant set of functions mapping objects of type σ to objects of type τ. Since we will use these objects to interpret algorithms, we have to

interpret nonterminating algorithms as well. We will do so by including an element for 'the undefined' in the base types, and carry this with us for higher types.

As usual the term language will consist of variables, constants and combined terms. What is new is that each term will be typed, that some constants are treated as function symbols, and that we need type matching when forming combined terms.

Definition 2.5.3 The terms in PCF are inductively defined as follows:

1. Variables:
   For each type σ there is an infinite list x_i^σ of variables of type σ.

2. Constants:
   In PCF we have the following typed constants:

   k_n of type ι for each natural number n.

   tt and ff of type o.

   Z of type $\iota \to o$.

   S and P of type $\iota \to \iota$.

   $\supset_\iota$ and $\supset_o$ of types $o \to (\iota \to (\iota \to \iota))$ and $o \to (o \to (o \to o))$ resp.

   Y_σ of type $(\sigma \to \sigma) \to \sigma$ for each type σ.

3. Combined terms:

   If M is a term of type $\sigma \to \tau$ and N is a term of type σ, then (MN) is a term of type τ. This construction is called *application*.

   If M is a term of type τ and x is a variable of type σ, then $(\lambda x.M)$ is a term of type $\sigma \to \tau$.
   This construction is called *abstraction*.

The intuition behind these constants and terms are as follows:
0 will denote the zero-element in $\mathbb{N}$, tt and ff the two truth values.
Z will test if a number is zero or not.
S and P are the successor and predecessor operators on the natural numbers.
$\supset$ will select the second or third argument depending on the boolean value of the first argument. Thus there will be one for each base type.
Y_σ will denote the fixed point operator. The idea is that whenever $f : \sigma \to \sigma$, then f will have a least fixed point a of type σ. Our challenge will be to formalize this in the formal theory and to find mathematical interpretations of these types such that this makes sense.
Application (MN) simply mean that M is thought of as a function, N as an argument and (MN) then just denotes the application. It may be confusing that we are not using the more conventual notation $M(N)$, but this has turned out to be less convenient in this context.

If M is a term of type τ, M may contain a variable x of a type σ. The intuition is that M denotes an unspecified object of type τ, an object that becomes specified when we specify the value of x. $(\lambda x.M)$ will denote the function that maps the specification of x to the corresponding specification of M.

The reader may have noticed that we have omitted a few parentheses here and there. We will do so systematically, both for types and terms.
If f is a function of type $(\sigma \to (\tau \to \delta))$ we will view f as a function of two variables of types σ and τ. We will write $\sigma, \tau \to \delta$ for such types. If then M is of type $\sigma, \tau \to \delta$, N is of type σ and K is of type τ we will write MNK instead of $(((M)(N))(K))$.This means that the application is taken from left to right.

Lemma 2.5.4 *Each type will be of the form $\sigma = \tau_1, \ldots, \tau_n \to b$ where $n \geq 0$ and b is one of the base types.*

The proof is easy by induction on the length of σ seen as a word in an alphabet.

With this convention we see that the type of $\supset_\iota$ is $o, \iota, \iota \to \iota$ and the type of $\supset_o$ is $o, o, o \to o$. We then view these as functions of three variables.

2.5.2 Operational semantics for PCF

An operational semantics for a language designed to describe algorithms will be the specification of how to carry out step-by-step calculations or computations. The operational semantics for PCF will be a set of rules for how to rewrite terms in order to ' compute' the value. This is of course most relevant when we are dealing with combined terms of base types.

In PCF we have a similar distinction between free and bounded occurrences of variables as in first order logic, x becomes bounded in $\lambda x.M$. As for first order languages, one term N may be substituted for a variable x in a term M if no variables free in N becomes bounded after the substitution. We write M^x_N for the result of the substitution, always assuming that N is substitutable for x in M.

Definition 2.5.5 We define the relation $\longrightarrow$, denoted by a long arrow, as the reflexive and transitive closure of the following one-step reductions (we assume that the typing is correct):

$Zk_0 \longrightarrow tt.$

$Zk_{n+1} \longrightarrow ff.$

$Pk_{n+1} \longrightarrow k_n.$

$Sk_n \longrightarrow k_{n+1}.$

$\supset_b ttMN \longrightarrow M$ where b is a base type.

$\supset_b ffMN \longrightarrow N$ where b is a base type.

$(\lambda x.M)N \longrightarrow M^x_N$.

$Y_\sigma M \longrightarrow M(Y_\sigma M)$.

$M \longrightarrow M' \Rightarrow MN \longrightarrow M'N$.

$N \longrightarrow N' \Rightarrow MN \longrightarrow MN'$.

$\lambda xM \longrightarrow \lambda y M^x_y$.

The last item tells us that we may replace a bounded quantifier with another one not occurring in M. As an example we will see how we may compute f(x) = 2x on the natural numbers:

Example 2.5.6 The function $f(x) = 2x$ is defined by primitive recursion from the successor operator as follows:

- $f(0) = 0$
- $f(S(x)) = S(S(f(x))$.

For the sake of readability, let g be a variable of type $\iota \to \iota$. Consider the term M of type $\iota \to \iota$ defined by

$$M = Y_{\iota\to\iota}\lambda g.\lambda x^\iota(\supset_\iota (Zx)k_0S(S(g(Px)))).$$

If we look at the expression $\lambda g.\lambda x^\iota(\supset (Zx)k_0S(S(g(Px))))$ we see that to each g we define a function that takes the value 0 on input 0 and $g(x-1)+2$ for positive inputs x. The least fixed point of this operator is exactly the function f. The verification of e.g. $Mk_2 \longrightarrow k_4$ is a lengthy process.

Example 2.5.7 We will show how the μ-operator can be handled by PCF, i.e. we will define a term of type $(\iota \to \iota) \to \iota$ that must be interpreted as the μ-operator.
Note that $F(f) = \mu x.f(x) = 0$ can in the same sense be defined in the following way: $F(f) = G(f, 0)$ where

$G(f, k) = 0$ if $f(k) = 0$.

$G(f, k) = G(f, k + 1) + 1$ if $f(k) > 0$.

G will be the fixed point of a PCF-definable operator, and then the μ-operator is definable.

The conditionals $\supset_o$ and $\supset_\iota$ can be extended to conditionals $\supset_\sigma$ for all types σ, see Exercise 2.14

Definition 2.5.8 Let $f : \mathbb{N} \to \mathbb{N}$ be a partial function. We say that f is PCF-*definable* if there is a term M of type $\iota \to \iota$ such that the following are equivalent for all numbers n and m:

1. $Mk_n \longrightarrow k_m$
2. $f(k)$ is defined and $f(n) = m$.

We used the expression 'PCF-definable' also for the μ-operator, trusting that the reader accepts this yet not clarified terminology. We then have

Theorem 2.5.9 *The PCF-definable functions are exactly the partial computable ones.*

Proof
We have shown how to handle the μ-operator and indicated how to handle primitive recursion. The full details are then easy and are left for the reader as Exercise 2.15.

2.5.3 A denotational semantics for PCF

The operational semantics for PCF explains how we may rewrite a term in such a way that we eventually may read off a number or a boolean value as the interpretation of the term. When we give a denotational semantics, we will interpret each term as a mathematical object in some set. One important aspect will be that the reductions of the operational semantics should not alter the denotational interpretations. If we look at the term M for the $f(x) = 2x$ function, we obtain that $Mk_2 \longrightarrow k_4$ in the operational semantics, while we want $[\![Mk_2]\!] = 4$, where we will use $[\![\cdot]\!]$ for the denotational semantics.

For the rest of this section, we leave out essentially all details. It is not expected that the reader will be able to fill in the details without consulting a textbook on domain theory or some other introduction to a similar topic. Most lemmas etc. will be left without proof.
Consider the term $N = Y_\iota \lambda x.S(x)$. If we try to evaluate Nk_0 we see that we get an infinite reduction sequence. Thus the only sensible way to interpret this term is by accepting 'undefined' as a possible value, and then using it in this case.

Definition 2.5.10 Let $\mathbb{N}_\perp$ and $\mathbb{B}_\perp$ be the set $\mathbb{N}$ of natural numbers and the set $\mathbb{B}$ of boolean values $\{true, false\}$ extended with a new element $\perp$ for the undefined.

We will interpret each type σ as a set $D(\sigma)$, and we let $D(\iota) = \mathbb{N}_\perp$ and $D(o) = \mathbb{B}_\perp$.
Each of these sets will be partial orderings by letting $\perp$ be the smallest element and the rest maximal elements that have no ordering between them. Such ordered sets are called *flat domains*.

Definition 2.5.11 A partial ordering $(D, \sqsubseteq)$ is called *bounded complete* if

1. Each bounded set has a least upper bound.
2. Each directed subset $X \subseteq D$ is bounded, with least upper bound $\sqcup X$.

It is easy to see that $\mathbb{N}_\perp$ and $\mathbb{B}_\perp$ both are bounded complete.

Definition 2.5.12 Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be two partial orderings that are bounded complete.
A function $f : D \to E$ is called *continuous* if f is monotone and for all directed subsets X of D we have

$$\sqcup_D X = \sqcup_E \{f(x) \mid x \in X\}.$$

Lemma 2.5.13 *Let $(D, \sqsubseteq)$ be bounded complete and let $f : D \to D$ be continuous. Then f has a least fixed point in D.*

Proof
The empty set has a least upper bound, which we call $\perp_D$.
Then

$$\perp_D \sqsubseteq_D f(\perp_D) \sqsubseteq_D f(f(\perp_D)) \sqsubseteq \cdots.$$

The least upper bound of this sequence will be the least fixed point of f.

Definition 2.5.14 Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be two orderings that are bounded complete.
Let $D \to E$ be the set of continuous maps from D to E.
If $f \in D \to E$ and $g \in D \to E$, we let $f \sqsubseteq_{D \to E} g$ if

$$\forall x \in D(f(x) \sqsubseteq g(x)).$$

Lemma 2.5.15 *Let $(D \to E, \sqsubseteq_{D \to E})$ be as above. This partial ordering will be bounded complete.*

Not mentioning the ordering, we now interpret each type σ by recursion on σ as follows:

$$D(\sigma \to \tau) = D(\sigma) \to D(\tau).$$

What remains to be done can briefly be described as follows:

1. An *assignment* will be a map from a set of typed variables x_i^σ to elements of $D(\sigma)$. The set of assignments can be viewed as elements of cartesian products of some $D(\sigma)$'s, and will be ordered in a bounded complete way by the coordinate-wise ordering.

2. Each term M of type σ will be interpreted as a continuous function $[\![M]\!]$ from the set of assignments to $D(\sigma)$. It will sometimes be convenient to consider assignments restricted to the free variables of M, sometimes convenient to accept dummy variables.

3. We must show that the least fixed point operator is a continuous map from $(D, \sqsubseteq D)$ to D. This is used to define $[\![Y_\sigma]\!]$.

4. We must show that application is a continuous map from $(D \to E) \times D$ to E. This is used to define $[\![MN]\!]$ from $[\![M]\!]$ and $[\![N]\!]$.

5. We must show that abstraction is a continuous map from $D \times E \to F$ to $D \to (E \to F)$. This is used to define $[\![\lambda x.M]\!]$ from $[\![M]\!]$.

6. We must give interpretations to the constants k_n, S, P, Z, $\supset_\iota$ and $\supset_o$. This can easily be done by the reader.

7. We must show that if $M \longrightarrow N$ then $[\![M]\!] = [\![N]\!]$. The only case that requires some work is the reduction

$$(\lambda x.M)N \longrightarrow M^x_N$$

where we must show by induction on M that

$$[\![M]\!](s^x_{[\![N]\!](s)}) = [\![M^x_N]\!](s).$$

The proof by induction is not very hard.

The denotational semantics for PCF that we have given is the one used originally by Scott when he formed the logic LCF that was turned into the programming language PCF by Plotkin . There are however alternative ways of interpreting PCF-terms reflecting interesting aspects of PCF. One consequence of the existence of a denotational semantics is that a closed term of type ι cannot be interpreted as two different numbers. Consequently, though the use of $\longrightarrow$ is a non-deterministic process, there is no canonical way of performing the next step, we do not risk to obtain different values to the same term.
We end this section by stating the converse, due to Plotkin, without any hints of proof:

Theorem 2.5.16 *Let M be a closed term og type ι. If $[\![M]\!] = n$, then $M \longrightarrow k_n$.*

2.6 Exercises to Chapter 2

Exercise 2.1 A *continued fraction* is a finite or infinite tree of fractions

$$\frac{1}{1 + n_0 + \frac{1}{1+n_1+\frac{1}{1+n_2+\cdots}}}.$$

A finite continued fraction will be a rational number while the value of an infinite continued fraction is defined as the limit of the finite subfractions. This limit will always exist. Why? (This part is not an exercise in logic, rather in elementary analysis.)

a) Show that if $0 < a < 1$, then there is a unique continued fraction with value a.

b) Show that the continued fraction of a will be infinite if and only if a is rational.

c) Show that the bijection obtained between the irrational elements in $[0,1]$ and $\mathbb{N}^{\mathbb{N}}$ using continued fractions is a homeomorphism, i.e.c̃ontinuous in both directions.

d) Discuss in which sense we may claim that the homeomorphism in c) is computable.

Exercise 2.2 Show that if F is a computable functional of type 2, then F will have a computable associate.

Exercise 2.3 Prove lemma 2.1.8.

Exercise 2.4 Let $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ be continuous, and let $T(F, f)$ be the set of finite sequences σ bounded by f such that F is not constant on B_σ.
Show that $T(F, f)$ will be a tree of sequences, and use König's lemma to show that $T(F, f)$ is finite for each $f \in \mathbb{N}^{\mathbb{N}}$. Use this to give an alternative proof of Lemma 2.1.9.

Exercise 2.5 Show that there is a continuous function $\hat{\Phi} : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$ such that whenever α is an associate for a continuous $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$, then $\hat{\Phi}(\alpha)$ is an associate for $\Phi(F)$.
Show that $\hat{\Phi}$ can be chosen to be computable in the sense that

$$G(x, \alpha) = \hat{\Phi}(\alpha)(x)$$

is computable.

Exercise 2.6 Let A and B be subsets of $\mathbb{N}$ and let K_A and K_B be the corresponding characteristic functions.
Show that the following are equivalent:

i) $B <_m A$

ii) There is a computable functional

$$F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$$

such that $K_B = F(K_A)$.

Hint: If F is computable, then we may use the compactness of $\{0,1\}^{\mathbb{N}}$ to see that for any $n \in \mathbb{N}$ there is an upper bound of the amount of information about an arbitrary element f of $\{0,1\}^{\mathbb{N}}$ we need in order to compute $F(f)(n)$, and then use that all truth-value functions of a fixed arity can be represented by a formula in propositional calculus.

Exercise 2.7 Tait showed that the fan functional is not Kleene-computable. Fill out the details in the following proof of Tait's result:

1. A *quasi-associate* for F is a function $\alpha : \mathbb{N} \to \mathbb{N}$ such that

- $\alpha(n) = m + 1 \Rightarrow F(f) = m$ when $\sigma_n \prec f$.
- $\alpha(n) > 0 \wedge \sigma_n \prec \sigma_m \Rightarrow \alpha(m) = \alpha(n)$.
- If $f \in 1 - sc(F)$ then for some n, $\sigma_n \prec f$ and $\alpha(n) > 0$.

If $\phi_e(\vec{a}, F)\downarrow$ and α is a quasi-associate for F, then there is some n such that whenever G has an associate extending $(\alpha(0), \ldots, \alpha(n))$ and $\phi_e(\vec{a}, G)\downarrow$ then

$$\phi_e(\vec{a}, G) = \phi_e(\vec{a}, F).$$

Hint: Use Use induction on the ordinal rank of the computation tree for $\phi_e(\vec{a}, F)$.

2. There is a quasi-associate for the constant zero function ${}^2O$ and a non-computable $f : \mathbb{N} \to \{0, 1\}$ such that $\alpha(\bar{f}(n)) = 0$ for all n.

3. For any finite part $\bar{\alpha}(n)$ of α there is an associate β extending $\bar{\alpha}(n)$ for a functional that is not constant on $\{0, 1\}^{\mathbb{N}}$.

4. Combining 2. and 3. we obtain a contradiction from the assumption that the fan functional is computable.

Exercise 2.8 If $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$, let $F_i(f) = F(\langle i, f\rangle)$.
Show that there is a total functional Γ satisfying the equation

$$\Gamma(F) = F_0(\lambda x.\Gamma(F_{x+1})),$$

and that we may compute $\Gamma(F)$ uniformly from an associate for F.
The historical interest is that Γ is not computable in the fan functional. This is too hard to be considered as an exercise.

Exercise 2.9 Work out detailed proofs of Lemmas 2.2.5, 2.2.6 and 2.2.9.
This cannot be seen as a simple exercise, but rather as a minor project.

Exercise 2.10 We reduce the definition of $\phi_e(\vec{a}, \vec{f}, \vec{F})$ by ignoring schemes v) and vi), and technically work with a subsystem.

a) Prove the S^n_m-theorem for computations with function and functional arguments as well as number arguments.

b) Prove the recursion theorem for computations with function and functional arguments as well as number arguments.

c) Use the recursion theorem to show that scheme v) is redundant in the original definition, i.e. that the class of functional computable in the subsystem is closed under primitive recursion.

d) Use the recursion theorem to show that scheme vi) is redundant in the original definition, i.e. that the μ-operator is definable in the subsystem. Hint: You may look at how the μ-operator is defined in PCF.

Exercise 2.11 Prove Lemma 2.4.10 and Lemma 2.4.11 in detail.

Exercise 2.12 We actually have a hierarchy for sets definable by second order formulas as well. Let

- $A \subseteq X$ is Π^1_{k+1} if for some Σ^1_k set $B \subseteq \mathbb{N}^{\mathbb{N}} \times X$,

$$\vec{x} \in A \Leftrightarrow \forall f \in \mathbb{N}^{\mathbb{N}}((f, \vec{x}) \in B).$$

- $A \subseteq X$ is Σ^1_{k+1} if for some Π^1_k set $B \subseteq \mathbb{N}^{\mathbb{N}} \times X$,

$$\vec{x} \in A \Leftrightarrow \exists f \in \mathbb{N}^{\mathbb{N}}((f, \vec{x}) \in B).$$

a) Prove that the class of Π^1_k sets are closed under finite unions and intersections, number quantifiers and universal function quantifiers. Prove that there are universal Π^1_k sets for any $k \geq 1$ and any dimension.

b) Formulate and prove the analogue results for the Σ^1_k-classes.

Exercise 2.13 a) Let $f : \mathbb{N} \to \mathbb{N}$ be computable in ${}^2E$. Show that the image of f will also be computable in ${}^2E$. Discuss what this means for the relation between semicomputable sets and c.e. sets relative to ${}^2E$.

b) Show that the class of sets semicomputable in ${}^2E$ is closed under finite unions and intersections, but not under complements.

c) Show that two disjoint Σ^1_1-sets can be separated by a Δ^1_1-set.
Hint: Use Gandy selection.

d) Show that there are disjoint Π^1_1-sets that cannot be separated by any Δ^1_1-sets.
Hint: Use an analogue of the construction of two computably inseparable c.e. sets.

Exercise 2.14 Let $\sigma = \tau_1, \ldots, \tau_n \to b$ be a type, where b is a base type. Show that there is a term $\supset_\sigma$ of type

$$o, \sigma, \sigma \to \sigma$$

such that whenever s and t are terms of type σ, then,

$$\supset_\sigma (tt)st \longrightarrow \lambda x_1^{\tau_1} \cdots \lambda x_n^{\tau_n}.sx_1 \cdots x_n.$$

$$\supset_\sigma (ff)st \longrightarrow \lambda x_1^{\tau_1} \cdots \lambda x_n^{\tau_n}.tx_1 \cdots x_n.$$

Exercise 2.15 Prove Theorem 2.5.9 in detail.

Chapter 3

Non-trivial exercises and minor projects

In this chapter we have collected some exercises or minor projects that are based either on the material in more than one chapter or that are too hard to be considered as exercises in the ordinary sense. We call them exercises, though some of the items in this chapter are much more demanding than ordinary exercises.

Exercise 3.1 Show that there is a computable partial ordering $(\mathbb{N}, \prec)$ such that every countable partial ordering $(X', <')$ can be embedded into $(\mathbb{N}, \prec)$.

Exercise 3.2 Let Σ be a fixed alphabet. Show that there is an enumeration $\{M_i\}_{i\in\mathbb{N}}$ of the set of Turing machines over Σ and a one-to-one enumeration $\{w_j\}_{j\in\mathbb{N}}$ of the set of words in Σ^* such that the function f satisfying the equation

$$f(i,j) \simeq k \Leftrightarrow M_i(w_j) = w_k$$

is computable, where $\simeq$ means that either is both sides undefined, or both sides are defined and with the same value.

Exercise 3.3 In the text we have indicated how to establish some of the properties of the hierarchy $\{F_\alpha\}_{\alpha<\epsilon_0}$. Work out all the details and write a small essay about it. You may well combine this with Exercise 3.4.

Exercise 3.4 Show that if $\alpha < \epsilon_0$ then F_α is provably computable. Suggest a fundamental sequence for ϵ_0 and thus a function F_{ϵ_0}. If properly defined, F_{ϵ_0} will not be provably computable in PA, but in some second order extension. Discuss how we may formulate a proof of the totality of F_{ϵ_0} in some second order extension of PA.

Exercise 3.5 Recall that a well ordering will be a total ordering $(X, <)$ such that any nonempty subset of X will have a least element. An initial segment of a well ordering $(X, <)$ will be a subset Y of X such that $y \in Y \wedge x < y \Rightarrow x \in Y$, together with the ordering $<$ restricted to Y.

a) Show that an initial segment of a well ordering is itself a well ordering.

b) Show that if $(X_1, <_1)$ and $(X_2, <_2)$ are two well orderings and π_1 and π_2 are two isomorphisms between initial segments of $(X_1, <_1)$ and $(X_2, <_2)$ resp. , then $\pi_1(x) = \pi_2(x)$ if they are both defined.
Hint: Assume not. Consider the maximal initial segment where they agree, and show that they must agree on the 'next' element as well.

c) Show that if $(X_1, <_1)$ and $(X_2, <_2)$ are two well orderings then they are either isomorphic or one is isomorphic to a proper subset of the other. In all cases the isomorphism is unique.
Hint: Let π be the union of all isomorphisms between initial segments of the two well orderings. Then π is the unique isomorphism in question.

An *ordinal number* will be a set α such that

α is *transitive*, i.e. if $\beta \in \alpha$ then β is a set and if moreover $\gamma \in \beta$, then $\gamma \in \alpha$.

$(\alpha, \in)$ is a well ordering.

d) Show that the empty set $\emptyset$ is an ordinal number, and that $\emptyset \in \alpha$ for all non-empty ordinal numbers. Describe the three smallest ordinal numbers.

e) Show that if α is an ordinal number and $\beta \in \alpha$, then β is an ordinal number.

f) Show that if α and β are two ordinal numbers, then $\alpha = \beta$, $\alpha \in \beta$ or $\beta \in \alpha$.
Hint: Show that identity functions are the only possible isomorphisms between initial segments of ordinal numbers.

g) Show that if α is an ordinal number, then $\alpha + 1$ defined as $\alpha \cup \{\alpha\}$ is an ordinal number.

h) Show that the union of any set of ordinal numbers is an ordinal number.

A binary relation (X, R) is *well founded* if each nonempty subset Y of X has an R-minimal element x, i.e. an element $x \in Y$ such that $y \notin Y$ whenever yRx.

i) Let (X, R) be a well founded relation. Let R^* be the transitive (but not reflexive) closure of R. Show that R^* is a well founded relation.

j) Show that if (X, R) is a well founded relation, there is a unique function ρ mapping X into the ordinal numbers such that

$$\forall x \in X(\rho(x) = \bigcup\{\rho(y) + 1 \mid yR^*x\}).$$

Hint: Use well foundedness to show that there is a maximal partial solution to the function equation for ρ, and once again to show that this maximal solution will be defined on all of X.

The function ρ will be called *the rank function* of R. The image of ρ will be an ordinal number and will be called the *rank* of R.

Bibliography

[1] S. Barry Cooper, *Computability Theory*, Chapman & Hall/CRC (2004).

[2] Christopher C. Leary, *A Friendly Introduction to Mathematical Logic*, Prentice hall, 2000.

[3] D. Normann *Mathematical Logic II*, Compendium December 21 - 2005, available from the authors home page.

[4] Piergiorio Odifreddi, *Classical Recursion Theory*, Elsevier (1989).

[5] Hartley Rogers Jr., The Theory of Recursive Functions and Effective Computability, MIT Press (1967 1.st ed. and 1987 2. ed.)

[6] G. E. Sacks, *Higher Recursion Theory*, Springer-Verlag (1990)

[7] Robert I. Soare, *Computably Enumerable Sets and Degrees*, Springer Verlag (1987).
Soare is writing a new book in two volumes under contract with Springer Verlag. This will soon be available.

[8] Thomas Streicher, *Domain-Theoretic Foundations of Functional Programming*, World Scientific (2006).

Index